**Design and Implementation of Email Classification System**

**——Based on Naive Bayes algorithm**

**Abstract**

With the widespread adoption of email communication, the identification of spam has become a crucial issue in network security and user experience optimization. This paper, based on Natural Language Processing (NLP) technology, designs and implements an efficient email classification system. It employs the Naive Bayes algorithm as the core classification model, combined with text preprocessing, TF-IDF feature extraction, and grid search tuning methods, ultimately achieving an accuracy rate of 98% on the test set. Experimental results demonstrate that the system can effectively distinguish between ham and spam, and the robustness of the model is verified through confusion matrix and threshold analysis. This paper further explores the feature weights of high-frequency spam keywords and their contribution to classification, providing data support for subsequent research.

**1 Problem Statement**

As a mainstream communication tool, the security of email is directly related to user privacy and enterprise operational efficiency. According to statistics, about 50% of the world's e-mail is spam, which covers advertising, phishing attacks and malware dissemination. Traditional rule filtering methods (such as keyword blacklist) are gradually replaced by machine learning because they are difficult to adapt to the dynamic spam pattern. However, the sparsity, unstructured features, and class imbalance of email text pose challenges to model performance.

This study aims to construct an automated email classification system using NLP technology to address the following core issues:

**Text noise processing:** Email content contains special characters, abbreviations, multilingual mixing, and HTML tags, which need to be standardized and cleaned to extract effective features;

**Feature representation optimization:** The high-dimensional nature of short texts can easily lead to model overfitting, and it is necessary to reduce computational complexity through feature selection and vectorization;

**Classification threshold tuning:** The probability output by Naive Bayes needs to be dynamically threshold adjusted to balance accuracy and recall;

**Enhanced interpretability:** Identifying keywords that contribute significantly to classification, assisting manual review and model iteration.

**2 Data Description**

The experiment uses the data set [Email Classification. csv] provided by the professor. Ham accounted for 86.6%, 4827; Spam accounts for 13.4%, 747%. A total of 5574 items. Key fields include:

**Message :** Original email body (including but not limited to daily messages, HTML tags, numbers, and punctuation)

**Class :** Category tags (ham, spam)

The training set and test set are divided in a 7:3 ratio, and stratified sampling is used to ensure consistency in class distribution. The data statistics are shown in Table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Set | Total Samples | Ham | Ham(%) | Spam | Spam(%) |
| Training Set | 3901 | 3378 | 86.6% | 523 | 13.4% |
| Test Set | 1673 | 1449 | 224 |
| Total | 5574 | 4827 | 747 |

**3 Methodology and Implementation**

**Data Preprocessing**

Raw email text undergoes multi-step cleaning:

Remove Non-Alphabetic Characters : Regex [^a-zA-Z\s] eliminates digits, punctuation.

Lowercasing : Normalize case variations.

Stopword Removal : Combine NLTK English stopwords and custom lists (e.g., 'i', 'me', 'my', 'myself') to filter noise.

Stemming : Apply Porter Stemmer to reduce inflectional forms (e.g., "running" → "run").

Example:

Raw: “Win a FREE iPhone! Click here NOW!!!”

Processed: “win free iphone click now

**Feature Engineering**

**TF-IDF Vectorization** converts text into numerical features: Here, N is the total documents, and DF(t) is the document frequency of term t. We set max\_features and include bigrams to capture contextual patterns.

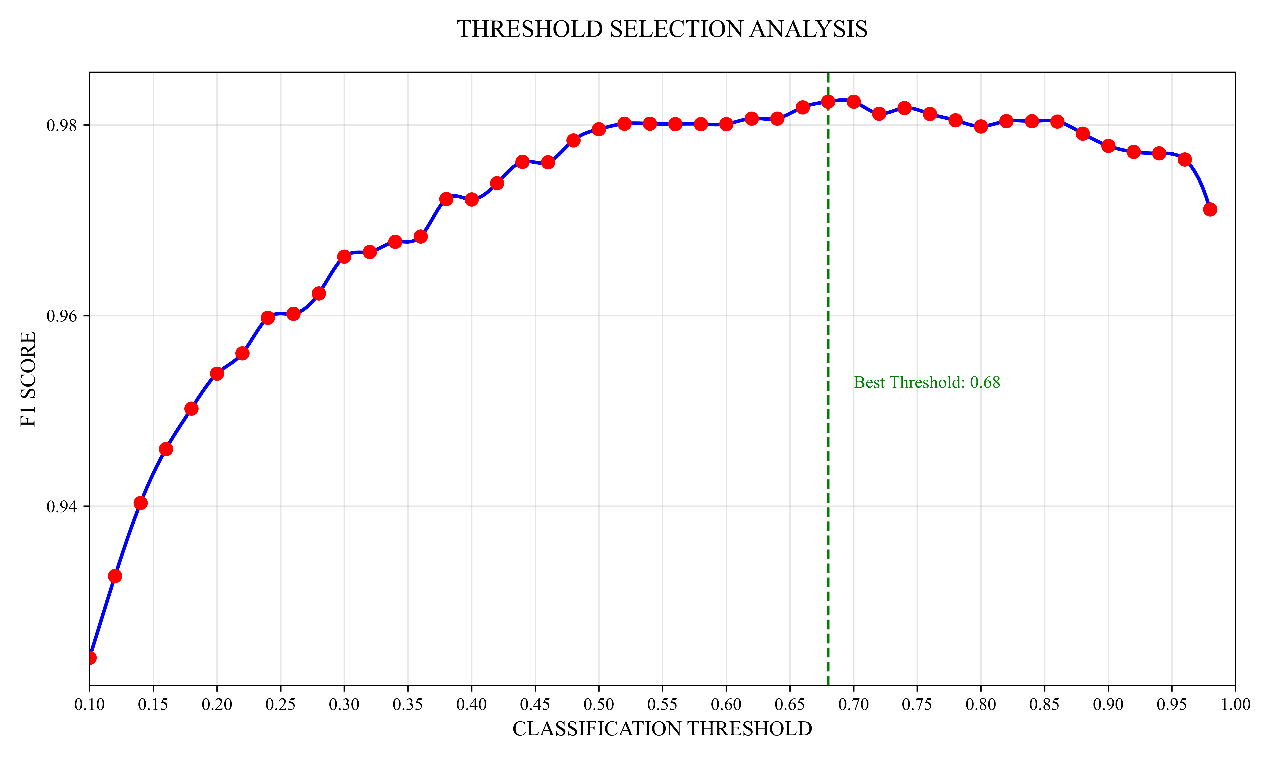
**Model Training and Optimization**

**Multinomial Naive Bayes (MNB)** serves as the classifier, calculating posterior probabilities via Bayes’ theorem. By using GridSearchCV to optimize the smoothing parameter α, the results show that the model performs the best when α=0.1 (as shown in Table 2)

The param\_grid defines the candidate values for the alpha parameter, ranging from 0.0001 to 10, with a total of 6 values. GridSearchCV uses cross-validation (cv=5) to evaluate the performance of each alpha value and selects the one that performs best. The best model is obtained through best\_estimator\_ , and then the best alpha value is printed out. After the training is completed, the alpha value in grid\_search.best\_params\_ will be output, which is the optimal hyperparameter obtained through grid search. Simply put, grid search selects the best alpha among the given candidate values using 5-fold cross-validation, and the final model is trained using this optimal alpha value. Cross-validation shows optimal performance at α=0.1.

**Classification Threshold Optimization**

As shown in the figure, a default threshold of 0.5 may result in suboptimal outcomes. By searching within a range of 0.10 to 1 with a step size of 0.02, the weighted F1 score reaches its peak at a threshold of 0.68, with a value of 0.982.

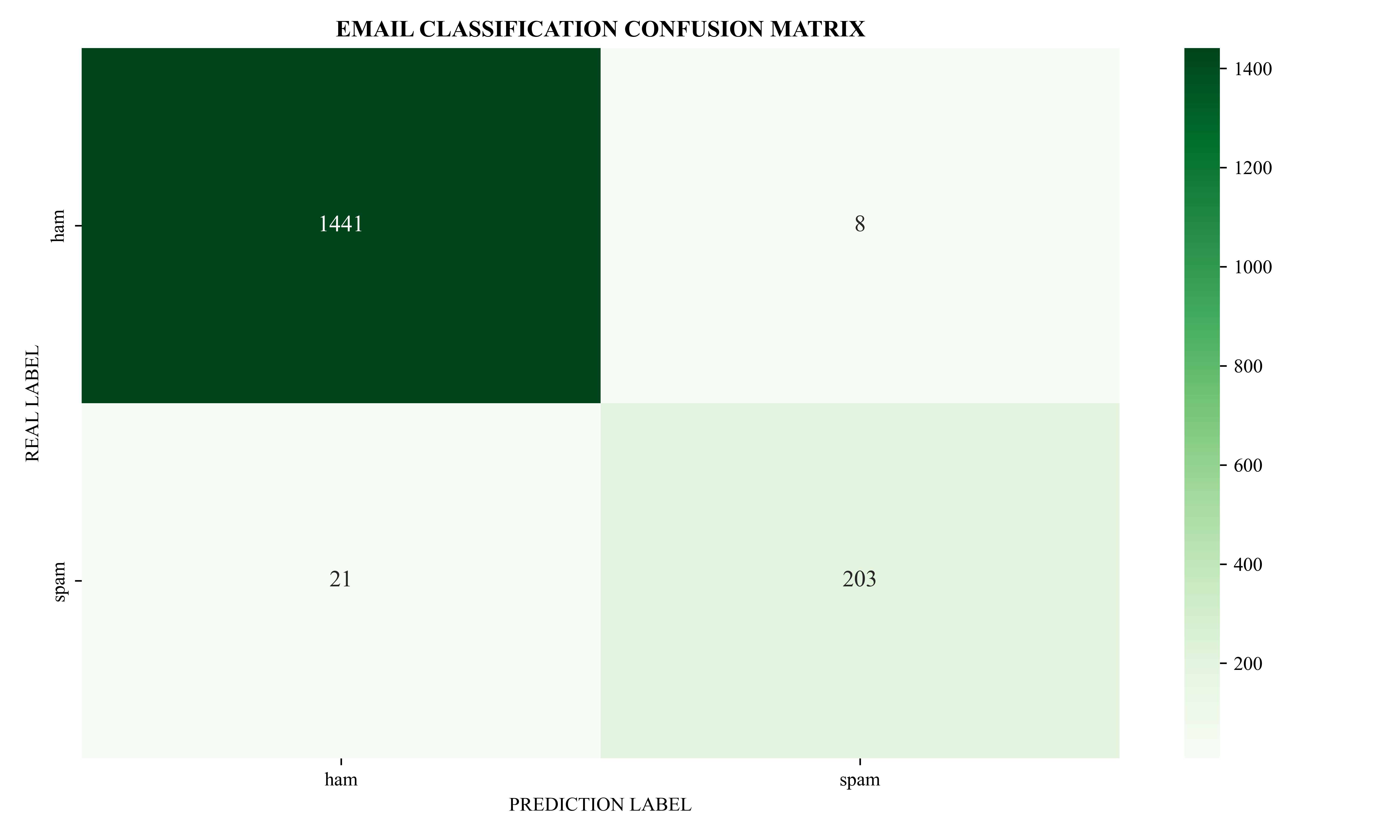


**4 Results and Discussion**

**Performance Evaluation**

Test set results. Test set accuracy:0.9827

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | precision | recall | f1-score | support |
| Ham | 0.99 | 0.99 | 0.99 | 1449 |
| Spam | 0.96 | 0.91 | 0.93 | 224 |
| accuracy |  |  | 0.98 | 1673 |
| macro avg | 0.97 | 0.95 | 0.96 | 1673 |
| weighted avg | 0.98 | 0.98 | 0.98 | 1673 |



The confusion matrix shows only 21 spam emails misclassified as ham, highlighting high sensitivity to risky samples.

**High-Frequency Feature Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Vocabulary | Weight | Count | Frequency |
| call | -5.522187274 | 735 | 0.188413227 |
| free | -5.835822142 | 457 | 0.117149449 |
| txt | -6.08192862 | 339 | 0.086900795 |
| text | -6.216611082 | 301 | 0.077159703 |
| claim | -6.268083579 | 230 | 0.058959241 |
| mobil | -6.340595121 | 239 | 0.061266342 |
| stop | -6.349740625 | 243 | 0.06229172 |
| ur | -6.369310124 | 299 | 0.076647014 |
| repli | -6.386827938 | 241 | 0.061779031 |
| prize | -6.478736463 | 169 | 0.043322225 |
| won | -6.611751728 | 141 | 0.036144578 |
| award | -6.618365714 | 147 | 0.037682645 |
| servic | -6.623939304 | 143 | 0.036657267 |
| new | -6.733328628 | 145 | 0.037169956 |

Extract the top 15 spam keywords with the highest TF-IDF weight (20 were selected from the actual code), among which words such as "call", "free", and "txt" significantly affect the classification results. Combined with word frequency statistics, the effectiveness of feature selection was verified. The spelling errors in the table are caused by data cleaning.

**5 Conclusion and Future Work**

The proposed system achieves high accuracy and interpretability. Future directions include:

Deep Learning Integration : Adopt BERT for contextual semantic analysis.

Dynamic Updates : Implement online learning to adapt to evolving spam patterns.

Introduce Ternary Grammar : Triple grammar features can capture semantic relationships over longer distances

Lemma : Stemming destroys semantics, Lemma preserves the grammatical form of vocabulary

Chi square feature selection : The current model retains all TF-IDF features, and using open feature selection can remove redundant features and improve model generalization

And, the code is not elegant enough.

**6 Codes**

*""""  
Email classification main program (v2.2)*

*Although I haven't completed the improvement yet, I will upload it to my GitHub:* *https://github.com/Prodigitree  
Function: A spam classification system based on Naive Bayes algorithm  
"""  
# --------------------------Environment Configuration--------------------------  
# Basic Library***import** os  
**import** sys  
**import** warnings  
**import** numpy **as** np  
*# DataProcessing***import** pandas **as** pd  
*# VIS***import** matplotlib.pyplot **as** plt  
**import** seaborn **as** sns  
**from** scipy.interpolate **import** make\_interp\_spline  
plt.rcParams.update({  
 **'savefig.dpi'**: 600,  
 **'font.sans-serif'**: **'Times New Roman'**,  
 **'axes.unicode\_minus'**: **False**,  
 **'figure.figsize'**: (10, 6),  
})  
*# TextProcessing***import** re  
**import** nltk  
**from** nltk.stem **import** PorterStemmer  
*# MachineLearning***from** sklearn.preprocessing **import** LabelEncoder  
**from** sklearn.model\_selection **import** train\_test\_split  
**from** sklearn.feature\_extraction.text **import** TfidfVectorizer  
**from** sklearn.naive\_bayes **import** MultinomialNB  
**from** sklearn.metrics **import** classification\_report, confusion\_matrix  
**from** sklearn.feature\_extraction.text **import** CountVectorizer  
**from** sklearn.model\_selection **import** GridSearchCV  
  
*# ConfigureWarningFiltering(Failure discovered in practical applications)*warnings.filterwarnings(**"ignore"**, category=DeprecationWarning)  
warnings.filterwarnings(**"ignore"**, category=FutureWarning)  
warnings.filterwarnings(**"ignore"**, category=UserWarning)  
warnings.filterwarnings(**"ignore"**, module=**"sklearn"**)  
  
os.environ[**"LOKY\_PICKLER"**] = **"pickle"**warnings.filterwarnings(**"ignore"**, category=DeprecationWarning)  
*# -------------------------- ConstantDefinition --------------------------*BASIC\_STOPWORDS = {  
 **'i'**, **'me'**, **'my'**, **'myself'**, **'we'**, **'our'**, **'ours'**, **'ourselves'**, **'you'**, **'your'**,  
 **'yours'**, **'yourself'**, **'he'**, **'him'**, **'his'**, **'himself'**, **'she'**, **'her'**, **'hers'**,  
 **'herself'**, **'it'**, **'its'**, **'itself'**, **'they'**, **'them'**, **'their'**, **'theirs'**, **'themselves'**,  
 **'what'**, **'which'**, **'who'**, **'whom'**, **'this'**, **'that'**, **'these'**, **'those'**, **'am'**, **'is'**, **'are'**,  
 **'was'**, **'were'**, **'be'**, **'been'**, **'being'**, **'have'**, **'has'**, **'had'**, **'having'**, **'do'**, **'does'**,  
 **'did'**, **'doing'**, **'a'**, **'an'**, **'the'**, **'and'**, **'but'**, **'if'**, **'or'**, **'because'**, **'as'**, **'until'**,  
 **'while'**, **'of'**, **'at'**, **'by'**, **'for'**, **'with'**, **'about'**, **'against'**, **'between'**, **'into'**,  
 **'through'**, **'during'**, **'before'**, **'after'**, **'above'**, **'below'**, **'to'**, **'from'**, **'up'**, **'down'**,  
 **'in'**, **'out'**, **'on'**, **'off'**, **'over'**, **'under'**, **'again'**, **'further'**, **'then'**, **'once'**, **'here'**,  
 **'there'**, **'when'**, **'where'**, **'why'**, **'how'**, **'all'**, **'any'**, **'both'**, **'each'**, **'few'**, **'more'**,  
 **'most'**, **'other'**, **'some'**, **'such'**, **'no'**, **'nor'**, **'not'**, **'only'**, **'own'**, **'same'**, **'so'**,  
 **'than'**, **'too'**, **'very'**, **'can'**, **'will'**, **'just'**, **'don'**, **'should'**, **'now'**}  
  
*# -------------------------- CoreFunction --------------------------***def** initialize\_nltk():  
 *"""  
1. Multipath detection: User directory → Project directory  
2. Automatic download: Automatically download when local resources are missing  
3. Alternative solution: Enable basic stop words when download fails  
!!!Warning, due to the inability to obtain stop words properly, offline BASIC\_STOPWORDS will be used directly!!!  
 """* **global** STOPWORDS  
 STOPWORDS = BASIC\_STOPWORDS  
 print(**f"\n Warning!Use\_BASIC\_STOPWORDS({**len(BASIC\_STOPWORDS)**})"**)  
  
  
**def** clean\_text(text):  
 *"""  
1. Cleaning: Remove non letter characters → Convert to lowercase → Remove spaces at both ends  
2. Word segmentation: Divide text by spaces  
3. Filtering: Remove stop words  
4. Stemming: Porter Stemming Processing  
 Args:  
 text (str): OriginalText  
 Returns:  
 str: StandardizedText  
 """  
 # cleaning* text = re.sub(**r'[^a-zA-Z\s]'**, **''**, text).lower().strip()  
  
 *# Segmentation And StopWordFiltering* tokens = text.split()  
 stems = [PorterStemmer().stem(w) **for** w **in** tokens]  
 bigrams = [**' '**.join(pair) **for** pair **in** zip(stems[:-1], stems[1:])]  
 **return ' '**.join(stems + bigrams)  
  
  
  
  
**def** analyze\_features(vectorizer, model, n\_top=20):  
 *"""  
1. Obtain feature names (compatible with both old and new versions of sklearn)  
2. Sort by logarithmic probability of spam category  
3. Output TopN important features and their weights  
 """* **global** STOPWORDS  
 features\_names = vectorizer.get\_feature\_names()  
 valid\_mask = [word **not in** STOPWORDS **for** word **in** features\_names]  
 filtered\_features = [(name, prob) **for** name, prob **in** zip(features\_names, model.feature\_log\_prob\_[1])  
 **if** valid\_mask[features\_names.index(name)]]  
 features = sorted(  
 filtered\_features,  
 key=**lambda** x: (-x[1], -len(x[0]), x[0])  
 )[:n\_top]  
 *# DataAlignment* vocab = [feat[0] **for** feat **in** features]  
 weights = [feat[1] **for** feat **in** features]  
  
 **return** vocab, weights  
  
  
**def** evaluate\_model(model, x\_test, y\_test, thresholds=np.arange(0.10, 1, 0.02)):  
 *"""  
Multi threshold model evaluation (version2)  
1. Detailed indicator output (including accuracy/recall/F1 of each classification)  
2. Optimal threshold recommendation based on weighted F1  
3. Return the optimal threshold for future use  
 """* y\_proba = model.predict\_proba(x\_test)[:, 1]  
  
 best\_score = 0  
  
 reports = []  
  
 *# Evaluation Threshold* **for** thresh **in** thresholds:  
 y\_pred = (y\_proba > thresh).astype(int)  
 report = classification\_report(y\_test, y\_pred, output\_dict=**True**)  
  
 *# Record Threshold Report* reports.append((thresh, report))  
 weighted\_avg = report[**'weighted avg'**]  
 current\_f1 = weighted\_avg[**'f1-score'**]  
  
 *# Update The bestscore* **if** current\_f1 > best\_score:  
 best\_score = current\_f1  
 best\_thresh = thresh  
  
 *# detailedIndicatorOutput* print(**"\n"** + **"-"** \* 50)  
 print(**f" threshold {**thresh**:.2f} detailedIndicators "**.center(45))  
 print(**"-"** \* 50)  
 print(**f"spam distinguish:"**)  
 print(  
 **f" Accuracy: {**report[**'1'**][**'precision'**]**:.3f} | Recall: {**report[**'1'**][**'recall'**]**:.3f} | F1: {**report[**'1'**][**'f1-score'**]**:.3f}"**)  
 print(**f"hamdistinguish:"**)  
 print(  
 **f" Accuracy: {**report[**'0'**][**'precision'**]**:.3f} | Recall: {**report[**'0'**][**'recall'**]**:.3f} | F1: {**report[**'0'**][**'f1-score'**]**:.3f}"**)  
 print(**f"F1: {**current\_f1**:.3f}"**)  
  
 *# best\_threshold\_recommendation* print(**"\n"** + **"="** \* 50)  
 print(**f" best\_threshold\_recommendation:{**best\_thresh**:.2f} "**.center(45))  
 print(**f" F1:{**best\_score**:.3f} "**.center(45))  
 print(**"="** \* 50)  
  
 *# AddDataCollectionBeforeReturning* f1\_scores = [report[**'weighted avg'**][**'f1-score'**] **for** \_, report **in** reports]  
 **return** best\_thresh, thresholds, f1\_scores  
  
  
**def** plot\_confusion\_matrix(y\_true, y\_pred, class\_names):  
 *"""  
Generate confusion matrix visualization  
1. Blue color scheme  
2. Display of numerical labels  
3. Chinese character support  
4. Automatically save to PNG format  
 """* cm = confusion\_matrix(y\_true, y\_pred)  
  
 plt.figure()  
 sns.heatmap(  
 cm,  
 annot=**True**,  
 fmt=**'d'**,  
 cmap=**'Greens'**,  
 xticklabels=class\_names,  
 yticklabels=class\_names,  
 annot\_kws={**'size'**: 12}  
 )  
 plt.title(**'EMAIL CLASSIFICATION CONFUSION MATRIX'**, fontweight=**'bold'**)  
 plt.xlabel(**'PREDICTION LABEL'**)  
 plt.ylabel(**'REAL LABEL'**)  
 plt.tight\_layout()  
 plt.savefig(**'confusion\_matrix.png'**)  
 plt.show()  
  
  
**def** plot\_threshold\_analysis(thresholds, f1\_scores, best\_thresh):  
 plt.figure()  
 x\_new = np.linspace(min(thresholds), max(thresholds), 300)  
 spl = make\_interp\_spline(thresholds, f1\_scores, k=3)  
 y\_smooth = spl(x\_new)  
  
 plt.plot(x\_new, y\_smooth, **'b-'**, linewidth=2)  
 plt.scatter(thresholds, f1\_scores, c=**'red'**, s=50, zorder=5)  
 plt.axvline(best\_thresh, color=**'green'**, linestyle=**'--'**, linewidth=1.5)  
 focus\_range = 0.1 *#control\_focus\_range* plt.xlim(0.10, 1.0)  
 plt.gca().xaxis.set\_major\_locator(plt.MultipleLocator(0.05)) *# Every 0.05 marks on the X-axis* plt.gca().yaxis.set\_major\_locator(plt.MultipleLocator(0.02)) *# Every 0.02 marks on the Y-axis* plt.title(**'THRESHOLD SELECTION ANALYSIS'**, fontsize=14, pad=20)  
 plt.xlabel(**'CLASSIFICATION THRESHOLD'**, fontsize=12)  
 plt.ylabel(**'F1 SCORE'**, fontsize=12)  
 plt.grid(**True**, alpha=0.3)  
  
 plt.text(best\_thresh + 0.02, max(f1\_scores) - 0.03,  
 **f'Best Threshold: {**best\_thresh**:.2f}'**,  
 fontsize=10, color=**'green'**)  
  
 plt.tight\_layout()  
 plt.savefig(**'threshold\_analysis.png'**)  
 plt.show()  
  
  
*# -------------------------- MainProgram --------------------------***if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 **try**:  
 print(**"\n"** + **"="** \* 50)  
 print(**" EmailClassificationSystem v2.1 "**.center(45))  
 print(**"="** \* 50)  
  
 *# Initialization* initialize\_nltk()  
 print(**"\n"** + **"="** \* 50)  
 print(**" DataLoading And Preprocessing "**.center(45))  
 print(**"="** \* 50)  
 df = pd.read\_csv(**"Email Classification.csv"**, encoding=**"utf-8-sig"**)  
 df[**"Class"**] = LabelEncoder().fit\_transform(df[**"Class"**])  
 print(**"DistributionOfDatasetCategories:\n"**, df[**"Class"**].value\_counts())  
  
 *# TextPreprocessing* print(**"\n TextPreprocessing..."**)  
 df[**"Cleaned\_Message"**] = df[**"Message"**].apply(clean\_text)  
  
 *# Feature Engineering* print(**"\n In feature engineering processing..."**)  
 tfidf = TfidfVectorizer(max\_features=30000, ngram\_range=(1, 2))  
 min\_df = 5,  
 max\_df = 0.85  
 X = tfidf.fit\_transform(df[**"Cleaned\_Message"**])  
 y = df[**"Class"**]  
  
 *# Add validation after feature engineering* print(**"\nFeature space analysis:"**)  
 print(**f"Actual feature dimension: {**X.shape[1]**}"**)  
  
 *# Add binary grammar validity validation* bigram\_samples = [phrase **for** phrase **in** tfidf.get\_feature\_names() **if ' ' in** phrase][:10]  
 print(**"Example binary grammar features:"**, bigram\_samples)  
  
  
 tfidf\_feature\_names = tfidf.get\_feature\_names()  
  
 *# DataPartitioning* X\_train, X\_test, y\_train, y\_test = train\_test\_split(  
 X, y,  
 test\_size=0.3,  
 random\_state=42,  
 *#42 is the ultimate answer to the universe* stratify=y  
 )  
  
  
 *# DataDistributionStatistics* **def** print\_distribution(name, y\_data):  
 ham = (y\_data == 0).sum()  
 spam = (y\_data == 1).sum()  
 print(**f"{**name**}:"**)  
 print(**f"ham: {**ham**} ({**ham / len(y\_data)**:.1%})"**)  
 print(**f"spam: {**spam**} ({**spam / len(y\_data)**:.1%})"**)  
  
 print\_distribution(**"\nTrainingSetDistribution"**, y\_train)  
 print\_distribution(**"\nTestSetDistribution"**, y\_test)  
  
 print(**"\n DuringModelTraining..."**)  
 *# GridSearch Configuration best\_alpha* param\_grid = {**'alpha'**: [0.0001, 0.001, 0.01, 0.1, 1, 10]}  
 grid\_search = GridSearchCV(  
 estimator=MultinomialNB(),  
 param\_grid=param\_grid,  
 cv=5,  
 n\_jobs=1  
 )  
  
  
  
 grid\_search.fit(X\_train, y\_train)  
 model = grid\_search.best\_estimator\_  
  
 *# Output best\_alpha* print(**f"ModelTrainingCompleted，best\_alpha: {**grid\_search.best\_params\_[**'alpha'**]**}"**)  
  
 *# spam WordFrequencyStatistics* print(**"\n CountingHighFrequencySpamKeywords..."**)  
 spam\_indices = y\_train[y\_train == 1].index  
 spam\_texts = df.loc[spam\_indices, **'Cleaned\_Message'**]  
 spam\_texts = spam\_texts.apply(  
 **lambda** x: **' '**.join([w **for** w **in** x.split() **if** w **not in** STOPWORDS])  
 )  
  
  
 count\_vec = CountVectorizer(  
 vocabulary=tfidf\_feature\_names,  
 ngram\_range=(1, 2))  
 spam\_counts = count\_vec.transform(spam\_texts)  
 spam\_counts\_array = spam\_counts.sum(axis=0).A1  
  
 *# Extract spam text from the training set* spam\_indices = y\_train[y\_train == 1].index  
 spam\_texts = df.loc[spam\_indices, **'Cleaned\_Message'**]  
 spam\_texts = spam\_texts.apply(  
 **lambda** x: **' '**.join([w **for** w **in** x.split() **if** w **not in** STOPWORDS])  
 )  
 feature\_names = (count\_vec.get\_feature\_names()) *# ← Abandon compatibility processing and directly use the old version* word\_counts = zip(feature\_names, spam\_counts.sum(axis=0).tolist()[0])  
 sorted\_words = sorted(word\_counts, key=**lambda** x: x[1], reverse=**True**)[:30]  
  
 train\_total = X\_train.shape[0]  
 freq\_data = [  
 (word, count, count / train\_total)  
 **for** word, count **in** sorted\_words  
 ]  
  
 *# Model evaluation* evaluate\_model(model, X\_test, y\_test)  
 *# best\_threshold = evaluate\_model(model, X\_test, y\_test)* best\_threshold, thresholds, f1\_scores = evaluate\_model(model, X\_test, y\_test)  
 feature\_vocab, feature\_weights = analyze\_features(tfidf, model, n\_top=20) *# ← Modify to output 20 features  
 # top\_features = analyze\_features(tfidf, model)  
  
 # Use the optimal threshold for final prediction* y\_proba = model.predict\_proba(X\_test)[:, 1]  
 final\_pred = (y\_proba > best\_threshold).astype(int)  
  
 *# Generate the final evaluation report* print(**"\n"** + **"="** \* 50)  
 print(**f" Final evaluation report(Use threshold {**best\_threshold**:.2f}) "**.center(45))  
 print(**"="** \* 50)  
 print(classification\_report(y\_test, final\_pred))  
  
 *# Add test accuracy output* print(**f"\nTest set accuracy:{**np.mean(final\_pred == y\_test)**:.4f}"**)  
  
 *# Generate confusion matrix using optimal threshold* plot\_confusion\_matrix(y\_test, final\_pred, [**'ham'**, **'spam'**])  
  
 count\_vec = CountVectorizer(vocabulary=feature\_vocab)  
 spam\_counts = count\_vec.fit\_transform(spam\_texts)  
 spam\_counts\_array = np.asarray(spam\_counts.sum(axis=0)).flatten()  
  
 *# Debugging dimension verification  
 #print(f"Feature dimension: {len(feature\_vocab)},  
 #print(f"Word frequency dimension: {spam\_counts.sum(axis=0).shape[1]}")  
 #print(f"Length of vocabulary column: {len(feature\_vocab)}")  
 #print(f"Weight column length: {len(feature\_weights)}")  
 #print(f"Word frequency statistics dimension: {spam\_counts.sum(axis=0).shape}")  
  
 # CreateDataBox* df\_merge = pd.DataFrame({  
 **'Vocabulary'**: feature\_vocab,  
 **'Weight'**: feature\_weights,  
 **'Count'**: spam\_counts\_array[:len(feature\_vocab)],  
 **'Frequency'**: spam\_counts\_array[:len(feature\_vocab)] / X\_train.shape[0]  
 }).head(20)  
 *# ConsoleOutput* print(**"\n top20\_high\_frequency\_spam\_keywords:"**)  
 print(**f"{'Vocabulary':<15} | {'Weight':<8} | {'Count':<8} | {'Frequency':<8}"**)  
 print(**"-"** \* 45)  
 **for** idx, row **in** df\_merge.iterrows():  
 print(**f"{**row[**'Vocabulary'**]**:15} | {**row[**'Weight'**]**:8.3f} | {**row[**'Count'**]**:8} | {**row[**'Frequency'**]**:8.3%}"**)  
  
 *# output2CSV* df\_merge.to\_csv(**'top20\_high\_frequency\_spam\_keywords.csv'**, index=**False**, encoding=**'utf-8-sig'**)  
 print(**" data output to top20\_high\_frequency\_spam\_keywords.csv"**)  
  
 *# outpot png* print(**"\n Generate\_png"**)  
 *# (y\_test, model.predict(X\_test), ['ham', 'spam'])* plot\_threshold\_analysis(thresholds, f1\_scores, best\_threshold)  
 print(**"\n ProgramRunningEnds"**)  
 **except** Exception **as** e:  
 print(**f"\n PROGRAM TERMINATED ABNORMALLY: {**str(e)**}"**)  
 sys.exit(1)