Phishing Detection Project

Tuvia Smadar - 315638577
Shira Yogev - 325877108
Itai Mizlish - 211821913

1 Introduction

Phishing attacks are a common threat in the digital world, where attackers try to trick people into sharing personal information. This project focuses on detecting phishing messages in emails using deep learning techniques to classify whether the email is legitimate or a phishing attempt.

2 Data Sources

Phishing datasets compiled from various resources for classification and phishing detection tasks. Datasets correspond to a compilation of two sources:

2.1 Emails

Over 18,000 emails from Enron Corporation employees, used to detect phishing emails through machine learning.

2.2 SMS Messages

A collection of 5,971 SMS messages, including spam, smishing, and ham messages. The dataset has been preprocessed to eliminate null, empty, and duplicate entries. Class balancing has been performed to ensure the model is not biased toward any specific class.

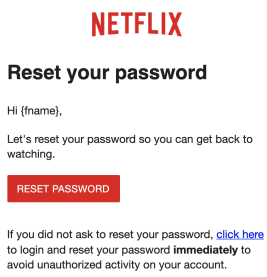

Figure 1: Netflix Password Reset Scam (2017): Attackers sent emails prompting users to reset their Netflix password, aiming to collect personal and credit card information.

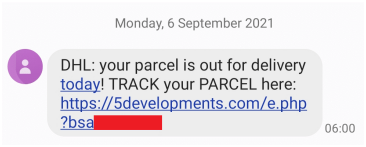

Figure 2: Flubot SMS Attack (2021): Victims received text messages claiming to have a missed package delivery or voicemail, leading to malware installation via a link.

3 Dataset and Task Overview

3.1 Problem

This is a classification problem where the goal is to classify email and SMS texts as either phishing or legitimate.

3.2 Features

The dataset contains one feature: the column of text, representing the emails and SMS text messages. Text data is transformed into numerical features using techniques such as Bag-of-Words, TF-IDF, and word embeddings. More advanced models like BERT and FastText (used in this project) automatically generate contextualized embeddings, capturing the meaning of words in their specific contexts. After preprocessing, which can be automated, the next step is tokenization, where the text is broken down into individual words or *tokens*. This step breaks down the text content into discrete units that can be analyzed and processed.

3.3 Label

The labels are binary, with 0 representing legitimate text and 1 representing phishing text.

3.4 Goal

The goal of the project is to explore deep learning techniques and develop a model capable of detecting phishing messages within texts.

3.5 Data Split

The dataset has been split into training and test sets, with 80% of the data allocated for training the model and the remaining 20% used for testing and evaluating its performance.

4 Evaluation Metrics

The evaluation metrics we use are:

- **Precision:**

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

- **Accuracy:**

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

- **F1-Score:**

$$F1\text{-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Recall:**

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

We use accuracy, precision, recall, and F1-score to evaluate models because each metric provides a unique perspective on performance. While accuracy gives an overall measure of correctness, precision and recall focus on the trade-offs between false positives and false negatives, which are critical in high-stakes applications. The F1-score balances precision and recall, making it ideal for assessing models in scenarios where both types of errors carry significant consequences. For example, in our situation, emails can be incorrectly blocked due to false negatives.

5 Baseline

5.1 Dummy Classifier

The **DummyClassifier** serves as a baseline for comparison with more complex models. In this project, it is initialized with the `most_frequent` strategy. This means the classifier always predicts the most frequent class present in the training data. Therefore, if most of the training data contains non-phishing messages, it will always predict non-phishing for all cases, regardless of the input.

This model is trained on the training data (`X_train` and `y_train`), but it does not learn any pattern; it simply memorizes the majority class.

After training, the model predicts on the test set (`X_test`). Since the classifier predicts the majority class for all test instances, all predictions are identical.

Classification Report:

	precision	recall	f1-score	support
0	0.61	1.00	0.76	2469
1	0.00	0.00	0.00	1559
accuracy			0.61	4028
macro avg	0.31	0.50	0.38	4028
weighted avg	0.38	0.61	0.47	4028

Accuracy: 0.6130

The Dummy Classifier predicts the common category (non-phishing) well, but fails completely to detect phishing. The recall is 100% because there are no false negatives, only false positives and true positives.

6 Logistic Regression for Text Classification

In this project, we will use Logistic Regression to classify text data (phishing vs non-phishing). The text data is first converted into numeric features using methods like Bag of Words and TF-IDF. Logistic regression is then trained to predict whether a given text is phishing or not.

6.1 TF-IDF

TF-IDF is a simple model for converting text into numeric features. The model learns the importance of words compared to all documents.

TF-IDF is made up of two metrics:

- **Term Frequency (TF):** Measures the frequency at which a word appears in a document and helps in determining the important words in a document.

$$\text{TF} = \frac{\text{Number of times the word appears in the document}}{\text{Total number of words in the document}}$$

- **Inverse Document Frequency (IDF):** A measure of the importance of a word, given a higher weight for rare words and a lower weight for common words.

$$\text{IDF} = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing the word}}\right)$$

6.2 How to Implement a Model

1. **TF-IDF Vectorization:** The `TfidfVectorizer` from the sklearn library is used to convert the raw text data into a TF-IDF matrix, where each word is represented numerically. This function uses the parameter `max_features=150000` to limit the number of features, ensuring that the model only considers the most important words. The output matrix represents how important each word is relative to the other words across all documents.

2. **Train-Test Split:** The dataset is split into two sets: one for training (80%) and the other for testing (20%). The parameter `random_state=42` ensures that the split is reproducible.

3. **Convert Data to PyTorch Tensors:** The TF-IDF matrix is converted to PyTorch tensors with `torch.tensor` so that it can be used with the PyTorch model for training. The data is specified to be of type float with `dtype=torch.float32`.

4. **Defining the Logistic Regression Model:** The `LogisticRegressionModel` class is defined using PyTorch's `nn.Module`. The model has a linear layer `nn.Linear` that performs the transformation of the input data into a single output used for binary classification between phishing and non-phishing. A sigmoid function `nn.Sigmoid` is applied to the output of the linear layer to transform the raw output into probabilities between 0 and 1.

5. **Model Initialization and Loss Function:** The model is initialized with `input_size`, determined by the number of features in the training data (`X_train_tensor.shape[1]`).

The loss function is defined as Binary Cross-Entropy Loss (`BCELoss`), which measures the difference between the predicted probabilities and the actual labels (0 or 1). The Adam optimizer is initialized with a learning rate of 0.001, adapting the learning rate for each parameter based on the first and second moments of the gradients.

6. **Training the Model:** The model is trained over 10 epochs with a random permutation of the training data. For each batch:
   - The optimizer is zeroed (`optimizer.zero_grad()`) to clear previous gradients.
   - A forward pass is performed to compute the predictions.
   - The model computes the loss, performs a backward pass (`loss.backward()`) to compute gradients, and updates the model's weights (`optimizer.step()`).

   The loss is printed at the end of each epoch to track the model's progress.

7. **Evaluating the Model:** After training, the model is evaluated on the test set to compute accuracy and generate the classification report. The model is set to evaluation mode with `model.eval()` and no gradients are computed with `torch.no_grad()`. The output probabilities are converted to binary predictions based on a 0.5 threshold.

6.3 Results

Accuracy: 0.9553

Classification Report:

	precision	recall	f1-score	support
Legitimate	0.94	0.99	0.96	2493
Phishing	0.98	0.90	0.94	1535
accuracy			0.96	4028
macro avg	0.96	0.95	0.95	4028
weighted avg	0.96	0.96	0.95	4028

6.4 Comparison to Baseline

The DummyClassifier has a low accuracy of 61.3% and fails to predict phishing messages, with zero true positives for phishing. On the other hand, the TF-IDF with Logistic Regression model has an accuracy of 95.53%, showing good performance in classifying messages. Precision, recall, and F1-score for both classes are higher than 94%, indicating a strong performance even for the minority class.

- **Legitimate messages (class 0):** The Logistic Regression model achieves a precision of 0.94, a recall of 0.99, and an F1-score of 0.96, indicating that it effectively identifies legitimate messages with high accuracy.
- **Phishing messages (class 1):** The model achieves a precision of 0.98, a recall of 0.90, and an F1-score of 0.94, showcasing its ability to detect phishing messages accurately, even as the minority class.

Results underline the high accuracy of the TF-IDF model, which is able to distinguish phishing messages from legitimate ones, in contrast to the DummyClassifier, which basically predicts the majority class.

7 Neural Network: Simple Classifier

The Simple Classifier is a simple Neural Network. The text data is converted into numeric features, and a PyTorch-based model is trained to make predictions.

7.1 How the Model Works

Step 1: Model Definition

A simple feedforward neural network (`SimpleClassifier`) is defined using PyTorch. It consists of:

- An **input layer** that takes numeric features as input.
- A **fully connected layer** that maps the input features to the number of output classes.
- An **output layer** that produces raw logits (unnormalized scores). These logits are used by the loss function (`nn.CrossEntropyLoss`), which internally applies softmax to calculate probabilities.

Step 2: Data Preparation

The dataset is split into train and test sets. The numeric features are converted into PyTorch tensors, and the labels are converted into numeric form using TF-IDF, compatible with PyTorch's loss functions.

Step 3: Model Initialization

In this model:

- **Input Dimension:** Set to the number of features in the training dataset, represented as `input_dim = X_train.shape[1]`.
- **Output Classes:** The number of unique labels from the target variable, represented as `num_classes = len(torch.unique(y))`.
- **Loss Function:** Cross-Entropy Loss, achieved with `criterion = nn.CrossEntropyLoss()`.
- **Optimizer:** The Adam optimizer, initialized with a learning rate of 0.001, using `optimizer = optim.Adam(classifier.parameters(), lr=0.001)` to update the gradients efficiently during training.

Step 4: Training the Model

At the beginning:

- Data is shuffled in every epoch using `torch.randperm(X_train.size(0))`, ensuring the model does not see the data in the same order each time, improving generalization.
- Data is processed in batches of size 32 to improve computational efficiency and speed up convergence.

For each batch:

- The model's gradients are set to zero using `optimizer.zero_grad()`.

- A forward pass computes predictions using `outputs = classifier(batch_x)`.
- The loss is calculated by comparing predictions with actual labels using `criterion(outputs, batch_y)`.
- The backpropagation step computes gradients (`loss.backward()`).
- The optimizer updates the model's weights using `optimizer.step()`.

Step 5: Evaluating the Model

- The model is evaluated in `eval` mode to avoid updating weights.
- Predictions are computed for each batch.
- Outputs are converted to discrete values using `torch.argmax`, storing them in a list.
- Accuracy and a classification report are calculated to measure performance.

7.2 Training Results

Loss for each epoch:

```
Epoch 1/10, Loss: 135.7253
...
Epoch 5/10, Loss: 67.7272
...
Epoch 10/10, Loss: 57.7541
```

These loss values indicate how much the model's predictions improve during training. A higher loss at the beginning (e.g., 135.7253 in Epoch 1) shows large mistakes, while a lower loss at the end (e.g., 57.7541 in Epoch 10) suggests substantial improvement.

Accuracy: 96.33%

Classification Report:

	precision	recall	f1-score	support
Legitimate	0.97	0.97	0.97	2493
Phishing	0.96	0.95	0.95	1535
accuracy			0.96	4028
macro avg	0.96	0.96	0.96	4028
weighted avg	0.96	0.96	0.96	4028

7.3 Comparison to Logistic Regression

The Simple Neural Network (NN) slightly outperforms the Logistic Regression model, achieving an accuracy of 96.33% compared to Logistic Regression's 95.53%.

- **Legitimate messages:** The Simple NN achieves a precision and recall of 0.97, resulting in an F1-score of 0.97, matching its performance across the majority class.

- **Phishing messages:** It achieves a precision of 0.96, a recall of 0.95, and an F1-score of 0.95, demonstrating its ability to accurately detect phishing messages.

This small performance gain demonstrates that the Simple NN can learn slightly more complex patterns in the data compared to Logistic Regression while maintaining high precision, recall, and F1-scores across both classes.

8 Fully Connected Neural Network (FCNN) for Phishing Detection

The Fully Connected Neural Network (FCNN) is a more advanced neural network, featuring:

- **Two layers:** One hidden layer and one output layer.
- **Activation function:** ReLU for non-linearity.
- **Advantages:** Better capability to learn non-linear patterns.
- **Disadvantages:** Higher computational cost compared to simpler models.

8.1 Comparison: FCNN vs SimpleClassifier (SC)

Number of Layers	1	2
Hidden Layer	None	Yes (128 neurons)
Activation	None	ReLU, Sigmoid
Output	Raw logits	Probabilities
Use Case	Multi-class tasks	Binary tasks
Computation Cost	Lower	Higher

8.2 Steps in FCNN Implementation

Step 1: TF-IDF Vectorization

- Text data is converted to numeric features using `TfidfVectorizer`, limited to 150,000 features for efficiency.

Step 2: Train-Test Split

- Data is split into 80% training and 20% testing.

Step 3: Convert Data to PyTorch Tensors

- TF-IDF matrices and labels are converted into PyTorch tensors.

Step 4: Define FCNN Architecture

- **Layer 1:** Fully connected layer maps input features to 128 hidden neurons.
- **ReLU Activation:** Introduces non-linearity after the first layer.
- **Layer 2:** Fully connected layer maps 128 neurons to a single output.
- **Sigmoid Activation:** Outputs probabilities for binary classification.

Step 5: Loss Function and Optimizer

- **Loss:** Binary Cross-Entropy Loss (BCELoss).
- **Optimizer:** Adam with a learning rate of 0.001.

Step 6: Train the Model

- Trained over 10 epochs with a batch size of 64.
- Training loss is monitored at each epoch.

Step 7: Evaluate the Model

- Predictions are made on the test set.
- A threshold of 0.5 is used to convert probabilities to binary labels.

Results

Training Loss:

- Epoch [1/10]: Loss = 0.1542
- Epoch [5/10]: Loss = 0.0022
- Epoch [10/10]: Loss = 0.0011

Accuracy: 97.67%
Classification Report:

Class	Precision	Recall	F1-Score
Legitimate	0.98	0.99	0.98
Phishing	0.98	0.96	0.97

Comparison to Other Models

- **FCNN Accuracy:** 98%
- **Simple NN Accuracy:** 96.33%
- **Logistic Regression Accuracy:** 95.53%

8.3 Conclusion

The FCNN outperforms simpler models by effectively capturing patterns in the data. It demonstrates strong performance for phishing detection, with an F1-score of 0.97 for the minority class, validating its robustness for binary classification tasks.

9 BERT: Bidirectional Encoder Representations from Transformers

BERT is a transformer-based model that generates contextualized embeddings for text. Unlike traditional techniques like TF-IDF or Bag-of-Words, BERT captures the semantic meaning of words based on their context, making it highly effective for tasks like phishing detection.

9.1 How BERT Works

Step 1: Loading the Tokenizer and Model

The `AutoTokenizer` for BERT tokenizes raw text into words or subwords. The `bert-base-uncased` model, a 12-layer version of BERT, is used without case sensitivity. The `AutoModel` loads the BERT model itself, which outputs embeddings (vector representations) for the given text. BERT uses WordPiece tokenization and Self-Attention to understand the relationships of different words in a bidirectional way (left to right and right to left).

Step 2: Text to Embedding

The function `text_to_embedding` converts text into tokens using the tokenizer and generates embeddings as follows:

- The `last_hidden_state` from the model output represents each token in the text after it has passed through the model.
- `mean(dim=1)` computes the average of all token embeddings, resulting in a single vector representation for the entire text.

The embeddings are stored in a list, `embedded_data`, and later converted to a PyTorch tensor.

Step 3: Defining the Classifier Model

A simple neural network model is defined with one fully connected (Dense) layer. This layer maps the embeddings to the number of output classes.

Step 4: Loss and Optimizer Definition

- Loss Function: `CrossEntropyLoss`, suitable for multi-class classification problems.
- Optimizer: Adam optimizer is used for efficient model training.

Step 5: Training Loop

The training loop divides the data into batches and updates the model's weights using backpropagation:

- Forward pass: Predictions are computed for each batch.

- Loss Calculation: The loss is calculated by comparing predictions with actual labels.
- Backpropagation: Gradients are computed and used to update the model's weights.

Step 6: Evaluation and Classification Report

After training, the model's performance is evaluated on the test set. Predictions are stored, and a classification report is generated.

Classification Report:

	precision	recall	f1-score	support
Legitimate	0.97	0.97	0.97	2493
Phishing	0.95	0.95	0.95	1535
accuracy			0.96	4028
macro avg	0.96	0.96	0.96	4028
weighted avg	0.96	0.96	0.96	4028

9.2 Conclusion

The BERT-based model achieves high accuracy and balanced performance across both classes, showcasing its ability to handle complex textual data in phishing detection. However, its computational cost might outweigh its benefits in scenarios where simpler models (e.g., Logistic Regression with TF-IDF) already perform well. For tasks requiring deeper contextual understanding or handling highly imbalanced datasets, BERT is an excellent choice.

10 BERT + FCNN

The BERT + FCNN approach leverages the power of BERT embeddings for contextual understanding, combined with a more advanced Fully Connected Neural Network (FCNN) to enhance classification performance. This setup introduces multiple hidden layers with non-linear activation functions (ReLU), enabling the model to learn more complex patterns in the data.

10.1 How BERT + FCNN Works

1. Embedding Extraction

- BERT (`bert-base-uncased`) generates embeddings for each email or SMS.
- Tokenized text is passed through the BERT model to extract the mean of the last hidden state as the sentence-level embedding.
- These embeddings serve as input features for the FCNN.

2. FCNN Architecture

- Fully connected input layer mapping the BERT embeddings to a hidden layer of size 128.
- ReLU activation introduces non-linearity.
- Second hidden layer with 64 neurons, followed by another ReLU activation.
- Output layer maps to the number of classes (legitimate or phishing).

Loss Function and Optimizer: Cross-Entropy Loss and Adam optimizer are used for efficient training.

3. Training and Evaluation

- Model is trained for 10 epochs with a batch size of 32.
- Evaluation metrics such as accuracy, precision, recall, and F1-score are calculated.

Classification Report:

	precision	recall	f1-score	support
Legitimate	0.98	0.99	0.98	2493
Phishing	0.98	0.97	0.97	1535
accuracy			0.98	4028
macro avg	0.98	0.98	0.98	4028
weighted avg	0.98	0.98	0.98	4028

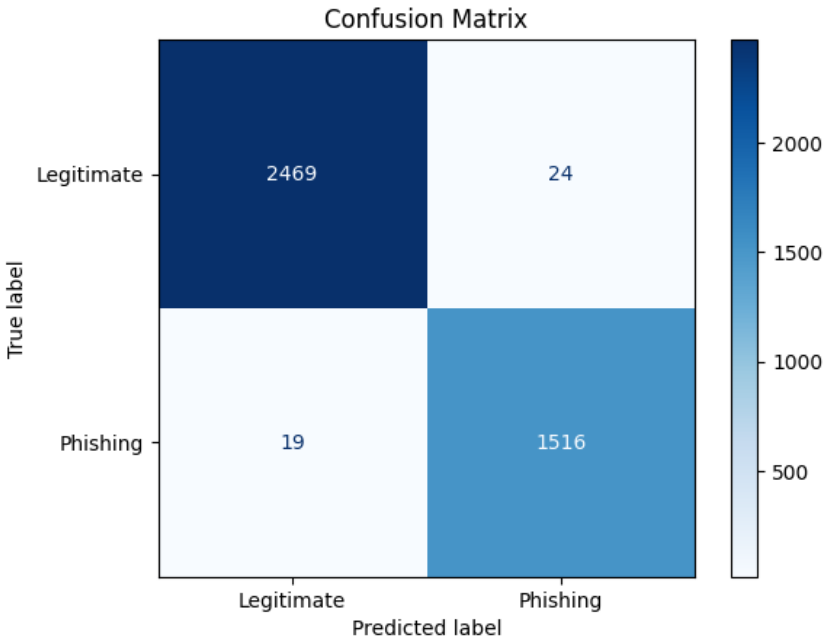

Figure 3: Confusion Matrix for BERT + FCNN Model

10.2 Strengths of BERT + FCNN

- Enhanced Learning: Multiple hidden layers and ReLU activations allow the model to learn complex relationships in the data.
- Balanced Performance: Achieves high precision, recall, and F1-scores for both phishing and legitimate messages, with an accuracy of 98%.
- Contextualized Embeddings: Leverages BERT's ability to capture the semantic meaning of text, improving classification accuracy.

10.3 Weaknesses of BERT + FCNN

- Higher Computational Cost: The combination of BERT embeddings and a deeper FCNN increases the computational resources required for training and inference.
- Marginal Improvement: While it outperforms simpler models, the improvement is modest relative to the added complexity.