Keyword Extraction by Deep Learning
Mid Term Report

Yi Cheng(yicheng1), Anoop Hallur(ahallur), Xiaoqiu Huang(xiaoqiuh)

November 5, 2014

1 Introduction

As the goal of our project, we plan to do keyword extraction using deep learning. We are aiming to improve the performance of keyword extraction by deep learning techniques as compared to other techniques used previously. We believe that Deep learning techniques can improve the performance because when it has been applied to other similar tasks, a significant improvement has been observed. For example , Google speech recognizer uses Deep Neural Networks and accuracy is extremely high[?].The area of applying deep learning to keyword extraction has not been explored much, hence we are trying to apply it and see how it performs.

2 Related Work

Before diving into the details of our model and algorithm, we want to summarize the related work that has been done in the field of Keyword Extractions, how they can be applied to our project and how deep learning can be applied to solve this particular problem and our thoughts on why deep learning should give better performance compared to other algorithms.

2.1 Baseline Algorithms of Keyword extraction

A survey was done by Lott.B[?], and he summarizes that whenever he have a large corpus of data already available(as in our case), TF-IDF is the most accurate algorithms of the existing ones.

In TF-IDF model, we assign weight to each term in the model, and we choose the top'n' weighted words as the keywords for the text. The weight of each word is computed by taking into account the Term Frequency(TF) and Inverse Domain Frequency(IDF), where Term Frequency indicates how significant is the term to a specific document, and IDF takes into account how common the word is in the domain. We have implemented the benchmarks for this algorithms on our dataset.

The other algorithm proposed by Lott.B was the one using Lexical Chain approach. In this approach, the extract features from the document, first by constructing a lexical tree of the nouns in the document. They extract the nouns and each noun word is a possible tag for the document. The lexical chain is constructed using semantic features of the words such as other synonym, hyperonyms and morphynyms of a noun. From the lexical tree, they extract four features, and three features from other positional aspects of the nouns. The features are used to train a C4.5 decision tree and this decision tree is used to make predictions of the nouns in the document from the test data. We have implemented this particular algorithm for our data set and analyzed the results.

Other algorithms presently being used are some variant of TF-IDF with domain specific modifications. For example , one technique uses a Bayes classifier with TF-IDF to compute the weights and extract the keywords. There are techniques which are to be used when we don't have a corpus of text available to us. Frequency based single Document Keyword Extraction is one such technique. This techniques computes word weights by measuring the frequency of text occurrence within two punctuation marks in the text. Since we have text corpus available, we did not want to be compared against these classes of algorithms.

2.2 Deep Learning Intuition

3 Model Description

3.1 Neural language model

In traditional NLP model, the input of the classifier is just one-hot features or TF-IDF features of text. However, this model suffers from the curse of dimensionality. When the number of documents in the corpus becomes large, the matrix of features will become sparse. Therefore, Bengio et al.[1] proposed a new neural language model which generates distributed representations for each word in the corpus. That means each word in the documents has a fixed length of feature vector. With the feature vectors of each term, we can measure the similarity between different words or generate the vector representation of each document.

In our model, each word will be first transformed to word vector by the tool named word2vec[2]. The tool will first train the neural language model on our data set and then generate the word vector of each term.

3.2 Recursive Autoencoder

Autoencoder is a neural network model that learns the function $f(x) = x$ in order to learn the reduced representation of the input text. As we can see in the figure 1, autoencoder is only a three-layer neural network. We want to utilize the hidden layer to generate the similar vector as the input vector. And after traning the model, the hidden layer can be treated as the condensed feature of the input vectors. Therefore, the training objective is to minimize the distance which is known as reconstruction error between the input and the output of NN. When applying such structure into our model, the input of the autoencoder is word vectors of two terms. After traning the model, the vector of hidden layer can be seen as the condensed feature of these two words.

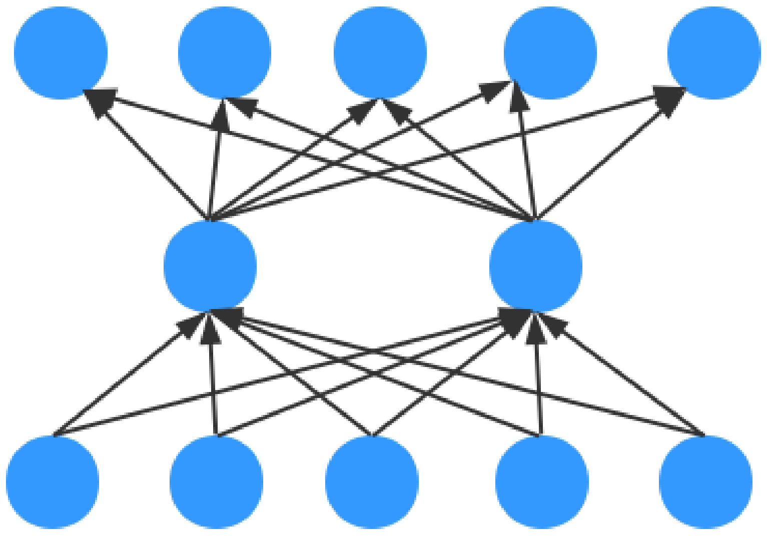


Figure 1: Autoencoder

However, only using the structure of autoencoder may not effectively model the text. If we set the input of autoencoder as the vectors of all terms in one document and try to generate the condensed features of the document, there will be so many parameters to optimize and the structure of the text is not utilized. In order to solve the problem, the structure of recursive autoencoder was proposed to model the hierarchical structure. Figure 2 shows the structure of recursive autoencoder.

In each layer, several terms are combined to generate the condensed features which will be utilized as the input of the next layer. At the final layer of recursive autoencoder, the vector of hidden layer will be the vector representation of the whole documents.

One of the key problem in such structure is how to combine different terms. There are mainly two ways. One way is to adopt the grammer tree which generated in advance. Each word will be combined in the same way as they are merge in the grammer tree. Another way is to utilize the greedy algorithm. At each layer, each pair of the adjacent words will be first combined and output reconstruction error. Then the pair of words with smallest error will be first combined. Using this greedy algorithm, the vector of each document can also be generated. In the experimental part, the effectiveness of the two structured will be

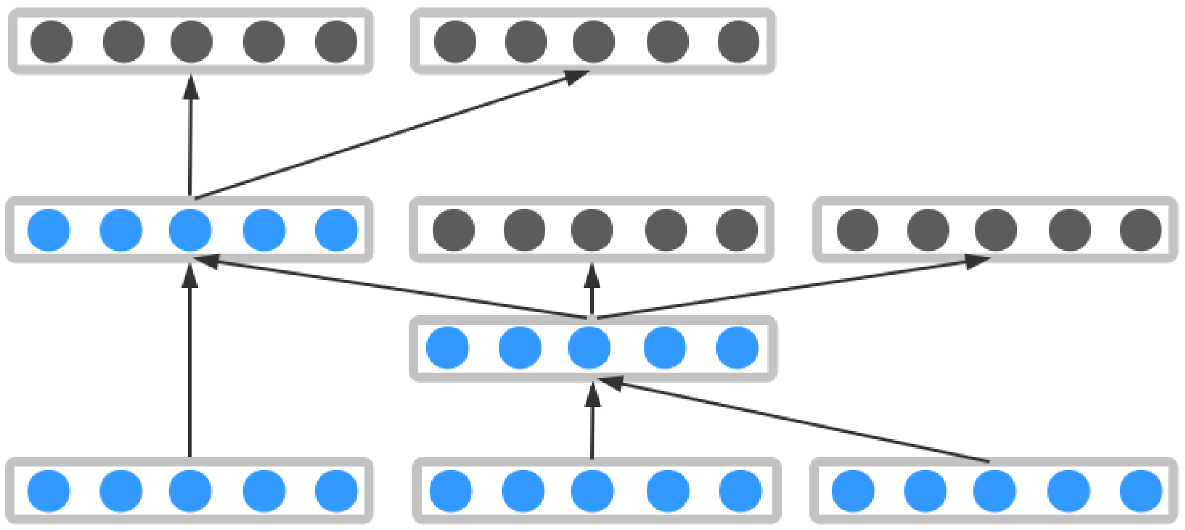


Figure 2: Recursive Autoencoder

compared.

3.3 Semi-supervised Recursive Autoencoder

In the traditional recursive autoencoder model, the network is trained only by the document itself. Therefore, it is the unsupervised model. In certain scenario, the label information of documents can be incorporated into the model and turn the model into a semi-supervised model.

One of the semi-superivised recursive autoencoder model has been proposed by socher et al.[3] in order to handle the issue of sentiment analysis. The structure of recursive autoencoder is presented as follows:

As we can see from the figure, the input of the model is also the distributed representation. After being processed by the hidden layer, the output layer reconstructs the input vector. In order to measure the performance of the representation, the reconstruction error is computed as the distance between the input vector and output vector. Also we can assign different terms with different weights. So

$$\begin{aligned} &E_{rec}([c_1; c_2]; \theta) \\ &= \frac{n_1}{n_1 + n_2}||c_1 - c_1'||^2 + \frac{n_2}{n_1 + n_2}||c_2 - c_2'||^2 \end{aligned} \tag{1}$$

, where n denote the number of words. In addition, c denotes the vector of input and c' present the vector of output.

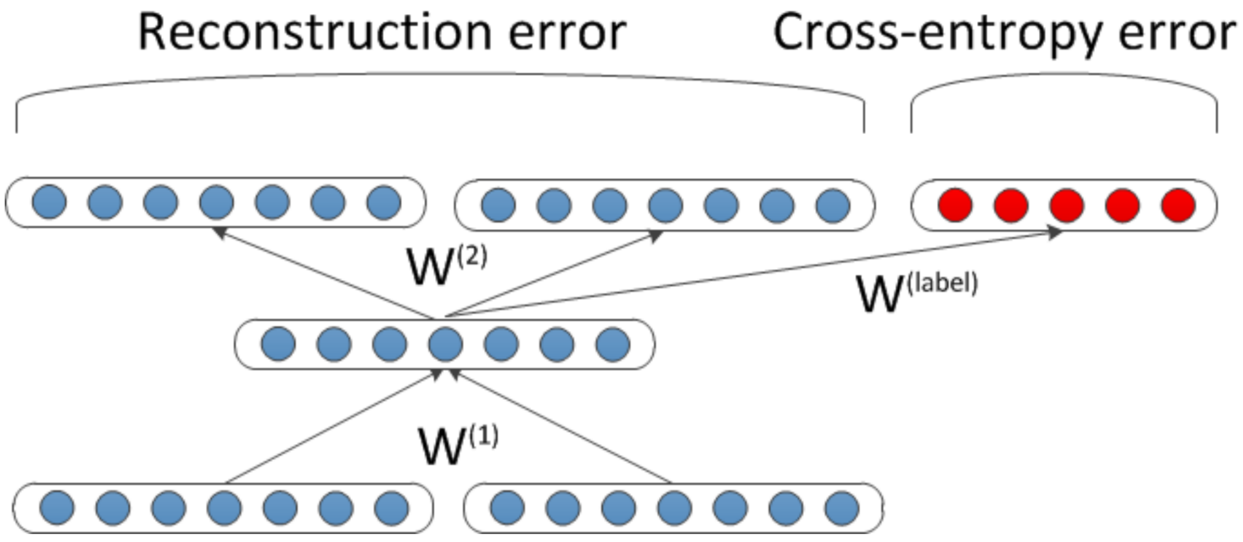


Figure 3: Semi-supervised Autoencoder

Also, in order to predict the sentiment of the sentence, an extra output unit is added to generate the sentiment label. And this generated label will be compared with the correct label. A softmax function is utilized in this scenario to generate the probability of each label. And author use the cross-entropy error to measure the correctness of the model.

$$\begin{aligned} & E_{cE}(p, t; \theta) \\ & = -\sum_{k=1}^{K} t_k \log d_k(p; \theta) \end{aligned} \tag{2}$$

, where t_k denotes the target label and d_k denotes the probability of each label.

After combining the two process, we can compute the reconstruction error and cross-entropy error. And the objective is to minimize the weighted sum of the two errors as follows:

$$\begin{aligned} & E([c_1; c_2]s, ps, t,) = \\ & = \alpha E_{rec}([c_1; c_2]_s; \theta) + (1-\alpha) E_{cE}(p_s, t; \theta). \end{aligned} \tag{3}$$

4 Experiment and Results

4.1 Baseline Algorithm Results

To evaluate the performance of our approach compared to existing techniques, we have used Precision and Recall rates as our benchmarking standards. We have formally defined these parameters as

$Precision = \frac{sizeof(A \bigcap B)}{sizeof(A)}$

Figure 4: Precision Comparison

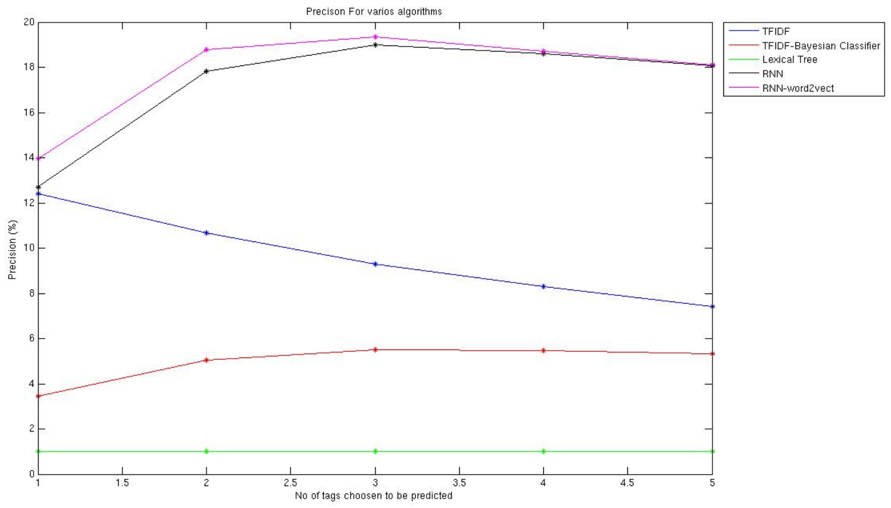
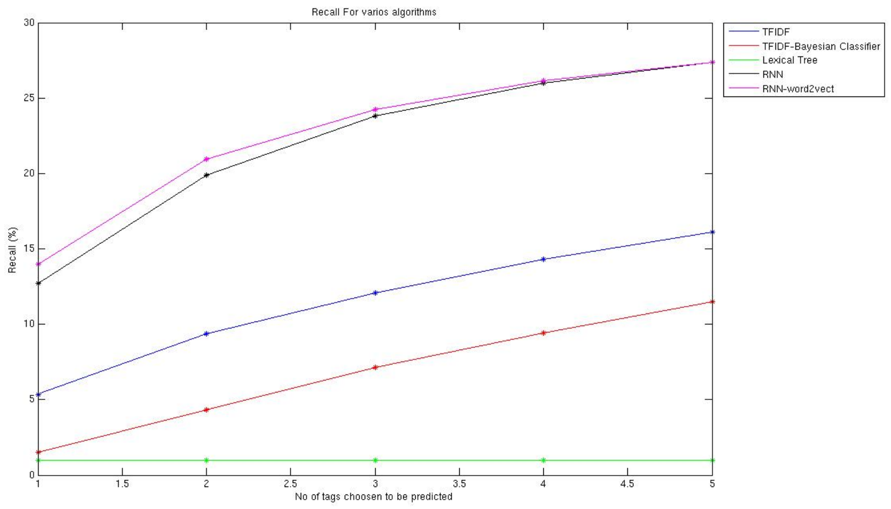


Figure 5: Precision Comparison



$Recall = \frac{sizeof(A \bigcap B)}{sizeof(A)}$
where A is the set of predicted tags and B is the set of actual tags.

We have implemented TF-IDF and Bayesian Classifier approach as described by the survey for Keyword Extraction on our data set. With plain vanilla TF-IDF, the average precision and recall rates were less than 10 %. However, after preprocessing the data by using a stemmer, the accuracy is a modest 15-20% on topics in our data set for as the best performer. We have used portland stemmer, available as part of open source NLTK(Natural Language Processing Toolkit) project. The Bayesian Classifier is similar to TF-IDF approach except that it takes into account the position of the word in text. The weight of each term is reduced as the log of the its position from the beginning of the sentence.

As part of survey paper{cite}, we found out that the existing standard for key word extraction uses Lexical Tree approach and hence we have summarized its performance on our data set as well. As per the lexical tree approach, the precision and recall were very poor, less than 1 % when tested against our data set. The reason for this can be attributed to the fact that our documents are very small posts with about a paragraph in them. Siber and McCoy[**?**] had implemented this feature for larger documents such as conference papers , essays etc and they would have probably had a document with more than 25 nouns, contrary to this, we only had 4-8 nouns per document on an average, hence we could not build a bigger graph for our documents.

4.2 Dataset

We had identified data from stackexchange(`https://archive.org/download/stackexchange/stackexchange_archive.torrent`) to be a suitable data set for our project. ‘ Of the 20 GB, available to us, for the feasibility of the experiment, we are using only 95 MB of data, which we have cleaned and formatted for our applications, on about 5 topics for our evaluation. Although the data set seems to be small compared with original one, we felt this was enough for testing purposes. Later we will try to **increase the data set** when our model is finished.

Here is an interesting question, why we choose several topic and not only one topic as small data set to test our model? In fact we did that, but found that it’s not a good way to do testing of the task like keyword extraction. For a single topic, some words will occur frequently, for example, in the topic of *apple*, the words like *osx* and *iphone* will occur frequently in most documents which according to *tf-idf*, they are not important words. But in fact they are the tags for many documents in this topic. And if we train the model on only a single dataset, then if the model only predict the top frequently tags, it can still gain good performance. That’s the reason we use several topics as dataset here.

The last step is to dived the data set into training dataset and testing dataset. We random sample 20% as the testing dataset and the rest are the training dataset.

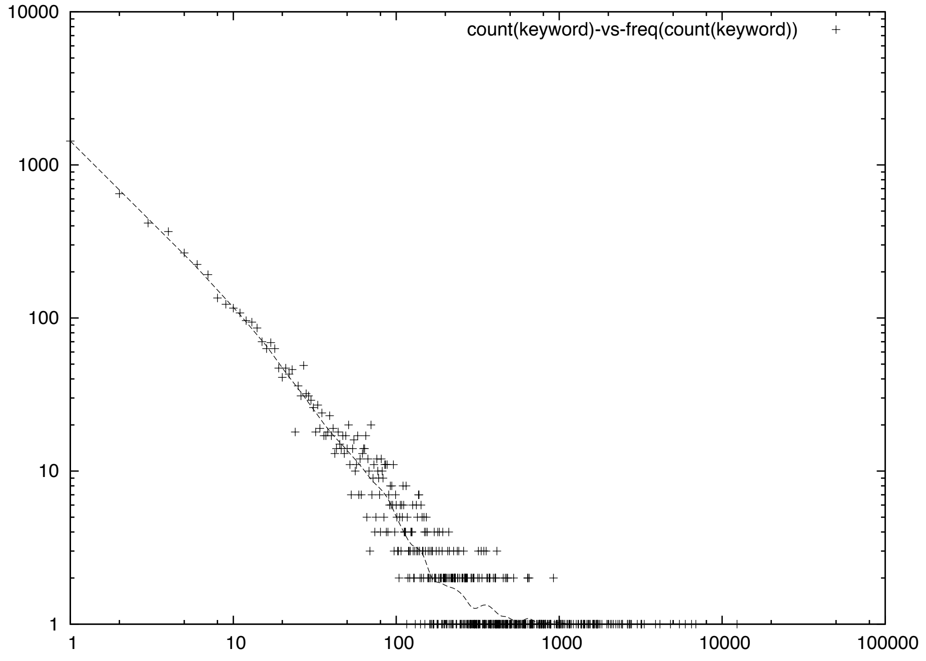


Figure 6: The distribution of tags follows Zipf's law. Here the x-axis is the count of the tag and the y-axis is the frequency of the count of tags. Drawn in log-log scale.

4.3 Our Model

Our work is based on the open source software [1]. For the midterm report, we use the tool to do some simple checking and we also modify some parts of the it to make use of the word vectors generated by *word2vec*[2], another tool developed by Google.

Before discussing about the experiment, we will introduce the method about training data preprocessing. We analyze the distribution of the tags in the dataset, and we find it follow Zipf's law from Figure 6. So we can't train the model with all dataset. And here, our model is a supervised model, and it's a multi-class classification task so it's better not to train the model on all the labels. Here, we just choose the top 200 big tags and for each tag we sample 100 documents.

Besides, our task is a little different with the classical classification task. For each document, we may give more than one tag, because it's common for a document having more than one tag. So when predicating, we generate the belief of each tag for each document and choose the top-n tags.

[1]https://github.com/sancha/jrae
[2]https://code.google.com/p/word2vec/

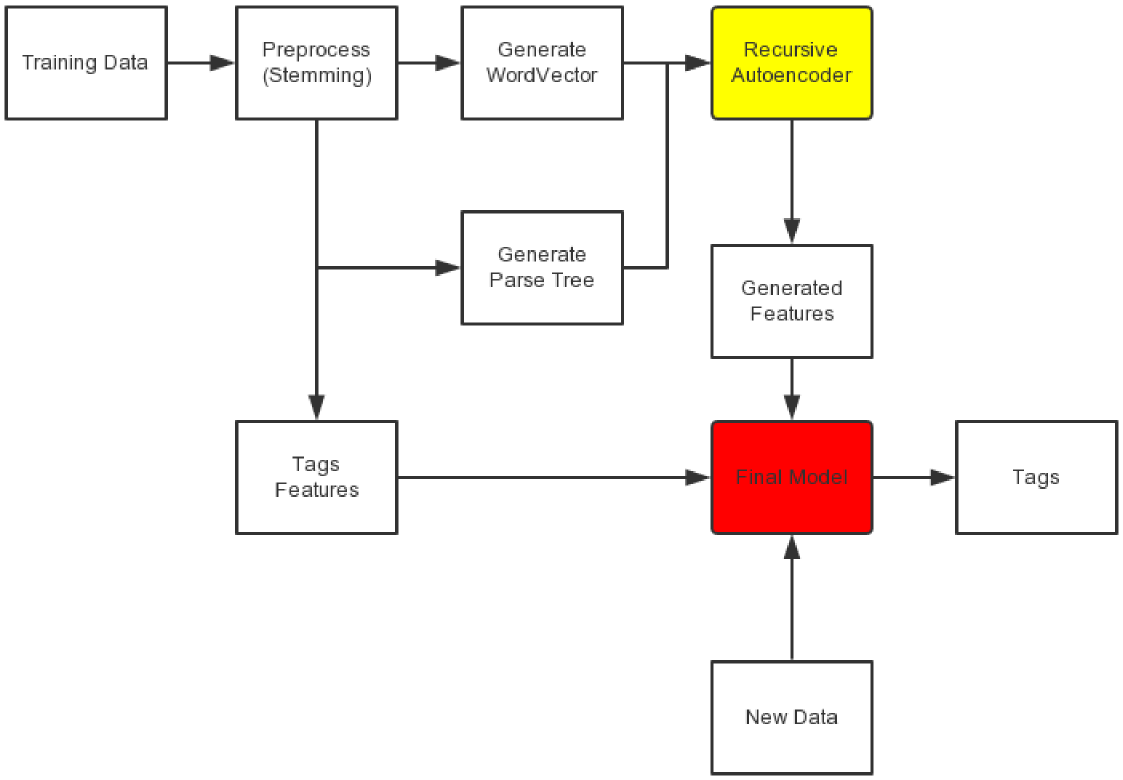


Figure 7: Our frame for keyword extraction. Here we have done some work on the yellow part: Recursive Autoencoder.

We do the testing on these settings: (i) original model (ii) pre-training the parameters of word vectors. (iii) using SVR instead of softmax. .

5 Pending Work

Now we have studied and implemented some baseline algorithms. We also the recursive autoencoder algorithm mentioned in the proposal. What's more we have done some experiment based on recursive autoencoder and also integrate *word2vect* with recursive autoencoder.

In Figure 7 is our big frame for the task. For the final report, we will add parse tree into the recursive autoencoder instead of using the greedy algorithm.

And for the final model:**TODO**

What's more we will also try to increase the size of the word vectors and the number of training data to observe the performance of our model.

References

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[3] Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 151–161. Association for Computational Linguistics, 2011.