TAAC Practical Work

Natural Language Processing with Disaster Tweets

João Fragão (up202204384)
Luís Henriques (up202204386)
Tomás Rodrigues (up202202467)

***Abstract*— This Natural Language Processing (NLP) project aims to address a binary classification problem involving disaster tweets. We implemented multiple pre-processing techniques, such as key-words engineering, lemmatization and Byte-Pair Encoding (BPE), and experimented with different setups, such as the vocabulary size and pre-training sub-word embeddings with either a Continuous Bag-of-Words (CBOW) model or a Skip-gram model. We implemented a Fully-Connected Neural Network (FCNN) with TF-IDF features, Recurrents Nets (RNN) with Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) cells, Encoder Transformers trained from scratch, and pre-trained encoder transformers, such as BERT. To assess the model's performance, considering the metric used by the respective Kaggle competition, the classification metric we gave more emphasis was the f1-score. The best model obtained has a f1-score of 81.13% on the test data.**

I. INTRODUCTION

As the communication is one of the most important aspects of out lives and as technology evolves, there's a large number of emerging internet platforms where people can share relevant insights on various topics as well as information they find interesting, such as social media; one example of such platforms is X (former Twitter). For this project, several thousand tweets from people either mentioning disasters, or not, will be used.

A. What is Natural Language Processing

The term Natural Language Processing (NLP) is referring to a branch of Data Science that enables computers to understand and process text and spoken words in a similar way that humans can. This field combines computational linguistics with statistical, machine learning and deep learning models. To this date, NLP consists of several tasks such as speech recognition, named-entity recognition, sentiment analysis, natural language generation, word sense disambiguation and others.

B. Objectives

This project's goal is to implement and evaluate several methodologies, such as RNN's with and without attention, transformer encoders trained from scratch and pre-trained ones, as well as conducting comprehensive and insightful experiments. On the other hand, we will also work on the interpretability of these methodologies, more specifically on the transformer encoders by interpreting and retrieve interesting insights from their attention maps.

C. Data Analysis

Data Analysis is not a mainstream point of our project. However, one important thing to take into account is the target distribution. Since we are working with a binary target, if the label was highly unbalanced a metric such as the accuracy would not be a good and important metric to compute.

There're 4342 non-disaster tweets (label 0) and 3271 disaster tweets (label 1), which corresponds to 57% and 43% accordingly. Therefore, the data is well balanced, and therefore accuracy will be used as a secondary classification metric.

II. PRE-PROCESSING

The initial step to deal with raw tweets was to remove URL's, non-alphabetic characters, lowercase the text, remove all one-character words and stop words from the english language, and then lemmatized the text using the spaCy tool.

With the tweets cleaner, we implemented Byte-Pair Encoding (BPE) into out pre-processing pipeline in order to tokenize the text. Depending on the model, several tokenization implementations of BPE were used. For example, for our models trained from scratch we used our own implementation of BPE with a vocabulary size of 20,000 tokens and a maximum length of 50. For pre-trained models, we retrieved the respective BPE tokenizer for the models, as all pre-trained models used apply BPE into their tokenization step.

That being said, there're two tokenizers that don't use the cleaner tweets, which are the tokenizers for the BERTweet base and BERTweet large models. These tokenizers are able to deal with raw tweets directly.

III. FEATURE EXTRACTION FROM TEXT

Because ML models aren't able to deal with tokenized text as they use mathematical operations, we extracted mathematical features that, in some way, reflect the characteristics of the tokens. For that, two main approaches were followed: one was the extraction of TF-IDF features and the other the extraction of dense token embeddings using two Word2Vec models, Continuous Bag-of-Words (CBOW) and Skip-gram.

So, instead of our models trained from scratch to learn embeddings with random weights for each semantic meaning of each token, we trained a CBOW and a Skip-gram model with the training data, and then gave those embeddings

directly to our models. Both models were trained with a vector size of 512 embeddings for each token, a window of 10 tokens, and a total of 256 epochs.

This feature extraction step wasn't implemented for the pre-trained models as they already have pre-trained token embeddings.

IV. Model Implementation

A. FCNN with TF-IDF features

1) ***FCNN:*** The most simple neural network implemented was a FCNN with 2 hidden layers of 2048 perceptrons each, ReLU activations, dropout with a rate of 0.1, and batch normalization. This network has one output node which returns logits.

B. RNN's with Word2Vec dense embeddings

Several experiments were done with RNN's in order to check what worked best. Initially, we compared training performance of LSTM models between the usage of CBOW or Skip-gram embeddings. Then, we added an extra type of RNN, GRU's, where we also implemented different embeddings extracted from different Word2Vec models.

The problem with these RNN's is that, although they're able to forward pass information even with a long sequence (specially the LSTM), for text classification we have a many-to-one architecture, meaning that only the last recurrent output is used for classification. So, we implemented attention into these RNN's. The attention mechanism assigns weights to different parts of the input sequence, dynamically emphasizing the most important tokens for the current task. This selective attention helps the model focus on the most relevant aspects of the text, improving its overall accuracy and performance.

We started with an attention layer with a single head, and then for the RNN with better performance we implemented a multi-head attention layer.

For all of these models we used our Word2Vec embeddings trained outside of the model's training cycle.

1) ***LSTM networks:*** We implemented three LSTM models, one without attention, the other with one layer with one attention head, and the other with multi-head attention. In fact, all three models are Bidirectional LSTM's (Bi-LSTM) with one recurrent layer each. The first two models have 64 hidden state features, while the third has 1024 features. The embeddings could either be frozen or not. Then, the outputs from the LSTM layer (weighed or not) were given to the classification head of the network, where either one or two fully-connected (FC) layers were implemented. The multi-head attention Bi-LSTM has a classification head of 2 FC layers, where batch normalization and dropout of 0.1 were implemented. All three networks have one output node which return logits.

2) ***GRU networks:*** We implemented two GRU models, one without attention and the other with one layer with one attention head. In fact, all these two models are Bidirectional GRU's (Bi-GRU) with one recurrent layer each. The models have 64 hidden state features. The embeddings could either be frozen or not. Then, the outputs from the GRU layer (weighed or not) were given to the classification head of the network, where either one (FC) layer, the output layer, was implemented. All GRU networks have one output node which return logits.

C. Transformers trained from scratch

Because we don't want to generate any text, there's no need for a decoder on the transformer. So, for text classification, we implemented transformers with only an encoder.

In the first transformer encoder, the encoder was implemented using the PyTorch function TransformerEncoder, while in the second transfomer the encoder was implemented from scratch ($\text{TransformerEncoder}_{\text{scratch}}$), from the scaled dot-product attention to the complete multi-head attention mechanism, all the way up. Another difference is that on the TransformerEncoder the positional encodings are embeddings the transformer learns, while on the $\text{TransformerEncoder}_{\text{scratch}}$ the positional encodings are computed with sinusoidal functions.

A significant advantage of transformers over RNN's (even with attention) is that in a transformer all tokens are processed at the same time (in parallel) in order to give a prediction, while on RNN's the tokens are processed one at a time (sequentially) given the output of one token depends on all the previous outputs.

1) ***TransformerEncoder:*** We implemented the TransformerEncoder with either CBOW or Skip-gram embeddings, followed by token embeddings and positional encodings. These features were added together, and to them layer normalization and dropout were applied. This output is then given to the encoder itself. The encoder's output is flatten in the sequence and feature dimensions, while keeping the mini batch size dimension. This transformed output is then given to the head of the transformer, where a ReLU and batch normalization are applied to a FC layer, and then the data is passed to the output layer with one neuron, returning logits. It was used a mini-batch size of 32 tweets, with 16 multi-head attention layers each with 16 heads, the FCNN inside the encoder has a hidden size of 2048 neurons, dropout rate of 0.1 and non-frozen token embeddings.

2) $\boldsymbol{\text{TransformerEncoder}_{\text{scratch}}}$***:*** The implementation of the $\text{TransformerEncoder}_{\text{scratch}}$ was pretty much the same as on the TransformerEncoder. The only two differences between these two transformers is the positional encodings and the fact that this transformer has an encoder implemented from scratch.

D. Pre-trained Transformers

A limitation of transformers trained from scratch is that they have only seen the data used for the problem at hand. More often than not Transfer Learning is used because knowledge gathered by the models in one use case could be very useful for other problems. Therefore, BERT has been widely adopted in a variety of downstream NLP tasks, such

as text classification, named entity recognition, and question answering.

There're transformers that have been trained with a large vocabulary size and with large amounts of text data, and therefore are a great start for fine-tuning models.

All the pre-trained transformers used were trained on lower-case english text, following the pre-processing implemented in this project.

1) ***BERT****:* Bidirectional Encoder Representations from Transformers, or BERT, was introduced by [2], and can be adjusted to a vast amount of task by just adding an output layer to it.

Two BERT models were used, $\text{BERT}_{\text{Base}}$ and $\text{BERT}_{\text{Large}}$, with 110M parameters and 340M parameters, respectively.

a) $\boldsymbol{BERT_{Base}}$*:* This pre-trained transformer encoder has a vocabulary size of 30522, 12 multi-head self-attention layers with 12 attention heads each, hidden size of 3072 on the FCNN inside the encoder, a hidden size of 768 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1, and layer normalization.

For this, and the $\text{BERT}_{\text{Large}}$ model, we used the model's adaptation to text classification implemented in a tool called hugging face, where several pre-trained transformers can be used.

b) $\boldsymbol{BERT_{Large}}$*:* In the case of the $\text{BERT}_{\text{Large}}$, we have a vocabulary size of 30522, 24 multi-head self-attention layers with 16 attention heads each, hidden size of 4096 on the FCNN inside the encoder, a hidden size of 1024 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1, and layer normalization.

2) ***DistilBERT****:* DistilBERT is a distilled version of BERT. DistilBERT was trained using a technique called Knowledge Distillation, where a lighter models tries to mimic the behaviour of a larger model.

To our knowledge, there's only one version of DistilBERT, which is the $\text{DistilBERT}_{\text{Base}}$. While $\text{BERT}_{\text{Base}}$ has 110M parameters, $\text{DistilBERT}_{\text{Base}}$ has 66M parameters, and according to [3] benchmarks show that their performance is very close, preserving around 95% of the performance of the teacher model. Also, $\text{DistilBERT}_{\text{Base}}$ runs 60% faster.

a) $\boldsymbol{DistilBERT_{Base}}$*:* This pre-trained transformer encoder has a vocabulary size of 30522, 6 multi-head self-attention layers with 12 attention heads each, hidden size of 3072 on the FCNN inside the encoder, a hidden size of 768 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1 (and 0.2 on the classification head), and layer normalization.

3) ***roBERTa****:* Robustly optimized BERT approach, or roBERTa, is a model developed by [4]. The study done to build BERT was replicated, however there were some changes. Relatively to BERT, roBERTa was trained for a longer time, with bigger mini-batch sizes and with more training data, as well as being trained with longer text sequences. Also, the next sentence prediction was a goal removed, and dynamically changing the masks of the training data.

Two BERT models were used, $\text{roBERTa}_{\text{Base}}$ and $\text{roBERTa}_{\text{Large}}$, with 125M parameters and 355M parameters, respectively.

a) $\boldsymbol{roBERTa_{Base}}$*:* This pre-trained transformer encoder has a vocabulary size of 50265, 12 multi-head self-attention layers with 12 attention heads each, hidden size of 3072 on the FCNN inside the encoder, a hidden size of 768 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1 (with no dropout on the classification head), and layer normalization.

b) $\boldsymbol{roBERTa_{Large}}$*:* In the case of the $\text{BERT}_{\text{Large}}$, it has a vocabulary size of 50265, 24 multi-head self-attention layers with 16 attention heads each, hidden size of 4096 on the FCNN inside the encoder, a hidden size of 1024 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1 (with no dropout on the classification head), and layer normalization.

4) ***BERTweet****:* BERTweet was introduced by [5], where the BERT architecture is used together with the roBERTa procedure. This model was specifically trained only on English tweets, stacking to 80GB corpus of 850M English tweets (873M for $\text{BERTweet}_{\text{Large}}$).

Given the problem of this project, it seemed ideal to use such a model that was specifically tailored for tweet-like syntax. Not only that, but also considering it gathered all the state-of-the-art advances, together with a huge tweet corpus of training data.

Two BERTweet models were used, $\text{BERTweet}_{\text{Base}}$ and $\text{BERTweet}_{\text{Large}}$, with 135M parameters and 355M parameters, respectively.

a) $\boldsymbol{BERTweet_{Base}}$*:* This pre-trained transformer encoder has a vocabulary size of 64001, 12 multi-head self-attention layers with 12 attention heads each, hidden size of 3072 on the FCNN inside the encoder, a hidden size of 768 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1 (0.3 on the classification head), and layer normalization.

b) $\boldsymbol{BERTweet_{Large}}$*:* In the case of the $\text{BERTweet}_{\text{Large}}$, it has a vocabulary size of 50265 (lower than the base version), 24 multi-head self-attention layers with 16 attention heads each, hidden size of 4096 on the FCNN inside the encoder, a hidden size of 1024 for the features inside the encoder, such as Keys, Queries, and Values, GeLU as the activation function inside the encoder, dropout rate of 0.1 (0.3 on the classification head), and layer normalization.

E. Training Process

We used two optimizers, AdamW for all transformers and Adam for the RNN's and FCNN models. Also for the transformer, we applied a linear learning rate scheduler with warmup of 10% of the training steps, as different

transformers were trained with different epoch numbers (2-4 epochs for pre-trained transformers and 8-24 epochs for transformers trained from scratch).

We did the usual hyper-parameter tuning, where we, looking at the results, changed the hyper-parameters we thought needed tuning. For example, if the training loss increases it's an indication that the learning rate is too large, or if the training loss is much lower than the test loss a higher dropout rate or regularization lambda would be used.

V. Results

The table below presents the outcomes corresponding to various combinations of preprocessing techniques and models. The reported metrics reflect the performance on the test dataset during the final epoch of each model, for the sake of simplicity. It's important to note that only the metrics from the test dataset are presented, excluding those from the training or validation datasets.

Model	Loss	Accuracy	F1-score
FCNN with TF-IDF	0.7249	67.36%	61.78%
LSTM Skip-gram	0.6834	57.03%	0.00%
$\text{LSTM}_{\text{Attention}}$ Skip-gram	0.4942	78.49%	71.06%
$\text{LSTM}_{\text{Multi-Head Attention}}$ Skip-gram	0.8627	72.42%	70.53%
LSTM CBOW	0.6844	57.03%	0.00%
$\text{LSTM}_{\text{Attention}}$ CBOW	0.4418	80.17%	75.37%
$\text{LSTM}_{\text{Multi-Head Attention}}$ CBOW	1.0553	68.00%	69.26%
GRU Skip-gram	nan	57.03%	0.00%
$\text{GRU}_{\text{Attention}}$ Skip-gram	nan	57.03%	0.00%
GRU CBOW	nan	57.03%	0.00%
$\text{GRU}_{\text{Attention}}$ CBOW	nan	57.03%	0.00%
EncoderTransformers Skip-gram	0.4975	76.86%	66.04%
EncoderTransformer CBOW	0.4679	78.24%	72.31%
$\text{TransformerEncoder}_{\text{scratch}}$ Skip-gram	0.4705	78.09%	68.74%
$\text{TransformerEncoder}_{\text{scratch}}$ CBOW	0.4550	78.85%	73.36%
$\text{DistilBERT}_{\text{Base}}$	0.4070	82.90%	78.51%
$\text{BERT}_{\text{Base}}$	0.4087	82.68%	78.82%
$\text{BERT}_{\text{Large}}$	0.6836	57.03%	0.00%
$\text{roBERTa}_{\text{Base}}$	0.4399	80.82%	77.38%
$\text{roBERTa}_{\text{Large}}$	0.6844	57.03%	0.00%
$\text{BERTweet}_{\text{Base}}$	0.4088	84.00%	80.72%
$\text{BERTweet}_{\text{Large}}$	**0.4070**	**84.46%**	**81.15%**

TABLE I
Final results

A. Results Analysis

Looking at the results, we can make several analysis. Firstly, models trained from scratch with CBOW embeddings as pre-training step gave better results than the ones with Skip-gram. On the other hand, larger models such as $\text{roBERTa}_{\text{Large}}$ and $\text{BERT}_{\text{Large}}$ gave bad results. We believe this might be due to the fact that for a relatively simple and light problem these larger models tend to overfit more, although we're not quite sure.

Something that we realized is that models like the LSTM, regardless of Word2Vec models, perform better with attention, and attention performs better than Multi-Head Attention.

Both TransformerEncoder and $\text{TransformerEncoder}_{\text{scratch}}$ perform quite well. The TransformerEncoder and the $\text{TransformerEncoder}_{\text{scratch}}$ CBOW reach an f1-score of 72% and 73%, and with Skip-gram, 66.04% and 68.74%, accordingly.

Regarding the pre-trained transformers, their Large versions usually do not perform good, with the exception of $\text{BERTweet}_{\text{Large}}$, which not only performs good, but is also a top performer. $\text{BERT}_{\text{Large}}$ and $\text{roBERTa}_{\text{Large}}$ do not perform well when compared to their baseline. $\text{DistilBERT}_{\text{Base}}$ performs quite good as well.

Overall, the best models were the BERTweet pre-trained transformers. Their performance is pretty much the same, although we're able to fine-tune a $\text{BERTweet}_{\text{Base}}$ model with a f1-score of 81.15%.

B. Interpretability Analysis

Regarding interpretability of the transformer models, we implemented a pipeline to extract the attention maps. The attention map in Fig. 1 comes from the $\text{BERTweet}_{\text{Large}}$ model. This tweet comes from the test data, and it refers to a disaster. We can see that the token "Ap" gives much more importance (weight) to the token "ocalipse", as well as the token "wild" gives much more weight to the token "fires". We believe this happens because the embeddings of these tokens are quite similar, given that the words "Apocalipse" and "wildfires" exist. Not only that, but because we're detecting disasters in tweets, when the token "wild" appears the model looks for "fires", as this combination is a strong indicator of a disaster being mentioned.

Fig. 1. Attention map from $\text{BERTweet}_{\text{Large}}$

VI. CONCLUSION

One of the things we gained by doing this project is the know-how of what methodologies could be more interesting to try out in NLP, such as adding attention to RNN's, as well as the usage of sub-word tokenization compared with word tokenization and character tokenization. Also, this project enabled us to better understand how the encoder of a transformer works by implementing one from scratch.

One thing we learned is that models usually perform better with attention, rather than without it. We realized that implementing a pre-trained transformer can be more a interesting solution in terms of results. Balancing both performance and computational power could be an important fact to take into account before implementing a model.

There some things we wanted to do more, such as using an LLM, such as GPT-2, to generate more training data. DL models are very data hungry, and even more when it comes to NLP tasks, so we believe that by increasing the training data with synthetic tweets the models would perform better.

One of the main difficulties we encountered was the implementation of the BPE algorithm given that it was confusing at first when looking for some already made implementation.

REFERENCES

[1] https://medium.com/data-from-the-trenches/decoding-nlp-attention-mechanisms-38f108929ab7

[2] Devlin, J., Chang, M.W., Lee, K. and Toutanova, K. (2019) BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 1, 4171-4186

[3] Sanh, V., Debut, L., Chaumond, J., Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. ArXiv, abs/1910.01108

[4] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. ArXiv, abs/1907.11692

[5] Nguyen, D.Q., Vu, T., Nguyen, A.G. (2020). BERTweet: A pre-trained language model for English Tweets. Conference on Empirical Methods in Natural Language Processing.

[6] https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb