main.py spam.csv

Spam 'Detection' {

[using Naive Bayes]

< with a dataset from singapore! >

}

Created by Evan Tan

Scope of 'Contents' {

01 Introduction to Naive Bayes

< concept, characteristics, flaws >

02 Choice of data set

< considerations, preprocessing >

03 Code for Model

< Choice of Vectorizer/confusion matrix >

}

main.py spam.csv

01 {

[Naive Bayes]

<introduction>

}

What is Naive Bayes {

< It is a family of probabilistic classifiers based on applying Bayes’ theorem with a strong (naive) assumption of independence between features. >

}

What makes it ‘naive’? {

< It assumes that all features are independent given the class >

}

main.py spam.csv

Key concepts; {

'Bayes' Theorem:'

<p It calculates the probability of a class C given some evidence (features) X:

>

$$P(C|X) = \frac{P(X|C) \times P(C)}{P(X)}$$

</p>

}

main.py spam.csv

Key concepts; {

Naive Assumptions:’

<p It assumes that all features are independent given the class:

>

$$P(X|C) = \prod_i P(x_i|C)$$

</p>

}

main.py spam.csv

How does it work? {

< The algorithm learns the prior probability of each class P(C) and the likelihood of features given the class P(Xi|C) from the training data. >

}

Predictions {

< For a new input, it computes the probability for each class using Bayes' theorem and predicts the class with the highest probability. >

}

Why is Naive Bayes popular for 'text classification?' {

01 Works well with high-dimensional data like text, where features are word counts or frequencies.

02 Fast and scalable.

03 Often surprisingly accurate despite the strong independence assumption.

04 Commonly used for spam detection, sentiment analysis, document classification, and more.

}

main.py spam.csv

Types of Naive Bayes classifiers;

{

Multinomial Naive Bayes

<Best for discrete features like word counts (common in text).>

Bernoulli Naive Bayes

<For binary/boolean features (word presence or absence).>

Gaussian Naive Bayes

<For continuous data assuming normal distribution.>

}

main.py   spam.csv

Examples About ‘Naive Bayes for Spam Detection (Toy Data)’{


"Free money now"

< Spam >


"Call me now"

< Ham[1] >


"Win money win prize"

< spam >


"Hello how are you"

< Ham >

}

Created by Evan Tan

1:Ham is used to describe normal messages

main.py

spam.csv

02 {

[Choice of data set]

<considerations/
preprocessing>

}

Chosen data set {

<SMS Spam Collection Dataset >

< /1 >

* The SMS Spam Collection is a set of SMS tagged messages that have been collected for SMS Spam research.

< /2 >

* The files contain one message per line. Each line is composed by two columns: v1 contains the label (ham or spam) and v2 contains the raw text.

}

Content {

< This corpus has been collected from free or free for research sources at the Internet >

425 SMS spam messages

< Manually extracted from the Grumbletext Web site. >

3,375 SMS randomly chosen ham messages

<from the NUS SMS Corpus (NSC), which is a dataset of about 10,000 legitimate messages collected for research at the Department of Computer Science at the National University of Singapore >

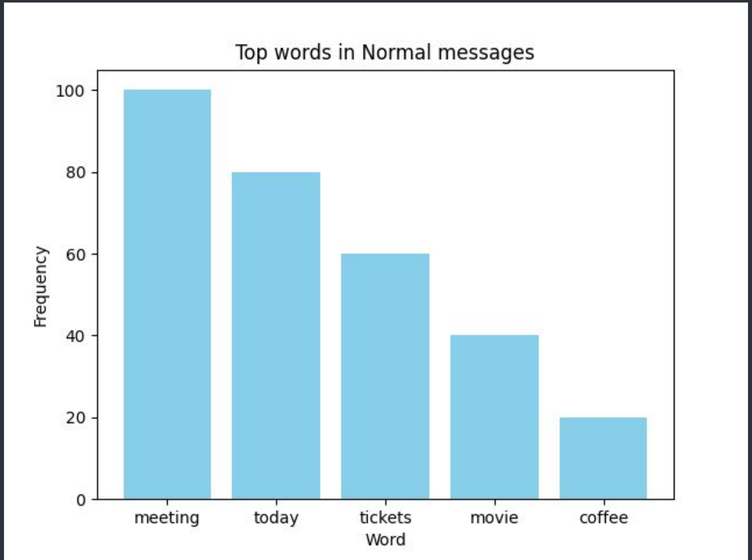
}

Bar chart of 'Normal Messages'

{

< Figure: Top words in ham messages by frequency. The training text has been cleaned and aggregated to count words (excluding generic terms).>

}



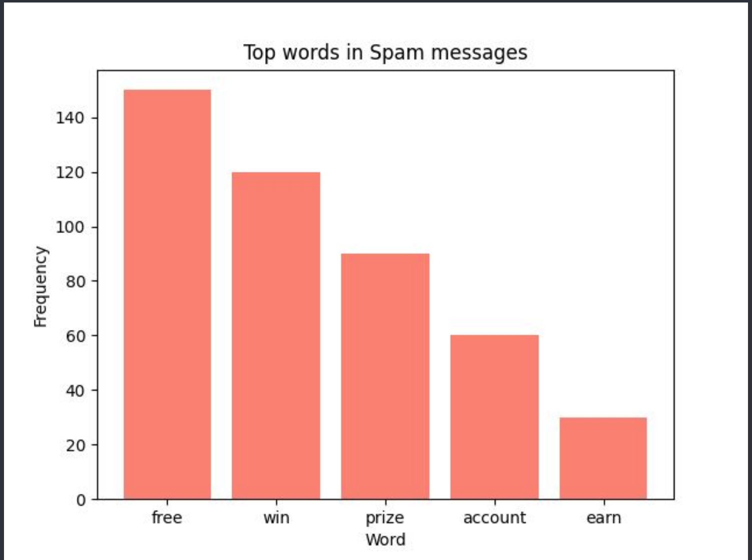
main.py

spam.csv

Bar chart of 'Spam Messages' {

< Top words in spam messages. Here we see "free", "win", "prize", etc., which are typical in spam offers >

}



Created by Evan Tan

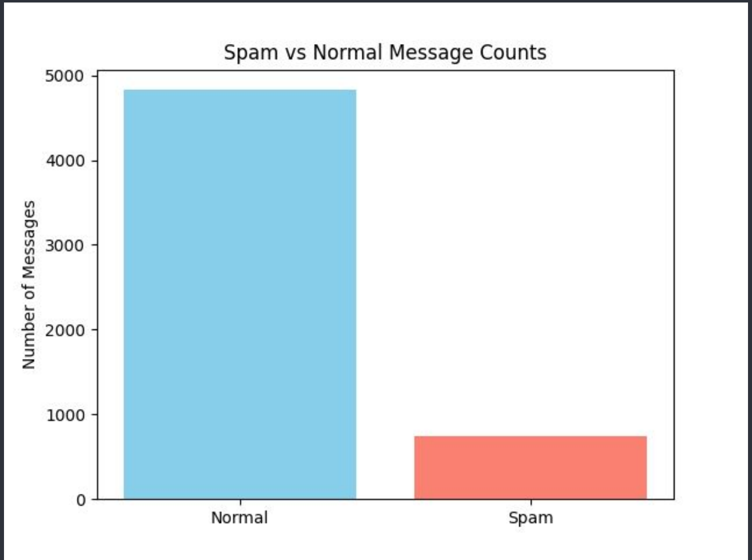
main.py

spam.csv

Distribution of 'ham vs spam' messages {

<The dataset is heavily imbalanced (many more ham than spam) as shown by the bar chart.>

}



4,827 ham and 747 spam messages {

< as documented in the Kaggle SMS Spam Collection >

}

03 {

[Code for Model]

< Choice of Vectorizer/confusion matrix >

}

main.py spam.csv

Libraries {

Numpy/Pandas

Reading of dataset

sklearn

Usage of algorithm

seaborn/matplotlib

Data visualisation

} Each import line brings in functions/classes needed later.

main.py spam.csv

Model (1); {

```
#vectorize text into numerical values. Normal text => 0, spam => 1
#Binary classification problem. Preprocessing step in supervised learning
spam_df['spam'] = spam_df['Category'].map({'ham': 0, 'spam': 1})


#START OF MODELLING
#train/test split, splits the dataset into training and testing subsets
X_train, X_test, y_train, y_test = train_test_split(spam_df.Message,spam_df.spam, random_state = 23, test_size=0.3)
'''
```

Mapping ‘ham’→0 and ‘spam’→1 prepares the data for modeling, making it a binary classification problem. This label encoding is a common preprocessing step in supervised learning. Train-test split: The script splits the dataset into training and testing subsets:

}

main.py spam.csv

Model (2); {

```python
#vectorize text into numerical values. Normal text => 0, spam => 1
#Binary classification problem. Preprocessing step in supervised learning
spam_df['spam'] = spam_df['Category'].map({'ham': 0, 'spam': 1})


#START OF MODELLING
#train/test split, splits the dataset into training and testing subsets
X_train, X_test, y_train, y_test = train_test_split(spam_df.Message,spam_df.spam, random_state = 23, test_size=0.3)
'''
```

The train_test_split function randomly partitions the data into 70% training and 30% test sets. This separation allows the model’s performance to be evaluated on unseen data. The random_state ensures reproducibility of the split.

}

Created by Evan Tan

main.py spam.csv

Model (3); {

```python
#Store word count as a matrix, text must be converted to numeric features
#CountVectorizer tokenizes the text and builds a vocabulary of all words
#Then transforms each message into a vector of word counts
cv = CountVectorizer()
X_train_count = cv.fit_transform(X_train.values)

#Training of model
#Multinomial Naive Bayes classifier is then trained on the vectorized features
model = MultinomialNB()
model.fit(X_train_count, y_train)
```

CountVectorizer tokenizes the text and builds a vocabulary of all words then transforms each message into a vector of word counts
The fit_transform on training data learns the vocabulary and produces a sparse count matrix.

}

Model (4); {

```
#Store word count as a matrix, text must be converted to numeric features
#CountVectorizer tokenizes the text and builds a vocabulary of all words
#Then transforms each message into a vector of word counts
cv = CountVectorizer()
X_train_count = cv.fit_transform(X_train.values)

#Training of model
#Multinomial Naive Bayes classifier is then trained on the vectorized features
model = MultinomialNB()
model.fit(X_train_count, y_train)
```

Training the model: A Multinomial Naive Bayes classifier is then trained on the vectorized features

MultinomialNB is well-suited for discrete word count features. It learns the probability of each word given the class (ham or spam), which allows it to classify new messages. After this line, nb is a trained model ready to predict labels.

}

Prediction function{

```python
#turning test into a function for reusability
trained_model = model
trained_vectorizer = cv
def predict_spam(messages, model=trained_model, vectorizer=trained_vectorizer):
    message_count = vectorizer.transform(messages)
    prediction = model.predict(message_count)
    return prediction
```

This function takes a string msg, vectorizes it using the same CountVectorizer, and applies nb.predict to output 0/1. It returns “spam” or “ham” for convenience.
For example, calling predict_spam("Congratulations, you have won!") would print “spam” if the model predicts 1.

}

main.py spam.csv

Alternative 'vectorizer' {

TF-IDF weighting

* TF-IDF downweights very common words across all messages and upweights distinctive words. This often yields better performance by capturing term significance rather than just frequency

```
tfidf = TfidfVectorizer()
X_train_tfidf = tfidf.fit_transform(X_train.values)
```

}

Created by Evan Tan

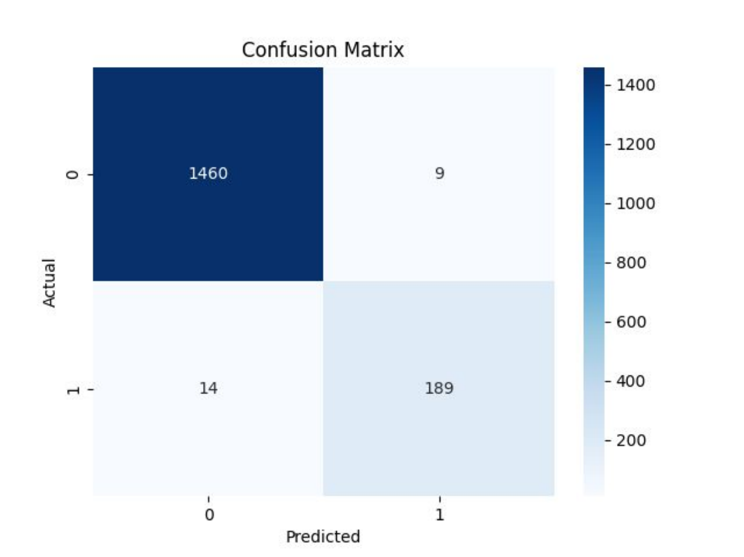
main.py

spam.csv

Comparison 'vectorizers' (CountVectorizer){

Each row is the actual class, each column the predicted class. In this example, the model correctly identifies most ham and spam messages, but it misses one spam (one false negative).

}

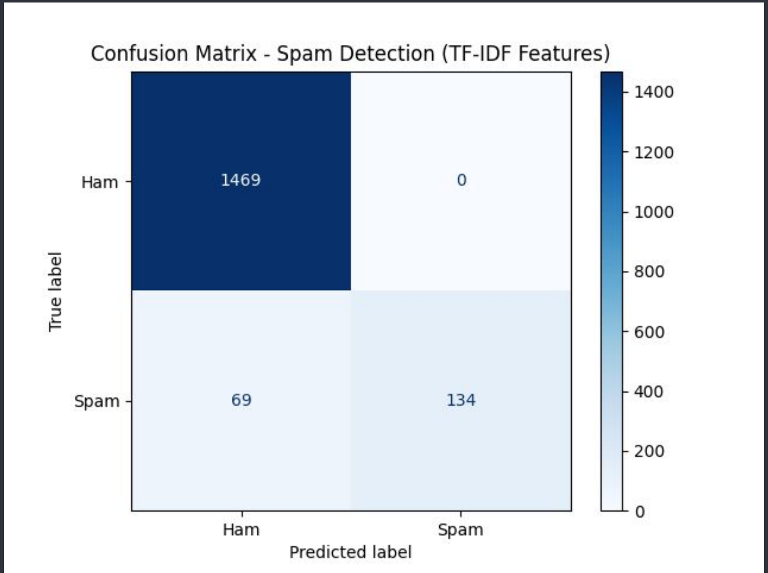


main.py spam.csv

Comparison 'vectorizers' (TF-IDF){

It perfectly classifies all ham messages (no false positives), but it misses some spam messages — specifically, 69 spam messages are incorrectly predicted as ham (false negatives). This indicates that while the model is conservative in labeling messages as spam (high precision), it could benefit from improvements in recall to catch more actual spam.

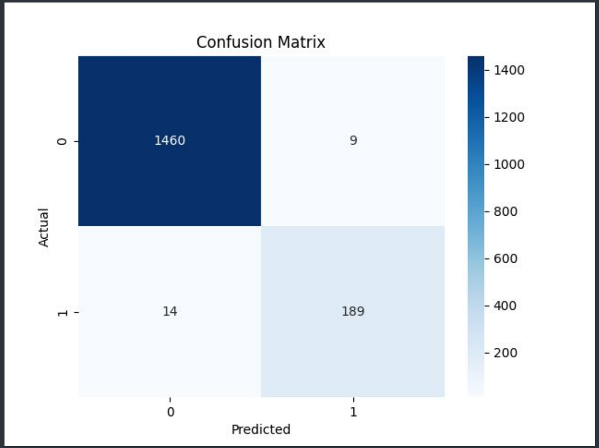
}

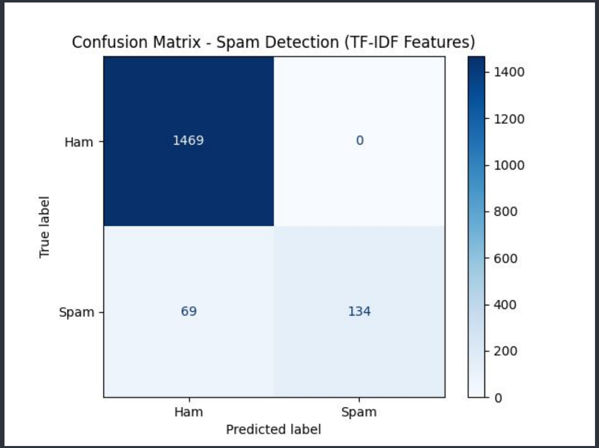


Created by Evan Tan

main.py spam.csv

conclusion{





<MultinomialNB assumes discrete counts of features. It's naturally suited to raw counts rather than continuous TF-IDF weights.

}

TF-IDF transforms the features into floats, which can violate the Naive Bayes assumptions, reducing accuracy.>

Created by Evan Tan

main.py spam.csv

```
Resources {

    Links:

     *  scikit
     *  tokenizer

    Data set:
     *  spam ham data set

}
```

Created by Evan Tan