Spam Email Detector Machine Learning Model

This is my project that I made during my free time to demonstrate knowledge in machine learning and cybersecurity. This model has been created to identify spam/phishing emails.

```
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, precision_score, recall_score, f1_score
import nltk
from nltk.corpus import stopwords
import re
from sklearn.model_selection import train_test_split
```

```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

The file that contains around 5000 emails classified into spam or not spam.

```
data = pd.read_csv('https://github.com/yushika-j/Playing-with-datasets/raw/master/Email%20Spam%20Filtering/emails.csv', encoding='utf-8')
data.head()
```

	text	spam
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

Next steps: Generate code with data | View recommended plots | New interactive sheet

Data Preprocessing

Once loaded, I proceeded to preprocess and vectorize the text data to convert it into a format that a machine learning model can understand. I used TF-IDF or CountVectorizer from scikit-learn to convert the email content into a term-document matrix.

```
print(data.isnull().sum())

# Drop any rows with missing values (if any)
data.dropna(inplace=True)

# Check for duplicates
data.drop_duplicates(inplace=True)
```

```
text    0
spam    0
dtype: int64
```

```
data['text'] = data['text'].str.lower()
#convert to lowercase
```

```
data['text'] = data['text'].apply(lambda x: re.sub(r'[^a-z\s]', '', x))
# remove special characters
```

```
#remove stopwords like 'the' and 'and'
stop_words = set(stopwords.words('english'))
data['text'] = data['text'].apply(lambda x: ' '.join([word for word in x.split() if word not in (stop_words)]))
```

Text Vectorization

To convert the cleaned text into a format that can be used by a machine learning model, use TF-IDF or Count Vectorizer to create a term-document matrix.

This will transform each email into a vector of numerical features where each feature will represent a frequency.

```
# Initialize the TF-IDF Vectorizer
vectorizer = TfidfVectorizer(max_features=1000)  # Limit to 1000 most frequent words
X = vectorizer.fit_transform(data['text']).toarray()  # Transforms text data to numerical array
# Target variable
y = data['spam']
#print the data
print(X)
```

```
[[0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.15747477 0.         0.117228   ... 0.         0.         0.        ]]
```

Train-Test split

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

```

Train a Classification Model

I started with a model like Logistic Regression for simplicity purposes. If the requirements increase, I might change the model to a neural network model.

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

Model Evaluation

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')

```

```
Accuracy: 0.9754170324846356
Precision: 0.9855072463768116
Recall: 0.918918918918919
F1 Score: 0.951048951048951
```

Hyperparameter tuning

For optimal performance, I used parameter tuning to ensure the model works perfectly. In this case, I used GridSearchCV.

```
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.1, 1, 10, 100]}
grid = GridSearchCV(LogisticRegression(), param_grid, refit=True, verbose=2)
grid.fit(X_train, y_train)

```

```
Fitting 5 folds for each of 4 candidates, totalling 20 fits
[CV] END ..............................................C=0.1; total time=   0.1s
[CV] END ..............................................C=0.1; total time=   0.1s
[CV] END ..............................................C=0.1; total time=   0.1s
[CV] END ..............................................C=0.1; total time=   0.1s
[CV] END ..............................................C=0.1; total time=   0.1s
[CV] END ................................................C=1; total time=   0.1s
[CV] END ................................................C=1; total time=   0.1s
[CV] END ................................................C=1; total time=   0.1s
[CV] END ................................................C=1; total time=   0.1s
[CV] END ................................................C=1; total time=   0.1s
[CV] END ...............................................C=10; total time=   0.1s
[CV] END ...............................................C=10; total time=   0.1s
[CV] END ...............................................C=10; total time=   0.1s
[CV] END ...............................................C=10; total time=   0.1s
[CV] END ...............................................C=10; total time=   0.2s
[CV] END ..............................................C=100; total time=   0.1s
[CV] END ..............................................C=100; total time=   0.1s
[CV] END ..............................................C=100; total time=   0.2s
[CV] END ..............................................C=100; total time=   0.1s
[CV] END ..............................................C=100; total time=   0.2s
```

```
▸ GridSearchCV
▸ best_estimator_: LogisticRegression
  ▸ LogisticRegression
```

The model applies a lower penalty for large coefficients, allowing it to fit the data more closely.

```
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)
```

Retrain on the full dataset

```
model = LogisticRegression(C=100)
model.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression(C=100)
```

Cross Validation

Cross-validation is a technique used in machine learning to assess how well a model generalizes to unseen data. Instead of training and testing a model once on a single split of the data, cross-validation divides the data into multiple parts (called "folds") and trains and tests the model multiple times. This process provides a more reliable estimate of model performance and helps prevent overfitting.

```python
# Convert cv_results_ to a DataFrame for easier viewing
cv_results = pd.DataFrame(grid.cv_results_)

# Display relevant columns
print("\nCross-Validation Results:")
print(cv_results[['param_C', 'mean_test_score', 'std_test_score']])
```

```
Cross-Validation Results:
   param_C  mean_test_score  std_test_score
0      0.1         0.911764        0.010151
1      1.0         0.975199        0.006305
2     10.0         0.984637        0.003249
3    100.0         0.985076        0.003768
```