CSCE 633 - Machine Learning HW2

Robert Quan
Department of Computer Science, Texas A&M University, College Station, Texas 77843-4242, USA
(Dated: October 23, 2017)

I. ABSTRACT

In this homework, I experiment with the creation of Decision Trees, Support Vector Machines and Kernels mathematics.

QUESTION I

Decision Tree

a) The equation to find the entropy of any given branch, the process is to first choose which feature will have the lowest amount of entropy from our calculation. The equation for entropy is the following:

$$H(x) = -\sum p(x) log_2(p(x)) \tag{1}$$

By choosing the feature with the least amount of entropy, we can then say that group with have the maximum amount of discrimination for our values. By hand performing my calculations, I discovered that the first group that will spli the data in half should be the "Sky Condition" feature.

The calculations are below for the "Cloudy" and "Clear" branches respectively:

$$H_{Cloudy} = -(\frac{25}{40} log_2(\frac{25}{40}) + \frac{15}{40} log_2(\frac{15}{40})) = 0.9544 \tag{2a}$$

$$H_{Clear} = -(\frac{11}{40} log_2(\frac{11}{40}) + \frac{29}{40} log_2(\frac{29}{40})) = 0.8485 \tag{2b}$$

Now we calculate the Outcome entropy

$$H(Rainy|SkyCond) = (\frac{40}{80} * 0.9544) \tag{3a}$$

$$+(\frac{40}{80} * 0.8485) = \boxed{0.9014} \tag{3b}$$

This was the **Lowest** entropy that I found when computing the values for all three features. The Entropy for the "Humid" class was: **0.9457** and for the "Hot class was **0.9457**.

Based on this first split, now I have halved the values and I retry each entropy euation now using the features based on "Humid" and "Hot". For the "Cloudy" branch, I discovered the the "Humid" class had a lower entropy than the "Hot" class. The calculation is below:

$$H_{High} = -(\frac{16}{20} log_2(\frac{16}{20}) + \frac{4}{20} log_2(\frac{4}{20})) = 0.7219 \tag{4a}$$

$$H_{Low} = -(\frac{9}{20} log_2(\frac{9}{20}) + \frac{11}{20} log_2(\frac{11}{20})) = 0.9927 \tag{4b}$$

$$H(Rainy|Humid|"Cloudy") = (\frac{20}{40} * 0.7218) \tag{5a}$$

$$+(\frac{20}{40} * 0.9927) = \boxed{0.855} \tag{5b}$$

At this point, we have reached the final and last class for the branch of Cloudy → High → Temp. Since there are not more features to split on, we take the highest number of outcomes and if it is higher than the average from the group, we will classify the ending node as either the decision boundary whether to run or not.

By using this method and calculating the different partial entropy for each branch, I obtain the following figure:

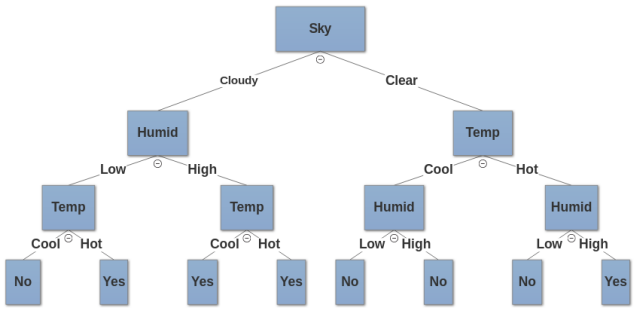


FIG. 1: Decision tree for problem 1 a

a larger image is included below

b) To prove that for any discrete probability distribution p with K classes, the Gini index is less than or equal to the corresponding value of the entropy we will see:

$$\sum_{k=1}^{K} p_k(1 - p_k) \leq -\sum_{k=1}^{k} p_k log_2(p_k) \tag{6}$$

we can set a relationship as:

$$1 - p_k - log_2(p_k) \leq 0 \tag{7}$$

We know that for any probability distribution, our values of the probability will be between [0,1]. If we set a function as:

$$f(p_k) = 1 - p_k + log_2(p_k) \leq 0 \tag{8}$$

then find its first derivative and set it at the maximum point:

$$f'(p_k = 1) = -1 - \frac{1}{1 * ln(2)} = 0.4426 \tag{9}$$

the first derivative is positive, so the function is monotonically increasing, now if we set our function at the maximum probability:

$$f(p_k = 1) = 1 - 1 - log_2(1) = 0 \tag{10}$$

Thus, we have show than the Gini Index is less than or equal to the Cross-entropy for any discrete probability distribution with K classes.

c) When I compute the number of samples for the benign I found **444 Benign** and **239 Malignant**. The two classes are not equally represented in the data, there is about a **65%** chance that the sample is Bening and a **35%** that it is malignant. **The rest of the problem is included in the pages below.**

QUESTION II

a) The most elementary algorithm that can be kernelized is ridge regression. By minimizing the quadratic cost function, we can find a closed for solution. However, if we work in feature space and replace our $\boldsymbol{x_i} -> \boldsymbol{\Phi(x_i)}$ there is a danger that we over-fit. Thus, we require the use of regularization.

$$J(w) = \sum_i (y_i - \boldsymbol{w}^T \boldsymbol{\Phi(x_i)})^2 + \lambda ||\boldsymbol{w}||^2 \tag{11a}$$

$$\frac{\partial J(w)}{\partial \boldsymbol{w}} = \sum_i (y_i - \boldsymbol{w}^T \boldsymbol{\Phi(x_i)})\boldsymbol{\Phi(x_i)} - \lambda \boldsymbol{w} = 0 \tag{11b}$$

$$\boldsymbol{w^*} = (\lambda \boldsymbol{I} + \sum_i \boldsymbol{\Phi(x_i)\Phi(x_i)}^T)^{-1}(\sum_j y_j \boldsymbol{\Phi(x_j)}) \tag{11c}$$

By mapping each feature to a nonlinear basis, the number of dimension can be much higher or even infinitely higher than the number of data-cases. There is an identity that we can use to help our derivation.

$$(\boldsymbol{P}^{-1} + \boldsymbol{B}^T \boldsymbol{R}^{-1} \boldsymbol{B})^{-1} \boldsymbol{B}^T \boldsymbol{R}^{-1} = \boldsymbol{P}\boldsymbol{B}^T (\boldsymbol{B}\boldsymbol{P}\boldsymbol{B}^T + \boldsymbol{R})^{-1} \tag{12}$$

we now obtain:

$$\boldsymbol{w^*} = (\lambda \boldsymbol{I_N} + \Phi\Phi^T)^{-1}\Phi y \tag{13a}$$

$$\boldsymbol{w^*} = \Phi(\Phi^T\Phi + \lambda \boldsymbol{I_N})^{-1} y \tag{13b}$$

which can be rewritten as:

$$\boxed{\boldsymbol{w^*} = \boldsymbol{\Phi}^T(\boldsymbol{\Phi}\boldsymbol{\Phi}^T + \lambda \boldsymbol{I_N})^{-1}\boldsymbol{y}} \tag{14}$$

b) By using the result above, we can now rewrite

$$y = \boldsymbol{w}^T \Phi(x) \tag{15}$$

as the following:

$$y = (\boldsymbol{y}(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \lambda \boldsymbol{I_N})^{-1})\boldsymbol{\Phi}^T)^T \boldsymbol{\Phi(x)} \tag{16}$$

which can be represented as:

$$\boxed{y = \boldsymbol{y}^T(\boldsymbol{K} + \lambda \boldsymbol{I_N})^{-1}\kappa(\boldsymbol{x})} \tag{17}$$

where $\kappa(\boldsymbol{x}) = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x})$ and where $\boldsymbol{K_{ij}} = \Phi^T\Phi$. The importance of this result is that we never actually need to access the feature vectors, we only need to access the kernel $\boldsymbol{K}$.

c) **Bonus:** The implementation requires solving a linear equation of size n. So the time complexity is for the Kernelized model

$$\boxed{O(N^3)} \tag{18}$$

and from class we know that the time complexity for solving the linear ridge regression is:

$$\boxed{O(D^3 + D^2N)} \tag{19}$$

The time complexity for the Kernalized Ridge Regression is harder for big data. Ultimately we come down to a trade-off of efficiency and flexibility.

Also for prediction we have

$$O(D) \tag{20a}$$

$$O(N) \tag{20b}$$

For predictions from the linear regression and kenrnel regression respectively.

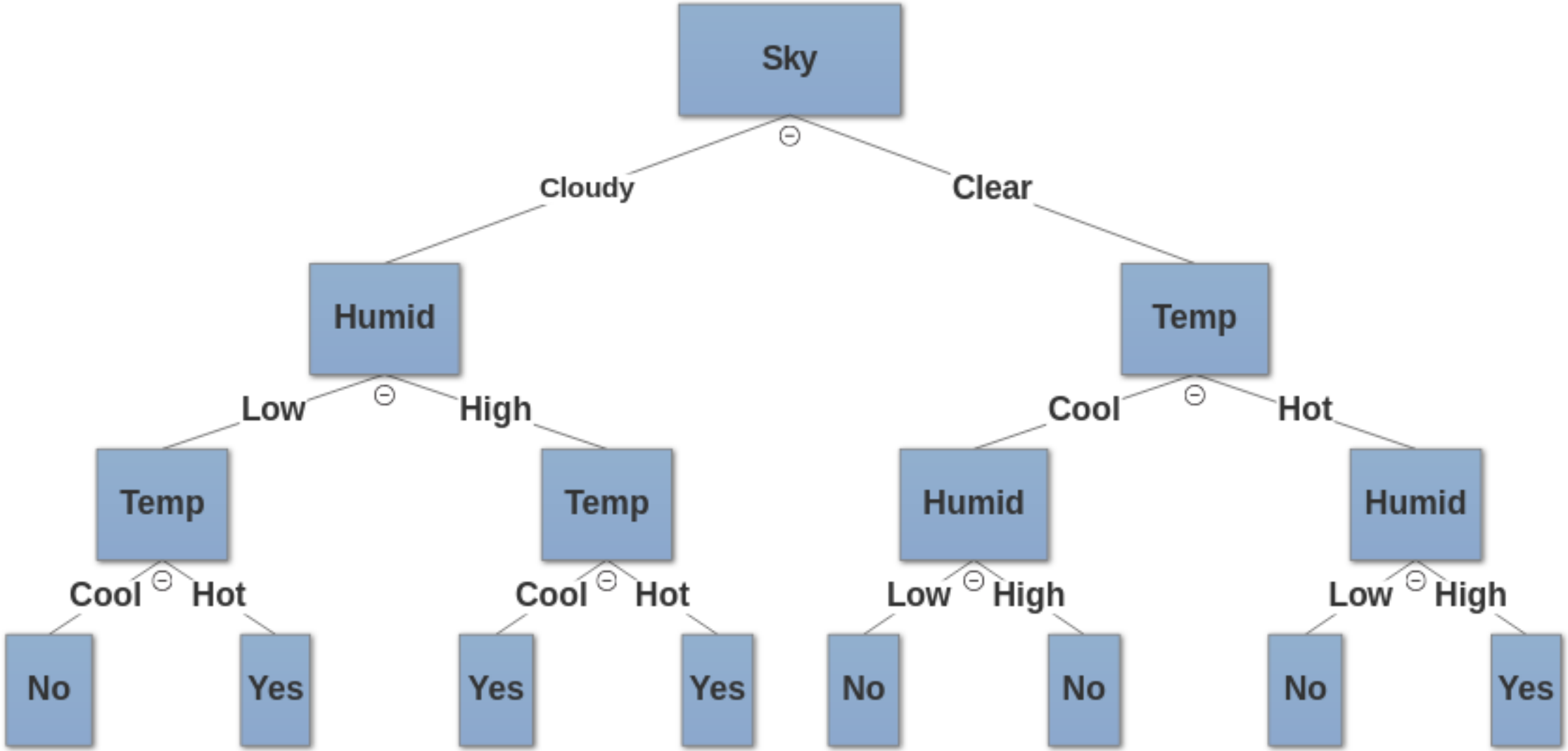
QUESTION III

a) By using pandas and scikit learn, I have replaced the columns which contained the triplet values with new columns that are:

$$\begin{vmatrix} f \\ 1 \\ 0 \\ -1 \end{vmatrix} = \begin{vmatrix} a & b & c \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix}$$

So for column 2 (index starting at 1), I obtain two new columns and then replace them with 2a,2b,2c and delete the original triplet column in the table.

I have included the code below with a writeup along with the code to follow the procedure.


Sky
Cloudy
Clear
Humid
Temp
Low
High
Cool
Hot
Temp
Temp
Humid
Humid
Cool
Hot
Cool
Hot
Low
High
Low
High
No
Yes
Yes
Yes
No
No
No
Yes

SVM Hw2 part A Data Preprocessing

October 23, 2017

1 Support Vector Machines using data from Phising Websites Data Set

```
In [2]: import pandas as pd
        from sklearn.model_selection import train_test_split
        from svmutil import *
        import matplotlib.pyplot as plt
        from time import time
```

Import pandas, when reading the csv sometimes we dont have any headers, if we choose the option header=None, now we have a 0:len scale for the columns.

```
In [3]: df = pd.read_csv("hw2_question3.csv", header=None)
```

This next function will reorder the data fram by inserting three new columns on a specific list of columns and it will delete the old data frame column.

```
In [4]: def reorder(df, index):
            df[str(index)+"a"] = (df[index] < 0).astype(int)
            df[str(index)+"b"] = (df[index] == 0).astype(int)
            df[str(index)+"c"] = (df[index] > 0).astype(int)
            del df[index]

In [5]: triplets = [1,6,7,13,14,15,25,28]

        for num in triplets:
            reorder(df,num)

In [6]: col = df.columns.tolist()

In [7]: das = col[:21] + col[23:]
```

Lets reorder the columns so that our target training column in on the last section.

```
In [8]: das.append(col[22])

In [9]: df = df[das]
        out = df[col[22]]
        del df[30]
```

Now lets split the data

```
In [10]:  X_train, Xtest, Y_train, Ytest = train_test_split(df, out,
                                           test_size =0.33, random_state=0)
```

```
In [11]: X = X_train.values.tolist()
         xtest = Xtest.values.tolist()
```

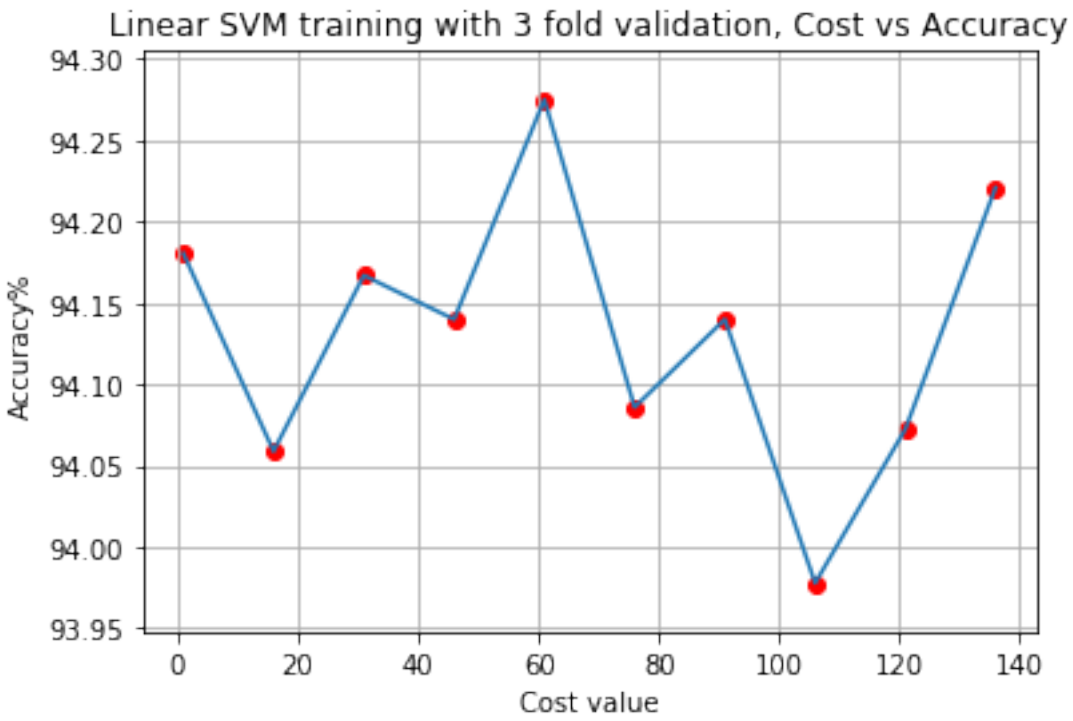
```
In [12]: Y = Y_train.tolist()
         ytest = Ytest.tolist()
```

Lets start to train our SVM

```
In [13]: prob = svm_problem(Y, X)
```

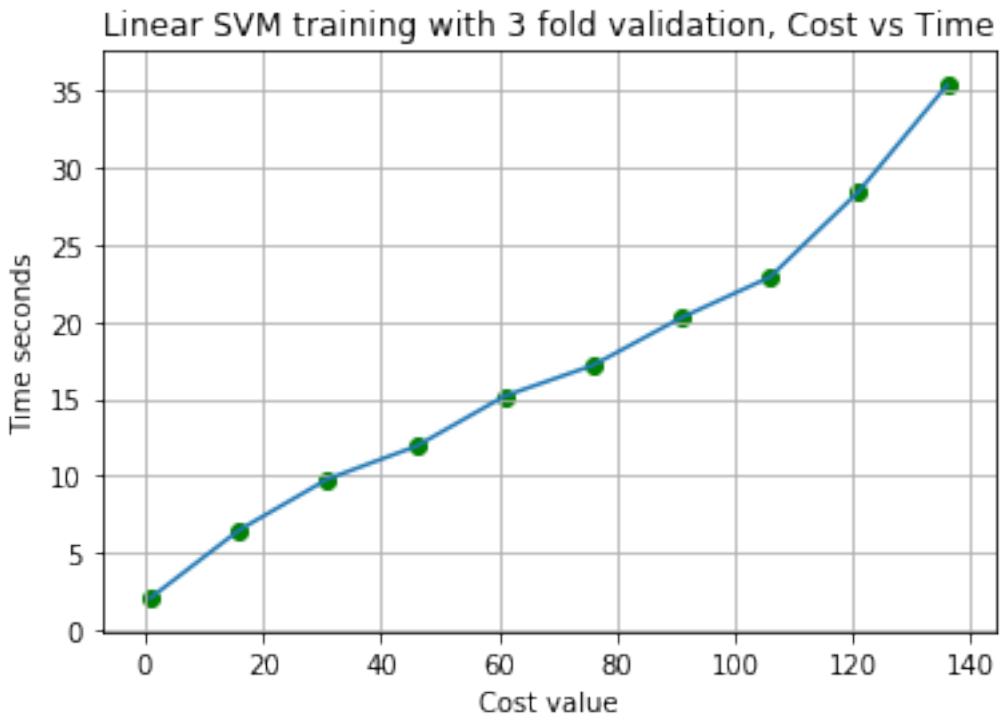
```
In [16]: c = 1
         cost = []
         lin_accuracy = []
         lin_timed = []

         for i in range(3):
             t0 = time()
             a = "-t 0 -v 3 -c " + str(c)
             param = svm_parameter(a)
             m = svm_train(prob, param)
             lin_accuracy.append(m)
             tp = time()
             lin_timed.append(round(tp-t0,3))
             print "Time is: ", round(tp-t0,3)
             cost.append(c)
             c +=15
         print lin_timed
```

```
In [69]: plt.scatter(cost,lin_accuracy, color='r')
         plt.plot(cost,lin_accuracy)
         plt.grid(True)
         plt.title("Linear SVM training with 3 fold validation, Cost vs Accuracy")
         plt.xlabel("Cost value")
         plt.ylabel("Accuracy%")
         plt.show()
```



The best value for C that I found using the 3fold cross validation is for C =61

```
In [74]: plt.plot(cost,lin_timed)
         plt.scatter(cost,lin_timed, color='g')
         plt.grid(True)
         plt.title("Linear SVM training with 3 fold validation, Cost vs Time")
         plt.xlabel("Cost value")
         plt.ylabel("Time seconds")
         plt.show()
         print lin_timed
```




```
[2.098, 6.492, 9.803, 11.926, 15.188, 17.187, 20.283, 22.913, 28.455, 35.372]
```

Now lets try our model on the prediction:

```
In [53]: m = svm_train(prob, '-t 0 -c 61 -n 3')

In [54]: svm_save_model('linear.model',m)
         m = svm_load_model('linear.model')
         p_label, p_acc, p_val = svm_predict(ytest, xtest, m)

Accuracy = 93.2036% (3401/3649) (classification)
```

The ***Accuracy*** for the testing set is **93.203%** when using the 3-fold cross validation on our data set. By using a set value of C to be 61, I found that this accuracy for the svm was a good one for prediciting the correct values of the test cases.

2 Kernel SVM in LIBSVM.

In this section I will experiment with different types of kernels which are the Polynomial kernel and the RBF kernel. I will also be switching the parameters to find the ideal parameters for the data:

The first kernel I will try is the polynomial kernel of degree 2.

```
In [23]: c = 1
         poly2_accuracy = []
         poly2_timed = []
         for i in range(10):
             t0 = time()
             a = "-t 1 -v 3 -d 2 -c " + str(c)
             param = svm_parameter(a)
             m = svm_train(prob, param)
             poly2_accuracy.append(m)
             tp = time()
             poly2_timed.append(round(tp-t0,3))
             c +=25

Cross Validation Accuracy = 94.5855%
Cross Validation Accuracy = 95.7737%
Cross Validation Accuracy = 95.6117%
Cross Validation Accuracy = 95.5442%
Cross Validation Accuracy = 95.3956%
Cross Validation Accuracy = 95.5982%
Cross Validation Accuracy = 95.0716%
Cross Validation Accuracy = 95.6252%
Cross Validation Accuracy = 95.7602%
Cross Validation Accuracy = 95.2336%
```

Now I will try training my data with the polynomial kernel of degree 3.

```
In [24]: c = 1
         poly3_accuracy = []
         poly3_timed = []
         for i in range(10):
             t0 = time()
             a = "-t 1 -v 3 -d 3 -c " + str(c)
             param = svm_parameter(a)
             m = svm_train(prob, param)
             poly3_accuracy.append(m)
             tp = time()
             poly3_timed.append(round(tp-t0,3))
             c +=25

Cross Validation Accuracy = 94.842%
Cross Validation Accuracy = 96.0708%
Cross Validation Accuracy = 96.0573%
Cross Validation Accuracy = 96.2463%
Cross Validation Accuracy = 95.8682%
Cross Validation Accuracy = 95.5847%
Cross Validation Accuracy = 95.7602%
Cross Validation Accuracy = 96.0032%
Cross Validation Accuracy = 95.7467%
```

```
Cross Validation Accuracy = 95.4361%
```

This last run is for the Radia Based Function Kernel.

```
In [25]: c = 1
         rbf_accuracy = []
         rbf_timed = []
         for i in range(10):
             t0 = time()
             a = "-t 2 -v 3 -c " + str(c)
             param = svm_parameter(a)
             m = svm_train(prob, param)
             rbf_accuracy.append(m)
             tp = time()
             rbf_timed.append(round(tp-t0,3))
             c +=25
```

```
Cross Validation Accuracy = 95.0851%
Cross Validation Accuracy = 96.1113%
Cross Validation Accuracy = 96.2868%
Cross Validation Accuracy = 95.9087%
Cross Validation Accuracy = 96.2193%
Cross Validation Accuracy = 96.1653%
Cross Validation Accuracy = 95.7197%
Cross Validation Accuracy = 96.2193%
Cross Validation Accuracy = 95.7602%
Cross Validation Accuracy = 95.7197%
```

It seems to be that the $\boldsymbol{RBF}$ kernel performs better than the linear or the polynomail functions.