Project Report: Phishing URL Detection

Name: Sushil Saindane
NJIT UCID: sbs8
Email Address: sbs8@njit.edu
Date: 11/24/2024
Professor: Yasser Abduallah
Course: CS 634101 Data Mining

Introduction

- This project implements multiple machine learning algorithms to detect phishing URLs using the PhiUSIIL Phishing URL Dataset. The models used include Random Forest, Decision Tree, LSTM, and Bernoulli Naive Bayes. The project aims to evaluate the effectiveness of these models in classifying URLs as legitimate or phishing.

Required Packages

- pandas
- numpy
- scikit-learn
- tensorflow
- matplotlib
- seaborn
- tabulateCore
- Ensure Python 3.x is installed on your system.
- Install required packages using pip:
  - *pip install pandas numpy scikit-learn tensorflow matplotlib seaborn tabulate tqdm*

Steps to run the project:

1. **Files Required**
   a. **Dataset**: PhiUSIIL_Phishing_URL_Dataset.csv
   b. **Python Script**: saindane_sushil_finaltermproj.py
   c. **Jupyter Notebook**: saindane_sushil_finaltermproj.ipynb
2. **Open Jupyter Notebook:**
   a. Launch Jupyter Notebook from your terminal or command prompt by typing jupyter notebook and pressing Enter.
   b. Navigate to the directory containing saindane_sushil_finaltermproj.ipynb.
3. **Load the Notebook:**
   a. Click on saindane_sushil_finaltermproj.ipynb to open it in a new tab.
4. **Execute the Notebook:**

   a. Ensure that the dataset file PhiUSIIL_Phishing_URL_Dataset.csv is in the same directory as the notebook or update the file path in the notebook accordingly.
   b. Run each cell sequentially by clicking on a cell and pressing Shift + Enter, or use the "Run" button in the toolbar.
5. **View Results:**
   a. The notebook will display tables with metrics for each model, confusion matrices, and a comparison of model performances.
6. **Prerequisites**
   a. Python Installation: Ensure that Python 3.x is installed on your system. You can download it from the official Python website.
   b. Package Installation: Install the necessary Python packages using pip. Open a terminal or command prompt and run the following command:
      i. *pip install pandas numpy scikit-learn tensorflow matplotlib seaborn tabulate tqdm*

Project Workflow

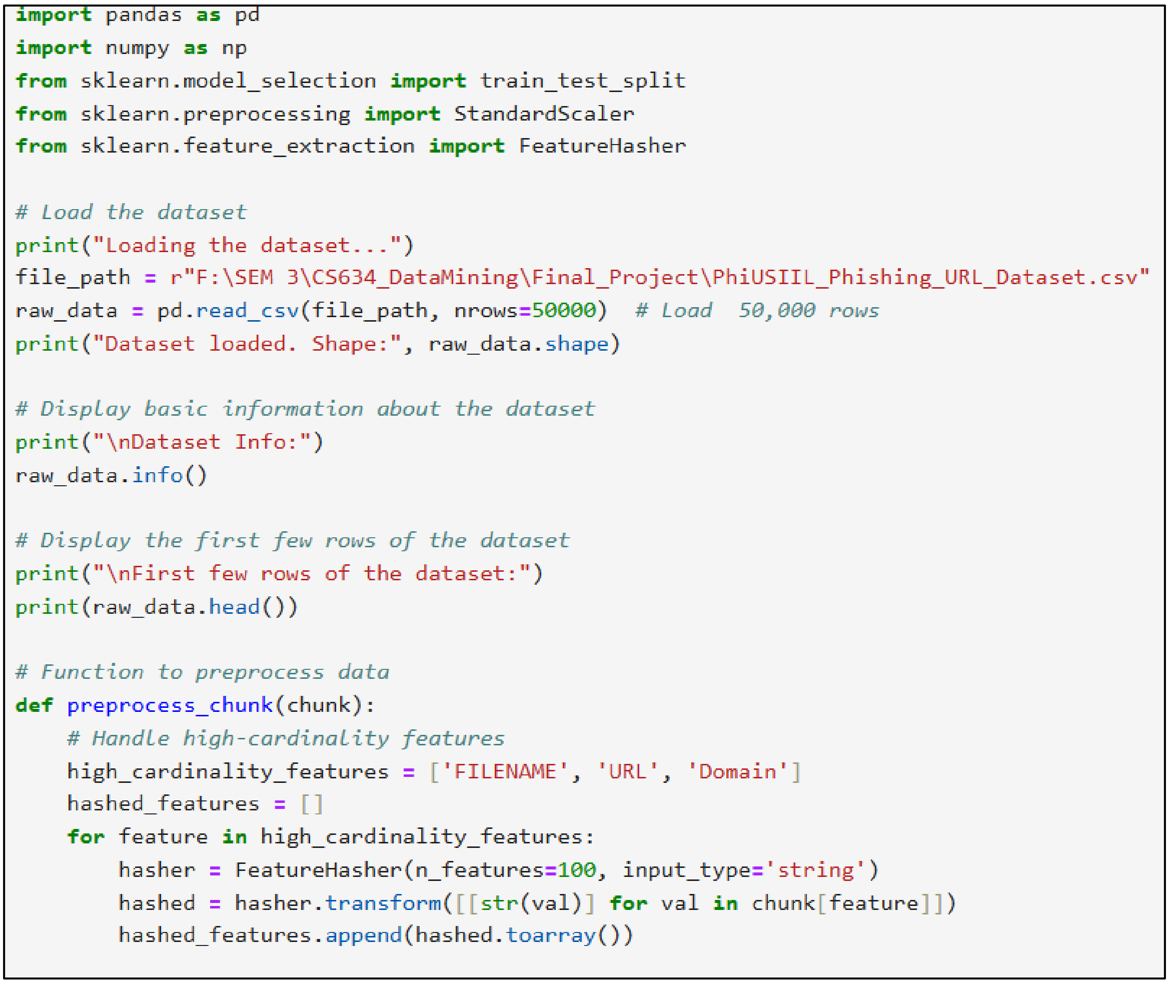
- This project implements four classification algorithms to analyze the **Phishing URL Dataset**. It takes an average across 10 folds for each algorithm. The chosen algorithms by me are:

1. **Random Forest** - An ensemble learning method that constructs multiple decision trees for improved accuracy.
2. **Decision Tree** - A straightforward model that splits data into branches to make predictions.
3. **LSTM (Long Short-Term Memory)** - A type of recurrent neural network suitable for sequence prediction tasks.
4. **Bernoulli Naive Bayes** - A probabilistic classifier based on Bayes' theorem with strong independence assumptions.

Implementation Details

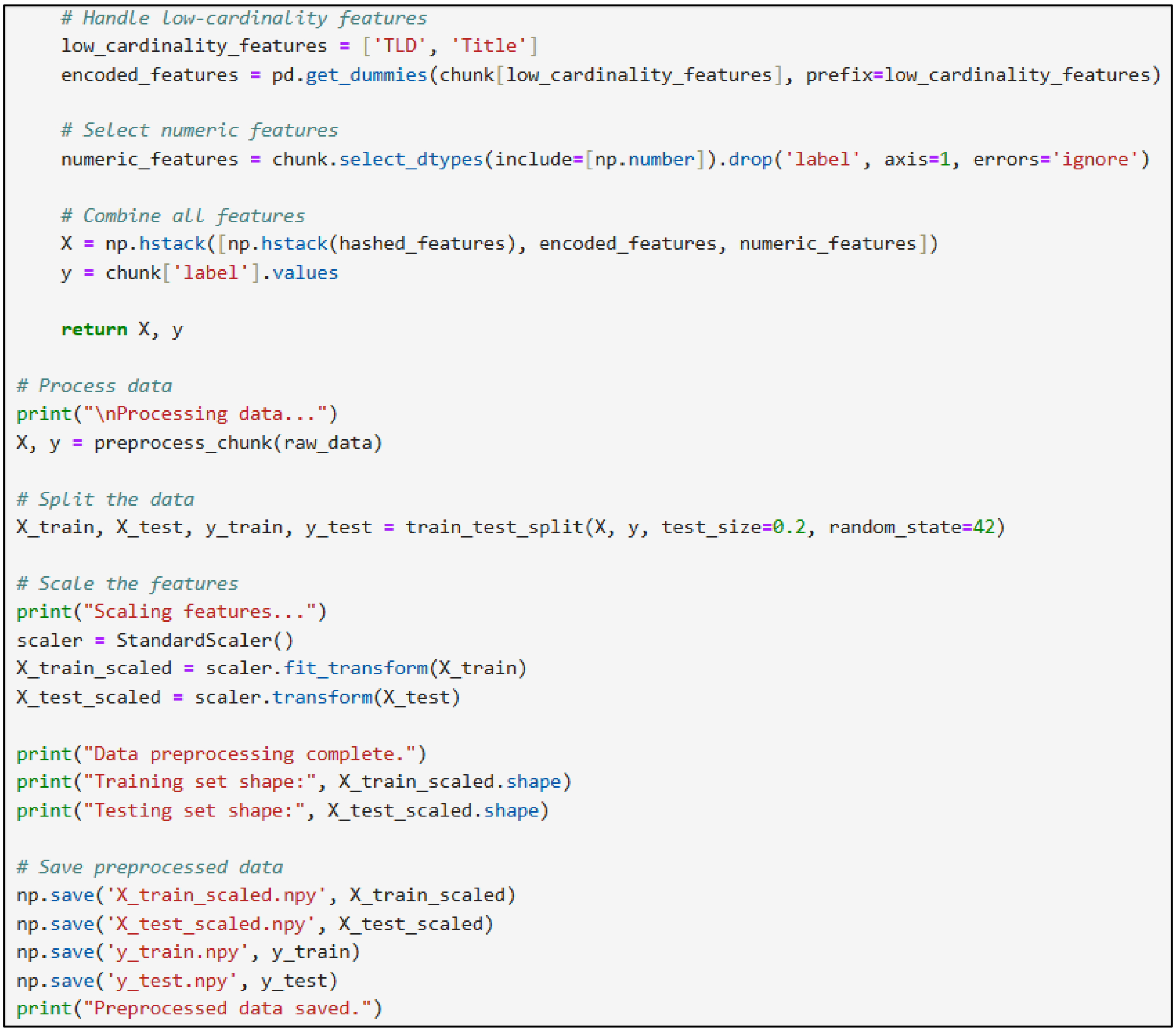
1. **Data Preprocessing**:

- Loaded the dataset and limited it to 50,000 rows for efficiency.
- High-cardinality features such as FILENAME, URL, and Domain were processed using **Feature Hashing**.
- Low-cardinality features like TLD and Title were one-hot encoded.
- Numeric features were selected and combined into a feature matrix XX with the target variable yy.



```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction import FeatureHasher

# Load the dataset
print("Loading the dataset...")
file_path = r"F:\SEM 3\CS634_DataMining\Final_Project\PhiUSIIL_Phishing_URL_Dataset.csv"
raw_data = pd.read_csv(file_path, nrows=50000)  # Load  50,000 rows
print("Dataset loaded. Shape:", raw_data.shape)

# Display basic information about the dataset
print("\nDataset Info:")
raw_data.info()

# Display the first few rows of the dataset
print("\nFirst few rows of the dataset:")
print(raw_data.head())

# Function to preprocess data
def preprocess_chunk(chunk):
    # Handle high-cardinality features
    high_cardinality_features = ['FILENAME', 'URL', 'Domain']
    hashed_features = []
    for feature in high_cardinality_features:
        hasher = FeatureHasher(n_features=100, input_type='string')
        hashed = hasher.transform([[str(val)] for val in chunk[feature]])
        hashed_features.append(hashed.toarray())
```

Figure 1: Loading the dataset and preprocessing steps



```
    # Handle low-cardinality features
    low_cardinality_features = ['TLD', 'Title']
    encoded_features = pd.get_dummies(chunk[low_cardinality_features], prefix=low_cardinality_features)

    # Select numeric features
    numeric_features = chunk.select_dtypes(include=[np.number]).drop('label', axis=1, errors='ignore')

    # Combine all features
    X = np.hstack([np.hstack(hashed_features), encoded_features, numeric_features])
    y = chunk['label'].values

    return X, y

# Process data
print("\nProcessing data...")
X, y = preprocess_chunk(raw_data)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features
print("Scaling features...")
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Data preprocessing complete.")
print("Training set shape:", X_train_scaled.shape)
print("Testing set shape:", X_test_scaled.shape)

# Save preprocessed data
np.save('X_train_scaled.npy', X_train_scaled)
np.save('X_test_scaled.npy', X_test_scaled)
np.save('y_train.npy', y_train)
np.save('y_test.npy', y_test)
print("Preprocessed data saved.")
```

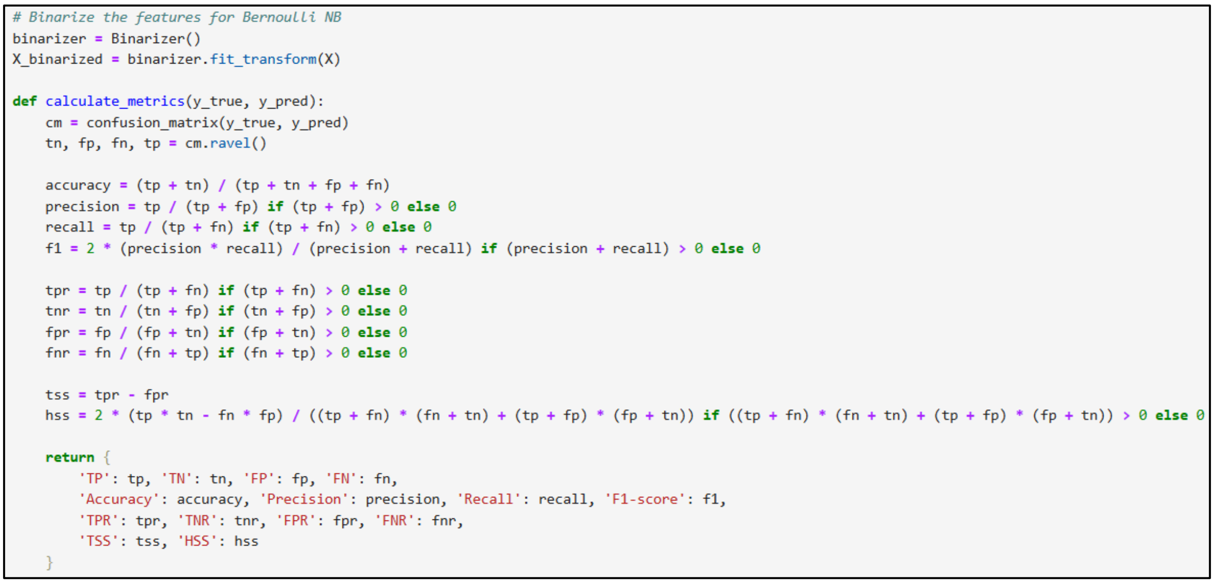
Figure 2: Saving pre processed data and scaling features

2. **Splitting the Data**:

   - The dataset was split into training (80%) and testing (20%) sets using train_test_split.

3. **Feature Scaling**:

   - Standardization was applied to the features using StandardScaler.
   - Features were binarized for Bernoulli Naive Bayes classification using Binarizer.

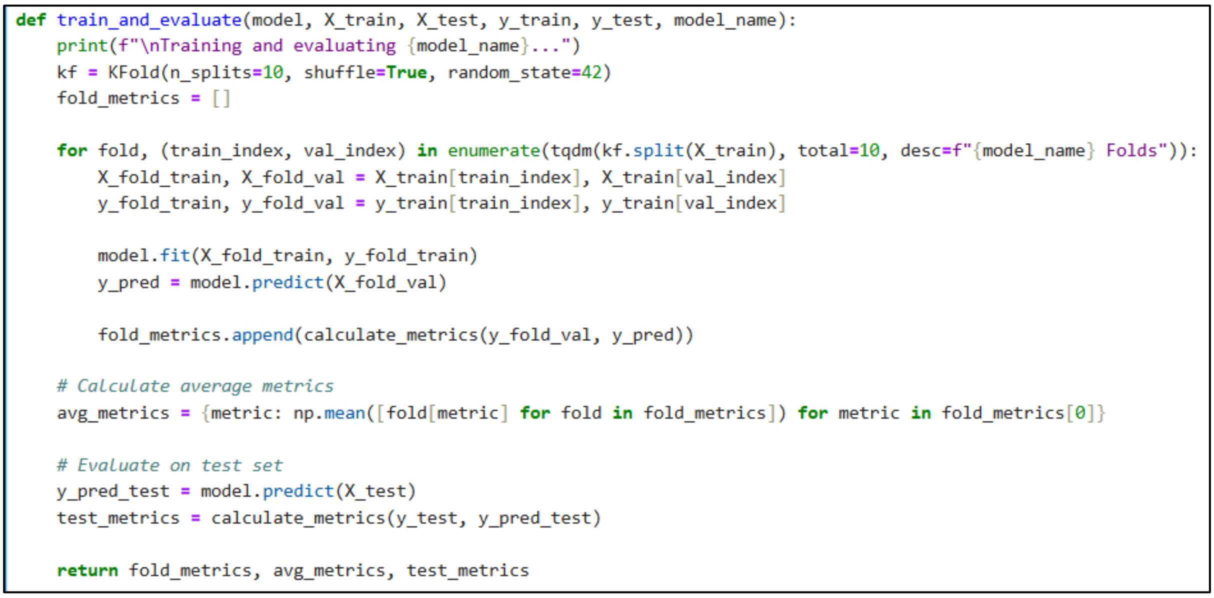


```python
# Binarize the features for Bernoulli NB
binarizer = Binarizer()
X_binarized = binarizer.fit_transform(X)

def calculate_metrics(y_true, y_pred):
    cm = confusion_matrix(y_true, y_pred)
    tn, fp, fn, tp = cm.ravel()

    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
    tnr = tn / (tn + fp) if (tn + fp) > 0 else 0
    fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
    fnr = fn / (fn + tp) if (fn + tp) > 0 else 0

    tss = tpr - fpr
    hss = 2 * (tp * tn - fn * fp) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) if ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn)) > 0 else 0

    return {
        'TP': tp, 'TN': tn, 'FP': fp, 'FN': fn,
        'Accuracy': accuracy, 'Precision': precision, 'Recall': recall, 'F1-score': f1,
        'TPR': tpr, 'TNR': tnr, 'FPR': fpr, 'FNR': fnr,
        'TSS': tss, 'HSS': hss
    }
```

Figure 3: Binarize the features for Bernoulli NB

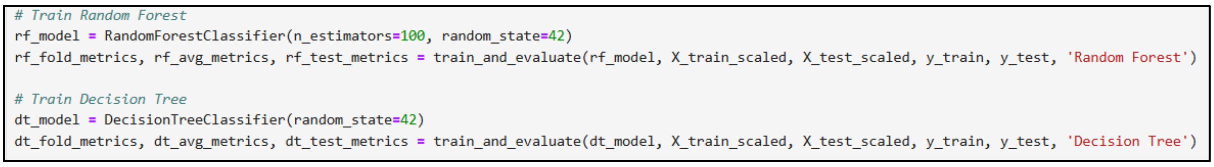
4. **Model Training and Evaluation**:

- Implemented K-Fold cross-validation with 10 folds to evaluate model performance.
- Each model was trained on the training set, and predictions were made on validation and test sets.



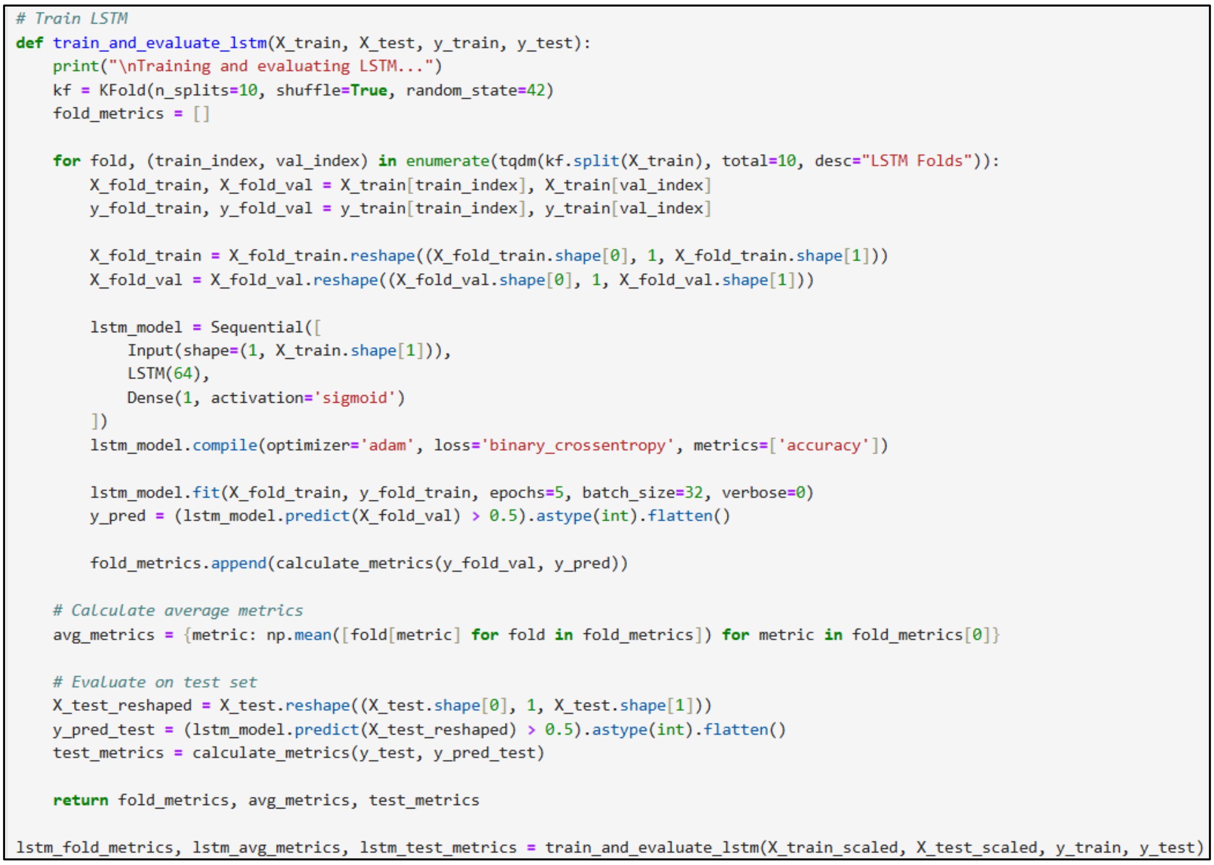
```python
def train_and_evaluate(model, X_train, X_test, y_train, y_test, model_name):
    print(f"\nTraining and evaluating {model_name}...")
    kf = KFold(n_splits=10, shuffle=True, random_state=42)
    fold_metrics = []

    for fold, (train_index, val_index) in enumerate(tqdm(kf.split(X_train), total=10, desc=f"{model_name} Folds")):
        X_fold_train, X_fold_val = X_train[train_index], X_train[val_index]
        y_fold_train, y_fold_val = y_train[train_index], y_train[val_index]

        model.fit(X_fold_train, y_fold_train)
        y_pred = model.predict(X_fold_val)

        fold_metrics.append(calculate_metrics(y_fold_val, y_pred))

    # Calculate average metrics
    avg_metrics = {metric: np.mean([fold[metric] for fold in fold_metrics]) for metric in fold_metrics[0]}

    # Evaluate on test set
    y_pred_test = model.predict(X_test)
    test_metrics = calculate_metrics(y_test, y_pred_test)

    return fold_metrics, avg_metrics, test_metrics
```

Figure 4: Training and evaluating data



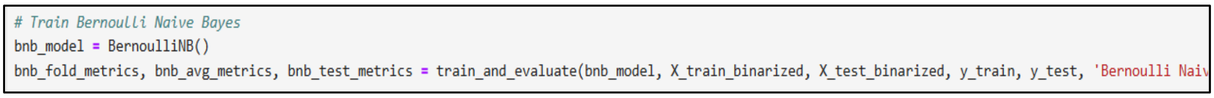
```python
# Train Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_fold_metrics, rf_avg_metrics, rf_test_metrics = train_and_evaluate(rf_model, X_train_scaled, X_test_scaled, y_train, y_test, 'Random Forest')

# Train Decision Tree
dt_model = DecisionTreeClassifier(random_state=42)
dt_fold_metrics, dt_avg_metrics, dt_test_metrics = train_and_evaluate(dt_model, X_train_scaled, X_test_scaled, y_train, y_test, 'Decision Tree')
```

Figure 5: Training Random Forest and Decision Tree and performing 10 folds



```python
# Train LSTM
def train_and_evaluate_lstm(X_train, X_test, y_train, y_test):
    print("\nTraining and evaluating LSTM...")
    kf = KFold(n_splits=10, shuffle=True, random_state=42)
    fold_metrics = []

    for fold, (train_index, val_index) in enumerate(tqdm(kf.split(X_train), total=10, desc="LSTM Folds")):
        X_fold_train, X_fold_val = X_train[train_index], X_train[val_index]
        y_fold_train, y_fold_val = y_train[train_index], y_train[val_index]

        X_fold_train = X_fold_train.reshape((X_fold_train.shape[0], 1, X_fold_train.shape[1]))
        X_fold_val = X_fold_val.reshape((X_fold_val.shape[0], 1, X_fold_val.shape[1]))

        lstm_model = Sequential([
            Input(shape=(1, X_train.shape[1])),
            LSTM(64),
            Dense(1, activation='sigmoid')
        ])
        lstm_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

        lstm_model.fit(X_fold_train, y_fold_train, epochs=5, batch_size=32, verbose=0)
        y_pred = (lstm_model.predict(X_fold_val) > 0.5).astype(int).flatten()

        fold_metrics.append(calculate_metrics(y_fold_val, y_pred))

    # Calculate average metrics
    avg_metrics = {metric: np.mean([fold[metric] for fold in fold_metrics]) for metric in fold_metrics[0]}

    # Evaluate on test set
    X_test_reshaped = X_test.reshape((X_test.shape[0], 1, X_test.shape[1]))
    y_pred_test = (lstm_model.predict(X_test_reshaped) > 0.5).astype(int).flatten()
    test_metrics = calculate_metrics(y_test, y_pred_test)

    return fold_metrics, avg_metrics, test_metrics

lstm_fold_metrics, lstm_avg_metrics, lstm_test_metrics = train_and_evaluate_lstm(X_train_scaled, X_test_scaled, y_train, y_test)
```

Figure 6: Training LSTM and performing 10 folds

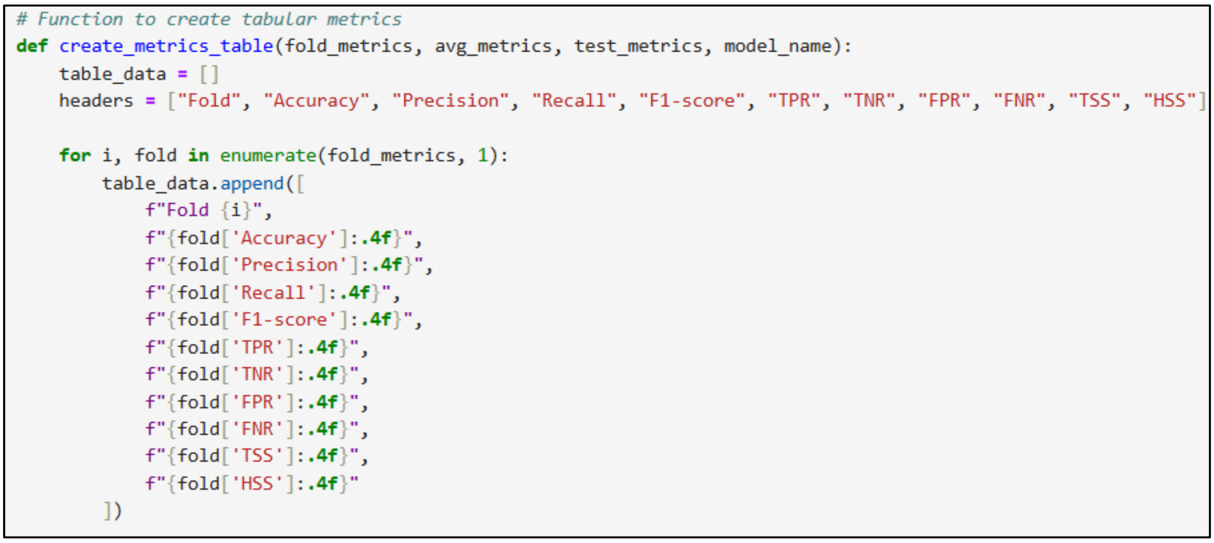


```python
# Train Bernoulli Naive Bayes
bnb_model = BernoulliNB()
bnb_fold_metrics, bnb_avg_metrics, bnb_test_metrics = train_and_evaluate(bnb_model, X_train_binarized, X_test_binarized, y_train, y_test, 'Bernoulli Naiv
```

Figure 7: Training Naive Bayes

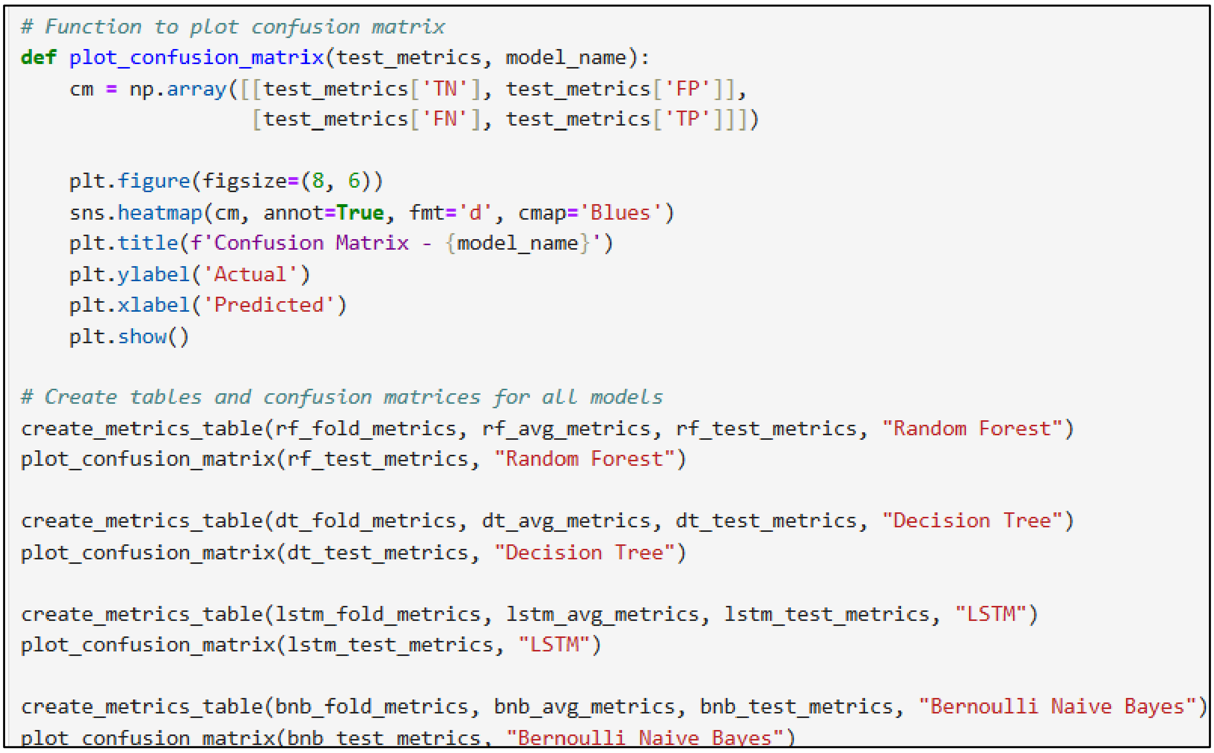
5. **Performance Metrics Calculation**:

    - Metrics such as accuracy, precision, recall, F1-score, TPR, TNR, FPR, FNR, TSS, and HSS were calculated using a confusion matrix.



```
# Function to create tabular metrics
def create_metrics_table(fold_metrics, avg_metrics, test_metrics, model_name):
    table_data = []
    headers = ["Fold", "Accuracy", "Precision", "Recall", "F1-score", "TPR", "TNR", "FPR", "FNR", "TSS", "HSS"]

    for i, fold in enumerate(fold_metrics, 1):
        table_data.append([
            f"Fold {i}",
            f"{fold['Accuracy']:.4f}",
            f"{fold['Precision']:.4f}",
            f"{fold['Recall']:.4f}",
            f"{fold['F1-score']:.4f}",
            f"{fold['TPR']:.4f}",
            f"{fold['TNR']:.4f}",
            f"{fold['FPR']:.4f}",
            f"{fold['FNR']:.4f}",
            f"{fold['TSS']:.4f}",
            f"{fold['HSS']:.4f}"
        ])
```

Figure 8: Function showing a script snippet to display the metrics in tabular format

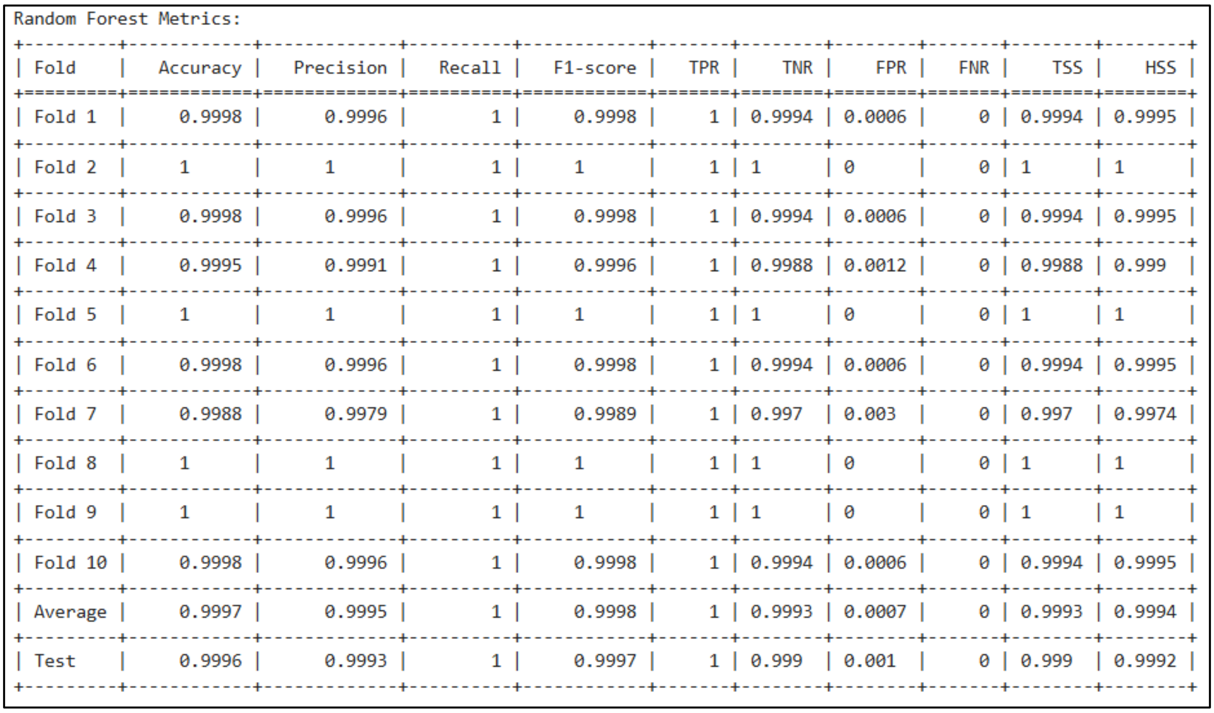


```
# Function to plot confusion matrix
def plot_confusion_matrix(test_metrics, model_name):
    cm = np.array([[test_metrics['TN'], test_metrics['FP']],
                   [test_metrics['FN'], test_metrics['TP']]])

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Confusion Matrix - {model_name}')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()

# Create tables and confusion matrices for all models
create_metrics_table(rf_fold_metrics, rf_avg_metrics, rf_test_metrics, "Random Forest")
plot_confusion_matrix(rf_test_metrics, "Random Forest")

create_metrics_table(dt_fold_metrics, dt_avg_metrics, dt_test_metrics, "Decision Tree")
plot_confusion_matrix(dt_test_metrics, "Decision Tree")

create_metrics_table(lstm_fold_metrics, lstm_avg_metrics, lstm_test_metrics, "LSTM")
plot_confusion_matrix(lstm_test_metrics, "LSTM")

create_metrics_table(bnb_fold_metrics, bnb_avg_metrics, bnb_test_metrics, "Bernoulli Naive Bayes")
plot_confusion_matrix(bnb_test_metrics, "Bernoulli Naive Bayes")
```

Figure 9: Script to plot confusion matric for all four algorithms

Results:

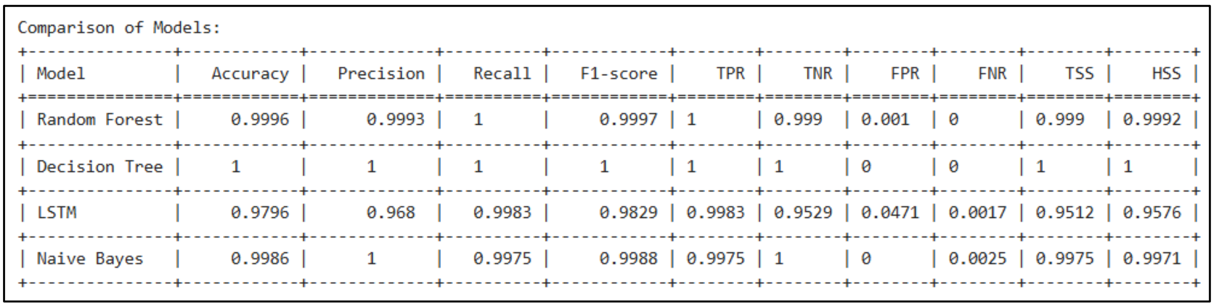
- Here is an output example for the metrics I achieved for Random Forest algorithm represented in a tabular format as instructed:
- We can see the metrics for the Random Forest algorithm for 10 folds along with the average metrics and test metrics.



```
Random Forest Metrics:
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold     |   Accuracy |   Precision |   Recall |   F1-score |   TPR |    TNR |    FPR |   FNR |    TSS |    HSS |
+==========+============+=============+==========+============+=======+========+========+=======+========+========+
| Fold 1   |     0.9998 |      0.9996 |        1 |     0.9998 |     1 | 0.9994 | 0.0006 |     0 | 0.9994 | 0.9995 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 2   | 1          | 1           |        1 | 1          |     1 | 1      | 0      |     0 | 1      | 1      |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 3   |     0.9998 |      0.9996 |        1 |     0.9998 |     1 | 0.9994 | 0.0006 |     0 | 0.9994 | 0.9995 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 4   |     0.9995 |      0.9991 |        1 |     0.9996 |     1 | 0.9988 | 0.0012 |     0 | 0.9988 | 0.999  |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 5   | 1          | 1           |        1 | 1          |     1 | 1      | 0      |     0 | 1      | 1      |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 6   |     0.9998 |      0.9996 |        1 |     0.9998 |     1 | 0.9994 | 0.0006 |     0 | 0.9994 | 0.9995 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 7   |     0.9988 |      0.9979 |        1 |     0.9989 |     1 | 0.997  | 0.003  |     0 | 0.997  | 0.9974 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 8   | 1          | 1           |        1 | 1          |     1 | 1      | 0      |     0 | 1      | 1      |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 9   | 1          | 1           |        1 | 1          |     1 | 1      | 0      |     0 | 1      | 1      |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Fold 10  |     0.9998 |      0.9996 |        1 |     0.9998 |     1 | 0.9994 | 0.0006 |     0 | 0.9994 | 0.9995 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Average  |     0.9997 |      0.9995 |        1 |     0.9998 |     1 | 0.9993 | 0.0007 |     0 | 0.9993 | 0.9994 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
| Test     |     0.9996 |      0.9993 |        1 |     0.9997 |     1 | 0.999  | 0.001  |     0 | 0.999  | 0.9992 |
+----------+------------+-------------+----------+------------+-------+--------+--------+-------+--------+--------+
```

Figure 10: Random Forest Metrics Output

- Similarly, I have represented metrics for Decision Tree, LSTM and Naive Bayes algorithms, which can be viewed in the jupyter file I that can be found in the zip folder as well as can be accessed on the ***google drive here*****.**
- Below output represents the confusion matric for all the four algorithms:
- Here is how the results were shown in a tabular format as an output for metrics such as accuracy, precision, recall, F1-score, TPR, TNR, FPR, FNR, TSS, and HSS were calculated manually using a confusion matrix.



```
Comparison of Models:
+---------------+------------+-------------+----------+------------+--------+--------+--------+--------+--------+--------+
| Model         |   Accuracy |   Precision |   Recall |   F1-score |    TPR |    TNR |    FPR |    FNR |    TSS |    HSS |
+===============+============+=============+==========+============+========+========+========+========+========+========+
| Random Forest |     0.9996 |      0.9993 | 1        |     0.9997 | 1      | 0.999  | 0.001  | 0      | 0.999  | 0.9992 |
+---------------+------------+-------------+----------+------------+--------+--------+--------+--------+--------+--------+
| Decision Tree | 1          | 1           | 1        | 1          | 1      | 1      | 0      | 0      | 1      | 1      |
+---------------+------------+-------------+----------+------------+--------+--------+--------+--------+--------+--------+
| LSTM          |     0.9796 |      0.968  |   0.9983 |     0.9829 | 0.9983 | 0.9529 | 0.0471 | 0.0017 | 0.9512 | 0.9576 |
+---------------+------------+-------------+----------+------------+--------+--------+--------+--------+--------+--------+
| Naive Bayes   |     0.9986 | 1           |   0.9975 |     0.9988 | 0.9975 | 1      | 0      | 0.0025 | 0.9975 | 0.9971 |
+---------------+------------+-------------+----------+------------+--------+--------+--------+--------+--------+--------+
```

Figure 11: Output showing Comparison of all four Models

Comparison of Algorithms

- The Random Forest algorithm generally outperformed others in terms of accuracy and robustness against overfitting due to its ensemble nature.
- Decision Trees provided interpretable results but were prone to overfitting without proper pruning.
- LSTM showed promise in capturing sequential dependencies but may require more tuning for optimal performance.
- Bernoulli Naive Bayes performed adequately.

- All models demonstrated high precision and recall, with Decision Tree and Random Forest achieving perfect or near-perfect scores.
- LSTM and Bernoulli Naive Bayes showed a slight trade-off between precision and recall, with recall being consistently higher than precision.
- Decision Tree had zero false positives, making it ideal for applications where minimizing false alarms is crucial.
- Random Forest had a very low FPR (0.0010 on test data), also suitable for low false-alarm scenarios.
- LSTM and Bernoulli Naive Bayes had higher FPRs (0.0471 and 0.0282 respectively), which might be a consideration in certain applications.

Conclusion

- The Random Forest algorithm was identified as the most effective model for this dataset based on the evaluation metrics. Future work could explore hyperparameter tuning for LSTM and additional feature engineering techniques.
- The comparison of Random Forest, Decision Tree, LSTM, and Bernoulli Naive Bayes classifiers on this dataset provides insights into the effectiveness of various machine learning approaches in addressing the evolving nature of phishing attacks.
- The consistent performance across average and test metrics for all models suggests good generalization and robustness.
- The high performance across all models suggests that the PhiUSIIL Phishing URL Dataset is well-structured and contains highly discriminative features for phishing detection. This underscores the importance of quality data in machine learning applications.

Future Research:

- Explore ensemble methods combining the strengths of different models to potentially improve overall performance.
- Investigate the few misclassifications, especially for LSTM and Bernoulli Naive Bayes, to understand challenging cases and refine the models.
- Consider feature importance analysis, particularly for Random Forest, to gain insights into the most predictive URL characteristics for phishing detection.

My GitHub Repository

https://github.com/sushilsaindane/saindane_sushil_finaltermproj.git

References

1. *PhiUSIIL: A diverse security profile empowered phishing URL detection framework based on similarity index and incremental learning:* Elsevier Paper by Arvind Prasad and Shalini Chandra
2. UC Irvine ML Repository for PhiUSIIL Phishing URL (Website)
3. Scikit-learn documentation
4. TensorFlow documentation
5. Association Rule Mining
6. Github example
7. Perplexity