### Problem.1

Part a)

Notice that kernel can be handily integrated into the dual perceptron algorithm. Given the normal dual perceptron algorithm:

2. Repeat
3. For
4. If
6. EndIf
7. EndFor
8. Until k iterations has reached
9. Output

We realize that we could modify the dot product to use the kernel function. In particular, the inequality in line 4 could be changed to:

Then, we have the perceptron training algorithm with kernel:

2. Repeat
3. For
4. If
6. EndIf
7. EndFor
8. Until k iterations has reached
9. Output

To apply the trained model , and given , we simply calculate , and classify the sample as +1 if , and -1 if .

Part b)

We have the following table for accuracy vs. digit & c value:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 100 | 97.71 | 99.22 | 99.48 | 99.2 | 99.74 | 99.74 | 98.93 | 94.75 | 96.83 |
| 0.0005 | 100 | 97.71 | 99.22 | 99.22 | 99.2 | 100 | 100 | 99.2 | 95.28 | 97.88 |
| 0.001 | 100 | 97.71 | 99.22 | 98.96 | 99.2 | 100 | 100 | 99.2 | 95.8 | 98.41 |
| 0.005 | 100 | 97.96 | 99.22 | 98.96 | 99.46 | 100 | 100 | 99.2 | 95.54 | 98.15 |
| 0.01 | 100 | 97.96 | 99.22 | 98.96 | 99.73 | 100 | 100 | 98.93 | 95.54 | 97.88 |
| 0.05 | 100 | 97.71 | 98.96 | 98.96 | 99.46 | 99.48 | 99.48 | 99.2 | 95.54 | 98.15 |
| 0.1 | 100 | 97.46 | 98.96 | 98.96 | 98.93 | 100 | 99.22 | 99.2 | 96.06 | 98.15 |

From the table, the best model (value of C) we picked was: (for each digit)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 0.005 | 0.0001 | 0.0001 | 0.01 | 0.0005 | 0.0005 | 0.0005 | 0.1 | 0.001 |

We chose the C values using the following method: We first find the C that maximizes accuracy on the validation set, and we broke ties by choosing the one with smallest C (this will hopefully reduce overfitting).

With this optimal model, we have the following results with the test set:

Accuracy rate is 1582/1797 = 0.880356

Type A error rate is 171/1797 = 0.095159 (this is when best classification is negative)

Type B error rate is 44/1797 = 0.024485 (this is when classification is positive, but classification is incorrect)

Sample execution commands:

|  |
| --- |
| # Train the models and print the accuracy for each c value.  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  for j in range(0,10):  os.mkdir('D:\\Documents\\Dropbox\\MLProj\\p4\\digits\\model\\'+str(j)+"\\")  os.mkdir('D:\\Documents\\Dropbox\\MLProj\\p4\\digits\\out\\'+str(j)+"\\")  stuff = re.compile(".\*(Accuracy on test set: )(.\*)% \\(.\*")  for i in (0.0001,0.0005,0.001,0.005,0.01,0.05,0.1):  for j in range(0, 10):  i = str(i)  j = str(j)  s1 = subprocess.check\_output('svm\_learn.exe -c '+i+' "D:\Documents\Dropbox\MLProj\p4\digits\digits'+j+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\model/'+j+'/digitsmod'+j+'\_'+i+'.train"')  s2 = subprocess.check\_output('svm\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits'+j+'.val" "D:\Documents\Dropbox\MLProj\p4\digits\model/'+j+'/digitsmod'+j+'\_'+i+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\out/'+j+'/digitsout'+j+'\_'+i+'.val"')  stemp = s2.split('\n')  for s in stemp:  res = stuff.match(s)  if res:  sys.stdout.write("%s " % res.group(2))  break  sys.stdout.write("\n")  # Classification for 10 digits  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  carray = [0.0001,0.0005,0.001,0.005,0.01,0.05,0.1];  bestmodel = [0, 3, 0, 0, 4, 1, 1, 1, 6, 2];  for j in range(0, 10):  c = str(carray[bestmodel[j]])  j = str(j)  s2 = subprocess.check\_output('svm\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits.test" "D:\Documents\Dropbox\MLProj\p4\digits\model/'+j+'/digitsmod'+j+'\_'+c+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\out\digitsout'+j+'.test"')  # Calculate the accuracy and errors.  [valuemax, indexmax] = max(out, [], 2);  error\_a = 0;  error\_b = 0;  for i = 1 : leng  if valuemax(i) < 0  error\_a = error\_a + 1;  elseif indexmax(i) ~= digitsout(i)  error\_b = error\_b + 1;  end  end  error = error\_a + error\_b; |

Part c)

Again, we have the following table for accuracy vs. digit & c value, for each d value:

d = 2:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.0005 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.001 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.005 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.01 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.05 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |
| 0.1 | 100 | 99.75 | 100 | 98.96 | 99.46 | 100 | 99.74 | 99.46 | 98.16 | 99.47 |

d = 3:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.0005 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.001 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.005 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.01 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.05 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |
| 0.1 | 100 | 99.75 | 100 | 99.22 | 99.73 | 99.74 | 100 | 99.46 | 98.43 | 99.47 |

d = 4:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.0005 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.001 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.005 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.01 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.05 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |
| 0.1 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.47 |

d = 5:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0.0001 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.0005 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.001 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.005 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.01 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.05 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |
| 0.1 | 100 | 99.75 | 100 | 99.22 | 100 | 100 | 100 | 99.46 | 98.69 | 99.21 |

We picked d = 4 for the maximum sum of accuracy across all digits. With that model, we have the following results on test set:

Accuracy rate is 1730/1797=0.962716

Type A error rate is 50/1797=0.027824 (this is when best classification is negative)

Type B error rate is 17/1797=0.009460 (this is when classification is positive, but classification is incorrect)

Sample execution commands:

|  |
| --- |
| # Look over d to calculate performance for each C value for all digits  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  for j in range(0,10):  os.mkdir('D:\\Documents\\Dropbox\\MLProj\\p4\\digits\\polymodel\\'+str(j)+"\\")  os.mkdir('D:\\Documents\\Dropbox\\MLProj\\p4\\digits\\polyout\\'+str(j)+"\\")  stuff = re.compile(".\*(Accuracy on test set: )(.\*)% \\(.\*")  for d in (2, 3, 4, 5):  d = str(d)  sys.stdout.write("\nd = " + d + ":\n");  for i in (0.0001,0.0005,0.001,0.005,0.01,0.05,0.1):  for j in range(0, 10):  i = str(i)  j = str(j)  s1 = subprocess.check\_output('svm\_learn.exe -c '+i+' -t 1 -d '+d+' "D:\Documents\Dropbox\MLProj\p4\digits\digits'+j+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\polymodel/'+j+'/digitsmod'+j+'\_'+d+'\_'+i+'.train"')  s2 = subprocess.check\_output('svm\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits'+j+'.val" "D:\Documents\Dropbox\MLProj\p4\digits\polymodel/'+j+'/digitsmod'+j+'\_'+d+'\_'+i+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\polyout/'+j+'/digitsout'+j+'\_'+d+'\_'+i+'.val"')  stemp = s2.split('\n')  for s in stemp:  res = stuff.match(s)  if res:  sys.stdout.write("%s " % res.group(2))  break  sys.stdout.write("\n")  # Apply the model and calculate classify the test set  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  d = '4'  bestmodel = [0, 3, 0, 0, 4, 1, 1, 1, 6, 2];  carray = [0.0001,0.0005,0.001,0.005,0.01,0.05,0.1];  for j in range(0, 10):  c = str(carray[bestmodel[j]])  j = str(j)  s2 = subprocess.check\_output('svm\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits.test" "D:\Documents\Dropbox\MLProj\p4\digits\polymodel/'+j+'/digitsmod'+j+'\_'+d+'\_'+c+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\polyout\digitsout'+j+'.test"')  # Print the error rate  [valuemax, indexmax] = max(polyout, [], 2);  error\_a = 0;  error\_b = 0;  for i = 1 : leng  if valuemax(i) < 0  error\_a = error\_a + 1;  elseif indexmax(i) ~= digitsout(i)  error\_b = error\_b + 1;  end  end  error = error\_a + error\_b; |

Part d)

The output results for each C values:

|  |  |
| --- | --- |
| C | Zero/one-error on test set |
| 0.00001 | 27.68 |
| 0.00005 | 67.10 |
| 0.0001 | 67.10 |
| 0.0005 | 58.75 |
| 0.001 | 56.40 |
| 0.005 | 32.11 |
| 0.01 | 15.67 |
| 0.05 | 13.58 |
| 0.1 | 12.01 |
| 0.5 | 9.92 |
| 1 | 10.70 |
| 5 | 7.05 |

With that data, we chose C = 5, and the results of SVM on test:

|  |
| --- |
| Reading model...done.  Reading test examples... (1797 examples) done.  Classifying test examples...done  Runtime (without IO) in cpu-seconds: 0.01  Average loss on test set: 7.7908  Zero/one-error on test set: 7.79% (1657 correct, 140 incorrect, 1797 total) |

Sample Commands:

|  |
| --- |
| # Loop over all C to choose the one with best performance  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  for i in ("0.00001", "0.00005", "0.0001", "0.0005", "0.001", "0.005", "0.01", "0.05", "0.1", "0.5", "1", "5"):  s1 = subprocess.check\_output('svm\_multiclass\_learn.exe -c '+i+' "D:\Documents\Dropbox\MLProj\p4\digits\digits.train" "D:\Documents\Dropbox\MLProj\p4\digits\multimodel/digitsmod\_c'+i+'.train"')  s2 = subprocess.check\_output('svm\_multiclass\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits.val" "D:\Documents\Dropbox\MLProj\p4\digits\multimodel/digitsmod\_c'+i+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\multiout/digitsout\_c'+i+'.val"')  sys.stdout.write("%s\n" % s2)  # Apply the model and run test on test set  import os, shutil, subprocess, sys, re;  os.chdir('D:\\Documents\\Dropbox\\MLProj\\p4\\')  best\_c = "5";  s2 = subprocess.check\_output('svm\_multiclass\_classify.exe "D:\Documents\Dropbox\MLProj\p4\digits\digits.test" "D:\Documents\Dropbox\MLProj\p4\digits\multimodel/digitsmod\_c'+best\_c+'.train" "D:\Documents\Dropbox\MLProj\p4\digits\multiout/digitsout\_c'+best\_c+'.val"')  sys.stdout.write("%s\n" % s2) |

### Problem.2

Part a)

We realize the equivalence that , then we have:

Then we have the linear format that we want, where:

(In the proof, we used without proof the property: , which could be easily showed by expanding the right hand side of the equality. )

Part b)

Again, we realize the equivalence that:

.

We also notice that we could the product, , to the product of a different base, , where denotes the whole vocabulary( is the word in the vocabulary), and denotes the number of occurrences of in . We see that:

This is because for any word in , we multiply to the total result for each occurrence. This is equivalent to multiplying to the power of the total occurrence to the result.

Then, we would denote the representation of input as , where denote the number of occurrence of word in the input . The above equation becomes:

Continue our equivalence relation, we have:

We again realize that the first term is simply the expanded version of a dot product. We could express this as a linear combination where we define as , and .

Part c)

We can firstly construct the training set as follows:

And

Now, say we want to classify .

Bayes optimal labeling would label it as 1, since:

And

Naïve Bayes labeling would label it as -1, since:

And

Thus we find a sample where the two prediction methods give different results.

### Problem.3

Part a)

For normalized data:

With default value of C:

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.01  Accuracy on test set: 96.81% (31451 correct, 1036 incorrect, 32487 total)  Precision/recall on test set: 95.92%/90.17%  false positive rate is 291/32487=0.008957  false\_negative rate is 745/32487=0.022932 |

When trying to change C:

Table for 4-fold validation accuracy:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0.001 | 0.01 | 0.1 | 1 |
| Seg 1 as validation | 77.14 | 81.50 | 95.77 | 97.30 |
| Seg 2 as validation | 76.43 | 80.77 | 95.46 | 96.64 |
| Seg 3 as validation | 75.50 | 79.67 | 95.37 | 96.72 |
| Seg 4 as validation | 76.70 | 81.08 | 95.44 | 97.03 |

C=1 is used in the best model since this is when the accuracy on the validation set is maximized

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.03  Accuracy on test set: 96.81% (31451 correct, 1036 incorrect, 32487 total)  Precision/recall on test set: 95.92%/90.17%  false positive rate is 291/32487=0.008957  false\_negative rate is 745/32487=0.022932 |

For unnormalized data:

With default C value:

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.01  Accuracy on test set: 96.08% (31214 correct, 1273 incorrect, 32487 total)  Precision/recall on test set: 96.66%/86.19%  false positive rate is 226/32487=0.006957  false\_negative rate is 1047/32487=0.032228 |

When trying to change C:

Table for 4-fold validation accuracy:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0.001 | 0.01 | 0.1 | 1 |
| Seg 1 as validation | 95.32 | 96.71 | 95.90 | 95.20 |
| Seg 2 as validation | 95.16 | 96.48 | 95.88 | 95.45 |
| Seg 3 as validation | 94.87 | 96.39 | 95.62 | 95.21 |
| Seg 4 as validation | 95.02 | 96.55 | 95.74 | 95.36 |

C=0.01 is used in the best model since this is when teh accuracy on teh validation set is maximized

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.02  Accuracy on test set: 96.43% (31327 correct, 1160 incorrect, 32487 total)  Precision/recall on test set: 95.91%/88.47%  false positive rate is 286/32487=0.008804  false\_negative rate is 874/32487=0.026903 |

It seems like that the normalized data set is slightly better from the results that we get. However, more notably, it seems that for the normalized data set, changing C makes a much bigger difference than it does for unnormalized dataset.

Part b)

Using the attached program, we are able to reach the accuracy of 96.04%, with 394(1.21%) false positives, and 892(2.75%) false negatives.

Part c)

With the new cost sensitive classifier, we are able to reach the accuracy of 96.01%, with 467(1.44%) false positives, and 829(2.55%) false negatives. Intuitively, if , that is, if we penalize false negative more than false positive, we should see less false negative errors being made. On the other hand, if this ratio is smaller than 1, we penalize false positive more, then we should see less false positive errors being made. If the ratio is 1, we are effectively minimizing the overall error rate.

Part d)

With default C value:

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.02  Accuracy on test set: 96.21% (31257 correct, 1230 incorrect, 32487 total)  Precision/recall on test set: 91.10%/92.85%  false positive rate is 688/32487=0.021178  false\_negative rate is 542/32487=0.016684 |

When trying to change C:

Table for 4-fold validation accuracy:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0.001 | 0.01 | 0.1 | 1 |
| Seg 1 as validation | 23.25 | 78.68 | 95.45 | 96.12 |
| Seg 2 as validation | 23.56 | 78.31 | 95.49 | 96.53 |
| Seg 3 as validation | 23.53 | 78.85 | 95.30 | 96.43 |
| Seg 4 as validation | 23.47 | 78.00 | 95.20 | 96.28 |

False positive rate for each C value:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0.001 | 0.01 | 0.1 | 1 |
| Seg 1 as validation | 5735/7472=0.767532 | 5735/7472=0.767532 | 5735/7472=0.767532 | 5735/7472=0.767532 |
| Seg 2 as validation | 5712/7473=0.764352 | 5712/7473=0.764352 | 5712/7473=0.764352 | 5712/7473=0.764352 |
| Seg 3 as validation | 5714/7472=0.764722 | 1562/7472=0.209047 | 285/7472=0.038142 | 132/7472=0.017666 |
| Seg 4 as validation | 5719/7473=0.765288 | 1626/7473=0.217583 | 292/7473=0.039074 | 145/7473=0.019403 |

False negative rate for each C value:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0.001 | 0.01 | 0.1 | 1 |
| Seg 1 as validation | 0/7472=0.000000 | 0/7472=0.000000 | 0/7472=0.000000 | 0/7472=0.000000 |
| Seg 2 as validation | 0/7473=0.000000 | 24/7473=0.003212 | 69/7473=0.009233 | 123/7473=0.016459 |
| Seg 3 as validation | 0/7472=0.000000 | 18/7472=0.002409 | 66/7472=0.008833 | 135/7472=0.018067 |
| Seg 4 as validation | 0/7473=0.000000 | 18/7473=0.002409 | 67/7473=0.008966 | 133/7473=0.017797 |

During cross validation, we are minimizing the value: false\_positive\_rate + false\_negative\_rate \* 10

This is because we assume that having a single false negative is 10 times as bad as having a false positive.

The best C is 0.1 since this minimizes (false\_positive\_rate + false\_negative\_rate \* 10) across all runs.

Run results:

|  |
| --- |
| Runtime (without IO) in cpu-seconds: 0.02  Accuracy on test set: 95.17% (30919 correct, 1568 incorrect, 32487 total)  Precision/recall on test set: 85.04%/96.24%  false positive rate is 1283/32487=0.039493  false\_negative rate is 285/32487=0.008773 |

Assuming having a single false negative is 10 times as bad as having a false positive, we let penalty = 10 \* false\_negative\_rate + false\_positive\_rate.

Before adjusting C, we have a penalty of 0.021178 + 10(0.016684) = 0.188018.

After adjusting C, we have a penalty of 0.039493 + 10(0.008773) = 0.127223

So, the results are better after adjusting C.

Furthermore, before introducing the j parameter, we have a penalty of 0.008957 + 10(0.022932) = 0.238277

So the j parameter definitely helps to get what we want in the case when false\_positive\_rate and false\_negative\_rate are weighed differently

Part e)

Advantages of Naive Bayes Classifier:

1. Much easier to implement than SVM
2. Training is more efficient with Naive Bayes
3. It only needs a small training set to estimate the parameters such as the mean, which are used for classification.
4. Furthermore, if the Naive Bayes assumption actually holds (i.e. features are independent), then Naive Bayes will converge quickly
5. You can handle multiclass quite easier

Advantages of SVM:

1. Prediction is more accurate. Furthermore, it is more stable and robust.
2. There are theoretical guarantees (i.e. against overfitting)
3. You can change the kernel very easily to adjust to the structure of the data (i.e. high-dimensional space)