Grammatical Error Correction as Multiclass Classification with Single Model

Zhongye Jia, Peilu Wang and Hai Zhao

MOE-Microsoft Key Laboratory for Intelligent Computing and Intelligent Systems,
Center for Brain-Like Computing and Machine Intelligence
Department of Computer Science and Engineering, Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{jia.zhongye,plwang1990}@gmail.com, zhaohai@cs.sjtu.edu.cn

Abstract

This paper describes our system in the shared task of CoNLL-2013. We illustrate that grammatical error detection and correction can be transformed into a multiclass classification task and implemented as a single-model system regardless of various error types with the aid of maximum entropy modeling. Our system achieves the F1 score of 17.13% on the standard test set.

1 Introduction and Task Description

Grammatical error correction is the task of automatically detecting and correcting erroneous word usage and ill-formed grammatical constructions in text (Dahlmeier et al., 2012). This task could be helpful for hundreds of millions of people around the world that are learning English as a second language. Although there have been much of work on grammatical error correction, the current approaches mainly focus on very limited error types and the result is far from satisfactory.

The CoNLL-2013 shared task, compared with the previous Help Our Own (HOO) tasks focusing on only determiner and preposition errors, considers a more comprehensive list of error types, including determiner, preposition, noun number, verb form, and subject-verb agreement errors. The evaluation metric used in CoNLL-2013 is Max-Matching (M$^2$) (Dahlmeier and Ng, 2012) precision, recall and F1 between the system edits and a manually created set of gold-standard edits. The corpus used in CoNLL-2013 is NUS Corpus of Learner English (NUCLE) of which the details are described in (Dahlmeier et al., 2013).

In this paper, we describe the system submission from the team 1 of Shanghai Jiao Tong University (SJT1). Grammatical error detection and correction problem is treated as multiclass classification task. Unlike previous works (Dahlmeier et al., 2012; Rozovskaya et al., 2012; Kochmar et al., 2012) that train a model upon each error type, we use one single model for all error types. Instead of the original error type, a more detailed version of error types is used as class labels. A rule based system generates labels from the golden edits utilizing an extended version of Levenshtein edit distance. We use maximum entropy (ME) model as classifier to obtain the error types and use rules to do the correction. Corrections are made using rules. Finally, the corrections are filtered using language model (LM).

2 System Architecture

Our system is a pipeline of grammatical error detection and correction. We treats grammatical error detection as a classification task. First all the tokens are relabeled according to the golden annotation and a sequence of modified version of error types is generated. This relabeling task is rule based using an extended version of Levenshtein edit distance which will be discussed in the following section. Then with the modified error types as the class labels, a classifier using ME model is trained. The grammatical error correction is also rule based, which is basically the reverse of the relabeling phase. The modified version of error types that we used is much more detailed than the original five types so that it enables us to use one rule to do the correction for each modified error type. After all, the corrections are filtered by LM, to remove those corrections that seem much worse than the original sentence.

As typical classification task, we have a training step and a test step. The training step consists three phases:

- Error types relabeling.
- Training data refinement.
- ME training.

The test step includes three phases:

- ME classification.
• Error correction according to labels.
• LM filtering.

2.1 Rebelling by Levenshtein Edit Distance with Inflection

In CoNLL-2013 there are 5 error types but they cannot be used directly as class labels, since they are too general for error correction. For example, the verb form error includes all verb form inflections such as converting a verb to its infinitive form, gerund form, past tense, past participle, passive voice and so on. Previous approaches (Dahlmeier et al., 2012; Rozovskaya et al., 2012; Kochmar et al., 2012) manually decompose each error types to more detailed ones. For example, in (Dahlmeier et al., 2012), the determinator error is decomposed into:

• replacement determiner (RD): \{ a → the \}
• missing determiner (MD): \{ e → a \}
• unwanted determiner (UD): \{ a → e \}

For limited error types such as merely determinatives error and preposition error in HOO 2012, manually decomposition may be sufficient. But for CoNLL-2013 with 5 error types including complicated verb inflection, an automatic method to decompose error types is needed. We present an extended version of Levenshtein edit distance to decompose error types into more detailed class labels and relabel the input with the labels generated.

The original Levenshtein edit distance has 4 edit types: unchange (I), addition (A), deletion (D) and substitution (S). We extend the “substitution” edit into two types of edits: inflection (T) and the original substitution (S). To judge whether two words can be inflected from each other, the extended algorithm needs lemmas as input. If the two words have the same lemma, they can be inflected from each other. The extended Levenshtein edit distance with inflection is shown in Algorithm 1. It takes the source tokens \( tok_{src} \), destination tokens \( tok_{dst} \) and their lemmas \( lem_{src}, lem_{dst} \) as input and returns the edits \( E \) and the parameters of edits \( P \). For example, for the golden edit \{ look → have been looking at \}, the edits \( E \) returned by \textbf{DISTANCE} will be \{A, A, I, T, A\}, and the parameters \( P \) of edits returned by \textbf{DISTANCE} will be \{have, been, looking, at\}.

Then with the output of \textbf{DISTANCE}, the labels can be generated by calculating the edits between original text and golden edits. For those tokens without errors, we directly assign a special label “\( \circ \)” to them. The tricky part of the relabeling algorithm is the problem of the edit “addition”, \( A \). A new token can only be added before or after an existing token. Thus for labels with addition, we must find some token that the label can be assigned to. That sort of token is defined as pivot. A pivot can be a token that is not changed in an edit, such as the “apple” in edit \{ apple → an apple \}, or some other types of edit such as the inflection of “look” to “looking” in edit \{ look → have been look-

\[\text{Algorithm 1 Levenshtein edit distance with inflection}\]

1: \textbf{function} \textit{DISTANCE}(\( tok_{src} \), \( tok_{dst} \), \( lem_{src} \), \( lem_{dst} \))
2: \quad (l_{src}, l_{dst}) \leftarrow (\text{len}(tok_{src}), \text{len}(tok_{dst}))
3: \quad D[0 \ldots l_{src}, 0 \ldots l_{dst}] \leftarrow 0
4: \quad B[0 \ldots l_{src}, 0 \ldots l_{dst}] \leftarrow (0, 0)
5: \quad E[0 \ldots l_{src}, 0 \ldots l_{dst}] \leftarrow \phi
6: \quad \text{for } i \leftarrow 1 \ldots l_{src} \text{ do}
7: \quad \quad D[i][0] \leftarrow i
8: \quad \quad B[i][0] \leftarrow (i - 1, 0)
9: \quad \quad E[i][0] \leftarrow D
10: \quad \text{end for}
11: \quad \text{for } j \leftarrow 1 \ldots l_{dst} \text{ do}
12: \quad \quad D[0][j] \leftarrow j
13: \quad \quad B[0][j] \leftarrow (0, j - 1)
14: \quad \quad E[0][j] \leftarrow A
15: \quad \text{end for}
16: \quad \text{for } i \leftarrow 1 \ldots l_{src}; j \leftarrow 1 \ldots l_{dst} \text{ do}
17: \quad \quad \text{if } tok_{src}[i - 1] = tok_{dst}[j - 1] \text{ then}
18: \quad \quad \quad D[i][j] \leftarrow D[i - 1][j - 1]
19: \quad \quad \quad B[i][j] \leftarrow (i - 1, j - 1)
20: \quad \quad \quad E[i][j] \leftarrow I
21: \quad \quad \else
22: \quad \quad \quad m \leftarrow \min(D[i - 1][j - 1], D[i - 1][j], D[i - 1][j - 1])
23: \quad \quad \quad \text{if } m = D[i - 1][j] \text{ then}
24: \quad \quad \quad \quad D[i][j] \leftarrow D[i - 1][j] + 1
25: \quad \quad \quad \quad B[i][j] \leftarrow (i - 1, j - 1)
26: \quad \quad \quad \quad E[i][j] \leftarrow D
27: \quad \quad \quad \text{else if } m = D[i][j - 1] \text{ then}
28: \quad \quad \quad \quad D[i][j] \leftarrow D[i][j - 1] + 1
29: \quad \quad \quad \quad B[i][j] \leftarrow (i - 1, j - 1)
30: \quad \quad \quad \quad E[i][j] \leftarrow A
31: \quad \quad \quad \end{end if}
32: \quad \quad \end{end if}
33: \quad \quad \end{end for}
34: \quad \end{end for}
35: \quad \text{while } i > 0 \text{ or } j > 0 \text{ do}
36: \quad \quad \text{insert } E[i][j] \text{ into head of } E
37: \quad \quad \text{insert tok}_{dst}[j - 1] \text{ into head of } P
38: \quad \quad (i, j) \leftarrow B[i][j]
39: \quad \quad \text{end while}
40: \quad \text{return } (E, P)
41: \textbf{end function}
ing at}. In the CoNLL-2013 task, the addition edits are mostly adding articles or determinaters, so when generating the label, adding before the pivot is preferred and only those trailing edits are added after the last pivot. At last, the label is normalized to upper case.

The BNF syntax of labels is defined in Figure 1. The non-terminal ⟨inflection-rules⟩ can be substituted by terminals of inflection rules that are used for correcting the error types of noun number, verb form, and subject-verb agreement errors. All the inflection rules are listed in Table 1. The ⟨stop-word⟩ can be substituted by terminals of stop words which contains all articles, determinaters and prepositions for error types of determiner and preposition, and a small set of other common stop words. All the stop words are listed in Table 2.

\[
\begin{align*}
\langle label \rangle & ::= \langle simple-label \rangle \mid \langle compound-label \rangle \\
\langle simple-label \rangle & ::= \langle pivot \rangle \mid \langle add-before \rangle \mid \langle add-after \rangle \\
\langle compound-label \rangle & ::= \langle add-before \rangle \langle pivot \rangle \\
& \mid \langle pivot \rangle \langle add-after \rangle \\
& \mid \langle add-before \rangle \langle pivot \rangle \langle add-after \rangle \\
\langle pivot \rangle & ::= \langle inflection \rangle \mid \langle unchange \rangle \mid \langle substitution \rangle \\
& \mid \langle deletion \rangle \\
\langle add-before \rangle & ::= \langle stop-word \rangle \oplus \\
& \mid \langle stop-word \rangle \oplus \langle add-before \rangle \\
\langle add-after \rangle & ::= \oplus \langle stop-word \rangle \\
& \mid \oplus \langle stop-word \rangle \langle add-after \rangle \\
\langle substitution \rangle & ::= \langle stop-word \rangle \\
\langle inflection \rangle & ::= \langle inflection-rules \rangle \\
\langle unchange \rangle & ::= \emptyset \\
\langle deletion \rangle & ::= \emptyset
\end{align*}
\]

Figure 1: BNF syntax of label

Algorithm 2 is used to generate the label from the extended Levenshtein edits according to the syntax defined in Figure 1. It takes the original tokens, tokorig and golden edit tokens, tokgold in an annotation and their lemmas, lemorig, lemgold as input and returns the generated label L. For our previous example of edit \{looked \rightarrow have\ been\ looking\ at\}, the L returned by RELABEL is \{HAVE\ BEEN\ GERUND\ AT\}. Some other examples of relabeling are demonstrated in Table 3.

The correction step is done by rules according to the labels. The labels are parsed with a simple LL(1) parser and edits are made according to labels. A bit of extra work has to be taken to handle the upper/lower case problem. For additions and substitutions, the words added or substituted are normalized to lowercase. For inflections, original case of words are reserved. Then a bunch of regular expressions are applied to correct upper/lower case for sentence head.

| Catalog | Rules |
|---------|-------|
| Noun Number | LEMMA, NPLURAL |
| Verb Number | VSINGULAR, LEMMA |
| Verb Tense | LEMMA, GERUND, PAST, PART |

Table 1: Inflection rules

| Catalog | Words |
|---------|-------|
| Determiner | a an the my your his her its our their this that these those |
| Preposition | about along among around as at beside besides between by down during except for from in inside into of off on outside over through to toward towards under underneath until up upon with within without |
| Modal Verb | can could will shall should must may might |
| BE | be am is are was were been |
| HAVE | have has had |
| Other | many much |

Table 2: Stop words

| Tokens | Edit | Label |
|--------|------|-------|
| to reveal | \{to reveal\−−revealing\} | GERUND |
| a woman | \{a woman\−−women\} | NPLURAL |
| developing world | \{developing world\−−the developing world\} | THE |
| a | \{a\−−\} | | |
| in | \{in\−−\} | ON |
| apple | \{apple\−−an apple\} | AN |

Table 3: Examples of relabeling

### 2.2 Training Corpus Refinement

The corpus used to train the grammatical error recognition model is highly imbalanced. The original training corpus has 57,151 sentences and only 3,716 of them contain at least one grammatical error, and only 5,049 of the total 1,161K words are needed to be corrected. This characteristic makes it hard to get a well-performed machine learning model, since the samples to be recognized are so sparse and those large amount of correct data will severely affect the machine learning process as it is an optimization on the global training data. While developing our system, we found that only using sentences containing grammatical errors will lead to a notable improvement of the result.
Algorithm 2 Relabeling using the extended Levenshtein edit distance

1: function RELABEL (tokorig, tokgold, lemorig, lemgold)
2: (E, P) ← DISTANCE (tokorig, tokgold, lemorig, lemgold)
3: pivot ← number of edits in E that are not A
4: L ← “
5: while i < length of E do
6: if E[i] = A then
7: L ← L + label of edit E[i] with P[i]
8: i ← i + 1
9: else
10: l ← L + label of edit E[i] with P[i]
11: pivot ← pivot - 1
12: if pivot = 0 then
13: i ← i + 1
14: while i < length of E do
15: l ← l + ⊕ + P[i]
16: i ← i + 1
17: end while
18: end if
19: end while
20: if pivot = 0 then
21: push l into L
22: L ← “
23: end if
24: L ← upper case of L
25: return L
26: end function

Algorithm 3 Training Corpus Refine Algorithm

1: function CORPUSREFINE(sentence, c, n)
2: context ← φ
3: if sentence contains no error then
4: return φ
5: end if
6: build the syntax tree T of sentence
7: enode ← the node with error in T
8: e ← enode
9: while True do
10: pnode ← parent node of e in T
11: cnodes ← all the children nodes of pnode in T
12: if len(cnodes) > c then
13: context ← cnodes
14: break
15: end if
16: e ← pnode
17: end while
18: i ← the position of enode in context
19: l ← size of context
20: if i < n then
21: add (n-i) words before context at the head of context
22: end if
23: if l-i<n then
24: append (l-i) words after context in context
25: end if
26: return context
27: end function

Inspired by this phenomenon, we propose a method to refine the training corpus which will reduce the error sparsity of the training data and notably improve the recall rate.

The refined training corpus is composed of contexts containing grammatical errors. To keep the information which may have effects on the error identification and classification, those contexts are selected based on both syntax tree and n-gram, of which the process is shown in Algorithm 3. For a word with error, its syntax context of size $c$ is those words in the minimal subtree in the syntax tree with no less than $c$ leaf nodes; and its n-gram context of size $n$ is $n − 1$ words before and after itself. In the CORPUSREFINE algorithm, the input $c$ gives the size of syntax context and $n$ provides the size of the n-gram context. These two parameters decide the amount of information that may affect the recognition of errors. To obtain a context, given a sentence containing a grammatical error, we first get a minimum syntax tree whose number of terminal nodes exceed the $c$ threshold, then check whether the achieved context containing the error word’s n-gram, if not, add the n-gram to the context. At last the context is returned by CORPUSREFINE.

An example illustrating this process is presented in Figure 2. In this example, $n$ and $c$ are both set to 5, the minimal syntax tree and the context decided by it are colored in green. Since the syntax context in the green frame does not contain the error word’s 5-gram, therefore, the 5-gram context in the blue frame is added to the syntax context and the final achieved context of this sentence is "have to stop all works for the development".

2.3 LM Filter

All corrections are filtered using a large LM. Only those corrections that are not too much worse than the original sentences are accepted. Perplexity (PPL) of the n-gram is used to judge whether a sentence is good:

$$PPL = 2^{-\sum_{w \in \text{n-gram}} p(w) \log p(w)}.$$

We use $r_{PPL}$, the ratio between the PPL before and after correction, as a metric of information gain.

$$r_{PPL} = \frac{PPL_{\text{corrected}}}{PPL_{\text{original}}}.$$ 

Only those corrections with an $r_{PPL}$ lower than a certain threshold are accepted.
3 Features

The single model approach enables us only to optimize one feature set for all error type in the task, which can drastically reduce the computational cost in feature selection.

As many previous works have proposed various of features, we first collected features from different previous works including (Dahlmeier et al., 2012; Rozovskaya et al., 2012; HAN et al., 2006; Rozovskaya et al., 2011; Tetreault, Joel R and Chodorow, Martin, 2008). Then experiments with different features were built to test these features’ effectiveness and only those have positive contribution to the final performance were preserved. The features we used are presented in Table 4, where word\(_0\) is the word that we are generating features for, and word and POS is the word itself and it’s POS tag for various components. NPHead denotes the head of the minimum Noun Phrase (NP) in syntax tree. word\(_{NP-1}\) represents the word appearing before NP in the sentence. NC stands for noun compound and is composed of the last \(n\) words \((n \geq 2)\) in NP which are tagged as a noun. Verb feature is the word that is tagged as a verb whose direct object is the NP containing current word. Adj feature represents the first word in the NP whose POS is adjective. Prep feature denotes the preposition word if it immediately precedes the NP. DPHead is the parent of the current word in the dependency tree, and DPRel is the dependency relation with parent.

4 Experiments

4.1 Data Sets

The CoNLL-2013 training data consist of 1,397 articles together with gold-standard annotation. The documents are a subset of the NUS Corpus of Learner English (NUCLE) (Dahlmeier et al., 2013). The official test data consists of 50 new essays which are also from NUCLE. The first 25 essays were written in response to one prompt. The remaining 25 essays were written in response to a second prompt. One of the prompts had been used for the training data, while the other prompt is new. More details of the data set are described in (Ng et al., 2013).

We split the entire training corpus ALL by article. For our training step, we randomly pick 90% articles of ALL and build a training corpus TRAIN of 1,258 articles. The rest 10% of ALL with 139 articles are for developing corpus DEV.

For the final submission, we use the entire corpus ALL for relabeling and training.

Figure 2: Example of training corpus refinement
Table 4: Features for grammatical error recognition. The example sentence is: “That man stared at the old oak tree.” and the current word is “the”.

Table 5: Features Continued

Table 6: All labels after relabeling
4.2 Resources

We use the following NLP resources in our system. For relabeling and correction, perl module Lingua::EN::Inflect\(^1\) (Conway, 1998) is used for determining noun and verb number and Lingua::EN::VerbTense\(^2\) is used for determining verb tense. A revised and extended version of maximum entropy model\(^3\) is used for ME modeling. For lemmatization, the Stanford CoreNLP lemma annotator (Toutanova et al., 2003; Toutanova and Manning, 2000) is used. The language model is built by the SRILM toolkit (Stolcke and others, 2002). The corpus for building LM is the EuroParl corpus (Koehn, 2005). The English part of the German-English parallel corpus is actually used. We use such a corpus to build LM for the following reasons: 1. LM for grammatical error correction should be trained from corpus that itself is grammatically correct, and the EuroParl corpus has very good quality of writing; 2. the NUCLE corpus mainly contains essays on subjects such as environment, economics, society, politics and so on, which are in the same domain as those of the EuroParl corpus.

4.3 Relabeling the Corpus

There are some complicated edits in the annotations that can not be represented by our rules, for example substitution of non-stopwords such as \{human → people\} or \{are not short of → do not lack\}. The relabeling phase will ignore those ones thus it may not cover all the edits. After all, we get 174 labels after relabeling on the entire corpus as shown in Table 6. These labels are generated following the syntax defined in Figure 1 and terminals defined in Table 1 and Table 2. Directly applying these labels for correction receives an M\(^2\) score of Precission = 91.43%, Recall = 86.92% and F1 = 89.12%. If the non-stopwords non-inflection edits are included, of course the labels will cover all the golden annotations and directly applying labels for correction can receive a score up to almost F1 = 100%. But that will get nearly 1,000 labels which is too computationally expensive for a classification task. Cut out labels with very low frequency such as those exists only once reduces the training time, but does not give significant performance improvement, since it hurts the coverage of error detection. So we use all the labels for training.

4.4 LM Filter Threshold

To choose the threshold for \(r_{PPL}\), we run a series of tests on the DEV set with different thresholds. From our empirical observation on those right corrections and those wrong ones, we find the right ones seldomly have \(r_{PPL} > 2\), so we test thresholds ranging from 1.5 to 3. The curves are shown in Figure 3.

![Figure 3: Different thresholds of LM filters](image)

With higher threshold, more correction with lower information gain will be obtained. Thus the recall grows higher but the precision grows lower. We can observe a peak of F1 around 1.8 to 2.0, and the threshold chosen for final submission is 1.91.

4.5 Evaluation and Result

The evaluation is done by calculating the M\(^2\) precision, recall and F1 score between the system output and golden annotation. All the error types are evaluated jointly. Only one run of a team is permitted to be submitted. Table 7 shows our result on our DEV data set and the official test data set.

| Data Set   | Precision | Recall | F1   |
|------------|-----------|--------|------|
| DEV        | 16.03%    | 15.88% | 15.95%|
| Official   | 40.04%    | 10.89% | 17.13%|

Table 7: Evaluation Results

5 Conclusion

In this paper, we presented the system from team 1 of Shanghai Jiao Tong University that participated in the HOO 2012 shared task. Our system achieves an F1 score of 17.13% on the official test set based on gold-standard edits.

References

DM Conway. 1998. An algorithmic approach to english pluralization. In Proceedings of the Second Annual Perl Conference. C. Salzenberg. San Jose, CA, O’Reilly.

Daniel Dahlmeier and Hwee Tou Ng. 2012. Better Evaluation for Grammatical Error Correction. In Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 568–572, Montréal, Canada, June. Association for Computational Linguistics.

---

\(^1\)http://search.cpan.org/˜dconway/Lingua-EN-Inflect-1.89/
\(^2\)http://search.cpan.org/˜jjnapiork/Lingua-EN-VerbTense-3.003/
\(^3\)http://www.nactem.ac.uk/tsuruoka/maxent/
Daniel Dahlmeier, Hwee Tou Ng, and Eric Jun Feng Ng. 2012. NUS at the HOO 2012 Shared Task. In Proceedings of the Seventh Workshop on Building Educational Applications Using NLP, pages 216–224, Montréal, Canada, June. Association for Computational Linguistics.

Daniel Dahlmeier, Hwee Tou Ng, and Siew Mei Wu. 2013. Building a Large Annotated Corpus of Learner English: The NUS Corpus of Learner English. In Proceedings of the 8th Workshop on Innovative Use of NLP for Building Educational Applications, Atlanta, Georgia, USA.

NA-RAE HAN, MARTIN CHODOROW, and CLAUDIA LEACOCK. 2006. Detecting Errors in English Article Usage by Non-Native Speakers. Natural Language Engineering, 12:115–129, 5.

Ekaterina Kochmar, Øistein Andersen, and Ted Briscoe. 2012. HOO 2012 Error Recognition and Correction Shared Task: Cambridge University Submission Report. In Proceedings of the Seventh Workshop on Building Educational Applications Using NLP, pages 242–250, Montréal, Canada, June. Association for Computational Linguistics.

Philipp Koehn. 2005. Europarl: A Parallel Corpus For Statistical Machine Translation. In MT summit, volume 5.

Hwee Tou Ng, Siew Mei Wu, Yuanbin Wu, Christian Hadiwinoto, and Joel Tetreault. 2013. The conll-2013 shared task on grammatical error correction. In Proceedings of the Seventeenth Conference on Computational Natural Language Learning.

Alla Rozovskaya, Mark Sammons, Joshua Gioja, and Dan Roth. 2011. University of Illinois System in HOO Text Correction Shared Task. In Proceedings of the 13th European Workshop on Natural Language Generation, pages 263–266. Association for Computational Linguistics.

Alla Rozovskaya, Mark Sammons, and Dan Roth. 2012. The UI System in the HOO 2012 Shared Task on Error Correction. In Proceedings of the Seventh Workshop on Building Educational Applications Using NLP, pages 272–280, Montréal, Canada, June. Association for Computational Linguistics.

Andreas Stolcke et al. 2002. SRILM-An Extensible Language Modeling Toolkit. In Proceedings of the international conference on spoken language processing, volume 2, pages 901–904.

Tetreault, Joel R and Chodorow, Martin. 2008. The Ups and Downs of Preposition Error Detection in ESL Writing. In Proceedings of the 22nd International Conference on Computational Linguistics: Volume 1, pages 865–872. Association for Computational Linguistics.

Kristina Toutanova and Christopher D Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13, pages 63–70. Association for Computational Linguistics.

Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 2003. Feature-rich Part-of-speech tagging with a cyclic dependency network. In Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1, pages 173–180. Association for Computational Linguistics.