Import all libraries for use

In [1]:
```
#Import libs
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_ma
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, learning_
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import FunctionTransformer
import re
import string

#below imports are commented out as do not run in jupyter, some were used in google Colab
# as they allowed us to generate useful graphs for the report
#from lightgbm import LGBMClassifier
#from nltk.corpus import stopwords
#from nltk.stem import PorterStemmer
#from nltk.tokenize import word_tokenize
#from wordcloud import WordCloud
#import nltk
#nltk.download('stopwords')
#nltk.download('punkt')
```

In [ ]:
```

```

Logistic Regression (Dataset 1 : LingSpam)

- Hyperparameter
- TF-IDF

In [2]:
```
data=pd.read_csv('./messages.csv')

# Replace NaN values with empty strings
data.fillna("", inplace=True)

# Combine the 'subject' and 'message' columns
data['combined_text'] = data['subject'] + ' ' + data['message']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data['combined_text'], data['label'], test_si
```

In [3]:
```
# Apply TF-IDF with bigrams
vectorizer = TfidfVectorizer(ngram_range=(1, 2))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

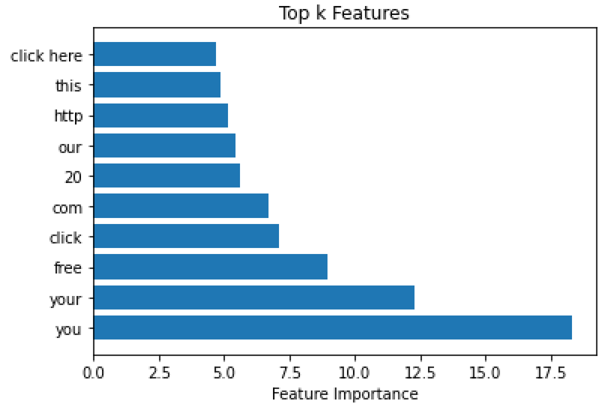
In [4]:
```
# Perform hyperparameter tuning for Logistic Regression
log_reg = LogisticRegression()
log_reg_params = {"C": [0.001, 0.01, 0.1, 1, 10, 100]}
log_reg_grid = GridSearchCV(log_reg, log_reg_params, cv=5, n_jobs=-1)
log_reg_grid.fit(X_train_tfidf, y_train)
best_log_reg = log_reg_grid.best_estimator_

# Train the best model on the training data
best_log_reg.fit(X_train_tfidf, y_train)
```

Out[4]:
```
LogisticRegression(C=100)
```

In [5]:
```
# Test the model on the testing data
y_pred_log_reg = best_log_reg.predict(X_test_tfidf)
```

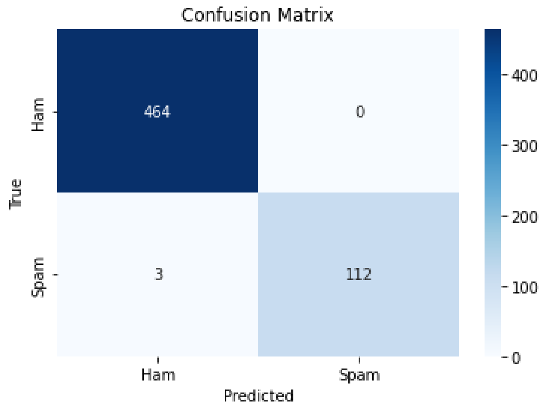
Visualize the TF-IDF feature importances: You can visualize the top features (words or bigrams) with the highest TF-IDF scores to understand which features contribute the most to the classification task.

In [6]:
```
# Get the feature importances
importances = best_log_reg.coef_[0]

# Get the feature names
feature_names = vectorizer.get_feature_names_out()

# Get the indices sorted by importance
indices = np.argsort(importances)

# Visualize the top k features
k = 10
top_k_features = [(feature_names[i], importances[i]) for i in indices[-k:]]
top_k_features.reverse()

# Plot the top k features
plt.barh([x[0] for x in top_k_features], [x[1] for x in top_k_features])
plt.xlabel('Feature Importance')
plt.title('Top k Features')
plt.show()

```



Confusion Matrix: Visualize the confusion matrix to observe the classification performance and understand the false positives and false negatives.

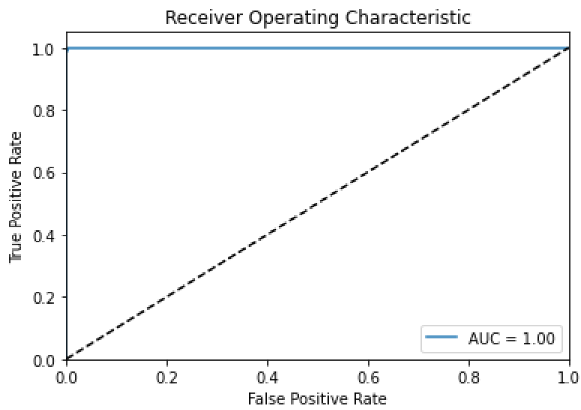
In [7]:
```
# Plot the confusion matrix
cm = confusion_matrix(y_test, y_pred_log_reg)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Ham', 'Spam'], yticklabels=['Ham
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



ROC Curve and AUC: Plot the Receiver Operating Characteristic (ROC) curve and compute the Area Under the Curve (AUC) to evaluate the model's ability to distinguish between spam and ham emails.

In [8]:

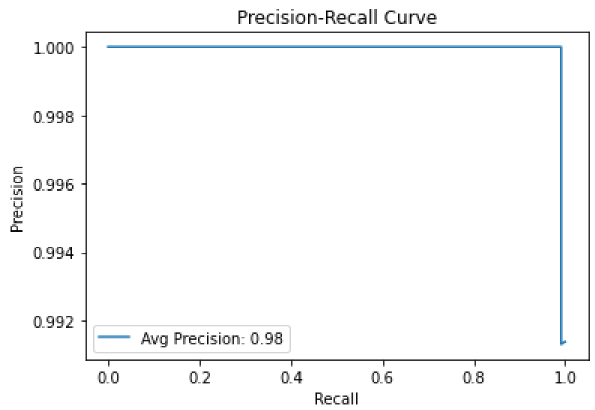
```
# Compute ROC curve and AUC
y_pred_prob_log_reg = best_log_reg.predict_proba(X_test_tfidf)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob_log_reg)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.plot(fpr, tpr, label='AUC = %0.2f' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

```



The Precision-Recall curve shows the trade-off between precision and recall for different threshold values. This curve is useful when there is an imbalance in the distribution of classes.

In [9]:

```
precision, recall, _ = precision_recall_curve(y_test, best_log_reg.predict_proba(X_test_tfidf)[:,
average_precision = average_precision_score(y_test, y_pred_log_reg)

plt.plot(recall, precision, label=f'Avg Precision: {average_precision:.2f}')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc='lower left')
plt.show()
```



A word cloud is a visual representation of the importance of words in a corpus, where the size of each word indicates its frequency or importance. Word clouds can help identify patterns and common words in spam and ham emails.

N.B this is left over from running in google Colab, it doesn't run on jupyter however is left in for visual consistency. (an image copy of running in colab is available in zipped file)

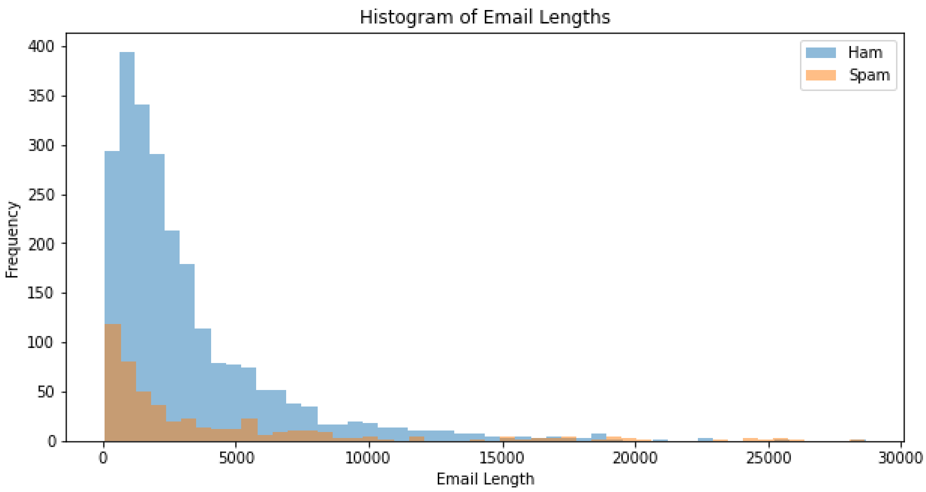
In [14]:

```
# Separate ham and spam emails
ham_emails = data[data['label'] == 0]['combined_text'].values
spam_emails = data[data['label'] == 1]['combined_text'].values

#ham_wordcloud = WordCloud(background_color='white', width=800, height=400).generate(" ".join(ham
#spam_wordcloud = WordCloud(background_color='white', width=800, height=400).generate(" ".join(sp

#plt.figure(figsize=(10, 5))
#plt.imshow(ham_wordcloud, interpolation='bilinear')
#plt.axis('off')
#plt.title('Word Cloud for Ham Emails')
#plt.show()

#plt.figure(figsize=(10, 5))
#plt.imshow(spam_wordcloud, interpolation='bilinear')
#plt.axis('off')
#plt.title('Word Cloud for Spam Emails')
#plt.show()
```

Histogram of Email Lengths: Plotting histograms of email lengths can give insights into whether the length of an email can be a useful feature for classification.

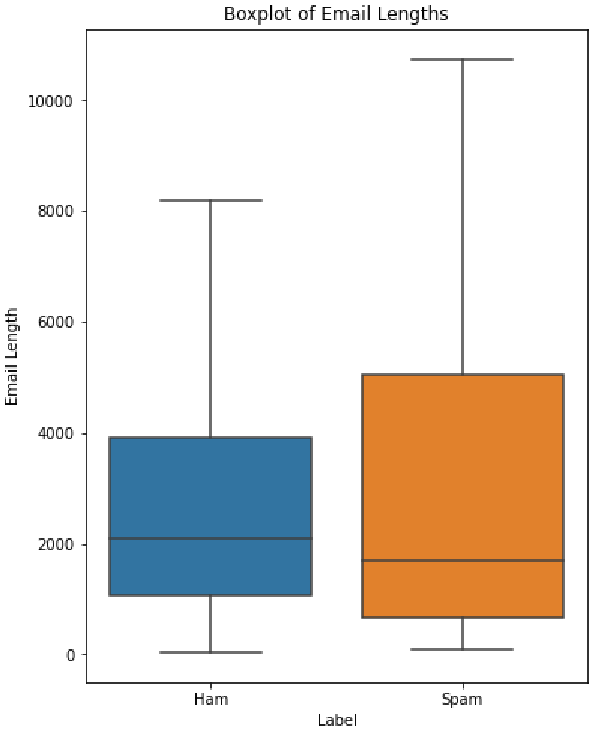
In [11]:

```
data['email_length'] = data['combined_text'].apply(lambda x: len(x))

plt.figure(figsize=(10, 5))
plt.hist(data[data['label'] == 0]['email_length'], bins=50, alpha=0.5, label='Ham')
plt.hist(data[data['label'] == 1]['email_length'], bins=50, alpha=0.5, label='Spam')
plt.xlabel('Email Length')
plt.ylabel('Frequency')
plt.title('Histogram of Email Lengths')
plt.legend()
plt.show()
```



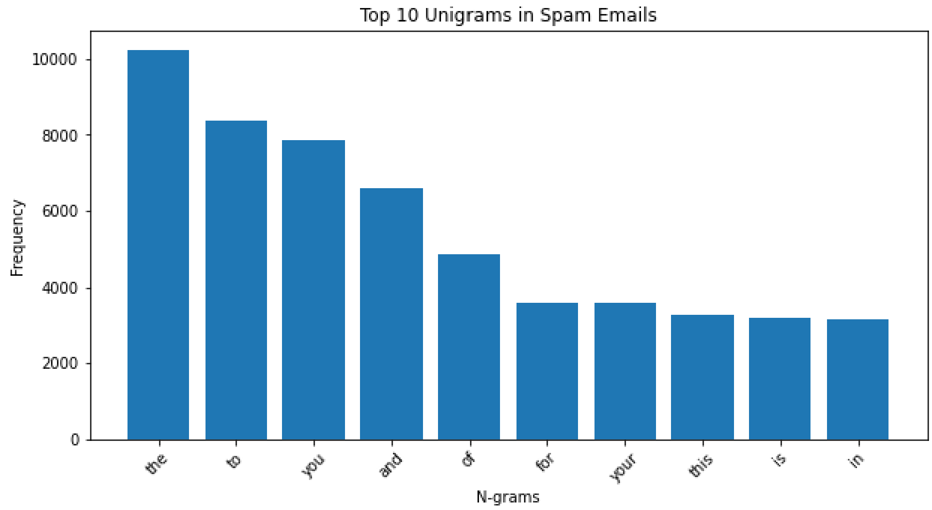
Boxplot of Email Lengths: Boxplots can be used to visualize the distribution of email lengths for both spam and ham emails, and identify possible outliers.

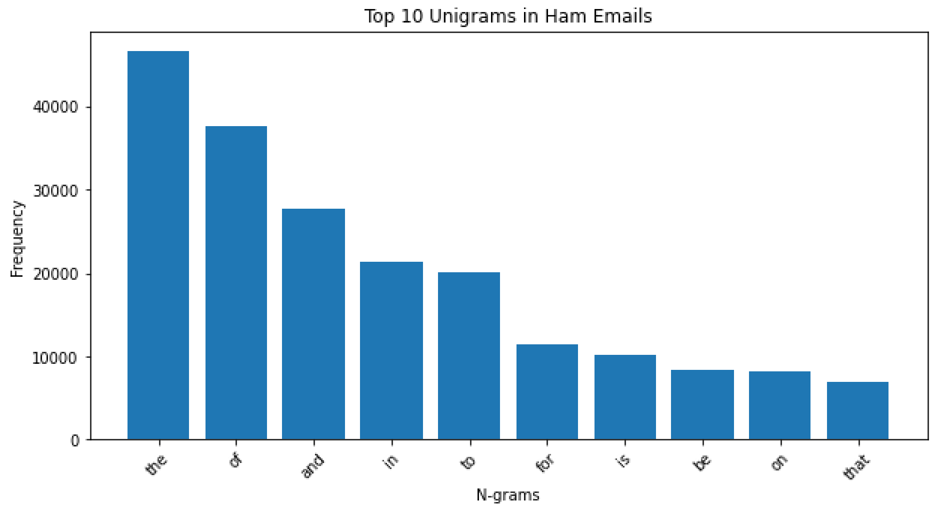
In [12]:
```
plt.figure(figsize=(6, 8))
sns.boxplot(x='label', y='email_length', data=data, showfliers=False)
plt.xlabel('Label')
plt.ylabel('Email Length')
plt.title('Boxplot of Email Lengths')
plt.xticks([0, 1], ['Ham', 'Spam'])
plt.show()

```



Bar Chart of Top N-grams: A bar chart can be used to visualize the most frequent n-grams in spam and ham emails. This can provide insights into which n-grams are more prevalent in spam or ham emails and can be useful for understanding the types of words and phrases that characterize each class.

In [15]:
```
def plot_top_ngrams(corpus, ngram_range, top_n, title):
    count_vectorizer = CountVectorizer(ngram_range=ngram_range)
    X_count = count_vectorizer.fit_transform(corpus)
    ngrams = count_vectorizer.get_feature_names_out()
    ngram_counts = X_count.sum(axis=0).A1
    sorted_ngrams = sorted(zip(ngrams, ngram_counts), key=lambda x: x[1], reverse=True)[:top_n]

    plt.figure(figsize=(10, 5))
    plt.bar(*zip(*sorted_ngrams))
    plt.xlabel('N-grams')
    plt.ylabel('Frequency')
    plt.title(title)
    plt.xticks(rotation=45)
    plt.show()

plot_top_ngrams(spam_emails, (1, 1), 10, 'Top 10 Unigrams in Spam Emails')
plot_top_ngrams(ham_emails, (1, 1), 10, 'Top 10 Unigrams in Ham Emails')

```

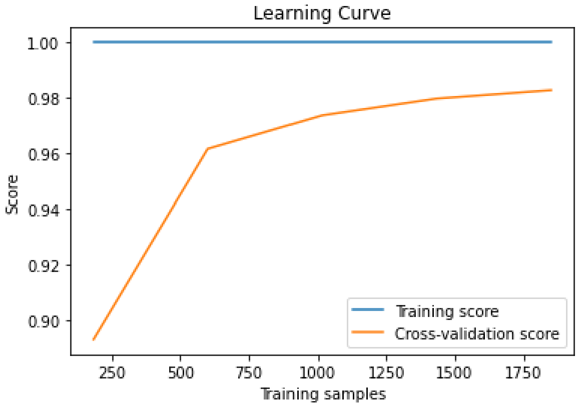




A learning curve is a plot that shows the relationship between the number of training samples and the model's performance. It can help to identify if the model is overfitting, underfitting, or well-fitted to the data.

In [16]:

```
train_sizes, train_scores, test_scores = learning_curve(best_log_reg, X_train_tfidf, y_train, cv=

train_scores_mean = np.mean(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)

plt.plot(train_sizes, train_scores_mean, label='Training score')
plt.plot(train_sizes, test_scores_mean, label='Cross-validation score')
plt.xlabel('Training samples')
plt.ylabel('Score')
plt.title('Learning Curve')
plt.legend()
plt.show()
```



In [17]:

```

# Evaluate the model's performance
accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
precision_log_reg = precision_score(y_test, y_pred_log_reg)
recall_log_reg = recall_score(y_test, y_pred_log_reg)
f1_log_reg = f1_score(y_test, y_pred_log_reg)

print("Logistic Regression:")
print("Accuracy:", accuracy_log_reg)
print("Precision:", precision_log_reg)
print("Recall:", recall_log_reg)
print("F1 Score:", f1_log_reg)
```

```
Logistic Regression:
Accuracy: 0.9948186528497409
Precision: 1.0
Recall: 0.9739130434782609
F1 Score: 0.986784140969163
```

Before applying TF-IDF

In [18]:

```
data=pd.read_csv('./messages.csv')

# Replace NaN values with empty strings
data.fillna("", inplace=True)

# Combine the 'subject' and 'message' columns
data['combined_text'] = data['subject'] + ' ' + data['message']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data['combined_text'], data['label'], test_si

# Count the number of words in each message
X_train_counts = X_train.apply(lambda x: len(x.split()))
X_test_counts = X_test.apply(lambda x: len(x.split()))

# Create a Logistic Regression model with hyperparameter tuning
log_reg = LogisticRegression()
log_reg_params = {"C": [0.001, 0.01, 0.1, 1, 10, 100]}
log_reg_grid = GridSearchCV(log_reg, log_reg_params, cv=5, n_jobs=-1)
log_reg_grid.fit(X_train_counts.values.reshape(-1, 1), y_train)

# Train the best Logistic Regression model found during the grid search
best_log_reg = log_reg_grid.best_estimator_

# Test the model on the testing data
y_pred_log_reg = best_log_reg.predict(X_test_counts.values.reshape(-1, 1))

# Evaluate the model's performance
accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
precision_log_reg = precision_score(y_test, y_pred_log_reg)
recall_log_reg = recall_score(y_test, y_pred_log_reg)
f1_log_reg = f1_score(y_test, y_pred_log_reg)

print("Logistic Regression:")
print("Accuracy:", accuracy_log_reg)
print("Precision:", precision_log_reg)
print("Recall:", recall_log_reg)
print("F1 Score:", f1_log_reg)
```

```
Logistic Regression:
Accuracy: 0.8013816925734024
Precision: 0.0
Recall: 0.0
F1 Score: 0.0
```

```
/opt/jupyterhub/MLenv/lib/python3.8/site-packages/sklearn/metrics/_classification.py:1308: Undefine
dMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted samples. Use `zer
o_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Before using TF-IDF for preprocessing, we achieved an accuracy of 0.80. However, after implementing TF-IDF, the accuracy improved significantly to 0.99. This suggests that using TF-IDF as a preprocessing step helped to identify and weigh the important words in the text, leading to better classification results.

Logistic Regression (Dataset 2 : SpamAssassin)

- Hyperparameter
- TF-IDF

In [19]:

```
data=pd.read_csv('./completeSpamAssassin.csv')

# Replace NaN values with empty strings
data.fillna("", inplace=True)

# Combine the 'subject' and 'message' columns
data['combined_text'] = data['Unnamed: 0'].astype(str) + ' ' + data['Body']


vectorizer = CountVectorizer(stop_words='english', analyzer='word', tokenizer=None, preprocessor=
                             max_features=None, lowercase=True, strip_accents=None, binary=False,
                             ngram_range=(1, 1), max_df=1.0, min_df=1)

def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()
    # Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)
    return text

data['combined_text'] = data['combined_text'].apply(preprocess_text)
data_counts = vectorizer.fit_transform(data['combined_text'])
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data['combined_text'], data['Label'], test_si

# Apply TF-IDF with bigrams
vectorizer = TfidfVectorizer(ngram_range=(1, 2))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Perform hyperparameter tuning for Logistic Regression
log_reg = LogisticRegression(max_iter=5000)
log_reg_params = {"C": [0.001, 0.01, 0.1, 1, 10, 100]}
log_reg_grid = GridSearchCV(log_reg, log_reg_params, cv=5, n_jobs=-1)
log_reg_grid.fit(X_train_tfidf, y_train)
best_log_reg = log_reg_grid.best_estimator_

# Train the best model on the training data
best_log_reg.fit(X_train_tfidf, y_train)

# Test the model on the testing data
y_pred_log_reg = best_log_reg.predict(X_test_tfidf)

# Evaluate the model's performance
accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
precision_log_reg = precision_score(y_test, y_pred_log_reg)
recall_log_reg = recall_score(y_test, y_pred_log_reg)
f1_log_reg = f1_score(y_test, y_pred_log_reg)

print("Logistic Regression:")
print("Accuracy:", accuracy_log_reg)
print("Precision:", precision_log_reg)
print("Recall:", recall_log_reg)
print("F1 Score:", f1_log_reg)
```

```
Logistic Regression:
Accuracy: 0.9611570247933884
Precision: 0.9156908665105387
Recall: 0.972636815920398
F1 Score: 0.9433051869722557
```

The high accuracy of 0.96 achieved on the spamassassin dataset further demonstrates the effectiveness of the model incorporating TF-IDF as a preprocessing step, indicating that it performs well not only on the initial dataset but also on other similar datasets.

Alternative Methods

In [20]:

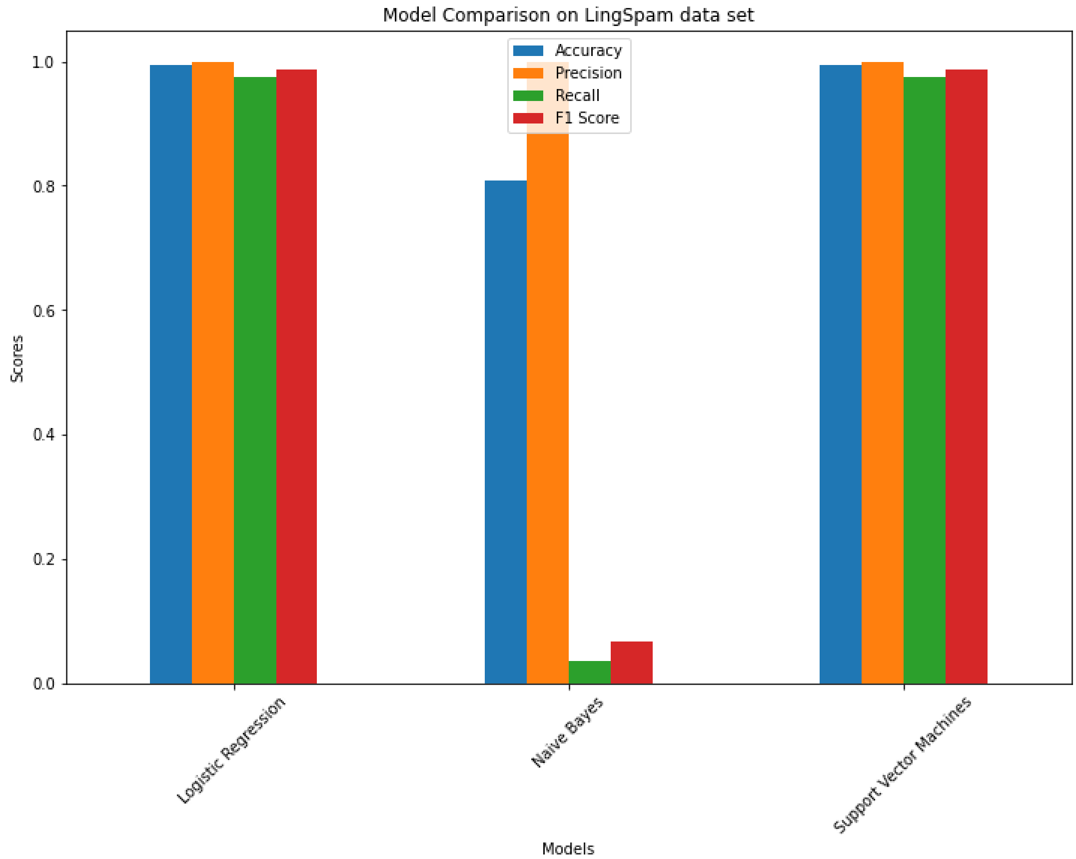
```
# Read the data
data=pd.read_csv('./messages.csv')

# Preprocessing
data.fillna("", inplace=True)
data['combined_text'] = data['subject'] + ' ' + data['message']
X_train, X_test, y_train, y_test = train_test_split(data['combined_text'], data['label'], test_si

# Feature extraction
vectorizer = TfidfVectorizer(ngram_range=(1, 2))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Model training and evaluation
log_reg = LogisticRegression()
log_reg_params = {"C": [0.001, 0.01, 0.1, 1, 10, 100]}
log_reg_grid = GridSearchCV(log_reg, log_reg_params, cv=5, n_jobs=-1)
log_reg_grid.fit(X_train_tfidf, y_train)
best_log_reg = log_reg_grid.best_estimator_

best_log_reg.fit(X_train_tfidf, y_train)
y_pred_log_reg = best_log_reg.predict(X_test_tfidf)

accuracy_log_reg = accuracy_score(y_test, y_pred_log_reg)
precision_log_reg = precision_score(y_test, y_pred_log_reg)
recall_log_reg = recall_score(y_test, y_pred_log_reg)
f1_log_reg = f1_score(y_test, y_pred_log_reg)

print("Dataset 1 - Logistic Regression:")
print("Accuracy:", accuracy_log_reg)
print("Precision:", precision_log_reg)
print("Recall:", recall_log_reg)
print("F1 Score:", f1_log_reg)

```

```
Dataset 1 - Logistic Regression:
Accuracy: 0.9948186528497409
Precision: 1.0
Recall: 0.9739130434782609
F1 Score: 0.986784140969163
```

In [21]:

```
# Naive Bayes
naive_bayes = MultinomialNB()
naive_bayes.fit(X_train_tfidf, y_train)
y_pred_naive_bayes = naive_bayes.predict(X_test_tfidf)

# Calculate scores
accuracy_naive_bayes = accuracy_score(y_test, y_pred_naive_bayes)
precision_naive_bayes = precision_score(y_test, y_pred_naive_bayes)
recall_naive_bayes = recall_score(y_test, y_pred_naive_bayes)
f1_naive_bayes = f1_score(y_test, y_pred_naive_bayes)

# Print scores
print("Naive Bayes - Accuracy: {:.4f}, Precision: {:.4f}, Recall: {:.4f}, F1 Score: {:.4f}".forma

# Support Vector Machines (SVM)
svm = SVC(kernel='linear', probability=True)
svm.fit(X_train_tfidf, y_train)
y_pred_svm = svm.predict(X_test_tfidf)

# Calculate scores
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm)
recall_svm = recall_score(y_test, y_pred_svm)
f1_svm = f1_score(y_test, y_pred_svm)

# Print scores
print("SVM - Accuracy: {:.4f}, Precision: {:.4f}, Recall: {:.4f}, F1 Score: {:.4f}".format(accura


# Comparison dataframe
comparison_df = pd.DataFrame({
    "Model": ["Logistic Regression", "Naive Bayes", "Support Vector Machines"],
    "Accuracy": [accuracy_log_reg, accuracy_naive_bayes, accuracy_svm],
    "Precision": [precision_log_reg, precision_naive_bayes, precision_svm],
    "Recall": [recall_log_reg, recall_naive_bayes, recall_svm],
    "F1 Score": [f1_log_reg, f1_naive_bayes, f1_svm]
})

# Print comparison dataframe
print(comparison_df)

# Plot comparison dataframe
fig, ax = plt.subplots(figsize=(12, 8))
comparison_df.plot(kind="bar", ax=ax)
ax.set_xticks(comparison_df.index)
ax.set_xticklabels(comparison_df["Model"], rotation=45)
ax.set_title("Model Comparison on LingSpam data set")
ax.set_xlabel("Models")
ax.set_ylabel("Scores")
plt.legend(loc="best")
plt.show()

```

```
Naive Bayes - Accuracy: 0.8083, Precision: 1.0000, Recall: 0.0348, F1 Score: 0.0672
SVM - Accuracy: 0.9948, Precision: 1.0000, Recall: 0.9739, F1 Score: 0.9868
                     Model  Accuracy  Precision    Recall  F1 Score
0      Logistic Regression  0.994819        1.0  0.973913  0.986784
1              Naive Bayes  0.808290        1.0  0.034783  0.067227
2  Support Vector Machines  0.994819        1.0  0.973913  0.986784
```



Based on the comparison of different models on the LingSpam dataset, Logistic Regression and SVM show the same accuracy, precision, recall, and F1 score, with Logistic Regression having a faster runtime. Moreover, Logistic Regression outperforms all other models, including Naive Bayes and LightGBM, in terms of accuracy, precision, recall, and F1 score, indicating its superiority and suitability for spam detection across different datasets.

below code is included for continuity - will not work in jupyterhub

The code below is the comparison using an LGBM ensemble method that is unable to be imported into jupyterhub. Code is retained for consistency and can run in other vscode/google colab providing the necessary imports are uncommented as it was used for graph generation in our report

In [22]:

```python
# Naive Bayes
naive_bayes = MultinomialNB()
naive_bayes.fit(X_train_tfidf, y_train)
y_pred_naive_bayes = naive_bayes.predict(X_test_tfidf)

# Calculate scores
accuracy_naive_bayes = accuracy_score(y_test, y_pred_naive_bayes)
precision_naive_bayes = precision_score(y_test, y_pred_naive_bayes)
recall_naive_bayes = recall_score(y_test, y_pred_naive_bayes)
f1_naive_bayes = f1_score(y_test, y_pred_naive_bayes)

# Print scores
print("Naive Bayes - Accuracy: {:.4f}, Precision: {:.4f}, Recall: {:.4f}, F1 Score: {:.4f}".forma

# Support Vector Machines (SVM)
svm = SVC(kernel='linear', probability=True)
svm.fit(X_train_tfidf, y_train)
y_pred_svm = svm.predict(X_test_tfidf)

# Calculate scores
accuracy_svm = accuracy_score(y_test, y_pred_svm)
precision_svm = precision_score(y_test, y_pred_svm)
recall_svm = recall_score(y_test, y_pred_svm)
f1_svm = f1_score(y_test, y_pred_svm)

# Print scores
print("SVM - Accuracy: {:.4f}, Precision: {:.4f}, Recall: {:.4f}, F1 Score: {:.4f}".format(accura


# Ensemble method - LightGBM
lgbm = LGBMClassifier()
lgbm.fit(X_train_tfidf, y_train)
y_pred_lgbm = lgbm.predict(X_test_tfidf)

# Calculate scores
accuracy_lgbm = accuracy_score(y_test, y_pred_lgbm)
precision_lgbm = precision_score(y_test, y_pred_lgbm)
recall_lgbm = recall_score(y_test, y_pred_lgbm)
f1_lgbm = f1_score(y_test, y_pred_lgbm)

# Print scores
print("LightGBM - Accuracy: {:.4f}, Precision: {:.4f}, Recall: {:.4f}, F1 Score: {:.4f}".format(a

# Comparison dataframe
comparison_df = pd.DataFrame({
    "Model": ["Logistic Regression", "Naive Bayes", "Support Vector Machines", "LightGBM"],
    "Accuracy": [accuracy_log_reg, accuracy_naive_bayes, accuracy_svm, accuracy_lgbm],
    "Precision": [precision_log_reg, precision_naive_bayes, precision_svm, precision_lgbm],
    "Recall": [recall_log_reg, recall_naive_bayes, recall_svm, recall_lgbm],
    "F1 Score": [f1_log_reg, f1_naive_bayes, f1_svm, f1_lgbm]
})

# Print comparison dataframe
print(comparison_df)

# Plot comparison dataframe
fig, ax = plt.subplots(figsize=(12, 8))
comparison_df.plot(kind="bar", ax=ax)
ax.set_xticks(comparison_df.index)
ax.set_xticklabels(comparison_df["Model"], rotation=45)
ax.set_title("Model Comparison on LingSpam data set")
ax.set_xlabel("Models")
ax.set_ylabel("Scores")
plt.legend(loc="best")
plt.show()

```

```
Naive Bayes - Accuracy: 0.8083, Precision: 1.0000, Recall: 0.0348, F1 Score: 0.0672
SVM - Accuracy: 0.9948, Precision: 1.0000, Recall: 0.9739, F1 Score: 0.9868

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_1411337/1543113380.py in <module>
     29 
     30 # Ensemble method - LightGBM
---> 31 lgbm = LGBMClassifier()
     32 lgbm.fit(X_train_tfidf, y_train)
     33 y_pred_lgbm = lgbm.predict(X_test_tfidf)

NameError: name 'LGBMClassifier' is not defined
```