**Support Vector Machines**

Jericho McLeod

CSI-873

Table of Contents

[Purpose 3](#_Toc25928969)

[Methods 3](#_Toc25928970)

[Data 4](#_Toc25928971)

[Experiments: 4](#_Toc25928972)

[Pixel Reduction 5](#_Toc25928973)

[Data Instances Reduction 6](#_Toc25928974)

[Singular Value Decomposition 7](#_Toc25928975)

[Odd Vs. Even Classifier 8](#_Toc25928976)

[Support vector Machines in Parallel 8](#_Toc25928977)

[Comparing Models 9](#_Toc25928978)

[Bibliography 10](#_Toc25928979)

[Appendix: Code 10](#_Toc25928980)

# Purpose

This project explores Support Vector Machines in several experiments. First, data reduction techniques are compared by reducing training examples, the dimensions within training examples, and singular value decomposition. Then, different classification problems are explored, such as comparing odd versus even and using a new support vector machine for each digit. Finally, a comparison to other machine learning algorithms is made using the results of this and previously obtained results.

# Methods

The support vector machines for this project were created by optimizing the following problem:

The kernel function used was the radial bias machine:

And B was found using:

For some optimal alpha where alpha is greater than 0 and less than C.

Classifications were made using the following function, where results greater than or equal to 0 where classified as true, and those less than 0 were classified as falls.

# Data

The dataset used for this project is a sampling of NMIST handwritten digits, ingested as vectors of 785 numbers. The first number indicates the class, and the remaining 784 digits represent the 28x28 matrix of pixel values from 0 to 255. These were scaled to a range of 0 to 1. The dataset contained 60,000 training examples and 10,000 validation examples. Of these, up to 500 training and validation examples for digits 2 and 5, and 100 training examples for all other digits, where used at any given time.

# Experiments:

Support Vector Machines were explored in five experiments. For the first three, the data dimensionality was reduced by eliminating pixels within training instances, removing training instances, and conducting singular value decomposition. Then, the machine was trained on and subsequently classified a subset of the data for digits “2” and “5”. The results summary of these are seen below, though more detailed results and discussions are presented in the following sections.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Reduction | Pixels | Training Instances | SVD |
| 0% | 0.992 | 0.992 | 0.992 |
| 50% | 0.990 | 0.983 | 0.893 |
| 75% | 0.977 | 0.962 | 0.896 |
| 90% | 0.969 | 0.876 | 0.907 |
| 95% | 0.830 | 0.700 | 0.914 |

## Pixel Reduction

Reducing the number of pixels had very little impact on overall accuracy until the images were greatly reduced. Runtimes improved, but less so than other mechanisms allowed for. This, however, may be due to construction of the program rather than any particular aspect of this type of data reduction; I would want to experiment further to make any claims in this regard.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2 Classified 2 | 2 Classified 5 | 5 Classified 2 | 5 Classified 5 | Accuracy | CI ± |
| 100% of pixels | 499 | 1 | 7 | 493 | 99.2% | 0.55% |
| 50% of pixels | 496 | 4 | 6 | 493 | 99.0% | 0.62% |
| 25% of pixels | 486 | 14 | 9 | 491 | 97.7% | 0.93% |
| 10% of pixels | 480 | 20 | 11 | 489 | 96.9% | 1.07% |
| 5% of Pixels | 338 | 162 | 8 | 492 | 83.0% | 2.33% |

Training Instance Dimensionality Reduction Accuracy



## Data Instances Reduction

Reducing training instances similarly improved run times, but also more steeply reduced the effectiveness of the algorithm in classifying new observations. This is possibly because less variability was present in the training data, making the learned model less robust at classifying new observations. Given the faster reduction in accuracy relative to reducing dimensionality within instances, this is a sub-optimal method.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2 Classified 2 | 2 Classified 5 | 5 Classified 2 | 5 Classified 5 | Accuracy | CI ± |
| 500 Examples | 499 | 1 | 7 | 493 | 99.2% | 0.55% |
| 250 Examples | 498 | 2 | 12 | 488 | 98.3% | 0.73% |
| 125 Examples | 498 | 2 | 36 | 464 | 96.2% | 1.19% |
| 50 Examples | 499 | 1 | 123 | 477 | 87.6% | 2.04% |
| 25 Examples | 494 | 6 | 294 | 206 | 70.0% | 2.84% |

Training Instance Reduction Accuracy



## Singular Value Decomposition

Singular value decomposition lowers the resolution while preserving much of the information in the data in datasets such as NMIST. It was interesting in that it caused a rather steep drop off from using all of the data, but then results improved incrementally until 97.5% decomposition; at this point no further improvements were obtained and the machine began to decrease in accuracy. This is noteworthy because running the algorithm across such a reduced data set is significantly faster than considering the entire dataset and reducing the dataset by eliminating pixels or training instances the same degree resulted in markedly worse results. Further, it indicates that there may be some noise in the dataset which is not useful for classification that SVD happens to remove. This may be the cause of the improving accuracy from 50% to 97.5% decompositions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2 Classified 2 | 2 Classified 5 | 5 Classified 2 | 5 Classified 5 | Accuracy | CI ± |
| No SVD | 499 | 1 | 7 | 493 | 99.2% | 0.55% |
| 50% SVD | 485 | 15 | 92 | 408 | 89.3% | 1.92% |
| 75% SVD | 484 | 16 | 88 | 412 | 89.6% | 1.89% |
| 90% SVD | 485 | 15 | 78 | 422 | 90.7% | 1.80% |
| 95% SVD | 483 | 17 | 69 | 431 | 91.4% | 1.74% |
| 97.5% SVD | 483 | 17 | 58 | 442 | 92.5% | 1.63% |
| 99% SVD | 429 | 71 | 50 | 450 | 87.9% | 2.02% |
| 99.5% SVD | 264 | 236 | 8 | 492 | 75.6% | 2.66% |

SVD Example Image SVD Accuracy

A picture containing text, newspaper

Description automatically generated ![A close up of text on a white surface

Description automatically generated]()

(Guo, Zhang, Zhang, & Liu, 2016)

Of the three methods, if a new dataset required dimensionality reduction, I would prefer eliminating by pixel reduction for small decreases, while singular value decomposition would be the ideal choice for large datasets requiring, or capable of supporting, significant dimensionality reduction.

## Odd Vs. Even Classifier

Classifying odd vs. even digits was the fourth experiment. The results were surprisingly accurate, resulting in a machine that classified training examples correctly 93.4% of the time, with 95% confidence that the true classification rate is within 1.54% of this rate.

|  |  |  |
| --- | --- | --- |
| Odd v. Even | Classified Even | Classified Odd |
| True Even | 473 | 27 |
| True Odd | 61 | 461 |

﻿Accuracy: 0.934 ± 0.0154

## Support vector Machines in Parallel

Classifying all digits was the final experiment, as this allowed comparison to other models previously constructed. This experiment conducted utilized 100 examples of each digit in the dataset for both the training and testing datasets. Ten separate machines were constructed such that, for each digit, there was a corresponding machine that would classify that digit as a 1 and any other digit as -1. The best results for classification were used, so cases where there was no positive classification, or more than one possible classification, were selected as the best guess to the true classification in the machine.

Accuracy: ﻿0.879 ± 0.020

A close up of a black background

Description automatically generated

This was relatively accurate, though 3 was frequently misclassified as 5, and 4 as 9. The accuracy was below that of KNN and ANN algorithms, and above Naïve Bayes, with 95% confidence. Further comparisons between algorithms are made in the following section.

# Comparing Models

The four models used to make classifications for NMIST data are listed in the table below. While the training and classification times were not collected explicitly, I can state experientially that a large component of accuracy appears to be a cost in time. For this dataset, K-Nearest-Neighbors and Artificial Neural networks had a high cost in time. It is noteworthy, for neural networks, that reducing training instances does not necessarily mean faster training. This is due to the iterative nature of training a neural network: it completes when a threshold of accuracy is reached rather than when a single iteration has been completed. With this in mind, it is likely that the neural network is the most accurate model of the four created, but that time constraints limited the training epochs permitted in the tests previously conducted.

In contrast, Naïve Bayes was extremely fast, but with reduced accuracy. This would be a reasonable model for situations where the cost of errors is low, and the cost of delay is high. Similarly, Support Vector Machines were able to be trained for 10-digit classification in minutes. However, given the reduction of the number of examples, this may not hold for larger tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Support Vector Machine | K-Nearest Neighbors | Artificial Neural Networks | Naïve Bayes |
| Accuracy | 0.879 | 0.922 | 0.918 | 0.840 |
| Lower 95% CI | 0.859 | 0.917 | 0.916 | 0.833 |
| Upper 95% CI | 0.900 | 0.927 | 0.920 | 0.847 |
| Training Examples Used | 1,000 | 10,000 | 50,000 | 10,000 |

![A screenshot of a cell phone

Description automatically generated]()

In addition to training time, classification time is an important component. K-Nearest-Neighbors must calculate distance to each existing point, if some optimization is not used, which places all of the computational expense in the classification component: no training is truly necessary outside of identifying the best K value for a dataset. The other algorithms shift some of the expense of usage to training and reduce the cost of making classifications, giving them higher costs to train, but lower costs to classify new data. This is useful in applications where fast decisions need to be made.

# Bibliography

Guo, Q., Zhang, C., Zhang, Y., & Liu, H. (2016). An Efficient SVD-Based Method for Image Denoising. *IEEE Transactions on Circuits and Systems for Video Technology*, 868-880.

# Appendix: Code

﻿#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Created on Sat Nov 16 12:44:59 2019

@author: jmcleod

"""

import csv,random,math,decimal,copy, datetime

import numpy as np

from PIL import Image

decimal.getcontext().prec = 100

#from numpy import array

from sklearn.decomposition import TruncatedSVD

import matplotlib.pyplot as plt

import cvxopt

#%%

def data\_import(file):

'''

This function imports the data from a particular file

and returns an array of arrays

'''

data = []

with open(file, 'r') as csvfile:

csv\_r = csv.reader(csvfile,delimiter=' ')

for row in csv\_r:

row\_nums = []

for i in range(len(row)):

try:

val = float(row[i])

if i > 0:

val = round(val/255,4)

# The above line scales the data imported

row\_nums.append(val)

except:

print('ERROR on import: non-numerical data:',row[i])

break

data.append(row\_nums)

return(data)

def data\_import\_loop(string,denom):

'''This function loops the data import across all files of the chosen type,

which is specified by the string argument passed to the function.

It then uses the first value in the set to add the imported arrays

to the correct dictionary key, created with values 0-9.

The resulting dictionary is returned.

'''

files = []

results = []

data\_matrix = []

data\_dict = {}

for i in range(10):

file\_name = string+str(i)+'.txt'

files.append(file\_name)

data\_dict[i]=[]

for i in files:

data = data\_import(i)

for j in range(len(data)):

if j<denom: # SUBSET data

data\_dict[data[j][0]].append(data[j][1:])

data\_matrix.append(data[j][1:])

results.append(data[j][0:1][0])

return(data\_dict,results,data\_matrix)

def create\_image\_data(char\_matrix):

'''

This function outputs a human-viewable copy of an input from matrix form

'''

data = np.zeros( (len(char\_matrix),len(char\_matrix[0]),3), dtype=np.uint8 )

for row in range(len(char\_matrix)):

for col in range(len(char\_matrix[row])):

val = 255 - char\_matrix[col][row]

data[row,col] = [val,val,val]

return(data)

def create\_large\_image(data\_dict):

'''This function creates an NxN image of 10 examples of 10 classes'''

shortest = 1000000

for k,v in data\_dict.items():

if len(v) < shortest:

shortest = len(v)

big\_matrix\_data = []

for m in range(10):

medium\_matrix\_data = []

for i in range(28):

medium\_matrix\_data.append([])

for i in range(10):

random\_num = random.randint(0,shortest-1)

array = data\_dict[m][random\_num]

for j in range(len(array)):

medium\_matrix\_data[j%28].append((array[j]\*255))

for i in medium\_matrix\_data:

big\_matrix\_data.append(i)

big\_image = create\_image\_data(big\_matrix\_data)

image = Image.fromarray(big\_image)

image.show()

def subset\_data(matrix,array,digits,P,denom):

matrix\_out = []

array\_out = []

for i in digits:

bottom = int(denom\*i)

top = bottom + int(denom\*P)

matrix\_out = matrix\_out +matrix[bottom:top]

array\_out = array\_out +array[bottom:top]

return(matrix\_out,array\_out)

def subset\_test\_data(matrix,array,digits,P,denom):

matrix\_out = []

array\_out = []

for i in digits:

bottom = int(denom\*i)

top = bottom + int(denom\*P)

matrix\_out = matrix\_out +matrix[bottom:top]

array\_out = array\_out +array[bottom:top]

return(matrix\_out,array\_out)

def radial\_bias(v1,v2):

x = np.array(v1)-np.array(v2)

n = math.e\*\*(- 0.05 \*(np.linalg.norm(x)\*\*2))

return(n)

def m\_matrix(labels,data):

m = []

if len(data) != len(labels):

print("Error: data length mismatch")

return(0)

else:

for i in range(len(labels)):

row = []

for j in range(len(labels)):

yy = labels[i] \* labels[j]

#print(data[i])

#print(data[j])

k = radial\_bias(data[i],data[j])

row.append(yy\*k)

m.append(row)

#print(m)

return(m)

def calc\_b(y,alphas,petals):

alpha\_0 = max(alphas)

alpha\_0\_index = alphas.index(alpha\_0)

b = 0

alpha\_m = []

for i in range(len(alphas)):

if alphas[i]>0.001:

b += y[i] \* alphas[i] \* radial\_bias(petals[i],petals[alpha\_0\_index])

alpha\_m.append(petals[i])

b -= y[alpha\_0\_index]

return(b,alpha\_m)

def clf(ind,b,y,alphas,data,test\_data):

sum\_val = 0

for i in range(len(alphas)):

if alphas[i] > 0.001:

sum\_val += y[i] \* alphas[i] \* radial\_bias(test\_data[ind],data[i])

sum\_val -= b

return(sum\_val)

def y\_vector(input\_vec,target):

output = []

for i in input\_vec:

if i == target:

output.append(1.0)

else:

output.append(-1.0)

return(output)

def qp\_vars(data\_ys,C,m):

'''

This function constructs the Quadratic Program matrices and vectors

'''

q = []

b = []

G = []

h = []

A = []

P = cvxopt.matrix(m)

for i in range(len(data\_ys)):

q.append(-1.0)

b.append(0.0)

g\_row = []

for j in range(len(data\_ys)):

if i==j:

g\_row.append(-1.0)

else:

g\_row.append(0.0)

G.append(g\_row)

h.append(0.0)

for i in range(len(data\_ys)):

row = []

for j in range(len(data\_ys)):

if i ==j:

row.append(1.0)

else: row.append(0.0)

G.append(row)

h.append(C)

q = cvxopt.matrix(q)

b = cvxopt.matrix(0.0)

A = cvxopt.matrix(data\_ys)

G = cvxopt.matrix(G)

G = G.T

h = cvxopt.matrix(h)

b = cvxopt.matrix(b)

A = cvxopt.matrix(A)

A = A.T

#print(P)

return(P,q,G,h,A,b)

#%%

'''Data Import'''

denom = 500

data\_dict,results,data\_matrix = data\_import\_loop('train',denom)

denom = 500

test\_dict,test\_results,test\_data\_matrix = data\_import\_loop('test',denom)

'''Create sample image of data'''

#create\_large\_image(data\_dict)

#%%

'''This section allows subsetting training examples off'''

subset\_matrix,subset\_results = subset\_data(data\_matrix,results,(2,5),(0.5\*0.05),denom)

subset\_test\_matrix,subset\_test\_results = subset\_data(test\_data\_matrix,test\_results,(2,5),1,denom)

#%%

# construct a data matrix and then a vector of y vals

data\_m = copy.deepcopy(subset\_matrix)

data\_ys = []

for i in subset\_results:

if i==2:

data\_ys.append(1.0)

else:

data\_ys.append(-1.0)

test\_ys = y\_vector(subset\_test\_results,2)

C = 100

lambda\_var = 0.05

m = m\_matrix(data\_ys,data\_m)

m = np.matrix(m)

P,q,G,h,A,b = qp\_vars(data\_ys,C,m)

sol = cvxopt.solvers.qp(P,q,G,h,A,b)

alphas = []

for i in sol['x']:

alphas.append(i)

b,alpha\_m = calc\_b(data\_ys,alphas,data\_m)

# Actual\_classified

true\_true, true\_false, false\_true, false\_false = 0,0,0,0

for i in range(len(test\_ys)):

y = test\_ys[i]

classification = clf(i,b,data\_ys,alphas,subset\_matrix,subset\_test\_matrix)

if y >= 0:

if classification >= 0:

true\_true +=1

else:

true\_false +=1

else:

if classification >= 0:

false\_true +=1

else:

false\_false +=1

#print(test\_ys[i],classification)

print(' True False')

print('True ',true\_true,' ',true\_false)

print('False ',false\_true,' ',false\_false)

#%%

'''This section allows eliminating pixels in each training example'''

subset\_matrix,subset\_results = subset\_data(data\_matrix,results,(2,5),0.5,denom)

subset\_test\_matrix,subset\_test\_results = subset\_data(test\_data\_matrix,test\_results,(2,5),1,denom)

keep\_one\_in = 20

trimmed\_subset\_matrix = []

trimmed\_subset\_test\_matrix = []

for i in range(len(subset\_matrix)):

row = []

for j in range(len(subset\_matrix[i])):

if j%keep\_one\_in == 0:

row.append(subset\_matrix[i][j])

trimmed\_subset\_matrix.append(row)

for i in range(len(subset\_test\_matrix)):

row = []

for j in range(len(subset\_test\_matrix[i])):

if j%keep\_one\_in == 0:

row.append(subset\_test\_matrix[i][j])

trimmed\_subset\_test\_matrix.append(row)

subset\_matrix = trimmed\_subset\_matrix

subset\_test\_matrix = trimmed\_subset\_test\_matrix

# construct a data matrix and then a vector of y vals

data\_m = copy.deepcopy(subset\_matrix)

data\_ys = []

for i in subset\_results:

if i==2:

data\_ys.append(1.0)

else:

data\_ys.append(-1.0)

test\_ys = y\_vector(subset\_test\_results,2)

C = 100

lambda\_var = 0.05

m = m\_matrix(data\_ys,data\_m)

m = np.matrix(m)

P,q,G,h,A,b = qp\_vars(data\_ys,C,m)

sol = cvxopt.solvers.qp(P,q,G,h,A,b)

alphas = []

for i in sol['x']:

alphas.append(i)

b,alpha\_m = calc\_b(data\_ys,alphas,data\_m)

# Actual\_classified

true\_true, true\_false, false\_true, false\_false = 0,0,0,0

for i in range(len(test\_ys)):

y = test\_ys[i]

classification = clf(i,b,data\_ys,alphas,subset\_matrix,subset\_test\_matrix)

if y >= 0:

if classification >= 0:

true\_true +=1

else:

true\_false +=1

else:

if classification >= 0:

false\_true +=1

else:

false\_false +=1

#print(test\_ys[i],classification)

print(' True False')

print('True ',true\_true,' ',true\_false)

print('False ',false\_true,' ',false\_false)

#%%

'''This section applies SVD'''

subset\_matrix,subset\_results = subset\_data(data\_matrix,results,(2,5),(0.5),denom)

subset\_test\_matrix,subset\_test\_results = subset\_data(test\_data\_matrix,test\_results,(2,5),1,denom)

# svd component here:

svd = TruncatedSVD(n\_components=(int(784/20)))

svd.fit(subset\_matrix)

result = svd.transform(subset\_matrix)

subset\_matrix = result

svd.fit(subset\_test\_matrix)

result = svd.transform(subset\_test\_matrix)

subset\_test\_matrix = result

# construct a data matrix and then a vector of y vals

data\_m = copy.deepcopy(subset\_matrix)

data\_ys = []

for i in subset\_results:

if i==2:

data\_ys.append(1.0)

else:

data\_ys.append(-1.0)

test\_ys = y\_vector(subset\_test\_results,2)

C = 100

lambda\_var = 0.05

m = m\_matrix(data\_ys,data\_m)

m = np.matrix(m)

P,q,G,h,A,b = qp\_vars(data\_ys,C,m)

sol = cvxopt.solvers.qp(P,q,G,h,A,b)

alphas = []

for i in sol['x']:

alphas.append(i)

b,alpha\_m = calc\_b(data\_ys,alphas,data\_m)

# Actual\_classified

true\_true, true\_false, false\_true, false\_false = 0,0,0,0

for i in range(len(test\_ys)):

y = test\_ys[i]

classification = clf(i,b,data\_ys,alphas,subset\_matrix,subset\_test\_matrix)

if y >= 0:

if classification >= 0:

true\_true +=1

else:

true\_false +=1

else:

if classification >= 0:

false\_true +=1

else:

false\_false +=1

#print(test\_ys[i],classification)

print(' True False')

print('True ',true\_true,' ',true\_false)

print('False ',false\_true,' ',false\_false)

#%%

'''This section compares even and odd numbers'''

denom = 500

subset\_matrix,subset\_results = subset\_data(data\_matrix,results,(0,1,2,3,4,5,6,7,8,9),.2,denom)

subset\_test\_matrix,subset\_test\_results = subset\_data(test\_data\_matrix,test\_results,(0,1,2,3,4,5,6,7,8,9),.2,denom)

def modified\_y\_vector(input\_vec,target):

output = []

for i in input\_vec:

if i in target:

output.append(1.0)

else:

output.append(-1.0)

return(output)

# construct a data matrix and then a vector of y vals

data\_ys = modified\_y\_vector(subset\_results,[0,2,4,6,8])

test\_ys = modified\_y\_vector(subset\_test\_results,[0,2,4,6,8])

#%%

data\_m = copy.deepcopy(subset\_matrix)

C = 100

lambda\_var = 0.05

m = m\_matrix(data\_ys,data\_m)

m = np.matrix(m)

P,q,G,h,A,b = qp\_vars(data\_ys,C,m)

sol = cvxopt.solvers.qp(P,q,G,h,A,b)

alphas = []

for i in sol['x']:

alphas.append(i)

b,alpha\_m = calc\_b(data\_ys,alphas,data\_m)

# Actual\_classified

true\_true, true\_false, false\_true, false\_false = 0,0,0,0

for i in range(len(test\_ys)):

y = test\_ys[i]

classification = clf(i,b,data\_ys,alphas,subset\_matrix,subset\_test\_matrix)

if y >= 0:

if classification >= 0:

true\_true +=1

else:

true\_false +=1

else:

if classification >= 0:

false\_true +=1

else:

false\_false +=1

#print(test\_ys[i],classification)

print(' True False')

print('True ',true\_true,' ',true\_false)

print('False ',false\_true,' ',false\_false)

#%%

'''This section compares all different digits and classifies them'''

denom = denom

subset\_matrix,subset\_results = subset\_data(data\_matrix,\

results,\

(0,1,2,3,4,5,6,7,8,9),0.2,denom)

subset\_test\_matrix,subset\_test\_results = subset\_data(test\_data\_matrix,\

test\_results,\

(0,1,2,3,4,5,6,7,8,9),\

0.2,denom)

#%%

def modified\_y\_vector(input\_vec,target):

output = []

try:

for i in input\_vec:

if i in target:

output.append(1.0)

else:

output.append(-1.0)

except:

for i in input\_vec:

if i == target:

output.append(1.0)

else:

output.append(-1.0)

return(output)

# construct a data matrix and then a vector of y vals

data\_y\_0 = modified\_y\_vector(subset\_results,0)

data\_y\_1 = modified\_y\_vector(subset\_results,1)

data\_y\_2 = modified\_y\_vector(subset\_results,2)

data\_y\_3 = modified\_y\_vector(subset\_results,3)

data\_y\_4 = modified\_y\_vector(subset\_results,4)

data\_y\_5 = modified\_y\_vector(subset\_results,5)

data\_y\_6 = modified\_y\_vector(subset\_results,6)

data\_y\_7 = modified\_y\_vector(subset\_results,7)

data\_y\_8 = modified\_y\_vector(subset\_results,8)

data\_y\_9 = modified\_y\_vector(subset\_results,9)

data\_y\_vals = [data\_y\_0,data\_y\_1,data\_y\_2,data\_y\_3,\

data\_y\_4,data\_y\_5,data\_y\_6,data\_y\_7,\

data\_y\_8,data\_y\_9]

test\_y\_0 = modified\_y\_vector(subset\_test\_results,0)

test\_y\_1 = modified\_y\_vector(subset\_test\_results,1)

test\_y\_2 = modified\_y\_vector(subset\_test\_results,2)

test\_y\_3 = modified\_y\_vector(subset\_test\_results,3)

test\_y\_4 = modified\_y\_vector(subset\_test\_results,4)

test\_y\_5 = modified\_y\_vector(subset\_test\_results,5)

test\_y\_6 = modified\_y\_vector(subset\_test\_results,6)

test\_y\_7 = modified\_y\_vector(subset\_test\_results,7)

test\_y\_8 = modified\_y\_vector(subset\_test\_results,8)

test\_y\_9 = modified\_y\_vector(subset\_test\_results,9)

test\_y\_vals = [test\_y\_0,test\_y\_1,test\_y\_2,test\_y\_3,\

test\_y\_4,test\_y\_5,test\_y\_6,test\_y\_7,\

test\_y\_8,test\_y\_9]

data\_m = copy.deepcopy(subset\_matrix)

#%%

def get\_b\_alphas(data\_ys,data\_m):

C = 100

m = m\_matrix(data\_ys,data\_m)

m = np.matrix(m)

P,q,G,h,A,b = qp\_vars(data\_ys,C,m)

sol = cvxopt.solvers.qp(P,q,G,h,A,b)

alphas = []

for i in sol['x']:

alphas.append(i)

b,alpha\_m = calc\_b(data\_ys,alphas,data\_m)

return(b,alphas)

b\_alpha\_dict = {}

for i in range(0,10):

b,alpha = get\_b\_alphas(data\_y\_vals[i],data\_m)

b\_alpha\_dict[i] = [b,alpha]

#%%

confusion = []

for i in range(10):

row = [0]\*10

confusion.append(row)

#%%

all\_classifications = []

# Actual\_classified

true\_true, true\_false, false\_true, false\_false = 0,0,0,0

for i in range(len(test\_ys)):

y = test\_ys[i]

classifications = []

for k,v in sorted(b\_alpha\_dict.items()):

classification = clf(i,b\_alpha\_dict[k][0],data\_y\_vals[k],b\_alpha\_dict[k][1],subset\_matrix,subset\_test\_matrix)

classifications.append(classification)

classed = classifications.index(max(classifications))

#print(subset\_test\_results[i],classed)

all\_classifications.append(classifications)

confusion[int(subset\_test\_results[i])][classed] +=1

#%%

good =0

bad =0

for i in range(len(all\_classifications)):

classed = all\_classifications[i].index(max(all\_classifications[i]))

if classed == subset\_test\_results[i]:

good+=1

else:

bad+=1

confusion[int(subset\_test\_results[i])][int(classed)] +=1

print(good,bad)

for i in confusion:

print(i)