Hiva Mohammadzadeh
3036919598

"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."

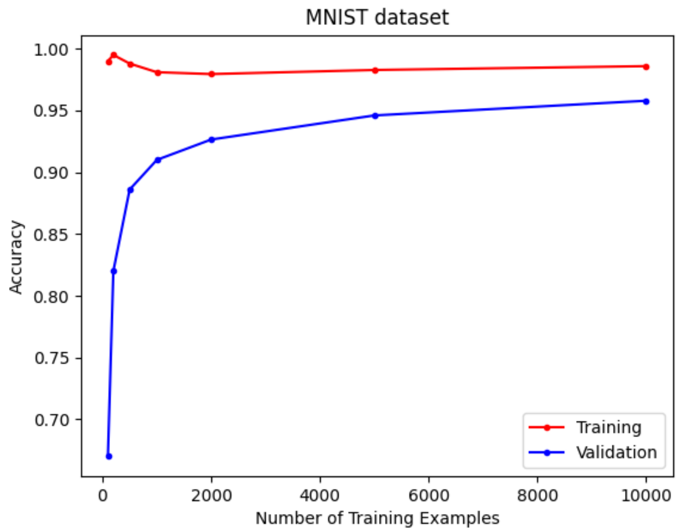
Question 2: Data Partitioning

```
MNIST Shuffling and Splitting:
Training data: (50000, 784)
Training labels: (50000, 1)
Validation data: (10000, 784)
Validation labels: (10000, 1)

SPAM Shuffling and Splitting:
Training data: (3337, 32)
Training labels: (3337, 1)
Validation data: (835, 32)
Validation labels: (835, 1)

CIFAR-10 Shuffling and Splitting:
Training data: (45000, 3072)
Training labels: (45000, 1)
Validation data: (5000, 3072)
Validation labels: (5000, 1)
```

Question 3: Support Vector Machines: Coding

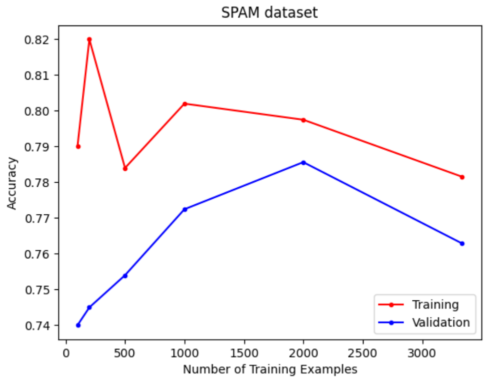
a) MNIST

```
MNIST Dataset accuracies:
Training with 100 examples
Training accuracy: 0.99
Validation Accuracy: 0.67

Training with 200 examples
Training accuracy: 0.995
Validation Accuracy: 0.82

Training with 500 examples
Training accuracy: 0.988
Validation Accuracy: 0.886

Training with 1000 examples
Training accuracy: 0.981
Validation Accuracy: 0.91

Training with 2000 examples
Training accuracy: 0.9795
Validation Accuracy: 0.9265

Training with 5000 examples
Training accuracy: 0.9828
Validation Accuracy: 0.946

Training with 10000 examples
Training accuracy: 0.9859
Validation Accuracy: 0.9579
```



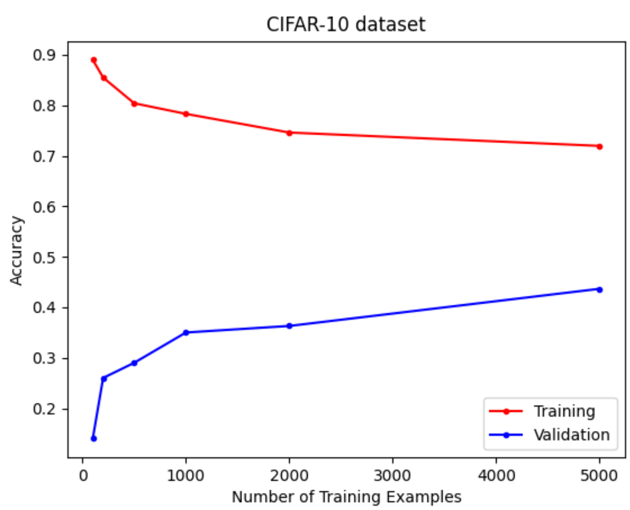
b) SPAM:

```
SPAM Dataset accuracies:
Training with 100 examples
Training accuracy: 0.79
Validation Accuracy: 0.74

Training with 200 examples
Training accuracy: 0.82
Validation Accuracy: 0.745

Training with 500 examples
Training accuracy: 0.784
Validation Accuracy: 0.754

Training with 1000 examples
Training accuracy: 0.802
Validation Accuracy: 0.7724550898203593

Training with 2000 examples
Training accuracy: 0.7975
Validation Accuracy: 0.7856287425149701

Training with 3337 examples
Training accuracy: 0.7815403056637699
Validation Accuracy: 0.7628742514970059
```



c) CIFAR-10

```
CIFAR-10 Dataset accuracies:
Training with 100 examples
Training accuracy: 0.89
Validation Accuracy: 0.14

Training with 200 examples
Training accuracy: 0.855
Validation Accuracy: 0.26

Training with 500 examples
Training accuracy: 0.804
Validation Accuracy: 0.29

Training with 1000 examples
Training accuracy: 0.783
Validation Accuracy: 0.35

Training with 2000 examples
Training accuracy: 0.746
Validation Accuracy: 0.363

Training with 5000 examples
Training accuracy: 0.7196
Validation Accuracy: 0.4366
```



Question 4: Hyperparameter Tuning:
I trained this model using 10000 training examples.
The C values that I tried were:

[0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000, 10000],
[0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10], and
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

But eventually went with this list:

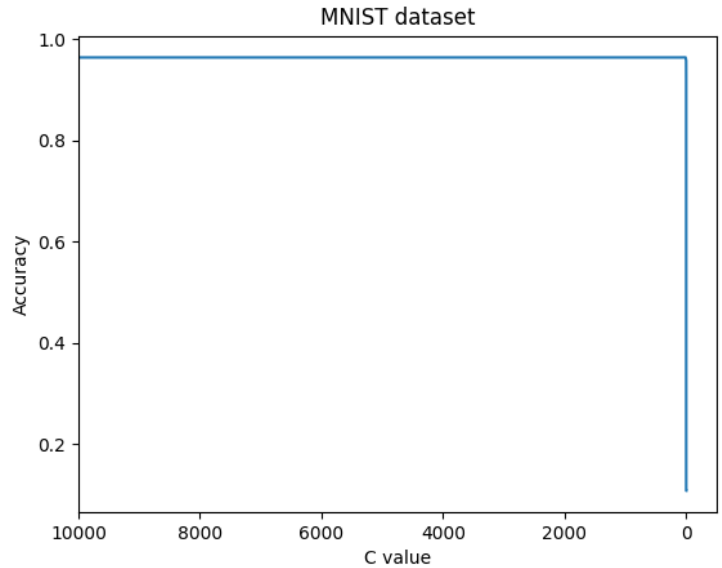
[0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000, 10000]

I generated these series with this website:
https://onlinenumbertools.com/generate-geometric-sequence

My code will print out all the validation accuracies for all the c values as they are calculated:

```
MNIST Dataset C value Calculation:

c_value: 1e-06
validation_accuracy: 0.1077

c_value: 1e-05
validation_accuracy: 0.1077

c_value: 0.0001
validation_accuracy: 0.1077

c_value: 0.001
validation_accuracy: 0.1077

c_value: 0.01
validation_accuracy: 0.7661

c_value: 0.1
validation_accuracy: 0.9254

c_value: 1
validation_accuracy: 0.9578

c_value: 10
validation_accuracy: 0.9641

c_value: 100
validation_accuracy: 0.964

c_value: 1000
validation_accuracy: 0.964

c_value: 10000
validation_accuracy: 0.964
```

I have also graphed the c values with respect to their validation accuracy:



Overall I got:

```
Best c value for mnist is: 10
With validation accuracy of: 0.9641
```

Question 5: K-Fold Cross-Validation:

I trained this model using all of the training examples inside of the set (3337).
The C values that I tried:

[0.00000001,0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000],
[0.0000001, 0.000001,0.00001,0.0001,0.001,0.01,0.1,1],
[0.0001, 0.001,0.01,0.1,1,10,100,1000, 10000, 100000], and
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

But eventually went with this list:

[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

I generated these series with this website:
https://onlinenumbertools.com/generate-geometric-sequence

My code will print out all the validation accuracies for all the c values as they are calculated:

```
SPAM Dataset C value
Calculation:

c_value: 1
validation_accuracy:
0.039568345323741004

c_value: 1
validation_accuracy:
0.6714628297362111

c_value: 1
```

```
validation_accuracy:
0.9437125748502994

c_value: 1
validation_accuracy:
0.9556354916067147

c_value: 1
validation_accuracy:
0.9461077844311377

c_value: 2
```

```
validation_accuracy:
0.09832134292565947

c_value: 2
validation_accuracy:
0.6906474820143885

c_value: 2
validation_accuracy:
0.9293413173652695

c_value: 2
```

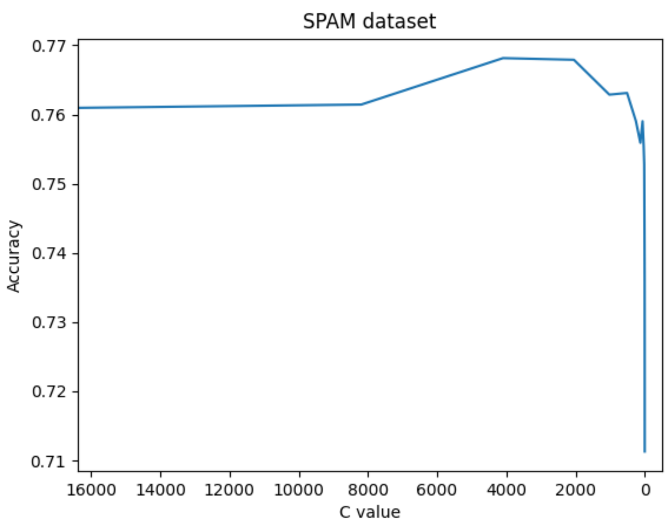
```
validation_accuracy:
0.9568345323741008

c_value: 2
validation_accuracy:
0.9425149700598803

c_value: 4
validation_accuracy:
0.1498800959232614

c_value: 4
validation_accuracy:
0.7062350119904077

c_value: 4
validation_accuracy:
0.925748502994012

c_value: 4
validation_accuracy:
0.9544364508393285

c_value: 4
validation_accuracy:
0.9437125748502994

c_value: 8
validation_accuracy:
0.17625899280575538

c_value: 8
validation_accuracy:
0.7218225419664268

c_value: 8
validation_accuracy:
0.9221556886227545

c_value: 8
validation_accuracy:
0.9580335731414868

c_value: 8
validation_accuracy:
0.9365269461077844

c_value: 16
validation_accuracy:
0.2182254196642686

c_value: 16
validation_accuracy:
0.7338129496402878

c_value: 16
validation_accuracy:
0.9281437125748503

c_value: 16
validation_accuracy:
0.9532374100719424

c_value: 16
validation_accuracy:
0.9305389221556887

c_value: 32
validation_accuracy:
0.22661870503597123

c_value: 32
validation_accuracy:
0.7422062350119905

c_value: 32
validation_accuracy:
0.925748502994012

c_value: 32
validation_accuracy:
0.9544364508393285

c_value: 32
validation_accuracy:
0.9281437125748503

c_value: 64
validation_accuracy:
0.26019184652278177

c_value: 64
validation_accuracy:
0.7553956834532374

c_value: 64
validation_accuracy:
0.9221556886227545

c_value: 64
validation_accuracy:
0.9460431654676259

c_value: 64
validation_accuracy:
0.911377245508982

c_value: 128
validation_accuracy:
0.26019184652278177

c_value: 128
validation_accuracy:
0.7565947242206235

c_value: 128
validation_accuracy:
0.9161676646706587

c_value: 128
validation_accuracy:
0.9424460431654677

c_value: 128
validation_accuracy:
0.9041916167664671

c_value: 256
validation_accuracy:
0.28896882494004794

c_value: 256
validation_accuracy:
0.7601918465227818

c_value: 256
validation_accuracy:
0.9125748502994012

c_value: 256
validation_accuracy:
0.9316546762589928

c_value: 256
validation_accuracy:
0.9017964071856287

c_value: 512
validation_accuracy:
0.2973621103117506

c_value: 512
validation_accuracy:
0.7589928057553957

c_value: 512
validation_accuracy:
0.9173652694610779

c_value: 512
validation_accuracy:
0.935251798561151

c_value: 512
validation_accuracy:
0.9065868263473054

c_value: 1024
validation_accuracy:
0.2961630695443645

c_value: 1024
validation_accuracy:
0.7613908872901679

c_value: 1024
validation_accuracy:
0.9173652694610779

c_value: 1024
validation_accuracy:
0.9328537170263789

c_value: 1024
validation_accuracy:
0.9065868263473054
```

```
c_value: 2048
validation_accuracy:
0.30815347721822545

c_value: 2048
validation_accuracy:
0.7889688249400479

c_value: 2048
validation_accuracy:
0.9173652694610779

c_value: 2048
validation_accuracy:
0.9256594724220624

c_value: 2048
validation_accuracy:
0.8994011976047904

c_value: 4096
validation_accuracy:
0.3129496402877698

c_value: 4096
```

```
validation_accuracy:
0.7913669064748201

c_value: 4096
validation_accuracy:
0.9149700598802395

c_value: 4096
validation_accuracy:
0.9232613908872902

c_value: 4096
validation_accuracy:
0.8982035928143712

c_value: 8192
validation_accuracy:
0.302158273381295

c_value: 8192
validation_accuracy:
0.7865707434052758

c_value: 8192
validation_accuracy:
0.9077844311377246
```

```
c_value: 8192
validation_accuracy:
0.920863309352518

c_value: 8192
validation_accuracy:
0.8898203592814371

c_value: 16384
validation_accuracy:
0.3057553956834532

c_value: 16384
validation_accuracy:
0.7817745803357314

c_value: 16384
validation_accuracy:
0.9041916167664671

c_value: 16384
validation_accuracy:
0.9244604316546763

c_value: 16384
validation_accuracy:
0.88862275449101
```

I have also graphed the c values with respect to their cross-validation accuracy:



Overall I got:

```
Best c value for SPAM is: 4096
With cross validation accuracy of: 0.7681503180688981
```

Question 6: Kaggle:

Kaggle username: **Hiva Mohammadzadeh**
Kaggle Scores:

a) MNIST: 96.633 %
b) SPAM: 82.666 %
c) CIFAR-10: 47.833 %

1. MNIST

I trained this model using 10000 training examples.
The C values that I tried were:

[0.0000001, 0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000],
[0.01,0.1,1,10],
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384], and
[0.001,0.01,0.1,1,10,100,1000]

But eventually went with this list:

[0.001,0.01,0.1,1,10,100,1000]

I generated these series with this website:
https://onlinenumbertools.com/generate-geometric-sequence

I used normalization to preprocess the data and then I used rbf as the kernel.

2.SPAM

I trained this model using all of the training examples inside of the set (3337).
The C values that I tried:

[0.00000001,0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000],
[0.0000001, 0.000001,0.00001,0.0001,0.001,0.01,0.1,1],
[0.0001, 0.001,0.01,0.1,1,10,100,1000, 10000, 100000], and
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

But eventually went with this list:

[1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

I generated these series with this website:
https://onlinenumbertools.com/generate-geometric-sequence

I used rbf as the kernel.

3.CIFAR-10

Repeated question 4 for CIFAR-10 to do hyperparameter tuning for the best C_value

```
CIFAR-10 Dataset C value Calculation:

c_value: 0.001
validation_accuracy: 0.0956

c_value: 0.01
validation_accuracy: 0.2456

c_value: 0.1
validation_accuracy: 0.3738

c_value: 1
validation_accuracy: 0.4778

c_value: 10
validation_accuracy: 0.4882

c_value: 100
validation_accuracy: 0.4824

c_value: 1000
validation_accuracy: 0.4824
```

I have also graphed the c values with respect to their validation accuracy:

Overall I got:

```
Best c value for cifar is: 10
```

Then tried tuning the c values, the kernel of svm.svc() and changed the kernel to be rbf. I also added a lot of features in to featurize.py. I used normalization to preprocess the data.

Question 7:

7 Theory of Hard-Margin Support Vector Machines

A *decision rule* (or *classifier*) is a function $r : \mathbb{R}^d \to \pm 1$ that maps a feature vector (test point) to $+1$ ("in class") or -1 ("not in class"). The decision rule for linear SVMs is of the form

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

where $w \in \mathbb{R}^d$ and $\alpha \in \mathbb{R}$ are the parameters of the SVM. The primal hard-margin SVM optimization problem (which chooses the parameters) is

$$\min_{w,\alpha} \ \|w\|^2 \ \text{ subject to } \ y_i(X_i \cdot w + \alpha) \geq 1, \ \ \forall i \in \{1, \ldots, n\}, \tag{2}$$

where $\|w\| = \sqrt{w \cdot w}$.

We can rewrite this optimization problem by using Lagrange multipliers to eliminate the constraints. (If you're curious to know what Lagrange multipliers are, the Wikipedia page is recommended, but you don't need to understand them to do this problem.) We thereby obtain the equivalent optimization problem

$$\max_{\lambda_i \geq 0} \min_{w,\alpha} \ \|w\|^2 - \sum_{i=1}^{n} \lambda_i (y_i(X_i \cdot w + \alpha) - 1). \tag{3}$$

Note: λ_i must be greater than or equal to 0.

(a) Show that Equation (3) can be rewritten as the *dual optimization problem*

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

Hint: Use calculus to determine and prove what values of w and α optimize Equation (3). Explain where the new constraint comes from.

(b) Suppose we know the values λ_i^* and α^* that optimize Equation (3). Show that the decision rule specified by Equation (1) can be written

$$r(x) = \begin{cases} +1 & \text{if } \alpha^* + \frac{1}{2} \sum_{i=1}^{n} \lambda_i^* y_i X_i \cdot x \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{5}$$

(c) Applying Karush–Kuhn–Tucker (KKT) conditions (See Wikipedia for more information), any pair of optimal primal and dual solutions w^*, α^*, λ^* for a linear, hard-margin SVM must satisfy the following condition:

$$\lambda_i^* (y_i(X_i \cdot w^* + \alpha^*) - 1) = 0 \ \ \forall i \in \{1, \ldots, n\}$$

This condition is called *complementary slackness*. Explain what this implies for points corresponding to $\lambda_i^* > 0$. What relationship do they have with the margin?

7 a) showing that $\max_{\lambda_i \geq 0} \min_{w, \alpha} ||w||^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)$ can be written as:

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

$$||w|| = \sqrt{w.w}$$

Following the hint and using calculus to find w and α: $\quad ||w||^2 = w^2$

$$\frac{\partial}{\partial w}\left(||w||^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)\right) = \frac{\partial}{\partial w}(w^2) + \frac{\partial}{\partial w}\left(-\sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)\right)$$

$$= \frac{\partial}{\partial w}(w^2) + \frac{\partial}{\partial w}\left(-\sum_{i=1}^{n} \lambda_i y_i X_i \cdot w + \lambda_i y_i \alpha) - \lambda_i y_i\right) = 2w - \sum_{i=1}^{n} \lambda_i y_i X_i = 0$$

set to 0 to find optimal value for w.

$$\rightarrow 2w = \sum_{i=1}^{n} \lambda_i y_i X_i \rightarrow \boxed{w = \frac{1}{2} \sum_{i=1}^{n} \lambda_i y_i X_i}$$

$$\frac{\partial}{\partial \alpha}\left(||w||^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)\right) = \frac{\partial}{\partial \alpha}(w^2) + \frac{\partial}{\partial \alpha}\left(-\sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)\right)$$

$$= \frac{\partial}{\partial \alpha}(w^2) + \frac{\partial}{\partial \alpha}\left(-\sum_{i=1}^{n} \lambda_i y_i X_i \cdot w + \lambda_i y_i \alpha) - \lambda_i y_i\right) = 0 - \sum_{i=1}^{n} \lambda_i y_i = 0$$

set to 0 to find optimal value for w.

$$\rightarrow \boxed{\alpha = \sum_{i=1}^{n} \lambda_i y_i}$$

In order to make sure that these values we found for w and α optimize equation 3, we have to show that they are minimum.

These values are minimum if the function that we took a derivative of is convex.

$$\underbrace{||w||^2}_{\text{L2 norm}} - \underbrace{\sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)}_{\text{linear}}$$ has to be convex.

L2 norm → convex by definition

linear → linear functions are convex.

Therefore, it is convex and the solution we found is the optimal solution.

To find where our new constraint $\left(\sum_{i=1}^{n} \lambda_i y_i = 0\right)$ comes from, we plug w in to eq 3:

→

7) cont. a) cont.

$\|w\|^2 - \sum_{i=1}^{n} \lambda_i (y_i (X_i \cdot w + \alpha) - 1)$ and $w = \frac{1}{2} \sum_{i=1}^{n} \lambda_i y_i x_i$, we get

$\sum_{i=1}^{n} \lambda_i y_i \alpha = 0 \rightarrow$ so, we get the constraint of equation 4.

---

b) know from eq 1) that $r(x) = \begin{cases} +1 & \text{if } w \cdot x + \alpha \geq 0 \\ -1 & \text{otherwise.} \end{cases}$, λ_i^* and α^* are known and from part a) that $W^* = \frac{1}{2} \sum_{i=1}^{n} \lambda_i y_i X_i$, then we just substitute and get:

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

---

c) complementary slackness holds if : $\lambda_i (y_i (X_i \cdot w^* + \alpha^*) - 1) = 0$

$\lambda_i^* > 0$, $\qquad\qquad$ $\lambda_i = 0 \rightarrow$ Not on margin

$y_i (X_i \cdot w^* + \alpha^*) = 1$

$\downarrow$

can be $+1$ or -1

and that will change the value of $x_i \cdot w^* + \alpha^*$ to be $+1$ or -1.

(d) The training points X_i for which $\lambda_i^* > 0$ are called the *support vectors*. In practice, we frequently encounter training data sets for which the support vectors are a small minority of the training points, especially when the number of training points is much larger than the number of features. Explain why the support vectors are the only training points needed to evaluate the decision rule.

(e) Assume the training points X_i and labels y_i are linearly separable. Using the original SVM formulation (not the dual) prove that there is at least one support vector for each class, $+1$ and -1.

Hint: Use contradiction. Construct a new weight vector $w' = w/(1 + \epsilon/2)$ and corresponding bias α' where $\epsilon > 0$. It is up to you to determine what ϵ should be based on the contradiction. If you provide a symmetric argument, you need only provide a proof for one of the two classes.

7 cont.

d) If not on margin ($\lambda_i^* = 0$) $\Rightarrow$ deleting the point will still give the same optimization $\Rightarrow$ No contribution on equation 5.

So, hard margin SVM classifies all training points correctly.

---

e) Proving that there's at least one support vector for each class, and following the hint to use contradiction and set $w' = \frac{w}{1+\varepsilon/2}$:

If we have no support vector on one of the classes (+1 or −1) we prove $y_i = 1$ and using symmetry, we know for $y_i = -1$.

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

Therefore $x_i^T \vec{w} + \alpha > 1$, so we want $\max x_i^T \vec{w} + \alpha = 1$ and $\min x_i^T \vec{w} + \alpha = 1 + \varepsilon$ for $\varepsilon > 0$

Letting $w' = \frac{w}{1+\frac{\varepsilon}{2}}$ $\qquad \frac{1+\varepsilon}{1+\varepsilon/2} = \frac{1+\varepsilon/2}{1+\varepsilon/2} + \frac{\varepsilon/2}{1+\varepsilon/2} = 1 + \frac{\varepsilon/2}{1+\varepsilon/2}$

$\alpha' = \frac{\alpha}{1+\varepsilon/2} - \frac{\varepsilon/2}{1+\varepsilon/2}$ Therefore $\min x_i^T \vec{w}' + \alpha' = 1$

Now, back to $y_i = -1 \Rightarrow \max \vec{x}_i^T \vec{w} + \alpha = -1$

$$\max x_i^T w' + \alpha' = \frac{-1}{1+\varepsilon/2} - \frac{\varepsilon/2}{1+\varepsilon/2} = \frac{-1-\varepsilon/2}{1+\varepsilon/2} = -1$$

$$\|w'\|^2 = \frac{1}{(1+\frac{\varepsilon}{2})^2} \|w\|^2 < \|w\|^2 \quad \checkmark$$

CODE APPENDIX:

Question 2: Data Partitioning

Lines 22 - 114 of Load.py

```
### QUESTION 2: shuffle and partition each of the datasets in the assignment.
#Shuffling prior to splitting crucially ensures that all classes are represented in
your partitions.

# MNIST
   print("\nMNIST Shuffling and Splitting:")
   data = np.load(f"../data/mnist-data.npz")
   training_data = data["training_data"]
   # print(training_data.shape)
   training_labels = data["training_labels"]
   # print(training_labels.shape)
   training_data = np.reshape(training_data, (60000, 784))
   training_labels = np.reshape(training_labels,(60000, 1))

   # Concatenating the labels with data in order to shuffle better
   concatenated_data = np.concatenate((training_data, training_labels), axis = 1)

   #Shuffling the data
   np.random.shuffle(concatenated_data)
   training_data = concatenated_data[:,:-1]
   training_labels = np.reshape(concatenated_data[:,-1], (60000, 1))
   # print(training_labels.shape)

   #Split using the amount we want in validation set
   amount_set_aside = 10000
   mnist_validation_data, mnist_validation_labels =
training_data[:amount_set_aside,:], training_labels[:amount_set_aside,:]
   mnist_training_data, mnist_training_labels = training_data[amount_set_aside:,:],
training_labels[amount_set_aside:,:]
   test_data = np.reshape(data["test_data"], (10000, 784))
   # print(test_data.shape)

   #Print the datasets' shapes
   print(f"Training data: {mnist_training_data.shape} \nTraining labels:
{mnist_training_labels.shape}")
   print(f"Validation data: {mnist_validation_data.shape} \nValidation labels:
{mnist_validation_labels.shape}")
#SPAM
```

```
    print("\nSPAM Shuffling and Splitting:")
    data = np.load(f"../data/spam-data.npz")
    training_data = data["training_data"]
    # print(training_data.shape)
    training_labels = data["training_labels"]
    # print(training_labels.shape)
    # training_data = np.reshape(training_data, (60000, 784))
    training_labels = np.reshape(training_labels,(4172, 1))

    # Concatenating the labels with data in order to shuffle better
    concatenated_data = np.concatenate((training_data, training_labels), axis = 1)

    #shuffling the data
    np.random.shuffle(concatenated_data)
    training_data = concatenated_data[:,:-1]
    training_labels = np.reshape(concatenated_data[:,-1], (4172, 1))

    #Split using the amount we want in validation set
    #Convert 20% percent to amount
    percent_set_aside = 0.20
    #print(training_data.shape)
    amount_set_aside = math.ceil(percent_set_aside * training_data.shape[0])
    spam_validation_data, spam_validation_labels = training_data[:amount_set_aside,:],
training_labels[:amount_set_aside,:]
    spam_training_data, spam_training_labels = training_data[amount_set_aside:,:],
training_labels[amount_set_aside:,:]

    #Print the datasets' shapes
    print(f"Training data: {spam_training_data.shape} \nTraining labels:
{spam_training_labels.shape}")
    print(f"Validation data: {spam_validation_data.shape} \nValidation labels:
{spam_validation_labels.shape}")



#CIFAR-10
    print("\nCIFAR-10 Shuffling and Splitting:")
    data = np.load(f"../data/cifar10-data.npz")
    training_data = data["training_data"]
    # print(training_data.shape)
    training_labels = data["training_labels"]
    # print(training_labels.shape)
```

```
    # training_data = np.reshape(training_data, (60000, 784))
    training_labels = np.reshape(training_labels,(50000, 1))

    # Concatenating the labels with data in order to shuffle better
    concatenated_data = np.concatenate((training_data, training_labels), axis = 1)

    #shuffling the data
    np.random.shuffle(concatenated_data)
    training_data = concatenated_data[:,:-1]
    training_labels = np.reshape(concatenated_data[:,-1], (50000, 1))

    #Split using the amount we want in validation set
    amount_set_aside = 5000
    cifar10_validation_data, cifar10_validation_labels =
training_data[:amount_set_aside,:], training_labels[:amount_set_aside,:]
    cifar10_training_data, cifar10_training_labels =
training_data[amount_set_aside:,:], training_labels[amount_set_aside:,:]

    #Print the datasets' shapes
    print(f"Training data: {cifar10_training_data.shape} \nTraining labels:
{cifar10_training_labels.shape}")
    print(f"Validation data: {cifar10_validation_data.shape} \nValidation labels:
{cifar10_validation_labels.shape}")
```

Question 3: Support Vector Machines: Coding

a) MNIST

Lines 119 - 160 of Load.py

```
### QUESTION 3: SVM. Training the models and calculating validation accuracies
# Part a) MNIST Dataset:
    print("\nMNIST Dataset accuracies:")
    mnist_model = svm.SVC()
    training_sizes = [100, 200, 500, 1000, 2000, 5000, 10000]
    accuracies = {"training": [], "validation": []}
    for training_size in training_sizes:
        print(f"Training with {training_size} examples")
        training_labels = np.asarray(mnist_training_labels).reshape(-1)
        validation_labels = np.asarray(mnist_validation_labels).reshape(-1)

        # Preprocessing and normalizing
        # print(np.max(mnist_training_data[:training_size,:]))
        # 255 is the maximum
        mnist_training_data = mnist_training_data / 255
        mnist_validation_data = mnist_validation_data / 255
```

```
        mnist_model.fit(mnist_training_data[:training_size, :],
training_labels[:training_size])

        # Calculate training and validation accuracies
        training_accuracy = metrics.accuracy_score(training_labels[:training_size],
mnist_model.predict(mnist_training_data[:training_size,:]))
        validation_accuracy = metrics.accuracy_score(validation_labels[:training_size],
mnist_model.predict(mnist_validation_data[:training_size,:]))
        accuracies["training"].append(training_accuracy)
        accuracies["validation"].append(validation_accuracy)
        print(f"Training accuracy: {training_accuracy} \nValidation Accuracy:
{validation_accuracy}\n")
    #Graph the plots
    plt.figure(1)
    plt.plot(training_sizes, accuracies["training"], '.r-')
    plt.plot(training_sizes, accuracies["validation"], '.b-')
    plt.legend(['Training', 'Validation'], loc=4)
    plt.xlabel("Number of Training Examples")
    plt.ylabel("Accuracy")
    plt.title("MNIST dataset")
    # plt.show()
    plt.savefig('mnist_accuracy.png')
```

b) SPAM:

Lines 164 - 199 of Load.py

```
# Part b) SPAM Dataset:
    print("\nSPAM Dataset accuracies:")
    spam_model = svm.SVC()
    training_sizes = [100, 200, 500, 1000, 2000, spam_training_data.shape[0]]
    # print(spam_training_data_x.shape[0])
    accuracies = {"training": [], "validation": []}

    for training_size in training_sizes:
        print(f"Training with {training_size} examples")

        training_labels = np.asarray(spam_training_labels).reshape(-1)
        validation_labels = np.asarray(spam_validation_labels).reshape(-1)
        spam_model.fit(spam_training_data[:training_size,:],
training_labels[:training_size])

        # Calculate training and validation accuracies
        training_accuracy = metrics.accuracy_score(training_labels[:training_size],
spam_model.predict(spam_training_data[:training_size,:]))
        validation_accuracy = metrics.accuracy_score(validation_labels[:training_size],
spam_model.predict(spam_validation_data[:training_size,:]))

        accuracies["training"].append(training_accuracy)
        accuracies["validation"].append(validation_accuracy)
```

```
        print(f"Training accuracy: {training_accuracy} \nValidation Accuracy:
{validation_accuracy}\n")

    #Graph the plots
    plt.figure(2)
    plt.plot(training_sizes, accuracies["training"], '.r-')
    plt.plot(training_sizes, accuracies["validation"], '.b-')
    plt.legend(['Training', 'Validation'], loc=4)
    plt.xlabel("Number of Training Examples")
    plt.ylabel("Accuracy")
    plt.title("SPAM dataset")
    # plt.show()
    plt.savefig('spam_accuracy.png')
```

c) CIFAR-10:

Lines 203 - 240 of Load.py

```
# Part c) CIFAR-10 Dataset:

    print("\nCIFAR-10 Dataset accuracies:")
    cifar10_model = svm.SVC(kernel = 'rbf')
    training_sizes = [100, 200, 500, 1000, 2000, 5000]
    accuracies = {"training": [], "validation": []}

    for training_size in training_sizes:

        print(f"Training with {training_size} examples")

        training_labels = np.asarray(cifar10_training_labels).reshape(-1)
        validation_labels = np.asarray(cifar10_validation_labels).reshape(-1)

        # Preprocessing and normalization:
        cifar10_training_data = (cifar10_training_data -
np.mean(cifar10_training_data))/np.std(cifar10_training_data)
        cifar10_validation_data = (cifar10_validation_data -
np.mean(cifar10_validation_data))/np.std(cifar10_validation_data)

        cifar10_model.fit(cifar10_training_data[:training_size, :],
training_labels[:training_size])

        # Calculate training and validation accuracies
        training_accuracy = metrics.accuracy_score(training_labels[:training_size],
cifar10_model.predict(cifar10_training_data[:training_size,:]))
```

```
        validation_accuracy= metrics.accuracy_score(validation_labels[:training_size],
cifar10_model.predict(cifar10_validation_data[:training_size,:]))

        accuracies["training"].append(training_accuracy)
        accuracies["validation"].append(validation_accuracy)

        print(f"Training accuracy: {training_accuracy} \nValidation Accuracy:
{validation_accuracy}\n")

    plt.figure(3)
    plt.plot(training_sizes, accuracies["training"], '.r-')
    plt.plot(training_sizes, accuracies["validation"], '.b-')
    plt.legend(['Training', 'Validation'], loc=4)
    plt.xlabel("Number of Training Examples")
    plt.ylabel("Accuracy")
    plt.title("CIFAR-10 dataset")
    # plt.show()
    plt.savefig('cifar10_accuracy.png')
```

Question 4: Hyperparameter Tuning:

Lines 245 - 295 of Load.py

```
### Question 4: Hyperparameter Tuning
# The best C value for MNIST Dataset

    print("\nMNIST Dataset C value Calculation: ")
    # mnist_training_size = mnist_training_data.shape[0]
    mnist_training_size = 10000
    #Function to train the dataset on the given c_value and calculate the validation
accuracy score for it
    #For Mnist and cifar since they need preprocessing
    def calculate_validation_accuracy_MNIST_CIFAR (train_data, train_labels, val_data,
val_labels, train_size, c_val):
        print(f"\nc_value: {c_val}")
        training_labels = train_labels.reshape(-1)
        validation_labels = val_labels.reshape(-1)
        model = svm.SVC( C = c_val)
        # Preprocessing and normalizing
        train_data = (train_data - np.mean(train_data))/np.std(train_data)
        val_data = (val_data - np.mean(val_data))/np.std(val_data)

        model.fit(train_data[:train_size], training_labels[:train_size])

        validation_accuracy = metrics.accuracy_score(validation_labels[:train_size],
model.predict(val_data[:train_size,:]))
        print(f"validation_accuracy: {validation_accuracy}")
```

```
        return validation_accuracy

    all_validation_accuracies =[]

    # Geometric series generated by
https://onlinenumbertools.com/generate-geometric-sequence using :
    # 1e-8 as the first element, 10 as the multiplication ratio, and 12 total numbers
in the sequesnce
    C_values = [0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000, 10000]
    # C_values = [0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10]
    # C_values = [1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

    # Calculate the accuracy for all the c_values
    for c_value in C_values:
        validation_accuracy =
calculate_validation_accuracy_MNIST_CIFAR(mnist_training_data, mnist_training_labels,
mnist_validation_data, mnist_validation_labels, mnist_training_size, c_value)
        all_validation_accuracies.append(validation_accuracy)

    # Dictionary of all c values and their validation accuracies
    dictionary = dict(zip(C_values, all_validation_accuracies))
    # Print the best C value that gives the highest validation accuracy
    print("\nBest c value for mnist is: " +
str(C_values[all_validation_accuracies.index(max(all_validation_accuracies))])
                                                + "\nWith validation accuracy of: " +
str(max(all_validation_accuracies)))

    plt.figure(4)
    plt.plot(C_values, all_validation_accuracies, label= "validation set")
    plt.xlabel("C value")
    plt.ylabel("Accuracy")
    plt.title("MNIST dataset")
    plt.xlim(max(C_values), -500)
    # plt.show()
    plt.savefig('MNIST_C_Value.png')
```

Question 5: K-Fold Cross-Validation:

Lines 300 - 377 of Load.py

```
## Question 5: K-Fold Cross Validation
# The best C value for Spam dataset
# Using 5-fold cross-validation, with at least 8 c_values

    #Function to train the dataset on the given c_value and calculate the validation
accuracy score for it
    #For SPAM since it doesn't need preprocessing
    def calculate_validation_accuracy_SPAM (train_data, train_labels, val_data,
val_labels, train_size, c_val):
        print(f"\nc_value: {c_val}")
        training_labels = train_labels.reshape(-1)
        validation_labels = val_labels.reshape(-1)
        model = svm.SVC( C = c_val)
```

```
        model.fit(train_data[:train_size], training_labels[:train_size])

        validation_accuracy = metrics.accuracy_score(validation_labels[:train_size],
model.predict(val_data[:train_size,:]))
        print(f"validation_accuracy: {validation_accuracy}")
        return validation_accuracy

    print("\nSPAM Dataset C value Calculation: ")
    spam_training_size = spam_training_data.shape[0]
    data = np.load(f"../data/spam-data.npz")
    #5-fold --> k=5
    k_value = 5
    spam_data = data["training_data"]
    spam_labels = data["training_labels"]
    # print(spam_labels.shape)
    spam_labels = np.reshape(spam_labels, (4172, 1))
    # print(spam_data.shape)
    fold = spam_data.shape[0]/k_value

    # Geometric series generated by
https://onlinenumbertools.com/generate-geometric-sequence using :
    # 1e-8 as the first element, 10 as the multiplication ratio, and 12 total numbers
in the sequence
    # C_values =
[0.00000001,0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000]
    # C_values = [0.0000001, 0.000001,0.00001,0.0001,0.001,0.01,0.1,1]]
    # C_values = [0.0001, 0.001,0.01,0.1,1,10,100,1000, 10000, 100000]
    C_values = [1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]

    #Function to train the dataset on the given c_value and calculate the 5-fold
cross-validation accuracy score for it
    def calculate_k_folds_validation(c_value):
        all_validation_accuracies = []
        #"This process is repeated k times with each set chosen as the validation set
once."
        for i in range(k_value):
            #"Model is trained on k − 1 sets and validated on the kth set."
            beginning_val = int(fold*i)
            ending_val = int(fold*(i+1))
            spam_validation_data = spam_data[beginning_val:ending_val]
            spam_validation_labels = spam_labels[beginning_val:ending_val]
            # print(spam_labels.shape)
            spam_training_data =
np.vstack((spam_data[:beginning_val],spam_data[ending_val:]))
            spam_training_labels =
np.vstack((spam_labels[:beginning_val],spam_labels[ending_val:]))

            #Call the function to calculate the validation accuracy

all_validation_accuracies.append(calculate_validation_accuracy_SPAM(spam_training_data
, spam_training_labels, spam_validation_data,
                                    spam_validation_labels, spam_training_size,
c_value))
        return all_validation_accuracies
```

```
   # Calculate the k-folds accuracies for each C value
   cross_validation_accuracies = [calculate_k_folds_validation(c_value) for c_value in
C_values]
   # "The cross-validation accuracy we report is the accuracy averaged over the k
iterations." Average the validation accuracies
   cross_validation_accuracy = [sum(i)/len(i) for i in cross_validation_accuracies]

   # dictionary of all c values and their validation accuracies
   dictionary = dict(zip(C_values, cross_validation_accuracy))

   #Take the c value that gives the maximum average validation accuracy
   best_c_value =
C_values[cross_validation_accuracy.index(max(cross_validation_accuracy))]
   print("Best c value for SPAM is: " + str(best_c_value) + "\nWith cross validation
accuracy of: " + str(max(cross_validation_accuracy)))

   plt.figure(5)
   plt.plot(C_values, cross_validation_accuracy, label= "validation set")
   plt.xlabel("C value")
   plt.ylabel("Accuracy")
   plt.title("SPAM dataset")
   plt.xlim(max(C_values), -500)
   # plt.show()
   plt.savefig('SPAM_C_Value.png')
```

Question 6: Kaggle:

a) MNIST

Lines 382 - 410 of Load.py

```
### QUESTION 6: Kaggle

# MNIST Dataset:
   #Function to test the model
   def test(training_data, training_labels, testing_data, training_size, C_value):
       training_labels = training_labels.reshape(-1,)
       model = svm.SVC(C=C_value)

       model.fit(training_data[:training_size,:],training_labels[:training_size])
       return model.predict(testing_data)

   #List to hold the final predictions on the test data
   mnist_final_test_predictions= []

   print("\nTesting data for mnist")
   data = np.load(f"../data/mnist-data.npz")
   mnist_test_data = data["test_data"]
```

```
    # print(mnist_test_data.shape)
    mnist_test_data = np.reshape(mnist_test_data, (10000, 784))
    # print(mnist_test_data.shape)
    # Preprocessing and normalizing
    mnist_training_data = (mnist_training_data -
np.mean(mnist_training_data))/np.std(mnist_training_data)
    mnist_test_data = (mnist_test_data -
np.mean(mnist_test_data))/np.std(mnist_test_data)
    # Calculate the final predictions (validation accuracies) for the test data
    # Using the best C value for MNIST calculated in question 4 (0.01)
    mnist_test_result = test(mnist_training_data, mnist_training_labels,
mnist_test_data, mnist_training_size, 10)
    # Save the result to a csv file
    results_to_csv(mnist_test_result)
    print("\nSuccessfully ran on test data for MNIST and saved to the csv file\n")
```

b) SPAM

Lines 414 - 430 of Load.py

```
# SPAM Dataset:
    #List to hold the final predictions on the test data
    spam_final_test_predictions= []
    print("\nTesting data for spam")

    data = np.load(f"../data/spam-data.npz")
    spam_test_data = data["test_data"]
    # print(spam_test_data.shape)
    # spam_test_data = (spam_test_data -
np.mean(spam_test_data))/np.std(spam_test_data)
    # Calculate the final predictions (validation accuracies) for the test data
    # using the best C value for MNIST calculated in question 5 (0.01)

    spam_test_result = test(spam_training_data, spam_training_labels, spam_test_data,
spam_training_size, 4096)
    # Save the result to a csv file
    results_to_csv(spam_test_result)
    print("\nSuccessfully ran on test data for SPAM and saved to the csv file\n")
```

And edited Features.py

```
# --------- Add your own feature methods ----------
def example_feature(text, freq):
   return int('example' in text)

def freq_scam(text, freq):
   return float(freq['scam'])

def freq_hello(text, freq):
   return float(freq['hello'])

def freq_cheap(text, freq):
   return float(freq['cheap'])

def freq_texas(text, freq):
   return float(freq['texas'])

def freq_adult(text, freq):
   return float(freq['adult'])

def freq_sincerely(text, freq):
   return float(freq['sincerely'])

def freq_free(text, freq):
   return float(freq['free'])

def freq_price(text, freq):
   return float(freq['price'])

def freq_congratulations(text, freq):
   return float(freq['congratulations'])

def freq_congrats(text, freq):
   return float(freq['congrats'])

def freq_buy(text, freq):
   return float(freq['buy'])

def freq_discount(text, freq):
   return float(freq['discount'])

def freq_fast(text, freq):
   return float(freq['fast'])

def freq_forwarded(text, freq):
   return float(freq['forwarded'])

def freq_question_mark(text, freq):
   return float(freq['?'])

def freq_urgent(text, freq):
```

```
    return float(freq['urgent'])

def freq_limited(text, freq):
    return float(freq['limited'])

def freq_ect(text, freq):
    return float(freq['ect'])

def freq_hou(text, freq):
    return float(freq['hou'])

def freq_enron(text, freq):
    return float(freq['enron'])

def freq_meter(text, freq):
    return float(freq['meter'])

def freq_cc(text, freq):
    return float(freq['cc'])

def freq_nbsp(text, freq):
    return float(freq['nbsp'])

def freq_td(text, freq):
    return float(freq['td'])

def freq_font(text, freq):
    return float(freq['font'])

def freq_computron(text, freq):
    return float(freq['computron'])

def freq_2004(text, freq):
    return float(freq['2004'])

def freq_pills(text, freq):
    return float(freq['pills'])

def freq_sex(text, freq):
    return float(freq['sex'])

    # --------- Add your own features here ---------
    # Make sure type is int or float

    feature.append(freq_hello(text, freq))
    feature.append(freq_scam(text, freq))
    feature.append(freq_cheap(text, freq))
    feature.append(freq_texas(text, freq))
    feature.append(freq_adult(text, freq))
    feature.append(freq_sincerely(text, freq))
    feature.append(freq_free(text, freq))
    feature.append(freq_price(text, freq))
```

```
    feature.append(freq_congrats(text, freq))
    feature.append(freq_congratulations(text, freq))
    feature.append(freq_buy(text, freq))
    feature.append(freq_discount(text, freq))
    feature.append(freq_fast(text, freq))
    feature.append(freq_forwarded(text, freq))
    feature.append(freq_question_mark(text, freq))
    feature.append(freq_urgent(text, freq))
    feature.append(freq_limited(text, freq))
    feature.append(freq_ect(text, freq))
    feature.append(freq_hou(text, freq))
    feature.append(freq_enron(text, freq))
    feature.append(freq_meter(text, freq))
    feature.append(freq_cc(text, freq))
    feature.append(freq_nbsp(text, freq))
    feature.append(freq_td(text, freq))
    feature.append(freq_font(text, freq))
    feature.append(freq_computron(text, freq))
    feature.append(freq_pills(text, freq))
    feature.append(freq_2004(text, freq))
    feature.append(freq_sex(text, freq)
```

c) CIFAR-10

Lines 435 - 508 of Load.py

```
## Repeating Question 4 for CIFAR-10: Hyperparameter Tuning
# The best C value for cifar-10 Dataset

   print("\nCIFAR-10 Dataset C value Calculation: ")
   # cifar_training_size = cifar10_training_data.shape[0]
   cifar_training_size = 10000

   #Function to train the dataset on the given c_value and calculate the validation
accuracy score for it
   def calculate_validation_accuracy (train_data, train_labels, val_data, val_labels,
train_size, c_val):
       print(f"\nc_value: {c_val}")
       training_labels = train_labels.reshape(-1)
       validation_labels = val_labels.reshape(-1)
       mnist_model = svm.SVC(C = c_val)
       mnist_model.fit(train_data[:train_size], training_labels[:train_size])

       validation_accuracy = metrics.accuracy_score(validation_labels[:train_size],
mnist_model.predict(val_data[:train_size,:]))
       print(f"validation_accuracy: {validation_accuracy}")
       return validation_accuracy
```

```
    all_validation_accuracies =[]

    # Geometric series generated by
https://onlinenumbertools.com/generate-geometric-sequence using :
    # 1e-8 as the first element, 10 as the multiplication ratio, and 12 total numbers
in the sequesnce
    # C_values = [0.0000001, 0.000001,0.00001,0.0001,0.001,0.01,0.1,1,10,100,1000]
    # C_values = [0.01,0.1,1,10]
    # C_values = [1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384]
    C_values = [0.001,0.01,0.1,1,10,100,1000]

    # Calculate the accuracy for all the c_values
    for c_value in C_values:
        cifar10_training_data = (cifar10_training_data -
np.mean(cifar10_training_data))/np.std(cifar10_training_data)
        cifar10_validation_data = (cifar10_validation_data -
np.mean(cifar10_validation_data))/np.std(cifar10_validation_data)
        validation_accuracy = calculate_validation_accuracy(cifar10_training_data,
cifar10_training_labels, cifar10_validation_data, cifar10_validation_labels,
cifar_training_size, c_value)
        all_validation_accuracies.append(validation_accuracy)

    # Dictionary of all c values and their validation accuracies
    dictionary = dict(zip(C_values, all_validation_accuracies))
    # Print the best C value that gives the highest validation accuracy
    print("\nBest c value for cifar is: " +
str(C_values[all_validation_accuracies.index(max(all_validation_accuracies))]))

    plt.figure(5)
    plt.plot(C_values, all_validation_accuracies, label= "validation set")
    plt.xlabel("C value")
    plt.ylabel("Accuracy")
    plt.title("Cifar dataset")
    plt.xlim(max(C_values), -500)
    # plt.show()
    plt.savefig('cifar_10_C_Value.png')

# CIfAR-10 Dataset:

    #List to hold the final predictions on the test data
    cifar10_final_test_predictions= []
```

```
    print("\nTesting data for cifar-10")

    data = np.load(f"../data/cifar10-data.npz")
    cifar10_test_data = data["test_data"]
    # print(cifar10_test_data.shape)

    ## cifar10_test_data = (cifar10_test_data -
np.mean(cifar10_test_data))/np.std(cifar10_test_data)
    #Calculate the final predictions (validation accuracies) for the test data
    # using the best C value for MNIST calculated in question 5 (0.01)
    cifar10_training_data = (cifar10_training_data -
np.mean(cifar10_training_data))/np.std(cifar10_training_data)
    cifar10_test_data = (cifar10_test_data -
np.mean(cifar10_test_data))/np.std(cifar10_test_data)

    # print("Hi")
    cifar10_test_result = test(cifar10_training_data, cifar10_training_labels,
cifar10_test_data, cifar_training_size, 10)
    # Save the result to a csv file
    results_to_csv(cifar10_test_result)
    print("\nSuccessfully ran on test data for CIFAR-10 and saved to the csv file\n")
```