CS771: Mini-Project 1

Chaitanya Vishwas Bramhapurikar
230305

Om Ji Gupta
230719

Pranshu Agarwal
230776

Sunij Singh Gangwar
231051

1 Task 1

Let us start with the first part of task 1, i.e. we shall train our model for the dataset involving emoticons.

1.1 Emoticon Dataset

We started off with unicode representation[1] of emojis and svm as our model (well, we have tried other models too but svm is giving us the best accuracy) for learning and predicting labels for the validation set. Given below is the accuracy plot for the same:

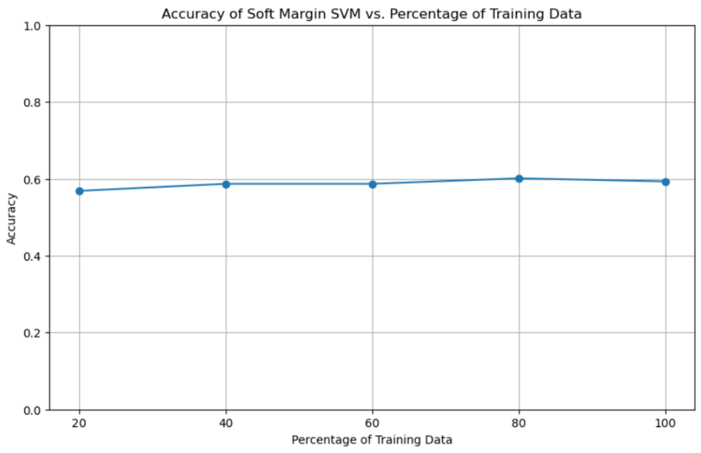

Figure 1: Accuracy plot for unicode representation

We then went on try one-hot encoding of the emoticons. The code snippet of the same is:

```python
unique_emojis = set()
for i in train_emoticon_X:
    unique_emojis.update(list(i))
unique_emojis = list(unique_emojis)

#All the unique emojis collected from each string of emoticons. Next part
     is creating emoji-to-index dictionary
```

[1]the main point is that we had to convert the emojis into some viable representation for the training of our the model, so we started with unicode representation

```

emoji_to_index = {emoji: idx for idx, emoji in enumerate(unique_emojis)}

# One-hot encoding function for a single emoji
def one_hot(emoji):
    encoding = np.zeros(len(unique_emojis), dtype=int)
    if emoji in emoji_to_index:
        index = emoji_to_index[emoji]
        encoding[index] = 1
    return encoding

# We finally need one_hot encoding function for a sequence of 13 emojis
def one_hot_encode_and_concatenate(emoji_sequence):
    concatenated_vector = np.zeros(13 * len(unique_emojis), dtype=int)
    # 13 one-hot vectors concatenated
    for idx, emoji in enumerate(emoji_sequence[:13]):
        one_hot_vector = one_hot(emoji)  # One-hot encode each emoji
        concatenated_vector[idx * len(unique_emojis):(idx + 1) * len(
    unique_emojis)] = one_hot_vector  # Concatenate
    return concatenated_vector

train_X = input_emoticon_X.apply(one_hot_encode_and_concatenate)

# Convert the result into a 2D numpy array for training (batch_size x 13
    * num_unique_emojis)
train_X = np.stack(train_X)

valid_X = valid_emoticon_X.apply(one_hot_encode_and_concatenate)
valid_X = np.stack(valid_X)
```

We have calculated the number of unique emojis as 214 for the training dataset (as well as test dataset) so the size of individual input is $214 * 13 = 2782$ (this will be useful later when we calculate the number of trainable parameters in task 2). On this encoding, we tried different models like DTs, Logistic Regression, SVMs and it lead us to the following accuracy plot:

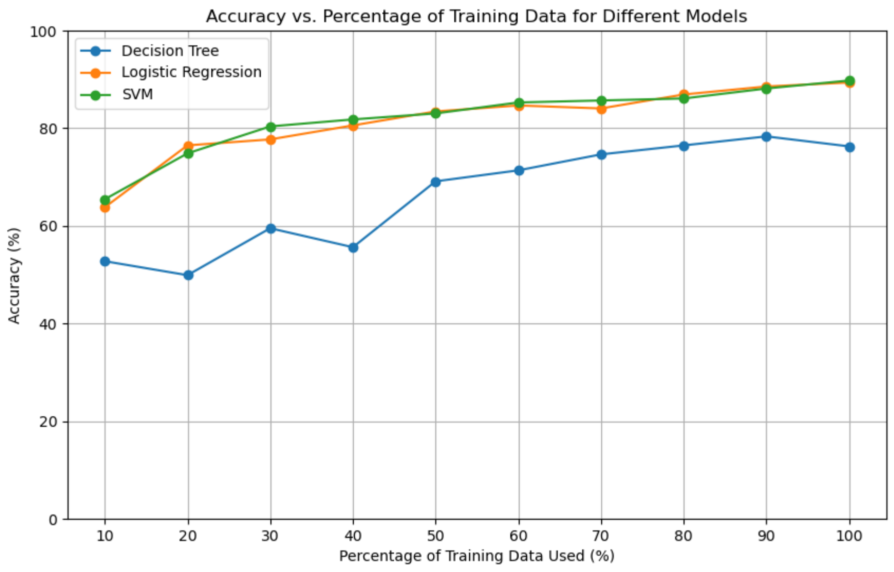

Figure 2: Plot of accuracy vs percentage of training data used

This was the maximum we could achieve using one-hot encoding. As it was allowed to use pre-trained deep learning model as feature extractor, we decided to learn[2] and use them directly for our feature transformation to improve our accuracy! That didn't go quite well (maybe we didn't use suitable neural-based encoding, who knows):

```
    models_to_test = [
    'all-MiniLM-L6-v2',
    'distilbert-base-nli-stsb-mean-tokens',
    'roberta-base-nli-stsb-mean-tokens',
    'bert-base-nli-mean-tokens'
    ]
```

Encoding the dataset emoticons using the first one in the above models gave us the graph given below:

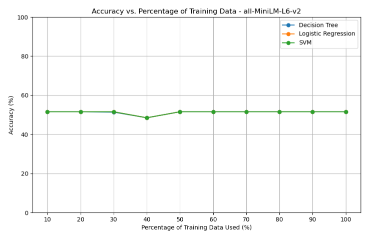

Figure 3: Using all-MiniLM-L6-v2 sentence transformer

We also tried KNN and LwP based on these transformers but their accuracy also wasn't good anyway. This wasn't nearly enough. We then tried DTs, Logistic Regression, SVM, etc for the distilbert-base-nli-stsb-mean-tokens neural-based transformer and other ones as well but they didn't yield any plots with decent accuracy. We realized later that the extracted features from transformers face lot of collisions i.e. for two different sequences of emoticons we get the same encoding, which results in bad accuracy. We also tried RNN and CNN for feature extraction but they also gave nearly 50% accuracy

1.2 Deep Features Dataset

For this dataset, we decided to use Averaging (which was mentioned in the question itself), Max Pooling, Concatenation, and a neural based feature extractor (CNN). For each of these encodings, we tried DTs, Logistic Regression, SVM and other models.

```
# 1. Averaging Embeddings
train_feat_X_avg = np.mean(train_feat_X, axis=1)
valid_feat_X_avg = np.mean(valid_feat_X, axis=1)
accuracies_avg = evaluate_decision_tree(train_feat_X_avg,
    valid_feat_X_avg)

# 2. Concatenation of Embeddings
train_feat_X_concat = train_feat_X.reshape(train_feat_X.shape[0], -1)
valid_feat_X_concat = valid_feat_X.reshape(valid_feat_X.shape[0], -1)
accuracies_concat = evaluate_decision_tree(train_feat_X_concat,
    valid_feat_X_concat)

# 3. MaxPooling of Embeddings
train_feat_X_maxpool = np.max(train_feat_X, axis=1)
valid_feat_X_maxpool = np.max(valid_feat_X, axis=1)
accuracies_maxpool = evaluate_decision_tree(train_feat_X_maxpool,
    valid_feat_X_maxpool)
# 4. Standard CNN-based Embedding Extraction
```

[2]Hugging face'sdocumentation was of big help

```
def build_cnn_model():
    model = models.Sequential()
    model.add(layers.Conv1D(filters=32, kernel_size=3, activation='relu',
     input_shape=(13, 768)))
    model.add(layers.MaxPooling1D(pool_size=2))
    model.add(layers.Conv1D(filters=64, kernel_size=3, activation='relu')
    )
    model.add(layers.MaxPooling1D(pool_size=2))
    model.add(layers.Flatten())
    return model

cnn_model = build_cnn_model()
cnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[
    'accuracy'])

# Get CNN features
train_feat_cnn = cnn_model.predict(train_feat_X)
valid_feat_cnn = cnn_model.predict(valid_feat_X)
accuracies_cnn = evaluate_decision_tree(train_feat_cnn, valid_feat_cnn, "
    CNN")
```

As the number of trainable parameters in CNN exceed the limit, we have not included it in the final code.

1.2.1 Decision Tree

We started off with the decision tree which gave us below plot:

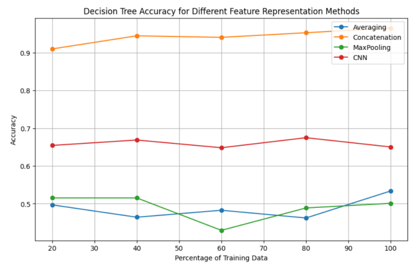

Figure 4: Decision Tree accuracy based on different feature extraction methods.

1.2.2 Logistic Regression

We then went on to try Logistic Regression which gave us the following plot:

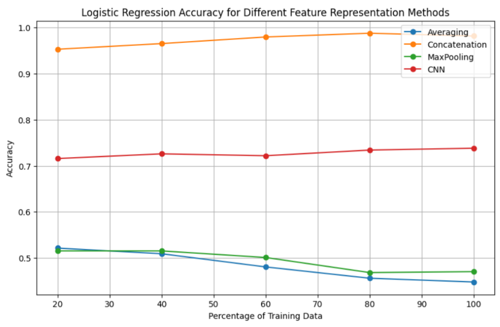

Figure 5: Logistic Regression accuracy based on different feature extraction methods.

1.2.3 Support Vector Machine

The plot for support vector machine is given below:

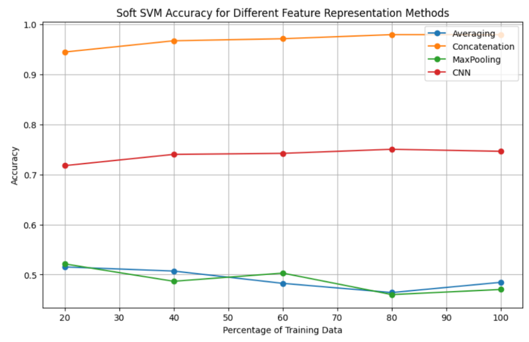

Figure 6: Soft Margin SVM accuracy based on different feature extraction methods.

1.2.4 Final Code

We observe that concatenation is giving us maximum accuracy across all the feature extraction methods. We have combined the plots of all the models for concatenation below:

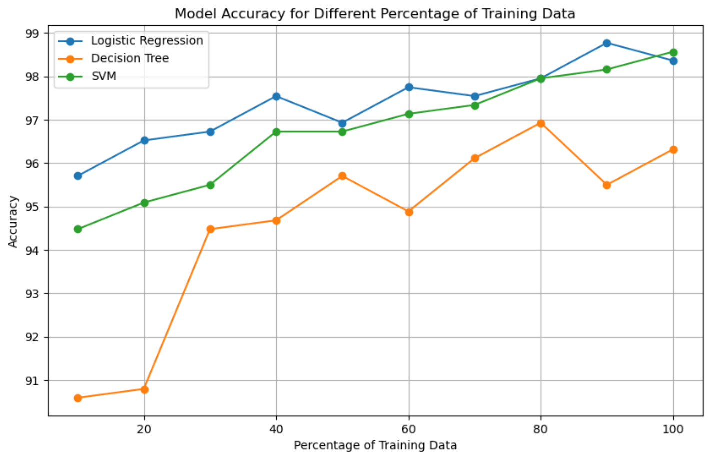

Figure 7: Accuracies of different models for concatenation

It can be seen from the above figure that the logistic regression is performing relatively better than the other models. The code snippet for logistic regression is given below:

```python
percentages = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

def evaluate_LR(train_feat_X_method, valid_feat_X_method, method_name):
    accuracies = []
    for percentage in percentages:
        n_samples = int(len(train_feat_X_method) * percentage)
        x_train_subset = train_feat_X_method[:n_samples]
        y_train_subset = train_feat_Y[:n_samples]
        LR_classifier = LogisticRegression(random_state=42)
        LR_classifier.fit(x_train_subset, y_train_subset)

        # Predict on the validation set
        y_pred = LR_classifier.predict(valid_feat_X_method)

        # Calculate accuracy
        accuracy = accuracy_score(valid_feat_Y, y_pred)
        accuracies.append(accuracy)

    return accuracies
# Concatenation of Embeddings
train_feat_X_concat = train_feat_X.reshape(train_feat_X.shape[0], -1)
valid_feat_X_concat = valid_feat_X.reshape(valid_feat_X.shape[0], -1)
accuracies_concat = evaluate_LR(train_feat_X_concat, valid_feat_X_concat,
     "Concatenation")
```

The corresponding plot is shown below:

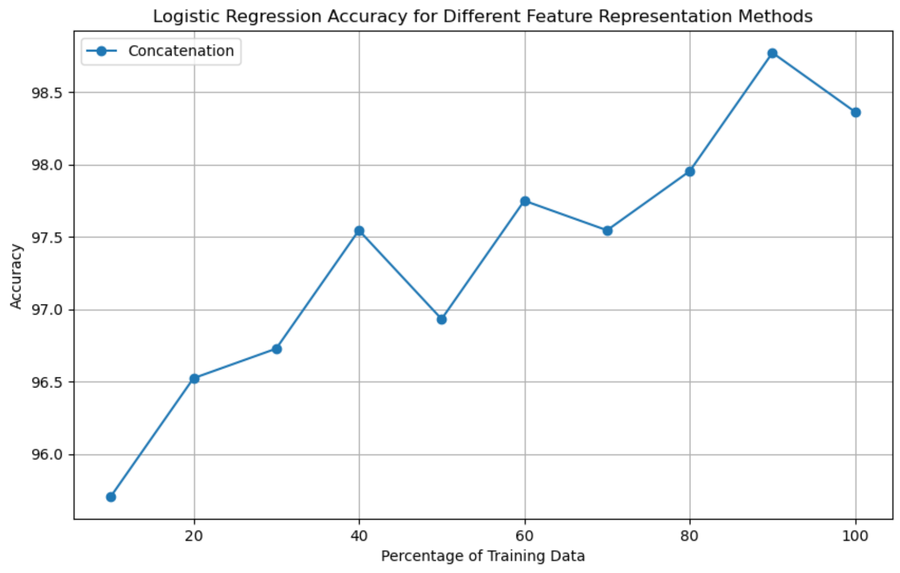

Figure 8: Logistic Regression accuracy with concatenation feature extractor.

1.3 Text Sequence Dataset

We decided to first convert the string into a sequence of integers, rather we could have directly passed string as an input parameter in the model but these models generally work with numerical data inputs, so changing it into a sequence of integers is a better choice.

```
# Preprocessing: Convert input strings to sequences of integers
max_length = 50
train_seq_X = [[int(digit) for digit in seq] for seq in train_seq_X]
valid_seq_X = [[int(digit) for digit in seq] for seq in valid_seq_X]
# Preprocess validation data
train_seq_X = np.array(train_seq_X)
valid_seq_X = np.array(valid_seq_X)

#Convert it into numpy array
train_seq_Y = np.array(train_seq_Y)
valid_seq_Y = np.array(valid_seq_Y)

# Define CNN Model architecture with reduced parameters
def build_cnn_model(input_shape):
    model = Sequential()
    model.add(Embedding(input_dim=10, output_dim=16, input_length=input_shape[1]))  # Reduced embedding dimension
    model.add(Conv1D(16, kernel_size=3, activation='relu'))  # Reduced number of filters
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(16, activation='relu'))  # Reduced dense layer size
    model.add(Dropout(0.3))
    model.add(Dense(1, activation='sigmoid'))  # For binary classification
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

# Function to train models and record validation accuracy
def train_model_on_subsets(model_func, model_type, X_train, y_train, X_val, y_val):
    percentages = [0.2, 0.4, 0.6, 0.8, 1.0]
    accuracies = []

    for p in percentages:
        subset_size = int(len(X_train) * p)
        X_train_subset = X_train[:subset_size]
        y_train_subset = y_train[:subset_size]

        if model_type == "CNN":
            # Build and train the CNN model
            model = model_func(X_train_subset.shape)
            model.fit(X_train_subset, y_train_subset, epochs=10, batch_size=64, verbose=0)
            y_val_pred = (model.predict(X_val) > 0.5).astype("int32")
            val_accuracy = accuracy_score(y_val, y_val_pred)
        else:
            # Train traditional machine learning models
            model = model_func()
            model.fit(X_train_subset, y_train_subset)
            y_val_pred = model.predict(X_val)
            val_accuracy = accuracy_score(y_val, y_val_pred)

        accuracies.append(val_accuracy)

    return percentages, accuracies

# Different models we have used
models = {
    "Logistic Regression": lambda: LogisticRegression(max_iter=1000),
    "kNN": lambda: KNeighborsClassifier(n_neighbors=5),
    "Decision Tree": lambda: DecisionTreeClassifier(),
    "Soft Margin SVM": lambda: SVC(kernel='linear', C=1),
    "CNN": build_cnn_model
}
```

Initially, the number of trainable parameters exceeded the 10,000 trainable parameters constraint. We then reduced the number of filters and dense layer size to satisfy the constraint.

1.3.1 Further Exploration

We decided to use neural-based feature extractors with the hope of achieving better accuracy.

```
def build_embedding_model():
    model = tf.keras.Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=max_length))
    model.add(GlobalAveragePooling1D())
```

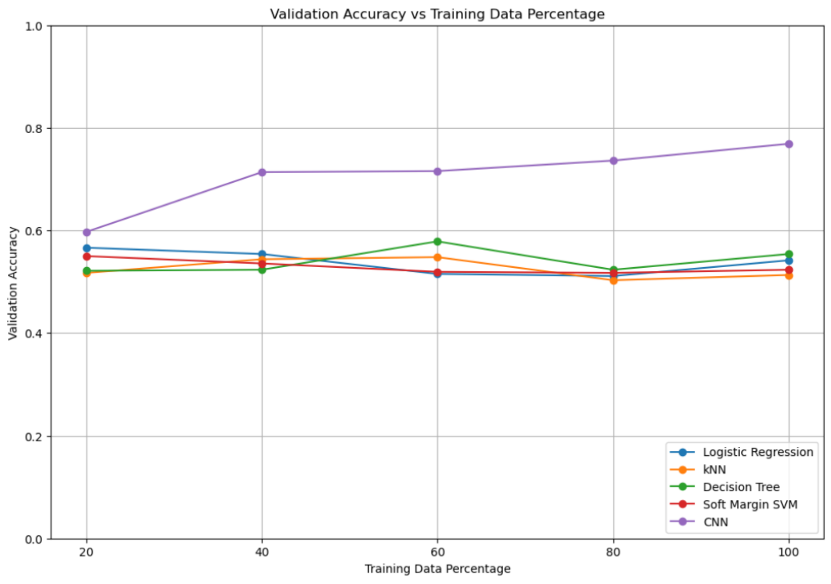

Figure 9: Final Plot of accuracies for different models.

```
    return model

def build_cnn_model():
    model = tf.keras.Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim,
    input_length=max_length))
    model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
    model.add(GlobalAveragePooling1D())
    return model

def build_rnn_model():
    model = tf.keras.Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim,
    input_length=max_length))
    model.add(SimpleRNN(64))
    return model

def build_lstm_model():
    model = tf.keras.Sequential()
    model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim,
    input_length=max_length))
    model.add(LSTM(64))
    return model
```

For the above feature extractors, we got the following accuracy plot for different models:

1.3.1.1. Decision Tree

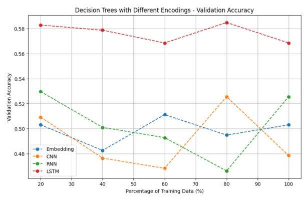

Figure 10: Decision Tree accuracy for different neural network-based feature extractors.

1.3.1.2. Logistic Regression

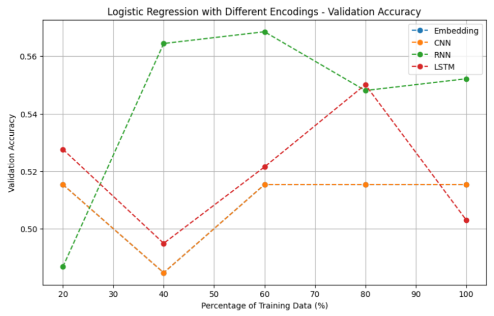

Figure 11: Logistic Regression accuracy for different neural network-based feature extractors.

1.3.1.3. Support Vector Machine

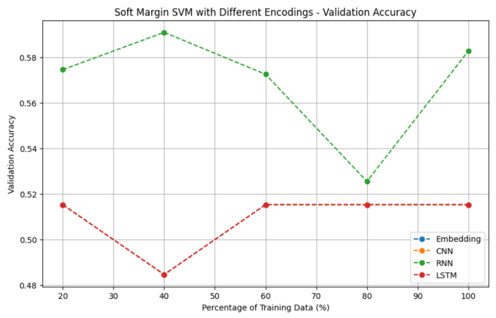

Figure 12: SVM accuracy for different neural network-based feature extractors.

2 Task 2

As we can see that in second part of task 1, we have achieved an accuracy of approximately 98.5%, so when we combine all the datasets from all the parts, we hope to get better accuracy than this. We tried to concatenate all the dataset inputs into one. The input size of first datset is 2782, second dataset is 9984, third dataset is 50 which in total exceeds our constraint of trainable parameters (1000). So we have used PCA to reduce the dimensionality of the second dataset to 7000.

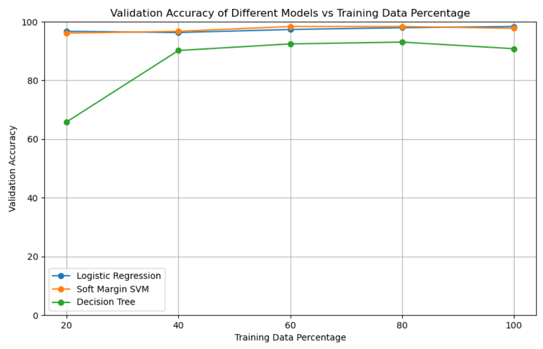

Figure 13: Accuracy plot for different models when the dimensionality of second dataset is 7000

The code snippet for the above plot is:

```
# Prepare the third dataset: digit encoding
max_length = 50
train_seq_X_encoded = [[int(digit) for digit in seq] for seq in
    train_seq_X]
train_seq_X_encoded = np.array(train_seq_X_encoded)

valid_seq_X_encoded = [[int(digit) for digit in seq] for seq in
    valid_seq_X]
valid_seq_X_encoded = np.array(valid_seq_X_encoded)

# Feature Transformation
# One-Hot Encoding for Emoticons
encoder = OneHotEncoder(sparse_output=False, handle_unknown = 'ignore')
train_emoticon_X_encoded = encoder.fit_transform(np.array(
    train_emoticon_X).reshape(-1, 1))
valid_emoticon_X_encoded = encoder.transform(np.array(valid_emoticon_X).
    reshape(-1, 1))


# Flatten the feature matrix
train_feat_X_reshaped = train_feat_X.reshape(train_feat_X.shape[0], -1)

# Flatten the feature matrix
valid_feat_X_reshaped = valid_feat_X.reshape(valid_feat_X.shape[0], -1)

# Apply PCA to reduce deep features to 7,000 features
pca = PCA(n_components=7000)  # Adjust number of components as needed
train_feat_X_reduced = pca.fit_transform(train_feat_X_reshaped)
valid_feat_X_reduced = pca.transform(valid_feat_X_reshaped)

# Concatenate Deep Features (after PCA) with One-Hot Encoded Features
combined_train_features = np.concatenate((train_feat_X_reduced,
    train_emoticon_X_encoded), axis=1)
combined_valid_features = np.concatenate((valid_feat_X_reduced,
    valid_emoticon_X_encoded), axis=1)

# Concatenate the Combined Features with Digit Encoded Features
final_train_features = np.concatenate((combined_train_features,
    train_seq_X_encoded), axis=1)
final_valid_features = np.concatenate((combined_valid_features,
    valid_seq_X_encoded), axis=1)


final_train_labels = train_seq_Y
final_valid_labels = valid_seq_Y

# Define models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Soft Margin SVM": SVC(kernel='linear', C=1),
    "Decision Tree": DecisionTreeClassifier()
}

# Store results for plotting
percentages = [0.2, 0.4, 0.6, 0.8, 1.0]  # Training percentages
results = {model_name: [] for model_name in models.keys()}  # Initialize
    results

# Train on subsets of the training data
for p in percentages:
    subset_size = int(len(final_train_features) * p)
    X_train_subset = final_train_features[:subset_size]
    y_train_subset = final_train_labels[:subset_size]
```

```python

    for model_name, model in models.items():
        model.fit(X_train_subset, y_train_subset)

        # Evaluate on the validation set
        y_val_pred = model.predict(final_valid_features)  # Use the validation features
        val_accuracy = accuracy_score(final_valid_labels, y_val_pred)

        results[model_name].append(val_accuracy*100)

# Plotting the results
plt.figure(figsize=(10, 6))
for model_name, accuracies in results.items():
    plt.plot([p * 100 for p in percentages], accuracies, marker='o', label=model_name)

plt.title("Validation Accuracy of Different Models vs Training Data Percentage")
plt.xlabel("Training Data Percentage")
plt.ylabel("Validation Accuracy")
plt.xticks([20, 40, 60, 80, 100])
plt.ylim(0, 100)  # Set y-axis limits for clarity
plt.grid()
plt.legend()
plt.show()
```

This PCA uses the entire dataset to extract features and reduce dimensionality of the deep features dataset, we have found the accuracy of logistic regression to be the highest across all the 3 models mentioned above, which is 98.16% which was already not a marginal improvement over our original accuracy for the second dataset, so it can be inferred that on applying PCA on incremental fractions of data will definitely give even lesser accuracy.

We have tried above approaches but we coudn't get any marginal improvement over accuracy obtained in deep features dataset. The second dataset seems to be the most accurate representation of the distribution on which the all the datasets are based. Therefore, we will use the deep features dataset to make predictions on test data.

References

- Used the sk-learn documentation for classification.
- Read about Sentence Transformers from Hugging Face.
- Learnt about tensor flow from its documentation.
- Took help of chat-gpt in debugging codes.