Financial Sentiment Analysis

Machine Learning Project

Michal Švec

04.01.2025

Abstract

This project focuses on classifying text from a financial dataset into positive, negative, or neutral categories. The primary objective was to explore ensemble methods like bagging, boosting and XGBoost library. Previously, we had the opportunity to work with Word2Vec, TF-IDF, and Bag-Of-Words in tutorials, so this time we employed BERT embeddings as the text preprocessing method. Additionally, classification using a large language model (LLM) was conducted to evaluate its performance against more traditional methods.

All models achieved an accuracy of at least 60%, with most exceeding 65%. Among the traditional models, the best-performing one achieved an accuracy of 69.86%. The LLM outperformed all traditional approaches, achieving an accuracy of 75.11%.

1 Dataset Overview

The dataset [1] is pretty straightforward, consisting of only two columns: *Sentence* and *Sentiment*. The *Sentence* column is self-explanatory and the *Sentiment* column is a categorical variable that labels the sentences as Neutral, Negative, or Positive. The dataset combines information from two sources: FiQA (Financial Q&A) and the Financial PhraseBank.

Below are 3 samples from the dataset. One can observe that the sentences are not very structured and may contain various special characters.

Sentence	**Sentiment**
The company closed last year with a turnover of about four million euros.	Neutral
$SAP Q1 disappoints as #software licenses down. Real problem #Cloud growth trails $MSFT $ORCL $GOOG $CRM $ADBE https://t.co/jNDphllzq5	Negative
The pretax profit of the group's life insurance business increased to EUR36m from EUR27m.	Positive

Table 1: Example of the dataset

The table below shows the distribution of class labels. The dataset is unbalanced, there is a prevalence of neutral sentences.

Number of sentences	**Positive**	**Negative**	**Neutral**
5322	$\sim 32\%$	$\sim 15\%$	$\sim 54\%$

Table 2: Distribution of labels

For all the experiments, the following split in Table 3 is used. It is important to note that the dataset was split using *Scikit-learn*'s split function, which supports stratification. This ensures that labels are evenly distributed across the split datasets.

Dataset size	Train	Validation	Test
5322	$\sim 70\% = 4090$	$\sim 15\% = 876$	$\sim 15\% = 876$

Table 3: Dataset train, validation, test split

2 Text Preprocessing

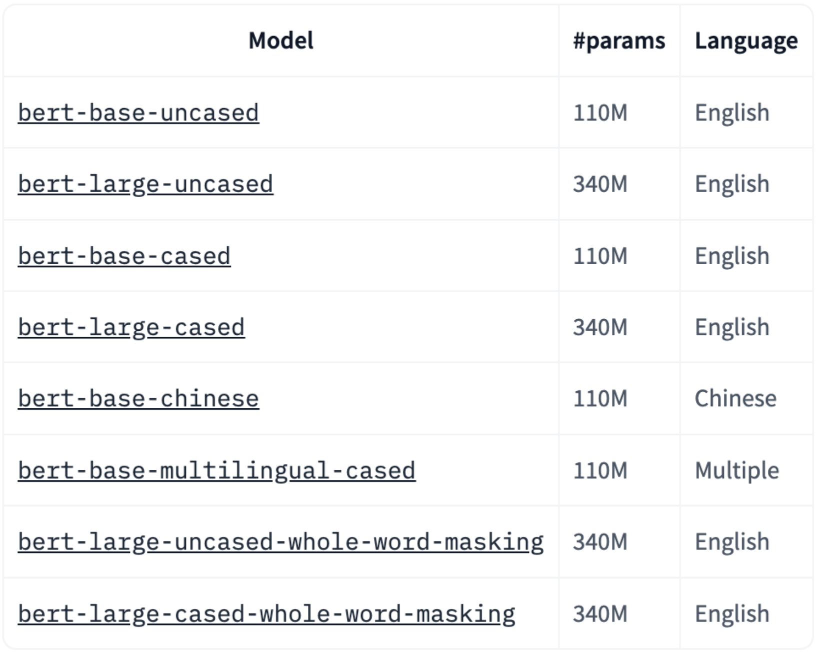
Since our dataset contains text, we need to perform text preprocessing before feeding it into a machine learning model. In the tutorials, we had the opportunity to work with Bag of Words, TF-IDF, and Word2Vec. This time, I decided to move directly to BERT model as a text preprocessing methods for the sake of learning something new.

BERT (Bidirectional Encoder Representations from Transformers)

BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained deep learning model developed by Google for natural language processing (NLP) tasks.

Unlike aforementioned techniques, BERT reads text and takes into account both the left and right context of a word in a sentence, which allows for a deeper understanding of the meaning.

There are many of variations of the BERT model. The two basic categories are large and base. The large model has more layers, more parameters and therefore it should perform better [2]. There is also a difference between uncased and cased, which indicates whether the model distinguishes between lowercase and uppercase letters. Additionally, there are multiple language variations.



Model	#params	Language
bert-base-uncased	110M	English
bert-large-uncased	340M	English
bert-base-cased	110M	English
bert-large-cased	340M	English
bert-base-chinese	110M	Chinese
bert-base-multilingual-cased	110M	Multiple
bert-large-uncased-whole-word-masking	340M	English
bert-large-cased-whole-word-masking	340M	English

Figure 1: BERT variations [3]

Apart from that, there are also multiple versions for specific use-cases [4]:

- **ALBERT** (A Little BERT)
    - modified to reduce the number of parameters and computational complexity

- **DistilBERT**
  - simplified for less computation resources
- **TinyBERT**
  - even more simplified for constrained resources
- **FinBERT**
  - fine-tuned on financial data
- **LegalBERT**
  - fine-tuned on law texts

There are many options available, which makes choosing one quite challenging. After careful consideration, we decided to use the *bert-large-cased* model without any fine-tuning, as we want to see how these embeddings from standard model perform compared to other text preprocessing techniques. This model is pretrained on a large corpus of English text. We selected the cased version because our input data includes uppercase characters, which can change the meaning of words, such as apple (the fruit) versus Apple (the stock/company).

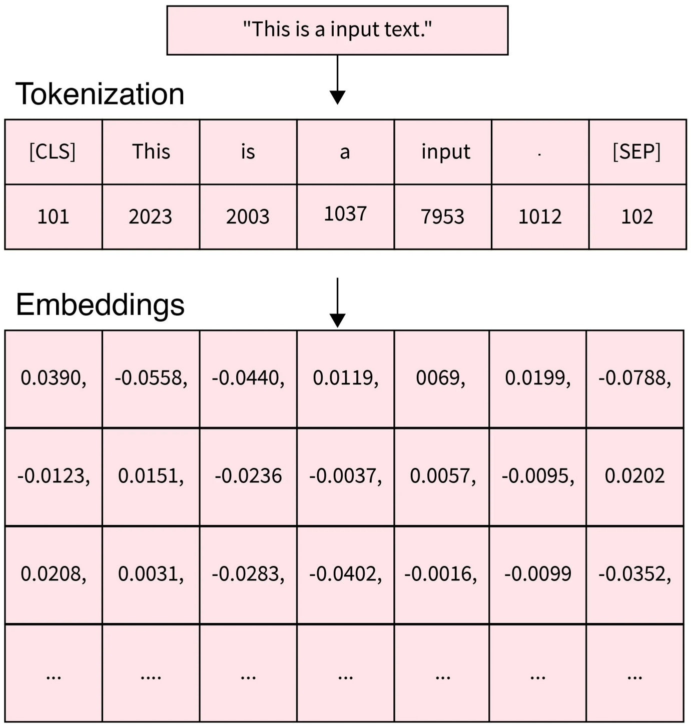
BERT usage

When we used encodings like Word2Vec, Bag-Of-Words, and TF-IDF, we removed words that did not add any value. We were wondering whether there is any preprocessing step required before feeding data into BERT. In general, BERT does not need any preprocessing as it uses the left and right context of words, so it is better to leave the text as it is [5], [6].

The inner architecture and working of the BERT model are complicated. However, here is a brief outline of what happens with our input text:

1. First, our input is tokenized. BERT uses the WordPiece algorithm, which breaks the input down into individual words or even subwords.
2. Then, these tokens are converted into IDs represented by numbers. As a result, our entire sequence is represented as a list of numbers.
3. Each token is converted into an embedding. In the case of the *bert-large* model, this vector has a size of 1024.

Since each token is represented with a vector of size 1024, then sequence is represented by a 2D array. When using more training samples, this data becomes 3D. One might notice that feeding this data into a model (at least into trees used in this project) can be problematic. To address this, we need to reduce the dimensionality. This can be achieved by taking the *mean*, *sum* or *max* over the embeddings, resulting in a single embedding that represents the entire sequence.



[CLS]	This	is	a	input	.	[SEP]
101	2023	2003	1037	7953	1012	102

0.0390,	-0.0558,	-0.0440,	0.0119,	0069,	0.0199,	-0.0788,
-0.0123,	0.0151,	-0.0236	-0.0037,	0.0057,	-0.0095,	0.0202
0.0208,	0.0031,	-0.0283,	-0.0402,	-0.0016,	-0.0099	-0.0352,
...	....	...	...	...	...	...

Figure 2: BERT: tokenization & embeddings [7]

In Figure 2, one can see a tokenized sentence that includes the special tokens *[CLS]* and *[SEP]*. The *[SEP]* token acts as a separator and is added at the end of each sequence. The *[CLS]* token is the first token and stands for ”classification”, as it can be used for classification tasks. Its respective embedding contains useful information obtained from all the embeddings of the tokens. This embedding should be calculated more intelligently to minimize the influence of tokens that do not add value [8]. This can be used instead of taking *mean*, *sum* or *max* over embeddings.

We loaded this model directly from *HuggingFace*, similar to what we did in the last tutorial. However, Google Colab was used for the entire development and we encountered multiple issues:

- we ran out of memory when we fed entire dataset into BERT at once
- CPU was too slow
- when using a TPU, we were unable to download models from HuggingFace due to an error, the notebook crashed for unknown reasons

We tried splitting the data into batches to solve the memory issues, however, that was not enough. We ran out of memory very quickly, even with small batches. The key point was to set *torch.no_grad()* before calling the model. As per my understanding, this disables any gradient computation. We do not need gradients because we are using BERT for inference and do not want to train the model [9], [10]. This, together with batches, fixed the problem. It was still very close to running out of memory though.

```
with torch.no_grad():
    outputs = model(input_ids=input_ids_batch, attention_mask=
        attention_mask_batch)
    all_sentences_embeddings.append(outputs.last_hidden_state)
```

Listing 1: Solution of memory issues

The speed problem was solved by moving all computations to the GPU. After that, we stored the *CLS*, *sum*, *mean*, and *max* embeddings into a *.npy* file for further use without recomputing them.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ")
model.to(device)
```

Listing 2: Switch to GPU

3 Models

We used several models from *Scikit-learn* and *XGBoost*. Before we list the used models, let us briefly explain *XGBoost* since we have not covered this in our course.

XGBoost

XGBoost is an optimized distributed library that implements machine learning algorithms under the Gradient Boosting framework [11]. Gradient boosting is an extension of boosting where the process of additively generating weak models is formalized as a gradient descent algorithm over an objective function [12].

I read that the implementation of gradient-boosted trees in *Scikit-learn* and *XGBoost* is fundamentally the same, but there are significant differences under the hood. *XGBoost*'s implementation is much faster, more memory-efficient, and capable of parallelization [13]. We have read many positive comments about this library, allegedly it has helped to win many competitions. Although there are other libraries that support trees, this one seemed to be the most prominent apart from *Scikit-learn*.

Training process

We trained all models in Google Colab. To achieve better accuracy, we wanted to use libraries for automated hyperparameter tuning (researched *Optuna* a bit). However, training took a significant amount of time, and trying different combinations of parameters would take too long. We also frequently ran out of free quota for more powerful runtimes. Instead, we read some articles on tuning to identify which parameters are worth focusing on. We combined this approach with plotting the training and validation losses (if applicable) and applied techniques like early stopping when the loss did not progress. It is important to note that all models were trained and tuned with *CLS* embeddings. These embeddings were not normalized, as normalization is not required for tree-based models or gradient boosting methods such as *XGBoost* [14]. Later, for the sake of comparison, we will run the same models with *sum*, *mean*, and *max* embeddings to determine which embeddings provide better results.

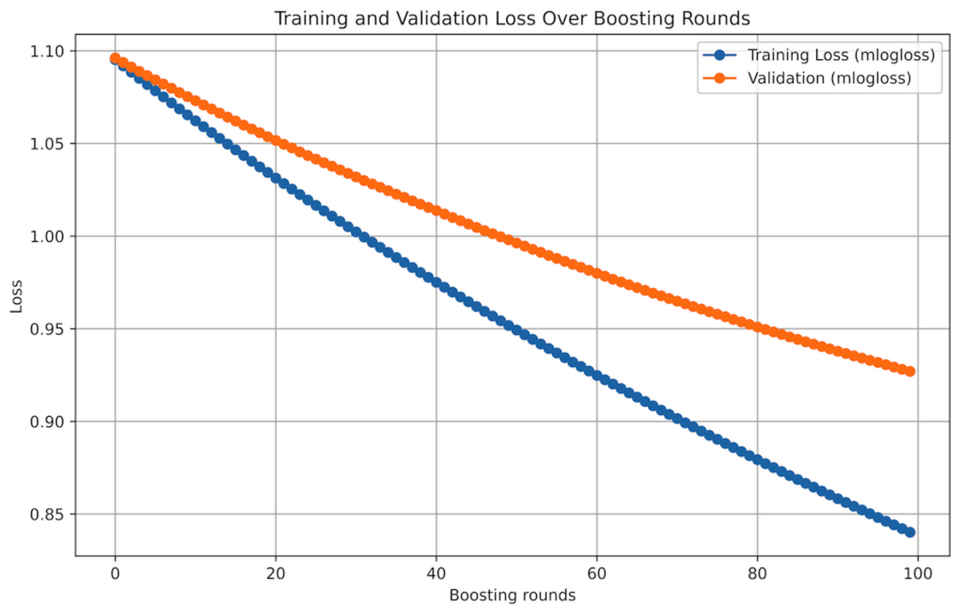


Figure 3: Loss example of overfitting, the goal was to avoid this

Used models

XGBClassifier

Gradient-boosted trees from *XGBoost* library. We were playing with the hyperparameters for an extensive amount of time, but in conclusion, the validation accuracy always hovered around 66%, reaching a maximum of 67%. The returns from hyperparameter tuning were very small.

```python
XGBClassifier(
    n_estimators=1000, #number of boosting rounds
    max_depth=3,
    num_class=3,
    colsample_bytree=0.8, #subsampling 80% of columns for each tree
    learning_rate=0.002,
    eval_metric=["mlogloss"],
    early_stopping_rounds=12
)
```

Listing 3: XGBoost gradient-boosted trees classifier

We tried to look at the confusion matrix, and it looks okay. We expected it to be worse. The main problem is that positive and negative examples are classified as neutral. There is a prevalence of neutral examples in this dataset, so the fix could be to balance the dataset. Interestingly, there does not exist a positive statement that was classified as negative. However, vice versa is not true.

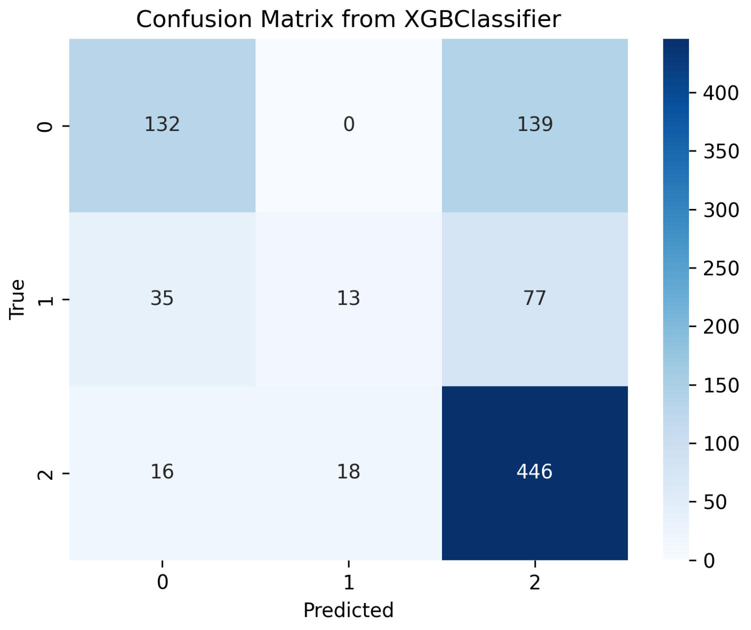


Figure 4: Confusion matrix from XGBClassifier (0=positive, 1=negative, 2=neutral)

We also wanted to try *Scikit-learn*'s implementation of gradient-boosted trees. However, it is much slower. Unlike XGBoost, it does not support parallelization. It took 10 minutes to train it with 100 trees and produced below-average results. Testing different variations to optimize it would take far too long, so we abandoned this model.

XGBRFClassifier
XGBoost's implementation of random forest. Again, we were fiddling around with the hyperparameters, and it hovers around 66% of accuracy on the validation dataset. However, this particular configuration yields 67%. It is intriguing that 6 trees with a maximum depth of 5 yield the best accuracy. Since it is a random forest, we would expect that more and deeper trees would be needed.

```
XGBRFClassifier(
    n_estimators=6,
    max_depth=5,
    num_class=3,
    learning_rate=0.003,
    eval_metric=["mlogloss"],
)
```

Listing 4: XGBoost Random Forest Classifier

RandomForestClassifier
Scikit-learn implementation of random forest. We experimented with various hyperparameters, but the results remained the same. The model achieves slightly above 67%, which is comparable to the *XGBoost*'s counterpart.

```python
RandomForestClassifier(
    n_estimators=40,
    max_depth=6,
    criterion="gini",
    n_jobs=-1 # paralleization
)
```

Listing 5: Scikit-learn Random Forest Classifier

AdaBoost
An AdaBoost [15] begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

On the validation dataset, this model performed worse than all the models above. I managed to achieve about 64.5% accuracy.

```python
AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=4),
    n_estimators=120,
    learning_rate=0.001
)
```

Listing 6: Scikit-learn AdaBoost Classifier

HistGradientBoostingClassifier
Histogram-based gradient boosting classification tree [16]. This algorithm uses histograms to approximate data distributions, growing trees level by level. It is faster and more efficient on large datasets compared to traditional gradient boosting. The implementation is inspired by LightGBM library.

After previous attempts, we thought we had reached the ceiling of 67% with these boosting and bagging methods. Surprisingly, this model achieved 70% on the validation dataset.

```python
HistGradientBoostingClassifier(
    max_iter=800,
    learning_rate=0.005,
    max_depth=2,
)
```

Listing 7: Scikit-learn HistGradientBoostingClassifier

Llama-3.3-70b-versatile
We were very curious to see if any large language model (LLM) without any fine-tuning could outperform traditional methods. To test this, we used the Llama-3.3-70B-versatile model hosted on *Groq*. If we understood the pricing correctly, 30 requests per minute should be free. This was not an issue, as our test data set contains only 876 samples. Basically, we downloaded Groq package and ran 876 API calls to Llama with the prompt included in Listing 8. At the end, we compared the results with the true labels.

```python
def generate_prompt(text):
  return [
      {
        "role": "system",
        "content": "Act as a classifier for financial sentiment
           analysis."
      },
      {
        "role": "user",
        "content": "Classify the following text as a 0=positive, 1=
           negative, or 2=neutral. Output only the single number.\n
           \nText:" + text,
      }
  ]
```

Listing 8: Classification prompt for Llama-3.3-70b-versatile

4 Results

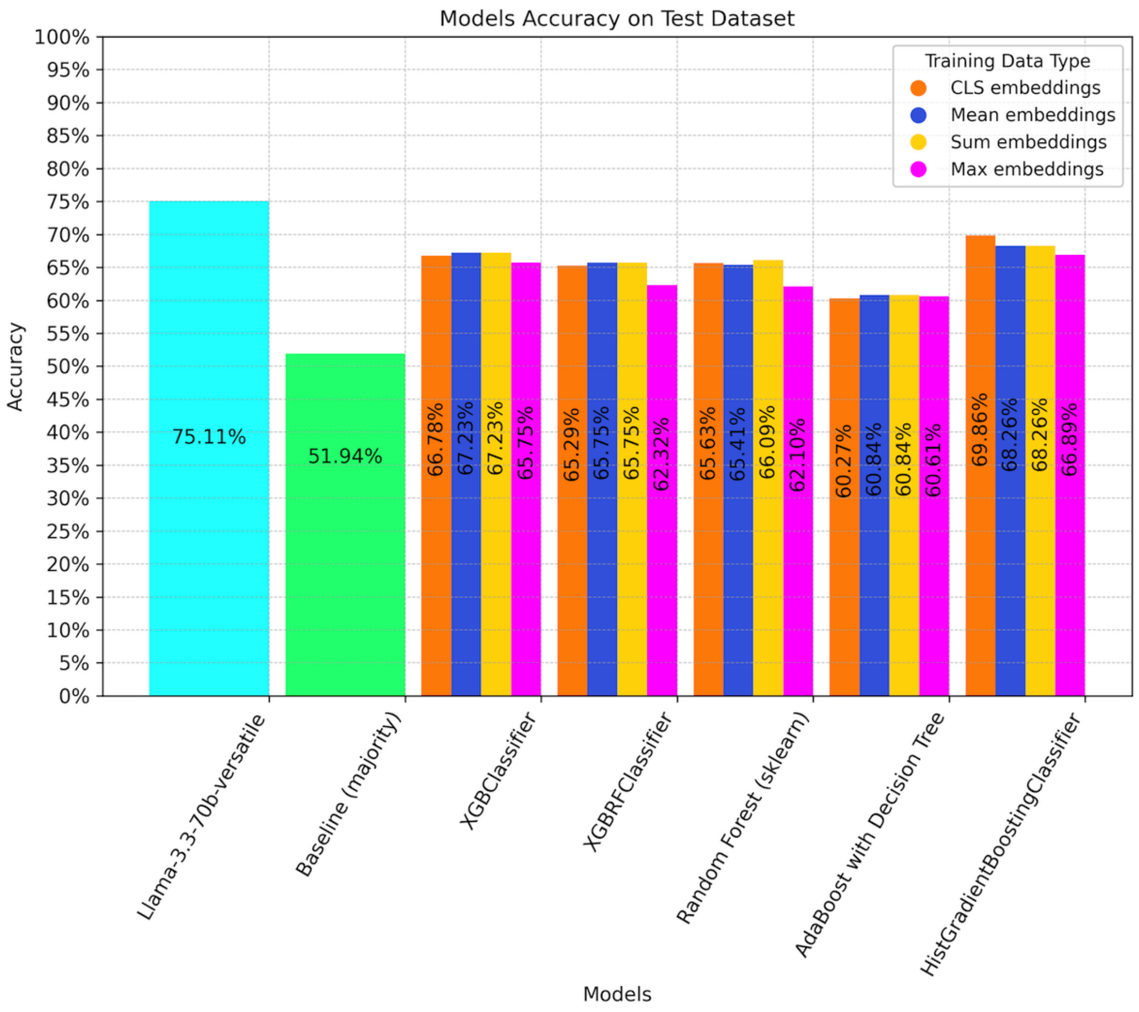


Figure 5: Reached accuracy of models

The baseline achieves 53% accuracy because it relies on majority voting, and the dataset is unbalanced. *LLama* outperformed all models, achieving an accuracy of 75.11%. All traditional models reached an accuracy above 60%, most of them exceeded threshold of 65%. Among the traditional methods, the winner is *HistGradientBoostingClassifier*, which achieved an accuracy of 69.86%, thereby outperforming the highly regarded *XGBoost*'s models.

Regarding embeddings, the results are generally consistent across methods. However, *max* embeddings tend to perform worse than the other three types.

Comparison with other solution

Since we found this dataset on Kaggle, we looked for solutions to compare ours with. The most liked solution on Kaggle appears to be a good candidate for comparison [17]. It is not a completely 1-to-1 comparison, as the author used a 25% testing split, but we do not think it makes noticeable differences.

The author cleverly filtered the important words and on top of that, applied TF-IDF. The figure below shows the accuracies. Among all the models we have in common, it seems that their models perform better. Based on this, we can conclude that BERT is not always the best solution for text preprocessing. In this case, it was outperformed by the simpler TF-IDF method. Among standard methods, SVM is the best performer. The winner is RoBERTa (Robustly Optimized BERT Approach), which was fine-tuned for this classification task, achieving an accuracy of 82.62%.

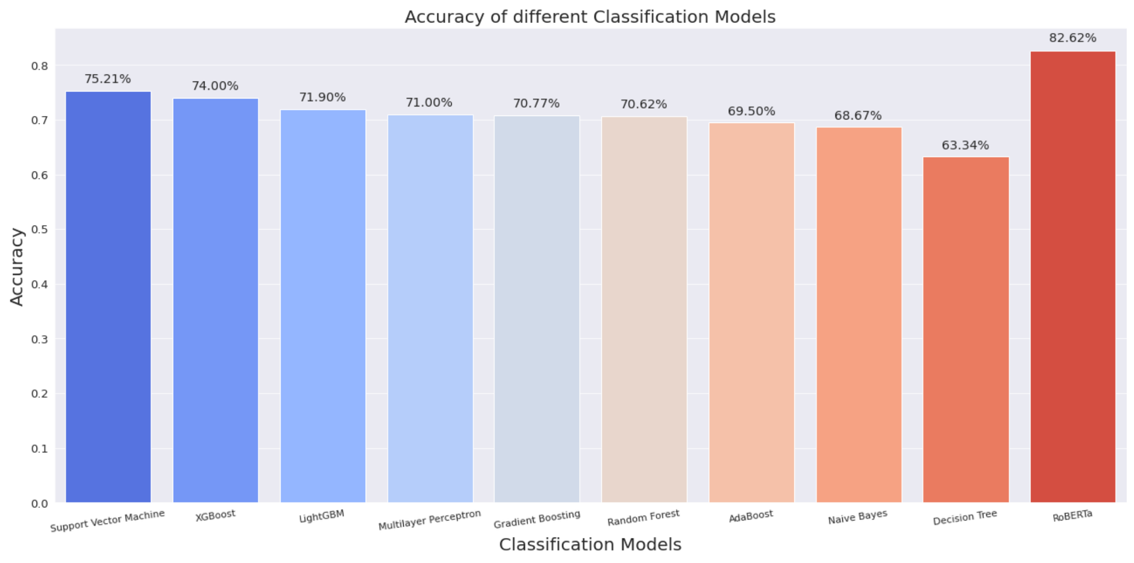


Figure 6: Competitor's results

As SVM leads in terms of traditional approaches in this case, we were curious how it would perform with embeddings, so we tested it. We normalized the CLS embeddings, trained an SVM, and surprisingly achieved 76.94% accuracy on the test dataset. Literally almost 77% without any hyperparameter tuning. There is potential to increase it even further. It surpassed all models in both solutions except RoBERTa, with even Llama being left behind. This is really surprising. We spent hours training and tuning various trees, yet it was outperformed by just a simple SVM.

5 Conclusion

In hindsight, we are not sure whether we would have done anything differently. We definitely learned a lot, so this project probably fulfilled its purpose. The methods used were probably not the best for these types of problem, but we primarily chose them for learning purposes. When we were researching about state-of-the-art in the area of natural language processing (NLP) everything revolves around models with transformer architecture. Selecting the right methods is challenging because the possibilities are literally endless.

The tree-based methods do not perform exceptionally well. This may be due to the large number of features, 1024 seems like a lot. *XGBoost* offers some feature selection methods, but we have not achieved better accuracy with those, only training speed improved.

On the other hand, the performance of SVM + CLS embeddings was surprising. It was better than the competitor's SVM with the TF-IDF approach. The takeaway from this project is that, at the beginning, it is probably better to try multiple models and only fine-tune the best-performing one. This approach can save a lot of time.

The code produced in this project is available in this *GitHub repository*.

References

[1] URL: https://www.kaggle.com/datasets/sbhatti/financial-sentiment-analysis.

[2] URL: https://iq.opengenus.org/bert-base-vs-bert-large/.

[3] URL: https://huggingface.co/google-bert/bert-base-uncased.

[4] URL: https://www.scaler.com/topics/nlp/bert-variants/.

[5] URL: https://datascience.stackexchange.com/questions/113359/why-there-is-no-preprocessing-step-for-training-bert.

[6] URL: https://stackoverflow.com/questions/70649831/does-bert-model-need-text.

[7] URL: https://www.scaler.com/topics/nlp/huggingface-transformers/.

[8] URL: https://stackoverflow.com/questions/62705268/why-bert-transformer-uses-cls-token-for-classification-instead-of-average-over.

[9] URL: https://stackoverflow.com/questions/63785319/pytorch-torch-no-grad-versus-requires-grad-false.

[10] URL: https://stackoverflow.com/questions/77669550/is-there-a-solution-for-this-word-embedding-problem-with-bert.

[11] URL: https://xgboost.readthedocs.io/en/stable/.

[12] URL: https://www.nvidia.com/en-eu/glossary/xgboost/.

[13] URL: https://stats.stackexchange.com/questions/282459/xgboost-vs-python-sklearn-gradient-boosted-trees.

[14] URL: https://datascience.stackexchange.com/questions/60950/is-it-necessary-to-normalize-data-for-xgboost.

[15] URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.

[16] URL: https://scikit-learn.org/1.5/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html#sklearn.ensemble.HistGradientBoostingClassifier.

[17] URL: https://www.kaggle.com/code/supreethrao/bert-s-a-stock-market-guru-86-22-huggingface.