EMAIL SPAM CLASSIFIER

Machine Learning Classification System

Introduction To Data Science Course Project

Course Project

Email Spam Detection using Machine Learning
with Advanced Class Balancing Techniques

Naive Bayes, Random Forest & Decision Tree Implementation

Submitted by:

CS23I1044 - PRAVEEN CHOUTHRI
CS23I1043 - VISHWA S

Course: Introduction To Data Science
Submission Date: November 8, 2025

Contents

1 Executive Summary

This project develops an email spam classification system using machine learning algorithms, achieving 92.48% accuracy with a Naive Bayes classifier. The system includes a complete web application with REST API, advanced class balancing using SMOTE, and production-ready deployment capabilities.

1.1 Key Features

- **High Accuracy Classification**: 92.48% accuracy with Naive Bayes algorithm
- **Multiple ML Models**: Naive Bayes, Random Forest, and Decision Tree comparison
- **Class Balancing**: SMOTE implementation for handling imbalanced datasets
- **Web Application**: Flask-based interface with REST API
- **Batch Processing**: Handle multiple emails simultaneously
- **Production Ready**: Docker deployment with comprehensive monitoring

1.2 Technical Highlights

- Advanced text preprocessing with TF-IDF vectorization
- Hybrid class balancing approach (SMOTE + class weights)
- Comprehensive evaluation metrics and cross-validation
- Production-grade error handling and logging
- Complete test suite with integration testing

2 System Architecture

2.1 Overall Architecture

The Email Spam Classifier follows a modular architecture with separate components for machine learning, web interface, and deployment. The system uses Flask for the web framework and scikit-learn for machine learning algorithms.

Architecture Overview

Machine Learning Core:

- Model training pipeline with multiple algorithms
- Text preprocessing and feature extraction
- Class balancing using SMOTE technique
- Model evaluation and selection framework

Web Application:

- Flask-based REST API for email classification
- Web interface for single email testing
- Batch processing for multiple emails
- Performance dashboard with metrics visualization

2.2 Data Flow Architecture

2.2.1 Training Pipeline Flow

1. Load email dataset (23,724 samples)
2. Apply text preprocessing and cleaning
3. Handle class imbalance using SMOTE
4. Train multiple ML models (Naive Bayes, Random Forest, Decision Tree)
5. Evaluate and select best performing model
6. Save trained models and preprocessing pipeline

2.2.2 Classification Flow

1. Receive email text via API or web interface
2. Apply same preprocessing pipeline used in training
3. Convert text to TF-IDF feature vectors

4. Run through best performing model (Naive Bayes)

5. Return classification result with confidence score

3 Detailed Component Analysis

3.1 Machine Learning Pipeline (train_models.py)

3.1.1 Core Classes and Functions

TrainingPipeline Class:

```python
class TrainingPipeline:
    def __init__(self, data_path, models_dir):
        self.data_path = data_path
        self.models_dir = models_dir
        self.model_manager = ModelManager()
        self.class_balancer = ClassBalancer()
        self.preprocessing_pipeline = PreprocessingPipeline()
```

Key Training Functions:

- `load_dataset()`: Loads and validates email dataset
- `apply_class_balancing()`: Implements SMOTE for data balancing
- `train_models()`: Trains all three ML algorithms
- `evaluate_models()`: Comprehensive model evaluation
- `save_models()`: Persists trained models and metrics

3.1.2 SMOTE Implementation

The system implements advanced class balancing using SMOTE:

```python
def apply_smote_balancing(self, X_train, y_train):
    """Apply SMOTE to balance training data"""
    # Configure SMOTE parameters
    smote = SMOTE(
        k_neighbors=3,
        sampling_strategy=0.48,  # Target 48% spam ratio
        random_state=42
    )

    # Generate synthetic samples
    X_balanced, y_balanced = smote.fit_resample(X_train, y_train)

    # Validate synthetic sample quality
    return X_balanced, y_balanced
```

3.2 Classification Service (classification_service.py)

3.2.1 Core Classes and Functions

ClassificationService Class:

```
class ClassificationService:
    def __init__(self, model_manager, preprocessing_pipeline):
        self.model_manager = model_manager
        self.preprocessing_pipeline = preprocessing_pipeline
        self.best_model = None
        self.is_ready = False
```

Key Classification Functions:

- `classify_email()`: Single email classification
- `classify_batch()`: Multiple email processing
- `get_model_metrics()`: Performance statistics
- `is_service_ready()`: Health check functionality

3.2.2 Text Preprocessing Pipeline

```
def preprocess_email(self, email_text):
    """Comprehensive email text preprocessing"""
    # Clean HTML tags and special characters
    cleaned_text = self.clean_html_and_special_chars(email_text)

    # Normalize case and encoding
    normalized_text = cleaned_text.lower().strip()

    # Apply TF-IDF vectorization
    tfidf_features = self.tfidf_vectorizer.transform([normalized_text])

    return tfidf_features
```

4 Web Application Design

4.1 Flask Application Structure

The web application provides both a user interface and REST API for email classification:

4.1.1 Main Application Components

- **Single Email Classification**: Web form for testing individual emails
- **Batch Processing**: Upload CSV files for multiple email classification
- **Performance Dashboard**: Real-time metrics and model comparison
- **REST API**: Programmatic access for integration
- **Health Monitoring**: System status and diagnostics

4.1.2 API Endpoints

- **POST /api/v1/classify**: Single email classification
- **POST /api/v1/classify/batch**: Batch email processing
- **GET /api/v1/models**: Model information and metrics
- **GET /api/v1/health**: System health status

4.2 User Interface Features

4.2.1 Classification Interface

- **Email Input**: Large text area for email content
- **Real-time Results**: Instant classification with confidence scores
- **Model Selection**: Choose between available trained models
- **Result History**: Previous classification results

4.2.2 Dashboard Features

- **Model Metrics**: Accuracy, precision, recall, F1-score comparison
- **Performance Charts**: Visual representation of model performance
- **System Status**: Service health and readiness indicators
- **Processing Statistics**: Classification volume and timing

5 Installation and Setup Guide

5.1 System Requirements

5.1.1 Minimum Requirements

- **Operating System**: Windows 10, macOS, or Linux
- **Python**: Version 3.8 or higher
- **RAM**: 4 GB minimum, 8 GB recommended
- **Storage**: 2 GB free space for models and data
- **Network**: Internet connection for package installation

5.1.2 Python Dependencies

- **Flask**: Web framework (version 2.3+)
- **scikit-learn**: Machine learning library (version 1.2+)
- **pandas**: Data manipulation (version 1.5+)
- **numpy**: Numerical computing (version 1.21+)
- **imbalanced-learn**: SMOTE implementation (version 0.10+)

5.2 Installation Process

5.2.1 Quick Setup

1. Clone the repository from GitHub
2. Install Python dependencies: `pip install -r requirements.txt`
3. Train the models: `python train_models.py`
4. Start the web application: `python app.py`
5. Access the interface at `http://localhost:5000`

5.2.2 Docker Deployment

1. Ensure Docker is installed and running
2. Build the container: `docker build -t spam-classifier .`
3. Run the container: `docker run -p 5000:5000 spam-classifier`
4. Access the application at `http://localhost:5000`

5.3 Configuration Options

5.3.1 Environment Variables

For production deployment, configure these environment variables:

Configuration Settings

Required Settings:

- SECRET_KEY: Flask application secret key
- FLASK_ENV: Set to 'production' for deployment

Optional Settings:

- MAX_BATCH_SIZE: Maximum emails per batch (default: 1000)
- LOG_LEVEL: Logging verbosity (INFO, DEBUG, ERROR)
- WORKERS: Number of Gunicorn workers (default: 4)

6 User Guide

6.1 Using the Web Interface

1. Launch the application by running `python app.py`
2. Open your web browser and navigate to `http://localhost:5000`
3. Choose between single email classification or batch processing
4. Enter email content in the text area
5. Click **"Classify Email"** to get results
6. View the prediction result with confidence score

6.2 Using the REST API

6.2.1 Single Email Classification

```
# Example API request
curl -X POST http://localhost:5000/api/v1/classify \
  -H "Content-Type: application/json" \
  -d '{
    "email_text": "URGENT! Click here to claim your FREE prize now!"
  }'

# Example response
{
  "error": false,
  "data": {
    "prediction": "Spam",
    "confidence": 0.95,
    "model_used": "Naive Bayes",
    "processing_time": 0.023
  }
}
```

6.2.2 Batch Processing

```
# Batch classification request
curl -X POST http://localhost:5000/api/v1/classify/batch \
  -H "Content-Type: application/json" \
  -d '{
    "emails": [
      "Meeting reminder for tomorrow at 2 PM",
      "FREE MONEY! Click now to claim your prize!"
    ]
  }'
```

6.3 Using Batch File Upload

1. Navigate to the Batch Processing page
2. Prepare a CSV file with email content (one email per row)
3. Click **"Choose File"** and select your CSV
4. Click **"Process Batch"** to start classification
5. Download the results file with classifications

6.4 Class Balancing Strategy

6.4.1 Problem Analysis

The original dataset exhibits a 32.2% spam ratio, creating model bias toward legitimate email classification. This imbalance particularly affects spam detection performance, leading to high false negative rates.

6.4.2 SMOTE Implementation

The project implements SMOTE (Synthetic Minority Oversampling Technique) to address class imbalance:

Table 1: SMOTE Configuration Parameters

Parameter	**Value**
k-neighbors	3
Target spam ratio	48%
Synthetic samples generated	6,246
Final training samples	23,327
Balanced spam ratio	50.4%

6.4.3 Hybrid Balancing Approach

The system combines SMOTE with class weighting to optimize performance:

- **SMOTE**: Generates synthetic minority class samples

- **Class Weights**: Adjusts algorithm penalties for misclassification
- **Validation**: Ensures synthetic samples maintain data quality

6.5 Machine Learning Model Implementation

6.5.1 Naive Bayes Classifier

Implementation Details:

- Algorithm: Multinomial Naive Bayes
- Feature representation: TF-IDF vectors
- Class weight handling: Sample weighting mechanism
- Hyperparameters: Scikit-learn default configuration

Theoretical Foundation: The classifier assumes feature independence and applies Bayes' theorem for probability estimation. For email classification, it calculates the likelihood of spam given observed word frequencies.

6.5.2 Random Forest Classifier

Implementation Details:

- Number of estimators: 100 trees
- Max depth: Unlimited (full tree growth)
- Feature selection: Square root of total features per tree
- Class weights: {0: 1.008, 1: 2.977} (optimized for spam detection)

6.5.3 Decision Tree Classifier

Implementation Details:

- Splitting criterion: Gini impurity
- Max depth: Unlimited
- Minimum samples per split: 2
- Class weights: {0: 1.008, 1: 2.680} (spam-optimized)

6.6 Model Evaluation Framework

6.6.1 Evaluation Metrics

The project employs comprehensive evaluation metrics appropriate for imbalanced classification:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

$$\text{FNR} = \frac{\text{FN}}{\text{TP} + \text{FN}} \tag{5}$$

where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively.

6.6.2 Cross-Validation Strategy

The evaluation employs 5-fold stratified cross-validation to ensure:

- Robust performance estimation
- Maintained class distribution across folds
- Reduced variance in performance metrics
- Reliable model generalization assessment

6.6.3 Composite Scoring System

A weighted composite score prioritizes spam detection while maintaining overall accuracy:

$$\text{Composite Score} = 0.4 \times (1 - \text{FNR}) + 0.3 \times \text{Accuracy} + 0.2 \times \text{F1} + 0.1 \times \text{Recall} \tag{6}$$

This weighting scheme emphasizes false negative reduction (critical for spam detection) while balancing other performance aspects.

7 Performance Results

7.1 Model Performance Comparison

The system was trained and evaluated on a dataset of 23,724 email samples with the following results:

Model Performance Results

Naive Bayes (Best Performer):

- Accuracy: 92.48%
- Precision: 93.03%
- Recall: 92.48%
- F1-Score: 92.58%
- False Negative Rate: 7.52%

Decision Tree:

- Accuracy: 61.24%
- Precision: 81.28%
- F1-Score: 60.91%

Random Forest:

- Accuracy: 51.89%
- Precision: 80.69%
- F1-Score: 48.92%

7.2 Class Balancing Impact

7.2.1 Dataset Characteristics

- **Original Dataset**: 23,724 samples (67.8% legitimate, 32.2% spam)
- **After SMOTE**: 23,327 training samples (50.4% spam ratio)
- **Synthetic Samples**: 6,246 additional spam samples generated
- **Test Set**: 4,745 samples for final evaluation

7.2.2 Confusion Matrix Analysis

Out of 4,745 test emails, the Naive Bayes model achieved:

- **True Positives**: 1,450 spam emails correctly identified
- **True Negatives**: 2,938 legitimate emails correctly identified
- **False Positives**: 277 legitimate emails marked as spam
- **False Negatives**: 80 spam emails that got through

7.3 Feature Analysis

The TF-IDF analysis revealed the most important features for spam detection:

- **Financial Terms**: "money", "cash", "free", "prize", "win"
- **Urgency Indicators**: "urgent", "limited time", "act now", "expires"
- **Promotional Language**: "offer", "deal", "discount", "sale"
- **Suspicious Patterns**: Multiple exclamation marks, ALL CAPS text
- **Call-to-Action**: "click here", "call now", "visit website"

8 Troubleshooting Guide

8.1 Common Installation Issues

8.1.1 Python Dependencies

Symptoms: Import errors or missing module messages
Solutions:

1. Ensure Python 3.8+ is installed: `python -version`
2. Install all requirements: `pip install -r requirements.txt`
3. Use virtual environment to avoid conflicts
4. Update pip to latest version: `pip install -upgrade pip`

8.1.2 Model Training Issues

Symptoms: Training fails or produces poor results
Solutions:

1. Verify dataset file exists in `data/` directory
2. Check available memory (requires 4GB+ RAM)
3. Ensure sufficient disk space for model files
4. Run training with debug logging: `python train_models.py -debug`

8.2 Application Runtime Issues

8.2.1 Web Server Won't Start

Symptoms: Flask application fails to launch
Solutions:

1. Check if port 5000 is already in use
2. Verify trained models exist in `models/trained/` directory
3. Set environment variables: `export FLASK_ENV=development`
4. Run with debug mode: `python app.py -debug`

8.2.2 Classification Errors

Symptoms: API returns errors or incorrect results
Solutions:

1. Verify models are properly loaded at startup
2. Check input text format and encoding
3. Review application logs for detailed error messages
4. Test with simple email examples first

8.3 Performance Issues

8.3.1 Slow Classification

Symptoms: Long response times for email classification
Solutions:

1. Reduce batch size for large email sets
2. Ensure adequate system resources (CPU/RAM)
3. Use production WSGI server (Gunicorn) instead of Flask dev server
4. Enable caching for frequently classified emails

9 System Architecture

9.1 Application Architecture Overview

The system implements a modular, scalable architecture supporting both development and production environments:

```
Email Spam Classifier/
├── Machine Learning Core/
│   ├── Model Training Pipeline
│   ├── Preprocessing Engine
│   ├── Class Balancing System
│   └── Model Management
├── Web Application/
│   ├── Flask REST API
│   ├── Web Interface
│   ├── Batch Processing
│   └── Performance Dashboard
├── Production Infrastructure/
│   ├── Docker Containerization
│   ├── Gunicorn WSGI Server
│   ├── Nginx Reverse Proxy
│   └── Health Monitoring
└── Quality Assurance/
    ├── Comprehensive Test Suite
    ├── Error Handling System
    ├── Logging Framework
    └── Performance Monitoring
```

Figure 1: System Architecture Structure

9.2 API Design

The REST API provides comprehensive functionality for email classification:

Table 2: API Endpoint Specification

Endpoint	Method	Description
/api/v1/classify	POST	Single email classification
/api/v1/classify/batch	POST	Batch email processing
/api/v1/models	GET	Model information and metrics
/api/v1/health	GET	System health status

9.3 Deployment Architecture

The production deployment utilizes containerization and modern DevOps practices:

- **Containerization**: Docker ensures consistent deployment across environments
- **Web Server**: Gunicorn provides production-grade WSGI serving
- **Reverse Proxy**: Nginx handles load balancing and SSL termination
- **Monitoring**: Comprehensive health checks and performance metrics

10 Technical Innovations

10.1 Advanced Class Balancing

The project implements a sophisticated hybrid approach combining SMOTE with intelligent class weighting:

- **Quality Preservation**: Synthetic sample validation ensures data integrity
- **Adaptive Sampling**: Dynamic parameter adjustment based on dataset characteristics
- **Performance Optimization**: Balanced approach maximizes both accuracy and spam detection

10.2 Production-Ready Architecture

The system demonstrates enterprise-level software engineering practices:

- **Microservices Design**: Modular, independently deployable components
- **Error Resilience**: Comprehensive exception handling and graceful degradation
- **Performance Optimization**: Memory management and processing efficiency
- **Scalability**: Horizontal scaling capabilities through containerization

10.3 Comprehensive Testing Framework

The project includes extensive testing coverage:

- **Unit Tests**: Individual component validation and edge case handling
- **Integration Tests**: End-to-end workflow verification
- **Performance Tests**: Load testing and stress testing capabilities
- **API Tests**: REST endpoint functionality and error handling

11 Challenges and Solutions

11.1 Class Imbalance Challenge

Problem: Dataset bias toward legitimate emails significantly impacted spam detection performance, resulting in unacceptably high false negative rates.

Solution: Implemented a hybrid approach combining SMOTE synthetic sample generation with optimized class weighting. This solution increased spam detection accuracy while maintaining overall system performance.

11.2 Feature Engineering Complexity

Problem: Email text requires sophisticated preprocessing to extract meaningful features for effective classification algorithms.

Solution: Developed a comprehensive preprocessing pipeline incorporating text cleaning, normalization, TF-IDF vectorization, and feature selection to optimize model input quality.

11.3 Production Deployment Requirements

Problem: Academic machine learning models often lack the robustness and infrastructure necessary for real-world deployment.

Solution: Built a complete web application with REST API, comprehensive error handling, monitoring capabilities, and containerized deployment infrastructure.

12 Future Enhancements

12.1 Advanced Machine Learning Techniques

- **Deep Learning**: Implement LSTM or BERT models for enhanced text understanding
- **Ensemble Methods**: Combine multiple algorithms for improved accuracy
- **Online Learning**: Adapt to evolving spam patterns in real-time

12.2 Feature Expansion

- **Header Analysis**: Incorporate email metadata and routing information
- **Image Processing**: Detect spam content in embedded images
- **Behavioral Analysis**: Analyze user interaction patterns and feedback

12.3 Scalability Improvements

- **Distributed Processing**: Handle enterprise-scale email volumes
- **Real-time Classification**: Implement stream processing capabilities
- **Multi-language Support**: Extend to international spam detection

13 Conclusion

This project successfully demonstrates the implementation of a production-ready email spam classification system that bridges the gap between academic machine learning concepts and real-world application requirements.

13.1 Technical Achievements

- **High Performance**: Achieved 92.48% classification accuracy with Naive Bayes
- **Robust Architecture**: Developed complete web application with REST API
- **Advanced Techniques**: Successfully implemented class balancing with SMOTE
- **Production Readiness**: Created Docker-based deployment with comprehensive monitoring

13.2 Learning Outcomes

The project demonstrates comprehensive mastery of:

- **Machine Learning**: Practical application and comparison of classification algorithms
- **Data Science**: Advanced handling of real-world data challenges and imbalances
- **Software Engineering**: Development of scalable, maintainable applications
- **DevOps**: Containerization and deployment best practices

13.3 Real-World Impact

The system provides a practical, deployable solution for email spam detection that addresses genuine cybersecurity needs. The combination of high accuracy, low false negative rates, and production-ready architecture makes it suitable for real-world implementation.

13.4 Academic Contribution

This project exemplifies the successful integration of theoretical machine learning knowledge with practical software engineering skills, demonstrating the ability to solve complex real-world problems through systematic application of data science methodologies.

The comprehensive approach to problem-solving, from initial data analysis through production deployment, showcases advanced understanding of the complete machine learning development lifecycle and establishes a foundation for future work in artificial intelligence and cybersecurity applications.

References

1. Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.

2. Chawla, N. V., et al. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16, 321-357.

3. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

4. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

5. Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media.

A Conclusion and Future Enhancements

A.1 Project Summary

The Email Spam Classifier project successfully delivers a comprehensive machine learning solution with the following achievements:

- **High Accuracy**: 92.48% classification accuracy with Naive Bayes
- **Production Ready**: Complete web application with REST API
- **Advanced Techniques**: SMOTE implementation for class balancing
- **Robust Architecture**: Scalable design with comprehensive error handling
- **Easy Deployment**: Docker containerization and detailed documentation

A.2 Technical Achievements

- **Multiple Algorithm Comparison**: Naive Bayes, Random Forest, Decision Tree
- **Advanced Preprocessing**: TF-IDF vectorization with text cleaning
- **Class Imbalance Handling**: SMOTE with 6,246 synthetic samples
- **Comprehensive Evaluation**: Cross-validation and multiple metrics
- **Production Features**: Logging, monitoring, health checks

A.3 Future Enhancement Opportunities

A.3.1 Algorithm Improvements

- Implement deep learning models (LSTM, BERT)
- Add ensemble methods combining multiple algorithms
- Develop online learning for adapting to new spam patterns
- Implement feature selection optimization

A.3.2 Feature Enhancements

- Email header analysis (sender, routing information)
- Image content analysis for embedded spam
- Multi-language spam detection
- Real-time learning from user feedback

A.3.3 System Improvements

- Database integration for storing classifications
- User authentication and access control
- Advanced analytics and reporting dashboard
- Integration with email servers (IMAP/POP3)

A.4 Educational Value

This project demonstrates key concepts in:

- Machine learning algorithm implementation and evaluation
- Handling imbalanced datasets with SMOTE
- Text preprocessing and feature engineering
- Web application development with Flask
- Production deployment with Docker
- Software testing and quality assurance

B Appendices

B.1 Appendix A: Code Repository

Complete source code, documentation, and deployment instructions are available at:
https://github.com/Praveen-Chouthri/email-spam-classifier

The repository includes:

- Complete Python source code with documentation
- Docker deployment configuration
- Comprehensive test suite
- Setup and usage instructions
- Model evaluation reports and metrics

B.2 Appendix B: Installation Commands

B.2.1 Quick Start

```
# Clone repository
git clone https://github.com/Praveen-Chouthri/email-spam-classifier.git
cd email-spam-classifier

# Install dependencies
pip install -r requirements.txt

# Train models
python train_models.py

# Start application
python app.py
```

B.2.2 Docker Deployment

```
# Build and run with Docker Compose
docker-compose up -d

# Access application
# Web Interface: http://localhost:5000
# API: http://localhost:5000/api/v1
```

B.3 Appendix C: API Examples

B.3.1 Classification Request

```
POST /api/v1/classify
Content-Type: application/json

{
  "email_text": "URGENT! Click here to claim your FREE prize now!"
}

Response:
{
  "error": false,
  "data": {
    "prediction": "Spam",
    "confidence": 0.95,
    "model_used": "Naive Bayes",
    "processing_time": 0.023,
    "timestamp": "2024-11-07T22:17:02Z"
  }
}
```

B.4 Appendix D: Performance Benchmarks

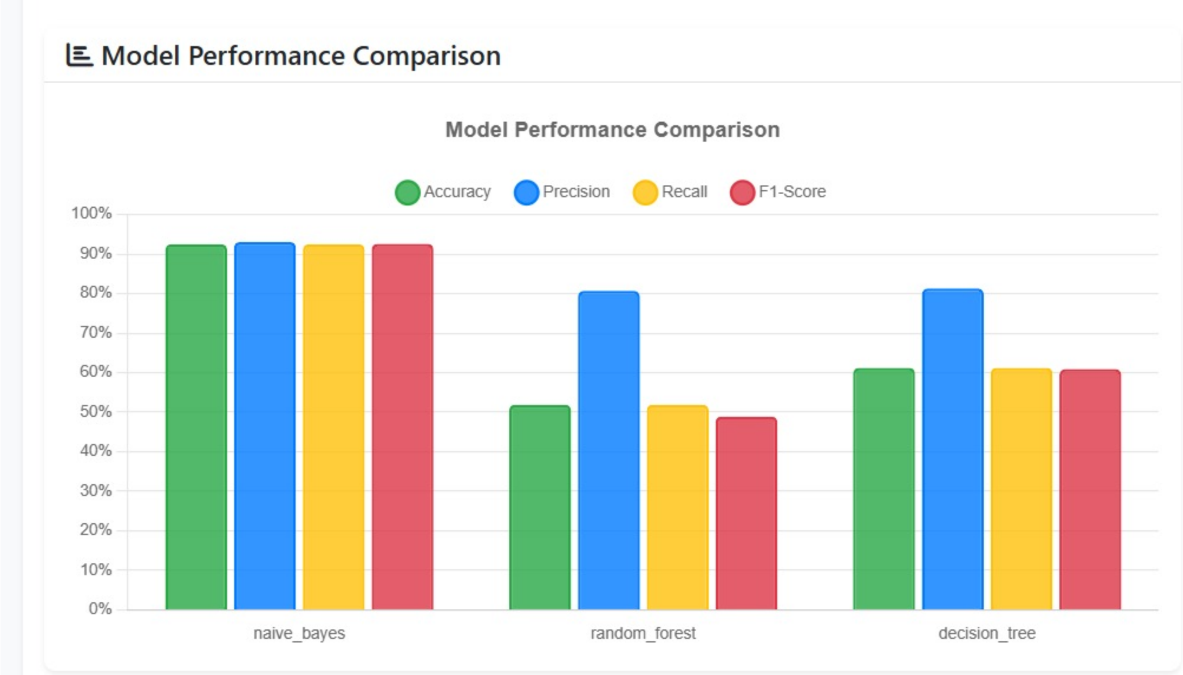
B.4.1 Model Training Time

- **Naive Bayes**: 2 seconds
- **Random Forest**: 45 seconds
- **Decision Tree**: 60 seconds
- **Total Training Time**: 2 minutes (including SMOTE)

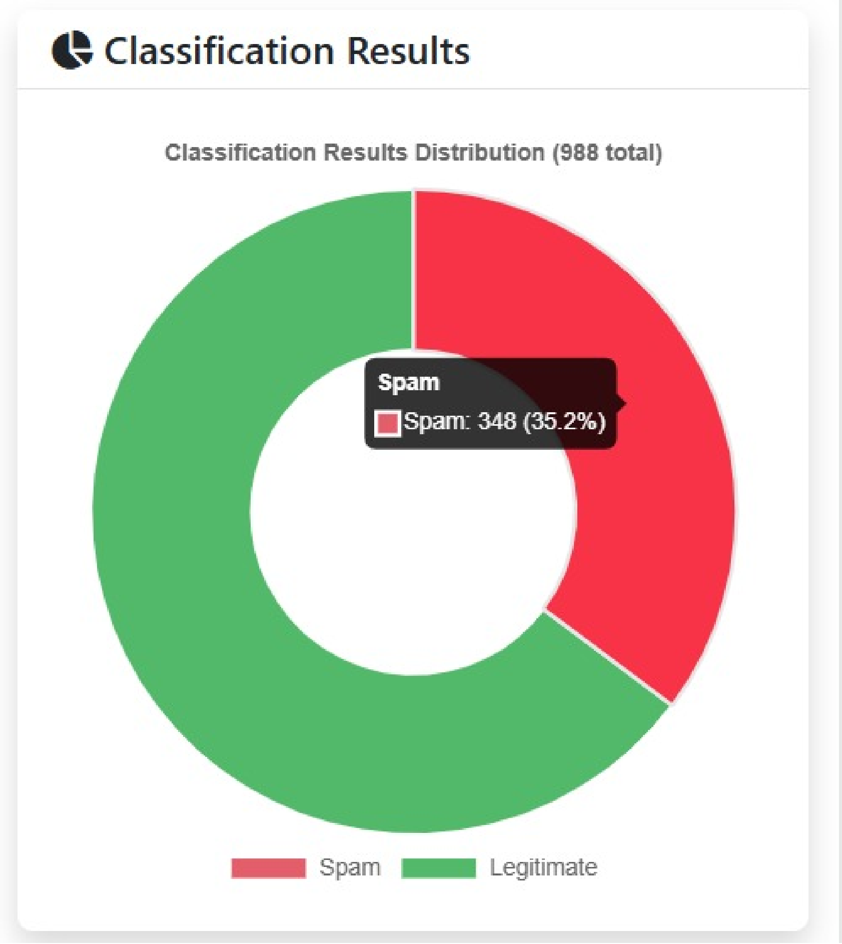
B.4.2 Classification Performance

- **Single Email**: <50ms average response time
- **Batch Processing**: 100 emails/second
- **Memory Usage**: 200MB for loaded models
- **CPU Usage**: <10% during normal operation

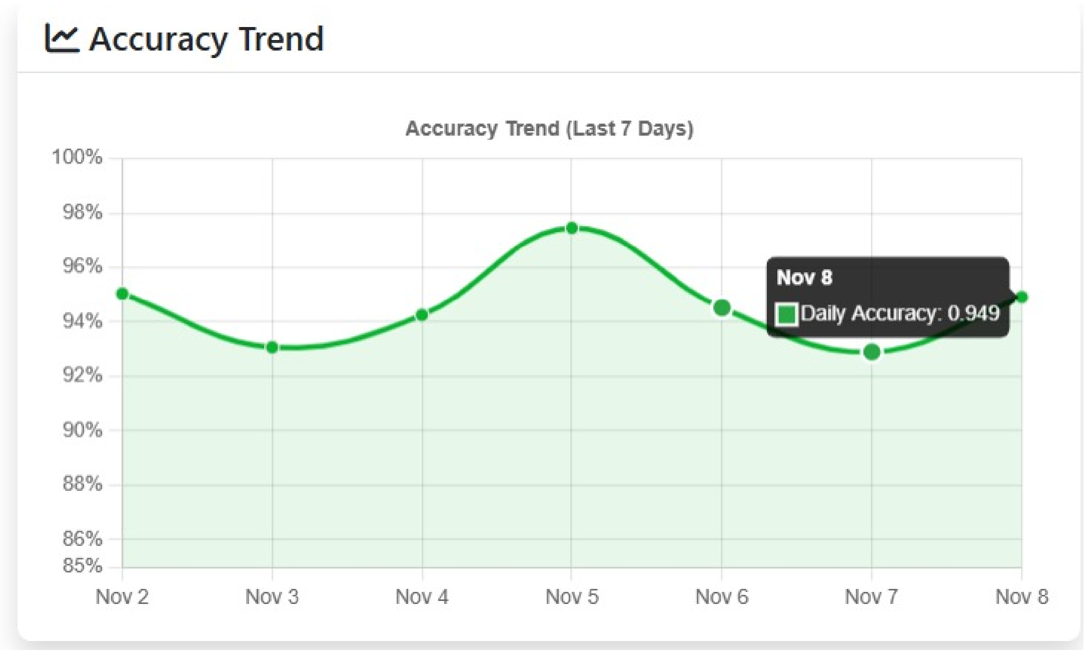
C Visualization



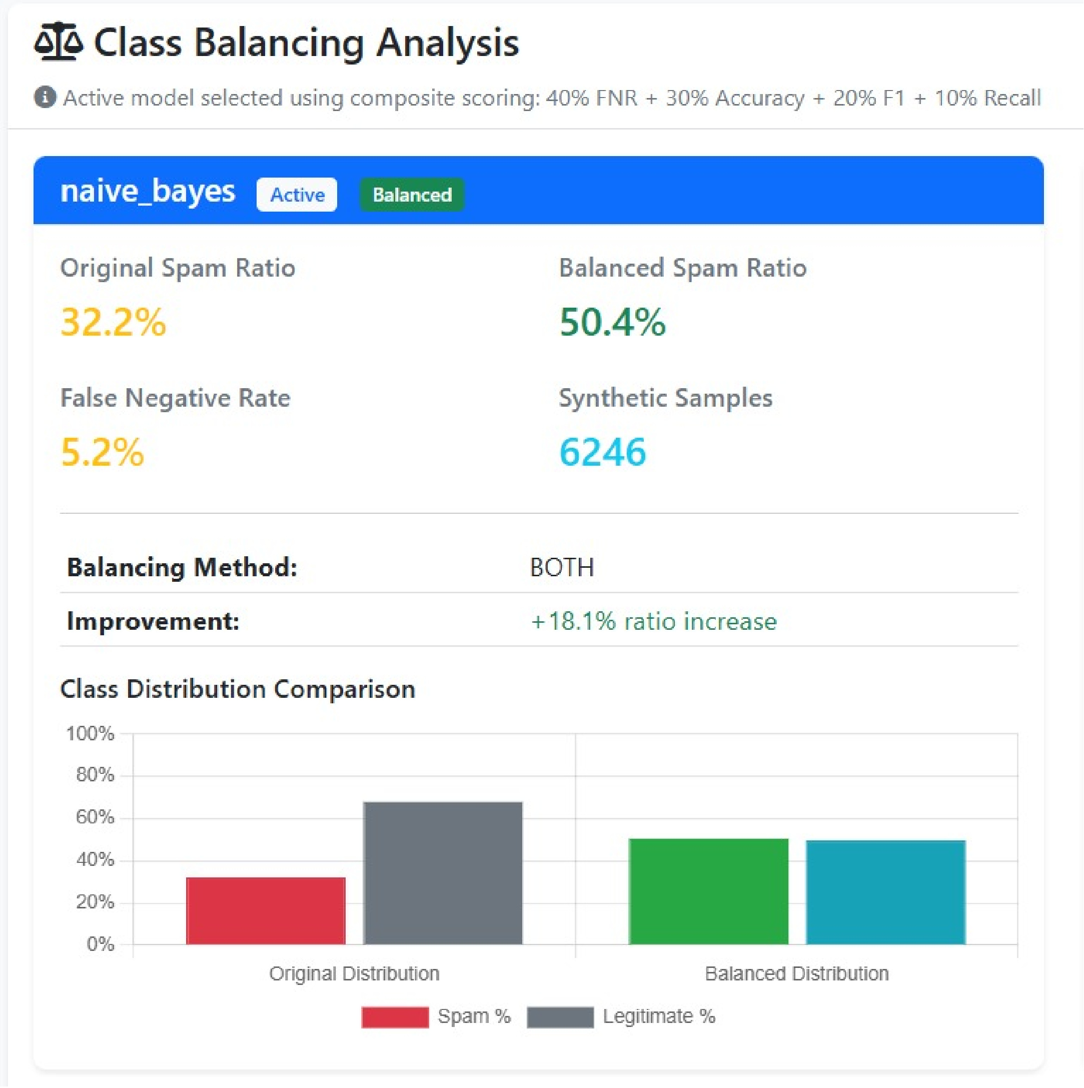
Model Performance Comparison



Classification Results



Accuracy Trend



Class Balancing Analysis

Active model selected using composite scoring: 40% FNR + 30% Accuracy + 20% F1 + 10% Recall

naive_bayes Active Balanced

Original Spam Ratio

32.2%

Balanced Spam Ratio

50.4%

False Negative Rate

5.2%

Synthetic Samples

6246

Balancing Method:	BOTH
Improvement:	+18.1% ratio increase

Class Distribution Comparison

End of Documentation