Machine Learning Engineer Nanodegree

Capstone Project

Tatiana Gaponova (May 17, 2018)

Definition

Project Overview

The project was inspired by Toxic Comment Classification competition on Kaggle and is devoted to the problem of toxic online behavior. This problem is inherently important for the modern world where it is possible to easily reach much more people than it was only 15-20 years ago. In order to make online environment healthier and safer it is important to timely detect potentially harmful content. Perspective API already offers a model which can distinguish between toxic and nontoxic text (*toxicity is defined as a cause of someone's leaving a discussion*) but it cannot discriminate between different types of toxicity and can be further improved in terms of accuracy. In their article, they used logistic regression and variations of multilayer perceptron to tackle this issue achieving 0.966 ROC AUC score[1]. However, in one last-year paper has been shown that the system is sensitive to adversarial examples[2]: inserting additional punctuation or using other means of non-conventional formatting results in seeing toxic text as not toxic.

Problem Statement

The main objective is to create a model that will be able to process a chunk of text and output a probability of this text to be toxic. The model should also discriminate between different types of toxicity (*general toxicity*, ***severe toxicity***, *obscene*, *threat*, *insult*, *identity hate)*, because this will help us understand why the text is labeled as toxic and will provide the means of fine-tuning the final system (e.g. some users may tolerate obscenity but not identity hate).

The indented solution involves bidirectional LSTM neural network used together with word embeddings layer trained against twitter and wikipedia data.

---

[1] Ex Machina: Personal Attacks Seen on Scale, Wulczyn E., Thain N., Dixon L., Oct. 2016

[2] Deceiving Google Perspective API, Hosseini H., Kannan S., Zhang B., Poovendran R., Feb. 2017

Metrics

The problem falls into multi-label classification framework because one comment can manifest more than one type of toxicity. The final prediction table will have 6 columns for every toxicity type and n rows (number of examples in the test set). The performance for each toxicity type can be measured by ROC AUC (as long as the model outputs probabilities). This metric shows how close our probability estimates are to the actual labels by varying discrimination threshold and computing the area under ROC curve. There are 6 types of toxicity, so there will be 6 AUC scores, that can be averaged to get one score to represent general quality of the model.

ROC AUC score is indeed a good fit to the data and to the problem because this metric considers all possible thresholds, while metrics like accuracy usually uses the most natural threshold 0.5. This feature especially useful because, as will be shown in the next chapter, the dataset is very imbalanced. ROC AUC score helps to overcome the bias that will inevitably appear if we define only one threshold. Also the classification problem we are dealing with is probabilistic by nature. If we say that one comment is toxic, some other comment may be more or less toxic. Therefore, it is very important to deviate from hit/miss paradigm (which is basically setting only one threshold even if a model outputs probabilities) and explore less strict metrics. ROC AUC is a perfect tool in regards to this.

However, in addition to the ROC AUC score, I would like to also explore precision (how relevant are found items), recall (how many relevant items are found) and F1 scores at the threshold equal to 0.5. Despite that the ROC AUC already incorporates the notions of precision and recall, separate tracking may provide more insights about performance of the model.

Analysis

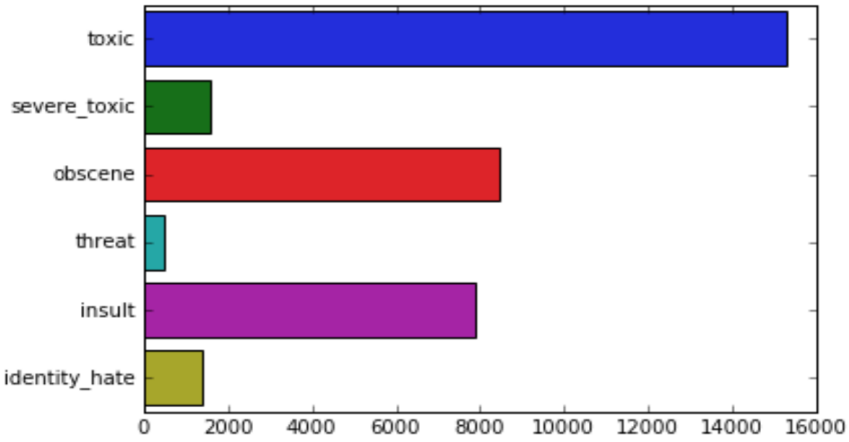
Data Exploration and Visualization

The dataset consists of 159571 wikipedia comments. Each comment is associated with 6 toxicity labels. A comment can fall into multiple toxic categories (like shown in the second example of the dataset snapshot). However, only 16225 comments (10 % of the data) are labeled as toxic.

comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
I shalt sever thy head at the neck. It will be dreadfully slow and painful, in accordance with Wikipedia policies.	0	0	0	1	0	0
Die Whore Die you whore.	1	1	1	1	1	0
I can accept the topic ban and understa	0	0	0	0	0	0

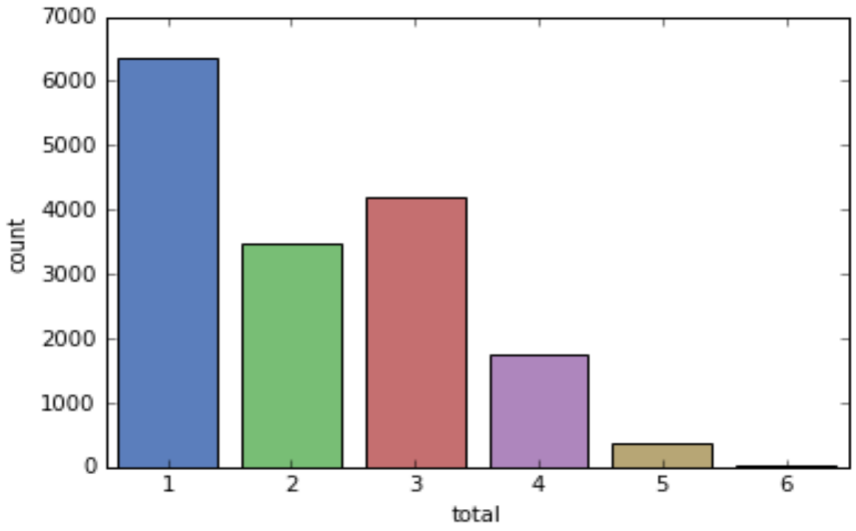
Pic. 1. Dataset snapshot

Let’s explore the distribution of toxicity among positively labeled examples. Pic. 2 provides a bar plot which shows that the dataset is very imbalanced. There are less than 500 examples of threat comments while obscene and insult categories have more than 8 thousands cases. As was shown earlier, one comment may be associated with multiple labels, therefore the total number of examples does not sum up to 16225.



Pic. 2. Toxicity distribution

Pic. 3. shows the label distribution. Less than a half of toxic comments have just one label associated with them, the majority of cases have two and more. Pic. 4. examines pairwise spearman correlations between the labels. Insult and obscene categories have the highest correlation score of 0.5, therefore they can often appear together.



Pic. 3. Label distribution

Pic. 4. Pairwise correlations between labels

The text itself also requires in-depth exploration. Text characteristics may provide a lot of hints and guidelines about dataset processing. Pic. 5-9 demonstrate various features that I observed such as presence of unique identifiers (usernames, emails, ips), presence of urls and various numbers. Toxic comments can also be repetitive, contain non-ascii chars and a lot of uppercase letters. Such features can be useful as they might be distinctive characteristics of toxic text.

```
'"(talk)]] \n\n There was absolutely nothing wrong with my edit. WTF???   \n\nThe pro
blem with your edit was that it violated your topic ban (again) which stood until nex
t month. Since this is the second time this has happened, you\'ve been blocked for a
longer term and the topic ban has been extended until 25 February 2016.  (talk) \n\n
Ah, so basically, [[User:Acroterion|"'
```

Pic. 5. Usernames

```
He is professor in Macedonia , member of NYAS, he publ. 160 scholarly art., 17 books in Int. Law,
  ULTRA VIRES act of UN in the process of admission of Macedonia in UN (published in AJIL, Vol.93.

http://www.mia.mk/en/Inside/RenderSingleNews/289/105947751
or at MINA   http://macedoniaonline.eu/content/view/21668/45/
or http://www.makemigration.com/iselenistvoweb/index.php?page=iselenici&id;=247&tip;_iselenici=7

http://s241910817.onlinehome.us/html/articles/janev/janev.html

or http://www.mkd.mk/makedonija/politika/nekoj-go-brishe-igor-janev-od-vikipedija
```

Pic. 6. Urls

```
u'0440567351 0445z 045jp7 046 04645 0467 047 048 0486407268 0
49 0495 049774 0498 04980 04arnols 04d 05 050 0500z 0501 0505
051211 0514560 0521270642 0525z 05263 0528dude 053 0535 0536
055 056359 0567 057 0576 058 058133 058162 0582493242 059 059
5292690 0595333001 05am 05d 05mm 05reading 05t10 05z 06 0600z
0601 0608 061 0612 0614 0615 0616 062 064 0642hrs 0645ad 0646
7 065 0650 0658 067 0674006623 0684822628 06848657680 069 069
1 069131994 06_22 06_december_2011 06am 06d 06t10 06t20 07 07
0 0704 070605 0707397 071 0711 0713990619 0714 0714631205 071
8 071828895 071d49 072 0722skull 073 0730z 0732224055 0738202
711 074 0742 0742562964'
```

Pic. 7. Numbers

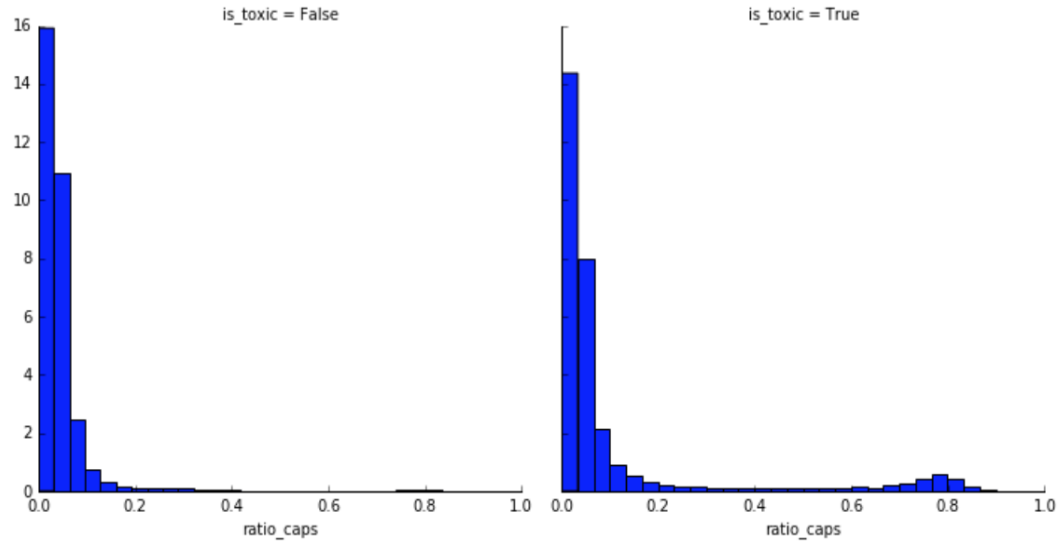
```
'FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U
YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELE
USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT
SS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK
T FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU
K YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USE
U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS B
LESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FU
BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YOU U USELESS BOT FUCK YO
```

Pic. 8. Repetition and uppercase letters

```
---
labels: toxic, obscene, threat
text: I ŵill diśṗośe of ŷour body by deṗosiţinǵ iţ in a duṃṗ ŵhere iţ belonǵś.
---
```

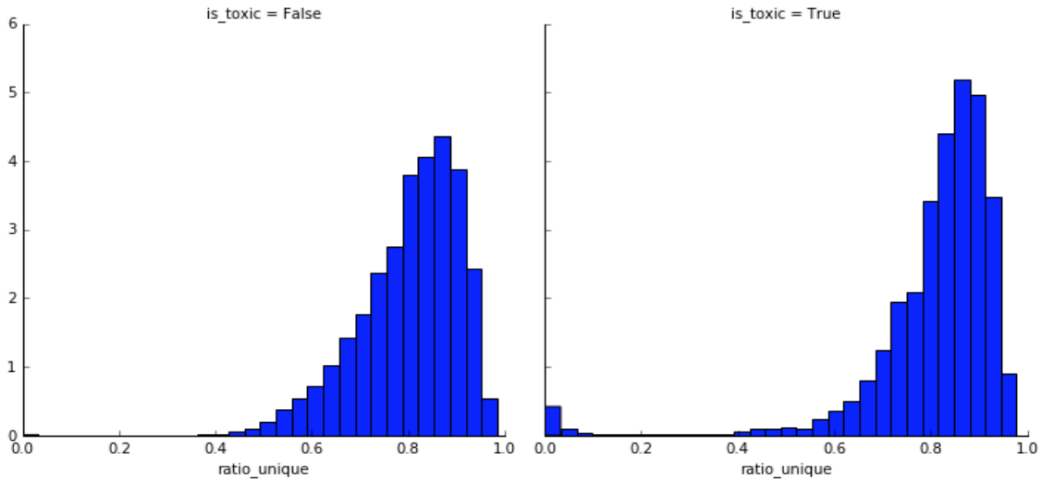
Pic. 9. Non-ascii characters

Pic. 10 explores the ratio between number of uppercase chars and total number of chars in a comment. X axis represent the ratio and Y axis represent the normalized number of cases. As we can see, both toxic and non-toxic text follow similar patterns. However, toxic text has distinctive tail at the end, which means that some number of toxic comments are written almost entirely in uppercase. Pic. 8 perfectly illustrates this phenomenon.



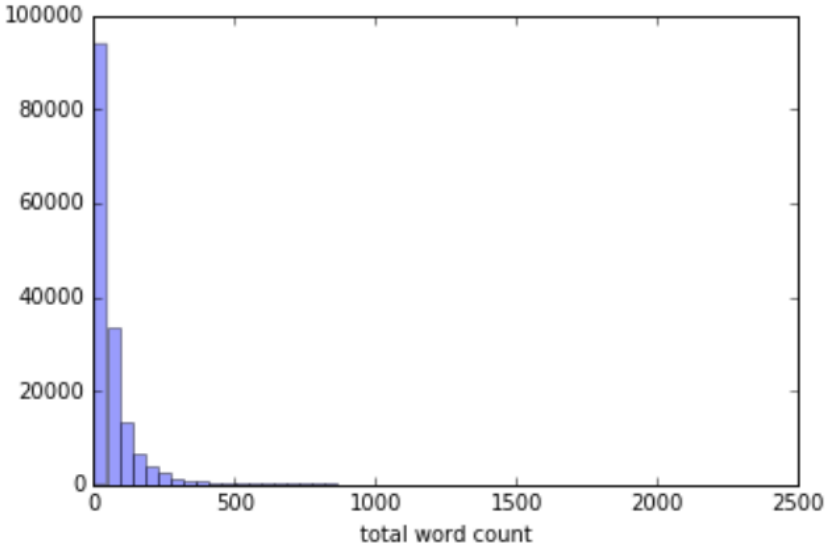
Pic. 10. Ratio between uppercase chars and total number of chars

Pic. 8 is also a good example of repetitiveness. Pic. 11 shows that the repetitiveness situation is similar to the uppercase situation: toxic distribution is a little bit narrower, but general patterns are alike. There is also a tail at the end of the toxic distribution, which means that some toxic comments consist only of couple of unique words.

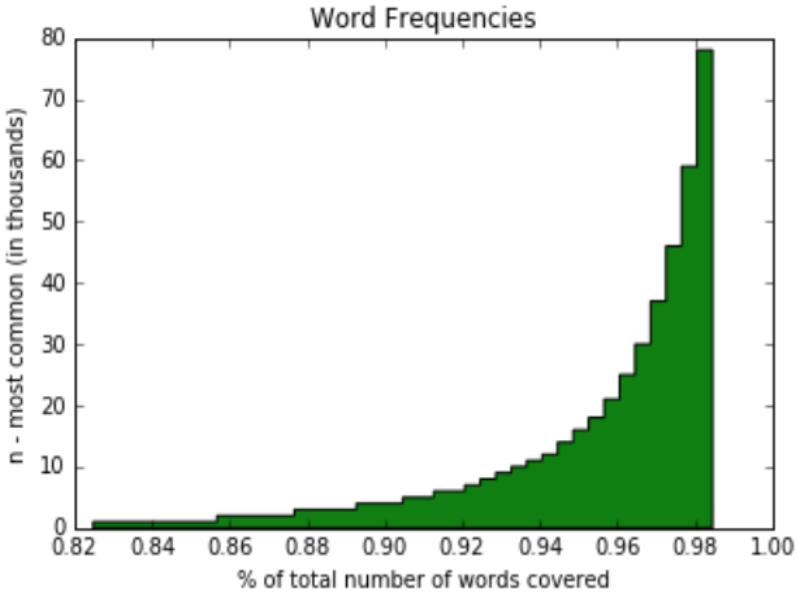


Pic. 11. Ratio between unique words and total number of words

The last two graphs will help us to determine the best hyper parameters for preprocessing and model training. Pic. 12 shows that most of the comments have less than 250 words. It means that it might be sufficient to provide vectors of length 250 to a neural network (with word embeddings as vectorization technique). Pic. 13 shows that 50k of most common words cover around 97 % of the whole dataset.



Pic. 12. Number of words in comments



Pic. 13. Word frequencies

Algorithms and Techniques

The core algorithm to solve the problem is bidirectional LSTM (long short-term memory) neural network with word embeddings layer. Bidirectional LSTM is a popular and powerful algorithm for solving NLP problems. LSTM is a subtype of RNN (recurrent neural network). Both LSTMs and simple RNNs are usually employed to process sequential data as they can capture long-term dependencies. However, LSTM builds on the notion of RNN by adding more control over the information flow. It introduces neurons with memory cells which have an internal state and several gates responsible for forgetting useless information and enforcing helpful information. In practice simple RNN is limited in what dependencies they can learn, while LSTM has instruments to overcome such limitations.

Bidirectionality property implies that the data is read in both directions - from left to right and from right to left. This way the algorithm has more chances to encounter and consider non-trivial text dependencies.

Pictures 14 and 15 show the default architecture that is used as a starting point for deep learning experiments. As you can see, BiLSTM will take word embeddings matrix as an input which is an alternative to TFIDF bag-of-words vectorization (will be discussed later in this section). Keras has predefined embedding layer specifically for this purpose which can be initialized with pretrained weights and further tuned during general training. I chose 100-dimensional embeddings as a part of the default architecture.

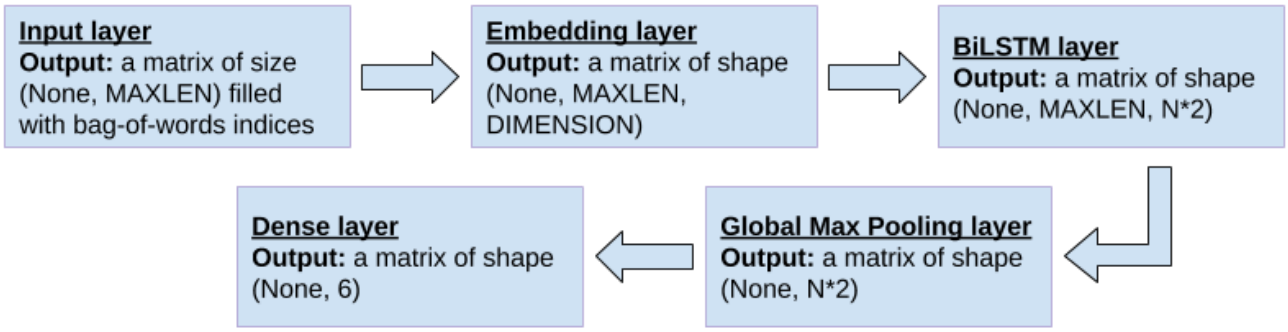
The output of embedding layer serves as an input to BiLSTM layer, which recursively feeds word vectors to the network and after some number of recursive calls produces a new sequence with the same length as the original sequence (MAXLEN parameter). Here we also have the possibility to return the results of intermediate recursive calls. The N (default value is 50) parameter defines the number of recursive calls, so the output will have shape of (MAXLEN, N). However, bidirectionality property doubles the size of intermediate results making the dimension of the final output of BiLSTM layer equal to (MAXLEN, N*2).

In order to prevent overfitting we can also add two additional parameters for BiLSTM layer: dropout (default value is 0.3) and recurrent dropout (default value is 0.1). Both of these parameters specify the probability of ‘dropping’ a neuron for the sake of general network stability and robustness. Recurrent dropout applies to the connections between the recurrent units, while regular dropout applies to the input connections.

Global Max Pooling layer allows to drastically reduce the dimension and lower our chances for overfitting even further. It is doing so by choosing the max value in each columns of the matrix produced by BiLSTM layer. There are other options available that will be explored in the refinement section such as global average pooling, regular max pooling and regular average pooling (regular pooling layers take additional parameter to determine the size of pooling window).

The final Dense layer has 6 neurons responsible for 6 toxic categories, each neuron will output probability of a comment to be associated with corresponding toxic category. This network trains during 2 epochs because after the second epoch the validation score starts to drop which is a clear sign of overfitting.

In addition, I also would like to clarify the meaning of $None$ value in the tensors' dimensions. This value is a placeholder for the batch size, e.g. how many cases will be propagated through the network at once. This parameter may be changed during training and does not affect the general architecture.



Pic 14. Default architecture

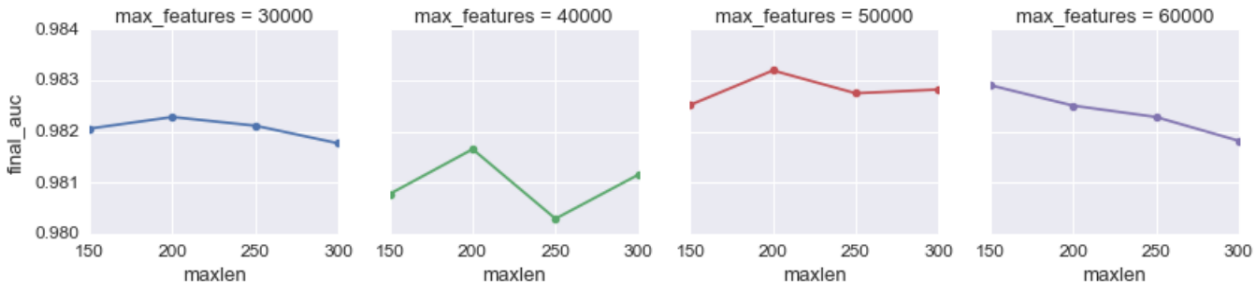
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 200)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 200, 100)          5000000
_________________________________________________________________
bidirectional_1 (Bidirection (None, 200, 100)          60400
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 606
=================================================================
Total params: 5,061,006
Trainable params: 5,061,006
Non-trainable params: 0
_________________________________________________________________
```

Pic 15. Default architecture (Keras output)

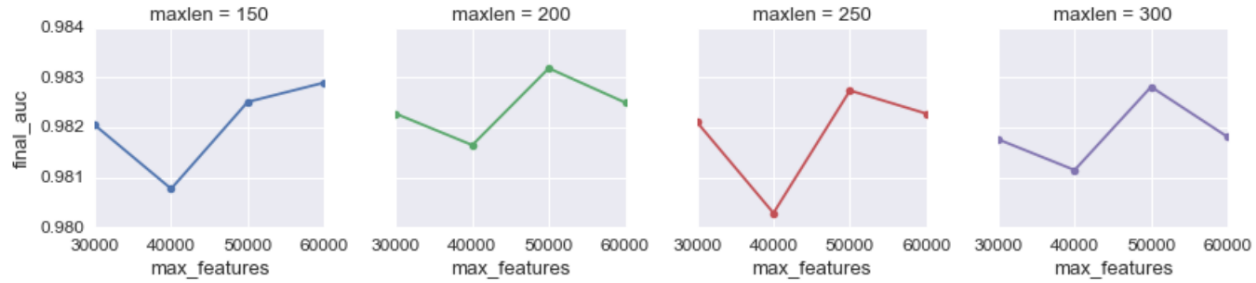
Let's discuss the topic of vectorization in more detail. As described in the next section, for the benchmark model I used TFIDF with no vocabulary restrictions as it is the simplest (yet powerful) NLP vectorization technique. It produces sparse document representation: a document is represented by a TFIDF vector with dimension equal to the size of vocabulary, so the most of the entries will be equal to zero as the document will not contain all the words from vocabulary simultaneously. However, for the LSTM model I would like to use different vectorization technique - dense vectors produced by pretrained word embeddings. This technique implies that every word in a document is transformed into its corresponding embedding vector, so we will have a sequence of vectors insteads of a sequence of words.

Here we have a lot of flexibility: how many words to take from a document, what is the optimal vocabulary size and what is the best pretrained embeddings. It is, of course, possible to take all words from the corpus and use full-length documents. However, this will add unnecessary computation complexity to the problem and slow down the experimentation process. I chose to use 200 as my maximum sequence length and 50000 for my vocabulary size. The pictures (pic. 16 and

17) below help to justify this choice. Here we can see that this combination of parameters provides the best tradeoff between simplicity and performance.



Pic 16. ROC AUC score: Max sequence length (maxlen) plotted against vocabulary size (max_features)



Pic 17. ROC AUC score: Vocabulary size (max_features) plotted against max sequence length (maxlen)

In relation to pretrained embeddings, there are also a number of options available such as the actual algorithm to produce them (word2vec, fasttext, glove), embedding dimension and the corpus against which it was trained. I decided to start from 100-dimensional glove embeddings trained against wikipedia data as it provides a good tradeoff between size and granularity and seems like a good fit for our corpus. Other options will also be explored.

In order to summarize this section, I would like to mention that all default parameters were chosen by a combination of rules of thumb, intuition and some exploratory experiments as it is computationally heavy task to run a parameter grid search for sophisticated neural networks (which is complicated even further by non-trivial reproducibility). However, I undertook an iterative approach for optimizing the parameters and cutting many unviable combinations. The results of this approach is reported in the sections devoted to model refinement and evaluation.

Benchmark

The benchmark model can be represented by the lines of python-style pseudocode below. Here we have simple tf-idf vectorization with some preprocessing (e.g. removal of unique identifiers,

the reason is explained in methodology section) and train 6 different logistic regression (LR) classifiers which provide us with probability estimates for each label. The performance of this model is quite good with the average ROC AUC score equal to 0.976. Please, refer to the results section for the separate scores and the comparison.

```
train, test = load_data()
vec = TfidfVectorizer()
trainX = vec.fit_transform(train.text)
testX = vec.transform(test.text)

for label in labels:   # ['toxic', 'severe_toxic', 'obscene', 'threat', 'insult', 'identity_hate']
   y = train.label
   model = LogisticRegression()
   model.fit(trainX, y)
   test.label = model.predict_proba(testX)
```

Methodology

Data Preprocessing

Data preprocessing is a very important step when dealing with NLP problems. There are two main areas that should be covered: normalization (cleaning, tokenization, lemmatization) and vectorization (how to represent text numerically).

For cleaning, I removed any unique identifiers that were shown in the visualization section. Such tokens may cause overfitting through information leakage. However, it is worth to mention that there is also a possibility to adapt those features to the final system with blacklisting as some emails and usernames may be specifically associated with toxic content. Yet, it is beyond the scope of the current project as I would like to concentrate solely on inherent text properties.

Tokenization (splitting sentences into words) is carried by **nltk** built-in function (*word_tokenize*). **keras** tokenizer has also been used as an auxiliary module as it allows to specify the vocabulary size. Lemmatization (transforming a word to its base form) is omitted because word embeddings used in vectorization were trained against non-lemmatized text.

The second part, vectorization, has already been discussed in the Algorithm section. There I briefly mentioned two parameters which are important for vectorization with embeddings: max sequence length and number of words (features) in the vocabulary. These parameters add one additional step to the preprocessing stage: cutting or padding comments so that they conform to the predefined length (which is 200, as was explained earlier). This step was performed by *pad_sequences* function which can be imported from **keras** library.

Implementation

The implementation was entirely carried out in JupyterLab environment. There are 5 ipython notebooks (the root directory of the repository) which represent 5 stages of the development:

1. **Exploration.ipynb** - data exploration and visualisation
2. **Preprocessing.ipynb** - data preprocessing
3. **Baseline.ipynb** - baseline implementation
4. **Learning.ipynb** - core development
5. **Evaluation.ipynb** - model evaluation and final visualisations

All those notebooks are documented, so the reader may refer to the corresponding notebooks to gain comprehensive understanding of the implementation process. The core libraries that were used to tackle the problem are ***pandas*** (data handling), ***numpy*** (numerical manipulations), ***nltk*** (preprocessing), ***seaborn*** (visualisation), ***sklearn*** (baseline implementation and metrics) and ***keras*** (deep learning).

Pic. 18 shows the information flow between different notebooks and data directories. In relation to the notebooks cells, incoming arrows signify that the notebook reads the data from the directory, outgoing arrows signify that the notebook writes the data to the directory. Pic. 19 shows the dependence between different notebooks. For example, **Preprocessing** should be run before **Baseline** or **Learning** because those two notebooks act on the data produced by **Preprocessing**.

Let's briefly go through the last three mentioned notebooks. **Preprocessing** is responsible for all text-related manipulations, it produces the data which is ready to feed into the neural network (or logistic regression for baseline). It is doing so by, firstly, creating different versions of original text (e.g. tokenized text, tokenized and lowercase text etc.) and, secondly, producing numerical representations with predefined parameters (e.g. maxlen, max_features, lowercase vocabulary etc.) It also saves all fitted tokenizers into the *pickle* directory (we will need them in order to restore the vocabulary and initialize the weights of the embedding layer).

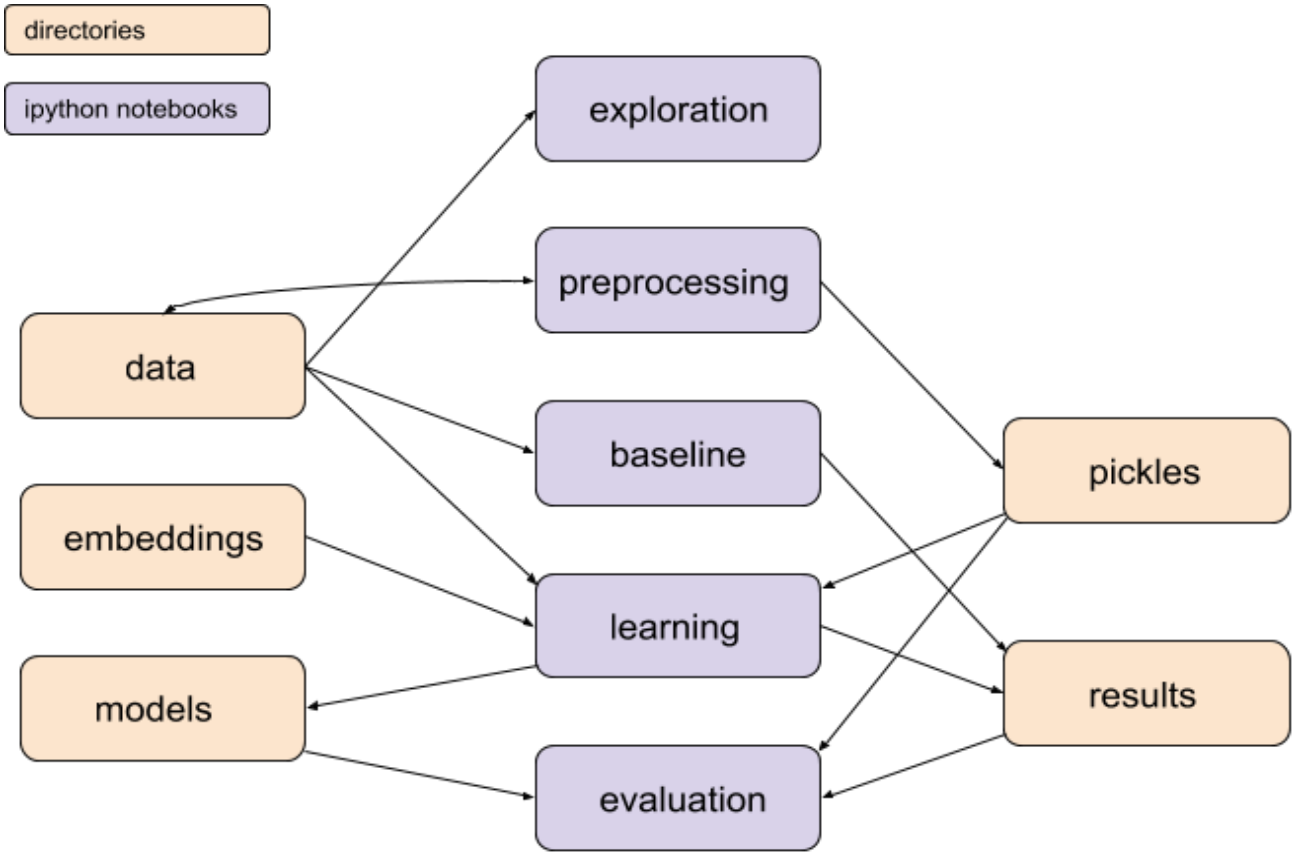
Baseline, as it follows from the name, uses LogisticRegression from ***sklearn*** and runs the baseline on the default variant of preprocessed text (simple cleaning and tokenization). The extended results (individual scores and the final score) are written into the *results* directory in order to be explored in the **Evaluation** notebook**.**

Learning contains the core code for the project, namely, the code for all deep learning experiments. There I implemented parameter grid search to find the optimal model (although, it is not recommended to explore all possible combination, unless substantial computational resources are available). Some helper functions have been defined to simplify the interactions with the notebook, for example:

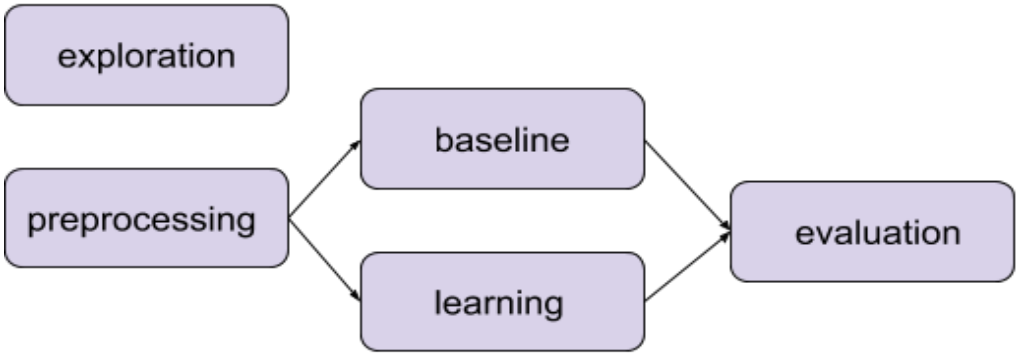
1) $get\_embedding\_matrix(embeddings\_index, tokenizer)$ - given the embedding dictionary {word: vector} and the tokenizer outputs the initial weights for the Embedding layer

2) $get\_model\_lstm(embedding\_matrix, n = 200, dropout = 0.2, recurrent\_dropout = 0.1, pool = 'global\_max')$ - given the defined parameters outputs the model with the specified architecture

In overall, implementation process was quite straightforward as the mentioned libraries provide a simple way to interact with complex tools. However, I had to confront Keras reproducibility issues in order to produce stable results while testing various network architectures. I couldn't allow multiple runs for the same experiment to average the results because it is computationally expensive, so I decided to control the randomness by setting various seeds (set_seed function in **learning.ipynb**) as explained here. Despite the fact that this method does not fully solve the problem it provides sufficient reproducibility for comparing the results of various experiments.



Pic 18. Interaction between data directories and ipython notebooks



Pic 19. Dependence scheme

Refinement

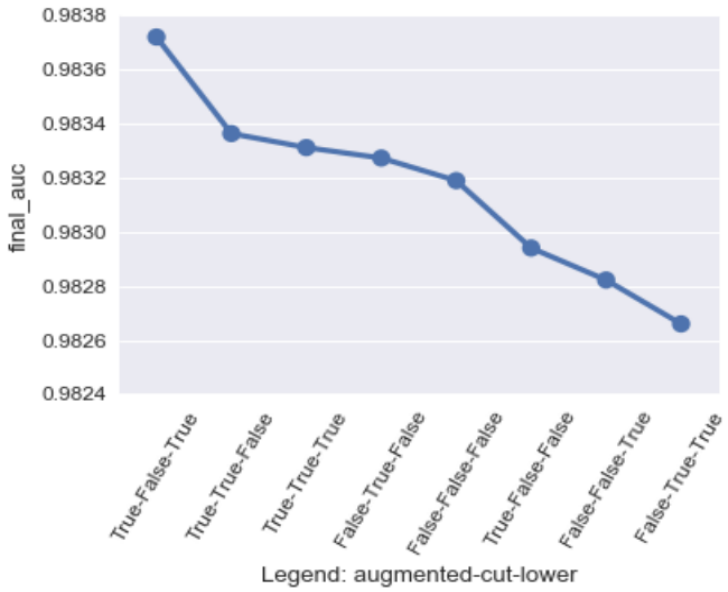
In this section I would like to concentrate on three steps which aim to provide significant improvement: (1) additional preprocessing and data augmentation, (2) choice of embeddings and (3) architecture-related tuning. In order to avoid unnecessary computation complexity, the refinement process was performed iteratively, that is, only the best parameters from step one moved to step two, et cetera. The starting point is the default architecture described in the Algorithm section, this architecture yields 0.9831 for the final ROC AUC score (average of 6 toxic categories) on the privately held test set.

For additional preprocessing I tested three ideas: lowercase transformation, different padding options and data augmentation through machine translation. Usually lowercase transformation is a default step during preprocessing. However, as was shown in pic. 8, the original formatting may also provide some clues about text toxicity so I decided to explore this transformation separately. The second idea explores whether we can improve the score by how we cut the comments which exceed 200 tokens. The default approach is just to take first 200 words from a comment. However, the 'distribution' of toxicity may not be even, so I decided to explore another option - take first 100 words and last 100 words.

The data augmentation trick was suggested by a fellow Kaggle member. The idea behind it is simple and clever: we can translate a comment to some target language and then translate it back to english. Usually this yields slightly different result from original text so we can augment the dataset without directly duplicating the content. I decided to use three european languages (french, german and spanish) and apply this trick only to categories with fewer entries (pic. 2), namely, severe toxic, threat and identity hate. The augmentation was performed only on the data explicitly used for training in order to not interfere with the testing process.

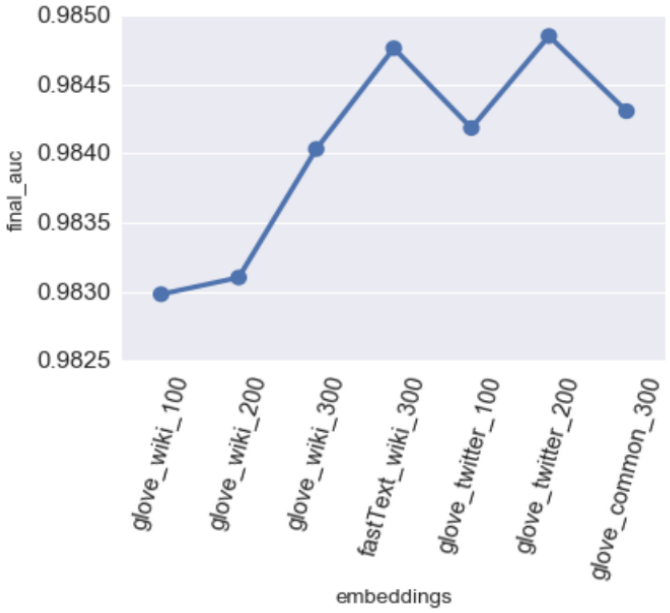
The results of those experiments are presented on pictures 20 and 21: augmentation together with lowercase transformation allow the model to improve the score, so the further experiments will use augmented data with lowercase text.

Pic 20. Results of additional preprocessing (cut value means whether the new cutting technique has been applied)



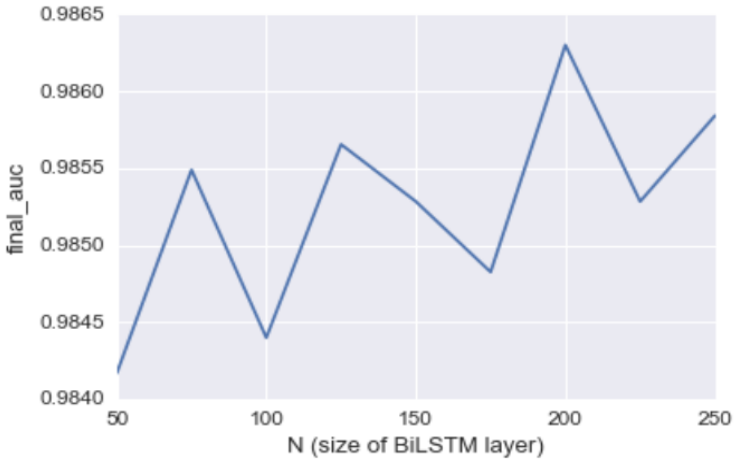
Pic 21. Combined results of additional preprocessing

The second step is to explore how various pretrained embeddings affect the performance of the model. Pic. 22 shows that the default embeddings (100-dimensional glove pretrained on wikipedia) are actually the worst choice for the problem. The best results are given by 200-dimensional glove pretrained on twitter. I assume that the twitter data is indeed the best fit for the type of the problem we are trying to solve (which in essence is an extended sentiment analysis). Twitter-like communication exhibits similar to our data properties because it consists of personal, sometimes emotional, short messages which are very similar to our use case.



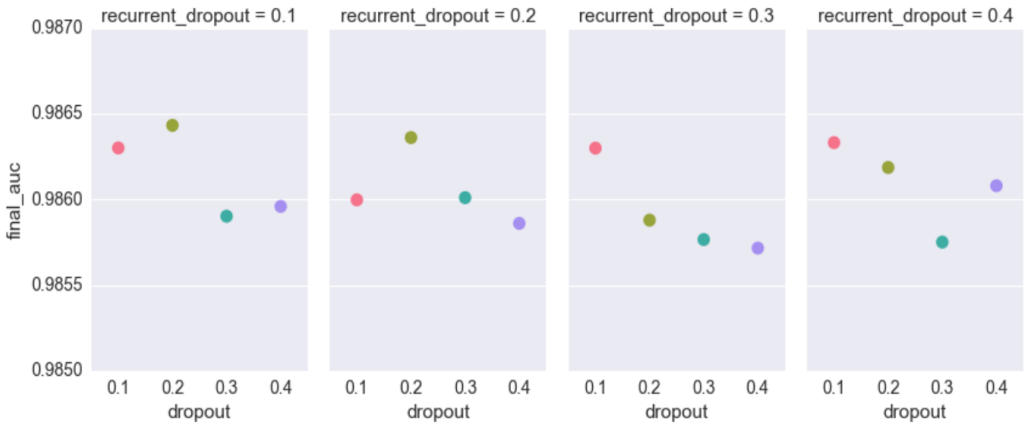
Pic 22. Embedding results

At last we will explore how various architecture tweaks can improve the quality of the model. Pic. 23 shows the influence of BiLSTM layer size on the performance: the graph has interesting regular shape with the peak on $N = 200$. Interestingly, this value coincides with the final dimension of pretrained embeddings.



Pic 23. Size of BiLSTM layer

Dropout parameters help to prevent overfitting, but it is important to avoid overusing them, limiting the model's predictive power. Pic. 24 shows that the optimal combination is 0.1 for the recurrent dropout and 0.2 for the regular dropout.



Pic 24. Dropout values

In addition to the size of BiLSTM layer and dropout parameters I also tried different pooling techniques and usual (non-bidirectional) LSTM. However, that resulted in drastic performance drop (~0.98 score), so I decided to omit these experiments from further consideration.

Results

Model Evaluation and Validation

The evaluation was done on the held-out test set that was not available to any model during the training process. The general architecture outlined in the pic. 14 stayed the same, however, the final hyperparameters (and the choice of embeddings) changed because the combination of hyperparameters depicted in the pic. 26 performed the best among other alternatives.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 200)               0
_________________________________________________________________
embedding_1 (Embedding)      (None, 200, 200)          10000000
_________________________________________________________________
bidirectional_1 (Bidirection (None, 200, 400)          641600
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 400)               0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 2406
=================================================================
Total params: 10,644,006
Trainable params: 10,644,006
Non-trainable params: 0
_________________________________________________________________
```

Pic 25. The final architecture (Keras output)

parameter	value
N	200
dropout	0.2
recurrent dropout	0.1
maxlen	200
max_features	50000
embeddings	glove/twitter
embeddings size	200

Pic 26. The final parameters

In order to verify the robustness of the final model I conducted sensitivity analysis by slightly varying the original text and the discriminative threshold. The model proved to be quite stable: drastic threshold variations had a little effect on the final output. Pic. 27 demonstrates this quality, here we can see that the core categories - toxic and threat - stay intact despite threshold variations. Two additional categories - obscene and insult - also appear in this example while they are not directly applicable in this case. However, their confidence is below 0.5, so they have little chances to appear in the final system.

The second example is taken from the twitter comment section to the infamous resource Russia Today. Here we can see that the wording is very important to the model: adding the word ‘crap’ results in toxic mark. Moreover, adding the word ‘pussy’ to the same sentence structure brings forward obscene category. Same is true vice-versa, rephrasing and dropping specific words of a sentence removes toxic mark. Additionally, second comment in the pic. 28 ideally should also be marked as insult, but it didn’t happen probably due to the absence of similar constructions in the training data.

```
text = 'I will kill you'
preds = am_i_toxic(text, model, tokenizer, th=0.1)
```

```
Preprocessed text:
 will kill you

Threshold: 0.1
Toxic categores: toxic, obscene, threat, insult
```

```
text = 'I will kill you'
preds = am_i_toxic(text, model, tokenizer, th=0.5)
```

```
Preprocessed text:
 will kill you

Threshold: 0.5
Toxic categores: toxic, threat
```

```
text = 'I will kill you'
preds = am_i_toxic(text, model, tokenizer, th=0.9)
```

```
Preprocessed text:
 will kill you

Threshold: 0.9
Toxic categores: toxic, threat
```

Pic 27. Threshold variations

```
text = 'All this crap did not just happen yesterday. It has been building for decades. \
        Being passive wont help prevent a war, being firm and able to talk to each other just might. \
        And forget that prophecy BS.'
preds = am_i_toxic(text, model, tokenizer, th=0.5)
```

```
Threshold: 0.5
Toxic categores: toxic
```

```
text = 'All this crap did not just happen yesterday. It has been building for decades. \
        Do not be a pussy, being firm and able to talk to each other just might prevent the war. \
        And forget that prophecy BS.'
preds = am_i_toxic(text, model, tokenizer, th=0.5)
```

```
Threshold: 0.5
Toxic categores: toxic, obscene
```

```
text = 'All this did not just happen yesterday. It has been building for decades. \
        Being passive wont, being firm and able to talk to each other just might prevent the war. \
        And forget that prophecy BS.'
preds = am_i_toxic(text, model, tokenizer, th=0.5)
```

```
Threshold: 0.5
The text is not toxic
```

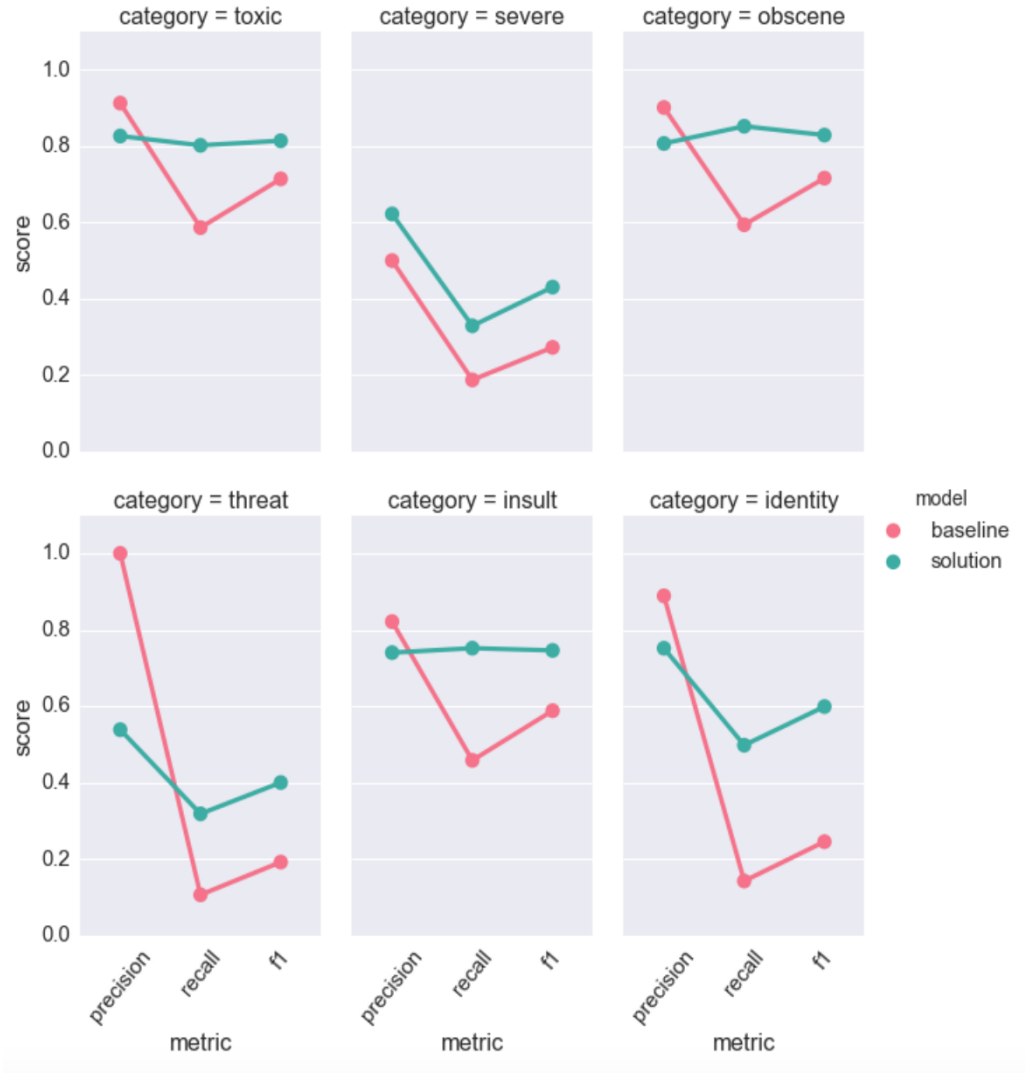
Pic 28. Wording variations

Justification

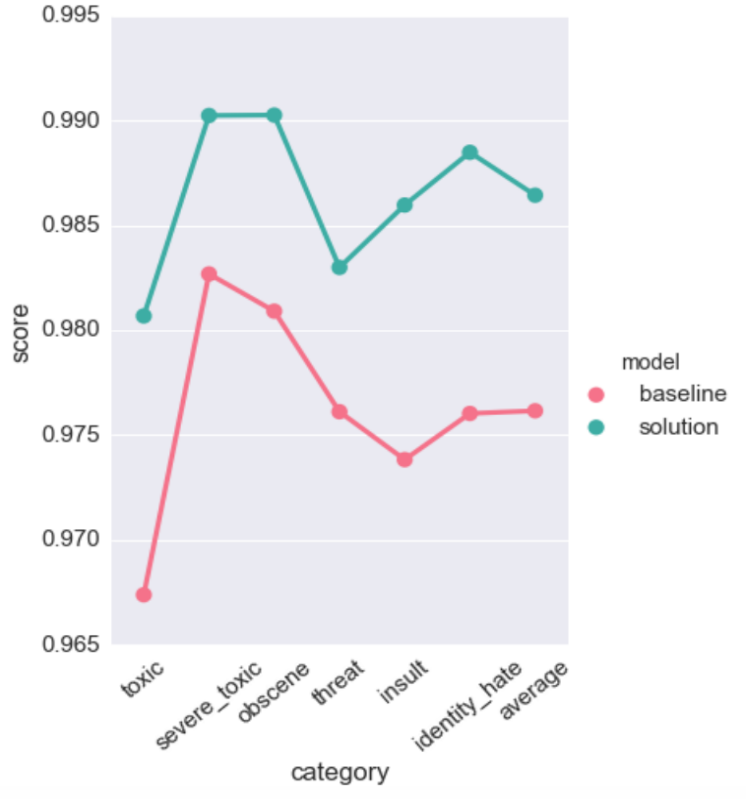
Pic. 29 and 30 show the comparison between the benchmark and the final model. The average ROC AUC score of the final model is 0.987 against 0.976 on the benchmark. The pictures illustrate the imbalance between different categories. Some categories, like obscene, turned out to be fairly easy for the model, while others, like threat, will definitely benefit from additional training data. Threat detection is indeed a subtle problem as the threat comment may not contain any easy to digest features (like obscene vocabulary), so the model should discover an intention behind the text rather than solely relying on shallow text properties.

There is also a difference between precision and recall scores for the categories like severe toxic, threat and identity hate. The difference is more pronounced on the baseline model. The balance between precision and recall is an interesting problem per se. Given the nature of the use-case we cannot really put neither precision nor recall first. By focusing on the precision we risk missing a lot of harmful content, while by focusing on recall can make the system too harsh and totalitarian and let in many false-positives. The community will not benefit from neither of those scenarios. F1 score treats precision and recall equally so in our case it is a viable metric to optimize.

Regarding the question of whether the solution is significant enough to have solved the posed problem, I would like to say that the model is quite robust to be deployed in the real world setting (especially in online learning mode). However, the notion of confidence comes handy here, as it is possible to set up a confidence threshold. A comment will require consideration of a moderator if its confidence is below the threshold.



Pic 29. Separate precision, recall and f1 scores of the baseline and the final solution (with 0.5 threshold)



Pic 30. Separate ROC AUC scores of the baseline and the final solution

Conclusion

Free-Form Visualization

To play with the model you can use the **evaluation** notebook. To run the model it is enough to run the first cell with import statements and then run two code cells after the line 'Playing with the model' (Pic. 31). This will define some helper functions and load the model and the tokenizer which is needed to preprocess the user's input (make sure to have the actual model in the model folder and tokenizer in the pickles folder). After these actions the function am_i_toxic will be available. Pic. 32 shows an example of it's usage.

Pic. 31. Exploration section

```
text = 'What a bunch of crap. You are trying to stir up trouble.'
preds = am_i_toxic(text, model, tokenizer, th=0.5)
```

```
Preprocessed text:
 what bunch of crap you are trying to stir up trouble

Threshold: 0.5
Toxic categores: toxic, obscene
```

Pic. 32. Example of input and output

Reflection

The end-to-end development process can be summarized by these steps:

1. Explore the data in order to gain general understanding of the problem and find some clues and hints that may help to solve the problem.
2. Define the metrics that will reflect the model's quality in the most descriptive way. For example, accuracy will not do well here especially for the categories with a small number of positive entries (e.g. threat).
3. Implement a simple baseline in order to detect overengineering in the next steps.
4. Preprocess the data, ideally this step should decrease the noise and make helpful features more prominent.
5. Implement the first version of the intended solution.
6. Perform an iterative search over possible parameters and configurations that may improve the model.
7. Validate and evaluate the final solution.

I found particularly interesting the problem of text augmentation. This concept comes from computer vision field where it is clearly defined and formalized (e.g. various image rotations). In case of image processing, it is indeed cheap yet very helpful way to improve the model and extend it's generalization ability. However, when it comes to natural language processing there are very few (if any) ready-to-use solutions with proven records. Text augmentation is nowhere near the success of image augmentation, so the results I have seen after augmenting the data through translation inspired me to explore further this research topic.

As I mentioned in the justification section, the suggested model provides substantial improvement in the metrics that I employed. I do believe that the final model efficient detection of toxic online behavior.

Improvement

From the algorithmic perspective the low-hanging fruit of improvement is getting more data for training. But from the timeline perspective it is not the most optimal solution, because usually this task requires labor of human annotators. However, in every competition Kaggle provides two separate datasets - for training and for testing. The testing dataset contains no label information, so we cannot directly incorporate it in the development process. One way to overcome this limitation is to apply pseudo-labelling, namely, using the final model to label the testing set, then adding the entries with the highest confidence to the training set and training the model again[3].

Another way to improve the model and tune in specifically for the wikipedia use-case involves tweaking the Embedding layer. A lot of complexity is concentrated there so it may be worthwhile to dig into it. As was shown in the pic. 22 the actual data used for embeddings training makes a lot of difference. Pretraining words embeddings against wikipedia editorial comments will require computational resources but may result is significant improvement.

Inspirations

This work was inspired by the following Kaggle threads:

1. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/discussion/52557
2. https://www.kaggle.com/jagangupta/stop-the-s-toxic-comments-eda
3. https://www.kaggle.com/jagangupta/lessons-from-toxic-blending-is-the-new-sexy/notebook
4. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/discussion/48038

[3] Also, it is possible to just ask competition organizers to reveal the full data.