A4: Emotion Classification using LLMs

Alexandre St-Aubin, Jonathan Campana, & Jake Gameroff

Comp 551: Applied Machine Learning

January 13, 2025

Abstract

In this paper we explore emotion classification on the GoEmotions dataset. We implement and compare a Naive Bayes classifier (from scratch), a baseline model (such as XGBoost or Random Forest), and a fine-tuned Large Language Model (LLM) based on Distilled-GPT2. Our Naive Bayes model achieved about 46% accuracy, while XGBoost and Random Forest had accuracies around 56%. More optimistically, the fine-tuned LLM obtained a pique classification accuracy of about 69%. We also investigate the impact of data balancing and examine attention patterns to study the decision-making processes of our models. These results highlight the value of LLMs for emotion detection tasks.

Introduction

In recent years, the rise of large language models (LLMs) has significantly advanced natural language processing (NLP) tasks, including sentiment and emotion analysis. While traditional models like Naive Bayes and advanced machine learning methods like XGBoost have been extensively used due to their simplicity and efficiency, LLMs promise deeper contextual understanding of text [1]. This report aims to compare these approaches in the context of classifying emotions in Reddit comments, a challenging task due to the diversity, informality, and ambiguity of user-generated content. By evaluating their performance, we can gain insights into their relative strengths, trade-offs, and applicability to real-world social media analysis tasks.

We began by building two baseline models, XGBoost and Random Forest, using TF-IDF vectorizers. While these models achieved reasonable overall accuracy, they struggled with rare labels. The XGBoost model had a weighted average precision, recall, and F1-score of 0.570, 0.566, and 0.507, respectively, with an average accuracy of 56.56%, highlighting its limitations in handling infrequent classes. Similarly, the Random Forest model performed comparably, with a weighted average precision, recall, and F1-score of 0.534, 0.559, and 0.510, and an average accuracy of 55.95%.

To explore alternative representations, we implemented Word2Vec embeddings for both models. However, the performance was significantly worse, with the XGBoost model achieving an accuracy of 39% and the Random Forest model reaching only 40%. Additionally, we experimented with a Naive Bayes model, which, after optimizing the smoothing parameter, achieved a training accuracy of 73.08% and a test accuracy of 46.43%, slightly lower than the baseline models. Finally, we fine-tuned GPT-2, a decoder-based transformer model, which achieved a test accuracy of 68.84%, significantly outperforming all other approaches.

Related Work

Data Processing

The data set was imported from parquet files to pandas data frames, one for each of the training, validation, and test data sets. The data points that had multiple labels were removed from the data frames, which left us will approximately 85% of the data points from the original data set.

For the naive Bayes method, we designed a data pre-processing pipeline using the bag-of-words (BOW) representation. With BOW we represent text data as vectors of word counts based on the dataset's unique vocabulary; we used scikit-learn's **CountVectorizer** to do this. For the baseline method, XGBoost in our assignment, we used TF-IDF-Vectorizer to transform the text data to numerical features. This transformed each text to normalized vectors with norm 1.

1 XGBoost

The eXtreme Gradient Boosting is the machine learning model that we used as our baseline model for the assignment. XGBoost is a gradient boosted decision tree model that has seen great success in classification and regression, due to its speed and efficiency, and regularization to combat overfitting (a common problem for decision trees). Given the preprocessed data with vectorized text input and the single label outputs, an XGBoost model was trained from the xgboost library using multiclass log loss, with the sklearn implementation. We used 100 boosting rounds, given that the test loss would decrease at each iteration, and would only start to plateau around iteration 90. We also used the **eval_set** input for the XGBoost model with the validation dataset from the GoEmotion dataset [2], to evaluate the performance of the model as it is learning. We used the default **max_depth = 3**, **learning_rate = 0.1**, and **reg_lambda = 1**. The model was not multi-threaded, therefore, we also kept the respective arguments as default. The remainder of arguments used is: **silent=True**, **booster='gbtree'**, **n_jobs=1**, **nthread=None**, **gamma=0**, **min_child_weight=1**, **max_delta_step=0**, **subsample=1**, **colsample_bytree=1**, **colsample_bylevel=1**, **reg_alpha=0**, **scale_pos_weight=1**, **base_score=0.5**, **random_state=0**, **seed=None**, **missing=None** [3]. Once the model was trained, we used the predict function on the normalized test data set and compared it with the true labels to evaluate the performance of the XGBoost model on unseen data.

Since the data is very skewed, we have some labels that show up a disproportionate amount of times. For example, in the dataset, label 27 makes up around one third of the entire dataset. Given the number of data points with this output, the model was a lot better at predicting data that had label 27 given how many examples it could learn from. Some classes have very few data points for our model to properly learn from, and as a result have poor accuracy when testing with the test data set. For example, there were only 39 data points with label 16, compared to 12823 data points with label 27. Thus, it is not a surprise that at test time, we obtained an accuracy of 51% accuracy on the 1606 data points with label 27, while getting 0% accuracy for the 2 data points with label 16. Overall, XGBoost achieved a score of 0.570 in the weighted average precision, 0.566 in average weighted recall, and an average weighted f1-score of 0.507, as seen in Figure 10. The overall accuracy on the test set is **0.565577**.

2 Random Forest

Random Forest is an ensemble learning method primarily used for classification and regression tasks. It operates by constructing multiple decision trees during training and outputs the class that is the majority vote for classification or the mean prediction for regression. Due to its ability to handle imbalanced data and mitigate overfitting, Random Forest served as a comparative model to our baseline XGBoost implementation.

The preprocessed data, including vectorized text inputs and corresponding single-label outputs, was used to train a Random Forest classifier from the **scikit-learn** library. We employed the **RandomForestClassifier** with the default **Gini** impurity criterion for node splitting. A total of 100 trees, **n_estimators=100**, were grown in the forest, balancing computational efficiency and model performance. Other default hyper parameters included **max_depth=None** and **bootstrap=True** [4].

Unlike XGBoost, Random Forest models rely on bagging, which trains each decision tree on a random subset of the data and features, reducing variance and improving robustness. The **oob_score** argument was enabled to evaluate the model's performance on out-of-bag samples, providing an internal estimate of generalization error. We however see that the model had very similar learning and predictive power, by also having good accuracy on the data points with labels that were common in the training cases, but poor accuracy with labels that it did not have much data to train on. The random forest model achieved a weighted precision of 0.534, recall of 0.559 and f1-score of 0.510, as seen in Figure 11. The overall accuracy on the test set is **0.559477**. Compared to XGBoost, precision and recall are both slightly lower for Random Forest, while the f1-score is almost identical, and accuracy is half a percentage point below.

3 Word2Vec Implementation

To compare the performance of XGBoost and Random Forest models using different data preprocessing techniques, Word2Vec was implemented as an alternative to TF-IDF for text representation. Word2Vec embeddings were generated by tokenizing the text data into individual words and training a Word2Vec model on the training dataset using the skip-gram architecture. Each document's vector representation was obtained by averaging the Word2Vec embeddings of its constituent words. This approach captures the semantic relationships between words, as Word2Vec is designed to learn vector representations where similar words are located closer together in the embedding space.

Despite its potential, the models trained with Word2Vec embeddings showed significantly lower performance compared to those trained with TF-IDF features. For instance, the XGBoost model achieved an overall accuracy of 39% with Word2Vec 12, compared to 57% with TF-IDF. For the Random Forest model, the model had an overall accuracy of 40% 13, which is still significantly lower than the 56% accuracy achieved with TF-IDF. It is interesting to note that with Word2Vec, the Random Forest model has better accuracy. Precision, recall, and F1-scores were consistently lower across most classes, particularly for infrequent categories. The under performance of Word2Vec in this context can be attributed to its inability to emphasize important contextual or frequency-based features that TF-IDF captures effectively, as well as the loss of textual structure when averaging word embeddings. These results highlight the importance of feature representation and suggest that Word2Vec, while useful for capturing semantics, may require more advanced models or fine-tuning to achieve competitive results in traditional classification tasks. This takes us to the next section on LLMs.

4 Fine-tuned LLM

GPT-2 is a decoder transformer, which means it uses the last token of the input sequence to predict the next token. This is because the last token holds all the information needed for the prediction. With this idea, we can repurpose the model to make predictions for classification tasks instead of generating text. For our implementation, we choose to fine-tune **Distilled-GPT2**, which is a smaller, faster and lighter version of GPT-2 with 82 million parameters, as opposed to 124 million.

When evaluating the pretrained LLM on the test set without fine-tuning, we observe an accuracy of approximately 3%. This result is expected because, although the pretrained LLM has learned general word representations, it has no knowledge about the classification task at hand, so it simply "guesses". For instance, the model cannot infer that the label **2** corresponds to the emotion **admiration**. As a result, we get an accuracy that corresponds almost exactly to $1/28 = 0.0357$, the probability of guessing correctly the class. This is where fine-tuning comes in play. By adjusting the weights of the entire model, especially those in the output layer, which maps word embeddings to emotion labels, we enable the model to adapt to the specific requirements of our task.

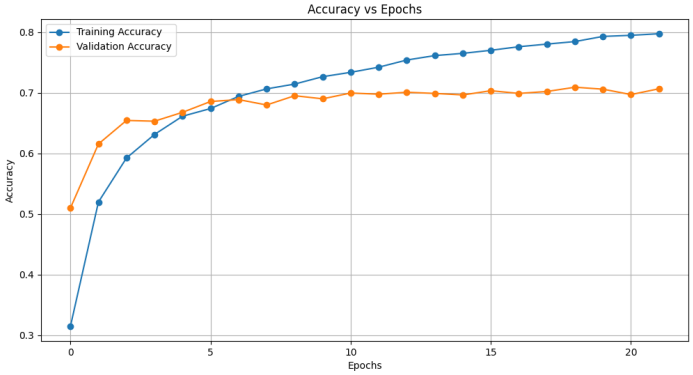
4.1 Training

Fine-tuning Distilled-GPT2 involves adapting the pre-trained language model using the GPT2ForSequenceClassification class from the Hugging Face Transformers library. This class modifies the original GPT-2 architecture, allowing it to perform classification tasks. Instead of predicting the next token in a sequence, as in a standard language model, this class performs classification based on the representation of the last token in the input sequence. This modification makes it suitable for tasks like sequence classification, where the final token's embedding encapsulates the features necessary for predicting the class label. The architecture of the model is shown in Figure 6. We optimize training using AdamW with `lr = 1e-5` and `eps = 1e-8`. This improves generalization by mitigating overfitting through weight decay, as in [5]. To ensure smooth and stable convergence during training, we employ a learning rate scheduler. Specifically, we use a linear scheduler that decreases the learning rate linearly from the initial value after a warm-up period, as illustrated in Figure 5a. The model's accuracy and loss as a function of the training epoch is presented in Figure 1. To prevent overfitting and unnecessary computation, we implement early stopping with a patience of 3 epochs, terminating training at epoch 21. The highest accuracy achieved on the test set was **0.6877**.

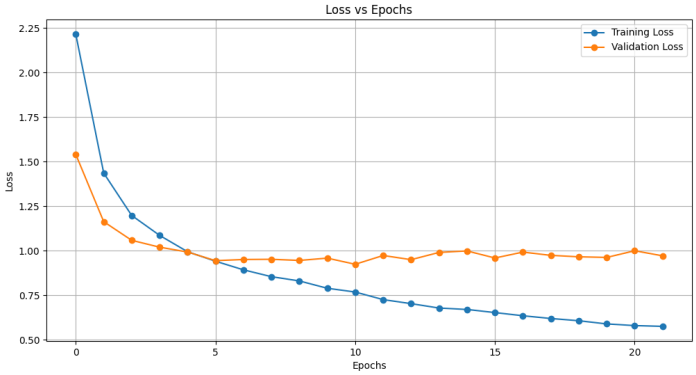
We explore the use of a cosine learning rate scheduler with hard restarts (Figure 5b) and report the results in Figure 7. Additionally, we test a constant learning rate scheduler, shown in Figure 5c and Figure 8. Our findings indicate that the cosine scheduler with hard restarts underperforms compared to the linear scheduler. In contrast, the constant scheduler shows faster learning, achieving comparable accuracy on both the training and validation sets. It achieves a test accuracy of **0.68836** after only 12 epochs, outperforming the cosine scheduler in both efficiency and effectiveness. Overall, all of our fine-tuned LLM Models significantly outperform both baselines in terms of accuracy. This is expected, as the LLM model we use is much more expressive than XGBoost and Random Forest.

4.2 Attention

In this analysis, we explore how attention is distributed between individual words in a sentence and the "class token" (CLS token) in a transformer model. The CLS token, which is added at the start of every sentence, gathers overall information about the sentence and is used for tasks like classifying emotions. By examining the attention patterns from one transformer block and a specific attention head, we can identify which words influence the CLS token the most. This helps us understand



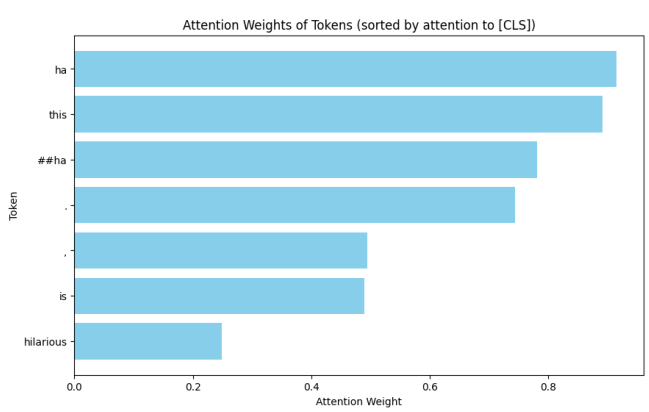
(a) Accuracy vs. Epochs.



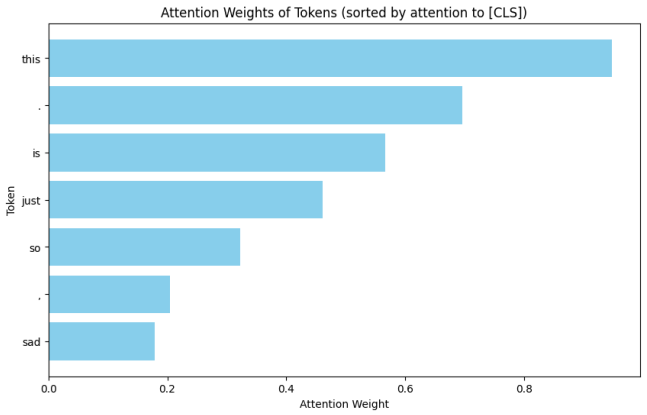
(b) Loss vs. Epochs.

Figure 1: Training of the LLM with linear learning rate scheduler.

how the model focuses on certain words to make predictions, especially for sentences where the model gets the classification right or wrong. This insight can reveal how well the model captures important parts of the sentence to decide its emotion. We examine the comments "Haha, this is hilarious." (correctly classified as *amusement*), see fig. 2a, and "This is just sad. So, so sad." (incorrectly classified as *desire*, correct class is *sadness*), see fig. 2b. We notice that for the case of the correctly



(a) "Haha, this is hilarious."



(b) "This is just sad. So, so sad."

Figure 2: Tokens sorted by attention weights associated with [CLS] for different inputs.

classified input, the token that has the most attention with [CLS] is "ha". This corresponds to a word heavily associated with the *amusement* emotion [2]. On the other hand, the incorrectly classified comment gives high attention to the word "this", which gives no information about any emotion.

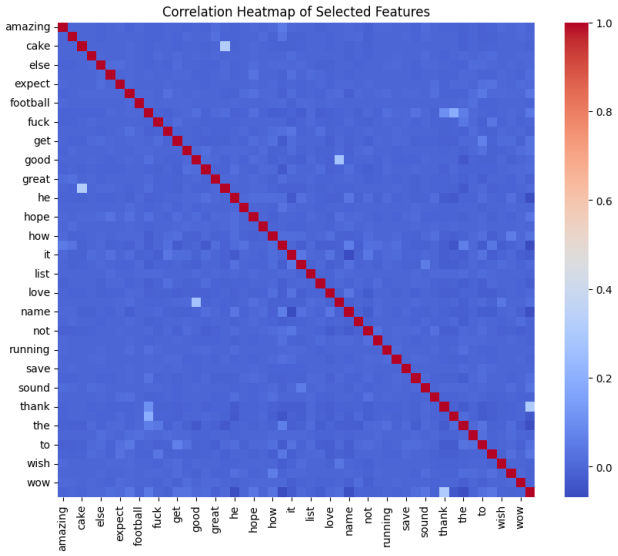
5 Naive Bayes

Our Naive Bayes model is a probabilistic classifier based on Bayes' theorem. We assume the features $x_1, x_2, \ldots, x_D$ are conditionally independent given the class $y = c$. The posterior probability for each class is $P(y = c|\mathbf{x}) \propto P(y = c) \prod_{d=1}^{D} P(x_d|y = c)$, where $P(y = c)$ is the ratio of the number of samples with $y = c$ to the total number of samples, and the likelihood $P(x_d|y = c)$ is calculated with Laplace smoothing:

$$P(x_d|y = c) = \frac{\text{count of feature } x_d \text{ in class } c + \alpha}{\text{total count of all features in class } c + \alpha \cdot D}.$$

We predict the class $y = c$ that maximizes the log posterior probability: $\hat{y} = \arg\max_c \left(\log P(y = c) + \sum_{d=1}^{D} \log P(x_d|y = c)\right)$.

The heatmap in Figure 3a shows the pairwise correlations between the top 50 features (in terms of frequency) in the dataset. We first observe that most off-diagonal values are close to zero, indicating low correlation between features. This supports our assumption that the features are conditionally independent given the class.



(a) Correlation Heatmap of Selected Features

α	Train Accuracy	Test Accuracy
0.000000	0.8530	0.3625
0.001000	0.8515	0.4052
0.050000	0.8297	0.4416
0.100000	0.8077	0.4523
0.150000	0.7819	0.4612
0.200000	0.7536	0.4636
0.239373	0.7308	0.4643
0.300000	0.7001	0.4617
0.500000	0.6087	0.4479
1.000000	0.4911	0.4214

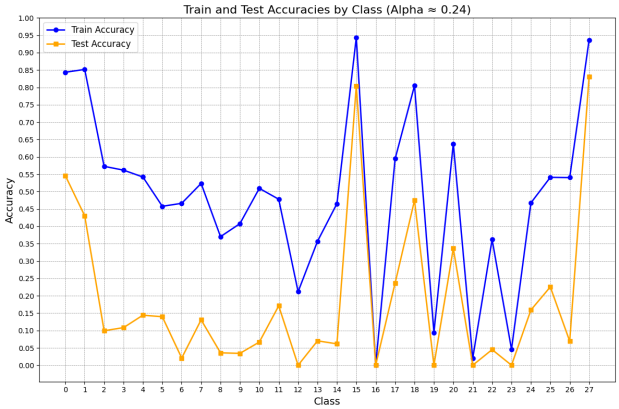
(b) Train and Test Accuracies for Different Alpha Values ($\alpha = 0.239373$ is optimal)

Nonetheless, there are some light-colored off-diagonal cells, suggesting some degree of dependency. For example, the words "love" and "kind" appear to have a moderate correlation. It is therefore possible that these words frequently co-occur in sentences associated with specific emotions, such as "happiness" or "gratitude", which may affect performance.

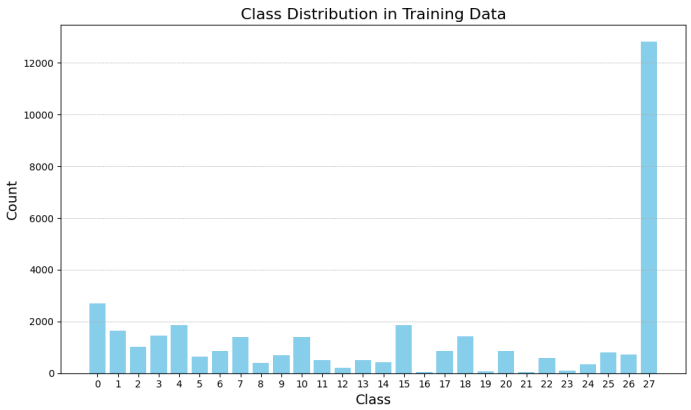
A search over around 2000 smoothing rates yielded an optimal smoothing parameter of $\alpha \approx 0.24$. With this optimum, the model's train accuracy was around 73.08% and its test accuracy was about 46.43%. Table 3b summarizes the train and test accuracies for different values of α (see also Figure 9 for a visual plot).

We see that as α increases, the train accuracy falls and the test accuracy initially increases to its optimum around $\alpha \approx 0.24$. This is likely due to a reduction in overfitting. When the model has no smoothing (i.e. $\alpha = 0$), the model overfits maximally, achieving the highest train accuracy but the lowest test accuracy. After its optimum, the test accuracy decreases, likely because the model becomes too biased and starts underfitting.

Compared with our other assignments, an optimal test accuracy of 46% seems quite low! However, an accuracy of 46% is significantly higher than the chance level of 3.57% for this 28-class task. Nonetheless, our model does make a very powerful assumption: namely, that the features (i.e. the words) are conditionally independent given the class. Consequently, our model may ignore contextual relations between words, and this may limit its accuracy. Another pertinent issue is the serious class imbalance of the GoEmotions dataset, as illustrated by Figure 4b. This may affect model accuracy because Naive Bayes assumes all classes are equally important, but this is not the case here. Further, some classes have very few examples, and so the model struggles to fit to these classes. Likewise, the model overfits to the overrepresented classes.



(a) Train and Test Accuracies by Class (Alpha ≈ 0.24).



(b) Class Distribution in Training Data.

Figure 4: Comparison of train/test accuracies by class and the class distribution in the training data.

This observation is supported by Figure 4a. We see that the overrepresented classes, such as 27, 15, and 0 (pertaining to *neutral*, *gratitude*, and *admiration* respectively) have training accuracies around 94%, 95%, and 85% respectively, which

implies that the model is overfitting these classes. Underrepresented classes such as 16, 19, or 23 (pertaining to *grief*, *nervousness*, and *relief* respectively) all have train accuracies under 10%, which shows that the model is underfitting classes.

Initially, we hypothesized that these low test results could be a consequence of the significant imbalance in the GoEmotions data set. To test this, we trained our model on a balanced subset of the data. With optimal parameters, the Naive Bayes model achieved a high train accuracy of 96.25%. However, the test accuracy was much lower at 25%. This suggests that while balancing the dataset helped the model learn all classes better during training, it may have introduced overfitting. This reduction may have also stopped the model from capturing certain meaningful and important patterns.

We note that in a BOW representation, common words may overshadow more informative ones; using TF-IDF helps by weighting emotionally distinctive words more effectively. With optimal parameters, we obtained a train accuracy of about 48.69% and a test accuracy of around 39.75%. However, the TF-IDF representation may not have outperformed BOW because it reduces the influence of frequent words that could still be important for predicting emotions in this dataset.

Conclusion

In this assignment, we evaluated a range of models for emotion classification in Reddit comments, highlighting the trade-offs between traditional and modern approaches. XGBoost and Random Forest performed similarly using TF-IDF, with accuracies of 56.56% and 55.95%, respectively, but struggled with infrequent labels. Word2Vec embeddings underperformed significantly for both models, compared to TF-IDF. The Naive Bayes model, though straightforward, achieved slightly lower accuracy (46.43%) compared to our other baseline methods. The fine-tuned GPT-2 achieved the highest accuracy (around 69%) by leveraging pre-trained language representations, excelling in capturing complex patterns in text. Overall, while simpler models are efficient, LLMs provide a significant edge for nuanced text data, albeit with greater computational costs.

Creativity

1. Conducted a **correlation analysis** of the top 50 most frequent features to assess their relationships and plotted the results as a heatmap (Figure 3a). This highlights how features interact. It also supports the assumption of conditional independence given the class label. We also performed a **class distribution analysis** to examine the balance of class labels in the dataset and plotted the results (Figure 4b). This helped us identify potential imbalances in the data that might impact model performance. More information about these two plots is given in Section 5 about Naive Bayes.

2. Conducted a **TF-IDF analysis** to emphasize distinctive terms and reduce the impact of common ones. We systematically tuned the smoothing parameter α. Results and further discussion are presented in Section 5 about Naive Bayes.

3. Trained the Naive Bayes model on a balanced subset of the data by creating a dataset with equal representation of all classes to address potential imbalances. This approach aimed to improve model fairness and performance across underrepresented classes. Please see Section 5 about Naive Bayes for more details.

4. Performance comparison of different learning rate schedulers for fine-tuning of the LLM in section 4.1.

5. Created several Baseline models to compare performance between more similar models (tree based models), and also comparing them to the other models in the assignment, mainly Naive Bayes and the LLM models.

6. Used Word2Vec on the baseline models to test the performance of different embedding methods.

Contributions

- Alex: Everything in the fine-tuned LLM section (code + report), conclusion.
- Jake: Everything about the Naive Bayes model (code + report). Wrote the abstract.
- Jonathan: Everything for the baseline models (code + report), introduction

References

[1] A. Strasser, "Pitfalls (and advantages) of sophisticated large language models," in *Ethics in Online AI-based Systems*, pp. 195–210, Elsevier, 2024.

[2] D. Demszky, D. Movshovitz-Attias, J. Ko, A. Cowen, G. Nemade, and S. Ravi, "Goemotions: A dataset of fine-grained emotions," 2020.

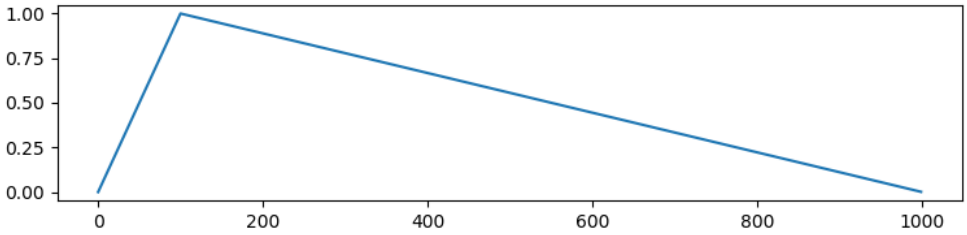
[3] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

[4] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

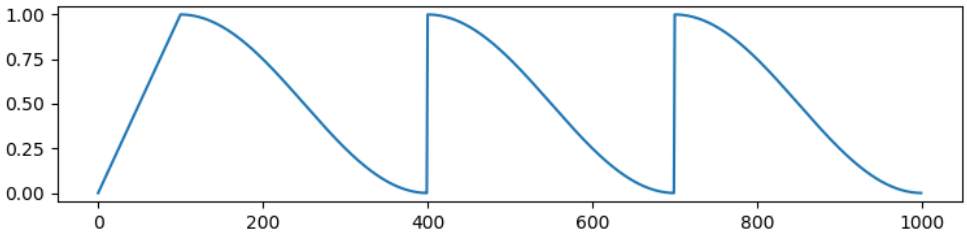
[5] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

Appendices

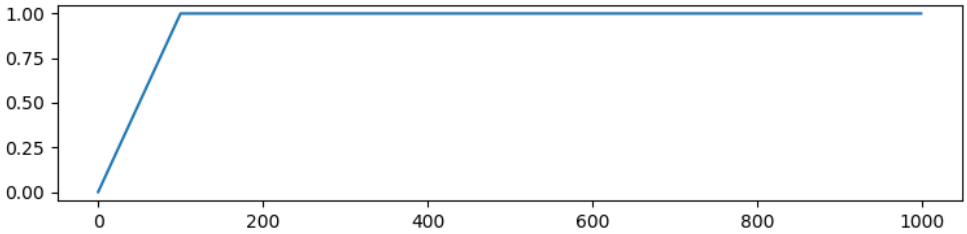
A Appendix A



(a) Linear scheduler with initial warmup period.



(b) Cosine scheduler with initial warmup period and hard resets.



(c) Constant scheduler with initial warmup period.

Figure 5: Different learning rate schedulers.

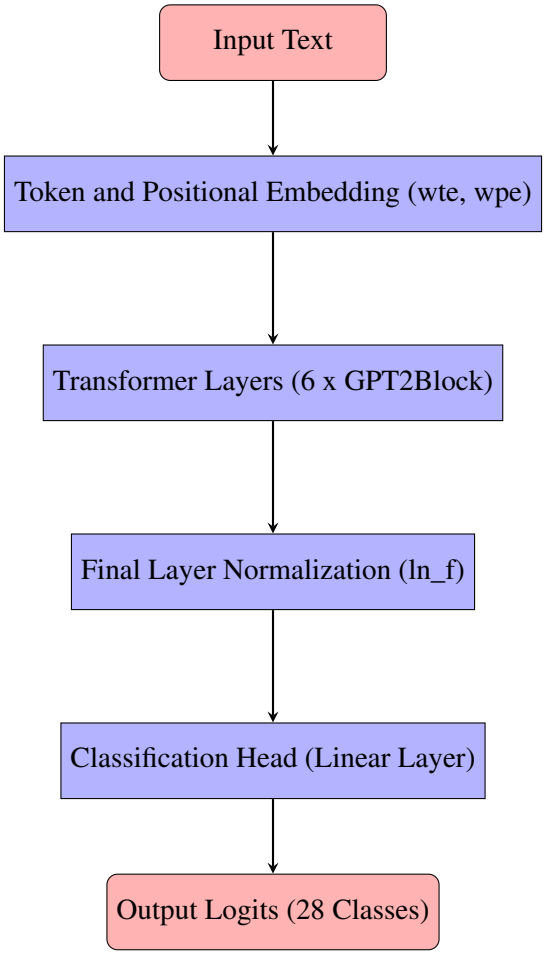
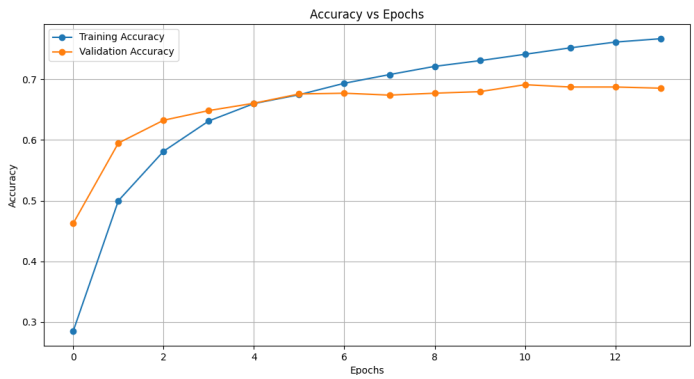
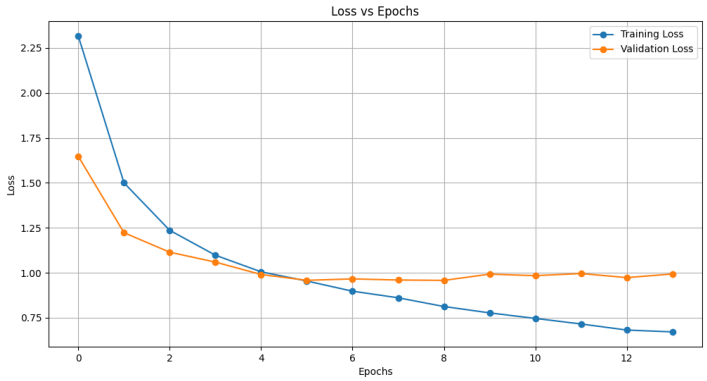


Figure 6: GPT-2 for Sequence Classification Architecture

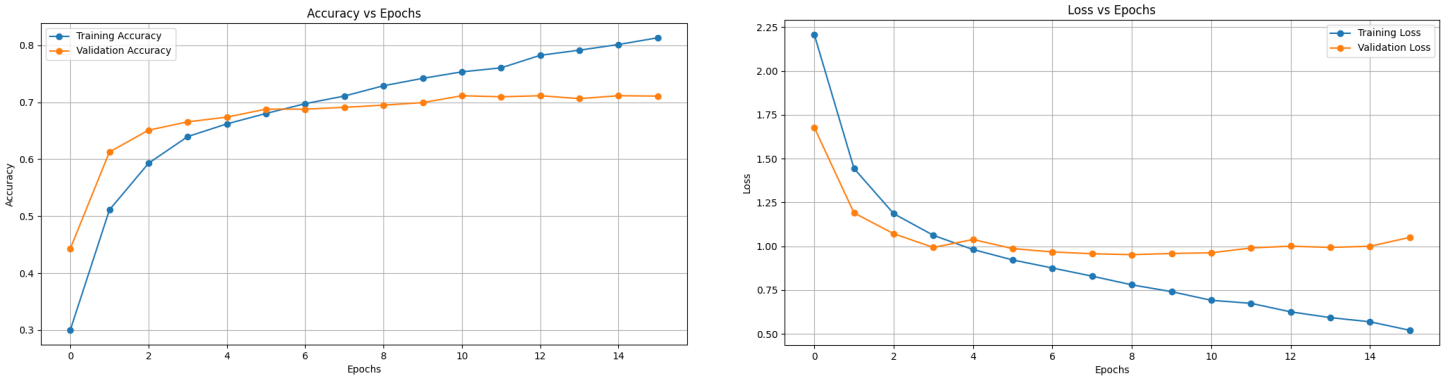


(a) Accuracy vs. Epochs.



(b) Loss vs. Epochs.

Figure 7: Training of the LLM with cosine learning rate scheduler with hard resets.



(a) Accuracy vs. Epochs.

(b) Loss vs. Epochs.

Figure 8: Training of the LLM with constant learning rate scheduler.

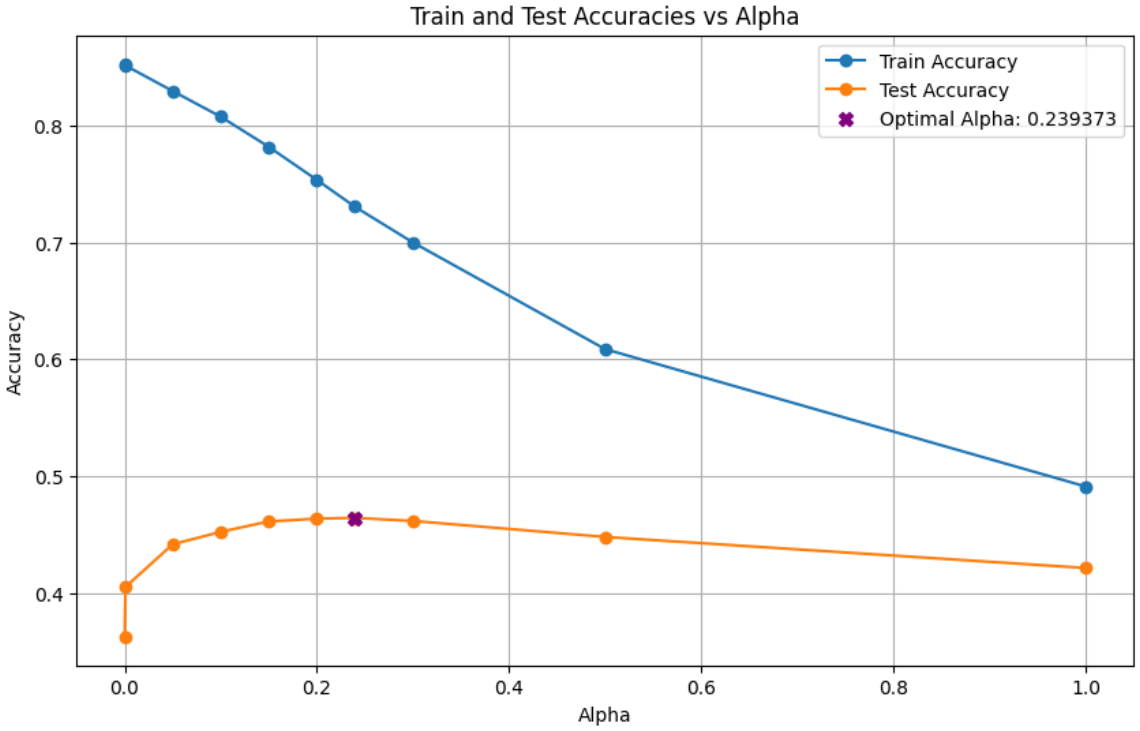


Figure 9: Train and test accuracies as a function of α. The optimal alpha value, $\alpha = 0.239373$, is marked.

	precision	recall	f1-score	support
0	0.628571	0.568966	0.597285	348.000000
1	0.722488	0.811828	0.764557	186.000000
2	0.511628	0.335878	0.405530	131.000000
3	0.500000	0.097938	0.163793	194.000000
4	0.507937	0.135593	0.214047	236.000000
5	0.571429	0.139535	0.224299	86.000000
6	0.444444	0.123711	0.193548	97.000000
7	0.714286	0.056818	0.105263	176.000000
8	0.545455	0.214286	0.307692	56.000000
9	0.529412	0.102273	0.171429	88.000000
10	0.320000	0.041026	0.072727	195.000000
11	0.510204	0.328947	0.400000	76.000000
12	0.500000	0.304348	0.378378	23.000000
13	0.517241	0.263158	0.348837	57.000000
14	0.725490	0.569231	0.637931	65.000000
15	0.923077	0.876923	0.899408	260.000000
16	0.000000	0.000000	0.000000	2.000000
17	0.564706	0.516129	0.539326	93.000000
18	0.733333	0.825000	0.776471	160.000000
19	0.272727	0.250000	0.260870	12.000000
20	0.674699	0.523364	0.589474	107.000000
21	0.500000	0.142857	0.222222	7.000000
22	0.600000	0.067416	0.121212	89.000000
23	0.000000	0.000000	0.000000	7.000000
24	0.557377	0.772727	0.647619	44.000000
25	0.696429	0.382353	0.493671	102.000000
26	0.428571	0.344828	0.382166	87.000000
27	0.509091	0.889166	0.647472	1606.000000
accuracy	0.565577	0.565577	0.565577	0.565577
macro avg	0.525307	0.345868	0.377330	4590.000000
weighted avg	0.569521	0.565577	0.507134	4590.000000

Figure 10: Classification Report for XGBoost

	precision	recall	f1-score	support
0	0.550505	0.626437	0.586022	348.000000
1	0.743961	0.827957	0.783715	186.000000
2	0.443478	0.389313	0.414634	131.000000
3	0.390625	0.128866	0.193798	194.000000
4	0.355556	0.135593	0.196319	236.000000
5	0.380952	0.186047	0.250000	86.000000
6	0.440000	0.113402	0.180328	97.000000
7	0.361111	0.073864	0.122642	176.000000
8	0.541667	0.232143	0.325000	56.000000
9	0.250000	0.056818	0.092593	88.000000
10	0.375000	0.046154	0.082192	195.000000
11	0.620000	0.407895	0.492063	76.000000
12	0.533333	0.347826	0.421053	23.000000
13	0.472222	0.298246	0.365591	57.000000
14	0.709091	0.600000	0.650000	65.000000
15	0.921569	0.903846	0.912621	260.000000
16	0.000000	0.000000	0.000000	2.000000
17	0.540816	0.569892	0.554974	93.000000
18	0.729730	0.843750	0.782609	160.000000
19	0.428571	0.250000	0.315789	12.000000
20	0.677778	0.570093	0.619289	107.000000
21	0.500000	0.142857	0.222222	7.000000
22	0.600000	0.101124	0.173077	89.000000
23	0.000000	0.000000	0.000000	7.000000
24	0.578947	0.750000	0.653465	44.000000
25	0.682540	0.421569	0.521212	102.000000
26	0.417722	0.379310	0.397590	87.000000
27	0.519685	0.821918	0.636758	1606.000000
accuracy	0.559477	0.559477	0.559477	0.559477
macro avg	0.491602	0.365176	0.390913	4590.000000
weighted avg	0.534175	0.559477	0.510096	4590.000000

Figure 11: Classification Report for Random Forest

XGBoost with Word2Vec Test Classification Report:

	precision	recall	f1-score	support
0	0.35	0.25	0.29	348
1	0.19	0.03	0.06	186
2	0.00	0.00	0.00	131
3	0.00	0.00	0.00	194
4	0.14	0.03	0.04	236
5	0.22	0.05	0.08	86
6	0.22	0.08	0.12	97
7	0.32	0.18	0.23	176
8	0.33	0.04	0.06	56
9	0.00	0.00	0.00	88
10	0.24	0.05	0.08	195
11	0.00	0.00	0.00	76
12	0.00	0.00	0.00	23
13	0.33	0.05	0.09	57
14	0.17	0.02	0.03	65
15	0.54	0.60	0.57	260
16	0.00	0.00	0.00	2
17	0.19	0.03	0.06	93
18	0.52	0.42	0.46	160
19	0.00	0.00	0.00	12
20	0.43	0.03	0.05	107
21	0.00	0.00	0.00	7
22	0.00	0.00	0.00	89
23	0.00	0.00	0.00	7
24	0.27	0.07	0.11	44
25	0.22	0.04	0.07	102
26	0.00	0.00	0.00	87
27	0.40	0.88	0.55	1606
accuracy			0.39	4590
macro avg	0.18	0.10	0.11	4590
weighted avg	0.29	0.39	0.29	4590

Figure 12: Classification Report for XGBoost with Word2Vec

Random Forest with Word2Vec Test Classification Report:

	precision	recall	f1-score	support
0	0.40	0.22	0.29	348
1	0.19	0.02	0.03	186
2	0.09	0.01	0.01	131
3	0.00	0.00	0.00	194
4	0.23	0.01	0.02	236
5	0.50	0.02	0.04	86
6	0.33	0.01	0.02	97
7	0.42	0.09	0.14	176
8	0.00	0.00	0.00	56
9	0.00	0.00	0.00	88
10	0.27	0.02	0.03	195
11	0.33	0.01	0.03	76
12	0.00	0.00	0.00	23
13	0.25	0.04	0.06	57
14	0.33	0.02	0.03	65
15	0.55	0.62	0.58	260
16	0.00	0.00	0.00	2
17	0.25	0.01	0.02	93
18	0.59	0.29	0.39	160
19	0.00	0.00	0.00	12
20	1.00	0.03	0.05	107
21	0.00	0.00	0.00	7
22	0.00	0.00	0.00	89
23	0.00	0.00	0.00	7
24	0.50	0.02	0.04	44
25	0.00	0.00	0.00	102
26	0.50	0.01	0.02	87
27	0.39	0.94	0.55	1606
accuracy			0.40	4590
macro avg	0.25	0.09	0.08	4590
weighted avg	0.34	0.40	0.28	4590

Figure 13: Classification Report for Random Forest with Word2Vec