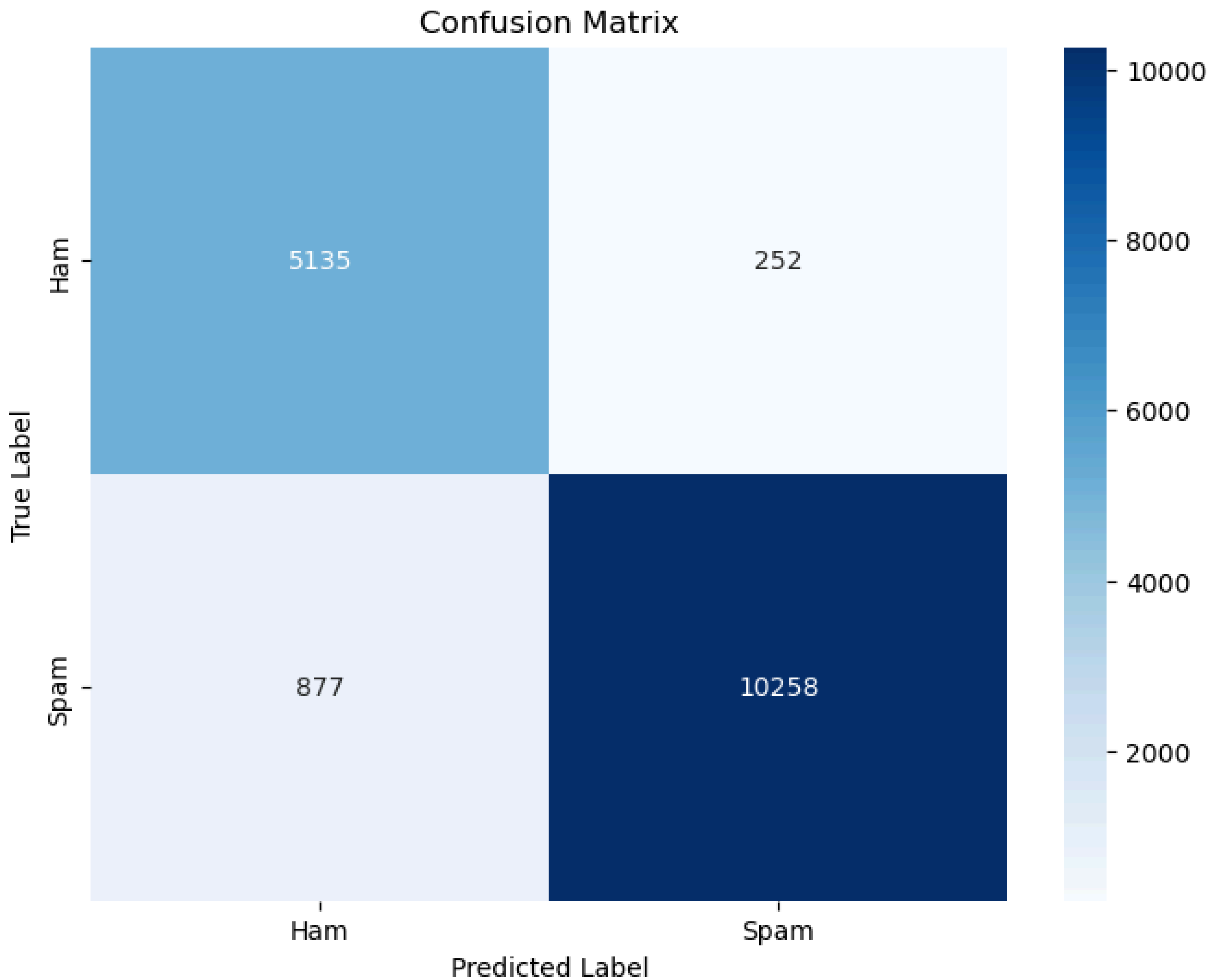
```python
# test_df should contain the columns with true and predicted labels
metrics_no_stopwords = evaluate_model(test_df, 'category', 'predicted')
```

```
Accuracy: 0.9316668684178671 = 93.17%
Precision: 0.976022835394862 = 97.60%
Recall: 0.921239335428828 = 92.12%
F1-Score: 0.9478401478401478 = 94.78%
```



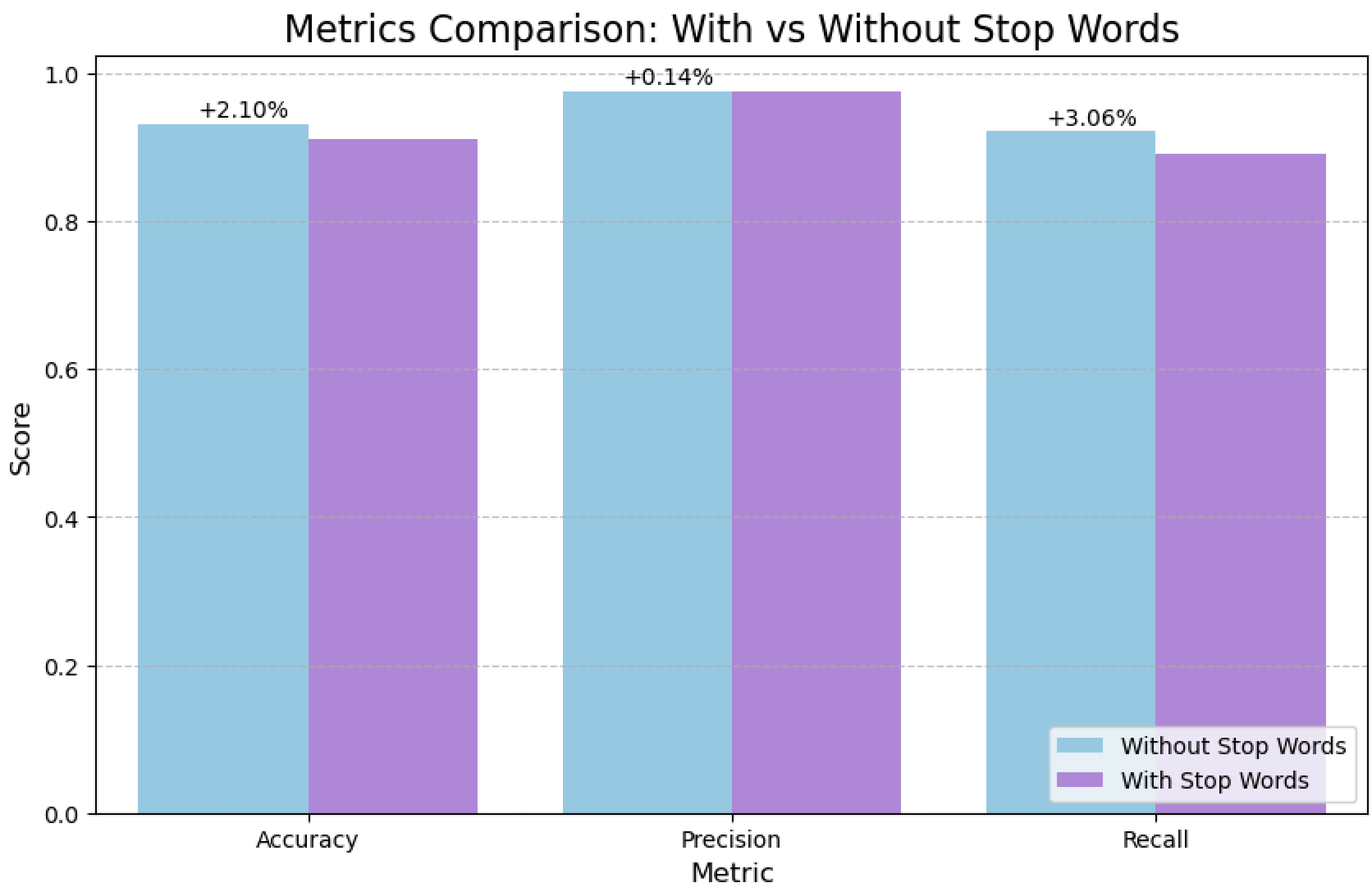
Results and Discussion

1. What is the effect of removing stop words in terms of precision, recall, and accuracy? Show a plot or a table of these results.

I created a **different notebook (with-stopwords.ipynb)** that contains the metric results for the model without removing the stop words in the dataset. So, I will just manually enter here the values that were already computed.

In [53]:
```
colors = ['#89CFF0', '#b47ee5']

# Create a df for the metrics
metrics_w_stopwords = {
    'Accuracy': 0.9106645684541823,
    'Precision': 0.9746437346437347,
    'Recall': 0.8906151773686574
}

metrics_wo_stopwords = {
    'Accuracy': 0.9316668684178671,
    'Precision': 0.976022835394862,
    'Recall': 0.921239335428828
}

metrics_df = pd.DataFrame({
    'Metric': ['Accuracy', 'Precision', 'Recall'],
    'Without Stop Words': [metrics_wo_stopwords['Accuracy'], metrics_wo_stopwords['Precision'], metrics_wo_stopwords[
    'With Stop Words': [metrics_w_stopwords['Accuracy'], metrics_w_stopwords['Precision'], metrics_w_stopwords['Recal
})

# Melt the DataFrame for easier plotting
metrics_melted = metrics_df.melt(id_vars='Metric', var_name='Condition', value_name='Value')

# Plot the bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x='Metric', y='Value', hue='Condition', data=metrics_melted, palette=colors)

# Add titles and labels
plt.title('Metrics Comparison: With vs Without Stop Words', fontsize=16)
plt.ylabel('Score', fontsize=12)
plt.xlabel('Metric', fontsize=12)
plt.legend(loc='lower right')
```

```
# Adding grid lines for better readability
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Show the difference between the two conditions
for index, row in metrics_df.iterrows():
    diff = row['Without Stop Words'] - row['With Stop Words']
    plt.text(index - 0.15, row['Without Stop Words'] + 0.01, f"+{diff:.2%}", color='black', ha="center", fontsize=10)

# Show plot
plt.show()
```



It is evident that removing stopwords improved the model's performance across the metrics, since removing non-informative words made it focus more on meaningful content. This leads to a better spam detection and fewer misclassifications.

2. Experiment on the number of words used for training. Filter the dictionary to include only words occurring more than k times (1000 words, then k > 100, and k = 50 times). For example, the word "offer" appears 150 times, that means that it will be included in the dictionary.

In [124…

```
# Filter the dictionary by word frequency for different k thresholds
filtered_dict_1000 = {word: idx for idx, (word, count) in enumerate(word_counter.items()) if count > 1000}
filtered_dict_100 = {word: idx for idx, (word, count) in enumerate(word_counter.items()) if count > 100}
filtered_dict_50 = {word: idx for idx, (word, count) in enumerate(word_counter.items()) if count > 50}

# Create lists of the filtered words
filtered_list_1000 = list(filtered_dict_1000.keys())
filtered_list_100 = list(filtered_dict_100.keys())
filtered_list_50 = list(filtered_dict_50.keys())

# Output the number of words for each threshold
print(f"Words with frequency > 1000: {len(filtered_dict_1000)}")
print(f"Words with frequency > 100: {len(filtered_dict_100)}")
print(f"Words with frequency = 50: {len(filtered_dict_50)}")
```

```
Words with frequency > 1000: 168
Words with frequency > 100: 2755
Words with frequency = 50: 4833
```

In [140…

```
# Redefine likelihood function to be used with filtered dictionary
def calculate_likelihoods_with_laplace_filtered(matrix, filtered_dict, lambda_smoothing):

    # Initialize likelihoods for the filtered dictionary size
    vocab_size = len(filtered_dict)
    likelihoods = np.zeros(vocab_size)

    # Calculate the total word count in the feature matrix (sum over all emails)
    word_counts = np.sum(matrix, axis=0)  # Sum across all rows (emails)
    total_word_count = np.sum(word_counts)  # Total number of words in the entire feature matrix

    # Calculate likelihoods with Laplace smoothing for each word in the filtered dictionary
    for i in range (vocab_size):
        likelihoods[i] = (word_counts[i] + lambda_smoothing) / (total_word_count + lambda_smoothing * vocab_size)

    return likelihoods
```

In [148…

```python
# Redefine the classify function to be used with filtered dictionary
def classify(email, spam_word_probs, ham_word_probs, p_spam, p_ham, filtered_dict, filtered_list):
    # Initialize the log probability of spam and ham
    log_p_spam = 0
    log_p_ham = 0

    # Split the email into words
    words = str(email).split()

    # Compute the log probability of spam and ham
    for word in words:
        if word in filtered_dict:
            log_p_spam += np.log(spam_word_probs[filtered_list.index(word)])
            log_p_ham += np.log(ham_word_probs[filtered_list.index(word)])

    # Add the log probability of spam and ham
    log_p_spam += np.log(p_spam)
    log_p_ham += np.log(p_ham)

    # Return the class with the highest log probability
    return 1 if log_p_spam > log_p_ham else 0
```

In [107…

```python
print(f"Filtered Dictionary Size (k > 1000): {len(filtered_dict_1000)}")
print(f"Feature Matrix Shape (spam): {spam_train_matrix_1000.shape}")
print(f"Feature Matrix Shape (ham): {ham_train_matrix_1000.shape}")
```

```
Filtered Dictionary Size (k > 1000): 168
Feature Matrix Shape (spam): (13777, 168)
Feature Matrix Shape (ham): (7523, 168)
```

In [136…

```python
# Create feature matrices for spam and ham emails using the re-indexed filtered dictionaries
spam_train_matrix_1000 = create_feature_matrix(train_spam_df, list(filtered_dict_1000.keys()))
ham_train_matrix_1000 = create_feature_matrix(train_ham_df, list(filtered_dict_1000.keys()))

spam_train_matrix_100 = create_feature_matrix(train_spam_df, list(filtered_dict_100.keys()))
ham_train_matrix_100 = create_feature_matrix(train_ham_df, list(filtered_dict_100.keys()))

spam_train_matrix_50 = create_feature_matrix(train_spam_df, list(filtered_dict_50.keys()))
ham_train_matrix_50 = create_feature_matrix(train_ham_df, list(filtered_dict_50.keys()))
```

In [150…

```
# Calculate likelihoods using Laplace smoothing for each threshold
lambda_smoothing = 1  # Laplace smoothing parameter

# For k > 1000
likelihood_spam_1000 = calculate_likelihoods_with_laplace_filtered(spam_train_matrix_1000, filtered_dict_1000, lambda
likelihood_ham_1000 = calculate_likelihoods_with_laplace_filtered(ham_train_matrix_1000, filtered_dict_1000, lambda_s

# For k > 100
likelihood_spam_100 = calculate_likelihoods_with_laplace_filtered(spam_train_matrix_100, filtered_dict_100, lambda_sm
likelihood_ham_100 = calculate_likelihoods_with_laplace_filtered(ham_train_matrix_100, filtered_dict_100, lambda_smoo

# For k > 50
likelihood_spam_50 = calculate_likelihoods_with_laplace_filtered(spam_train_matrix_50, filtered_dict_50, lambda_smoot
likelihood_ham_50 = calculate_likelihoods_with_laplace_filtered(ham_train_matrix_50, filtered_dict_50, lambda_smoothi
```

In [156…

```
test_df_k = test_df.copy()
test_df_k.drop('predicted', axis=1, inplace=True)
# Classify test emails for k > 1000
test_df_k['predicted_k1000'] = test_df_k['email_message'].apply(
    lambda x: classify(x, likelihood_spam_1000, likelihood_ham_1000, p_spam, p_ham, filtered_dict_1000, filtered_list
)

# Classify test emails for k > 100
test_df_k['predicted_k100'] = test_df_k['email_message'].apply(
    lambda x: classify(x, likelihood_spam_100, likelihood_ham_100, p_spam, p_ham, filtered_dict_100, filtered_list_10
)

# Classify test emails for k > 50
test_df_k['predicted_k50'] = test_df_k['email_message'].apply(
    lambda x: classify(x, likelihood_spam_50, likelihood_ham_50, p_spam, p_ham, filtered_dict_50, filtered_list_50)
)

test_df_k
```

Out[156…

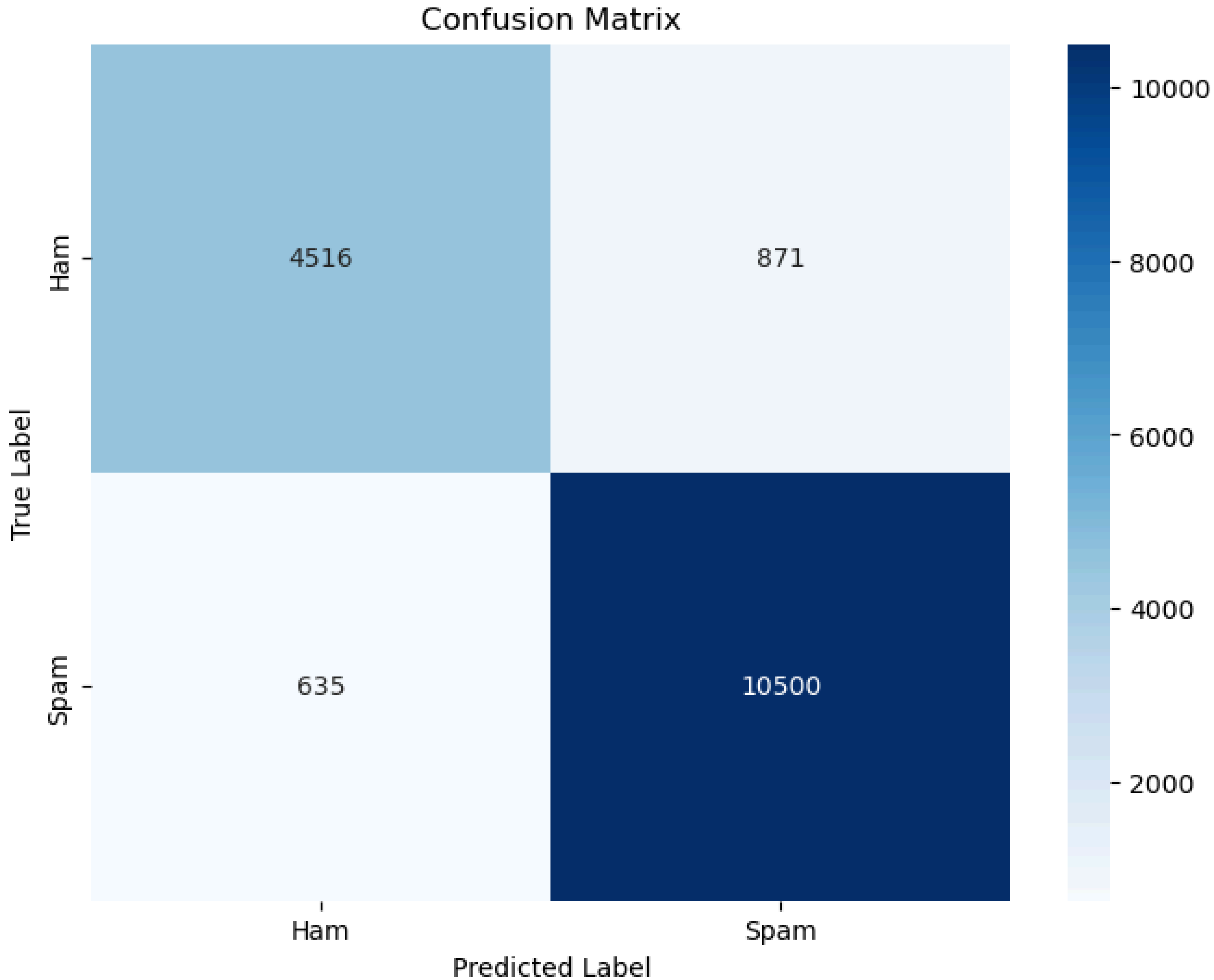
	folder	file	email_message	category	predicted_k1000	predicted_k100	predicted_k50
21300	71	0	hesitantly derive perverse satisfaction clodho...	1	1	1	1
21301	71	1	things perform experiment display will remain ...	0	0	0	0
21302	71	2	best offer month viggra ci ialis vaiium xa naa...	1	1	1	1
21303	71	3	de ar wne cr doesnt matter ow real st mmed ia ...	1	1	1	1
21304	71	4	special offer adobe video collection adobe pre...	1	1	1	1
...	...	...	...	...	...	...	...
37817	126	17	great news expec ted infinex ventures infx pri...	1	1	1	1
37818	126	18	oil sector going crazy weekly gift kkpt thing ...	1	1	1	1
37819	126	19	httpvdtobjdocscaninfo suffering pain depressio...	1	1	1	1
37820	126	20	prosperous future increased money earning powe...	1	1	1	1
37821	126	21	moat coverall cytochemistry planeload salk	1	1	1	1

16522 rows × 7 columns

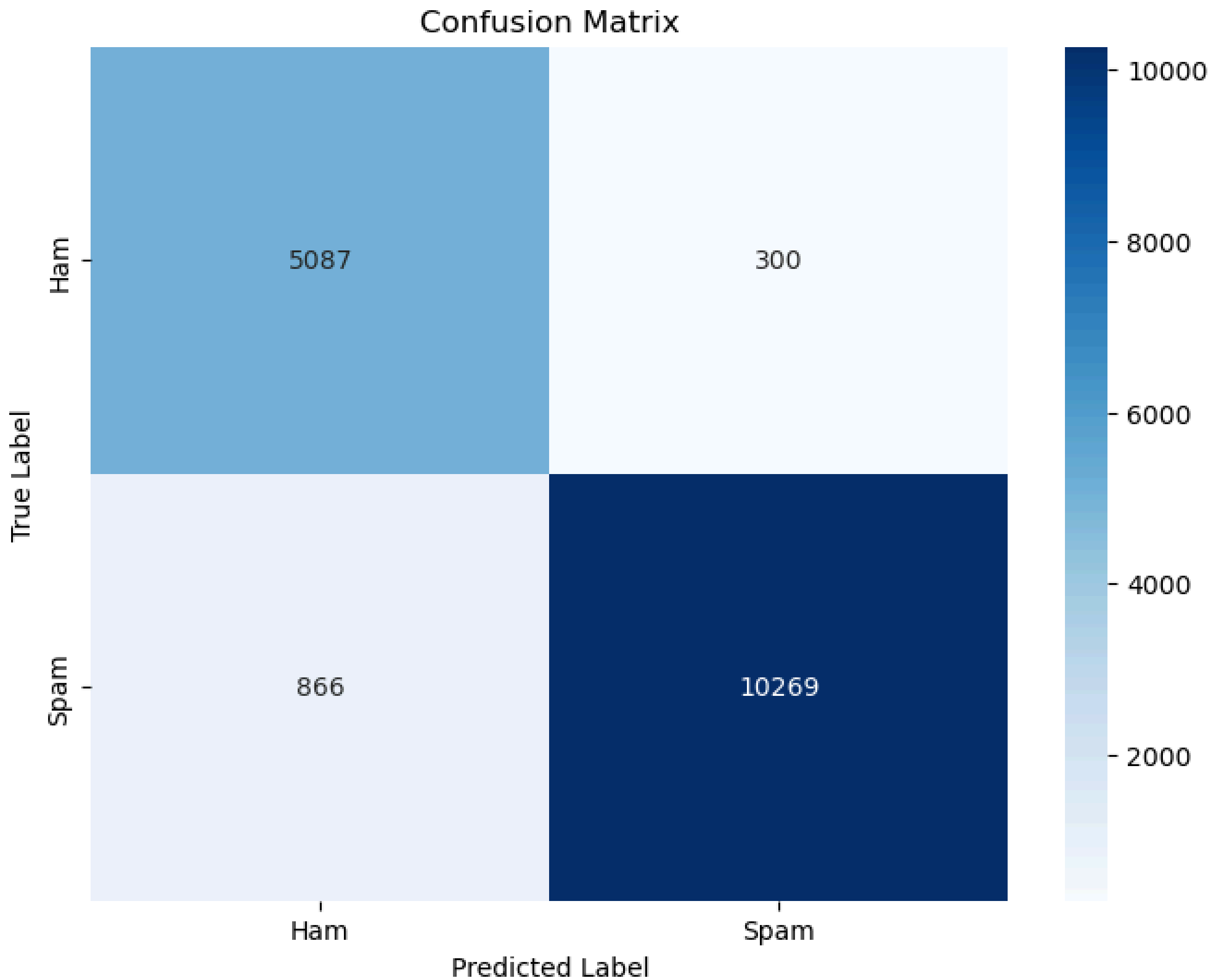
In [166…

```
# Evaluate the model for k > 1000
print("Metrics for k > 1000:")
metrics_k1000 = evaluate_model(test_df_k, 'category', 'predicted_k1000')

# Evaluate the model for k > 100
print("Metrics for k > 100:")
metrics_k100 = evaluate_model(test_df_k, 'category', 'predicted_k100')

# Evaluate the model for k > 50
print("Metrics for k > 50:")
metrics_k50 = evaluate_model(test_df_k, 'category', 'predicted_k50')
```

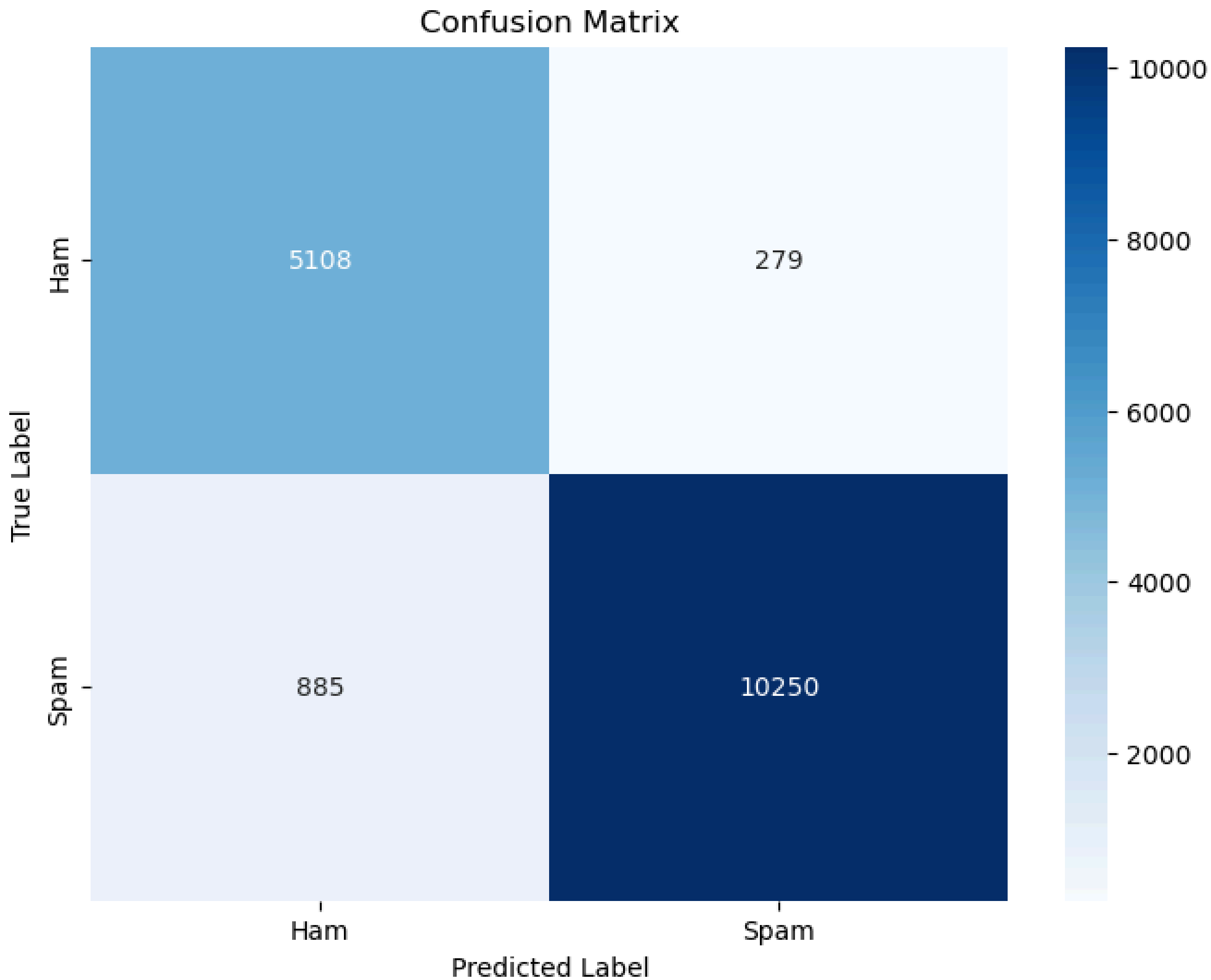
```
Metrics for k > 1000:
Accuracy: 0.9088488076504055 = 90.88%
Precision: 0.9234016357400404 = 92.34%
Recall: 0.9429726088908846 = 94.30%
F1-Score: 0.9330845107971207 = 93.31%
```




```
Metrics for k > 100:
Accuracy: 0.929427430093209 = 92.94%
Precision: 0.9716151007663922 = 97.16%
Recall: 0.9222272114952851 = 92.22%
F1-Score: 0.9462771839292295 = 94.63%
```



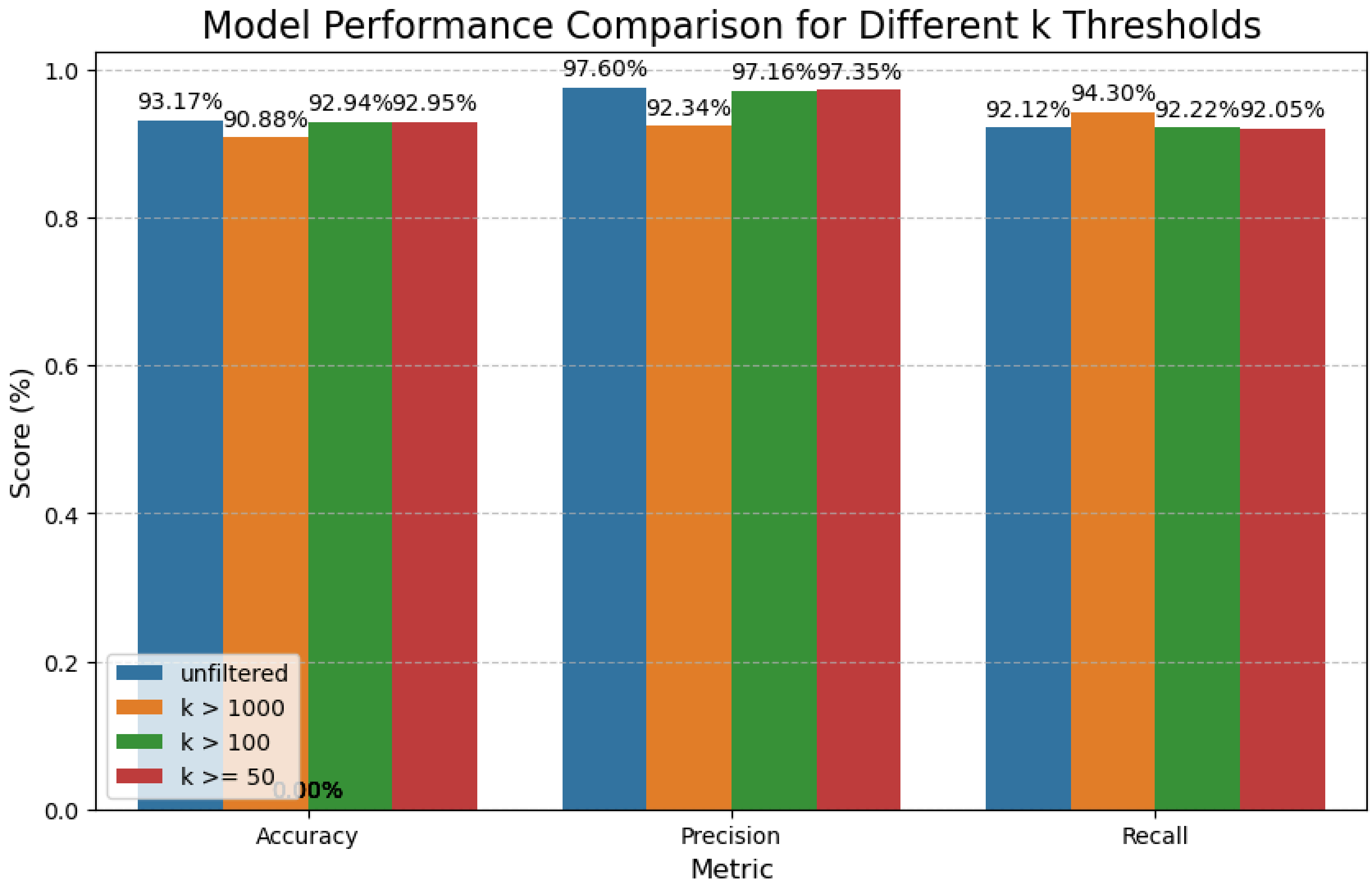

```
Metrics for k > 50:
Accuracy: 0.9295484808134609 = 92.95%
Precision: 0.9735017570519517 = 97.35%
Recall: 0.9205208801077683 = 92.05%
F1-Score: 0.9462703101920236 = 94.63%
```



In [203…

```
# Store the metrics in a dictionary for comparison
metrics_comparison = {
    'Metric': ['Accuracy', 'Precision', 'Recall'],
    'unfiltered': [metrics_no_stopwords['Accuracy'], metrics_no_stopwords['Precision'], metrics_no_stopwords['Recall
    'k > 1000': [metrics_k1000['Accuracy'], metrics_k1000['Precision'], metrics_k1000['Recall']],
    'k > 100': [metrics_k100['Accuracy'], metrics_k100['Precision'], metrics_k100['Recall']],
    'k >= 50': [metrics_k50['Accuracy'], metrics_k50['Precision'], metrics_k50['Recall']]
}
```

```python
# Convert the dictionary to a pandas df
metrics_df = pd.DataFrame(metrics_comparison)
metrics_melted = metrics_df.melt(id_vars='Metric', var_name='Condition', value_name='Value')

plt.figure(figsize=(10, 6))

# Create the bar plot
ax = sns.barplot(x='Metric', y='Value', hue='Condition', data=metrics_melted)

# Add titles and labels
plt.title('Model Performance Comparison for Different k Thresholds', fontsize=16)
plt.ylabel('Score (%)', fontsize=12)
plt.xlabel('Metric', fontsize=12)

# Add value labels on top of each bar
for p in ax.patches:
    ax.annotate(f"{p.get_height() * 100:.2f}%",
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha='center', va='baseline',
                fontsize=10, color='black', xytext=(0, 5),
                textcoords='offset points')

# Add grid lines
plt.grid(True, axis='y', linestyle='--', alpha=0.7)

# Move the legend to the best position
plt.legend(loc='lower left')

# Display the plot
plt.show()
```



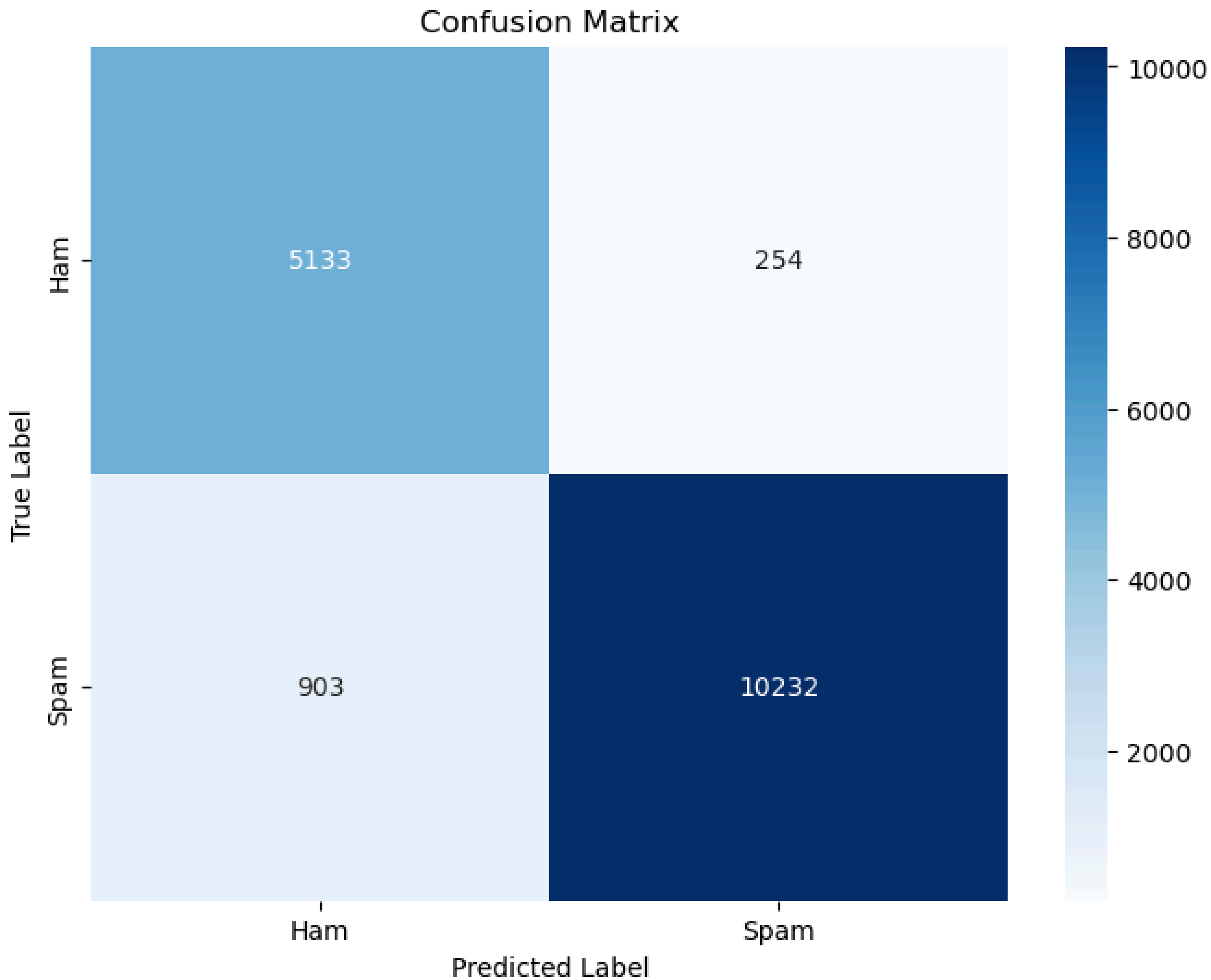
The plot shows that the **unfiltered model** performs a bit better in **Accuracy** and **Precision**, but the filtered models (with different word thresholds) still hold up well, especially in **Recall**. So, removing less frequent words doesn't hurt the model much, though it slightly lowers precision.

3. Discuss the results of the different parameters used for Lambda smoothing. Test it on 5 varying values of the λ (e.g. λ = 2.0, 1.0, 0.5, 0.1, 0.005), Evaluate performance metrics for each.

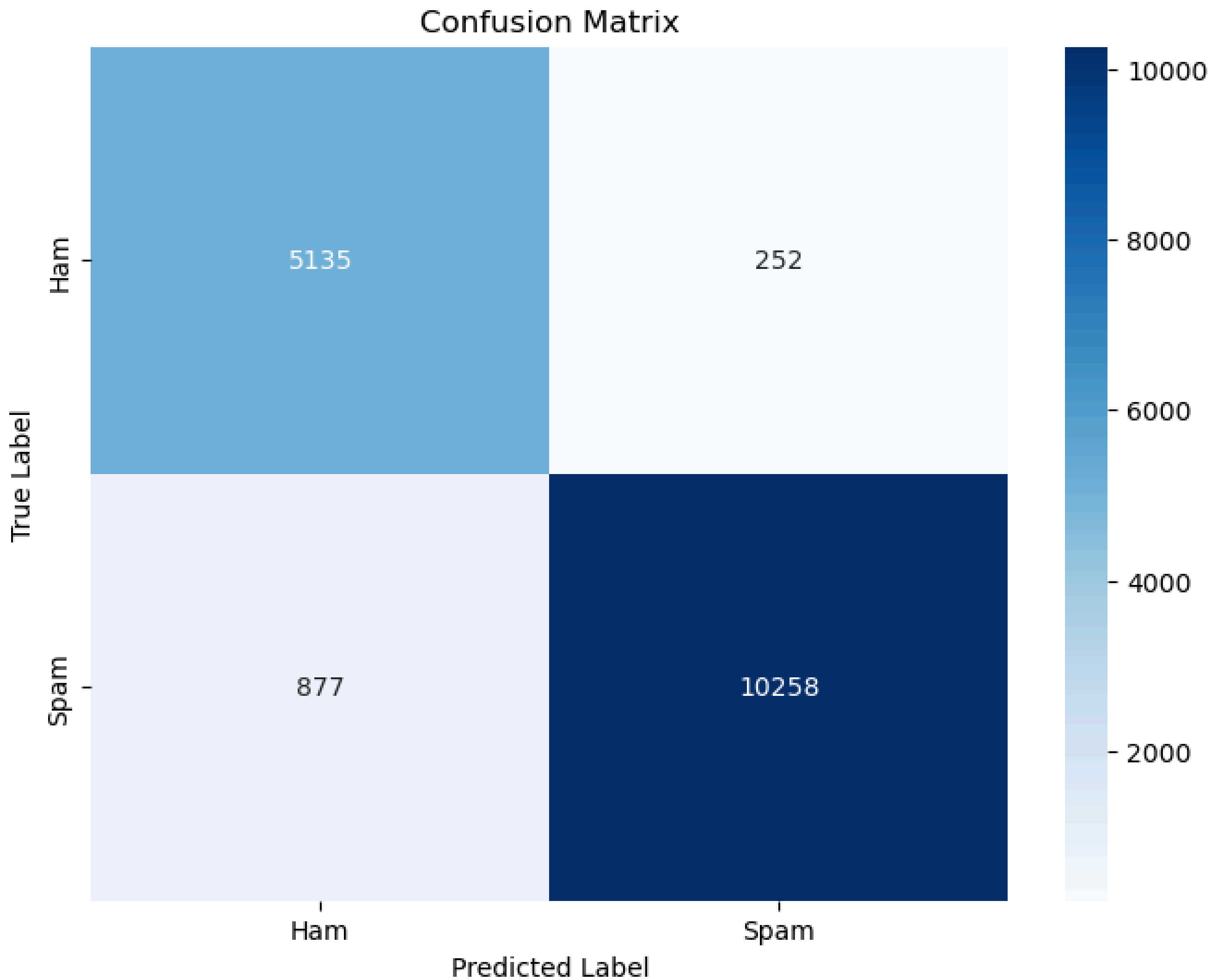
In [209…

```
# Define new lambda values to test
lambda_values_new = [2.0, 1.0, 0.5, 0.1, 0.005]

# Dictionary to store results
lambda_results_new = {}

for lambda_val in lambda_values_new:
    test_df_copy = test_df.copy()

    # Recalculate likelihoods
    likelihood_spam_lambda = calculate_likelihoods_with_laplace(spam_train_matrix, len(top_10000_words_list), lambda_
    likelihood_ham_lambda = calculate_likelihoods_with_laplace(ham_train_matrix, len(top_10000_words_list), lambda_va

    # Classify test emails
    test_df_copy[f'predicted_lambda_{lambda_val}'] = test_df_copy['email_message'].apply(
        lambda x: classify_email(x, likelihood_spam_lambda, likelihood_ham_lambda, p_spam, p_ham)
    )
    print(f"Results for Lambda = {lambda_val}:")
    # Evaluate performance metrics
    metrics_lambda = evaluate_model(test_df_copy, 'category', f'predicted_lambda_{lambda_val}')

    # Store the metrics
    lambda_results_new[lambda_val] = metrics_lambda
```

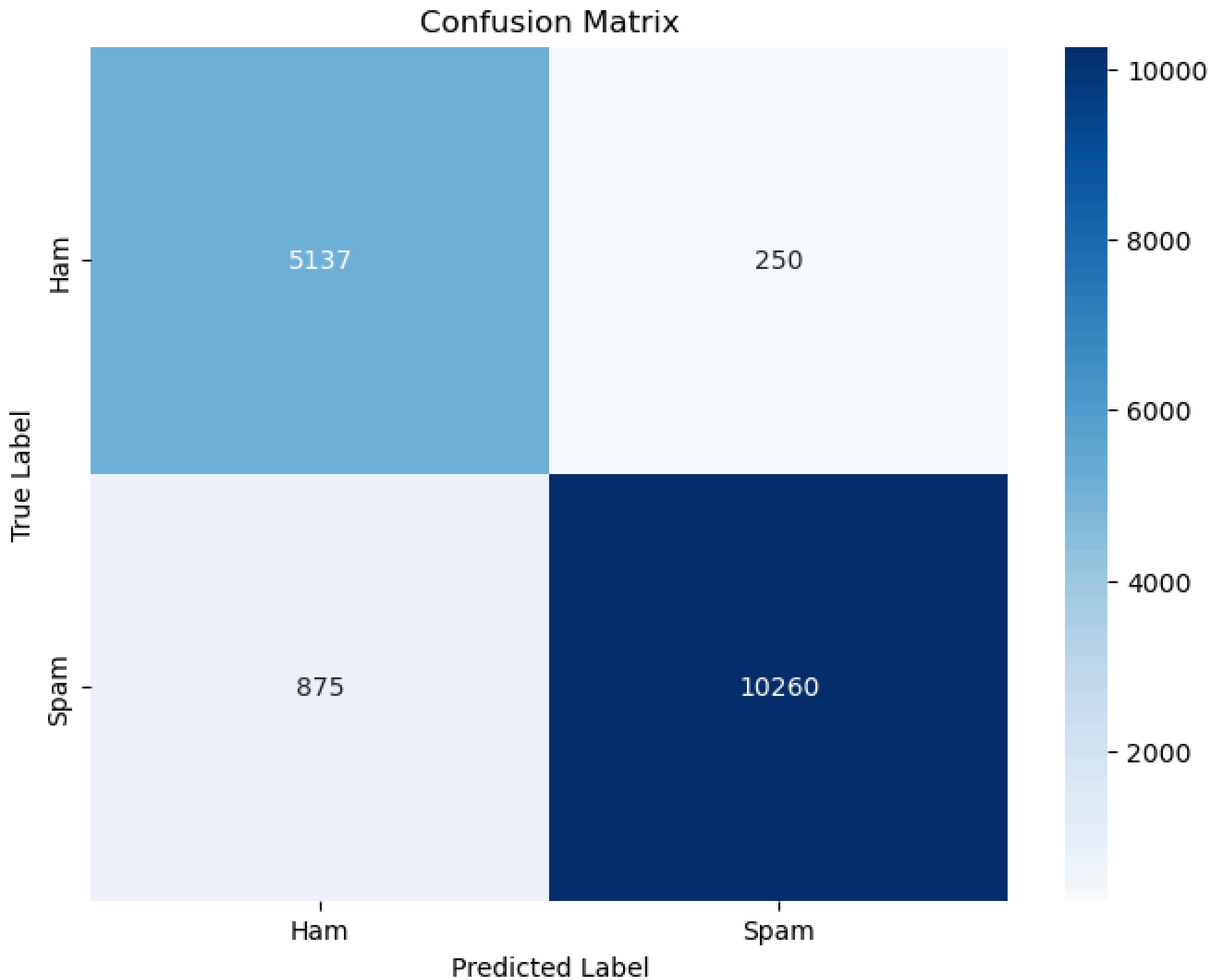
```
Results for Lambda = 2.0:
Accuracy: 0.9299721583343421 = 93.00%
Precision: 0.9757772267785619 = 97.58%
Recall: 0.918904355635384 = 91.89%
F1-Score: 0.946487211507331 = 94.65%
```



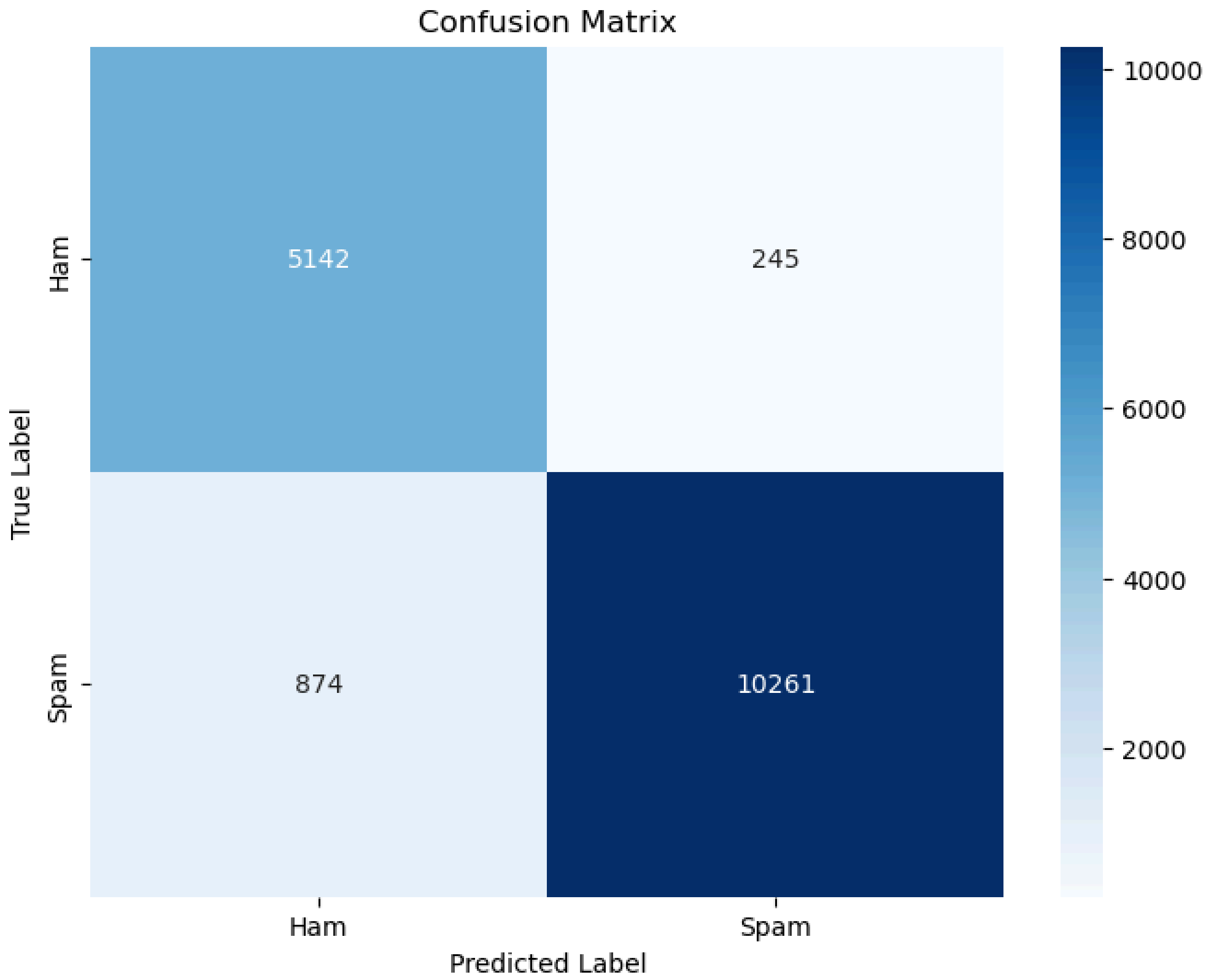

```
Results for Lambda = 1.0:
Accuracy: 0.9316668684178671 = 93.17%
Precision: 0.976022835394862 = 97.60%
Recall: 0.921239335428828 = 92.12%
F1-Score: 0.9478401478401478 = 94.78%
```



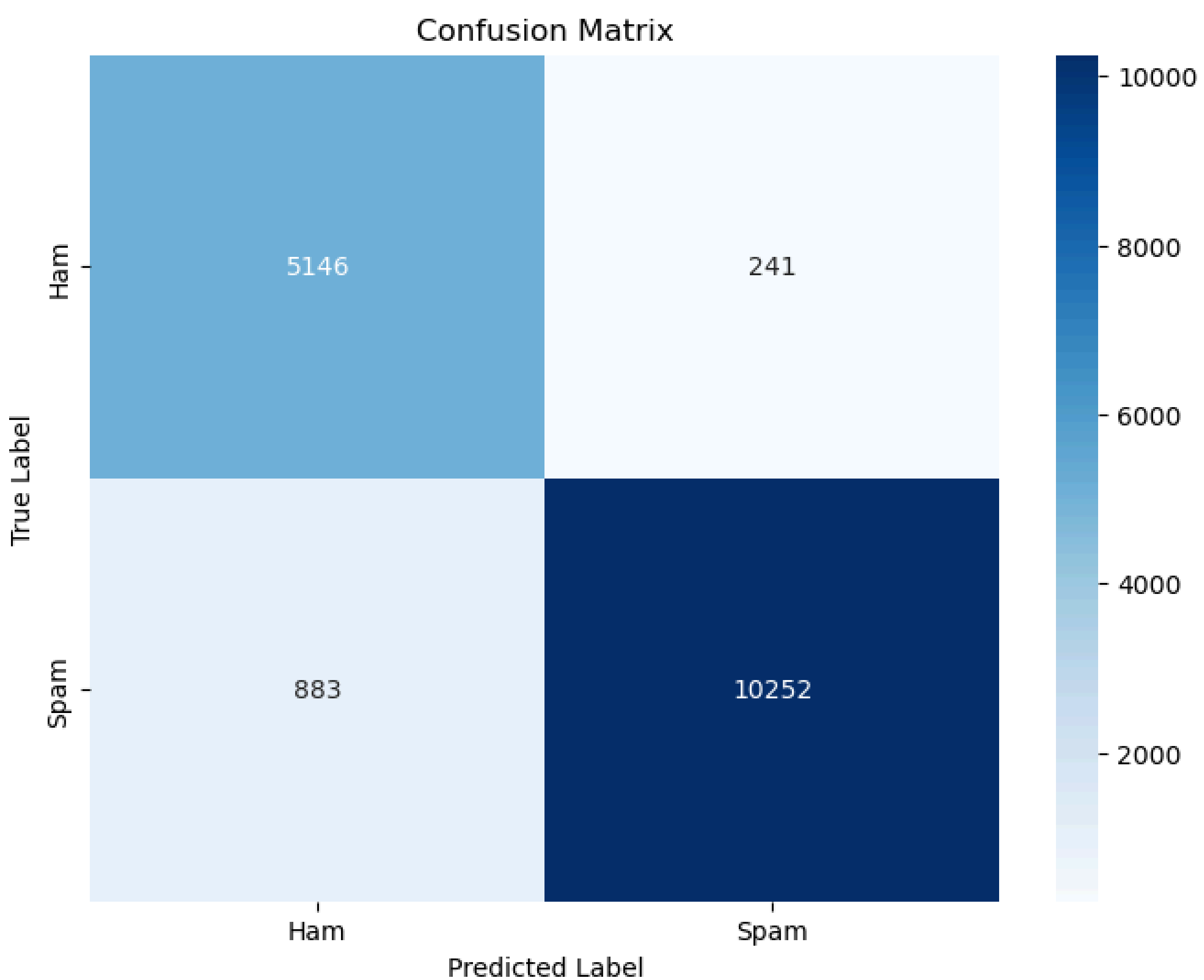

```
Results for Lambda = 0.5:
Accuracy: 0.9319089698583707 = 93.19%
Precision: 0.9762131303520457 = 97.62%
Recall: 0.921418949259093 = 92.14%
F1-Score: 0.9480249480249481 = 94.80%
```




```
Results for Lambda = 0.1:
Accuracy: 0.932272122019126 = 93.23%
Precision: 0.9766799923853037 = 97.67%
Recall: 0.9215087561742255 = 92.15%
F1-Score: 0.9482925927637355 = 94.83%
```



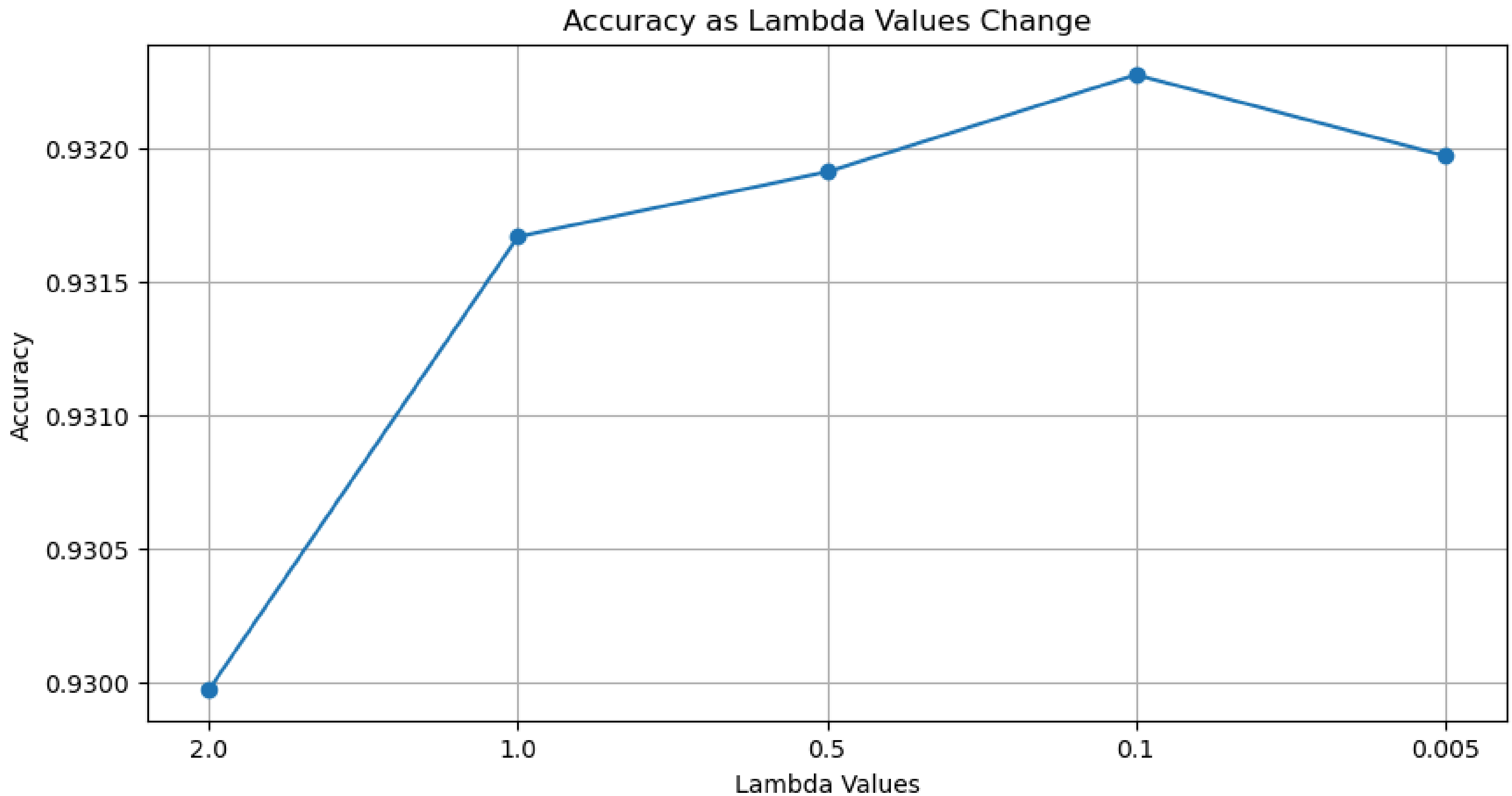

```
Results for Lambda = 0.005:
Accuracy: 0.9319694952184966 = 93.20%
Precision: 0.977032307252454 = 97.70%
Recall: 0.9207004939380332 = 92.07%
F1-Score: 0.9480303310523395 = 94.80%
```



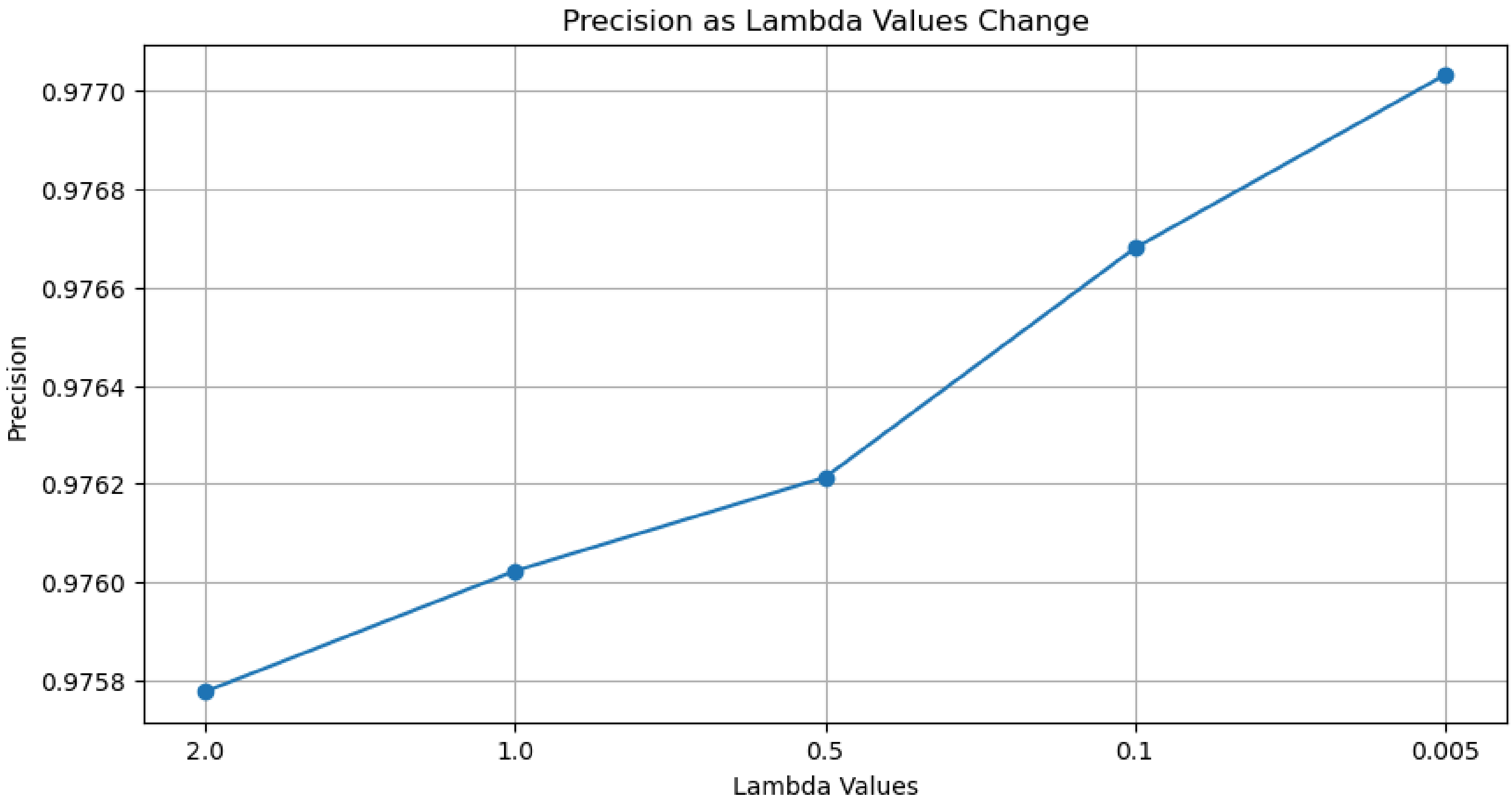
In [211…

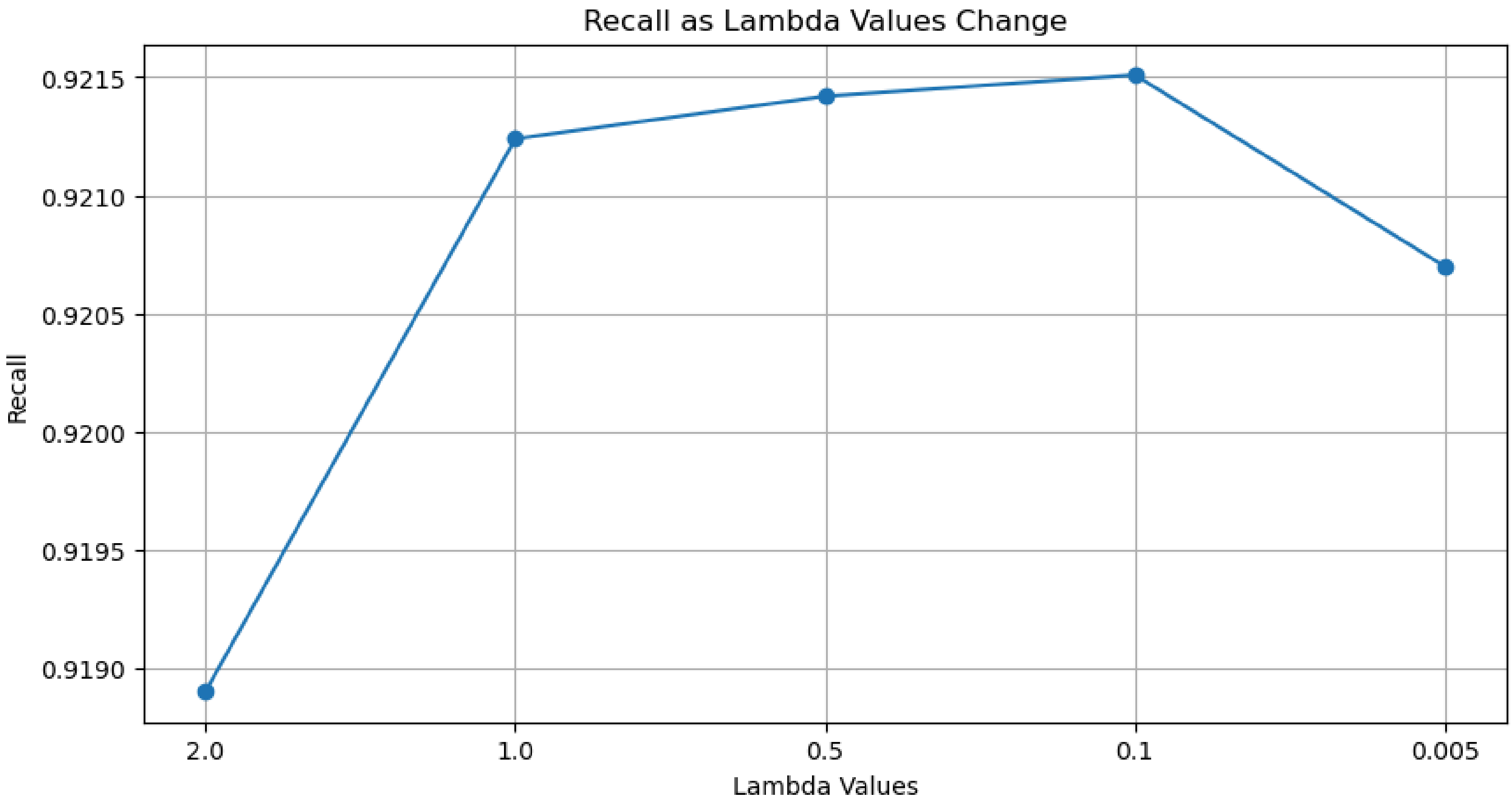
```
# Lambda values and corresponding metrics from lambda_results_new
lambda_values_labels = ['2.0', '1.0', '0.5', '0.1', '0.005']
lambda_values = [2.0, 1.0, 0.5, 0.1, 0.005]

# Extract Accuracy, Precision, and Recall from lambda_results_new for each lambda
accuracy_lambda_metrics = [lambda_results_new[lambda_val]['Accuracy'] for lambda_val in lambda_values]
precision_lambda_metrics = [lambda_results_new[lambda_val]['Precision'] for lambda_val in lambda_values]
recall_lambda_metrics = [lambda_results_new[lambda_val]['Recall'] for lambda_val in lambda_values]
```

```python
# Plot of Accuracy as lambda values change
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
ax.plot(lambda_values_labels, accuracy_lambda_metrics, marker='o')
ax.set_title('Accuracy as Lambda Values Change', fontsize=12)
ax.set_ylabel('Accuracy')
ax.set_xlabel('Lambda Values')
plt.grid(True)
plt.show()

# Plot of Precision as lambda values change
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
ax.plot(lambda_values_labels, precision_lambda_metrics, marker='o')
ax.set_title('Precision as Lambda Values Change', fontsize=12)
ax.set_ylabel('Precision')
ax.set_xlabel('Lambda Values')
plt.grid(True)
plt.show()

# Plot of Recall as lambda values change
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
ax.plot(lambda_values_labels, recall_lambda_metrics, marker='o')
ax.set_title('Recall as Lambda Values Change', fontsize=12)
ax.set_ylabel('Recall')
ax.set_xlabel('Lambda Values')
plt.grid(True)
plt.show()
```



Accuracy as Lambda Values Change
0.9320
0.9315
0.9310
0.9305
0.9300
Accuracy
2.0
1.0
0.5
0.1
0.005
Lambda Values


Precision as Lambda Values Change
0.9770
0.9768
0.9766
0.9764
0.9762
0.9760
0.9758
Precision
2.0
1.0
0.5
0.1
0.005
Lambda Values



From the line graph, we can see the that when the lambda is around 1.0 or 0.5, the model performs best in terms of accuracy and recall. But as lambda gets smaller (0,005), precision keeps improving, suggesting that smaller values of lambda help the model focus more on correctly identifying relevant spam emails. However, Recall reaches its highest point around lambda = 0.5 and then declines as λ decreases, indicating that while precision improves, the model might start missing more spam emails as lambda decreases. Overall, the intermediate values (around 0,5 to 1.0) provides the best trade-ofs by doing a good job in balancing between identifiying spam and not missing too many.

4. What are your recommendations to further improve the model?

Based on my experience with this problem set, here are some recommendations to improve the model:

General suggestion (because this one was really a headache): Since the notebook took a long time and consumed a lot of resources, we could speed up the preprocessing stages by using stemming instead of lemmatization, or we could reduce the size of the feature set by limiting the number of top words. Furthermore, faster training and testing could be achieved without compromising too much accuracy by streamlining the handling of the data, like employing an efficient algorithm or batch processing.

Other suggestions:

- Experiment with how stop words are handled, or maybe only eliminate some of them to see if it improves outcome.
- Including bigrams or trigrams (two or three-word phrases) could capture more context from the emails.
- Make sure the dataset is balanced between spam and ham emails so the model doesn't favor one over the other.
- Try filtering out rare words or setting different frequency limits for words to improve feature selection.
- Fine-tune parameters like lambda and the number of top words to see if we can boost the model’s accuracy.