# CS771 : Mini-Project 1 - Binary Classification

**Group No: 72**

**Pankaj Nath**
221188

**Manikanta**
220409

**Prashant**
220803

**Sai Nikhil**
221095

**Venkatesh**
220109

## 1 Introduction

This mini-project focuses on developing and evaluating binary classification models for three distinct datasets derived from an original raw dataset. Each dataset represents the same underlying data but with different feature representations. Our goal is to identify the best-performing model based on validation accuracy while also considering the amount of training data used.

### 1.1 Project Objectives

The project consists of two main tasks:

- **Task 1:** Developing individual binary classification models for each dataset, exploring different model architectures and training strategies.
- **Task 2:** Creating a unified model by combining the datasets to potentially leverage complementary information from different feature representations.

## 2 Dataset Description

The three datasets provided represent the same underlying data in different formats. Each dataset has unique characteristics that influence the choice of modeling techniques:

### 2.1 Emoticons as Features Dataset

This dataset contains 13 categorical features where each input is represented by emoticons. Due to its categorical nature, this representation requires careful preprocessing to ensure effective model training.

### 2.2 Deep Features Dataset

In this dataset, each input is represented as a $13 \times 786$ matrix of embeddings. This format provides a rich but complex feature space for classification, necessitating advanced modeling techniques to capture intricate patterns.

### 2.3 Text Sequence Dataset

The features in this dataset are represented as strings of 50 digits, offering a sequential representation of the data. This format allows for the application of sequence-based models that can effectively handle ordered data.

## 3 Task 1: Developing Models for Individual Datasets

### 3.1 Emoticons Dataset

#### 3.1.1 Data Processing

Two data preprocessing strategies were employed to convert the emoticon sequences into formats suitable for training machine learning models:

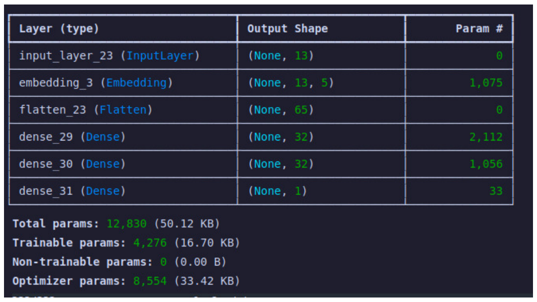
1. **Strategy I:**
   - In this approach, each sequence of 13 emoticons was split into 13 individual columns.
   - One-hot encoding was then applied to this $7080 \times 13$ matrix, resulting in a sparse $7080 \times 2195$ matrix.
   - Various binary classification models were trained on this transformed dataset.
2. **Strategy II:**
   - Here, instead of splitting the emoticon string, spaces were inserted between the emoticons, and tokenization was performed using a TensorFlow tokenizer.
   - The tokenizer assigned unique numerical values to each emoticon, producing a $7080 \times 13$ matrix.
   - which was then used to train our custom neural network.

We have also performed data mining to extract useful information from the training data and obtained the plots of label distribution and Emoticon frequency by label.

#### 3.1.2 Model Training and Evaluation

For the classification task, several machine learning models were implemented, including:

1. Custom Neural Network A deep learning model was constructed with an embedding layer followed by two dense layers. The network used ReLU activation functions and a softmax output layer. The model was compiled and fitted with the following hyperparameters:
   - **Learning rate:** 0.01
   - **Batch size:** 32
   - **Number of epochs:** 5
   - **Optimizer:** Adam
   - **Loss function:** Cross-entropy loss

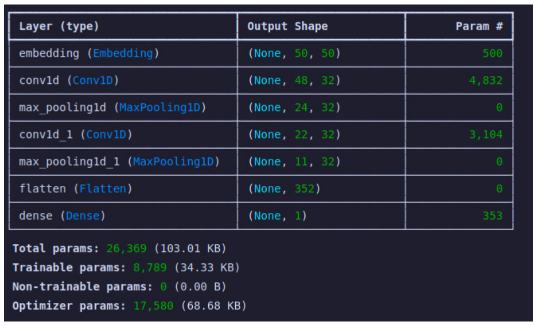


```
┌──────────────────────────────┬──────────────────────┬──────────────┐
│ Layer (type)                 │ Output Shape         │      Param # │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ input_layer_23 (InputLayer)  │ (None, 13)           │            0 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ embedding_3 (Embedding)      │ (None, 13, 5)        │        1,075 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ flatten_23 (Flatten)         │ (None, 65)           │            0 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense_29 (Dense)             │ (None, 32)           │        2,112 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense_30 (Dense)             │ (None, 32)           │        1,056 │
├──────────────────────────────┼──────────────────────┼──────────────┤
│ dense_31 (Dense)             │ (None, 1)            │           33 │
└──────────────────────────────┴──────────────────────┴──────────────┘
Total params: 12,830 (50.12 KB)
Trainable params: 4,276 (16.70 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 8,554 (33.42 KB)
```

   -

To prevent overfitting, techniques such as early stopping and cross-validation were employed. Hyperparameter tuning was performed using grid search and random search.

2. Traditional Models In addition to the neural network, several traditional machine learning models were trained and evaluated, including:
    - Logistic Regression
    - Support Vector Machines (SVM)
    - Random Forest
    - XGBoost
    - K-Nearest Neighbors (KNN)
    - parameters:



```
Layer (type)                      Output Shape          Param #
embedding (Embedding)             (None, 50, 50)            500
conv1d (Conv1D)                   (None, 48, 32)          4,832
max_pooling1d (MaxPooling1D)      (None, 24, 32)              0
conv1d_1 (Conv1D)                 (None, 22, 32)          3,104
max_pooling1d_1 (MaxPooling1D)    (None, 11, 32)              0
flatten (Flatten)                 (None, 352)                 0
dense (Dense)                     (None, 1)                 353

Total params: 26,369 (103.01 KB)
Trainable params: 8,789 (34.33 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 17,580 (68.68 KB)
```

Hyperparameters for these models were fine-tuned using grid search.

The results of these models, evaluated on the validation set, are summarized in Tables 1 and 2.

Table 1: Model Performance Comparison (Strategy-I)

| **Model** | **20%** | **40%** | **60%** | **80%** | **100%** |
|---|---|---|---|---|---|
| Logistic Regression | 0.7198 | 0.8200 | 0.8650 | 0.9162 | 0.9243 |
| Random Forest | 0.6033 | 0.7301 | 0.7464 | 0.8384 | 0.8589 |
| SVM | 0.5440 | 0.6585 | 0.6830 | 0.7464 | 0.7587 |

Table 2: Model Performance Comparison (Strategy-II)

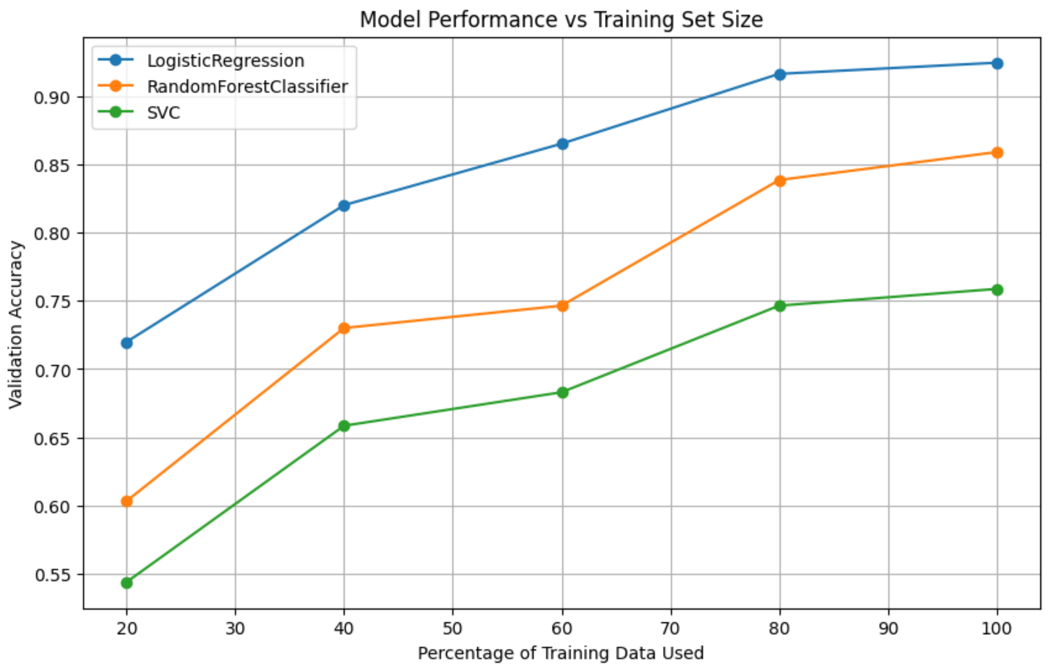
| **Model** | **20%** | **40%** | **60%** | **80%** | **100%** |
|---|---|---|---|---|---|
| Custom Neural Network | 0.9141% | 0.9284% | 0.9427% | 0.9714% | 0.9734 |

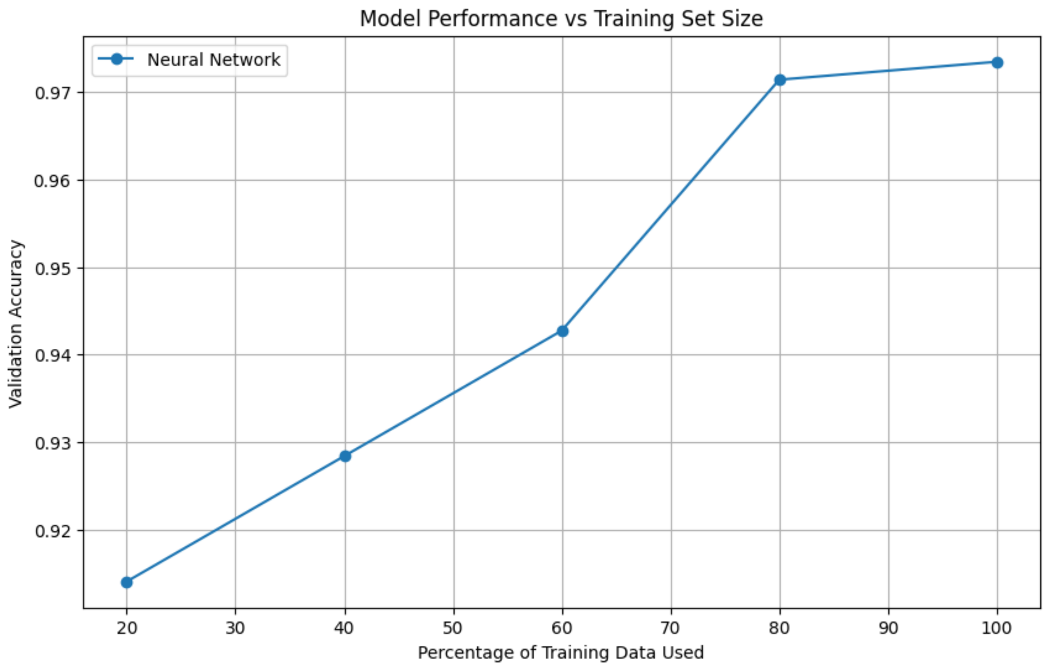
### 3.1.3 Results and Analysis

The performance of each model is assessed based on validation accuracy. Neural networks using strategy-II consistently outperformed other models, particularly as the size of the training data increased. In the case of primitive classifiers, logistic regression has given highly accurate results compared to other models. We have achieved an accuracy of 92.43% using logistic regression alone and an accuracy of 97.34% using neural networks.

### 3.1.4 Accuracy vs Training Size

The following figure illustrates how validation accuracy varies with different percentages of training data used for each model.





## 3.2 Deep Features Dataset

### 3.2.1 Data Processing

- We first reshaped the $13 \times 786$ dimensional features into a flattened representation to make them compatible with our models while preserving the information content
- We maintained the original scale of the deep features the same since they were already normalized through the neural network extraction process

### 3.2.2 Model Selection

For the deep features dataset, which consists of $13 \times 786$ dimensional embeddings, we carefully selected a diverse set of models that could effectively handle high-dimensional data, we experimented with various models:

- **XGBoost:** Selected as our primary model due to its ability to handle complex non-linear relationships in high-dimensional embedding spaces. We implemented extensive hyperparameter tuning using RandomizedSearchCV, exploring key parameters:
  - $n_{\text{estimators}}$: [100, 200, 300] for ensemble size optimization.
  - learning rate: [0.01, 0.1, 0.2] to control the boosting step size.
  - max depth: [3, 4, 5] to manage model complexity given our deep features.
- **Random Forest:** Implemented with 200 estimators to capture complex relationships in the embedding space while maintaining robustness to overfitting.
- **Logistic Regression:** Included as a baseline model with liblinear solver, specifically chosen for its efficiency with high-dimensional data and L2 regularization.
- **Support Vector Machine (SVM)**: Chosen with RBF kernel for its effectiveness in handling high-dimensional feature spaces, particularly relevant for our 786-dimensional embeddings.

### 3.2.3 Model Training and Evaluation

Training included:

- **Hyperparameter Tuning:** We implemented extensive hyperparameter tuning using RandomizedSearchCV, exploring key parameters:
  - $n_{\text{estimators}}$: [100, 200, 300] for ensemble size optimization.
  - learning rate: [0.01, 0.1, 0.2] to control the boosting step size.
  - max depth: [3, 4, 5] to manage model complexity given our deep features.
- **Training Set Size Experimentation:** The models were trained with different proportions of the training data (20%, 40%, 60%, 80%, and 100%).

### 3.2.4 Results and Analysis

XGBoost demonstrated superior performance on the full deep features dataset, achieving the highest validation accuracy across all models.

| **Model** | **20%** | **40%** | **60%** | **80%** | **100%** |
|---|---|---|---|---|---|
| XGBoost | 0.9489 | 0.9734 | 0.9816 | 0.9836 | 0.9877 |
| Logistic Regression | 0.9652 | 0.9755 | 0.9775 | 0.9796 | 0.9836 |
| Random Forest | 0.9652 | 0.9775 | 0.9816 | 0.9836 | 0.9857 |
| SVM | 0.9489 | 0.9550 | 0.9530 | 0.9611 | 0.9734 |

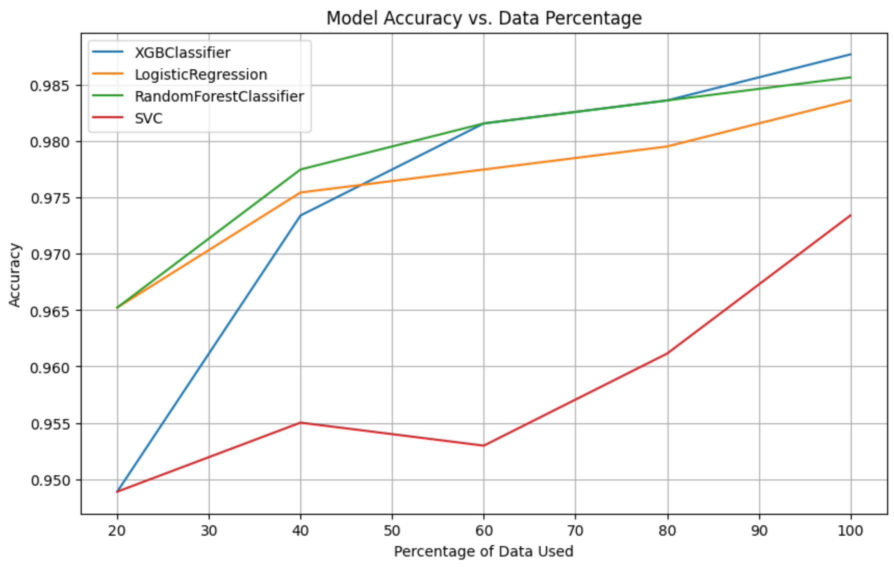
Table 3: Validation Accuracy for Different Models at Varying Training Sizes

### 3.2.5 Accuracy vs Training Size

## 3.3 Text Sequence Dataset

### 3.3.1 Data Processing

- Converted string sequences to numerical format.
- Implemented dynamic padding using `pad_sequences` to handle variable-length inputs.
- Standardized CNN-extracted features using `StandardScaler` before classification.



### 3.3.2 Model Selection

For the text sequence dataset, we implemented a sophisticated hybrid architecture combining deep learning for feature extraction and traditional classifiers. Our model selection was driven by the sequential nature of the input data and the need to maintain parameter efficiency:

**CNN Feature Extractor:**

- Embedding Layer (dimension = 16) to learn dense representations of numerical sequences.
- Two-layer CNN architecture with:
  - First Conv1D layer: 32 filters, kernel size = 3, ReLU activation.
  - Second Conv1D layer: 32 filters, kernel size = 3, ReLU activation.
- MaxPooling1D layers (pool size = 2) for dimensionality reduction.
- Dense layer (units = 8) for final feature extraction.
- Parameter-efficient design ensuring total trainable parameters stayed under $10,000$.

**Classifiers:**

- XGBoost: Selected for its ability to capture complex patterns in CNN-extracted features.
- Logistic Regression: Implemented as a linear baseline classifier to evaluate the quality of CNN-extracted features.

The hybrid architecture was specifically designed to handle the sequential nature of the input data while maintaining computational efficiency and avoiding overfitting.

### 3.3.3 Training and Validation Strategy

We implemented a comprehensive training strategy that focused on both feature extraction and classification:

**Feature Extraction Pipeline:**

- Trained the CNN model for 10 epochs with Adam optimizer.
- Used binary cross-entropy loss appropriate for binary classification.

- Extracted features from the trained CNN for both training and validation sets.

**Hyperparameter Optimization:**

- Conducted grid search over key parameters:
  - Embedding dimensions: [16].
  - Conv1D filters: [32, 32].
  - Dense units: [8].
- Monitored total parameter count to ensure model efficiency.
- Tracked validation accuracy across different configurations.

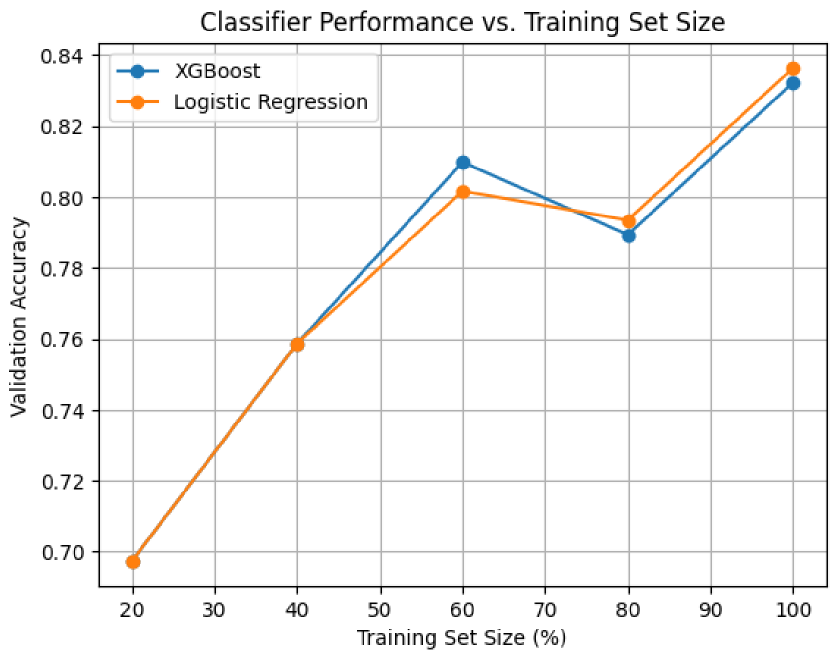
### 3.3.4 Results and Analysis

From working with the previous 2 datasets, we realized that logistic regression and XGBoost classifiers are doing great for the given dataset, we have focused on fine-tuning and experimenting with these 2 classifiers, and as expected, they turned out to be the best performers.

Table 4: Validation Accuracy for Different Models at Varying Training Sizes

| **Model** | **20%** | **40%** | **60%** | **80%** | **100%** |
|---|---|---|---|---|---|
| XGBoost | 0.6973 | 0.7587 | 0.8098 | 0.7894 | 0.8323 |
| Logistic Regression | 0.6973 | 0.7587 | 0.8016 | 0.7935 | 0.8364 |

### 3.3.5 Accuracy vs Training Size



## 4 Task 2: Developing a Unified Model by Combining Datasets

### 4.1 Feature Extraction

1. **Emoticons Dataset:** Similar to Task 1, we trained our custom neural network on this dataset, with slight parameter modifications. Features were extracted from the learned model by removing the output layer, converting each 13-length emoticon string into a feature vector of length 32.
2. **Features Dataset:** Each input in this dataset consists of a $13 \times 768$ matrix, which was flattened into a vector of length 9984.
3. **Text Sequence Dataset:** Similar to Task 1.3, we trained another custom neural network with adjusted parameters and extracted the learned features from the network (removing the output layer). Each 50-length text sequence was transformed into a feature vector of length 176.

### 4.2 Feature Combination

1. **Data Standardization:** To ensure that all features contributed equally, we applied 'StandardScaler' to each dataset, standardizing the features to have a mean of 0 and a standard deviation of 1.
2. **PCA Dimensionality Reduction:** After standardization, Principal Component Analysis (PCA) was applied to reduce dimensionality while retaining 95% of the variance. For the training data, PCA objects were fitted, and the same objects were applied to the validation data for consistency.
3. **Concatenation of Features:** The PCA-transformed data from the three datasets were concatenated horizontally using 'np.stack()', forming a unified feature vector of length 343 for each input instance.

### 4.3 Model Training and Validation

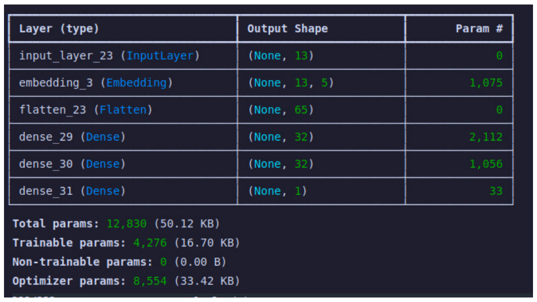
Once the combined dataset was prepared, we trained and validated the following models on it:

- Logistic Regression
- Decision Tree
- Random Forest
- K-Nearest Neighbors (KNN)
- Support Vector Machine (SVM)
- Neural Networks
- XGBoost
- LightGBM

### 4.4 Neural Network Configurations

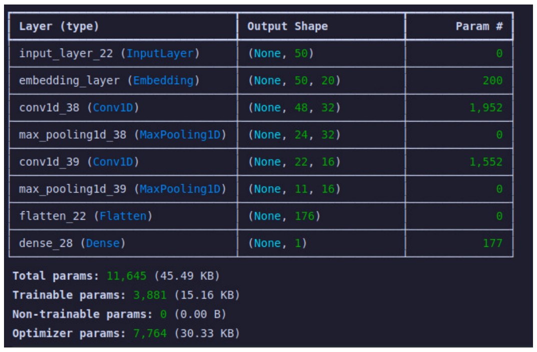
For feature extraction, two neural networks were trained:

- **Neural Network for Emoticons:**
  - Learning rate: 0.01
  - Batch size: 32
  - Epochs: 5
  - Optimizer: Adam
  - Loss function: Cross-entropy
  - Parameters:



```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_23 (InputLayer)     │ (None, 13)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ embedding_3 (Embedding)         │ (None, 13, 5)          │         1,075 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_23 (Flatten)            │ (None, 65)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_29 (Dense)                │ (None, 32)             │         2,112 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_30 (Dense)                │ (None, 32)             │         1,056 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_31 (Dense)                │ (None, 1)              │            33 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 12,830 (50.12 KB)
 Trainable params: 4,276 (16.70 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 8,554 (33.42 KB)
```

- **Neural Network for Text Sequences:**
  - Conv1D filters: [32, 32]
  - Dense units: 8
  - Trained for 20 epochs using the Adam optimizer
  - Loss function: Binary cross-entropy
  - Conducted grid search for hyperparameter optimization
  - Parameters:



```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ input_layer_22 (InputLayer)     │ (None, 50)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ embedding_layer (Embedding)     │ (None, 50, 20)         │           200 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv1d_38 (Conv1D)              │ (None, 48, 32)         │         1,952 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling1d_38 (MaxPooling1D) │ (None, 24, 32)         │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv1d_39 (Conv1D)              │ (None, 22, 16)         │         1,552 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling1d_39 (MaxPooling1D) │ (None, 11, 16)         │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_22 (Flatten)            │ (None, 176)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_28 (Dense)                │ (None, 1)              │           177 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 11,645 (45.49 KB)
 Trainable params: 3,881 (15.16 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 7,764 (30.33 KB)
```

## 4.5 Results and Analysis

The performance of each model was evaluated on different percentages of the dataset (20%, 40%, 60%, 80%, and 100%), as shown in the following table:

| **Model** | **20%** | **40%** | **60%** | **80%** | **100%** |
|---|---|---|---|---|---|
| Logistic Regression | 0.9366 | 0.9611 | 0.9755 | 0.9836 | 0.9816 |
| Decision Tree | 0.8896 | 0.9530 | 0.9652 | 0.9693 | 0.9775 |
| Random Forest | 0.8691 | 0.9489 | 0.9530 | 0.9816 | 0.9755 |
| KNN | 0.9468 | 0.9591 | 0.9571 | 0.9530 | 0.9550 |
| SVM | 0.9571 | 0.9714 | 0.9775 | 0.9877 | 0.9898 |
| Neural Network | 0.9223 | 0.9550 | 0.9591 | 0.9796 | 0.9734 |
| XGBoost | 0.9448 | 0.9648 | 0.9652 | 0.9877 | 0.9836 |
| LightGBM | 0.9264 | 0.9489 | 0.9693 | 0.9836 | 0.9796 |

Table 5: Performance of Various Models on Combined Dataset

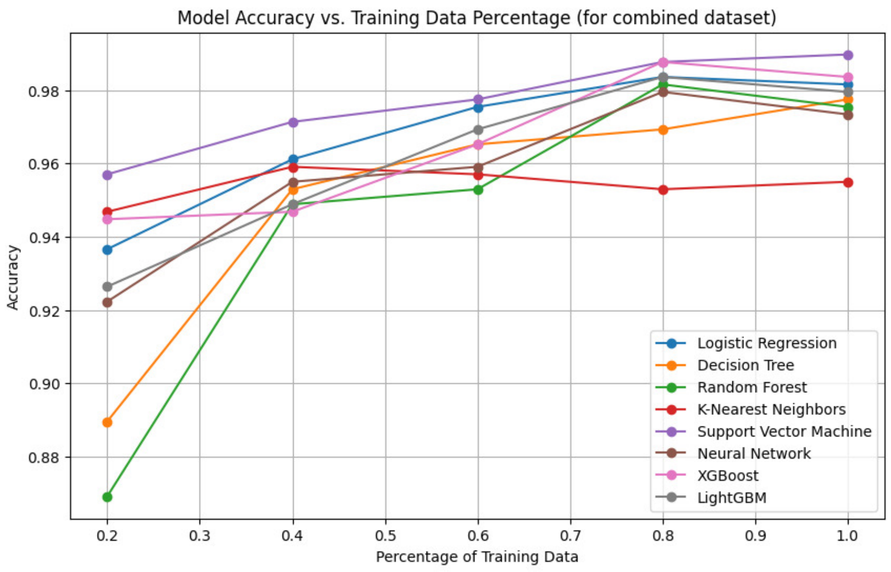


Figure 1: Accuracy vs Training Data Percentage

## 5 Conclusion

Table 6: Final Models and Validation Accuracies of Each Task

| **Task** | **Model** | **Feature Extraction Method** | **Training %** | **Validation Accuracy** |
|---|---|---|---|---|
| Task-1.1 | Deep Neural Network (DNN) | Deep Neural Network (DNN) | 20% | 91.41% |
| | | | 40% | 92.84% |
| | | | 60% | 94.27% |
| | | | 80% | 97.14% |
| | | | 100% | 97.34% |
| Task-1.2 | XGBoost | pre-extracted features | 20% | 94.89% |
| | | | 40% | 97.34% |
| | | | 60% | 98.16% |
| | | | 80% | 98.36% |
| | | | 100% | 98.77% |
| Task-1.3 | Neural networks | CNN | 20% | N/A |
| | | | 40% | N/A |
| | | | 60% | 88.34% |
| | | | 80% | 89.57% |
| | | | 100% | 92.02% |
| Task-2 | Logistic Regression | PCA | 20% | N/A |
| | | | 40% | 95.91% |
| | | | 60% | 97.13% |
| | | | 80% | 97.75% |
| | | | 100% | 98.77% |

## References

[1] scikit-learn: Machine Learning in Python. Available at: `https://scikit-learn.org/stable/`

[2] Keras: Deep Learning API in Python. Available at: `https://keras.io/`

[3] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.