

Development of neural network based rules for confusion set disambiguation in LanguageTool

Markus Brenneis¹

Abstract: Confusion set disambiguation is a typical task for grammar checkers like LanguageTool. In this paper we present a neural network based approach which has low memory requirements, high precision with decent recall, and can easily be integrated into LanguageTool. Furthermore, adding support for new confusion pairs does not need any knowledge of the target language. We examine different sampling techniques and neural network architectures and compare our approaches with an existing memory-based algorithm.

Keywords: Confusion Set Disambiguation; Grammatical Error Correction; Machine Learning; Neural Networks

1 Introduction

Grammar checkers are used to detect errors which cannot be detected by a simple spell-checker, e.g., confusion of words and agreement errors. For instance, the sentence **Then lecture is great.* contains no spelling mistake, but the words “then” and “the” have been confused. We have developed rules for confusion set disambiguation based upon neural networks and integrated them into the existing grammar checker LanguageTool.

Existing approaches for grammatical error correction have several problems: If rule based correction algorithms are used, the rules must typically be written by hand, which is time-consuming and prone to errors. Moreover, the creator of the rule must be familiar with the language the rule is written for.

On the other hand, when using an approach based on machine learning, extensive knowledge of the target language is not needed. But machine learning based grammatical error correction often suffers from creating too many false positives, which is annoying for a user of a grammar checker. Furthermore, many machine learning models require several hundreds of megabyte storage space (e.g. deep neural networks) and are slow at classification time, which both limit the usefulness of such models for end users, who often do not want a grammar checker to take up much storage space or take a long time to check a text.

¹ Heinrich-Heine-Universität Düsseldorf, Institut für Informatik, Universitätsstraße 1, Germany, markus.brenneis@uni-duesseldorf.de

Therefore, we focused on creating a classifier which has high precision to minimize false alarms, low memory requirements and little evaluation time.

We will now explain what LanguageTool is and how it works, what the task of confusion set disambiguation is, and state the goals of our work.

1.1 LanguageTool

LanguageTool is a free, open-source and rule-based grammar and style checker originally developed by Naber [Na03] and written in Java. The majority of rules are manually written in either XML or Java, hence rule development is a time-consuming task which requires knowledge of the target language.

When a text is checked, LanguageTool uses its own language-specific sentence splitter, tokenizer and part-of-speech tagger to assign part-of-speech (POS) tags to every token in the input. After POS tagging, each sentence is checked against the style and grammar rules.

1.2 Confusion Set Disambiguation

A typical type of mistake which is not detectable by a spell checker are confused words. Confusion set disambiguation is the task of choosing the right word from a finite set of words (e.g. {to, too, two}). In this paper, we will focus on confusion sets with exactly two tokens t and t' . LanguageTool already supports detecting commonly confused words. Currently, there are basically two types of rules: Pattern rules written in XML or Java, which are usually created by hand; therefore, creating new rules is time-consuming and prone to errors.

As an alternative, there are 3-gram based rules in LanguageTool, which require a copy of a large 3-gram corpus (e.g. 10 trillion tokens for English, stored in a 11 GB database) which bases upon the Google n-gram corpus [Li12]. The error detection algorithm is memory-based and works as follows: Let t be a token in a confusion pair (t, t') and t_{+n} the n th token after t in the text being checked. When t is encountered in the text, the number of occurrences m of the 3-grams (t_{-2}, t_{-1}, t) , (t_{-1}, t, t_{+1}) , and (t, t_{+1}, t_{+2}) are counted and compared with the number of occurrences m' of the same 3-grams containing t' instead of t . If m' is x times greater than m (where a suitable x with good precision and recall is determined beforehand), t is considered incorrect.

The 3-gram based rules have the advantage that rules have not to be written manually. On the other hand, there are several disadvantages: First, the rules fail to detect errors if the exact 3-gram is not part of the corpus. For instance, the mistake in *We go *too Gimli's birthday party.* is not detected, because the 3-grams *(go to Gimli)* and *(to Gimli 's)* are not part of the corpus, although the individual tokens are.

Furthermore, the user of LanguageTool needs to download a big corpus in order to use the rules and must have a sufficiently fast hard drive and enough memory, in order not to slow down the process of text checking too much.

1.3 Goals of our Work

The main goal of our work was to create confusion set disambiguation rules using neural networks for LanguageTool which are at least as good as the existing 3-gram based rules. Storing the new rules should not require much memory, preferable less than 100 MB, and the rules should cause as few false alarms as possible to be suitable for everyday use. What is more, the rules must be able to deal with unseen contexts and may not have a negative impact on the performance of LanguageTool. Furthermore, we wanted to examine the influence of different sampling methods and model sized on the performance of the classifier.

In the following section we will introduce our neural network architectures and the training process. Afterwards we compare our classifiers and the existing memory-based 3-gram rules with regard to precision, memory usage and speed. Finally, we have a look at alternative approaches and related work.

2 Model Architecture and Training Process

We will now describe how our classifier works and how it has been trained. In particular, we introduce our data set, discuss different approaches to sampling, our input representation and the architecture of our neural network.

2.1 Data Set

Our neural network has been trained on a large, unannotated corpus which can be considered to have no or at least very few mistakes. As shown by Banko; Brill [BB01], using larger data sets can improve the performance of a classifier significantly. Furthermore, some words like second person verb forms can only seldom be found in some corpora, for example newspaper articles. Thus, a corpus with sentences randomly chosen from newspaper articles from *Project Deutscher Wortschatz* [GEQ12] and sentences from *Tatoeba* [Ho] has been created. The final corpus for English contains more than 30,000,000 words and has been divided in a training (90 %) and testing set (10 %). This unusual split ratio has been used because we think that the data set is large enough to get decent test results with only 10 % of the data, and the model has a chance to learn more using 90 % of the data.

The corpus has been tokenized using the tokenizer of LanguageTool. For each confusion pair, we trained a separate classifier on a subset of the training set which only contains those sentences which contain the tokens of the confusion set, which typically are between 1.000 and 100.000 sentences per token, depending on how common it is in the corpus.

2.2 Sampling

It is often the case that one word of a confusion set occurs several times more often in the training corpus than the other word. Considering the German confusion set {wider, wieder}, there are around 40.000 sentences containing “wieder” in the training corpus, but only 471 sentences with “wider”. Our experiments discussed in section 3.3 have shown that this class imbalance leads to heavy overfitting, since the classifier is biased towards the majority class.

Sampling Technique	occurences of <i>wider</i> in training set	occurences of <i>wieder</i> in training set
None	471	44.751
Undersampling	471	471
Oversampling	44.751	44.751
Combination Over-/Undersampling	942	942

Tab. 1: Overview of sampling techniques using the confusion set {wider, wieder} as example

To overcome the issue of class imbalance, we compared three different approaches which can commonly be found in research [Ch09]: Random undersampling, random oversampling, and a combination of over- and undersampling. In the latter case, the oversampling has been limited to a factor of 2, and the majority class has been undersampled such that the class label ratio is 1. This approach seemed to be feasible because we did not want to throw away too many training samples as would be done in undersampling, but we also wanted to prevent the classifier to overfit on the few samples of the minority class. Table 1 summarizes the sampling techniques we studied.

2.3 Word Representation

As neural networks can only be applied to numeric input, the input token must be mapped to numerical values. One possibility would be a simple one-hot encoding, i.e., given a vocabulary of size n , the i th word is represented by the vector $[x_1, \dots, x_n]$ with $x_i = 1$ and $x_j = 0$ for $j \neq i$. This encoding has two crucial disadvantages: The size of the representation is very big, and any linguistic information about the relations of different tokens is lost, which is why we are using a word embedding to map tokens into a vector space.

Tokens are represented using a 64 dimensional word embedding created using the word2vec approach by [Mi13]. We used the continuous skip-gram model for creating the word2vec model, i.e., the model has learned to predict tokens which are likely to appear in the same context as another token. In the final vector representation, words with similar meaning are mapped to vectors which are close to each other, which enables the neural network to detect errors in contexts it has not seen before.

All tokens which appeared at least five times in the training corpus are part of the word2vec model’s dictionary. This way, the model is kept small by ignoring less frequently used tokens, and possible typos in the training corpus, which probably do not occur very often. Tokens which are not part of the dictionary are replaced by the special token “UNKNOWN”. To minimize storage space, the same embedding is used for each confusion pair.

2.4 Neural Network Architecture

The artificial neural network gets the two tokens before and after a confusion word candidate as input. It outputs a number y_i for each token in the confusion set, which can be interpreted as the logits, i.e., the logarithm of the odd $\log\left(\frac{p}{1-p}\right)$, where p is the probability for the corresponding token to be correct in the given context.

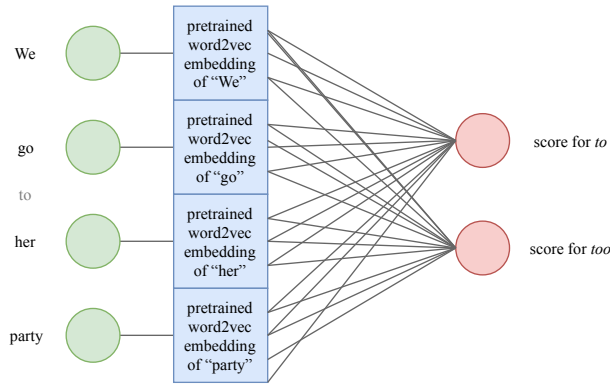


Fig. 1: An illustration of our “NN” architecture, which uses a separately trained word2vec embedding and no further hidden layers, for the confusion set $\{to, too\}$.

Our main architecture is a single layer network without any hidden layers and activation function, i.e., a linear model (called “NN”, depicted in figure 1). For comparison, we also trained a network with one hidden layer with 8 neurons and ReLU activation function (“NNH”) and variants which get only two tokens from the context as input (“NN2” and “NNH2”, respectively). We did not train any deep models or models with large hidden layers because our goal was to create a classifier which does not require much storage memory.

All architectures are trained for 1.000 epochs using the Adam Optimizer by Kingma; Ba [KB14] to minimize the softmax cross entropy loss

$$L = -\log\left(\frac{e^{y_i}}{e^{y_i} + e^{y_j}}\right) \quad (1)$$

where y_i is the output for the correct label and y_j the output for the wrong label.

2.5 Output Interpretation

The output (y, y') of the neural network is used as follows: Given a threshold $\theta \in \mathbb{R}^+$, the token t of the confusion set is considered incorrect and t' is considered correct, if and only if $y < -\theta$ and $y' > \theta$ (i.e., the network thinks t' is much more likely than t and t' seems to fit).

The reasonableness of this approach can be explained like this: Assuming that in a given context the probabilities for t and t' are independent, i.e., it is possible that both tokens are correct, the output (y, y') can be transformed into probabilities using the sigmoid function. So we assume that

$$p_t = \frac{1}{1 + e^{-y}} \qquad p_{t'} = \frac{1}{1 + e^{-y'}} \qquad (2)$$

are the probabilities that the first or second token are correct, respectively. Then the aforementioned approach is equivalent to saying that $p_t < 0.5 + \sigma$ and $p_{t'} > 0.5 - \sigma$, with

$$\sigma = \frac{1}{1 + e^{-\theta}} - 0.5 \in [0, 0.5) \qquad (3)$$

i.e., t' is considered at least 2σ more probable to be correct than t and $p_t < 0.5$ and $p_{t'} > 0.5$.

The practical advantage of the first criterion is that it requires fewer calculations, and is therefore used in our implementation.

3 Rule Quality and Comparison

In this section we will have a look at the quality of the rules with regard to precision and recall, comparing our different architectures and the existing 3-gram-based rules.

3.1 Precision and Recall

In order to be useful for a grammar checking application, the neural network based rules must not cause any or at least very few false alarms. In the context of the error detection task, we define true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn) as depicted in table 2.

Note that a true positive is an incorrect usage of a token which is marked as error, and not solely the case where the neural network would choose the right token (which is $tp + tn$).

	marked as error	not marked as error
correct usage	fp	tn
incorrect usage	tp	fn

Tab. 2: Definition of true positives, true negatives, false positives, false negatives. Note that a true positive is a wrong token correctly detected as wrong, and not the case where the neural network would insert the correct token.

For each confusion pair (t, t') we evaluated, we created a grammar checker rule and checked it against 5,000 sentences containing t and 5,000 sentences containing t' from the test set, and another 10,000 wrong sentences which were created by swapping t and t' in the correct sentences. We calculated precision P and recall R for different thresholds θ .

$$P = \frac{tp}{tp + fp} \qquad R = \frac{tp}{tp + fn} \qquad (4)$$

A rule is considered good if $P > 0.99$ (i.e., the probability for false alarms is less than 1 %) and $R > 0.5$ (i.e., more than 50 % of incorrect usages are detected as error). For comparison: The average recall of the existing 3-gram rules for English in LanguageTool is 0.56 with $P > 0.99$.

3.2 Comparison of network architectures

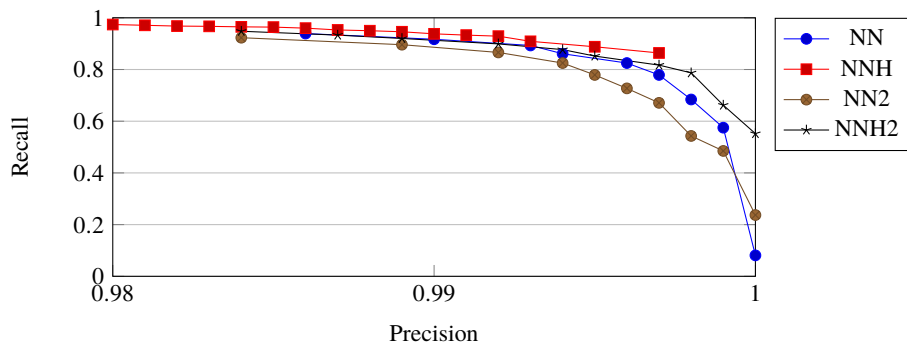


Fig. 2: Precision and recall for different network architectures for the confusion pair $\{to, too\}$

The neural network architectures show different recalls at the same level of precision on the test corpus. In general, looking at different confusion pairs, the architectures having 2 tokens as input have for a fixed precision lower recall than the corresponding architecture with 4 input tokens.

Moreover, the architectures with hidden layer perform better than those without hidden layer. Whether “NN” or “NNH2” performed better depended on the confusion set. The distance between the smaller “NN” architecture and the larger “NNH” within the interesting precision interval $[0.99, 0.995]$ has, in general, been rather small.

3.3 Comparison of Sampling Techniques

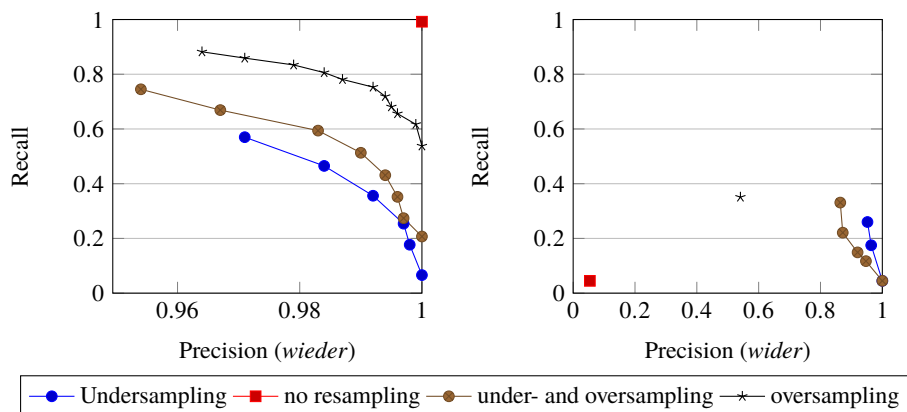


Fig. 3: Precision and recall for the confusion pair {wieder, wider}

We also had a look on how different sampling methods during the training process influenced the performance on the test set. Figure 3 shows precision and recall for the {wieder, wider} confusion pair using the “NN” architecture. For the diagram for *wieder*, only sentences where *wieder* is correct has been used, i.e., sentences with correct usage of *wieder* and sentences with incorrect usage of *wider*.

While there are decent results for detecting the right use of the more common word *wieder* when oversampling is used, the recall for the around 100 times less common *wider* is much worse, with a maximum precision of around 0.5, probably due to overfitting. If no resampling is used, the network is very good at dealing with contexts where *wieder* is correct, but has very low precision in contexts where *wider* must be used. Using a mixture of over- and undersampling produces relatively close results, where undersampling is worse for the recall of the more common *wieder* case and better for the less common *wider*.

For other imbalanced confusion pairs like {to, too} (factor 10), the differences have not been that big, such that undersampling has been used in the other experiments.

3.4 Comparison with 3-gram Rules

As our goal was to be at least as good as the existing 3-gram rules, we also compared the performance of our system with the existing rules. Note, however, that the comparison is not

confusion pair	$R_{3\text{-gram}}$	R_{NN}	confusion pair	$R_{3\text{-gram}}$	R_{NN}
and/end	0.81	0.84	da/dar	0.82	0.74
five/give	0.97	0.94	das/dass	0.43	0.81
it/its	0.95	0.92	den/denn	0.70	0.90
our/out	0.98	0.93	fielen/vielen	0.85	0.94
then/the	0.45	0.57	ihm/im	0.96	0.94
to/too	0.82	0.95	schon/schön	0.73	0.44
some/same	0.99	0.98	seid/seit	0.98	0.93

Tab. 3: Comparison of recall at $P = 0.99$ for some English and German confusion pairs.

completely accurate, since the 3-gram rules use a different tokenization algorithm, which is compatible with Google’s n-gram database. For instance, the 3-gram rules can detect the error in **give-year-old*, because this expression consists of 5 tokens according to the Google style tokenizer, whereas our rules fail to detect the error, since the expression is one token for the LanguageTool tokenizer. In order not to end up with a lot of “false” false negatives for our rules, we changed the existing testing algorithm in LanguageTool to exclude those cases.

The results depicted in table 3 show that our rules have, on average, a performance comparable to those using the memory-based 3-gram rules. In around half of the cases, our simple one-layer architecture outperforms the 3-gram rules.

3.5 Memory Usage and Runtime Performance

The files for the word2vec embedding for English have a size of around 65 MB (uncompressed, stored as plain text data). The files containing the neural network weights for the “NN” architecture have a size of 13 KB for each confusion pair. Thus, around 800,000 neural network based rules would need the same amount of storage memory as the 3-gram corpus, which is stored as 11 GB Lucene database index.

The start-up wall-clock time of the LanguageTool standalone GUI without 3-gram and neural network rules, from the start till an English example sentence has been checked against the intergrated pattern rules, is about 4.8 seconds on our test system with SSD. If only the 3-gram rules (and the pattern rules) are enabled, the start-up time is 1.2 seconds longer, with only neural network rules (and pattern rules) enabled, the time is 1.5 seconds longer.

The memory usage of the GUI 10 seconds after start-up and a garbage collection call is around 80 MB without 3-gram and neural network rules, 130 MB with 3-gram rules enabled and 100 MB with neural network rules loaded.

Checking a German text with around 3,000 words using the command line version of LanguageTool takes around 2.9 seconds with both rule types disabled and only pattern rules enabled, 4.5 seconds with 76 3-gram rules enabled and 3.0 seconds with 29 neural network rules of the “NN” architecture enabled. Hence we can see that our approach has a much lower impact on the time of the grammar checking process.

To sum up, the calculation done by the neural network code have a lower impact on the performance than the 3-gram lookup, and storing as well as loading the 3-gram index requires more memory than the word2vec model and the neural network data. Only the start-up time of LanguageTool is slightly negatively affected.

4 Alternative Approaches

During development of the architectures discussed in the previous section, we also had a look at other architectures and classifiers. In this section, we summarize which other approaches for the confusion pair disambiguation task have also been tested, and why we think that those approaches are inferior.

4.1 Classical Machine Learning

We also did experiments with classical machine learning classifiers which also got word2vec encoded context words as input.

Random forest are fast to train, had a precision of over 85% for most confusion pairs, but were unable to reach our target of 99% without further optimization. Another drawback is their model size of around 1.5 MB for a random forest with 20 decision trees, which is 100 more than needed by the “NN” architecture. Furthermore, random forests were considerably slower at evaluation time.

On the other hand, Support Vector Machines (SVMs) required more time at training time, were fast at test time, and reached results comparable with those for the “NN” architecture. But like random forests, the model sizes were larger than 1.5 MB, which was the main reason why we did not further evaluate SVMs.

4.2 Recurrent and Convolutional Neural Networks

As recurrent and convolutional neural networks are able to handle inputs of different lengths, they can easily be used to analyze sentences of different lengths. Although it seems promising to use these architectures, there are several drawbacks. First, the model size is much bigger because the model must be able to deal with larger inputs. It also takes more

time to check a sentence, because more computations have to be done, which can slow down the performance of the grammar checker considerably, especially if no GPU can be used. Furthermore, especially the training of recurrent neural networks takes a considerable amount of time, which would make rule creation a very time-consuming task.

5 Related Work

Miłkowski [Mi12] has studied automatic and semi-automatic creation of symbolic rules using transformation-based learning. The created rules have very good recall, but often suffer from a low precision, i.e., cause many false alarms, unless there is human intervention.

Support vector machines, convolution neural networks with fixed context size and recurrent neural networks for detecting grammar errors at the word level using unlabeled data have been compared by Liu; Liu [LL17]. A bidirectional LSTM-based classifier performed best, but still has an F-measure below 20% which makes it unsuitable for application in a grammar checker because it would cause too many false alarms. Because of that, we focused on the simpler task of confusion set disambiguation.

Banko; Brill [BB01] have compared different classifiers for the confusion set task with regard to their performance if the training corpus is increased from 1 million words to 1 billion words. They have shown that a memory based algorithm is outperformed by a more complex perceptron algorithm when the training corpus has more than 1 million words.

6 Conclusion and Future Work

In this paper we have presented a new kind of rule for the free style and grammar checker LanguageTool which uses neural networks, and tested them successfully on a confusion set disambiguation task. The rule quality is similar to the memory based rules which are already part of LanguageTool, but our rules require less memory and are faster. Hence, our rule can be used instead or in addition to the existing 3-gram rules. It has to be noted, though, that creating new neural network based rules requires several minutes of computation time for the training process, which is not needed for a new 3-gram rule.

Possible next steps include using information from the part-of-speech tagger to handle words which are not part of the training vocabulary more appropriately than projecting all out-of-vocabulary tokens to a single “UNKNOWN” token. Furthermore, the current neural network architecture can easily be extended to support bigger confusion sets, such that rules for {to, too}, {to, two} and {two, too} can be merged in one {to, too, two} rule. Adding support for confusion sets containing larger expressions instead of single tokens (e.g. {das, dass,} or {in dem, indem}) is also planned. In addition, training on even larger corpora might further improve the performance. It is also possible to reduce the storage space for the neural network rules by using a binary format for storing weight matrices.

Moreover, adding support of new confusion pairs could be simplified by creating a neural network which is not specialized on a confusion pair, but can calculate probabilities for any target tokens.

Acknowledgements

Computational support and infrastructure was provided by the “Center for Information and Media Technology” (ZIM) at the University of Düsseldorf (Germany). I would like to thank my supervisor Sebastian Krings for his useful advises, and I also thank the LanguageTool community for the feedback during the integration of the new rules into LanguageTool.

References

- [BB01] Banko, M.; Brill, E.: Scaling to very very large corpora for natural language disambiguation. In: Proceedings of the 39th annual meeting on association for computational linguistics. Association for Computational Linguistics, pp. 26–33, 2001.
- [Ch09] Chawla, N. V.: Data mining for imbalanced datasets: An overview. In: Data mining and knowledge discovery handbook. Springer, pp. 875–886, 2009.
- [GEQ12] Goldhahn, D.; Eckart, T.; Quasthoff, U.: Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages. In: LREC. Pp. 759–765, 2012.
- [Ho] Ho, T.: Tatoeba Downloads, URL: <https://tatoeba.org/eng/downloads>, visited on: 11/01/2017.
- [KB14] Kingma, D.; Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980/, 2014.
- [Li12] Lin, Y.; Michel, J.-B.; Aiden, E. L.; Orwant, J.; Brockman, W.; Petrov, S.: Syntactic annotations for the google books ngram corpus. In: Proceedings of the ACL 2012 system demonstrations. Association for Computational Linguistics, pp. 169–174, 2012.
- [LL17] Liu, Z.-R.; Liu, Y.: Exploiting Unlabeled Data for Neural Grammatical Error Detection. Journal of Computer Science and Technology 32/4, pp. 758–767, 2017.
- [Mi12] Miłkowski, M.: Automating rule generation for grammar checkers. arXiv preprint arXiv:1211.6887/, 2012.
- [Mi13] Mikolov, T.; Chen, K.; Corrado, G.; Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781/, 2013.
- [Na03] Naber, D.: A rule-based style and grammar checker./, 2003.