Project Report: SMS Spam Classification Application

# 1. Problem Statement

The increasing volume of SMS messages has led to a rise in spam messages, which are unsolicited and often malicious. Manual filtering of spam messages is inefficient and error-prone. This project aims to develop an automated system to classify SMS messages as spam or ham (not spam) using machine learning techniques, improving user experience and security.

# 2. Abstract

SMS spam messages pose a significant challenge to mobile users, leading to inconvenience and potential security risks. This project presents an SMS Spam Classification App that leverages machine learning to automatically classify messages as spam or ham. The system uses a Multinomial Naive Bayes classifier trained on a dataset of labeled SMS messages. Text preprocessing techniques such as stopwords removal and lemmatization are applied to improve model accuracy. The app is built using Python and Flask, providing a web interface for real-time classification and message history management. The system is deployed on Heroku with PostgreSQL as the backend database, ensuring scalability and accessibility.

# 3. Introduction

With the proliferation of mobile communication, SMS spam has become a widespread problem, causing annoyance and potential security threats. Traditional manual filtering methods are inadequate for handling the volume and variety of spam messages. Machine learning offers a promising solution by enabling automated, accurate classification of messages. This project develops an SMS Spam Classification App that integrates a machine learning model with a web-based interface, allowing users to classify messages in real time and maintain a history of predictions. The app uses natural language processing techniques to preprocess text data and a Naive Bayes classifier for prediction. The system supports both SQLite and PostgreSQL databases and is deployed on Heroku for public access.

# 4. Methodology

## 4.1 Data Collection and Preprocessing

A dataset of SMS messages labeled as spam or ham is used. Text preprocessing includes removal of non-alphabetic characters, conversion to lowercase, stopwords removal using NLTK, and lemmatization to reduce words to their base forms.

## 4.2 Feature Extraction

TF-IDF vectorization is applied to convert text data into numerical features suitable for machine learning.

## 4.3 Model Training

A Multinomial Naive Bayes classifier is trained on the processed dataset. The model is evaluated on training data to ensure accuracy.

## 4.4 Application Development

The Flask web framework is used to develop the backend API and web interface. The app provides endpoints for message classification, retrieving message history, and clearing stored messages.

## 4.5 Database Integration

The app supports SQLite for local development and PostgreSQL for production deployment. Messages and predictions are stored in a database table for persistence.

## 4.6 Deployment

The app is deployed on Heroku, providing a live demo accessible via a public URL.

# 5. Results

- The model achieved high accuracy on the training dataset.

- The web app successfully classifies messages in real time and stores message history.

- The database integration allows persistent storage and retrieval of messages.

- The app UI is responsive and user-friendly.

# 6. Conclusion

The SMS Spam Classification App effectively automates the detection of spam messages using machine learning. It provides a practical tool for users to filter unwanted messages, enhancing communication security. The integration of a web interface and database storage makes the system accessible and reliable. Future work can include expanding the dataset, improving the model with deep learning techniques, and adding features like multi-language support.

# 7. Future Scope

- Incorporate deep learning models such as LSTM or transformers for improved accuracy.

- Expand the dataset with more diverse and larger samples.

- Add support for multiple languages and regional dialects.

- Implement real-time SMS filtering on mobile devices.

- Enhance the web UI with additional analytics and visualization features.

# 8. Application Overview

This application is a Flask-based SMS spam classifier deployed on Heroku. It uses a machine learning model (Multinomial Naive Bayes) trained on a spam dataset to classify SMS messages as 'spam' or 'ham' (not spam). The app supports both SQLite and PostgreSQL databases for storing messages and their predictions.

# 9. Features

• API Endpoints:

- /predict (POST): Accepts a JSON payload with a 'message' field and returns the spam classification.

- /messages (GET): Returns the 10 most recent messages and their predictions from the database.

• Web UI:

- A simple interface to submit SMS messages for classification.

- Displays recent messages with color-coded predictions (red for spam, green for ham).

# 10. Architecture

• Backend: Flask app with endpoints for prediction and data retrieval.

• Model: Multinomial Naive Bayes trained on a preprocessed subset of the spam dataset.

• Database: Supports SQLite (local) and PostgreSQL (Heroku addon).

• Frontend: Basic HTML/CSS/JavaScript UI served via Flask templates.

# 11. Deployment

• Hosted on Heroku with a PostgreSQL addon.

• Uses Gunicorn as the WSGI server (configured in Procfile).

• Environment variable DATABASE\_URL manages database connection.

# 12. Testing Summary

• Tested /predict endpoint with various spam and ham messages; predictions are accurate.

• Tested /messages endpoint to retrieve saved messages.

• Verified database integration by saving and fetching messages.

• Tested web UI for message submission and displaying results with color coding.

• Memory optimizations implemented to reduce Heroku memory quota exceedance.

# 13. Known Issues

• The app occasionally exceeds Heroku memory quota, causing warnings.

• No advanced error handling for malformed requests.

• No load testing or long-term stability monitoring performed yet.

# 14. Recommendations

• Further optimize memory usage or upgrade Heroku plan for stability.

• Implement comprehensive error handling for API endpoints.

• Conduct load testing to assess performance under concurrent requests.

• Enhance UI for better user experience and additional features.

# 15. Next Steps

• Proceed with thorough testing of edge cases and performance.

• Monitor app stability and resource usage over time.

• Consider adding user authentication and message management features.