UTILIZING WEAKLY-SUPERVISED LEARNING FOR HASHTAG SEGMENTATION AND NAMED ENTITY DISAMBIGUATION

by

Arda Çelebi

B.S., Computer Engineering and Information Science, Bilkent University, 2002

M.S., Computer Engineering, Boğaziçi University, 2012

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2020

UTILIZING WEAKLY-SUPERVISED LEARNING FOR HASHTAG SEGMENTATION AND NAMED ENTITY DISAMBIGUATION

APPROVED BY:

DATE OF APPROVAL: 29.06.2020

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere and heartfelt gratitude to my advisor Assoc. Prof. Arzucan Özgür. Before I met you, I was not sure about my future at the department. You were like an angel who appeared out of nowhere and gave me a second chance I desperately needed at the time. You always have trust in me and that empowered me all the way to the finish line. Thank you for guiding me through all those years. Thank you for pushing me to apply for BÜVAK and ASELSAN scholarships, as well as researcher position at TETAM. In none of these I thought that I could get it. They all happened just because of you and your belief in me. I am and will be grateful for your encouragement forever. Thank you for your excellent feedback during our studies, your always smiling face in our meetings, and your uplifting words and motherly support whenever something bad happens. I owe you my deepest gratitude.

I would like to thank my thesis committee members, Prof. Tunga Güngör and Prof. Murat Saraçlar for their valuable recommendations and encouragement during the progress stage of the thesis, as well as Assoc. Prof. Gülşen Cebiroğlu Eryiğit and Prof. Olcay Taner Yıldız for spending their time on my thesis and their valuable contributions.

I would like to give my special thanks to Prof. Kemal Oflazer. You are the one who introduced me to the Natural Language Processing field back in 2000 when I was an undergraduate student at Bilkent University. You were (and still are) such a charismatic and excellent teacher that you made me sell my soul to the "NLP God" and worship him for the rest of my life. Moreover, thanks to you, I was selected for the Johns Hopkins (JHU) Summer Workshop for the internship as well as PhD degree at the University of Southern California (USC). During my time in those places, I am also grateful to have the support of Prof. Dragomir Radev in JHU and Prof. Kevin Knight in USC. My career in USC didn't pan out as expected. After returning back to Turkey, I became a programmer who developed information management tools on

the Net, such as Newzie and Ziepod. Yet, when I said I want to get back to the academics, it was Prof. Kemal Oflazer who made me get into Boğaziçi University with his recommendation to Prof. Murat Saraçlar.

If today I am graduating from Boğaziçi University, one of the main reasons is Prof. Murat Saraçlar and Prof. Ethem Alpaydın as they got me accepted to Boğaziçi University. Both are the best in their departments. I would especially like to thank Prof. Murat Saraçlar. I am so grateful that you opened your arms and supported me in my first years at Boğaziçi. You gave me a research job in your TUBITAK project. That not only helped me keep my head above the water financially, but also kick-started me in the academic life beyond my imagination. In a very short time, we were able to do a lot of good science and published three papers. You really put me back on track after all those years away from academic life. I am so much grateful for your support, your excellent guidance in the field, your straight-to-the-point feedback, and your professionalism. Moreover, over that time, I was also so lucky to have two colleagues and friends, namely Haşim Sak and Erinç Dikici. First of all, when I met Haşim, you were already getting ready to be a PhD graduate. In that short time, you supported me so much and eventually became a role model for my PhD career. Who doesn't want to work at Google after graduation, right? Thank you for giving me that inspiration back in those years. Secondly, my dearest friend Erinç. We worked together for a couple of years under the guidance of Murat Saraçlar. I was so inspired by your professionalism and friendship. I am grateful to share those first years with you and learn the subtleties of the research studies. Thank you.

I am grateful that I am a part of the Computer Engineering Department at Boğaziçi. All teachers are the best in their fields. I especially like to give my thanks to Prof. Tunga Güngör. Thank you for listening to my PhD progress reports in the last four years and giving insightful feedback. Not only that, I had the chance to take your Machine Translation class and I have never met a teacher who is more methodical and yet humble as you are. Such inspiring traits. Another inspiring teacher throughout my existence in the department is of course Suzan Üskudarlı. To me, you are a one-in-a-million teacher. I took all your classes and none of them is like any other class I

took in my entire student life. I felt too much excitement in your classes. I learned so much about philosophy of the Web, people's interaction, and much more. And your infinite enthusiasm towards the subject as well as students stuns me even right now. I am grateful to be acquainted with such a personality as yours. In the department, there is one more teacher that truly changed me in the early years and she is Prof. Pınar Yolum. You are one of the true professional researchers I have met in my life. You are the teacher who taught me how to write a scientific report and paper. This is something that I told this to people over and over again. When I started taking your class, at the beginning I was spending the entire week writing a review. When I finished your class, it took only 4-5 hours of my time. Such a progress! Writing reports and papers is the most time consuming and frankly boring time of the graduate studies. Thanks to you, they all went so quickly. So thank you one more time.

I was a member of this department for 10 years and most of the time was spent with friends in the lab. Especially when I was living near the university, there were many late nights. In the early days, I shared the lab with great people like Akin Günay, Barış Gökce and Nadin Kökciyan. They are all very hard-working and excellent friends. I am especially grateful to work closely with Nadin on the sentiment analysis competition project. I remember we worked very hard day and night and finished the project on time and achieved pretty good results in that international competition, making our department proud. Thank you for making me experience that success. I am really happy that you graduated quickly and got a teaching job at the University of Edinburgh. You earned it! Later on, we got our own lab as an NLP group. Those times were great with Çağıl Uluşahin, Mert Tiftikci, Göksu Öztürk, Betül Bilgin, and even people from other groups like Nefise Yağlıkcı and Can Kurtan. I feel very lucky to share the lab with these clever and hard-working people. I especially remember those days when we conducted weekly AI lab meetings and even movie nights at the roof. I enjoyed every minute of them and you all made that happen. Around the end of my time in the department, we moved to a new lab and I became friends with a whole set of young and brilliant people. Engaging conversations with Onur Güngör, Mert Tiftici's passion for trying new things, Abdullatif Köksal's brilliance and humbleness, Riza Özcelik's love for research and teaching, Gönül Ayci's perseverance, Hakime Öztürk's excellence

in small details, Selen Parlar's love of plants and people, and Gökce Uludoğan's love of cats. I am grateful to experience all this and know you all. There are also very interesting people from other labs. I am especially fascinated by the people from our robotics lab. How can I forget Ibrahim Özcan's love to build things with parts shipped from China as well as parts printed by his 3D printer. You are one of a kind. I feel lucky to be acquainted with Okan Aşık and his award-winning work on the rescue robots as well as Binnur Görer's work on robots helping elderly people. Both such inspiring studies for me. Thank you all.

Computer Engineering Department at Boğaziçi might be the best computer engineering department in Turkey. The department is in excellent hands in the leadership of Prof. Cem Ersoy and Tuna Tuğcu. I am grateful to be acquainted with Prof. Cem Ersoy. I had never had the chance to be in your class. Nevertheless, I listened to your stories on the roof many times with a big smile on my face. Your storytelling is unmatched. Not only that, when I wrote "Open Letter to the Department" regarding the need for more computing resources, you personally took the responsibility and bought a Nvidia DGX server to the department (like made it happen out of thin air) so that all graduates can do their research much faster. I truly believe that if you did not buy that computer, I would not be able to finish my research in time. So I owe you so much gratitude. Moreover, I am also thankful for being selected for the TETAM scholarship. Even in just one year, I was lucky to meet with great people in this program, like Cansu Canbek. Such an inspiring and enthusiastic person. Your expertise in nano-networks subjects excites me so much. Alongside Prof. Cem Ersoy, there is another leader figure, namely Prof. Tuna Tuğcu. I have never met such a hard-working teacher and quite frankly a person like you. You are so professional at your work and really spare no effort at teaching and guiding people around you. I am inspired by just seeing you in the hallways. So thank you for making me feel that.

In our department, I am also thankful for the support staff, namely Birol Yamaç, Şeker Biliç, Mustafa Tunç and Erhan Demir. When I get to the rooftop to have tea or to have breakfast, my encounter with you guys always puts a smile on my face. I admire your friendliness and easy-to-talk personalities. I especially miss Birol singing

Karadeniz songs. You are the invisible heroes of our department. Thank you for everything.

I am lucky to be supported by multiple institutions and programs during my PhD studies. For the most part of my PhD studies, I was doing the research under two BAP projects (Boğaziçi Research Fund Grant Numbers 11170 and 14201) and I am grateful for their support. Similarly, I am also grateful of the support of BÜVAK scholarship program for doctoral students at Boğaziçi University. Thanks to Arzucan Hoca's insistence, I applied for the ASELSAN Graduate Scholarship for Turkish Academicians and got accepted to my surprise. In that, I also like to thank Aykut Koç at ASELSAN for believing in me and helping me get this scholarship. Towards the end of my time here, again thanks to Arzucan Hoca's insistence, I also applied to be a researcher at TETAM (2007K12-873) and got accepted. I feel very privileged to attend TETAM's bi-weekly meetings at the Kandilli campus of Boğaziçi University and get to know other researchers and their studies there.

My most special thanks go to my parents; my mother Afet and my father Ünal. They are my shining north star. Their unstoppable continuous infinite support has always fueled me in my life and it was no different while I was at Boğaziçi, even though most of the time they are in Izmir. I never felt their absence while living in Istanbul. When I was down, they lifted me into the air and made me keep going. When I progressed through the thesis, my happiness and excitement bloomed further by sharing those feelings with them. Not just this thesis, I dedicate all my life's work to them.

ABSTRACT

UTILIZING WEAKLY-SUPERVISED LEARNING FOR HASHTAG SEGMENTATION AND NAMED ENTITY DISAMBIGUATION

Today's high-performing machine learning algorithms learn to predict by the supervision of large amounts of human-labeled data. However, the labeling process is costly in terms of time and effort. In this thesis, we design weakly-supervised approaches, which are based on automatically labeling raw data, for two different Natural Language Processing (NLP) tasks, namely hashtag segmentation and Named Entity Disambiguation (NED). Hashtag segmentation's aim is to identify the words in the hashtags, so as to process and understand them better. We propose a heuristic to obtain automatically segmented hashtags using a large tweet corpus and use these data to train a maximum entropy classifier. State-of-the-art accuracy is achieved for hashtag segmentation without using any manually labeled training data. The target of NED, which is the second task that we address, is to link the named entity (NE) mentions in text to their corresponding records in the Knowledge Base. We hypothesize that the types of the NE mentions may provide useful clues for their correct disambiguation. The standard approaches for identifying mention types require a type taxonomy and large amounts of mentions annotated with their types. We propose a cluster-based mention typing approach, which does not require a type taxonomy or labeled mentions. This weakly-supervised approach is based on clustering the NEs in Wikipedia by using different levels of contextual information and automatically generating data for training a mention typing model. The mention type predictions lead to significant F-score improvement when incorporated to a supervised NED model. This thesis shows that designing weakly-supervised approaches by considering the underlying characteristics of the addressed problem can be an effective strategy for NLP.

ÖZET

ZAYIF DENETİMLİ ÖĞRENME YAKLAŞIMI KULLANARAK HASHTAG AYRIŞTIRMA VE VARLIK İSMİ ANLAMLANDIRMA

Günümüzün yüksek başarımlı makine öğrenmesi yöntemleri başarılarını etiketlenmiş çok miktarda veri üzerinde öğrenme yapmalarına borçludur. Fakat etiketleme çok fazla zaman ve efor gerektirir. Bu tezde iki Doğal Dil İşlemesi (DDİ) alanında zayıf denetimli öğrenim yapmak için otomatik veri etiketleme yöntemi önerdik. İlk uygulama alanımız olan Hashtag Ayrıştırması, hashtag'lerin otomatik olarak işlenmesi ve anlamlandırılması için orijinal sözcüklerine bölünmesidir. Büyük bir tweet veri setinden elde edilen istatistiklere göre hashtag'leri otomatik olarak ayrıştırdık ve en güvenilir ayrıştırmaları maksimum entropi sınıflandırıcısını eğitmek için kullandık. Elle etiketli eğitim verisi kullanmadan hashtag ayrıştırma problemi için literatürdeki en yüksek doğruluk oranlarını elde edebildik. Çalıştığımız ikinci alan olan Varlık İsimlerinin Anlamlandırılmasında (VİA) amaç metinde tanınan varlık isimlerini bilgi bankasında karşılık gelen kayıtlara bağlamaktır. Bahsedilen varlığın türünü önceden tespit edersek bu bilginin VİA'da başarıyı artıracağını öngördük. Varlık türü tanımlanması için standart yaklaşımlar elle hazırlanmış tür taksonomisine ve metinde bahsi geçen varlıkların türlerinin etiketlendiği büyük miktarda veriye ihtiyaç duymaktadır. Bizim önerdiğimiz yöntem ile varlıkları değişik seviyelerde bağlamsal benzerliklerine göre kümelendirip, küme kimliklerini tür olarak varlıklara atadık. Bu sayede, tür taksonomisine olan ihtiyaç giderilirken, Wikipedia makalelerindeki varlıkları, onlara atanan türler ile işaretleyerek, tür tahminini yapacak sistem için eğitim verisini otomatik olarak oluşturduk. Tür tahminlerinin ek bilgi olarak kullanılması VİA sisteminin başarısını anlamlı seviyede artırdı. Bu tez, problemin özelliklerini dikkate alarak tasarlanan zayıf denetimli öğrenme yaklaşımlarının DDİ'de etkili bir strateji olabileceğini gösterdi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

WC	Word-based Context
SFC	Surface Form-based Context
EC	Entity-based Context
E_{WC}	WC-based Entity Embeddings
E_{SFC}	SFC-based Entity Embeddings
E_{EC}	EC-based Entity Embeddings
E_{SYN}	Synset-based Entity Embeddings
W_{CC}	Cluster-centric Word Embeddings
W_{SF}	Surface Form-based Word Embeddings
C_{WC}	Entity Clustering based on WC
C_{SFC}	Entity Clustering based on SFC
C_{EC}	Entity Clustering based on EC
C_{SynC}	Entity Clustering based on Synset-based Context
C_{BRO}	Entity Clustering based on Brown Clustering
c	Candidate Entity
SF_m	Surface Form of the Mention
SF_c^{best}	Closest Surface Form of c wrt SF_m
D_c	Doc2vec Embedding of c
D_t	Doc2vec Embedding of the test document t
E_c	Entity Embedding of c
T_c^{SF}	Type of the Surface Form SF of c
T_c^e	Entity Type of c
P_c^n	Typing Model Probability for c
R_c	Ranking Probability of c at the First Stage of Ranking
S_c	All Surface Forms for c seen in Surface Form Data Set
C^m	All Candidates for the Mention m
C_c^t	All Occurrences of the same c in test Document t
C_c^{prev}	All Previous Occurrences of c wrt the Mention m

C_c^{next}	All Future Occurrences of c wrt the Mention m
C_{topNxM}	Highest ranked N candidates in surrounding M Mentions
F^e	Entity frequency of candidate named entity
F^s	Surface form frequency of candidate named entity

LIST OF ACRONYMS/ABBREVIATIONS

AGCCS	Average Gold Candidate Cluster Size
AMI	Average Mutual Information
BiLSTM	Bidirectional Long-Short Term Memory
CBOW	Continuous Bag-of-Words
CNN	Convolutional Neural Network
CR	Co-reference Resolution
CRF	Conditional Random Field
EDA	Exploratory Data Analysis
EC	Entity-based Context
HMM	Hidden Markov Model
FSM	Finite State Machine
FST	Finite State Transducer
IE	Information Extraction
IR	Information Retrieval
KB	Knowledge Base
kNN	k Nearest Neighbor
LM	Language Model
LREC	Language Resources and Evaluation Conference
LSTM	Long-Short Term Memory
MEMM	Maximum Entropy Markov Model
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MUC	Message Understanding Conference
NE	Named Entity
NEN	Named Entity Normalization
NER	Named Entity Recognition
NED	Named Entity Disambiguation
NLP	Natural Language Processing
NN	Neural Network

MaxEnt	Maximum Entropy
POS	Part-of-Speech
RNN	Recurrent Neural Network
Seq2Seq	Sequence-to-Sequence
SFC	Surface form-based Context
SOTA	State-of-the-art
SVM	Support Vector Machines
TAC	Text Analytics Conference
TF-IDF	Term Frequency-Inverse Document Frequency
VSM	Vector Space Model
WC	Word-based Context
xDB	Experiment Database

1. INTRODUCTION

1.1. Motivation

When a machine learning algorithm is trained with annotated data, it is called supervised learning as the annotations in the data supervise the learning process. Today, in order to achieve human-level performance in many Artificial Intelligence-related tasks, more and more annotated data are being used. In the literature, near human-level performance has been reported for image recognition [1], speech processing [2], and identification of linguistic components in text [3]. However, such supervised learning with large annotated data may not be feasible, since obtaining such data is labor intensive, time consuming and costly. Recent studies [4, 5] tend to focus on techniques that use less annotated data and even use only unlabeled or raw data. Considering the availability of a gigantic amount of raw data compared to the generally small amount of manually annotated data, utilizing the raw data effectively for learning is a promising approach to follow.

While supervised learning is still desirable, weakly supervised and unsupervised learning methods have also been achieving high level performance. Studies [6–8] showed that using large amounts of raw data in combination with small amounts of annotated data can improve the learning performance. With the help of automating the labelling process by making use of the domain knowledge, dependency to manually annotated data is reduced in weakly supervised learning. Unsupervised learning, on the other hand, does not require any annotated data. Instead, such methods look for patterns in data. Clustering is the best example for unsupervised learning where data points that exhibit similar characteristics are grouped together.

Weak supervision is an umbrella term that covers different ways of making use of large amounts of unlabeled data. Zhou [9] breaks down weak supervision into three categories. The first one is incomplete supervision, where only a subset of the training data has labels. The technique called semi-supervised learning falls into this subcate-

gory. Such approaches combine a small amount of labelled data with the automatically extracted intuition from large amounts of unlabeled data in order to enhance the final trained model. The second type of weak supervision is inexact supervision, where the training data are given with only coarse-grained labels. In this case, the labels in the training data set do not reflect the learned task exactly, but give a rough idea about it. Lastly, the third type is inaccurate supervision. This happens when the labels in the training set are not always ground-truth. There are many examples of inaccurate supervision in the literature. Most of the studies start with an initial model trained on a small amount of annotated data and use that model to annotate unlabeled data. These pseudo-labeled instances are used to expand the actual annotated data so that a better model is learned out of it. Other methods such as co-training [10] and self-training [11] also fall into this type of weak supervision.

All in all, it can be time consuming and costly to build manually annotated data. By using weak supervision, we can automatically create training data with a heuristic or by re-purposing existing data, which eliminates the need for manually annotated data and saves time as well as effort. In this thesis, we propose weakly supervised approaches for two NLP tasks, namely hashtag segmentation and named entity disambiguation.

1.2. Problem Statement

When we are looking to make use of a large amount of unlabeled data, one obvious area involves tweets, since more than half a billion of them are posted every day, 6000 tweets every second. Tweets are very short documents, mostly around 140 characters in length. Compared to the language used in regular text like news articles, the language used in tweets is very noisy and irregular. It was shown that the accuracy of NLP applications drops up to 20 percent when they are applied on tweets [12]. For example, the average F_1-score of the Stanford Named Entity Recognizer [13], which is trained on the CoNLL-2003 [14] shared task data set and achieves state-of-the-art performance on that task, drops from 90.8% [15] to 45.8% on tweets [16]. That hardness attracts many research studies. One particular aspect of tweets that has not been studied much till our research is hashtags. Hashtag is a special token that consists of one or more

words concatenated one after another along with a special prefix pound character "#". It originated as a label to mark the content of tweets. Later on, it evolved into an information conveyor tool where long phrases and even sentences are turned into a hashtag, such as *#ifiwereaboy*. Other hashtags may also include complex noun phrases that include numbers like *#o2007comp*. As more content is carried through hashtags, it became more and more important to break them down into their original words. This enables us to reach and process the conveyed content inside them. In essence, this task is very similar to word segmentation seen in languages like Chinese and Arabic, where no boundary is used between words. In our study, we consider tweets in the English language. Hence, what we need is just regular text in English, where all word boundaries already exist. Then, we can train our model on this naturally annotated data in supervised fashion. However, we argue that such regular text might not exhibit the same characteristics as the words inside the hashtags. Hence, we specifically look for whether we can create a training data set out of the hashtags themselves. Instead of segmenting them manually, we proposed a heuristic to automatically segment millions of hashtags and picked the ones that we are confident about their correct segmentation with acceptable degree. This is a weak supervision approach, or more specifically inaccurate supervision, since the word boundaries in auto-segmented hashtags may not be accurate. Nevertheless, based on our results given in Section 3.6.3, we show that the model trained with inaccurate supervision outperforms the model trained on the regular text.

As a second area for applying weak supervision, we addressed the named entity disambiguation (NED) task, which has been studied for a long time. The goal is to associate the recognized named entity mention with the corresponding entry in the reference knowledge base (KB). This can be very helpful to identify which specific named entities are being mentioned in the context, so that we can apply more precise processes on them. For example, we can count how many times a specific product is mentioned or measure the sentiment towards it. However, it is not an easy task. The mention of the word "Washington" in text may correspond to many different named entities such as the city "Washington D.C." or the newspaper "Washington Post." There are tens of cities and counties in the United States that are named

"Washington." It is hard to decide which among hundreds of candidates are mentioned. In the literature, many studies [17–19] have tried to incorporate the external data such as context into the named entity disambiguation task in order to help the learning algorithm do better and more informed decisions. Likewise, in our research, we focus on how to make use of context better in order to improve the disambiguation accuracy. We built a special mention typing model, which predicts the entity type of the mentioned named entity based on its surface form and surrounding context, so that the predicted type information can be used as an extra clue for the actual disambiguation task. We proposed cluster-based types which are automatically generated by clustering all named entities in our KB based on their contextual similarity observed in Wikipedia articles. Entities in the same cluster are labeled with the same label, that is the Cluster ID. Previous works on the the mention typing task [20–22] use manually curated type taxonomy. Considering that there are over five million named entities in Wikipedia, manually curated taxonomies are inherently incomplete and require a lot of work to create. Our cluster-based approach gives us automatically generated types, which eliminates the need for the taxonomy. Then, we label each hyperlinked mention of the named entity in the Wikipedia articles with the corresponding cluster-based type of that entity. This automatically generates training data to train a model to predict the cluster-based type of a given mention. This approach is weakly supervised, since the auto-generated training data is not exactly meant to be curated for this task, not to mention the cluster-based types as inexact labels. As shown in our results in Section 4.5.4, when we use the predictions of mention typing as an extra input at the disambiguation task, our model achieves better than state-of-the-art results on a number of de facto test sets.

1.3. Publication Notes

Parts of the work in this thesis have appeared in the following publications:

- "Segmenting Hashtags Using Automatically Created Training Data", Çelebi, Arda and Arzucan Özgür. Proceedings of the Language Resources and Evaluation Conference (LREC), pp. 2981-2985, 2016. (Chapter 3) [23]

- "Segmenting Hashtags and Analyzing Their Grammatical Structure", Çelebi, Arda and Arzucan Özgür. Journal of the Association for Information Science and Technology (JASIST), Vol. 69(5), pp. 675-686, 2018. (Chapter 3) [24]
- "Cluster-based Mention Typing for Named Entity Disambiguation", Çelebi, Arda and Arzucan Özgür. Natural Language Engineering (NLE), 2020, *accepted*. (Chapter 4) [25]

The other relevant works that are not part of this thesis are:

- "Description of the BOUN System for the Tri-lingual Entity Detection and Linking Tasks at TAC KBP 2017", Çelebi, Arda and Arzucan Özgür. Proceedings of the Text Analysis Conference (TAC), 2017. [26]
- "BOUNCE: Sentiment Classification in Twitter using Rich Feature Sets", Kökciyan, Nadin, Arda Çelebi, Arzucan Özgür and Suzan Üsküdarlı. Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013), pp. 554-561, 2013. [27]

1.4. Summary of Contributions

In this thesis, we focused on developing weak supervision approaches to create automatically annotated training data for two NLP tasks: hashtag segmentation and named entity disambiguation. The results on both tasks show us that when we create large amounts of automatically annotated data by designing heuristics based on our domain knowledge, such a weakly-supervised approach can be an effective strategy for NLP. Below we outline the contributions for each task, starting with the contributions achieved for the hashtag segmentation task.

- We introduced a heuristic in order to segment hashtags automatically using almost half a billion tweets and obtained 803,000 automatically segmented hashtags. We used this data set as a training set in our hashtag segmentation studies.
- We proposed a feature-rich approach for hashtag segmentation based on the max-

imum entropy model. In addition to various vocabulary- and orthography-based features, we also exploited the local and global context information of the hashtags by using the tweets in which the auto-segmented hashtags reside. Moreover, we incorporated word boundary clues suggested by a language modeling approach. The developed hashtag segmentation system achieved state-of-the-art results without using any manually labeled training data.

- With our state-of-the-art hashtag segmentor, we segmented 60 million (2.1 million distinct) hashtags and studied their internal structure. We found out that around 80% of the hashtags that contain either positive or negative sentiment are multi-word hashtags. We also conducted detailed grammatical analysis of those 60 million auto-segmented hashtags, which is the first of its kind in the literature.
- We publicly shared the data sets that we curated and the script of our hashtag segmentor on the TABI Lab web site [28].

The second part of this thesis involves proposing a new cluster-based mention typing model to improve the named entity disambiguation (NED) task. The following contributions are achieved:

- Our research is the first that uses clustering to obtain cluster-based mention typing models. We clustered over five million named entities and used the cluster IDs as type labels. Our mention typing models were trained on the hyperlinked mentions of the named entities in Wikipedia articles after we automatically label them with their corresponding cluster-based types in a weakly supervised fashion. At the end, we used the predicted cluster-based types of a given mention as extra clues to improve the NED task.
- We introduced five different ways of cluster-based mention typing based on representing the context around a mention at three different levels. We showed that improved NED results are achieved when the different typing models are used together.
- In the candidate generation step of NED, we proposed using the candidates of the co-occurring mentions in the same document, which leads to higher gold recall values than the previously reported results. We publicly shared our tool and data

sets on the TABI Lab web site [29]

1.5. Thesis Overview

This thesis studies the application of weakly supervised learning on two NLP tasks: hashtag segmentation and named entity disambiguation. Before laying out our research and results on these two tasks, we first visit the background related to the research done in this thesis in Section 2. We define the tasks related to the studies performed in this thesis, such as word segmentation, clustering, and named entity disambiguation. Then, we explain the methods related to these tasks.

After describing the background, we present our research on hashtag segmentation in Section 3. Only a few studies have addressed this task in the NLP literature so far. We start by discussing these relevant studies. Then, we provide the details of our proposed rich feature-based approach for hashtag segmentation and describe how we extend it with context and language model-based approaches. In Section 3.3, we explain our automatic hashtag segmentation heuristic, which helps us obtain segmented versions of almost a million hashtags out of half a billion tweets. Before giving the results, in the Evaluation Metric section, we propose using F-score in addition to the accuracy metric, as we argue that partial results can be as important as predicting the exact segmentation. In the Experimental Results section, we show that we achieve the state-of-the-art results on two test sets. More importantly, we show that our weakly supervised approach to hashtag segmentation achieves better than or comparable results with respect to the model that is trained on regular tweet text. Now that we have the state-of-the-art hashtag segmentor in our hands, in Section 3.7, we explain our findings after we automatically segmented 60 million hashtags from SNAP tweet data set [30]. In those subsections, we not only study the internal grammatical structure of hashtags in detail, but also show that most of the time sentiment is trapped inside multi-word hashtags. Our findings indicate the need for a hashtag segmentor, in order to make use of hashtags more effectively while processing tweets.

Section 4 presents the second half of this thesis, which is our research on the named entity disambiguation task in conjunction with our newly proposed cluster-based mention typing. We again start by providing the latest related work on named entity disambiguation and mention typing. Then, we describe how we build our cluster-based mention typing model in Section 4.2. It starts with the explanation of different ways of representing a mention's context. Then, we describe how to represent named entities based on those different context models and do the clustering based on those representations. After clustering over five million named entities in Wikipedia and assigning the Cluster IDs as cluster-based types to those named entities, we explain how to train a model that can predict the cluster-based type of a named entity mention. To accomplish that, we show how we automatically generate training data from hyperlinked mentions in Wikipedia articles. After obtaining the mention typing model, in Section 4.3, we start explaining the components of our named entity disambiguation pipeline. First our state-of-the-art candidate generator comes, which outputs all possible named entities being referred to by a given mention. After that, we explain our ranking model which is based on a simple feed-forward neural network. Before getting into the results, we describe the details on the training and test data sets as well as how we tune the clustering for better mention typing. In Section 4.5, we first report the results for the mention typing model, then for the candidate generator, and finally for the disambiguation model. We also conduct a detailed analysis of the results with ablation tests.

Before concluding and giving the future directions in Section 6, Section 5 gives the detailed description of the tools that we developed during this thesis and made publicly available. These are the hashtag segmentor, named entity disambiguation related tools, and experiment result database toolkit called xDB.

Have a good read!

2. Background

2.1. Related Tasks

2.1.1. Word Segmentation

Word segmentation is one of the oldest tasks in the NLP literature. The problem that word segmentation solves is to correctly identify word boundaries in a given character string. It can be very challenging, especially for languages without explicit word boundary separators, such as Chinese and Japanese. In case of Semitic languages like Arabic and Hebrew, they have rich non-segmental multi-word tokens which makes the boundary decision harder. Even for languages like English, it can be misleading to rely on white spaces as word boundaries alone due to punctuation and even typos. To give an example, while writing tweets, which are restricted to be no longer than 140 characters, people tend to merge words (*e.g.* "doubledown") together in order to save space. Another application of word segmentation involves the recognition of words in hand-written documents [31].

Word segmentation is the first step before applying any high-level NLP, such as part-of-speech (POS) tagging, named entity recognition etc. It has been studied for a long time and today it is almost a solved problem. Early methods used for this task were as simple as using a dictionary to find words in a given sequence of characters. One of the best known dictionary-based methods is maximum matching and its variations [32] enhanced with heuristics, such as greedy algorithm. These methods basically try to find the longest matching word in the given input. While a dictionary-based method works well for certain cases, its performance depends on the coverage of the dictionary and how well the method handles ambiguous cases.

Later state-of-the-art systems employed statistical approaches, since they are in general more successful at handling unknown words and picking the best possible alternative when there is ambiguity. Such methods treat the problem as tagging, where

they assign a label to each character based on whether it indicates a word boundary. Conditional Random Fields (CRFs) [33], which is a commonly used method for sequence labeling was also used [34, 35]. Probabilistic methods such as HMM-based methods can learn the word boundaries by modeling character sequences. For Chinese word boundary detection, discriminative models such as word-based perceptron algorithm [36], as well as unsupervised methods [37, 38] are also used. Neural networks [39] and lazy learning approaches [40] have also been applied in this domain. In a more generic boundary detection study, corpus type frequency information along with maximum length frequency and entropy rate was also used [41].

Another approach for word segmentation is using a language model to find the best possible word segmentation of a given sequence of characters among many possible segmentations. Trigram LM to determine the best possible morpheme sequence for Arabic word segmentation was used [42]. In a web-scale word segmentation study, it was showed that they can unify and generalize word breaking techniques under the Bayesian minimum risk framework which can consider multiple document styles with minimal heuristics [43]. Their tool, namely Word Breaker achieves an accuracy of 97.18% with a trigram model.

In this thesis, we tackle the hashtag segmentation problem, which is in essence a word boundary detection task. Hashtag is a special token that consists of one or more words concatenated one after another without any white space along with special prefix pound character "#". Considering that they occur in tweets and the fact that the language of tweets can be quite noisy, it can be considered as a harder task than regular word boundary detection.

2.1.2. Named Entity Recognition

Named Entity Recognition (NER) is a task of identifying regions of text (*i.e.* mentions) corresponding to entities and categorizing them into a predefined list of types. Recognition of mentions in unstructured text is an important step for Information Extraction (IE) purposes. This is first realized in the Message Understanding Conference

(MUC) series. Starting in MUC-6, the research community focused on detecting four main types of named entities (NEs) in text: person, organization, location, as well as temporal expressions. In later systems, abbreviations and numeric expressions are also considered. In CoNLL-2002/2003 [14, 44], in addition to person, organization and location categories, miscellaneous category was added, which covered different concepts like project names, team names etc. More recently, ACE used categories like vehicles and facilities [45]. ACE also considered identifying temporal expressions as a separate task.

Early approaches for the NER task involved rule-based methods [46, 47]. Rule-based systems extensively use curated dictionaries and gazetteers. Rules are constructed in the form of finite state patterns. They used gazetteers, manually crafted proper name grammar rules as well as the discourse interpretation. However, building such rule-based systems are quite difficult. It requires expertise of curating necessary vocabularies as well as knowledge of grammar and other related linguistic aspects. Later on, they were outperformed by Machine Learning (ML) approaches. ML methods automatically learn a statistical model to classify given input based on labeled examples. First ML methods require manually annotated high quality training data to achieve their potential. Maximum Entropy (MaxEnt) model was used as well as dictionaries and gazetteers in order to create the features for modeling [48]. In various studies, multiple learning approaches have been tried together in combination, even mixed with rule-based approaches. For example, rules and MaxEnt model together were used [49]. Finite State Transducer (FST) based pattern matching rules in combination with MaxEnt and bigram-based HMM was employed [50]. They incorporated the rules into a constrained HMM network in order to remove errors due to lack of training corpus. In CoNLL-2003 evaluations [14], most of the participants used the MaxEnt-based approach in isolation or in combination with other methods. [13] proposed one of the most known and used systems in this field, which is called Stanford-NER. They use Conditional Random Fields (CRFs) [33] and replaced the Viterbi decoding with the simulated annealing in order to incorporate non-local features.

In order to model the NER, ML methods require large amount of training data. However, lack of annotated data made the research community reduce the annotation requirements by making use of large amounts of unlabeled data. Semi-supervised methods are such methods that start with small set of labeled data and incrementally expand this set by automatically extracting new data till certain threshold is reached. This is called bootstrapping. [51] proposed mutual bootstrapping. They started with small set of entities. By using large corpus, they collected context patterns that surround those known entities. Then those contexts are ranked and used to find new entities. Similar to [51], [52] use syntax information like subject-object connection to discover more accurate contextual patterns around the entities. [53] also employed mutual bootstrapping approach to detect named entities from 100 million web page corpus. Unsupervised approaches have been considered as well. One of the first studies of unsupervised techniques on NER task was done by [54]. Instead of using high number of rules or large training data, they only consider 7 simple seed rules. [55] introduced a system called KNOWITALL, which is an unsupervised, domain-independent information extraction system. It uses 8 pattern rules to extract named entities from the web pages. In [56], they proposed a named-entity recognition system which combines named entity extraction and named-entity disambiguation in an unsupervised setup. By processing semi-structured HTML formatted web pages, they automatically created a large gazetteer of named entities. In [57], they applied existing hyponyms/hypernyms identification method to detect the hypernyms of sequence of capitalized words. They use search query patterns to retrieve web pages that suggest the hypernym of looked up word sequence. For example, in a pattern like "X such as Y", X can be considered as hypernym of Y, as in the case of "city such as Istanbul". By retrieving large amounts of web pages containing such pattern instances, one can identify the hypernym correctly.

In last 5 years, deep learning approaches took over the NLP landscape by achieving state-of-the-art results in every applicable field, including image processing, speech recognition, machine translation, language modeling, and sequence labelling, which includes NER. [3] presented groundbreaking study on applying deep learning on various NLP tasks including NER. They employed Convolutional Neural Network (CNN) over a sequence of word embeddings with CRF model on top. While success of deep

learning at NER task started with the hybrid of CNN and CRF models in [3], more obvious solution is to replace CNN with the model that suited for sequence labelling. Special type of neural networks called Recurrent Neural Network (RNN) is designed to handle model time series based problems, like NER. However, they suffer from vanishing gradient problem, which prevent them from capturing long-range relations in the input. Hence, its variant called Long Short-Term Memory (LSTM) is currently mostly used deep learning architecture for NER. In fact, the best results achieved by combining two LSTMs to consider both previous and future context in the input, which are called Bidirectional LSTM (BiLSTM). [58] replaced CNN of [3] with BiLSTM and also used hand-crafted spelling features. [59] proposed hybrid of BiLSTM and CRF model like [60] but they manage to model both character- and word-level information with BiLSTM. [61] combined BiLSTM, CNN and CRF models together and used CNNs specifically for character-level representation, instead of utilizing BiLSTM for that like [59].

In this thesis, we did not do any named entity recognition or use any recognizer. Instead, we use the already recognized mentions in our named entity disambiguation studies. In any case, it is an important precursory step for the disambiguation task.

2.1.3. Named Entity Disambiguation

Named Entity Disambiguation (NED) is the task of disambiguating entity mentions in text by associating them with entries in a provided knowledge base, like Wikipedia and DBPedia. It was first introduced in 2009 in the Knowledge Base Population track of the Text Analysis Conference (TAC-KBP) [62].

A typical entity disambiguation system assumes that the named entities mentioned in the text are already recognized by the NER step. Hence, it starts with recognized named entities as well as their predicted entity types, such as person. Having said that, traditional entity disambiguation involves two steps: candidate detection and candidate selection. In the candidate detection step, a list of all possible candidate named entities from reference knowledge base are generated. In this process, basically

surface form of the mention is used to as a search query to retrieve all named entities that are named same or similarly. The second stage is to decide which candidate is the best match for that mention. If there is no candidate or none of the candidates are strong enough for the selection, then that mention is disambiguated as NIL entry. This means that there is no corresponding entry at KB for that particular mention. Normally NIL mentions with the similar surface forms are clustered together in order to reflect the fact that they refer to the same unknown entity. This is also called NIL clustering.

Aside from the candidate generation, NED is mainly considered as a ranking task as we are expected to choose the most likely candidate that mention is referring to. To do this ranking, the similarity between the context of a mention and the document associated with a candidate entity (e.g. its page in Wikipedia) is used. While applying this ranking, there are two main approaches applied in the literature: local approaches where every mention is considered independently from others in the context. Ranking is done by considering whether each candidate is the one we are looking for or not. Score produced by this binary classification is used for ranking. Some studies [63] use local statistics about each mention and considered candidate entity. In case of global approaches, all mentions are disambiguated together, simultaneously in order to achieve the highest cohesion score in the context.

One of the important aspects of both local and global approaches is to be able to measure the semantic similarity between the context and candidate entity. The more similar candidate entity and the context are, the more likely that that entity is mentioned in that context. Various techniques have been proposed in the literature. [64] directly optimizes document and entity representations for a given similarity measure, instead of utilizing simple similarity measures such as cosine similarity etc. and their disjoint combinations. They use two-stage approach where first stage is Stacked Denoising Auto-encoder to discover general concept encodings and second stage is to fine-tune the parameters based on selected similarity measure as an optimization criteria. Their hierarchical model allows them to model and context and entity with different levels of abstraction. They train their model on Wikipedia data set and

achieved then state-of-the-art results on TAC 2010 and AIDA data sets. [65] use Convolutional Neural Networks (CNNs) to capture the similarity between context of the mention and candidate target entity. In the literature, CNNs are proven to be good at modeling sentences and documents for classification task. [65] use CNNs with different context granularities and model the topic semantics of context and entity better. [66] created embeddings for mention, context, and entity separately, and used a neural network framework to integrate these representations for entity disambiguation. In [67], they first learn low-dimensional embeddings for entities and words by jointly modelling knowledge base and text in the same vector space and then utilize these embeddings in a two-layer disambiguation model. [68] focused on learning entity embedding and selectively leveraging contexts through a local attention mechanism over local context windows. [69] proposed a method, which jointly maps words and entities into the same continuous vector space. Their extension to skip-gram model employs two models: KB Graph model and anchor context model. The first model learns the relatedness of entities using the link information in the KB. The second model aims to align vectors so that similar words and entities occur close to one another in the same vector space by looking at KB anchors and their context words. [70] proposed attention-like mechanisms for coherence, where the evidence for each candidate is based on a small set of strong relations, rather than relations to all other entities in the document. Their multi-focal attention model is expected to enforce coherence.

In case of global approach, to achieve the highest cohesion while resolving all mentions in the context [71]. This is based on the assumption that all mentioned entities in the context occurs in the same context, thus related to each other. This is also called topical coherence. In such type of approaches, all mentions are resolved together collectively. In the literature, multiple methods have been proposed over the years. In [72], their method builds a weighted graph of mentions and candidate entities, and computes a dense subgraph that approximates the best joint mention-entity mapping. Many other works are based on the Random Walk [73] and PageRank algorithm [74]. Scores of entities obtained with these algorithms will be used to select the matching entity for each mention in the context. In [75], their method called Iterative Substitution jointly optimizes the identification of the mapping entities while

maximizing the sum of pair-wise semantic similarities between all linked entities. In more recent study [76], fast linking is achieved by applying the well-known Forward-Backward algorithm [77]. Their approach only considers the adjacent assignments in coherence optimization. Although graph-based approaches are shown to produce robust and competitive performance, they are computationally expensive because the graph may contain hundreds of vertices for documents with multiple mentions. Unlike previous studies, [78] used the position of the mention in the context and avoided using bag-of-words like approach by modeling the context with Long short- term Memory (LSTM) and attention mechanism. On top of that, they also propose a pair-linking algorithm, which iteratively identifies and resolves pairs of mentions, starting from the most confident pair. They reported that Pair-Linking achieves comparable or even better results than the state-of-the-art collective linking algorithms. [79] use memory network [80] based model which leverages the importance of context words in an explicit way, unlike other neural models such as RNNs and CNNs. Their approach used two external memories; one to represent context words, other to represent descriptive words in entity's Wikipedia page. They regard context words as the memory and the mention as the query to find important evidences from the memory. They trained their system on annotated data collected from Wikipedia and tested on TAC-KBL 2010 evaluation data set.

2.1.4. Mention Typing

Recall that named entity recognizer detects where mentioned named entity is located in given text and also categorizes them into a predefined list of types. In the early days, only three major types [81] are considered: those are person, location, and organization. Even though we see the addition of the miscellaneous type in CoNLL-03 [14] and later on types of geopolitical entities, weapons, vehicles and facilities [45], it was not sufficiently large enough to make the NER effective for the relation extraction. Because there can be hundreds or thousands of different relations between named entities and that requires being able to categorize named entities of many types. To solve this problem, [20] introduced the task of fine-grained entity recognition which categorizes mentions into 112 unique groups based on Freebase types. For example,

instead of just identifying person type, their approach looked to identify actor, doctor and 13 other specific types of person. Apart from recognizing the mention in given text, the task of identifying its type is called mention typing. In other words, mention typing assumes that mentioned named entities in given text are already recognized beforehand.

One of the important components of mention typing is the taxonomy of types itself. While early studies like Ling and Weld relied on hand-picked set of types, later studies expand the type set even further. [21] derived a very fine-grained type taxonomy from YAGO [22] based on a mapping between Wikipedia categories and WordNet [82] synsets. Their taxonomy contains a large hierarchy of 505 types. More recent studies used tens of thousands of types [83] extracted from the Wikipedia category hierarchy. In another study, [84] assumed the type set a open set by allowing a free-form phrase to be the predicted type. Apart from the last approach, most of the studies in this task used manually curated type taxonomy, such as Wikipedia categories. Even though it requires tedious human work to curate such a large taxonomy, one important advantage of using Wikipedia categories is that it also provides a huge set of annotated mentions of named entities in Wikipedia articles in the form of hyperlinks. Being able to know the possible types of named entities and having many examples of mentions allow researchers to use that data as a training data to model the mention typing task.

Note that in the literature, mention typing is also referred as mention-level entity typing or fine-grained entity typing. However, the term "entity typing" is also used to refer to the task of identifying all possible types of a named entity given a large set of corpus. This task is specifically called corpus-level entity typing. It is used for knowledge base completion [85] which is the task of automatically inferring missing facts by making use of the information already present in the knowledge base.

2.1.5. Clustering

Clustering is a task that groups similar objects into sets known as clusters so that objects in the same cluster are similar to each other, while objects in different

clusters are dissimilar. It is very popular technique in the NLP literature, specifically for two major types of use: exploratory data analysis (EDA) and generalization. EDA is technique to discover the properties of the data with the minimal knowledge about that data and summarize those main properties in the form of visual methods. Visualization of clustered data points may give clues about the underlying structure of that data. The second type of use is the generalization. In this case, as we group similar items together, we can start to generalize certain information from which we know about some members of clusters to others in the same cluster.

There are number of different clustering algorithm types depending on their underlying methodology. However, most used ones are called partitional and hierarchical clusterings. Partitional clustering partitions the entire data set into either a predetermined or an automatically derived number of clusters. It tries to create high quality clusters according to selected criterion function which measures the similarity or distance. The most famous clustering of this type is K-means clustering, which is the one we use in our experiments. On the other hand, hierarchical clustering algorithms create clusters with a hierarchical relation between them. Each node represents the cluster that contains all the objects of its descendants. Depending on whether we use top-down or bottom-up approach, there are two types of hierarchical clustering, namely agglomerative and divisive clustering. Agglomerative clustering is a greedy bottom-up approach and starts with a separate cluster for each item. In each step, the two most similar clusters are merged into a new cluster. In case of divisive clustering, it is a top-down approach and requires a method for splitting a cluster until singleton cluster. Hierarchical clustering is good at recognizing underlying hierarchical structure of the data, if exists. However, this advantage comes with a cost of computational efficiency, which has a time complexity of $O(n^3)$. Whereas, K-means has a linear time complexity. Having said that, partitional clustering is recommended when dealing with the high number of data points.

Apart from different clustering methods based on underlying methodology, they are also grouped into two types depending on how we assign data points to clusters; those are hard clustering or soft clustering. In hard clustering, each data point either

belongs to a cluster completely or not. In case of soft clustering, instead of assigning each data point to a distinct cluster, a probability of that data point to be in each cluster is calculated. In our experiments, we use hard clustering due to efficiency reasons.

The most advantageous side of clustering is that it does not need any annotated data. Hence, it is an unsupervised method. The only needed component is a similarity measure so as to be able to calculate the level of similarity between two data points. There are various similarity measures. Euclidean distance is considered as the standard metric for the K-means algorithm [86]. It is the ordinary distance between two data points. It calculates the root of square differences between the coordinates of a pair of objects. Another measure is cosine distance (or similarity) which calculates the cosine of the angle between two vectors. The higher the degree gets, the more distant two vectors are from each other. There are also other measures like Jaccard distance, Manhattan distance, Chebyshev distance, etc.

When it comes to its application in the literature, their unsupervised nature help In early studies, clustering has been mostly used for grouping documents for better Information Retrieval (IR) and extraction. For example, in case of cluster-based retrieval, it is hypothesized that similar documents match the same information needs so retrieved documents is listed based on retrieved clusters [87]. Other studies categorized named entities in order to improve document retrieval [88, 89]. Later on, studies introduced clustering words so that instead of dealing with hundreds of thousands words, we map them to a few thousand clusters and use those clusters. This particular idea is used specifically in class-based language models. [90] introduced the hypothesis that similar words have similar distributions of words to their immediate left and right, which is also called Brown clustering to be described in Section 2.2.6.

In this thesis, we proposed a cluster-based mention typing approach, where we cluster named entities in our KB based on their contextual similarity as described in Section 4.2.3. Then we use each cluster as a automatically generated named entity type. We train a mention typing model based on these cluster-based types and use

the predictions of the mention typing model as an extra clue for the named entity disambiguation task.

2.2. Related Methods

2.2.1. Hidden Markov Models

A Hidden Markov Model (HMM) [91] is a statistical Markov model. Markov models are used to model randomly changing environments. It relies on the basic assumption that future state of the environment only depends on the current state of the environment rather than its past states. It is called markov assumption. This helps to reduces the complexity of the model of the environment by ignoring any complicated dependency. In case of HMMs, we assume there are unobservable or hidden states. HMMs are used to model time-based sequential series. We define the process in terms of states and observations. Like the Markov assumption, we also have observation independence assumption in HMMs, which states that current observation depends only on the current state that outputs that observation. Even though the current state of the system is not directly observable, the output is considered to be visible. Hence, by observing these outputs, HMM can assign a probability to that sequence of outputs without knowing the internal states. This way, we can predict the best possible state sequence that generates that output.

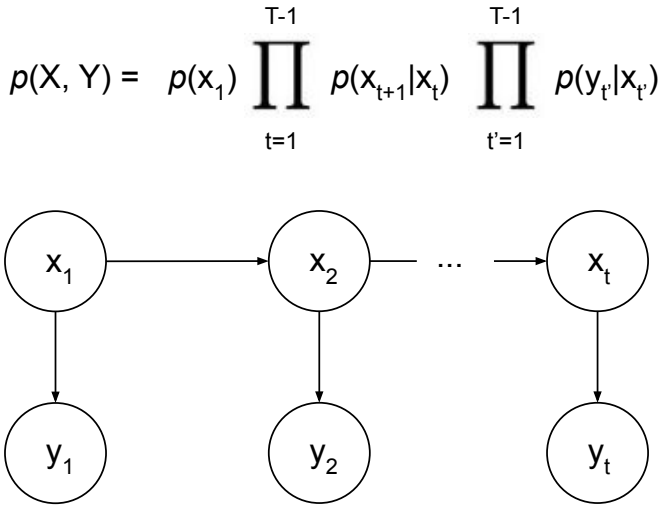

Figure 2.1. Sample HMM shown for joint probability calculation, p(X,Y).

Figure 2.1 depicts a sample HMM model to calculate $p(X, Y)$. X's are states and Y's are observations. We calculate the p(X,Y) by multiplying initial prior state probability, transition probabilities $p(x_{t+1}|x_t)$ and observation probabilities $p(y_{t'}|x_{t'})$ at each state together. Note that transition and observation probabilities follow the markov and observation independence assumptions, respectively. Figure 2.1 shows one hidden state with one observation. However, for an HMM with N hidden states and an observation sequence of T observations, there are N^T possible hidden sequences. Depending on the values of N and T, it may not be feasible to calculate the joint probability. Instead of considering all possible cases, an algorithm called Forward algorithm is used, which is written based on the dynamic programming paradigm and reduces the complexity to $O(N^2T)$. As it goes over the observation sequence, it keeps track of intermediate values. At the end, it then computes the observation probability by summing over the probabilities of all possible hidden state paths that could generate the observation sequence. The task of obtaining the sequence of states from the sequence of observations is called decoding. For HMMs, Viterbi algorithm [92] is used for decoding, which is again based on the dynamic programming.

To give a real life example for HMMs within the context of the word segmentation task, characters of the word can be represented as the observations. And the situation which is whether there is a boundary or not at the observation point (i.e. before corresponding character) can be represented as the state since we do not know if there is a boundary or not. During our hashtag segmentation studies, we also implemented a HMM-based segmentation model. Instead of representing each character as a state, we expand each state with two previous characters, making it a 3-gram character model.

2.2.2. Language Models and OpenFST

A language model (LM), or statistical language model to be precise, defines a probability distribution over sequences of words. Its main functionality is to calculate the likelihood of the sequence of words in given input text. This capability makes it very valuable tool for number of tasks, such as machine translation, speech recognition, spell checking etc. Being able to calculate the probability of a word sequence also

means that LMs can be use to determine the best next word coming after that word sequence. This becomes especially useful in natural language generation related tasks. LM calculates the joint probability of all words in the sequence which is not feasible as the length of that sequence increases. Hence at its core it uses various assumptions and heuristics like the chain rule in probability theory and Markov assumption as mentioned in previous section. The main parameter of the language model is the length of history we desire to consider at the calculations. If we consider none of the any previous words, it is called unigram model, which is the most basic LM model. However, in order to calculate the probability more accurately, we need to consider as much history as possible. Modern LMs use up to 5-6 previous words, which makes them very powerful yet hard to calculate models.

In our experiments, we use LM in our hashtag segmentation studies. With the help of OpenFST tool, we represent all the possible segmentation of hashtags in a graph and then use a LM to score this graph and pick the highest scored path. This path gives us the best possible segmentation calculated by the LM.

OpenFST [93] is a open-source library to construct and use weighted finite-state transducers (FSTs). FST is a finite automaton which consists of set of states and arcs connecting those states. Specific to FST, each arc includes input and output labels where input label is acted as constraint to transition from one state to another by using that arc. When it is used, it produces output label on the same arc. In case of weighted FSTs, each arc is also associated with a weight. In the literature, OpenFST is frequently used tool to work with LMs.

We implemented our OpenFST approach in C++ by using corresponding libraries of the OpenFST. Our implementation requires a LM and a lexicon in FST format. We use SRILM tool in order to obtain the language model on given training data set. However, it outputs the model in its own SRI format. By using make-ngram-pfsg tool, we first convert it to AT&T format and then use pfsg-to-fsm to get it in FSM format. As we print out the arcs in plain text, we are able to create FST out of that plain text. In case of lexicon, we extract it from the data set that we train our LM.

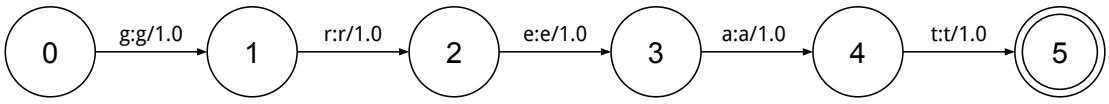

Figure 2.2. FST representation of word 'great'. Arc weights are 1.0.

When input text with no spaces is given to our OpenFST program, it first creates a simple FST where each character of the input is converted to an arc. Hence, the input text is represented as single line FST. To give an example, Figure 2.2 depicts a FST of word "great". After converting input to an FST, we compose this FST with our lexicon FST. This process matches character arc sequences with words in the lexicon and creates word arcs on top of sequence of character arcs. The next step is to compose the expanded FST with LM FST. This assigns score to word arc sequences so that we can choose the highest scored paths. As we traverse the paths, we extract the highest scored segmentations.

2.2.3. Maximum Entropy Model

Statistical models like HMMs, Naive Bayes etc. assume that features are independent from each other. Even though this assumption simplifies the estimation of feature weights, in reality interaction between features can be complex. And training data may not include all possible interactions. Hence the model starts to fill or predict the missing interactions. Complexity of the model may increase when it starts to make assumptions about that missing data. Maximum Entropy (MaxEnt) Models [94] can handle feature interactions without falling into that complexity trap. Compared to other models, MaxEnt models make no assumption about missing or unknown part of the data. It basically makes a minimum assumption about the prior distribution over the data. It achieves this by assuming uniform distribution for unknown cases. That maximizes the uncertainty, that is entropy $H(p)$ of the model, hence the term maximum entropy.

$$H(p) = -\sum_{i=1}^{n} p_i log p_i \tag{2.1}$$

$$\begin{aligned} & p_i \geq 0 \\ & \sum_{i=1}^{n} p_i = 1 \\ & \sum_{i=1}^{n} p_i r_{ij} = \alpha_j, 1 \leq j \leq m \end{aligned} \tag{2.2}$$

Shannon [95] introduced the definition of the entropy $H(p)$, which is given in Equation 2.1 subject to the Conditions 2.2. It is a way to estimate the average minimum number of bits needed to encode a string of symbols based on the frequency of the symbols. The maximum entropy principle is a problem that is solved with the constrained optimization. To solve, we apply the method of Lagrange multipliers on $H(p)$ and then calculate the derivative of that with respect to p_i. The result is maximum entropy distribution given in Equation 2.3 where $\lambda_0, \lambda_1, ..., \lambda_m$ are chosen based on the Conditions 2.4.

$$p^* = \arg\max_p H(p) p_i^* = \frac{e^{\sum_{j=1}^{m} \lambda_j r_{ij}}}{e^{1-\lambda_o}} \tag{2.3}$$

$$\begin{aligned} & p_i^* \geq 0 \\ & \sum_{i=1}^{n} p_i^* = 1 \\ & \sum_{i=1}^{n} p_i^* r_{ij} = \alpha_j, 1 \leq j \leq m \end{aligned} \tag{2.4}$$

In our research, we use maximum entropy model in our hashtag segmentation studies. We specifically use C++ version of the maxent toolkit developed by Le Zhang [96].

2.2.4. Word Embeddings and word2vec

From computer's point-of-view, words themselves are no more than symbols. Symbols are good tool for labeling concepts and referring to things when needed. However, they are very hard to operate computationally. We can only compare them by looking at the differences between their surface forms. In order to achieve high levels of natural language processing, we should be able to compare words at the semantic level, measure their similarity to each other, and even do other computational operations like adding or subtracting their meanings and taking the average meaning of set of words.

The motivation behind representing words in terms of vectors comes from two fields, namely Information Retrieval (IR) and Language Modeling (LM). IR research field involves searching relevant documents to a given query. Hence being able to represent documents and query in the same representation format is essential for the success. Vector Space Model (VSM) [97] came to the light to achieve this. VSM is a encoding procedure where each document in a collection is represented by a t dimensional vector and each element represents a distinct term contained in that document. These elements may be binary or real numbers. With the introduction of vector representation, one can apply mathematical operations like calculating the similarity between document vectors or between document and query vectors. VSM is the earliest method of representing words as vectors, yet it is not that informative and efficient.

The second area that is interested in word representations is LM. As discussed in Section 2.2.2, LMs defines a probability distribution over sequences of words and calculate the probability of seeing the next word after given sequence o words. It is basically calculated with the Maximum Likelihood Estimation (MLE), over all words in the vocabulary. However, as the size of the vocabulary gets highers, problems tend to occur. To cope with this, different smoothing techniques and method called backing-off [98] have been introduced over the years. Another solution was to cluster words [90] and generalize words with corresponding classes so that required number of calculations is reduced significantly. More recently, neural networks [99–101] have been used to obtain word representations in vectors, which is better known as word

embeddings today. Despite recent ground-breaking improvements in this area, it is worth mentioning that using neural networks to obtain word representations was not that new. In 1986, [102] used the early neural networks to calculate some form of word representation.

Today, *word2vec* [101] is one of the most used tools to obtain word embeddings, in addition to GloVe [103] and FastText [104]. Word2vec uses a neural network approach to calculate the word embeddings. However, unlike recent deep neural networks which contain many number of hidden layers, word2vec uses a single hidden layer as depicted in Figure 2.3. It takes in the vector representation of a single word, passes it through that hidden layer and uses a softmax activation layer at the end to predict the nearby word which is located inside the boundary of a window with predefined width. This model is called skip-gram model. This approach is based on the notion that "you shall know a word by the company it keeps" [105]. Having said that, words that occur in similar context is expected to have similar word representations in the vector space.

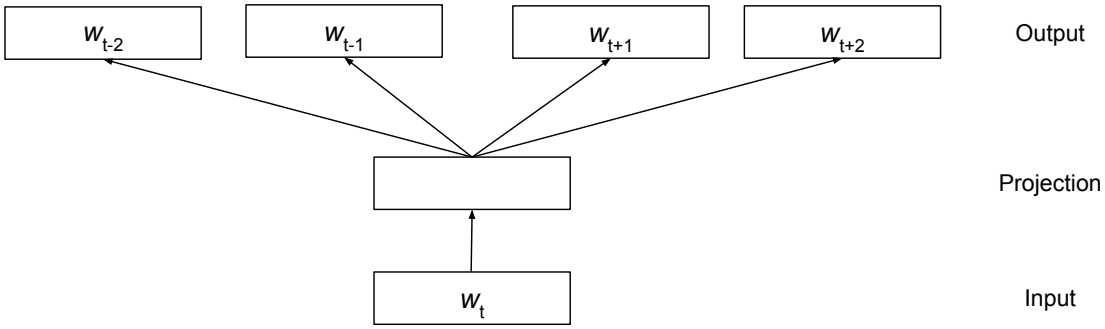

Figure 2.3. Word2vec's Neural Network with the Single Hidden Layer.

The training objective of the skip-gram model is to calculate the word representations that are good at predicting the surround words in a given sentence. In formal way, the objective function is given in Equation 2.5 given a sequence of words $w_1, w_2, ..., w_T$, where c is the window size to be considered around w_t.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} log\ p(w_{t+j}|w_t) \tag{2.5}$$

The final layer of the neural network is the softmax layer as formulated in Equation 2.6, where v_w and $v_w^{'}$ are the input and output vector representations of the word w respectively and W is the number of words in the vocabulary. However, it is not practical to use this softmax function as its complexity is proportional to W which can be a very large number in the order of 10^5 to 10^6. One of the of word2vec is the introduction of two heuristic methods called hierarchical softmax and negative sampling. Thanks to these methods, word2vec is able to reduce to complexity of the algorithm to acceptable range.

$$p(w_O|w_I) = \frac{exp({v_{w_O}^{'}}^T v_{w_I})}{\sum_{w=1}^{W} exp({v_w^{'}}^T v_{w_I})} \tag{2.6}$$

The word2vec tool accepts raw text as input, which makes it very easy to use. It assigns a randomly initialized vector to each seen word and iterates through the input until the convergence. At the end, it outputs tuned word embeddings for each word. The size of the embedding vector is given as parameter which is chosen between 50-1000 depending on the intended purpose of the embeddings. The important parameter of the tool is the window size. Usually 5-10 is chosen as the window size. The smaller it is set, the more syntax oriented similarity is encoded into the word embeddings. As it gets higher, word embeddings convey more semantic similarity as the considered context around the center word gets larger. In our experiments with the named entity disambiguation, we use word2vec extensively. In addition to that, we also use altered version called *word2vecf* [106] which works the same way but accepts the input in two columns rather than raw text.

2.2.5. K-means Clustering

K-means clustering is one of the most popular unsupervised machine learning algorithms. it aims to group similar data points together into fixed number of clusters. "K" in the K-means corresponds to that fixed number. K is either predefined given number or it can be determined after testing multiple K values and picking the right one based on some criteria, such as the one that gives the minimum mean distance between each data point and the centroid of its assigned cluster.

$$c^{(i)} = \arg\min_{x} \left\| x^{(i)} - \mu_j \right\|^2 \tag{2.7}$$

$$\mu_j = \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}} \tag{2.8}$$

K-means algorithm starts with assigning each data point $x^{(i)}$ to randomly selected cluster $c^{(i)}$. The cluster centroid $\mu_j \in \mathbb{R}^n$ is the arithmetic mean position of the cluster. The algorithm then runs iteratively and looks for the nearest centroid for each data point and re-assigns it to that centroid, if it is not already assigned to that. This way, it optimizes the positions of the centroids after each full pass over the data points. It halts when either predefined number of iterations have been achieved or no more than significant number of data points have been re-assigned at the last iteration.

2.2.6. Brown Clustering

Brown clustering [90] is a hierarchical agglomerative clustering method based on distributional information. It is based on the hypothesis that similar words have similar distributions of words to their immediate left and right. It uses a binary merging

criterion based on the log-probability of a given tokens under a class-based language model. Class-based language model considers the fact that each token is a member of a class. Brown clustering uses Average Mutual Information (AMI) in Equation 2.9 as the optimization function. AMI measures the information contained in random variable Y about the random variable X. Brown clustering merges clusters such that the merge results in the least loss in global mutual information.

$$I(X;Y) = \sum_{X,Y \in A} P(X,Y) log(\frac{P(X,Y)}{P(X)P(Y)}) \tag{2.9}$$

In our experiments, we use brown clustering in the named entity disambiguation studies. We obtained brown clusters for named entities in Wikipedia. We did this by collecting a list of named entity ids that occur in each Wikipedia article. This represents co-occurrence relation among entities. At the end we get large file which contains rows of entity id listing, each row corresponding to one Wikipedia article. We used fast C++ implementation of Brown clustering [107] in order to obtain the entity clusters.

2.2.7. Deep Neural Networks

Neural Networks mimic the way human brain works. They take in an input and transform it through a series of hidden layers. Each hidden layer consists of a set of nodes and those nodes are connected to other nodes in the previous layer. Whole interconnected structure resembles and functions like a network of neurons in our brain. Like neurons, these nodes receive an input value and transforms it into a another value by multiplying it with the weights, which amplifies or dampens the input value. Multiplication of input values with weights are summed and the final value goes into the activation function where it is determined whether input signal is allowed to go through. A typical neural network consists of three types of layers as

depicted in Figure 2.4; input and output layers and number of hidden layers inbetween. The higher the number of hidden layers is, the deeper the network gets. As the neural network is being trained, it tunes its weights by going through each training instance with the objective of minimizing the difference between the real output and predicted output. This difference is calculated at the output layer and it is redirected into the neural network in backward direction, which is called back-propagation. As the difference follows backward from upper layers to the lower layers, weights of the nodes inside layers are adjusted to reduce the future observed difference at the output. This forward-backward process continues until convergence.

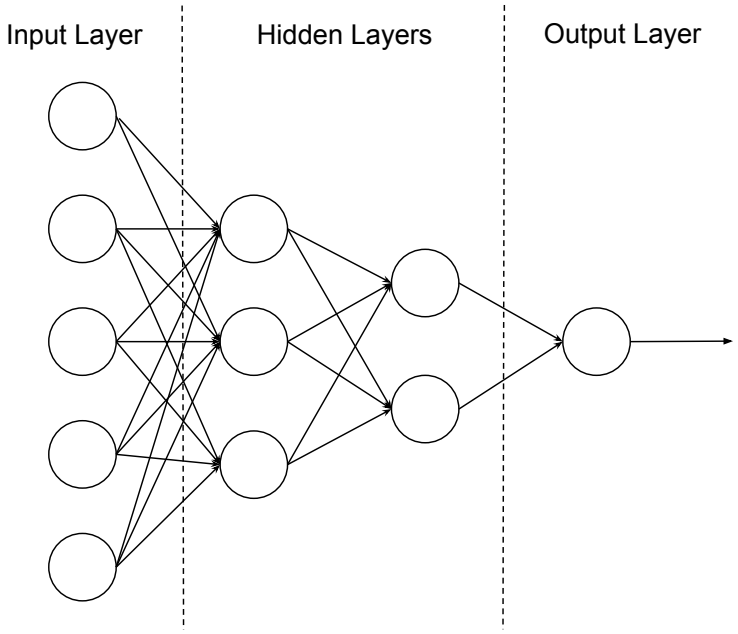

Figure 2.4. Simple neural network.

In this thesis studies, we use neural networks while doing named entity disambiguation. While there are various types of architectures available in the neural network repertoire, we used specifically Long-short Term Memory (LSTM) networks which are derived from the Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). In the following subsections, we describe how these two networks work.

2.2.7.1. Recurrent Neural Networks. A recurrent neural network is a type of neural network architecture specifically suited for modeling sequential data. As shown in Figure 2.5, they are networks with loops which allows information to be carried from

one step of the network to the next. In other words, loops allows information persist within the network.

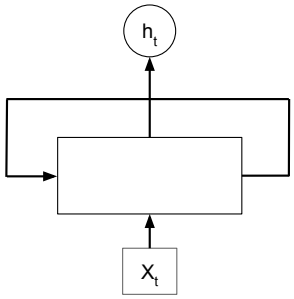

Figure 2.5. Recurrent neural network having a loop.

In order to see how recurrency works, you can imagine the network at each time step as shown in Figure 2.6. This can also be seen as multiple copies of the same network at different time points. This chain-like structure is inherently related to sequences hence their ability to model sequences. Input and even output of RNNs can be sequence of vectors.

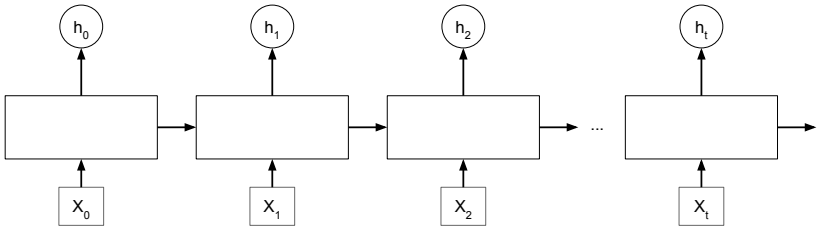

Figure 2.6. An unrolled recurrent neural network.

In more formal language, at each time step t, an RNN takes the input vector $x_t \in R^N$ and the hidden state vector $h_{t-1} \in R^M$ from the previous time step and produces the next hidden state h_t by applying the following recursive operation:

$$h_t = f(Wx_t + Uh_{t-1} + b) \tag{2.10}$$

where $W \in R^{M \times N}$, $U \in R^{M \times M}$, and $b \in R^M$ are parameters. f is a non-linear activation function applied element-wise. In theory, RNNs can compress and represent all historical information up to time t with the hidden state h_t. However, in practice it was shown that vanilla RNNs cannot learn long-range dependencies due to vanishing or exploding gradients [108] because as we multiply the gradient of the typical activation

function (eg. tanh or sigmoid) many times, gradient is lost along the way. To cope with this, variant of RNN called Long short-term Memory (LSTM) network was introduced [109]. LSTM does not use activation function for hidden state, which eliminates the vanishing gradient. Instead, it contains three multiplicative gates which control how much information to forget and to pass on to the next time step. Note that LSTM still experience exploding gradient which can be handled by optimization strategies like gradient clipping. Following details how to update LSTM cell at time t:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ g_t &= tanh(W_g x_t + U_g h_{t-1} + b_g) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ h_t &= o_t \odot tanh(c_t) \end{aligned} \tag{2.11}$$

Here $\sigma(\cdot)$ and $tanh(\cdot)$ are the element-wise logistic sigmoid and hyperbolic tangent functions respectively. $\odot$ is the element-wise product, and i_t, f_t, o_t are referred to as input, forget, and output gates, respectively, all having the same size as the hidden vector h. c_t is cell state and LSTM can add or remove information to the cell state by gates. Sigmoid function of each gate outputs numbers between zero and one, deciding how much information can pass through the gate. At t=1, h_0 and c_0 are initialized to zero. W_i, W_f, W_g and W_o are weight matrices of different gates for input x_t, while U_i, U_f, U_g and U_o are weight matrices for hidden state h_{t-1}. Whereas b_i, b_f, b_g and b_o are bias vectors.

For sequential tagging tasks, when we have access to both past (left) and future (right) context, we can make use of both context by using what is called bidirectional LSTM (BiLSTM) network [110]. It works by giving the input sequence backwards and forwards to two hidden states separately in order to capture past and future information, respectively. Then those two hidden states are combined into final output.

It is also possible to extend RNN or LSTM by putting multiple layers on top of each other and using lower layer's hidden state as the input for the upper layer. Such

stacked layers increases the capability of the architecture to learn complex relations. However optimum number of layers depends on the problem at hand.

2.2.7.2. Convoluational Neural Networks. The Convolutional Neural Network (CNN) is a specialized type of neural network architecture designed for working specifically with two-dimensional data, such as images, while it is still possible to apply them on one-dimensional and three-dimensional data. Their main application area is the image recognition which involves classifying objects in images and even pinpointing their exact location. For example, its earliest application is the Optical Character Recognition (OCR) to digitize written documents.

CNNs are different than traditional neural networks not only due to being able to handle multi-dimensional input. The nodes in the layers are not necessarily connected to every other node in the previous layer, but only to a small region of it. Depicted in Figure 2.7, a simple CNN consists of two major parts; feature extraction part which includes convolution and pooling operations and fully-connected part at the end to do the actual desired classification.

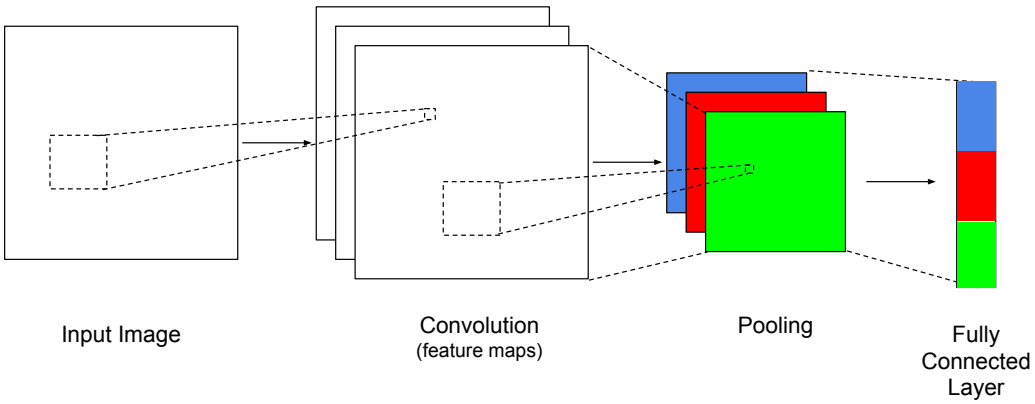

Figure 2.7. A Simple Convoluational Neural Network.

CNNs starts with applying the convolution on the input, which is the mathematical combination of two functions to produce the third function. In case of CNN, it runs the filter (or kernel) on the input and produces a feature map. This process involves sliding the two-dimensional filter over the two-dimensional image and at each location a matrix multiplication is performed and results are gathered inside another two dimen-

sional vector, which is called the feature map. As in case of regular neural networks, activation function can be used to make the output of convolution non-linear. *TanH* and $ReLU$ are the two of the most used activation functions in the literature. *TanH* squash the values into a range between -1 and 1, whereas $ReLU$ maps the negative values to zero and leave positive values as they are.

The second operator of the feature extraction part of the CNN is pooling layer, which is also called downsampling or subsampling. It is responsible for reducing the spatial size of the feature map. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model. The most used pooling function is maxpooling which returns the maximum value from the portion of the image covered by the kernel. Only the part of the images that shows the strongest correlation to each feature is selected by the maxpooling. There can be number of convolution and pooling layer pairs on top of each other in a CNN.

After series of convolution and pooling layers, one or more fully connected layers are placed on top of them. These layers are like regular neural network layers, each node in the layer is connected to every node in the previous layer. This part is responsible for using the extracted features from the first part to classify the input.

While applying CNN on any task, there are number of parameters to choose. Kernel size of the convolutions, number of filters to use, stride (the size of steps while sliding filter), padding at the edges, activation function are the major parameters for the convolution. There are also different pooling methods such as average pooling. The number of convolution and pooling layers on top of each other is also another parameter, not to mention each such layer can have its own customized parameter settings. Finally, the number of fully connected layers at the top and number of nodes in each of them are the other set of parameters to decide.

CNNs are not only used with images. They have been extensively used for text classification, sentiment detection etc. in the NLP literature. This is possible since a sequence of words forms a two dimensional input which consists of time and word embedding dimensions. In our named entity disambiguation experiments, we use CNN in order to characterize mentions, which is also defined as a classification task. We input sequence of words at the left and right context of the mention into two CNNs and connect them together with the another fully connected layer at the top as described in Section 4.2.6.

3. Segmenting Hashtags

After one year Twitter was debut in 2006, on August 23, 2007 at 10:25pm, Chris Messina suggested something that no one ever has suggested before. He tweeted "how do you feel about using # (pound) for groups. As in #barcamp [msg]?" This was the birth of hashtags. Their existence came out of necessity to tag channels in Twitter so that people can filter those tweets that are labeled by specific hashtags. Since then, we use hashtags not only to label channels, but also to convey the actual message that we want people to hear. When big events happen, Twitter users get coordinated and use hashtags that are specific to that event. That way, frequently mentioned hashtags in a short time period show up in the trending hashtags list, thus the messages that these hashtags convey. Some hashtags are short and compact, while others, especially the ones that convey some message, can be long, particularly when they are formed from clauses or complete sentences. *#IfTheyGunnedMeDown*, *#TwitterIsBlockedInTurkey*, *#BringBackOurGirls* are example hashtags that were seen in the trending lists in the past years. As these examples suggest, in order to do complete social network content analysis, we should be able to segment hashtags into their original words and unleash their content for Natural Language Processing (NLP) tasks such as sentiment analysis and named-entity recognition.

Considering the history of the NLP field, hashtags are quite a new concept. Recently, a number of studies have shown that they can be effectively used for various social media NLP tasks such as for text classification [111], query expansion [112], and emotion detection [113]. These studies reveal that hashtags have started to attract the attention of the NLP community. However, so far, no extensive study has been done on segmenting hashtags and analyzing their grammatical structure. Most prior studies use their function of being a label without breaking them into their constituent words [114]. When they need segmented hashtags, they either use the traditional word segmentation tools [115] or employ simple glossary and rule based approaches [111]. Despite their high accuracy at segmenting words, such tools have not been originally designed for hashtag segmentation. Our study [23] and [116] showed that hashtag

segmentation presents a more challenging task than traditional word segmentation, as both studies outperformed the state-of-the-art word segmentation tool Word Breaker [43] at segmenting hashtags.

In the first part of this thesis, we develop a feature-rich machine learning based approach for hashtag segmentation. Here are the list of contributions for this part:

(i) Unlike the traditional supervised machine learning setting, where training is performed using manually annotated training data, we utilize the large amount of unlabeled data available in the social media and use two approaches to automatically create our training data. In this respect, our approach can be considered as weakly-supervised. First, we use normalized tweets to create synthetic hashtag segmentations. Second, we design a special heuristic approach which automatically extracts hashtag segmentations from a large set of tweets.

(ii) We create and share two data sets each one consisting of 1000 manually segmented hashtags.

(iii) Most prior work do the segmentation in isolation from the context in which a hashtag occurs. To utilize the words that occur together with the hashtag, we introduce context-based features. Moreover, we use not only the single tweet that hosts the hashtag, but also multiple tweets that hold the same hashtag. We distinguish the difference as local and global context.

(iv) Apart from the feature-based approach, we present a language model (LM) based hashtag segmentation approach. We look into how LM alone performs at segmenting hashtags and then, blend the LM-based approach into the feature-based one by introducing new features based on the top best LM-based segmentations. We show that combining context-based features and LM-based features leads to improved performance.

(v) We run our best hashtag segmentor on 2.1 million distinct hashtags and obtain their automatically segmented forms. With that set, we first measure how much sentiment is trapped inside multi-word hashtags. Then, we parse this set by using a dependency parser and study their grammatical structure. To the best of our knowledge, this is the first study that reports on the constituent word analysis of

a large set of automatically segmented hashtags. We argue that such an analysis reveals the extend of hashtags' complexity as well as the need to use hashtag segmentors.

3.1. Related Work

Precursor to hashtag segmentation, the earliest related work is word segmentation, which is described in Section 2.1.1. Compared to traditional word segmentation studies, there are only a handful of studies on hashtag segmentation. [117] use an unsupervised method with a joint probability model learned from multiple corpora. However, they evaluate their segmentor only by observing improvement in the Twitter search recall measure. [118] tackle the problem as a compound word segmentation task and apply the well-known Viterbi algorithm [92] to choose the best possible word sequence. Their hashtag segmentor has not been evaluated independently, but has been used as a component in a search system for tweets. [119] develop a hashtag tokenizer for GATE [120] by applying a Viterbi-like algorithm to look for the best possible match by using multiple gazetteers in the GATE framework. [121] focus on normalizing the surface forms of hashtags in the form of case normalization, lemmatization and syntactic segmentation. They examine tagged and parsed versions of *CamelCased* hashtags in a domain specific data set. [116] introduce a hashtag segmentor which starts with a set of possible segmentations and ranks them by using five features, including context similarity. They achieve 87.3% accuracy using 5-fold cross validation on their manually annotated test set. While evaluating our hashtag segmentation model, we also use their test set, which we call Test-STAN. In our recent study, we developed a feature-rich approach using MaxEnt and CRFs as the learning algorithms [23]. Various vocabulary- and orthographic-based features were designed. The use of auto-segmented hashtags for training versus tweets was evaluated. While using tweets for training resulted in better performance, MaxEnt was shown to outperform CRFs. An accuracy of 88.2% was achieved on our manually annotated test set, Test-BOUN [28]. On Test-STAN, the best accuracy value obtained was 85.4%.

3.2. Our Approaches to Segment Hashtags

3.2.1. Feature-based Approach

We formulate the segmentation problem as a boundary detection task in a given sequence of characters. In this setup, each character becomes an individual training instance for the learning algorithm and is labeled as being the first character (boundary) of a word or not in the sequence. Imagine we have an imaginary cursor pointing to a character in the given sequence as in Figure 3.1. At each position, we determine the active features and the learning algorithm uses these features to model this binary classification task. For each training instance, its class and the list of active features are given to a MaxEnt model for learning [96]. Each feature is represented as a string constructed as feature_name=feature_value, where feature value is obtained based on the current cursor position. We consider various features with different characteristics. These features and their initialized values based on the cursor position shown in Figure 3.1 are listed in Tables 3.1, 3.2, 3.3, 3.4, and 3.5. The first column shows how to set the values for the corresponding features by using functions[1] . The second column contains the active feature instances, each one constructed as a string consisting of feature name and value pairs. We group the features into four categories:

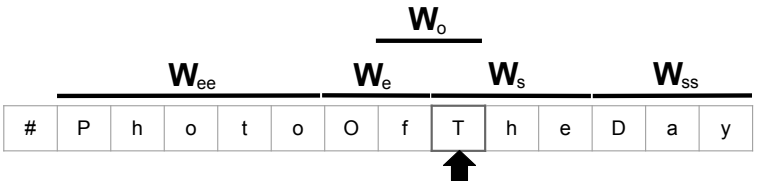

Figure 3.1. Showing the words detected around the cursor position for the hashtag #PhotoOfTheDay; W_s: longest word at cursor position; W_e: longest word before W_s; W_{ee}: longest word before W_e; W_{ss}: longest word after W_s; W_o: overpassing word "ft"

[1]The descriptions of the functions that are not already described in text are as follows: *ifExist*$(\cdot)$ checks if the given input exists; *isUpper*$(\cdot)$ and *isLower*$(\cdot)$ check if there is an upper and lower case letter in the given position, respectively; *isNumber*$(\cdot)$ checks if there is a digit in the given position; *ifRestIsNumber*$(\cdot)$ checks if all characters from the given position till the end of the input are digits; *countRepeatedCharactersAt*$(\cdot)$ counts the number of repeating character starting at the given position and *countRepeatedCharactersAtBackward*$(\cdot)$ does the same backward; *ifShortWord*$(\cdot)$ checks if a given word is a short word; *ifBigramExists*$(\cdot, \cdot)$ checks if the given two words are seen as a bigram in the training set.

Table 3.1. Listing all active vocabulary-based feature instances generated based on the cursor position shown in Figure 1, where W_s=the, W_{ss}=day, W_e=of, W_{ee}=photo, and W_o=ft. The plus sign (+) is used as a separator to make the reading easy.

TEMPLATE OF FEATURE VALUE	ACTIVE FEATURE INSTANCE
W_s	longest_word_starting_at_cursor=the
$len(W_o)$	length of overpassing word=2
$ifExists(W_o)$	if overpassing word exists=TRUE
$ifExists(W_e\&W_s)$	if words around cursor position exist=TRUE
$ifExists(W_s)\&isUpper(cursor[0])$ $\&isLower(cursor[-1])$	if word at cursor position exists and cursor position in upper case=TRUE
$ifExists(W_e\&W_s)\&isUpper(cursor[0])$ $\&isLower(cursor[-1])$	if words around cursor position exist and cursor position in upper case=TRUE
$len(W_s)$	length of longest word starting at cursor=3
$len(W_o)+flNegLogProb(W_o)$	length and floored neg log prob. of overpassing word=2+3
$len(W_s)+flNegLogProb(W_s)$	length and floored neg log prob. of longest word at cursor=3+1
$len(W_e)+len(W_s)+len(W_o)$	length of words around cursor and overpassing word=2+3+2
$shortWordAt(cursor[0])+len(W_e)+len(W_{ss})$	short word in middle and length of surr words=the+2+3
$len(W_e)+flNegLogProb(W_e)$	length and floored neg log prob. of word ended before cursor=2+1
$len(W_{ee})+flNegLogProb(W_{ee})+len(W_e)$ $+flNegLogProb(W_e)$	length and floored neg log prob. of two words before cursor=5+3+2+1
$len(W_e)+len(W_s)$	length of longest words around cursor=2+3
$len(W_e)+flNegLogProb(W_e)+len(W_s)$ $+flNegLogProb(W_s)$	length and floored neg log prob. of words around cursor=2+1+3+1
$len(W_s)+flNegLogProb(W_s)+len(W_{ss})$ $+flNegLogProb(W_{ss})$	length and floored neg log prob. of bigram starting at cursor=3+1+3+1
$flNegLogProb(W_e)+flNegLogProb(W_s)$	floored neg log probability of words around cursor=1+1
$wordClass(W_e)+wordClass(W_s)$	word class bigram around cursor=10110+110100
$wordClass(W_s)+wordClass(W_{ss})$	word class bigram at cursor=110100+11100111010

3.2.1.1. Vocabulary-based Features. Vocabulary-based features look for words around the cursor position in a character sequence based on a given vocabulary. We create our vocabulary from all used training data sets, which are described in the Training and Test Data Sets section. As shown in Figure 3.1, we look at five positions w.r.t. the cursor position for the longest matching words[2] . While the existence of an overpassing word (W_o) such as 'ft' in Figure 3.1 suppresses the boundary decision, words starting at the cursor position (W_s) and ending just before the cursor position (W_e) are positive indicators for a boundary for that position. The longer they are, the more likely there is a boundary. Apart from using the longest word W_s as a feature, we also define features that represent words in terms of their lengths ($len(\cdot)$) and floored negative unigram

[2]Assume that there is no longer word in our vocabulary that can match in this specific example.

log probabilities[3] (*flNegLogProb*$(\cdot)$). For example, the unigram log probability of the word "the" is -1.808. Hence, one of the features in Table 3.1 represents the word "the" (W_s) with the combination of its length 3 and its floored negative log probability 1 (i.e., *length and floored negative log probability of longest word at cursor* $= 3 + 1$). We also use the class codes of the words (*wordClass(·)*), which we obtain from CMU's Twitter Word Clusters [122]. To give an example, since the words "of" and "the" are represented with Word Cluster codes 10110 and 110100, respectively, one of the features in Table 3.1 (i.e., *word class bigram around cursor* $= 10110 + 110100$) uses this information to represent the word bigram, where the first word (W_e) occurs before the cursor position and the next one (W_s) starts at the cursor position. Such representations group words into equivalence classes and reduces the number of model parameters, hence reducing the complexity of the model. We observed that especially features where words are represented with their lengths and floored negative log probabilities together are more effective in general compared to using the words themselves. Moreover, since short words[4] like "of", "the" etc. may be seen very frequently inside any hashtag, one of our effective features looks for a short word *shortWordAt(·)* at cursor position which is also supported by the existence of surrounding words W_e and W_{ss}.

3.2.1.2. N-gram based Features. Listed in Table 3.2, N-gram based features are extensions of word-based features. Detecting word bigrams around the cursor position like (W_e, W_s) and (W_s, W_{ss}) can be quite indicative of a boundary. Moreover, we define features based on the bigram frequencies obtained from all our training data sets. These frequencies are clustered according to their ranges and represented by their cluster indices. *bigramFreqClass(·, ·)* assigns a cluster index of 1, for example, if frequency is higher than 500[5] . Similarly to the *shortWordAt(·)* feature, there is also a feature that looks for bigrams that are constructed from short words only, like "of the". *shortNgramAround(·)* returns such bigram around the cursor position.

[3]We use Microsoft's Web N-Gram Service to collect unigram log probabilities of words in our vocabulary.

[4]The list of short words and short bigrams (consisting of short words) that we compiled is available at our project web site.

[5]Cluster indices of 6, 5, 4, 3, 2, and 1 are assigned when frequency is lower than 5, 10, 50, 100, 200, and 500, respectively.

Table 3.2. Listing all active ngram-based feature instances generated based on the cursor position shown in Figure 1, where W_s=the, W_{ss}=day, W_e=of, W_{ee}=photo, and W_o=ft. The plus sign (+) is used as a separator to make the reading easy.

TEMPLATE OF FEATURE VALUE	ACTIVE FEATURE INSTANCE
$shortNgramAround(cursor[0])$+$len(W_{ee})$ +$len(W_{ss})$	short ngram over cursor and length of surrounding words=of the+5+3
$bigramFreqClass(W_e,W_s)$+$flNegLogProb(W_o)$ +$len(W_o)$	freq class of bigram around cursor and overpassing word length=1+3+2
$ifBigramExists(W_e,W_s)$	if words around cursor seen as bigram=TRUE
$bigramFreqClass(W_e,W_s)$	frequency class of bigram around cursor=1
$bigramFreqClass(W_s,W_{ss})$	frequency class of bigram starting at cursor=1
$bigramFreqClass(W_s,W_{ss})$ +$bigramFreqClass(W_e,W_s)$	frequency classes of bigram starting at cursor and around cursor=1+1
$len(W_e)$+$len(W_s)$+$len(W_o)$ +$bigramFreqClass(W_e,W_s)$	len of words around cursor and overpassing and freq class of bigram around cursor=2+3+2+1
W_e+W_s	bigram with short word around cursor=of+the

Table 3.3. Listing all active orthography-based feature instances generated based on the cursor position shown in Figure 1, where W_s=the, W_{ss}=day, W_e=of, W_{ee}=photo, and W_o=ft. The plus sign (+) is used as a separator to make the reading easy.

TEMPLATE OF FEATURE VALUE	ACTIVE FEATURE INSTANCE
$orth(cursor[-1])$+$orth(cursor[0])$	orthographic shape of previous and current character=c+C
$orth(cursor[-1])$+$orth(cursor[0])$ +$orth(cursor[1])$	orthographic shape of previous current and next characters=c+C+c
$orth(cursor[-2])$+$orth(cursor[-1])$ +$orth(cursor[0])$	orthographic shape of previous two and current characters=C+c+C
$orth(cursor[-3])$+$orth(cursor[-2])$ +$orth(cursor[-1])$+$orth(cursor[0])$	orthographic shape of previous three and current characters=c+C+c+C
$cursor[0]$+$cursor[1]$+$cursor[2]$	three character starting at cursor=t+h+e
$isNumber(cursor[0])$ & $ifRestIsNumber(cursor[0])$	if number starts at cursor position and continues till end=FALSE
$countRepeatedCharactersAt(cursor[0])$	num of times character at cursor repeats forward=0
$countRepeatedCharactersAtBackward(cursor[0])$	num of times character at cursor repeats backward=0

3.2.1.3. Orthographic shape-based Features. Orthography-based features complement the vocabulary-based ones. They are listed in Table 3.3. $orth(\cdot)$ converts characters to their orthographic shapes[6] . A capitalized letter or a number around the cursor position can be a good indicator of a boundary. Orthography-based features are especially effective when no word is detected at the cursor position.

[6]Upper-case letter is replaced with 'C', lower-case one with 'c', digit with 'd'; otherwise the character itself is used

3.2.1.4. Context-based Features. Context-based features make use of words that occur in the same tweet that hosts the hashtag. We consider three context-based features. Table 3.4 presents the context feature instances generated for the example tweets in Figure 3.2. The simplest feature looks at whether a word in the tweet is seen in the hashtag starting at the cursor position. To give an example, in Figure 3.2, the first tweet includes a hashtag #nationalwineday, while the word "wine" also occurs in the tweet. Since such detection doesn't require any vocabulary, this feature is complementary to vocabulary-based features.

The second feature looks for semantically similar words in the tweet to the word starting at the cursor position (W_s) in the hashtag. If W_s is semantically similar to a word in the tweet, then that signals the existence of W_s in the hashtag. We use the distributed representations of words to compute their semantic similarity. To obtain that, we train the *word2vec* tool [101] on the Stanford Twitter Sentiment Analysis Data set [123] with default parameters to produce a 100-dimensional vector for each word. The value of this feature is set as the cosine similarity value between the vectors of W_s and the word that is most similar to W_s in the tweet. In Figure 3.2, the word "prevent" in Tweet 2 can help to identify the boundary starting with the semantically similar word "blocked".

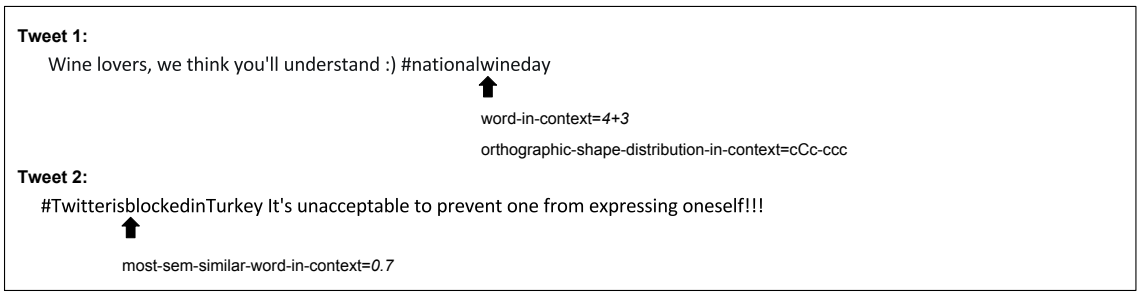

Figure 3.2. Sample tweets exemplifying the context-based features.

Considering the fact that the same hashtag may occur in a number of tweets, we can collect a set of tweets that contain the same hashtag and use all their words as an extended context. We call this global context, whereas the case where only one tweet that hosts the hashtag is used is called local context. This increases the chance of identifying words in hashtags that are mentioned in the global context or semantically related to them. Moreover, since different capitalized versions of the same hashtag may

Table 3.4. Listing all context-based feature instances generated based on the cursor positions shown in Figure 3.2, where W_s=wine for Tweet 1, W_s=blocked for Tweet 2. The plus sign (+) is used as a separator to make the reading easy.

TEMPLATE OF FEATURE VALUE	**ACTIVE FEATURE INSTANCE** (feature_name=feature_value)
$len(W_s)$+$flNegLogProb(W_s)$	word-in-context=4+3
$orthoShapeDist(W_s)$	ortho-shape-dist-in-context=cCc-cc
$cosine\text{-}similarity(W_s, W_c)$	highest-sem-similarity-in-context=0.7

occur in a given tweet set, as a third feature, we look for orthographic shape differences among all occurrences of a hashtag. We count the number of different capitalization cases around the cursor position in a window of 3 characters, order them based on their frequencies, and create a feature from these ordered orthographic shapes. For example, assume that we have 10 tweets that include #NationalWineDay and 2 tweets with #nationalwineday. As the cursor is at the starting character of word 'wine', there are 10 "cCc" and 2 "ccc" orthographic shapes observed around that cursor position. For this case, the feature has the value "cCc-ccc", which is a concatenation of the ordered orthographic shapes with a hyphen in between.

3.2.2. LM-based Approach

Besides the feature-based approach to detect word boundaries, we also consider a completely different approach, which takes into account all likely segmentations, scores each with a language model (LM), and chooses the highest scoring word sequence as the best segmentation for the input character sequence. An LM-based approach is generally successful at detecting word sequences and such a capability is particularly helpful when it comes to detecting short words (e.g. "in", "at" etc.) between other words. This is where the feature-based approach struggles. Nevertheless, the performance of the LM-based approach is hindered by unknown words as they disrupt the true word sequence, in which case, LM inevitably scores an incorrect word sequence as the best possible segmentation. Considering that tweets are full of typos and frequently introduced new words, the LM-based approach may not perform as expected in this domain. One

way to cope with this is to use as much data as possible to train the LM. Hence, we generate the training data set using randomly selected tweets from the SNAP data set [124]. The SNAP data set contains 20-30% of all public tweets posted during the seven months period between June and December, 2009. It includes 476 million tweets. While creating the training set, we filter out the hashtags, links, mentions and punctuations, which leaves out only word sequences. In order to investigate the effect of training set size on results, we created multiple training sets with increasing sizes, starting from 10 million tokens up to 1 billion tokens. From each training set, we generate the language model with the SRILM tool. We generate the LM from 4-grams and prune the vocabulary to contain the most frequent 1M words. We use Kneser-Ney smoothing as the discounting option and use the default values for the rest of the parameters. In our experiments, we use the learned LM models with the OpenFST tool[7] , which considers all likely segmentations for a given input character sequence and selects the highest scoring segmentation as the best based on the used model.

3.2.2.1. LM-based Features. Considering the noisy nature of tweets, the LM-based approach might not always find the exact segmentation. However, if we consider the N-best segmentations, instead of the best suggested one, it is more likely to have the correct (or very close to correct) segmentation among them. We argue that the N-best segmentations can be a supplementary data for the feature-based approach and we consider three features that make use of that data as boundary clues. This effectively combines the feature-based and LM-based approaches. First, we use the rank of the highest scored segmentation that has a word boundary at the cursor position. Secondly, as an extension to the first one, we concatenate the ranks of all segmentations that have a boundary at the cursor position. Thirdly, we use the number of segmentations that have a boundary at the cursor position as the feature value. To illustrate these with an example, Figure 3.3 shows the top 10 highest scoring segmentations of “greatmovie”, highest at the top. Alongside, the active feature instance for each LM-based feature is listed based on the shown cursor position in this example. The equal sign separates the name of the feature and its value.

[7]We use *make-ngram-pfsg* and *pfsg-to-fsm* tools to make the output of the SRILM tool compatible with OpenFST.

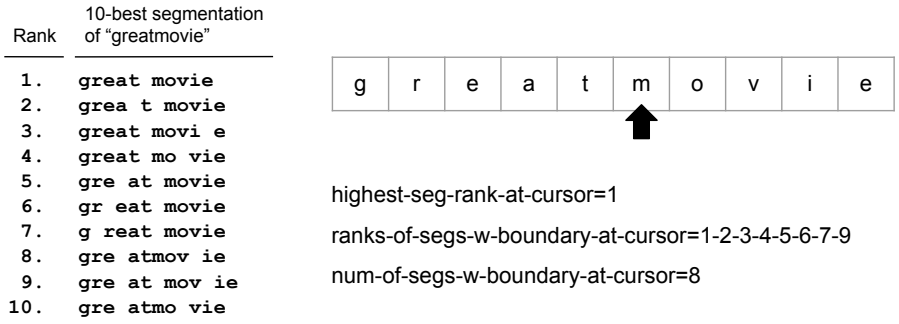

Figure 3.3. Showing the values of LM-based features when the cursor is at the 6th position while segmenting “greatmovie.”

Table 3.5. Listing all LM-based feature instances generated based on the cursor position shown in Figure 3.3, where i is the position of the cursor and *N-bests* holds the N-best LM-produced segmentations of the input.

TEMPLATE OF FEATURE VALUE	**ACTIVE FEATURE INSTANCE** (feature_name=feature_value)
highestSegmentRankAt(*N-bests*, i)	highest-seg-rank-at-cursor=1
segmentRanksAt(*N-Bests*, i)	ranks-of-segs-w-boundary-at-cursor=1-2-3-4-5-6-7-9
numOfSegmentsAt(*N-Bests*, i)	num-of-segs-w-boundary-at-cursor=8

In case of the first feature, the rank represents the index of the segmentation in the N-best list and the higher it is (i.e., the smaller the index), the more likely that there is a boundary at that position. For the other two features, the more segmentations have the boundary at the cursor, again, the more likely that there is an actual boundary there. We observe the state-of-the-art results when all three LM features are used together.

3.3. Automatically Generating Training Data

Instead of manually segmenting thousands of hashtags for training purposes, we considered two approaches. The first one involves acquiring automatically segmented hundreds of thousands of hashtags and using them for training. For this purpose, we used the SNAP Stanford Twitter data set [30]. We extracted 2.6M distinct hashtags from 476M tweets and applied simple heuristics to automatically segment those hash-

tags. We searched for consecutive word sequences in the SNAP tweets such that their concatenation corresponds to one of those hashtags. For 1.25M hashtags, we detected at least one word sequence. We selected the most frequent word sequence for each hashtag, if the hashtag's total occurrence is higher than 10 and if the word sequence corresponds to 75% of the total occurrences of all word sequences that correspond to that hashtag. For example, in case of *#twittermarketing* hashtag, we detected 29892 occurrences of *"twitter marketing"* and 116 occurrences of *"twittermarketing"* in the SNAP set. We ended up with 734K hashtags and their automatic segmentation, which we call it the Hashtag set.

In our second approach, we generated synthetic hashtags by concatenating the words in tweets. Since the word boundaries are known from the tweets, we can use these for training purposes. We create two sets of tweets from different sources. The first set is selected from the Stanford Sentiment Data set. The second tweet set BOUN was collected with the Twitter Search API by using the names of popular people, movies, tv shows, sports teams etc. as query.

3.4. Vocabulary Building

Vocabulary might be the most essential part of a word segmentation system. Because when we give string of characters to the system, it traverse over each character and look for possible word boundary at that position. Knowing existing words helps the system to better predict whether a word starts at the current position or ends. All our vocabulary-based features described in Section 3.2.1.1 depend on the quality of the vocabulary.

Building a vocabulary is especially difficult in the context of tweets. You need to make sure that all possible words that can occur in tweets are in your vocabulary. However it is a very difficult task because there can be many proper or special names occur in this environment. Moreover, the language used in tweets is so impure, if tweets are used to build the vocabulary, you may end up having many wrongly spelled words in that vocabulary. And the ones that hurt the segmentation system most are those

cases where two or more words are concatenated to each other. Such cases is highly likely because people can save space due to 140 character length restriction on tweets in the past. To given an example, “ilove” token can be seen many times in tweets and due to its high occurrence it might be considered as a valid word. However, it comprised of words “i” and “love.” It is hard to detect all such cases and clean your vocabulary.

Despite all disadvantages, having a vocabulary that is derived from the original source domain provides a great advantage. To filter out invalid words, after building our vocabulary, we retrieve unigram score for each word from Microsoft Web N-Gram Service and discard those words having a log score less than -8. Since the language model (LM) of this service is calculated from a huge web corpus, it is fair to assume that such threshold will remove most of the invalid words. To support that, we also run experiments and the best results were taken with -8 threshold.

3.5. Experimental Setup

3.5.1. Training and Test Data

In order to learn how to segment words, learning algorithms need training data which include already segmented words. For this purpose, we collected tweets with the Twitter Search API by using the names of popular people, movies, tv shows, sports teams etc. [23] as query. Moreover, in order to make our results more comparable, we use the tweets in the Stanford Sentiment data set used by [116]. After filtering the links and other non-literal tokens in tweets, boundaries of the words were used as gold standard for training. Two tweet data sets were generated: Tw-STAN and Tw-BOUN. Since tweets in the Tw-BOUN are collected with specific topic-based queries, this data set can be considered to be specific to certain domains. On the other hand, tweets in the Tw-STAN are collected with the emoticons only. Hence, the difference between the collection methods for the two data sets allows us to measure the effect of “almost” randomly collected tweets (Tw-STAN) compared to relatively more domain-centric tweets (Tw-BOUN) on performance. Our third training data set, which is a hashtag

data set (HASHTAGS), was obtained from the SNAP Twitter data set as described in Section 3.3

In our trainings, we use portions of these three data sets with increasing sizes. Table 3.6 lists the number of tokens in those data sets, which roughly gives the number of positive training instances for hashtag segmentation. Note also that the column at the right hand-side of the table indicates the number of hashtags selected from the HASHTAGS data set to make the token counts compatible with the tweet data sets.

Table 3.6. Number of tokens in Tw-BOUN, Tw-STAN, and HASHTAGS datasets with increasing sizes.

# of Tweets	# of Tokens			# of Hashtags
	Tw-BOUN	Tw-STAN	HASHTAGS	
5K	56K	60K	57K	24K
10K	115K	118K	118K	49K
20K	237K	249K	237K	98K
50K	594K	602K	595K	246K
100K	1189K	1210K	1180K	489K

For the development and testing purposes, we used the test set manually labeled by [116] as well as two data sets that we created manually [23]. We call the test set of [116] Test-STAN. It includes 1268 randomly selected hashtags from the Stanford Twitter Sentiment Analysis data set. To complement that, for development purpose, we randomly selected 1000 more hashtags from Stanford Twitter data set and segmented them manually. It is called Dev-STAN. Our second data set includes 1000 hashtags randomly selected from Tw-BOUN collection. We divided this data set into equal parts and called them Dev-BOUN and Test-BOUN and used them for development and testing purposes, respectively. Similar to the difference between Tw-STAN and Tw-BOUN, these two test sets have different characteristics. STAN test sets are more likely to contain complex hashtags with high randomness, whereas BOUN test sets contain more domain-centric hashtags with less randomness. It would be interesting to observe how the segmentor performs in both cases.

3.5.2. Evaluation Metrics

Evaluation of word segmentation is usually done by measuring the percentage of the test instances that are segmented exactly as expected. It is called accuracy measure. Despite this defacto standard, we argue that F_1-score can be a more appropriate evaluation metric depending on in which task the segmentor will be used. The F_1-score awards partially correct segmentations. Even if not all words are detected correctly, detecting most of them might still be useful for certain applications. For example, in case of sentiment analysis, if it can separate the words with sentiment correctly, it does not matter whether it detect the right boundaries for the other words. More importantly, since it is more granular, it shows the progress of the segmentor better during its development period. F_1-score is the harmonic mean of precision and recall. Precision is what percentage of the words outputted by the segmentor is correct, while recall measures what percentage of the actual words in the test set are identified. In the following sections, we report our results in both accuracy and F_1-score.

3.5.3. Baseline

For both test sets, we consider three approaches at different strengths as baselines. Our first baseline is based on HMM which was trained on character tri-grams. It considers both the current and previous two characters with respect to the cursor position for the boundary detection. We trained it with 100K randomly selected tweets from Tw-BOUN. However, it performed poorly. As a second approach, we use another off-the-shelf tool, called Hashtag Tokenizer [119] in the GATE framework [120]. This gazeteer-based approach achieves slightly better performance than the previous baseline results, yet still obtains low performance.

While the previous two approaches can be considered as weak baselines, the third approach, Microsoft's Word Breaker [43], provides a strong baseline, considering that it outperforms the other baselines significantly in all metrics. Hence, we consider it as our actual baseline. As shown by [112], hashtags can be a more challenging segmentation task for Word Breaker, since it has originally been designed to break URLs.

Table 3.7. Baseline results on Test-BOUN and Test-STAN sets.

Approach	Test-BOUN		Test-STAN	
	F_1-score	Accuracy	F_1-score	Accuracy
HMM	74.0	69.3	63.0	64.3
GATE Hashtag Tokenizer	76.5	70.2	73.3	71.5
Word Breaker	**84.4**	**86.2**	**84.6**	**83.6**

3.6. Experimental Results

3.6.1. Context-based Results

Context-based features require us to collect tweets for both training and test hashtags. However, we observe that some hashtags in the test sets occur rarely. In order to measure the ideal effect of context and its size correctly, we need to make sure that all hashtags in the test sets have the same number of tweets in their global context. To do that, from our training sets, we collected all tweets that contain hashtags from our test sets. The number of hashtags in Test-BOUN and Test-STAN that occurred in at least 100 tweets was 300 and 500, respectively. We call these new sets Test-BOUN-300 and Test-STAN-500 and use them to evaluate the effects of context features.

When we train the best (BEST) feature combination which was reported by our study [23] on the 100K HASHTAGS data set without adding any context-based features, the accuracies on Test-BOUN-300 and Test-STAN-500 are calculated as 90.0% and 88.9%, respectively. As we add each context feature into this best feature combination, we measure how much increase in accuracy each context feature provides as shown in Table 3.8 and Table 3.9. We start the size of the context from one tweet (C=1), which we consider as the local context case. Then, we gradually increase the number of tweets up to 100 tweets (C=100).

In case of Test-BOUN-300, we achieve the highest scores when we use all context-based features. Even though using 100 tweets as global context results in the highest

Table 3.8. Accuracy on the Test-BOUN-300 test set, while the baseline (BEST) accuracy where no context is used is 90.0%.

Features	C=1	C=5	C=10	C=20	C=50	C=100
BEST + Word in Context (WIC)	90.0	90.0	90.3	90.0	90.3	90.7
BEST + Sem. Similar Word in Context (SSWIC)	89.7	90.3	89.7	90.3	90.3	90.3
BEST + Orthographic Shape Distribution in Context (OSDIC)	-	91.3	90.3	90.7	90.7	90.0
BEST + WIC + SSWIC + OSDIC	89.7	91.3	91.3	91.7^a	91.3	**92.7**^{a}

accuracy of 92.7%, using as few as 20 tweets still outperforms the baseline accuracy, which is 90.0%, statistically significantly[8] . On the other hand, with Test-STAN-500, even though all context features in general result in improvement, the orthographic shape distribution in context (OSDIC) feature results in the best performance. These results suggest that even if we can collect as few as 20 tweets for global context purposes, it can still increase the segmentation accuracy.

Table 3.9. Accuracy on the Test-STAN-500 test set, while the baseline (BEST) accuracy where no context is used is 89.9%.

Features	C=1	C=5	C=10	C=20	C=50	C=100
BEST + Word in Context (WIC)	90.1	89.5	90.4	90.1	89.7	89.9
BEST + Sem. Similar Word in Context (SSWIC)	90.6	90.4	89.7	90.1	90.1	90.1
BEST + Orthographic Shape Distribution in Context (OSDIC)	-	90.8	**91.6**^{a}	**91.6**^{a}	91.4	91.0
BEST + WIC + SSWIC + OSDIC	89.9	91.4^b	90.8	90.6	90.8	90.6

3.6.2. LM-based Results

The LM-based results for different training set sizes are shown in Table 3.10. LM achieves an accuracy of 90% on Test-BOUN when trained with 1 billion tokens. This score is higher than the best previously reported feature-based result of 88.2% [23]. On the other hand, the best LM-based accuracy obtained on Test-STAN (80.4%) is lower than the best previously reported feature-based accuracy (85.4%), it is even lower than the accuracy of the Word Breaker baseline (83.6%). One way to explain

[8]In Tables 3.8 and 3.9, "a" and "b" indicate statistically significant result compared to the baseline for $p<0.05$ and $p<0.1$ (based on a paired two-tail t-Test), respectively.

the difference between the two test sets is to look at the perplexity scores[9] of those test sets. Perplexity measures how successful LM is at predicting the next word in a sequence. The lower the score is, the better the LM is at predicting the next word. For Test-BOUN and Test-STAN, the perplexity scores are 750 and 6920, respectively. The huge difference in these scores explains why LM fails on Test-STAN.

Table 3.10. Results of the best (top 1) LM-based word segmentation on Test-BOUN and Test-STAN sets.

LM Training Size	Test-BOUN		Test-STAN	
in # of tokens	F_1-score	Accuracy	F_1-score	Accuracy
10 Millions (10M)	89.3	84.6	81.0	78.0
100 Millions (100M)	92.2	88.4	82.4	79.7
1 Billion (1B)	**93.2**	**90.0**	**82.9**	**80.4**

Even if the most likely segmentation returned by LM is not the correct one, it is likely that the correct segmentation is among the top N segmentations produced by LM. In Tables 3.11 and 3.12, we calculate the accuracy of LM at top N, by considering the result as correct, if the gold standard segmentation is among the top scored N segmentations. F_1-score is computed by considering the segmentation with the lowest edit distance to the gold standard. The accuracy increases up to 94.8% and 93.2% on Test-BOUN and Test-STAN, respectively. On both data sets, the best segmentation is most of the time in the top 2. These results indicate that the top N segmentations contain valuable clues for segmentation.

3.6.3. Results with LM-based and Context-based Features Combines

As described in the LM-based Features section, we designed three LM-based features using the 10-best LM segmentations. Due to run-time constraints, we used LM trained on 100M tokens, rather than 1 billion. Table 3.13 shows how much increase these LM-based features add onto the best feature combination obtained on the Test-BOUN set in [23]. For any type of training set, we can observe up to 5.6 points increase

[9]We use 100M-token LM for perplexity calculation. We report *ppl1* value given by SRILM's *ngram* tool as the perplexity score.

Table 3.11. Best possible results in Top N on Test-BOUN.

TopN	**LM Training Data Size in # of tokens**					
	10M	100M	1B	10M	100M	1B
	F_1-Score at Top N			Accuracy at Top N		
N=1	89.3	92.2	93.2	84.6	88.4	90.0
N=2	93.1	95.8	96.2	89.8	93.6	94.4
N=5	93.4	96.2	96.6	90.0	94.0	94.8
N=10	93.4	96.2	96.6	90.0	94.0	94.8

Table 3.12. Best possible results in Top N on Test-STAN.

TopN	**LM Training Data Size in # of tokens**					
	10M	100M	1B	10M	100M	1B
	F_1-Score at Top N			Accuracy at Top N		
N=1	81.0	82.4	82.9	78.0	79.7	80.4
N=2	87.2	91.4	92.9	84.8	89.8	91.6
N=5	87.7	92.7	94.4	85.3	91.2	93.2
N=10	87.8	92.9	94.4	85.4	91.5	93.2

in accuracy and 3.9 points increase in F_1 score on Test-BOUN test set. Especially the increase in case of the HASHTAGS training set is consistently high. Note that there is no statistically significant difference between using all context features or using only the OSDIC context-based feature.

Likewise, Table 3.14 depicts a similar situation for the Test-STAN set. Using the LM-based features with 10-best segmentations improves the accuracy by up to 3 points. The increase is relatively lower than what we observe in Test-BOUN. This is again due to the fact that LM is better at detecting word sequences in TEST-BOUN than in Test-STAN. Similarly, using LM-based features on the HASHTAGS training set does not bring that much improvement compared to the 5.6-points increase on the Test-BOUN set. Again, in case of the HASHTAGS training set, as we add context-based features,

Table 3.13. Best results on Test-BOUN set. Baseline (MS Word Breaker) F_1-score = 84.4%, Accuracy = 86.2%

Training Set	**LM-based features used?**	**Training Data Size**									
		5K	10K	20K	50K	100K	5K	10K	20K	50K	100K
		F_1-score					Accuracy				
Tw-BOUN	No	90.5	91.3	91.4	91.5	91.5	85.6	86.8	87.2	87.4	87.4
	Yes	93.5	94.6	93.9	93.6	94.1	90.2	91.8	90.4	90.0	90.8
Tw-STAN	No	92.0	92.1	92.1	92.4	91.8	88.0	88.2	87.8	88.2	87.6
	Yes	93.8	93.3	93.7	93.6	93.1	90.4	89.6	90.2	90.2	89.2
HASHTAGS	No	89.9	89.4	89.7	90.8	91.0	85.8	85.0	85.0	86.2	86.6
	Yes	93.4	92.3	93.6	94.4	94.6	90.2	88.6	90.6	91.4	91.8
	Yes+OSDIC	93.2	88.3	93.1	**94.9**	93.9	89.8	83.0	90.0	**92.2**	90.6
	Yes+All-C	93.5	93.6	94.3	94.5	93.9	90.4	90.4	91.6	91.6	90.8

we observe up to 3 points improvements in accuracy. The best accuracy in Table 3.14 is 88.5%, which is higher than our first published result of 85.4% [23]. To make this score comparable with the accuracy of 87.3% obtained by [116], we recalculated it on their original data set which includes over 100 duplicate hashtags compared to our Test-STAN. In that case, our best score increases to 88.8%[10] .

Additionally, in both Table 3.13 and Table 3.14, the segmentors trained with auto-segmented hashtags (HASHTAGS) tend to produce better results as the size of the training set increases. Such behavior is less consistent with the other types of training sets.

[10]The difference between 88.8 and 87.3 is statistically significant with p<0.05 based on one sample t-Test.

Table 3.14. Best results on Test-STAN set. Baseline (MS Word Breaker) F_1-score = 84.6%, Accuracy = 83.6%

Training Set	**LM-based features used?**	**Training Data Size**									
		5K	10K	20K	50K	100K	5K	10K	20K	50K	100K
		F_1-score					Accuracy				
Tw-BOUN	No	84.9	86.9	87.0	87.0	87.0	83.0	85.1	85.4	85.3	85.4
	Yes	88.2	89.3	90.0	89.9	**90.2**	86.1	87.6	88.4	88.3	**88.5**
Tw-STAN	No	84.4	86.4	85.8	85.9	86.7	83.9	84.2	83.5	84.8	84.8
	Yes	86.8	87.8	88.1	88.3	87.9	84.8	85.8	86.3	86.5	85.8
HASHTAGS	No	84.8	85.3	85.4	85.8	86.2	82.9	83.5	83.7	84.1	84.5
	Yes	85.4	84.6	86.9	87.6	86.8	83.5	82.4	85.2	85.8	84.9
	Yes+OSDIC	86.7	87.3	88.1	87.8	89.5	84.8	85.4	86.5	86.1	87.9
	Yes+All-C	86.6	85.8	87.1	87.5	89.2	84.7	83.8	85.4	85.7	87.6

3.6.4. Error Analysis

In order to better understand in which cases our hashtag segmentor fails, we examine the output of the highest scoring configuration[11] from Table 3.14 on the Dev-STAN set. We observe that almost all of mis-segmented hashtags contain only a single misclassified boundary. 42% of the erroneous cases are due to special words with low frequency, such as foreign words or proper names. 23% involve incorrectly segmented ordinary words like *outdoorsport*. The next most seen error type is caused by numeric expressions in single word names such as *o2007comp* with 13% of coverage. Similarly, 8% of errors are caused by capitalized characters inside single word names, such as *AbleGamers*. Such cases are very difficult to handle unless that word is seen in the training data. Otherwise, the segmentor tends to break the words from the capitalized characters. More surprisingly, 9.5% of errors are caused by lack of apostrophe in hashtags. For example, if a hashtag contains the string "thats" corresponding to "that's", the manually segmented form in the gold standard is "that s". In such cases,

[11]The one that achieves 88.5% accuracy when it is trained on 100K tweets from Tw-BOUN.

the segmentor fails to separate the contracted form of a word from the word that it is attached to.

3.7. Understanding The Content of Hashtags

In this section, we investigate the importance of having a hashtag segmentor. We first count the number of words in millions of auto-segmented hashtags and then measure how much of the sentiment is trapped inside multi-word hashtags. Then, we analyze the grammatical structure of auto-segmented hashtags to observe the complexity of hashtags. In these experiments, we use the hashtags extracted from the SNAP tweet data set and apply our best model to get their segmented versions. We calculate these statistics on both 2.1 million distinct hashtags[12] and all 60 million hashtag occurrences in the SNAP data set.

3.7.1. Length of Hashtags

The simplest thing we can do with this data set is to calculate the length histogram of hashtags as shown in Figure 3.4. In this figure, we can see that most of the hashtags have a length of 10, while the average length is 12.5 characters. Even though we cut the tail at length 30, there are hashtags as long as character limit allows, which was 139, excluding the # sign.

3.7.2. Orthography of Hashtags

Orthography deals with the shape and type of the characters in a string of characters. Whether character is capitalized or if it is a digit or letter is important information for boundary detection. For example, seeing capitalized letter after lower cased letter might be a very good indicator of a word boundary. Same is also true for sequence of numbers preceded by a letter. Having said that, we measure general statistics about orthographic shape of hashtags seen in this 2.6M data set. In 40.8% of hashtags we observe at least one capitalized character. 76.7% of these cases have the capitalization

[12]The count is taken in case-insensitive mode.

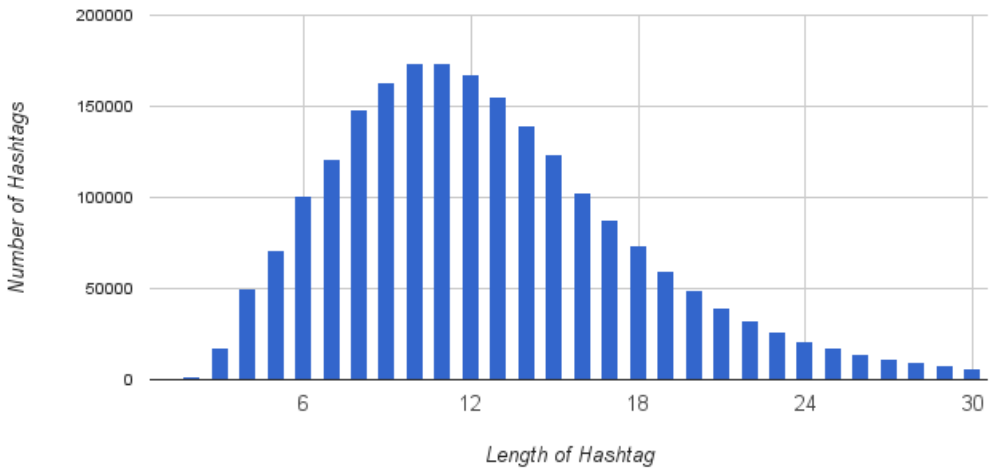

Figure 3.4. Hashtag Character Length Histogram.

inside the word rather than the first character being capitalized. This supports the common thinking that using capitalization as a feature should be very beneficial. It is interesting to note that out of 2.6M hashtags, 6% of them are completely capitalized. In case of digits, there is at least one numeric value in 7.9% of the hashtags. In those cases, 44.7% of them have at least one capitalized value along with that digit.

3.7.3. Word Count in Auto-segmented Hashtags

When we look at the word count calculated on the set of distinct hashtags in Table 3.15, 87.7% of the hashtags consist of multiple words, which means that hashtags are not simple one-word labels. Even when we calculate the percentages in all 60 million occurrences, this value only drops to 48.6%. In other words, half of the time we need a hashtag segmentor to break down a hashtag into its constituent words. Most of distinct multi-word hashtags contain two or three words and people tend to use hashtags that are no longer than three words in 93.8% of all cases.

Table 3.15. Number of words in auto-segmented hashtags.

# of Words	**% in Distinct 2.1M Hashtags**	**% in All 60M Occurrences**
1	12.3	51.4
2	38.5	30.8
3	24.7	11.6
4	12.5	4.3
5	6.1	1.3
> 5	5.9	0.6

3.7.4. Trapped Sentiment inside Multi-word Hashtags

Considering that half of the hashtag occurrences consist of multiple words, it is important to measure how often a word with sentiment occurs in multi-word hashtags. We use the AFINN sentiment analysis tool [125] [126], which assigns a sentiment score to a given text.

Table 3.16. Percentage of observed sentiment in auto-segmented hashtags.

Sentiment	**% in Distinct 2.1M Hashtags**	**% in All 60M Occurrences**
Neutral	75.7	86.8
Positive	10.7	7.1
Negative	13.6	6.1

Table 3.16 shows that, out of 2.1M distinct hashtags, 10.7% convey positive sentiment and 13.6% have negative sentiment. When we do the same calculation on all hashtag occurrences, the percentages drop to 7.1% and 6.1%, respectively. One thing to point in Table 3.16 is that people tend to use positive hashtags more often than negative hashtags, yet there are more distinct negative hashtags than positive ones. In other words, people are more creative at creating negative hashtags.

Our further analysis of hashtags containing positive or negative sentiment reveals that only 0.5% of distinct hashtags with positive sentiment, and 0.8% of distinct hashtags with negative sentiment are single-word, and the remaining ones are multi-word. When we consider all hashtag occurrences, we observe that only 20.7% of positive hashtags and 19.5% of negative hashtags are single-word. This means that, only around 20% of sentiment (either positive or negative) are seen in single-word hashtags. To put it another way, around 80% of sentiment is trapped inside multi-word hashtags.

3.7.5. Parsing Auto-segmented Hashtags

As the language in tweets are noisy and irregular, we chose to use TweeboParser [127] to parse segmented hashtags. TweeboParser is a dependency parser which is originally trained to parse tweets. It outputs which word in the given input grammatically depends on which other word, along with the part-of-speech (POS) tag assigned to each word. A word that does not depend on any other word is called root. Figure 3.5 shows an example dependency parse tree for the "Definition of Fail" hashtag. It is a noun phrase which includes the noun "Definition" as the root (or head) of the phrase and has an attached prepositional phrase which is headed by the preposition "of" and its dependent noun "Fail."

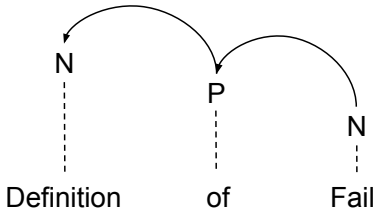

Figure 3.5. Dependency parse tree for "Definition of Fail" where the arrows point which word depends on which other.

While the dependency parser outputs the grammatical structure of the entire sentence or phrase, to keep our analysis simple, we only consider the root of the whole parse and its dependent tags one level below. When we just look at the root tag, it tells us which type of phrase or clause the whole structure is. When we look at its dependent tags, we basically observe how it is made of. While most of the hashtags have single root like the one shown in Figure 3.5, there can also be multiple roots, where each root

corresponds to an independent clause. However, for the sake of simplicity we perform our analysis on hashtags with single root in their parses.

3.7.6. Analysis of Root Tags

Table 3.17 shows the percentages of the most frequent POS tags at the root level from the parses of segmented hashtags. One might argue that most hashtags are supposed to be noun phrases, since they are mostly used for labeling. However, it indicates that almost half of the 2.1M distinct hashtags are noun headed expressions, namely noun phrases. Following that, with 26.0%, verb headed expressions come in the form of various clauses. As we consider all occurrences, we observe that noun-based expressions are used more often (77.1%) and usage of verb-based expressions drops down to 11.9%.

Table 3.17. The most frequent root tags.

Root Tag	**% in Dist. 2.1M Hashtags**	**% in All 60M Occurrences**
Noun (N)	48.1	77.1
Verb (V)	26.0	11.9
INTJ (!)	5.8	3.4
Adjective (A)	2.5	1.3
Preposition (P)	2.0	1.0
Coord. Conjunction (CC)	1.0	0.4
Adverb (R)	0.4	0.3
Multi-root cases	12.5	3.5

The third most common root tag is interjection (INTJ). However, we observe that many words tagged as interjection are actually unknown words not recognized by TweeboParser. Most of such cases include out of vocabulary words originating from foreign languages, wrong segmentations, and proper names. The other single-root POS tag cases, namely adjective (A), preposition (P), coordinating conjunction (CC), and adverb (R) cover a very small portion. The rest of the cases covering 12.5% of distinct

hashtags are complex expressions made of multi-root tags.

3.7.7. Analysis of Tag Patterns Around the Root

In order to inspect the structure of a hashtag, we investigate the tags that are connected to the root tag in the TweeboParser's output. We call these "tag patterns around the root tag." In the tables below, we show these patterns as a sequence of tags in the order of their occurrence. We also include the root tag, which is surrounded by brackets to make it distinguishable. In Figure 3.5 above, while the root tag is N, it has only one dependent tag, which is P. Hence, the tag pattern around the root tag is denoted as **[N] P**. Below, we analyze the most frequent tag patterns around each major root tag separately.

Table 3.18. The most frequent tag patterns headed by a noun.

Tag Pattern	**% in Dist.**	**% in All**	**Example Segmented Hashtag**
N [N]	41.0	23.3	Lindas Piernas; Beanie Siegel
[N]	11.3	59.1	MUNICES; Agression; Punicorn
A [N]	10.8	5.4	EXCELLENT GINA; New Iberia
N N [N]	7.1	1.3	Alex Volta Logo; buen servicio VE
[N] P	5.6	1.9	Definition Of Fail; apps to come

Table 3.18 lists the most frequent tag patterns headed by a noun root tag. As shown in the first row, 41% of distinct hashtags in noun form are constructed by combining two nouns together. It is almost four times more than the ones with single noun, consisting of 11.3% of cases. 10.8% of noun-headed hashtags have a single adjective modifier. When we consider the percentages in all occurrences, hashtags with single noun become dominant.

Table 3.19 lists the most frequent patterns headed by a verb tag. This time the most frequent pattern is sentence in imperative form, which covers 14.4% of distinct verb-based hashtags. Note that the fourth case is also in imperative form, which increases the total to 20.2%. The other three cases can be considered as regular sentence

Table 3.19. The most frequent patterns headed by a verb.

Tag Pattern	**% in Dist.**	**% in All**	**Example Segmented Hashtag**
[V] N	14.4	23.7	Hate Camilla Belle; Unfollow diddy
N [V]	11.3	8.9	Louis Presume; vegas sucks
N [V] N	6.4	4.0	dont judge medotcom; beat stabber
[V] V	5.8	4.2	dont judge medotcom
O [V] N	4.7	3.9	this is genius; i chrisf baby

formations, whose percentage sums up to 22.4%. When looking at all occurrences, we observe that people tend to use imperative form more often than regular form.

Table 3.20. The most frequent patterns headed by an adjective.

Tag Pattern	**% in Dist.**	**% in All**	**Example Segmented Hashtag**
N [A]	29.1	9.2	Vincents gay; cambio social
[A] P	15.2	3.1	sick of the heat; scared of iPhone
R [A]	13.1	3.9	finally safe; very confused
[A]	11.2	74.6	Willing; INSPIRING; Tempting
A [A]	6.2	1.3	deep undercover; the daily green

Considering trapped sentiment inside hashtags, another important root tag to investigate is adjective. According to Table 3.20, surprisingly, 29.1% of adjective-headed hashtags include the adjective as a post modifier following a noun. On the other hand, 74.6% of all adjective-headed hashtags consist of single word adjective. However, when we consider all hashtags, not only the ones headed by adjectives, adjectives are mostly used inside expressions especially as a noun modifiers (**A [N]** in Table 3.18). This also supports our finding that sentiment is trapped inside multi-word hashtags considering the fact that adjectives carry most of the sentiment compared to other word types.

3.8. Discussion and Future Work

In our experiments, two annotators manually segmented the hashtags in the Test-BOUN and Dev-STAN sets. On Test-BOUN, 330 of the hashtags were segmented by both annotators and there were 21 hashtags where the two annotators disagreed with each other (6.3%). In case of Dev-STAN, 600 hashtags were annotated by both annotators and the two annotators disagreed on 20 cases (3.3%). Hashtag segmentation is a difficult task for machines due to lack of context and the constantly evolving environment of Twitter. Almost half of the failures of the developed hashtag segmentor are related to uncommon and foreign words and only quarter of the failures look preventable as they involve regular words. The performance of Microsoft Word Breaker, which is the state-of-the-art word segmentor, on the hashtags indicates the difficulty of this task. Our context-based approaches might be helpful. Especially, when we make use of multiple tweets that contain the same hashtag as global context, that can remedy the lack of context. It is not ideal to rely on a single tweet as the source of context because people tend to do typos, deliberately shorten words, and omit putting space between words while writing tweets. However, if we collect statistics from multiple tweets, we enable the sampled data to better reflect the characteristics of the general population.

The simplicity of our proposed heuristic to obtain automatically segmented hashtags for training a hashtag segmentor resembles the bootstrapping approach of [128] while doing word sense disambiguation. He makes two simple assumptions regarding the relation between the sense of a word and its surrounding words. By using collocation-based statistics, he learns which words are indicative of a particular sense. He also trains a classifier with a small set of labeled data to predict the sense of polysemous words. When he uses that classifier on unlabeled data, he picks those predictions with probability above a certain threshold as new labeled instances, which iteratively increases the size of the labeled data set. Similar to Yarowsky's approach, our heuristic also assumes that an already segmented form of a hashtag occurs in the same tweet as the hashtag and the correctly segmented form can be chosen based on its frequency and its relative frequency with respect to alternative forms.

There are also various areas that have a potential for improvement. First off, we conducted our experiments on the SNAP data set, which contains tweets posted around 2009. It would be interesting to apply the same grammatical study on hashtags that are collected from more recent tweets. Moreover, showing the evolution of those statistics over a large time span would be a very interesting research area. It can be suspected that today people may tend to use longer and more complex hashtags than earlier times. That can amplify the idea that we need a hashtag segmentor to understand the hashtags and their effects on the hosting tweets.

Another improvement area for the hashtag segmentor is the replacement of the MaxEnt model with deep learning techniques. In our experiments (not reported), we briefly tried deep learning, but did not achieve good results. We believe application of deep learning techniques must be explored more extensively. We formulate hashtag segmentation as a binary classification task and neural networks (NNs) are very good at classification tasks. Especially, character-based NN models achieve great results on various tasks and hashtag segmentation can be good fit for such models. An NN model may also take the hosting tweet as an extra input along with the hashtag itself and use it as context information as we did in our experiments. At this point, it needs to be said that one of the drawbacks of our solution is that it requires input from external SRILM toolkit in order to feed Language Model (LM)-based features. This means that we need to run another program offline, which is not ideal. Instead, today the best LMs are RNN-based models. So, we can train an RNN-based LM at the side and then train the hashtag segmentor NN along with that model together. This gives us a stand-alone single neural network at the end, which requires no external tool. Moreover, NN-based approach especially with RNN-based LM extension may also cancel out the need for a vocabulary which makes it more robust and flexible. All these would simplify the solution.

In the experiments, we formulated hashtag segmentation as a binary classification task. However, it can also be formulated as a sequence labeling task. In fact, we applied Hidden Markov Model (HMM) and it performed very poorly as shown in Table 3.7. Another alternative method is Maximum Entropy Markov Models (MEMMs), but they

have label bias problem due to local variance normalization. Instead, Conditional Random Fields (CRFs) is a better alternative, as it adapts global variance normalization which solves the label bias. Sequence labeling requires us to have labels. Our task becomes assigning those labels to characters. We can use different labeling schemes. For example, we may have fine-grained labels like FIRST_CHAR, SECOND_CHAR, NTH_CHAR, and LAST_CHAR. Note that this is basically increasing the number of labels from two (i.e. BOUNDARY and INTERNAL) to four. In sequence labeling tasks like named entity recognition extending the label set (i.e. from BIO to BILOU tagging format) is shown to improve the results [15]. Hence, it is possible that a model with more fine-grained labeling can associate specific features with specific labels better and that may improve its predictions. However, such a labeling scheme is not suitable for MaxEnt, since it is not originally designed for sequence classification. Instead, we can use such a fine-grained labeling approach with deep learning algorithms and we can even extend the neural network model with a Conditional Random Fields (CFRs) layer, which tends to improve the results for sequence labeling tasks [58].

Apart from these major improvement points, there are a number of other things to try. For example, we observe that using auto-segmented hashtags as training data does not provide the best results on both test sets. One thing to try is to apply some kind of filtering mechanism (automatic or another heuristic) on those auto-segmented hashtags so that we can exclude the ones with less confidence.

3.9. Conclusion

In this part of this thesis, we explore extending a feature-based machine learning approach with context-based features and LM-based approaches for hashtag segmentation. We observe that context-based features improve the results without needing thousands of tweets in the context. As few as 20 tweets are sufficient. While using LM alone does not improve the results, utilizing N best segmentations given by LM as features helps us obtain the state-of-the-art results on both test sets. Moreover, as we add context-based features on top of that, we see up to 3 point increase when training is done on the HASHTAGS set, which makes HASHTAGS the best training set on

Test-BOUN. Error analysis shows that most of the incorrectly segmented hashtags are due to low frequency words like foreign words, expressions with numbers, and special capitalized cases. We used the word boundaries in tweets to create the Tw-BOUN and Tw-STAN training data sets. One possible future work would be using segmented tweets [129] to create training data, since tweet segments can be seen as phrases with similar characteristics to hashtags and may be more successful at representing word boundaries in hashtags compared to whole tweets.

In the second part of our research, we segment millions of hashtags extracted from the SNAP Tweet data set and then analyze their structure. We observe that almost 90% of 2.1M distinct hashtags include multiple words. Moreover, we observe that in 80% of sentiment bearing hashtags, sentiment is trapped inside multi-word hashtags. As we further analyze the parses of auto-segmented hashtags, we discover that about quarter of distinct hashtags are written as verb headed expressions, especially in imperative form. However, as we consider all occurrences, people tend to use noun-based hashtags more often. Adjectives are mostly used inside expressions rather than as single word hashtags. All these show that hashtags are not simple one-word labels, hence hashtag segmentation is necessary for a better understanding and utilization of hashtags, especially in the sentiment analysis task.

4. Named Entity Disambiguation

We humans name things around us in order to refer to them. Many times, different things can have the same name. For example, when you visit the disambiguation page [130] for the word "Washington" in Wikipedia, you see tens of cities and counties in the United States that are named "Washington." Moreover, the same word is used as a hyperlink title to refer to several different articles throughout Wikipedia. Things that can be denoted with a proper name are called named entities. The Knowledge Base (KB) of a machine keeps the list of all known named entities to that machine. When they are mentioned in a document, the task of identifying the correct named entity that a mention refers to among all the possible entities in the KB is called Named Entity Disambiguation (NED). Figure 4.1 gives three example sentences with the mention "Washington", each referring to a different named entity. For a human, it is not hard to figure out which entity is being referred to by considering the clues in the surrounding context of the mention. However, from a machine point of view, each mention may refer to any one of the hundreds of named entities in its KB.

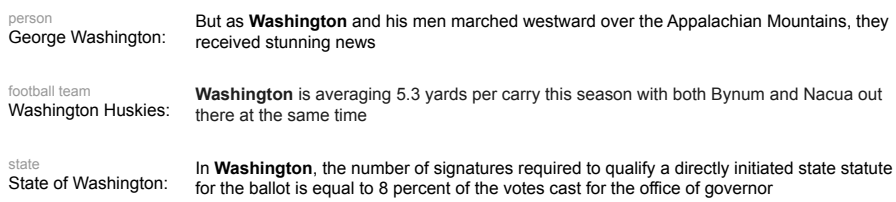

Figure 4.1. Mentions of different named entities with the same surface form "Washington."

NED, in general, is done in two steps. In the first step, the candidate named entities in the KB are identified based on their lexical similarities to the given entity mention. In the second step, which is the actual disambiguation step, each candidate is scored based on some extracted features. The one with the highest score is returned as the predicted named entity corresponding to the input mention. In this field, the reference KB is most commonly based on Wikipedia, as is in our case.

In the literature, various approaches have been proposed to solve the NED task. Early studies used entity-specific local features like the similarity of the candidate to the document topic in order to score them individually [17,131]. [71] proposed the idea that entities in a document should be correlated with each other and consistent with the topic of that document. How well a named entity is connected to the surrounding named entities is measured by coherence. They tried to optimize the coherence by including global context-based features that take into account the surrounding candidates into the disambiguation decision. Later studies looked at collective disambiguation, where all candidates are considered together in the decision process, rather than individually [132]. Most of the collective models employed computationally complex graph-based approaches in order to find the sub-graph of candidates with the highest coherence. As the deep learning approaches advanced, Long Short-term Memory (LSTM) [78] models have been used to capture the long-range relations between words in the context, and attention-based neural networks have been used [68] to pay more attention to the most relevant segments in the surrounding context of mentions. Entity embeddings, which are continuous vector-based representations of named entities, have been optimized to detect the relations between the candidate named entities and their context [69]. A number of recent studies have investigated utilising the category hierarchy of Wikipedia for Named Entity Disambiguation. [133] proposed to integrate the symbolic structure of the category hierarchy of the named entities in Wikipedia in order to constrain the output of a neural network model. [134] used the same principle of making use of type hierarchy, but proposed a hierarchically-aware training loss. [83] followed a similar approach, but rather than predicting the entities directly, they only modeled the fine-grained entity properties, which represent detailed entity type information. Then, they used that as a filtering mechanism at the disambiguation step. In addition to all these new techniques, more and more studies have been using Wikipedia as a vast resource for representation learning from its text and the hyperlinked mentions in the text.

This research focuses on identifying the type of a mentioned named entity first and then using this information to improve the prediction of the identity of that named entity at the disambiguation step. The first step is called mention-level entity typing, or

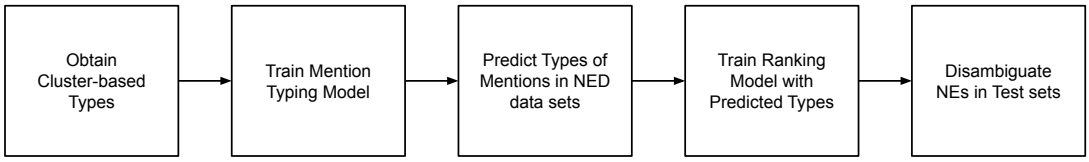

Figure 4.2. The Workflow of the Proposed System (NE: Named Entity, NED: Named Entity Disambiguation).

mention typing in short. It is the task of predicting the type (e.g. politician, city etc.) of a single mention given its surface form and its surrounding context. Prior studies on this task use manually curated type taxonomies, like Wikipedia categories, and assign each named entity mention to one or more such types. However, considering there are millions of named entities, such manual curation is inherently incomplete. Hence, we propose to obtain types automatically in an unsupervised fashion and use these types to improve NED. The workflow of the proposed approach is summarized in Figure 4.2. In the first step, we cluster named entities based on their contextual similarities in Wikipedia and use the cluster ids as types, hence cluster-based types. These types no longer correspond to conventional discrete types as exemplified in Figure 4.1. Instead, each cluster-based type represents a cluster of named entities that are mentioned in similar contexts. Since the entities with the same conventional type tend to occur in similar context, it is also likely to obtain clusters that implicitly correspond to these types, like person or football team. In the second step, we train a mention typing model on the hyperlinked mentions in Wikipedia, which have been labeled with their cluster-based types through distant supervision. In the third step, mentions in the NED data sets are classified with this typing model. The fourth step involves using those type predictions as features for training an entity candidate ranking model. In the final step, for each entity mention in the NED test sets, the trained ranking model selects the most possible entity candidate in the KB. We use a simple feed-forward neural network model to score the candidates based on local context-independent and global context-based features, which include the aforementioned predictions. Moreover, in order to maximize the contribution from the context, we use five different ways of representing entities, which leads to five different clusterings of them and, thus, five cluster-based types for each entity. By using five different typing predictions together, our system achieves better or comparable results based on randomization tests [**?**] with

respect to the state-of-the-art levels on four defacto test sets.

Here is a summary of our contributions to the literature:

(i) This research is the first at using clustering to obtain cluster-based mention types and using these to improve the NED task.
(ii) We introduce five different ways of cluster-based mention typing based on representing the context around a mention at three different levels. We show that improved NED results are achieved when the different typing models are used together.
(iii) In the candidate generation phase, we propose using the candidates of the co-occurring mentions in the same document, which leads to higher gold recall values than the previously reported results. We publicly share our tool and data sets.

The rest of this chapter is organized as follows. In the next section, we give the related work on NED, mention typing, and clustering. Section 4.2 introduces cluster-based mention typing, where the methods for clustering named entities and then predicting the cluster-based types are presented. Section 4.3 describes how to disambiguate the entities. It starts with our candidate generation method and then explains the local context-independent and global context-based features used to represent the mentions for disambiguation. It ends with the description of our ranking model for disambiguation. Section 4.4 gives the details on our experimental setup. Next, in Section 4.5, we present our results on the mention typing model and disambiguation model, along with a detailed error analysis. Section 4.6 discusses the issues and suggests new ways to extend our research on mention typing and named entity disambiguation. Finally, Section 6 finishes this chapter with concluding remarks.

4.1. Related Work

As introduced in Section 2.1.3, **Named entity disambiguation** is one of the most studied tasks in the NLP literature. The main function of NED is the ranking of possible entity candidates for the recognized mention so as to pick the best one as

the predicted referred entity. There are various approaches to formalize the ranking of the candidates. Early studies [17, 131] ranked the candidates individually by using features like similarity between the candidate and the test document. [71], hypothesizing that entities should be correlated and consistent with the topic of the document, used features that optimize the global coherence. [69, 135] used a two-step approach where they select the most likely candidates in the first step, calculate their coherence score based on their surrounding selected candidates and then use that score in the second step (i.e., the ranking step). Due to its simplicity, we also adapt this two-step approach. Later studies [132] considered the ranking collectively, rather than individually. Graph-based approaches [72, 136] were proposed for collective disambiguation, where the topic coherence of an entity is modeled as the importance of its corresponding node in a graph. Several studies were conducted where different algorithms were used to model node importance, including the personalized version of the PageRank algorithm [137], probabilistic graphical models [138], inter-mention voting [139], random walk [73, 136], minimum spanning tree [140], and gradient tree boosting [141]. Unless heuristics are used, these models are, in general, computationally expensive, as they consider many possible relations between the nodes.

In addition to the way to rank, representing the context, mention and entity is another important aspect in NED. [142, 143] used topic modeling to obtain word-entity associations. However, such models learn a representation for each entity as a word distribution. [144] argued that counting-based distributional models are usually outperformed by context-predicting methods, such as embeddings. That said, improved embedding calculations with word2vec [101] led to many studies. [67, 69, 145] investigated learning entity and word embeddings jointly, which enables a more precise similarity measurement between a context and a candidate. They used the text in Wikipedia as the main source for entity representations. In addition, the knowledge graph of Wikipedia has also been exploited [19]. [146] learned multiple semantic embeddings from multiple KBs. In our study, we obtain the entity embeddings with their EAD-KB approach. However, instead of using multiple KBs, we make use of the context of the same KB in five different levels. When it comes to deep learning approaches, [78] employed Long Short-Term Memory (LSTM) networks with atten-

tion mechanism. [68] used attention mechanism over local context windows to spot important words and [147] expanded this to the important spans with conditional random fields. While these approaches used neural networks with attention mechanism to model the named entities and mentions together and pick the best matching candidate entity, we used a simple LSTM to model the type prediction of the mentions only and then used that information as an extra clue for a simple feed-forward neural network-based ranking model. Other studies [66, 78, 148] modeled context using the words located at the left- and right-hand sides of the mention. Either the sentence or a small window is used as the context boundary. Similar to these studies, we model the left and right context separately. In addition, we propose representing the local and global context separately, in three different ways, which in our results is empirically shown to provide a richer way of characterizing entity mentions.

Mention typing is the task of classifying the mentions of named entities with their context dependent types as introduced in Section 2.1.4. It is a relatively new study area and a specialized case of corpus-level entity typing, which is the task of inferring all the possible type(s) of a named entity from the aggregation of its mentions in a corpus. Some of the recent studies on corpus-level entity typing used the contextual information [149], the entity descriptions in KB [85,150] as well as multi-level representations at word, character and entity level [151]. The way [151] represent the entities in terms of these three levels with increasing granularity resembles our way of considering the context at different scales by representing local and global context separately. In case of the mention-level entity typing or mention typing in short, [20] proposed an entity recognizer called FIGER, which uses a fine-grained set of 112 types based on Freebase [152] and assigns those types to mentions. They trained a linear-chain conditional random fields model for joint entity recognition and typing. [21] derived a very fine-grained type taxonomy from YAGO [22] based on a mapping between Wikipedia categories and WordNet [82] synsets. Their taxonomy contains a large hierarchy of 505 types organized under 5 top level classes (person, location, organization, event, and artifact). They used a SVM-based hierarchical classifier to predict the entity mention types. These studies usually created their training data sets from Wikipedia using a distant supervision method, which is the practice that we also employed. Mention

typing has also recently been used to improve NED. [133] used Wikipedia categories to incorporate the symbolic information of types into the disambiguation reasoning. [153] used type, description and context representations together to obtain entity embeddings. [134] employed CNN with position embeddings to obtain a representation of the mention and the context. [83] formulated NED as purely a mention typing problem. However, all of these studies rely on manually crafted type taxonomies. The main difference of our approach from these studies is that we generated types automatically. We use clustering to partition the named entity space of our KB into clusters, each holding entities that occur in a similar context. Then, each cluster is assigned as a type to the entities in that cluster. This makes our cluster-based types more context oriented than manually crafted types. Moreover, since we obtain multiple clusterings based on different contextual scopes, we ended up having multiple type sets, each exhibiting the characteristics of the context differently, unlike the traditional manually crafted type sets in the literature.

Clustering is a powerful tool to partition the data set into similar groups without any supervision as introduced in Section 2.1.5. There is a large variety of methods in the literature. They can be mainly grouped into partitional (or centroid-based) clustering, such as K-means [154], density-based clustering, like DBSCAN [155], distribution-based clustering, like the Expectation-Maximization algorithm [156], and hierarchical clustering. Among them, partitional clustering, and more specifically K-means, is one of the most practical algorithms due to its simplicity and time complexity. One of the early application areas of clustering includes the clustering of search results [87] in the Information Retrieval field. Later studies categorized named entities in order to improve document retrieval [88,89]. In the NLP field, clustering has been used to group similar words together. [90] introduced Brown clustering which assumes that similar words have similar distributions of words to their immediate left and right. While this method assigns each word to one cluster (i.e., hard clustering), [157] proposed a distributional clustering method which calculates the probability of assigning a word to each cluster (i.e., soft clustering). Word clustering has been used to improve many tasks like statistical language modeling [158], text classification [159], and induction of part-of-speech tags [160]. In the Named Entity Recognition task, [161] clustered

text patterns that represent relations between certain types of named entities in order to better recognize their mentions in text. In mention typing, [162] proposed a joint hierarchical clustering and linking algorithm to cluster mentions and set their types according to the context. Their approach to clustering is similar to ours. However, they rely more on knowledge base taxonomy in order to generate human-readable types. In our case, we do not need to have human-readable types, as mention typing is merely an intermediate step in order to obtain additional contextual clues for named entity disambiguation. In the NED task, the term "Entity Clustering" has exclusively been used for the co-reference resolution (CR) task [163], which is to detect and group the multiple mentions of the same entity within a document or multiple documents (*i.e.* cross-documents). If this task is done on mentions that have no corresponding entries in KB, it is called "NIL clustering" [164]. In these studies, hierarchical agglomerative clustering is mainly used due to its efficiency as it merges similar mentions into a new group recursively in a bottom-up approach. When CR is done within a document, the clustering only considers the merge combinations within that document, which can be in the order of thousands. However, the number of combinations in cross-document CR can be in the order of millions, which requires more efficient clustering algorithms. Some of the proposed methods include a distributed Markov-Chain Monte Carlo approach to utilize parallel processing [165], a discriminative hierarchical model that defines an entity as a summary of its children nodes [166] and the use of latent features derived from matrix factorization of the surrounding context [167]. Moreover, [168] proposed a discriminative model which is trained on a distantly-labeled data set generated from Wikipedia. A recent review of the CR literature is provided by [169]. CR has also been used in a joint task with entity linking [170, 171]. Apart from using mention clustering directly, [172] clustered 10000 lemmatized nouns into 1000 groups based on syntactic relations in order to learn features that are useful for the CR task. While clustering has been explicitly used on mentions of named entities, to the best of our knowledge, our work is the first study on clustering millions of named entities. Moreover, we represent entities at different contextual levels and do the clustering for each level.

4.2. Cluster-based Mention Typing

In the mention typing task, named entities in a KB are in general assumed to be assigned to manually-crafted predefined type(s), and the task is to classify the mentions of named entities to these predefined types based on their context. Considering that there can be millions of named entities in a KB, manually designed predefined types are inherently incomplete. Therefore, we propose using clustering to define the entity types in an unsupervised fashion. We cluster named entities that occur in similar contexts together and assign the corresponding cluster id of an entity as a type label to that entity. As [101] argued, similar words occur in similar contexts, hence have similar word embeddings. That said, similar entities should have similar entity embeddings. When we cluster named entities based on these embeddings, similar entities are expected to be grouped into the same cluster and each cluster is expected to contain entities of similar types based on common contextual clues. Note that these cluster-based types do not necessarily correspond to regular named entity types, and they do not need to. In our work, their only purpose is to represent the context so that the named entity disambiguation model can decide how likely it is that a certain candidate named entity is mentioned in the given context.

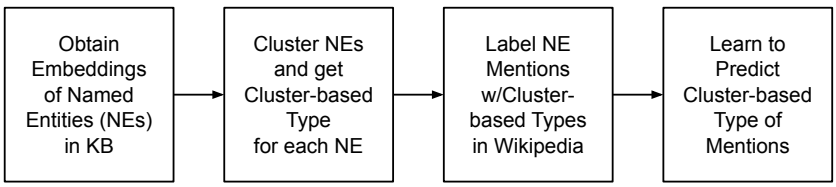

Figure 4.3. The Four Steps of Cluster-based Mention Typing.

As depicted in Figure 4.3, our cluster-based mention typing involves four steps. First, we calculate the entity embeddings based on some contextual representation. In the second step, clustering is applied to group similar entities based on their embeddings. As we get the clusters, we assign the cluster id as a cluster-based type to each entity in that cluster. In the third step, to train a typing model, we prepare a training data set by automatically labeling the hyperlinked mentions of named entities in Wikipedia articles with the assigned cluster-based types. In the final step, we train a typing model with this auto-generated training data. At test time, the typing

model only uses the mention surface form and its surrounding context to predict the cluster-based type. These cluster-based type predictions are used as extra features at the entity disambiguation step.

We cluster entities in five different ways based on different representations of entities. We obtain four embeddings for each entity based on four different (three contextual, one synset based) representations and use them to produce four separate clusterings of the entity space with K-means. To increase the variety, we also use Brown clustering, which requires no embeddings but takes simpler input. At the end, we get five different mention typing models, one for each clustering. Before explaining each step in the following subsections, we first describe how to represent the context of a mention in three different formats.

4.2.1. Three Different Representations of a Mention's Context

The context of a mention is basically the content that resides at the left- and right-hand side of the mention. These are called left and right contexts, respectively. In this research, we represent this context in three different ways. Figure 4.4 shows a document at the left-hand side where three entity mentions are underlined. At its right, it shows the three different ways of representing the context specific to the mention "Democratic Party". For that particular mention, the dimmed sections are not part of the used context.

Word-based Context (WC) is a traditional context in the form of sequence of words adjacent to the mention at its left- and right-hand side. This context has a local viewpoint, since only the words of the sentence that holds the mention are used. This is shown in the second box of Figure 4.4, where the words in the same sentence with the mention are kept as the word-based context. **Surface Form-based Context (SFC)** keeps the surface forms of the mentions at the left- and right-hand side of each mention, excluding all other words in between as shown in the third box. Compared to WC, this includes words farther than the mention into the context. They reflect the topic of the document better than the other words. **Entity-based Context (EC)** only consists

Figure 4.4. A Sample Document and the Different Types of Context used for the Named Entity Mention of "Democratic Party".

of entity ids surrounding the mention, excluding all the words as shown in the fourth box. Considering the fact that co-occurring entities are related and consistent with the topic of the document, EC also reflects the topic of the document. Both SFC and EC present a global viewpoint at the document level compared to the local viewpoint of WC that is mostly based on the sentence level. Having said that, cluster-based types generated with WC-type context may act more like traditional named entity types, as the surrounding words might reflect their semantic roles better. On the other hand, cluster-based types based on SFC and EC may act more like topic labels.

4.2.2. Obtaining Entity Embeddings

We obtain four different embeddings for each entity in KB using Wikipedia articles as training data. Three of those embeddings are based on the three different ways of representing the context of the named entity mentions in text. The forth embedding is based on the synsets (or types in BaseKB terminology) in the YAGO and BaseKB data sets that are associated with each entity. Figure 4.5 gives overviews the process.

WC-based Entity Embeddings (E_{WC}) are obtained by using the *word2vec* tool [101] on Wikipedia articles. *Word2vec* gets the input as sequence of tokens and calculates an embedding for each token (i.e., target token) in the given input based on its window of surrounding tokens at the left- and right-hand side. Note that *word2vec*

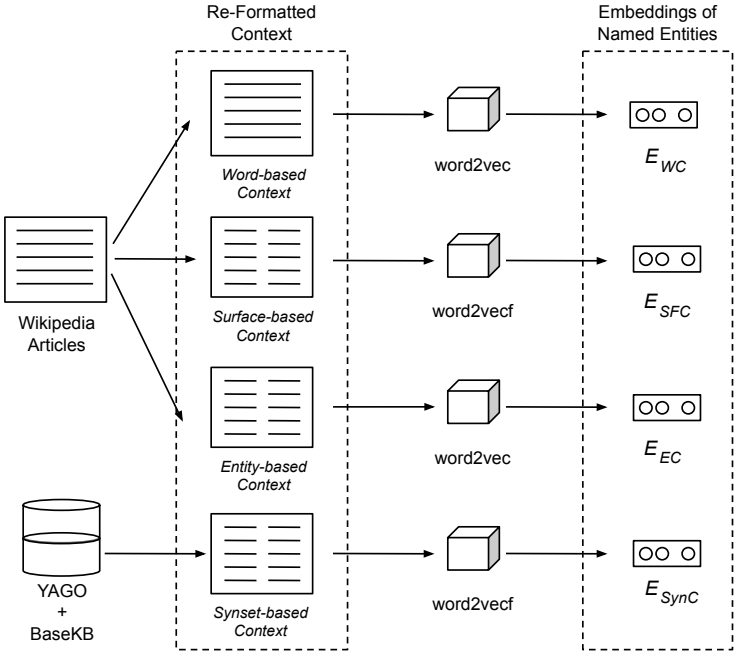

Figure 4.5. Steps of Obtaining Entity Embeddings by using Different Types of Context.

does not use the sentence boundaries as context boundaries as in the definition of WC. Instead, it uses a window of words around the mention. In order to obtain the embeddings for the entities, we reformat the Wikipedia articles based on the EAD-KB approach [146]. We replace the hyperlinked mention (i.e., a-href tag and the surface form together) of each named entity with its Wikipedia id, which is a single unique token that corresponds to that entity. As we run the *word2vec* tool on these reformatted articles, we obtain embeddings for both regular words and entity ids.

SFC-based Entity Embeddings (E_{SFC}) do not rely on the immediate adjacent words of the entity mention as in E_{WC}. It is hard to represent this in linear bag-of-words context as *word2vec* expects. Hence, we use the *word2vecf* tool [106], which is an altered version of *word2vec*. While *word2vec* assumes a window of tokens around the target token as context, *word2vecf* allows us to define the context tokens arbitrarily, one-by-one for the target token. It takes the input in two columns, where the first column holds the target token and the second column has a single context token. As exemplified in Figure 4.6, the input file contains one row for each target and context token pair. We select a window of mentions around each hyperlinked mention in

Wikipedia and use the words in their surface forms to create the input data for the *word2vecf* tool.

Input to get E_{SFC} w/word2vecf:

Barack_Obama	United
Barack_Obama	States
Barack_Obama	Democratic
Barack_Obama	Party
...	...

Input to get E_{SYN} w/word2vecf:

Barack_Obama	wordnet_person
Barack_Obama	wordnet_us_president
Barack_Obama	wordnet_politician
Barack_Obama	wordnet_african_american
...	...

Figure 4.6. Input Samples from the Example Contexts in Figure 4.4 to Obtain E_{SFC} and E_{SYNC} with the word2vecf Tool.

EC-based Entity Embeddings (E_{EC}) are calculated by using the entity ids of the surrounding mentions in the given EC. In this case, we remove all the words in the Wikipedia articles and only keep the entity ids of hyperlinked mentions as tokens. As we run *word2vec* on these reformatted articles, we get an embedding for each mentioned named entity, which is calculated based on a window of the mentioned named entities around it.

Synset-based Entity Embeddings (E_{SYN}) are the only type of entity embeddings that do not reflect the context-based similarity of entities, but their synset-based similarity. In WordNet, a synset is a set of synonymous words grouped together. WordNet uses the synset records to define a hypernym hierarchy between them to reflect which synset is a type of which other. YAGO v3.1 [22] uses those synsets as category labels to represent certain types of named entities, such as person, musician, city etc. Moreover, YAGO extends the WordNet synsets with its own specialized synsets, prefixed "wikicat" (e.g., wordnet_person_100007846 is a hypernym of wikicat_Men in the YAGO synset taxonomy). In YAGO data set, named entities are labeled with one or more synsets. In addition to YAGO, we also use BaseKB Gold Ultimate Edition [152] data set which is based on the last Freebase dump of April 19, 2015. It contains over 1.3 billion facts about 40 million subjects, including 4 million from Wikipedia. It is similar to YAGO, except it has its own simple type taxonomy, independent of WordNet synsets. In our experiments, we combine the type definitions from both YAGO and BaseKB data sets and call them synsets for the sake of simplicity. By combining them, we aim to have a synset for as many named entities as possible. We then use

the associated synsets of named entities as their context tokens as *word2vecf* allows us to define custom context. We give the entity ids in the first column and the associated types in the second column as shown in Figure 4.6. In this process, we do not use all available synsets though. We replace any specialized synset (ones starting with wikicat_*) with its hypernym WordNet synset. We also filter out some BaseKB types that do not reflect type information (e.g., base.tagit.topic and types with user names).

4.2.3. Clustering Named Entities

Now that we have entity embeddings, we can use them to cluster named entities. In order to do that, we primarily use the K-means algorithm due to its simplicity and time complexity. It is a partitional (or centroid-besed) clustering algorithm. After setting the number of centroids (i.e. K) manually at the beginning, it assigns every data point to the nearest centroid. We use the euclidean distance between the entity embedding and the centroid vectors. After the assignment step, the centroids are recomputed. This process continues till a stopping criterium is met[13] . In addition to K-means, we use the Brown clustering which is originally introduced to group words that have similar distributions of words to their immediate left and right. Both algorithms have time complexity that is linear in terms of the number of items being clustered [173] provided that other factors are fixed. Considering that we have over five million named entities in KB, this property makes them very eligible for our experiments.

As in Figure 4.7, we give entity embeddings (E_X) to K-means as input. In case of Brown, we use the Wikipedia articles in the Entity-based context format. After getting clusters (C_X), the cluster ids are assigned to entities as cluster-based types.

It is important to note that each clustering of the named entity space breaks that space into groups. In terms of the named entity normalization task, the discriminative power of a clustering depends on how well it distinguishes the correct candidate for a named entity mention from the other candidates. The ideal case occurs when the

[13]We stop iterating when there is no more than 1% change in the assignment of entities to the clusters. Also, we allow at most 50 iterations.

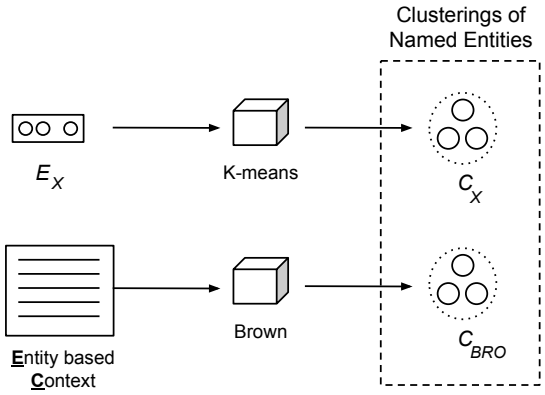

Figure 4.7. Creating Different Clusterings of Named Entities with the K-means and Brown Clustering Algorithms, where $X \in${WC,SFC,EC,SynC}.

correct named entity for a mention is placed in a different cluster from the other most likely candidates and the cluster of the mention is also predicted to be the same as its corresponding named entity. Using high number of clusters makes the clustering more discriminative as each cluster corresponds to lower number of entities. However, that also makes the typing model task harder as it increases the ambiguity. In Section 4.4.3, we explore the right number of clusters for our experiments. Moreover, since we have five different clusterings and each breaks the entity space differently, using a combination of them is expected to make the aggregate discriminative power even higher. Hence, using multiple clusterings can be seen as an alternative to using higher number of clusters. Note that Section 4.4.3 also explains our heuristic to select the best combination in order to achieve a better performance at the disambiguation step.

4.2.4. Preparing Training Data for the Typing Model

So far, we have described our approach for clustering named entities and assigning the cluster ids to entities as types. In order to train a typing model to predict the type of an entity mention in an input text, we need training data that includes mentions labeled with these types. We create the training data from the hyperlinked mentions in Wikipedia articles using distant supervision. We label each hyperlinked named entity mention with its assigned cluster-based type. Since we have five different clusterings, we produce five different training data sets.

As described in Section 4.2.1, we represent the context of a mention in three different formats and three of our clusterings (namely, C_{WC}, C_{SFC}, C_{EC}) are obtained according to the corresponding context formats. Take C_{WC} for example. Clusters in C_{WC} hold entities that are similar to each other in terms of word-based context (WC). It is compatible to train a typing model for C_{WC} with input in the WC format because labels are created from the characteristics that made up the input in that format. Same is true for C_{SFC} that goes with the surface form-based context (SFC) and C_{EC} with entity-based context (EC). Figure 4.8 exemplifies three training instances in WC, SFC, and EC formats, respectively. They are generated based on the example in Figure 4.4. Each instance consists of three input fields in addition to the label; surface form, left and right context. Surface form is the common input. Like surface form, contexts in the first two cases are in the form of words. In case of the third, it is in entity ids.

Label	Surface Form	Left Context	Right Context
Cluster_10	Democratic Party	A member of the	, he was the first African American to be elected to the presidency .
Cluster_232	Democratic Party	Barack Obama U.S.	
Cluster_43	Democratic Party	Barack_Obama United_States	

Figure 4.8. Example Training Instances from Figure 4.4 in WC, SFC, and EC Formats, Respectively.

In case of C_{SynC} and C_{BRO}, we use word-based context and surface form-based context, respectively. In order words, we use the same formatted input for C_{WC} and C_{SynC} pair and C_{SFC} and C_{BRO} pair with the exception of the labels. Note that clusters in the C_{SynC} hold entities that are associated with similar synsets. Synsets are like semantic categories and local context is better suited to infer which synsets the mentioned entity is associated with. Hence, the word-based context is used. On the other hand, using global context is better suited to infer C_{BRO} based types. We choose to use the surface form-based context over the entity-based context. This is due to the fact that surface form-based context can easily be obtained by gathering the surface forms of the surrounding named entities. However, entity-based context is only available after we get the best predictions from the first stage of our two-stage disambiguation approach described in Section 4.3. In other words, by using the surface form-based context, we are making the Brown-based mention typing model available at the first stage, which increases the success rate of the first stage.

4.2.5. Specialized Word Embeddings for the Typing Model

When it comes to predicting the cluster of a mention with word-based context, it is common practice to use word embeddings that are obtained from a data set that is similar to the domain data. In our case, it is Wikipedia, which is a widely used source to obtain word embeddings in the literature. In our experiments, we did not use regular word embeddings calculated with word2vec. Instead, we propose two word embeddings that are more discriminative at predicting the cluster of a given mention. We use these word embeddings together to represent the word-based input for the mention typing model.

Cluster-centric Word Embeddings (W_{CC}): We hypothesize that word embeddings, which are obtained from the data that reflect the characteristics of the problem, may be more effective at solving that problem. In this case, the problem is to predict the cluster-based type of a mention. Hence, we inject a piece of cluster information into the context in hope that word embeddings are being influenced by their presence and become better at predicting the cluster-based type. In order to accomplish that, in Wikipedia articles, we filter out the html tag (i.e., a-href) of the named entity mentions and leave its surface form alone. At the same time, we add a special token to the right of that surface form. That token corresponds to the assigned cluster id of the mentioned named entity. Embeddings of those words that are close to the specific cluster token are expected to be affected by that and become indicators of that cluster. We use the word2vec tool on this modified data set. Since the clustering of named entities is done at training time and does not change based on the given test input, the same embeddings that are obtained at training time are used at the test time. We obtain different word embeddings for each of the five named entity clusterings, since entities are assigned to different clusters in each case.

Surface Form-based Word Embeddings (W_{SF}): While word embeddings in W_{CC} are only based on the words surrounding the named entity mention, another type of word embedding can be obtained with the words inside the surface forms exclusively.

In order to do that, we use the Anchor Text data set[14] of DBpedia [175]. It contains over 150 million (entity_id, surface_form) records, one for each mention occurrence in Wikipedia. We extract around fifteen million unique such records for almost all named entities in our KB, along with their frequencies. We name this data set as the **Surface Form Data Set**. Since this data set only includes the surface forms of the entities, but not their surrounding sentences, we use the associated cluster ids of entities as context tokens with the word2vecf tool. In other words, for each word in the surface form of a named entity, the cluster id of that entity is given as a context word, which results in (surface_form_word, cluster_id) pairs. We also take the frequencies of the surface form records into account. The more frequent one surface form is, the more important it is. Therefore, while creating the training data set for W_{SF} from the (surface_form_word, cluster_id) pairs, the number of instances included for each surface_form_word is proportional to the logarithm of its frequency in the Surface Form Data Set.

4.2.6. Model to Predict Cluster-based Types

Now that we generate a training data that includes labeled mentions with their corresponding named entities' cluster-based type, we are ready to train a mention typing model to predict those types. At the end, we end up having five typing models: *Word* model based on C_{WC}, *Surface* model on C_{SFC}, *Entity* model on C_{EC}, *Brown* model on C_{BRO} and *Entity* model on C_{EC}.

We experiment with two different typing model architectures: Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) models. CNNs are widely used for text classification, whereas LSTMs are effective at modeling sequences of items. Both models receive the same three-part input described in the previous section. However, since we have two sets of word embeddings for surface words as described in 4.2.5, we can represent the Surface-part as two inputs for both models, one with W_{CC} and the other with W_{SF}. In our experiments, we seek to compare both models and

[14]We use the English version of the Anchor Texts data set from DBPedia Downloads 2016-10 available [174].

pick the best performing one. Details on these models are given below.

4.2.6.1. LSTM-based Typing Model. In this model, each input part is given to a separate LSTM layer and their output is concatenated and fed into the final softmax layer as illustrated in Figure 4.9. $Surface_1$ gets the Cluster-centric Word Embeddings (W_{CC}), which are better at discriminating cluster-specific words. $Surface_2$ gets the Surface Form Word Embeddings (W_{SF}), which are optimized for discriminating cluster-specific surface words seen in large set of surface forms. We use dropout layers [176] before and after the LSTM layers in order to prevent overfitting. Our preliminary experiments led us to using bi-directional LSTM (BiLSTM) for the $Surface_x$ and uni-directional one for the left and right ones. Moreover, reversing the order of words in the right context produces better results. We argue that words that are closer to the mention are more related to the mention and play more important role at the prediction.

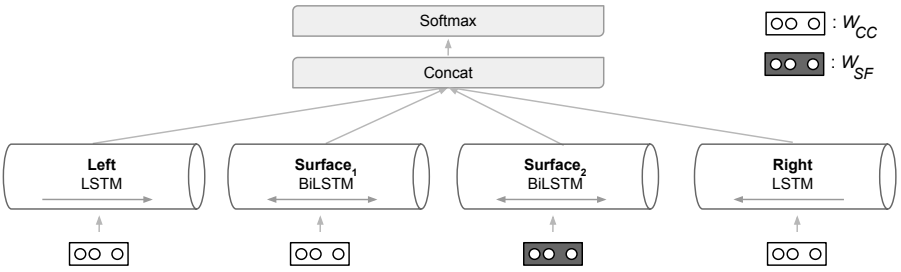

Figure 4.9. LSTM-based Model for Mention Typing.

4.2.6.2. CNN-based Typing Model. CNN-based model is very similar to the LSTM-based model, except that the input parts are given to the convolution layers. As pictured in Figure 4.10, each input part is fed into the six convolution layers with kernel sizes 1 through 6. These layers model the sequence of words in different lengths, like n-grams in a language model. We use hyperbolic tangent as an activation function over the convolution layers and apply max pooling layer prior to merging all into one layer before feeding to the softmax layer. Like in the LSTM-based model, we use dropout after the word embeddings and before the softmax layer.

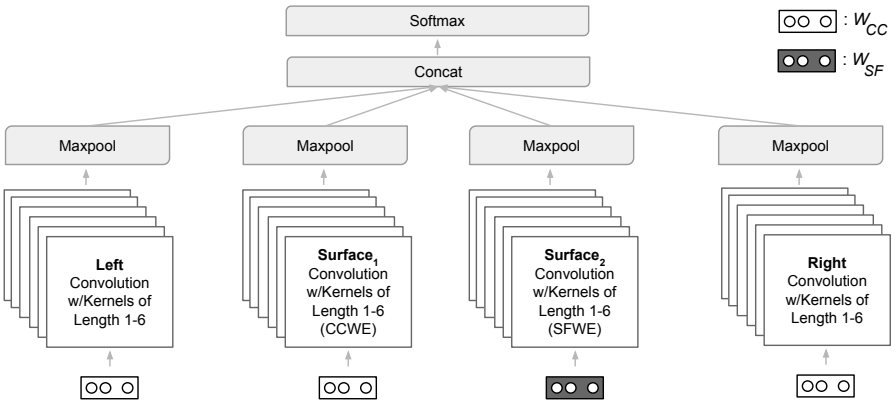

Figure 4.10. CNN-based Model for Mention Typing.

4.3. Disambiguating Named Entities

Disambiguation of named entities is often formulated as a ranking problem. First, a set of candidate named entities for a given mention are obtained. Then, each of them is scored individually, resulting in a ranked list of candidates. Finally, the highest ranked candidate is returned as the final prediction. In our experiments, we use a two-stage approach for ranking. At the first stage, we train our model and rank the candidates with the available features and get a ranking probability for each candidate. As a result of the first stage, the highest scored candidates are used to define the entity-based context shown in Figure 4.4. This allows us to run the *Entity* typing model, which accepts the context in terms of entities only. At the second stage, we use those ranking probabilities to define new features that encapsulate future insight. With the addition of new features, we again train our ranking model and get the final ranking of the candidates. In the following subsections, we first describe how we obtain the candidates. Next, we explain our disambiguation model by first describing the features we used and then, the model itself.

4.3.1. Candidate Generation

In the literature, most of the studies do not mention how to index and search entities to generate candidates based on the surface form of a mention. Some of the ones [18, 177] providing this information report that they use Apache Lucene [178]. Lucene provides fuzzy search property which tolerates certain percentage of mismatch

at character level with the Damerau-Levenshtein distance. However, such off-the-shelf tools consider each mention individually and do not use contextual clues from other mentions in the same document. Moreover, they do not provide additional information, such as the type of the matched surface form, which we use as a feature (i.e. *surface-form-type-in-binary* in Table 4.2) at the disambiguation step. Hence, we implemented our own candidate generation tool.

For the indexing, as in [72], we use all available surface forms of named entities in DBpedia and Wikipedia (including anchor texts, redirect and disambiguation records). In other studies, while some [68, 179] used additional web corpora, others [135, 140] used only Wikipedia. We index surface forms based on character tri-grams. In case of searching, the pseudocode of our search procedure is in Algorithm 1.

Our algorithm works in two stages. In the first stage, it starts with searching named entities matching with the mention surface form (Lines 17-25). It first retrieves indexed surface forms that have at least certain amount of character tri-gram overlap (T=60%) with the search query (i.e., mention surface form). Then, it picks the ones that have the ratio of the edit distance at the character level to the query length less than certain value (E=25%). If not, then it checks for the word overlap. It can tolerate up to one word mismatch (D=1) if query contains more than one word (W=2). After this selection, all matched candidates are returned. This first stage is similar to what off-the-shelf third party indexing and search tools provide.

In the second stage of our algorithm, we expand the candidates of each mention with the candidates from other mentions in the same document, since the same entity can be mentioned more than once. We do this in two steps. The first step looks at if surface form of one mention is completely seen in the other's (Lines 6-8). For example, "Barack Obama" can be first mentioned in full name and then as "Obama" in the same document. The second expansion step uses the most frequently co-occurring named entities (Lines 9-14) observed in Wikipedia articles. We only include the ones that have a surface form that includes the mention surface form in itself. At the end, we

```
1: procedure getMatchedNamedEntitiesForDocument(set_of_mentions, N)
2:     matchedEntitiesForMentions = []
3:     for each mention in set_of_mentions do
4:         matchedEntities = getCandidatesForMention(mention)
5:         add matchedEntities to matchedEntitiesForMentions
6:     for each mention_a and mention_b pair in set_of_mentions do
7:         if mention_a is completely seen in mention_b then
8:             add matchedEntities_b to matchedEntities_a
9:     for each mention_a in set_of_mentions do
10:        for each matched_entity in other mentions do
11:            for each most frequently co-occurring entity of matched_entity do
12:                for each surface_form of co-occurring entity do
13:                    if mention_a is completely seen in surface_form then
14:                        add that entity to matchedEntities_a
15:    return scoreCandidatesThenPickTopN(matchedEntitiesForMentions, N)
16: procedure getCandidatesForMention(query)
17:    matchedEntities = []
18:    for each entity in the knowledge_base do
19:        for each surface_form of entity do
20:            if trigramOverlap(surface_form, query) ≥ T% then
21:                if editDistance(surface_form, query) ≤ E% of query length then
22:                    add entity to matchedEntities
23:                else if numWords(surface_form) ≥ W and numWordDiff(surface_form, query) ≤ D then
24:                    add entity to matchedEntities
25:    return matchedEntities
```

Figure 4.11. Pseudocode of the candidate generation algorithm.

order the final set of candidates by scoring each candidate based on a specific formula[15] (Line 15) and return the highest scored 100 candidates as the final output.

4.3.2. Ranking Features

Each candidate is scored according to certain properties. These properties are named as features and their scalar values are turned into a vector which is given to the ranking model as input. In our experiments, we use a total of twelve features. In order to understand how they are calculated, we first give the definitions of the relevant variables, sets of values, and functions in Table 4.1.

[15]We observed that the following candidate scoring gave the highest recall on the AIDA.train set: score = *entity_frequency* + *num_of_occ_in_test_doc**100 - *jaro_winkler_distance**10000, where *entity_frequency* is the total number of times the entity is seen in our Surface Form Data Set. We observe that Jaro-winkler distance performs better than Levenstein edit distance.

Table 4.1. Variables, Sets, and Functions used at Defining Ranking Features.

Variable:	**Description**
SF_m	Surface form of the mention
SF_c^{best}	Closest surface form of the entity c wrt SF_m given by the candidate generator
D_c	Doc2vec embedding of the candidate entity c
D_t	Doc2vec embedding of the test document t
E_c	Entity embedding of the candidate entity c
T_c^{SF}	Type of the surface form SF given by the candidate generator
T_c^e	Type of the candidate entity c
P_c^n	Probability given by the typing model n for the entity c being in its cluster-based type
R_c	Ranking probability of candidate entity c calculated by the first stage ranking
Set:	**Description**
S_c	Set of all surface forms for the candidate entity c seen in Surface Form Data Set
C^m	Set of all candidates for the mention m
C_c^t	Set of all occurrences of the same candidate c in test document t
C_c^{prev}	Set of all previous occurrences of the candidate c wrt the mention m
C_c^{next}	Set of all future occurrences of the candidate c wrt the mention m
C_{topNxM}	Set of highest ranked N candidates in the surrounding window of M mentions
Function:	**Description**
$freq(SF_c)$	Number of times the surface form SF is seen for entity c in Surface Form Data Set
$dist(A, B)$	Levenstein edit distance between two strings A and B
$cos(A, B)$	Cosine similarity between two vectors A and B
$log(A)$	Natural logarithm of the scalar value A
$avg(\sum)$	Average of the sum of the values
$binarize(A)$	Binarized code of the number A
$argmax(S)$	The maximum value among the set of values S
$argsecmax(S)$	The second maximum value among the set of values S

Variables in Table 4.1 represent either strings, scalar values, or vectors. SF_m and D_t are the only variables that are not related to the candidate. Whereas, D_c, E_c, and T_c^e are candidate-specific and calculated offline, hence they do not depend on the mention being considered. The rest of the variables depend on the mention. **Sets** are variables that represent a set of values. For example, S_c corresponds to all the surface forms of the candidate entity c. The rest includes set of candidate entities that are determined based on specific position. **Functions** are applied on these variables and sets in order to define more detailed features.

We group the ranking features into four main categories as shown in Table 4.2. In the literature, the features used for disambiguation are basically divided into two main categories, namely local context-independent and global context-based features. In order to make it more explicit, we further break down the global features into three sub-categories. Those are mention-level, document-level and second stage features.

Table 4.2. List of Features used by the Ranking Model and their Descriptions.

Local Features :	
surface-form-edit-distance	dist(SF_c^{best}, SF_m)
entity-log-freq	log($\sum_{s \in S_c} freq(SF_c^s)$)
surface-from-type-in-binary	binarize($T_c^{SF_c^{best}}$)
entity-type-in-binary	binarize(T_c^e)
id^t-typing-prob	P_c^n, where $n \in$ {*Word*,*Surface*,*Synset*,*Brown*,*Entity*}
Mention-level Features :	
avg-surface-form-edit-distance	avg($\sum_{c' \in C^m}$ dist($SF_{c'}^{best}$, SF_m))
max-diff-surface-form-log-freq	$argmax_{c' \in C^m}$ $freq(SF_{c'})$-$freq(SF_c)$, if $freq(SF_c)$ is not max $freq(SF_c)$ - $argsecmax_{c' \in C^m}$ $freq(SF_{c'})$, otherwise
max-diff-doc-similarity	$argmax_{c' \in C^m}$ cos($D_{c'}$,D_t)-cos(D_c,D_t),if cos(D_c,D_t) is not max cos(D_c,D_t) - $argsecmax_{c' \in C^m}$ cos($D_{c'}$,D_t), otherwise
max-diff-id^t-typing-prob	$argmax_{c' \in C^m}$ $P_{c'}^n$ - P_c^n, if P_c^n is not max P_c^n - $argsecmax_{c' \in C^m}$ $P_{c'}^n$, otherwise
Document-level Features :	
max-id^t-typing-prob-in-doc	$argmax_{c' \in C_c^t}$ $P_{c'}^n$
Second-stage Features :	
max-ranking-score	$argmax_{c' \in C_c^{prev}}$ $R_{c'}$ if C_c^{prev} is not empty $argmax_{c' \in C_c^{next}}$ $R_{c'}$ if C_c^{next} is not empty & $T_{c'}^{SF} = WikiID$ 0 , otherwise
max-cos-sim-in-context	$argmax_{c' \in C_{topNxM}}$ cos(E_c, $E_{c'}$) * $P_{c'}^n$

Local Features are those that consider the individual properties of the candidate without taking into account any other candidate. For example, the feature *surface-form-edit-distance* considers the edit distance between the most similarly matched sur-

face form (SF_c^{best}) of the candidate in our Surface Form Data Set and the actual surface form (SF_m) of the mention. The more distant they are, the less likely the candidate is the actual one. In order to take into account the popularity of the candidate, the feature *entity-log-freq* uses the total number of times that named entity is seen in the Surface Form Data Set. It uses the logarithm to scale down and smooth the values. The more popular the candidate is, the more likely it might be mentioned. Another local feature *doc-similarity* is the cosine similarity between the doc2vec[16] embeddings of the test document and the Wikipedia article of the candidate. The similarity between the two corresponds to the similarity between the context of the test document and the context in which the candidate is expected to be mentioned.

Table 4.3. Entity Types and Surface Form (SF) Types in SFDB.

	Type	**Description**
Entity Types		**Entities labeled w/following YAGO synsets and BaseKB types**
	Person	wordnet_person_100007846, people.person, ...
	Organization	wordnet_organization_108008335, organization.organization, ...
	Location	wordnet_site_108651247, location.location, ...
	SportsTeam	wordnet_team_108208560, wordnet_club_108227214, ...
	Misc	*Any other dissimilar synset or type*
		When ...
Surface Form Types	WikiID	SF is same as the $MainTitle$ of the entity
	Redirect	SF is labeled as "redirect" in Wikipedia dump files
	Disambiguation	SF is labeled as "disambiguation" in Wikipedia dump files
	FirstName	the entity is Person-type and SF is a known first name (eg. John)
	Surname	the entity is Person-type and SF is a known surname (eg. Smith)
	FirstWord	SF is the first word in the main title of the entity
	LastWord	SF is the last word in the $MainTitle$ of the entity
	PrefixPhrase	SF is prefix phrase in the $MainTitle$ of the entity
	SuffixPhrase	SF is suffix phrase in the $MainTitle$ of the entity
	BeforeComma	SF is the phrase before comma in the $MainTitle$ of the entity
	OrgAcronym	SF comprised of first letters of $MainTitle$ of the entity

The two features *entity-type-in-binary* and *surface-form-type-in-binary* are binary vectors that encode the type of the candidate entity and the surface form, respectively. These types are based on the categorization of the entities and surface forms defined in Table 4.3. We assign a type to each named entity in our KB based on whether

[16]We train Gensim's Doc2vec tool on Wikipedia articles with embedding size=300 and set the other parameters to their default values in order to obtain document embeddings

it is labeled with one of the pre-defined YAGO synsets or BaseKB types. We define five main types. While *Person*, *Organization*, *Location* and *Misc* types are common in the named entity recognition literature, we added the *SportsTeam* type since many mentioned entities in AIDA data sets are of this type. In case of the surface form types, we assign one or more applicable types to the SF_c^{best}. Other than *Redirect* and *Disambiguation* types in Table 4.3, the rest is based on the similarity between the surface form and the $MainTitle$ of the entity, which is obtained from its Wikipedia ID by replacing any underscore character with the blank space and discarding any phrase given in parenthesis. Certain combinations of these two types can give clue about the likelihood of the candidate. For example, it is more likely that mention of *Person* type entity is made in surface form of type *FirstWord* or *Organization* type entity in *OrgAcronym* type surface form.

The final local feature *id^t-typing-prob* is the probability (P_c^n) of labeling the mention with the pre-assigned cluster-based type of the candidate by the typing model n. More precisely, when the typing model is applied to the input text containing the mention, it outputs the probability of the mention belonging to each of the cluster-based types. As we know the pre-assigned cluster-based type of the candidate, the probability of that type is set as P_c^n. Since we have five different typing models, this feature contributes with five separate values. Note that each feature that includes P_c^n does that.

Mention-level Features consider the relative value of individual candidate's feature with respect to the other candidates' for the same mention (C^m). The first feature *avg-surface-form-edit-distance* takes the average of all *surface-form-edit-distance* feature values of C^m. Averaging helps us represent the edit distance of an average candidate. We can argue that the higher it is, the more likely that none of the candidates are the actual entity being mentioned. The features prefixed with *max-diff-* compare the candidate's corresponding feature value with respect to the best value seen for the mention. If the value is already the best, then it uses the second best value. The higher positive value it is, the more likely that the candidate is the actual one. Or, the lower negative value it is, the less likely that the candidate is the one. *max-diff-* converts a

context independent value to a context dependent by representing it with respect to the highest (or the next highest) seen value.

Document-level Features consider all occurrences (C_c^t) of the same candidate in the same test document. We have one such feature and it uses the highest seen P_c^n value for the considered candidate. When the same entity is seen as candidate in multiple mentions, each has its own P_c^n value depending on the position of the mention in the document. The highest of them increases the chance of the other occurrences of the same candidate with the lower P_c^n in the same document.

Second-stage Features are only available after the ranking model with the previously described features is trained and applied on the candidates once. These features use the ranking probability of each candidate (R_c). The first feature *max-ranking-score* keeps the highest R_c among the previous occurrences of the same candidate (C_c^{prev}) in the same document. If there is no previous occurrence of the candidate before the considered mention, then it looks at the future occurrences (C_c^{next}) only if $T_{c'}^{SF}$ of that candidate is *WikiID*. This means that having an occurrence of the candidate with high R_c increases the chance of the next occurrences of the same candidate in the same document. The same affect for the early occurrences can only happen if the future occurrence is mentioned in its full title. The second feature *max-cos-sim-context* uses the cosine similarity between the entity embeddings (E_c) of the candidates[17] . This feature is set to the highest cosine similarity value between the candidate and the highest ranked N candidates in the surrounding window of M mentions.

4.3.3. Neural Network Model for Disambiguation

Our neural network for ranking candidate named entities is a two-layer feed-forward neural network and one softmax layer[18] at the top. We use dropout layer after each feed-forward layer. We turn the ranking task into a binary classification task,

[17]We use co-occurrence based entity embeddings obtained with the word2vecf tool. We set the window size to 5 and number of iterations to 20 on top of default settings.

[18]In our experiments, we observed that using the softmax layer instead of logistic regression in the PyTorch provides higher results for this binary classification task.

where we classify each candidate as true or false candidate at the training time and then use the output probability of the true class to rank the candidates at the test time. For each candidate, we create its own input vector, which includes all numeric values for the features described in Section 4.3.2. Since we do re-ranking, we have two disambiguation models in our experiments, the first one does the initial ranking and the second one does the final ranking based on additional ranking insight obtained from the first model. In both cases, we use the same network architecture, except the fact that the first model does not include the second-stage features. The values for those features are calculated from the output of the first model. Hence, in case of the test sets, we first need to run the first model on them and get the ranking scores for each candidate. After that, we can run the second model to get the final ranking results.

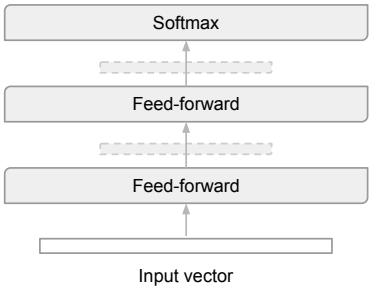

Figure 4.12. Neural Network Structure for our Disambiguation Model.

4.4. Experimental Setup

4.4.1. Training and Test Data Sets

4.4.1.1. Data Sets for the Mention Typing. We derived our data sets from Wikipedia which includes over five million well-written documents, each describing one named entity. Most of the mentions of named entities and concepts in the documents are manually annotated with HTML anchor tag. However, Wikipedia Editing Guidelines suggest authors to annotate only the first occurrence of a named entity in the Wikipedia articles, which means that most of the mentions are not annotated in the articles. Hence, we try to auto-annotate the remaining occurrences in the documents. To do this, we look for the word sequences (with greedy match) that are previously annotated in the same document and then auto-annotate them at the rest of the document. This process increases the number of mentions considerably with an acceptable error rate.

However, we did not try to annotate mentions with the partial names.

Table 4.4. Statistics on the Data Sets for the Mention Typing Models.

Data Set	Num. of Instances	Average Num. of Tokens per Context
WC-SmallTrain	981,000	22.2
WC-LargeTrain	9,828,000	22.2
WC-Test	9,700	22.2
SFC-SmallTrain	500,000	28.9
SFC-LargeTrain	5,000,000	28.9
SFC-Test	50,000	29.0
EC-SmallTrain	500,000	14.8
EC-LargeTrain	5,000,000	14.8
EC-Test	50,000	14.8

After this pre-processing step, we create a separate data set for each context representation. Note that we represent the context in three different levels as described in Section 4.2.2. In case of word-based context (WC), we break the Wikipedia articles into sentences and collect only those sentences that have at least one named entity mention and the length is between 10-50 words. We use the sentence boundary detector tool and tokenizer in the Stanford CoreNLP [180]. We ended up having 46.8M such sentences. Out of those, we randomly selected sentences and created WC-* data sets. In case of surface form-based context (SFC), we start with the same sentences and then for each mention, we get the mention surface forms of the previous and next 10 mentions in the same document. Again we randomly selected instances and created SFC-* data sets. For the entity-based context (EC), around each mention we collected the previous and next 10 named entities in the same document. This is called EC-* data sets. Sample training instances are already exemplified in Figure 4.8 in Section 4.2.4. For each type, we created small and large training sets and a test set as given in Table 4.4. The number of instances and the average number of tokens in the context (left and right combined) are also provided.

4.4.1.2. Named Entity Disambiguation Data Sets. There are a number of publicly available data sets with different characteristics for the NED task. For training, development and test purposes, we use **AIDA** [72], which is derived from Reuters news

articles of the CoNLL 2003 NER task. Being the widely used and largest NED data set, it comes in three pieces: AIDA.train (train), AIDA.testa (dev), and AIDA.testb (test). We report the results on AIDA-testb as in-domain evaluation results, since all our training is done on AIDA-train. In order to see how the model that is trained on AIDA.train achieves on the data sets that exhibit different characteristics, we test that model on a number of other test sets. This is called cross-domain evaluation. We also use **MSNBC** [71], **AQUAINT** [181] and **ACE2004** [135], which are the next three most frequently used test sets. ACE2004 is a subset of ACE2004 Coreference documents, while AQUAINT contains news articles from the Xinhua, New York Times and Associated Press. Since they are also used for wikification, they include Wikipedia concepts apart from named entities. As the recent studies used the revised versions of these data sets prepared by [73], we report our results on these revised versions as well. Apart from these, we also consider three more test sets in order to observe how our system performs on shorter documents. **KORE50** [182] includes 50 short sentences on various topics such as celebrities, music etc. Most of the mentions are single-word such as first names, which makes the deduction of the actual mentioned named entity very difficult. **RSS-500** [183] includes short formal text collected from a data set containing RSS feeds of the newspapers compiled in [184]. The texts are on a wide range of topics such as world, business, science etc. **Reuters-128** [183] is a small subset of the well-known Reuters-21587 corpus containing short news articles about economy.

Details about these data sets are given in Table 4.5. Note that since we use Wikipedia as our reference KB, we map the DBpedia-based annotations in KORE50, RSS-500 and Reuters-128 to the corresponding Wikipedia-based IDs. The AIDA sets have the most annotations among all, which makes them suitable for training and development. KORE50, RSS-500, ACE2004, and Reuters-128 are the sets that have the smallest average number of mentions per document (Avg#M/D). In terms of sentence-based context size, the KORE50 and AIDA sets have the smallest average number of words per sentence (Avg#W/S). Some of these data sets include NIL annotations, which have no a corresponding named entity at the reference KB. As a design decision, we exclude such annotations and report the results accordingly in our experiments.

Table 4.5. Statistics on the Disambiguation Data Sets.

Data Set	**Domain**	**RefKB**	**#Mentions**	**Avg#M/D**	**Avg#W/S**
AIDA.train	news	Wikipedia	18,448	19.5	15.7
AIDA.testa (dev)	news	Wikipedia	4,791	22.2	17.2
AIDA.testb (test)	news	Wikipedia	4,485	19.4	14.5
MSNBC	news	Wikipedia	656	32.3	28.8
MSNBC^{rev}	news	Wikipedia	655	32.8	27.5
AQUAINT	news	Wikipedia	730	14.3	28.5
AQUAINT^{rev}	news	Wikipedia	722	14.4	28.5
ACE2004	news	Wikipedia	255	4.5	37.0
ACE2004^{rev}	news	Wikipedia	256	4.5	38.6
KORE50	mixed	DBpedia	143	2.9	14.6
RSS-500	RSS-feeds	DBpedia	517	1.0	32.2
Reuters-128	news	DBpedia	621	4.9	28.0

4.4.2. Evaluation Metrics

During the evaluation of the cluster prediction model and the disambiguation model, we report the results in standard micro-average F1-score. It is the harmonic mean of the precision and recall measures. Precision measures the percentage of machine's predictions that are correct compared to the gold (human) annotations. On the other hand, recall measures the percentage of the gold annotations that are predicted correctly by the machine. In case of the disambiguation results, a number of studies reported their results in bag-of-title (BoT) F1-score [135, 181], which is designed for Wikification systems. It is used to evaluate a NED system for indexing purposes. Hence, it discards the duplicate gold annotations and predictions in a document and uses the same F1-score metric on the filtered out numbers. Moreover, we report our results in InKB accuracy in case of the AIDA test set due to the convention in the literature. On the other test sets, we use a threshold on the ranking probability and do not assign an entity to a mention if our system is not confident with the assignment (i.e., the ranking probability is below the threshold). In our experiments, we use 0.03 as the threshold.

4.4.3. Optimizing Clustering for Better Disambiguation

Important thing before getting into the experimental results is tuning of the clustering for better disambiguation. As described in Section 4.2.3, how we cluster named entities determines the effect of the discriminative power of clustering on the disambiguation step. The more clusters we use, the fewer entities each cluster has, hence the more discriminative the clustering becomes. For example, consider the cluster containing the gold standard entity. When we have fewer entities in that cluster, we expect it to be relatively easier to identifying the correct entity among the others, compared to when we have more entities in the same cluster. However, when we have more clusters, this makes the typing model difficult to solve due to the increase in the number of classes (i.e. types). There are two main factors we considered for optimization. The first is the window size chosen while calculating the entity embeddings on which we run the K-means algorithm. The second and more important factor is the number of clusters to be generated by the clustering algorithms. However, it is not feasible to try all possible combinations and measure which one achieves the best performance at the disambiguation step. That is because for each combination, we need to get the clusterings, train their typing models, train the ranking model on their predictions and then finally measure the disambiguation success. Instead, we introduce the following three-step approach.

The first step is to produce a wide range of clusterings for the named entity space with different window sizes for embeddings and different numbers of clusters. The second step includes selecting a good sample of clusterings and training a typing model with each selected clustering. Since we use multiple mention typing models for disambiguation, the third step is to select one clustering for each typing model so as to achieve high at the disambiguation step. In order to choose this combination, we also propose a heuristic that minimizes the number of times we train the ranking model. Below, we explain each step in more detail.

The first step involves creating various clusterings of named entities for each of the five clustering approaches. The first and primary parameter is the number of clusters.

In case of the *Brown*-based approach, we run the tool with number of clusters from 1000 to 1500 in 100 incremental steps. For the other four clustering approaches, we run the K-means algorithm with number of clusters from 600 to 3000 in 200 incremental steps. The second parameter is related to the calculation of the entity embeddings which are only used by K-means. In case of *Synset*-based approach, since there is no sequential context, we use word2vecf and the only parameter we change is the *iteration*. For the other three approaches, we use *word2vec* and experiment with different values of the *window* parameter. We use window sizes of 2 and 3. Using other values does not provide any better results. We set the *iteration* parameter for the *Synset*-based approach to 20. At the end of the first step, we get a total of 70 clusterings.

At the second step, we pick one or more parameter combinations for each clustering approach. However, instead of selecting randomly, we proposed a metric that helps us evaluate each clustering based on how well it might help discriminate the candidates for the benefit of the disambiguation model. This metric is called Average Gold Candidate Cluster Size (AGCCS). When we group the candidates of the mention based on the clustering at hand, we end up having clusters of candidates, or candidate clusters. The one that holds the gold (true) candidate is called gold candidate cluster. The smaller the size of the gold candidate cluster is, the less number of candidates are being favored by the cluster prediction model at the disambiguation step. In other words, it becomes more discriminative. For each parameter combination, we calculated AGCCS value on the AIDA.train data set, which includes over 18000 mentions with an average of 84 candidates for each mention.

Figure 4.13 shows how AGCCS changes, as we increase the number of clusters. Generally, it drops, because using more clusters leads to less number of entities inside the clusters, hence the smaller size for the gold candidate cluster. Note that AGCCS generally decreases fast at first but does not drop below 2.5. The dots in Figure 4.13 show the chosen cases that we train a typing model for. We particularly pick the ones with the lowest AGCCS value. Moreover, we also consider some other local minimums, since they are obtained with less number of clusters. At the end, we have 19 different

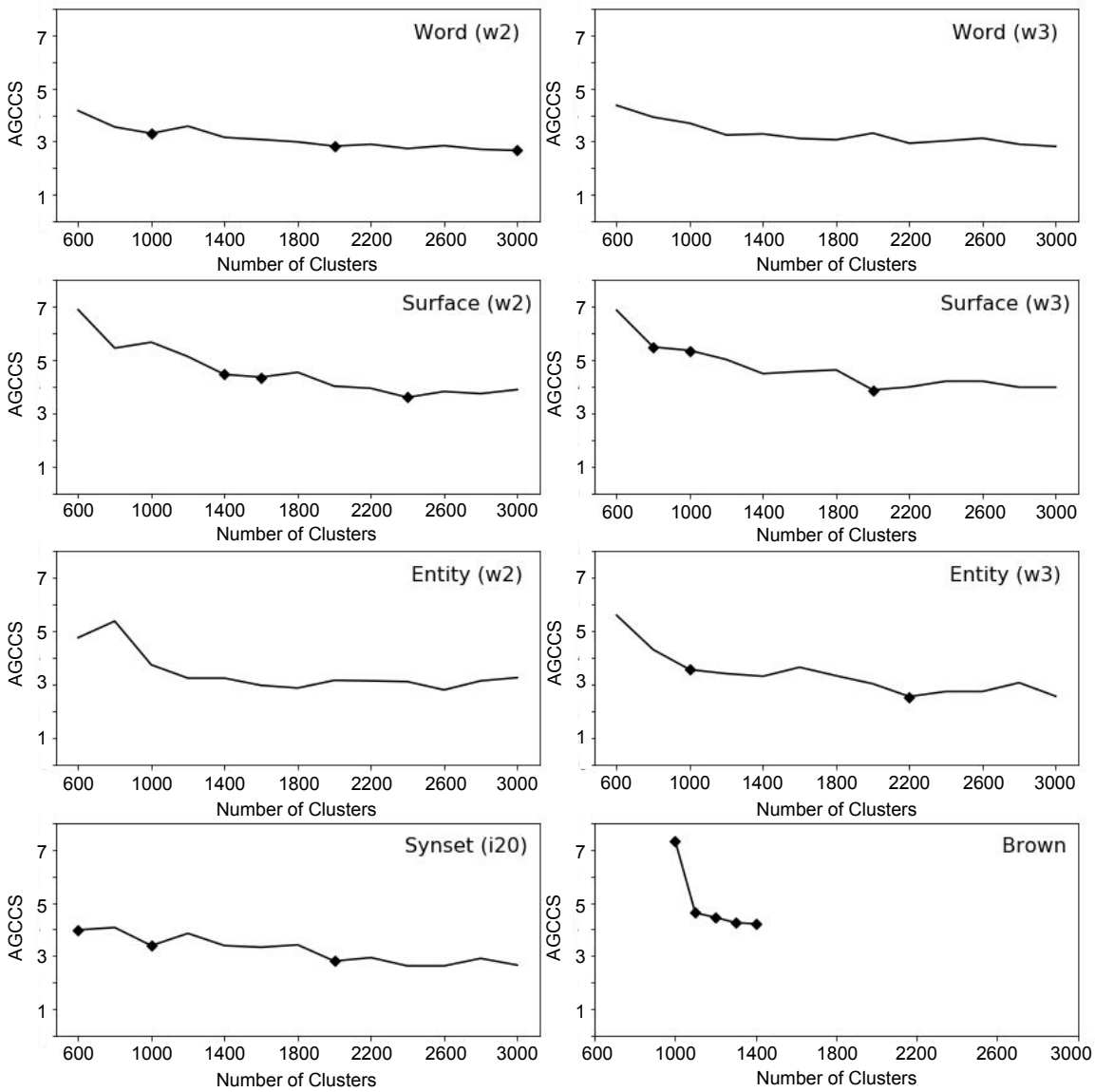

Figure 4.13. Change of Average Gold Candidate Cluster Size (AGCCS) as the Number of Clusters is Increased for each Clustering Approach.

typing models calculated[19] , hence 19 dots in Figure 4.13.

The third step is to pick the best clustering combination. Note that based on 19 calculated typing models, we ended up having 540 different combinations[20] . However, it is not feasible to train a disambiguation model for each combination and pick the

[19]We have three *Word* models for 1000, 2000, 3000 clusters with window=2; six *Surface* models for 1400, 1600 and 2400 clusters with window=2 and 800, 1000 and 2000 clusters with window=3; two *Entity* models for 1000 and 2200 clusters with window=3; three *Synset* models for 600, 1000, and 2000 clusters with iter=20; and five *Brown* models for 1000, 1100, 1200, 1300, and 1400 clusters

[20]540 = 3 *Word* x 6 *Surface* x 3 *Synset* x 6 *Brown* x 2 *Entity*.

best performing one. Instead, we proposed a heuristic that scores the typing model combinations according to Equation 4.1 . We omit the *Entity* typing model in order not to involve the second-stage ranking into this procedure. After getting the first-stage fixed, we pick the best performing *Entity* model which is obtained with 2200 clusters.

$$\begin{aligned} &\underset{w\in Word, s\in Surface, y\in Synset, b\in Brown}{\arg\min} \quad P^w + P^s + P^y + P^b \\ &\text{where} \quad P^t = \sum_{m\in M} \sum_{c_{ng}\in\{C^m - c_g\}} P^t_{c_{ng}} - P^t_{c_g}, \quad \text{if} \quad P^t_{c_{ng}} > P^t_{c_g} \end{aligned} \tag{4.1}$$

Note that a typing model t outputs the probabilities of the cluster-based types for a given *mention*. Since we know the pre-assigned cluster-based type of each candidate c_{type}, we also know $P(c_{type}|mention)$, or P^t_c in short. The ideal case from the disambiguation point of view is to have all typing models assign the highest probability to the gold candidate c_g so that it can be easily distinguishable from the rest of the candidates (i.e. $C^m - c_g$), which are called non-gold candidates, or c_{ng}. However, in real world scenario, typing models can make a classification mistake which leads to c_{ng} having higher probability than c_g. We call them competing non-gold candidates. Having higher typing model probability falsely favors them in the disambiguation step. Having said that, Equation 4.1 chooses the model combination such that it minimizes the aggregate probability difference between gold candidate and competing non-gold candidates calculated over all mentions M. For these calculations, we use the AIDA.train data set.

At the third step, instead of picking the typing model combination that has the lowest value according to Equation 4.1, we select the lowest scored 10 combinations and train a ranking model for each. Then, we score them on the AIDA.testa development set and pick the one that achieves the highest disambiguation score. The selected combination is *Word* model with 1000 clusters, *Synset* model with 1000 clusters, *Brown* model with 1300 clusters, and *Surface* model with 2400 clusters. For the *Entity* model

in the second-stage of the ranking, we use the model with 2200 clusters.

4.5. Experimental Results

4.5.1. Evaluation of the Candidate Generator

In order to evaluate the candidate generator, we use the gold recall measure, which is defined as the percentage of the annotated (gold) named entities in the data set that have been suggested by the candidate generator. Table 4.6 gives the gold recall values for our candidate generator and two other candidate generators in the literature. Note that most of the NED studies use a smaller number of candidates in order to discard the least possible cases before doing the ranking. Hence, *Ratinov* [135] and *Ganea* [68] use the highest scored 20 and 30 candidates per mention, respectively. In our experiments, we use top 100 ($N = 100$) candidates in order to increase the recall of the NED system and to have more negative examples during the training of the model. To evaluate our candidate generator, we also calculate gold recall for top 20 and 30 candidates. Moreover, in order to measure the contribution of the second stage of our candidate generation algorithm described in Section 4.3.1, we also calculate these recalls without applying the second stage. The results of the proposed candidate generator as well as the ones of [135] and [68] are given in Table 4.6. In addition to these, [18] reported a recall of 97.7 on AIDA.train when $N = 100$.

When we compare our recall values with other studies, in almost all cases the proposed candidate generator achieves better performance except with respect to the performance of [135] on the AQUAINT. The results also show that the second stage in the candidate generator produces a substantial improvement. It means that using candidates from similarly titled surrounding mentions and extending the candidates even further with their co-occurring entities from Wikipedia results in higher recall values. The only exception is with the AQUAINT and AQUAINT$^{\text{rev}}$.

When we perform error analysis, we observe that in case of KORE50, small context and mostly one word mentions result in low recall. In case of RSS-500, the

Table 4.6. Gold Recall Values for Candidate Generation on the NED Data Sets.

	Ours w/o 2nd Stage			**Ours w/ 2nd Stage**			**Ganea**	**Ratinov**
Data Set	N=100	N=30	N=20	N=100	N=30	N=20	N=30	N=20
AIDA.train	97.97	97.56	97.16	99.74	99.22	98.61	-	-
AIDA.testa	97.63	97.26	97.07	99.85	99.22	98.24	96.6	-
AIDA.testb	97.87	97.07	96.58	99.62	98.59	97.36	98.2	-
MSNBC	91.63	91.63	91.01	99.70	99.22	98.29	98.5	88.67
MSNBC^{rev}	91.91	91.91	91.30	99.85	99.08	98.47	-	-
AQUAINT	96.93	96.23	95.25	97.40	96.09	94.69	94.2	97.83
AQUAINT^{rev}	97.92	97.37	96.68	98.06	96.95	96.12	-	-
ACE2004	96.86	95.29	93.73	96.86	95.65	94.51	90.6	86.85
ACE2004^{rev}	96.88	95.70	94.14	96.88	96.09	94.92	-	-
KORE50	86.01	82.52	81.82	92.31	88.81	87.41	-	-
RSS-500	88.39	86.27	85.69	89.56	87.43	86.85	-	-
Reuters-128	88.41	87.60	87.28	95.65	90.82	89.21	-	-

annotated mentions only include part of the existing surface form (e.g., only the word "France" is annotated for the available surface form "Tour de France"). Since we do not take into account the immediate surrounding words of the annotated mentions during the search or use an off-the-shelf named entity recognizer [78] to expand the boundaries, we achieve relatively low recall on RSS-500. Table 4.7 reports the average number of candidates per mention generated for $N = 100$. It also shows the average length of the mentions in characters and the average edit distance between the surface form (SF_m) of the mention and the best matched surface form of the candidate (SF_c^{best}) per candidate. The highest values for both metrics are seen in Reuters-128, while the average mention length values on the AIDA sets also show their characteristic difference from the other sets, which is important considering that we train our disambiguation model on the AIDA.train and test it on the other sets.

4.5.2. Contribution of Specialized Word Embeddings in Mention Typing

Instead of using regular word embeddings (i.e. W_R) as input for our mention typing models, we introduced two different word embeddings in Section 4.2.5. Those

Table 4.7. Statistics on the Candidates Generated for each NED Data Set.

Data Set	Average Num. of Candids (N=100)	Average Mention Length	Average Edit Distance
AIDA.train	84.65	8.9	1.7
AIDA.testa	84.35	9.0	1.8
AIDA.testb	84.45	8.9	1.9
MSNBC	92.97	10.2	2.6
MSNBC^{rev}	94.22	9.9	2.5
AQUAINT	78.07	11.6	2.1
AQUAINT^{rev}	78.22	11.6	2.1
ACE2004	79.15	11.0	2.2
ACE2004^{rev}	79.17	11.0	2.2
KORE50	85.44	6.3	1.2
RSS-500	70.90	11.3	2.7
Reuters-128	72.79	12.6	3.8

are the cluster-centric word embeddings (W_{CC}) and surface form based word embeddings (W_{SF}). W_{CC} is proposed as an optimized version of W_R, as it is influenced by the applied clustering during the calculation. Both embeddings are obtained with word2vec[21] on all Wikipedia articles. They are used as an input to $Left$, $Right$ and $Surface_1$ components (either LSTM or CNN layers shown in Figures 4.9 and 4.10) of the typing model. In case of W_{SF}, it is obtained with word2vecf[22] from the large surface form data set as described in Section 4.2.5. Different from the previous two embeddings, W_{SF} is used as an input to the $Surface_2$ component of the typing model. Note that, like $Surface_1$ component, $Surface_2$ also takes the words of mention's surface form as input but in W_{SF} embeddings instead.

In order to measure the contribution of using W_{CC} over W_R and using additional W_{SF}, we trained both LSTM[23] - and CNN[24] -based models with different embedding combinations for each typing model. Note that each typing model is trained and tested

[21]We set window=2, size=300, and use the default values for the other parameters.

[22]We set size=300, and use the default parameter values for the other parameters.

[23]We set the learning rate to 0.1 and use the standard gradient descent optimizer with Nesterov momentum of 0.9. We set the weight decay rate to 1.2e-06 and clip the gradients at 2.0. We set the hidden state size to 600 for all LSTM layers. Wherever applied, dropout probability is set to 0.5. We use batch size of 200.

[24]We use the same parameter values as in the LSTM-based model, except the clip value is set to 1.0. The CNN filter sizes are set to 64.

on its own designated data set described in Section 4.4.1.1. To consider all combinations in a feasible time frame, the small version of those data sets (the ones named *-SmallTrain and *-SmallTest) are used. In Table 4.8, we measured how accurately the typing model predicts the cluster-based type labels assigned to each mention in the test sets. The results are given in F1 score.

Table 4.8. Showing the Contribution of the W_{CC} (over W_R) and W_{SF} Embeddings When Typing Models are Trained and Tested on *-SmallTrain and *-SmallTest Sets.

Arch.	Inputs				Mention Typing Models				
Type	**Left**	$Surface_1$	$Surface_2$	**Right**	***Word***	***Synset***	***Surface***	***Brown***	***Entity***
LSTM	W_R	W_R	*NotUsed*	W_R	67.5	66.6	66.1	68.6	67.1
LSTM	W_{CC}	W_{CC}	*NotUsed*	W_{CC}	+2.4	+1.9	+5.3	+4.4	+0.9
LSTM	W_R	W_R	W_{SF}	W_R	+4.6	+1.0	+6.4	+4.8	+2.2
LSTM	W_{CC}	W_{CC}	W_{SF}	W_{CC}	+5.3	+3.0	+7.0	+6.3	+2.5
CNN	W_R	W_R	*NotUsed*	W_R	69.1	61.1	58.3	60.3	61.4
CNN	W_{CC}	W_{CC}	W_{SF}	W_{CC}	+3.7	+7.5	+14.5	+10.6	+6.3

The first rows for the LSTM and CNN-based models in Table 4.8 consider the case where we do not use any special word embedding except the W_R. The following rows show how much the F1-scores change with respect to the first row (i.e., the baseline), first when we replace W_R with W_{CC}, then when we use W_{SF}. Shown at the last rows, using both special embeddings together increases the scores considerably, up to 3 to 7 points. Their contribution is even more visible in case of the CNN-based models. When we do the same analysis on the other NED sets, we also observe very similar contribution levels. Based on these results, we use W_{CC} and W_{SF} together in the rest of our experiments.

4.5.3. Cluster-based Mention Typing Results

In order to evaluate the cluster-based mention typing models, we train and test our five different models on the *-LargeTrain and *-LargeTest sets defined in Section 4.4.1.1. The results in F1-score are given in Table 4.9 for both the LSTM- and CNN-based models. However, since they are tested on different sets, the F1-scores are

not comparable[25] across the table. Hence, the average loss per instance is included in parenthesis. It is calculated by normalizing the total cross entropy loss value given by the model on the test set with the number of instances in that set. Note that average loss per instance is a better evaluation metric than F1-score for assessing the quality of the predictions for the disambiguation step, since, the prediction probabilities are used as features in the ranking model. F1-score only measures how accurate the model predicts the true label, whereas average loss per instance indirectly takes the model's probability of that predicted label into account.

Table 4.9. F1-scores and Average Loss per Instance Values for the Mention Typing Models Trained and Tested on the *-LargeTrain and *-LargeTest Sets.

Arch. Type	***Word***	***Synset***	***Surface***	***Brown***	***Entity***
LSTM	81.2 (1.10)	78.6 (1.52)	83.4 (0.80)	83.8 (0.68)	82.1 (0.74)
CNN	81.0 (1.09)	79.0 (2.14)	81.0 (0.83)	81.6 (0.75)	79.6 (1.43)

When the context is local as in the *Word* and *Synset* models, CNN performs similarly to LSTM. However, in general the LSTM-based models outperform the CNN-based models. This is supported by both F1-score and average loss per instance values. Hence, in the rest of our experiments, we use LSTM for all mention typing models.

When we compare the different models with each other through the average loss per instance value, the *Synset* model turns out to be the worst performer and the *Word* model comes after that. This means that these two sentence-based models are outperformed by the document-level typing models, namely *Surface*, *Brown*, and *Entity*. This might be expected due to the larger context at the document-level. In case of the worst performer, the synset-based model is based on clustering of the entity embeddings that have not originated from the context, but are based only on the similarity of the assigned synsets. In other words, the cluster-based type labels used for training the *Synset* model are not optimized for the contextual similarity. This might affect the success of the typing model which learns to predict based on the contextual similarities. The *Synset* model not being optimal for typing model is also supported by the results in Table 4.8. Specialized word embeddings do not help the

[25]In order to keep the results as much comparable as possible, we train all typing models based on the same number of clusters, which is 1000.

Synset model much compared to the other models.

4.5.4. Disambiguation Results

For the disambiguation step, we trained our disambiguation model[26] on the AIDA.train data set and did the feature selection based on the AIDA.testa development set. We train our ranking model in two-stages. At the first stage, the model is trained and run on all data sets. The ranking probabilities for the candidates are stored and used in the additional features, which are classified as the second stage features in Section 4.3.2. Our final results[27] are obtained after training the ranking model with all the features. The results are reported in Table 4.10 (for test sets with large context) and Table 4.11 (for test sets with small context).

Table 4.10. Results in F1 and BoT F1 on the NED Test Sets with Large Context.

System	MSN	MSN^{rev}	AQU	AQU^{rev}	ACE	ACE^{rev}	AIDAb
Phan *et al.* 2017	91.8	-	-	-	92.9	-	-
Ganea and Hofmann 2017	-	93.7	-	88.5	-	88.5	92.2
Guo and Barbosa 2016	-	92.0	-	87.0	-	88.0	-
Yamada *et al.* 2016	-	-	-	-	-	-	93.1
Phan *et al.* 2018	91.0	-	87.9	-	88.3	-	-
Sil *et al.* 2018	-	-	-	-	-	-	94.0
Le and Titov 2018	-	93.9	-	88.3	-	89.9	93.1
Radhakrishnan *et al.* 2018	-	-	-	-	-	-	93.0
Raiman and Raiman 2018	-	-	-	-	-	-	94.8
Fang *et al.* 2019	-	92.8	-	87.5	-	90.5	94.3
Liu *et al.* 2019	-	-	-	87.3	-	86.6	87.6
Cheng and Roth 2013 [BoT]	*90.0*	-	*90.0*	-	*86.0*	-	-
Yang et al. 2018 [BoT]	-	*92.6*	-	*90.5*	-	*89.2*	*95.9*
Ours							
w/o Typing Models (Baseline)	87.3	88.4	84.5	87.0	80.8	82.0	81.4
w/4 T.Models (1st Stage)	91.9	92.6	88.4	90.7	89.5	90.3	89.9
w/5 T.Models (2nd Stage)	**92.6**	93.0	**89.0**	**90.7**	90.0	**91.1**	93.2
w/5 T.Models (2nd Stage) [BoT]	*90.8*	*92.1*	*89.8*	*91.9*	*91.8*	*93.2*	*92.6*

[26]We set the learning rate to 0.05 and use the standard gradient descent optimizer with Nesterov momentum of 0.9. The first and the second layers contain 500 and 300 units, respectively. The dropout values after the first and the second layers are set to 0.1 and 0.7, respectively. The training data are given in batches of 400 instances.

[27]We train our system 20 times with different seed values.

The upper part of the tables provides the results reported by the previous studies in the literature. Note that the results from the two studies at the bottom are given in BoT F1-score. The lower part of the tables presents our results given in traditional micro F1-score and BoT F1-score. The first line in "Ours" part shows the results taken without using any features related to the mention typing models or the second stage. We call it our baseline. As we add the features calculated with the four typing models (namely *Word*, *Synset*, *Surface*, and *Brown*), we can observe increase of 2 to 8 points on all test sets. Next, we apply the second-stage, where we add the second stage features and the features calculated with the *Entity* typing model. The results are improved further, especially on AIDA.testb with 3.3 points increase.

In order to compare our results with the SOTA results in Table 4.10, we perform the randomization test with respect to the studies that achieved close to our results. On AQUAINT, we achieve better than Phan *et al.* (2018) at statistically significant level[28] . On MSNBC, our higher F1-score turned out to be not statistically significant compared to Phan *et al.* (2017). On the revised ACE2004, our higher result is not statistically significant[29] compared to Fang *et al.* (2019). On the revised AQUAINT, our results are 2.2 points higher than Ganea and Hoffman (2017) and 1.4 points higher in BoT F1 with respect to Yang *et al.* (2018). However, we are unable to perform valid randomization tests[30] with respect to their results.

In case of test sets with shorter context, Table 4.11 shows that our best system cannot achieve better performance than its counterparts. The worst performed test sets compared to the SOTA results are KORE50 and RSS-500. KORE50 has the lowest average number of words per sentence and RSS-500 has only one mention per document on average. Our context-centric approach cannot utilize such short context enough. This is in fact the most common problem for all NED systems. On the other hand,

[28]We compared our 20 runs with the one run of them that produced the same reported F_1-score and observed p-value=0.014 on average.

[29]Table 4.10 includes the adjusted F1-score as they did not output prediction for 5 mentions of 256 that we have. Their reported F1-score of 91.2 is on 251 mentions, on which we achieve an F1-score of 91.7.

[30]Ganea and Hoffman (2017) provided the output of one run. The F1-score calculated on that run is 91.1, while the score reported in their study as the average of five runs is 88.5 $\pm$ 0.4. The provided output might be from their best run, while the score from our best run is 91.5.

Table 4.11. Results in F1 and BoT F1 on the NED Test Sets with Short Context.

System	KORE50	RSS-500	Reuters-128
Phan *et al.* 2017	79.4	80.4	91.8
Phan *et al.* 2018	78.7	82.3	85.9
Ours			
w/o Typing Models (Baseline)	40.4	68.9	72.0
w/4 T.Models (1st Stage)	56.1	74.4	76.5
w/5 T.Models (2nd Stage)	57.5	76.1	79.3
w/5 T.Models (2nd Stage) [BoT]	*58.5*	*77.7*	*88.6*

Reuters-128 has a relatively larger context based on those metrics, however its average number of edit distance per candidate is the highest with respect to the other sets. That may cause the lower scores on Reuters-128. This is actually connected to the fact that AIDA.train on which we train our disambiguation model is characteristically different from these three data sets. One particular difference to mention is that the average edit distances given in Table 4.6 in the AIDA sets are lower than many of the other sets. Whenever our NED model gets candidates with relatively higher edit distances on any of the test sets, it assigns low scores to such candidates as expected. This affects the gold candidates disproportionately, when there are alternative candidates with lower edit distances.

It is interesting to note that the AQUAINT test set contains many concepts (*e.g.,* "power plant", "radioactive waste" etc.) along with named entities. Our success might be related to the fact that we train our typing models on Wikipedia, which also includes the mentions of concepts.

4.5.5. Analysis of the Experiments

4.5.5.1. Ablation Study on the Ranking Features. In order to understand the contribution of the ranking features, we perform two ablation tests. Moreover, since we do the ranking in two stages, we calculate the contributions for both stages as well as for the baseline, which is the system that doesn't use any features obtained from the typing models. Before getting into the analysis, one factor has to be noted while evaluating

the contributions. The second stage uses additional "second stage" features and their success depends on the success of the features used in the first stage. Since those first stage features are still used in the second stage, their contribution drops as they share their success with the second stage features. Hence, it is more appropriate to evaluate the contribution of the first stage features based on the results at the first stage.

The first test examines the contribution of each feature by excluding that feature from the model. Table 4.12 lists all the features used in our experiments. Note that some of them are not available (N/A) for certain stages. The results[31] are shown in the F1-scores taken on the AIDA.testa (dev) set. As each feature is excluded, the drop in the F1 score is given. The first thing to notice is the high-level contribution of edit distance based features. Surface form is the major factor at the disambiguation task. The second thing is the decreasing contributions towards the second stage.

Table 4.12. Showing the Contribution of each Feature in F1 Scores by its Exclusion at Different Stages on the AIDA.testa Development Set.

System	Baseline	1^{st} Stage	2^{nd} Stage
All Features Included	82.1	93.3	94.4
Local Features:			
- *surface-form-edit-distance* (1)	-9.7	-4.0	-1.3
- *entity-log-freq* (2)	-3.0	-1.1	-0.1
- *surface-from-type-in-binary* (3)	-2.4	-1.6	-0.6
- *entity-type-in-binary* (4)	-1.8	-0.9	-0.4
- *id^t-typing-prob* (5)	N/A	-0.2	-0.2
Mention-level Features:			
- *max-diff-surface-form-log-freq* (6)	-4.2	-0.8	-0.4
- *max-diff-doc-similarity* (7)	-4.1	-0.6	-0.2
- *avg-surface-form-edit-distance* (8)	-0.6	-0.6	-0.3
- *max-diff-id^t-typing-prob* (9)	N/A	-0.5	-0.3
Document-level Features:			
- *max-id^t-typing-prob-in-doc* (10)	N/A	-0.7	-0.2
Second-stage Features:			
- *max-ranking-score* (11)	N/A	N/A	-0.5
- *max-cos-sim-in-context* (12)	N/A	N/A	-0.8

Even though there are four groups of features shown in Table 4.12, for the sake of contribution analysis, it is better to group them as surface form based (numbered

[31] We train each model 10 time with different seed values and report the average results.

1,3,6,8), candidate based (2,7), typing model based (5,9,10), and ranking based (11,12) features. Each group considers the disambiguation task from different aspects.

Surface form based features are mention oriented, completely independent from the context. The results in Table 4.12 show that they keep their contribution high at all stages. This can be attributed to the fact that other features do not consider surface form. In other words, there is no contribution overlap between surface form based features and other features.

Typing model features are mostly context driven. Their contribution in Table 4.12 does not look notable at the individual level. However, the change between the Baseline and the 1^{st} stage, which is 11.2, comes from adding the typing model features. The low individual contributions can be explained by the fact that those three features are derived from the same value hence, their contributions overlap. Interestingly, using the highest typing model probability seen in other occurrences of the same candidate in the same document (10) is more powerful than using the typing model probability of the candidate (5) for the considered mention.

Candidate based features are more candidate specific, with no direct involvement of the surface form. Their contribution drops substantially at the $2^n d$ stage. Even though (2) is widely used in the NED literature, it is calculated offline and independent of the mention or its context. This might explain its low performance. In case of (7), it is based on doc2vec embedding similarity between candidate's Wikipedia page and the test document. In other words, it is similar to the other typing model features. The results support that their contributions overlap.

Ranking based features help us take into account the surrounding candidates. The results show that they contribute quite well compared to others. Especially (12) might be the main driver behind the F1 score improvement of the 2^{nd} stage.

In our further analysis, we observe that using *max-diff* version of the features performs better than using the feature itself. *max-diff* compares the value of the feature

with respect to the highest value seen among its sibling candidates. For example, using only *doc-similarity* does not contribute better than *max-diff-doc-similarity*. The same is true for *max-diff-surface-form-log-freq*. We argue that this is related to the fact that *doc-similarity* and *surface-form-log-freq* are context independent and applying *max-diff* makes them context dependent.

Table 4.13. Showing the Contribution of each Mention Typing Model in F1 Scores by its Exclusion at Different Stages on the AIDA.testa Development Set.

System	1^{st} Stage	2^{nd} Stage
All Mention Typing Model Features Included	93.3	94.4
- *Word*	-1.1	-0.6
- *Synset*	-1.4	-0.7
- *Brown*	-0.6	-0.2
- *Surface*	-0.6	-0.3
- *Entity*	N/A	-0.2

Our second ablation analysis evaluates the contribution of each typing model. Table 4.13 shows the amount of drop in F1 score when we exclude each typing model from the 1^{st} and 2^{nd} stages on the AIDA.testa (dev) set. Note that features (5,9,10) shown in Table 4.12 have multiple values, one for each used typing model t. Considering that the *Synset* typing model is the worst one at predicting the types as shown in Table 4.9, it is surprising to see that it is the best contributing model. Like *Synset*, the other local-context based model, namely the *Word* typing model comes second. This indicates that typing models based on local (sentence-based) context contribute to the success of NED more than typing models based on global (document-based) context.

4.5.5.2. Error Analysis of the Ranking Model. In order to understand where our ranking model fails, we analyzed the failed cases on the AIDA.testa (dev) set in detail. Our first analysis involves measuring the role of the popularity of a candidate and the frequency of its surface form. It is intuitive to assume that popular entities are more likely to be mentioned than less popular entities. In our case, we define the popularity of a named entity as the total number of times that named entity is seen in the Surface Form Data Set. Similarly, when one surface form is seen more times with a certain entity than others, it is more likely that that surface form refers to that entity when-

ever it is used. When one surface form may refer to many entities, it causes ambiguity, which is the main source of the failures for the ranking model.

In Table 4.14, we calculate certain ratios and percentages based on the popularity of the candidate (F^e) and the frequency of its surface form (F^s). During these calculations, G is the set of the gold (true) candidates, one for each mention in the data set. G' is its subset that only includes the ones (i.e. failed golds) that are failed to be predicted correctly by our model. P' is the set of wrong predictions that are suggested instead of gold candidates. Note that X is a variable in shown calculations and it should be replaced with either F^e or F^s, when appropriate. Moreover, X_{max} refers to the maximum value among all non-gold candidates of the same mention for the selected X.

Table 4.14. Analyzing the impact of the popularity of the candidate entity and its surface form frequency when our system fails on the AIDA.testa (dev) set.

Case	**Description**	**Calculation**	$X = F^e$	$X = F^s$
[1]	Ratio of failed golds' X to all golds' X	$avg\sum_{g' \in G'} X_{g'} / avg\sum_{g \in G} X_g$	0.17	0.05
[2]	Ratio of predictions' X to failed golds' X	$avg\sum_{p' \in P'} X_{p'} / avg\sum_{g' \in G'} X_{g'}$	2.78	10.69
[3]	% of cases when gold's X is higher than the max's X	$\sum_{g \in G}[X_g > X_{max}]/size(G)$	0.36	0.48
[4]	% of cases when failed gold's X is higher than the max's X	$\sum_{g' \in G'}[X_{g'} > X_{max}]/size(G')$	0.29	0.16

Case [1] in Table 4.14 looks at the ratio of failed golds' popularity and form frequency to the all golds' in terms of averages. The values 0.17 for F^e means that the average popularity of failed golds is about five times lower than the average popularity of the golds. In case of F^s, we can say that when our system fails, the form frequency of the gold candidate is 20 times less frequent than the average gold candidate's surface form.

In Case [2], we compare the values of wrongly predicted candidates and the corresponding gold candidates. For F^e, it shows that the popularity of the wrongly predicted candidate is 2.78 times higher than the popularity of the actual gold candidate. This is 10.69 times in case of F^s, which means that the form frequency plays more role than

the popularity.

Case [3] measures the percentage of the cases when gold's popularity or form frequency is higher than any other sibling candidate. The nominator of [3] in Table 4.14 counts for how many g X_g is higher than X_{max}. The denominator normalizes that value to get the percentage. Case [4] measures the same value for the failed golds. In case of F^e, the value 0.36 means that 36% of the time the golds are the most popular candidate and it is 29% for the failed golds. In other words, most of the time golds are not the most popular ones among the candidates[32] and that ratio does not change much among the failed golds. In case of F^s, half of the time gold candidates are the ones that the surface form refers to most. That ratio drops considerably in case of failed golds. All in all, our system pays more attention to the form frequency than the popularity and prefers the candidates that the surface form refers to most. This also aligns with Case [2].

In addition to this analysis, we also measure the role of edit distance at failures. There are 15 gold candidates out of 4791 golds that have non-zero edit distance value in the AIDA.testa set. Meaning that, none of their surface form in our database matches with the surface form of the mention exactly. Our system fails 10 of those 15 cases. Compared to a total of 282 failures, it is less than 4% of errors. However, in case of test sets like Reuters-128, it makes the difference.

Another aspect we analyze is how accurate our system predicts when there are multiple mentions of the same entity. Out of 4791 mentions, 2863 of them involve the multi-mention cases. Our system fails at 153 of those cases, which does not look significant. Yet, it also means that more than half of the total failures of our system involve such cases. Moreover, in 60% of 153 cases, our system predicts none of the instances of the multiple-times mentioned entity correctly.

[32]This might be the reason why the *entity-log-freq* feature does not have a notable impact on the disambiguation results.

4.6. Discussion and Future Work

In this research, we cluster named entities with two clustering algorithms: Brown clustering algorithm and primarily K-means algorithm. Since we have over five million named entities to cluster, we did not focus on picking the best clustering approach due to computation time overhead. Instead, we choose K-means specifically for its wide-acceptance in the literature and more importantly its acceptable linear time complexity $O(Kn)$, where K is the number of clusters and n is the number of items to be clustered. In fact, we experience the burden of quadratic time complexity in case of the Brown clustering algorithm and the highest number of clusters we were able to run with Brown was 1400, as shown in Figure 4.13. In case of other clustering algorithms, one possible candidate might be hierarchical clustering, or more specifically top-down hierarchical clustering. While bottom-up approach can go as far as $O(n^2 logn)$, top-down approach's time complexity is comparable to K-means. Moreover, hierarchical approaches do not require pre-defined number of clusters. Nevertheless, in order to determine that number, we still need to look at the dendrogram created during the hierarchical clustering procedure and find the optimum point to cut. Since we have over five million data points to cluster, that task might be as difficult as finding the optimum number of clusters for K-means. Another alternative is the K-medoid algorithm, which is closely related to K-means. The motivation to use K-medoid might be to represent the centroids of the clusters in terms of the existing named entities in the given input, rather than the average of the entities in a cluster. Considering that named entities are unique concepts that are denoted with proper names, it is in general unlikely that one of them can act as medoid (center) and others are grouped around it. This can also be explained as follows: since there are over five million data points to cluster, it might be the case that items are not grouped together in spherical shapes. They might be scattered more arbitrarily. However, the K-medoid clustering algorithm is known to have a disadvantage in such situations, as it relies on minimizing the distances between the medoid and non-medoid items. Furthermore, the time complexity of K-medoid is $O(n^2)$, which also makes it less ideal compared to K-means.

There are a number of areas that can be addressed as future work to improve the proposed system. One of the disadvantages of our mention typing model is its dependency on context. The shorter it is, the worse it performs. One solution is to utilize the existing context as much as possible with techniques like attention. Similarly, the latest advances in embeddings such as BERT [185] can be another alternative as their context-customized embeddings help better represent the context. Another problem is the fact that our weakest typing model is the *Synset*-based model. It can be argued that lack of contextual information in the *Synset*-based types leads to a low performing typing model. Despite its high performance at the disambiguation step, finding a solution to this drawback might improve our weakest model and lead to even better results. Part of the problem might be related to insufficient context while learning the entity embeddings with word2vecf before obtaining the synset-based clustering of named entities. Compared to regular context which may include hundreds or thousand words, synset-based context can be very small like three or five synsets depending on how many synsets are associated with the named entity in the YAGO and BaseKB data sets. One way to expand it is to incorporate the synset hierarchy to extend the assigned synsets because YAGO synsets are based on the WordNet synsets which are connected with IsA relations forming a taxonomy hierarchy. For example, for each synset associated with a named entity in YAGO, we can also include its hypernym (i.e. parent) synset as another context item for that named entity while calculating entity embeddings with word2vecf. Moreover, we can even consider each synset entry as a named entity and use its hypernym as its context. This way, word2vecf may learn the embedding of the synset based on its hypernyms and that eventually may affect the embedding calculations of the related named entities.

Clustering of named entities is another improvement area. For example, there are other clustering methods like Expectation Maximization clustering, which provides soft clustering. In our experiments, we use K-means which does hard-clustering. It would be interesting to convert our mention typing model to make use of a soft clustering approach. For example, instead of doing classification, we may compare the output vector of the mention typing model with the soft-clustering of the mentioned named entity. Such vector comparison might hold more detail for loss generation which might

lead to better training of the model. In our study, we obtain each clustering of the named entity space independently from the others and pick the best combination for the disambiguation step in Section 4.4.3. We face the dilemma of using larger cluster numbers to improve their discriminative power for the disambiguation step versus being less accurate at predicting the cluster-based types. These two contradicting criteria may lead to a non-optimal solution. Nevertheless, we still achieve SOTA results. However, a more optimized clustering approach (such as customized K-means) that considers these two criteria together can produce better clusters. Customized clustering can even be designed to minimize the overlap between different types of clusterings. Alternatively, feedback from the disambiguation step can be circled back to the clustering step for iterative revision of clusters. One more ideal solution to consider is to design an end-to-end system that does the mention typing and named entity disambiguation together. However, that might be difficult to design. In the ranking step, the most prominent features are edit distance and popularity related features. The loss values of the ranking model more likely originate from those types of features. Considering that we would like to push the loss values from the ranking model back to the mention typing model, this may not result in any significant improvement. Moreover, the training data set for the named entity disambiguation task is relatively very small. This also means that the mention typing model may not learn much from that. One possible end-to-end approach is to do the clustering and mention typing modeling together. This means that clustering opseeration must be differentiable so that it generates loss value to make the model learn. There are studies in the literature [186] that adapt clustering algorithms like K-means into neural network architecture by applying the perspective of differentiable programming. By combining such a network with the mention typing model network, we can both learn the mention typing and clustering together.

4.7. Conclusion

In the second part of the thesis, we introduce a cluster-based mention typing approach. We cluster named entities based on their contextual embeddings and assign those cluster ids as type labels to entities. Our analysis shows that using window as

short as two to calculate entity embeddings with word2vec turns out to give better clusterings for the disambiguation task. We calculate five different clusterings of over five million named entities, each considering different contextual aspect. Based on these, we train five different mention typing models. The results show that LSTM-based models achieve better results than CNN-based models. Moreover, the models that use document-level context predict the cluster-based types better than the models with sentence-level context. We also introduce two specialized word embeddings that are influenced by the presence of cluster information during their calculation. Their analysis shows that such influence helps the typing model better predict the cluster-based types.

The second contribution of our study is to use the predictions of the mention typing model as features for the disambiguation of named entities. Our analysis shows that each typing model improves the disambiguation performance. However, using one typing model is not enough to achieve state-of-the-art (SOTA) results. As we use five of the models together, our system achieves better or comparable results based on randomization tests with respect to the state-of-the-art levels on four defacto test sets. Considering that our ranking model is a simple two-layer feed forward neural network, we score each candidate individually in a binary classification-based approach rather than employing a collective disambiguation approach. Moreover, we use the top 100 candidates rather than the top 20 and 30 as in previous works. Achieving SOTA results indicates the potential of using mention typing models. Our further analysis shows that even though the typing models with sentence-level context obtain lower scores at predicting the types, they are the most contributing models compared to document-level models at the disambiguation step.

Additionally, we study the candidate generation step. Our analysis shows that using candidates from similarly titled surrounding mentions and extending the candidates even further with their co-occurring entities from Wikipedia gives higher recall values.

5. Tools

We compiled our studies on two different tasks into two publicly available tools. Those are hashtag segmentor and named entity disambiguator. In addition to those two, we also developed an experiment result management platform which is called xDB. In the following sections, we describe these three tools in detail.

5.1. Hashtag Segmentor

We implemented a feature-based hashtag segmentor. This means that the raw input is converted into a set of features on which the model can train and eventually be able to do segmentation predictions. Each feature is expected to represent one aspect of the input that gives a clue about whether there is a boundary in considered position in given input character sequence as described in Section 3.2.1.

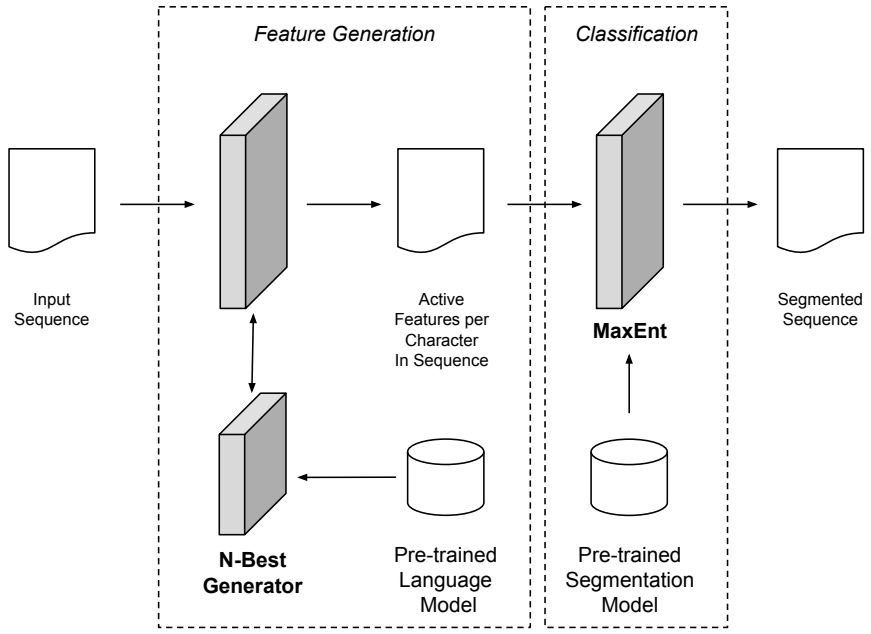

Figure 5.1. The Flow of the Process in the Hashtag Segmentor.

Figure 5.1 depicts the flow of the process in the segmentor. It can be broken down to two major steps. The first step involves reading the input and converting it to set of features. While the input is raw and unreadable for the machine, its transformation into feature space makes it representable and thus learnable by the machine. After

feature generation, at the second step, it gives these features to the off-the-shelf tool, namely maxent. This tool acts in two ways depending on the purpose. If you run the segmentor in order to segment a character sequence (or a set of sequences in a file), it makes the maxent tool use those generated features at calculating the probability of whether there is word boundary at each character in a given sequence. This is done based on pre-trained segmentation model. After getting boundary probability for each character, if the probability of having a boundary at position i is higher than not having a boundary, then the segmentor get to the conclusion that there is a boundary at position i. After making this decision for each position in the input, it outputs its final segmented version.

On the other hand, instead of segmenting any input, if you want to train your own segmentation model, then you can give a already segmented input to the segmentor and it generates those features as before at the first step and then make the maxent tool use those features at training a new model. Since segmented input already includes the word boundary positions, it can learn which features are more likely indicators of the word boundary. During the training, the maxent tool goes several iterations over the input and then saves the model into a file which can be used later to do segmenting.

5.1.1. Requirements

The segmentor was implemented in Python 2.7. Hence, the first requirement is to have Python 2.7 installed on your system. The second requirement is the *maxent* tool which is used as a classifier. This tool was mainly written in C++ and you should be able to compile it on your system by installing appropriate compiler. Note that you may use another compatible classifier on the Internet as long as it accepts the input and outputs the results in the same format. Check the format details in the manual of the *maxent* tool. Note that one of the components of the hashtag segmentor toolkit is the N-best generator, which provides information for LM-based features described in Section 3.2.2.1. N-best generator has its own set of requirements which are provided in Section 5.1.6.1. However, if you choose not to use LM-based features, you do not need to run N-best generator.

5.1.2. Files and Folders

In this section, we explain the files and folder hierarchy of the hashtag segmentor toolkit. When you download the distributed version of this toolkit and you unzip that compressed file and following files and folders come out of it. The main two scripts at the top folder are *runHashtagSegmentor.py* and *runNBestGenerator.py* scripts, both written in Python. All the functionality of the toolkit can be reached by executing these scripts with the appropriate arguments. For available arguments for these two scripts, check the Sections 5.1.3 and 5.1.6.2, respectively.

5.1.2.1. "data" Folder. This folder keeps the data files that are used by the segmentor while segmenting given input or training a new model. Following describes the function of each file.

- **default.vocab**: It is the default vocabulary file that is used in our experiments. It contains list of words along with their log frequency. Check the Section 5.1.5 for more detail in vocabulary.
- **recommended.features**: It contains a list of features that are recommended based on our studies with this segmentor on various data sets. This file is used as a default parameter setting when the segmentor is used to train a new model.
- **cmu.word.classes**: This file contains a list of word and word class pairs from a data set called Twitter Word Clusters [122]. In this tool, certain features use these classes instead of words themselves. This effectively reduces the number of unique features and makes the learning more feasible.
- **default.bigrams**: It contains bigram statistics extracted from a large set of text. It basically includes a frequency value of seeing a word followed by another word. Hence data is given in three columns.
- **default.stop.words**: It keeps a list of stop words. These are small set of words (e.g. "the") that have no specific semantic meaning but are used more for functional purposes. Stop words are generally ignored during language processing. We also use them to ignore in specific cases.

- **default.short.stop.words**: This file keeps a list of stop words that has a length up to four characters. Certain features use this list to ignore certain cases.
- **default.short.ngrams**: Short words are very problematic cases for the hashtag segmentor. There can be many words in the vocabulary that include the word "to" or "on" in itself. Word-based features and NGram-based features are directly affected by such words because they create too many false-positive clues. When two short word comes one after another, it creates a extra problematic case. In order to take into account these cases, we collected a set of most frequent bigrams where both words are short, that is no longer than four characters.
- **default.special.words**: This is a special set of words that contain at least one capitalized character inside of them. For example, YouTube, iPhone and McDonald. Even though given examples are known cases and hence can be segmented correctly, this file enables the system to be aware of such special words so that it takes those cases into account. There are special features that look for these special words in the cursor position.

5.1.2.2. "src" Folder. This folder keeps the source files of the hashtag segmentor. Following describes the function of each file.

- **Modeler.py**: It is the base class for the classifier class. It also keeps the functions to measure the accuracy of the classifier if a test file is given with the *test_file* parameter.
  - **MaxEntModeler.py**: It derives from the Modeler.py and is responsible for calling the "maxent" tool based on given parameters. In the parameter file, you can specify further settings such as *iteration_count=[integer]*, *cutoff=[integer]*, *guassian_prior=[float]*, *gis=[0—1]* parameters. These values are passed directly to the *maxent* tool at the command line.
- **FeatureGenerator.py**: It is the base class for different types of feature generators. It keeps track of which features are set as active ones and calls corresponding functions to look for those features for considered cursor position in the input.
  - **WordFeatureGeneratory.py**: It is special type of feature generator class

that looks for basic word-based features around the currently considered cursor position. For example, one word-based feature looks at the longest word starting at the cursor position. Note that valid words are given in the vocabulary, which is *files/default.vocab* by default.

  - **NGramFeatureGenerator.py**: This feature generator class is specialized in the features that look for the existence of consecutive words (i.e. n-grams). For example, one particular such feature checks if there a word that ended before the cursor position and another word that starts at the cursor position. Existence of such case might give a clue about whether there is a word boundary at that considered cursor position.
  - **OrthographicFeatureGenerator.py**: It is another type of feature generator that defines features related to orthographic shape rather than words. To give an example, if there is a capitalized character at the cursor position and lower-cased characters before and after that capitalized character, it is highly likely that there is a word boundary at that position. Such features look for the orthographic shape of cursor's surroundings.
  - **ContextFeatureGenerator.py**: Context-based features are defined in this generator. These features look for clues collected from the context information, which is given in a separate file. For example in case of segmenting hashtag, the tweet that contains that hashtag is the context for that hashtag. There can be many tweets given as context for the same hashtag. Check Section 5.1.4 for more information on how to set the context file.
  - **NBestFeatureGenerator.py**: This generator holds the functions that define language model-based features. These features use the output of the N-Best Generator which calculates the most likely N segmentations of the input based on pre-trained language model. For example, one such feature looks at how many of those N segmentations include a word boundary at the currently considered cursor position.

- **Vocabulary.py**: It holds the functions that deal with the vocabulary, such as loading and filtering based on given parameters.
- **Generator.py**: It keeps the instances of all feature generators and coordinate the feature generation. It is responsible for loading the input sequence one-

by-one and calling each special feature generator's *createSequenceData* function with that sequence so that they can create all required data to be used at the feature generation. After that, for each character in the sequence it first calls *createPositionData* function to create position-specific data and then calls the function of each activated feature with the created sequence and position-specific data. If the feature is active at that position, the function generates a value for the feature, which is specific to the cursor position. This value represents the situation at that position.

5.1.2.3. "segmentation_models" Folder. This folder keeps a folder for each saved model. The hashtag segmentor toolkit comes with four pre-trained model. Each model folder holds the model file itself and the parameter settings file which holds the parameters used to create that model. The model file is generated by the *maxent* toolkit. Following is the explanation of each pre-trained model in this toolkit.

- **default.model**: This folder keeps the default model that is trained based on the basic set of recommended features. These exclude context-based and language model-based features as they require extra set of inputs that a regular user might not be able to run right after they install this toolkit.
- **default.model.wLM**: This folder keeps a model that considers LM-based features on top of the default model. It requires you to be able to run nbest-generator as it needs to create N-best segmentation for each input.
- **default.model.wContext**: This folder holds a model that take into account the context-based features in addition to the default recommended ones. In order to segment any input with this model, you should also be able to provide context data. The details are described in Section 5.1.4.
- **default.model.wLM_and_Context**: This is another default model that considers all types of recommended features.

Note that in order to segment any input with these models, you need to give the name of the model (i.e. its folder name) as input with the *--model* parameter of

the *runHashtagSegmentor.py* script. Moreover, when you instruct the *runHashtagSegmentor.py* script to train a new model, it saves that model with the name given with *--output_model* parameter under the *segmentation_models* folder. Check the following sections for more details on this.

5.1.2.4. "language_models" Folder. This folder holds the pre-trained language models to be used by the N-best generator. The original distribution of the toolkit does not contain any pre-trained language model due to their large size. However, it can be downloaded at our project website [28]. For example, after downloading the zip file named "default.lm.targ.gz", unzip its content into this folder which should be a folder named "default.lm", so that it can be recognized by the N-best generator automatically.

5.1.2.5. "nbest_generator" Folder. This folder contains the source files for the *nbest_generator* toolkit that comes with the hashtag segmentor toolkit. Its content is described in Section 5.1.6.3.

5.1.3. Command Line

runHashtagSegmentor.py script can be used for two main purposes. The first one is to break the given input into original words and the second one is to train a new segmentation model.

- *--input_text*: If you want to segment a specific character sequence for the test purposes, you enter that sequence following this parameter.
- *--input_file*: If you have a set of character sequences to segment in a file, you can use this parameter to enter that file's position. Input file contains one character sequence per row.
- *--output_file*: If you give a file to segment with *input_file* parameter, the results can be outputed in two ways. If you do not set *output_file*, then it automatically generates a output file by appending ".segmented" to the end of the input file name. However, if you want to have the output in specific file, then you can set

that file name with *output_file* parameter.

- *--model*: If you want to run the segmentor to segment any input, you need to specify the model which is used at predicting the word boundaries. If not defined, it uses the default model which comes with the distribution.
- *--training_file*: In order to train a new segmentation model, you need to have a training file which includes a set of segmented character sequences. This parameter is used to define the position of that file. Note that when this parameter is set, the segmentor goes into the training mode.
- *--output_model*: At the training time, you need to name the model so that all data related to newly calculated model is saved under the "models" folder with that name. If you want to use that model for the segmentation later on, then you use that name with the *model* parameter.
- *--test_file*: In order to measure the accuracy of the segmentor, you can give a test file which includes a set of segmented character sequences. Based on used model, the segmentor ignores the already given word boundaries and predicts them from scratch. Following that, it compares those predictions with respect to given true cases. It measures the results in the Accuracy and F_1-score (including precision and recall) metrics.
- *--output_details*: If you like to see the generated features for each character in given input, you need to set this parameter to 1. If you are segmenting a single sequence with the *input_text* parameter, then it outputs the details onto the screen directly. In case of segmenting a file with the *input_file* parameter, then it automatically generates an extra output file with ".features" extension. Note that the output file may vary depending on whether you use *output_file* parameter or not.
- *--parameters*: The segmentor supports many features as listed in Appendix A. When you train a new model, it uses the recommended subset of those features by default. However, if you like to select your own subset, you can list them row-by-row in a file and use this parameter to specify that file. Note that, copy of this file is saved under the generated model folder.

To give an example, for the test purposes, if you like to segment a character sequence such as “greatmovie” by using default model, you can run the following command line:

```
python runHashtagSegmentor.py --input_text greatmovie
```

When you run the command above, it outputs the segmented version of the input on the screen. Moreover, if you like to see the generated features per each character, you can also add *--output_details 1* to the command line. In this case, the output on the screen will include 10 more lines, each listing a set of active features for corresponding character in the input “greatmovie”. Next, if you have a set of character sequences in a file named “my.hashtags” which includes one sequence per row and want the segmentor to segment them and put the results in a specific file called “my.segmented.hashtags”, here is how to do that:

```
python runHashtagSegmentor.py --input_file my.hashtags
 --output_file my.segmented.hashtags
```

After executing this command, the output file includes a segmented version of each line from the input file. Again you can extend this command with *--output_details 1* in order to reach generated features. In this case, an extra output file named “my.segmented.hashtags.details” will be generated. The format of this file is a set of rows separated by the blank row. Each row set corresponds to one character sequence in the input file and each row lists active features per character in that sequence.

If you have your own training data in a file named “my.training.data” and like to train your own model and name it “my.model”, then in order to train a following command should do that:

```
python runHashtagSegmentor.py --training_file my.training.file
 --model my.model
```

When you run this command, it first generates all active features per training instance and then call *maxent* tool to train a model. The resulting model file is saved under "models/my.model" folder along with the parameter file that keeps the record of the model parameters (i.e. features). The command above uses the default recommended features[33] in order to generate active features per training case. However, if you like to specify your own set of features, then you list them in a file, say "my.features" and add *--parameters my.features* to the end of the command line above.

5.1.4. Context File for Context-based Features

In order to be able to use context-based features, you need to specify context information for every training instance in training file. Note also that this only works when training file contains list of segmented hashtags, that is not any raw text. Then context file should include list of tweets separated by the tab character for each hashtag in the training file. These tweets should also include that hashtag inside them, which is why they provide context for that hashtag. Name of the context file should be given by appending ".context" extension to the name of the training file. For example, if you have an input file named "my.input.file", then its corresponding context file should be named as "my.input.file.context". Similarly, if you want to train a new model with context-based support, then you need to provide the context file corresponding to that training file. Again, its name should start with the name of the training file, followed by the ".context" extension. This is also the case when you define the test file with *test_file* parameter.

[33] You can see those recommended features in the *files/recommended.features* file.

5.1.5. Vocabulary

The hashtag segmentor toolkit comes with its own vocabulary. However, if you are interested in building your own vocabulary for further training purposes, you should note that there are two types of vocabulary format that the segmentor accepts. First option is a simple list of words in a single column. However, many features of the segmentor rely on the word frequency information. Hence, the second type of vocabulary allows you to specify that frequency information. In that case, the vocabulary again the frequency and word itself in two columns, each column separated by the tab character. You can also specify the frequency in the negative logarithmic scale and use *logfreq2int=yes* parameter to convert it back to integer scale inside the tool. Plus, you can also use *minneglogfreq* parameter to specify minimum negative log frequency value for filtering undesired words without having to change the vocabulary. Moreover, you can also specify minimum word length with *minwordlen* parameter for further filtering.

5.1.6. N-best Generator

N-best Generator consists of two scripts:

- **nbest_generator**: It is the main executable file that is responsible for generating the highest scored N segmented versions (i.e. n-best segmentations) of given input character sequence. It is written in C++ and compiled with OpenFST library support so that it can read a LM encoded in OpenFST format.
- **runNBestGenerator.py**: It is a wrapper script written in Python. It is used to prepare the raw input to the format that is acceptable for the *nbest_generator*. It can also be used to train a new LM, which is required by *nbest_generator*.

5.1.6.1. Requirements. Our Hashtag Segmentor toolkit comes with a dummy LM that is generated from a small text file. You can use this LM to run hashtag segmentor with the LM-based features. As described before, Hashtag Segmentor calls *runNBestGenerator.py* script at the background, which subsequently calls *nbest_generator* script.

Hence, in order to make this call chain to work, the minimal requirement is to setup OpenFST toolkit [93] on your system and make sure that path to */path/to/openfst/src/lib* is included into the system library path variable, which is *LD_LIBRARY_PATH* for the Linux-based systems.

If you are not satisfied with the dummy LM distributed with this toolkit, then you can generate your own by using *runNBestGenerator.py* script. However, before doing that, you need to install two additional 3^{rd} party packages. Those are AT&T's FSM Library [187] and SRI's SRILM [188]. You may add the path to their *bin* directories to your system path environment variable, such as *PATH* in case of Linux-based systems. You may also provide those paths as extra input arguments to *runNBestGenerator.py*.

5.1.6.2. Command Line. *runNBestGenerator.py* script accepts two types of input. One is a single character sequence to test the script and the second input type is a file that contains one or more character sequences (e.g. hashtag without '#' sign), each in its own row. In order to generate n-best segmentations, you can run this script with following arguments:

- --*input_text*: If you want to try out a character sequence, you put that sequence after this argument.
- --*input_file*: If you have a file to segment, you put the full path of that file after this argument.
- --*output_file*: If you want the script to output the results into specific file, you can set the full path of that output file this argument. By default, it is not set, which means that the script determines the output file name automatically by appending ".segmented.default" to the end of the input file.
- --*top_n*: This argument allows you set the number of highest scored segmented versions of input to be outputted. By default, it is set to 10.
- --*lm*: This is to set the LM. By default, it is set to be the dummy LM which comes with the toolkit.

To give an example, for the test purposes, if you like to segment "greatmovie" character sequence by using our dummy LM and get highest scored 5 segmentations, here is the command line:

```
python runNBestGenerator.py --input_text greatmovie --top_n 5
```

If you have a file, named "my.file", which contains one sequence of characters per each row and get the segmentation of highest scored 20 segmentations of each by using the dummy LM, here is the command line to run:

```
python runNBestGenerator.py --input_file my.file --top_n 20
```

If you like to train a new LM, you can give extra input arguments to the runNBest-Generator.py script. Here are those arguments:

- *--training_file*: This script is capable of generating LM by running series of commands from OpenFST toolkit and AT&T's FSM Library. LM is generated from a raw text file and this argument is used to give the full file path to that file.
- *--output_lm*: This is to set the name of the LM that is being generated.
- *--vocab_size*: In order to filter out least frequent words, this parameter allows you to keep the most frequent N words in the vocabulary of the LM. By default, N is set to 300,000.
- *--min_word_freq*: If you want to filter out words that do not occur more than some specific value, you can use this argument. By default, it is disabled.
- *--ngram*: This parameter is a specific setting used while training language models. It represents the maximum number of sequential words (i.e. n-grams) to be used to train a LM. By default, it is set to 4.
- *--srilm*: This is to give the path for the binary files of the SRILM toolkit. By

default, it is disabled provided that you add that path to your system.

- *--att_fsm*: This is to give the path for the binary files of the AT&T's FSM Library. By default, it is disabled provided that you add that path to your system.
- *--openfst*: This is to give the path for the binary files of the OpenFST toolkit. By default, it is disabled provided that you add that path to your system.

To give an example, if you have a text file named "my.text.file" and like to create a new LM named "my.lm"[34] by only keeping the most frequent 50,000 words in that text file, here is the command line:

```
python runNBestGenerator.py --training_file my.text.file
 --vocab_size 50000 --output_lm my.lm
```

5.1.6.3. Files and Folders. We provide all source code files for *nbest_generator* executable, along with *Makefile* in order to be able to compile the sources codes on your platform. Those files are located under the "*nbest_generator*" folder of the Hashtag Segmentor toolkit. In case of the *runNBestGenerator.py* script, it is located at the root folder of the toolkit.

It is also important to note the LM files, which are used by *nbest_generator*. In fact, LM is a bundle of six files. Those are: (1) vocabulary file, (2), vocabulary symbol file, (3) open fst file, (4) lexicon fst file, (5) determinized lexicon fst file, and (6) the file that keeps the parameters used while training the LM. You do not worry about these files. Each LM is stored in its own directory and that folder path identifies each individual LM. Our dummy LM comes inside the "*lms/dummy*" folder.

[34]By default, generated LMs are stored under "*lms*" folder of this toolkit.

5.2. Named Entity Disambiguation Tools

5.2.1. Knowledge Base Server and Candidate Generator

The first step for the named entity disambiguation task is to generate a list of candidate named entities for a given mention so that disambiguation step can choose the most likely one as the predicted output. In the literature, most of the studies [18, 177] use off-the-shelf indexers like Apache Lucene. They index the all possible known titles of the named entities and then find the entities that have similar title with respect to given mention surface form. Such methods do not make use of context fully. In Section 4.3.1, we argue that same named entities might be occur multiple times in the document and later mentions in the document tend to have the short form of the actual title of the named entity. Hence, they can be hard to detect. However, if we know which named entities are already suggested as candidate entities in surrounding mentions, we can give more chance to them when the surface form of the mention matches so little with the titles of those named entities.

For our studies, we implement the Knowledge Base (KB) Server and Candidate Generator together in one program. It is a stand-alone executable written in C programming language. It runs at the background and listens to specific port for commands. The goal of the KB Server is to keep certain information about named entities in the KB and make them accessible from web browser for fast exploration. Such information includes all known titles of the named entity, redirects (i.e. alternative titles) from Wikipedia dump, entity types seen in YAGO and BaseKB data sets. Since KB Server includes all title information, it can also act as a candidate generator. KB Server runs at the background because it requires to load and index too much data so that loading time can be as long as an hour. Hence, it is not practical to run it every time we need to generate candidates for a document. Instead, we run it once at the background and send our requests any time.

```
./KBServer port=<PORT_INDEX> params=<FILE_NAME>
```

5.2.1.1. Input Parameters. Our KB Server accepts the input parameters in a file at the execution time as shown above. It consists of two-column structure where the first column keeps the name of the parameter and the second column keeps the value of the parameter. Here are the parameters and description of their values:

- **wikipedia_ids**: This parameter points to a file where the list of all named entity ids are kept. In our research, we use Wikipedia as our actual Knowledge Base. Hence, each row in this file is the unique named entity identified with the Wikipedia ID. This ID is basically the main title of the named entity, such as "Barack Obama". Note that it is not single token, but contains white spaces. While loading it, KB Server replaces all the white spaces with underscore character.
- **wikipedia_redirects**: This parameter points to a file where redirects obtained from the Wikipedia dump are mapped to corresponding Wikipedia ID. Hence, it has two-column input. Note that there can be multiple redirects for a single Wikipedia ID. For example, "Barak Obama", "Barack H. Obama", "Borrack Obama" are some of the redirects of "Barack Obama". When people search for certain entities in Wikipedia, they may wrongly type the name of that entity. However, these redirects most often match with those wrongly entered names so that Wikipedia can "redirect" people to the right entity. Basically, redirects provide alternative titles for named entities.
- **wiki2basekb**: This file keeps the mapping between BaseKB ID and Wikipedia ID. It is used to connect BaseKB type taxonomy information to Wikipedia entries.
- **wiki2dbpedia**: This file keeps the mapping between DBPedia ID and Wikipedia ID. DBPedia is the main source of the named entity titles. Hence, this file helps us to know which DBPedia entry corresponds to which Wikipedia entry.
- **basekb_entity_types**: This file keeps the BaseKB types. It maps each BaseKB entry to a one BaseKB type. Hence, it has two-column structure.
- **yago_taxonomy**: This file keeps the YAGO type taxonomy. It maps child type to a parent type. For example, it maps "wikicat_American_female_musicians" to "wordnet_musician_110339966". Note that YAGO types are extension of WordNet synsets.

- **yago_entity_synsets**: It maps the named entity defined in YAGO data set to a YAGO type. Note that YAGO is built on top of Wikipedia. Hence, named entities in YAGO has Wikipedia IDs. In other words, this file maps Wikipedia IDs to YAGO types.
- **dbpedia_titles**: This files contains all seen titles for named entities in DBPedia namespace. It has three-column structure. The first column keeps the named entity ID in DBPedia namespace. The second and third columns keep one particular title and its frequency seen in DBPedia data set, respectively.

Once all these files are loaded, KB Server combines a lot of different information from Wikipedia, YAGO, BaseKB, and DBPedia and serves them from one point. Thanks to DBPedia, it knows which titles are used most to refer to each named entity. Redirect information from Wikipedia expands the list of alternative titles for each named entity. Thanks to YAGO and BaseKB, it knows the entity types of each named entity. Moreover, KB Server also automatically populates alternative titles based on the type of the named entity. For example, if named entity is a person type and has three words in its main title, it also generates a new alternative title with the first and third word from that main title, if no such title exists already.

5.2.1.2. Request Script and Output Format. As mentioned before, KB Server runs at the background and listens to specified port number for any request. We also implemented a separate Python script that gets the document with recognized mentions and converts that data to a special format that KB Server can understand and then sends it to the KB Server. Note that our candidate generator needs to see the whole document so that it can make use of the context better, as described earlier. Hence each input file should only contain mentions from one document. This script converts each recognized mention into four column input, each column separated by tab character and each such entry separated by other entry with the new line character. Those four columns are the (1) start position, (2) end position of the mention in the document, (3) mention surface form and (4) recognized named entity type. Here is how to call *getCandidates.py* script.

```
python getCandidates.py
   --server_ip <IP_ADDRESS>
   --server_port <INT>
   --input <FILE_NAME>
   --output <FILE_NAME; optional>
   --topn <INT; default 100>
   --max_edit_distance <INT; default -1>
```

You need to specify the IP address of the machine on which KB Server is running as well as the port number that you make it run. Then you need to specify the input file name with *–input* parameter. If you specify the output file name with *–output parameter*, it will write the candidates into that file. Otherwise, it will create a new file name by appending ".candidates" suffix to the input file name. You can also specify how many candidates you like to see at most with *–topn* parameter. If you have specific edit distance constraint, you can also specify maximum allowed edit distance value.

The output is produced in JSON format. Each candidate is given in a separate line and following fields are given for each candidate:

- **start**: Given start position for the mention
- **end**: Given end position for the mention
- **wiki_id**: Wikipedia ID of candidate named entity
- **basekb_id**: BaseKB ID for the candidate named entity
- **matched_title**: Matched title of the candidate that is most similar to given mention surface form
- **title_freq**: Title frequency for matched title of the candidate
- **total_title_freq**: Total frequency of all titles of the candidate named entity
- **num_other_occ**: Number of occurrences of the candidate named entity in given document
- **edit_distance**: Edit distance between matched title of the candidate and given mention surface form

- **jw_distance**: Jaro-Winkler distance between matched title and given mention surface form
- **matched_title_func**: Title function which refers to origin of the matched title. It can redirect, or auto-generated.

5.3. xDB (Experiment DB)

During our studies, we conducted over thousands of experiments. Each experiment runs with set of parameters and produces a result that is specific to those given parameter values. Keeping track of which parameter combinations results in the what outcome is a very important but difficult task. It is not only about keeping the note of the highest scored experiment at hand. There can various sets of independent experiments, not to mention re-calculating the experiments after every significant improvement to the algorithm. From the start, we needed a platform to save, manage, and navigate thousands of experiment results. In order to handle that load, we developed a web-based tool.

While designing this tool, we have couple of objectives in our mind:

- We should be able to create a personal account for each user in the system so that multiple researchers can use the tool at the same time.
- We should be able to create different experiments for different tasks. Results of various parameter combinations in each experiment can be saved under corresponding experiment record.
- We should be able to select specific parameter value and see only those experiment results in which that parameter is set to selected value. In other words, we should be able to filter results.
- We should be able to save different metric values such as precision, recall, F_1-score, accuracy and loss. Moreover, we should be able to save these values per each epoch so that we can see the change in their values over time.
- We should be able to create a task record from available parameters and keep it in the Task Queue so that an external script checks this queue for new tasks and

start the experiment with selected parameter values as soon as possible.

- Since multiple people (e.g. student and teacher) can be interested in the same experiment, they should be able to share the results with each other over the system.

We addressed each of above points and made our tool available for download at GitHub [189]. Following sections explain these points in detail. If you need more help, please visit the GitHub page of the xDB.

5.3.1. General Layout of the System

xDB tool consists of couple of components as depicted in Figure 5.2. The first component is the web user interface of the xDB. It allows user to interact with xDB platform through his or her web browser, such as Chrome. It consists of two parts; back-end and front-end. Back-end of the website is written in PHP. It resides on the web server and accepts requests from the front-end of the website and executes them. Front-end is written in JavaScript. By making use of AJAX methodology, the front-end presents a responsive user interface such that the web page does not need to be reloaded in order to show the response to each request. Instead, it executes the request at the background and shows the results on the fly.

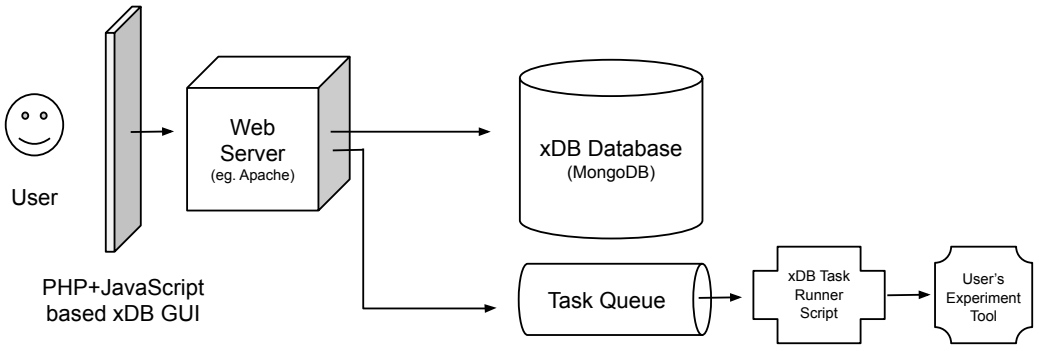

Figure 5.2. Diagram Showing the Components of xDB Platform and Its Interactions.

The second component of the xDB is the database where all records of the system are saved. For this purpose, we use external database toolkit, namely MongoDB. PHP can interface with the MongoDB through a driver that should be installed into the

web server. Once installed, through web server, xDB can execute commands such as reading from the database or changing the records.

The third component of the xDB is the Task Queue. This component allows user to create new runs through xDB user interface quickly and automatically run the user's experiment tool with selected parameter values. It basically consists of two parts. First part is the queue section in the website where we see the list of current tasks in the queue. The second part is a python script called "Task Running Script" in Figure 5.2. It runs at the background and checks the queue periodically. Once it detects new task, it runs the user's experiment tool with selected parameter values defined in the created task. Moreover, this task runner script can fire multiple scripts one after another depending on maximum allowed number of scripts that can run simultaneously on the same computer.

5.3.2. Setting up xDB

In order to install xDB and run it, you need to do following steps. However, before installing, you need to make sure that you satisfy the system requirements. xDB is a web application written in PHP and JavaScript. Hence, first and foremost, it requires a web server to run, such as Apache server. PHP has to be installed and activated in Apache settings. Moreover, since it assumes that the data is kept inside a MongoDB server, it also requires MongoDB server running on the machine. In addition to that, in order to be able to connect to the MongoDB server through PHP, PHP-MongoDB driver [190] must be installed. Also specific to PHP version 7, alcaeus' *mongo-php-adapter* module [190] must be installed and activated for further compatibility. As an extension to the xDB, it includes a Task Queue which manages submitted actions. In order to run these actions as they are created, there is also an Python script which runs at the background and checks for newly submitted actions every second or so. In order to be able to run that script, Python must be installed on the server, as well.

xDB is publicly available at GitHub. In order to install it, you need to clone or download xDB repository from GitHub onto your machine. After opening it up,

you need to copy the contents of the “web” folder into a folder under the root folder of your web server such as Apache. Assume that you have a Linux environment and web server folder is located at “/var/www/html” and you choose the name “xdb” for that folder, then files under web folder will be copied under “/var/www/html/xdb”. Having said that, if you are using your localhost, then the web page address to see xDB is “localhost/xdb”. If you are using PHP version 7, then as aforementioned, you also need to setup alcaeus’ mongo-php-adapter and copy the ”vendor” folder of the mongo-php-adaptor under the “xdb” folder.

Once you enable PHP to interface with MongoDB server running on a machine, you need to set the address of that MongoDB database in the “php/utils.php” file. Without doing that, when you try to see the content of “xdb” folder with your web browser, it will show an error message. You can change the “$salt” value in the “php/utils.php”. This is one layer of security around saved user passwords in the xDB. Note that xDB does not keep passwords in plain text is uses crypt function of PHP and adds “salt” onto that to make it more secure. Note that you are not advised to change the salt after setting up the xDB because already signed up users cannot log in anymore. Also make sure that you do not share the salt value publicly. Otherwise saved passwords in the xDB becomes vulnerable to being decoded.

Next step is to set the admin account. You need to open the address of newly setup xDB into your web browser. In example above, it is “localhost/xdb”. If everything is okay so far, then it requests you to enter a password for admin account. Enter the same password twice. This will create an admin account in the xDB so that you can create user accounts.

In order to make the task queue component of the xDB operational, you need to open “runTaskQueue.py” script and set the path of your experiment tool. Note that your tool needs to accept input arguments through the command line. It should also handle accept specific parameter values that is specific to xDB, such as experiment id, experiment token. Once it can accept the input as specified, you need to run “runTaskQueue.py” with experiment id and leave it.

5.3.3. User Accounts in xDB

There can be many people in a research lab and you can create one account for each user inside xDB. xDB comes with "admin" user, which manages all the users in the xDB platform. He or she can approve new user requests or even delete the user.

In order to create a new account, you need to enter a valid email address and a password. There is no email confirmation in xDB. Your request will be saved into xDB so that later on admin user can see the new requests and approve or reject them. When it is approved, you can get to the login panel and enter your user information to start using the platform.

5.3.4. Experiments in xDB

If you are experimenting on any task like named entity disambiguation or sentiment analysis, you implement a tool and start testing it on the development and test sets. You may run that tool multiple times with different parameter values and normally keep all the results you obtained in a text file or maybe in spreadsheet for quick access. In xDB, all the runs you do regarding one task is called experiment. All the data in xDB is kept inside experiments. Hence, first thing you need to do is to create a new experiment by clicking "Create a new Experiment" link shown at the top of the xDB panel after logged in. It opens up a new dialog window where you enter the name of the experiment.

Once you create it, you are ready to save your experimental results inside xDB. Keep in mind that you do not fill some kind a form to save the results. Instead, you call the xDB API from your tool whenever you calculate the results. In order to do that, you need three things: (1) web address of the xDB API, (2) experiment ID (3) experiment token. The token is like a secret key that only the owner of the experiment can know, like a password. Yet, owner can share it with other people so that they can also run their tools and insert new results to the owner's experiment. Experiment ID and token can be reached by opening the settings of the experiment. Once you have

them, you can make an API call like shown below.

```
[XDB-WebAddress]/php/xdbEngine.php?cmd=create_new_run&
  experiment_id=[id]&token=[experiment-token]&
  params=[list-of-parameters]&dev=[dev-file]&test=[test-file]
```

5.3.4.1. Runs in Experiments. In the API call shown above, the command you call is to create a new run entry. Other than experiment ID and token, it also needs the parameters and development and test file names. Note that you can run your tool multiple times with different parameter values. In xDB, each time you execute your tool with different parameter set, it is called "run". Since this is a URL link, parameters should be given in key=value format, each separated with ampersand character. Once you make this call, it creates a new run entry in xDB database and returns its ID. Now, you are ready to save the result for this specific run. In order to do that, you need to do the following API call with experiment id and token, as well as run id which was returned by the first call.

```
[XDB-WebAddress]/php/xdbEngine.php?cmd=report_iteration_result&
  run_id=[id]&experiment_id=[id]&token=[experiment-token]&
  is_test=[0/1]&is_best=[0/1]&epoch=[number]&fscore=[value]&
  precision=[value]&recall=[value]&accuracy=[value]&loss=[value]
```

As the command (i.e. "cmd") field suggested, it is to report the iteration result. Meaning, you can not only save the final result, but also result of each epoch which happens when you finish looping over the training data set. If you look at the other fields, you can see that, you can give F_1-score, precision, recall, accuracy and even loss value to be saved as the result. You can also specify the epoch index as well as whether this result is taken on development or test set (by setting "is_test" field 0 or 1, respectively)

Once you save the results by making these calls, you are ready to see them inside xDB. In order to do that, you will a list of available experiments after logged in. Then when find the experiment, you will see list of development files. If you save the results on one development file, there will be one listed then. Once you click on that file, you will see a list of saved results.

5.3.4.2. Browsing the Results. xDB allows you browse the results of all runs you saved in an experiment. First, you can order them based on specific evaluation metric like F_1-score or accuracy. Note that you can alter shown evaluation measure(s) via the settings of the experiment. Each run is shown along with its given specific parameter values. In order to make the readability easy, if there are certain parameter values are all shared among all listed runs, then it is shown separated at the top of the list and removed from the list of parameters from all runs. That way, you can more easily focused on differences (i.e. different parameters values) between the runs. Moreover, you can even click on any specific parameter and filter only those runs that have that specific parameter value. The list will update itself and selected parameter will be shown above the list so that you can remove that selection and make all runs re-appear.

If you want to see the saved results of one particular run in more detail, you can click on the F_1-score or accuracy or loss value and new popup window opens up and show the graph of how that selected evaluation measure value changes over the epochs. Note that listed runs are ordered based on development set results, not test set results. That is because in an scientific experiment, you need to select the best possible parameter combination based on development set results and then report the result of those parameter combination(s) on the test set. Hence, in Figure **??**, the result on the test set is separately given at the point where the result is the highest on the development set.

5.3.4.3. Adding a Baseline. One of the components of a typical experiment is the baseline. It measures the success of a simple method on the development or test set. The goal of the experiment is to suggest a new method that exceeds the baseline

method. Hence, it is the case that you may have one or multiple baseline scores in your hand and you may like to see where it is positioned with respect to the results you obtained from your runs.

In order to do that, you need to go to the experiment menu at the toolbar and select “Add a new baseline” option. The dialog panel pops up. You can set the name of the baseline and enter the different evaluation measure values for that baseline. Once you add it, it will be listed as a run among other runs with highlighted differently. Now, you can easily see which runs are better than the baseline. You can delete the baseline and add new ones, if you like.

5.3.5. Tasks in xDB

As described in previous section, when you run your tool, you call xDB API to create the run entry with parameter values that are given to that tool. In other words, you should be able to provide those parameter values to your tool as an argument on your own. This might be not feasible if you have many parameters, like fifty. You may write your own program to generate all possible parameter value combinations and run all of them. Such search grid approaches can be used while experiment. However, you may need to wait for a long time to finish.

xDB allows you to create a task in an experiment where you manually set the value of parameters. Note that when you save the results of the run into the xDB platform, it also keeps track of all used parameter values. You may see and manage the list by selecting the “Manage Parameters” option under the experiment menu. You can edit existing ones or add new parameter values. While creating the task, these parameters are shown to you so that you can select among them. Nevertheless, it is still not easy to select the value for each parameter from shown list. Hence, you can also click “run” link shown for each listed run inside the experiment and xDB shows you the “Create a Task” window where all parameters of the experiment are listed but this time the parameters of the chosen run come as pre-selected. This enables you to change only small number of parameter values and create a new task with selected

values. In other words, you can easily create a new task that resembles closely to existing run.

Once you create a new task, it is placed into the task queue. Each experiment has it own task queue. Hence, from the experiment menu, you can open it and see all the tasks inside. After placed into the queue, now it is the job of external script that checks the queue in frequent periods and start your tool automatically with the selected parameter values at the background. This way, you can quickly start new runs and get their results into the xDB platform.

5.3.6. Sharing Results

One of the important objectives of xDB is to be able to share the results of experiments with others in the system so that they can track the progress. One particular scenario is student-teacher relation. Students run the experiments and teacher observes the results. You can also see it like reporting the results from inside the system directly rather than copy-and-pasting them into an email and send it to related party.

In xDB, you can share the results of a development set in an experiment, rather than whole experiment. When you open the experiment, you go to the file menu which resides at the right hand-side of the experiment menu and then select "Share the Results of This File" option. A popup window will emerge, you need to enter the username of the other user. The experiment and shared development file will appear at the panel of selected person.

6. Conclusion

6.1. Contributions

Today's high-performing machine learning algorithms generally depend on manually annotated training data, which can be hard and costly to curate. In this thesis, we explored weakly-supervised learning for two NLP tasks: hashtag segmentation and named entity disambiguation. We showed that we can achieve the state-of-the-art (SOTA) results by designing heuristics to automatically create training data by utilizing the patterns in the raw data as well as by re-purposing the existing data sets that have not been originally created for the target tasks.

In case of the hashtag segmentation task, manually segmented hashtag data sets are not sufficiently large for training. For example, the data set curated by [116] contains 1268 manually segmented hashtags. More recently, [191] shared a larger data set of 12,594 hashtags. People tend to give the same message both in the regular tweet text and in the hashtags. Therefore, thanks to half a billion tweets in the SNAP data set [30], there are tweets that include both a hashtag and its segmented formation in their text. We hypothesize that if we detect such pairs of tweet text and hashtags and filter out the infrequent cases, the rest may form a viable set of training data for hashtag segmentation. In case of the named entity disambiguation task, we hypothesize that being able to predict the entity type of a mention, which is called mention typing, beforehand can increase the performance of the disambiguation task. Yet, curating a type taxonomy is very hard and there is no an explicitly created training data set for mention typing. Instead, we hypothesize that named entities that occur in similar context should be assigned to the same type because context is half of the input for mention typing, while the other half is the surface form of the mention. Following that, we cluster named entities based on the semantic similarity of their context and assign cluster IDs as types, namely cluster-based types. Moreover, we re-purpose the hyperlinked mentions in Wikipedia articles by labeling them with corresponding cluster-based types, which gives us the training data for mention typing.

For both tasks, we design heuristics that rely on the underlying characteristics of the problem space. The experimental results for both tasks support our hypotheses. Here are the detailed contributions of this thesis work.

In the first part of this thesis, we tackled the hashtag segmentation task, which is in essence a word boundary detection task. In this particular task, if the language is a space-delimited one like English, it is easy to obtain training data, because any space-delimited text can be used as a training data set. In other words, the word boundaries come already annotated within a regular text. However, segmenting hashtags into their constituent words is not as easy as segmenting words in regular text like news articles. That was shown by the poor performance of the SOTA Word Breaker tool of Microsoft on hashtags in Table 3.7. Word Breaker achieves only 84.4 and 84.6 F_1-score on the Test-BOUN and Test-STAN sets, respectively. That means that if we use regular space-delimited text as training data, it may not be enough. Instead, we proposed a simple heuristic based on traversing half a billion tweets and breaking the seen hashtags into the most possible original words with certain confidence. That gave us 803,000 automatically segmented hashtags. We used this as a training data set for hashtag segmentation. Even though it might include inexact segmentations, the results on the Test-BOUN and Test-STAN test sets indicate its potential. Especially on the Test-BOUN set, the training set that includes automatically segmented hashtags achieves 94.9 F_1-score, which is better than using regular space-delimited text obtained from tweets, which achieves 93.6 F_1-score. On Test-STAN, the difference is not significant.

At the time of our study, there was no extensive work on hashtags, even though tweets and thus hashtags have been widely used on the Web. We proposed a feature-rich approach, which converts the raw input into a list of features and learns to predict the word boundary positions based on the observed features at the target character. We formulated the problem as a binary classification task and used the maximum entropy classifier (MaxEnt) to model the task. We explored over 100 features, which are grouped into four types.

Vocabulary-based features look for known words around the target character. The existence of long words before or starting at the target character indicates a boundary with high probability. One important point regarding vocabulary-based features is that we did not use the words themselves as feature values. If we used the words themselves, there could be as many features as the number of different words in the dictionary, or more if we consider n-grams of words. This would considerably increase the complexity of the model to be learned. Instead, we encoded the words in terms of their frequencies (seen in Wikipedia) and lengths. If a word is observed 1000 times and consists of 5 characters, we represented it as "3:5". "3" represents the floored negative logarithm of its frequency and "5" is its length. In other words, we grouped words based on their frequencies and lengths and used the group ID as a feature. This effectively reduced the number of unique features and made the model simpler and easier to learn.

Vocabulary-based features are limited due to the possibility of new or out-of-vocabulary words. Hence, we also designed orthography-based features as a second type of features. These features depend on the shapes of the characters around the target character. For example, if the target character is capitalized, while the previous one is not, this might indicate a word boundary.

The third type of features are context-based features. Even though the words of the hosting tweet have been used as context features in [116], we extended that approach with context coming from multiple tweets that host the same hashtag. While words in tweets can be good indicators of boundaries, people may also capitalize the same hashtag differently. We used all these clues together and showed that as few as 20 tweets that host the same hashtag can provide effective information about how to segment that hashtag.

The forth type of features that we investigated are Language Model (LM)-based features. LMs are originally designed to score the next word in a sentence and give the probability of the sentence being uttered. An LM can also be used to get the N-best segmentations of a character sequence into its original words. We trained a LM on a large corpus, which consists of 1 billion words. We obtained the N-best segmentations

and encoded the boundary positions as features in our feature-based approach. With the help of context-based features and especially LM-based features, we achieved SOTA results on both Test-BOUN and Test-STAN test sets.

We used our state-of-the-art hashtag segmentor to segment 2.1 million distinct hashtags and studied their internal structure. This is the first study of its kind in the literature. The first thing we observed is that 90% of 2.1 million hashtags include multiple words. This indicates that in order to understand the hashtags, we first need to have a hashtag segmentor. Moreover, our analysis revealed that around 80% of the hashtags that contain positive or negative sentiment consist of multiple words. In other words, 80% of the time sentiment is trapped inside multi-word hashtag. This also means that in order to perform sentiment analysis on tweets, a hashtag segmentor may help to identify the sentiment inside multi-word hashtags. Recently, [191] showed that applying hashtag segmentation in the SemEval 2017 sentiment analysis task increases the average recall by 2.6%. After segmenting the hashtags, we applied TweeboParser [127] on those and conducted a grammatical analysis of 2.1 million hashtags. It is yet again the first study of its kind. We discover that about quarter of distinct hashtags are written as verb headed expressions, especially in imperative form. However, as we consider all occurrences, people tend to use noun-based hashtags more often. Adjectives are mostly used inside expressions rather than as single word hashtags. This observation might explain why sentiment is trapped more often inside multi-word hashtags, since sentiment is mostly carried by the adjectives. All in all, all our observations show that hashtags are not simple one-word labels. Hence, hashtag segmentation is necessary for the understanding and utilization of hashtags.

As a final contribution in this phase of this thesis, we publicly shared our code and data sets. The source code for the hashtag segmentor is available at GitHub along with explanatory guidelines. We shared our automatically segmented 803,000 hashtags as well as the Test-BOUN test set, so that other researchers can test and evaluate their systems with respect to ours.

Since our publication, there have not been many study done on hashtag segmentation. More recently, [191] framed the hashtag segmentation task as a pairwise ranking problem, where they order the candidate segmentations of a given hashtag. They applied a neural network based multi-task learning approach and achieved 94.5% accuracy on the Test-STAN test set, whereas they reported that our segmentor achieved 92.4% accuracy. However, they trained our system on their own training set and we believe the version without the LM-based features was used. Our results showed that the LM-based features significantly improve the performance of our segmentor.

At the second phase of this thesis, we focused on the named entity disambiguation (NED) task. This field has been well studied and several recent deep learning techniques have been already investigated in the NED literature. Unlike most prior studies, we approached the problem from weakly-supervised learning perspective. We re-purposed a large amount of data that is not directly related to the NED task. Hence, we explore the idea that if we predict the type of a mentioned named entity, we can use that information at the disambiguation step as an extra clue. This task of predicting the type of a mention is called mention typing. However, the success of mention typing depends directly on the type set or type taxonomy. If it is a small set, it does not provide much information for disambiguation, even though the type prediction task is easier to learn. If it is a large set, then the task of predicting the types becomes harder to learn. The available type taxonomies are curated by humans and there are a number of disadvantages. Firstly, it is labor intensive. Secondly, the types are not designed based on the contexts of the mentions. Instead, they are buckets that hold categorically the same type of entities. However, predicting the type of a mention directly depends on the context. Thirdly, considering that there are over five million named entities in Wikipedia-based Knowledge Base, such a type taxonomy is inherently incomplete.

All these considered, we proposed to generate our type set by clustering named entities based on their contextual similarity. Each cluster becomes a type, hence named as cluster-based type. Our research is the first at using clustering to obtain a cluster-based mention typing model. We clustered over five million named entities. Then, a mention typing model is trained on the hyperlinked mentions of the named entities in

Wikipedia articles, after we label them with their corresponding cluster-based types. At the end, we use the cluster-based type prediction of a given mention as an extra clue to improve the NED task. We implemented a basic two layer feed-forward neural network with no addition of attention mechanism. We convert each candidate into an input feature vector and calculate its probability of being the real mentioned named entity. The highest scored one becomes the prediction of our system. Our system achieves better or comparable results based on randomization tests with respect to the state-of-the-art levels on four defacto test sets. The successful contribution of cluster-based mention typing modeling is also supported by the fact that our disambiguation model is simple compared to other studies in the field.

As specified before, being able to utilize context as much as possible directly affects the success of the NED task. Hence, we proposed three different ways of representing context. The first context type is word-based context, which considers the local (or sentence-based) context around the mention. Types predicted based on this context are expected to be like semantic types, because the surrounding words like verbs and adjectives are likely to give clues about the semantic meaning of the entity. The second type of context is surface form-based context, which includes all surface forms of the surrounding mentions in the same document. This is more like document-level context. Types predicted based on this context can be more like topics, because context only includes nouns and it is broader compared to sentence-level context. The third type of context is entity-based context, which includes the entity IDs of the surrounding mentions. It is more entity-specific, because context is represented directly in terms of the entity IDs instead of the words composing the entity mention. Modeling based on this type of context is also modeling the co-occurrence and order relations (i.e. which entity is often seen before which) between named entities. Representing context in these three formats allows us to obtain three different types of entity embeddings and thus three different clusterings of named entities. We run word2vec on each context type to obtain embeddings and run K-means on those embeddings to cluster them. We also represent context in terms of synsets assigned to named entities in the YAGO and BaseKB data sets and obtain a forth way of clustering named entities. As a fifth approach, we use Brown clustering on entity-based context as an alternative to

K-means. At the end, we ended up having five different clusterings of named entities and thus five mention typing models, each looking at the context from different level. Our experiments show that these five models complement each other, since the best results are achieved when all are used together.

Another important contribution in these studies is the candidate generation tool that we design. We again utilize the idea that context is very important. We propose using the candidates of the surrounding mentions in the same document. It is observed that the same named entity can be mentioned multiple times in the same document and its later mentions in the document are most likely to be in their shorter forms. Moreover, if the later mentions consists of only one word, it may refer to many possible named entities in the knowledge base. Think about how many people there are with the name "Mike". However, if there is already a mention of "Mike Tyson" in a document, it becomes more likely that the single word "Mike" in the same document refers to "Mike Tyson". By using this intuition, we implemented our own candidate generator instead of using off-the-shelf tools, which achieved higher gold recall values than the previously reported results in the literature. We publicly share our tool and data sets.

All in all, in this thesis, we focused on automatically generating training data by employing weakly-supervised approaches for two NLP tasks. For both tasks, we used simple learning algorithms, namely the Maximum Entropy classifier for hashtag segmentation and a shallow feed-forward neural network for named entity disambiguation. Nevertheless, state-of-the-art performance is obtained by utilizing automatically generated training data and by a detailed representation of the input in terms of features or different levels of context information. Our results show that when we design heuristics that rely on the underlying characteristics of the problem space, the training data generated by those heuristics can help achieve state-of-the-art results.

6.2. Impact

In this thesis, we designed weakly-supervised approaches to automatically generate labeled data for training purposes. We developed and evaluated the proposed

approaches for English, but these approaches can be easily applicable to other languages. Our heuristic to obtain auto-segmented hashtags from a tweet collection is completely language independent. For example, we can apply it on tweets in Turkish and obtain auto-segmented hashtags in Turkish. One possible constraint might be the size of the tweet collection. In case of frequently used languages, it would be very easy to collect millions of tweets. But that may not be the case for less frequently used languages on the Internet like Catalan [192] from Spain, or even endangered languages like Siwi [193] from Egypt. In case of our cluster-based mention typing approach, we can use Wikipedia in Turkish to obtain mention typing models for Turkish. This is again limited by the size of Wikipedia in the target language. For example, there are over five million articles in the English version of Wikipedia, but that number goes down to just over 350,000 in case of Turkish. The proposed mention typing model also relies on DBpedia for the title database. Nevertheless, DBpedia comes in 125 languages. For the synset-based typing model, we use the YAGO and BaseKB data sets. They can be considered as language independent assuming that named entities in those data sets also include foreign (non-English) named entities. To sum up, we believe sufficient data are available on the Internet to train and use the proposed cluster-based typing models for most of the widely-spoken languages.

The utility of predicting the cluster-based type of a mention can go beyond named entity disambiguation. For example, cluster-based mention typing can be applied to improve co-reference resolution. One possible challenge for this particular application is the existence of pronoun-based mentions. Since the surface form of a pronoun-based mention is generic, meaning any named entity might be referred to by that, the mention typing model will not be able to utilize the surface form and it will rely only on the left and right context of the mention. Like co-reference resolution, another application area is NIL clustering, which is the task of clustering the mentions of NIL entities (i.e., entities that have no corresponding entries in the KB) and identifying all the mentions that correspond to the same NIL entity. In addition to using the surface forms of the mentions, utilizing their context is key to success. Prediction of the cluster-based types might provide extra clues for this task. This can be especially helpful at the corpus level (*i.e.* cross-document) as there is more context to predict cluster-based types.

The proposed cluster-based mention typing model can also be applied to the relation extraction task. The goal of this task is to detect relations between entities that are mentioned in the text. In the literature, various approaches are used to extract relations such as rule-based approaches like trying to match a given text with phrase templates that are known to indicate specific relations. There are also feature-based approaches, which extract syntactic and semantic features from a given text and try to predict the relation between two entities that occur in the same sentence. In such cases, prediction of the cluster-based types of the entities might be used as an extra clue. Another method for relation extraction is using string kernels [194]. Originally proposed for the text classification task, string kernels compute the similarity of two strings. [195] applied string kernels to relation extraction by modeling the context around the two entities, which they refer to as the before, middle and after portions of the context. Considering that our cluster-based mention typing models also take into account the left and right context of the mention separately in the model architecture, the mention typing approach is very similar to string kernels. Hence, our cluster-based mention typing model has potential to improve the relation extraction modelling, like string kernels.

Another application area of cluster-based mention typing might be question answering. In a configuration where possible candidate answers are generated and the output is determined by scoring them, the task becomes a ranking task. Then, our ranking approach in named entity disambiguation can be applied for question answering as well. As in our case, cluster-based types can be assigned to candidate answers beforehand. The question can be used as context to calculate the probability of each cluster-based type and that probability can be used for scoring the candidate answers.

All in all, the approaches proposed in this thesis can be extended in various directions.

REFERENCES

1. Krizhevsky, A., I. Sutskever and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

2. Hinton, G., L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition", *IEEE Signal Processing Magazine*, Vol. 29(6), pp. 82–97, 2012.

3. Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, "Natural Language Processing (Almost) from Scratch", *Journal of Machine Learning Research*, Vol. 12, pp. 2493–2537, 2011.

4. Radford, A., L. Metz and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", *arXiv preprint arXiv:1511.06434*, 2016.

5. Veit1, A., N. Alldrin, G. Chechik, I. Krasin, A. Gupta and S. Belongie, "Learning From Noisy Large-Scale Datasets With Minimal Supervision", *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6575–6583, 2017.

6. Bilen, H. and A. Vedaldi, "Weakly Supervised Deep Detection Networks", *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2846–2854, 2016.

7. Durand, T., T. Mordan, N. Thome and M. Cord, "WILDCAT: Weakly Supervised Learning of Deep ConvNets for Image Classification, Pointwise Localization and Segmentation", *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 642–651, 2017.

8. Ni, J., G. Dinu and R. Florian, "Weakly Supervised Cross-Lingual Named Entity Recognition via Effective Annotation and Representation Projection", *Proceedings of the Association for Computational Linguistics (ACL)*, pp. 1470–1480, 2017.

9. Zhou, Z.-H., "A brief introduction to weakly supervised learning", *National Science Review*, Vol. 5(1), pp. 44–53, 2017.

10. Blum, A. and T. Mitchell, "Combining Labeled and Unlabeled Data with Co-Training", *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, pp. 92–100, 1998.

11. Nigam, K. and R. Ghani, "Analyzing the effectiveness and applicabilbity of co-training", *Proceedings of the 9th International Conference on Information and Knowledge Management*, pp. 86–93, 2000.

12. Ritter, A., S. Clark, M. Etzioni and O. Etzioni, "Named Entity Recognition in Tweets: An Experimental Study", *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1524–1534, 2011.

13. Finkel, J. R., T. Grenager and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling", *Proceedings of the ACL*, pp. 363–370, 2005.

14. Sang, E. and F. Meulder, "Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition", *Proceedings of Conference on Natural Language Learning*, pp. 142–147, 2003.

15. Ratinov, L. and D. Roth, "Design challenges and misconceptions in named entity recognition", *Proceedings of CoNLL*, pp. 147–155, 2009.

16. Liu, X., S. Zhang, F. Wei and M. Zhou, "Recognizing Named Entities in Tweets", *Proceedings of ACL*, pp. 359–367, 2011.

17. Bunescu, R. and M. Pasca, "Using Encyclopedic Knowledge for Named Entity Disambiguation", *Proceedings of European Chapter of the Association for Computational Linguistics (EACL)*, pp. 9–16, 2006.

18. S. Hakimov, H. H., S. Jebbara, M. Hartung and P. Cimiano, "Combining Textual and Graph-based Features for Named Entity Disambiguation Using Undirected Probabilistic Graphical Models", *Knowledge Engineering and Knowledge Management (EKAW)*, pp. 288–302, 2016.

19. Radhakrishnan, P., P. Talukdar and V. Varma, "ELDEN: Improved Entity Linking using Densified Knowledge Graphs", *Proceedings of NAACL-HLT*, pp. 1844–1853, 2018.

20. Ling, X. and D. Weld, "Fine-grained entity recognition", *Proceedings of AAAI*, Vol. 12, pp. 94–100, 2012.

21. Yosef, M., S. Bauer, J. Hoffart, M. Spaniol and G. Weikum, "HYENA: Hierarchical type classification for entity names", *Proceedings of 24th International Conference on Computational Linguistics (COLING)*, pp. 1361–1370, 2012.

22. Mahdisoltani, F., J. Biega and F. Suchanek, "YAGO3: A Knowledge Base from Multilingual Wikipedias", *Proceedings of Conference on Innovative Data Systems Research (CIDR)*, pp. 177–185, 2015.

23. Celebi, A. and A. Ozgur, "Segmenting hashtags using automatically created training data", *Proceedings of Language Evaluation and Resources Conference (LREC)*, pp. 2981–2985, 2016.

24. Celebi, A. and A. Ozgur, "Segmenting Hashtags and Analyzing Their Grammatical Structure", *Journal of the Association for Information Science and Technology (JASIST)*, Vol. 69(5), pp. 675–686, 2018.

25. Celebi, A. and A. Ozgur, "Cluster-based Mention Typing for Named Entity Dis-

ambiguation", *Natural Language Engineering, accepted for publication*, 2020.

26. Celebi, A. and A. Ozgur, "Description of the BOUN System for the Trilingual Entity Detection and Linking Tasks at TAC KBP 2017", *Proceedings of Text Analysis Conference (TAC)*, 2017.

27. Kökciyan, N., A. Çelebi, A. Özgür and S. Üsküdarlı, "BOUNCE: Sentiment Classification in Twitter using Rich Feature Sets", pp. 554–561, 2013.

28. Celebi, A., "*Hashtag Project Web Page at TABI Lab Website*", `https://tabilab.cmpe.boun.edu.tr/projects/hashtag_segmentation/`, 2020, accessed in July 2020.

29. Celebi, A., "*NER Project Web Page at TABI Lab Website*", `https://tabilab.cmpe.boun.edu.tr/projects/ned_w_cmt/`, 2020, accessed in July 2020.

30. Yang, J. and J. Leskovec, "Patterns of temporal variation in online media", *Proceedings of the fourth ACM international conference on Web Search and Data Mining*, pp. 177–186, 2011.

31. Gatos, B., N. Stamatopoulos and G. Louloudis, "ICDAR 2009 handwriting segmentation contest", *Proceedings of International Conference of Document Analysis and Recognition (ICDAR)*, pp. 1393–1397, 2009.

32. Wong, P. and C. Chan, "Chinese word segmentation based on maximum matching and word binding force", *Proceedings of the 16th conference on Computational Linguistics*, pp. 200–203, 1996.

33. Lafferty, J., A. McCallum and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data", *Proceedings of the ICML*, pp. 282–289, 2001.

34. Xue, N., "Chinese word segmentation as character tagging", *International Journal*

of Computational Linguistics and Chinese Language Processing, Vol. 8(1), pp. 29–48, 2003.

35. Peng, F., F. Feng and A. McCallum, "Chinese segmentation and new word detection using conditional random fields", *Proceedings of the 20th international conference on Computational Linguistics*, pp. 562–568, 2004.

36. Zhang, Y. and S. Clark, "Chinese segmentation with a word-based perceptron algorithm", *Annual Meeting of the Association of Computational Linguistics*, pp. 888–896, 2008.

37. Magistry, P. and B. Sagot, "Unsupervized Word Segmentation: the case for Mandarin Chinese", *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pp. 383–387, 2001.

38. Chen, S., Y. Xu and H. Chang, "A simple and effective unsupervised word segmentation approach", *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 866–871, 2012.

39. Rumelhart, D. and J. McClelland, "On learning the past tenses of english verbs", *Parallel distributed processing: explorations in the microstructure of cognition*, Vol. 2, pp. 216–271, 1986.

40. Daelamans, W., A. van den Bosch and A. Weijters, "Igtree: Using trees for compression and classification in lazy learning algorithms", *Artificial Intelligence Review*, Vol. 11, pp. 407–423, 1997.

41. Islam, A., D. Inkpen and I. Kiringa, "A generalized approach to word segmentation using maximum length descending frequency and entropy rate", *Computational Linguistics and Intelligent Text Processing Lecture Notes in Computer Science*, Vol. 4394, pp. 175–185, 2007.

42. Lee, Y., K. Papineni and S. Roukos, "Language model based arabic word seg-

mentation", *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pp. 399–406, 2003.

43. Wang, K., C. Thrasher and B. Hsu, "Web scale nlp: A case study on url word breaking", *Proceedings of The International World Wide Web Conference*, pp. 357–366, 2011.

44. Sang, E., "Introduction to the CoNLL-2002 Shared Task: Language- Independent Named Entity Recognition", *Proceedings of Conference on Natural Language Learning*, pp. 155–158, 2002.

45. NIST, "NIST:ACE08 Evaluation Plan", `https://my.eng.utah.edu/~cs6961/papers/ACE-2008-description.pdf`, 2008, accessed in July 2020.

46. Grishman, R., "The NYU system for MUC-6 or Wheres the Syntax", *Proceedings of Sixth Message Understanding Conference*, pp. 167–195, 1995.

47. Wakao, T., R. Gaizauskas and Y. Wilks, "Evaluation of an Algorithm for the Recognition and Classification of Proper Names", *Proceedings of COLING-96*, pp. 418–423, 1996.

48. Bikel, D., S. Miller, R. Schwartz and R. Weischedel, "Nymble: a high performance learning name-finder.", *Proceedings of the Fifth conference on Applied Natural Language Processing*, pp. 194–201, 1997.

49. Borthwick, A., *Maximum Entropy Approach to Named Entity Recognition*, Ph.D. Thesis, New York University, 1999.

50. Srihari, R., C. Niu and W. Li, "A Hybrid Approach for Named Entity and Sub-Type Tagging", *Proceedings of the Sixth Conference on Applied Natural Language Processing*, pp. 247–254, 2000.

51. Riloff, E. and R. Jones, "Learning Dictionaries for Information Extraction by

Multi-level Bootstrapping", *Proceedings National Conference on Artificial Intelligence*, pp. 1044–1049, 1999.

52. Cucchiarelli, A. and P. Velardi, "Unsupervised Named Entity Recognition Using Syntactic and Semantic Contextual Evidence", *Computational Linguistics*, Vol. 27(1), pp. 123–131, 2001.

53. Pasca, M., D. Lin, J. Bigham, A. Lifchits and A. Jain, "Organizing and Searching the World Wide Web of FactsStep One: The One-Million Fact Extraction Challenge", *Proceedings National Conference on Artificial Intelligence*, pp. 1400–1405, 2006.

54. Collins, M. and Y. Singer, "Unsupervised Models for Named Entity Classification", *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pp. 100–110, 1999.

55. Etzioni, O., M. Cafarella, D. Downey, A.-M. P. T. Shaked, S. Soderland, D. S. Weld and A. Yates, "Unsupervised Named-Entity Recognition: Generating Gazetteers and Resolving Ambiguity", *Artificial Intelligence 165. Essex: Elsevier Science Publishers*, pp. 91–134, 2005.

56. Nadeau, D., P. Turney and S. Matwin, "Unsupervised Named-Entity Recognition: Generating Gazetteers and Resolving Ambiguity", *Proceedings of Canadian Conference on Artificial Intelligence*, pp. 266–277, 2006.

57. Evans, R., "A Framework for Named Entity Recognition in the Open Domain", *Proceedings of Recent Advances in Natural Language Processing*, pp. 267–274, 2003.

58. Huang, Z., W. Xu and K. Yu, "Bidirectional LSTM-CRF models for sequence tagging", *arXiv preprint arXiv:1508.01991*, 2015.

59. Lample, G., M. Ballesteros, S. Subramanian, K. Kawakami and C. Dyer, "Neural

architectures for named entity recognition", *Proceedings of the NAACL*, pp. 260–270, 2016.

60. Chiu, J. and E. Nichols, "Named entity recognition with bidirectional LSTM-CNNs", *Transactions of the Association for Computational Linguistics (TACL)*, Vol. 4, pp. 357–370, 2015.

61. Ma, X. and E. Hovy, "End-to-end sequence labeling via bidirectional LSTM-CNN-CRF", *Proceedings of ACL*, pp. 1064–1074, 2016.

62. McNamee, P., H. Simpson and H. Dang, "Overview of the TAC 2009 knowledge base population track", *Proceedings of Text Analysis Conference (TAC)*, pp. 111–113, 2009.

63. Zheng, Z., F. Li, M. Huang and X. Zhu, "Learning to link entities with knowledge base", *Proceedings of the NAACL*, pp. 483–491, 2010.

64. He, Z., S. Liu, M. Li, M. Zhou, L. Zhang and H. Wang, "Learning Entity Representation for Entity Disambiguation", *Proceedings of ACL*, pp. 30–34, 2013.

65. Francis-Landau, M., G. Durrett and D. Klein, "Capturing Semantic Similarity for Entity Linking with Convolutional Neural Networks", *Proceedings of NAACL/HLT*, pp. 1256–1261, 2016.

66. Sun, Y., L. Lin, N. Yang, Z. Ji and X. Wand, "Modeling Mention, Context and Entity with Neural Networks for Entity Disambiguation", *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1333–1339, 2015.

67. Fang, W., J. Zhang, D. Wang, Z. Chen and M. Li, "Entity Disambiguation by Knowledge and Text Jointly Embedding", *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1787–1796, 2016.

68. Ganea, O. and T. Hofmann, "Deep Joint Entity Disambiguation with Local Neu-

ral Attention", *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 1667–1676, 2017.

69. Yamada, I., H. Shindo, H. Takeda and Y. Takefuji, "Joint learning of the embedding of words and entities for named entity disambiguation", *Proceedings of Conference on Natural Language Learning (CoNLL)*, pp. 250–259, 2016.

70. Globerson, A., N. Lazic, S. Chakrabarti, A. S. M. Ringgaard and F. Pereira, "Collective Entity Resolution with Multi-Focal Attention", *Proceedings of ACL*, pp. 621–631, 2016.

71. Cucerzan, S., "Large-scale Named Entity Disambiguation based on Wikipedia Data", *Proceedings of EMNLP-CoNLL*, pp. 708–716, 2007.

72. Hoffart, J., M. Yosef, I. Bordino, H. Furstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, G. Weikum, Z. Guo and D. Barbosa, "Robust Disambiguation of Named Entities in Text", *Proceedings of of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 782–792, 2011.

73. Guo, Z. and D. Barbosa, "Robust Named Entity Disambiguation with Random Walks", *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 1–28, 2016.

74. Hachey, B., W. Radford and J. Curran, "Graph-Based Named Entity Linking with Wikipedia", *Proceedings of Web Information Systems Engineering (WISE)*, pp. 213–226, 2011.

75. Shen, W., J. Wang, P. Luo and M. Wang, "LIEGE: Link entities in web lists with knowledge base", *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pp. 1424–1432, 2012.

76. Pappu, A., R. Blanco, Y. Mehdad, A. Stent and K. Adani, "Lightweight Multilingual Entity Extraction and Linking", *Proceedings of Web Search and Data*

Mining (WSDM), pp. 365–374, 2017.

77. Austin, S., R. Schwartz and P. Placeway, "The forward-backward search algorithm", *Proceedings of IEEE ICASSP*, pp. 697–700, 1991.

78. Phan, M., A. Sun, Y. Tay, J. Han and C. Li, "Neupl: Attention-based Semantic Matching and Pair-linking for Entity Disambiguation", *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, pp. 1667–1676, 2017.

79. Yaming, S., Z. Jia, L. Lina, X. Wanga and D. Tang, "Entity disambiguation with memory network", *Neurocomputing*, Vol. 275, pp. 2367–2373, 2018.

80. Weston, J., S. Chopra and A. Bordes, "Memory networks", *arXiv preprint arXiv:1410.3916*, 2014.

81. Hirschman, L. and N. Chinchor, "MUC-7 Coreference Task Definition", *Proceedings of Message Understanding Conference*, pp. 1–18, 1997.

82. Miller, G., "WordNet: A Lexical Database for English", *Communications of the ACM*, Vol. 38(11), pp. 39–41, 1995.

83. Onoe, Y. and G. Durrett, "Fine-Grained Entity Typing for Domain Independent Entity Linking", *arXiv preprint arXiv 1909.05780*, 2019.

84. Choi, E., O. Levy, Y. Choi and L. Zettlemoyer, "Ultra-Fine Entity Typing", *Proceedings of the Association for Computational Linguistics (ACL). Association for Computational Linguistics*, pp. 87–96, 2018.

85. Neelakantan, A. and M. Chang, "Inferring missing entity type instances for knowledge base completion: New dataset and methods", *Proceedings of the 2015 Conference of the NAACL-HLT: Human Language Technologies*, pp. 515–525, 2006.

86. Singh, A., A. Yadav and A. Rana, "K-means with Three different Distance Metrics", *International Journal of Computer Applications*, pp. 1424–1432, 2012.

87. Rijsbergen, V., "Information Retrieval", *Journal of the American Society for Information Science*, Vol. 30(6), pp. 374–375, 1979.

88. Pasca, M., "Acquisition of Categorized Named Entities for Web Search", *Proceedings of Conference on Information and Knowledge Management*, pp. 137–145, 2004.

89. Teffera, H., *Automatic Construction of Labeled Clusters of Named Entities for Information Retrieval*, M.S. Thesis, Universitat Des Saarlendes, 2010.

90. Brown, P., V. Pietra, P. deSouza, J. Lai and R. Mercer, "Class-based N-gram Models of Natural Language", *Computational Linguistics*, Vol. 18(4), pp. 467–479, 1992.

91. Rabiner, L. and B. Juang, "An introduction to hidden Markov models", *IEEE ASSP Magazine*, Vol. 4-15, pp. 44–53, 1986.

92. Forney, J. G. D., "The viterbi algorithm", *Proceedings of the IEEE*, Vol. 61(3), pp. 268–278, 1973.

93. "*OpenOSF Website*", `http://www.openfst.org`, 2020, accessed in July 2020.

94. Jaynes, E., "Information Theory and Statistical Mechanics", *Phys. Rev. 106*, pp. 620–630, 1957.

95. Shannon, C., "The Mathematical Theory of Communication", *Bell System Technical Journal*, Vol. 27, pp. 379–423, 1948.

96. "*MaxEnt Tool Download Web Page*", `https://github.com/lzhang10/maxent`, 2020, accessed in July 2020.

97. Salton, G., A. Wong and C. Yang, "A vector space model for automatic indexing", *Communications of the ACM*, Vol. 18(11), pp. 613–620, 1975.

98. Katz, S., "Estimation of probabilities from sparse data for the language model component of a speech recognizer", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 35(3), pp. 400–401, 1987.

99. Bengio, Y., R. Ducharme, P. Vincent and C. Janvin, "A neural probabilistic language model", *The Journal of Machine Learning Research*, Vol. 3, pp. 1137–1155, 2003.

100. Collobert, R. and J. Weston, "A unified architecture for natural language processing: Deepneural networks with multitask learning", *Proceedings of International Conference on Machine Learning*, pp. 160–167, 2008.

101. Mikolov, T., I. Sutskever, K. Chen, G. Corrado and J. Dean, "Distributed Representations of Words and Phrases and Their Compositionality", *Proceedings of Neural Information Processing Systems (NIPS)*, pp. 3111–3119, 2013.

102. Rumelhart, D., G. Hinton and R. Williams, "Learning representations by back-propagating errors", *Nature*, Vol. 323(6088), pp. 533–536, 1986.

103. Pennington, J., R. Socher and C. Manning, "GloVe: Global Vectors for Word Representation", *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

104. Bojanowski, P., E. Grave, A. Joulin and T. Mikolov, "Enriching Word Vectors with Subword Information", *Transactions of the Association for Computational Linguistics (TACL)*, Vol. 5, pp. 135–146, 2017.

105. Firth, J., "A synopsis of linguistic theory 1930-1955", *In Studies in Linguistic Analysis*, pp. 1–32, 1957.

106. Levy, O. and Y. Goldberg, "Dependency-based Word Embeddings", *Proceedings of the 52nd Annual Meeting of Association for Computational Linguistics (ACL)*, pp. 302–308, 2014.

107. "*Brown Cluster Tool Download Web Page*", `https://github.com/percyliang/brown-cluster`, 2020, accessed in July 2020.

108. Bengio, Y., P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", *IEEE Transactions on Neural Networks*, Vol. 5(2), pp. 157–166, 1994.

109. Hochreiter, S. and J. Schmidhuber, "Long short-term memory", *Neural Computation*, Vol. 9(8), pp. 1735–1780, 1997.

110. Graves, A., N. Jaitly and A. Mohamed, "Hybrid speech recognition with deep bidirectional LSTM", *Automatic Speech Recognition and Understanding (ASRU) Workshop at IEEE*, pp. 273–278, 2013.

111. Billal, B., A. Fonseca and F. Sadat, "Named Entity Recognition and Hashtag Decomposition to Improve the Classification of Tweets", *Proceedings of the 2nd Workshop on Noisy User-generated Text*, pp. 64–73, 2016.

112. Bansal, P., R. Bansal and V. Varma, "Towards semantic retrieval of hashtags in microblogs", *Proceedings of the 24th International Conference on World Wide Web Companion*, pp. 7–8, 2015.

113. Qadir, A. and E. Riloff, "Learning emotion indicators from tweets: Hashtags, hashtag patterns, and phrases", *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1203–1209, 2014.

114. Stilo, G. and P. Velardi, "Hashtag sense clustering based on temporal similarity", *Computational Linguistics*, pp. 181–200, 2017.

115. Chen, T., X. He and M. Kan, "Context-aware image tweet model-ling and recommendation", *Proceedings of the 2016 ACM on Multi-media Conference*, pp. 1018–1027, 2016.

116. Bansal, P., R. Bansal and V. Varma, "Towards deep semantic analysis of hash-

tags", *Proceedings of the 37th European Conference on Information Retrieval*, pp. 453–464, 2015.

117. Srinivasan, S., S. Bhattacharya and R. Chakraborty, "Segmenting web-domains and hashtags using length specific models", *Proceedings of the 21st ACM International Conference on Information and Knowledge*, pp. 1113–1122, 2012.

118. Berardi, G., A. Esuli, D. Marcheggiani and F. Sebastian, "Isti@trec microblog track 2011: Exploring the use of hashtag segmentation and text quality ranking.", *Proceedings of Twentieth Text REtrieval Conference*, 2011.

119. Maynard, D. and M. Greenwood, "Who cares about sarcastic tweets? investigating the impact of sarcasm on sentiment analysis", *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC)*, pp. 4238–4243, 2014.

120. Cunningham, H., D. Maynard, K. Bontcheva and V. Tablan, "Gate: an architecture for development of robust hlt applications", *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pp. 168–175, 2002.

121. Declerck, T. and P. Lendvai, "Processing and normalizing hashtags", *Proceedings of Recent Advances in Natural Language Processing (RANLP)*, pp. 104–109, 2015.

122. Owoputi, O., B. O'Connor, C. Dyer, K. Gimpel and N. Schneider, *Part-of-speech tagging for Twitter: Word clusters and other advances*, Tech. Rep. CMU-ML-12-107, Carnegie Mellon University, Machine Learning Department, Pittsburgh, Pennsylvania, USA, 2012.

123. Stanford, "*Twitter Sentiment Analysis Data Set Web Page*", `http://help.sentiment140.com/for-students/`, 2020, accessed in July 2020.

124. "*SNAP Data Set Web Page*", `http://snap.stanford.edu/data/twitter7.html`, 2020, accessed in July 2020.

125. Nielsen, F. A., "*Afinn Sentiment Analysis Tool Web Page*", `https://github.com/fnielsen/afinn`, 2020, accessed in July 2020.

126. Nielsen, F., "A new ANEW: Evaluation of a word list for sentiment analysis in microblogs", *Proceedings of the ESWC2011 Workshop on Making Sense of Microposts: Big things come in small packages*, pp. 93–98, 2011.

127. Kong, L., N. S. S. Swayamdipta, A. Bhatia, C. Dye and N. Smith, "A dependency parser for tweets", *Proceedings of Empirical Methods on Natural Language Processing (EMNLP)*, pp. 1001–1012, 2014.

128. Yarowsky, D., "Unsupervised Word Sense Disambiguation Rivaling Supervised Methods", *Proceedings of 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 189–196, 1995.

129. Li, C., A. Sun, J. Weng and Q. He, "Tweet Segmentation and Its Application to Named Entity Recognition", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, pp. 558–570, 2015.

130. Wikipedia, "*Wikipedia Washington Disambiguation Web Page*", `https://simple.wikipedia.org/wiki/Washington_(disambiguation)`, 2020, accessed in July 2020.

131. Mihalcea, R. and A. Csomai, "Wikify! Linking Documents to Encyclopedic Knowledge", *Proceedings of Conference on Information and Knowledge Management (CIKM)*, pp. 233–242, 2007.

132. Kulkarni, S., A. Singh, G. Ramakrishnan and S. Chakrabarti, "Collective Annotation of Wikipedia Entities in Web Text", *Proceedings of the ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 457–466, 2009.

133. Raiman, J. and O. Raiman, "DeepType: Multilingual Entity Linking by Neural Type System Evolution", *arXiv preprint arXiv:1802.01021*, 2018.

134. Murty, S., P. Verga, L. Vilnis, I. Radovanovic and A. McCallum, "Hierarchical Losses and New Resources for Fine-grained Entity Typing and Linking", *Proceedings of the ACL*, Vol. 1, pp. 97–109, 2018.

135. Ratinov, L., D. Roth, D. Downey and M. Anderson, "Local and Global Algorithms for Disambiguation to Wikipedia", *Proceedings of ACL-HLT*, pp. 19–24, 2011.

136. Han, X., L. Sun and J. Zhao, "Collective Entity Linking in Web Text: A Graph-based Method", *Proceedings of the 34th International ACM SIGIR*, pp. 765–774, 2011.

137. Pershina, M., Y. He and R. Grishman, "Personalized Page Rank for Named Entity Disambiguation", *Proceedings of North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*, pp. 238–243, 2015.

138. Ganea, O., M. Ganea, A. Lucchi, C. Eickhoff and T. Hofmann, "Probabilistic Bag-of-hyperlinks Model for Entity Linking", *Proceedings of the 25th International Conference on World Wide Web*, pp. 927–938, 2016.

139. Ferragina, P. and U. Scaiella, "Tagme: On-the-fly Annotation of Short Text Fragments (by Wikipedia Entities)", *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pp. 1625–1628, 2010.

140. Phan, M., A. Sun, Y. Tay, J. Han and C. Li, "Pair-linking for Collective Entity Disambiguation: Two Could Be Better Than All", *Proceedings of Computing Research Repository (CoRR)*, Vol. 31(7), pp. 1383–1396, 2018.

141. Yang, Y., O. Irsoy and K. Rahman, "Collective entity disambiguation with structured gradient tree boosting", *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 777–786, 2018.

142. Han, X. and L. Sun, "An Entity-topic Model for Entity Linking", *Proceed-*

ings of Joint Conference on Empirical Methods in Natural Language Processing (EMNLP) and CoNLL, pp. 105–115, 2012.

143. Kataria, S., K. Kumar, R. Rastogi, P. Sen and S. Sengamedu, "Entity Disambiguation with Hierarchical Topic Models", *Preceedings of the ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 1037–1045, 2011.

144. Baroni, M., G. Dinu and G. Kruszewski, "Dont count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors", *Proceedings of ACM ACL*, Vol. 1, pp. 238–247, 2014.

145. Zwicklbauer, S., C. Seifert and M. Granitzer, "Robust and Collective Entity Disambiguation Through Semantic Embeddings", *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 425–434, 2016.

146. Zwicklbauer, S., C. Seifert and M. Granitzer, "DoSeR - A Knowledge-Base-Agnostic Framework for Entity Disambiguation Using Semantic Embeddings", *The Semantic Web. Latest Advances and New Domains, ESWC'16*, pp. 182–198, 2016.

147. Liu, C., F. Li, X. Sun and H. Han, "Attention-Based Joint Entity Linking with Entity Embedding", *Information*, Vol. 10(2), pp. 1–46, 2019.

148. Sil, A., G. Kundu, R. Florian and W. Hamza, "Neural Cross-Lingual Entity Linking", *Proceedings of Association for the Advancement of Artificial Intelligence (AAAI)*, pp. 5464–5472, 2018.

149. Yaghoobzadeh, Y. and H. Schutze, "Corpus-level fine-grained entity typing using contextual information", *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 715–725, 2015.

150. Xie, R., Z. Liu, J. Jia, H. Luan and M. Sun, "Representation learning of knowledge graphs with entity descriptions", *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 2659–2665, 2016.

151. Yaghoobzadeh, Y. and H. Schutze, "Multi-level Representations for Fine-Grained Typing of Knowledge Base Entities", *Proceedings of the 15th Conference of the European Chapter of the ACL*, Vol. 1, pp. 578–589, 2017.

152. Bollacker, K., C. Evans, P. Paritosh, T. Sturge and J. Taylor, "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge", *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 1247–1250, 2008.

153. Gupta, N., S. Singh and D. Roth, "Entity Linking via Joint Encoding of Types, Descriptions, and Context", *Proceedings of EMNLP*, pp. 2681–2690, 2017.

154. Steinley, D., "K-means clustering: A half-century synthesis", *The British Journal of Mathematical and Statistical Psychology*, Vol. 59, pp. 1–34, 2006.

155. Ester, M., H. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise", *AAAI Press*, pp. 226–231, 1996.

156. Jin, X. and J. Han, "Expectation Maximization Clustering", C. Sammut and G. Webb (Editors), *Encyclopedia of Machine Learning*, Springer, Boston, MA, 2011.

157. Pereira, F., N. Tishby and L. Lee, "Distributional Clustering of English Words", *Proceedings of the 31st Annual Meeting of the ACL*, pp. 183–190, 1993.

158. Kneser, R. and H. Ney, "Improved Clustering Techniques for Class-based Statistical Language Modeling", *Proceedings of Eurospeech*, Vol. 2, pp. 973–976, 1993.

159. Slonim, N. and N. Tishby, "The Power of Word Clusters for Text Classification",

European Colloquium on Information Retrieval Research, pp. 191–200, 2001.

160. Clark, A., "Combining Distributional and Morphological Information for Part of Speech Induction", *Proceedings of European Chapter of the Association for Computational Linguistics (EACL)*, pp. 59–66, 2003.

161. Ren, X., A. El-Kishky, C. Wang, F. Tao, C. Voss, H. Ji and J. Han, "Clustype: Effective Entity Recognition and Typing by Relation Phrase-based Clustering", *Proceedings of Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 995–1004, 2015.

162. Huang, L., J. May, X. Pan and H. Ji, "Building a fine-grained entity typing system overnight for a new x (x= language, domain, genre)", *arXiv preprint arXiv:1603.03112*, 2016.

163. Cardie, C. and K. Wagstaff, "Noun phrase coreference as clustering", *Proceedings of the 1999 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pp. 82–89, 1999.

164. Ratford, W., B. Hachey, M. Honnibal, J. Nothman and J. Curran, "Naive but effective NIL clustering baselines - CMCRC at TAC 2011", *Proceedings of TAC*, 2011.

165. Singh, S., A. Subramanya, F. Pereira and A. McCallum, "Large-scale cross-document coreference using distributed inference and hierarchical models", *Proceedings of the Association for Computational Linguistics: Human Language Technologies*, Vol. 1, pp. 793–803, 2011.

166. Wick, M., S. Singh and A. McCallum, "A Discriminative Hierarchical Model for Fast Coreference at Large Scale", *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pp. 379–388, 2012.

167. Ngomo, A., M. Roder and R. Usbeck, "Cross-document Coreference Resolution

using Latent Features", *Proceedings of International Conference on Linked Data for Information Extraction (LD4IE)*, pp. 33–44, 2014.

168. Singh, S., M. Wick and A. McCallum, "Distantly Labeling Data for Large Scale Cross-Document Coreference", *CoRR, abs/1005.4298*, 2010.

169. Beheshti, S., B. Benatallah, S. Venugopal, S. Ryu, H. Motahari-Nezhad and W. Wang, "A Systematic Review and Comparative Analysis of Cross-document Coreference Resolution Methods and Tools", *Computing*, Vol. 99(4), pp. 313–349, 2017.

170. Monahan, S., J. Lehmann, T. Nyberg, J. Plymale and A. Jung, "Cross-lingual cross-document coreference with entity linking", *TAC 2011 Workshop*, pp. 14–15, 2011.

171. Dutta, S. and G. Weikum, "A Joint Model for Cross-Document Co-Reference Resolution and Entity Linking", *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 846–856, 2015.

172. Hendrickx, I. and W. Daelemans, "Adding Semantic Information: Unsupervised Clusters for Coreference Resolution", *Workshop notes on Machine Learning for Natural Language Processing*, pp. 45–48, 2007.

173. Steinbach, M., G. Karypis and V. Kumar, "A Comparison of Document Clustering Techniques", *Proceedings of Workshop on Text Mining in Knowledge Discovery and Data Mining (KDD)*, Vol. 16, pp. 645–678, 2000.

174. DBPedia, "*DBPedia Download Web Page*", `https://wiki.dbpedia.org/downloads-2016-10`, 2020, accessed in July 2020.

175. Auer, S., C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, "Dbpedia: A Nucleus for a Web of Open Data", *Proceedings of the 6th International Semantic Web Conference (ISWC)*, pp. 722–735, 2007.

176. Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research*, Vol. 15, pp. 1929–1958, 2012.

177. Hachey, B., W. Radford, J. Nothman, M. Honnibal and J. Curran, "Evaluating entity linking with Wikipedia", *Artificial Intelligence*, Vol. 194, pp. 130–150, 2013.

178. Apache, "*Apache Lucene Web Page*", `http://lucene.apache.org/`, 2020, accessed in July 2020.

179. Ling, X., S. Singh and D. Weld, "Design Challenges for Entity Linking", *Transactions of the Association for Computational Linguistics*, Vol. 3, pp. 315–328, 2015.

180. Manning, C., M. Surdeanu, J. Bauer, J. Finkel, S. Bethard and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit", *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics (ACL): System Demonstrations*, pp. 55–60, 2014.

181. Milne, D. and I. Witten, "Learning to Link with Wikipedia", *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, pp. 509–518, 2008.

182. Hoffart, J., S. Seufert, D. Nguyen, M. Theobald and G. Weikum, "Kore: Keyphrase Overlap Relatedness for Entity Disambiguation", *Proceedings of the 21st ACM international conference on Information and Knowledge Management*, pp. 545–554, 2012.

183. Roder, M., R. Usbeck, S. Hellmann, D. Gerber and A. Both, "N3 - A Collection of Datasets for Named Entity Recognition and Disambiguation in the NLP Interchange Format", *Proceedings of Language Resources and Evaluation Conference (LREC)*, pp. 3529–3533, 2014.

184. Goldhahn, D., T. Eckart and U. Quasthoff, "Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages", *Proceedings of Language Resources and Evaluation Conference (LREC)*, pp. 759–765, 2012.

185. Devlin, J., M. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding", *Proceedings of North American Association for Computational Linguistics (NAACL)*, pp. 4171–4186, 2019.

186. Peng, X., I. W. Tsang, J. T. Zhou, and H. Zhu, "k-meansNet: When k-means Meets Differentiable Programming", *arXiv preprint arXiv:1808.07292*, 2018.

187. AT&T, "*FSM Library Web Page*", `https://www3.cs.stonybrook.edu/~algorith/implement/fsm/distrib/`, 2020, accessed in July 2020.

188. SRI, "*SRILM Download Web Page*", `http://www.speech.sri.com/projects/srilm/download.html`, 2020, accessed in July 2020.

189. Celebi, A., "*xDB Github Page*", `https://github.com/ardax/xDB`, 2020, accessed in July 2020.

190. Github, "*Mongo PHP Driver Github Page*", `https://github.com/mongodb/mongo-php-driver`, 2020, accessed in July 2020.

191. Maddela, M., W. Xu1 and D. Preotiuc-Pietro, "Multi-task Pairwise Neural Ranking for Hashtag Segmentation", *Proceedings of ACL*, pp. 2538–2549, 2019.

192. Wikipedia, "*Wikipedia Languages used on the Internet Page*", `https://wikipedia.org/wiki/Languages_used_on_the_Internet`, 2020, accessed in July 2020.

193. Wikipedia, "*Wikipedia List of endangered languages in Africa*",

https://wikipedia.org/wiki/List_of_endangered_languages_in_Africa, 2020, accessed in July 2020.

194. Lodhi, H., C. Saunders, J. Shawe-Taylor, N. Cristianini and C. Watkins, "Text classification using string kernels", *Journal of Machine Learning Research*, pp. 419–444, 2002.

195. Bunescu, R. C. and R. J. Mooney, "Subsequence kernels for relation extraction", *Neural Information Processing Systems (NIPS)*, pp. 171–178, 2005.