Mini Project-1 Group No. 32

Abhijeet Kumar(220028)
Vikash Kumar(211168)
Himanshu Gupta(220453)
Manikya Pathak(231290403)
Gautam Hinduja(200384)

October 20, 2024

Our project explores various machine learning techniques applied to different datasets. Here, we describe in detail the LSTM model that we developed for the **emoticon dataset**.

1. First Model Used on the Emoticon Dataset(EmojiLSTMModel)

This is the first model applied to the emoticon dataset for classification purposes. The unique nature of emoticons as sequential data makes this a novel approach, particularly in exploring how LSTM networks can capture the underlying patterns in emoticon sequences.

Model Overview:

The core objective of this model is to classify emoticon sequences into predefined labels, using a Long Short-Term Memory (LSTM) network. LSTMs are particularly suited for sequential data due to their ability to capture dependencies over time, making them ideal for emoticon sequence classification.

The model is designed using the following hyperparameters:

- **Maximum Sequence Length:** 10 – The input emoticon sequences are padded or truncated to have a uniform length of 10.
- **Embedding Dimension:** 20 – This defines the size of the dense vector representation for each emoticon.
- **LSTM Units:** 16 – The number of memory units in the LSTM layer.
- **Dense Units:** 8 – The number of neurons in the fully connected dense layer before the output layer.

Dataset Preprocessing:

The preprocessing steps are crucial for feeding data into the LSTM network. Emoticons are non-alphanumeric characters, so we convert them into integer sequences that the model can process.

Step 1: Creating a Mapping for Emoticons to Integers

Each unique emoticon is assigned an integer value using a dictionary. For example, an emoticon might be mapped to an integer 1, while another might map to 2. This creates a lookup table for encoding emoticons into a numeric format.

Step 2: Encoding and Padding the Sequences

The emoticon sequences are encoded using the above mapping and then padded to a uniform length of 10. Padding ensures that all input sequences have the same length, which is necessary for the LSTM model.

Padding: Sequences shorter than 10 are padded with zeros, while sequences longer than 10 are truncated. This ensures that the model can process input efficiently.

Example:

$$\text{Sequence:}[,,] \quad \Rightarrow \quad \text{Encoded:}[1,2,3,0,0,0,0,0,0,0]$$

Model Architecture:

The architecture follows a standard LSTM pipeline with embedding, recurrent, and fully connected layers.

Step 1: Embedding Layer

This layer converts each integer-encoded emoticon into a dense vector of size 20 (the embedding dimension). The embedding helps the model learn relationships between emoticons, as similar emoticons are likely to have similar vector representations.

Step 2: LSTM Layer

The LSTM layer, consisting of 16 units, processes the sequence of embeddings. LSTMs are chosen because they can handle sequential dependencies. Each unit maintains a hidden state and updates it through time steps, making them ideal for sequence-based data such as emoticons.

Step 3: Dropout Layer

A dropout layer with a rate of 0.5 is added after the LSTM layer to reduce overfitting. Dropout randomly turns off 50% of the neurons during training, forcing the model to generalize better.

Step 4: Flatten Layer

The output from the LSTM layer is 2D (timesteps and features), and we flatten this output into a single vector, so it can be passed to the fully connected layers.

Step 5: Dense Layer

The first dense layer with 8 units and ReLU activation compresses the learned features into a more compact form, helping the model focus on the most relevant information.

Step 6: Output Layer

The output layer uses softmax activation, producing probabilities for each class (emoticon category). It outputs the predicted class based on the highest probability.

Training and Optimization:

The model is trained using the **Sparse Categorical Crossentropy** loss function, which is suitable for multi-class classification problems with integer labels. The **Adam optimizer** is used to minimize the loss, as it is effective in handling complex optimization landscapes.

Training Details:

- **Epochs:** 10 – The model is trained over 10 iterations on the training data.
- **Batch Size:** 32 – Training is done in batches of 32 samples at a time to balance memory usage and learning stability.
- **Metrics:** Accuracy is used to evaluate the performance during training and validation.

Validation and Evaluation:

The validation dataset is used to monitor the model's performance during training. By observing the validation loss and accuracy, we ensure that the model does not overfit to the training data.

After training, the model achieved a validation accuracy of **89.78%**.

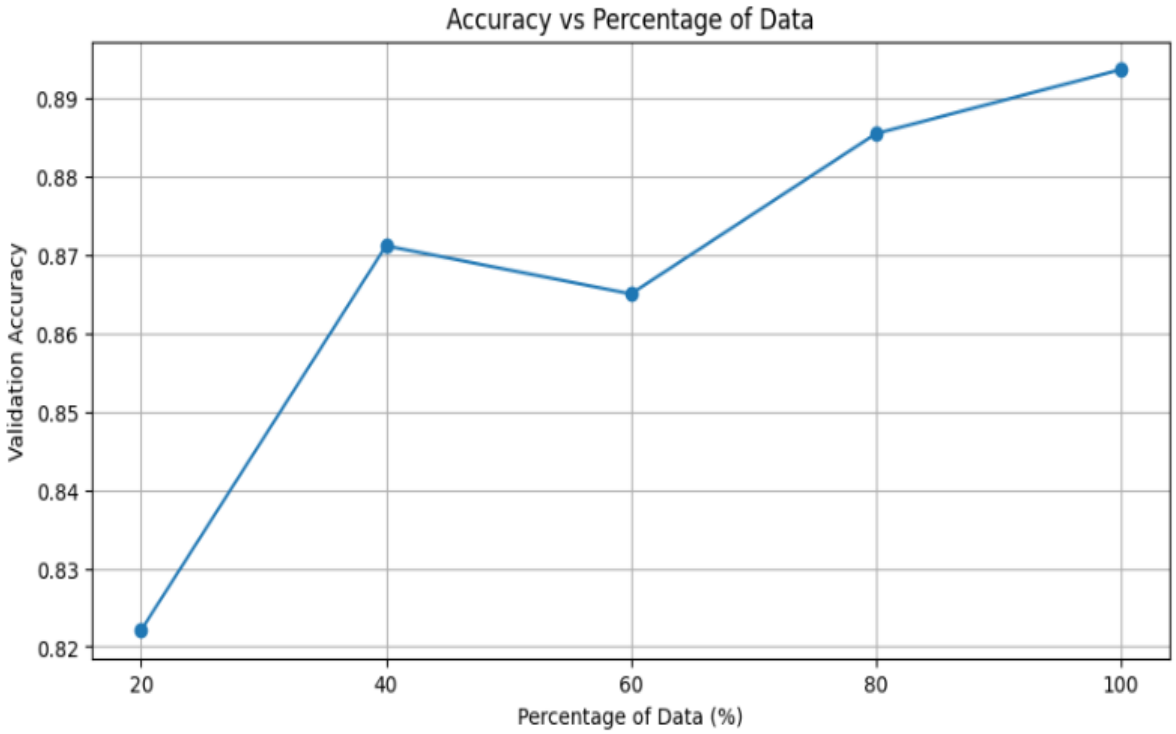

Figure 1: Validation accuracy graph

Classification Report: The model achieved an overall accuracy of 90

- Class 0: Precision = 0.92, Recall = 0.87, F1-Score = 0.90
- Class 1: Precision = 0.87, Recall = 0.92, F1-Score = 0.90

The macro average and weighted average for the precision, recall, and F1-score are all 0.90, indicating balanced performance across both classes.

Model Size: The total number of trainable parameters in the model is **7974**.

Predictions:

For testing, new unseen emoticon sequences are passed through the model. The steps include:

- Loading the test dataset.
- Encoding the emoticon sequences as done in the training phase.
- Padding them to a length of 10.
- Predicting the output label using the trained LSTM model.

The predicted labels are then compared with the actual labels to measure performance.

Feature Transformation and Encoding:

- **Emoji-to-Integer Encoding:** Each emoji is represented by a unique integer. This transformation allows for the direct input of emojis into the LSTM.
- **Padding:** Ensures all input sequences are of uniform length, which is required for batch processing in LSTMs.

Improvements and Future Work:

While the model shows promising results with sequential emoji data, some areas for improvement include:

- **Hyperparameter Tuning:** The embedding size, number of LSTM units, and dense layer size can be fine-tuned to improve performance.
- **More Complex Architectures:** Exploring deeper LSTM or Bidirectional LSTM networks could capture even more nuanced relationships in the emoji sequences.
- **Regularization:** Additional regularization techniques like L2 weight decay could help in reducing overfitting further.

In future work, this model can be combined with other datasets to explore the effects of transfer learning or domain adaptation.

2.Second Model Used on Deep Feature Dataset(LogisticRegressionModel)

The aim of this project was to train a Logistic Regression model to classify data from the `deep_feature` dataset into two classes. The performance of the model was evaluated using different portions of the training dataset.

Data Preprocessing

The data was loaded from `.npz` files for training, validation, and testing. The features were flattened and normalized using `StandardScaler` to ensure all features were on a similar scale.

Model Training

A Logistic Regression model was trained using varying percentages of the training data (20%, 40%, 60%, 80%, and 100%). The model was configured with a maximum of 1000 iterations to ensure convergence.

The validation accuracy for each training percentage is shown in Table 3.

Model Complexity

The Logistic Regression model has a total of $N + 1$ trainable parameters, where N is the number of features. The additional parameter accounts for the bias term. Total trainable parameters are **9985**.

Test Predictions

The model was tested on the test set after training. The predictions were saved to a file named `pred_deepfeat_class.txt`.T he model accuracy was **98.36%**

Conclusion

The model's performance improved as the size of the training data increased. Logistic Regression provided a reasonable solution for this binary classification problem, but further improvements could be made by exploring more complex models or tuning hyperparameters.

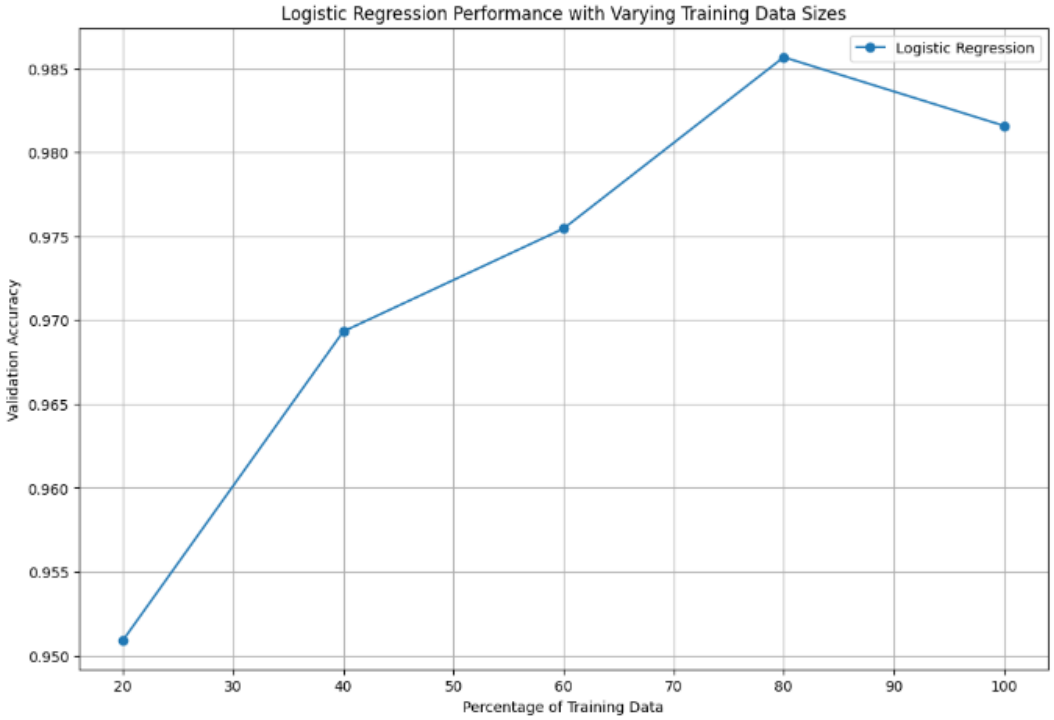

Figure 2: Logistic model performance across varying percentages of training data

Training Percentage	Validation Accuracy
20%	95.09%
40%	96.93%
60%	97.55%
80%	98.57%
100%	98.16%

Figure 3: Validation accuracy at different training percentages

3. Third Model Used on Text Sequence Dataset(CNNLSTMModel)

The `CNNLSTMModel` class implements a Convolutional Neural Network (CNN) combined with Long Short-Term Memory (LSTM) layers for the classification of digit sequences. This model is designed to process sequences of digits, which can be useful in various tasks such as sentiment analysis or numerical data classification.

Model Architecture

The architecture of the CNN-LSTM model consists of the following layers:

- **Embedding Layer:** Converts the input digit sequences into dense vector representations.
- **Convolutional Layers:**
  - The first convolutional layer uses `filters_1` (default: 32) filters with a kernel size of 3 to extract local features from the sequences.
  - A second convolutional layer applies `filters_2` (default: 8) filters with the same kernel size.
- **Max Pooling Layers:** Reduce the dimensionality of the feature maps, allowing the model to focus on the most significant features.

- **Dropout Layers:** Introduced after pooling layers to prevent overfitting by randomly setting a fraction (default: 0.5) of input units to 0 during training.
- **Flatten Layer:** Converts the 2D feature maps into a 1D vector.
- **Dense Layers:**
  - A fully connected layer with `dense_units` (default: 64) units and ReLU activation.
  - An output layer with a sigmoid activation function for binary classification.

Data Loading and Transformation

The `load_data` method is responsible for loading and transforming the training and validation datasets. The following steps outline the data processing pipeline:

1. **Loading Datasets:** The method reads the training and validation datasets from specified CSV files using the `pandas` library, which enables efficient handling of tabular data.
2. **Encoding Sequences:** The digit sequences in the `input_str` column are converted to lists of integers. A lambda function is applied to each row in the DataFrame to achieve this transformation, converting each character in the string to an integer.
3. **Padding Sequences:** The method utilizes the `pad_sequences` function from Keras to ensure that all sequences have the same length. This is essential for neural network models, which require input data to be of uniform shape. The `maxlen` parameter is set to `self.max_sequence_length`, and padding occurs at the end of the sequences (`padding='post'`).
4. **Defining Labels:** The labels corresponding to the training and validation datasets are extracted from the respective DataFrames, serving as the target variable for classification.
5. **Encoding Labels:** A `LabelEncoder` is employed to convert the categorical labels into integers, facilitating the training process by making the labels suitable for model input.
6. **Returning Data:** The method concludes by returning the padded input sequences and encoded labels for both training and validation datasets, preparing them for model training.

Methods

The `CNNLSTMModel` class includes several key methods:

- `load_data(train_data_path, val_data_path)`: Loads training and validation datasets, encodes digit sequences, and pads them to a uniform length.
- `build_model()`: Constructs the CNN-LSTM architecture and compiles the model using binary cross-entropy loss and the Adam optimizer.
- `train(X_train, y_train_encoded, X_val, y_val_encoded, epochs=10, batch_size=16)`: Trains the model on the training data and validates it on the validation data.
- `evaluate(X_val, y_val_encoded)`: Evaluates the model's performance on the validation dataset and prints the accuracy.
- `save_model(model_path)`: Saves the trained model to a specified file path.
- `load_saved_model(model_path)`: Static method to load a previously saved model.
- `predict(model, test_data_path, output_file_path)`: Loads test data, predicts labels for the input sequences, and saves the predictions to a specified output file.

Conclusion

The model accuracy achieved **82.82%**. The `CNNLSTMModel` class provides a robust framework for classifying sequences of digits using a hybrid CNN-LSTM approach. It integrates data preprocessing, model training, evaluation, and prediction functionalities in a cohesive manner, making it suitable for various sequence classification tasks.

4. COMBINED MODEL

The `COMBINED_MODEL` class implements an ensemble approach that combines three different machine learning models: LSTM, Logistic Regression, and CNN-LSTM. The architecture is designed to handle various data types and leverage their strengths for improved prediction accuracy.

Data Loading

The training, validation, and test datasets are loaded using `pandas` and `numpy`. The training data consists of both textual and emoji features:

- **Emoji Dataset:** The emoji dataset is loaded from a CSV file, and emoticons are encoded into integers. The sequences are then padded to ensure uniformity across inputs.
- **Feature Datasets:** Additional features for the Logistic Regression model are loaded from a `.npz` file. These features are reshaped to be compatible with the model.
- **Text Sequences:** Input strings are converted from digit sequences into lists of integers and padded for the CNN-LSTM model.

Model Training

The training process involves fitting each of the three models separately:

- **LSTM Model:** The LSTM model is loaded from a pre-trained `.h5` file and compiled. It is then trained on the padded emoji sequences.
- **Logistic Regression Model:** A Logistic Regression model is trained using the flattened feature arrays obtained from the training dataset.
- **CNN-LSTM Model:** Similar to the LSTM model, the CNN-LSTM model is loaded and trained on the padded text sequences.

Prediction Making

For prediction, the class uses separate validation datasets for each model. The predictions are generated as follows:

- **LSTM Predictions:** Predictions are obtained by passing the validation emoji sequences through the LSTM model.
- **Logistic Regression Predictions:** The validation features are flattened and fed into the Logistic Regression model for predictions.
- **CNN-LSTM Predictions:** The validation text sequences are passed through the CNN-LSTM model to obtain binary predictions.

Combining Predictions

The predictions from the three models are combined using a weighted voting scheme:

- **Accuracy Calculation:** The accuracy for each model is calculated based on the validation data.
- **Weight Normalization:** Each model's accuracy is normalized to compute weights that reflect their performance.
- **Weighted Voting:** For each set of predictions, a weighted sum is calculated using the respective weights. The final prediction is determined by whether this sum exceeds a threshold (0.5).

Test Predictions

Finally, the ensemble model is evaluated on a test dataset, and predictions are saved for future analysis.
The validation accuracy for LSTM Model **89.78**
for Logistic Model **98.36**
for CNN-LSTM Model **82.82**
for Combined Model **94.27**
.
Hence the best model is logistic model which was trained on second dataset deep feature.