Offensive Language Detection with Neural Networks for Germeval Task 2018

Dominik Stammbach
Saarland University
dominiks@coli.uni-saarland.de

Azin Zahraei
Saarland University
azinz@coli.uni-saarland.de

Polina Stadnikova
Saarland University
polinas@coli.uni-saarland.de

Dietrich Klakow
Saarland University
dietrich.klakow@lsv.uni-saarland.de

Abstract

In this paper we describe our submissions to task I of the GermEval 2018 Shared Task with the goal of identifying offensive language in a set of German tweets. We experiment with two neural architectures and different features. Our submission consists of 3 runs using ensembles of different neural network architectures, each achieving approximately 78 % macro-F1 measure on the last 500 tweets from the training set. The source code for our experiments is publicly available on Github. [1]

1 Introduction

In recent years, it has become increasingly important to come up with countermeasures to deal with offensive language in social media. The NetzDG law which has been in effect since January 1 2018 in Germany requires tech companies like Twitter to delete obviously illegal content. (Wikipedia contributors, 2018) The huge amount of data posted on Twitter and the fact that German is in the top 10 languages of this social media platform (Hong et al., 2011) makes manually monitoring the data unfeasible and calls for automatic methods of identifying offensive language.

The GermEval 2018 Shared Task is focused on detecting offensive comments in a set of German tweets in two subtasks. Task I is a binary classification of tweets. Task II requires a more fine-grained classification of the offensive tweets into 3 subcategories, namely profanity, insult and abuse. But because of the small number of examples for the profanity class, training a neural network to detect profanity was infeasible. Because of the nature of the evaluation metric it was unlikely to get competitive results in task II so we only submit our model for task I.

[1] https://github.com/polinastadnikova/-neurohate

For our submission we have used neural networks which have become the top-performing technique for many tasks in the field of natural language processing. Convolutional Neural Networks (CNN), which were initially invented for the computer vision domain, have proven to be effective for many Natural Language Processing tasks. This architecture allows for extraction of local features in text, e.g. word order. This way, we are able to make use of combinations of words and use fixed size regions of text, e.g. bigrams, trigrams and so on as features. Yoon Kim (2014) shows the effectiveness of using a CNN for text classification by comparing results on different benchmarks. Recurrent Neural Networks (RNN), on the other hand, are able to extract long term dependencies. This is a feature that is definitely useful in offensive language detection. RNN-based methods have produced state-of-the-art scores for offensive language detection in other languages (Del Vigna et al., 2017). Thus, we have implemented both a CNN and a RNN model for this task.

Following many experiments with different ways of handling the data and different architectures for our prediction model, we selected our best models based on their macro-averaged F1 scores. More specifically, we compared the models based on their mean F1 score when 10 fold cross-validating on all the training data. We submit three runs, where the first, second and third runs are an ensemble of RNNs, an ensemble of CNNs and an ensemble of CNNs and RNNs respectively. After describing the data and how we preprocessed them in Section 2, we introduce the architectures and hyperparameters used in our best models in Section 3. In Section 4, we talk about our experimental setups and their results.

2 Data

The training data consists of 5009 tweets in German, where some tweets contain different types

of hate speech. The data is annotated according to the tasks: binary and fine-grained classification. Therefore each tweet has two labels, OFFENSE or OTHER as the first label and as the second label one of the following: INSULT, ABUSE, PROFANITY, OTHER. In our work, we focus on the binary classification, that means we have 1688 training examples containing offensive language and 3321 without hate speech. The reason for our decision not to participate in the fine-grained classification task is that there are only 71 examples for the PROFANITY label, 1022 examples for ABUSE out of 1688 tweets. We believe it is not enough for neural network training and furthermore our system would be biased towards the ABUSE label.

2.1 Preprocessing

For classification, as well as for many other NLP tasks, preprocessing of the training data has an impact on the system's performance (Kannan and Gurusamy, 2014; Qu et al., 2015). Since we use neural networks for our classifier and such approaches are data-driven, preprocessing becomes a crucial part of the system.

First of all, we tokenize the data using the *twokenize* package [2] for Python, which was specially designed for tokenization of tweets. This forms the *basic* preprocessing.

For the *advanced* preprocessing, we continue working with the tokenized tweets. We remove punctuation and words containing non-alphanumerical characters (including emojis) and we lowercase all the words. We consider hashtags, words with the # sign, as a special case since they are widely used on Twitter. We do not want to remove them because hashtags can be repetitive and capture some relevant information. For this reason, we just remove the hash sign. Mentions, denoted by the @ sign, are also popular on Twitter but they are often random and we decided that they are not relevant for our classifier. By removing them, we back down from using implicit information captured in the word embeddings about specific users.

Since neural networks cannot handle categorical features as input, we need to convert the input tweets into a numerical representation. Following convention, we make use of pre-trained word embeddings. We use the German Twitter embeddings collected by the researchers at Heidelberg University[3]. The embeddings are trained using word2vec, with 100 dimensions for each word, a context window size of 7 and a minimum occurence of at least 50 times per word in the data. They are also tokenized using the *twokenize* package, hence our decision to use the same library to tokenize the tweets.

We vectorize tweets in the following way: each tweet is a vector with word IDs as its elements. Word IDs correspond to the row of a word in the embeddings matrix. For words which occur only in the training data but not in the embeddings (OOV) we introduce the label UNKNOWN.

2.2 Features

Features have a large impact on performance, especially in domain specific tasks (Schmidt and Wiegand, 2017). The information, relevant for the features, is extracted during preprocessing.

- **Word embeddings** represent one of the most common features in neural NLP (Ruder et al., 2017) . As already introduced above, they are vector-based word representations which are usually pre-trained on large datasets. The embeddings which we use perfectly fit our purpose since they are trained on the Twitter data. It is known that word embeddings trained on out-of-domain data lower performance of systems(Qu et al., 2015). Interestingly, words in the embeddings are true-cased, most nouns appear twice in the embeddings, once true-cased and once lowercased. Therefore the question arises whether we benefit from lowercasing the data. We design our experiments with regard to this fact.

We also tried out other features like emphasizing some categories or considering punctuation, all of which lowered the performance and thus are not included in our final models. We will briefly describe them in Section 4.

3 Model

We experimented with two different neural network architectures, namely convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

[2] https://github.com/nryant/twokenize_py

[3] http://www.cl.uni-heidelberg.de/english/research/downloads/resource_pages/GermanTwitterEmbeddings/GermanTwitterEmbeddings_data.shtml

3.1 CNN

When using CNNs in NL, a window size is defined and a shared weight matrix is trained which is slided along sentences to produce a feature map for every n-gram in the sentence where n is the window size. Afterwards, we do max pooling over the different features generated and use this as a hidden representation for the sequence. The main benefit is that it is very fast and has few trainable parameters, but can only consider local information. For our final CNN model, we use word embeddings which are initialized with the values from the Heidelberg embeddings and can be trained. We max pool over 1 layer of bi- and trigram features with 64 filters per filter feature. We use a stride of 1 to extract such features and to do max pooling over all the resulting feature maps. Then this hidden representation is fed into a two-layer deep feed-forward network with the first layer having 128 hidden units and the second layer with only two units to perform classification. These parameters were chosen by grid searching over a number of different settings.

3.2 RNN

While using RNNs, one can encode the sequence in a very intuitive way, namely as word representations for every word. In this case, a recurrent neural network starts at the beginning of the sequence and computes a hidden state given the input. This hidden state is propagated through the sequence and updated at each timestep given the current input. The hidden state can also be thought of as the memory of the network and thus is able to capture global information from the sentence. The downsides consist of having more trainable parameters to be optimized using a limited amount of training data. For our final RNN model, we use bidirectional gated recurrent units (GRUs) (Cho et al., 2014) with 50 hidden units for each direction. We also experimented using LSTMs but they perform worse. We think this is explainable by the lower numbers of trainable parameters in the GRU-case which performs better on the small number of training examples we actually have. We performed a max-pool operation over the hidden timesteps because important features at a given timestep may be forgotten towards the end of the sequence and this is a straight-forward way to keep such features. The resulting hidden representation of the sequence (output of the GRUs) is fed into a 4-layer deep feed-forward neural network with 100 hidden units for the first three layers and two neurons in the final layer to perform classification.

Both architectures share some common settings which we describe here: All the layers in the feed-forward neural networks use a dropout-rate of 0.2, a ReLU-activation and L2-regularization with λ 0.0001. We also apply the same dropout to the input sequence and the output representation of the CNN/RNN respectively. We used cross-entropy as a loss function and optimized it using the Adam optimizer with default parameters. Additionally, we weighted offensive tweets twice as much as the non-offensive ones to overcome the imbalance with respect to the number of training examples in the data.

The training was completed using a batch size of 64 examples per batch, with the data shuffled after every epoch and early stopping on a development set with a patience of 4. We selected all the parameters described above by peforming grid search over the training set in a 10-fold cross-validating fashion. The two configurations described above turned out to be the ones yielding the highest average macro F1 measure on different parts of the data. The ensemble method is a loose version of bagging which furthermore increases the robustness and accuracy of the classification. We decided to use it since the high fluctuations in the results were observed when running the same configuration multiple times. A possible reason for this might be the random parameter initialization. Moreover, the problem of finding the right seed in training neural networks also plays an important role here (Bajgar et al., 2018). To counter such behaviour while grid searching, we use 10-fold cross-validation. Finally, using an ensemble of 9 identical models trained on different parts of the data [4], we do predictions based on the majority vote from these models and observe an increase of approximately 2% F1 measure compared to when only one model was used. Our final macro-F1 scores are discussed in the next section.

4 Experiments and Discussion

In Table 1 we show our results with different experiments. All experiments (except the Character CNN) are conducted using the GRU-architecture described above. For each experiment, we use 10-fold cross-validation and in each fold, we split the

[4] one part of training data is reserved for performing early stopping

data in three parts: a training set, a validation set for early stopping and a testset to evaluate. We report the average macro-F1 score over all the ten folds. Our system is optimized for the F1-measure and not for precision and recall, for this reason we report only the first one. Throughout the experiments, we fixed the different splits so that we do not evaluate every experiment on different parts of the data.

In the first row, we just looked up the true-cased version of a word in our embeddings vocabulary. In case we cannot find it there, we try to back off to the lowercased version of the word and otherwise, we just use the UNKNOWN token.

In the second row, we report the results for replacing tokens which appear in a swear word dictionary[5] by a special SWEAR token. The motivation for this feature was the fact that offensive tweets tend to contain some swear words. Interestingly, compared to true-cased data, this significantly improves performance, but by just lowercasing all the words, we get even better results(row 3). This can be justified by the fact that for most nouns, two versions, one true-cased and one lowercased copy, exist in the embeddings and words are not always accurately true-cased in tweets. Thus, by lowercasing all words, we avoid confusing the network with inconsistently true-cased words.

In row 5, we run the model without excluding non-alphanumerical tokens, punctuation and emojis. This again decreases the system's performance. Another issue we tried to overcome here is the out-of-vocabulary (OOV) words treatment, which is common in NLP, especially with small datasets like ours. For this, we use *hunspell* spellchecker [6]. Many tweets contain spelling errors, therefore the spellchecker helps to reduce the number of OOVs: from 2511 OOV tokens to 91. The only problem here is that the spellchecker generates words which are correct but do not occur in the embeddings and therefore are not very useful[7]. This might be an explanation for the slightly worse model performance.

Row 7 shows the results from running our RNN model using LSTMs instead of GRUs. We speculate that since LSTMs have a larger number of trainable parameters, training them on our small training data is producing worse results than GRUs.

[5]https://www.schimpfwoerter.de/

[6]https://pypi.org/project/hunspell/

[7]For instance, SPDler is corrected to Spieler, and Antifantenbrut to quantifizieren.

In row 8, we see the results when using our CNN model with character embeddings. We grid-searched over a number of settings and our best result was a setting with 50 hidden units, a dropout of 0.1 and a batch size of 256. Despite the fact that using character embeddings solves the OOV issue, the model still fails to capture lots of the more broad-scale features in a sentence and therefore yielded very low results compared to our other runs.

Table 2 summarizes the runs which we submit for task I. For each run, we evaluated our system on the last 500 tweets from the training set. The last run consists of an ensemble of 18 models, 9 RNN GRUs and 9 CNNs. We expect that this might slightly boost the performance. We combined the predictions from the two sets of models on the test set and predicted offense tags if at least half the models predicted a tweet as offensive.

Note that the results from Table 1 and 2 are not directly comparable since we evaluate the features using 10-fold cross-validation and the submission runs using the last 500 examples which we excluded during the training time. For the final submission, we retrained the ensembles including the last 500 tweets as additional training material.

Method	**F1(%)**
True-cased	61.6
True-cased + swear word dictionary	74.2
Lowercased	75.9
Lowercased + swear-word dictionary	74.9
Lowercased + non-alpha numerical tokens	72.6
Spellchecker for OOVs	69.7
Using LSTM instead of GRU	68.3
Character embeddings	49

Table 1: Results for different experiments

Submission File	**Ensemble**	**F1(%)**
SaarOffDe_coarse_1.txt	RNN	77.7
SaarOffDe_coarse_2.txt	CNN	78.6
SaarOffDe_coarse_3.txt	CNN + RNN	77.6

Table 2: Submitted runs

5 Conclusion

In this paper, as part of the Germeval 2018 shared task, task I, we implemented neural networks for

the Identification of Offensive Language in German.

We evaluated the two most common neural network approaches for sequence classification on a new German dataset and reported different preprocessing techniques and their impact on the final classification. The most surprising fact seems to be that the best models rely on lowercased words even though the word embeddings we use are true-cased. The overall best performance was achieved with a CNN model with a bi- and trigram filter.

We submit three runs for task I consisting of an ensemble of RNNs[8], CNNs[9] and a combination of both RNNs and CNNs together[10].

References

Lichan Hong, Gregorio Convertino, and Ed Chi. 2011. *Language matters in Twitter: A large scale study* In International AAAI Conference on Weblogs and Social Media.

Wikipedia contributors. The Free Encyclopedia, 30 Jul. 2018. Web. 3 Aug. 2018. *Netzwerkdurchsetzungsgesetz. Wikipedia, The Free Encyclopedia. Wikipedia*. American Psychological Association, Washington, DC.

Anna Schmidt and Michael Wiegand. 2017. *A Survey on Hate Speech Detection using Natural Language Processing* In: Proceedings of the Fifth International Workshop on Natural Language Processing for Social Media.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio 2014. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*

Ondrej Bajgar, Rudolf Kadlec, Jan Kleindienst 2018. *A Boo(n) for Evaluating Architecture Performance* In: Proceedings of the 35th International Conference on Machine Learning, PMLR 80:344-352, 2018.

Yoon Kim 2014. *Convolutional Neural Networks for Sentence Classification* CoRR abs/1408.5882

Fabio Del Vigna, Andrea Cimino, Felice Dell'Orletta, Marinella Petrocchi, and Maurizio Tesconi . 2017. *Hate me, hate me not: Hate speech detection on Facebook.* In: Proceedings of ITASEC.

Subbu Kannan and Vairaprakash Gurusamy. 2014. *Preprocessing Techniques for Text Mining.* In: Proceedings of RTRICS.

Sebastian Ruder, Ivan Vulić and Anders Sogaard. 2017. *A Survey of Cross-lingual Embedding Models.*

Lizhen Qu, Gabriela Ferraro, Liyuan Zhou, Weiwei Hou, Nathan Schneider, and Timothy Baldwin. 2015. *Big Data Small Data, In Domain Out-of Domain, Known Word Unknown Word: The Impact of Word Representation on Sequence Labelling Tasks* . In: Proceedings of the 19th Conference on Computational Language Learning.

[8] corresponds to the run SaarOffDe_coarse_1.txt from our submission.

[9] corresponds to SaarOffDe_coarse_2.txt.

[10] corresponds to SaarOffDe_coarse_3.txt.