CS 189: Homework 1

Vinay Maruri

January 30, 2019

I did not work with anybody on this homework.
“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.”
Signature: Vinay Maruri

1 Python Configuration and Data Loading

Code Block 1:

```
import sys
if sys.version_info[0] < 3:
    raise Exception("Python 3 not detected.")
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from scipy import io
for data_name in ["mnist", "spam", "cifar10"]:
    data = io.loadmat("data/%s_data.mat" % data_name)
    print("\nloaded %s data!" % data_name)
    fields = "test_data", "training_data", "training_labels"
    for field in fields:
        print(field, data[field].shape)
```

This code block executed without error on my local environment. I configured my Python environment properly for this homework.

2 Data Partitioning

(a) For the MNIST dataset, write code that sets aside 10,000 training images as a validation set.

Code Block 2:

```
m_nist_data = io.loadmat("data/%s_data.mat" % "mnist")
len(m_nist_data['training_data'])
mindex = np.arange(60000)
np.random.shuffle(mindex)
mvalidindex = mindex[0:10000]
mtrainindex = mindex[10000:]
mtraindata = np.array(m_nist_data['training_data'])
mtrainlabels = np.array(m_nist_data['training_labels'])
mnist_validation = mtraindata[mvalidindex]
mnist_validationlabels = mtrainlabels[mvalidindex]
mnist_training = mtraindata[mtrainindex]
mnist_traininglabels = mtrainlabels[mtrainindex]
```

The code block above loads the data and then shuffles the mnist data using np.random.shuffle, and then using a non-random index,(10000), I partition the training data into a validation dataset (10000 images) and the rest is reserved for the training dataset.

(b) For the spam dataset, write code that sets aside 20% of the training data as a validation set.

Code Block 3:

```
spam_data = io.loadmat("data/%s_data.mat" % "spam")
len(spam_data['training_data'])
sindex = np.arange(5172)
np.random.shuffle(sindex)
svalidindex = sindex[0:1034]
strainindex = sindex[1034:]
straindata = np.array(spam_data['training_data'])
strainlabels = np.array(spam_data['training_labels'])
spam_validation = straindata[svalidindex]
spam_validationlabels = strainlabels[svalidindex]
spam_training = straindata[strainindex]
spam_traininglabels = strainlabels[strainindex]
```

The code block above shuffles the spam data using np.random.shuffle, and then using a non-random index, (1034), we partition the training data into a validation dataset (1034 datapoints) and the rest is reserved for training dataset.

(c) For the CIFAR-10 dataset, write code that sets aside 5,000 training images as a validation set. Be sure to shuffle your data before splitting it to make sure all the classes are represented in your partitions.

Code Block 4:

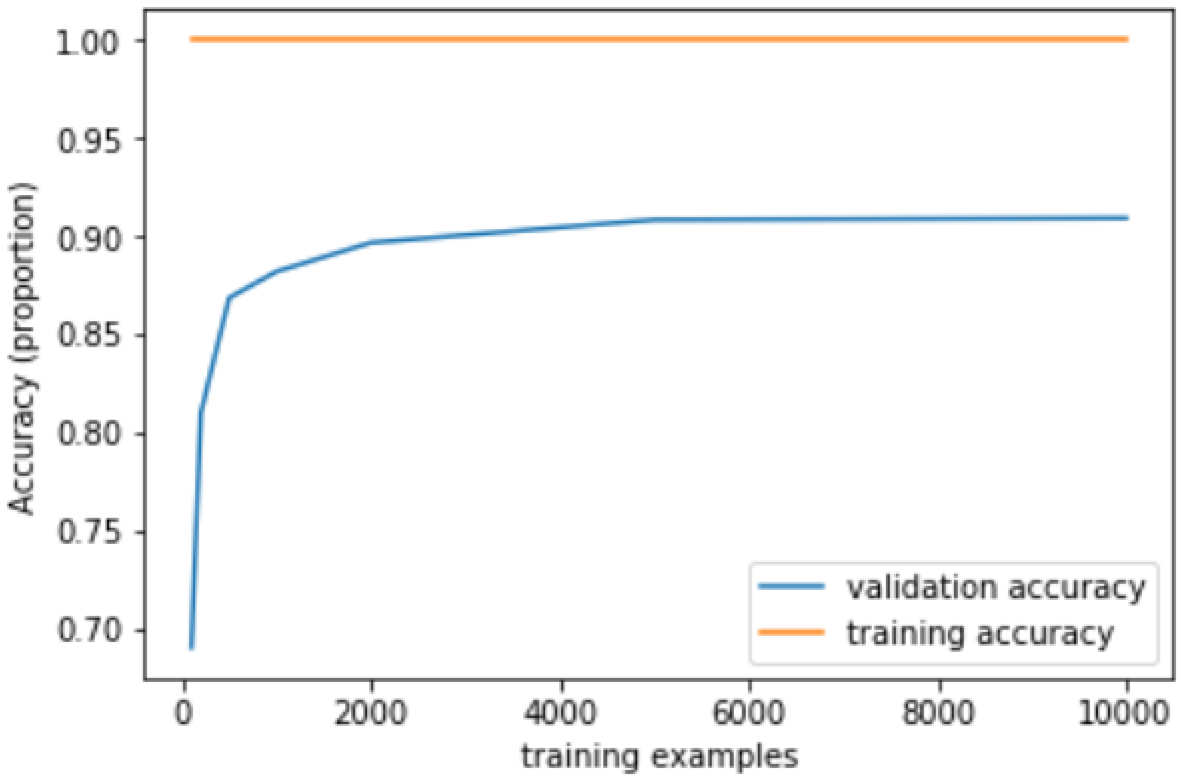
```
cifar_data = io.loadmat("data/%s_data.mat" % "cifar10")
len(cifar_data['training_data'])
cindex = np.arange(50000)
np.random.shuffle(cindex)
cvalidindex = cindex[0:5000]
ctrainindex = cindex[5000:]
ctraindata = np.array(cifar_data['training_data'])
ctrainlabels = np.array(cifar_data['training_labels'])
c_validation = ctraindata[cvalidindex]
c_validationlabels = ctrainlabels[cvalidindex]
c_training = ctraindata[ctrainindex]
c_traininglabels = ctrainlabels[ctrainindex]
```

The code above shuffles the cifar-10 data using np.random.shuffle, and then using a non-random index, (5000), we partition the training data into a validation dataset (5000 images) and the rest is reserved for training dataset.

3 Support Vector Machines: Coding

(a) For the MNIST dataset, use raw pixels as features. Train your model with the following numbers of training examples: 100, 200, 500, 1,000, 2,000, 5,000, 10,000. At this stage, you should expect accuracies between 70% and 90%.
Code Block 5:

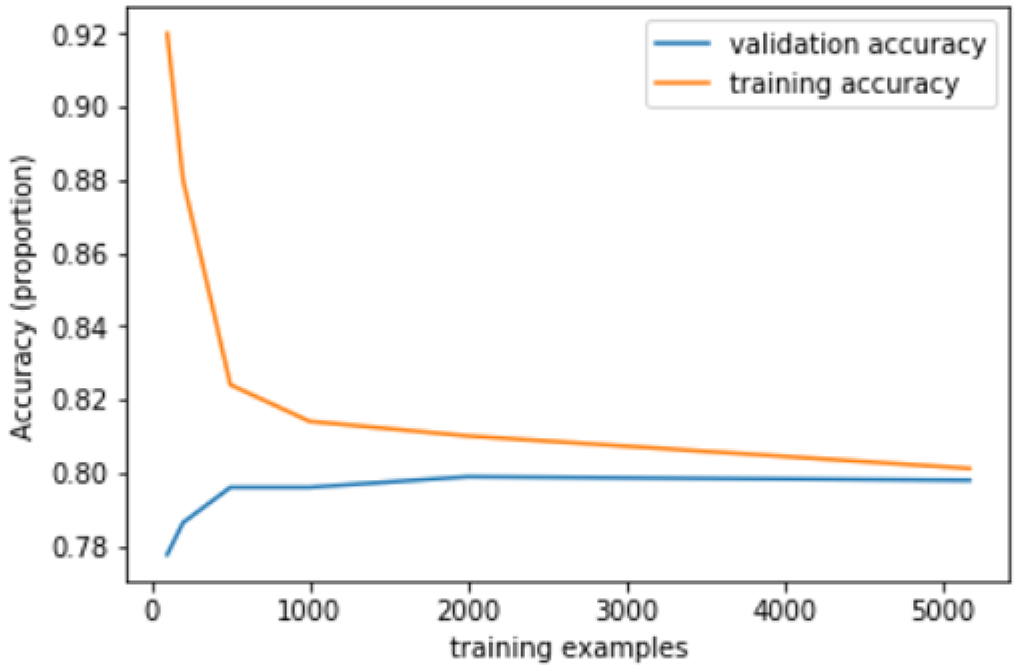
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
mnist_traininglabels = mnist_traininglabels.reshape(-1,)
def train_mnist(batchsize):
    clf = SVC(kernel = 'linear')
    clf.fit(mnist_training[0:batchsize], mnist_traininglabels[0:batchsize])
    prediction = clf.predict(mnist_training[0:batchsize])
    trainacc = accuracy_score(mnist_traininglabels[0:batchsize], prediction)
    validation_result = clf.predict(mnist_validation)
    validacc = accuracy_score(mnist_validationlabels, validation_result)
    print([trainacc, validacc, batchsize])
train_mnist(100)
train_mnist(200)
train_mnist(500)
train_mnist(1000)
train_mnist(2000)
train_mnist(5000)
train_mnist(10000)
mnist_train_lst = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
mnist_valid_lst = [0.6908, 0.8101, 0.8686, 0.882, 0.8967, 0.9084, 0.9092]
mnist_sizes = [100, 200, 500, 1000, 2000, 5000, 10000]
#results
ax = plt.plot(mnist_sizes, mnist_valid_lst, label = 'validation accuracy')
ay = plt.plot(mnist_sizes, mnist_train_lst, label = 'training accuracy')
plt.xlabel('training examples')
plt.ylabel('Accuracy (proportion)')
plt.legend()
#graph code
```



The code block above trains my support vector machine on 100, 200, 500, 1,000, 2,000, 5,000, and 10,000 training examples from our mnist training set. Then, after collecting my results for the validation and training errors, I have created 2 line graphs for training and validation accuracies for my classifier.

(b) For the spam dataset, use the provided word frequencies as features. In other words, each document is represented by a vector, where the ith entry denotes the number of times word i (as specified in featurize.py) is found in that document. Train your model with the following numbers of training examples: 100, 200, 500, 1,000, 2,000, ALL.
Code Block 6:

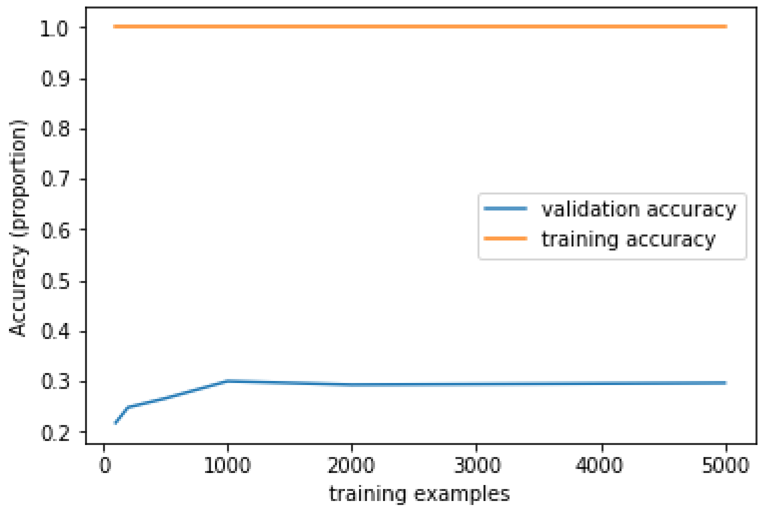
```
spam_traininglabels = spam_traininglabels.reshape(-1,)
def train_spam(batchsize):
    clf2 = SVC(kernel = 'linear')
    clf2.fit(spam_training[0:batchsize], spam_traininglabels[0:batchsize])
    prediction = clf2.predict(spam_training[0:batchsize])
    accuracy = accuracy_score(spam_traininglabels[0:batchsize], prediction)
    validationp = clf2.predict(spam_validation)
    validacc = accuracy_score(spam_validationlabels, validationp)
    print([accuracy, validacc, batchsize])
train_spam(100)
train_spam(200)
train_spam(500)
train_spam(1000)
train_spam(2000)
train_spam(5172)
spam_train_lst = [0.92, 0.88, 0.824, 0.814, 0.81, 0.8011116481391977]
spam_valid_lst = [0.7775628626692457, 0.7862669245647969, 0.7959381044487428,
                  0.7959381044487428, 0.7988394584139265, 0.7978723404255319]
spam_sizes = [100, 200, 500, 1000, 2000, 5172]
#results
ax = plt.plot(spam_sizes, spam_valid_lst, label = 'validation accuracy')
ay = plt.plot(spam_sizes, spam_train_lst, label = 'training accuracy')
plt.xlabel('training examples')
plt.ylabel('Accuracy (proportion)')
plt.legend()
#graph code
```



The code block above trains my support vector machine on 100, 200, 500, 1,000, 2,000, and 5,172 training examples in the spam training set. Then, after collecting my results, I created 2 line graphs for training and validation accuracies for my spam classifier.

(c) For the CIFAR-10 dataset, use raw pixels as features. At this stage, you should expect accuracies between 25% and 35%. Be forwarned that training SVMs for CIFAR-10 takes a couple minutes to run for a large training set. Train your model with the following numbers of training examples: 100, 200, 500, 1,000, 2,000, 5,000.
Code Block 7:

```
def train_cifar(batchsize):
    clf3 = SVC(kernel = 'linear')
    clf3.fit(c_training[0:batchsize], c_traininglabels[0:batchsize])
    prediction = clf3.predict(c_training[0:batchsize])
    accuracy = accuracy_score(c_traininglabels[0:batchsize], prediction)
    validationp = clf3.predict(c_validation)
    validacc = accuracy_score(c_validationlabels, validationp)
    print([accuracy, validacc, batchsize])
c_traininglabels = c_traininglabels.reshape(-1,)
train_cifar(100)
train_cifar(200)
train_cifar(500)
train_cifar(1000)
train_cifar(2000)
train_cifar(5000)
cifar_train_lst = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
cifar_valid_lst = [0.2178, 0.2484, 0.2662, 0.3002, 0.2932, 0.2968]
cifar_sizes = [100, 200, 500, 1000, 2000, 5000]
#results
ax = plt.plot(cifar_sizes, cifar_valid_lst, label = 'validation accuracy')
ay = plt.plot(cifar_sizes, cifar_train_lst, label = 'training accuracy')
plt.xlabel('training examples')
plt.ylabel('Accuracy (proportion)')
plt.legend()
#graph code
```



The code block above trains my support vector machine on 100, 200, 500, 1000, 2000, and 5000 training examples in the cifar training set. Then, after collecting my results, I created 2 line graphs for training and validation accuracies for my spam classifier.

4 Hyperparameter Tuning

(a) For the MNIST dataset, find the best C value. In your report, list the C values you tried, the corresponding accuracies, and the best C value. As in the previous problem, for performance reasons, you are required to train with up to 10,000 training examples but not required to train with more than that.
Code Block 8:

```
tuningset = mnist_validation
tuninglabels = mnist_validationlabels
tuninglabels = tuninglabels.reshape(-1,)
def c_tune(param):
    clf = SVC(kernel = 'linear', C=param)
    clf.fit(tuningset, tuninglabels)
    predictions = clf.predict(tuningset)
    return accuracy_score(tuninglabels, predictions)
accs = []
#list of training accuracies for the different C parameters
C_params = [1/1000000000, 1/100000000, 1/10000000, 1/100000, 1/10000,
1/1000, 1/100, 1/10, 1, 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8]
#list of C params being tested
for y in C_params:
    result = c_tune(y)
    accs.append(result)
print(accs)
```

In this problem, I tested the following C values: $[10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100, 1000, 10^4, 10^5, 10^6, 10^7, 10^8]$. Their corresponding validation accuracies were (same order as C values): [0.6671, 0.8975, 0.9373, 0.9959, 0.9999, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0].
The code block above shows the process I used to select the best C value. For every potential C value, I fitted a linear SVM classifier using that C value onto the tuning (MNIST validation) set and its labels, and then predicted a set of labels using the tuning set again, and finally returning the validation set accuracy associated with the classifier with that C value.
The best C value appears to be 1/1000.

5 K-Fold Cross Validation

(a) For the spam dataset, use 5-fold cross-validation to find and report the best C value. In your report, list the C values you tried, the corresponding accuracies, and the best C value.
Code Block 9:

```
sindex = np.arange(5172)
np.random.shuffle(sindex)
#5172/5 = 1034.4
set1 = sindex[0:1034]
set2 = sindex[1034:2068]
set3 = sindex[2068:3102]
set4 = sindex[3102:4036]
set5 = sindex[4036:]
spam_data['training_data'] = np.array(spam_data['training_data'])
spam_data['training_labels'] = np.array(spam_data['training_labels'])
vset1 = spam_data['training_data'][set1]
vset2 = spam_data['training_data'][set2]
vset3 = spam_data['training_data'][set3]
vset4 = spam_data['training_data'][set4]
vset5 = spam_data['training_data'][set5]
vlabel1 = spam_data['training_labels'][set1].reshape(-1,)
vlabel2 = spam_data['training_labels'][set2].reshape(-1,)
vlabel3 = spam_data['training_labels'][set3].reshape(-1,)
vlabel4 = spam_data['training_labels'][set4].reshape(-1,)
vlabel5 = spam_data['training_labels'][set5].reshape(-1,)
C_params = [1/1000000000, 1/100000000, 1/10000000, 1/100000, 1/10000, 1/1000,
1/100, 1/10, 1, 10, 100]
#list of C params being tested
vacc1 = []
vacc2 = []
vacc3 = []
vacc4 = []
vacc5 = []
for c in C_params:
    clf = SVC(kernel = 'linear', C = c)
    clf.fit(vset1, vlabel1)
    clf.fit(vset2, vlabel2)
    clf.fit(vset3, vlabel3)
    clf.fit(vset4, vlabel4)
    predictions = clf.predict(vset5)
    vacc5.append(accuracy_score(vlabel5, predictions))

for c in C_params:
    clf = SVC(kernel = 'linear', C = c)
    clf.fit(vset2, vlabel2)
    clf.fit(vset3, vlabel3)
    clf.fit(vset4, vlabel4)
    clf.fit(vset5, vlabel5)
    predictions = clf.predict(vset1)
    vacc1.append(accuracy_score(vlabel1, predictions))

for c in C_params:
    clf = SVC(kernel = 'linear', C = c)
    clf.fit(vset3, vlabel3)
    clf.fit(vset4, vlabel4)
    clf.fit(vset5, vlabel5)
    clf.fit(vset1, vlabel1)
    predictions = clf.predict(vset2)
    vacc2.append(accuracy_score(vlabel2, predictions))

for c in C_params:
    clf = SVC(kernel = 'linear', C = c)
    clf.fit(vset1, vlabel1)
    clf.fit(vset2, vlabel2)
    clf.fit(vset4, vlabel4)
    clf.fit(vset5, vlabel5)
    predictions = clf.predict(vset3)
    vacc3.append(accuracy_score(vlabel3, predictions))

for c in C_params:
    clf = SVC(kernel = 'linear', C = c)
    clf.fit(vset1, vlabel1)
    clf.fit(vset2, vlabel2)
    clf.fit(vset3, vlabel3)
    clf.fit(vset5, vlabel5)
    predictions = clf.predict(vset4)
    vacc4.append(accuracy_score(vlabel4, predictions))

results = {}

i = 0
while i < 11:
    templst = []
    templst.append(vacc1[i])
    templst.append(vacc2[i])
    templst.append(vacc3[i])
    templst.append(vacc4[i])
    templst.append(vacc5[i])
    results[C_params[i]] = templst
    i = i+1

keys = results.keys()
avgs = []
for key in keys:
    avgs.append(np.average(results[key]))
print(avgs)
#best c value on average appears to be 10. (for spam)
```

The code block shows the process I used to implement 5-fold cross validation on the spam dataset. I randomly partitioned the training data into 5 disjoint sets, trained the linear SVM classifier on 4 of the sets and validated on the 5th set for a given C value. The process repeats for all C values, and the process is run 5 times, so that each set is chosen as the validation set once. I tried C values of 1/1000000000, 1/100000000, 1/10000000, 1/100000, 1/10000, 1/1000, 1/100, 1/10, 1, 10, 100. The reason why I did not test a C value greater than 100 was because doing so yielded unreasonable run times on my local machine of more than 30 minutes without a solution. The cross-validation accuracy for C values is listed as key-value pairs here (keys are C values, values is the cross-validation accuracy for that C value): {1/1000000000: 0.710224783981537, 1/100000000: 0.710224783981537, 1/10000000: 0.710224783981537, 1/100000: 0.710224783981537, 1/10000: 0.7109984783722527, 1/1000: 0.7324030815470317, 1/100: 0.7681882263557196, 1/10: 0.7933279789447882, 1: 0.8014197196982483, 10: 0.8058918557797441, 100: 0.8035333539180485}. It appears that a C value of 10 is the best value according to this 5-fold cross-validation process.

6 Kaggle

Code Block 10 (MNIST):
Kaggle Name: Vinay Maruri; Score: 0.94433
I didn't implement a non-linear SVM kernel or add more features. For the MNIST challenge, I only adjusted the C value of the classifier to the optimum value found in question 4 of 1/1000 and did pre-processing using StandardScaler().

```
mindex = np.arange(60000)
np.random.shuffle(mindex)
mvalidindex = mindex[0:10000]
mtrainindex = mindex[10000:]
mtraindata = np.array(m_nist_data['training_data'])
mtrainlabels = np.array(m_nist_data['training_labels'])
mnist_validation = mtraindata[mvalidindex]
mnist_validationlabels = mtrainlabels[mvalidindex]
mnist_training = mtraindata[mtrainindex]
mnist_traininglabels = mtrainlabels[mtrainindex]
mnist_traininglabels = mnist_traininglabels.reshape(-1,)
from sklearn.preprocessing import StandardScaler
mscaler = StandardScaler()
mnist_training = mscaler.fit_transform(mnist_training)
mnist_validation = mscaler.transform(mnist_validation)
clf = SVC(kernel = 'linear', C = 1/1000, gamma = 'scale', cache_size = 1000)
def train_mnist(batchsize):
    clf.fit(mnist_training[0:batchsize], mnist_traininglabels[0:batchsize])
    prediction = clf.predict(mnist_training[0:batchsize])
    trainacc = accuracy_score(mnist_traininglabels[0:batchsize], prediction)
    validation_result = clf.predict(mnist_validation)
    validacc = accuracy_score(mnist_validationlabels, validation_result)
    print([trainacc, validacc, batchsize])
train_mnist(100)
train_mnist(200)
train_mnist(500)
train_mnist(1000)
train_mnist(2000)
train_mnist(5000)
train_mnist(10000)
train_mnist(20000)
train_mnist(40000)
train_mnist(50000)
test_mnist = np.array(m_nist_data['test_data'])
test_mnist = mscaler.transform(test_mnist)
prediction = clf.predict(test_mnist)
import save_csv
```

```
save_csv.results_to_csv(prediction)
```

Code Block 11 (Spam):
Kaggle Name: Vinay Maruri; score: 0.82754
I didn't implement a non-linear SVM kernel or add more features. For the MNIST challenge, I only adjusted the C value of the classifier to the optimum value found in question 5 of 10 and did pre-processing using StandardScaler().

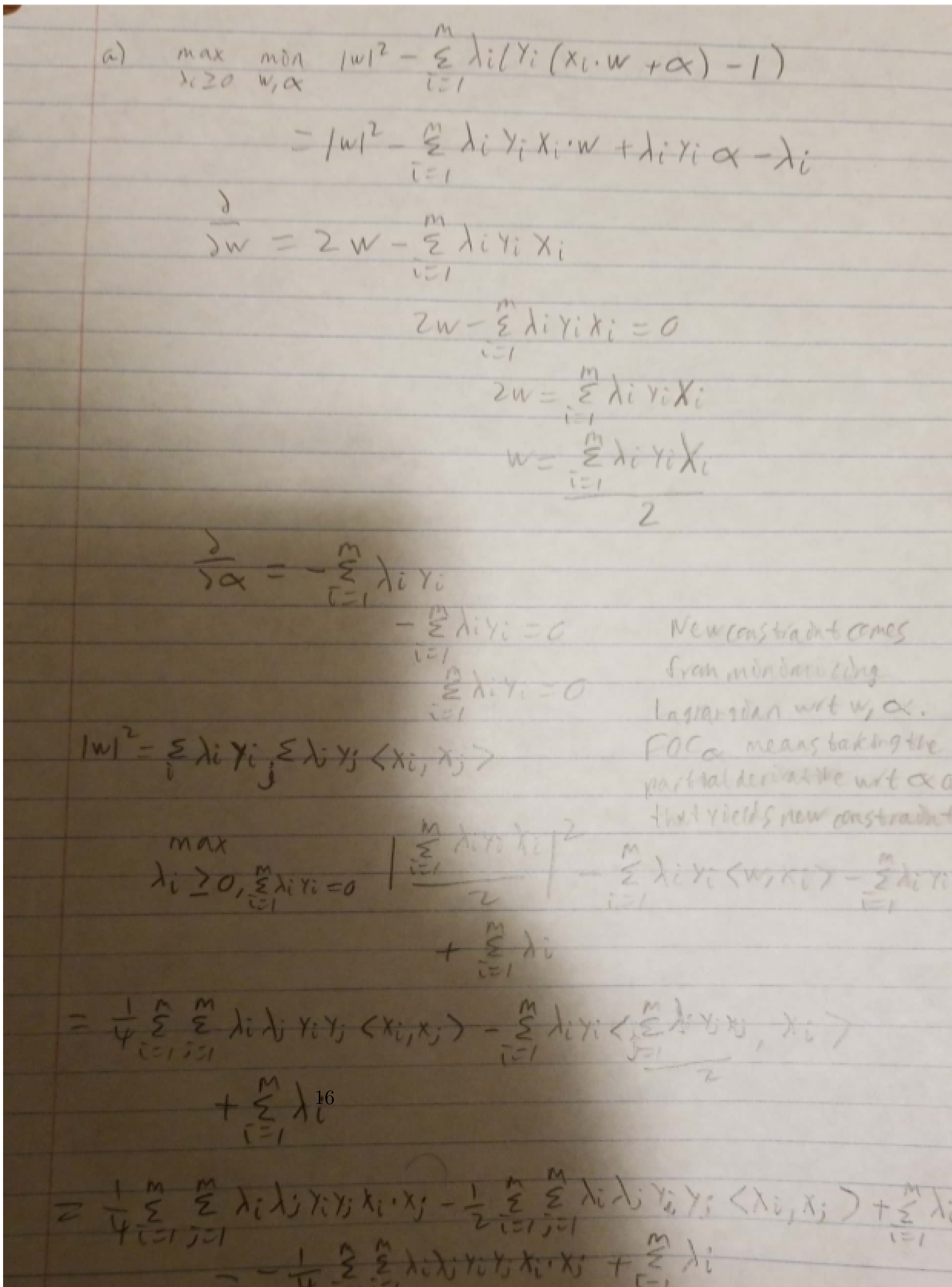
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
clf = SVC(kernel = 'linear', C = 10, gamma = 'scale', cache_size = 7000)
def trainspam(indexset):
    clf.fit(spam_training[indexset], spam_traininglabels[indexset])
    prediction = clf.predict(spam_training[indexset])
    print(accuracy_score(spam_traininglabels[indexset], prediction))

spam_traininglabels = np.array(spam_data['training_labels']).reshape(-1,)
spam_training = np.array(spam_data['training_data'])
indices = np.arange(5172)
np.random.shuffle(indices)
s_scaler = StandardScaler()
spam_training = s_scaler.fit_transform(spam_training)
sizes = [100, 200, 500, 1000, 2000, 3500, 5172]
for size in sizes:
    lst = indices[0:size]
    trainspam(lst)
spam_test = np.array(spam_data['test_data'])
spam_test = s_scaler.transform(spam_test)
test_prediction = clf.predict(spam_test)
import save_csv
save_csv.results_to_csv(test_prediction)
```

Code Block 12 (CIFAR):
Kaggle Name: Vinay Maruri; score: 0.29066
I didn't implement a non-linear SVM kernel or add new features. After iterated testing of C values and implementing pre-processing using StandardScaler(), I arrived at the model that performed best on my local machine and that is what I submitted to kaggle.

```
cindex = np.arange(50000)
np.random.shuffle(cindex)
cvalidindex = cindex[0:5000]
ctrainindex = cindex[5000:]
ctraindata = np.array(cifar_data['training_data'])
ctrainlabels = np.array(cifar_data['training_labels'])
c_validation = ctraindata[cvalidindex]
c_validationlabels = ctrainlabels[cvalidindex]
```

```
c_training = ctraindata[ctrainindex]
c_traininglabels = ctrainlabels[ctrainindex]
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
clf3 = SVC(kernel = 'linear', cache_size = 1000)
def train_cifar(batchsize):
    clf3.fit(c_training[0:batchsize], c_traininglabels[0:batchsize])
    prediction = clf3.predict(c_training[0:batchsize])
    accuracy = accuracy_score(c_traininglabels[0:batchsize], prediction)
    validationp = clf3.predict(c_validation)
    validacc = accuracy_score(c_validationlabels, validationp)
    print([accuracy, validacc, batchsize])
c_traininglabels = c_traininglabels.reshape(-1,)
ci_scaler = StandardScaler()
c_training = ci_scaler.fit_transform(c_training)
c_validation = ci_scaler.transform(c_validation)
train_cifar(100)
train_cifar(200)
train_cifar(500)
train_cifar(1000)
train_cifar(2000)
train_cifar(5000)
cifartest = np.array(cifar_data['test_data'])
cifartest = ci_scaler.transform(cifartest)
predictions = clf3.predict(cifartest)
import save_csv
save_csv.results_to_csv(predictions)
```

7 Theory of Hard-Margin Support Vector Machines



a) $\max_{\lambda_i \geq 0} \min_{w,\alpha} |w|^2 - \sum_{i=1}^{m} \lambda_i (y_i (x_i \cdot w + \alpha) - 1)$

$= |w|^2 - \sum_{i=1}^{m} \lambda_i y_i x_i \cdot w + \lambda_i y_i \alpha - \lambda_i$

$\frac{\partial}{\partial w} = 2w - \sum_{i=1}^{m} \lambda_i y_i x_i$

$2w - \sum_{i=1}^{m} \lambda_i y_i x_i = 0$

$2w = \sum_{i=1}^{m} \lambda_i y_i X_i$

$w = \frac{\sum_{i=1}^{m} \lambda_i y_i X_i}{2}$

$\frac{\partial}{\partial \alpha} = -\sum_{i=1}^{m} \lambda_i y_i$

$-\sum_{i=1}^{m} \lambda_i y_i = 0$

$\sum_{i=1}^{m} \lambda_i y_i = 0$

New constraint comes from minimizing Lagrangian wrt w, α. FOC_α means taking the partial derivative wrt α a that yields new constraint

$|w|^2 = \sum_i \lambda_i y_i \sum_j \lambda_j y_j \langle x_i, x_j \rangle$

$\max_{\lambda_i \geq 0, \sum_{i=1}^{m} \lambda_i y_i = 0} \left| \frac{\sum_{i=1}^{m} \lambda_i y_i x_i}{2} \right|^2 - \sum_{i=1}^{m} \lambda_i y_i \langle w, x_i \rangle - \sum_{i=1}^{m} \lambda_i y_i$

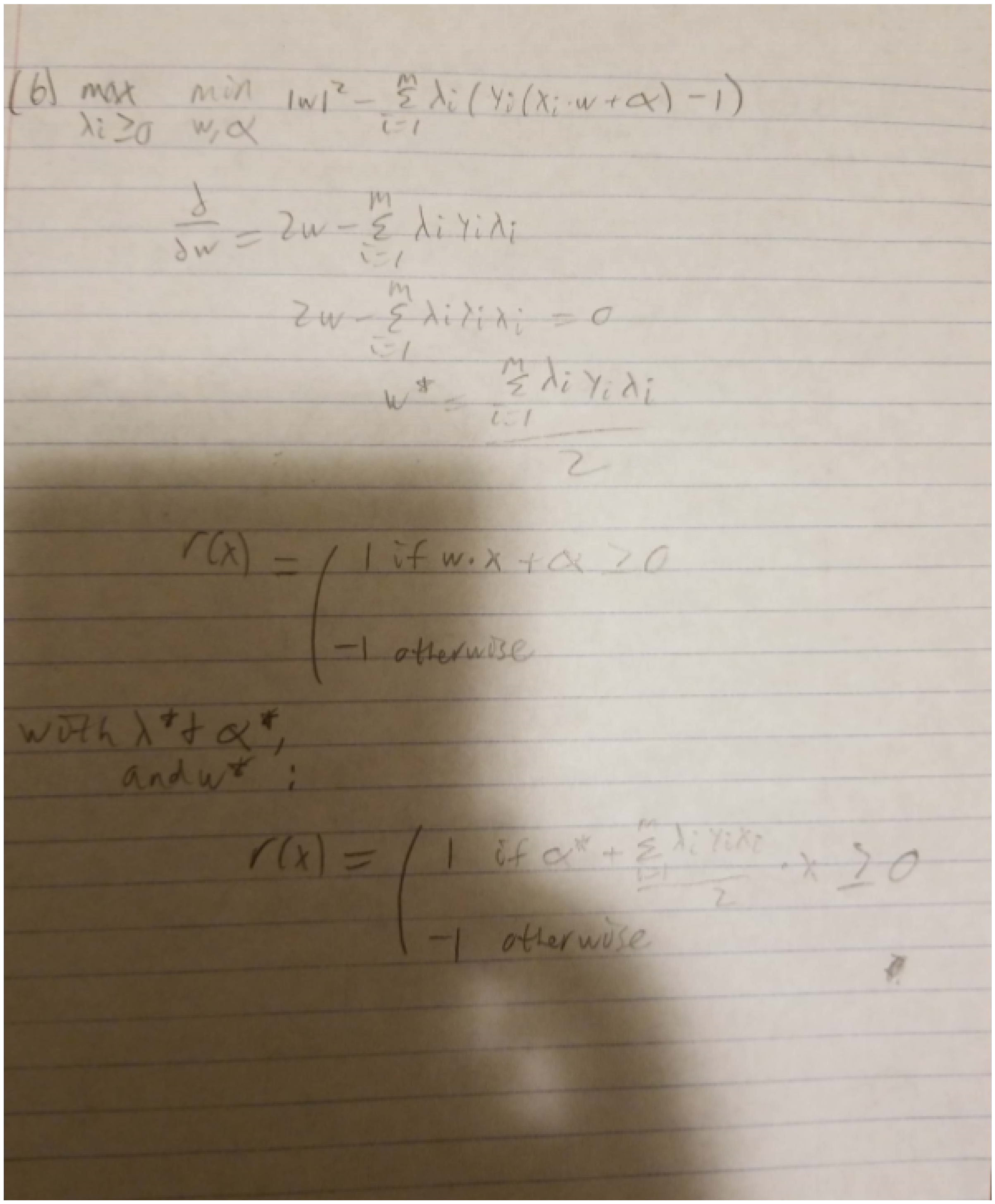
$+ \sum_{i=1}^{m} \lambda_i$

$= \frac{1}{4} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \langle x_i, x_j \rangle - \sum_{i=1}^{m} \lambda_i y_i \langle \frac{\sum_{j=1}^{m} \lambda_j y_j x_j}{2}, x_i \rangle$

$+ \sum_{i=1}^{m} \lambda_i$

$= \frac{1}{4} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j x_i \cdot x_j - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \langle \lambda_i, x_j \rangle + \sum_{i=1}^{m} \lambda_i$

$= -\frac{1}{4} \sum \sum_{i=1}^{m} \lambda_i \lambda_j y_i y_j x_i \cdot x_j + \sum_{i=1}^{m} \lambda_i$



(b) $\max_{\lambda_i \geq 0} \min_{w,\alpha} |w|^2 - \sum_{i=1}^{m} \lambda_i (y_i (x_i \cdot w + \alpha) - 1)$

$$\frac{\partial}{\partial w} = 2w - \sum_{i=1}^{m} \lambda_i y_i \lambda_i$$

$$2w - \sum_{i=1}^{m} \lambda_i y_i \lambda_i = 0$$

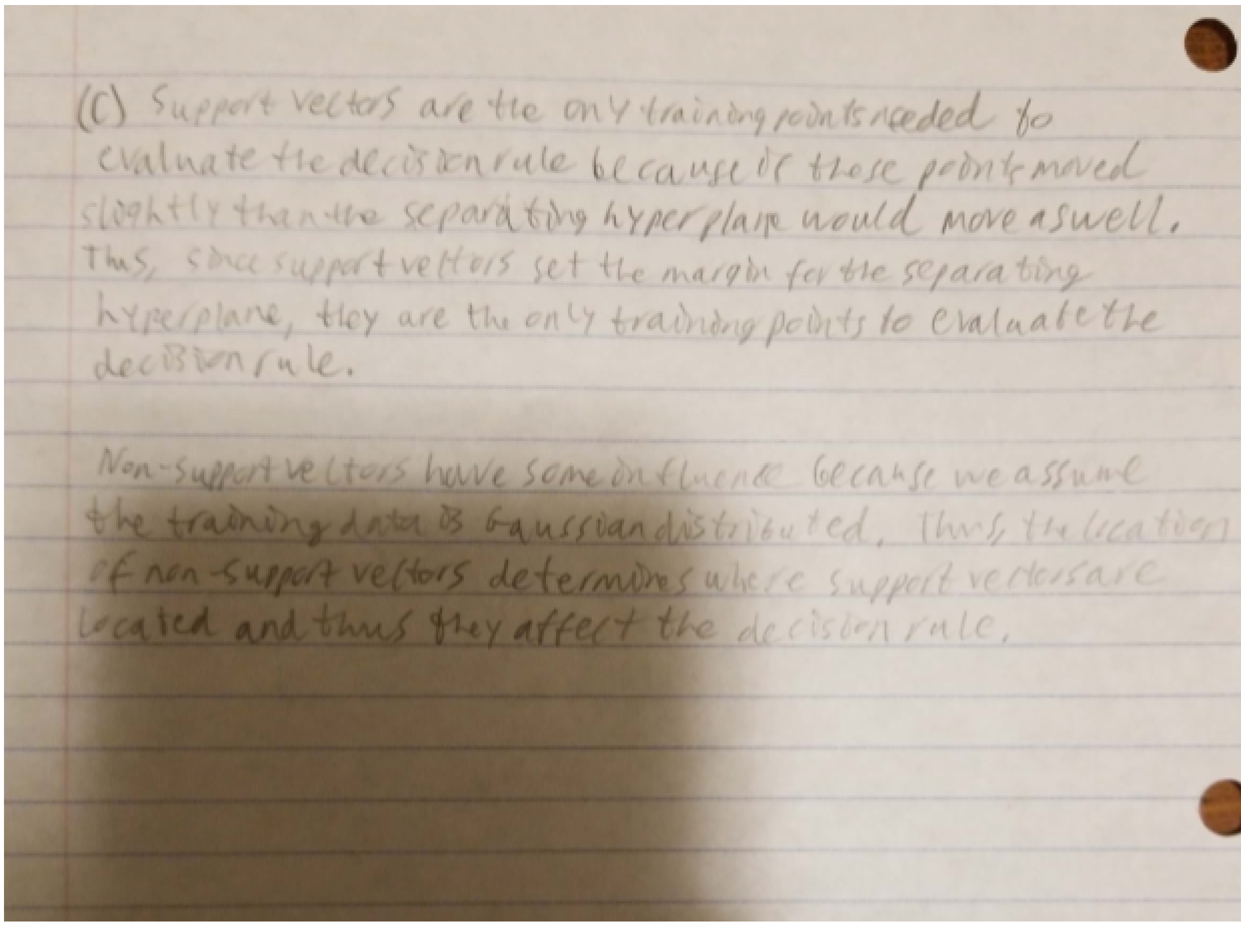
$$w^* = \frac{\sum_{i=1}^{m} \lambda_i y_i \lambda_i}{2}$$

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

with λ^* & α^*, and w^*:

$$r(x) = \begin{cases} 1 \quad \text{if } \alpha^* + \frac{\sum_{i=1}^{m} \lambda_i y_i x_i}{2} \cdot x \geq 0 \\ -1 \quad \text{otherwise} \end{cases}$$

(b)

(c) 

(c) Support vectors are the only training points needed to evaluate the decision rule because if those points moved slightly than the separating hyperplane would move as well. Thus, since support vectors set the margin for the separating hyperplane, they are the only training points to evaluate the decision rule.

Non-support vectors have some influence because we assume the training data is Gaussian distributed. Thus, the location of non-support vectors determines where support vectors are located and thus they affect the decision rule.