Real or Not? A Natural Language Preprocessing system for Disaster Tweets

DAVIDE MONTAGNO BOZZONE, 535910

LUCA CORBUCCI, 516450

RAFFAELE VILLANI, 608837

Abstract: Twitter is an online social network where users can post messages with a restriction, all the tweets must be at most 280 characters, this restriction allows to focus on the content of the message. Thanks to this feature Twitter can also be used to send short real-time notification about possible disaster. Unfortunately, it's not easy to automatically understand if a tweet is really about real disaster or if the user is just using some words that are commonly used in a disaster context. In this report, we will explain how we managed to solve this problem.

1 INTRODUCTION

Twitter is a social network that has become an important communication channel in times of emergency, everyone can tweet a message to allow other people to stay informed about the situation. In this report, we will explain our solution for "Real or Not? NLP with Disaster Tweets" competition [1]. We will analyze the performances of our solutions and we will also explain the experiments that we made during the development of this project.

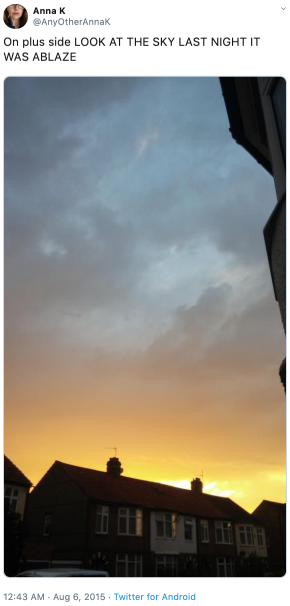
We used Keras [1] to develop this project.

2 COMPETITION

2.1 Task

"Real or Not? NLP with Disaster Tweets" is a Kaggle competition in which the goal is to create a machine learning model that can predict which tweets are about the real disaster and which ones are not. Kaggle provides a hand classified dataset with 10.876 tweets (already divided in training and test set), those tweets could be a real disaster like in Figure 1a or unreal like in Figure 1b.

(a) Real disaster tweet



(b) Unreal disaster tweet

Fig. 1

Kaggle allows us to upload our predictions to compute the accuracy of our model using the provided test set.

2.2 Related Works

Before starting the project we researched to understand the current state of the art in the NLP for disaster field. We found some papers that were useful for our work, in particular, "Deep Learning and Word Embeddings for Tweet Classification for Crisis Response" [2] helped us a lot because gave us the idea to use the Fine Tuned Glove Word Embedding (we will talk more about this in subsection 5.1).

We also found two articles where are explained some techniques to identify disaster-related tweets. More specifically in the paper [3] the researcher explained how to classify tweets concerning the Hurricane Sandy while in another paper [4] they explained how to apply NLP techniques to classify tweets concerning the 2011 East Japan Earthquake.

[1] Real or Not? NLP with Disaster Tweets: https://www.kaggle.com/c/nlp-getting-started/overview.

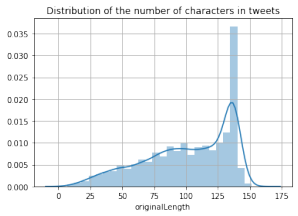
3 DATASET

3.1 Dataset Exploration

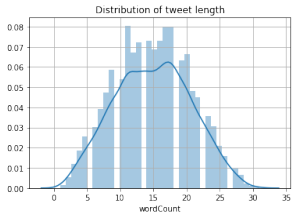
We started our project doing some preliminary analysis of our data. Kaggle provides for this competition two datasets:

- Train Dataset: this one contains 7613 samples. For each sample we have:
  - The text of the tweet, all of them are in English;
  - A keyword contained in the tweet;
  - The location from which the tweet was sent;
  - A label that classifies the tweet as real (1) and unreal (0).
- Test Dataset: this one contains 3263 samples, for each of them we have the same fields of the training dataset except for the target. This dataset is not labelled and to understand the accuracy that our model produces with it we have to upload the predictions on Kaggle.

Our training dataset contains 3271 real tweets and 4342 unreal tweets, each tweet is at most 280 characters and it could contain links, hashtag or mentions to other users. In Figure 2a we can see the distribution of the tweet length in term of characters and Figure 2b we show the distribution concerning the words contained in a tweet.



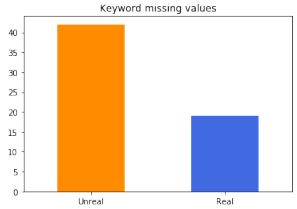
(a) Lenght distribution (characters)



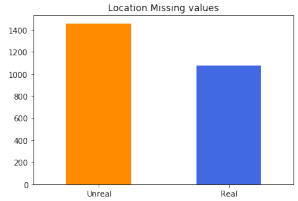
(b) Lenght distribution (words)

Fig. 2

As we already said in our dataset we have two fields apart from the tweet, they are "keyword" and "location". For both of them, we noticed that we have some missing values and we will explain in 3.2 what we did to solve this problem. During the data exploration phase, we tried to understand if there is a correlation between the missing values and the label of our tweet.



(a) Missing values for the keyword field



(b) Missing values for the location field

Fig. 3

As we can see in Figure 3a and 3b seems that if we have a missing value then the label of the tweet is more likely "Unreal" but we also have to consider that in the dataset the number of "Unreal" Tweet is higher than the number of "Real" one so the presence of the keyword or the "location" should not be too correlated with the label.

3.2 Data Cleaning

After we finished the dataset exploration we started cleaning our dataset, we needed to do this because the text of the tweets contains links, mentions and hashtag and this could be a problem for the classification. For each tweet we did the following operations:

- We used the library "Contractions" to remove the contractions from our tweets. This library is available on Github [2];
- We removed the mentions that are in the text, we recognized the mention thanks to the use of "@" as the first character and we removed the entire word because it is just the username of the user that has been mentioned;
- We removed links;
- We removed just the character "#" for the hashtag because we guessed that the word used as a hashtag could have been useful for the tweet's classification;
- We removed punctuation;
- We removed numbers and we lowered the case of the text.

After reasoning about that, we decided to keep the stopwords in our dataset because we guessed that they could be useful in a classification task. As an example, we thought about a tweet with a negation. If we had removed the stopword the tweet would have had a completely different meaning.

Besides this first data cleaning process, we also filled the missing values in the Keyword and Location fields. We decided to fill the missing values using a placeholder, more specifically we used "No_Keyword" and "No_Location".

After we finished the data cleaning process we did some analysis on the cleaned data and we produced three word clouds for the tweets:

[2] Contractions codebase: https://github.com/kootenpv/contractions.

- The one in Figure 4 is for the complete dataset (real tweets and unreal);

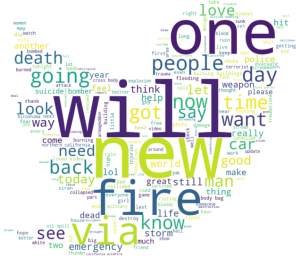
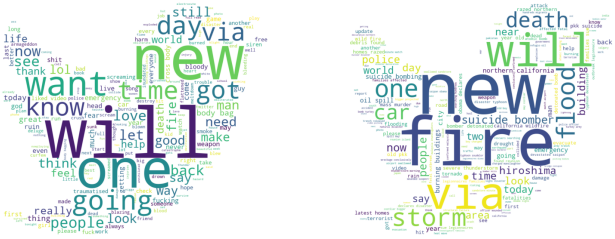


Fig. 4

- The one in Figure 5a is for unreal tweets.
- The one in Figure 5b is for real tweets.



(a) (b)

Fig. 5

We can notice that we have a different distribution of words in the real tweet compared with unreal tweets. From Figure 6 we can see that in real disaster tweets we have, among the most common words, "Fire", "Disaster" and "Suicide".

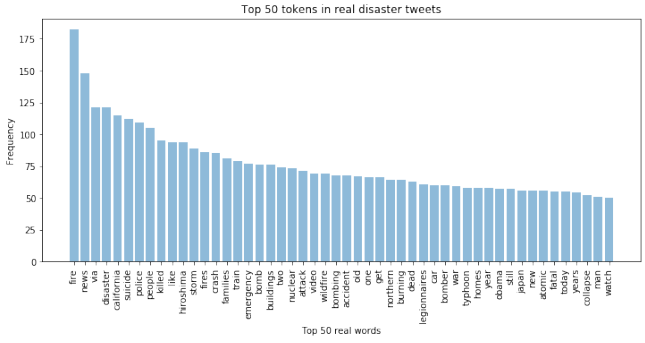


Fig. 6. Top 50 words in real disaster tweets

From Figure 7 we can see that in unreal disaster tweets we can find words that are not used in dangerous situations, we have among the most common words, "Like", "New" and "Video".

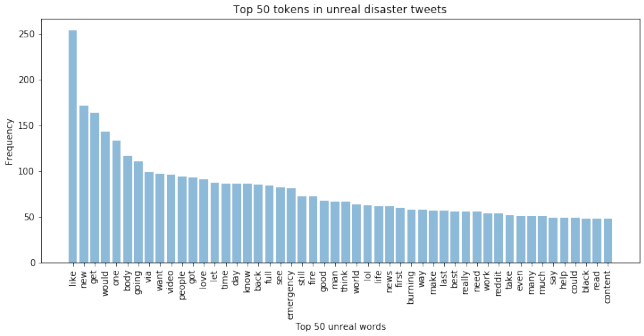


Fig. 7. Top 50 words in unreal disaster tweets

3.3 Studying the data distribution in train and test set

After we finished the data cleaning process we decided to analyze the data distribution in the training and in the test set to understand if they are different. We did this because we know that if we train our model on some data that have certain distribution and the test set has a different distribution, the model will perform poorly on the test set.

To be sure that training and test set have the same distribution we merged all the data in a single dataset and we trained a classifier to understand if a certain tweet is from training set or the test set.

We got always bad performances on the training set and the test set, our classifier was not able to understand from which dataset the tweet was taken. We can conclude that the training and test set have the same distribution.

4 IMPLEMENTATION CHOICES

In this section, we will present all the architectures that we tried for this tweet classification task and how we chose the hyperparameters, in Section 6 we will present the score that we got with of these methods on Kaggle and we will make a comparison between them.

We trained these models using the training set provided by Kaggle, we used the 80% of the data to train and validate our models and 20% to test them. Once we reached good performances, we trained again the model on the entire training dataset and we tested the models using the test dataset provided by Kaggle. In the end, we got the score for each model uploading the produced predictions on Kaggle.

4.1 Baseline Models

When we started our project we decided to implement some basic models mainly for two reasons:

- We wanted to have some baseline model to compare their performances with more difficult models;
- We wanted to understand what kind of performance we should have expected.

We started with the Naïve Bayes Classifier, a non-neural model, and then we moved to more and more complex "neural" models.

4.1.1 *Naïve Bayes Classifier*

The first model that we tried was a Naïve Bayes Classifier, we thought that for this kind of task where we have a small dataset (7613 tweets in the training set) it could perform decently and so we implemented it. To choose the best hyperparameters for our classifier we did a GridSearch, more specifically we tried different values for the alpha of the classifier, the ngram range of the Bag of Words, the number of epoch for training and the batch size. We tried this model using the original dataset, the oversampled one and an augmented dataset (we will talk more about this in Section 5).

4.1.2 *Neural Network*

As part of our first tests, we decided to try to use a simple neural network to understand how it performs compared with Naïve Bayes and with models that are more suitable for NLP tasks such as LSTM or Bert. Our goal, in this case, was to start with a neural network with a really simple architecture and then try some possible variations. We started with a Neural network with a single layer and a single hidden neuron, then we added another layer increasing the number of the hidden neurons.

4.1.3 *Recurrent Neural Network*

As part of baseline models, we also implemented a Vanilla Recurrent Neural Network to understand how it performs compared with more suitable models like LSTM and GRU. The recurrent neural network has recently been shown to produce state-of-the-art [5] results in perplexity and word error rate across a variety of tasks. These networks differ from classical feed-forward neural network language models in that they maintain a hidden-layer of neurons with recurrent connections to their previous values. This recurrent property gives an RNN the potential to model long span dependencies. However, theoretical analysis indicates that the gradient computation becomes increasingly ill-behaved the farther back in time an error signal must be propagated and that therefore learning arbitrarily long-span phenomena is difficult.

4.2 LSTM

Another model that we tried was the Long-short-term memory. This model is well-suited to classifying, processing and making predictions based on time series data. LSTM, unlike Recurrent Neural Network, has feedback connections and can partially solve the vanishing gradient problem, because its units allow gradients to also flow unchanged; we said partially because of LSTM networks can still suffer from the exploding gradient problem. In the equations below, the lowercase variables represent vectors. Matrices W_q and U_q contain, respectively, the weights of the input and recurrent connections, where the subscript q can either be the input gate i, output gate o, the forget gate f or the memory cell c, depending on the activation being calculated. Also, σ_g is referred as sigmoid function, $\sigma_c = \sigma_h$ as hyperbolic tangent function.

- $f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$
- $i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$
- $o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$
- $\tilde{c}_t = \sigma_h(W_c x_t + U_c h_{t-1} + b_c)$
- $c_t = f_t \bigcirc c_{t-1} + i_t \bigcirc \tilde{c}_t$
- $h_t = \bigcirc \sigma_h(c_t)$

In our case the baseline model is a single layer LSTM neural network with hidden size 128. The model receives pre-trained GloVe Twitter 27B embeddings as input.

Below, we can see the best parameters that we found for this task.

- **Batch size**: 32
- **Optimizer:**: Adam
- **Number of epochs**: 25
- **Loss**: Binary cross-entropy
- **Dropout LSTM layer**: 0.2
- **Recurrent dropout LSTM layer**: 0.5
- **LSTM units**: 128
- **Split training set**: 0.2

4.2.1 *Bidirectional LSTM*

We also used Bidirectional-LSTM to understand if it allows us to obtain better performances with respect to the standard LSTM. This model allowed us to manage data in two ways, one from past to future and one from future to past. The model differs from unidirectional since LSTM, which runs backwards, can preserve information from the future. So, by using the two hidden states combined we will be able at any point in time to preserve information from both past and future. Bidirectional-LSTM shows quite good results as they understand the context better, but it requires more time in the training phase. While bidirectionality is useful, it can't be used in situations where a sequence model is used for inference. For example, in machine translation. Below, we can see the best parameters that we found for this task.

- **Batch size**: 128
- **Optimizer:**: Adam
- **Number of epochs**: 20
- **Loss**: Binary cross-entropy
- **Dropout LSTM layer**: 0.2
- **Recurrent dropout LSTM layer**: 0.5
- **LSTM units**: 128
- **Split training set**: 0.2

4.2.2 GRU

As a comparison with LSTM we also implemented GRU, this is like a Long Short-Term Memory with a forget gate but has fewer parameters than LSTM, as it lacks an output gate. We decided to try GRU because it has been shown to exhibit even better performance on certain smaller and less frequent datasets. There are several variations on the full gated unit, with gating done using the previous hidden state and the bias in various combinations, and a simplified form called minimal gated unit. We used the full one. Below we can find the best parameters for this task.

- **Batch size**: 32
- **Optimizer:**: Adam
- **Number of epochs**: 5
- **Loss**: Binary cross-entropy
- **Dropout LSTM layer**: 0.2
- **Recurrent dropout LSTM layer**: 0.5
- **LSTM units**: 128
- **Split training set**: 0.2

4.2.3 Bidirectional GRU

With the same reasoning that we did with LSTM and Bidirectional LSTM we decided to also try Bidirectional GRU.

As it happens with Bidirectional LSTM, here the input sequence is fed in normal time order for one network, and in reverse time order for another. The outputs of the two networks are usually concatenated at each time step, though there are other options, e.g. summation. By default, the outputs of the reversed RNN is ordered backwards. Keras will reverse it when *return_sequences* is true (it's false by default)

- **Batch size**: 32
- **Optimizer:**: Adam
- **Number of epochs**: 10
- **Loss**: Binary cross-entropy
- **Dropout LSTM layer**: 0.2
- **Recurrent dropout LSTM layer**: 0.5
- **LSTM units**: 128
- **Split training set**: 0.2

4.3 Going Deeper

After trying the models that we have just described we started some experiments and we decided to combine them in a deeper network. Thanks to GridSearch we tried a lot of parameters and a lot of possible architectures for our model and in the end, we chose the one that gives us the best score on the validation data. In Figure 8 we show the model that we chose, we used a Bidirectional GRU layer followed by a LSTM

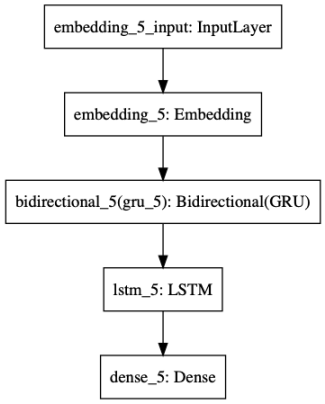


Fig. 8. A deeper architecture that we tried combining Bidirectional GRU and LSTM

We did this experiment to see if by adding more layers we could have increased the performance of our model.

4.4 Keras Functional API

In section 3 we explained the content of our dataset and we said that we also have, for most of the tweet, a field with a keyword contained in the tweet. In the previous models, we did not consider these keywords and so we decided to try to take into account them in a more advanced model. To develop this model we used the Keras Functional API [3] that gives more freedom during the development of a neural network, in our case we exploited the possibility to have more than a single input for the network.

In Figure 9 we show the architecture of the network we developed, in this model we have two inputs layers:

- The first one is the input layer for the tweet;
- The second one is the input layer for the Keywords extracted from the tweet (we have this information in the dataset).

We embedded both the input using Glove and then we input the tweet's embedding to the Bidirectional LSTM and the Keyword's embedding to a Dense Layer. We concatenate these two outputs and we apply LSTM to this concatenation.

[3]Keras Functional API: https://keras.io/guides/functional_api/.

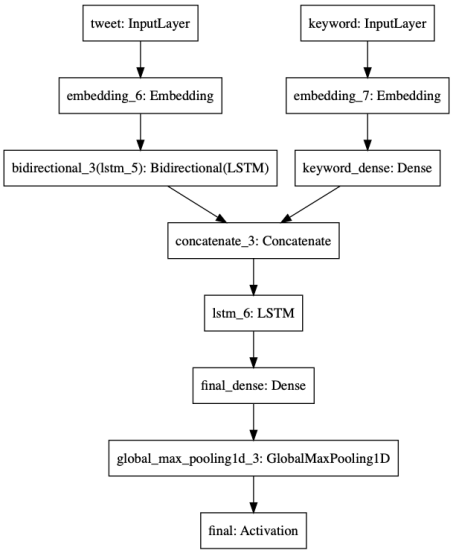


Fig. 9. Architecture of the network we developed using Keras Functional API

This model is a bit more complicated concerning the classic LSTM and to the other models that we have seen until now and it took longer to train it.

4.5 BERT

One of the models that we tried is the BERT (Bidirectional Encoder Representation from Transformer) model [6], we used the large BERT model already trained [7].

The reason to choose BERT is that it has achieved state-of-art performance in many NLP tasks and it is open source.

To preprocess the tweets for the BERT model we performed the text preprocessing that we explained before, after that, we performed the BERT tokenization that consists to tokenize the sentence and finally add the tokens [CLS] and [SEP] at beginning and end of every sentence. Finally, we used a max sequence in the BERT input layer of 150 words because the tweet with the max length in the dataset has a length of 53 words, but some tweet, in general, could be longer.

To perform our classification task we added a classification layer on top of the Transformer output for the [CLS] token, in particular, the classification layer is a simple sigmoid classifier that performs the classification task. We used the pre-trained model of BERT and we performed the final-tuning on this model, for doing that we did a grid search on the parameters shown below. BERT changed the rules of the game. This is because compared with the design of complex and ingenious network structure, the experimental results of BERT language representations pre-trained on massive unsupervised data and the simple network model with a small amount of fine-tuning training data achieved great advantages.

- **Batch size**: 6
- **Learning rate(Adam)**: $1e^{-5}$, $2e^{-5}$, $1e^{-4}$, $2e^{-4}$
- **Number of epochs**: 2,3

with this different hyperparameters, we choose the best combination of these on the validation data that are 3 epochs with $2e^{-5}$ learning rate and a batch size of 6. The fine-tuning process lasted about 10-20 minutes every epoch. To complete the training in acceptable time we performed it on Google Collaborative Environment(Colab).

We used TensorFlow and k-train library [8] to do the fine-tuning of BERT and the results were similar, so we preferred to use k-train because it is a very recent library and really easy to use.

4.6 BERT and Bidirectional LSTM

To extend the BERT model [9], we explored the ideas of adding an LSTM to the final fully connected layer of the transformers within Bert. We recognized the advantages that a bidirectional LSTM would have over an NN or an RNN such as resource efficiency and bidirectional word awareness. We thought that incorporating a bidirectional LSTM at the end of the attention layers in BERT would have a much stronger effect than we saw in our implementations. We saw worse performance and scores of roughly half a point lower than the baseline. We still believe in this strategy, but perhaps there may be further optimizations in our hyperparameters or architecture.

- **Batch size**: 32
- **Learning rate(Adam)**: $1e^{-4}$
- **Number of epochs**: 3
- **Dropout LSTM layer**: 0.2

5 EXPERIMENTS

5.1 Embedding Fine Tuning

When we started the first tests using the Recurrent Neural Network and the LSTM we needed to create the embedding for our tweets. We decided to use the Glove Embeddings [10], more specifically we used the "glove.twitter.27B.200d" [4]. Starting from the downloaded embedding we decided to fine-tune them using a Library called Mittens [11]. After the fine-tuning, we created a new Embedding containing the words which were not already in the original one.

5.2 Oversampling

As we said in subsection 3.1 we have an imbalanced training set, we decided to perform oversampling so that we could have the same number of real and unreal tweets. We used the imbalanced-learn API [5] and more specifically the RandomOverSampler to perform this task. During this task, the library samples some tweets in the smaller class and insert them again in the dataset.

At the end of this process, we created an oversampled dataset with 4342 tweets in the unreal class and 4342 in the real one.

[4] glove.twitter.27B.200d: https://nlp.stanford.edu/projects/glove/.
[5] Imbalanced-learn API: https://imbalanced-learn.readthedocs.io/en/stable/index.html.

5.3 Data Augmentation

We described our dataset in the section 3 and we saw that we don't have too many data to train our models. Therefore we decided to try two data augmentation techniques.

5.3.1 EDA: Easy Data Augmentation

The first technique we tried is called EDA [12] and it allowed us to augment our training dataset adding more tweets that contain synonyms of the words that we had in our original tweets. We produced 3 new tweets for each tweet present in our dataset keeping the original target in the new data.

5.3.2 GPT-2

We also made another experiment using GPT-2 [13]. In this case, we generated new tweets starting from our dataset and using the 117M pre-trained model.

This approach is different from the one we used in EDA. With EDA we just changed some words in every tweet, with GPT-2 instead we generate a brand new tweet based on the tweet that we gave as input to this pre-trained model.

Using this technique we did not have an improvement in term of accuracy concerning the architectures we described in the previous sections trained with the original dataset. We trained the model using these data and then we tried it using the test set provided by Kaggle. The accuracy that we get is usually worse than the one that we got with the other models we described in the previous sections trained with the original dataset.

5.3.3 Back Translation

Another technique that we used to augment the dataset is Back Translation [14]. We translated all our tweets from English to French and then back to English, doing so we obtain a tweet that has the same meaning of the original one but it is different in term of words because in some cases the translation will use synonyms of the words.

At the end of this process, we obtained a dataset with 15.900 samples.

5.3.4 A few comments on the Data Augmentation Techniques

We tried data augmentation techniques and got new data, we used them to train our models and to understand which was the best augmented dataset that better fit the necessity of this task. None of the augmented datasets allowed us to obtain a better score on the Kaggle's test set. We compared the three techniques and we can conclude that for our task EDA is the best solution (in term of accuracy that we got on the test set), we always had bad performances with the dataset augmented using the other two techniques. We guess that GPT-2 and the BackTranslation produce training examples that are too different from the one that we have in our original dataset.

We will present in section 6 the score that we got using the EDA augmented dataset with all our models.

6 RESULTS

In this section, we provide all the results of the models described in Section 4.

We present the results considering 3 types of datasets, Original, Oversampled and Augmented using EDA. All the following results were obtained using our models with the test set provided by Kaggle. We produced the predictions and then we uploaded them on Kaggle to obtain a score. In the following sections, we can see how some base model works better for this task, and more "clever and complete" model did not. As we can see with a small number of parameters (Naïve Bayes) we can achieve good results as having a large number of parameters (BERT).

6.1 Score with Naïve Bayes

Dataset	Kaggle Score
Original	0.7983
Oversampled	0.7943
Augmented	0.7511

Table 1. Performances of Naïve Bayes Classifier

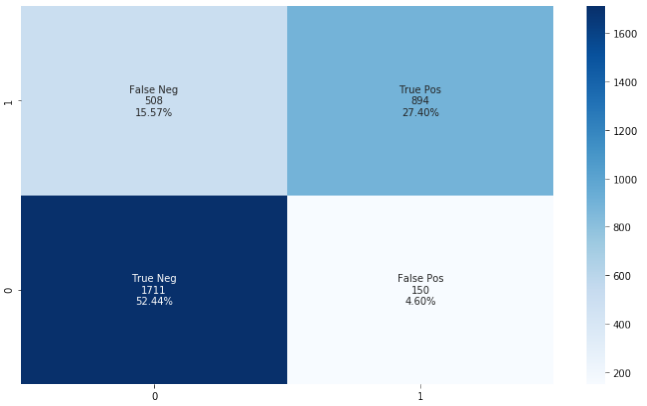


Fig. 10. Confusion matrix for Naïve Bayes classification on Kaggle test data

6.2 Score with Neural Network

Model	Dataset	Kaggle Score
Neural Network	Original	0.7428
Neural Network	Oversampled	0.7603
Neural Network	Augmented	0.6585

Table 2. Performances of Neural Network Classifier

6.3 Score with RNN

Model	Dataset	Kaggle Score
Recurrent Neural Network	Original	0.6380
Recurrent Neural Network	Oversampled	0.7428
Recurrent Neural Network	Augmented	0.5703

Table 3. Recurrent Neural Network performances

6.4 Score with LSTM and GRU (also bidirectional)

Model	Dataset	Kaggle Score
LSTM	Original	0.7836
Bidirectional LSTM	Original	0.7716
GRU	Original	0.7817
Bidirectional GRU	Original	0.7722
LSTM	Oversampled	0.7588
Bidirectional LSTM	Oversampled	0.7719
GRU	Oversampled	0.7842
Bidirectional GRU	Oversampled	0.7860
LSTM	Augmented	0.7355
Bidirectional LSTM	Augmented	0.7235
GRU	Augmented	0.7162
Bidirectional GRU	Augmented	0.7137

Table 4. Performances of LSTM and GRU

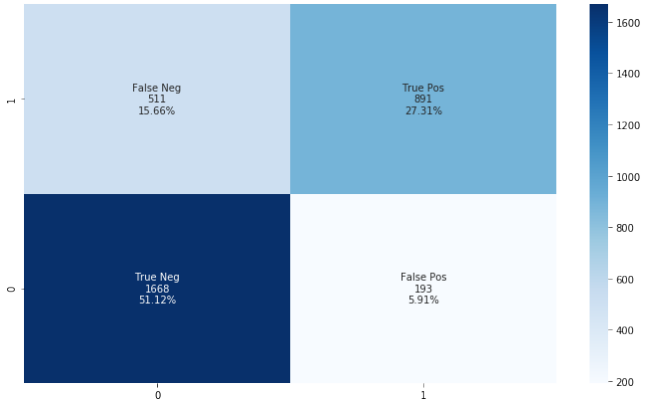


Fig. 11. Confusion matrix for GRU (trained on Oversampled data) classification using Kaggle test data

6.5 Score with Deeper Models

Dataset	Kaggle Score
Original	0.7808
Oversampled	0.7652
Augmented	0.7030

Table 5. Performances of Deeper model

6.6 Score with Keras Functional API

Dataset	Kaggle Score
Original	0.7781
Oversampled	0.7631
Augmented	0.7189

Table 6. Performances of the model developed using Keras Functional API

6.7 Score with Bert

Model	Dataset	Kaggle Score
BERT	Original	0.8345
BERT	Oversampled	0.8179
BERT	Augmented	0.7857
Bert with LSTM on top	Original	0.8338
Bert with LSTM on top	Oversampled	0.8185
Bert with LSTM on top	Augmented	0.7943

Table 7. Performances of BERT Classifier

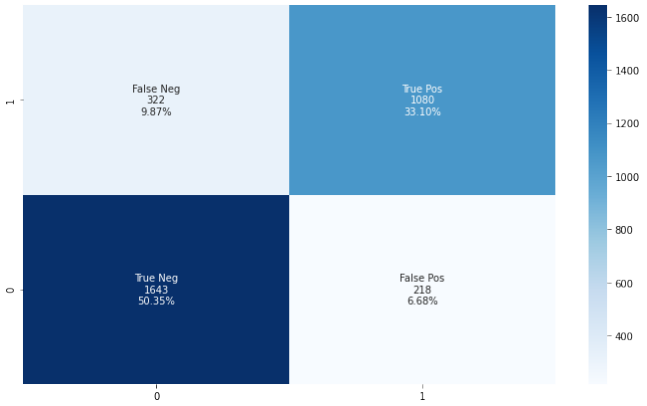


Fig. 12. Confusion matrix for Bert (fine-tuned on original data), classification using Kaggle test data

6.8 Comparisons

After trying all the models we decided to understand which model is more preferable than another and why.

As we said we have a small dataset and so we got good results using Naïve Bayes that outperform all the neural models except Bert. Due to small dataset, a simple Neural Network can not perform correctly this task.

Surprisingly the LSTM and GRU outperform the respective Bidirectional models and we guessed that it is caused by the small dataset and maybe by the small size of the tweets. Bidirectionality models would be a good solution if we had gotten more data and more complex structures, in fact with fewer parameters (only one direction) we obtain in major cases best results.

We can also state that using Bert and Bert with LSTM, thanks to the available pre-trained models, we can obtain the best score for our task.

Among these two Bert models, we choose **BERT without LSTM**, on top of it, because it is a simpler model with fewer parameters and the results are not too much different. [15].

Model	Kaggle Score	Type dataset
Naïve Bayes	0.7983	Original
Neural Network	0.7603	Oversampled
Vanilla RNN	0.7425	Oversampled
LSTM	0.7836	Original
GRU	0.7842	Oversampled
Bidirectional LSTM	0.7719	Oversampled
Bidirectional GRU	0.7860	Oversampled
Multi Layer Model	0.7808	Original
Keras Functional Model	0.7781	Original
Bert Model	0.8345	Original
BERT with LSTM on top	0.83389	Original

Table 8. Comparison of the performances obtained using different models

7 CONCLUSIONS

We discussed a lot of models that can be used for this specific task. We analyzed all the components for each architecture trying to better understand which model was the best. We started with basic approaches moving gradually to "clever and more complete" models.

In the experiments, we had some problems due to small dataset and unsuitable resources that did not allow us to explore the entire research space.

Anyway, the best model (or at least it seems) that we found is BERT, because of large pre-training phase done by the creators that allow us to fine-tune the last layer added in it to make predictions for this task.

As we said, we tried also data augmentation techniques without obtaining any better results. We wanted to do more experiments and more analysis for this task; due to COVID-19 however, the synchronization of different works and results, done by us, was difficult. Moreover, due to the computation, and not the best performances of our laptops, we did not try a complete grid search.

Finally, we compared our results with the public Kaggle LeaderBoard. We got results that are comparable to those obtained by the other participants of the challenge; excluding the results with 100% of accuracy that are probably not correct or real.

REFERENCES

[1] François Chollet et al. Keras. https://keras.io, 2015.

[2] Reem ALRashdi and Simon O'Keefe. Deep learning and word embeddings for tweet classification for crisis response, 2019.

[3] Kevin Stowe, Michael J. Paul, Martha Palmer, Leysia Palen, and Kenneth Anderson. Identifying and categorizing disaster-related tweets. In *Proceedings of The Fourth International Workshop on Natural Language Processing for Social Media*, pages 1–6, Austin, TX, USA, November 2016. Association for Computational Linguistics.

[4] Graham Neubig, Yuichiroh Matsubayashi, Masato Hagiwara, and Koji Murakami. Safety information mining — what can NLP do in a disaster—. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 965–973, Chiang Mai, Thailand, November 2011. Asian Federation of Natural Language Processing.

[5] Tomas Mikolov and Geoffrey Zweig. Context dependent recurrent neural network language model. Association for Computational Linguistics.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[7] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

[8] Arun S. Maiya. ktrain: A low-code library for augmented machine learning. *arXiv*, arXiv:2004.10703 [cs.LG], 2020.

[9] Zac Farnsworth Adam Thorne and Oscar Matus. Research of lstm additions on top of squadbert hidden transform layers.

[10] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[11] Nicholas Dingwall and Christopher Potts. Mittens: An extension of glove for learning domain-specialized representations. *CoRR*, abs/1803.09901, 2018.

[12] Jason Wei and Kai Zou. EDA: Easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6383–6389, Hong Kong, China, November 2019. Association for Computational Linguistics.

[13] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[14] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale, 2018.

[15] Carl Edward Rasmussen and Zoubin Ghahramani. Occam's razor. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 294–300. MIT Press, 2001.