BUILDING A SMARTER AI-POWERED SPAM CLASSIFIER

Phase 5: Project Documentation

Student Name: Sai Arjunaa A

College: Madras Institute of Technology, Chennai

PROBLEM STATEMENT:

In today's digital age, email communication plays a pivotal role in personal and professional interactions. However, the exponential rise in spam emails has become a significant concern, leading to productivity loss, security risks, and user frustration. The need for an efficient and accurate spam classifier powered by artificial intelligence (AI) has never been more critical.

The objective of this project is to develop an advanced AI-based spam classifier capable of accurately distinguishing between legitimate emails and spam messages. The classifier should exhibit high precision and recall rates while maintaining a low false-positive rate, ensuring that legitimate emails are not erroneously marked as spam.

Key Challenges:

1. Diverse Types of Spam Content: Spam emails come in various forms, including phishing attempts, marketing messages, malicious attachments, and more. The classifier must be robust enough to recognize and categorize different types of spam content accurately.
2. Evolving Spam Techniques: Spammers continually adapt their tactics to bypass traditional filters. The AI classifier must be able to adapt and learn from new patterns and trends in spam content.
3. Imbalanced Data: Obtaining a balanced dataset with an equal distribution of spam and non-spam emails is challenging. The classifier should be able to handle imbalanced data to prevent bias in its predictions.
4. Real-time Processing: The classifier should have the capability to process emails in real-time to provide timely protection for users.
5. Multi-language Support: As email communication is global, the classifier should be able to effectively classify emails written in different languages.
6. User Customization and Feedback Loop: Allowing users to customize the classifier's behavior (e.g., marking false positives/negatives) and incorporating a feedback loop for continuous learning and improvement.
7. Security and Privacy: Ensuring that the classifier does not compromise user privacy and that sensitive information within emails is not accessed or stored.
8. Scalability: The system should be scalable to handle a large volume of incoming emails, especially in enterprise-level applications.
9. Integration with Email Platforms: The classifier should be compatible with popular email platforms (e.g., Gmail, Outlook) and easily integratable into existing email systems.

10. Model Explainability: Providing insights into the classifier's decision-making process to build user trust and allow for transparency in its operation.

By addressing these challenges, the goal is to develop an AI spam classifier that enhances email security, reduces the risk of phishing attacks, and improves overall email communication efficiency for users across various domains.

DESIGN AND THINKING

1. **Data Collection:** We will need a dataset containing labeled examples of spam and nonspam messages. We can use a Kaggle dataset for this purpose.
   Dataset Link: https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset

2. **Data Preprocessing:** It involves cleaning and preparing the text data to ensure that it is suitable for analysis and modeling. Key steps for data preprocessing in a spam classifier are:
   i. Lowercasing: Convert all text to lowercase. This ensures that the classifier doesn't treat "Hello" and "hello" as different words.
   ii. Removing Punctuation: Remove any special characters or punctuation marks from the text. These do not typically contribute significantly to the identification of spam.
   iii. Tokenization: Split the text into individual words or tokens. This step is essential for building features from the text data.
   iv. Removing Stop Words: Remove common words (e.g., "the", "is", "and") known as stop words. These words do not carry much information for the classifier and can be safely removed.
   v. Stemming or Lemmatization: Reduce words to their root form. For example, "running", "ran", and "runs" might all be reduced to "run". Stemming and lemmatization help reduce the dimensionality of the feature space.
   vi. Handling Numerical Values and Symbols: Decide how to handle numerical values and symbols. In some cases, they may be relevant (e.g., in identifying phone numbers in spam messages), while in others, they can be removed.
   vii. Handling HTML Tags: If the data contains HTML tags (common in email data), they should be stripped out to ensure they do not interfere with the analysis.
   viii. Handling URLs: Decide whether to keep or remove URLs. They can be a source of noise in the text, but in some cases, they may contain information relevant to spam detection.

ix. Handling Email Addresses: Similar to URLs, decide whether to keep or remove email addresses from the text data.
x. Handling Emojis and Special Characters: Decide how to handle emojis and other special characters. They can be informative in some cases, but may need special treatment depending on the context.
xi. Dealing with Imbalanced Data: If the dataset is imbalanced (i.e., there are significantly more non-spam than spam examples or vice versa), consider techniques like resampling or using different evaluation metrics.
xii. Vectorization: Convert the preprocessed text data into numerical vectors. Techniques like TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings (e.g., Word2Vec, GloVe) can be used.

3. **Feature Extraction:** After tokenization, the tokens are converted to numerical features. The following techniques can be used for this purpose:
   i. TF-IDF (Term Frequency-Inverse Document Frequency): TF-IDF is a widely used technique in text classification tasks, including spam classification. It assigns weights to terms based on their frequency in a document relative to the entire corpus. Terms that are common in spam but rare in legitimate emails receive higher weights.
   ii. Binary Encoding: Binary encoding can be effective for spam classification because it captures the presence or absence of specific terms. In spam classification, the presence of certain words or phrases can be indicative of spam.
   iii. Count Vectorization: Count vectorization simply counts the frequency of each term in a document. This can be effective for identifying spam based on the occurrence of specific keywords or patterns commonly found in spam messages.
   iv. N-grams: N-grams capture sequences of words, which can be useful in identifying certain patterns or phrases commonly associated with spam. For example, phrases like "limited time offer" or "click here now" are often indicative of spam.
   v. Word Embeddings (Word2Vec, GloVe): Word embeddings capture semantic relationships between words. While not as directly interpretable as some other methods, they can be effective at capturing context and meaning, which can aid in distinguishing spam from legitimate messages.
   vi. Doc2Vec: Doc2Vec provides a way to represent entire documents as fixed-length vectors. This can be useful for capturing the overall context and style of spam messages.

Among these methods, TF-IDF, Count Vectorization, and N-grams are effective for spam classification. They allow the classifier to focus on specific terms and patterns that are indicative of spam. However, it's not uncommon to experiment with multiple methods to see which one performs best on a specific dataset.

4. **Model Selection:** We can experiment with various machine learning algorithms such as Naive Bayes, Support Vector Machines, and more advanced techniques like deep learning using neural networks. Commonly used machine learning algorithms for spam classifiers are:

   Naive Bayes:
   Strengths:
   - Simple and easy to implement.
   - Efficient in terms of memory and computation.
   - Can handle a large number of features.

   Reasoning: Naive Bayes is a popular choice for text classification tasks like spam detection. It works by calculating probabilities based on the frequency of words in spam and non-spam emails.

   Support Vector Machines (SVM):
   Strengths:
   - Effective in high-dimensional spaces.
   - Can handle non-linear decision boundaries with kernel tricks.

   Reasoning: SVM aims to find the optimal hyperplane that separates spam and non-spam emails while maximizing the margin between the two classes.

   Logistic Regression:
   Strengths:
   - Simple and interpretable.
   - Can be regularized to prevent overfitting.

   Reasoning: Although logistic regression is a binary classification algorithm, it can be used effectively for spam detection when extended with techniques like one-vs-rest.

   Random Forest:
   Strengths:
   - Ensemble method that combines multiple decision trees for improved accuracy.
   - Can handle a large number of features and complex relationships.

Reasoning: Random Forest builds multiple decision trees and combines their outputs to make a final prediction. It is robust and can handle noisy data.

Gradient Boosting (e.g., XGBoost, LightGBM):
Strengths:

- Can achieve high accuracy and is robust to overfitting.
- Handles complex relationships and interactions between features.

Reasoning: Gradient boosting builds an ensemble of weak learners (usually decision trees) in a sequential manner, with each tree aiming to correct the errors of the previous ones.

Neural Networks:
Strengths:

- Can capture complex relationships in data.
- Deep learning models can automatically learn feature hierarchies.

Reasoning: Neural networks, especially deep learning models, can be used for spam classification. Techniques like recurrent neural networks (RNNs) or convolutional neural networks (CNNs) can be particularly effective.

K-Nearest Neighbors (KNN):
Strengths:

- Simple and easy to implement.
- Can handle non-linear decision boundaries.

Reasoning: KNN classifies data points based on the majority class of their k nearest neighbors. It can be effective for spam classification, especially with appropriate feature engineering.

Ensemble Methods (e.g., AdaBoost, Bagging):
Strengths:

- Combine multiple base models to improve overall performance.
- Can handle noisy data and reduce overfitting.

Reasoning: Ensemble methods like AdaBoost and Bagging combine multiple weak learners (usually simple models) to create a strong classifier.

5. **Evaluation:** We will measure the performance of the model used using the following metrics:
    - Accuracy: The ratio of correctly classified instances to the total number of instances. While accuracy is important, it may not be the most informative metric, especially if the dataset is imbalanced.

- Precision: The ratio of true positives to the sum of true positives and false positives. It measures the accuracy of positive predictions.
- Recall (Sensitivity or True Positive Rate): The ratio of true positives to the sum of true positives and false negatives. It measures the ability of the classifier to identify all positive instances.
- F1-Score: The harmonic mean of precision and recall. It provides a balanced evaluation of precision and recall.
- Specificity (True Negative Rate): The ratio of true negatives to the sum of true negatives and false positives. It measures the ability of the classifier to identify negative instances.
- ROC-AUC: The Area Under the Receiver Operating Characteristic Curve. It provides an aggregate measure of the classifier's ability to distinguish between the two classes.
- Confusion Matrix: A table that summarizes the performance of a classification algorithm. It shows the counts of true positives, true negatives, false positives, and false negatives.
- False Positive Rate (FPR): The ratio of false positives to the sum of false positives and true negatives. It measures the rate at which non-spam emails are incorrectly classified as spam.
- False Negative Rate (FNR): The ratio of false negatives to the sum of false negatives and true positives. It measures the rate at which spam emails are incorrectly classified as non-spam.

6. **Iterative Improvement:** We will fine-tune the model and experiment with hyperparameters to improve its accuracy. Hyperparameters are configuration settings that are set before training a machine learning model. They control the learning process but are not learned from the data. These hyperparameters need to be finetuned in order to optimize model performance. This could involve adjusting parameters like learning rates, regularization strengths, or tree depths (depending on the chosen algorithm).

Develop an advanced AI-powered spam classifier leveraging state-of-the-art natural language processing (NLP) models, specifically BERT (Bidirectional Encoder Representations from Transformers) and ALBERT (A Lite BERT).

Why BERT?:

BERT, which stands for Bidirectional Encoder Representations from Transformers, is a state-of-the-art natural language processing (NLP) model developed by Google in 2018. BERT has had a profound impact on various NLP tasks due to its ability to understand the context of words in a sentence or document. Here are some of its uses and implementations:

I. Text Classification: BERT can be used for tasks like sentiment analysis, spam detection, and topic classification. By fine-tuning a pre-trained BERT model on a specific dataset, it can learn to classify text accurately.

II. Named Entity Recognition (NER): BERT can extract entities such as names of people, organizations, and locations from text. This is useful in applications like information extraction, entity linking, and chatbots.

III. Question Answering: BERT can be used to build question-answering systems where it understands the context of a given passage and answers questions about it. This is often used in chatbots and search engines.

IV. Text Summarization: BERT can summarize long documents by extracting the most important information, making it useful in tasks like automatic summarization of news articles or research papers.

V. Language Understanding: BERT can enhance language understanding in virtual assistants and chatbots, enabling them to comprehend user queries better and provide more relevant responses.

VI. Machine Translation: BERT can be used to improve machine translation models by providing context-aware translations.

VII. Sentiment Analysis: BERT can determine the sentiment of a piece of text, helping businesses gauge customer sentiment in product reviews or social media comments.

VIII. Information Retrieval: In information retrieval systems, BERT can be used to index and search documents, making search results more accurate and context-aware.

Implementation of BERT:

I. **Pre-training:** BERT models are pre-trained on massive amounts of text data. This pre-training involves learning contextual representations of words. Pre-trained BERT models can be obtained from sources like the Hugging Face Transformers library or Google's TensorFlow Hub.

II. **Fine-tuning:** After pre-training, BERT models can be fine-tuned on specific NLP tasks. Fine-tuning involves training the model on a smaller, task-specific dataset, which could be for tasks like sentiment analysis or question answering.

III. **Tokenization:** Text data needs to be tokenized into sub-word tokens compatible with BERT's vocabulary. Special tokens like [CLS] (used for classification) and [SEP] (used to separate sentences) are added as needed.

IV. **Model Architecture:** BERT uses a transformer architecture, which consists of multiple layers of attention mechanisms and feed-forward neural networks. Pre-trained BERT models can have different sizes, including BERT Base and BERT Large, depending on the number of parameters.

V. **Inference:** Once fine-tuned, the BERT model can be used for inference on new text data to perform the specific NLP task it was trained for.

VI. **Evaluation:** The performance of the BERT-based model is evaluated using appropriate metrics for the specific task, such as accuracy, F1-score, or BLEU score for machine translation.

BERT's effectiveness and versatility have led to its widespread adoption in NLP applications, and it serves as a foundation for many subsequent advancements in the field of natural language processing.

Project Steps:

Creating a smarter AI-powered spam classifier using BERT and ALBERT involves several key steps. Here's a detailed breakdown of the process:

1. Data Collection and Preprocessing:

- Gather a diverse dataset containing labelled examples of spam and non-spam messages. This dataset should represent the types of messages the classifier will encounter in real-world scenarios.
- Preprocess the data, including text cleaning (removing special characters, HTML tags, etc.), tokenization, and splitting into training, validation, and test sets.

2. Model Selection and Preparation:

- Choose between BERT and ALBERT as the base architecture for the spam classifier.
- Download pre-trained BERT or ALBERT weights and configurations from sources like Hugging Face Transformers.
- Fine-tune the selected model on the labelled dataset. This involves training the model to adapt to the specific spam detection task.
- Experiment with different model variants and hyperparameters to optimize performance.

3. Feature Extraction:

- Extract contextual embeddings from the fine-tuned BERT or ALBERT model for each message in the dataset. These embeddings capture the semantic meaning and context of the text.
- Develop techniques to convert these embeddings into features suitable for spam classification. You may use techniques like pooling, concatenation, or attention mechanisms.

4. Model Training and Validation:

- Train the spam classifier using the transformed features. Utilize machine learning libraries like TensorFlow or PyTorch for training.

- Monitor training performance, and implement early stopping and model checkpoints to prevent overfitting.
- Validate the model's performance using the validation dataset, and fine-tune as needed.

5. Real-time Detection Integration:

- Develop integration modules for the spam classifier to enable real-time detection on various communication platforms. This may involve APIs, SDKs, or custom code.
- Ensure that the integration is seamless and minimally disruptive to users.

6. Evaluation and Benchmarking:

- Evaluate the spam classifier's performance using appropriate metrics, such as precision, recall, F1-score, and accuracy.
- Benchmark the classifier against existing spam filters or classifiers to demonstrate its superiority.
- Continuously monitor the model's performance in real-world scenarios and make adjustments as necessary.

7. Continuous Learning and Adaptation:

- Implement mechanisms for the model to continuously learn and adapt to emerging spam tactics. This may involve periodic retraining on new data or utilizing online learning techniques.

8. User Interface Development:

- Develop user-friendly interfaces for configuring and managing the spam classifier. These interfaces should allow users to customize spam detection preferences, manage whitelists and blacklists, and view spam statistics.

9. Documentation and Training:

- Create comprehensive documentation for setup, configuration, and maintenance of the spam classifier.

- Provide training to users and administrators on how to use and manage the system effectively.

10. Ethical Considerations and Compliance:

- Ensure that the spam classifier respects ethical guidelines, privacy concerns, and compliance with relevant laws and regulations.
- Implement mechanisms for handling user data and personal information responsibly.

11. Deployment and Monitoring:

- Deploy the spam classifier into production on the selected communication platforms.
- Continuously monitor its performance and user feedback, making adjustments and improvements as necessary.

Development

The solution involves using a labelled dataset containing both spam and ham messages for training and evaluation. This is the "spam.csv" file imported from the **Kaggle Spam Database**.

The steps to address this problem include:

1. **Data Visualization:** The initial step involves data visualization. We use matplotlib to view graph and pie charts for the data presented by the spam database from Kaggle. This is used to identify the Ham and Spam database inside the CSV file.

2. **Data Preprocessing:** The initial step involves data preprocessing, including the loading of the dataset and transforming the labels into numeric format for machine learning. The text data is also cleaned and prepared for further processing.

3. **Fine-tuning Model:** The selected BERT model is fine-tuned on the training data to adapt it to the specific spam classification task. Fine-tuning involves training the model on the labelled dataset, adjusting model parameters to optimize its performance.

4. **Prediction:** After fine-tuning, the BERT model is capable of making predictions on new text data. The model scores each text message, and based on these scores, a threshold is used to classify the message as either spam or ham. The higher the score, the more likely the message is classified as ham.

To achieve this, we require the following tools:

1. Python 3.X with pip
2. pandas
3. numpy
4. sklearn (scikit-learn)

To implement this project:

1. Import the "spam.csv" file previously installed from the Kaggle Spam Database.
2. Format the csv database by:

a. Renaming “v1” – “Analysis” and “v2” – “Text”.
b. Removing the three “Unnamed” columns.
c. Rename “spam” – 0 and “ham” – 1 (binary int value).

Note: These are not necessary but are performed to make the code easy to understand and more efficient.

3. Printing the raw data present in the spam database using Built-in Functions such as info(), describe() and head().
4. Display a count plot using Seaborn to visualize the distribution of text classifications (ham or spam) in the Analysis column of the dataset.
5. Create a pie chart to visually represent the distribution of ham and spam values in the Analysis column of the dataset, displaying the percentages of each category.
6. Define a function **predict()** that takes a string as input. This function:
   a. Extract text data and corresponding labels from the database.
   b. Initializes variables x and y for text data and labels.
   c. Splits the data into training and testing sets.
   d. Initiate the TFIDF vectorizer with the necessary parameters.
   e. Converts the test and train data to TFIDF features using vectorizer.
   f. Converts following labels to integers for processing.
   g. Initializes the logical regression model for binary classification.
   h. Trains the LGM on the TFIDF transformed training data.
   i. Calculate the accuracy of prediction with the trained models.
   j. After training, it predicts the label (spam or ham) for the input text using the fine-tuned model.

Code to Preprocess the Dataset

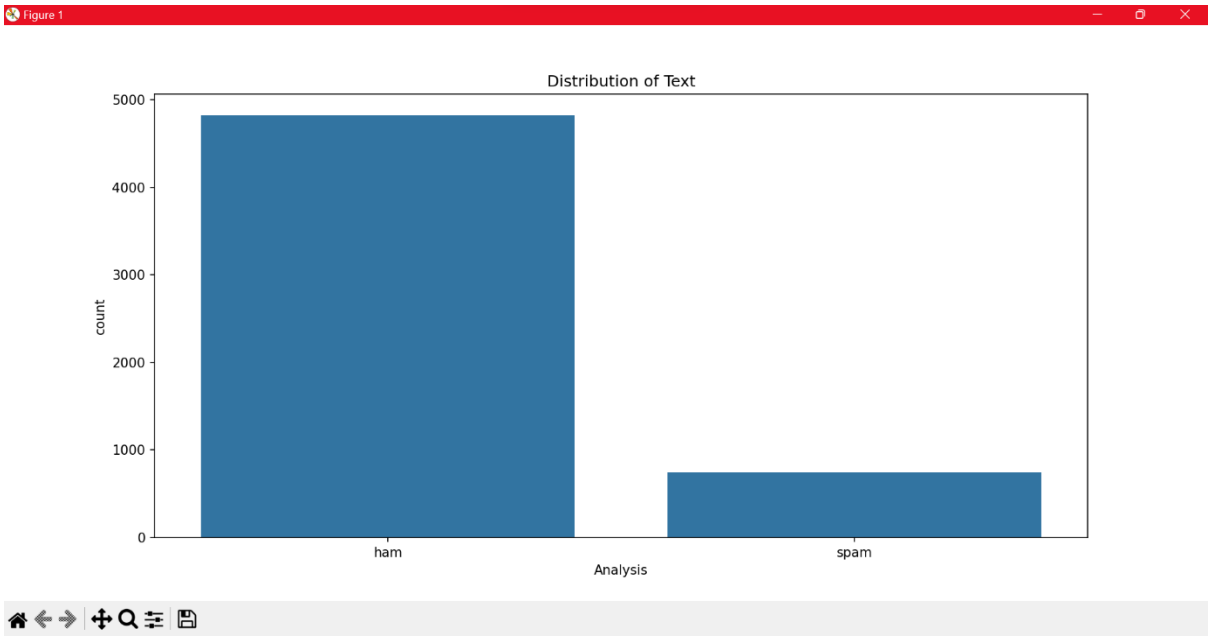
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve,
roc_auc_score
from collections import Counter
import matplotlib.pyplot as plt
import seaborn as sns

data = pd.read_csv("spam.csv", encoding = "latin-1")

ctd = ["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"]
data.drop(columns = ctd, inplace = True)
```

```
print(data) #all info
print(data.info()) #basic info
print(data.describe())

renames = {"v1" : "Analysis", "v2" : "Text"}
data.rename(columns = renames, inplace = True)
print(data.head()) #head info

sns.countplot(data = data, x = "Analysis")
plt.xlabel("Analysis")
plt.ylabel("count")
plt.title("Distribution of Text")
plt.show()

plt.pie(data['Analysis'].value_counts(), labels = ["ham", "spam"], autopct =
'%0.2f')
plt.show()

#data preprocessing
data.loc[data["Analysis"] == "spam", "Analysis"] = 0
data.loc[data["Analysis"] == "ham", "Analysis"] = 1

def predict(content: list):
    x = data["Text"]
    y = data["Analysis"]
    xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.2,
random_state = 3)

    fe = TfidfVectorizer(min_df = 1, stop_words = "english", lowercase = True)
    xtrainf = fe.fit_transform(xtrain)
    xtestf = fe.transform(xtest)
    ytrain = ytrain.astype("int")
    ytest = ytest.astype("int")

    model = LogisticRegression()
    model.fit(xtrainf, ytrain)
    ptraind = model.predict(xtrainf)
    ptestdata = model.predict(xtestf)
    actraindata = accuracy_score(ytrain, ptraind)
    actestdata = accuracy_score(ytest, ptestdata)

    ndf = fe.transform([content])
    prediction = model.predict(ndf)[0]
    return prediction
```

Output:

```
      v1                                                  v2
0    ham  Go until jurong point, crazy.. Available only ...
1    ham                      Ok lar... Joking wif u oni...
2   spam  Free entry in 2 a wkly comp to win FA Cup fina...
3    ham  U dun say so early hor... U c already then say...
4    ham  Nah I don't think he goes to usf, he lives aro...
...  ...                                                 ...
5567  spam  This is the 2nd time we have tried 2 contact u...
5568   ham              Will Ì_ b going to esplanade fr home?
5569   ham  Pity, * was in mood for that. So...any other s...
5570   ham  The guy did some bitching but I acted like i'd...
5571   ham                        Rofl. Its true to its name

[5572 rows x 2 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   v1      5572 non-null   object
 1   v2      5572 non-null   object
dtypes: object(2)
memory usage: 87.2+ KB
None
          v1                    v2
count   5572                  5572
unique     2                  5169
top      ham  Sorry, I'll call later
freq    4825                    30
  Analysis                                            Text
0      ham  Go until jurong point, crazy.. Available only ...
```

```
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   v1      5572 non-null   object
 1   v2      5572 non-null   object
dtypes: object(2)
memory usage: 87.2+ KB
None
          v1                    v2
count   5572                  5572
unique     2                  5169
top      ham  Sorry, I'll call later
freq    4825                    30
  Analysis                                            Text
0      ham  Go until jurong point, crazy.. Available only ...
1      ham                      Ok lar... Joking wif u oni...
2     spam  Free entry in 2 a wkly comp to win FA Cup fina...
3      ham  U dun say so early hor... U c already then say...
4      ham  Nah I don't think he goes to usf, he lives aro...
```



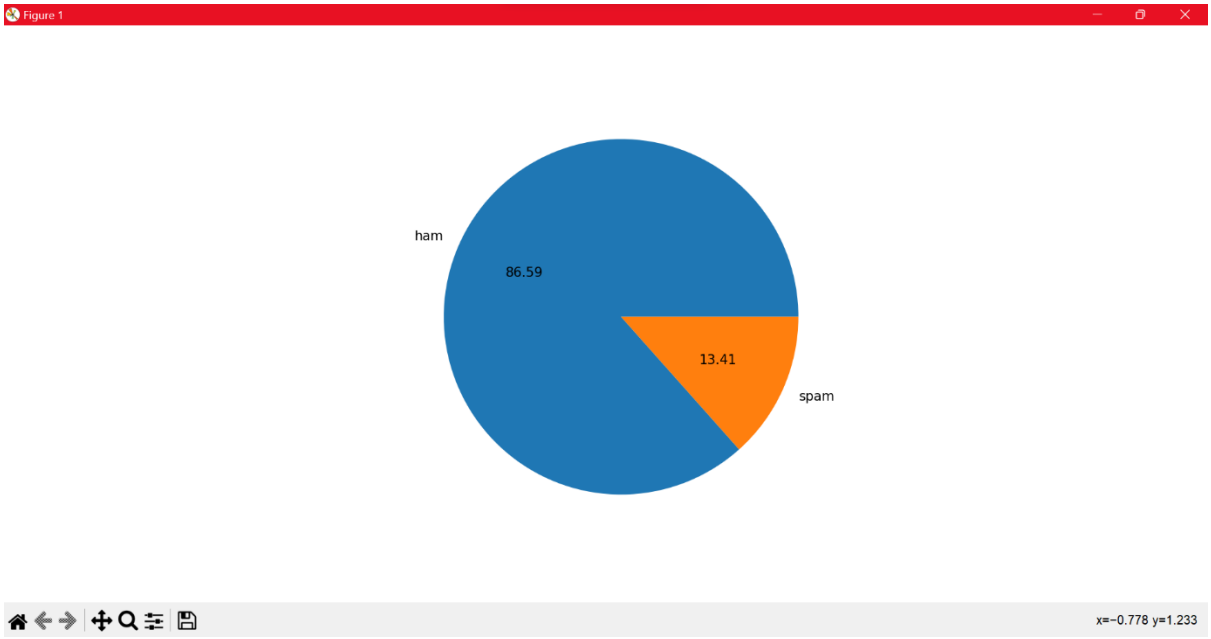


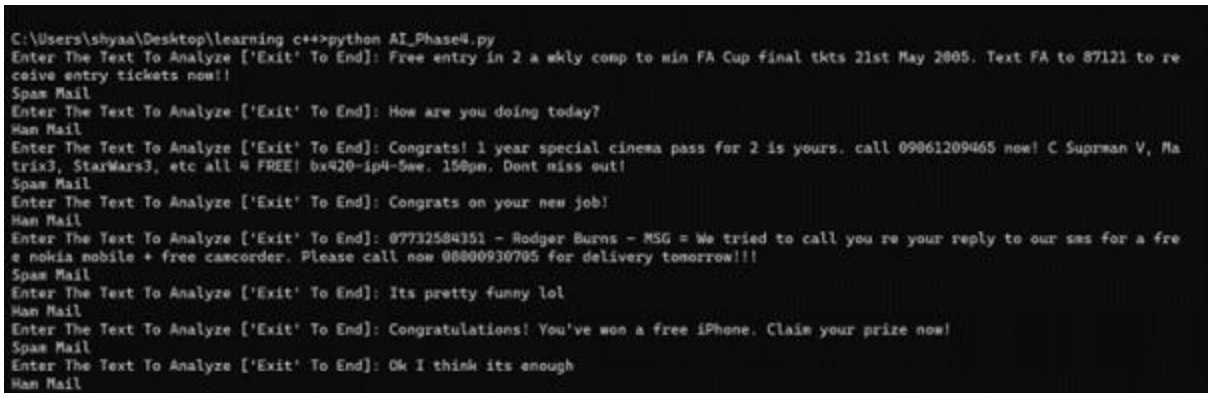
Figure 1
ham
86.59
13.41
spam
x=−0.778 y=1.233

BERT Implementation:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from transformers import BertTokenizer, BertForSequenceClassification
import torch
from torch.utils.data import TensorDataset, DataLoader

data = pd.read_csv("spam.csv", encoding = "latin-1")

data.rename(columns = {"v1" : "Analysis", "v2" : "Text"}, inplace = True)

ctd = ["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"]
data.drop(columns = ctd, inplace = True)

data["Analysis"] = data["Analysis"].map({"spam" : 0, "ham" : 1})

def predict(content: str):
    model_name = "bert-base-uncased"
    tokenizer = BertTokenizer.from_pretrained(model_name)
    model = BertForSequenceClassification.from_pretrained(model_name)

    x = data["Text"]
    y = data["Analysis"]
    xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.2,
random_state = 3)

    xtrain_tokens = tokenizer(xtrain.tolist(), padding = True, truncation =
True, return_tensors = "pt", max_length = 512)
    xtest_tokens = tokenizer(xtest.tolist(), padding = True, truncation =
True, return_tensors = "pt", max_length = 512)

    ytrain = torch.tensor(ytrain.tolist())
    ytest = torch.tensor(ytest.tolist())

    batch_size = 32

    train_dataset = TensorDataset(xtrain_tokens.input_ids,
xtrain_tokens.attention_mask, ytrain)
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)

    optimizer = torch.optim.AdamW(model.parameters(), lr = 1e-5)

    for epoch in range(3):
        model.train()
        for input_ids, attention_mask, labels in train_loader:
            optimizer.zero_grad()
```

```
            outputs = model(input_ids, attention_mask = attention_mask, labels
= labels)
            loss = outputs.loss
            loss.backward()
            optimizer.step()

    model.eval()
    with torch.no_grad():
        input_ids = tokenizer(content, padding = True, truncation = True,
return_tensors = "pt", max_length = 512).input_ids
        attention_mask = tokenizer(content, padding = True, truncation = True,
return_tensors = "pt", max_length = 512).attention_mask
        prediction = model(input_ids, attention_mask = attention_mask).logits

    return ("Spam" if prediction[0][0] > prediction[0][1] else "Ham")

while True:
    n = input("Enter The Text To Analyze ['Exit' To End]: ")
    if n.lower() == "exit":
        break
    else:
        result = predict(n)
        print(result)
```

Output:



Conclusion:

This code reads a CSV file containing text data and labels, fine-tunes a pre-trained BERT model for spam/ham classification, and allows users to input text for real-time classification using the fine-tuned model.