Machine Learning Mini Project -1

TEAM: Parameter Hunters

Author	**Roll No**
Pokala Dattatreya	241110048
Voora Nagendra Bhaskara Swamy	241110044
Telugu Sudhakar	241110077
Edula Vinay Kumar Reddy	241110024
Tippireddy Yashwanth	241110082

1 TASK 1

1.1 Emoticon Dataset

1.1.1 Feature Exploration and representations:

Emoji to Unicode Representation:

- The input_emoticon column contains sequences of 13 emojis. Each emoji was placed in a separate column, after which it was replaced by its corresponding Unicode representation.
- It was observed that each Unicode representation contained a common prefix, which was removed to optimize the input format.
- The final transformed input was passed to a Feed-Forward Neural Network, resulting in a training accuracy of 75% and validation accuracy of 52% (using validation data extracted from the training dataset).
- GRU , LSTM, BiDirectional LSTM are also used to train and predict , but none of them crossing 70% accuracy, found this won't be a good choice of feature representation.

Possible reasons of failure:

- The Unicode values assigned to emojis are not related to their meanings or relationships. Since the Unicode encoding is not based on any semantic or visual similarity between emojis, the neural network is not learning any useful patterns from these numeric representations.
- By replacing emojis with their Unicode values, may lose the rich meaning each emoji conveys. The network might treat emojis with nearby Unicode values as more similar, even if they are conceptually very different, leading to poor performance.
- Even after removing the common prefix in the Unicode representations, the remaining part of the Unicode might still carry no meaningful relationship for classification, resulting in random or irrelevant patterns being learned.

Positional Encoding:

- A set of unique emojis was created, and each emoji was assigned a unique index.
- A positional encoding scheme was applied to the features.
  Data transformation:
  - A zero matrix of dimensions (sequence_length, number_of_unique_emojis) was created. Where each row represents an emoji in the sequence, and each column corresponds to a unique emoji.
  - Number of unique emojis were 214 over the training data.
  - Created a matrix of 13X214 for each row in training data. for each emoji in the sequence, the corresponding index was looked up in a dictionary and the respective column was set to 1(if it presents in dictionary).
  - By using method, we are encoding emoji and its position. since if an emoji "x" present in 3 rd position in input_emoticon row 1 and 5^{th} position in input_emoticon row 7 , then in the 3 rd row of matrix of input_emoticon row 1 is equal to 5^{th} row of matrix of input_emoticon row 7.
  - This encoded matrix was flattened into a 1D array to make it compatible with machine learning models.

Model Performance:

Using this positional encoding representation improved the model's accuracy, as it allowed the model to differentiate between distinct emojis more clearly. In contrast, representing emojis by their Unicode numeric suffixes yielded poor results, since the numeric values of Unicode are irrelevant when it comes to distinguishing between different emojis. The Positional encoding provided a better separation between features, leading to improved model performance.

Model	Training Accuracy	Validation Accuracy
KNN	56.53%	53.17%
Decision Tree	100%	76.28%
Logistic Regression	97.60%	89.16%
SVC	96.7%	89.78%

Table 1: Different Models Performance

Removing Irrelevant Features and Positional Encoding:
Upon exploring the data, it was found that certain emojis had little importance for classification. We attempted to remove these irrelevant emojis to enhance model performance.
The importance of each emoji was determined using the permutation_importance function from the sklearn.inspection module.
After finding out the importance of each emoji , we tried removing each one of them in the order of least importance , it was found that after removing 7 least emojis model performance is increased signifacantly. Those seven least important emojis were removed. Among these, four appeared only once per row, and three appeared twice per row.
Data transformation:

- Each row left with 3 emojis after removing those 7 emojis since four appeared only once per row, and three appeared twice per row.
- A zero matrix of dimensions (sequence_length,number_of_unique_emojis) was created. Each row represents an emoji in the sequence, and each column corresponds to a unique emoji.
- Number of unique emojis were 207 over the training data after Removal 7 least important emojis.
- Created a matrix of 3X207 for each row in training data. For each emoji in the sequence, the corresponding index was looked up in a dictionary and the respective column was set to 1(if it presents in dictionary).
- By using this method, we are encoding emoji and its position. since if an emoji "x" present in 1 st position in input_emoticon row 1 and 3 rd position in input_emoticon row 7 , then in the 1 st row of matrix of input_emoticon row 1 is equal to 3 rd row of matrix of input_emoticon row 7.
- This encoded matrix was flattened into a 1D array to make it compatible with machine learning models.
- Finally each row is represented with 621 length 1D array.
- Same transformation is applied while doing validation and test predictions.

Model Performance:

Model	Training Accuracy	Validation Accuracy
KNN	100%	56%
Decision Tree	100%	87%
Logistic Regression	100%	97%
SVC	100%	97%

Table 2: Different Models Performance(removing irrelevant features)

Logistic regression and Support Vector Classifier is working good in this case. After observing both of them over training with low to high no of training samples. Logistic Regression is performing better than SVC. We are choosing logistic regression as classification model.

1.1.2 Final Model:

Logistic regression with removing irrelevant features and positional encoding has chosen for the final model. The results were as follows:

Percentage of training data	Validation Accuracy
20%	88.34%
40%	93.87%
60%	95.50%
80%	96.11%
100%	97.14%

Table 3: Model accuracy

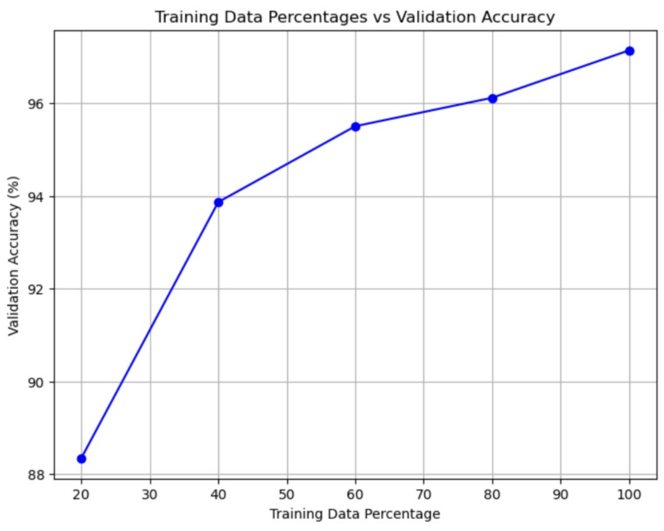

Figure 1: % of training data vs Accuracy

Number of parameters:
622 (no of features(621) + intercept(1))

1.1.3 Conclusion:

Removing the emojis which are less important improved the model's performance. By reducing the number of features and giving an efficient representation, the model focused more on the relevant data, allowing it to generalize better. This helped streamline the feature space and enabled the model to work more effectively with the remaining features and finally given an accuracy of 97.14% on validation data.

1.2 Deep Features Dataset

1.2.1 Feature Exploration and representations:

1. **Feature Transformation (Averaging)**
   Each input example, originally represented as a 13x768 matrix, was transformed by averaging the values in each column, resulting in a 768x1 vector.
   Model Performance on Averaged Features:

Model	Training Accuracy	Validation Accuracy
Logistic Regression	55.25%	45.60%
SVM with Linear Kernel	54.59%	47.44%
SVM with RBF Kernel	56.55%	48.87%

Table 4: Model Performance

   Analysis: Averaging the columns resulted in significant information loss, as the model no longer captured the row-wise dependencies between the features. This led to poor performance across all models, as reflected by the low validation accuracy.

2. **Feature Transformation (Flattening)**
   Flattened each 13x768 matrix into a 9984x1 vector by concatenating the rows sequentially. This preserves all feature information but at a much higher dimensionality.
   Model Performance on Flattened Features:

Model	Training Accuracy	Validation Accuracy
Logistic Regression	99.97%	98.77%
SVM with Linear Kernel	99.25%	98.56%
SVM with RBF Kernel	100%	53.9%

Table 5: Model Performance

   Analysis: Flattening the features significantly improved model performance, especially for Logistic

Regression, which achieved the best results on both the training and validation datasets. Based on this strong performance, we proceeded with Logistic Regression for subsequent experiments.
Feature Selection Since flattening yielded high accuracy, we then re-ran the Logistic Regression model with L1 regularization L1 regularization encourages sparsity by forcing some feature weights to zero, that highlights the most important features. From this process, we identified that rows 1, 7, and 12 (0-based index) were the most important, as the remaining rows had weights close to or equal to zero.

3. **Feature Transformation (Flattening Just Important Rows)**
To reduce dimensionality and focus only on the significant features, we transformed each 13x768 matrix by flattening only the important rows 1, 7, and 12. This resulted in a reduced 2304x1 vector (3 rows x 768 columns).

Model Training with Reduced Features
We trained **Logistic Regression model** without regularization on these flattened vectors containing only the important rows. The model again achieved **99.9% accuracy** on the training dataset and **98.7%** accuracy on the validation dataset, showing that using only the important features resulted in the same high performance as using all rows.
Total parameters: 2305 (including the bias term)
Results of Model: Even with only 20% of the training data, the model achieved over 96% accuracy,

Percentage of training data	Validation Accuracy
20%	96.32%
40%	97.34%
60%	98.16%
80%	98.77%
100%	98.77%

Table 6: Model performance on % of training data used

demonstrating its robustness and efficiency. The performance remains consistent as the dataset size increases, confirming that the model generalizes well with fewer features.

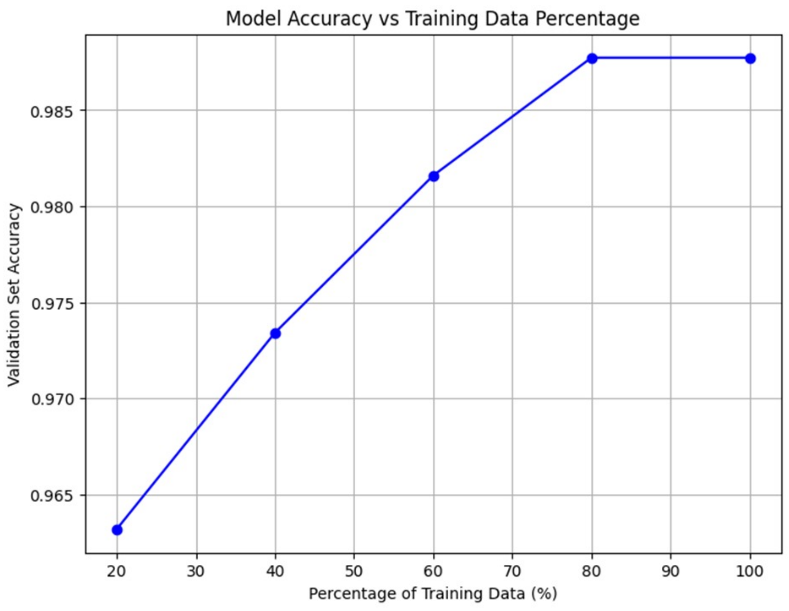

Figure 2: % of training data vs Accuracy

1.2.2 Conclusion

The exploration of feature transformation techniques revealed that flattening the input matrix yielded the best performance. L1 regularization helped identify the most important rows 1, 7, and 12 (0 based indexing), which allowed us to reduce dimensionality without compromising accuracy. The model trained on Reduced Features maintained excellent performance, achieving 98.77% accuracy on the full training dataset, even when trained on smaller subsets of the data, the model generalized well, achieving 96.32% accuracy with just 20% of the training data.

1.3 Text Sequence Dataset

Assumption: The dataset provided is an encoded version of the original dataset, where all the data from different columns is combined into a single string. At first, we did not know how the information from the columns was encoded into this string. We started by assuming that each feature (column) had a fixed length and that these features were joined together to create the full string. Based on this assumption, we experimented with different models, treating each feature as having a fixed length.

1.3.1 Naive Bayes Classification on Grouped Text Sequences

Feature transformation:
The input sequence is split into groups of k digits. This means every k consecutive digits form a "group," and there will be a total of num_features groups (calculated as num_features = math.ceil(50/k)). The grouping ensures that the 50-digit input is converted into structured features that can be used as inputs to a model. If the last group does not have enough digits (e.g., only 2 digits are left), the code will pad the group with zeroes to maintain consistency (e.g., 90 becomes 900).The input sequence was initially transformed using TF-IDF (Term Frequency-Inverse Document Frequency) to represent the digits as numerical values based on their importance in the dataset. This method allowed us to capture the significance of individual characters and character groups as ngrams (unigrams, bigrams, trigrams). However, after experimenting with this approach, we found that assigning integers to each group of digits produced better results for our model.
Algorithm & Training:
Categorical Naive Bayes assumes that the probability of each feature (grouped digits) is independent of the others, given the class label.
For each feature (group), it calculates the probability of that group given the class label and multiplies these probabilities across all groups to predict the most likely class for new data.
Validation (Prediction & Evaluation):
After training, the algorithm uses the validation dataset to test the model's performance. The accuracy of the model is calculated by comparing the predicted labels (y_pred) with the actual labels (y_valid).
We have experimented with different values of K (group size) ranging from 2 to 10. For each value of K, the model is trained and validated, and its accuracy is stored. Highest accuracy of all is achieved when K=3. Classification report on validation dataset after training on 100% of train dataset.
Validation Accuracy : 0.7137
Validation Classification Report :

	Precision	Recall	f1-score	Support
0	0.73	0.70	0.72	252
1	0.70	0.73	0.71	237
Accuracy			0.71	489
Macro Avg	0.71	0.71	0.71	489
Weighted Avg	0.71	0.71	0.71	489

Table 7: Classification Report

1.3.2 LSTM Classification on Integer-Encoded Text Sequences

We choose, LSTM because it is ideal for capturing patterns in the given text sequences that may not be immediately obvious and they are designed to handle sequential data, allowing them to learn the temporal dependencies between text effectively.
Feature Transformation:
The input sequences are pre-processed by converting each character of the string into an integer representation. The integers represent the individual digits in the original sequences, which allows the model to effectively learn patterns in the data.
Model Creation and Training:

- The model architecture consists of:
  - An embedding layer to transform integer sequences into dense vector representations.
  - An LSTM layer to process the sequential data.

– A dense output layer with a suitable activation function to produce binary classifications.

K-Fold Cross-Validation:
To ensure robust model evaluation, we used K-Fold cross-validation with 3 splits. This technique helps assess the model's performance more reliably by utilizing different subsets of the training data for validation. We experimented with various optimizers (Adam, RMSprop, SGD), loss functions (Binary Cross entropy), and activation functions (sigmoid,relu) to find the best-performing model configuration.

Total parameters: Embedding layer with 640 parameters that transforms input sequences into dense vector representations, an LSTM layer with 5,184 parameters that captures sequential dependencies in the data, and a Dense layer with 17 parameters for output prediction. Overall, the model has a total of 17,525 parameters, of which **5,841** are trainable.

Validation (Prediction & Evaluation):

- After training across all folds, we calculated the average accuracy for each configuration. The highest accuracy achieved was identified, and the corresponding parameters were saved for the final model. Highest accuracy is achieved at optimizer = Adam, activation = sigmoid.
- The final model was trained on the entire training dataset using the optimal parameters found during cross-validation. We implemented callbacks such as ModelCheckpoint and EarlyStopping to save the best model based on validation accuracy and prevent overfitting.
- We achieved an accuracy of 86.71%

Final Validation Accuracy : 86.71%
Model:"sequential_10"

Layer (type)	Output shape	Param
embedding_10 (Embedding)	(None, 50, 64)	640
lstm_10 (LSTM)	(None, 16)	5,184
dense_10 (Dense)	(None, 1)	17

Table 8: Model Summary

Total params: 17,525
Trainable params: 5,841
Non-trainable params: 0
Optimizer params: 11,684

1.3.3 Feed forward neural network with Custom Tokenization:

After experimenting with various models on this dataset, we recognized that the data corresponding to each feature must be of variable length. We also noted that the dataset contains numerous strings of different lengths, many of which repeat. Consequently, we decided to transition to tokenization models that tokenize each text into frequent tokens, then ML models are applied.

Data Preprocessing and Token Extraction

1. **Generating tokens from the dataset:**
   - **Find All Substrings:**For each input string, all possible substrings are generated. These substrings include combinations of characters of varying lengths, starting from the first character.A frequency count is maintained for how often each substring appears across all input strings.
   - **Filtering and Extracting Substrings for Tokenization:** First, common substrings that appear in all inputstrings are identified, sorted by length, and filtered to remove redundant or overlapping substrings.Then, the input strings are split into smaller parts using these common substrings as delimiters, and theremaining smaller substrings are counted. Substrings that appear frequently enough are retained.Finally, the token list is created by combining both the common substrings and the frequent smaller substrings, which will be used for future tokenization.
2. **Custom Tokenization: Apply_tokenization Function:**
   In this function:
   - **Token Matching:** For each input string, the function attempts to match the longest possible token from the extracted set of tokens. If no match is found, the string is labeled with an "[UNK]" token (unknown).
   - **Token Representation:** The tokenized strings are represented as sequences of tokens, with spaces separating them.

   The tokenized strings will then be used as input to the neural network model.

Model Preparation and Training

1. **Tokenization and Sequence Padding**
   Before feeding the tokenized data into the model, further preprocessing steps are applied:
   - **Fitting a Keras Tokenizer:**The Tokenizer object from the tensorflow.keras.preprocessing.text module is used to convert the tokenized text into sequences of integers.

- **Padding Sequences:** Since the neural network requires input sequences of uniform length, the sequences are padded to the length of the longest sequence.

2. **Model Architecture**
   A simple neural network model is implemented using TensorFlow's Keras API
   - **Embedding Layer:** Converts the integer token sequences into dense vector representations. The embedding dimension is set to 8.
   - **Flatten Layer:**The output of the embedding layer is flattened to create a 1D vector.
   - **Dense Layers:** Two fully connected (dense) layers follow, with ReLU and sigmoid activations. The final layer outputs a single probability for binary classification.
3. **Experimentation with Various Deep Learning Models**
   We have experimented with various types of deep learning models to identify the best-performing architecture for our task. The following models were tested:
   - **GRU (Gated Recurrent Unit)**
   - **CNN (Convolutional Neural Network)**
   - **Bi-directional LSTM/GRU**
   - **Transformer Models**
   - **Pre-trained Language Models (BERT, GPT, etc.)**
   - **Simple Feedforward Neural Network (FFNN)**

   In addition to different model architectures, various loss functions and hyperparameters were also explored. Notably, the Simple Feedforward Neural Network (FFNN) performed exceptionally well, achieving over 90% accuracy with just 20% of the training data.

Model Compilation and Callbacks
The model is compiled with the Adam optimizer, binary cross-entropy loss, and accuracy as the evaluation metric. Two callback mechanisms are employed:
- **Early Stopping:** Stops training when the validation loss does not improve for five consecutive epochs.
- **Model Checkpoint:** Saves the best version of the model based on validation performance.

Parameters:

Layer (type)	Output shape	Param
embedding_14 (Embedding)	(None, 20, 8)	1,712
flatten_14	(None, 160)	0
dense_28 (Dense)	(None, 8)	1,288
dense_29 (Dense)	(None, 1)	9

Table 9: Model Summary with Layer Parameters

Total params: 9,029
Trainable params: 3,009
Non-trainable params: 0
Optimizer params: 6,020
Results:
Validation Accuracy with 20% training data: 0.91
Validation Accuracy with 40% training data: 0.94
Validation Accuracy with 60% training data: 0.96
Validation Accuracy with 80% training data: 0.97
Validation Accuracy with 100% training data: 0.98
Graph:

2 TASK 2

2.1 Approach 1: Taking Majority Output / Weighted Average Based on Accuracy

In this approach, we tried to use the predictions from all three models individually, to make one final prediction. We looked at two different ways of doing this:
The first way was to take the output from each model and then:
- **Majority Voting:** Where we simply see what majority of the three models predict and make that the final output.

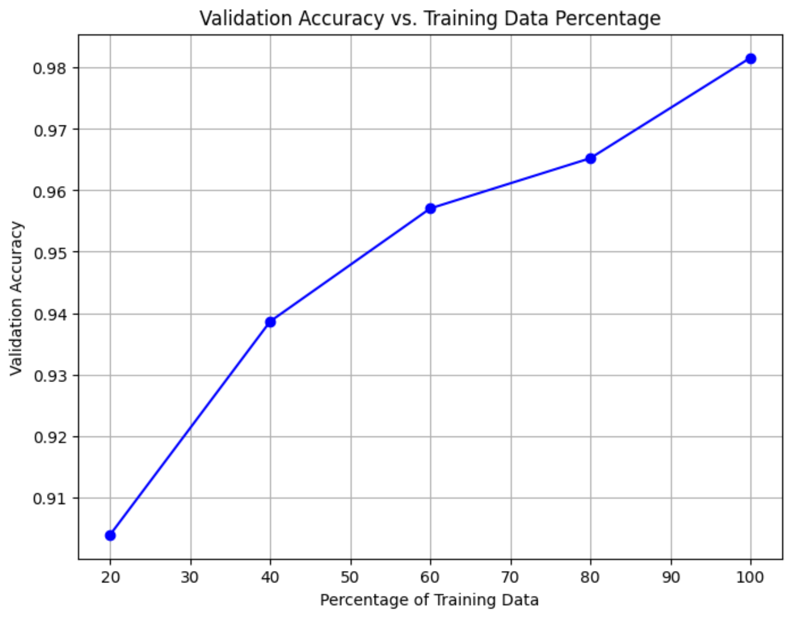

Figure 3: % of training data vs Accuracy

- **Weighted Average:** Here we used the accuracy score of each model. The model with a better accuracy score gets more weight, and we combine the predictions based on those weights to decide the final prediction.

As the accuracies of all the three models are almost same, weighted average method also produced similar results as majority voting. Using this approach, following are the validation accuracies we achieved using various fractions of training data for training,

Percentage of Training data	Accuracy
20%	94%
40%	96.5%
60%	97.3%
80%	97.7%
100%	98.1%

Table 10: Model Performance

2.2 Approach 2: Using Confidence of Model prediction

The second approach we tried was to look at the confidence score of each model's prediction. Instead of only considering the model's accuracy as weights, we considered how sure (confident) the model is about each specific prediction. If a model is more confident in a prediction, we trust it more, even if the other models are accurate overall but are not as confident for that specific prediction. The main reason for choosing this approach is that all three models have similar validation accuracy. Therefore, it is preferable to rely on the output from the model with the highest confidence rather than considering predictions from models that are less certain.

Implementation:

To implement this, we followed these steps:

1. We took the data points from each of the three datasets (text sequence, emoticon, and deep features) and passed them through their respective models.
2. Each model gave us a prediction, and also a confidence score (how sure the model is about that prediction).
3. Finally, we compared the confidence scores of all three models for each prediction.
4. The final prediction was taken from the model with the highest confidence score.

Using this approach, following are the validation accuracies we achieved using various fractions of training data for training:

By using the model that had the highest confidence for each prediction, we ensured that we trusted the

Percentage of Training data	Accuracy
20%	95.2%
40%	97.3%
60%	98.3%
80%	98.7%
100%	98.5%

Table 11: Model performance

prediction where the model was the most confident, rather than just relying on majority or weighted accuracy. This way, we focused on the model that felt strongest for that particular case.

2.3 Approach 3: Concatenating Features from All Three Models and Training a New Machine Learning Mode

In this approach, we combined features learned from three different models and then trained a new machine learning model on this combined dataset. The shapes of training dataset after applying feature transformations in task-1 are as follows:

1. **Shape of Emoticon Dataset after Feature Transformation:** (7080, 621)
2. **Shape of Deep Features Dataset after Feature Transformation:** (7080, 9984)
3. **Shape of Text Sequence Dataset after Feature Transformation:** (7080, 18)

After concatenating all these datasets, we created a new combined dataset with a shape of (7080, 10623). This combined dataset was used for Task 2, where we applied different machine learning models to evaluate their performance. Below are the results of our experimentation:

Model Performance

1. **Logistic Regression:** After training Logistic Regression on the combined dataset, we achieved a validation accuracy of **97%.**
2. **K-Nearest Neighbors (KNN):** The KNN model, however, performed poorly, achieving a validation accuracy of only **50%**. This could be due to the high dimensionality of the dataset, which makes distance-based algorithms like KNN less effective.
3. **Feedforward Neural Network:** Tried various layers and activation functions and the best model we got have two hidden layers with 128 and 64 neurons, using ReLU activation functions and dropout to prevent overfitting. The output layer had a sigmoid activation function, and the model was compiled using the Adam optimizer with binary cross-entropy loss. The validation accuracy achieved was **97%**.
4. **Gradient Boosting Classifier:** We experimented with different combinations of hyperparameters like n_estimators, learning_rate, and max_depth. The best result we achieved with this model was a validation accuracy of **98%**.
5. **Support Vector Machine (SVM):** After training the SVM model on the combined dataset, we achieved a validation accuracy of **96%**.

Best Model: Random Forest Classifier

Our best results were achieved using the **Random Forest Classifier**. We trained the model using various n_estimators(number of trees) ranging from 10 to 100. The best validation accuracy we obtained was **99.39%**. Since the Random Forest Classifier involves random sampling of data, random feature selection, and tree construction, the results slightly vary each time the model is run, even with the same parameters. For this reason, we saved the model with the best weights, which correspond to the highest accuracy achieved during our experiments. The concatenation of features from all three models provided a rich feature space, and after testing various machine learning models, the **Random Forest Classifier** performed the best with a **99.39%** validation accuracy which is greater than individual models in task-1

Result:

Percentage of Training data	Accuracy
20%	97.14%
40%	98.16%
60%	98.57%
80%	99.18%
100%	99.39%

Table 12: Model Performance

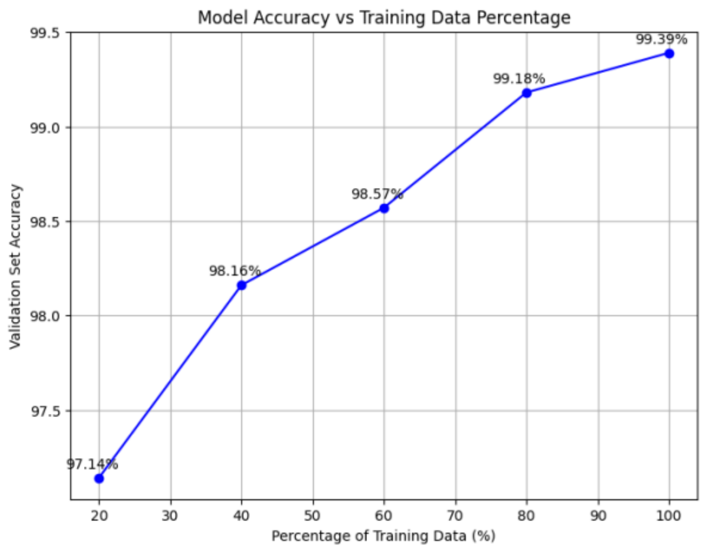

Figure 4: % of training data vs Accuracy