Tweet Language Identification

Neural Networks Practical Course - KIT
28 February 2018

Dominik Sauter, Alexander Heilig, Tabea Kiupel
{dominik.sauter, alexander.heilig2, tabea.kiupel}@student.kit.edu

Abstract

The goal of this project was to develop a neural network-based solution for the language identification of tweets from Twitter. A tweet is a socially-oriented short text, which may contain abbreviations, misspellings or special textual segments, such as hash tags or URLs. This poses distinct requirements to the language identifier compared to regular text, e.g. from news articles. Since big datasets are available from Twitter (T, 2015), a neural network-based approach is especially suited for this task. First, a character embedding via a Skip-Gram model with Negative Sampling is calculated. Subsequently, the embedded characters of a tweet are used as input to a Recurrent Neural Network (RNN) model in the form of a bidirectional Gated Recurrent Unit (bi-GRU). After evaluation on the test set, the trained RNN model may be further tested in an interactive terminal with arbitrary input text or live tweets fetched directly from Twitter. Our approach thereby showed good performance on all tested datasets. The project was conducted in the context of the Neural Networks practical course at the Karlsruhe Institute of Technology (KIT) [1] , Germany. Its source code is publicly available under the MIT license on GitHub [2] .

Keywords: Twitter, tweet, short, social, text, language, identification, neural, network, kit

1. Introduction

There has been growing interest in understanding the content of social media, e.g. from Twitter or Facebook. This understanding is important for many reasons: Fighting spam or hate speech, grasping a user's interest and improving search engines, like from Google or Yahoo. There are many steps involved in a typical natural language processing (NLP) pipeline. One of the first steps is the language identification - determining the language in which a text, e.g. a tweet or a post, is written. This fundamental step is a prerequisite for correct further processing.

Earlier approaches, like Grefenstette (1995) used a small and simple model for identifying the language of news articles and achieved good accuracy scores for a limited set of European languages on sentences with 20 and more words. However, tweets are different from such news articles or web pages. They are especially short and do not use a formal language. Twitter users must squeeze their thoughts into originally only 140 characters, which was slightly increased to 280 in November 2017 (Kuri, 2017). The language in tweets often significantly differs from common language: The users may use abbreviations, special symbols like "#" or "@" and also emojis for expression.

In this work, we developed a neural network-based solution for the task of language identification of tweets from Twitter. Our resulting network approach is oriented on the model of Jaech et al. (2016). Additionally, we added functionality for fetching our data from Twitter and a terminal for interactive testing of trained models.

1.1. Structure

The organization of the rest of the paper is as follows: In section 2., we present an overview over our general system architecture. Important design decisions are described in section 3.. Subsequently, the individual parts of the architecture are described in the following sections: In section 4. we address data fetching and processing followed by the character embedding in section 5.. Then we describe our network in section 6., before presenting our results and evaluation in section 7.. Finally, we present our terminal in section 8. and draw a conclusion in section 9..

[1]https://www.kit.edu
[2]https://github.com/DominikSauter/NNLangID

2. Architecture Overview

In fig. 1 the general architecture of our approach is shown. It can be subdivided into three major parts with individual modules, which will be explained in more detail in the following sections.

In the *Data processing* part we use the `TweetRetriever` to establish a connection to Twitter and fetch tweets as data for training and testing our model. After storing the data, we perform different preprocessing steps in `DataSplit` and `InputData`. Subsequently, we calculate the embedding for the characters and train our network in the *Training* part. The resulting embedding weights as well as the trained model checkpoint are saved to file and may be used later by the `Terminal` in the *Evaluation* part.

3. Important Design Decisions

As there are already many different approaches for identifying the language of tweets or short utterances, we got an insight into possible challenges for this task (see Chang and Lin (2014), Balazevic et al. (2016) Zazo et al. (2016), Lopez-Moreno et al. (2014) and Tromp and Pechenizkiy (2011)). Therefore, we made some important design decisions, which are explained in the following sections.

3.1. Training, Validation and Test Data Files

In the field of machine learning a base dataset is usually randomly split into three disjunct subsets for the use for

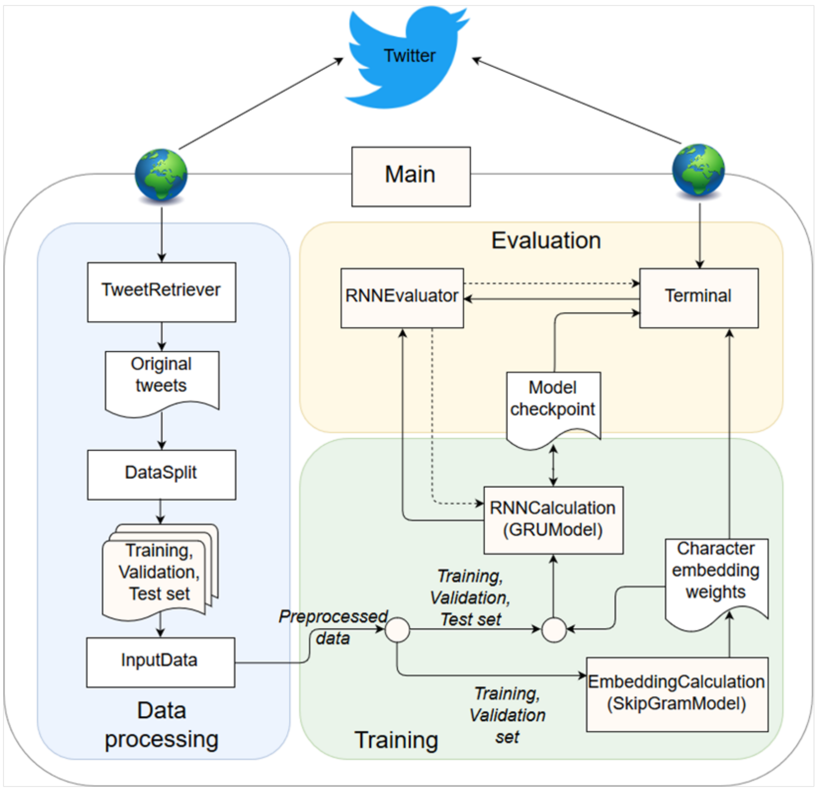


Figure 1: General system architecture

training and evaluation of models.

- Training set: The dataset used for training the model. The parameters of the model are iteratively learned while training. In the case of artificial neural networks the parameters are the connection weights (Ripley, 2007)(Shah, 2017).
- Validation set: This dataset is used to tune the parameters of the model, i.e. to find the optimal hyperparameters for a neural network (Ripley, 2007)(Shah, 2017).
- Test set: The test dataset is used to provide the evaluation of the final model. It is only used once, when the model is completely trained and shows generalization capabilities (Ripley, 2007)(Shah, 2017).

To ensure this clear separation between training and evaluation, we decided to split a received original tweets file into three separate files, for training, validation and testing, respectively.

3.2. Filtering

Prior to working with the data retrieved from T (2015), the tweets are filtered by removing parts which would add noise to the training for language prediction.
This mitigates confusion such as a tweet containing ”#MANvsFCB”, where we cannot predict the language this hash-tag belongs to without further background knowledge. It could be, that ”MAN” stands for the soccer club Manchester United. In that case, the tweet would probably be an English one. But having a look on the second part, ”FCB” can stand for FC Bayern Munich, a German soccer club, or FC Barcelona, a Spanish club. So it is not clear, if the tweet should be labeled as a German, a Spanish or an English one.
Therefore, removing irrelevant and ambiguous parts for language prediction helps to improve the overall learnability.

3.3. Character-Level Embedding

A major design decision was how to learn languages by. When learning the language of a tweet, it is possible to do so on a word and/or character basis.
While word-based prediction has the advantage of discerning between languages by utilizing the different vocabularies, which are more distinct than characters used over multiple languages, building such vocabularies relies on the orthography of words. This, however, cannot be guaranteed on tweets written by different Twitter users, where countless misspellings and word abbreviations would need to be included in these vocabularies to guarantee the correctness on the small amount of data that one tweet provides. On the other hand, character-level embedding offers the flexibility to not have to take misspellings and the like into account, as small word differences will only have an influence on the language prediction near the characters where the difference occurred.
To decrease the vocabulary and thus the embedding size and increase the accuracy when accounting for different spellings used in the same language, this work uses a character-level embedding to train and predict tweets.

3.4. RNN as Bidirectional GRU

As already mentioned, our approach is oriented on the model of Jaech et al. (2016). Jaech et al. are using a RNN in the form of a bidirectional Long Short-Term Memory (bi-LSTM) to map a sequence of word embedding vectors to a language label. In Schuster and Paliwal (2673 2681), the bidirectional RNN (bi-RNN) was introduced to overcome the limitations of a regular unidirectional RNN. A bi-RNN can be trained using all available input information from both the past and future of a specific time frame.
However, training on our huge Twitter data corpus on the character-level had to be done very efficiently. Our first approach was simply to use a bi-LSTM like (Jaech et al., 2016). But in (Chung et al., 2014) they compared a LSTM and the simplified version of a Gated Recurrent Unit (GRU) in the task of polyphonic music and speech signal modeling. The GRU outperformed the LSTM on all the datasets for the music modeling. For speech modeling, the performance was similar to the LSTM. They also found out, that in most cases the convergence for the GRU is often faster and the final solutions tend to be better.
Based on this results, we decided to use a bi-GRU instead of a bi-LSTM in our approach.

3.5. Modularization

In large and complex pieces of software there may be million lines of code. In such an environment, it is easy to lose track of what a particular part of code does and it is also hard to debug. With the use of modular programming, the code will be organized based on the task it executes.
Modular programming is a development paradigm that emphasizes self-contained, flexible and independent pieces of the functionality (Hare and Kaplan, 2017). The resulting modularity of a software offers several advantages to developers and users: In particular, functionality can be dynamically loaded and unloaded depending on the particular use case (Jackson, 2015). The code becomes reusable and easier to manage. Such managed code has a better readability, too. All these advantages result in a higher reliability.
To benefit from these advantages, we decided to modularize our different parts of the language identification task. Another reason for this decision was that we wanted our different steps, such as preprocessing and training, independently from one another. So we can use results from one step in our processing pipeline for another step, without executing our whole pipeline. In fig. 1 we have the following modules:

1. *Data processing*:
   - `TweetRetriever` for retrieving the tweets from Twitter and storing them in original files.
   - `DataSplit` to split the original files into training, validation and test set files.
   - `InputData` for the preprocessing of the tweets in the datasets.
2. *Training*:
   - `EmbeddingCalculation` to calculate the embedding and get the character embedding weights.
   - `RNNCalculation` for training the RNN model to get a trained model checkpoint.
3. *Evaluation*:
   - `RNNEvaluator` to evaluate the trained RNN model checkpoint.
   - `Terminal` for testing a trained model checkpoint with arbitrary input text or live tweets from Twitter.

4. Data Fetching and Processing

In the first step of our language identification task we need to fetch the data which we are going to use for training and testing from Twitter. Subsequently, it has to be preprocessed in a way, that yields an easily learnable and readily usable representation for the embedding and RNN.

4.1. Data Fetching from Twitter

First of all, we fetched the data from T (2015), which are tweets collected in July 2014 that cover 70 languages. The tweets are separated into three different sets: *uniformly sampled*, *recall-oriented* and *precision-oriented* dataset. However, we chose to only use the *uniformly sampled* and *recall-oriented* sets, since the *precision-oriented* set is not human- but machine-labeled and thus may not be very reliable. It was labeled by an old version of Twitter's internal language identifier and only determines if a tweet is e.g. German or non-German. The characteristics of the used datasets are listed in tab. 1.

	Uniformly sampled	Recall-oriented
Languages	67	70
Tweets	ca. 55,000	ca. 53,000
Distribution of languages	Close to real Twitter distribution (36% English, 17% Japanese, 11% Spanish, ...)	Nearly equally distributed

Table 1: Used datasets from T (2015) and their characteristics.

The *uniformly sampled* dataset was created by taking uniform samples on Twitter. Therefore, it represents the real distribution of languages on Twitter with the most common ones being English (36 %), Japanese (17 %) and Spanish (11 %), and some languages very sparsely or not at all represented in the set. The equally distributed *recall-oriented* set provides more insight into the task of learning all languages equally well. In this dataset we have enough tweets for each language to perform the language identification task for all languages quite well. Finally, to improve the results for the most common languages and to get a real big dataset, we used the union of the *uniformly sampled* and the *recall-oriented* dataset, yielding the *uniformly-recall-merged* set.
For interactive testing of our model, we decided to fetch public live tweets in the terminal, which adds another

source of data. Options are fetching random tweets or tweets filtered by keywords or a certain language.

4.2. Input Data Processing

After fetching the relevant data, we need to split and preprocess it to be readily used by subsequent steps.

4.2.1. Splitting

The splitting of an original tweets file into separate training, validation and test set files is performed in three consecutive steps with intermediate sets:

1. Language sets:
   At the beginning the original tweets file is split into sets for each single language called language sets.

2. Ratio sets:
   Then each language set is splitted according to a predefined splitting ratio (hyperparameter: `tr_va_te_split_ratios`), where we used the ratios 80% training, 10% validation and 10% test throughout our evaluation to get a large training corpus. After this step there are three different ratio sets for each language.

3. Merged sets:
   Finally, the individual ratio sets of each language are merged together, e.g.: 80% English + 80% German + 80% Spanish + ...
   The result are the three merged sets for training, validation and test. Fig. 2 visualizes an example of this merging step.

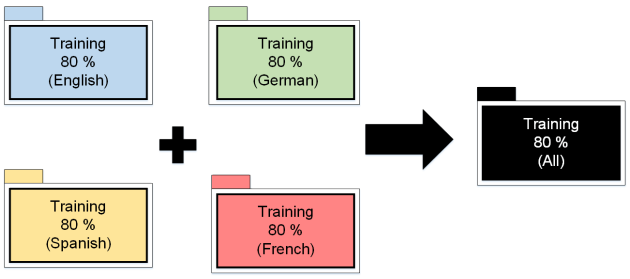


Figure 2: Example for merging language ratio sets over different languages retrieving the final training set.

This way, we can ensure that each language actually occurs in each set with frequency according to the splitting ratios.

4.2.2. Filtering

As mentioned in section 3.2. we made the decision to filter out irrelevant parts of tweets which would add noise to the language indentification task. For this reason we removed hash-tags, "@-names" and URLs:

- Hash-tags:
  Typical characteristic of tweets are the frequently included hash-tags, starting with the "#"-symbol, e.g. "#neuralnetworks". They are included to indicate the association of a tweet with different topics. But as they mostly consist of named entities or English terms independent of the actual language of a tweet, they are of little use for the language identification task. Therefore, we chose to replace them by a single "#" to reduce their influence but keep logical sentence structures.

- "@"-names:
  The "@"-symbol followed by a named entity is a special expression on Twitter, which is used to direct the tweet to the named entity called after the "@"-symbol, e.g. "@JohnDoe how are you?". As the "@"-symbol is only followed by named entities, there is almost no relation to the language of the tweet and therefore of no use for the language indentification. To keep the logical sentence structure they are not completely removed but reduced to a single "@"-symbol signifying a named entity at the given position.

- URLs:
  Sometimes tweets will refer to a website by linking a URL. As the model presented in this work is based on a character-level prediction, long URLs would pose drastic noise since they mostly do not correlate with the language of a tweet at all, e.g. "http://bit.ly/2C6Ugjt". However, to maintain possible logical structures, URLs are replaced by the otherwise seldomly used "_"-symbol.

4.2.3. Index Representation

The last step for readily usable data for embedding and RNN is to create vocabularies for characters and languages from the training dataset and transform the tweets into an index representation.
Every character that occurred at least with a predefined minimum frequency in the original dataset (hyperparameter: `min_char_frequency`) and also occurs in the training set, will be added to the vocabulary. The same procedure is done with the languages, except there is no occurrence threshold applied. Both vocabularies are filled with tuples in the form of: $(onehot\_index, number\_of\_occurrences)$.
The vocabularies are then used to transform the tweets into an indexed form. Only the characters which are in the character vocabulary will remain and be replaced by their unique onehot-indices. The annotated languages will be replaced by their unique onehot-indices, too. See fig. 3 for an example:

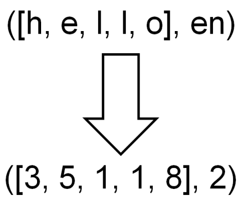


Figure 3: Example of the transformation into the index representation.

5. Character Embedding

As we want to feed a whole tweet character by character into the RNN, this has to be done very efficiently. Therefore, we chose to implement a character embedding for already providing low dimensional input vectors to the RNN.

5.1. Skip-Gram Model with Negative Sampling

For the embedding calculation we chose to implement a Skip-Gram model with Negative Sampling as suggested by Mikolov et al. (2013). Since the original proposed Skip-Gram model is designed for word-level embeddings, there are some slight adaptations for working on the character-level. In our case, given a target character $w(t)$, our Skip-Gram predicts its surrounding context characters $w(t-2)$, $w(t-1)$, $w(t+1)$ and $w(t+2)$ (see fig. 4 and 5).

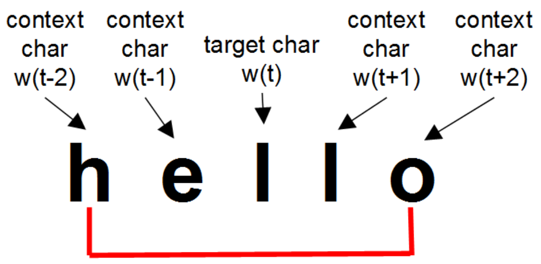


Figure 4: Example of a context window of size 2.

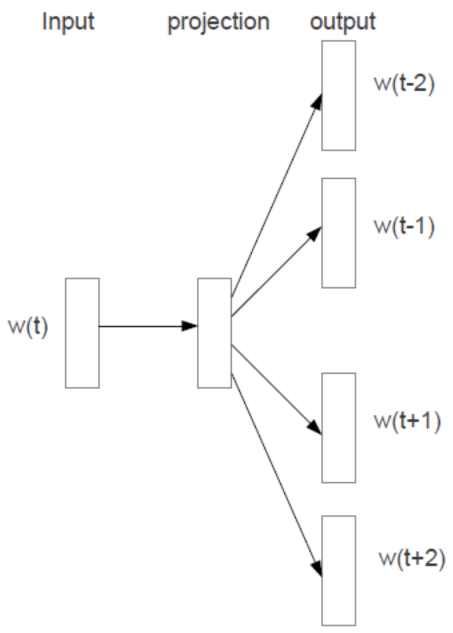


Figure 5: The Skip-Gram model architecture as presented by Mikolov et al. (2013).

Concrete context window sizes are sampled within a specified maximum context window (hyperparameter: `max_context_window_size`). Therefore, characters which are further away from the target character get lesser weight. Thereby, direct neighbours are always(!) sampled, while the maximum context window is chosen with probability: 1/`max_context_window_size`. This supports the fact, that characters which are further away are more likely to vary than those that are closer to the target character, and therefore should be less important in the embedding creation.

When using a Skip-Gram, there is always the issue of the vocabulary-sized output (context) vectors for which Softmax has to be calculated. Since calculating Softmax over large vectors is inefficient, training in a naive way is not feasible for large vocabularies. Therefore, two common solutions exist to address this problem. The first is called ”Hierarchical Softmax”, which computes Softmax in a tree-like fashion, thereby reducing calculation costs. The other is called ”Negative Sampling”, which is faster for training and yields better embedding vectors for frequent words than Hierarchical Softmax according to Mikolov et al. (2013). Therefore, we chose to use Negative Sampling in this project. It simplifies training by only updating the weights attributed to the 1-position of the output vector and some randomly sampled 0-positions (”negative samples”), instead of all of them. As some languages are only sparsely represented in our datasets, we chose to use 5 negative samples throughout our trainings, as suggested for small datasets by Mikolov et al. (2013). However, this is also a hyperparameter (`num_neg_samples`), and may be freely chosen by the user in our implementation.

The calculation is then defined by the Negative Sampling Objective (NEG) function:

$$\log \sigma({v'_{w_O}}^\top v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-{v'_{w_i}}^\top v_{w_I}) \right] \tag{1}$$

It yields the loss for a given input vector's 1-position weights (v_{w_I}) paired with a given output vector's 1-position weights (v'_{w_O}) and sampled 0-positions weights (v'_{w_i}). The NEG function itself is based on the Noise Contrastive Estimation (NCE). Further details may be found in the paper by Mikolov et al. (2013).

The probability $P(c)$ according to which a character c is chosen as a negative sample is given based on its frequency f_c as suggested by Mikolov et al. (2013):

$$P(c) = \frac{(f_c)^{\frac{3}{4}}}{\sum_C (f_c)^{\frac{3}{4}}} \tag{2}$$

For efficiency purposes this probability is precalculated for every character in the vocabulary and a large sampling table is filled with an amount of each character according to its probability. Afterwards efficient sampling may be executed via `numpy.random.choice` on the filled table. However, the size of the sampling table showed to have a big performance impact, leading to drastically slow sample operations for the used size of 100M in the original `C`-code by Mikolov et al. (2013). Therefore, we chose to determine the size of the table in a dynamical way based on the character with the minimum frequency in the vocabulary. The user may specify an amount of this character that he wishes to be in the table (hyperparameter: `sampling_table_min_char_count`), whereas the amounts of the other more frequent characters are then determined based on that. Throughout our trainings we used an amount of 1, which leads to minimum overhead while at the same time guaranteeing that all characters are actually represented in the table and have a chance to be sampled. However, this approach may easily lead to exploding table sizes for some vocabularies. Therefore, we also introduced a maximum table size (hyperparameter: `sampling_table_specified_size_cap`) to cap the table size no matter how big it would be due to the previous calculation. Having both parameters, the user eventually possesses full control of the table size, and the performance impact it leads to. For the general Skip-Gram implementation, we also used some further PyTorch optimizations suggested in a guide by Sun (2017).

5.2. Training

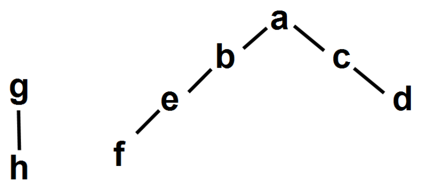
Training is done automatically until the model does not improve anymore or a specified maximum number of epochs is reached (hyperparameter: `max_num_epochs_embed`). The former is determined by evaluating the validation set regularly (hyperparameter: `eval_every_num_batches_embed`), thereby calculating the mean loss. If the loss improved, an embedding model checkpoint as well as the extracted embedding weights are saved to file. After a "patience"-period of no validation mean loss improvement (hyperparameter: `max_eval_checks_not_improved_embed`), the training is stopped. There is also a learning rate adaptation, which starts when half of the patience-period has passed. The learning rate is then decreased every further step of the patience-period by a factor (hyperparameter: `lr_decay_factor_embed`) until the loss improves again.

5.3. Evaluation

A common approach for word embedding evaluation is projecting the embedding vectors onto a 2D plane, to see which words are closely related. This is however not really practical for character embeddings, since character relations are not so intuitive to see, and especially since our vocabulary contains many to us unfamiliar characters from Arabian or Asian languages. Therefore, we used a different way of testing our model, by constructing some artificial tweets as follows:

```
"abababababababab"
"cdcdcdcdcdcdcdcd"
"acacacacacacacac"
"bebebebebebebebe"
"efefefefefefefef"
"ghghghghghghghgh"
```

The "relation"-graph in fig. 6 shows the (transitive) relations of the characters based on their vicinity and is therefore expected to manifest in the embeddings. To prove this,



$$\begin{tikzcd} & & & a \arrow[dl, dash] \arrow[dr, dash] & \\ g \arrow[d, dash] & & b \arrow[dl, dash] & & c \arrow[dr, dash] & \\ h & e \arrow[dl, dash] & & & & d \\ f & & & & & \end{tikzcd}$$

Figure 6: "Relation"-graph for the constructed tweets.

the differences between the embedding vectors of a trained embedding may be calculated, e.g. for `g`:

$$\begin{array}{lcl} sum\left\{\left|v_g - v_h\right|\right\} & \approx & 0.13 \\ sum\left\{\left|v_g - v_a\right|\right\} & \approx & 8.82 \\ sum\left\{\left|v_g - v_b\right|\right\} & \approx & 6.30 \\ sum\left\{\left|v_g - v_c\right|\right\} & \approx & 9.39 \\ sum\left\{\left|v_g - v_d\right|\right\} & \approx & 7.54 \\ sum\left\{\left|v_g - v_e\right|\right\} & \approx & 7.86 \\ sum\left\{\left|v_g - v_f\right|\right\} & \approx & 6.51 \end{array} \tag{3}$$

This clearly shows that `g` and `h` are closely related (difference almost 0), while `g` is not related to the other characters (differences all above 6). For a more detailed evaluation, e.g. the differences for `f` may be observed:

$$\begin{array}{lcl} sum\left\{\left|v_f - v_f\right|\right\} & = & 0 \\ sum\left\{\left|v_f - v_e\right|\right\} & \approx & 1.73 \\ sum\left\{\left|v_f - v_b\right|\right\} & \approx & 4.87 \\ sum\left\{\left|v_f - v_a\right|\right\} & \approx & 7.83 \\ sum\left\{\left|v_f - v_c\right|\right\} & \approx & 10.85 \\ sum\left\{\left|v_f - v_d\right|\right\} & \approx & 9.00 \end{array} \tag{4}$$

Here, a strong trend of increasing embedding differences can be observed the further the respective character is away from `f` in the relation-graph (apart from some irregularity at `d`). This trend of increasing differences can roughly also be observed for all other characters when moving away from them in the relation-graph. Consequently, based on this simple but fundamental evaluation, we are confident of a correctly learning model.

6. RNN

The previous processes lead to the training of the main language identification network. For this network, the first choice to be made was its general type. The task presented in this work is to ascertain the relation between different characters relative to the underlying language the input is written in. The natural choice for a network that takes into account past predictions is a Recurrent Neural Network (RNN). Due to the shortcomings in standard RNNs to remember long-term dependencies due to the vanishing gradient issue (Schuster and Paliwal, 2673 2681), a special type is needed.

6.1. Bidirectional GRU Model

A Gated Recurrent Unit (GRU) is a derivative of Long Short-Term Memory (LSTM) cells. GRUs extend typical RNN cells by adding two gates to the cell as shown in fig. 7. The reset-gate r_t determines how to combine the new input x_t at time t to the previous output h_{t-1}, which represents the GRU's memory, for the activation of x_t. The second gate z_t has the property of being able to ration how much of the previous memory is kept for the new final output h_t. This is achieved by splitting the result of the z_t sigmoid layer in a z_t and a $1 - z_t$ direction.

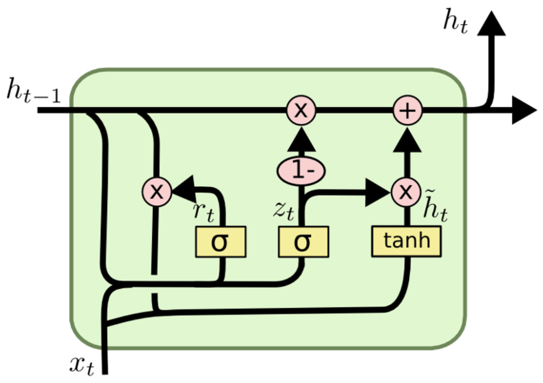


Figure 7: Outline of a Gated Recurrent Unit (GRU) (Olah, 2015).

For the task of a character-based language prediction, the GRU was implemented in a bidirectional way. Thus, not only the previous characters were taken into account, but also the subsequent ones. This does not only improve the network's performance, but also reduces the time it takes to train it (Graves and Schmidhuber, 2005).

6.2. Data Stream

As previously mentioned, the network learns a set of languages l on a character basis. Therefore, the batch input size of the bi-GRU layer equals the number of characters c in each input tweet, while each character input size is equal to the embedding dimension, as the character embeddings described in section 5. are fed into the RNN. After forwarding the input through the bi-GRU, it goes through another layer which computes the logarithmic softmax function for that bi-GRU i:

$$f_i(x) = log(\frac{exp(x_i)}{\sum exp(x_j)}) \tag{5}$$

The model then outputs a character-based prediction for all languages based on eq. 5. The model's output is a $(c \times l)$-dimensional tensor where each character is associated with the logarithmic probabilities for each language learned.

6.3. Weight Update

For training purposes, the network's weights need to be updated by using a measurement of the error in each training step. The model presented in this work uses the Negative Log Likelihood Loss (NLLLoss) described in PyTorch (2018). Due to taking the logarithm of the likelihood function, the NLLLoss can compute the loss with sums instead of multiplications. Furthermore, the logarithm transforms very small numbers (i.e. likelihoods < 1) to relatively large numbers, for which the computer precision will be higher. This results in the NLLLoss being numerically stable as well as being easier to compute.

The resulting loss is then fed to an optimizer, which updates all weights in the model respective to the error in the training step. This model uses the Adam optimizer, first introduced by Kingma and Ba (2014). The Adam optimizer, similar to other optimizers such as Adagrad, maintains individual learning rates for all weights in the system. Furthermore, additionally to tracking an exponentially decaying moving average of past squared gradients v_t like Adagrad, it also keeps an exponentially decaying moving average of past gradients m_t as described in eq. 6. β_1 and β_2 are parameters which are set at the initialization of the optimizer. This model uses the default values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_1^t} \end{aligned} \tag{6}$$

These two momentums are then used to update each weight by eq. 7. η describes the learning rate and is initially set to `initial_lr_rnn` $= 0.01$ in this model. ϵ is a sufficiently small number added to the denominator in order to increase the numerical stability.

$$\theta_{t+1} = \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{7}$$

The Adam optimizer has been shown to converge to the optimal weights faster than its competitors such as Adagrad or RMSProp (Kingma and Ba, 2014). Furthermore, the Adam optimizer integrates optional L2 regularization, which holds a penalty to large weights and therefore acts against overfitting. The squared weights are summed up, multiplicated with a weight decay factor (hyperparameter: `weight_decay_rnn` $= 0.00001$) and finally added to the error itself, i.e. the NLLLoss.

6.4. Training

The general training procedure is similar to the embedding. The neural network uses mini-batch training, so that it updates its weights every `batch_size_rnn` $= 10$ number of training steps. It is evaluated regularly on the validation set (hyperparameter: `eval_every_num_batches_rnn`) and a model checkpoint is saved to file, if the average loss of the validation set is lower than the previous lowest average loss of a past evaluation. Furthermore, a "patience"-period is implemented, thus, if the loss has not improved over `max_eval_checks_not_improved_rnn` $= 10$ validation set evaluations, the training is stopped. There is also the possibility to specify a maximum number of trained epochs (hyperparameter: `max_num_epochs_rnn`).

7. Evaluation & Results

To validate the embedding and RNN models described in section 5. and 6., they have been evaluated on different kind of sets derived from the data provided by T (2015). The data was trained on models with one layer of 100 bidirectional GRUs. For evaluation, this work compares the different models not only to each other, but also to the most common languages occurring on Twitter (Statista, 2013). It should be noted that Malaysian was excluded from the presented figures due to not being consistently marked in the initial datasets.

7.1. Evaluation Sets

The first set chosen for training was a subset of the *recall-oriented* dataset (in the following shortly: *Recall*) and contained the five uniformly distributed languages French, German, English, Italian and Spanish which share a similar (Latin) alphabet. It is called *5-Latin Recall*. The second set was the full *recall-oriented* dataset, a uniformly distributed set of 70 languages and over 53,000 tweets (see tab. 1). The third set was the *uniformly sampled* set (in the following shortly: *Uniformly*), containing 67 languages and over 54,000 tweets, too. Yet the distribution was chosen to represent the real language distribution of all tweets on Twitter (36% English, 17% Japanese, 11% Spanish, ...). The second and third set were disjunct, so they did not share any tweets with each other. The fourth training set was the conjunction of the second and third and contained over around 107,000 tweets called the *uniformly-recall-merged* set (in the following shortly: *Recall* + *Uniformly*). The reasoning behind putting the uniformly and real distribution together was to gain a large dataset for nearly all represented languages as described in section 4.1.. However, this set showed the limitations of the provided computing resources which lead to mid-way training abortion. Therefore, no final results on the test set could be retrieved for this set.

7.2. Results and Comparison

All sets were evaluated using embedding average validation loss, bi-GRU model test accuracy as well as each languages' test precision, recall and F1-score if available. The models were trained on the KIT-ISL cluster. Fig. 8 shows the results for the four trained datasets.

		5-Latin Recall	Recall	Uniformly	Recall+ Uniformly
Embed. loss (μ)		0.38	0.35	0.08	0.48
RNN accuracy (%)		88.4	74.4	88.6	75.0 (validation)
RNN precision	**μ**	0.884	0.749	0.416	n.a.
	σ	0.037	0.172	0.446	n.a.
RNN recall	**μ**	0.884	0.734	0.308	n.a.
	σ	0.026	0.178	0.357	n.a.
RNN F1-score	**μ**	0.884	0.738	0.340	n.a.
	σ	0.023	0.170	0.374	n.a.

Figure 8: Comparison of different sets on embedding average validation loss and RNN test set evaluation metrics.

The best test set accuracy was achieved by the *Uniformly* set with 88.6% as well as having the lowest embedding average validation loss of 0.08. Whilst the *Uniformly* set scored high on these metrics, its precision, recall and thus F1-score average are comparably low due to being skewed towards the higher occurrences of common languages in the *Uniformly* test set. From the 67 languages contained in this set, 33 of them were not learned at all (precision and recall score of 0.0). The *Recall* set only achieved a much lower accuracy of 74.4% and a higher embedding average validation loss of 0.35. However, the 70 languages had better precision, recall and F1-scores as the model had enough data to learn languages which are not often used on Twitter. The *5-Latin Recall* model consisting of five Latin languages showed good results overall with a test set accuracy of 88.4% and a F1-score of 0.884.

These results are especially interesting when comparing the *Uniformly* and *Recall* sets' ability to predict the most common languages on Twitter by comparing their respective F1-scores in fig. 9.

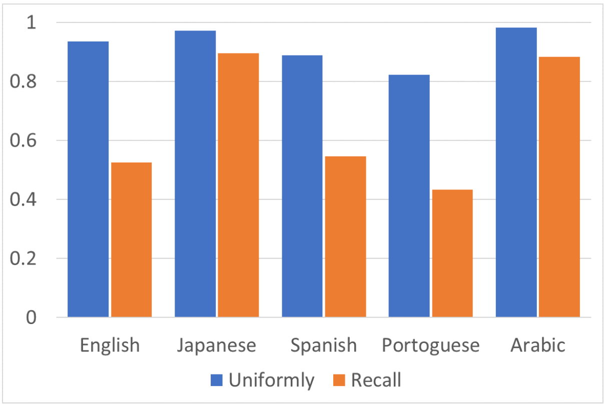


Figure 9: Comparison of F1-score on Twitter's most common languages (except Malay) (Statista, 2013).

As expected, the *Uniformly* set performed remarkably well in this area. The *Recall* set, on the other hand, achieved a good performance on Japanese and Arabic, yet suffered on the prediction of Latin languages English, Spanish and Portuguese. As the *5-Latin Recall* set performed better on a smaller subset of Latin languages, the large amount of languages with the similar Latin-based alphabet may be detrimental for the *Recall* set's ability to generalize these languages.

Despite this potential drawback, the *Recall* set still outperforms the *Uniformly* set when predicting lots of different languages. Fig. 10 shows the F1-scores for the *Uniformly*, *Recall* and *5-Latin Recall* sets for each of their trained languages summarized as a box plot figure.

The *Uniformly* set's huge variance in performance becomes evident as its F1-scores span from 0.0 to 0.981 (standard deviation of 0.374, fig. 8) while the *Recall* set only spans from 0.364 to 0.996 (standard deviation of 0.170, fig. 8). The small dataset of *5-Latin Recall* contains the lowest variance in the data and highest F1-score mean, yet never exceeds a score of 0.913.

8. Terminal

In addition to the evaluation of a trained model on the test set, we implemented an interactive terminal for manual entry of arbitrary input text or randomly fetched live tweets from Twitter. For the latter, some criteria, such as language or keywords, may also be specified for selective evaluation. The prediction of the five most probable languages is then shown for each input, sorted in descending order. Fig.

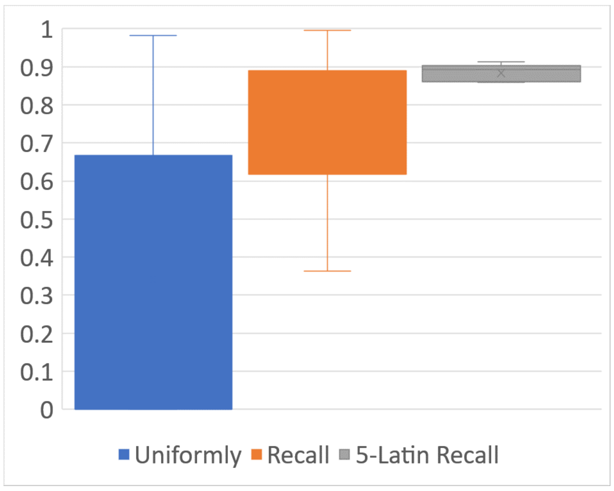


Figure 10: F1-score distribution over all languages.

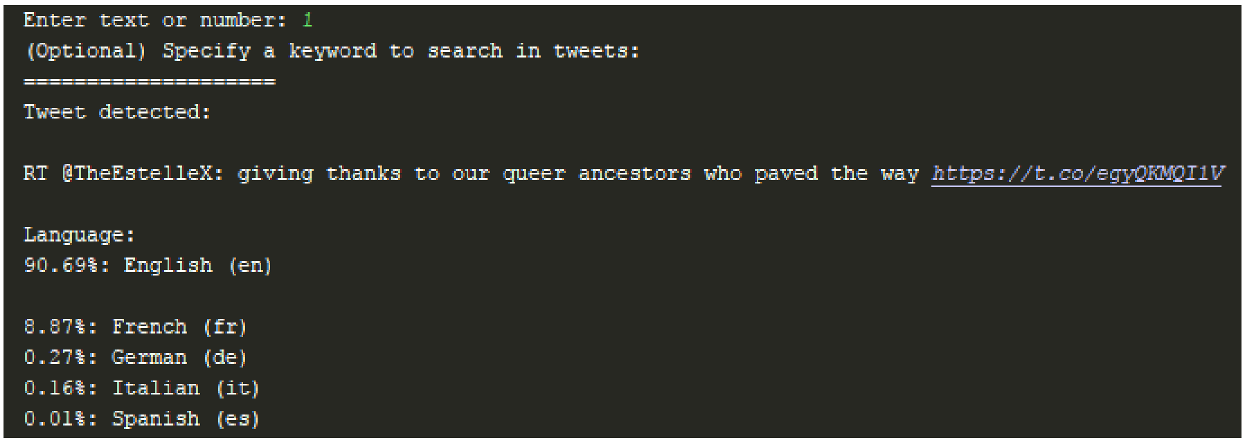
11 shows the prediction of a randomly fetched tweet from Twitter by using the *5-Latin Recall* trained model. Fig. 12 respectively demonstrates the prediction on the same model, but with a manually entered tweet.

9. Conclusion

The neural network-based solution developed in this work has successfully been able to predict the languages associated with tweets. The used training and evaluation data was provided by a Twitter blog post (T, 2015), which had been collected in July 2014 and was retrieved in December 2017. Due to the large time difference, many tweets were deleted and could not be retrieved. The *uniformly sampled* set, for example, was supposed to contain about 120,000 tweets, but retrieving them resulted in only around 55,000 tweets.
Our approach splits the provided datasets into training, validation and test set files, and the characters occurring in the data are fed into a Skip-Gram model to get a character embedding. The retrieved embedding weights are subsequently used to get the embedded characters of a tweet and feed them into the language identification network. The latter has been implemented using a bi-GRU yielding good results for all evaluated datasets.
One of the difficult aspects of the Twitter language identification task is the adherence to the online colloquial language inherent to Twitter. As language, especially colloquial language, changes over time and from generation to generation, a Twitter language identification model needs to be up to date to accommodate for new colloquialisms. Furthermore, our training data only consisted of the old 140-character tweets, whereas Twitter increased the maximum tweet character length to 280 in November 2017 (Kuri, 2017). These new and longer tweets therefore might hold more information for better training of the model, whilst also learning new expressions for different languages.
Another interesting extension to our approach would be using CNN-layers for filtering the embedded characters of a word to get a word-level representation, which is then used as input to the RNN. This approach was presented by Jaech et al. (2016), who also predict the language for every word instead of a whole tweet. Since usually the smallest entities which are attributed to a language are the single words and not individual characters, this could lead to further improvements of the results.

10. Bibliographical References

Balazevic, I., Braun, M., and Müller, K.-R. (2016). Language Detection For Short Text Messages In Social Media. *arXiv preprint arXiv:1608.08515*.
Chang, J. C. and Lin, C.-C. (2014). Recurrent-neural-network for language detection on Twitter code-switching corpus. *arXiv preprint arXiv:1412.4314*.
Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.
Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18:602–610.
Grefenstette, G. (1995). Comparing two language identification schemes.
Hare, E. and Kaplan, A. (2017). Designing Modular Software: A Case Study in Introductory Statistics. *Journal of Computational and Graphical Statistics*, 26(3):493–500.
Jackson, G. (2015). The Advantages of Modularization in Programming. `https://www.techwalla.com/articles/the-advantages-of-modularization-in-programming`. Access: 20.02.2018.
Jaech, A., Mulcaire, G., Ostendorf, M., and Smith, N. A. (2016). A neural model for language identification in code-switched tweets. In *Proceedings of The Second Workshop on Computational Approaches to Code Switching*, pages 60–64.
Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980.
Kuri, J. (2017). Twitter verdoppelt maximale Länge der Tweets auf 280 Zeichen. `https://www.heise.de/newsticker/meldung/Twitter-verdoppelt-maximale-Laenge-der-Tweets-auf-280-Zeichen-3883047.html`. Access: 14.02.2018.
Lopez-Moreno, I., Gonzalez-Dominguez, J., Plchot, O., Martinez, D., Gonzalez-Rodriguez, J., and Moreno, P. (2014). Automatic language identification using deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 5337–5341. IEEE.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *CoRR*, abs/1310.4546.
Olah, C. (2015). Understanding LSTM Networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.
PyTorch. (2018). Pytorch NLLLoss. `http://pytorch.org/docs/master/nn.html#nllloss`.
Ripley, B. D. (2007). *Pattern recognition and neural networks*. Cambridge university press.



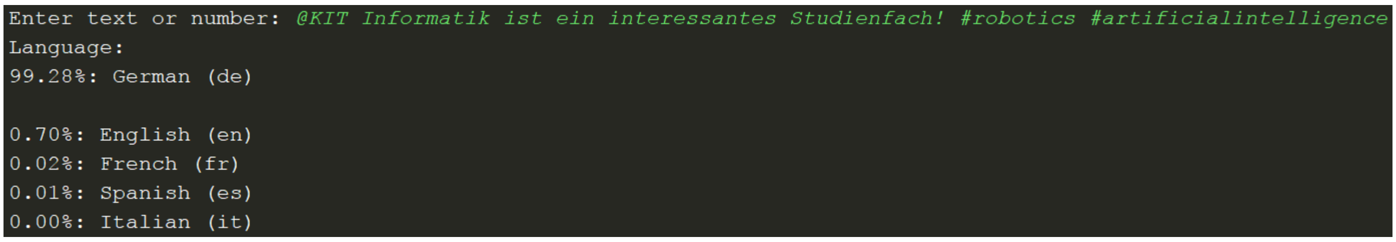
```
Enter text or number: 1
(Optional) Specify a keyword to search in tweets:
====================
Tweet detected:

RT @TheEstelleX: giving thanks to our queer ancestors who paved the way https://t.co/egyQKMQI1V

Language:
90.69%: English (en)

8.87%: French (fr)
0.27%: German (de)
0.16%: Italian (it)
0.01%: Spanish (es)
```

Figure 11: Online tweet language prediction.



```
Enter text or number: @KIT Informatik ist ein interessantes Studienfach! #robotics #artificialintelligence
Language:
99.28%: German (de)

0.70%: English (en)
0.02%: French (fr)
0.01%: Spanish (es)
0.00%: Italian (it)
```

Figure 12: Offline tweet language prediction.

Schuster, M. and Paliwal, K. (2673 - 2681). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45:1403–1409.

Shah, T. (2017). Train, Validation and Test Sets. `http://tarangshah.com/blog/2017-12-03/train-validation-and-test-sets/`. Access: 20.02.2018.

Statista. (2013). Twitter's most common languages of 2013. `https://www.statista.com/statistics/267129/most-used-languages-on-twitter/`.

Sun, X. (2017). Word2vec with Pytorch. `https://adoni.github.io/2017/11/08/word2vec-pytorch/`. Access: 11.01.2018.

T, M. (2015). Evaluating language identification performance. `https://blog.twitter.com/engineering/en_us/a/2015/evaluating-language-identification-performance.html`. Access: 20.01.2018.

Tromp, E. and Pechenizkiy, M. (2011). Graph-based n-gram language identification on short texts. In *Proc. 20th Machine Learning conference of Belgium and The Netherlands*, pages 27–34.

Zazo, R., Lozano-Diez, A., Gonzalez-Dominguez, J., Toledano, D. T., and Gonzalez-Rodriguez, J. (2016). Language identification in short utterances using long short-term memory (LSTM) recurrent neural networks. *PloS one*, 11(1):e0146917.