Dustin Vasquez – 1917828

## Part1: SVM classification for Simulated Data

### Question 1: Generate a Data Set by Simulations

In this report we are going to display how to predict classifications using support vector machines (SVM) on a constructed data set with a separator determined by a polynomial of degree four. This data set will have four features and one response variable that will either be +1 or -1. Please not I will be using the term feature and explanatory variable interchangeably. There will be 10,000 cases to start with and we will reduce that down to 5,000 for the SVM algorithms. I will denote k as an individual case and n as the total number of cases in the dataset. Please note that I will be using cases and observations interchangeably.

All random numbers generated for the data set and polynomial factors will be generated with $Unif[-2,2]$ . The polynomial we will use to form our response for a single case is as follows:

$$Pol(x_k) = \Sigma_{i=1}^{4}\Sigma_{j=1}^{4}A_{ij} * x_{ki} * x_{kj} + \Sigma_{i=1}^{4}B_i * x_{ki} + \frac{c}{20} \qquad \forall \;\; k = 1,2,\dots,n$$

*Equation 1*: Polynomial Equation

First, we form X, or our dataset, by creating 10,000 vectors in dimension 4 with each variable being a random number. This is where we will get our $x_{ki}$ and $x_{kj}$ from. Then we will form A as a 4x4 matrix of random numbers. Then we form B as a vector in dimension 4 with each variable being a random number. Then we form c as a single random number. Now we have everything needed to generate our response vector using the polynomial stated above.

After generating our response vector there will be some that are greater than 0, and some that are less than 0. The ones that are less than 0 will be grouped into class -1, and the ones greater than 0 will be grouped into class +1. When generating data using random numbers, this will be random every time unless you set your seed. I set my seed at 229 and was lucky enough to have 4936 positive values and 5064 negative values. The equation form for assigning classes for our polynomial is as follows.

$$\Sigma_{i=1}^{4}\Sigma_{j=1}^{4}A_{ij} * x_{ki} * x_{kj} + \Sigma_{i=1}^{4}B_i * x_{ki} + \frac{c}{20} > 0 \rightarrow y_k = +1 \qquad \forall \;\; k = 1,2,\dots,n$$

$$\Sigma_{i=1}^{4}\Sigma_{j=1}^{4}A_{ij} * x_{ki} * x_{kj} + \Sigma_{i=1}^{4}B_i * x_{ki} + \frac{c}{20} < 0 \rightarrow y_k = -1 \qquad \forall \;\; k = 1,2,\dots,n$$

*Equation 2*: Equations to determine class of poly(x)

From here we form our dataset that we will use the remainder of the time by randomly sampling 2500 values that are positive and 2500 values that are negative. Note, before randomly sampling make sure you attach your response variables to your explanatory variables to keep them with each other. After this the data had the following description. The first 4 columns, 0-3, are descriptions of the explanatory variables, y is a description of the response variable.Notice that I went ahead and kept our calculation of the polynomial for future plots, but with real world data you will not always know this value. The only way you will have a value for this is if you do a regression problem that you turn into a classification problem by determining a certain class with a certain value that splits the data. The distance of your actual value to this reference point can be used as a distance.

| | 0 | 1 | 2 | 3 | y | U |
|---|---|---|---|---|---|---|
| **count** | 5000 | 5000 | 5000 | 5000 | 5000 | 5000 |
| **mean** | -0.03 | -0.01 | 0.02 | 0.01 | 0 | -0.64 |
| **std** | 1.15 | 1.15 | 1.15 | 1.16 | 1 | 4.69 |
| **min** | -2.00 | -2.00 | -2.00 | -2.00 | -1 | -18.67 |
| **25%** | -1.03 | -0.99 | -0.98 | -0.98 | -1 | -3.40 |
| **50%** | -0.03 | -0.02 | 0.05 | -0.01 | 0 | 0.00 |
| **75%** | 0.95 | 0.98 | 1.01 | 1.03 | 1 | 2.27 |
| **max** | 2.00 | 2.00 | 2.00 | 2.00 | 1 | 13.59 |

*Table 1*: Description of Data

After we get our full data set of 5000 observations, we want to standardize the data for the use of our SVM algorithms. Each feature will need to be standardized so that the mean is 0 and the standard deviation is 1 in order to run the SVM model. This is done by the following equation:

$$x = \frac{x_k - \overline{x_i}}{\sigma_{x_i}} \quad \forall \quad i = 1,2,3,4 \quad ; \quad k = 1, \dots, n$$

*Equation 3*: Equation for standardizing a feature

Using this formula, we standardized each explanatory variable in the data set and got the following description. Notice in this description that the mean values of the explanatory values, the first 4 columns have a value of 0 and a standard deviation of 1.

| | 0 | 1 | 2 | 3 | y | U |
|---|---|---|---|---|---|---|
| **count** | 5000 | 5000 | 5000 | 5000 | 5000 | 5000 |
| **mean** | 0 | 0 | 0 | 0 | 0 | -0.64 |
| **std** | 1 | 1 | 1 | 1 | 1 | 4.69 |
| **min** | -1.71 | -1.73 | -1.76 | -1.74 | -1 | -18.67 |
| **25%** | -0.86 | -0.85 | -0.87 | -0.86 | -1 | -3.40 |
| **50%** | 0.00 | -0.01 | 0.02 | -0.02 | 0 | 0.00 |
| **75%** | 0.85 | 0.87 | 0.86 | 0.88 | 1 | 2.27 |
| **max** | 1.76 | 1.75 | 1.72 | 1.72 | 1 | 13.59 |

*Table 2*: Description of Data after standardization

With the data being in more than 3 dimensions, it is hard to get a visual of the classes and their separation. One method in trying to do this is by calculating the correlations between the explanatory variables and the response variable and plot the two explanatory variables that are most correlated with the data. The two explanatory variables that correlated the most with y were feature 0 and feature 2 with correlations of -.207 and .305 respectively. Notice that these correlations are pretty low for being the best, that is because we randomly created the data and it will not always be this way. Either way, you can see that the data will not be easily classed and I definitely do not see where a linear separator can do a good job.

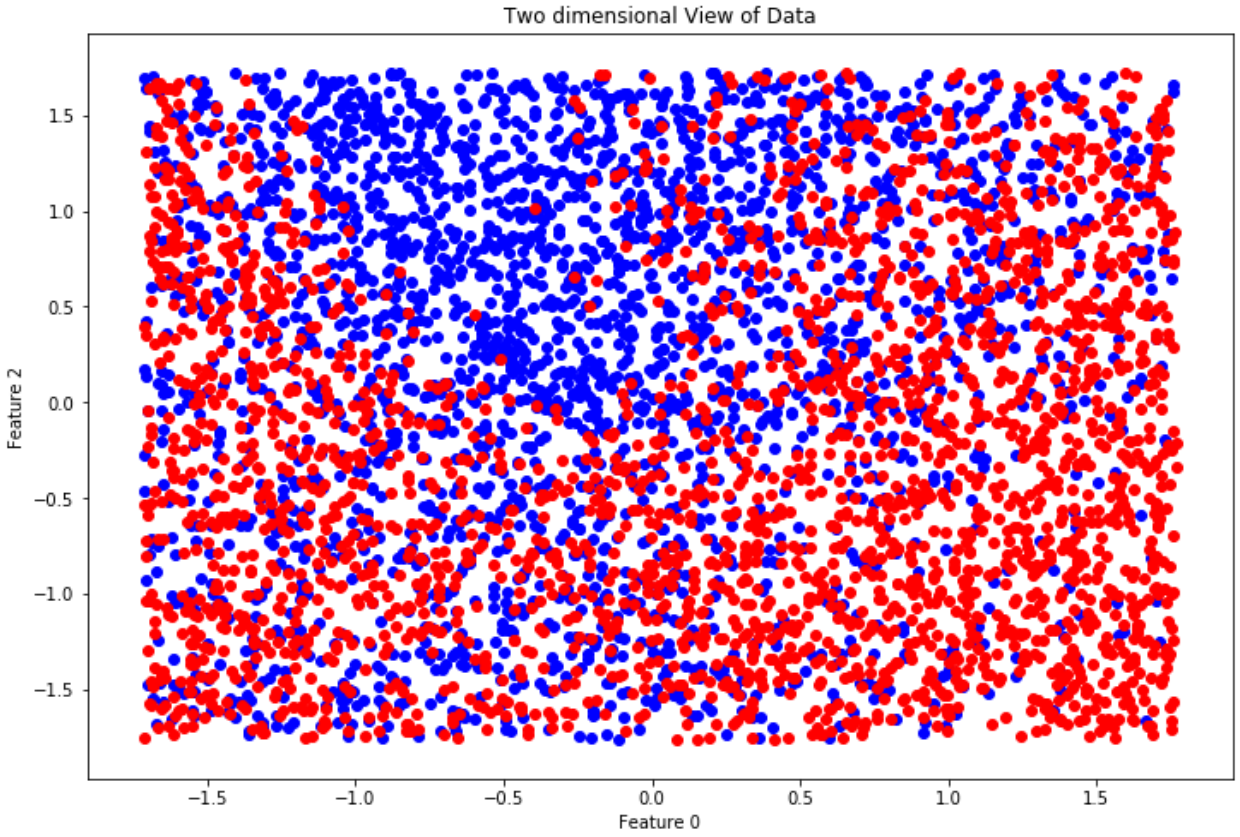


*Figure 1*: 2D plot of Data. The red points represent class -1, the blue points represent class +1

We can now split the data into two groups, our training set and our test set. We will use 80% of the observations for our training set and 20% of the observations for our test set, giving a quantity of 4,000 and 1,000 respectively. We need to split up the data in order to train the algorithm and then test how well that algorithm performed. The names of the data set line up with the name of their function. Now that we have our data sets the way we want, we can move on to tuning and training our algorithms, making predictions, and analyzing the accuracy of our predictions. We will start with a linear kernel, then do a radial or Gaussian kernel, and finally do a polynomial kernel. Each kernel has parameters that can be tuned in order to generate the best separating hyperplane, and this is one of the main factors that we will be touching on.

The simplest algorithm to use for predicting the class of each case is a linear kernel. This is done by finding the best possible hyperplane that separates the data by their respective response class. The equation of a hyperplane is as follows:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

$$p = number\ of\ features$$

*Equation* 4: Equation for a hyperplane

But there can be an infinite number of lines to separate the two classes, especially if they are distinct. In order to find the optimal line, we look at the margin between the separating hyperplane and the closest vectors for each class. Our goal is to find a line that has an equal and maximal margin between the two classes of data, called the maximal margin classifier. This is done by optimizing the following conditions:

$$Maximize\ M$$

$$\Sigma_{i=1}^{p}\beta_i = 1$$

$$y_k\left(\beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki}\right) \geq M \quad \forall \quad k = 1,2,\dots,n_{train}$$

*Equation 5*: Equation's for optimizing separating hyperplane

Note, in our case p is 4 because this is how many features we have, and n is 4000 because this is how many cases we have in our training set. If our data is perfectly separated, then $y_k\left(\beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki}\right) > 0$ for all cases because $\beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki}$ is the direction and distance from the hyperplane and $y_k$is the sign of the side of the hyperplane. So, by making it greater than or equal to M, we are saying that not only are they all perfectly separated, but they have a cushion of M which we have maximized. We want $\Sigma_{i=1}^{p}\beta_i = 1$ because we get more meaning from $\beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki}$ by saying the coefficients are a unit vector telling us strictly the direction and let the values of $x_{ki}$ play the role in determining distance. The following image from The Introduction to Statistical Learning with Applications in R, written by Gareth James, displays this description of a separating hyperplane, margin, and support vectors well.

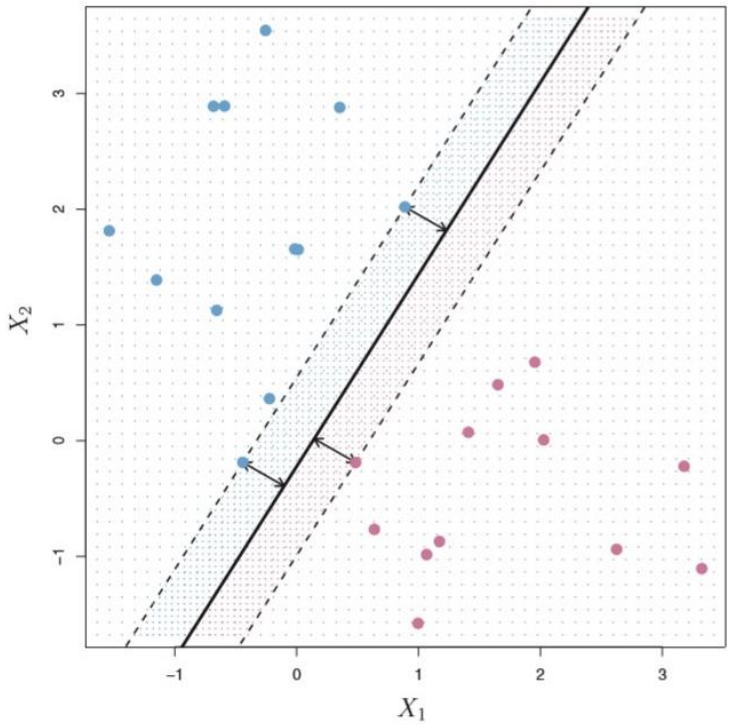


*Figure 2*: Image displaying structure of Maximal Margin Separator, margin, and support vectors.

You can see that these two-dimensional vectors are distinctly classed into blue and purple. It is easy to see that the separating hyperplane is determined by the closest vectors in each class. The margin on each side of the hyperplane is equal and maximized, so the hyperplane is centered between the two classes closest borders. These vectors that determine the hyperplane are called support vectors.

In our case, we generated our response variable by a polynomial of degree four, which is not linear and will not be separated by a linear hyperplane, as can be seen in figure 1. Most, real life instances will not have classes that are separated linearly. So, we need to allow for error to optimize our hyperplane by the following:

$$Maximize\ M$$

$$\Sigma_{i=1}^{p}\beta_i = 1$$

$$y_k\left(\beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki}\right) \geq M(1-\epsilon_k) \quad \forall \quad k = 1,2,\dots,n_{train}$$

$$\epsilon_k \geq 0\,, \qquad \Sigma_{k=1}^{n}\epsilon_k \leq C \qquad -or- \qquad \epsilon_k \leq \frac{1}{C} \quad \forall \quad k = 1,\dots,n_{train}$$

*Equation 5*: Equation's for optimizing separating hyperplane and allowing error

It can be seen from the above equations that if $0 \leq \epsilon_k \leq 1$, then that vector falls inside of the margin. If $1 < \epsilon_k$, then that vector falls on the wrong side of the separating hyperplane. The reason I gave those two equations involving the parameter C is because it is referenced both ways. On the left, C is the 'budget' for all of the errors, where $\frac{1}{C}$ is the cost for each individual error, which is how Python uses parameter C. By setting the cost of the error to $\frac{1}{C}$, we are saying that the smaller our cost the larger our margin and the higher our cost the tighter our margin. If our margin is large, we will have an algorithm that may have a low variance between test and training scores, but a high bias. On the flip side, a tight margin has more potential for a low bias but a high variance. To walk the tightrope of finding the best bias variance trade off, we tune the cost parameter to look for best score on both the training and the test set. Using this new set of equations and restrictions to find the best hyperplane is called the Support Vector Classifier, which is similar to saying it is Support Vector Machine with a linear kernel. There is a slight difference in notation between a SVC and a SVM with a linear kernel. The function for deciding which side of the hyperplane goes from the first equation to the second equation when you use a kernel function do determine the separating function.

SVC: $$f(x_k) = \beta_0 + \Sigma_{i=1}^{p}\beta_i x_{ki} = \beta_0 + \beta_1 x_{k1} + \beta_2 x_{k2} + \cdots + \beta_p x_{kp} \quad \forall \quad k = 1,\dots,n_{test}$$

*Equation 6*: Equation for predicting response in SVC Model

SVM: $$f(x_k) = \beta_0 + \Sigma_{i=1}^{n_{train}}\alpha_i < x_k^*, x_i > \quad \forall \quad k = 1,\dots,n_{test}$$

*Equation 7*: Equation for predicting response in SVM Model with linear kernel function

Please note that the * symbolizes the transpose of that vector. The $\alpha_i$ for this case is determined by the computer program. One thing to note, is that it may seem weird going from a formula that had $p$ factors to estimate to $n_{train}$ factors to estimate, especially when $p < n_{train}$ such as in our case. Luckily, it turns out the only $\alpha's$ that are not 0 are the ones that are support vectors, which helps reduce the number of estimates. Writing it in this form also sets it up for the use of kernels, which allows us to calculate the $f(x)$ in a greater dimension for a non-linear separator in our current $p$ dimensions. In terms of the kernel functions, the above equation can be written as:

$$f(x_k) = \beta_0 + \Sigma_{i=1}^{S}k(x_k, x_i) \quad \forall \quad k = 1,\dots,n_{test}$$

*Equation 8*: Equation for predicting response in SVM Model in terms of the kernel function

Notice how I changed the summation from $n_{train} to\ S$, which symbolizes number of support vectors. The kernel can be easily changed in the programming between the common types, or even a custom one.

**Question 2: SVM classification by linear kernel**

We will start by running our data through the SVM linear kernel without tuning any parameters and examine its results. Here is the equation for a linear kernel:

$$k(x_k, x_i) = < x_k^*, x_i > \qquad \forall \quad k = 1,\dots,n_{test} \;\; ; \;\; i = 1,\dots,n_{support\ vectors}$$

*Equation 9*: Linear Kernel Function

Using our training set we train the algorithm by giving it our explanatory variables and their respective Y (response) values from the training set. Doing this gives us a score on the training set of $68 \pm 1.4\%$. This score is not very good and would need to be improved for it to be useful. We can also see that there are 1454 support vectors for class -1, and 1455 support vectors for class +1, summing up to 2909 support vectors total. We can also find the ratio of support vectors by the following equation.

$$Ratio_{support\ vectors} = \frac{S}{n_{training}}$$

$$S = number\ of\ Support\ vectors, \qquad n_{training} = observations\ in\ training\ set$$

*Equation 9*: Ratio of support vectors

This is equivalent to saying that approximately 73% of the data are support vectors for the separating hyperplane. Since we have so many support vectors, it further supports that we should not feel very confident about our predictions because all of the data points are in the margin or wrongly classified. The following confusion matrix displays the percent confusion for the prediction of each class in the training set. The rows display the percent of actual observations in that classification and the columns represent the percent of predictions in that classification. With this, the sum of the rows is 100% because it represents all of the observations in that class. The diagonals represent the percent of correct predictions, or scores for that class. The table following the confusion matrix is the confidence intervals for each of the classes score plus the global score for its respective set of observations.

| N(-1)=2000<br>N(+1)=2000 | **-1** | **1** |
|---|---|---|
| **-1** | 64.75% | 35.25% |
| **1** | 28.85% | 71.15% |

*Table 3*: Percent Confusion Matrix for training set

| | **Lower Limit (%)** | **Percent Correct (%)** | **Upper Limit (%)** |
|---|---|---|---|
| **-1** | 63.27 | 64.75 | 66.23 |
| **1** | 69.75 | 71.15 | 72.55 |
| **Global** | 66.50 | 67.95 | 69.40 |

*Table 4*: Confidence intervals for training set accuracies

The confidence intervals were calculated assuming the error followed a normal distribution with a significance level of $\alpha = .05$. All of the confidence intervals presented through out the document will be found with the following equation.

$$Score \pm Z(.975) * \sigma$$

$$\sigma = \sqrt{\frac{Score * (1 - Score)}{Size_{class}}} \quad , \quad Z(.975) \cong 1.96$$

*Equation 10*: Equation for getting marginal error and confidence interval of proportions

Note that in the case of global scores, the size is not of a single class, but the combination of the classes in that data set since it is a global description. So, what we have learned from this confusion matrix is that the model we created can predict with about 64% accuracy for group -1 and about 71% accuracy for group +1 with the data set that trained it. This model has already seen this data and it still cant get better than a 70% global accuracy. This isn't terrible for it being the most basic algorithm of the SVM family, but I feel that we can do better with some other kernels. But first, lets look at how the model handles data it has not seen by looking at the percent confusion matrix for the test set.

| N(-1) = 500<br>N(+1) = 500 | -1 | 1 |
|---|---|---|
| **-1** | 62.4% | 37.6% |
| **1** | 31.2% | 68.8% |

*Table 5*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 59.4 | 62.4 | 65.4 |
| **1** | 65.9 | 68.8 | 71.7 |
| **Global** | 62.7 | 65.6 | 68.5 |

*Table 6*: Confidence intervals for test set accuracies

As you can see, it appears that it did a little worse than the training set, but this is expected because the algorithm has not seen any of these explanatory variables and their respective response variables. But if you look closely at the intervals of each global accuracy, you will see that they overlap, meaning we cannot say one did better than the other one. This makes me feel good about the robustness of the model since there was not really any change between accuracies on both training and test set.

Another way to examine how confident you can be about the model is by looking at the distance of each case, or observation, from the separating hyperplane. In python this is given to you through the decision_function method, and it gives you a distance from the model's hyperplane for every observation you give it. Naturally you will think that the closer an observation is to the hyperplane, the less confident you will be about that decision. Also, the further from the hyperplane, you can feel more confident in that decision. One of the ways to visually look at this is by creating a histogram. This looks at the frequency of observations within a specific distance away from the hyperplane. The hyperplanes reference in the graph is 0 because that is where we are measuring our distance from. The x-axis on the graph represents the distance from the hyperplane.

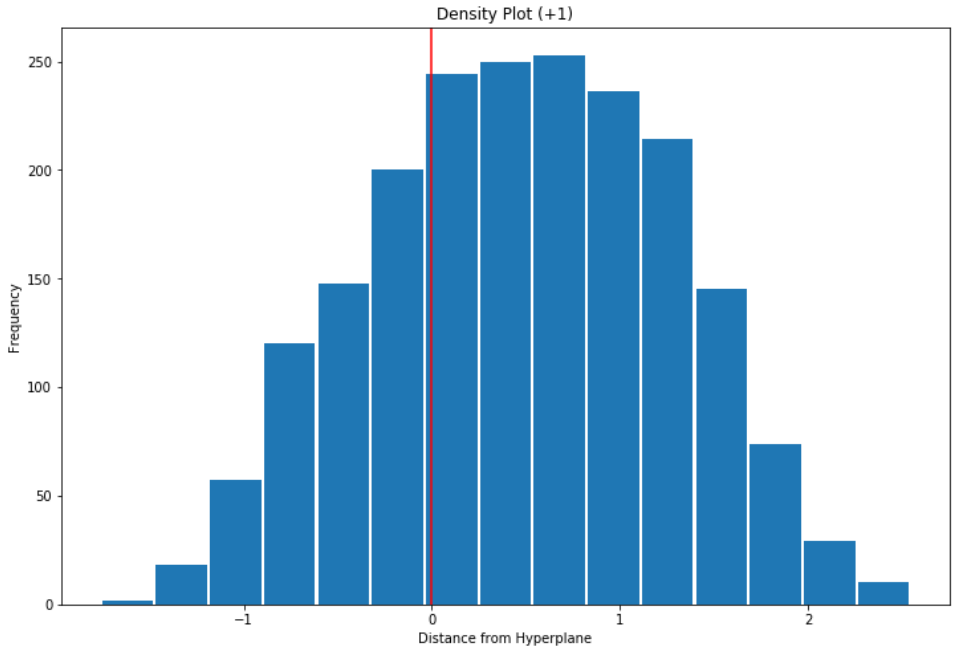


*Figure 3*: Histogram for +1 class

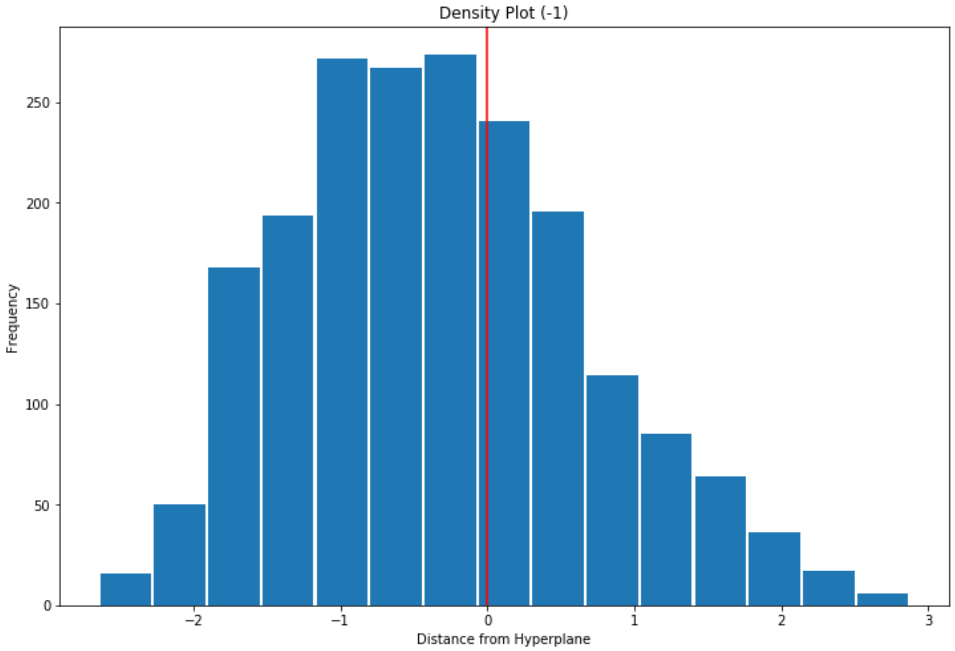


*Figure 4: Histogram for -1 class*

Looking at these histograms you can reference them to their respective confusion matrix. These histograms were created from the distances calculated on the training set. So, looking at the first histogram, which is for the +1 class, it is believable that about 70% of the observations fall to the right of the red line, or are correctly classified in the +1 class. The same is true for the -1 class, the area to the left of the red line looks like it is around 65%, which is the value that the percent confusion matrix tells us.

Another way to visually evaluate the confidence of your decisions is to look at the scatter plot of each observation with the x-axis being your polynomial value used for generating the classes and your y-axis being the distance from the model's hyperplane. In the case with SVM models, where you are only ever looking at two different classes at a time, each quadrant of the graph can be viewed as a quadrant of correct predictions or a quadrant of incorrect predictions. It makes sense that that the sign of the polynomial of x should match the sign of the distance from the hyperplane if it is a correct prediction. The quadrants with red points are the quadrants with incorrect predictions, and the quadrants with blue points are the quadrants with correct predictions. This scatterplot is for the training set. This is the graph we will be using to display visually what the confusion matrix tells us through numbers and help us get a feel for how confident we can be of the model. One of these graphs can be created for both the training set and the test set, but we won't waste the space of the test set until we start optimizing some parameters.

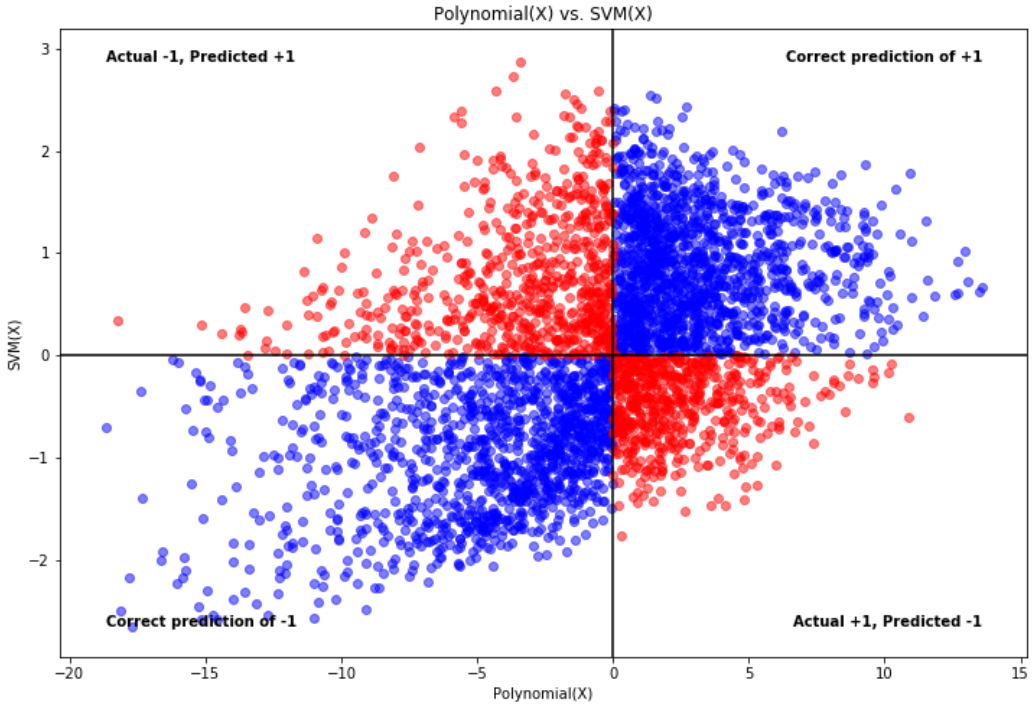


*Figure 5: Scatter plot of SVM(x) vs. Poly(x)*

**Question 3: Optimize the Parameter "Cost"**

When tuning for the best parameter to use for your model we have to remember that this is an iterative process in which we don't know exactly where to start. So, if we approach it from a broad picture and then focus in on the areas of interest, we may come out with better results. For the linear kernel, the only parameter that can be tuned is the cost of each error, described earlier. There are multiple ways to do this in any program. Python has a built-in function called grid search that allows you to specify a list of values or a specific method in which to analyze the parameters for optimizing the algorithm. The method I used for this report is by creating my own function that plotted the important factors of the decision process in determining the value of a parameter. I did this to break the steps down in order to better explain and graph what is going on.

There are 3 main factors I looked at when analyzing the models: the global percent accuracy of the training set, global percent accuracy of the test set, and the number of support vectors. These factors will be plotted vs. each parameter value to visualize how each change in the parameter effects the model. The reason we want to look at these three features is explained below:

1) Training set Score: Of course, we want an algorithm that can correctly classify the data that has trained it. More specifically we want to see how these scores change relative to the test set scores.
2) Test set Score: Like stated above, we want to see how these scores change relative to the training set scores. We expect these values to be less, but in order to pick a robust model, we want to try and find a point that has a high score and the difference between the test and train scores is low.
3) Number of Support Vectors: This kind of gives us an idea of how confident we should be in the model by telling us how much of our training data was used in making the hyperplane. The more support vectors, the more robust but also more variance and error. So we are looking for a model that has a fairly low amount of support vectors.

Dustin Vasquez – 1917828

Since I did not know what a good cost was for the linear model, I started by looking broadly at these six values of cost [.00001, .001, 1, 10, 30, 50]. Using these six values I got the following plots. The first plot displays the scores as the y-axis and the cost, or parameter of interest, as the x-axis. You can see that the score changes between $10^{-5}$ and 1, but levels out after that. Looking at the other plot below with the ratio of support vectors as the y-axis and the parameter of interest, cost, as the x-axis, we see that with a cost of $10^{-5}$ has 100% of our training observations used as support vectors. We also see that after the cost reaches 1, our ratio of support vectors levels out around 73%. Looking at these two graphs, we can say it is safe to zoom in between .001 and 2 to see what it looks like.

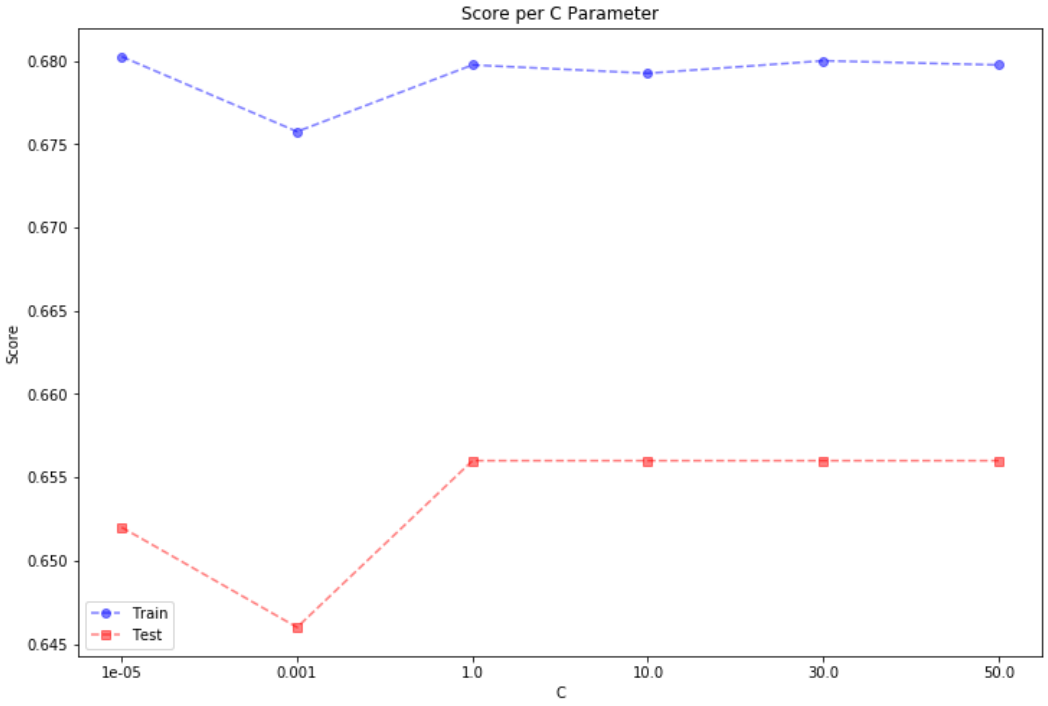


*Figure 6: Test and Train Scores per Parameter value*

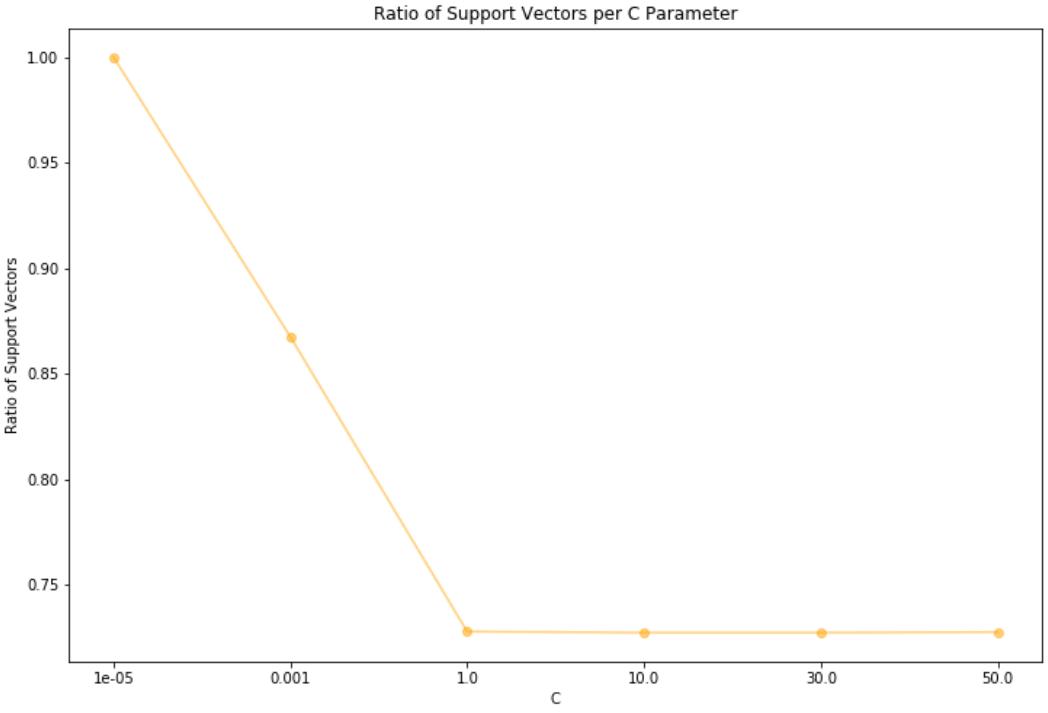

*Figure 7: Ratio of support vectors to training observations per Parameter value*

Running the tune function with these new values we get the following plots. By looking at these plots, I want to use .1 as my cost per error. This is the point where it reaches its max test score and really starts to level out in terms of scores and support vectors. If I use a higher cost, the algorithm is going to try harder to reduce the cost by tightening the margin, which takes time but doesn't give any benefit as the graph shows.

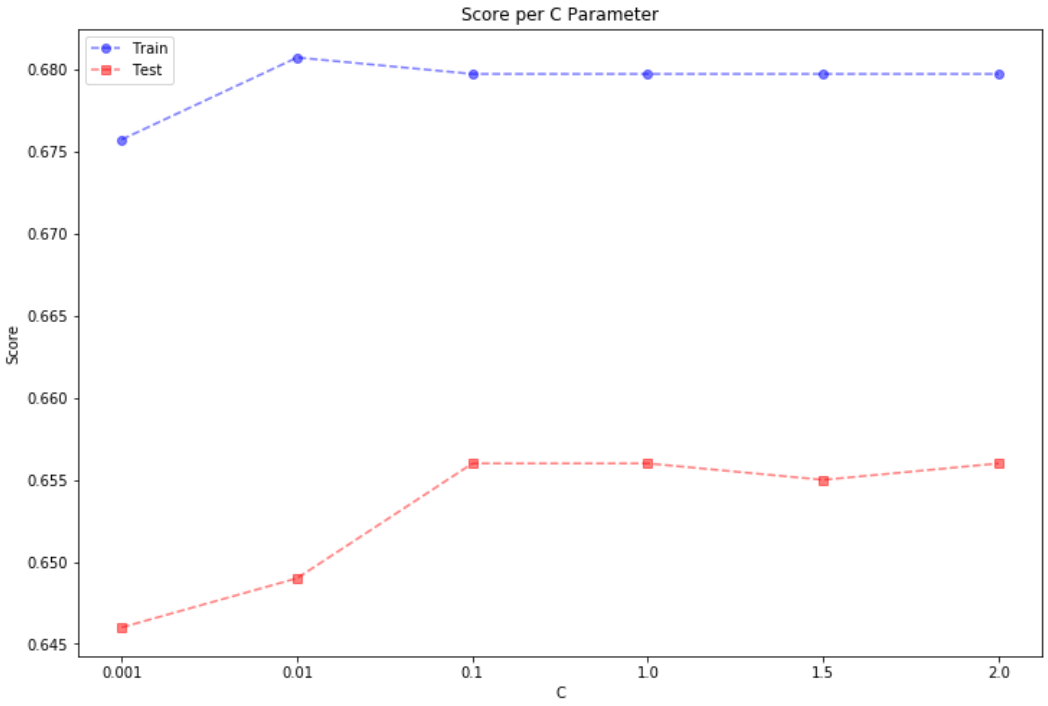


*Figure 8: Test and Train Scores per Parameter value*

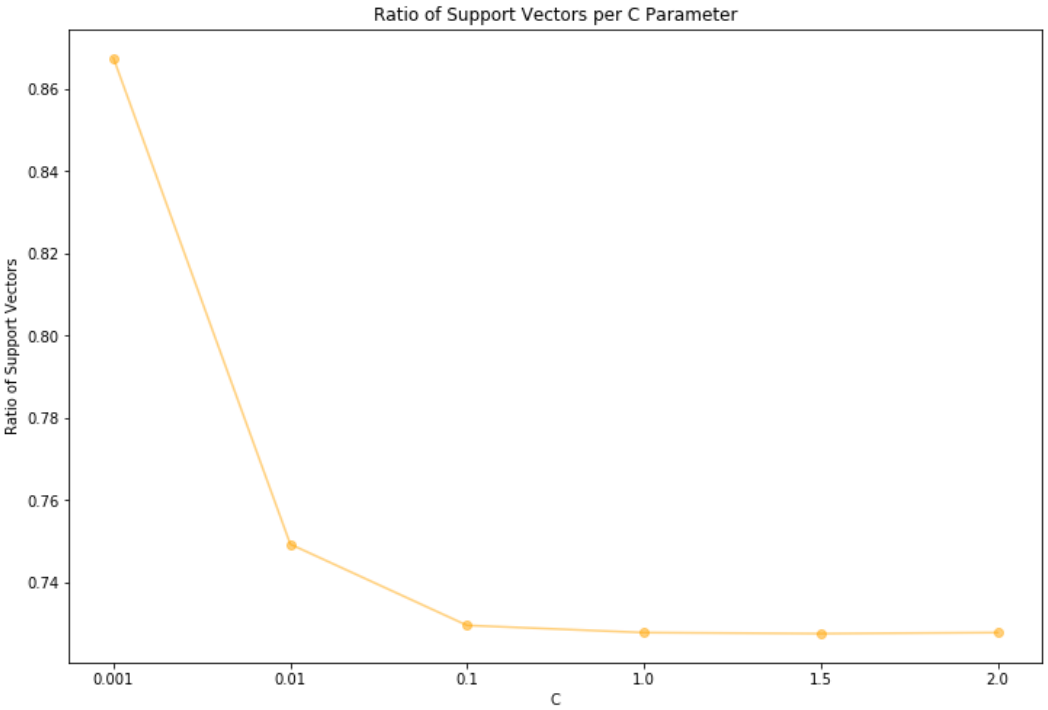


*Figure 9: Ratio of support vectors to training observations per Parameter value*

Using a cost of .1, I get the following confusion matrices for training and test sets. Notice that the global accuracy for the training set is $68\% \pm 1.4\%$ and a global accuracy for the test set of $65.6 \pm 2.9\%$.

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 64.8% | 35.2% |
| **1** | 28.85% | 71.15% |

*Table 7*: Percent Confusion Matrix for train set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 63.3 | 64.8 | 66.3 |
| **1** | 69.7 | 71.2 | 72.6 |
| **Global** | 66.5 | 68.0 | 69.4 |

*Table 8*: Confidence intervals for train set accuracies

| N(-1) = 500<br>N(+1) = 500 | -1 | 1 |
|---|---|---|
| **-1** | 62.6% | 37.4% |
| **1** | 31.4% | 68.6% |

*Table 9*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 59.6 | 62.6 | 65.6 |
| **1** | 65.7 | 68.6 | 71.5 |
| **Global** | 62.7 | 65.6 | 68.5 |

*Table 10*: Confidence intervals for test set accuracies

Comparing the values in these plots to the values in the original percent confusion matrices, there is no significant difference. In fact, they are practically identical. Another thing to note is that the ratio of support vectors stayed at 73%. With that being said, I do not feel the need to plot anything describing these matrices and we can move on to see if we can do better with different kernel.

**Question 4: SVM Classification by Radial Kernel**

Changing the kernel in the program is as easy as changing kernel type from 'linear' to 'radial', but theoretically it is quite another story. Changing the kernel from a linear kernel to a radial kernel changes the separating hyperplane from a linear one to a non-linear one by doing the calculations in another dimension.

The equation for a radial kernel is as follows:

$$k(x_k, x_i) = e^{-\gamma||x_k - x_i||^2} \qquad \forall \quad k = 1, \dots, n_{test} \;\; ; \;\; i = 1, \dots, n_{support\ vectors}$$

*Equation 11*: Radial Kernel Function

By plugging $k(x_k, x_i)$ into the equation for the separator in equation 8, you can calculate a non-linear separator for classification. With the radial kernel there are two parameters to optimize, $C$ and $\gamma$. Using our best C of .1 from the linear kernel and the standard $\gamma$ of 1, we get the following results. Notice that the global score for the training set is $97.2\% \pm 0.5\%$ and the test set is about $96.2 \pm 1.1\%$. This is without optimization. Using the radial kernel increased the model's accuracy significantly, even compared to the best linear model we could find in our last step.

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 96.4% | 3.7% |
| **1** | 2.0% | 98.1% |

*Table 11*: Percent Confusion Matrix for train set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 95.8 | 96.4 | 96.9 |
| **1** | 97.6 | 98.1 | 98.5 |
| **Global** | 96.7 | 97.2 | 97.7 |

*Table 12*: Confidence intervals for train set accuracies

| N(-1) = 500<br>N(+1) = 500 | -1 | 1 |
|---|---|---|
| **-1** | 95.6% | 4.4% |
| **1** | 3.2% | 96.8% |

*Table 13*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 94.3 | 95.6 | 96.9 |
| **1** | 95.7 | 96.8 | 97.9 |
| **Global** | 95.0 | 96.2 | 97.4 |

*Table 14*: Confidence intervals for test set accuracies

Let's visually see what these percent confusion matrices are telling us. Looking at the graphs below, the first most immediate notice is how few errors there are, relatively. Another thing that caught my eye is the pattern. This pattern shows me that there are a lot more observations that are further from separating hyperplane and this makes me feel more confident about the ability of this model to make predictions.

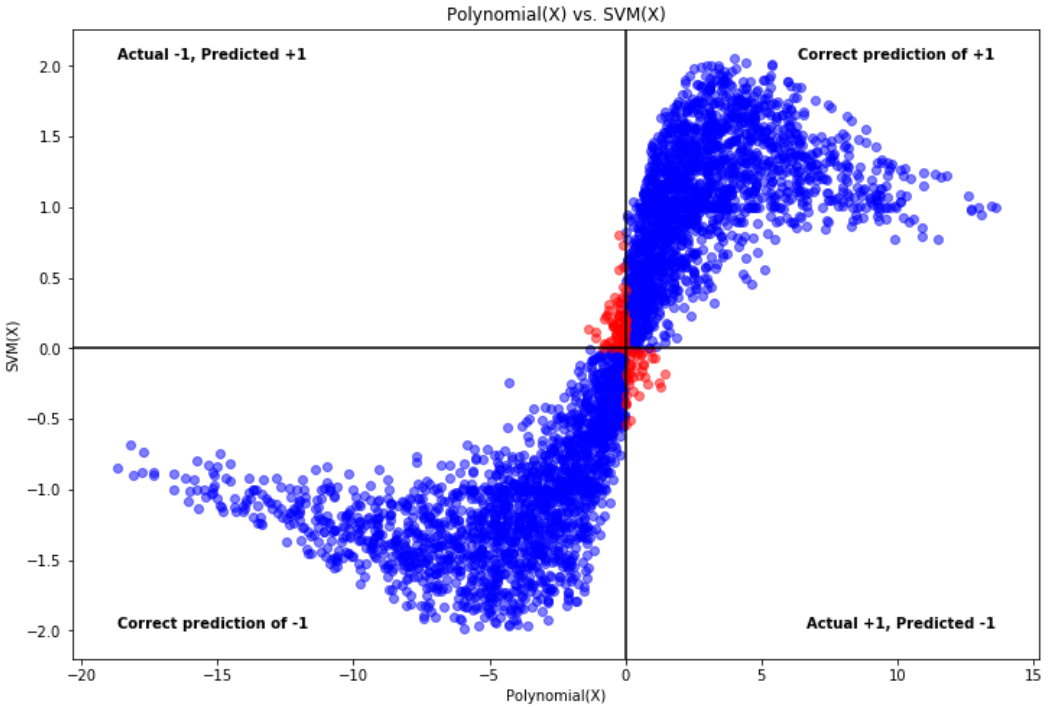


*Figure 10: Scatter plot of SVM(x) vs. Poly(x) for training set*

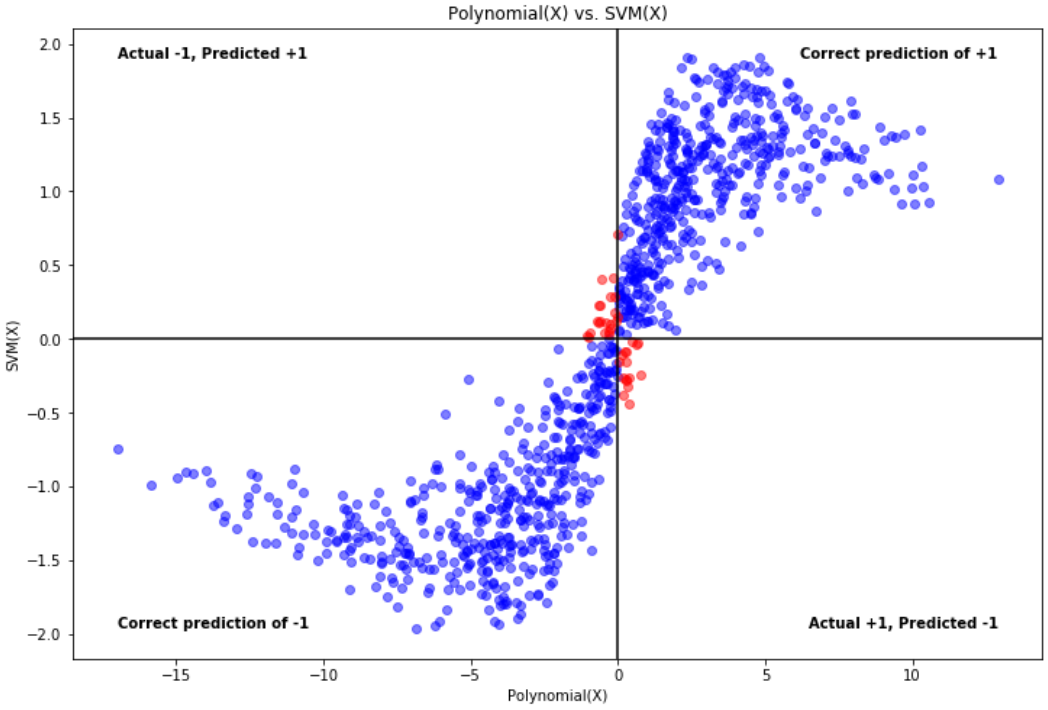


*Figure 11: Scatter plot of SVM(x) vs. Poly(x) for test set*

Another thing that is pretty interesting is that there were only 879 support vectors for -1 and 878 support vectors for +1. This is a total of 1757 support vectors which says that 44% of the observations in the training set determine the separating hyperplane. This is a big difference from the linear kernel which was 73% of the observations in the training set. We can have a lot more confidence in this model than the linear model already, and we have not optimized any parameters.

**Question 5: Optimize the Parameter "Cost" and "Gamma"**

This process is a little more extensive now that we have two parameters to optimize. Doing this step with the built in tuning functions will be much more efficient than doing it this way, but sometimes it is good to do both to make sure the program is making the same decision you would make. The method I am going to take is to optimize the $\gamma$ parameter. Use the best value for that optimization, then optimize the $C$ parameter. Then with the best $C$ parameter I will re-evaluate different $\gamma$ values to make sure that the new $C$ value doesn't affect the optimized $\gamma$ value.

We will start by looking a broad range of $\gamma$ values of [.00001, .001, .01, .1, .5, 1]. Looking at the plots below, we can see that the score for the test and train set make a significant leap from .01 to .1. It even looks like the test set did better than the training set at a $\gamma$ of .1, remember that there are naturally some errors in calculating these percentages so we cannot say they are significantly different. They are at their closest value there though, which makes me think that a $\gamma$ in that area may be the $\gamma$ that we want. Looking at the other graph below, we see the best support vector ratio of 39% is at a $\gamma$ of .5.

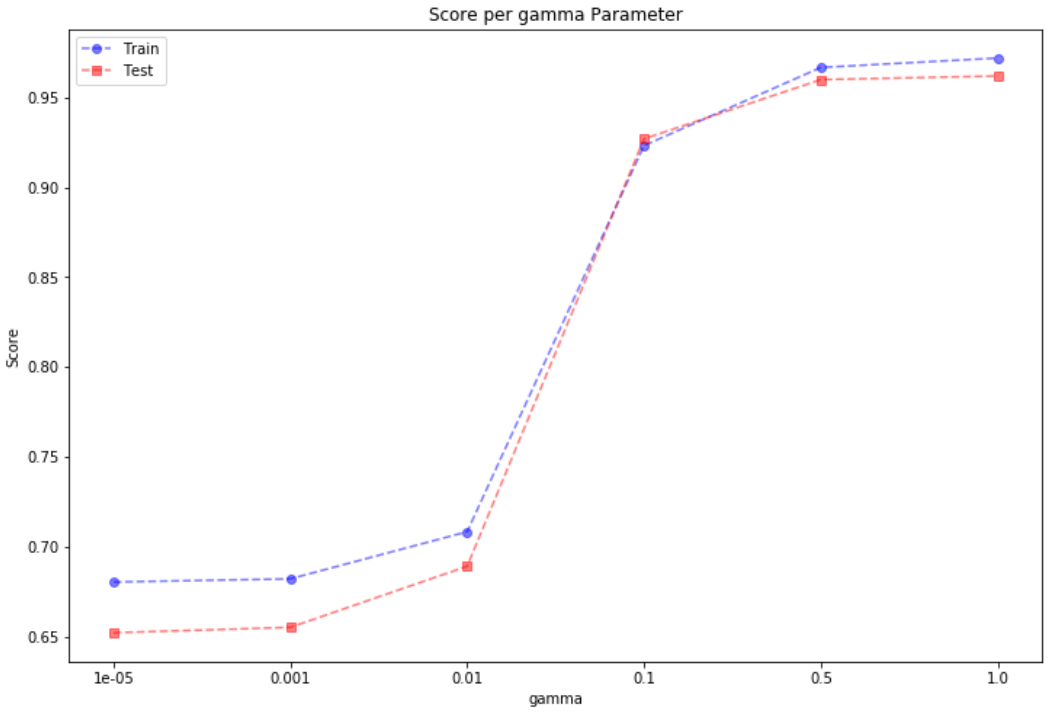


*Figure 12: Test and Train Scores per Parameter value*

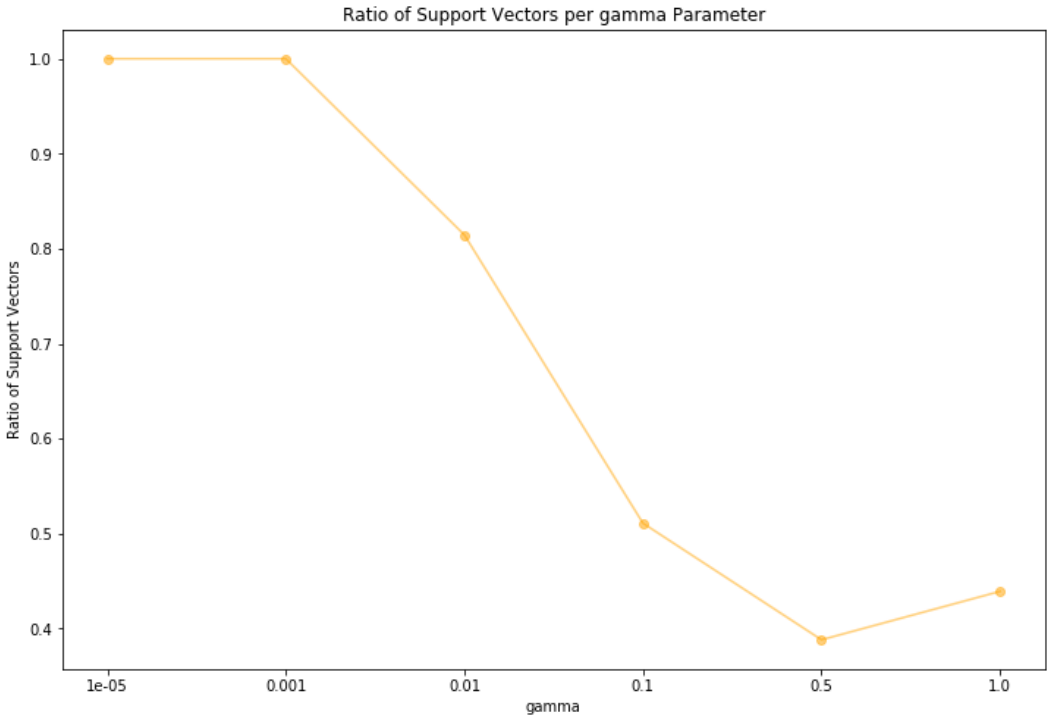


*Figure 13: Ratio of support vectors to training observations per Parameter value*

So, lets tune $\gamma$ again with a focus on the area between .05 and .75. After focusing in on our $\gamma$ parameter and using the values [.05, .075, .1, .25, .45, .65, .75], we get the following graphs.

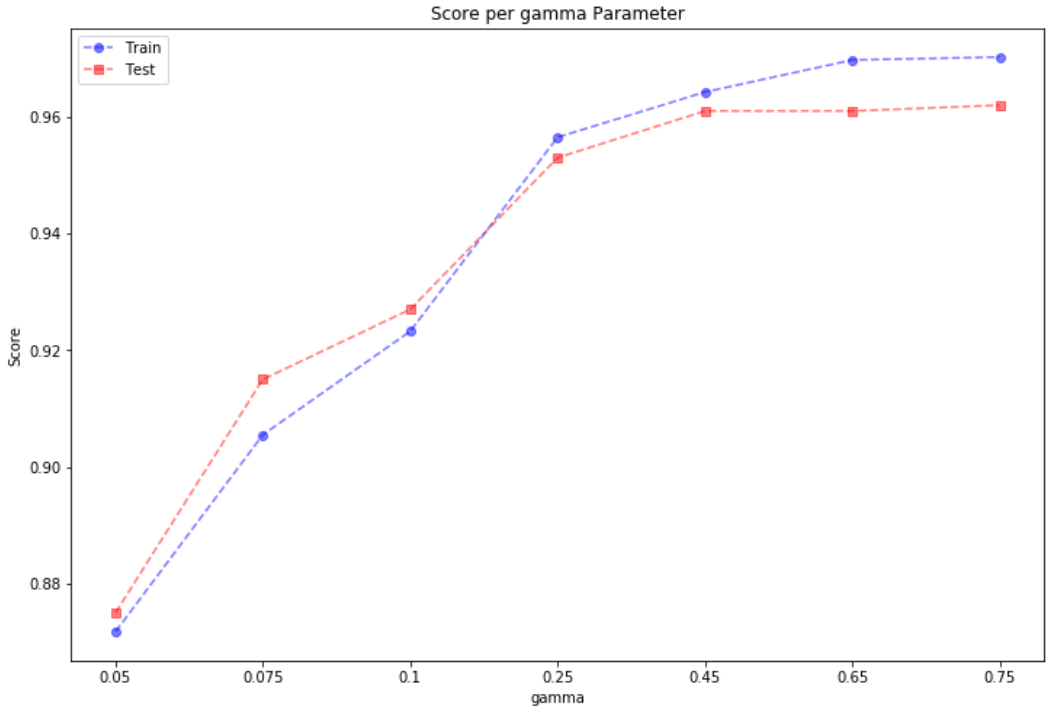


*Figure 14: Test and Train Scores per Parameter value*

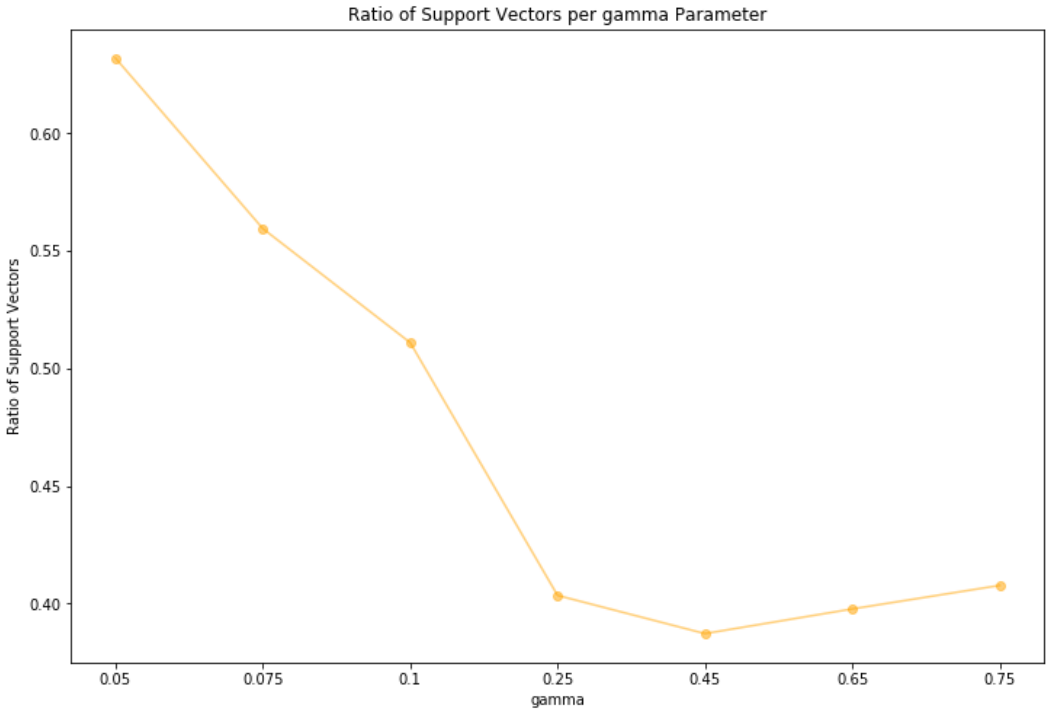


*Figure 15: Ratio of support vectors to training observations per Parameter value*

Looking at these two graphs above we see that the best score for our test set is at a $\gamma$ of 0.45, this also seems to be the point where the train and test scores start to separate. Looking at the ratio of support vectors, the lowest ratio is at a $\gamma$ of 0.45. I feel confident with this value for our parameter $\gamma$, so we will lock $\gamma$ to 0.45 and move on to optimizing the $C$ parameter with the following broad values [.00001, .01, 10, 35, 70, 100]. After running those values, I wanted to focus in on a $C$ of around 30, so I used the following values [ 15, 20, 25, 30, 35, 40, 45] and got the following plots. Note that I did not plot the original broad view of the Cost parameter to save reduce the amount of space taken by plots.

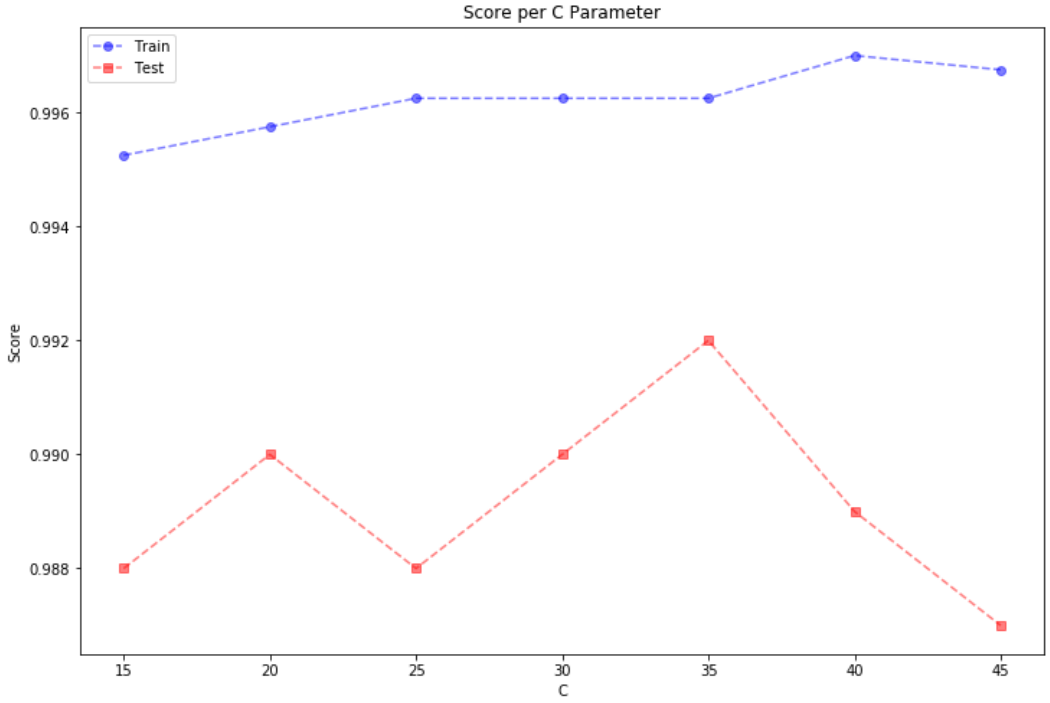


*Figure 16: Test and Train Scores per Parameter value*

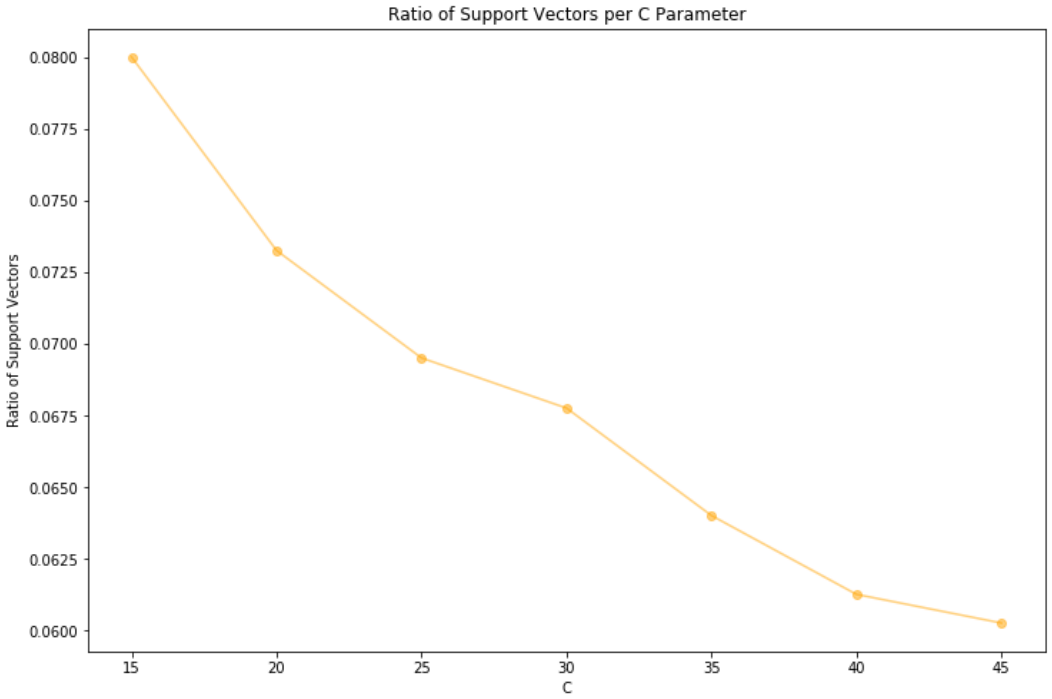


*Figure 17: Ratio of support vectors to training observations per Parameter value*

Looking at these plots it seems like a $C$ of 35 will be the value I choose. I chose this value because on the graph of scores between test and train, it looks like after that point the test and train start to spread from each other. One thing to notice is how low the ratio is for support vectors to observations in the training set. You can see as you raise the cost of each error that the number of support vectors reduces. That is because as you raise the cost, you are putting a limit on the size of the error allowed. As your cost goes up, it will try and reduce the error size until it reaches that point where it cannot reduce the error size anymore. The concern with only looking at ratio of support vectors, is that you can over fit the data and get yourself a good score for your training set, but a potentially worse score for your test set, this is also referred to as low bias but high variance.

Now we will see how changing the $C$ parameter affected the $\gamma$ parameter by locking the $C$ at 35 and re-analyzing results with a $\gamma$ in the focused range of values [.05, .075, .1, .25, .45, .65, .75].

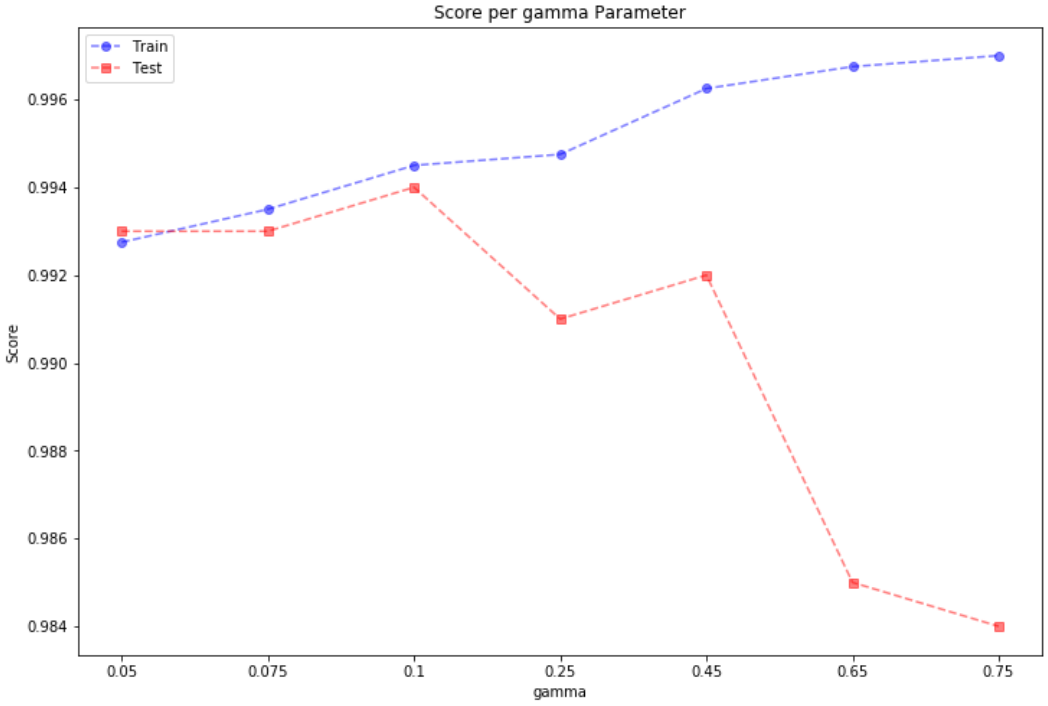


*Figure 18: Test and Train Scores per Parameter value*

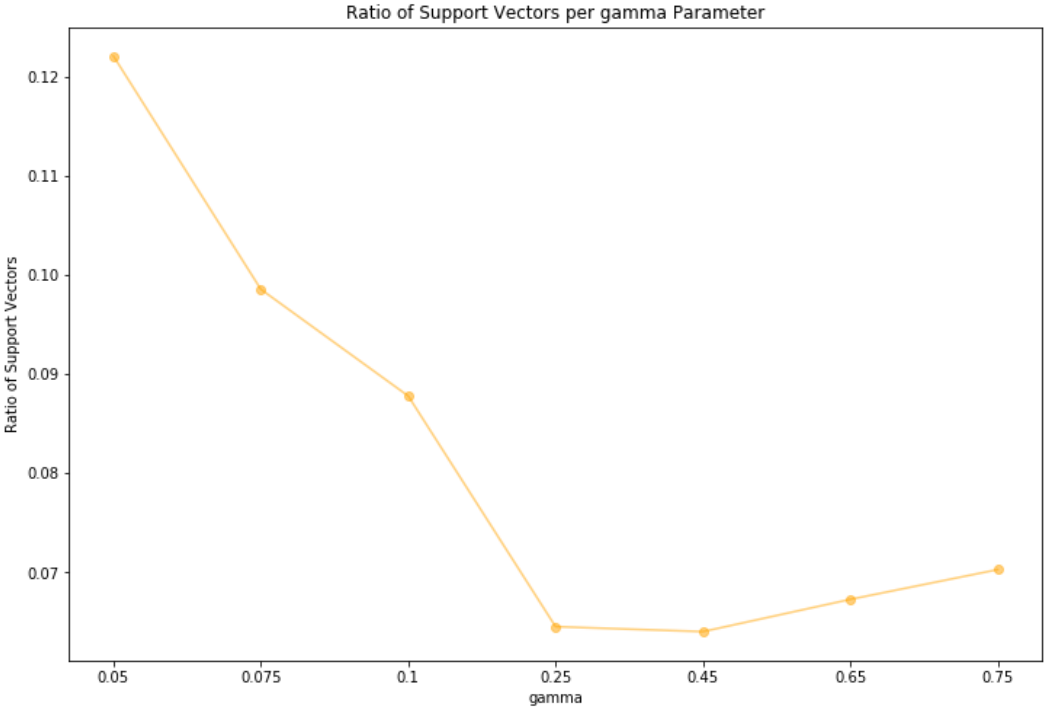


*Figure 19: Ratio of support vectors to training observations per Parameter value*

It is easy to see that a $\gamma$ of 0.45 is no longer the best value, 0.1 seems to be the best value. This process is done until you converge at some values or you are happy with the results you are getting. I changed $\gamma$ to 0.1 and then looked at the $C$ for the same range of values I used earlier. After doing this I found the best value for $\gamma$ to be 0.1 and the best value for $C$ to be 35. Now that we have some parameters that we are happy with, we look at the percent confusion matrices and scatterplots for the train and test set to analyze the model.

Dustin Vasquez – 1917828

Looking at the percent confusion matrices below, you will see that the global training score is $99.5\% \pm .2\%$ and the global test score is $99.4\% \pm .4\%$.

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 99.4% | 0.6% |
| **1** | 0.4% | 99.5% |

*Table 15*: Percent Confusion Matrix for train set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 99.2 | 99.4 | 99.6 |
| **1** | 99.3 | 99.5 | 99.7 |
| **Global** | 99.2 | 99.5 | 99.7 |

*Table 16*: Confidence intervals for train set accuracies

| N(-1) = 500<br>N(+1) = 500 | -1 | 1 |
|---|---|---|
| **-1** | 99.6% | 0.4% |
| **1** | 0.8% | 99.2% |

*Table 17*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 99.2 | 99.6 | 99.99 |
| **1** | 98.7 | 99.2 | 99.8 |
| **Global** | 98.9 | 99.4 | 99.9 |

*Table 18*: Confidence intervals for test set accuracies

Looking at the difference between the train and test results, we can see that there is no significant difference between the percent accuracies of the two. This is telling me that the model is pretty robust and has low variance, meaning we can expect to get similar accuracy scores whether it has seen the data before or not. Another thing to note, is when comparing our optimized results to our original results, we see that both our training set and test set significantly change for the better. This is good, and why went through the effort of optimizing the parameters. If after optimizing the parameters, our final test set score did not improve, then our tuning effort would have been fruitless because our whole purpose is to improve and make better predictions of data the algorithm has not seen. Another thing to note about the final tuned model is that our ratio of support vectors has reduced down to about 9%. This is a big difference and greatly increases confidence in the model.

Looking at the plots below, you can see how little red there is in them. You can also see how well the algorithm learned the training set by how dense the observations are and the pattern with which the observations take. The similarity in the two graphs is another sign of confidence that the model is robust and has low variance. Another thing to notice is the scale of the y-axis for the tuned scale is about (-25,20) which is much greater than the original radial kernels scale of around (-2,2). This increase in distance from the separating hyperplane also gives me much more confidence in the model.

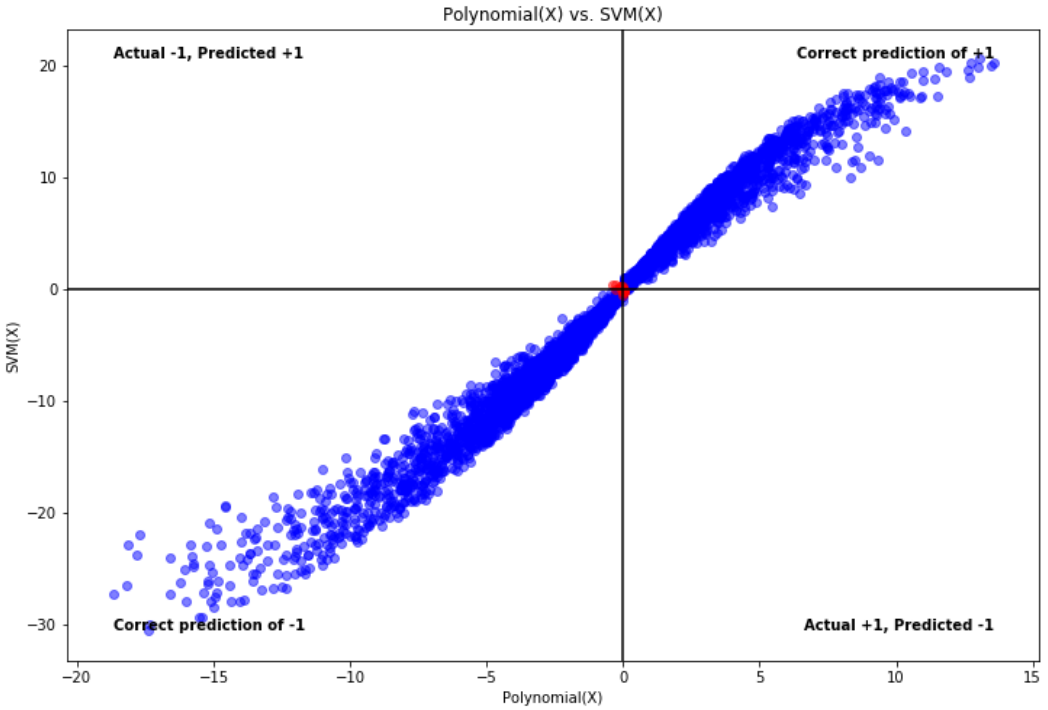


*Figure 20: Scatter plot of SVM(x) vs. Poly(x) for training set*

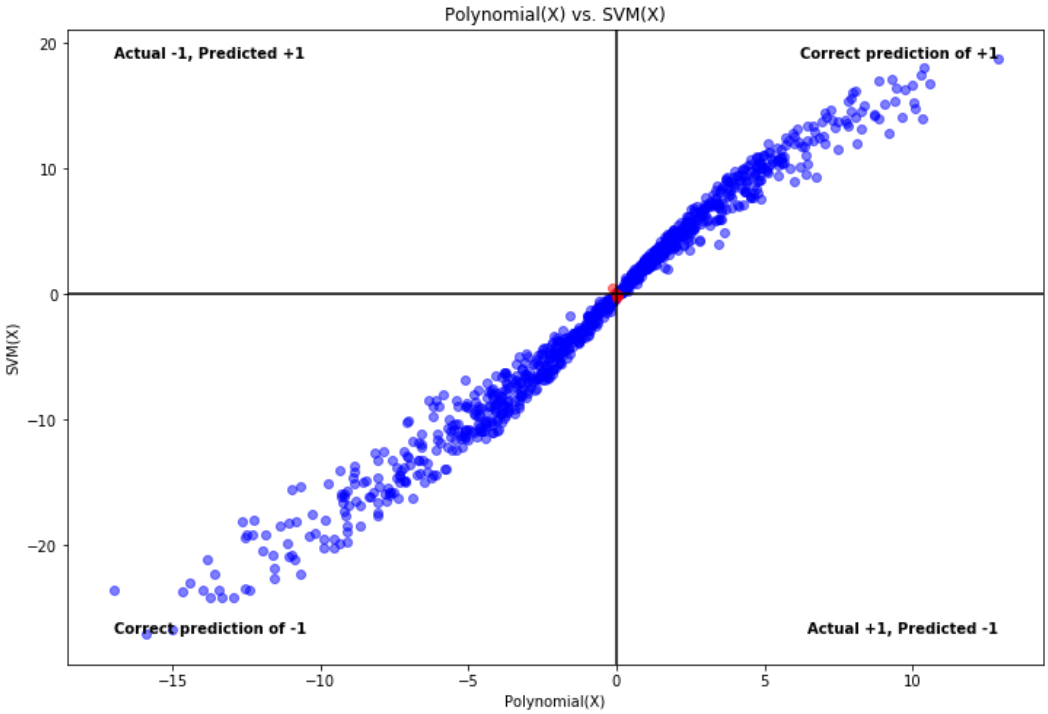


*Figure 21: Scatter plot of SVM(x) vs. Poly(x) for test set*

Overall, I feel pretty confident in the process of our tuning for $\gamma$ and $C$ and think it is time to move onto looking at the polynomial kernel and optimizing those parameters.

**Question 6: SVM classification using a polynomial kernel**

The polynomial kernel is as follows:

$$k(x_k, x) = (Coef_0 + < x_k^*, x >)^d$$

*Equation 12*: Polynomial Kernel Function

With the polynomial kernel there are 3 different parameters to optimize: $C$, $Coef_0$, $d$. In this case we are going to leave the degree, $d$, as 4 and just optimize the other two. First, we will look at how the model runs without any optimization so that we can compare our optimized results in the end and see if our effort was worth it.

Looking at the scores for the test and training set, we see that they are pretty similar, telling us that the model is pretty robust. But The accuracies are not that high, with $83.2\% \pm 1.1\%$ for the training set and $82.3\% \pm 2.4\%$ for the test set. I feel that some optimization will raise the accuracy of this polynomial kernel significantly. Another thing to look at is how there are 902 support vectors for each class, summing up to 1804 support vectors total, telling us that 45% of the training observations are support vectors.

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 80.2% | 19.8% |
| **1** | 13.9% | 86.1% |

*Table 19*: Percent Confusion Matrix for train set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 79.0 | 80.2 | 81.4 |
| **1** | 85.0 | 86.1 | 87.2 |
| **Global** | 82.0 | 83.2 | 84.3 |

*Table 20*: Confidence intervals for train set accuracies

| N(-1) = 500<br>N(+1) = 500 | -1 | 1 |
|---|---|---|
| **-1** | 79.2% | 20.8% |
| **1** | 14.6% | 85.4% |

*Table 21*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 76.7 | 79.2 | 81.7 |
| **1** | 83.2 | 85.4 | 87.6 |
| **Global** | 79.9 | 82.3 | 84.7 |

*Table 22*: Confidence intervals for test set accuracies

Looking at the plots, I notice that the distance from observations to the hyperplane has increased drastically for a polynomial with degree 4. It is definitely a better model than the linear kernel right out the gates, but the polynomial kernel definitely has some improvements to make if it wants to compete with the radial kernel.

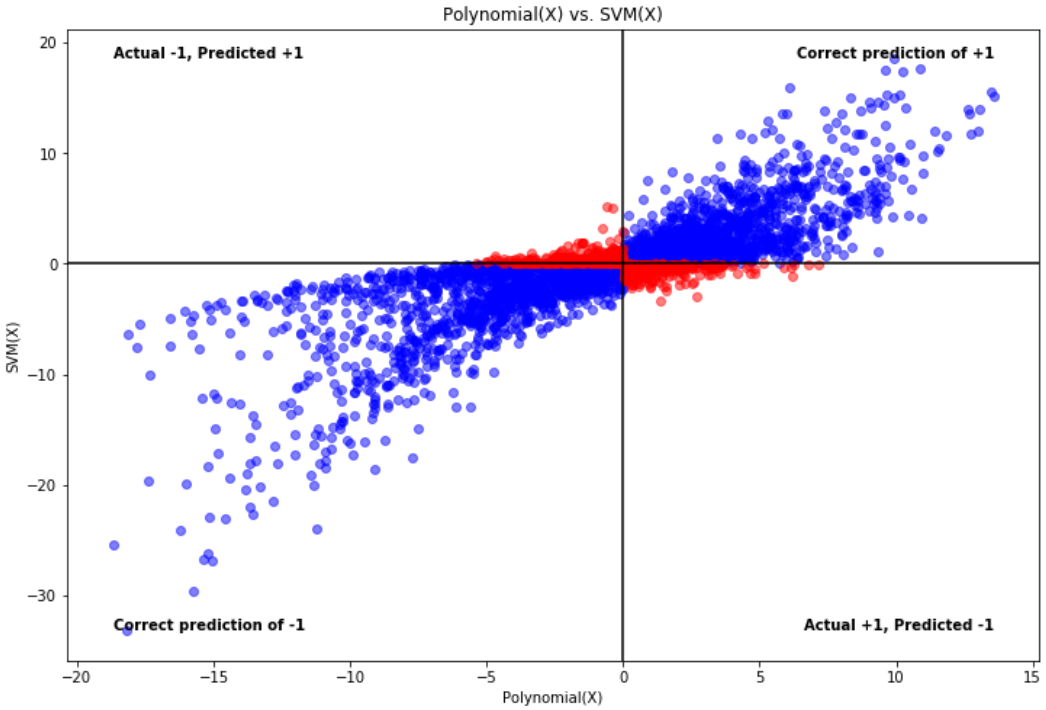


*Figure 21: Scatter plot of SVM(x) vs. Poly(x) for training set*

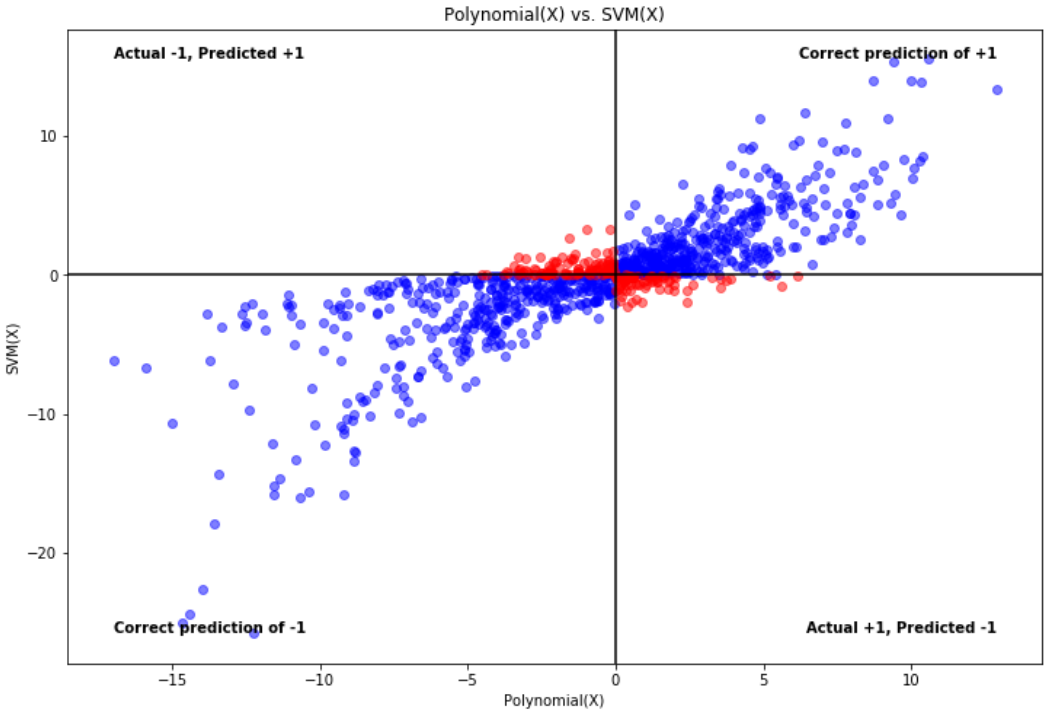


*Figure 22: Scatter plot of SVM(x) vs. Poly(x) for test set*

In tuning the parameters for the polynomial kernel, I will display the tables with values instead of all the graphs and then display the final tuning graph to show the relationship between train and test scores and the number of support vectors. The tables on the left are the values for our broad view of the parameters and the tables on the left are the values of our focused view on the parameters. The highlighted values show my values of interest.

| $Coef_0$ | Train | Test | SV Ratio |
|---|---|---|---|
| **0.001** | 0.916 | 0.908 | 0.414 |
| **0.1** | 0.981 | 0.981 | 0.220 |
| **10** | 0.998 | 0.996 | 0.020 |
| **30** | 1.000 | 0.998 | 0.011 |
| **60** | 1.000 | 0.996 | 0.008 |
| **100** | 0.999 | 0.993 | 0.009 |
| **200** | 0.997 | 0.996 | 0.011 |

(focusing on 30)

| $Coef_0$ | Train | Test | SV Ratio |
|---|---|---|---|
| **15** | 1.000 | 0.996 | 0.016 |
| **20** | 1.000 | 0.997 | 0.013 |
| **25** | 1.000 | 0.998 | 0.013 |
| **30** | 1.000 | 0.998 | 0.011 |
| **35** | 1.000 | 0.998 | 0.010 |
| **40** | 1.000 | 0.997 | 0.010 |
| **45** | 1.000 | 0.997 | 0.009 |

*Table 23*: Optimization tables for $Coef_0$

Using a $Coef_0$ of 30, we will now optimize the cost.

| Cost | Train | Test | SV Ratio |
|---|---|---|---|
| **0.001** | 0.995 | 0.994 | 0.089 |
| **0.1** | 0.999 | 0.995 | 0.020 |
| **25** | 1.000 | 0.993 | 0.008 |
| **50** | 1.000 | 0.993 | 0.008 |
| **75** | 1.000 | 0.993 | 0.008 |
| **100** | 1.000 | 0.993 | 0.008 |
| **125** | 1.000 | 0.993 | 0.008 |

(focusing .1 – 25)

| Cost | Train | Test | SV Ratio |
|---|---|---|---|
| **0.1** | 0.999 | 0.995 | 0.020 |
| **1** | 1.000 | 0.998 | 0.011 |
| **5** | 1.000 | 0.995 | 0.009 |
| **10** | 1.000 | 0.995 | 0.009 |
| **15** | 1.000 | 0.993 | 0.008 |
| **20** | 1.000 | 0.993 | 0.008 |
| **25** | 1.000 | 0.993 | 0.008 |

*Table 24*: Optimization tables for $C$

Since the original cost used in determining the coefficient was 1, we don't really need to re-evaluate and make sure that the change in cost didn't affect the coefficient because there was no change in cost. Here are the plots of the focused parameter values for tuning the cost.

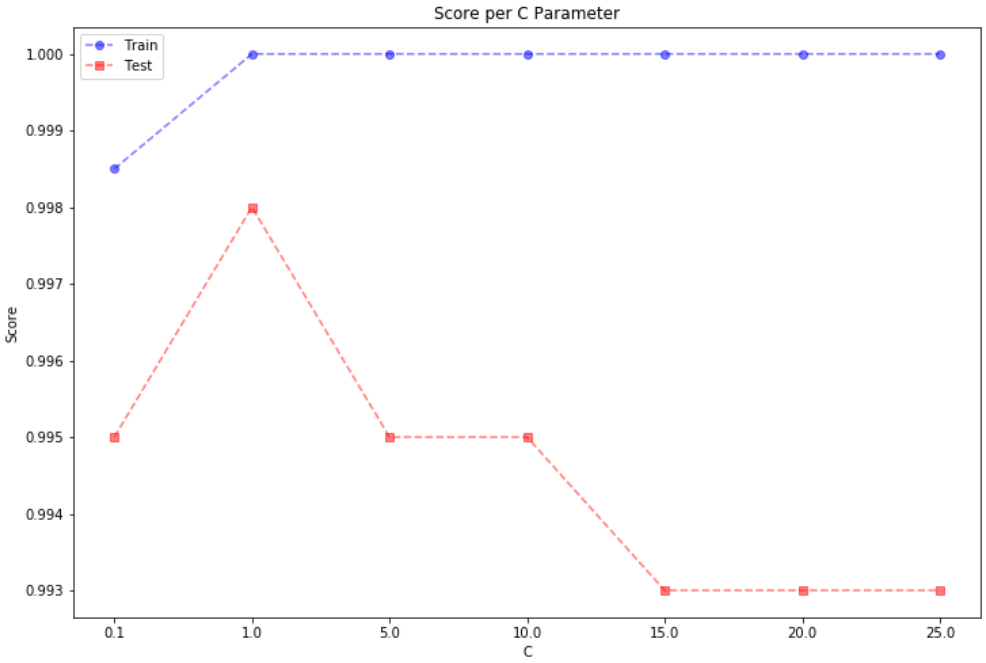


*Figure 23: Test and Train Scores per Parameter value*

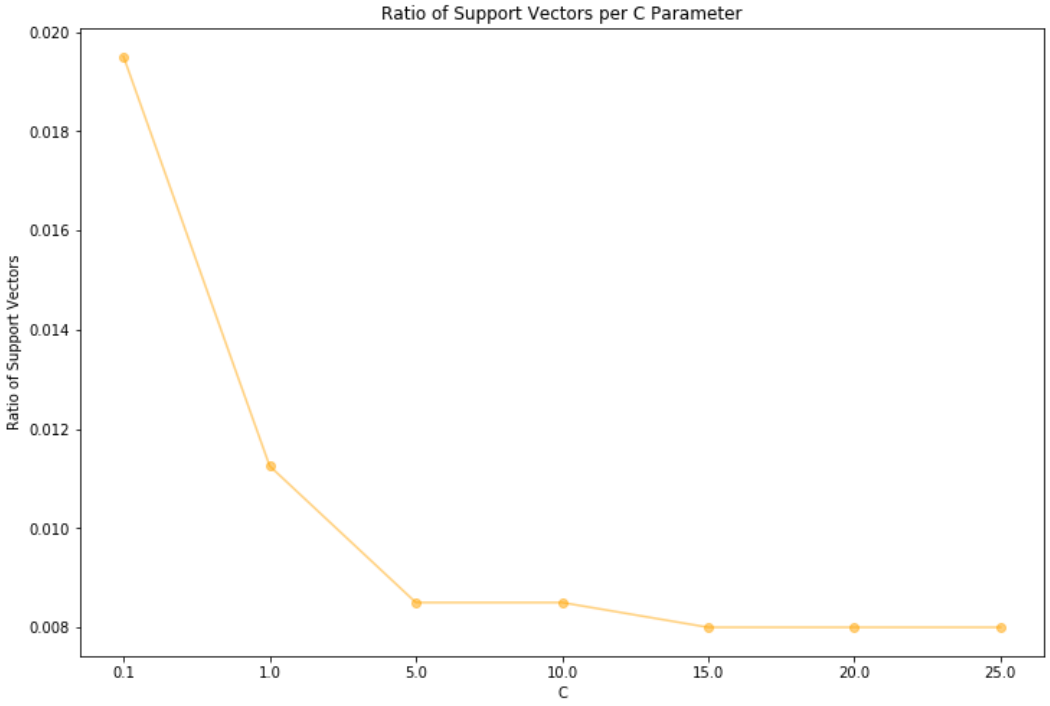


*Figure 24: Ratio of support vectors to training observations per Parameter value*

It seems like the most robust set of parameters falls with a coefficient of 30 and a cost of 1. The values on the results of these tuning functions are so close, I don't really believe there is that much of a difference. Let's look at our final result with our new optimized coefficient and cost. It is easy to see that this model has done by far the best with a training cost of $100\% \pm 0\%$ and a test score of $99.8\% \pm .003\%$. It is also interesting to note that the final ratio of support vectors is reduced all the way down to 1.1%.

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 100.0% | 0.0% |
| **1** | 0.0% | 100.0% |

*Table 25*: Percent Confusion Matrix for train set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 100.0 | 100.0 | 100.0 |
| **1** | 100.0 | 100.0 | 100.0 |
| **Global** | 100.0 | 100.0 | 100.0 |

*Table 26*: Confidence intervals for train set accuracies

| N(-1) = 2000<br>N(+1) = 2000 | -1 | 1 |
|---|---|---|
| **-1** | 100% | 0% |
| **1** | 0.4% | 99.6% |

*Table 27*: Percent Confusion Matrix for test set

| | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| **-1** | 100.0 | 100.0 | 100.0 |
| **1** | 99.2 | 99.6 | 100.0 |
| **Global** | 99.5 | 99.8 | 100.0 |

*Table 28*: Confidence intervals for test set accuracies

If you look at the y axis on the plots below, you can see that the distance of the svm(x) is really far away from the separating hyperplane giving me a lot of confidence in this model. I feel like a lot of this is due to the fact that the $Coef_0$ was at such a high value which was then drastically increased when raised to the $4^{th}$ power. You can also see how well the observations are grouped and the linear pattern they are displaying. Comparing the plots, and confusion matrices, between the training set and test set, you see there is really not any variance between them. This is really promising and gives me a lot of confidence that this model will accurately predict the class of any poly(x) formed with our generating formula used earlier in the document.

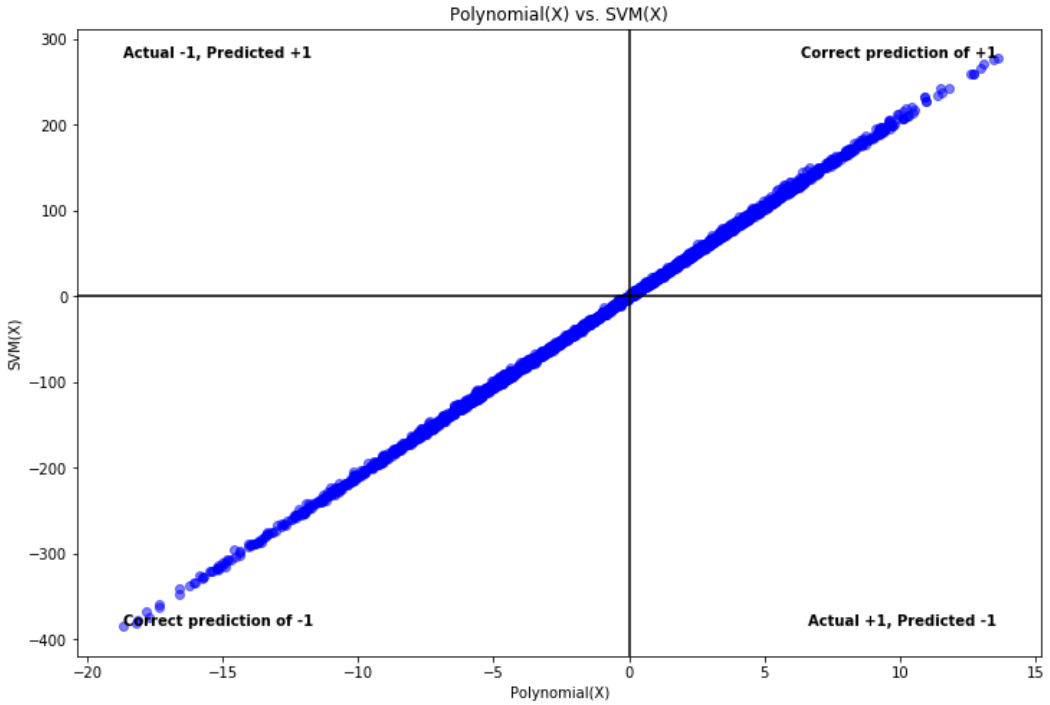


*Figure 25: Scatter plot of SVM(x) vs. Poly(x) for training set*

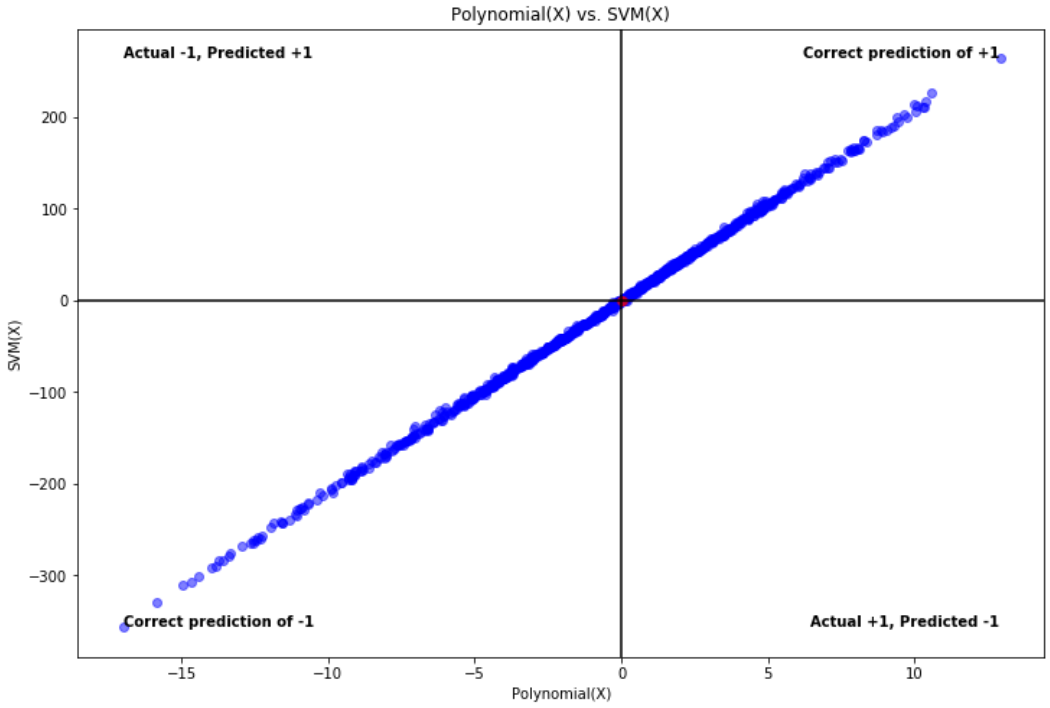


*Figure 26: Scatter plot of SVM(x) vs. Poly(x) for training set*

**Summary**

In summary, lets look at the best two model's tables and see if we can decide which model is better for us. Looking at the tables below you can see that the polynomial kernel, with a $Coef_0 = 0,\ C = 1,\ and\ d = 4$, is slightly better than the radial kernel, with a $\gamma = .1\ and\ C = 35$, on the training set. This is simply because it was 100% accurate and with 100% accuracy the marginal error is 0%. But for the test set, there is really not any significant difference between the

two models, and this is the accuracy we are most interested in because we want the accuracy on predictions of data the model has not already seen.

| Poly (Train) | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| -1 | 100.0 | 100.0 | 100.0 |
| 1 | 100.0 | 100.0 | 100.0 |
| Global | 100.0 | 100.0 | 100.0 |

| Radial (Train) | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| -1 | 99.2 | 99.4 | 99.6 |
| 1 | 99.3 | 99.5 | 99.7 |
| Global | 99.2 | 99.5 | 99.7 |

| Poly (test) | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| -1 | 100.0 | 100.0 | 100.0 |
| 1 | 99.2 | 99.6 | 100.0 |
| Global | 99.5 | 99.8 | 100.0 |

| Radial (Test) | Lower Limit (%) | Percent Correct (%) | Upper Limit (%) |
|---|---|---|---|
| -1 | 99.2 | 99.6 | 99.99 |
| 1 | 98.7 | 99.2 | 99.8 |
| Global | 98.9 | 99.4 | 99.9 |

*Table 29*: Percent Confusion Matrix Comparison for best two models

With that being said, I would go with the polynomial model simply because we know the explicit formula used a polynomial of degree 4. In real life we would not know this, and we may start looking at different factors such as speed of algorithm, or the plots of svm(x) vs poly(x) to get an idea of confidence and distance from hyperplane to make our decision. Looking at the below tables you will see the times it took for the different sections of the code. A thing to note is that this does not include the time of the iterative process involved in tuning the parameters. This includes the time of running the untuned model and the tuned model for each kernel. The 'Create Data' column is the time it took to create the data set, reduce the data set, then split the data set into test and train.

| Create Data | Linear Kernel | Radial Kernel | Polynomial Kernel |
|---|---|---|---|
| 0.18 | 3.38 | 2.7 | 4.96 |

*Table 30*: Time record of each Kernel and dataset creation

You can see that the radial kernel is quite a bit faster than the polynomial kernel, this could be an attractive attribute of using the radial kernel, especially if we start dealing with larger data sets. Another thing that I think is interesting to look at is the difference in the training set plots of SVM(x) vs Poly(x) between the 3 different kernels, refer to the plots below.

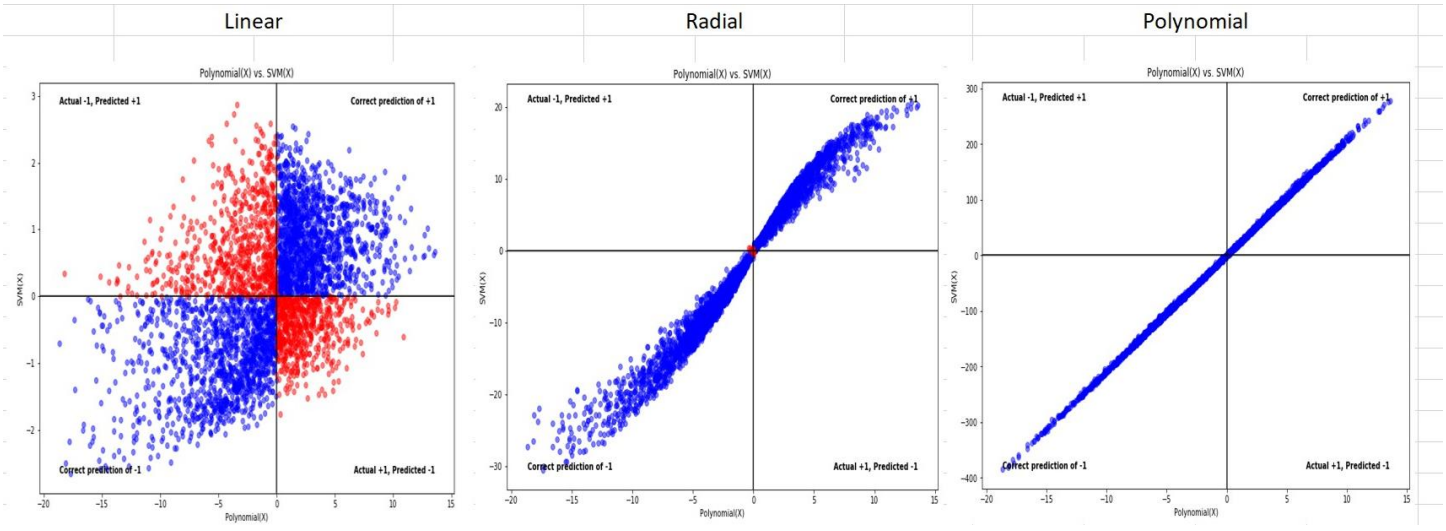

*Figure 27: Scatter plot of SVM(x) vs. Poly(x) comparison for each kernel's optimized training set*

Looking at it this way you can see the difference in confidence of the plots and even how the scale on the y-axis changes from model to model. You will notice that the x-axis does not change because Poly(x) doesn't change when using the same observations, which is our case because these are all plots of the training set. So, the difference in these plots strictly from the SVM(x) values for each kernel, which is the distance from the hyperplane. With the linear model you can see all of the observations on the left side are supposed to be in the -1 class and the ones on the right side are supposed to be in the +1 class. The SVM(x) creates a hyperplane that is supposed to mimic this explicit separator, and that is plotted as the y-axis. Meaning the observations in the bottom half are the predicted -1 class and the observations in the upper half are the predictions in +1 class. This is the only axis that has any shifting going on. Because of this, as the model gets better you can see how the SVM(x) values tend to mimic the Poly(x) values creating this convergence of values on the 45-degree line going across the graph in the polynomial plot.

Overall, I feel that walking through this optimization process to visualize the process of fine-tuning bias and variance trade off was very beneficial in understanding what is going on in the built in functions for tuning. I feel confident in using either the radial or polynomial kernel for this data set for different reasons. I look forward to walking through a real data set and using the built-in tune functions for the next part.

Dustin Vasquez – 1917828

## CODE:

```
# -*- coding: utf-8 -*-
"""
Created on Fri Nov  1 07:32:23 2019

@author: Dustin
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import copy
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
import time


timea = time.time()
times = [time.time()]
################################################################################
#PART ONE: GENERATE DATA BY SIMULATIONS {random numbers = uniform distribution over the interval [-2, +2]}
################################################################################

#STEP 1

np.random.seed(seed = 229)
#np.random.seed(seed = 313)
#select 16 random numbers Aij with i= 1 2 3 4 and j = 1 2 3 4
A = np.random.uniform(-2,2,size = (4,4))

#select 4 random numbers Bi with i= 1 2 3 4
B = np.random.uniform(-2,2, size = 4)

#select 1 random number c (c/20)
C = np.random.uniform(-2,2)/20

#display the values of these random numbers

#Define the polynomial of degree 2 in the 4 variables x1 x2 x3 x4 as follows

#STEP 2

#select 10, 000 vectors x1 ... x10,000 in R4
```

Dustin Vasquez – 1917828

```
X = np.random.uniform(-2,2, size = (10000, 4))

#for each selected xn compute U(n) = Pol(xn) and y(n) = sign[U(n)]
#  Do a for loop because it was faster than Numpy
U = []
for row in range(0,10000):
   u = np.dot(np.dot(X[row,:].T , A), X[row,:]) + np.dot(X[row,:], B) + C/20
   U += [u]
#Pol(x) = ∑i ∑j Aij xi xj + ∑ i Bi xi + c/20

del row, u

u = pd.Series(U)
print('\npositive:', len(u[u>0]))
print('negative:', len(u[u<0]))

#get y into 1 or -1
y = pd.Series(copy.deepcopy(U))
y[y>0] = 1; y[y<0] = -1

#turn into data frame
data = np.insert(X,4,[y,U],axis = 1)
data = pd.DataFrame(data, columns = [0,1,2,3,'y','U'])

#keep only 2500 cases in CL(1) and 2500 cases in CL(-1),
d = pd.concat([data[data['y']>0].sample(n=2500, axis = 0, random_state = 229),
               data[data['y']<0].sample(n=2500, axis = 0, random_state = 229)]
               , axis = 0)
d = d.reset_index(drop = True)
print('\nGreater than 0 for y, U:', len(d[d['y']>0]),len(d[d['U']>0]), '\n',
     'Less than 0 for y, U:', len(d[d['y']<0]),len(d[d['U']<0]))
U_prescale = pd.Series(d['U'])
#d.describe()
#              0          1  ...         y           U
#count  5000.000000  5000.000000  ...  5000.0000  5000.000000
#mean     -0.032034    -0.012180  ...     0.0000    -0.641322
#std       1.148448     1.148988  ...     1.0001     4.693225
#min      -1.999433    -1.998371  ...    -1.0000   -18.668459
#25%      -1.030560    -0.992777  ...    -1.0000    -3.403314
#50%      -0.031247    -0.021481  ...     0.0000    -0.000209
#75%       0.946375     0.984573  ...     1.0000     2.273674
#max       1.998916     1.998722  ...     1.0000    13.594242
```

Dustin Vasquez – 1917828

```
#Center and Rescale  this data set of size 5000  so that the standardized
#data set will have mean = 0 and dispersion =1
#d_scale = pd.DataFrame(preprocessing.scale(d), columns = [0,1,2,3,'y','U'])
d_scale = pd.DataFrame(preprocessing.scale(d), columns = [0,1,2,3,'y','U'])
print('\nGreater than 0 for y, U:', len(d_scale[d_scale['y']>0]),len(d_scale[d_scale['U']>0]), '\n',
     'Less than 0 for y, U:', len(d_scale[d_scale['y']<0]),len(d_scale[d_scale['U']<0]))
#do not carry scaled U because the mean isnt centralized, it is a skewed frequency
d_scale['U'] = U_prescale
x = d_scale[[0,1,2,3]]
y = d_scale['y']
#d_scale['U'] = U_prescale
odescription = d.describe()
description = d_scale.describe()
#                0           1  ...         y          U
#count  5.000000e+03  5.000000e+03  ...  5000.0000  5000.000000
#mean  -1.980624e-16  3.219647e-17  ...     0.0000    -0.641322
#std    1.000100e+00  1.000100e+00  ...     1.0001     4.693225
#min   -1.713265e+00 -1.728817e+00  ...    -1.0000   -18.668459
#25%   -8.695433e-01 -8.535299e-01  ...    -1.0000    -3.403314
#50%    6.850186e-04 -8.095595e-03  ...     0.0000    -0.000209
#75%    8.520248e-01  8.675927e-01  ...     1.0000     2.273674
#max    1.768606e+00  1.750326e+00  ...     1.0000    13.594242

#Then Split each class into a training set and a test set ,
#using the proportions 80% (2000) and 20% (500)
test = pd.concat([d_scale[d_scale['y']>0].sample(n=500, axis = 0, random_state = 229),
                  d_scale[d_scale['y']<0].sample(n=500, axis = 0, random_state = 229)]
                  , axis = 0)
train = d_scale.drop(index = test.index.tolist())
test = test.reset_index(drop = True); train = train.reset_index(drop = True)

#Plot two elements of the data
#first get highest correlation
d_scale[[0,1,2,3,'y']].corr()['y']

fig, ax = plt.subplots(figsize = (12,8))
plt.scatter(x = d_scale[[0]][d_scale['y']>0], y = d_scale[[2]][d_scale['y']>0],
          color = 'blue')
plt.scatter(x = d_scale[[0]][d_scale['y']<0], y = d_scale[[2]][d_scale['y']<0],
          color = 'red')
plt.title('Two dimensional View of Data')
plt.xlabel('Feature 0')
plt.ylabel('Feature 2')
```

```
times += [time.time()-timea]
timea = time.time()
###############################################################################
#PART TWO: SVM classification by linear kernel
###############################################################################

from SVM import SVM

mySVM = SVM()

#SVM on Training set, linear kernel
mySVM.set_kernel(kernel = 'linear')
#run the function
n_sv, sv, y_predict, score, conf_int, dist_hp, model = \
   mySVM.run_svm(train.drop(columns = ['y','U']), train['y'], C=5, random_state = 9989)

#Get density histrograms of accuracy on train
mySVM.get_hist(dist_hp, train['y'])

#plot polynomial vs. svm
mySVM.get_scatter(train['U'], dist_hp)

#get confusion matrix for the training set
train_conf, train_per_conf, train_limits, train_confidence = mySVM.get_confusions(train['y'], y_predict,
conf_desired = .95)

#SVM on test set
test_predict, test_score, test_conf_int, test_dist_hp  = \
   mySVM.run_svm(test.drop(columns = ['y','U']), test['y'], C=5, random_state = 9989,
            train = 'no', model = model)

#plot polynomial vs. svm
mySVM.get_scatter(test['U'], test_dist_hp)

#get confusion matrix on test set and limits
test_conf, test_per_conf, test_limits, test_confidence = mySVM.get_confusions(test['y'], test_predict,
conf_desired = .95)

#look at these two values to verify what column means what in confusion matrix
print('\nValue Counts in response:\n', pd.Series(test_predict).value_counts())
print('\nSum of columns in conf matrix:',test_conf.sum(axis = 0))
```

Dustin Vasquez – 1917828

```
################################################################################
#PART THREE: Optimize the parameter cost
################################################################################

#Tune the cost
mySVM = SVM()

#SVM on Training set, linear kernel
mySVM.set_kernel(kernel = 'linear')

#run through the different tunes: #1
C_scores, C_n_sv = mySVM.tune_svm(train.drop(columns = ['y','U']), \
                        train['y'], test.drop(columns = ['y','U']), test['y'], \
                        C=5, random_state = 9989, tune = 'C', \
                        params = [.00001, .001, 1, 10, 30, 50])

#run through the different tunes
C_scores, C_n_sv = mySVM.tune_svm(train.drop(columns = ['y','U']), \
                        train['y'], test.drop(columns = ['y','U']), test['y'], \
                        C=5, random_state = 9989, tune = 'C', \
                        params = [.001, .01, .1, 1, 1.5, 2])



##print the different scores for each c
#print(max(C_scores.values()), C_scores.keys()[max(C_scores)]))
best_c = .1

mySVM = SVM()

#SVM on Training set, linear kernel
mySVM.set_kernel(kernel = 'linear')
#run the function
n_sv, sv, y_predict, score, conf_int, dist_hp, model = \
        mySVM.run_svm(train.drop(columns = ['y','U']), train['y'], C=best_c,
                random_state = 9989)

#Get density histrograms of accuracy
mySVM.get_hist(dist_hp, train['y'])

#get confusion matrix for the training set
train_conf, train_per_conf, train_limits, train_confidence = mySVM.get_confusions(train['y'], y_predict,
conf_desired = .95)
```

```
#plot polynomial vs. svm
mySVM.get_scatter(train['U'], dist_hp)

#Do test run
#SVM on test set
test_predict, test_score, test_conf_int, test_dist_hp  = \
   mySVM.run_svm(test.drop(columns = ['y','U']), test['y'], C=5, random_state = 9989,
            train = 'no', model = model)

#plot polynomial vs. svm
mySVM.get_scatter(test['U'], test_dist_hp)

#get confusion matrix on test set and limits
test_conf, test_per_conf, test_limits, test_confidence = mySVM.get_confusions(test['y'], test_predict,
conf_desired = .95)

times += [time.time()-timea]
timea = time.time()
###############################################################################
#PART FOUR:  SVM classification by radial kernel
###############################################################################

radSVM = SVM()

#SVM on Training set, linear kernel
radSVM.set_kernel(kernel = 'rbf', gamma = 1)

#run the function
n_svr, svr, yr_predict, score_r, conf_r, dist_hp_r, model_r = \
   radSVM.run_svm(train.drop(columns = ['y', 'U']), train['y'], C=best_c,
            random_state = 9989, train = 'yes')

#Get density histrograms of accuracy
radSVM.get_hist(dist_hp_r, train['y'])

#get confusion matrix for the training set
train_conf_r, train_per_conf_r, train_limits, train_confidencer = radSVM.get_confusions(train['y'], yr_predict,
conf_desired = .95)

#plot polynomial vs. svm
radSVM.get_scatter(train['U'], dist_hp_r)

#SVM on test set
```

```
test_predict_r, test_score_r, test_conf_r, test_dist_hp_r = radSVM.run_svm(test.drop(columns = ['y', 'U']),
                    test['y'], C=best_c, random_state = 9989, train = 'no',
                    model = model_r)

#get confusion matrix on test set
test_conf_r, test_per_conf_r, test_limitsr, test_confidencer = radSVM.get_confusions(test['y'], test_predict_r,
conf_desired = .95)

#plot polynomial vs. svm
radSVM.get_scatter(test['U'], test_dist_hp_r)

################################################################################
#PART FIVE:  Optimize cost and gamma
################################################################################

#Tune the radial gamma and cost
tune_radSVM = SVM()

#SVM on Training set, linear kernel
tune_radSVM.set_kernel(kernel = 'rbf')

#run through the different tunes - first iteration
gamma_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=best_c, random_state = 9989, tune = 'gamma',
                        params = [.00001, .001, .01, .1, .5, 1])

#run through the different tunes - 2nd iteration
gamma_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y','U']), test['y'],
                        C=best_c, random_state = 9989, tune = 'gamma',
                        params = [.05, .075, .1, .25, .45, .65, .75])

"""
looks like gamma of .1 is the best for both Test and Train Set \
Use gamma = 0.45 and iterate through the cost
"""

tune_radSVM = SVM()

#SVM on Training set, linear kernel
tune_radSVM.set_kernel(kernel = 'rbf', gamma = .45)
```

```
C_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=5, random_state = 9989, tune = 'C',
                        params = [.00001, .01, 10, 35, 70, 100])

C_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=5, random_state = 9989, tune = 'C',
                        params = [.00001, .001, 1, 10, 30, 50])

C_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=5, random_state = 9989, tune = 'C',
                        params = [15, 20, 25, 30, 35, 40, 45])



"""
Looks like cost of 35 was the highest and closest match between test and train \
We will re-iterate through gamma using a cost of 30, with focus around previous \
gamma.
"""

tune_radSVM = SVM()

#SVM on Training set
tune_radSVM.set_kernel(kernel = 'rbf')

#run through the different tunes with best cost to make sure didnt change
gamma_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=35, random_state = 9989, tune = 'gamma',
                        params = [.05, .075, .1, .25, .45, .65, .75])

#SVM on Training set, linear kernel
tune_radSVM.set_kernel(kernel = 'rbf', gamma = .1)
C_scores, n_sv_gamma = tune_radSVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=5, random_state = 9989, tune = 'C',
                        params = [8, 15, 22, 30, 37, 45, 50])

"""
Run: gamma = 0.1, C = 35
"""
```

```
#run tuned radial SVM
C=35; gamma = 0.10

tune_radSVM = SVM()

#SVM on Training set
tune_radSVM.set_kernel(kernel = 'rbf', gamma = gamma)

n_svrt, svrt, yrt_predict, score_rt, conf_int_rt, dist_hp_rt, model_rt = \
   tune_radSVM.run_svm(train.drop(columns = ['y', 'U']), train['y'], C=C,
            random_state = 9989)

#Get density histrograms of accuracy
tune_radSVM.get_hist(dist_hp_rt, train['y'])

#get confusion matrix for the training set
train_conf_rt, train_per_conf_rt, limits_rt, train_confidence_rt = tune_radSVM.get_confusions(train['y'],
yrt_predict)

#plot polynomial vs. svm
tune_radSVM.get_scatter(train['U'], dist_hp_rt)

#SVM on test set
test_predict_rt, test_score_rt, test_conf_int_rt, test_dist_hp_rt = tune_radSVM.run_svm(test.drop(columns =
['y', 'U']),
                    test['y'], C=C, random_state = 9989, train = 'no',
                    model = model_rt)

#get confusion matrix on test set
test_conf_rt, test_per_conf_rt, test_limits_rt, test_confidence_rt = tune_radSVM.get_confusions(test['y'],
test_predict_rt)

#plot polynomial vs. svm
tune_radSVM.get_scatter(test['U'], test_dist_hp_rt)

times += [time.time()-timea]
timea = time.time()
###############################################################################
#PART SIX:  Run Polynomial with degree = 4, optimize Coeff and Cost
###############################################################################

"""
```

Dustin Vasquez – 1917828

```
Since we know degree is 4, lets get an initial look with default parameters
"""

poly_SVM = SVM()

#set polynomial
poly_SVM.set_kernel('poly', degree = 4)

#run poly SVM
n_svp, svp, yp_predict, score_p, conf_p, train_dist_hp_p, model_p = \
   poly_SVM.run_svm(train.drop(columns = ['y', 'U']), train['y'], C = 1,
            random_state = 9989, train = 'yes')

#Get density histrograms of accuracy
poly_SVM.get_hist(train_dist_hp_p, train['y'])

#get confusion matrix for the training set
train_conf_p, train_per_conf_p, train_limits_p, train_confidence_p = poly_SVM.get_confusions(train['y'],
yp_predict)

#plot polynomial vs. svm
poly_SVM.get_scatter(train['U'], train_dist_hp_p)

#SVM on test set
test_predict_p, test_score_p, test_conf_p, test_dist_hp_p = poly_SVM.run_svm(test.drop(columns = ['y', 'U']),
                  test['y'], C=1, random_state = 9989, train = 'no',
                  model = model_p)

#get confusion matrix on test set
test_conf_p, test_per_conf_p, test_limits_p, test_confidence_p = poly_SVM.get_confusions(test['y'],
test_predict_p)

poly_SVM.get_scatter(test['U'], test_dist_hp_p)

"""
Tune the parameters Coeff and Cost. Leave Degree = 4
"""
tpoly_SVM = SVM()

#set polynomial
tpoly_SVM.set_kernel('poly', degree = 4)

pt_scores, n_sv_pt = tpoly_SVM.tune_svm(train.drop(columns = ['y', 'U']),
```

```
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=1, random_state = 9989, tune = 'coef0',
                        params = [.001, .1, 10, 30, 60, 100, 200])

tpoly_SVM = SVM()

#set polynomial
tpoly_SVM.set_kernel('poly', degree = 4)

pt_scores, n_sv_pt = tpoly_SVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=1, random_state = 9989, tune = 'coef0',
                        params = [15, 20, 25, 30, 35, 40, 45])

"""
Coef of 30 and 50 were equally best on Test. 30 was 100% on Train, 50 was 99.975%.\
Use Coef 30, change params for Cost now.
"""
tpoly_SVM = SVM()

#set polynomial
tpoly_SVM.set_kernel('poly', degree = 4, coef0 = 30)

pt_scores, n_sv_pt = tpoly_SVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=1, random_state = 9989, tune = 'C',
                        params = [.001, .1, 25, 50, 75, 100, 125])

tpoly_SVM = SVM()

#set polynomial
tpoly_SVM.set_kernel('poly', degree = 4, coef0 = 30)

pt_scores, n_sv_pt = tpoly_SVM.tune_svm(train.drop(columns = ['y', 'U']),
                        train['y'], test.drop(columns = ['y', 'U']), test['y'],
                        C=1, random_state = 9989, tune = 'C',
                        params = [.1, 1, 5, 10, 15, 20, 25])

"""
Looks like best cost is 1, though they are so close it is not really different.\
Use Cost = 1, coef0 = 30, degree = 4 and run_SVM
"""
tpoly_SVM = SVM()
```

```
#set polynomial
tpoly_SVM.set_kernel('poly', degree = 4, coef0 = 30)

#run poly SVM
n_svpt, svpt, ypt_predict, score_pt, train_conf_pt, dist_hp_pt, model_pt = \
   tpoly_SVM.run_svm(train.drop(columns = ['y', 'U']), train['y'], C = 1,
            random_state = 9989, train = 'yes')

#Get density histrograms of accuracy
tpoly_SVM.get_hist(dist_hp_pt, train['y'])

#get confusion matrix for the training set
train_conf_pt, train_per_conf_pt, train_intervals_pt, train_confidence_pt =
tpoly_SVM.get_confusions(train['y'], ypt_predict)

#plot polynomial vs. svm
tpoly_SVM.get_scatter(train['U'], dist_hp_pt)

#SVM on test set
test_predict_pt, test_score_pt, test_conf_pt, test_dist_hp_pt = tpoly_SVM.run_svm(test.drop(columns = ['y',
'U']),
                    test['y'], C=1, random_state = 9989, train = 'no',
                    model = model_pt)

#get confusion matrix on test set
test_conf_pt, test_per_conf_pt, test_intervals_pt, test_confidence_pt = poly_SVM.get_confusions(test['y'],
test_predict_pt)

#plot polynomial vs. svm
tpoly_SVM.get_scatter(test['U'], test_dist_hp_pt)

times += [time.time()-timea]
timea = time.time()
##############################################################################
#Time
##############################################################################

print(times)
```