Tackling word sparsity through preprocessing

Ferdinand Kossmann, Philippe Mösch, Saiteja Reddy Pottanigari, Kaan Sentürk
Kaggle: moeschp
Department of Computer Science, ETH Zurich, Switzerland

***Abstract*—Compared to other sentiment analysis data, Twitter data contains a higher word sparsity and flatter word frequency distribution. This is caused by the use of hashtags, slang, abbreviations, emoticons, misspellings and more. While Twitter is similar to spoken language, 60% of the words only appear once in the given training data, while only 42% of the words appear once in transcribed English conversations. This causes problems such as noisy embeddings or a scarcity of training data containing certain words.**

In this work, we investigate how these problems can be addressed with preprocessing steps such as splitting hashtags or replacing slang. We furthermore propose MMST spelling correction, a novel method for choosing correction candidates of misspelled words, according to how well the candidate fits into context. MMST spelling correction hereby outperforms other popular spelling correction libraries.

I. Introduction

A key difficulty in sentiment analysis for Twitter is the data's high vocabulary sparsity. Although Tweets are very similar to spoken language [1], 60% of the words in the provided training data only appear once throughout the data. This stands in contrast to 42% of the words only appearing once in transcribed conversations [2]. Figure 1 visualizes the higher vocabulary sparsity of the provided data when compared to a corpus of 11 million words of transcribed face-to-face conversations.

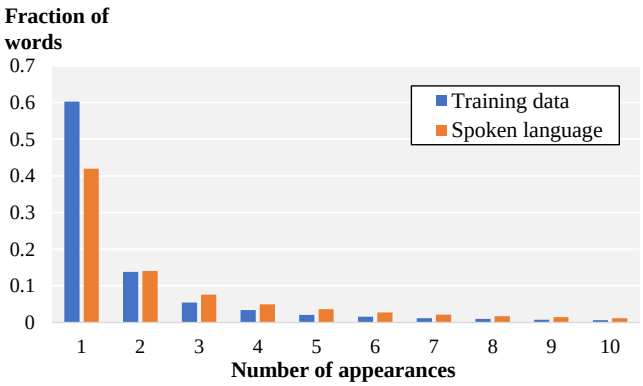

Figure 1: Fraction of words against appearances in text for less than 10 appearances.

This sparsity is problematic because it leads to noisy embeddings and a shortage of training data containing the words. It is furthermore especially problematic for Twitter data, since Tweets are limited to 140 characters and each token contains scarce, valuable information on the Tweet's sentiment.

The discrepancy between the sparsity in the Twitter data and spoken language can be explained by the use of non-standard language in Tweets. Such language includes hashtags, abbreviations, slang, misspelled words or emoticons. Figure 2 assigns the words, that only appear once in the training data, to those different categories.

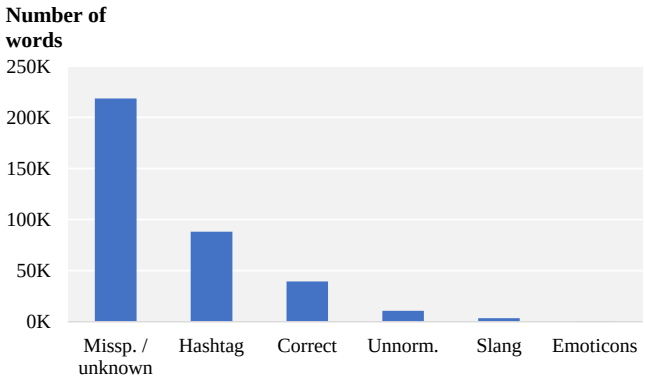

Figure 2: Types of words only appearing once.

To overcome the problem of vocabulary sparsity, this work investigates the effect of preprocessing methods that densen the word frequency distribution. We hereby present novel preprocessing techniques like a context sensitive spelling corrector that exploits clustering of similar words in GloVe embeddings. Furthermore, since densifying the word frequency distribution doesn't directly translate to better accuracy, all preprocessing steps are also evaluated in terms of their effect on model accuracy.

II. Preprocessing

The following section describes the preprocessing steps investigated in this work.

Prior to preprocessing, we applied the following data cleaning:

1) Remove duplicate Tweets from training data: Some Tweets appear multiple times in the training data. This effectively multiplies the weight of the Tweet, which is an unwanted effect.
2) Question mark separation: All punctuation marks terminating a sentence (e.g. !, ., ...) have a spacing between the sentence and the mark. Question marks

(?) are the exception. We insert a space to standardize this for easier use in the later preprocessing steps.

3) Some emoticons appear normally (e.g. ":'-(") and some have spaces between the characters (e.g. ": ' - ("). We remove spaces between emoticons if there are any. This enables recognition of same emoticons.

We then investigated the following preprocessing steps:

- Hashtag split: Split hashtags into space-separated words. Example: #pronetocry → prone to cry. Third-party library [3].
- Emoticon replace: Replace emoticons with their meaning in words. Example: :) → happy. Dictionary scraped from Wikipedia [4], then complemented and cleaned.
- Slang replace: Replace slang and abbreviations. Example: L8 → late. Dictionary scraped from noslang.com [5].
- Normalization: Remove falsely repeated letters. Example: looooveee → love. Naïve search for spelling that is in English, slang, or emoticon dict.
- Contraction: Contract words with apostrophe. Example: can't → can not. Dictionary taken from [6] and [7].

Finally, we investigated the effect of spelling correction. Spelling correctors have the flaw of introducing errors as the right correction might be non-trivial. For example, "the fx runs" could be corrected to "the fix runs". The only way to avoid such errors is to look at the context that the misspelled word appears in. In order to reduce error rates, we therefore developed *MMST spelling corrector*, a context-sensitive spelling corrector.

A. MMST spelling correction

Given a set $\mathcal{T}$ of correctly spelled words in a Tweet, and set $\mathcal{M} = \{(w, \mathcal{C}_w) |\ w$ misspelled and $\mathcal{C}_w$ candidate corrections for $w\}$, the MMST spelling corrector tries to choose the correct candidate correction out of $\mathcal{C}_w$ for each misspelled word w. To do so, the MMST spelling corrector creates a graph $\mathcal{G}$ in which each correct word and correction candidate is represented by one node. Definition 2.1 defines a subset of these graph nodes of this graph, such that this subset determines a valid correction for the input sentence.

Definition 2.1: Correction Subgraph. A correction subgraph is a subset S of nodes in $\mathcal{G}$, such that the node of exactly one word of each candidate set $\mathcal{C}_w$ is in S, and the node of each correct word is in S.

We now construct $\mathcal{G}$ the following way: Each correction candidate node is connected to all other nodes that are not in the same correction candidate set. Furthermore, all correct nodes are connected with each other. A connecting edge has the weight of the Euclidean distance between the embeddings of the two connected words.

Given graph $\mathcal{G}$, MMST correction will then find the Correction Subgraph, that has the minimal minimum spanning tree (MMST) out of all possible Correction Subgraphs in $\mathcal{G}$.

Algorithm 1 provides pseudo code for our implementation of MMST correction.

Algorithm 1: Spelling correction with MMST

Input: correct words $\mathcal{T}$, misspelled words with candidate sets $\mathcal{M}$

// build graph
Graph $\mathcal{G} \leftarrow$ nodes for each element in $\mathcal{T} \bigcup_{w \in \mathcal{M}} \mathcal{C}_w$
for *each pair of nodes* (i, j) *with* i *and* j *not element of the same candidate set* **do**
  $d_{i,j} \leftarrow$ euclidean_distance(emb(i), emb(j))
  insert edge (i, j) with weight $d_{i,j}$ into G
end

// build MMST
$\mathcal{S} \leftarrow$ minimum spanning tree of $\mathcal{G}$
while *there exists a* $\mathcal{C}_w$ *with* $|\mathcal{C}_w| > 1$ **do**
  calculate deletion costs of all nodes
  delete cheapest $i \in \mathcal{C}_w$ with $|\mathcal{C}_w| > 1$ from $\mathcal{S}$
end

MMST correction has the advanatage of being context sensitive for a learned notion of context (word embeddings). This enables the context to be highly fitted to the text that should be corrected: For example, we used GloVe embeddings obtained from 2 billion Tweets [8] to define how similar words are in the context of Twitter.

Furthermore, MMST correction choses the most likely context (minimal minumum spanning tree) considering all possible correction candidates. Like this, even when many words are misspelled, words in candidate sets will be chosen in consideration of other candidates from other candidate sets.

Figure 3 visualizes graph $\mathcal{G}$ and its MMST (black edges) for the sentence "My unt is the sister of my moter". Note that stop words are removed and that the embeddings have been mapped to 2D using tSNE. The tSNE mapping did not preserve the high-dimensional MMST.

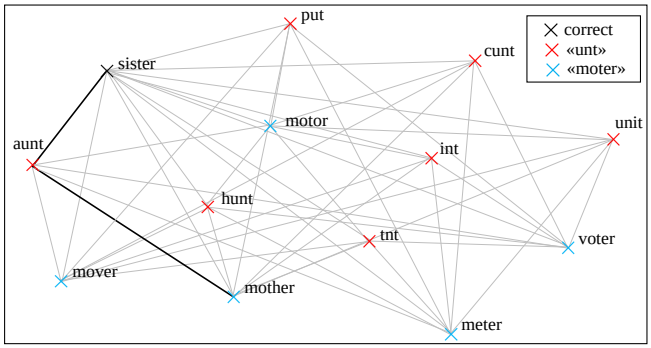

Figure 3: Visualization of graph and MMST produced for sentence "My unt is the sister of my moter".

III. Models and training

The following section describes the feature representations, as well as layer types used in the trained models.

A. Feature Representation

For feature representation we chose pretrained GloVE embeddings [8], as well as Contextualized BERT embeddings. The GloVe embeddings are suitable for our task, since they were trained on 2 billion Tweets. The BERT embeddings are suitable, as they represent state-of-the-art embeddings.

B. BiLSTM layer

As our tweet sentiment analysis is a sequence binary classification, recurrent models are a suitable choice. Bidirectional Long short-term memory cells (BiLSTM) have hereby proven to produce good results as they handle sequential dependencies over larger token sequences [9].

C. BiGRU layer

Bidirectional Gated Recurrent Units (BiGRU) are a faster alternative to BiLSTMs [10]. They are therefore suitable, especially when running as part of models with higher parameter size, as well as when performing larger trainings, for example with extensive pretraining.

D. BiLSTM + Self attention layer

Recurrent models suffer from vanishing gradients when predicting over a long sequence of tokens. To overcome this problem, self attention networks detect the words that have the most decisive effect on classification and capture the most important semantic information of the sequence [11]. We therefore also implemented a self attention network feeding from a BiLSTM. For H being the hidden output of the BiLSTM, and h^* being the classification output, the self attention layer performs the following:

$$M = \tanh(H)$$

$$\alpha = softmax(w^T M)$$

$$r = H\alpha^T$$

$$h^* = \tanh(r)$$

E. SepCNN

The SepCNN [12] (Depthwise Separable Convolutional Network) we are using consists of a depthwise convolution followed by a pointwise convolution.

The depthwise convolution $f^{(d)} : \mathcal{R}^{n \times c_{in}} \rightarrow \mathcal{R}^{n \times c_{out}}$ can be represented by a Kernel matrix $K^{(d)} \in \mathcal{R}^{w \times c_{in}}$, where w specifies kernel width. Is can be defined for spatial s and channel c (in computer vision) as:

$$f^{(d)}(X; K^{(d)})_{s,c} = \sum_{i=1}^{i \leq w} K^{(d)}_{i,c} X_{s+i,c}$$

The pointwise convolution $f^{(p)} : \mathcal{R}^{n \times c_{in}} \rightarrow \mathcal{R}^{n \times c_{out}}$ can also be represented by a Kernel matrix $K^{(p)} \in \mathcal{R}^{c_{in} \times c_{out}}$ and defined as:

$$f^{(p)}(X; K^{(p)})_s = \sum_{i=1}^{i \leq w} K^{(p)} X_s$$

The separable convolution is a composition of these two convolutions and therefore parameterized by $K^{(d)}$ and $K^{(d)}$:

$$f^{(s)}(X; K^{(d)}, K^{(p)}) = f^{(p)} \circ f^{(d)}$$

Due to the stacking of these convolution layers we are able to independently look at correlations and at spatial correlations with far fewer parameters than regular convolutions, which can be very beneficial in short text sequences as in tweets.

IV. Results

The following section describes the effect that the preprocessing steps and embeddings had on model performance.

A. Effect of preprocessing

The following subsection investigates the effect of the preprocessing steps excluding spelling correction. To evaluate how beneficial a preprocessing procedure is, we trained a BiLSTM with GloVe embeddings for 10 epochs on the procedure's output. We then evaluated the model's accuracy on a held-out 10% validation split.

To evaluate the effect of each preprocessing step, we first applied it singly on the data. Table I shows the effect each preprocessing step when applied alone.

Step applied	Accuracy
No preprocessing	87.12%
Normalization	87.21%
Hashtag split	87.16%
Emoticon replace	87.01%
Slang replace	87.05%
Contraction	87.27%

Table I: Effect of individual preprocessing steps on model accuracy.

Given the above findings, we then built a pipeline combining the positive effects of the performance-enhancing steps. To make the steps profit from each other, we ordered them in the following way:

$$\text{Hashtag split} \rightarrow \text{Normalize} \rightarrow \text{Contraction}$$

This follows their topological order: Before words in a hashtag can be normalized, the hashtag needs to be split. Before an unnormalized word can be contracted, the word needs to be normalized.

Table II shows the performance of that pipeline.

Pipeline applied	Accuracy
No preprocessing	87.12%
Hash. split → Norm. → Contr.	87.40%

Table II: Combination of the performance-enhancing preprocessing steps

B. Effect of spelling correction

We measure the effect of spelling correction. Hereby, we take the data preprocessed with the above pipeline and apply different spelling correctors to it.

Spelling corrector applied	Accuracy
No spelling correction	87.40%
pyenchant spelling corrector	87.32%
MMST spelling corrector	87.49%

Table III: Combination of the performance-enhancing preprocessing steps

We also compared MMST's performance to the one of TextBlob and pyspellchecker by looking at their outputs by eye. Both spelling correctors made significantly more mistakes on the data subset we looked at. However, processing the whole data set with these spelling correctors would have taken several days on our CPUs which is why we couldn't run a model on the fully processed data. TextBlob reports a 70% correction accuracy [13].

C. Effect of different training procedures

We finally investigated how further learning methods perform on the preprocessed data. We hereby combine the elements described in section III. with different Hyperparameter choices and pretraining. The pretraining data consisted of 3.6 million samples including 1 million Tweets [14], 600k Yelp reviews and 2 million Amazon reviews [15]. We selected the following training procedures for execution:

1) BERT embeddings, 2 stacked BiLSTMs: 64 batch size, 5 epochs.
2) BERT embeddings, BiLSTM with self attention: 64 batch size, 5 epochs.
3) BERT base embeddings with 2 stacked BiGRUs: Batch size 64, 1 epoch pretraining, 6 epochs training.
4) GloVe embeddings with SepCNN: Batch size 512, 10 epochs training.
5) BERT base embeddings with BiLSTM + self attention: Batch size 256, 1 epoch pretraining, 3 epochs training.

Table IV shows the results of the above procedures on the public test set on Kaggle.

Training	Test accuracy
Procedure 1	87.98%
Procedure 2	87.90%
Procedure 3	88.72%
Procedure 4	83.67%
Procedure 5	86.80%

Table IV: Selection of further models executed on best preprocessed data

V. Discussion

The results give a better understanding of how preprocessing can benefit and hurt model accuracy. They show, that some steps like slang and emoticon replacement worsened the model's performance. This is likely due to the fact, that both slang and emoticons have embedding representations that more accurately represent the tokens than the embeddings of the replacement words.

However, the results also show that other preprocessing steps benefit the model's performance. Such steps are hashtag splitting, normalization and contraction.

The results furthermore show, that the effect of spelling correction depends highly on correction accuracy. While pyenchant worsened prediction accuracy, MMST correction improved it. This highlights the importance of elaborated spelling correctors that are context-sensitive.

While MMST correction both outperformed pyenchant and improved the model's accuracy, there are many potential improvements of MMST that remain to be tried out.

- Edge weights: Based on cosine similarity of embeddings instead of euclidean distance.
- If embeddings are too far away from each other, don't include the edge. This might provide outlier robustness in graphs where all correction candidates seem bad.
- Incorporate prior scores, based on popularity of a word: Misspelled words are more likely to be popular words, as misspelled words also follow the typical English word frequency distribution.

Also, it remains open how the effect of preprocessing plays out for other models.

All in all, preprocessing improved the classification accuracy of the testing model by almost 0.4%. This can be significant but is typically lower than what variations in models and hyperparameters cause.

References

[1] J. Zayner, "We tweet like we talk and other interesting observations: An analysis of english communication modalities," 2014.

[2] R. Love, C. Dembry, A. Hardie, V. Brezina, and T. McEnery, "The Spoken BNC2014: designing and building a spoken corpus of everyday conversations," *International Journal of Corpus Linguistics*, vol. 22, p. 319, 2017.

[3] G. Jenks, "Python wordsegment," http://www.grantjenks.com/docs/wordsegment/.

[4] "Wikipedia list of emoticons," https://en.wikipedia.org/wiki/List_of_emoticons.

[5] "Slang dictionary," https://www.noslang.com.

[6] "Contractions dictionary," https://stackoverflow.com/questions/43018030/replace-apostrophe-short-words-in-python/.

[7] K. Parikh, "Contractions dictionary," https://gist.github.com/ParikhKadam/5eb89c8551dce2b2360c9bed707d8076.

[8] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," https://nlp.stanford.edu/projects/glove/.

[9] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *Signal Processing, IEEE Transactions*, pp. 2673–2681, 1997.

[10] J. Deng, L. Cheng, and Z. Wang, "Self-attention-based BiGRU and capsule network for named entity recognition," https://arxiv.org/pdf/2002.00735.pdf.

[11] P. Zhou, W. Shi, and J. Tian, "Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification," https://www.aclweb.org/anthology/P16-2034.pdf.

[12] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *CoRR*, vol. abs/1610.02357, 2016. [Online]. Available: http://arxiv.org/abs/1610.02357

[13] "Textblob spelling correction," https://textblob.readthedocs.io/en/dev/quickstart.html/.

[14] M. Michailidis, "Sentiment140," https://www.kaggle.com/kazanova/sentiment140.

[15] X. Zhang, "Yelp and amazon pretraining data," https://drive.google.com/drive/folders/0Bz8a_Dbh9Qhbfll6bVpmNUtUcFdjYmF2SEpmZUZUcVNiMUw1TWN6RDV3a0JHT3kxLVhVR2M.