ML Project Report

Name: Karunaya Garg

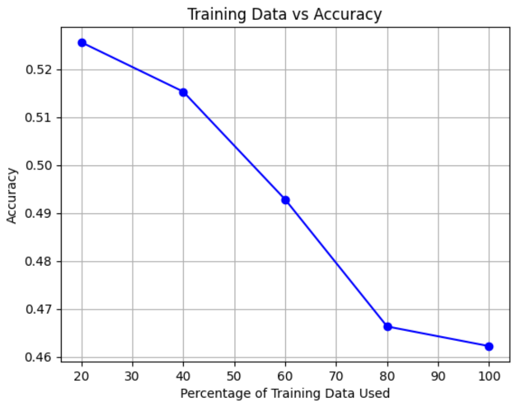
TASK - 1
DATA SET – 1

Emoticons

First, we implemented a **Logistic Regression model** with essential preprocessing steps, including converting all text to lowercase and stripping unnecessary whitespaces while retaining special characters like emoticons. This preserved the key features for classification. Using **TF-IDF Vectorization**, we transformed the emoticon-based text into numerical features. Character-level analysis in TF-IDF allowed us to capture the frequency and importance of each character, helping the model weigh the significance of emoticons effectively.

We trained the logistic regression model on varying portions of the training data (20%, 40%, 60%, 80%, and 100%) to evaluate its performance. To optimize the model's hyperparameters, we applied **GridSearchCV** to fine-tune the regularization parameter (C), balancing overfitting and generalization. The model was evaluated on the validation set using accuracy, precision, recall, and F1-score, and we observed consistent improvements as more training data was used.

Percentage of Training Data Used	Accuracy
20	**0.5256**
40	0.5153
60	0.4928
80	0.4663
100	0.4622



Pseudo Code:

```
1. Load Training and Validation Datasets:
- Load the emoticon classification data from CSV files.
2. Preprocess the Text:
- Convert text to lowercase.
- Strip unnecessary whitespaces but retain special characters (emojis).
3. Convert Text to Features using TF-IDF:
- Use the TF-IDF vectorizer to convert the preprocessed text into a matrix of numerical features.
- Set the analyzer to 'char' to handle individual emoticons and characters.
4. Encode Labels:
- Use LabelEncoder to transform the labels into numeric form for logistic regression.
5. Tune Hyperparameters (Optional):
- Use GridSearchCV to find the best regularization parameter (C) for logistic regression.
- Perform 5-fold cross-validation to determine the optimal parameter.
```

6. Train the Model on Varying Percentages of Training Data:

- For each subset of training data (20%, 40%, 60%, 80%, 100%):

- Train a logistic regression model.

- Evaluate the model on the validation data using accuracy, precision, recall, and F1-score.

7. Plot and Print Results:

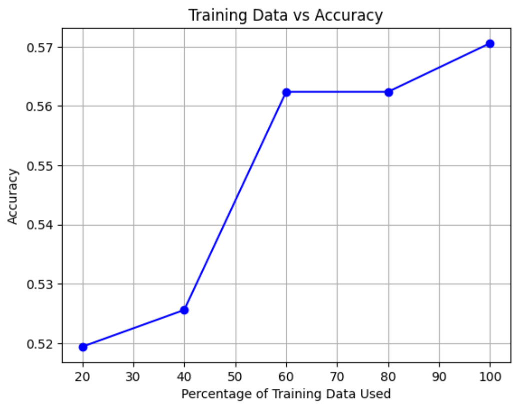
- Plot the performance metrics across different data percentages.

- Print the accuracy, precision, recall, and F1-score for each percentage.

We implemented a **Random Forest model** as the Logistic Regression model did not yield satisfactory results. The preprocessing pipeline involved converting text to lowercase, removing unnecessary punctuation, and stripping whitespace while retaining emoticons—the key features for classification. This ensured that irrelevant characters were removed while meaningful symbols remained intact. We used **TF-IDF Vectorization** to convert the emoticon text into numerical features, capturing both unigrams and bigrams of characters. Additionally, we limited the number of features to reduce noise and focus on the most important patterns.

The **Random Forest model** was trained with 200 trees and balanced class weights to handle class imbalances. We trained the model on varying portions of the data (20%, 40%, 60%, 80%, 100%) to evaluate performance scaling. Finally, the model was evaluated on the validation set using accuracy, precision, recall, and F1-score to assess its classification capabilities comprehensively.

Percentage of Training Data Used	Accuracy
20	0.5194
40	0.5256
60	0.5624
80	0.5624
100	**0.5706**



Pseudo Code:

1. Load Training and Validation Datasets:

- Load the emoticon classification data from CSV files.

2. Preprocess the Text:

- Convert text to lowercase.

- Strip leading and trailing whitespaces.

- Remove unwanted punctuation while retaining emoticons.

3. Convert Text to Features using TF-IDF:

- Use the TF-IDF vectorizer to convert the preprocessed text into a matrix of numerical features.

- Capture both unigrams and bigrams to recognize patterns in emoticons.

4. Train the Random Forest Model:

- Train a Random Forest classifier with 200 trees and balanced class weights on different portions of the training data (20%, 40%, 60%, 80%, and 100%).

5. Evaluate the Model:

- Evaluate the model on the validation set using accuracy, precision, recall, and F1-score.

6. Plot and Print Results:

- Plot the metrics across different data percentages.

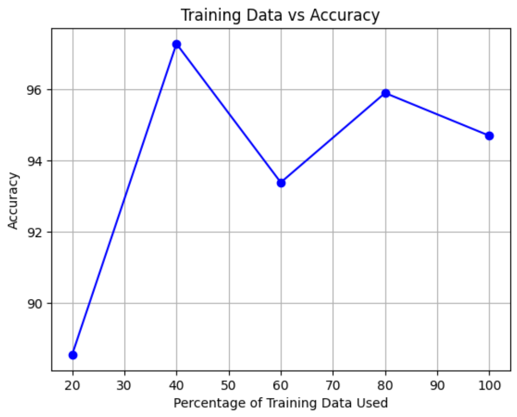
- Print the accuracy, precision, recall, and F1-score for each training percentage.

We implemented a **Neural Network model** combining both **LSTM** and **TF-IDF vectorization** to classify emoticons. The goal was to capture both the sequential nature of the emoticon patterns using LSTM and the frequency-based importance of character combinations through TF-IDF. First, we tokenized the input emoticons at the character level and padded the sequences to ensure uniform input length. We then created an **embedding layer** for the tokenized sequences to represent each emoji as a dense vector. These vector representations were fed into an LSTM layer to learn the temporal relationships between the emojis.

Parallel to the LSTM processing, we applied **TF-IDF Vectorization** to extract n-gram features (unigrams, bigrams, and trigrams) from the same emoji text. These two sets of features—LSTM output and TF-IDF—were concatenated and passed through dense layers to make a final binary classification decision using a **sigmoid activation function**. The model was trained on varying portions of the training data (20%, 40%, 60%, 80%, and 100%) to evaluate how the size of the dataset impacts model performance.

To optimize training and avoid overfitting, we employed **early stopping** by monitoring the validation loss. The results showed a steady improvement in accuracy as more training data was used, with accuracies ranging from 88.56% to 97.28% as the percentage of data increased.

Percentage of Training Data Used	Accuracy (%)
20	88.56
40	**97.28**
60	93.39
80	95.9
100	94.7



Pseudo Code:

1. Load and Preprocess Data:

- Tokenize emojis at the character level.

- Pad sequences to ensure uniform length.

- Apply TF-IDF vectorization on the emoji text.

2. Define Neural Network Model:

- Create an Embedding layer for tokenized emojis.

- Add an LSTM layer to capture sequential patterns.

- Use TF-IDF features in parallel.

- Concatenate LSTM output with TF-IDF features.

- Add dense layers for non-linear learning and a sigmoid output for binary classification.

3. Train the Model:

- Train on varying portions of data (20%, 40%, 60%, 80%, 100%).

- Use early stopping to prevent overfitting.

4. Evaluate the Model:

- Compute accuracy for each portion of training data and display results.

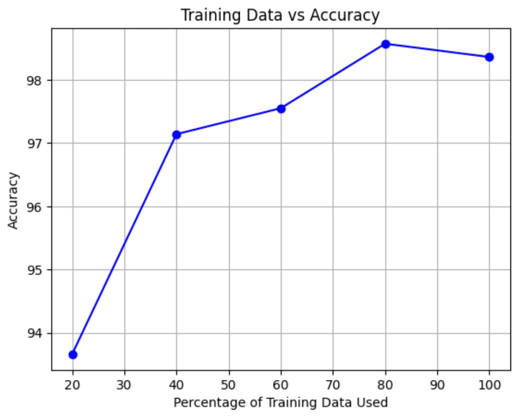
DATA SET – 2

Features

We focused on evaluating how the performance of a logistic regression model improves as we increase the amount of training data. We started by preprocessing the deep feature dataset, which originally had a 13x786 matrix for each sample. These matrices were flattened into 1D vectors (13 * 786 = 10218 features) to simplify the data structure. We then standardized the features using StandardScaler to normalize the data. Since the feature space was quite large, we applied Principal Component Analysis (PCA), reducing the dimensionality to 300 components. This step was crucial to make the model training more efficient without losing significant information.

After preprocessing, we trained the logistic regression model on varying amounts of data: 20%, 40%, 60%, 80%, and 99%. For each percentage, we split the training data accordingly, fit the model, and evaluated it on a fixed validation set. The results showed a clear improvement in accuracy as the training set size increased, from 93.66% with 20% of the data to 98.57% with 80%. However, beyond 80%, the improvement was marginal, suggesting diminishing returns. We also examined precision, recall, and the confusion matrix, which further confirmed that increasing the training data helped the model generalize better, with more accurate predictions and fewer misclassifications.

Percentage of Training Data	Accuracy (%)
20	93.66
40	97.14
60	97.55
80	**98.57**
100	98.36



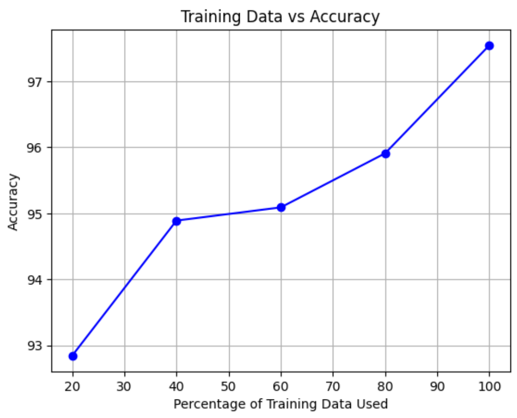
Pseudo Code:

```
1. Load the dataset (train, validation, test).
2. Preprocess the data:
a. Concatenate deep features (13x786 matrix) into a 1D vector (10218 features).
b. Standardize the features using StandardScaler.
c. Apply PCA to reduce dimensionality to 300 components.
3. Define a list of percentages [20%, 40%, 60%, 80%, 99%].
4. For each percentage in the list:
a. Split the training data according to the given percentage.
b. Train a logistic regression model using the reduced PCA features.
c. Predict on the validation set.
d. Calculate accuracy, precision, recall, f1-score, and confusion matrix.
e. Store and print the accuracy and other metrics for each percentage.
5. Plot a graph of accuracy vs percentage of training data.
6. Output a summary table showing the accuracy at each percentage of training data.
```

We used a **Random forest** classifier to evaluate how performance improves as we increase the size of the training data. After applying PCA to reduce the dimensionality of the features, we trained the random forest model on progressively larger subsets of the training data (20%, 40%, 60%, 80%, and 99%).

The model's accuracy improved from 92.84% with 20% of the data to 97.55% with 99% of the data. As more data was used, precision and recall for both classes steadily increased, demonstrating the model's improved ability to generalize and predict accurately with more training examples.

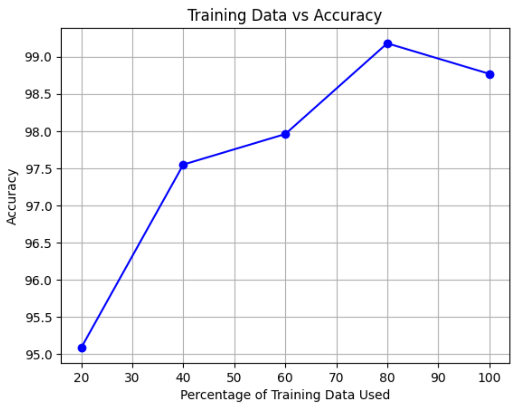
Percentage of Training Data	Accuracy (%)
20	92.84
40	94.89
60	95.09
80	95.91
100	**97.55**



In this experiment, we evaluated the performance of a **Support Vector Machine (SVM)** model using PCA-reduced features with different percentages of training data. The SVM model was trained on progressively larger subsets of the data (20%, 40%, 60%, 80%, and 99%), and we measured the model's accuracy, precision, recall, and f1-scores on a validation set after each step.

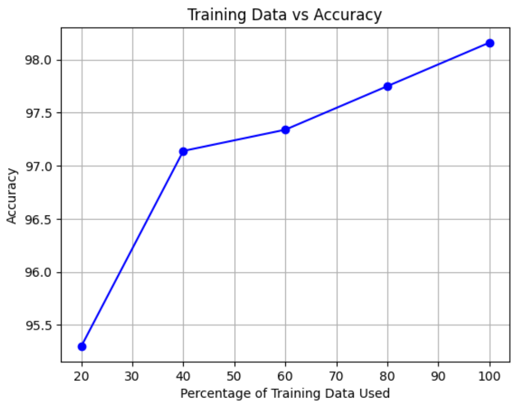
The model achieved an accuracy of 95.09% with 20% of the data, increasing to 99.18% with 80% of the data. The results show that SVM performs well as we increase the size of the training data, but with diminishing returns beyond 80% of the data.

Percentage of Training Data	Accuracy (%)
20	95.09
40	97.55
60	97.96
80	**99.18**
100	98.77



The table shows the accuracy of the **LightGBM model** as the percentage of training data increases from 20% to 99%. The accuracy steadily improves, starting at 95.30% with 20% of the data and reaching 98.16% with 99% of the data, indicating that the model benefits from additional data, though the gains become marginal after 60%.

Percentage of Training Data	Accuracy (%)
20	95.3
40	97.14
60	97.34
80	97.75
100	98.16



DATA SET – 3

We first preprocess text data using `CountVectorizer`, a tool that converts text sequences into numerical representations, specifically at the character level. Each character in the input string is treated as a feature, and we limit the number of features to 1,000, retaining only the 1,000 most frequent characters. This character-level vectorization provides a basic but effective way to convert text data into a format suitable for machine learning.

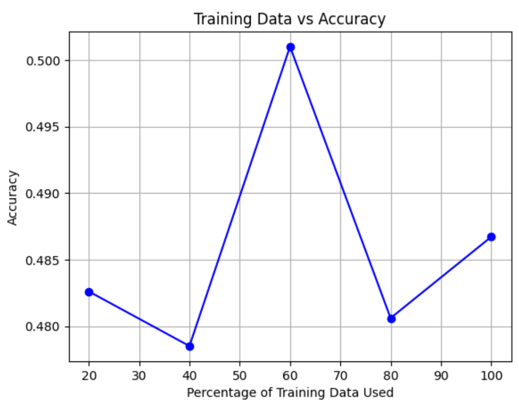
After converting the input text into a sparse matrix of token counts, we use it to train a **Logistic Regression model**, a simple yet powerful linear model for binary classification tasks. Logistic regression predicts probabilities for the binary classes, making it well-suited for text classification problems like this one.

Once the **Logistic Regression model** is trained on the vectorized training data, we use it to make predictions on the validation set. The model's performance is evaluated using the `classification_report`, which calculates precision, recall, F1-score, and accuracy. This provides insight into how well the model performs across different metrics, allowing for a detailed evaluation of the results. Finally, the model's accuracy is visualized across varying percentages of training data to understand the relationship between training data size and model performance.

Pseudocode

```
1. Data Preparation:
- Extract input text sequences (X_train, X_valid) and labels (y_train, y_valid) from the unique training and validation datasets.
2. Vectorization:
- Initialize CountVectorizer for character-level analysis and limit features to the top 1,000 characters.
- Use the vectorizer to convert X_train and X_valid text sequences into sparse matrices of character token counts.
- Convert the sparse matrix of X_train to a dense array for compatibility with some models.
3. Model Training:
- Initialize a Logistic Regression model with a maximum of 100 iterations for convergence.
- Train the Logistic Regression model on the vectorized X_train and y_train.
4. Model Evaluation:
- Use the trained model to make predictions on the vectorized X_valid dataset.
- Generate a classification report to evaluate the model's performance based on precision, recall, F1-score, and accuracy.
5. Accuracy Plotting:
- Use varying percentages of the training data (e.g., 20%, 40%, 60%, etc.) to evaluate model accuracy at different data sizes.
- Plot the relationship between the percentage of training data used and the model's accuracy.
```

Percentage of Training Data	Accuracy
20	0.4826
40	0.4785
60	0.501
80	0.4806
100	0.4867



In this code, we are working on a classification task using XGBoost, where we combine numerical and textual data for model training. The numerical features are processed using PCA to reduce dimensionality, while text data is tokenized at the character level. We first load the datasets, apply PCA for dimensionality reduction on the numerical features, and tokenize the text sequences, padding them to a uniform length. The PCA-transformed features and padded text sequences are concatenated to form the final input features. The model is trained using XGBoost with specific hyperparameters (e.g., `n_estimators=150`, `learning_rate=0.05`), and we experiment with different percentages of the training data (20%, 40%, 60%, 80%, and 100%) to evaluate how much data impacts the validation accuracy.

After training on different subsets of the data, we plot the relationship between the percentage of data used and the validation accuracy. Finally, we evaluate the fully trained model on the validation set, displaying the final accuracy, classification report, and confusion matrix to assess the model's performance. This step ensures we can analyze both the accuracy and detailed performance metrics (precision, recall, F1-score) across different classes.

Pseudo code:
Load and Clean Data:

- Load numerical features (train, validation, test) from .npz files.
- Load text sequences (train, validation, test) from CSV files.

Dimensionality Reduction:

- Apply PCA to reduce dimensionality of numerical features to 250 components.

Feature preprocessing:

- Reshape numerical feature arrays to 2D format.
- Standardize the numerical features using `StandardScaler`.

PCA transformation:

- Fit PCA on scaled training features, transform train, validation, and test sets.

Text preprocessing:

- Initialize tokenizer (char-level).
- Fit tokenizer on training text data.
- Convert text to sequences and pad them to a length of 50 characters.

Concatenate numerical PCA features and text padded sequences:

- Concatenate PCA-transformed features with padded text sequences for train, validation, and test sets.

Define XGBoost model parameters:

- Set XGBoost hyperparameters (e.g., `n_estimators=150`, `learning_rate=0.05`, etc.).

Train and evaluate with different percentages of training data:

- For each percentage (20%, 40%, 60%, 80%, 100%):
    1. Split the training data into subsets according to percentage.
    2. Train the XGBoost model on the subset.
    3. Predict on the validation set.
    4. Calculate validation accuracy and store it.

Plot the validation accuracies against percentages of training data.

Final model training and evaluation:

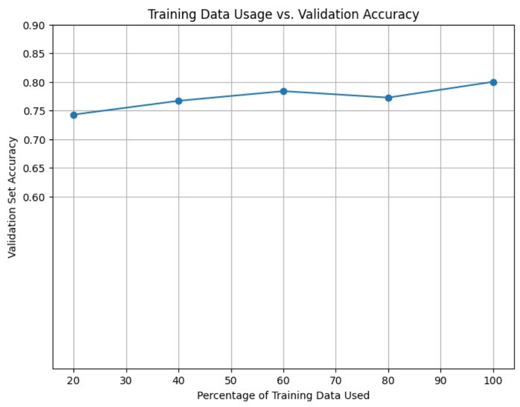
- Train XGBoost on full training data.
- Predict on validation data.

- Print final accuracy, classification report, and confusion matrix.

Plot Results:

- Plot the accuracy against the percentage of training data used.

Percentage	Max Accuracy
20	74.29
40	76.69
60	78.39
80	77.26
100	**80.01**



Task – 2
In this code, we are working on a classification task using XGBoost, where we combine numerical and textual data for model training. The numerical features are processed using PCA to reduce dimensionality, while text data is tokenized at the character level. We first load the datasets, apply PCA for dimensionality reduction on the numerical features, and tokenize the text sequences, padding them to a uniform length. The PCA-transformed features and padded text sequences are concatenated to form the final input features. The model is trained using XGBoost with specific hyperparameters (e.g., `n_estimators=150`, `learning_rate=0.05`), and we experiment with different percentages of the training data (20%, 40%, 60%, 80%, and 100%) to evaluate how much data impacts the validation accuracy.
After training on different subsets of the data, we plot the relationship between the percentage of data used and the validation accuracy. Finally, we evaluate the fully trained model on the validation set, displaying the final accuracy, classification report, and confusion matrix to assess the model's performance. This step ensures we can analyze both the accuracy and detailed performance metrics (precision, recall, F1-score) across different classes.

Performance Results

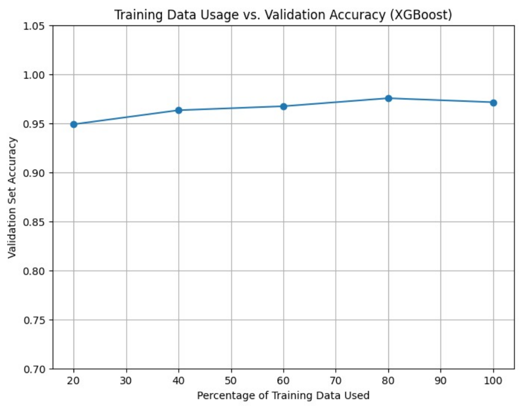
Percentage	Max Accuracy
20%	94.89
40%	96.32
60%	96.73
80%	**97.55**
100%	97.14

Pseudo Code

1. **Load the datasets:**
    - Load numerical features (train, validation, test) from `.npz` files.
    - Load text sequences (train, validation, test) from CSV files.
2. **Dimensionality Reduction:**

- Apply PCA to reduce dimensionality of numerical features to 250 components.

3. **Feature preprocessing:**
    - Reshape numerical feature arrays to 2D format.
    - Standardize the numerical features using `StandardScaler`.
4. **PCA transformation:**
    - Fit PCA on scaled training features, transform train, validation, and test sets.
5. **Text preprocessing:**
    - Initialize tokenizer (char-level).
    - Fit tokenizer on training text data.
    - Convert text to sequences and pad them to a length of 50 characters.
6. **Concatenate numerical PCA features and text padded sequences:**
    - Concatenate PCA-transformed features with padded text sequences for train, validation, and test sets.
7. **Define XGBoost model parameters:**
    - Set XGBoost hyperparameters (e.g., `n_estimators=150`, `learning_rate=0.05`, etc.).
8. **Train and evaluate with different percentages of training data:**
    - For each percentage (20%, 40%, 60%, 80%, 100%):
        (a) Split the training data into subsets according to percentage.
        (b) Train the XGBoost model on the subset.
        (c) Predict on the validation set.
        (d) Calculate validation accuracy and store it.
9. **Plot the validation accuracies against percentages of training data.**
10. **Final model training and evaluation:**
    - Train XGBoost on full training data.
    - Predict on validation data.
    - Print final accuracy, classification report, and confusion matrix.


Training Data Usage vs. Validation Accuracy (XGBoost)
Validation Set Accuracy
Percentage of Training Data Used