Machine Learning CS-433

Class Project 2 - Tweets classification

Arthur Arnoux SCIPER: 342321
Émile Friot SCIPER: 343371
Grégoire Pèlegrin SCIPER: 344583

December 23, 2021

Abstract

This project is about analyzing the sentiment described in tweets by predicting if they used to contain a sad or happy smiley. We shall for that use Natural Language Processing techniques and Machine Learning models to be as precise as possible.

1 Introduction

During this project, we will use a basic NLP method that will be regarded as a baseline regarding the accuracy of our more complex Machine Learning models. First we will describe our way to prepare the data. Then, we shall talk about the process of transforming the text into relevant information for the model. Finally, we will explore the different models we built to surpass the baseline method.

2 Data exploration

One of the interesting aspects of the project is that the data only contains text from tweets, which happens to be less well written than Shakespeare's litterature. That includes typos, words from a specific jargon (internet), abbreviations, or non alphanumeric characters.

The data are given as two .txt files, which we concatenate in one .tsv file, shuffle and reforme as one train file (80% of the total) and one test file (20%). the new .tsv files will contain the sentences as well as their respective label. It will allow us to easily read the files and prepare the data for the model.

We will discuss later how the text will be transformed to be usable as input for a neural network.

3 Baseline method

As a first approach to the problem, we have decided to only use some NLP methods, such as n-grams and tokenization. Indeed, we can decide to not split the sentences only by words, but rather by tokens such as 1, 2 or 3-word(s), on which we can for example consider the surroundings other n-grams. By saving for every appearance of 1-2-3-grams the corresponding label (positive or negative sentiment), we can then be aware of the probability for each one to appear in a positive or negative sentence. For a new sentence, we can retrieve the probability of every n-gram in it, and compare them to decide the most probable label of the sentence.

Using this method, we achieve an accuracy of approximately 80% when predicting the labels

of the test data, when using the small dataset for the "training phase". This small dataset contains about 10% of the total data, and the latter contains about 2.5 million entries.

This baseline method and accuracy will permit us to know if a new method appears to be more or less effective, and we shall now see those other methods, including more Machine Learning parts.

4 Preprocessing

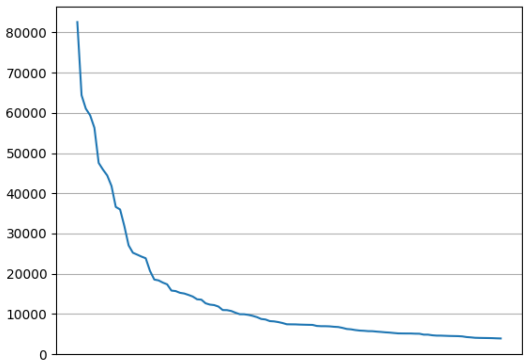
Given the fact that sentences come from twitter, we need to sanetize them as much as possible without giving up on too much information. We have used the nltk library to stem the words (eg. "working" → "work"), to remove the stop words (eg. "a", "for", "can"), and to remove non alphanumerical characters (eg. ","). However, the last one could contain more information than planned, as "..." may for example add a strong negative aspect to a sentence. We also replaced the numbers in the sentences by a <number> tag (like <user> and <url>), while trying to not misinterpret the number for words (eg. "u 2" could be "you too").

There exists a law called the "Zipf Law" arguing that the relevant words of a sentence, being the words that have an influence over its meaning, are usually not he most nor the least common words. This law follows a power law.

Thus, as we can observe in the figure 1, the distribution of words is very disproportionate (power law), as the words "i", "the", "to", "a", "and" are the 5 most common words with an occurrence up to 80,000. We decided to keep the words that have at most an occurrence up to 35,000, and that appear at least 2 or 3 times.

Finally, and before applying the law we just discussed, we also needed to fix the errors in words, such as typos. Indeed, a word could be underrepresented if errors are usually made while writing it, and the nltk library once again helped with the jaccard distance function to retrieve the word supposed to be initially written. However, the function seemed too costly to be computed for every sentences in the dataset.

Figure 1: Words distribution, top 100 and not from the full dataset



This whole preprocessing phase didn't prove to be as useful as expected, since it only helped a little for the first model we will present, and didn't change anything for the second one. It is difficult to evaluate either the processing was too strong and removed too much information in the sentence, or the rules applied were simply not very useful.

5 Word embeddings

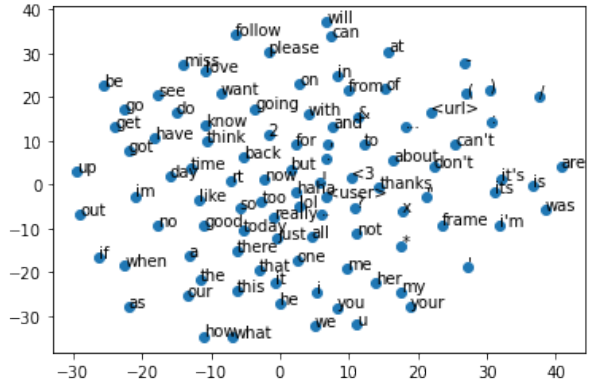
As we have studied neural networks during the course, we surely want to try to build and train one to answer the problem. However, the simple and unchanged words as inputs would not be the best approach since words don't have a real meaning for a machine. It could only sort of compare two words by checking the contained letters. For example, the words "hello" and "hell" may seem way more closer than "hello" and "greetings", although they are not perceived this way.

To remedy the problem, we will be using some word embeddings, transforming the words from the whole text data, into vectors, allowing them to have a spacial representation and so a distance to other words more meaningful.

However, to obtain those meaningful distance between words, we also need to train the vector space,

so that words that appears in some same context of other words are being attributed a close vector.

Figure 2: Word embeddings in a 2D space



On the figure 2, we can for example see on the bottom left that "if", "when" and "as" are in the same area of the vector space, just like "will" and "can" at the top of the image.

We used a Word2Vec model to which we gave all our known sentences and made it train so all the words we use have a good representation. We can observe an example on figure 1, although it has been reduced to a 2 dimension space for a simpler visualization.

In the end, we decided to use a GloVe pretrained representation with 200 dimensions, trained over a twitter dataset, that provided slightly better results.

6 Neural Network

6.1 What type of network ?

To come back to our problem, we need a Machine Learning model able to find out the sentiment label of a sentence. Considering the high dimensionality of our embeddings (100 to 300 depending on the model used), we will need to use a neural network to answer the curse of dimensionality.

Moreover, we can quickly see that the inputs given to the neural network will vary according to the length of the sentence, which would lead to a problem for a traditional multi-layer perceptron. We quickly tried using a MLP with the probabilities of each n-gram as inputs but it wasn't powerful enough.

Thus, we want to build a Recurrent Neural Network (RNN), allowing us to input sentences of different sizes, since the feed forward process is now saving a state value at every neurons, combining the current input and the last state. To achieve that, the model waits for the sentence as well as its length as inputs. It connects neurons between timestamps and the model can dynamically change depending on how the sequence is. Finnaly, the state value allows to track the information of all past words. The state and the output of each neuron (or LSTM block) are computed that in a way close to a normal neural network:

$$\text{state } h_t = \sigma(w_{hx}x_t + w_{hh}h_{t-1} + b_h)$$
$$\text{output } z_t = \sigma(w_{zh}h_t + b_z)$$

Still, there remain a problem, called "vanishing gradient problem", which declares that the first word's information will fade faster than the second one, and so on. It leads to a loss of information when reaching the more distant parts of the sentence, while its beginning could have potentially been a very important group of words.

The latter problem guides us to a Long Short Term Memory (LSTM) model, that uses more complex architectures for its neurons, mainly combining 3 gates, the forget gate, the input gate and the output gate, that introduces restrictive controls.

We can also use a Encoder-Decoder model. Its purpose is to encode a sequence fully with one model and use its representation to seed a second model that decodes another sequence. In our case, in can be seen like the encoder reads the input sentence while the decoder decides what smiley (happy or sad) it should use, giving us the sentiment initially felt in the sentence. The Encoder-Decoder model still contains LSTM blocks.

A last detail about RNNs and LSTMs is that they can't really decide weither nearby words should affect each other more than farther ones.

The idea to counter this is to use the output of the decoder LSTM to compute an "attention" over all the outputs of the encoder LSTM, giving a weighted average.

All those last paragraphs have finally led us to Transformers. For this final model, we will be using a pre-trained one (BERT), which is a huge model with a great ability to perform NLP-related tasks. It was initially trained from unlabeled data from Wikipedia, with around 2,500 million words.

6.2 What architecture ?

To make sure we are using a good network architecture, we have to make trials and errors by changing some hyperparameters each time. The best results/accuracy we have hit, was using a network containing 2 hidden layers with 128 LSTM blocks each, leading to a total of 5 million trainable parameters. Moreover, a probability of dropout of 50% was used.

To build the model, we have been using Pytorch offering a good range for possibilities. A summary of this model gives us:

```
RNN(
  (embedding): Embedding(25000, 200)
  (rnn): LSTM(200, 128, num_layers=3,
     dropout=0.5, bidirectional=True)
  (fc): Linear(in_features=256,
     out_features=1, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

To get a higher accuracy, we imported the BERT model whose encoder part contains 12 layers (or BERT blocks/layers), finished by one pooling layer, the BertPooler. It has over 100,000,000 trainable parameters, which is why we import it, freeze most of its parameters, to only train a part of the model over our own dataset. The training phase here is way longer than for the first model, and it appeared to be already good for the task before even training, showing the our best accuracy reached for this challenge right after the first epoch.

7 Results

Below are some results we have gathered after experimenting with our different methods.

	n-grams	LSTM	BERT
Raw data	0.83%	0.80%	0.88%
Preprocessed	0.83%	0.84%	0.88%

Table 1: Accuracy in predicting testing data, according to the method used

We can see that it is not trivial to get a significantly higher accuracy than basic but proven NLP methods, by simply using neural networks, and we need to use state-of-the-art models, such as Transformers (in our case the BERT model), to reach a notably higher accuracy.

8 Conclusion

During this project, we have been able to use algorithms and methods we have seen during the course in a practical manner, using mainly Pytorch for the neural network building, while also exploring some impressive state-of-the-art neural networks models and witnessing their potential.

We have achieved an accuracy of precisely 88.2% for the classification of sentiments in tweets. It feels like an honest accuracy given the fact that provided tweets are sometime very badly wirtten, use unknown slang, or simply that written text express less sentiment than orally, which led people to use smileys to specify their thoughts.

9 References

- Introduction to Natural Language Processing (CS-431)
- PyTorch Sentiment Analysis by bentrevett https://github.com/bentrevett/pytorch-sentiment-analysis