2024 AML Challenge 3 Group 20

Moaad MAAROUFI Barthelemy CHARLIER

EURECOM

Abstract

Sentiment analysis is still one of the most challenging areas in NLP due to the subjectivity of emotions, which are usually conveyed through more than just words, such as tone and sarcasm or facial expressions. This report will explore sentiment analysis on a tweet dataset in the classical sense of NLP, exposing a set of models and methods along with their limitations.

1 Introduction

In this report we will explore a tweet dataset from the Figure Eight's Data for Everyone platform and try to classify tweets as positive, negative or neutral. We will set two goals for this study:

- Compare how general yet powerful embeddings (OpenAI's embeddings) score vs specialized fine-tuned embedding for the task at hand.
- Investigate how small models perform against a behemoth multimodal LLM like GPT-4o, which a priori should have a better understanding of human given that it was pre-trained on the whole internet.

We will start with some baselines such as Naive Bayes, which will ignore a very important constraint (word precedence dependencies), and we will use Zero Shot on the embeddings. Then we will try to squeeze more performance from the embeddings by using KNN and a deep neural network. Next, we will move on to some memory models that are better suited for this task, such as LSTM and BERT, which will produce specialized embeddings encapsulating the contextual information in the tweets. We will try to improve upon their base versions using some engineering tricks.

2 Exploratory Data Analysis

The dataset is composed of a labeled training set and an unlabeled validation set. We will only explore the labeled set in the following analysis. The figure below shows that the neutral tweets have a count above the average, while the neutral and positive tweets have relatively the same count.

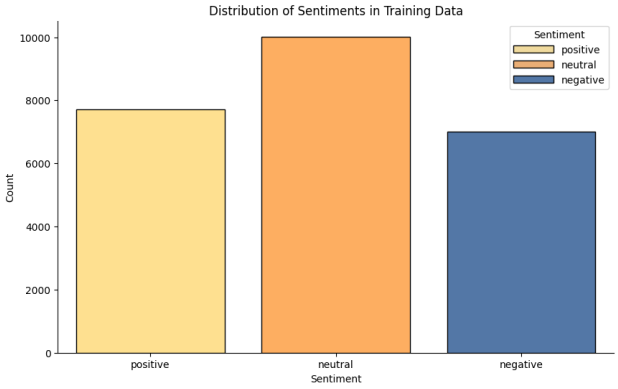

Figure 1: Distribution of sentiments in the training data

The following figure shows the most frequent words in each class after data cleaning (which we'll address in the next section). It's interesting that some words that don't seem positive or negative are very frequent in these classes, such as "get," "go," and "got." The Naive Bayes model will address this by giving little weight to these uninformative words, which are frequent and present in different classes.

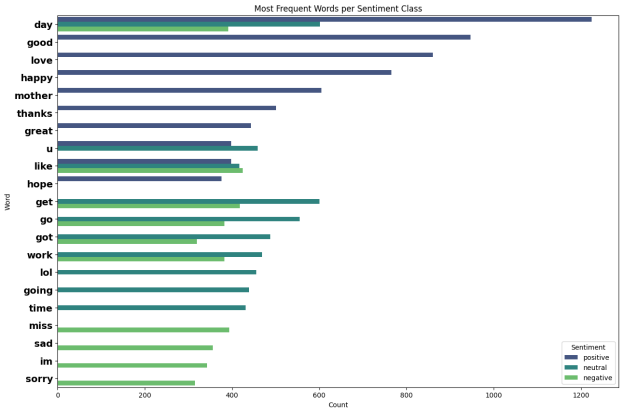

Figure 2: Most frequent words per sentiment class

Another interesting feature we should investigate is the length of the tweet to see if it is informative or not. The following box plot shows that this is not the case; the tweet classes have very similar distributions. Ignoring the outliers, the median, upper quartile, and lower quartile are very similar across classes.

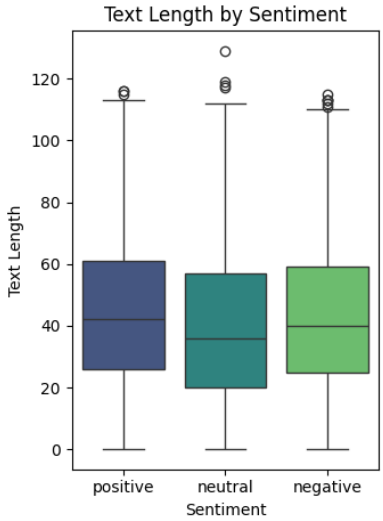

Figure 3: Box plot of text length by sentiment

3 Macro Average F1 score

We will report only this metric throughout the entire report and will refer to it simply as "score" to avoid redundancy. It takes the mean of recall and precision over all classes.

4 Data preprocessing

4.1 Techniques common to all models

Each of our models requires relatively different input, but we will try in this section to list the common pipeline and detail the specific processing steps for each model in the corresponding sections.

The common pipeline for all the models starts as follows: convert the tweets to lowercase (sometimes uppercase could convey emotion, but this is beyond the scope of this study), remove non-alphanumeric characters with regex, and remove uninformative characters in links such as "http" (we chose to keep the content of the links because they are very frequent in tweets and could convey some emotion). Then, tokenize using nltk.tokenize and apply a lemmatizer (to reduce words to their base forms). Next, remove stop words, and finally, concatenate the tokenized tweet back together into a single string.

The models do not take natural language tweets as direct inputs. Instead, most of them use a tokenizer, which maps each word to an index based on a predefined vocabulary. Other processing steps are unique to each model and will be discussed in the appropriate context to make sense.

The labeled training set was split in a stratified way (based on the sentiment in order to maintain the same proportions across all splits) into training **0.7%**, validation and testing each **15%**.

4.2 Labeling the validation data using GPT4o

GPT-4o is OpenAI's latest large language model, which is much faster and cheaper than the previous GPT-4 model, while maintaining the same level of reasoning. We thought it would be a good idea to let this model label the validation data, which was given to us unlabeled.

We wanted to explore how a large language model performs in this task rather than using a dedicated sentiment analysis model from GCP's API. To label the **2748** normalized tweets, it only cost us 0.75 EUR.

We gave the model a batch of 40 tweets at a time to reduce the number of API pings. The model was asked to provide the sentiment and the extracted text in JSON format. We considered using the extracted text, but eventually failed to do so due to time constraints. You can find the prompt in the appendix.

This batched approach reduces the risk of reaching API limits or getting throttled. Additionally, we used the Tenacity library to add exponential backoff to our requests and implement retry behavior according to good practices from the OpenAI API documentation.

Before starting to label, we need to check the performance of the model on the labeled training data instead of blindly trusting GPT. We asked it with the same prompt to label the labeled test set.

We expected the performance to be quite high, but the results were a bit disappointing. GPT-4o and GPT-3.5-turbo, given cleaned (lowercase, no links, etc.) tweets, scored **0.71** and **0.70** respectively. This is very unusual, given that these models were trained on the whole internet and should understand human sentiment well.

We manually investigated some tweets where the models were wrong. It appears that some labels are very controversial and that we ourselves would label them differently than the dataset's labels.

- **Tweet:** *Sigh*
  Predicted Sentiment: negative
  True Sentiment: neutral

- **Tweet:** *Wishes she had a pool*
  Predicted Sentiment: neutral
  True Sentiment: positive

- **Tweet:** *Lmao i know !! pleaseeee reply You got any replies yet? :L x*
  Predicted Sentiment: positive
  True Sentiment: neutral

Nonetheless, sentiment is very subjective, and GPT-4o aligns well with human sentiment and language. We think it is still worth seeing how our models will perform against a GPT-4o labeled dataset. We chose GPT-4o as it is the latest OpenAI model and benefits from multimodality. According to [2], multimodal LLMs have shown superior performance in common-sense reasoning compared to single-modality models, highlighting the benefits of cross-modal transfer for knowledge acquisition. Furthermore, GPT-3.5-turbo tends not to follow the prompt and JSON constraints if the batch size (number of tweets given in a single API call) is higher than 15.

4.3 OpenAI embeddings

We wanted to explore the embeddings route, which will enable us to move away from NLP and return to the more

familiar territory of machine learning. We decided to explore some commercial-grade embeddings, such as OpenAI embeddings, which are closed-source but are supposed to be quite good according to their claims. Additionally, we had some leftover credits, so we might as well use them before they expire.

We used OpenAI's best embedding model, "text-embedding-3-large." It is ranked 17th overall on the MTEB (Massive Text Embedding Benchmark) on HuggingFace Hub and 32nd in classification based on accuracy. We weren't satisfied with this ranking, so we considered using the open-source model ranked first (NV-Embed-v1) based on Mistral-7B-v0.1. We tried running it on a Colab A100 GPU (costing 1 EUR per hour), but it was quite slow, taking 20 seconds to produce a single embedding, which would be very costly. So, we gave up on that idea and stuck with OpenAI.

The OpenAI model produces an embedding of size **3072**. They also claim that if you truncate and normalize it, the performance wouldn't degrade much, which we will test here. The API calls are quite cheap, and we didn't encounter any rate limits or throttling, even with a dataset of **30,000** cleaned tweets for both training and validation (**0.13** dollars per **1M** tokens).

In the following sections we will explore how we can put these embeddings to good use and squeeze the most performance out of them.

5 Model selection

5.1 Baselines

5.1.1 Naive Bayes

A baseline model should ignore some constraints, return to familiar territory, and give us an idea of what we are dealing with and how much ignoring these assumptions/constraints would cost us. This is exactly what Naive Bayes will do, earning it the name "Naive."

Bayes' theorem is the basis of Bayesian machine learning. It is used to update prior beliefs with the observed data to get a new updated posterior, which we will use for inference. The catch here is that it assumes that the presence of a particular word in a tweet is independent of the presence of any other word, given the sentiment class, i.e., the observed data is i.i.d. (independent and identically distributed). This is not the case because words have contextual dependencies.

$$P(Y \mid X_1, X_2, \cdots, X_n) = \frac{P(X_1|Y)P(X_2|Y)\cdots P(X_n|Y)*P(Y)}{P(X_1)P(X_2)\cdots P(X_n)}$$

This is a very important baseline to show if these dependencies are informative or not. The model creates a sparse vector of word frequencies in each tweet. It scales down the frequencies of words that are frequent across many tweets in many classes, making them less informative. Then, the components of Bayes' theorem are calculated using these simple counts.

The model achieves a score of 0.57 on the test set and 0.53 on the GPT-4o labeled dataset. This proves that contextual dependencies are quite important in this task and shouldn't be ignored. Thus, our subsequent models will take this into account.

5.1.2 Zero shot

As a simple yet efficient way to evaluate the quality of the OpenAI embeddings, we will perform zero-shot inference without any training from these embeddings. We will do this by generating embeddings for the words "positive," "negative," and "neutral" and using the cosine similarity between each of these embeddings and the tweet embedding. We classify the sentiment based on the highest similarity score. This method yields a performance that is slightly better than random chance for the labeled test set **0.53** and **0.59** on the GPT-4o labeled data.

We thought that a single word ("negative," "positive," "neutral") might not be representative enough. So, we wrote some tweets filled with representative words for each class. We took the average embeddings of these tweets per class (3 tweets per class) and performed the same experiment. This approach slightly improved the performance to **0.56** on the test set and **0.64** on the GPT-4o labeled dataset.

This steered us in another direction: why not get the embeddings of actual training tweets and average them per class just like before? Then, we can calculate the cosine similarity of a new tweet with the average of each of the 3 classes. This approach resulted in a much bigger performance bump, achieving **0.70** on the test set and **0.76** on the GPT-4o labeled dataset. This is an interesting route, and we will try to pursue it further in the next sections.

5.2 K-Nearest-Neighbours

To continue using the training data embeddings, k-nearest neighbors (KNN) is one of the natural algorithms we could use. However, we face two problems: the curse of dimensionality, as our vectors are quite large, and the fact that scikit-learn (sklearn) doesn't support GPU-accelerated computing.

The first problem is actually an opportunity to test OpenAI's claim that their embeddings lose very little meaning when truncated, even to 256 dimensions. For the second problem, we will use cuML, which offers equivalent solutions to sklearn but with GPU acceleration support.

The difference is actually quite surprising. We managed to perform grid search for hundreds of values with cuML, unlike sklearn, which took 15 minutes and the model didn't even finish training.

We managed to push the performance up to **0.72** on the test set and **0.75** on the GPT-4o labeled data. We also tried truncating the embeddings to 512 dimensions to see if the curse of dimensionality was impacting the performance. We got slightly worse results, with scores of **0.69** and **0.71** on the GPT-4o labeled data and the test set, respectively. This slight decrease could be explained by the loss of some information due to truncating the embeddings. However, it's still very impressive that a fraction of the original dimension manages to achieve this score.

5.3 Feedforward Neural Network (FNN) Classifier

Now we move away from zero-shot inference and try to use a simple deep neural network as a classifier. We use a four-layer FNN with ReLU activation and dropout between each layer, and a softmax activation at the end. It was trained for only 4 epochs. It achieved a score of **0.76** from the very first epoch on the validation set and similarly at the end on the test set. It also achieved the exact same score on the GPT-4o labeled data.

The performance drops to **0.72** on both test sets when we truncate the embeddings to 512 which is still quite good.

As a conclusion, the model manages to generalize quite well to out-of-distribution data. The OpenAI embeddings are quite good, retaining most of their context even when truncated. However, they might be too general and not specifically suited for this task. We believe we can squeeze more performance from these deep models by fine-tuning the embeddings to the current task, which we will explore in the next sections.

6 LSTM

This is the first step towards training specialized embeddings. The OpenAI embeddings addressed a crucial aspect that Naive Bayes was missing: contextual dependencies. Unfortunately, the embeddings might be too general for the current tasks. The next step is to create specialized embeddings.

6.1 Base LSTM

LSTM is a gated model that can control memory flow and conserve relevant long-term dependencies. We will use this ability to create a vector from each tweet, leveraging these dependencies. We will use a tokenizer to index the words, as explained in the data preprocessing part. A vector of indices corresponding to a tweet will be fed to an embedding layer, which will create a 128-dense representation vector for each of the indices/words, thus creating a sequence of vectors or a matrix. These vectors will be fed to the LSTM sequentially, and it will eventually output a single vector summarizing the tweet. This vector will be given to two dense layers, which will perform the classification and output probabilities with a softmax.

The model scores **0.69** on the test set and **0.64** on the GPT-4o labeled data. While it performs better than the Naive Bayes baseline, it doesn't surpass the general OpenAI embedding. It was trained for 4 epochs and has **3M** trainable parameters. The model is overfitting to the training data, as the difference between the validation and training scores was substantial. Additionally, the random initialization of the embedding layers may not be optimal, and the tokenization process may also be suboptimal.

6.2 Bidirectional LSTM with dropout and GloVe embeddings

We try to remedy overfitting by adding dropout to the LSTM inputs and recurrent dropout to the LSTM's recurrent state. Furthermore, instead of initializing the weights randomly, we will use GloVe embeddings (pretrained vectors). We will also use bidirectional LSTM because we noticed that the upcoming model, BERT, performs very well and is bidirectional, so this might be a good choice.

The model was trained in the same way as before, scoring **0.74** on the test set and **0.68** on the GPT-4o labeled data. We noticed that while we managed to solve some problems, the model is now more inclined toward the labels of the dataset it was trained on, rather than the GPT-4o labeled data. This indicates a kind of trade-off between fitting the training data and generalizing to the GPT-4o labeled data.

7 Component wise fine-tuning of BERT

BERT [1] is an encoder-style transformer that was trained in a bidirectional manner on a large corpus of data. [3] used DistilBERT as a word embedding model and concluded that it surpasses all other common embeddings, such as GloVe, FastText, and Word2Vec, in the context of tweet sentiment analysis.

BERT uses its tokenizer to index the sentence using the vocabulary it was trained on. The embedding and encoding layers will learn a representation of these tokenized sentences during training, which the classifier will utilize (similar to LSTM, but with the attention mechanism of transformers instead of the gates of LSTM). We will first train the model end-to-end on the train set to get a baseline. Then, we will utilize an engineering trick recommended by the professor, which involves fine-tuning the model component-wise. First, we freeze the embedding layers and encoder, and train the classifier by disabling the gradients. Next, we unfreeze the embedding and encoder and freeze the classifier. Finally, we unfreeze the entire model and train it for one last pass end-to-end. The intent of this trick is to make the embeddings even more specialized. Each frozen component acts as a regularizer for the other component, reminding them of the objective. We hope that this approach will prevent overfitting and boost performance by creating a more specialized embedding.

We trained a BERT classifier without any layer freezing to establish a baseline and see if our component-wise fine-tuning would be effective. It was trained for 3 epochs on an L4 GPU with 23GB memory and a batch size of 64 (fully utilizing 21GB of GPU memory), comprising 110 million parameters. There was a performance bump compared to the FNN, with a score of **0.79** on the test set and **0.71** on the GPT-4o labeled data.

Next, we implemented the component-wise training. We froze the embedding and encoder layers and trained the model for one epoch (271*64 steps). Then, we froze only the classifier and trained the model for another epoch before unfreezing the whole model and training

it end-to-end for one last epoch. The model was trained for 3 epochs in total. We got a slightly worse result, with a score of **0.78** on the test set and **0.71** on the GPT-4o labeled data.

The decline in performance could be attributed to the fact that BERT was pre-trained end-to-end. Brutally freezing some components and continuing training may have hindered performance. A better approach might be to unfreeze layers gradually, layer by layer. Nevertheless, even the baseline experiment shows that specialized embeddings utilizing the attention mechanism manage to surpass the performance of general embeddings like OpenAI's.

It is also worth mentioning that the OpenAI embeddings agree more with the GPT-4o labeled data. Since OpenAI's methods are closed-source, we can only speculate that they probably use somewhat similar data in training, leading to the same biases.

8 Conclusion

We started with Naive Bayes to show that contextual dependencies of words are important in our case. Then we explored how we could squeeze more performance from the OpenAI embeddings using some very simple techniques. Next, we used specialized embeddings with LSTM and BERT, demonstrating that with some engineering tricks(some worked some didn't), we could achieve much better performance than with general embeddings.

We also showed that GPT-4o on the test set does not necessarily perform better than the smaller models. However, we noted that the labels were somewhat controversial, even for human labelers, so we decided it would be a good idea to compare the models against GPT-4o predictions on the unlabeled data.

Please note that we ran most of our models at least three times, and the results are consistent. We used random seeds for the test split and other functions that accept a random seed to make our work reproducible.

9 Future Work

It's a good idea to follow in the footsteps of [3] by using DistilBERT, which has 40% fewer parameters than BERT, or even a smaller version which can be tuned more efficiently before training a bigger version. We can use the same technique of freezing and unfreezing, but this time layer-wise rather than component-wise to make it less brutal. We can also explore creating synthetic tweets by asking GPT to generate some tweets for us based on the training data given in the prompt and play with the temperature to get different variations. Additionally, we can try using the validation set labeled by GPT in the training data to see if the performance improves on the test set.

References

[1] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[2] Jiayang Wu et al. *Multimodal Large Language Models: A Survey*. 2023. arXiv: 2311.13165 [cs.AI].

[3] Yichao Wu et al. *Research on the Application of Deep Learning-based BERT Model in Sentiment Analysis*. 2024. arXiv: 2403.08217 [cs.CL].

A GPT prompt for data Labeling using OpenAI API

```
messages=[
 {
   "role": "system",
   "content": "You are an expert in tweet sentiment  analysis. "
   "I will give you a batch of tweets and I want you
   to return one word based on the sentiment. Positive,

    Negative or Neutral. along with the part of the tweet
     that you used to chose the sentiment "

   "You will return the values in json format,
   where the key is the index of the tweet and the value is
    a list containing the sentiment and the extracted words
     that you used to chose the sentiment "
 },
 {
 "role": "user",
 "content": json.dumps({
 1: "yay to being smokefree  well done",
 2: "i hope you hoes are having so much fun not too much without me though lol",
 3: "wow  that s looks really good  i wish i had some  was it good",
 4: "i miss my friends so much",
 5: "i think i d be like phoebe s mom on friends   stop the movie before the sad part ",
 6: "can t sleep   feening for cigs  i m horrible   x" ,
 7: "hmm you can t judge a book by looking at its cover",
 8:  "says finally  im home   http plurk com p rr121",
 9: "talking to nat"
 })
 },
 {
 "role": "assistant",
 "content": json.dumps({
 1: ["positive", "yay well done"],
 2: ["positive", "so much fun"],
 3: ["positive", "really good"],
 4: ["negative, I miss my friends so much"],
 5: ["neagtive","sad"],
 6: ["negative","horrible"],
 7: ["neutral","Hmm..You can‘t judge a book by
 looking at its cover"],
 8: ["neutral","says Finally, Im home."],
 9: ["neutral","talking to nat"]
 })
 },
 {
 "role": "user",
 "content": structured_input

 }
  ]
```