Christoph Stenkamp, Johannes Schrumpf

Assessing alt-right Twitter use through LSTMs using TensorFlow

Abstract

The current president of the United States of America, Donald J. Trump, and his policies has been a controversial topic in public and social media. The social media platform Twitter offers the possibility for individuals to publicly speak their mind about political and social topics regarding these policies.

Distinguishing between political posts and posts that are concerned with other matters is a crucial ability when dealing with social media.

We trained a **long short-term memory (LSTM) network** with an implementation of **word2vec**[1]-data, generated from tweets. These tweets consisted of political content as well as content unrelated to current political developments. The goal of the training process was to achieve a classifier that can distinguish between alt-right political and non-political tweets. Further, we programmed a generator that, among others, uses the previously trained classifier to create political tweets.

The resulting Twitter-bot is active, and its tweets can be found here: https://twitter.com/TrumpFacts8/with_replies. The source code is publicly available on GitHub: https://github.com/cstenkamp/iannswtf-project

The task

Our Twitter-bot consists of several parts. In this section, we will at first introduce every part, and afterwards go into detail about every one of them:
The Bot is mainly a **generator** for natural language. As for that, it learns the structure of the sequence of words from the input-data via a sliding-window over input strings, and then tries to mimic this structure by creating one word after the other while trying to keep the perplexity of following words as low as possible. Since the generated tweets are supposed to be similar to the training data, the bot also incorporates two **discriminators**: one for discriminating between actual sentences and gibberish, the second for discriminating between realistic trump-follower-tweets and non-trump-follower-tweets. Because we were interested if pre-training **word2vec** on the dataset had a positive impact on the classification, we incorporated that as well (the source code is taken mostly from the Tensorflow tutorial though). Because there doesn't exist a dataset fulfilling our wishes yet, we created the dataset ourselves – by building a **crawler**, automatically collecting tweets from selected users from the Twitter-API, and afterwards **preprocessing** the data to make learning on it possible. One important fact about this is its generality: the bot could classify and mimic any opinion on twitter, as long as account-lists of positive and negative samples (with a few thousands tweets each) are provided. It is also possible to let the generator run on IMDB-dataset by simply changing one variable.

1.1) Twitter crawler

Before gathering and filtering tweets, it is crucial to decide which accounts to crawl. This policy aims at obtaining low false-positive and false-negative rates. We manually searched Twitter for accounts that frequently use hashtags related to political opinions. Because some hashtags are being used ironically, we read the most recent tweets of specific

[1] https://arxiv.org/pdf/1301.3781v3.pdf

twitter accounts and determined if they are posting ironic tweets. Since this pre-selection is based on the author's political assessment and sensitivity to irony, the resulting list of accounts supporting Trump may be biased.

To obtain tweets we used the **tweepy-API**[2]. This library allows to download the latest 3240 tweets from individual accounts in 200-tweet batches. These batches are appended to a list of all tweets from an individual account. Using the *json*-library, *tweepy*-Twitter-objects can be filtered for their tweet content, disregarding all additional information such as geolocation, number of retweets etc. The filtered Twitter object content is then stored in a .txt file.

When the crawler has finished collecting tweets from the list of accounts, all twitter messages are merged into one .txt-file. This step occurs once for the list of accounts of Trump-supporters as well as for accounts posting unrelated content.

These two files are then passed through the preprocessor to obtain the final lists of filtered tweets.

1.2) Preprocessing

Even though Twitter-objects have already been broken down into their text-content, preprocessing is still a vital step to obtain easy to use data. To lessen semantic noise, we filtered tweets that contained any sort of URL from the total list of tweets (because grasping the meaning of the website would be far too much). Additionally, we filtered tweets that are shorter than 50 characters to increase semantic content of tweets that are being used for the network to learn. Further, some formatting- and special characters are erased or changed to an explicit word, such that the Twitter-dataset is structurally similar to the *IMDB Large Movie Review Dataset*[3], because of known success in sentiment-analysis on this dataset. As for that, we also split the data into training-, test- and validation- data. This data is then fed to the next step.

1.3) Creating the Dataset-object

Since the LSTM cannot handle real words, at first every word from the input-string get converted into an index. During that, words that occur only once are replaced by a special sign, since it is not possible to figure out semantic content for those words. Further, when creating the dataset, two lookup-dictionaries are created for easily converting a word into its index and vice versa. All of this is then stored in a dataset-object, which is a saved using python's *pickle*, enabling for faster recreation of the dataset.

1.4) (optional:) Pre-training word-embeddings on the dataset

According to [Kim 2014][4], pre-training word2vec-like word-embeddings leads to faster and better learning of natural-language models. In an effort to reproduce that, we incorporated an optional pre-training of those embeddings. Since this was not the main task however, the code for that is mostly taken from TensorFlow's GitHub-repository[5], with only minor changes to it, like rewriting the *generate_batch*-function, such that it can learn on multiple tiny strings (remember that Twitter-posts are maximally 140 characters), instead of one big dataset. Looking at those word-embeddings, or rather clusters of semantically close word-embeddings, is however really informative: Next to obvious Clusters like [I, you, we, they] or [woman, man, guy] or [good, bad] or [<dot>, <exclamation>, <comma>, <colon>, etc] there are also some Clusters which are specific to our dataset, for example Donald Trump's Name being not only close to other policitians, but also to his Twitter-account or the his Hashtag: [@realdonaldtrump, trump, #trump, hillary, obama]. Another interesting thing is for example "for" and "4" being close to each other, being a result of typical abbreviations used in Twitter. For more funny Clusters, take a look at appendix A.

[2] http://www.tweepy.org/
[3] Maas et al, 2014: Learning Word Vectors for Sentiment Analysishttp://ai.stanford.edu/~amaas/data/sentiment/
[4] Yoon Kim, 2014: Convolutional Neural Networks for Sentence Classification https://arxiv.org/abs/1408.5882
[5] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py

If word2vec is performed, the pre-trained word-embeddings are added to the *pickle*-file containing the dataset, such that the word-embeddings can later on be used from the classifier and the generator. In practice, we didn't see a great increase in performance when using pre-trained word-embeddings (see Appendix B.1 and B.2). One really interesting thing however is, that if input-strings are relatively long (in multiple examples we tried a maximum of ~80 words per tweet, which is far over average for 140-character-twitterposts, leading to a lot of strings consisting to far over 50% of the <END>-token), the network doesn't learn anything without pre-trained word-embeddings, and with them, the classification performance sometimes arbitrarily experiences sudden leaps from chance to ~80% classification rate, where it plateaus then (See Appendix B.3)

The Class making up the dataset additionally contains functions to read its content, iterate over it, as well as automatically shorten the strings inside it. Especially the latter function is very useful, since it allows for high generality: All strings longer than the string representing the X's percentile are either clipped or split, and all strings shorter than a minimal string length are left out. Afterwards, all strings are made exactly as long as X's percentile, by filling the strings up with a special <END>-token, represented by the zero vector.

When running the iterator over the dataset (for the language model and generator), the strings are split with another special token (<EOS>, end-of-string). That will later be very useful for the generator, as it learned how a tweet likely starts: with one of the tokens following <EOS>.

1.5) The discriminators

The bulk of the work went into the discriminators. The original network for them was implemented for the IMDB-Dataset (hence the occasional "reviews" instead of "strings"), because the generation of the dataset happened simultaneously to the implementation of the network. The discriminator is a two-layer stateful LSTM network. The embedding dimension is 128, which appears enough for the discrimination. During Training, dropout is performed with a probability of 0.5 to avoid overfitting, which is left out in the actual discrimination. While testing the network, we noticed that the Adam-Optimizer, without a specified learning rate, performs at least as well as a simple gradient descent optimizer with manual weight decay, which is why we settled for this, easier, version in the discriminator network. Our network performed around 85% on the IMDB dataset and equally well on the (very, very noisy!) Twitter-dataset. Since [Maas et al, 2014] report a performance of maximally 90% of sophisticated on the IMDB dataset, we consider our performance to be well enough to justify our choice of hyper parameters, especially since our network even occasionally deals with the rear parts of the split of strings, in part even starting in the middle of a sentence. We still decided to leave the split in there like this, for our network to be more generally applicable to different datasets.

The discriminator-LSTM-class contains several functions for multiple things from classifying a single string to letting the entire network learn from scratch.

I mentioned earlier that there are two discriminator-LSTMs. Those two networks use the same class, only training on different data: The one mentioned before discriminates between trump-follower-tweets and non-trump-follower-tweets, the other discriminates between tweets and non-grammatical gibberish. The implementation of this gibberish happens in the *create_random.py* function: in here, a lot of strings which look on the surface equal to a twitter-post but is a random combination of words gets created. The reason to discriminate between those and real-twitter-posts is, next to gaining knowledge (easily 99% accuracy on the training-set), gaining an understanding of grammar, which is necessary for sorting out tweets generated by the generator-network that are non-grammatical.

1.6) The generator

At last, however most crucial, there is the generator network. This network is basically a language model implemented with another two-layer stateful LSTM, roughly following the TensorFlow Tutorial for Language Modelling [6]on a subset of the Penn Treebank, however almost fully re-implemented and changed. The language model works like this: During training, a sliding window runs over the string, having a bag of words as input value, and an upcoming bag of words

[6] https://www.tensorflow.org/tutorials/recurrent

as target value, trying to optimize in the sense of keeping the "surprise" upon seeing the following target-bag-of-words low. For this, we noticed that the gradient descent optimizer performs better than the Adam Optimizer (and that the model trains generally longer than the discriminator, so probably the former is a result of the latter). Again, during training dropout turned out to be useful.

When generating text using this network, there is of course no need to use the target-values, instead we aim for the maximally probable expected follow-up word, or rather a probability distribution (the *softmax*) over them. To generate a text, we sample then over this probability distribution. It is important to reset the initial state of our stateful LSTM after each generated string, such that the generation of one string is not dependent on the previously generated string. Also, we were able to play with the probability of the <EOS>-token at certain lengths of strings, to keep the probability of strings that are too long or too short as low as we wish.

1.7) Putting it all together

The result of putting all this together can be seen in the *main.py* file. In here, all the necessary datasets are either read, or, if not existent yet, created, and a tweet is created at the call of the respective function. Also, there are many additional constraints which need to apply in order for a tweet to be considered post-able, as can be seen in the *main.py* method. Another feature of our implementation is, that we can abort a running network in the middle of the learning process and continue later on, thanks to a counter of iterations, stated in the *file_functions.py*

For an example of tweets generated by our network, please see Appendix C.

1.8) Related work

The work most closely resembling our type of network is the SequenceGAN by Lantao Yu et al[7]. The network implemented by them is however far more sophisticated, as the generator and the discriminator are trained simultaneously in a reinforcement-learning paradigm with Monte Carlo Tree Search over half-created sentences. Trying to copy that would go far beyond the scope of this task, however we do think that a similar paradigm could be implemented into our two networks, making them learn simultaneously instead of the discriminator simply forbidding certain strings from the generator.

The discriminator and language model are themselves very similar to existent implementations, whose code can be online, for the links please see the footnotes above.

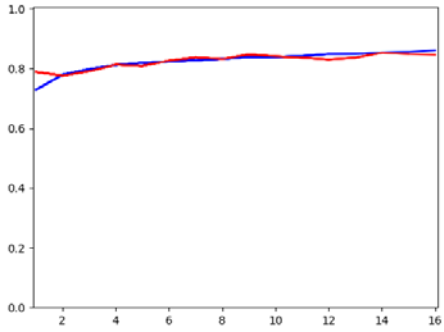
[7] Lantao Yu et al: SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient https://arxiv.org/pdf/1609.05473.pdf

Appendix A: Clusters of word2vec

```
Nearest to dude: <UNK>, generous, @jimlibertarian, foundations, @realvinniejames, youll, @neiltyson, vs,
Nearest to <comma>: <dot>, clearance, <semicolon>, @suzanbay, dobbs, networks, viciously, south,
Nearest to ronald: content, landowners, disable, @warrior4victory, biden, america, able, ninth,
Nearest to <semicolon>: <comma>, entrances, tolerant, @richardgarriott, nazis, clarification, blocked, @chasestraight,
Nearest to support: aspiring, motor, drums, reps, peo, face, @frozenyogurt3, <UNK>,
Nearest to reagan: ladies, names, dec, @bobafettfanclub, inauguration, karma, respected, slander,
Nearest to a: fart, humans, bashing, sworn, the, snakes, @djfood, attorneys,
Nearest to those: these, #trade, their, you, nnot, lots, coating, apparent,
Nearest to using: celebrating, keeping, questions, books, posting, @drudge_report, prevented, @aappres,
Nearest to you: u, we, mourning, they, +0, i, them, unleash,
Nearest to wing: seeking, kappa, thanksgiving, sells, makes, quality, happen, store,
Nearest to against: for, waking, refuses, mass, hasn, 9pm, continued, baldwin,
Nearest to evil: wod, uncool, @impsy, @marlibu, @wikileaks, filled, suggestions, showed,
Nearest to violation: progressive, whites, generous, donating, feed, cake, candidates, workforce,
Nearest to going: trying, emailed, losing, convince, #blogher11, #snowflakes, therapy, meal,
Nearest to but: so, alright, @jewschoosetrump, and, suffered, @thejuanwilliams, centers, btw,
Saved word2vec-Results.
Word2vec ran through 316896 different strings.
Close to ' 4 ':  ['for', '2', 'items', 'citizens', '5']
Close to ' evil ':  ['wod', 'uncool', '@impsy', '@marlibu', '@wikileaks']
Close to ' trump ':  ['he', 'hillary', '#trump', 'obama', '@realdonaldtrump']
Close to ' woman ':  ['eliminate', 'script', 'counteract', 'dictatorship', 'play']
Close to ' <dot> ':  ['<dots>', '<UNK>', '<exclamation>', '<comma>', 'appears']
Close to ' his ':  ['their', 'your', 'my', 'her', 'our']
Close to ' bad ':  ['@policycast', 'sad', 'art', 'spooky', 'shows']
Close to ' three ':  ['phoenix', 'festivals', 'filtering', '@rtoro20', 'consent']
```

Appendix B: Performance with and without pre-trained word embeddings

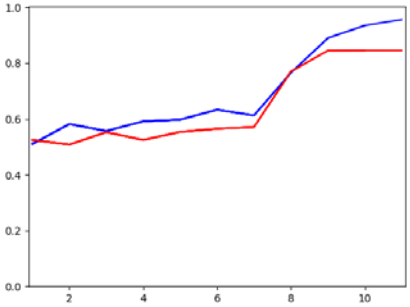
B.1) sample-performance without word2vec



x-axis: Iteration
y-axis: Classification performance

red graph: test-set
blue graph: train-set

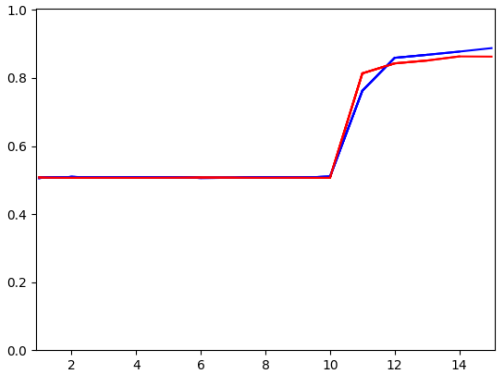
B.2) sample-performance with pre-trained word2vec



x-axis: Iteration
y-axis: Classification performance

red graph: test-set
blue graph: train-set

B.3) Performance with word2vec for very long strings



x-axis: Iteration
y-axis: Classification performance

red graph: test-set
blue graph: train-set

Appendix C: Generated tweets