"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

(2a) Show that equation (3) can be written as the *dual optimization problem*

$max_{\lambda_i \geq 0}\ min_{w,\alpha} ||w||^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i * w + \alpha) - 1)$

$max_{\lambda_i \geq 0}\ min_{w,\alpha}\ w^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i * w + \alpha) - 1)$

$= max_{\lambda_i \geq 0} [\nabla (\frac{df/dw}{df/d\alpha})]$

$[\nabla f (\frac{df/dw}{df/d\alpha})] \ =$

$= \frac{df}{dw} [w^2 - \sum_{i-1}^{n} [\lambda_i (y_i (X_i * w + \alpha) - 1)]$

$= \frac{df}{dw} w^2 - \sum_{i-1}^{n} \frac{df}{dw} [\lambda_i (y_i (X_i * w + \alpha) - 1)]$

$2w - \sum_{i-1}^{n} \frac{df}{dw} [\lambda_i (y_i (X_i * w + \alpha) - 1)]$

$2w - \sum_{i-1}^{n} [\frac{df}{dw} \lambda_i y_i X_i * w + \frac{df}{dw} \alpha \lambda_i y_i - \frac{df}{dw} \lambda_i \ ]$

$= 2w - \sum_{i-1}^{n} [\lambda_i y_i X_i] = 0$

$w = \frac{\sum_{i-1}^{n} [\lambda_i y_i X_i]}{2}$

And,

$\frac{df}{d\alpha} [w^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i * w + \alpha) - 1)]$

$\frac{df}{d\alpha} w^2 - \sum_{i=1}^{n} \frac{df}{d\alpha} [(\lambda_i (y_i X_i * w + \alpha y_i \lambda_i) - \lambda_i)] = 0$

$= - \sum_{i=1}^{n} [\frac{df}{d\alpha} \lambda_i y_i X_i w + \frac{df}{d\alpha} \alpha y_i \lambda i - \frac{df}{d\alpha} \lambda_i]$

$= - \sum_{i=1}^{n} [0 + y_i \lambda_i - 0] = 0$

$= - \sum_{i=1}^{n} [y_i \lambda_i] = 0$

$= \sum_{i=1}^{n}[y_i\lambda_i] = 0$, where $\sum_{i=1}^{n}[y_i\lambda_i] = 0$

$\alpha = 0$

Plugging in w and α into (3):

$max_{\lambda_i \geq 0} || \frac{\sum_i^n \lambda_i y_i X_i}{2} ||^2 - \sum_{i=1}^{n} \lambda_i((y_i X_i * w + (0)\lambda_i) - 1)$

$= \frac{(\sum_{i=1}^{n} \lambda_i y_i X_i)(\sum_{i=1}^{n} \lambda_i y_i X_i)}{4} - \frac{(\sum_{i=1}^{n} \lambda_i y_i X_i)(\sum_{i=1}^{n} \lambda_i y_i X_i)}{2} - \sum_{i=1}^{n} \lambda_i$

$= \frac{(\sum_{i=1}^{n} \lambda_i y_i X_i)(\sum_{i=1}^{n} \lambda_i y_i X_i)}{4} - \frac{\sum_{i=1}^{n}\sum_{i=1}^{n} \lambda_i y_i \lambda_j y_j * X_i * X_j}{2} - \sum_{i=1}^{n} \lambda_i$

$= \frac{(\sum_{i=1}^{n} \lambda_i y_i X_i)(\sum_{i=1}^{n} \lambda_i y_i X_i)}{4} - 0 - \sum_{i=1}^{n} \lambda_i$

Therefore,

$max_{\lambda_i \geq 0} - \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j X_i * X_j$ subject to $\sum_{i=1}^{n} \lambda_i y_i = 0$

(2b)

$$r(x) = \begin{cases} +1, & \text{if } w * X + \alpha \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

Substituting w, and α,

We get:

$$r(x) = \begin{cases} +1, & \text{if } \alpha^* + \frac{\sum_{i-1}^{n} [\lambda_i y_i X_i]}{2} * X_i \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

(2c)

From my understanding, we have a decision rule $r(x)$ that requires we solve the optimization problem (3) in order to get α and w. When solving the optimization problem we apply lagrange multipliers. This is where the condition comes into play. The original condition is $min||w||^2$ with the condition that $(y_i(X_i * w^* + \alpha) - 1) \geq 1$ which means that for any point X_i it must be ≥ 1 in order for the point to be classified correctly with a margin of ≥ 1. By introducing lagrange multiplier λ_i^* we have, $\lambda_i(y_i(X_i * w + \alpha) - 1)$ which implies that the value lies on the margin, thus identifying the support vectors.

(2d)

The support vectors are the only training points needed to evaluate the decision rule because these training points define the position and orientation of the margins, thus, since all the other points that are not support vectors. they are not necessary because they don't contribute to finding the margins because only support vectors do so.

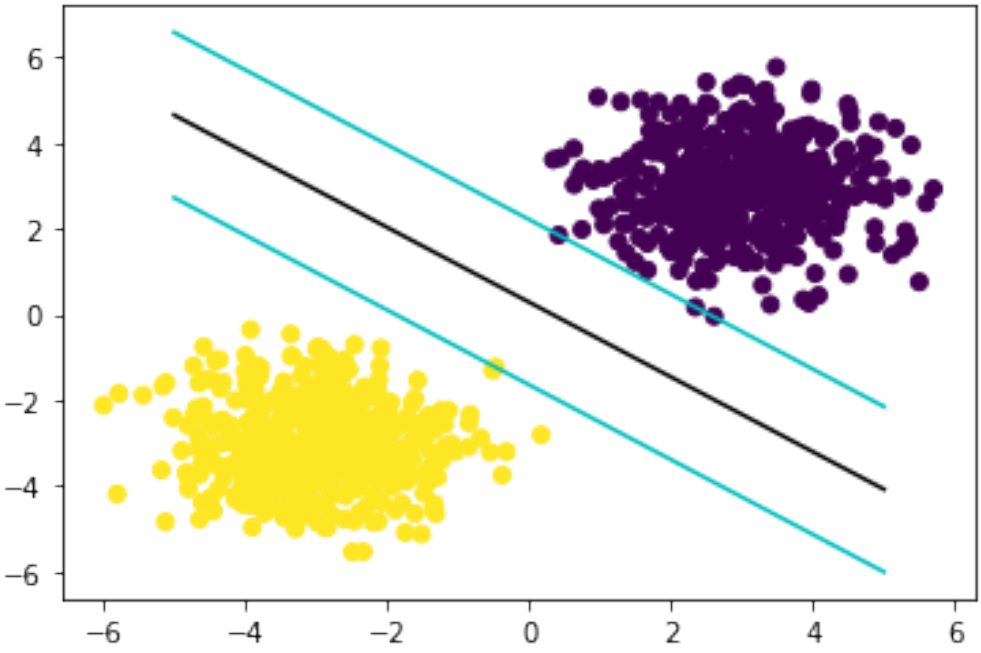


Figure 1: Toy Data Scatterplot

(2e)

```
# Scatterplot of toy data
plt.scatter(X_train[:,0], X_train[:,1], c=Y_train);

# Plot decision boundary
w = np.vstack(np.array([-0.4528, -0.5190]))
b = 0.1471
x = np.linspace(-5, 5, 100)
y = -(w[0] * x + b) / w[1]

# Define Margins
margin_1x = np.linspace(-5, 5, 100)
margin_1y = -((w[0] * x + b) - 1) / w[1]
margin_2x = np.linspace(-5, 5, 100)
margin_2y = -((w[0] * x + b) + 1) / w[1]

# Plot margins
plt.plot(margin_1x, margin_1y, 'c');
plt.plot(margin_2x, margin_2y, 'c');
plt.plot(x, y, 'k');
```

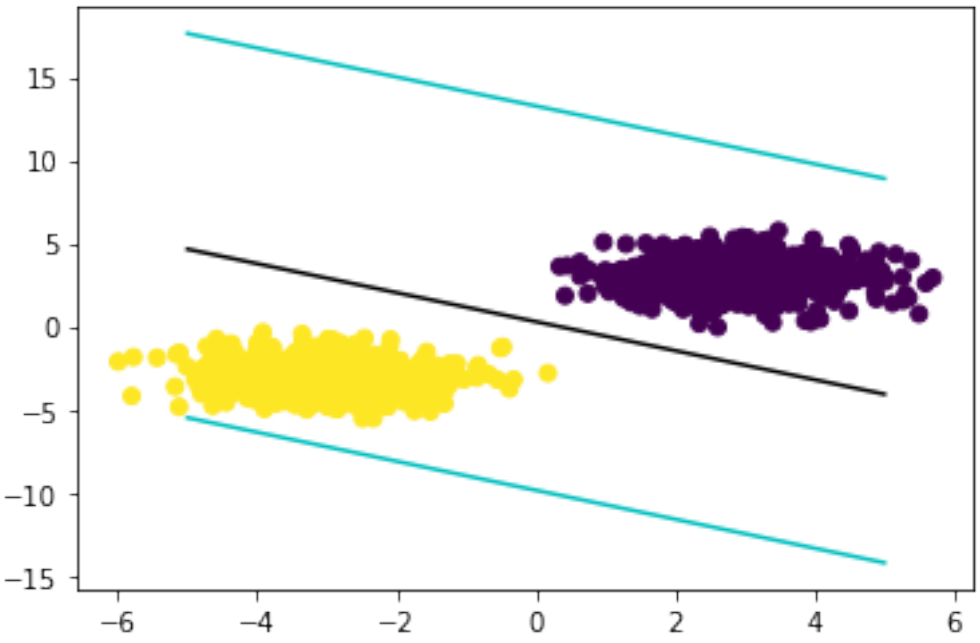


Figure 2: Enter Caption

(2f)

Contradiction: We assume there are no support vectors for class $+1$. This means all points lie within the margin. Referring to the toy dataset this looks like all points inside the margin. When applying $w' = \frac{w}{1+\frac{\epsilon}{2}}$ where $\epsilon = 10$ and $\alpha = 0.1471$ we can observe that this violates the maximization margin principle because although there's a hyperplane, it's not the one with the widest possible gap between the two classes. However, the hyperplane still seperates the two classes.

(2f: Plot used for Contradiction)

```
# Scatterplot of toy data with w' and epsilon
plt.scatter(X_train[:,0], X_train[:,1], c=Y_train);

# Plot decision boundary
w = np.vstack(np.array([-0.4528, -0.5190]))
epsilon = 10
w_prime = w / (1 +(epsilon / 2) )
b = 0.1471
x = np.linspace(-5, 5, 100)
y = -(w[0] * x + b) / w[1]

# Define Margins
margin_1x = np.linspace(-5, 5, 100)
margin_1y = -((w_prime[0] * x + b) - 1) / w_prime[1]
margin_2x = np.linspace(-5, 5, 100)
margin_2y = -((w_prime[0] * x + b) + 1) / w_prime[1]

# Plot margins
plt.plot(margin_1x, margin_1y, 'c');
plt.plot(margin_2x, margin_2y, 'c');
plt.plot(x, y, 'k');
```

(3a)

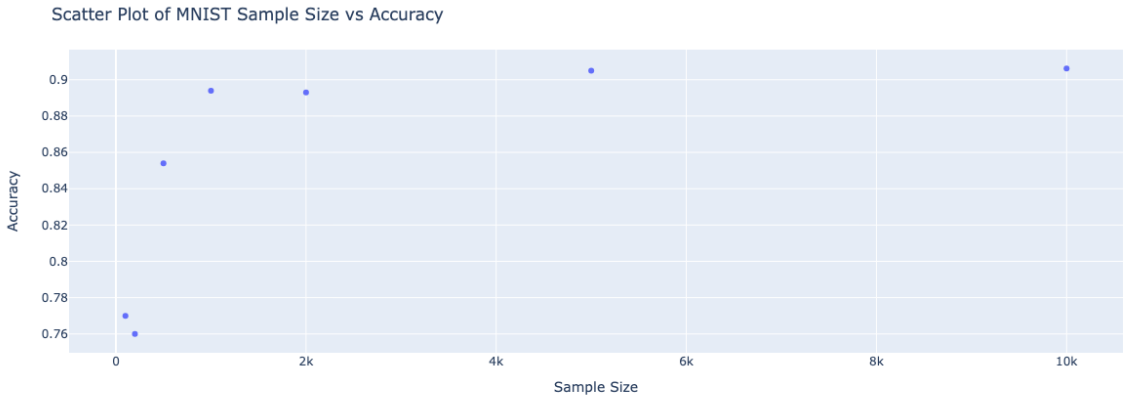
```
# Load MNIST data
mnist_data = np.load(f"../data/mnist-data.npz")

# Extract training data containing features
MNIST_features = mnist_data["training_data"]
MNIST_labels = mnist_data["training_labels"]

# Generate permutation indices from MNIST_training_data
indices = np.random.permutation(len(MNIST_features))

# Apply permutation to the training data and labels
MNIST_features_shuffled = MNIST_features[indices]
MNIST_labels_shuffled = MNIST_labels[indices]

# Define the training data
X_train_MNIST = MNIST_features_shuffled[10_000:]
Y_train_MNIST = MNIST_labels_shuffled[10_000:]

# Set aside 10,000 images from the training set as a validation set
X_test_MNIST = MNIST_features_shuffled[:10_000]
Y_test_MNIST = MNIST_labels_shuffled[:10_000]

# Load spam data
spam_data = np.load(f"../data/spam-data.npz")

# Extract the features and labels
spam_features = spam_data['training_data']
spam_labels = spam_data['training_labels']

# Generate permutation indices
indices = np.random.permutation(len(spam_features))

# Shuffle the the features and labels
spam_features_shuffled = spam_features[indices]
spam_labels_shuffled = spam_labels[indices]

# Define the index for 20th percentile
twenty_percentile_index_features = int(len(spam_features_shuffled) * 0.20)
twenty_percentile_index_labels = int(len(spam_labels_shuffled) * 0.20)

# Define the training data
X_train_spam = spam_features_shuffled[twenty_percentile_index_features:]
Y_train_spam = spam_labels_shuffled[twenty_percentile_index_labels:]

# Define the validation set as 20% of the data
X_test_spam = spam_features_shuffled[:twenty_percentile_index_features]
Y_test_spam = spam_labels_shuffled[:twenty_percentile_index_labels]
```

(3b)

```
def evaluation_metric(_y, _y_hat, sample_size):
    """Function returns classification accuracy"""
    correct_predictions = sum(_y_hat == _y)
    return (correct_predictions / sample_size)
```

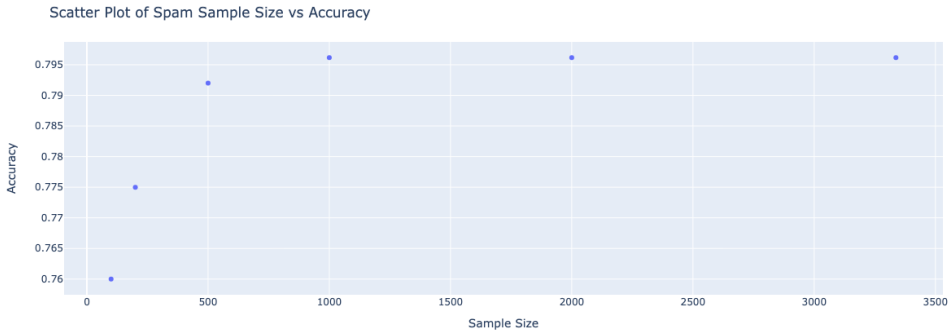
4a

```
MNIST_training_sizes = [100, 200, 500, 1_000, 2_000, 5_000, 10_000]
MNIST_accuracies = []

# Define models
MNIST_model = SVC(kernel='linear', random_state=42)

for size in MNIST_training_sizes:

    # Transform Y_train_MNIST
    Y_train_MNIST_flat = Y_train_MNIST[:size].reshape(
                                                    Y_train_MNIST[:size].shape[0],
                                                    -1)

    # Transform Y_test_MNIST
    Y_test_MNIST_flat = Y_test_MNIST[:size].reshape(Y_test_MNIST[:size].shape[0], -1)

    # Flatten Y_train_MNIST_flat
    Y_train_MNIST_flat = Y_train_MNIST_flat[:,0]

    # Flatten Y_test_MNIST_flat
    Y_test_MNIST_flat = Y_test_MNIST_flat[:,0]

    MNIST_model.fit(
                    X_train_MNIST[:size].reshape(X_train_MNIST[:size].shape[0],-1),
                    Y_train_MNIST_flat)

    y_hat = np.array(MNIST_model.predict(
                            X_test_MNIST[:size].reshape(X_test_MNIST[:size].shape[0],
                            -1)))

    MNIST_accuracies.append(evaluation_metric(Y_test_MNIST_flat,y_hat, len(y_hat)))

# MNIST Accuracy Plot
import plotly.express as px

MNIST_accuracies_df = pd.DataFrame({'Sample Size' : MNIST_training_sizes, 'Accuracy' : MNIST
fig = px.scatter(MNIST_accuracies_df, x = "Sample Size",y = "Accuracy", hover_data=['Accuracy
fig.update_layout(title='Scatter Plot of MNIST Sample Size vs Accuracy')
fig.show()
```


Scatter Plot of MNIST Sample Size vs Accuracy
Accuracy
0.9
0.88
0.86
0.84
0.82
0.8
0.78
0.76
0
2k
4k
6k
8k
10k
Sample Size

4b

```
spam_sample_sizes = [100, 200, 500, 1_000, 2_000, len(X_train_spam)]
spam_accuracies = []

# Define model
spam_model = SVC(kernel='linear', random_state=42)

for i in spam_sample_sizes:
    spam_model.fit(X_train_spam[:i], Y_train_spam[:i])
    spam_y_hat = spam_model.predict(X_test_spam[:i])
    accuracy = evaluation_metric(Y_test_spam[:i], spam_y_hat, len(spam_y_hat))
    spam_accuracies.append(accuracy)

spam_accuracies_df = pd.DataFrame({'Sample Size' : spam_sample_sizes, 'Accuracy' : spam_accur

fig = px.scatter(spam_accuracies_df, x = "Sample Size",y = "Accuracy", hover_data=['Accuracy']
fig.update_layout(title='Scatter Plot of Spam Sample Size vs Accuracy')
fig.show()
```


Scatter Plot of Spam Sample Size vs Accuracy
Accuracy
0.795
0.79
0.785
0.78
0.775
0.77
0.765
0.76
0
500
1000
1500
2000
2500
3000
3500
Sample Size

5: Hyperparameter Tuning

```
# Increase value of C to reduce training error, however this can lead to overfitting
MNIST_c_values = [0.0001, 0.001, 0.1, 10, 100, 1_000, 10_000, 100_000]
MNIST_accuracies_with_hyperparameter_tuning = []
sample_size = 20_000
MNIST_models = []

for c_value in MNIST_c_values:
    Y_train_MNIST_flat = Y_train_MNIST[:sample_size].reshape(
        Y_train_MNIST[:sample_size].shape[0], -1
        )
    # Transform Y_test_MNIST
    Y_test_MNIST_flat = Y_test_MNIST[:sample_size].reshape(
        Y_test_MNIST[:sample_size].shape[0], -1
        )
    # Flatten Y_train_MNIST_flat
    Y_train_MNIST_flat = Y_train_MNIST_flat[:,0]
    # Flatten Y_test_MNIST_flat
    Y_test_MNIST_flat = Y_test_MNIST_flat[:,0]

    # Define model
    MNIST_model = SVC(C = c_value, kernel='linear', random_state=42)
    # Fit the model
    MNIST_model.fit(X_train_MNIST[:sample_size].reshape(
        X_train_MNIST[:sample_size].shape[0], -1),
        Y_train_MNIST_flat)
    # Make predictions
    y_hat_ = np.array(MNIST_model.predict(
        X_test_MNIST[:sample_size].reshape(
            X_test_MNIST[:sample_size].shape[0], -1)
            )
            )
    # Determine accuracy
    accuracy_c_with_cvals = evaluation_metric(Y_test_MNIST_flat, y_hat_, len(y_hat_))
    # Append current models accuracy to list of MNIST accuracies
    MNIST_accuracies_with_hyperparameter_tuning.append(accuracy_c_with_cvals)
    # Append the model to list of models
    MNIST_models.append(MNIST_model)

top_MNIST_model = MNIST_models
[
    MNIST_accuracies_with_hyperparameter_tuning.index
    (
        max(MNIST_accuracies_with_hyperparameter_tuning)
    )
]
top_MNIST_accuracy = max(MNIST_accuracies_with_hyperparameter_tuning)
top_MNIST_C_value  = MNIST_c_values
[
```

```
    MNIST_accuracies_with_hyperparameter_tuning.index
    (
        max(MNIST_accuracies_with_hyperparameter_tuning)
    )
]
# Deliverable:
print(f"C values used: {MNIST_c_values}")
print(f"Corresponding Accuracies: {MNIST_accuracies_with_hyperparameter_tuning}")
print(f"Top C Value: {top_MNIST_C_value}")
```

```
C values used: [0.0001, 0.001, 0.1, 10, 100, 1000, 10000, 100000]
Corresponding Accuracies: [0.9124, 0.9109, 0.9109, 0.9109, 0.9109, 0.9109, 0.9109, 0.9109]
Top C Value: [0.0001]
```

6: K-Fold Cross-Validation

```
# c values to test
spam_c_values = [0.001, 0.01, 0.1, 100, 1000, 10_000, 100_000, 1_000_000]

# Determine size of dataset
training_size = len(X_train_spam)

# Variables to hold models
spam_models = []
spam_model_mean_accuracy = []

# number of folds
K = 5

# shuffle spam dataset training features (X_train)
shuffled_indexes = np.random.permutation(len(X_train_spam))
X_train_spam_shuffled = X_train_spam[shuffled_indexes]
# shuffle spam dataset training labels (Y_train)
Y_train_spam_shuffled = Y_train_spam[shuffled_indexes]
# determine splits into k - 1 disjoint sets
folds = np.array_split(np.arange(training_size), K)

for c_val in spam_c_values:
    k_fold_accuracy= []
    for i in range(K):
        # Define model
        model = SVC(C=c_val, kernel='linear', random_state=42)
        test_indices = folds[i]
        train_indices = np.concatenate([fold for j, fold in enumerate(folds) if j != i])

        X_train = X_train_spam_shuffled[train_indices]
        Y_train = Y_train_spam_shuffled[train_indices]
        X_test = X_train_spam_shuffled[test_indices]
        Y_test = Y_train_spam_shuffled[test_indices]

        # Fit the model on the training sets
        model.fit(X_train, Y_train)

        # Predict X_test
        y_hat = model.predict(X_test)

        # calculate the accuracy of the model with the current fold and C value
        accuracy = evaluation_metric(Y_test, y_hat, len(y_hat))

        # append the accuracy to a list of accuracies
        k_fold_accuracy.append(accuracy)

    spam_models.append(model)
    # after all folds have been tested, get the mean accuracy for each k
```

```
    spam_model_mean_accuracy.append(np.mean(k_fold_accuracy))

# Deliverable
top_spam_model = spam_models
[
    spam_model_mean_accuracy.index
    (
        max(spam_model_mean_accuracy)
    )
]
top_spam_accuracy = max(spam_model_mean_accuracy)
top_spam_C_value = spam_c_values
[
    spam_model_mean_accuracy.index(max(spam_model_mean_accuracy))
]

# Deliverable
print(f"C values used: {spam_c_values}")
print(f"Corresponding Accuracies: {spam_model_mean_accuracy}")
print(f"Top C Value: {top_spam_C_value}")
```

7: Kaggle

```
# Load MNIST test data
MNIST_test_data = mnist_data['test_data']
# Make predictions on test data
kaggle_y_hat = MNIST_model.predict(
    MNIST_test_data.reshape(X_test_MNIST[:len(MNIST_test_data)].shape[0], -1)
    )

# Export MNIST predictions to CSV
results_to_csv(kaggle_y_hat)

# Load spam test data
spam_test_data = spam_data['test_data']
# Make predictions using best spam model
spam_y_hat = top_spam_model.predict(spam_test_data)
# Export spam predictions to CSV
results_to_csv(spam_y_hat)
```

MNIST Kaggle Score: .906

Spam Kaggle Score: .803

What worked for improving my accuracy for the MNIST dataset was increasing the sample size (as expected) and increasing the size of the C parameter. However, due to computatioin limitations I had to experiment on the optimal sample size to train the model. Some sample sizes were too large, causing the model to take a very long time to train and make predictions. What worked to increase the accuracy on the spam dataset was modifying the C parameter as well as increasing the training size. K-fold's also improved accuracy. What didn't work was using small values for the C parameter.