Assignment 3: kNN and SVM - Machine Learning 6316

Jacob Dineen / JD5ED

October 21, 2019

1 KNN and Model Selection (k) (programming)

This first block deals with the k-nearest-neighbors algorithm and its implementation from scratch, mainly using numpy for vectorized operations. We are provided, as per usual, a template that we were required to fill out that would eventually get us to the point where we would run k-fold cross validation while iterating through a list containing scalar values deciding the prospective decision boundary for each model, which I'll get into a bit later re: model complexity and bias-variance trade-off. The data is high dimensional (in regards to p features)- There are 2000 observations and 100 features for each which makes exploratory analysis/visualization a bit difficult. It should be noted that we are dealing with perfect class balance, meaning that a majority vote model (a model that were to guess the same class for each instance) would register 50 percent accuracy. We'll use this as a threshold for now, before turning to Sklearn KNN to understand/debug any of the code that was written.

Num Features 100
Num Obs: 2000
Count of Postive Class labels: 1000
Count of Negative Class labels: 1000

Code Explanation The first method that we were to implement was to return a specified fold (a set within a set of sets) that would give us our respective train test splits. I borrowed most of the skeleton from my Ridge Regression implementation, but did alter it so that I only returned a specified fold according to the parameter *i*. There is moderate error handling included that should deal with instances where the number of specified folds splits the data into bins with a remainder. In this case, much like in the last assignmnet, the 'extras' are thrown into the last bins and trained on.

We'll be using Euclidean or L2 Distance as our scoring metric throughout this portion of the assignment: $d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$, but note that this is an imperfect distance measurement - A feature with more variance may have an unreasonable impact on the model output. I have worked with the Census dataset in the past (problem 2, to come) and I believe that standardizing/normalizing/scaling data works well in cases where you may have a feature like Salary that has a range from (0, 100,000) and a number of binary features. I have used cosine similarity in the past to distance between two encoded feature vectors representing some notion of text (tf, tfidf), but never for strictly continuous data.

The classify method returns an array containing all of the predictions on the test test. We have to run the fold method first to generate the train test splits, which we do in the main loop. This method loops through each instance of the test set, computes all of the distances from itself to each point in the train set, and ranks those distances. We then have to see which responses were most common amongst the k neighbors. I chose to deal with odd valued k's throughout this assignment - If there was an even count amongst negative and positive classes, there is usually some design heuristic that dictates which class mapping to perform. With an odd number k, you

marginalize this issue. The code, as written, takes the k nearest neighbors from the sorted list of distances and returns the argmax response from the dictionary. I used a loop initially to compute accuracy, but realized that numpy was much quicker, so we just calculate the sum of matches between two arrays (ypred, ytrue) and return the result divided by the length of the prediction array. Barplot is a simple plot showing the number of neighbors against the cross validation accuracy.

The last method is the heart of the program - findBestK. This was subject to a bit of interpretation, as the *i* parameter in the fold method was a bit ambiguous. What I decided to do was add a few more variables and counters, along with some nested logic, to perform this in the following manner:

iterate through the list of k (neighbors)
  iterate through each fold of nfolds
    split the data into train test sets
    perform inference according to k
    calculate accuracy

VAR: accuracylist will contain the mean CV accuracy across all folds as per the second loop. VAR: accuracy will contain the mean cv accuracy at the current time step. If the accuracy at timestep *t* is greater than the best accuracy, the best accuracy will get updated. This wasn't incredibly clear when reading through the instructions, but I assume this is what was desired - We are, for each k, computing the mean of the accuracies for each fold within the master set of splits.

Results

A model improving over the majority vote threshold will be useful in proving that the code was accurately written, although I am not entirely sure of what constitutes a great score on this particular data set. We score all test data using k fold cross validation with $k \in \{4, 8, 10\}$, not to be confused with k determining the number of neighbors used to calculate the majority vote response.

Number of neighbors / CV Score, Folds = 4

Neighbors: 3 , cv accuracy .574
Neighbors: 5 , cv accuracy .594
Neighbors: 7 , cv accuracy .614
Neighbors: 9, cv accuracy .616
Neighbors: 11, cv accuracy .624
Neighbors: 13, cv accuracy .612
Best k: 11/13 *dependent on randseed

Number of neighbors / CV Score, Folds = 8
Neighbors: 3, cv accuracy 0.572
Neighbors: 5, cv accuracy 0.616
Neighbors: 7, cv accuracy 0.628
Neighbors: 9, cv accuracy 0.632
Neighbors: 11, cv accuracy 0.644
Neighbors: 13, cv accuracy 0.628
Best k: 11 or 13 *dependent on randseed

Number of neighbors / CV Score, Folds = 10
Neighbors: 3, cv accuracy 0.58
Neighbors: 5, cv accuracy 0.625
Neighbors: 7, cv accuracy 0.61
Neighbors: 9, cv accuracy 0.63
Neighbors: 11, cv accuracy 0.63
Neighbors: 13, cv accuracy 0.63

Best k: 11 or 13 *dependent on randseed

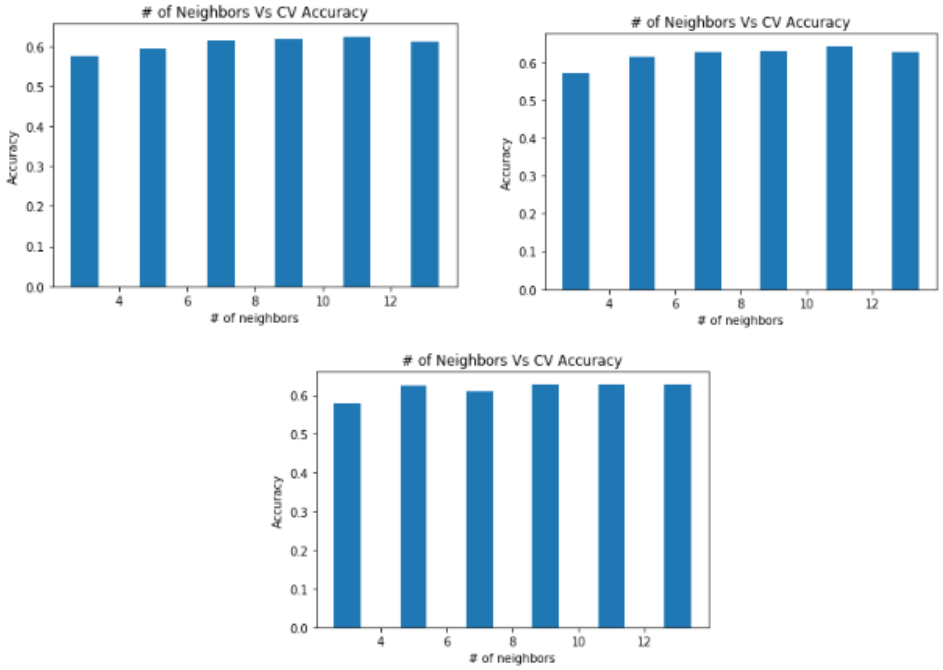


Figure 1: CV Accuracy vs Number of Neighbors. Left to Right: Number of Folds $\in \{4, 8, 10\}$

I ran this through a number of different k-fold schemes (4,8,10), and the best number of neighbors hyper parameter was usually 11 across the broad. I transferred this code from an ipython notebook and hadn't been using the same random seed, so performance may vary slightly from knn.py. Performance generally increased with the size of this parameter. Let's take our best model, which was found using 11 neighbors and 8-fold validation, and compare it to Sklearn's KNN implementation - We'll just use a simple train test split of 0.8 and fix the number of neighbors to 11, for brevity.

Ours: Neighbors: 11, cv accuracy 0.644
Sklearn: Neighbors: 11, Test accuracy 0.626

```
]: from sklearn.neighbors import KNeighborsClassifier
   from sklearn.model_selection import train_test_split
   model = KNeighborsClassifier(11)
   x_train, x_test, y_train, y_test = train_test_split(x,y)
   model.fit(x_train, y_train)
   preds = model.predict(x_test)
   calc_accuracy(preds, y_test)
   executed in 111ms, finished 17:18:11 2019-10-13
]: 0.626
```

Figure 2: Rudimentary implementation of SKlearn's KNN

The comparison isn't apples to apples, but this shows that the implementation from scratch is likely coded correctly. With more feature engineering and additional model selection techniques, the accuracy for both models would likely improve. This is saved as knn.py.

KNN is a lazy learning algorithm, in the sense that we aren't learning a functional mapping from the input to

the output space. All of the compute is done come inference time (polynomial time complexity). Training the model is done in constant time. We have implemented a KNN classifier, although a similar methodology, as above, could be taken to turn this into a regressor (compute and return the conditional mean of k nearest neighbors). There is an inverse relationship between the model complexity of the KNN algorithm and the parameter chosen for the number of neighbors. As k increases, the model complexity and the bias of the model decreases, and vice versa. A model with k = 1 will have high variance, represented by an incredibly complex decision boundary. This may result in 100 percent training accuracy, but does not generalize well to unseen data. To loosely quote an example from class, a high variance model is such that if we were to remove one single data point from the train set and retrain the model, we would get a drastically different decision boundary, i.e. our model's decision boundary is incredibly sensitive/elastic to change. As seen above, increasing the size of k, and decomplexifying the model, resulted in better cross-fold validation accuracy, which could be used as an expectation on the unseen test dataset. Our final choice of k was relatively small in relation to the size of the test set (particularly with 4-fold CV), so we aren't too worried that our neighborhood is so large as to provide an innacurate estimate of the responses in the neighboring areas of our input space. What we didn't do, but would likely result in a better classify: experiment with some of the variants of KNN, e.g. weighting neighboring responses based on proximity, experiment with feature scaling/feature engineering to mitigate the impact of high-range features.

2 Support Vector Machines with Scikit-Learn and preprocessing

This next task deals with a kaggle-style competition, in which we are to try to perform inference on some hold out test set in hopes of finishing in the top 20 percent of our class leaderboard. I have worked with the particular dataset (Census) before, and most of the time I achieved better performance relative to the amount of data cleaning and feature engineering performed, as opposed to exhaustive hyperparameter searching - although I haven't used SVMs here. I'll get up the writeup here as brief as possible, simply noting a table with the impending grid search of SVM hyperparams, some of the data cleaning techniques I use, or chose not to use, and a few of my initial hypotheses on why I believe some parameters work better than other for this particular distribution. As the task mainly centers around model selection, I'll leave out any exploratory data analysis that I do on my own from my final code submission (will comment out in docstrings). It should be noted that we are dealing with a 75/25 negative/positive class imbalance problem here, so that should be our benchmark threshold for model improvement.

My understanding of the PDF was as follows - Although it has been taking quite a while to search about 10 sets of hyperparameters, so I'll likely hardcode in a fixed set for submission to alleviate compute time on your end:

- Load the Data.
- Clean the data, but don't scale on the entire set until it's split into k folds.
- Grid Search on hyperparameters with self-written code
- Run through K fold CV (self-written). Each train fold will have a scaler fit to it. Each test set will be transformed per the scaler.
- Fit model on train set, inference on test set.
- Calculate Mean Train/Test Accuracy across all folds
- Report out Accuracy for each set of hyperparams.
- Select the best CV holdout accuracy model.
- Retrain using the full dataset and the best hyperparams.
- Perform inference on unseen test data for submission to instructors.

My first run at this - after getting the manual grid search / k fold cross validation code off the ground - resulted in the following results (Fig 3). Some of the initial preprocessing I did:

- Used Pandas to read in the data. This is important, as I had to tinker with my old K-Fold Validation code to make this work.
- Binarize the target variable.
- Map all non 'United-States' entries to an 'other' bucket. This was mainly done to speed up model fitting.
- Removed any data from the numerical columns (column-wise) that was more than three standard deviations from the mean (outlier detection/removal).
- Encode the categorical features using pandas.
- Fit a scaler to the trainset for each fold when hyperparam searching. Then transform the testset before inference. This helps us in creating a useful model - Scaling the data originally and fitting to it would allow the model to learn the decision boundaries exactly (data leakage) - I did this at first and received 100 percent CV accuracy across the board.

Figure 4 shows the results of my grid search, which mainly consisted of shifting the C (regularization parameter) between [0.1,1,10], the degree between [1,7]* and the kernel between rbf,linear and poly. The results were relatively inelastic for the linear and rbf kernels, with the best model being produced using: C = 100, Degree = 1, Kernel = rbf, which resulted in 85.4 percent 3-fold mean cross validation accuracy on the holdout sets. *I realized that the degree hyperparam. wasn't impacting my cv accuracy (train or test) in any way. I decided to move forward using degree = 1 for all model variants.

Model #	C	Degree	Kernel	CV Train Accuracy	CV Test Accuracy
3	100	1	rbf	0.86	0.854
2	10	1	rbf	0.86	0.853
4	0.1	1	linear	0.85	0.852
10	10	1	poly	0.85	0.852
5	1	1	linear	0.85	0.852
11	100	1	poly	0.85	0.851
6	10	1	linear	0.85	0.851
7	100	1	linear	0.85	0.851
1	1	1	rbf	0.85	0.850
9	1	1	poly	0.85	0.847
0	0.1	1	rbf	0.84	0.838
8	0.1	1	poly	0.83	0.829

Figure 3: Param Search- Light Preprocessing

I then tried the following to reduce the sparsity of the matrix that was generated via encoding the categorical features, which actually did not improve upon the best CV accuracy from above (Fig 5).

More Preprocessing:

- Marital Status into two bins: [single, married]
- Removing native-country entirely
- Mapping race into two bins.
- Mapping relationship into two bins.

Model #	C	Degree	Kernel	CV Train Accuracy	CV Test Accuracy
3	100	1	rbf	0.860	0.854
2	10	1	rbf	0.856	0.853
4	0.1	1	linear	0.852	0.852
1	1	1	rbf	0.852	0.851
10	10	1	poly	0.851	0.851
5	1	1	linear	0.850	0.850
11	100	1	poly	0.850	0.850
6	10	1	linear	0.850	0.850
7	100	1	linear	0.850	0.849
9	1	1	poly	0.848	0.848
0	0.1	1	rbf	0.840	0.839
8	0.1	1	poly	0.836	0.835

Figure 4: Param Search - Heavier Preprocessing

Because of the marginal/incremental improvement, I believe that the 'over-preprocessing' may not be worth it here. My final model will be trained on the 'light' preprocessing noted above. RBF kernel was the most successful, both at fitting the data and generalizing to the holdout. A higher C enforced a smaller margin and fewer misclassifications. In general, when we increase the value of C we are enforcing a tighter margin, which is actually a question below. The way I look at this is that if we have a very tight margin, our model has high variance in that if were to remove a single training point, we'd likely have a noticeably different decision boundary. From some stackexchange posts, I've found that increasing the value of C does in fact lead to a scenario where the boundary is almost entirely decided by the regularization term - We're injecting more bias into our model the lower C we choose.

We can draw some insights from the distribution of our predictions in accordance with the underlying distribution of the response variables presented in our train set. Professor noted some form of semi supervised learning could be used to add knowledge to this step, but I didn't have time to thoroughly dive into those techniques. The distribution of my predictions was 80/20 negative/positive class. I assume this came from the same distribution as the training data (75/25) so I do expect a fair amount of false negatives.

The next step was just wrapping some of this code into main, which involves reading in the unseen test data (with the same preprocessing techniques (minus outlier removal) and performing inference). The output file containing the predictions is stored locally and will be submitted to Collab for rankings.

I ended up circling back to this with the following methodology, as I was a bit hampered by the runtime performance of kfold CV + grid search. For faster model selection, I've down sampled to the first 10k observations of the train set. I've also removed all categorical features to reduce dimensionality. What I found was that: This is definitely quicker - Grid search with Kfold CV searched through 13 sets of params in about 45 seconds with the reduced dimensionality. There was also considerable performance (accuracy) drop when omitting the categorical features (See Fig 5). Fig 6 shows the same, but without omitting cat features, speaking to the loss of information when excluding them. One important thing that I did - Generally, in the past, I've done encoding and cleaning on a full dataset before splitting it. In this case we did it via batches, which caused some headaches come inference time. To solve some of the dimensionality issues that result from doing this, e.g. one hot encoded of the test set resulted in fewer columns than the train set, I added some catching to add null columns not present in the test set that were present in the train set. Also, I had to remove columns from the test set that weren't present during training. Most of these were high level encoded features that didn't have much bearing on the decision boundaries, but I felt it was worth noting. Let's also look at the impact of C on a subset of training examples. As discussed, the limit as C approaches infinity converges to the hard-margin SVM decision boundary, with less margin for error, in terms of misclassification. Figure 7 shows the train and test accuracy for a radial basis kernel while varying C from a small to a very large value. It's fairly noticeable how we begin to over-fit our train set and lose some generalization ability

NOTE: On my code submission, I've downsampled the trainsets to include only the first 10k observations, and

Model #	C	Degree	Kernel	CV Train Accuracy	CV Test Accuracy
0	0.1	1	rbf	0.80	0.80
1	1	1	rbf	0.81	0.81
2	10	1	rbf	0.82	0.82
3	100	1	rbf	0.82	0.82
4	1000	1	rbf	0.82	0.82
5	0.1	1	linear	0.81	0.81
6	1	1	linear	0.81	0.81
7	10	1	linear	0.81	0.81
8	100	1	linear	0.81	0.81
9	0.1	1	poly	0.79	0.79
10	1	1	poly	0.81	0.81
11	10	1	poly	0.81	0.81
12	1000	1	poly	0.81	0.81

Figure 5: Param Search - Removed Cat Features and Downsampled Trainset

Model #	C	Degree	Kernel	CV Train Accuracy	CV Test Accuracy
7	10	1	linear	0.857936	0.854985
12	1000	1	poly	0.858136	0.854685
8	100	1	linear	0.858286	0.854585
3	100	1	rbf	0.870637	0.854085
6	1	1	linear	0.857486	0.853985
2	10	1	rbf	0.859736	0.853885
11	10	1	poly	0.857036	0.853485
5	0.1	1	linear	0.854485	0.851985
4	1000	1	rbf	0.882988	0.849785
1	1	1	rbf	0.849985	0.848385
10	1	1	poly	0.846235	0.844884
0	0.1	1	rbf	0.814381	0.813581
9	0.1	1	poly	0.774777	0.774777

Figure 6: Param Search -Downsampled Trainset

disabled the overwriting of my true prediction file, to allow for better run-time. I feel like this allows for a quicker proof of concept to let you know that my code works, while using the original best model to perform inference.

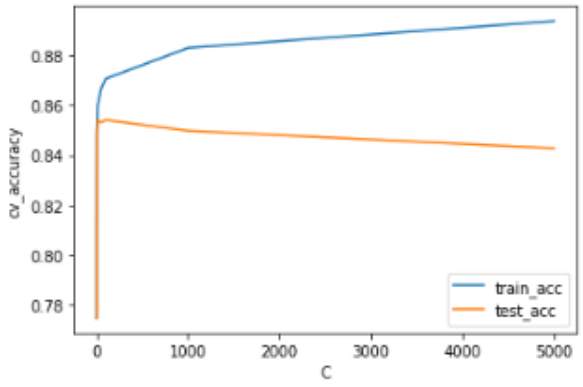


Figure 7: Sensitivity of Accuracy vs C

3 Sample QA Questions:

Question 1. Support Vector Machine
Soft-margin linear SVM can be formulated as the following contrained quadratic optimization problem:
such that

$$y_i \left(w^T x_i + b \right) \geq 1 - \epsilon_i$$

where C is the regularization parameter, which balances the margin (smaller $w^T w$) and the penalty of mis-classification (smaller $\sum_{m=1}^{m} \epsilon_i$)

3.1

(a) (True/False) Number of support vectors do not depend on the selected value of C. Please provide a one-sentence justification.

There was a slide that Professor Qi noted that said there was not a clear relationship between C and the number of support vectors, which would make this statement true.

My initial thinking was as follows: A soft-margin linear SVM is an extension of a hard-margin SVM, except we introduce this idea of slack variables to allow for inseparable data to fall on the 'wrong' side of the decision boundary, e.g. penalizing misclassified instances during training. C is a (regularization) parameter to control the trade-off between minimizing training accuracy and 'controlling' model complexity [Bishop]. A large value of C results in a smaller margin, and a smaller value of C results in a larger margin. Thus, the number of support vectors **could** be dependent on the selected value of C - false could be the answer to the question if this were always the case, but it could also just depend on the data, so I think that's what is meant by 'no clear relationship'.

3.2

In the following figure, $n_{C=0}$ is the number of support vectors for C getting close to 0 ($\lim_{C \to 0}$) and $n_{C=\infty}$ is the number of support vectors for C gets close to ∞ .

(1) $n_{C=0} > n_{C=\infty}$
(2) $n_{C=0} < n_{C=\infty}$
(3) $n_{C=0} = n_{C=\infty}$
(4) Not enough information provided

This is also subject to the interpretation noted in the first line above about the Professor's slides from the final SVM lecture which noted no analytical relationship between C and the number of support vectors, which would likely make this answer (4).

My thinking was that as you increase the value of C, the margin decreases. Decrease C, the margin increases. A high value of C means that we want to minimize the number of training errors/margin violations noted in the previous example. As C approaches infinity, we converge to the hard margin solution (assuming data is linear separably). The correct answer could be (1), if this were always true - In this case, because we would be dealing with a soft margin classifier as the data is not linearly separable, e.g. we would have to increase the amount of regularization we inject into our model to loosen the constraints that all data points fall on the correct side of the plane. Doing so would result in atleast a single additional non-zero a_i. *I tested this empirically on the Iris Dataset with various C to confirm, re: Figure 4. I do believe that the underlying distribution of the training will impact how you'd want to tune C.

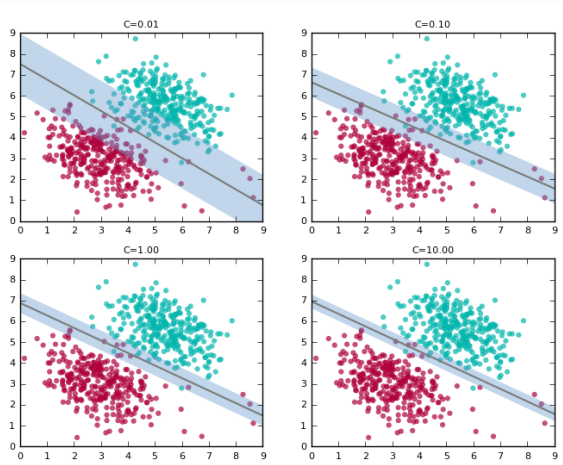


Figure 8: SVM w/ various C - blog.statsbot.co/

```
from sklearn import svm, datasets
# import some data to play with
iris = datasets.load_iris()
X = iris.data[:, :1]  # we only take the first two features. We could
                      # avoid this ugly slicing by using a two-dim dataset
y = iris.target
C = 0.00001
for i in [1e-5, 1e-1, 1, 1e5, 1e10]:
    svc = svm.SVC(kernel='linear', C=i).fit(X, y)
    print('C: {} support vectors: {}'.format(i, len(svc.support_vectors_)))
```

executed in 804ms, finished 19:32:13 2019-10-13

```
C: 1e-05 support vectors: 150
C: 0.1 support vectors: 126
C: 1 support vectors: 107
C: 100000.0 support vectors: 79
C: 10000000000.0 support vectors: 70
```

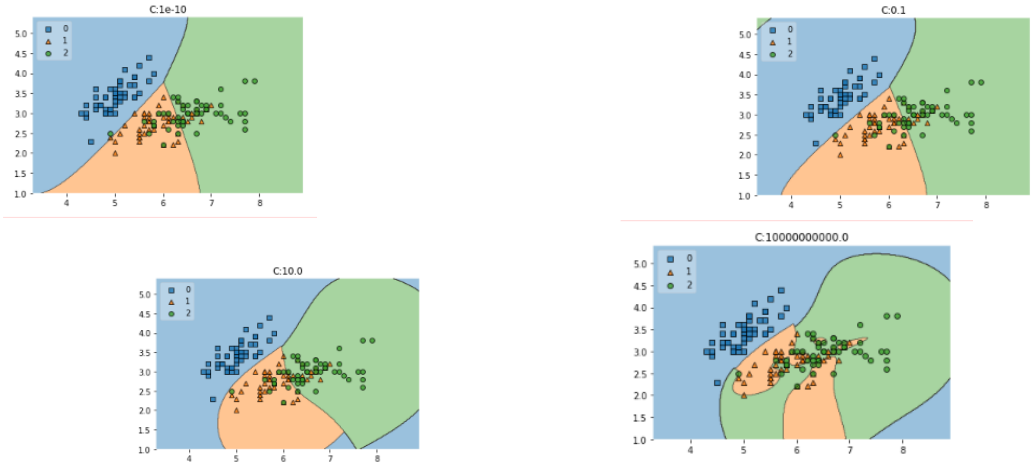


Figure 9: SVM w/ various C - Iris Decision Boundaries

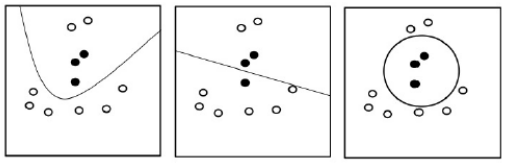
3.3

(c) Match the following list of kernels used for SVM classification with the following figures:

(1) Linear Kernel: $K(x, x') = x^T x'$

(2) Polynomial Kernel (order $= 2$) : $K(x, x') = \left(1 + x^T x'\right)^2$

(3) Radial Basis Kernel: $K(x, x') = \exp\left(-\frac{1}{2}\|x - x'\|^2\right)$



First Picture: Polynomial Kernel
Second Picture: Linear Kernel
Third Picture: Radial Basis Kernel