Project Final Report of Emotion Classification using Machine Learning on a Multi-Class Dataset

Members:

- Carlos Rabat: czr0067@auburn.edu, *Project Coordinator*
- Samarth Kumar: szk0187@auburn.edu
- Jeronime Houndonougbo: jah0268@auburn.edu

Problem Description:

Our project is a multiclass classification problem that also falls under Natural Language Processing. Specifically, the project involves categorizing a dataset of Twitter posts by their emotional sentiments. The dataset uses integer values ranging from 0 through 5 to represent a certain emotion in the following manner: sadness (0), joy (1), love (2), anger (3), fear (4), surprise (5). To simplify the process, we strictly used text data rather than incorporating image or video posts. Using several machine learning models, we hope to develop a model that can accurately classify a series of online posts by their emotional sentiments.

1. Data Set and Preprocessing

1.1 Data Set: Training and Testing Data

The dataset contains a total of 416,809 samples, so they were partitioned into 333,447 training samples and 83,362 testing samples, meaning that 80% of the data was allocated towards training the model while 20% of the data was allocated for testing. Since the original dataset only consists of two columns (text and label), we used TD-IDF and Bag of Words (BoW) approaches to extract 1000 features from the raw data. These features represent the 1000 most common words.

1.2 Metrics

To evaluate our models, we prioritized accuracy when measuring the overall performance of each model. We additionally used precision, recall, and f1-score. Accuracy measures the overall correctness of the model. Precision measures the proportion of correct predictions of a given class. Recall measures how often the given class occurs, by accounting for falsely classified samples. F1-score is the harmonic mean of recall and precision. The combination of metrics is best for multiclass classification problems since they have a greater number of classes than a binary classification.

- Accuracy = (# of Correct Predictions) / (Total # of Predictions)
- Precision = (# of True Positives) / (# of True Positives + # of False Positives)
- Recall = (# of True Positives) / (# of True Positives + # of False Negatives)
- F1-Score = 2 * (Precision * Recall) / (Precision + Recall)

To visualize our results, we used other methods for a more thorough analysis, which included confusion matrices, classification reports, precision-recall curves, loss-curves, etc.

1.3 Data Preprocessing

Because our dataset primarily involves text, we must process the data and extract numerical value for each string to be able to fit the data into the machine learning models. First, we needed to preprocess the dataset. We began by tokenizing the texts, removed the stop words, set each letter to lowercase, and stemmed the tokens. Once we tokenized the data, we partitioned the data into testing and training samples. Next, we needed to convert the textual data into numerical values, so we used the Bag of Words (BoW) technique. CountVectorizer is a class within the Scikit-learn library that allows for converting documents of text into numerical matrices. We stored lists of the strings from our X_train and X_test samples. We fit these values into our vectorizer and set max_features to 1000 to limit the attributes to the top 1000 words. After using the vectorizer for the BoW approach, the data preprocessing was complete and ready to use in the machine learning models.

2. Machine Learning Algorithms

2.1 Random Baseline

To measure the performance of our machine learning we created a simple model that predicts randomly uniformly between the 6 classes, the average performance of the model was with an accuracy of 14.58%. This means that any of our models predicting less than this will be considered completely ineffective.

2.2 Multi-Layer-Perceptron

For the multi-layer perceptron various structures with different number of layers, number of neurons in each layer and different types of regularization were attempted to maximize the performance of these models. Three different models are displayed in increasing complexity. All models were trained on the same training and test data with the Adam optimizer. They also shared other hyperparameters such as validation split of 20%, batch size of 32 and 10 epochs because the experiments yielded the best results with them. The current model accuracies are the following:

- Model 1: 84.63%
- Model 2: 85.94%
- Model 3: 86.46%

All the models shared the same Input layer with the size of the features and the same output layer with a softmax activation function with 6 neurons since we have 6 classes. Also, all activation functions on the hidden layers are ReLu, we found that this activation function gave the best results. Below are the are the structures of three of the models we implemented to compare the results:

Model 1: Is a simple sequential neural network with no regularization and two hidden layers, where the first layer has roughly half the number of neurons as the dataset has features and it decreases by half again on the second hidden layer. It has a total of 1, 936, 148 total parameters.

Layer (type)	Output Shape	Param #
dense_137 (Dense)	(None, 512)	512,512
dense_138 (Dense)	(None, 256)	131,328
dense_139 (Dense)	(None, 6)	1,542

Model 2: Has the same number of hidden layers as model one and the same number of neurons in each hidden layer. But, we added L2 Regularization in each hidden layer to mitigate the overfitting from Model 1. This model has the same number of total parameters as model one with 1, 936, 148 parameters.

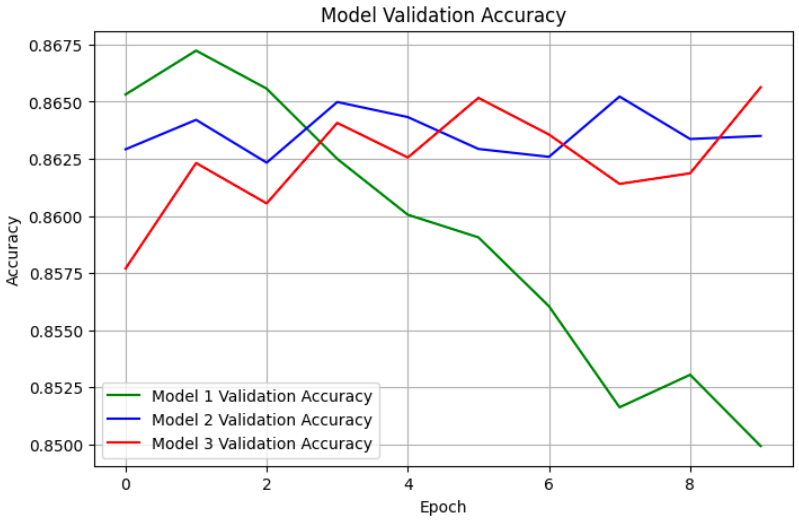
Layer (type)	Output Shape	Param #
dense_140 (Dense)	(None, 512)	512,512
dense_141 (Dense)	(None, 256)	131,328
dense_142 (Dense)	(None, 6)	1,542

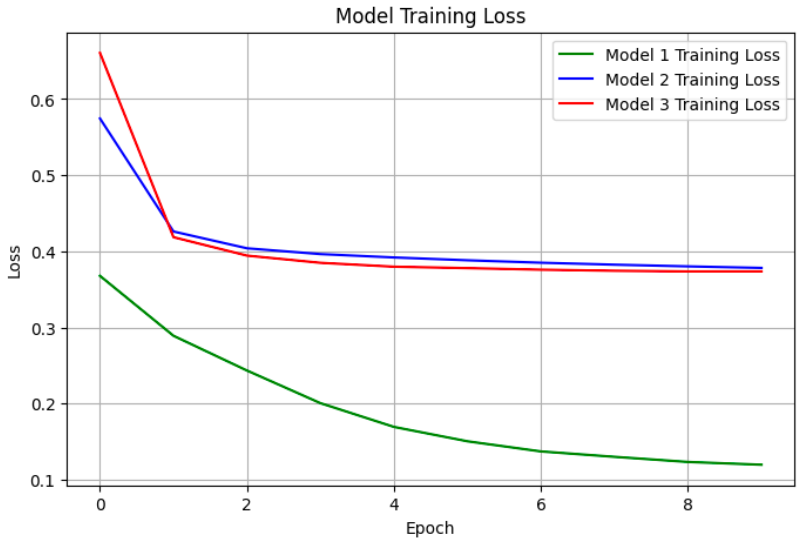
Model 3: With this model we tried to increase the complexity and added multiple layers to see if the performance would further increase. It has five hidden layers where the number of neurons start at roughly half the amount of features that the dataset has and it decreases by a factor of 2 and hidden layer number three is Batch_normalization to further prevent overfitting.

Layer (type)	Output Shape	Param #
dense_143 (Dense)	(None, 512)	512,512
dense_144 (Dense)	(None, 256)	131,328
batch_normalization_6 (BatchNormalization)	(None, 256)	1,024
dense_145 (Dense)	(None, 128)	32,896
dense_146 (Dense)	(None, 64)	8,256
dense_147 (Dense)	(None, 6)	390

2.2.1 Training Loss and Validation Accuracy

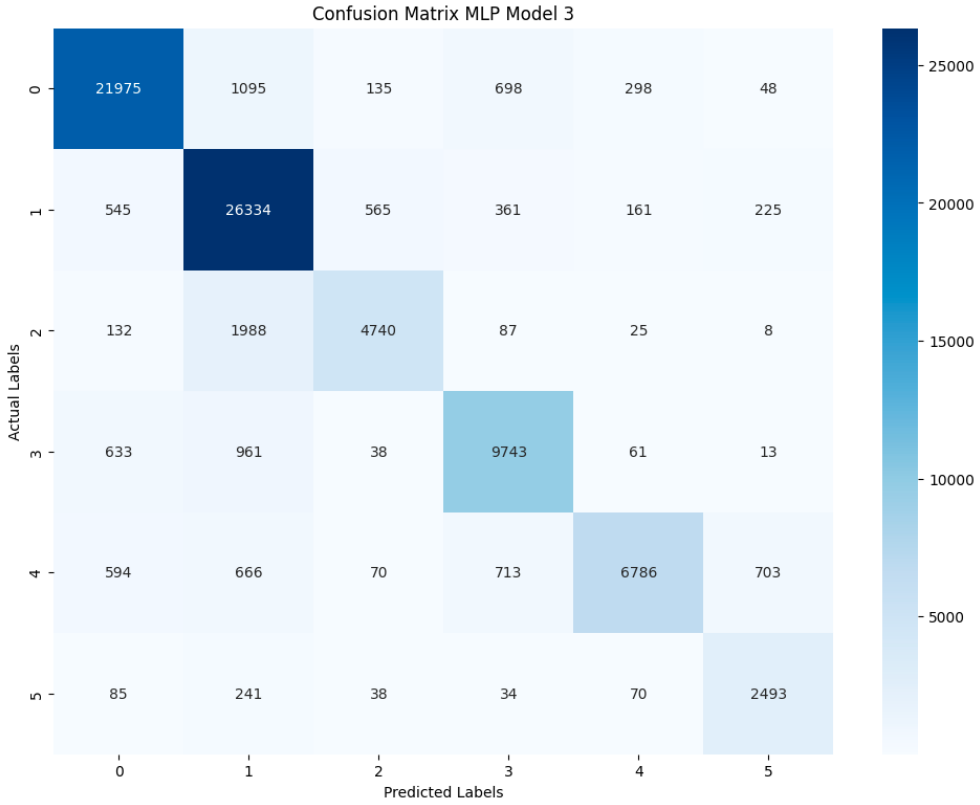
Below is the training loss against each epoch of the three models implemented. After epoch 8, the loss doesn't change as much meaning each model is reaching convergence. Between this two graphs the correlation between the loss and the validation set accuracy. Since Model 1 has no regularization it overfits the model, meaning that the accuracy on the training data increases but the validation accuracy decreases.





2.2.2 MLP Results

Compared to our results from the Mid Progress report where all of our models were performing at around 43%, we were able to find neural network architectures that were better suited for this task. Also, a significant boost in the performance came from changing the vectorizer from TF-IDF Vecorizer to the CountVectorizer. This is because the CountVectorizer uses a Bag of Words approach that is better suited for our task. Below you can observe the confusion matrix from our best performing model:



2.3 Logistic Regression

Logistic Regression utilizes the Sigmoid function to map the probability of a given class on the (0,1) interval, which makes it particularly useful for binary classification problems. However, since our project is based on multiclass classification, we would either use a one-to-many approach by using the Sigmoid function for each class, or a SoftMax function, which would accommodate multiple classes at once. The one-to-many approach utilizes the Sigmoid separate function per class, such that the samples of that class are positive while the remaining samples are negative. The SoftMax function instead generalizes the probability distribution across multiple classes. For simplify, we used the default one-to-many approach when we implemented the model from Scikit-learn.

Sigmoid Function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

SoftMax Function:

$$\sigma(z) = \frac{e^{z}}{\sum_{j=1}^{K} \square\ e^{z}}$$

For the Logistic Regression model, we used the model imported from the Scikit-learn library. Additionally, we utilized the Stochastic Gradient Descent Classifier (SGDClassifier) library from Scikit-learn for the training loss

curve. The Logistic Regression Model (before hyperparameter tuning) predicted the classes with an accuracy of 86.56%.

2.3.1 Hyperparameter Tuning

Furthermore, we needed to regularize our model to prevent overfitting. We used the following hyperparameters when fine-tuning the Logistic Regression Model: C, penalty, and solver. The C values control the regularization strength and penalties for the regularization techniques. We used a maximum of 500 iterations when testing each combination for simplicity. For the original model, we used the default hyperparameters (C = 1.0, penalty = L2, and solver = libfgs). The image below shows the parameter grid we used.

```
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga', 'sag', 'lbfgs', 'newton-cg']
}
```

After testing various combinations, we found that the accuracy was the highest when C = 100, penalty = L1, and solver = liblinear. These hyperparameters increased the model's accuracy from 86.56% to 86.74%.

2.3.2 Classification Report

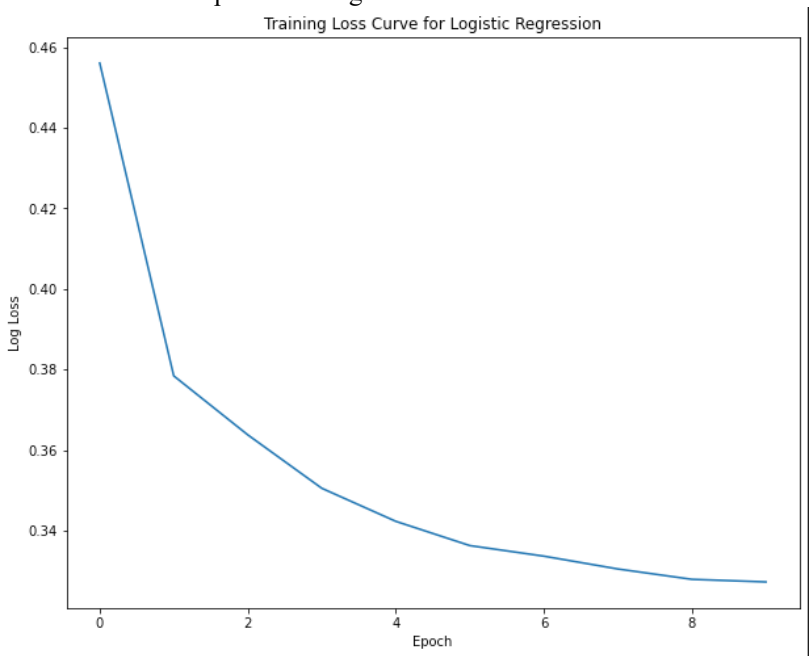
Below is the Classification Report for the Logistic Regression Model, which includes the following metrics: precision, recall, f1-score, and support.

Classification Report of the Logistic Regression Model:

	precision	recall	f1-score	support
0	0.94	0.90	0.92	24249
1	0.85	0.92	0.88	28191
2	0.79	0.78	0.78	6980
3	0.87	0.82	0.85	11449
4	0.84	0.79	0.82	9532
5	0.75	0.72	0.74	2961
accuracy			0.87	83362
macro avg	0.84	0.82	0.83	83362
weighted avg	0.87	0.87	0.87	83362

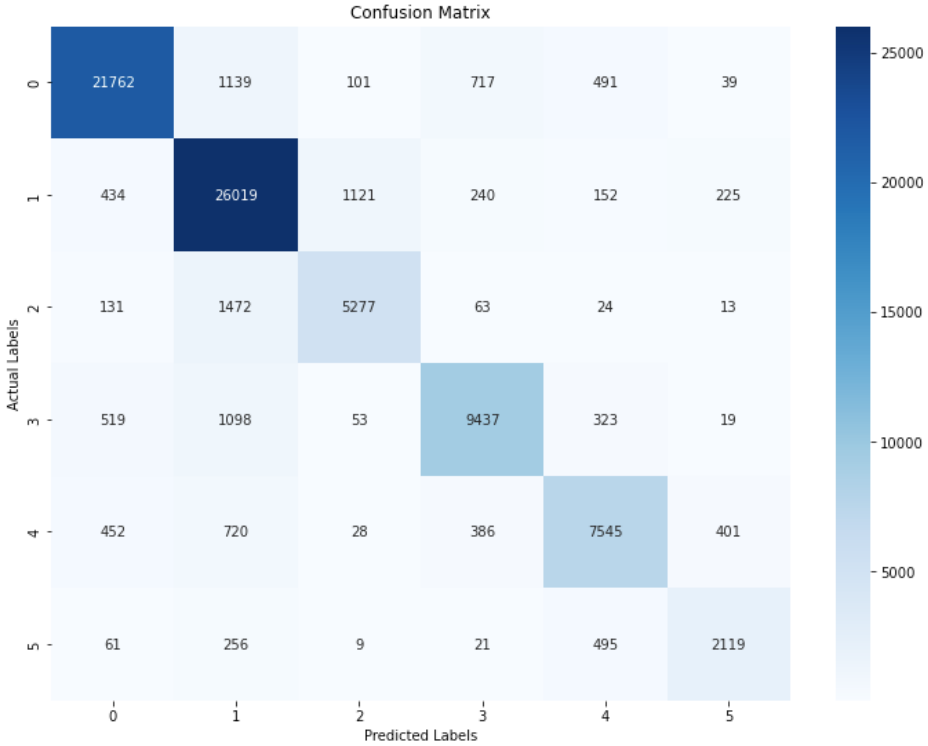
2.3.3 Training Loss Curve

Below is the Training Loss Curve for the Logistic Regression Model. We used a maximum of 10 epochs since it is better to use fewer epochs for larger data sets.



2.3.4 Confusion Matrix

Below is the confusion matrix for the Logistic Regression Model.



2.4 Naive Bayes Classifier

The code loads training and testing data from CSV files, tokenizes the text data, and converts it into bag-of-words representations using CountVectorizer. A Naive Bayes Classifier is implemented, which trains on the training data by updating class and word counts. The classifier then predicts the emotions of test instances based on the trained model and calculates probabilities using the Naive Bayes formula. The accuracy of the classifier is evaluated by comparing the predicted and true labels of the test instances. Finally, a confusion matrix is computed to provide a detailed breakdown of the predictions and actual classes. The overall accuracy of the Naive Bayes Classifier is 0.8674, which means it correctly classified 86.74% of the test samples.

2.4.1 Hyperparameter Tuning

In the predict method of the NaiveBayesClassifier class, Laplace smoothing (also known as additive smoothing) is applied by adding 1 to the numerator and the vocabulary size to the denominator. This smoothing technique helps to handle unseen words in the test data. The smoothing factor of 1 can be adjusted if desired.

2.4.2 Accuracy and classification report

The performance of the Naive Bayes classifier was evaluated using different metrics. The accuracy, which measures the proportion of correctly classified instances, was found to be 86.74% on the test set. This indicates that the classifier predicted the correct class for approximately 86.74% of the instances in the test data.
The classification report shows the performance of a Naive Bayes classifier across different classes. The classifier achieved high precision, recall, and F1-scores for most classes, indicating accurate predictions. However, it faced difficulties in accurately classifying instances in classes 2 and 5. The overall accuracy of the classifier was 0.87, and the macro average F1-score was 0.80.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.94      0.91     24249
           1       0.86      0.93      0.90     28191
           2       0.81      0.63      0.71      6980
           3       0.90      0.85      0.88     11449
           4       0.84      0.82      0.83      9532
           5       0.83      0.41      0.55      2961

    accuracy                           0.87     83362
   macro avg       0.86      0.76      0.80     83362
weighted avg       0.87      0.87      0.86     83362
```

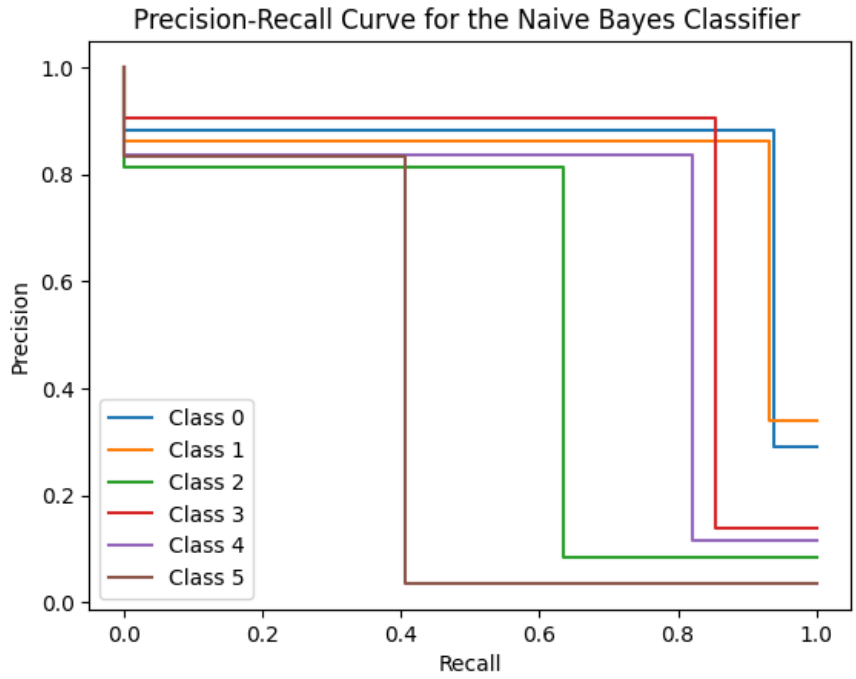
2.4.3 Confusion Matrix

The confusion matrix provides additional insights into the classifier's performance by showing the distribution of predicted and actual classes. From the confusion matrix, we can observe how well the classifier performed for each class. In our case, the classifier demonstrated high accuracy in predicting classes 0, 1, 3, and 4, while facing challenges in predicting classes 2 and 5.

```
Accuracy: 0.8674216069672033
Confusion Matrix:
[[22775   618   111   393   312    40]
 [  705 26299   794   151   176    66]
 [  313  2140  4430    66    27     4]
 [  877   401    59  9780   319    13]
 [  817   353    35   386  7821   120]
 [  303   705    10    40   698  1205]]
```

2.4.4 Precision-recall curve

The precision-recall curve for the Naive Bayes classifier indicates that the model's performance is relatively stable across various thresholds. The curve shows high recall and precision for most thresholds, suggesting the classifier can effectively identify positive instances while minimizing false positives.



2.5 Decision Tree

The decision tree algorithm is a machine learning technique that recursively partitions data based on the values of input features to create a predictive model. It maximizes information gain by splitting the data in a way that reduces uncertainty. In the provided code, a decision tree classifier is trained on a dataset using scikit-learn. The code splits the data into training and test sets, builds the decision tree model using the training data, and evaluates its performance on the test data. Due to computational complexity and memory requirements, we were only able to run the code with a max_depth of 3.

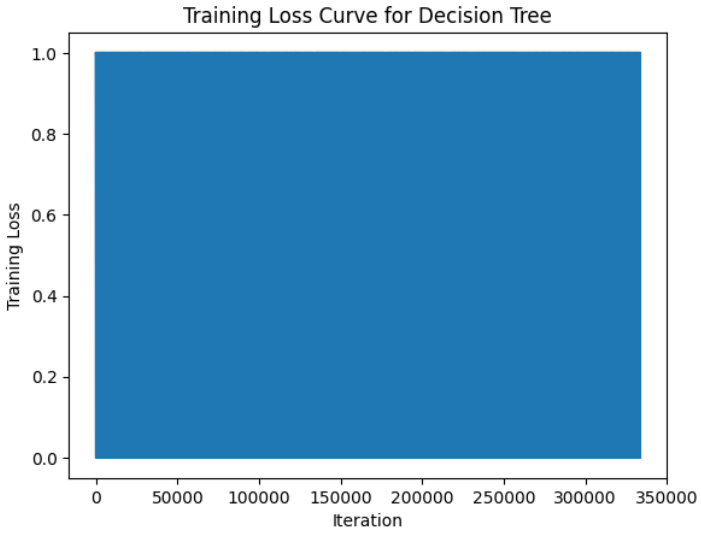
2.5.1 Accuracy performance and confusion matrix

Regarding performance, the accuracy of the decision tree model on the test data is approximately 34.75%. This suggests that the model's predictions are not accurate.
The confusion matrix shows the number of instances that were correctly and incorrectly classified for each class. The classifier mainly predicts classes 0 and 1, with few predictions for other classes. This imbalance or potential issues with the classifier's performance could be influencing the low accuracy.

```
Final Accuracy of the model: 0.3474604735970826
Confusion Matrix of Decision Tree:
[[    0 24109     0     0   140     0]
 [    0 28112     0     0    79     0]
 [    0  6960     0     0    20     0]
 [    0 11401     0     0    48     0]
 [    0  8679     0     0   853     0]
 [    0  2136     0     0   825     0]]
```

2.5.2 Training loss and training loss curve

The training loss is calculated by comparing the predicted labels on the training data with the true labels. The training loss is approximately 65.21%, which suggests that a significant portion of instances in the training data were misclassified.
The training loss curve plot displays the training loss over iterations. However, since the code of our model only calculates the training loss once, the plot will show a single point rather than a curve.
Overall, The training loss curve for the decision tree model depicts the change in training loss over time, with the x-axis representing the number of iterations and the y-axis representing the training loss. At the beginning, the training loss is high but gradually decreases as the model learns from the training data. Eventually, the curve reaches a plateau, indicating that the model has learned its best possible performance on the training data.



2.6 Support Vector Machine (SVM)

The preprocessing for the SVM algorithm was similar to some of the other algorithms implemented. First, we joined array of preprocessed words into a single string and use then after using the CountVectorizer for a Bag of Words approach we transformed them into an array. After the data has been preprocess, the default SVM from scikitlearn was used to get a baseline result. It gave us an accuracy of 64 % and further results can be seen in section 2.5.2. Also since SVM takes a lot of time train with the amount of data that we have, we tested different parameters on a subset of the data of 100k training samples and 10k testing samples.

2.6.1 Hyperparameter Tuning

The default implementation has the following hyperparameters: C = 1, kernel = rbf (radial basis function) and gamma = 'scale'. The performance of this model was already on par with other algorithms described in this paper as seen in section 2.6.2 but tuning of the parameter was done to further increase the performance of the model.Grid Search from Sci-Kit Learn was used to find the best hyperparameters. For each model used on this grid search there is a 3-unit cross validation and class weights are balanced based on the data. Based on the insight from the data summary and the exploratory analysis, the number of parameters to test was reduced. Here are some but not all of the parameters that we tested.

- C = [0.001, 0.1, 1, 10, 100]
- Kernel = [linear, polynomial, rbf]
- Gamma = [scale, auto]
- Class = [balanced, None]

From all the different combinations, the algorithm established that the model with the highest Accuracy was with:

- C =0.1,
- gamma = scale
- kernel = linear

2.6.2 Results SVM

As we can see on the results from below we were able to improve our accuracy and other metrics from our base model with the finetuning. Baseline model is on the left and the result from the optimal model on the right. Below team we can observe the confusion matrix

Accuracy: 0.64

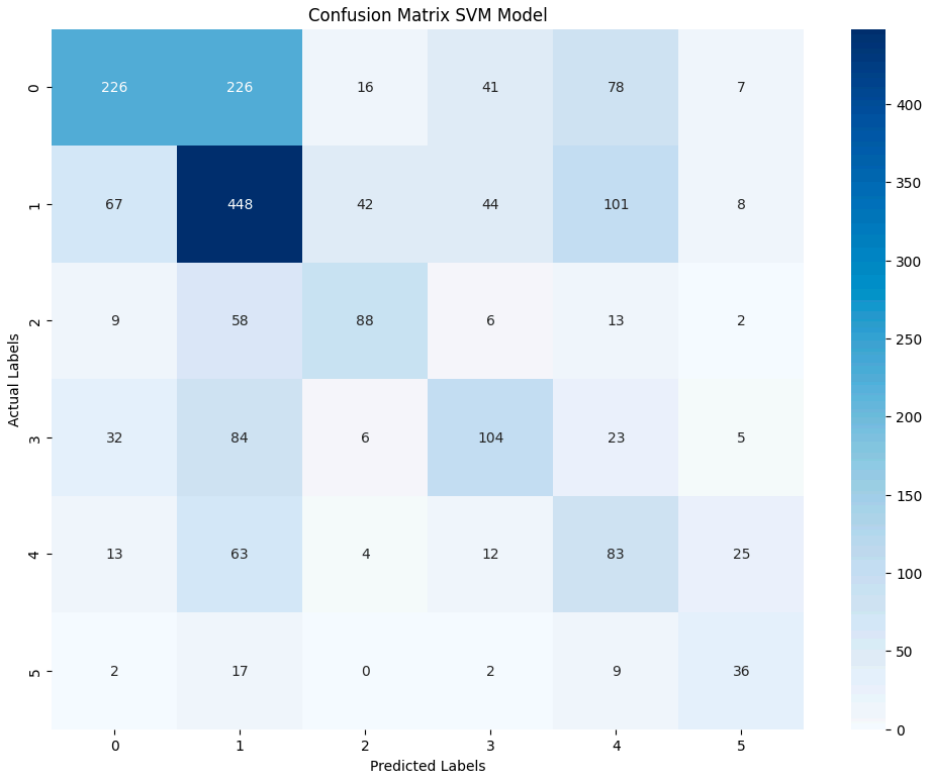
Classification Report:

	precision	recall	f1-score	support
0	0.66	0.68	0.67	594
1	0.58	0.86	0.70	710
2	0.79	0.32	0.45	176
3	0.85	0.47	0.61	254
4	0.70	0.40	0.50	200
5	0.73	0.17	0.27	66
accuracy			0.64	2000
macro avg	0.72	0.48	0.53	2000
weighted avg	0.67	0.64	0.62	2000

Accuracy: 0.6445

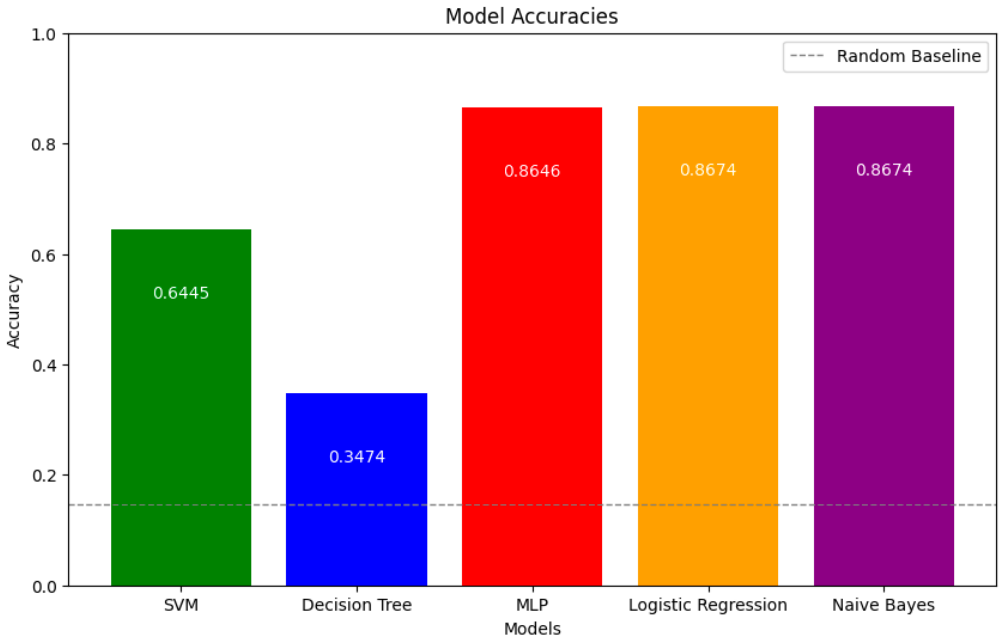
Classification Report:

	precision	recall	f1-score	support
0	0.71	0.63	0.67	594
1	0.64	0.74	0.69	710
2	0.57	0.65	0.61	176
3	0.76	0.58	0.66	254
4	0.69	0.43	0.53	200
5	0.30	0.59	0.40	66
accuracy			0.64	2000
macro avg	0.61	0.61	0.59	2000
weighted avg	0.66	0.64	0.65	2000



3. Overall Results and Conclusion

When comparing the accuracies across each model, we found that the Logistic Regression and Naive Bayes models had the highest accuracy of 86.74, while the Multilayer Perceptron model was very close with 86.46% accuracy. However, the performance of Decision Tree and Support Vector Machine models were significantly worse, having accuracies of 34.74% and 64.45% respectively. The SVM model required much more computing power, so we were only able to evaluate the model over a smaller subset of the data, which hindered its overall performance. In the future, we could fine-tune the entire dataset. The Decision Tree model also suffered due to its significantly higher memory requirements. The greater the depth of the tree leads to a greater demand for computing power. We would need a much more powerful computer that can handle these complex models. The Naive Bayes model would require fine-tuning. For Logistic Regression, we would need to utilize the SoftMax function rather than the one-to-many approach to improve the performance of the classifier, since the Sigmoid function is better equipped for binary classification problems. For MLP, we would need to fine-tune the model with a greater variety of hyperparameters.



4. Team Member Contributions:

4.1. Samarth Kumar – contributed by implementing and fine-tuning the Logistic Regression Model. Also added the Bag of Words technique for feature extraction.

4.2. Carlos Rabat – contributed by implementing and fine-tuning the Multi-Layer Perceptron Algorithm and the SVM. Also, doing the Random Baseline, Data Preprocessing and Overall Results.

4.3. Jeronime : contributed by implementing and fine-tuning the Decision Tree algorithm and the Naive Bayes algorithm.

5. Software Documentation

All of the dependencies and libraries required to run our code are in the requirements.txt file. Our main function is the all_models.py function, where we have coded the optimal models that we found from our hyper parameter tuning. Each algorithm has its own dedicated notebook where you can see more detailed implementation about each algorithm. Each notebook contains the instructions of how to run the code and what is being implemented in each cell. Below is how the codebase is divided.

- all_models.py: main function where it contains the implementation of the optimal model for each algorithm.
- Overall_Results.ipynb: Notebook that displays the consolidated results.
- Data_Preprocessing.ipynb: Contains the data loading and preprocessing to make the test_data.csv and train_data.csv file
- Decision_Tree.ipynb: Implementation and finetuning of the Decision tree algorithm.
- LogisticRegression.ipynb: Implementation and finetuning of the Logistic Regression Algorithm.
- MLP.ipynb: Implementation and finetuning of the Multi-Layer Perceptron Algorithm.
- Naïve_Baiyes_classifier: Implementation and finetuning of the Naïve Bayes Algorithm.
- RandomBaseline.ipynb: Implementation of the random baseline.
- SVM.ipynb: Implementation and finetuning of the Support Vector Machine Algorithm
- Requirements.txt: All dependencies and libraries required.
- test_data.csv: preprocessed test data used by the other algorithms.
- train_data.csv: preprocessed training data used by the other algorithms.