Classifying Poker Hands

Project Repo: <https://github.com/DanielDeMarco/AIFinalProject>

# Group Members

* Michael Neas - mrn11001
* Sean Bridges - smb10013
* Angela Onofrio - amo13008
* Cindy Lin - cil11002
* Nick Barber - njb08005
* Edward Hanlon -  ejh12002
* Daniel DeMarco - dad13001

Introduction

# Objective

We wanted to do something different than an EEG or smartphone based project.  The problem we wish to address as an AI project dealing with the game of Poker. We found a dataset (<https://archive.ics.uci.edu/ml/datasets/Poker+Hand>) containing over 1 million unique 5 card poker hands (considering order, suit, and rank). We used it to develop a model using several classification algorithms that can classify what class a hand falls under; where class is the degree of a given poker hand (one pair being the lowest, royal flush the highest). The problem is easy to understand and we want to take advantage of it’s simplicity by applying multiple classification algorithms to it, allowing us to compare the performance of each. This project also eased the process of dividing work between 7 group members as a result of the independence of each implementation. Each of the method, results, and conclusion requirements are declared in each subsection, and there is an overall conclusion at the end.

# Dataset

*Training Data*: 25,000+ poker hands

*Test Data*: 1,000,000 poker hands

The dataset itself has a series of different poker hands classified into different poker hand “ranks”. Each line is showing a hand consisting of five playing cards drawn from a 52 card deck. Totaling 10 predictive attributes in a 11 size vector, the first 10 numbers correspond to a suit number (1-4) and a rank order (1-13) of each card in a 5 card poker hand. The last attribute of the vector is the number the poker hand is classified into. The order of cards is important, for example there are 480 possible Royal Flush hands as compared to 4.

The classes are:

0: Nothing in hand; not a recognized poker hand

1: One pair; one pair of equal ranks within five cards

2: Two pairs; two pairs of equal ranks within five cards

3: Three of a kind; three equal ranks within five cards

4: Straight; five cards, sequentially ranked with no gaps

5: Flush; five cards with the same suit

6: Full house; pair + different rank three of a kind

7: Four of a kind; four equal ranks within five cards

8: Straight flush; straight + flush

9: Royal flush; {Ace, King, Queen, Jack, Ten} + flush

As an example, the vector <1,10,1,11,1,13,1,12,1,1,9> is a poker hand that has a 10 of clubs, 11 (Jack) of clubs, 13 (King) of clubs, 12 (Queen) of clubs, and 1 (Ace) of clubs, and is classified into class 9 which is a royal flush.

# Running Our Code

Each subgroup has written code in Python 2.7, 3.5, and Java. In order to run the python code it is heavily suggested that you download Anaconda, <https://www.continuum.io/downloads>, which automatically paths various python packages for science, math, engineering, and data analysis. Through anaconda we were able to leverage sklearn and various other libraries that helped us complete the project. Once Anaconda is downloaded you can run any of the .py files within your terminal window (cd to directory type “python filename.py” or python IDE. Knearest was implemented in java and dependencies are included in the package. KNearest would be easiest to run within a java IDE such as Eclipse. More explicit details on running programs are located within the subsections of the report. Please reach out to any of us if you need assistance running any of the code from our project.

Python 2.7 – Support Vector Machine(svm.py)

Python 3+ - Decision Trees(DesTree.py) & Linear Discriminant Analysis (Test.py)

Java – Knearest

Support Vector Machine

# Overview

Support Vector Machine learning algorithm (SVM), is a supervised learning classifier formally defined by a separating hyperplane. These hyperplanes are constructed in a multidimensional space, separating cases of different class labels. It supports regression and classification tasks.

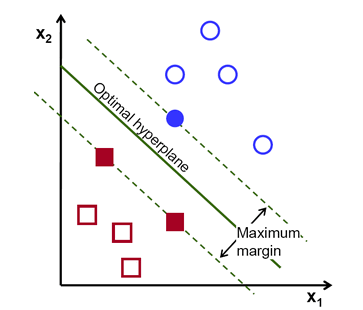
 Given a set of data, there could be many qualifying hyperplanes that separate the clusters of data. SVM algorithm is based on finding the hyperplane that gives the largest minimum distance to the training examples of any class. Twice this distance is the margin. Therefore, the optimal separating hyperplane *maximizes* the margin of the training data as shown in the Figure 1.

Figure 1

When the data is not linear, SVM can handle this situation particularly well by using what is known as a kernel trick. These kernel functions take low dimensional input space and transform it into a higher dimensional space. Figure 2 and 3 describes this situation. Say there is a data set that

represents Figure 2. Clearly, there is not a linear hyperplane (in this 2D space) that can separate the red circle points and the blue stars. By using the kernel trick, the data points are mapped into a higher dimension that can separate the data points and when we look at the hyperplane in its original input space, we see a circle (Figure 3).

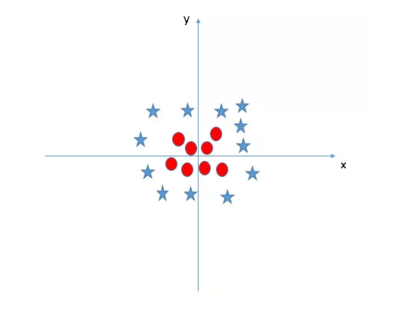
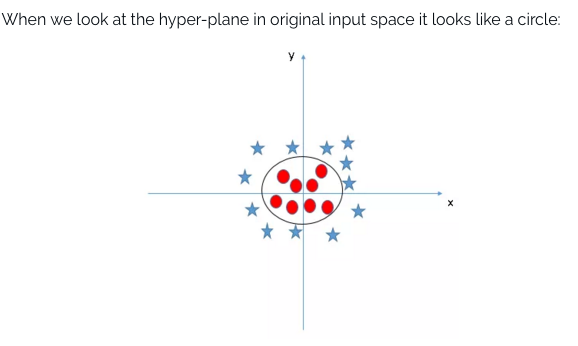


Figure 2

Figure 3

# Advantages and Disadvantages

Advantages of Support Vector Machines –

* Performs well in high dimensional space
* Still effective in cases where the number of dimensions is greater than the number of samples
* Uses a subset of training points in the decision function (called support vectors), so it’s memory efficient
* Versatile: different kernel functions can be specified

Disadvantages of Support Vector machines –

* If the number of features is much greater than the number of samples, the method is likely to give poor performances
* Do not directly provide probability estimates; they are calculated using an expensive five-fold cross-validation

# Mathematical Background

If we were given a training set of instance-label pairs the support vector machine require the solution of the optimization problem below:

subject to the constraints:

C is the penalty parameter, w is the vector of coefficients, b is a constant, and represents parameters for handling non separable data inputs. Index I labels the n training cases. The kernel is used to transform the data into a higher dimension. The parameter C tells the SVM optimization how much you want to avoid misclassifying each training example. For a large C value, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. On the other hand, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, regardless is the hyperplane misclassifies more points. The goal is to identify a good C value so that the classifier can accurately predict the class of each testing data.

# Python Setup

The support vector machine learning algorithm is implemented using Python 2.7 to guarantee compatibility with some of the libraries. Specifically we are using an existing package called scikit-learn. The libraries included in this specific code are: numpy v1.11.0, pandas v0.18.0, and scipy v0.17.0. In order to run the script, the necessary libraries will have to installed in the computer. From the command prompt, the script can be executed. The directory will have to be changed to wherever the script is located. Then in the command prompt ‘python svm.py’ is entered without the apostrophes surrounding it.

If the libraries aren’t installed in the computer, a python distribution called Anaconda may be used which already has a lot of the necessary libraries built in.

The overall program will take a few minutes to execute until it’s done. It takes a few minutes before anything is printed out in the console.

# Support Vector Machine Code (methods)

The svm.py script fetches the training and testing data from <https://archive.ics.uci.edu> using urllib2. After the data is fetched from the web, the data is given a header and sorted by class. Then, the training and testing data is split into two variables. The trainingInput variable takes the first 10 attributes of the data (first 10 indexes) and the trainingOutput variable takes the last attribute which is the class label. The same goes for testInput and testOutput. Next the linear SVM classifier is ran with a C parameter of 1 and trained on the training data. The testInput is then used to predict the output value. Then, the accuracy is calculated based on the predict value and actual value. This whole process is repeated again through a different kernel (radial basis function) and then both are ran against a C value of .001 and 10.

A high C value was not chosen to be run because of the amount of time it would have taken to run the script. Because of the high amount of data that we have in our dataset, a high C value would have required a lot of CPU utilization and therefore have taken a lot of time to ensure that the margin of error was very little to none. Therefore, a total of 3 reasonable C values were chosen to see if there were any differences with the result of this change.

The time it took for each classifier method to train and test the data was also given in the output. Please give the program some time to be interpreted as it is fetching data through url and connection speeds can vary the time to get back results.

# Empirical Results

Classifier: Linear SVM with C value = 1

training...

Training: 60.181483 s used.

Predicting

Testing: 58.216740 s used.

accuracy = 0.55861

----------------------------------

Classifier: Radial Basis Function SVM with C value = 1

training...

Training: 24.165355 s used.

Predicting

Testing: 56.171810 s used.

accuracy = 0.92262

----------------------------------

Classifier: Linear SVM with C value = .001

training...

Training: 19.413530 s used.

Predicting

Testing: 57.356842 s used.

accuracy = 0.55674

----------------------------------

Classifier: Radial Basis Function SVM with C value = .001

training...

Training: 29.888195 s used.

Predicting

Testing: 94.317109 s used.

accuracy = 0.501

----------------------------------

Classifier: Linear SVM with C value = 10

training...

Training: 295.772652 s used.

Predicting

Testing: 53.664476 s used.

accuracy = 0.55627

----------------------------------

Classifier: Radial Basis Function SVM with C value = 10

training...

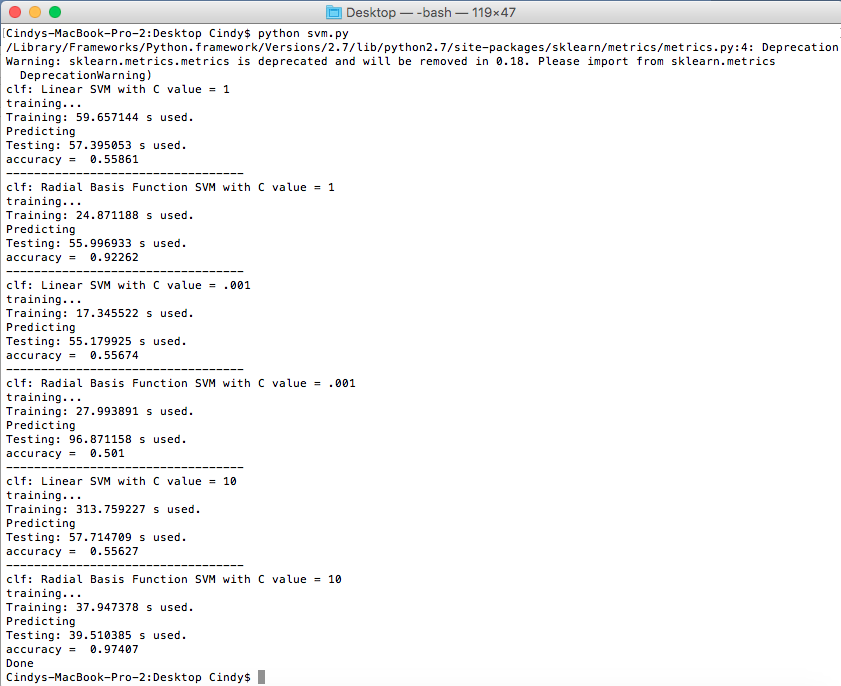
Training: 36.215836 s used.

Predicting

Testing: 37.738813 s used.

accuracy = 0.97407

Done



Command Prompt Screenshot

# Conclusion

In summary the resulting accuracies for different C values per different SVM kernels are:

|  |  |  |  |
| --- | --- | --- | --- |
|  | C = 10 | C = 1 | C = .001 |
| Linear | 0.55627 | 0.55861 | 0.55674 |
| RBF | 0.97407 | 0.92262 | 0.501 |

From the table above, we see that the accuracy value remains relatively the same for each C value for a linear kernel. This was different than what I expected, since I thought that the accuracy would increase if the C value increases, because the point of a high C value is to decrease margin of error. So, the data might not be linearly separable, which is why radial basis function kernel performs much better with almost twice the accuracy rate. We see that the accuracy score increases as the C value increases. The accuracy increases dramatically from when C = .001 and C = 1.

K-Nearest Neighbor

Introduction

Given a dataset of possible card combinations, we wished to evaluate which classification algorithm would be most effective at identifying poker hands. Using the K-Nearest Neighbor algorithm, we would be able to classify the test dataset by determining the “k” closest neighbors to the test data in the training dataset. The classification of the test data entry would be equal to the most popular classification of the “k” closest neighbors. This should be repeated for all the entries in the test dataset.

In order to compute the accuracy, we can take the sum of all of the true predicted classifications, divided by the length of the test dataset. This accuracy will be an acceptable indicator of how the K-Nearest Neighbor algorithm compares to the other classification methods we evaluated.

Methods

We opted to write our component in Java, included in the zipped KNearestNeighbor package attached with the submission.

Initially, we believed it would be most efficient to represent the data two-dimensionally so that we could easily calculate the distance between data entries using the Euclidean distance formula. This can be seen in our code in the unused getXY() and customClassify() methods (as well as a handful of helper methods). These would have run the data sets through a formula to work them into two dimensional data that would have been classified through a 2-dimensional Euclidean distance equation. The accuracy calculation would have been the same, had we followed through with this line of thinking.

However, because the Euclidean formula accepts n-dimensions, we decided to see what kind of results we could get from using ten dimensions instead; two for each of the five cards: the suit and the rank. This way, we would be able to classify the raw data points without having to give the program any additional information about what certain combinations of cards means. This is the final process we settled on.

First, we read in the data through the readFile() method, and its helper methods. Then we call the basicClassify() method, taking as parameters the training data, test data, and the number of neighbors to find. The number of neighbors can be set through the “nearestneighbors” variable. It should be noted that this method alone takes 5-6 minutes to run, given the size of the data. It carries out the basic k-nearest-neighbor algorithm, classifying the test data based on the k nearest neighbors in the training data. This function outputs an array of proposed classifiers for the test data.

From here, called the findAccuracy() method, taking the raw test data and classifier array as parameters. This method calculates the accuracy, accounting for all true matches divided by the dataset size. This methods takes nearly an hour to run through due to the sheer volume of the arrays being compared. Our method of k-Nearest-Neighbors ends here, returning the calculated accuracy.

Empirical Results

K=1

Length of test data: 1000000

Length of training data: 25010

5 Test Data entries:

1: [1,1,1,13,2,4,2,3,1,12,0]

2: [3,12,3,2,3,11,4,5,2,5,1]

3: [1,9,4,6,1,4,3,2,3,9,1]

4: [1,4,3,13,2,13,2,1,3,6,1]

5: [3,10,2,7,1,2,2,11,4,9,0]

5 Training Data entries:

1: [1,10,1,11,1,13,1,12,1,1,9]

2: [2,11,2,13,2,10,2,12,2,1,9]

3: [3,12,3,11,3,13,3,10,3,1,9]

4: [4,10,4,11,4,1,4,13,4,12,9]

5: [4,1,4,13,4,12,4,11,4,10,9]

Finished. Took :

1350

Correct! #1.0: 0 | 0

Correct! #2.0: 1 | 1

Correct! #3.0: 0 | 0

Correct! #4.0: 0 | 0

Correct! #5.0: 0 | 0

Correct! #6.0: 0 | 0

Correct! #7.0: 0 | 0

Correct! #8.0: 0 | 0

Correct! #9.0: 1 | 1

Correct! #10.0: 1 | 1

Correct! #11.0: 0 | 0

Correct! #12.0: 1 | 1

Correct! #13.0: 0 | 0

Correct! #14.0: 1 | 1

Correct! #15.0: 0 | 0

Correct! #16.0: 0 | 0

Correct! #17.0: 1 | 1

Correct! #18.0: 0 | 0

Correct! #19.0: 1 | 1

Correct! #20.0: 1 | 1

Correct! #21.0: 0 | 0

Correct! #22.0: 1 | 1

Correct! #23.0: 0 | 0

Correct! #24.0: 1 | 1

Correct! #25.0: 0 | 0

Correct! #26.0: 0 | 0

Correct! #27.0: 0 | 0

Correct! #28.0: 1 | 1

Correct! #29.0: 1 | 1

Correct! #30.0: 1 | 1

Correct! #31.0: 1 | 1

Correct! #32.0: 1 | 1

Correct! #33.0: 1 | 1

Correct! #34.0: 0 | 0

Correct! #35.0: 0 | 0

Correct! #36.0: 1 | 1

Correct! #37.0: 1 | 1

Correct! #38.0: 1 | 1

Correct! #39.0: 0 | 0

Correct! #40.0: 0 | 0

Correct! #41.0: 1 | 1

Correct! #42.0: 0 | 0

Correct! #43.0: 0 | 0

Correct! #44.0: 1 | 1

Correct! #45.0: 1 | 1

Correct! #46.0: 0 | 0

Correct! #47.0: 1 | 1

Accuracy: 47.0

Finished. Took :

422196

Accuracy: 47.0

Accuracy = 43.1822

K=2

Length of test data: 1000000

Length of training data: 25010

5 Test Data entries:

1: [1,1,1,13,2,4,2,3,1,12,0]

2: [3,12,3,2,3,11,4,5,2,5,1]

3: [1,9,4,6,1,4,3,2,3,9,1]

4: [1,4,3,13,2,13,2,1,3,6,1]

5: [3,10,2,7,1,2,2,11,4,9,0]

5 Training Data entries:

1: [1,10,1,11,1,13,1,12,1,1,9]

2: [2,11,2,13,2,10,2,12,2,1,9]

3: [3,12,3,11,3,13,3,10,3,1,9]

4: [4,10,4,11,4,1,4,13,4,12,9]

5: [4,1,4,13,4,12,4,11,4,10,9]

Finished. Took :

1796

Finished. Took :

324228

Accuracy: 44.1531%

We ran the algorithm for k values of 1 through 3 over the 1,000,000 entry test data. For the first case, we found an accuracy of 43.1822%. For the second, we calculated an accuracy of 44.1531%. When looking for the 3 nearest neighbors, we found the accuracy to be 43.1822%. We consider these observations to be relatively representative of the data set, though we would have liked to have run tests for higher values of k. Time constraints related to the length of time required to fully carry out the algorithm restricted us from carrying these out. These results should be repeatable, as the dataset will not change, though we expect the algorithm will take just as long to evaluate.

Conclusion

According to our results, we found that the K-Nearest Neighbor algorithm was not the most accurate method that could be used to classify sets of cards into poker hands. From the data that we collected, there does not seem to be a pattern in accuracy changes as the value of k changes. As the highest accuracy was only 44.1531% (when k = 2), there are other methods that will not take quite as long to complete and will yield a higher accuracy. We believe the other methods attempted by our team members should return more accurate classification results.

Decision Trees

Overview

Decision trees break down of inputs into various attributes into a tree-like structure. Every level of the tree is generated based on the decision of the attributes, which can be thought of as breaking down a classification into smaller and smaller sub problems to try to guess correctly. Decisions are based on whether an attribute is either true or false, or greater or less than a desired target value. The final result of a decision can be seen as a path from the root to a leaf, where the current values passing through satisfy given conditions. Decision trees are easy to understand and can be visualized, they are capable of both numerical and categorical data, and can be easily cross validated. However, they can be biased if certain classes dominate, which sort of happened to our data due to the large amount of lower class hands, and decision trees can quickly become complicated as they don’t generalize all data well.

We used a CART implementation of decision trees which is a combined classification and regression algorithm that recursively partitions data space and fits a prediction on each split. The CART algorithm gives us opportunity to prune our tree since it is sparse due to all the distinctly different types of hands one can get in a poker hand. This type of algorithm is non-parametric so there are no assumptions about probability distributions. Decision trees are notoriously seen within the Iris classification problem which is somewhat similar to the classification of poker hands that we were trying to solve. We had to deal with the potential problems of poker hands being wrongly classified based off a magnitude of a hand. For example, one may receive an off suit queen and nine, and that hand could be classified better than a pair of two’s based on the values of the cards. Combinations of numerically lower valued cards can increase the cumulative hand value even though that might not necessarily be a “good” hand to get, where good is defined to be a hand you would place a bet on.

Methods

formatData(path) – in order to manipulate the large amount of data as desired we saved the data to a relative path, which took a lot of time to save, but saved a large amount of read time during the actual running of the algorithm since we did not require a http request to communicate with the data set. In this method the attributes are thrown into an array of length 10 and the classifiers in a singleton array. This gave back a more structured data set that we were able to then reshape.

Reshape(arg, arg,) – reshaping data is required in order to put data in the correct format to be used by the scikit-learn algorithms. This method is transforming the data like you would a normal matrix.

Fit(attributes, classifier) – this method builds the decision tree off the training set based on the CART algorithm. It iterates through all the attributes and classifiers and produces a model from applying the class to the values.

Predict(x) – takes the test data and predicts classes based on the previously fitted decision tree. The predicted class for each sample gets returned.

From the predict method optimizations were made by combining the attributes into unique suits and ranks, taking 5 attributes of both into 1. This combination of attributes allowed us to get significantly better results, explained more in the conclusion.

Empirical Results

With Raw Attributes

Number Correctly Identified: 479371

Number Incorrectly Identified: 520627

Percent Accuracy:47.94%

With Unique Attributes

Number Correctly Identified: 974747

Number Incorrectly Identified: 25251

Percent Accuracy: 97.47%

Improvement: 2

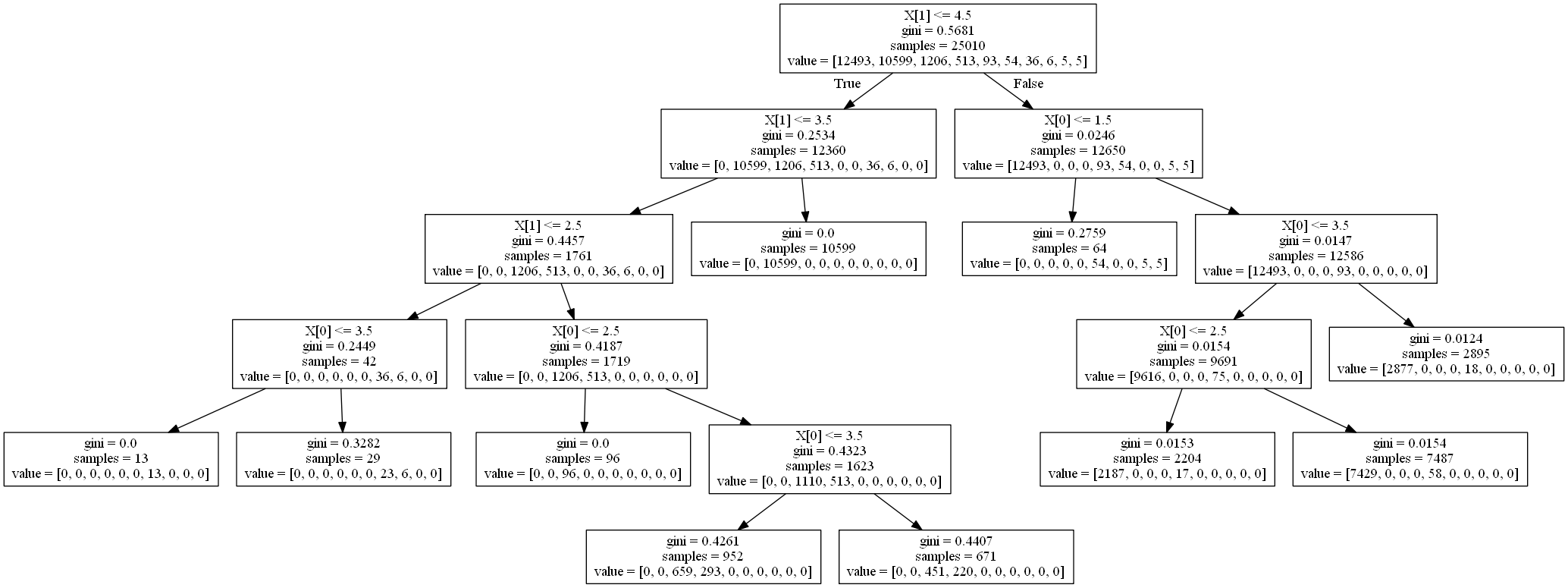
[Finished in 165.1s]

Conclusion

Based off the results we’ve determined that decision trees are not the best algorithms for classifying poker hands if you don’t do pre or post processing. Initially our algorithm generally output around 48% prediction accuracy from different subsets of the test data, which is somewhat caused by the decision tree tending to overfit the data with the amount of classifiers. However, since the depth of our tree, with only 10 attributes, would not provide substantial overfitting, it probably wasn’t the largest issue. Due to the fact that we had a large amount of training data, the tree became rather complicated and somewhat biased. There is quite a large amount of class 0-3 hands in the data set so the output seemed to lean towards deciding outputs were in that range. On average it took several minutes to query all million test hands and graph them, even though querying the tree for a single test hand provided an O(log(n)) runtime. Our first attempt of graphing the data resulted in over 20,000 nodes and 20,000 vertices which takes a significant amount of time to plot on a decent computer. The output was 30,000 pixels wide and looks incredibly distorted.

CSE%204705/project/AIFinalProject/out/tree.png

From there we had to find a way to synthesize the poker hands into the least number of attributes without increasing the entropy of the result. In order to accomplish making the entire tree compressed from a factor of 910 choices, from 9 different choices each level to a factor of 2. This is done by finding similarities within the data, we can lower the number of attributes. The issue is, how do you lower the number of attributes, and still retain the meaning in the data? This was accomplished by turning 5 rank attributes into 1 by their uniqueness and turning 5 suit attributes into 1 by their attribute uniqueness. This works because the strength of a poker hand heavily relies on the non-uniqueness of ranks and suits. So two of a kind, three of a kind, four of a kind, and a full house are all an example of ranks being non-unique, you’re always looking for combinations of the same rank, and the only hand that does not win on uniqueness is a straight. Having hands with different unique cards becomes a good metric because in general, you win a poker hand based on the non uniqueness of rank. Therefore, the concatenation these attributes allows us to have a significantly less complicated tree and it fits what a “good” hand is. The above picture that doesn’t look like a tree becomes reduced to the tree below.



It is important to say that decision trees are generally very good at classifying data. In fact they are one of the most used tools used throughout machine learning and data mining. Pre-pruning is a good idea to stop the tree from growing to a point where it will overfit a majority of the data, but this is not the easiest task, as it is difficult to determine when the perfect time to prune is. Another method similar to what we did with unique attributes allows for a smaller tree and more accurate results. This is the reason why finding the optimal decision tree is viewed as an NP class problem. However, clever tricks to combine attributes can allow for better decisions.

Linear Discriminant Analysis

Overview

Linear discriminant analysis is used for classification, and is typically used to reduced dimensionality and prevent overfitting. It assumes a normal distribution for mean and variance and class covariances are similar. It uses orthogonal point projection to a line that creates a Gaussian distribution to partition the set.

Methods

Coding in Python, we used the linear and quadratic discriminant analysis methods in the sklearn library to build our models and make predictions. The numpy library allowed us to convert easily from the input read from the database into workable arrays for analysis. The matplot library gave the necessary tools to plot ROC curves, together with roc\_curve and auc methods from sklearn. The input binarizer, one-vs-rest classifier, and test-train split, all from sklearn, allowed the training data to be partitioned for the purpose constructing the ROC curve.

Empirical Results

Linear discriminant analysis:

Class 0: 25009 predicted, of which 12493 are true positives out of 12493 true total

Class 1: 0 predicted, of which 0 are true positives out of 10599 true total

Class 2: 0 predicted, of which 0 are true positives out of 1206 true total

Class 3: 0 predicted, of which 0 are true positives out of 513 true total

Class 4: 0 predicted, of which 0 are true positives out of 93 true total

Class 5: 0 predicted, of which 0 are true positives out of 54 true total

Class 6: 0 predicted, of which 0 are true positives out of 36 true total

Class 7: 0 predicted, of which 0 are true positives out of 6 true total

Class 8: 0 predicted, of which 0 are true positives out of 5 true total

Class 9: 0 predicted, of which 0 are true positives out of 4 true total

For quadratic discriminant analysis:

Class 0: 18424 predicted, of which 10263 are true positives out of 12493 true total

Class 1: 6566 predicted, of which 3448 are true positives out of 10599 true total

Class 2: 0 predicted, of which 0 are true positives out of 1206 true total

Class 3: 2 predicted, of which 1 are true positives out of 513 true total

Class 4: 2 predicted, of which 0 are true positives out of 93 true total

Class 5: 0 predicted, of which 0 are true positives out of 54 true total

Class 6: 0 predicted, of which 0 are true positives out of 36 true total

Class 7: 6 predicted, of which 6 are true positives out of 6 true total

Class 8: 5 predicted, of which 5 are true positives out of 5 true total

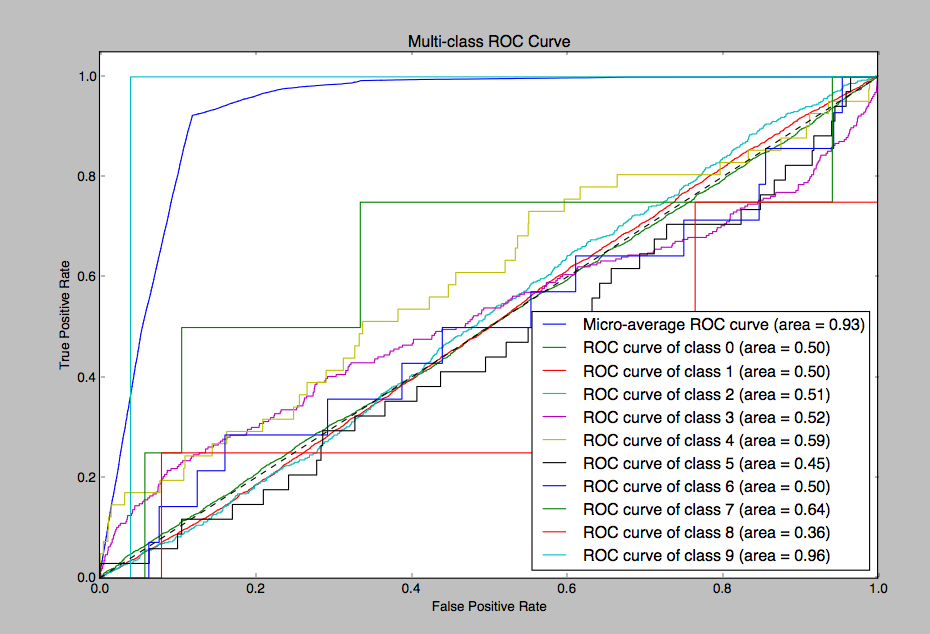
Class 9: 4 predicted, of which 4 are true positives out of 4 true total

Using the initial approach of linear discriminant analysis, every instance, including those in the training model, were assigned to class 0. Out of the 25009 instances in the training model, the two most prominent true classes were 0 and 1 with 12493 and 10599 instances, respectively. Our hypothesis is that because linear discriminant analysis aims to project the data onto a line in 2-D space, there were no components that could be identified as more important than others in such a low-order dimension. From a practical point of view this makes sense, as it’s often not possible to make a confident classification until every attribute has been accounted for. For instance, if the first nine attributes of a given instance result in a prediction of class 0 (nothing; not recognized), there’s still a reasonable chance the remaining attribute will match in rank to one of the others (which would result in prediction of class 1). The dependency among attributes makes it difficult to imagine a dimensional reduction would be possible, let alone a classification scheme.

For further analysis, we were able to partition the training set in two, and use one half to train the data and the other half to test. These results were then plotted using an ROC curve. Individual classes did not fare substantially better than guessing, but the micro-average ROC curve which accounts for the accuracy over all classes, was significantly better than a random assignment.

Conclusion

Our subsequent approach of quadratic discriminant analysis faired slightly better, predicting primarily classes 0 and 1. Out of the 25009 instances in the training model, 18493 were predicted as class 0, accounting for 10263 out of 12493 true instances, and 6566 were predicted as class 1, accounting for 3448 out of 10599 true instances. Additionally, the model predicted all instances in classes 7, 8, and 9, correctly without misapplying those labels elsewhere. These classes correspond to four-of-a-kind, straight flush, and royal flush, respectively. This perfect conversion rate represents the success the model had of identifying the underlying structure and in particular, how few instances were required to do so. However, barely any predictions were made for classes 2, 3, 4, 5, and 6, corresponding to two-pair, three-of-a-kind, straight, flush, and full house, respectively. While the training set was conceivably designed so that the number of instances of a certain class present reflected the probability of it occurring randomly, this deficiency represents the failure of the model to identify underlying structure. For instance, when a full house occurs, some hierarchically lower hands such as pair, two-pair, and three-of-a-kind are simultaneously present. But since only one classification can be made, there is no constructive way the model can develop any kind of helpful intuition. For this reason, the model may have more readily chosen classes which constituted a higher proportion of the training set when in doubt.



Through the above process, we were able to discern the effectiveness of our model without having to resort to additional testing data. While typically feeding an instance’s attributes back into a model it was trained on predicts the true class, the projection produced by LDA/QDA for classification is a generalization and inherently loses accuracy.

One of the concerns we had about our model is that LDA/QDA is conventionally used with continuous variables for attributes and discrete variables for classes. Although our dataset required discrete variables for both, research has shown that either model is robust enough to adjust accordingly. Problems can arise when the attributes are a mix of discrete and continuous variables.

Conclusion

# In an overall conclusion aside from each sub components deductions, our best results came from using the Support Vector Machine and radial basis function with a c of 10. The graph shows the best accuracy amongst all the tests. When we leveraged the fact that common ranks are a good way to win a poker hand the increase of accuracy was a close second for decision trees. If we defined this classification scheme to the other algorithms the results may have differed in our favor.

# Screen%20Shot%202016-05-06%20at%2011.23.58%20PM.png

# With all classifying algorithms, it is important to choose the right tool for the job. Thank you for letting us learn a lot more about classifier algorithms and gain some experience working in groups.

# Bibliography

1. "1.4. Support Vector Machines." *1.4. Support Vector Machines — Scikit-learn 0.17.1 Documentation*. N.p., n.d. Web. 05 May 2016.
2. Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. "A Practical Guide to Support Vector Classification." *Practical Guide to Machine Vision Software* (2014): n. pag. 15 Apr. 2015. Web. 5 May 2016.
3. "Introduction to Support Vector Machines." *Introduction to Support Vector Machines — OpenCV 2.4.13.0 Documentation*. N.p., n.d. Web. 05 May 2016.
4. Steinwart, Ingo, and Andreas Christmann. *Support Vector Machines*. New York: Springer, 2008. Web. 5 May 2016.
5. Dipartimento Di Elettronica. "Clustering - K-means." Clustering - K-means. N.p., n.d. Web. 02 May 2016. <http://home.deib.polimi.it/matteucc/Clustering/tutorial\_html/kmeans.html>.
6. Onmyphd. "K-means Clustering." N.p., n.d. Web. 03 May 2016. <http://www.onmyphd.com/?p=k-means.clustering&ckattempt=1>.
7. Cambridge University Press. "K-means." K-means. Stamford, n.d. Web. 02 May 2016. <http://nlp.stanford.edu/IR-book/html/htmledition/k-means-1.html>.
8. Open Source. SKLearn. INRIA, Google, Other, n.d. Web. <http://scikit-learn.org/stable/>.
9. Colorado School of Mines. "Decision Tree Classifier." Decision Tree Classifier. HumanOriented, 2010. Web. 05 May 2016. <http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio\_exports/lguo/decisionTree.html>.
10. Li, Jia. "Classification Trees." SpringerReference (n.d.): n. pag. Pdf. Penn State, 2011. Web. <http://sites.stat.psu.edu/~jiali/course/stat597e/notes2/trees.pdf>.
11. Moore, Andrew. "Decision Trees." Decision Trees. Carnegie Mellon University, 2013. Web. 06 May 2016. <http://www.autonlab.org/tutorials/dtree.html>.
12. Raschka, Sebastian. "Linear Discriminant Analysis." Sebastian Raschka's Website. Self Post, 03 Aug. 2014. Web. 03 May 2016. <http://sebastianraschka.com/Articles/2014\_python\_lda.html>.
13. Li, Jia. "Linear Discriminant Analysis." Statistical Pattern Recognition (n.d.): 123-68. Class Notes. PDFThe Pennsylvania State University, 2011. Web. 3 May 2016. <http://sites.stat.psu.edu/~jiali/course/stat597e/notes2/lda.pdf>.
14. ALGLIB. "Linear Discriminant Analysis - ALGLIB." Linear Discriminant Analysis - ALGLIB. ALGLIB Project, 1999. Web. 01 May 2016. <http://www.alglib.net/dataanalysis/lineardiscriminantanalysis.php>.