Datasets used:

UCI Wisconsin Breast Cancer Dataset

The first dataset I used was the UCI Cancer database, a classification dataset to determine if a tumor was 'malignant' or ' benign':
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)
This dataset was interesting to me as I've had several family members with cancer. Using ML to solve hard hitting problems such as this really intrigues me and reveals the true potential that machine learning has – we can solve so many diverse and complex problems, sometimes to even save lives. The dataset has 10 initial features on a 1-10 continuous scale with a binary classifier. Around 600 records are in this dataset, so not extensively large. From my understanding it's a classic UCI ML dataset so I figured it'd be a good place to start learning these algorithms.

Kaggle basketball shots dataset.

My second dataset involves a large number of records from the 2014-2015 NBA season ultimately recording if a shot was 'made' or 'missed'. Some fields involve amount of dribbles taken before shooting, distance from the hoop, player taking the shot, closest defender distance, etc. I actually used this data set to apply for an internship about a year ago, well before I became interested in machine learning. They tasked me with creating a prediction algorithm for the dataset, and this is actually when I first discovered logistic regression, so it has a special place in my heart. Of course my simple analysis didn't land me the job, but it really got me interested in data science. This data set involves several null fields and numerous categorical variables, so I performed some preprocessing in excel myself - this involved omitting fields with null values and using one-hot encoding on discrete variables, all of which are binary. I chose omission over say, selecting the mean/median/mode of a particular feature and using that for nulls, as there were only around 5000/100k records with null values so fairly small, and I can cut my preprocessing time down using this method. I've also omitted some columns which are redundant like "points scored", which is just a combination of the "fgm" column (field goal made) and the "shot type" column (2, or 3). Additionally, I found that encoding for some variables would be very difficult for the dataset, as either the dimensionality would significantly increase due to so many potential possibilities (one-hot, player name) or the encoding would misrepresent the variable (label encoding). Luckily, these variables I felt went against the general purpose of my algorithm, which would be to predict the likelihood of making a shot, no matter what player was shooting/defending, what day, etc. I felt these were extraneous to the solution. Some variables, however, were very relevant to the outcome, just based on my personal knowledge of the sport – home vs away, win or loss, etc. I used practicality and my background knowledge of basketball to shape the dataset beforehand.
https://www.kaggle.com/dansbecker/nba-shot-logs

Structure of analysis

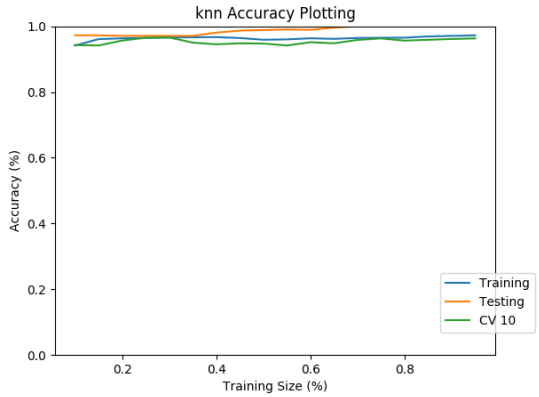
I will conduct my analysis by analyzing the data sets separately, and going through each algorithm per data set. I generally had the same method for analysis -> preprocessing (scaling, random shuffling, and feature selection through variance thresholding), training/testing split size selection, hyper parameter optimization, and comparison of initial performance (no hyper parameter tuning) to final performance (hyper-parameters optimal).

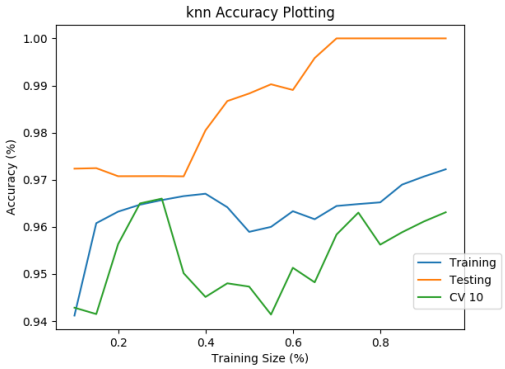
Breast Cancer Dataset:

K-Nearest Neighbor:

Knn ran quite quickly for me on the UCI dataset. I understand this is a computationally exhaustive algorithm for testing but with a smaller dataset with fewer features the testing processed rather quickly (even with several iterations of training size).

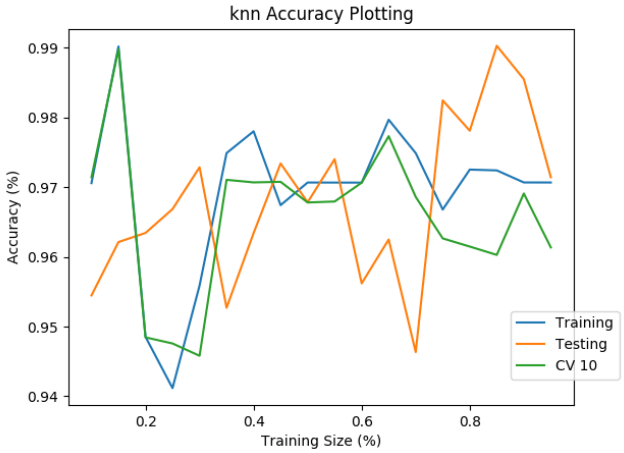
Interestingly enough, running a KNN on the UCI dataset with no alterations (base run) produces a learning curve like this:



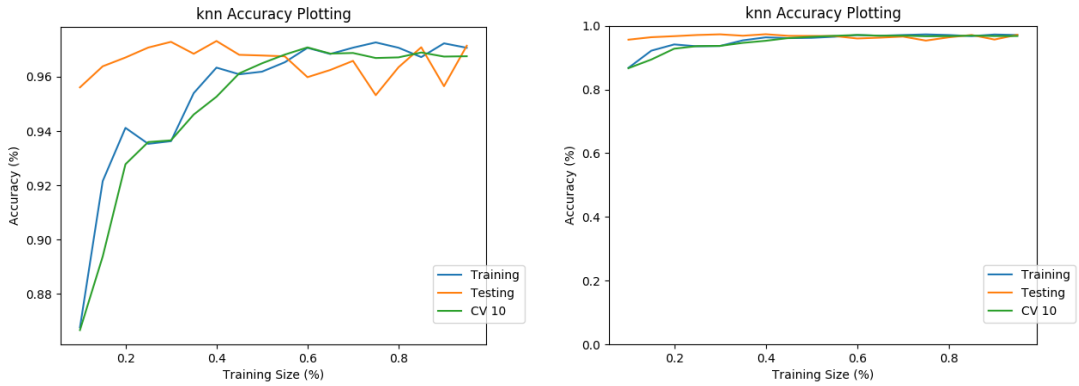


The blue and orange lines denote the training/testing split, while the green line represents a 10 fold cross validation. I quickly noticed how the second plot illustrates 100% accuracy on the test set after a 70% training set split, which I felt was quite bizarre to attain that accuracy so quickly, and to have a higher test accuracy than training. I attributed this to the dataset: the collection had occurred in several groups, so I assumed the last 3rd of the data was not balanced against the first 7/10ths. In order to combat this, I randomly shuffled and split the data so as to avoid

any imbalance in the training and testing sets, and came up with this new plot



(*Note – I realize that the plots do not have much variation on a 0-100 scale, I just wanted to point out this strange noise on a small scale and what I realized was causing it). I discovered that I had been generating a random shuffle split for each iteration of test size when creating this graph, which cause this strange noise on a small scale. I combatted this by initializing one random split before iterating through a loop of incrementing training size, which resulted in a much smoother corresponding graph:

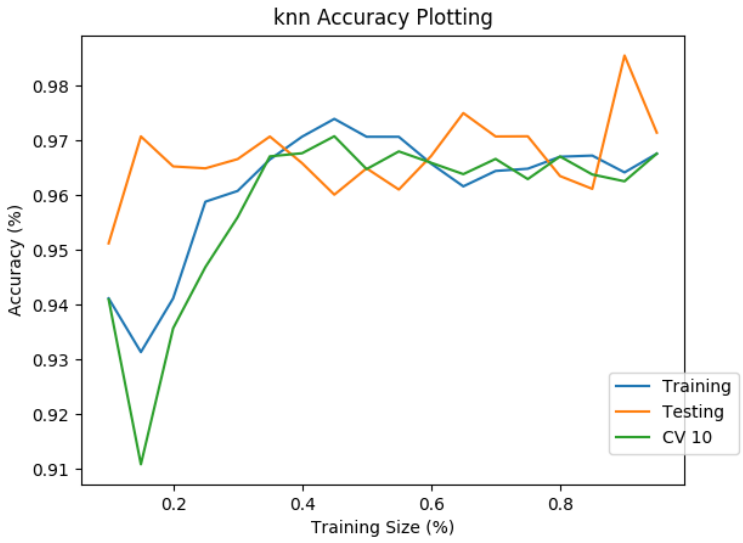


I had some strange behavior with the testing accuracy running different replications. It seemed to frequently have a higher % than its training and cv counterparts, and would either significantly drop or spike after the training size had increased to .95. Although I realized this was probably negligent, as the fluctuations were only around 2% and all of the accuracy scores were consistently in the upper 90s. These were my base metrics for accuracy with no parameter tuning or modifications: my test accuracy consistently averaged around 96%. I was somewhat worried about overfitting, especially with the small dataset.

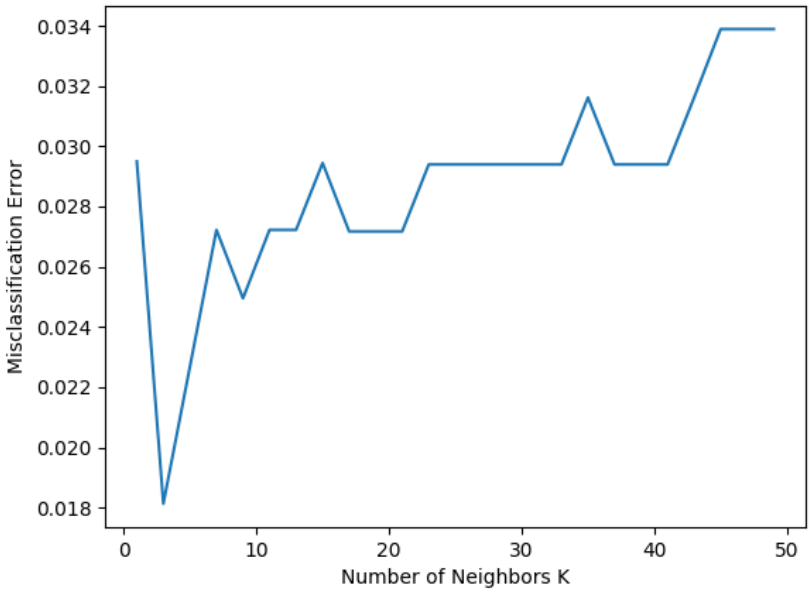
My first attempt to combat this was to standardize the features on a [0,1] scale based on their minimum and maximum values. Knn depends on some scaling/normalization/standardization of its attribute to avoid bias from smaller scale attributes; the Euclidean distance will favor smaller (and thus closer) data. This did not have much effect however, as each attribute already was on a 1-10 scale, so they already had essentially been normalized.

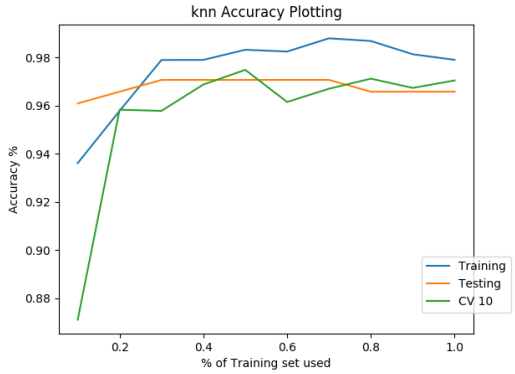
With 10 attributes, I next tried to cut down some using feature selection. I decided to use variance threshold as it seemed like the most straightforward (I tried to use Recursive Feature Elimination but it requires an estimator to base its feature selection off of – I'm assuming for algorithms that are not instance based). The threshold of 1.0 was the sweet spot for the variance (any higher and only one feature remained), and this cut my features down to 5. This still however produced similar results. This was likely because most if not all of my features seemed to have low variance, so taking some out would not impact predictive power.

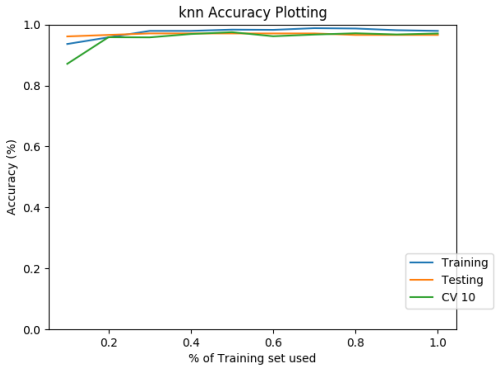
It was time to tune my hyper parameter, k. It was in my best interest to keep the feature reduction, random splits, and feature scaling, as these all contribute to the efficiency of knn while avoiding overfitting and inaccuracies. At a more focused scale the highest average CV accuracy (using CV to avoid fitting to testing data) seemed to be at around a 65% training/testing split, so I used this in my parameter tuning (CV was higher around 40%, but I did not want to limit my training dataset to below 60%, as this would likely affect accuracy).



Running several trials of increasing values of k through a 10 fold cross validation gives me and average optimal value of k=3. I used CV in this situation because I know that hyper parameter tuning using the testing set is an easy way to over-fit your model. Essentially I produced the graph multiple times, and received a mode of k=3 (changing due to random shuffling). See graph below:







My hyper parameter tuning was based on MSE of classification error, so the parameter was not exactly tuned to predictive CV accuracy. So, running the final model at a training split of .65/.35 and a ten fold CV with k=3 yields me the following (100 random shuffling replications seemed to yield the best, least variant results). The % of Training set used vs accuracy yielded normal results. Testing and cross validation yielded slightly lower scores then the training accuracy which is what I expected.

INITIAL (random shuffling, no scaling, no feature selection, no tuning of k (k=10), training size = .65):

Testing accuracy (average across 100 runs): **96.38%**

Standard deviation of 100 runs for testing accuracy: **.012**

CV accuracy (average across 100 runs): **96.67%**

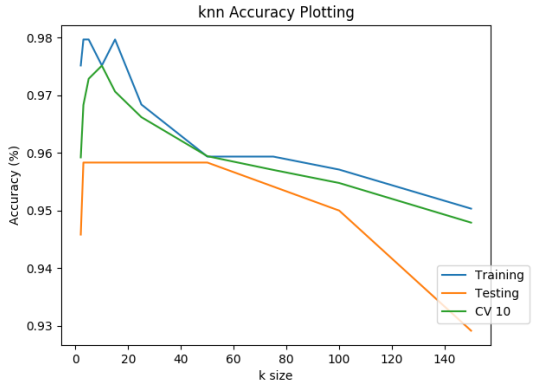
Standard deviation of 100 runs for CV accuracy: **.006**

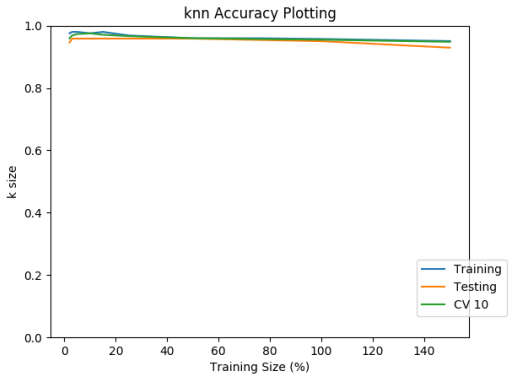
Average Standard Deviation of k folds(across 100 runs): **.0244**

Standard deviation of the average stdevs for k-folds: **.007**

FINAL:
Testing accuracy (average across 100 runs): **96.08%**
Standard deviation of 100 runs for testing accuracy: **.01**
CV accuracy (average across 100 runs): **96.18%**
Standard deviation of 100 runs for CV accuracy: **.007**
Average Standard Deviation of k folds(across 100 runs): **.026**
Standard deviation of the average stdevs for k-folds: **.006**

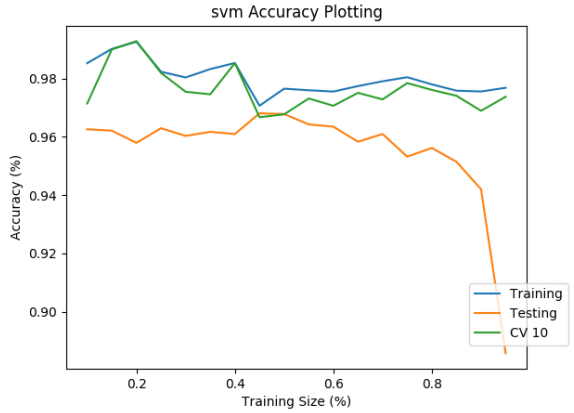
Disappointingly, the results are not that different. In fact they're very similar. Curious as to why this was, I plotted k size vs accuracy:

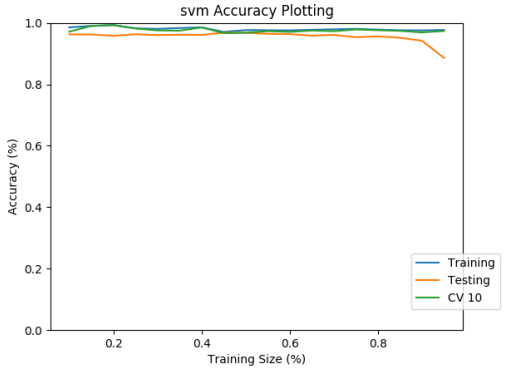




Based on these graphs, it appears that testing accuracy remains constant for k 3 – 60, which explains why the results were so similar. I believe it is because of low variance in the data set – points nearby as well as points very far away had similar values, so they did not skew results. It also reveals that feature selection did not play a big role in the overall results.

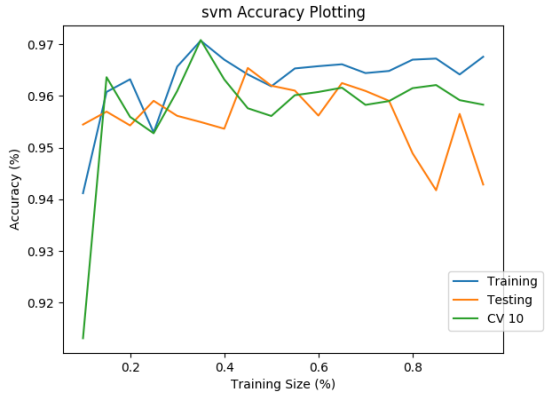
SVM:

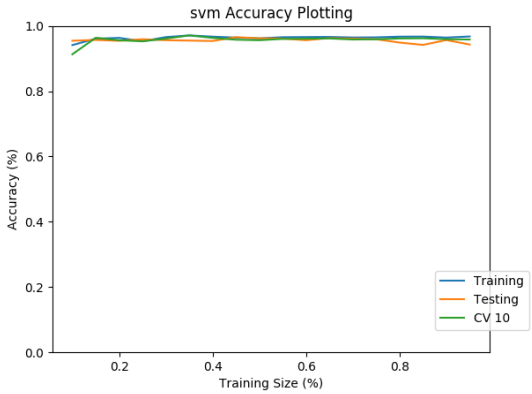




An initial look at my graph signals that 0.8 training split maximizes our CV accuracy. I've already scaled the attributes since this is important for SVM (again don't want to favor attributes with larger scales) and shuffled the attributes as well. I'm using a linear kernel to start off with as

well as this often yields the best results and I don't want to try a non-linear kernel until parameter tuning. SVM is good with high dimensional spaces so I want to check if feature selection will even make a difference (especially with the already low number).





Interestingly enough, feature selection (reducing to 5) seems to have *decreased* my cv accuracy by about 2%. This may be a good thing however, as we may have removed noise from our dataset (low variance), and thus we have probably reduced overfitting! I'll keep the feature size moving forward.

With a multi-variate selection of parameters (kernel, "c" value, and "gamma"), I wasn't entirely sure how to optimize this for maximize my prediction over CV. Luckily there is an sklearn function (GridSearchCV) which maximizes the scoring metric (in this case test accuracy and CV accuracy) according to the parameters given. Because of very slow run time (likely due to the optimization of parameters), I ran 10 iterations to determine the following optimal hyper parameters selected as well as an average accuracy for all 10 runs (*note – each iteration utilized a different set of optimal parameters – this is due to the random shuffling instantiated during each iteration):

{'kernel': 'rbf', 'C': 1, 'gamma': 0.02}
{'kernel': 'rbf', 'C': 2, 'gamma': 0.2}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.5}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.3}
{'kernel': 'linear', 'C': 2, 'gamma': 0.01}
{'kernel': 'rbf', 'C': 5, 'gamma': 0.03}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'rbf', 'C': 3, 'gamma': 0.5}
{'kernel': 'linear', 'C': 5, 'gamma': 0.01}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.04}

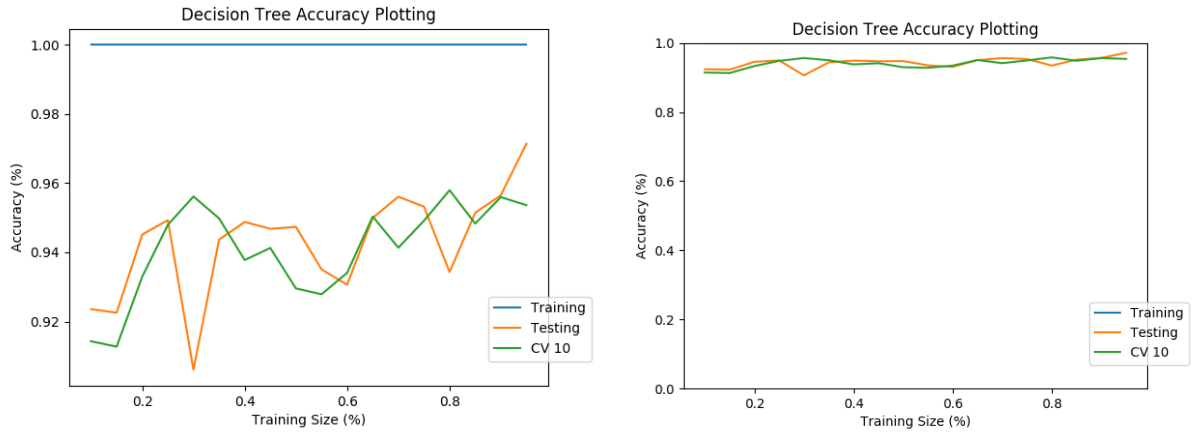
INITIAL (for strict comparison purposes of parameter optimization, all factors are held constant except for parameters: kernel=linear):
Testing accuracy (average across 10 runs): **95.62**
Standard deviation of 10 runs for testing accuracy: **.012**
CV accuracy (average across 10 runs): **96.24%**

Standard deviation of 10 runs for CV accuracy: **.004**
Average Standard Deviation of k folds(across 100 runs): **.0227**
Standard deviation of the average stdevs for k-folds: **.008**

FINAL:
Testing accuracy (average across 10 runs): **96.42%**
Standard deviation of 10 runs for testing accuracy: **.01**
CV accuracy (average across 10 runs): **96.1%**
Standard deviation of 10 runs for CV accuracy: **.0037**
Average Standard Deviation of k-10 folds(across 10 runs): **.025**
Standard deviation of the average stdevs for k-10-folds: **.004**

We definitely see some improvement in both testing accuracy as well as CV accuracy.
It seems that in most cases, rbf fits our model more accurately than linear, implying that the data is non-linear (I assumed that during tuning, the 'c' and 'gamma' parameter were ignored if the optimizer chose 'linear' as the optimal kernel, as c and gamma to not apply to linear kernels). Fitting an rbf kernel was likely the result of our slight improvement in accuracy over most runs.

<u>Decision Tree:</u>
A first glance at an initial training size vs accuracy plot reveals something interesting:



The training accuracy is 100%. I can think of 2 reasons why this is occurring: (1) trees are designed this way inherently. The training data is split in such a way so that training accuracy is always 100% - the function completely encompasses every data point in the training set so the training predictions are entirely accurate, or (2) there are one or several variables that perfectly split the data; high entropy variables at the top split data into *independent* subsets, and variables with low entropy already likely have low variance (i.e. they are all the same), so the tree can predict the correct outcome on the training data always (because it is built on the training data). Referring to part (2), I would expect this result after feature selection, because the likelihood of having a high entropy variables is greater with fewer variables from the get-go, but it seems that there is a possibility of several high entropy variables (we know that there should be around 5, because feature selection eliminates the 5 others). Once again, the CV

seems to reach its highest point around .8 training size, so I will use this value when tuning parameters. *Note: feature selection did not seem to improve accuracy, but I will of course use it to avoid overfitting. I will use feature selection, random shuffling, and attribute scaling for the remainder of the algorithms when using the Cancer dataset.

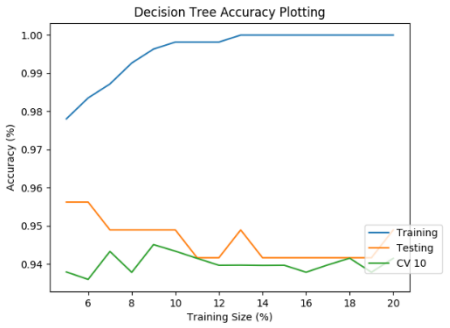
I once again used the GridSearchCV method to optimize my parameter tuning. The four parameters I used were criterion, max depth, min_samples_split, and splitter method. I believe all of these could potentially be seen as "pruning" methods (since they have the potential to reduce the number of nodes), but max depth is the true, "most pure" pruning parameter, as it directly controls the number of nodes created in the algorithm.

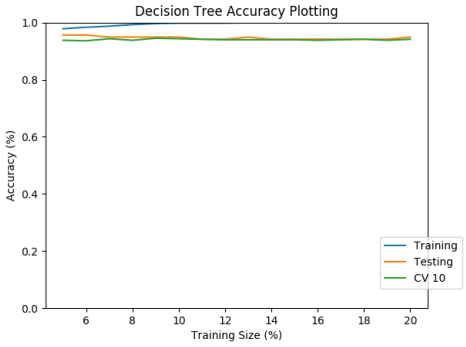
INITIAL (Split at 0.8(maximizes CV accuracy), short run time):
Testing accuracy (average across 10 runs): **95.18%**
Standard deviation of 10 runs for testing accuracy: **.0139**
CV accuracy (average across 10 runs): **94.84%**
Standard deviation of 10 runs for CV accuracy: **.009**
Average Standard Deviation of k folds(across 100 runs): **.0267**
Standard deviation of the average stdevs for k-folds: **.005**

FINAL:
Testing accuracy (average across 10 runs): **95.77%**
Standard deviation of 10 runs for testing accuracy: **.013**
CV accuracy (average across 10 runs): **94.9%**
Standard deviation of 10 runs for CV accuracy: **.007**
Average Standard Deviation of k-10 folds(across 10 runs): **.028**
Standard deviation of the average stdevs for k-10-folds: **.007**

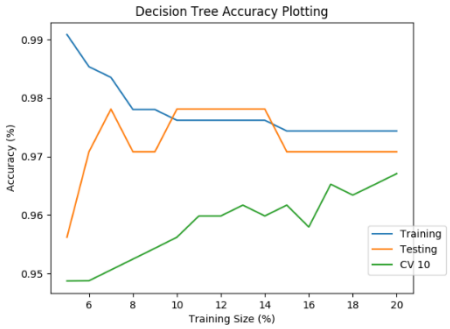
The following optimal hyper parameters were produced over 10 iterations:

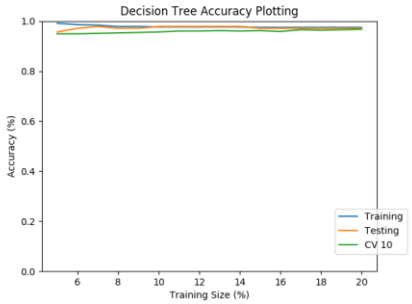
'min_samples_split': 18, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 5}
{'min_samples_split': 14, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 10}
{'min_samples_split': 9, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 11}
{'min_samples_split': 20, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 6}
{'min_samples_split': 16, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 6}
{'min_samples_split': 15, 'splitter': 'random', 'criterion': 'entropy', 'max_depth': 17}
{'min_samples_split': 12, 'splitter': 'random', 'criterion': 'entropy', 'max_depth': 17}
{'min_samples_split': 8, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 8}
{'min_samples_split': 10, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 12}
{'min_samples_split': 13, 'splitter': 'random', 'criterion': 'gini', 'max_depth': 19}





*X axis should read max_depth





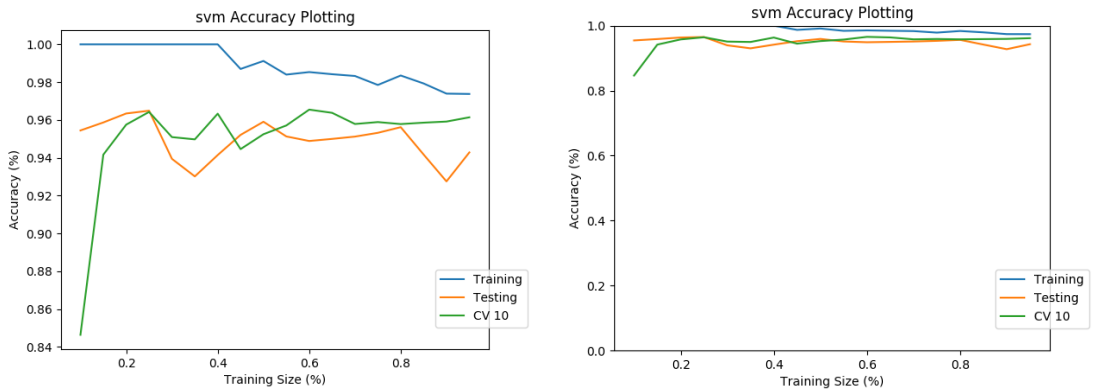
*X axis should read min_samples splits

Graphs for accuracy vs max depth and minimum samples required to split vs accuracy are plotted above. For the max tree graph, we can see exactly how trees over-fit! As the max depth increases, we attain 100% accuracy on the training data because the tree fits all of the samples in the training set. We can see accuracy on the testing set decline as max depth increases, which furthers the point. For the min_samples_split plot, CV accuracy seems to increase as we increase our minimum sample splits by almost 2%, while training accuracy declines by 1.5%. Since the tree requires more samples before splitting, we essentially avoid over fitting because we don't cater to specific records. This is why the training accuracy starts at 100% with low minimum samples required to split. In the future I'll certainly pay close attention to these two hyper-parameters to avoid over-fitting.

In terms of overall accuracy the decision tree accuracies for test and CV over ten replications very slightly improved. I would say this is a good thing – our accuracies for all these algorithms seem to be held in the same small range, so any increase is helpful. Because of the small dataset, I think it would be beneficial next time to set max depth to a constant, smaller value to avoid over fitting and just alter the splitting metric and min_samples_split required.

Adaboost:

Baseline learning curves for the adaboost algorithm look as follows:



The first noticeable thing about this algorithm is its runtime. It seems to be very slow, which is understandable as it's an ensemble classifier, so it must compile lots of weak classifiers to reach a prediction. Majority of the baseline learning curves (default parameters -> Decision Tree Classifier for the base estimator, 50 maximum estimators, learning rate of 1) seem to hang around the lower to upper 90s for all accuracies, so it is on par with the other algorithms. Another interesting observation is all random shuffles of the training set accuracy appear to be at 100% until the training size grows to over 0.4. This doesn't seem too farfetched– with a low variance in the features and small training size, this leads me to believe the algorithm does not need a lot of trees to be generated since the algorithm stops after a perfect fit anyway so the estimators probably perfectly fit on the data very fast. Actually after more thought, adaBoost probably only needs one estimator as it only creates additional estimators to target incorrect points, so it perfectly fits after the first tree. After several replications CV accuracy appears to be maximized at a .8/.2 training testing split, so I'll use this to tune the hyper parameters.

INITIAL (over 10 replications):
Testing accuracy (average across 10 runs): **95.1%**
Standard deviation of 10 runs for testing accuracy: **.018**
CV accuracy (average across 10 runs): **95.7%**
Standard deviation of 10 runs for CV accuracy: **.005**
Average Standard Deviation of k folds(across 100 runs): **.024**
Standard deviation of the average stdevs for k-folds: **.005**

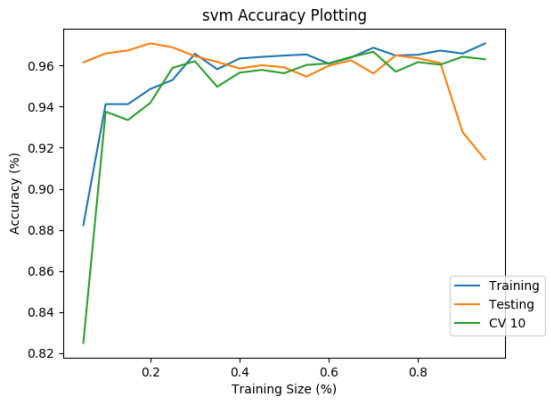
FINAL (over 10 replications)**:**
Testing accuracy (average across 10 runs): **95.6%**
Standard deviation of 10 runs for testing accuracy: **.017**
CV accuracy (average across 10 runs): **95.16%**
Standard deviation of 10 runs for CV accuracy: **.003**
Average Standard Deviation of k-10 folds(across 10 runs): **.026**
Standard deviation of the average stdevs for k-10-folds: **.006**

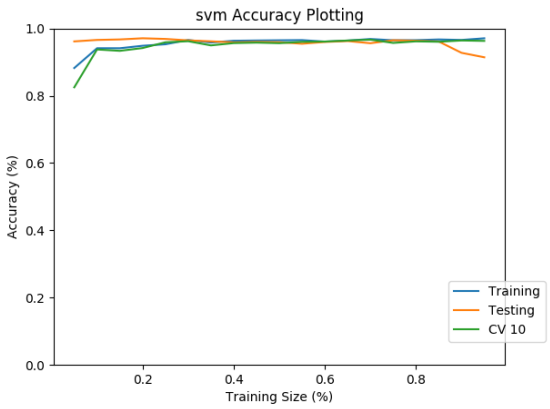
With the following optimal hyper parameters chosen after each iteration:

{'n_estimators': 150, 'learning_rate': 0.01}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 10, 'learning_rate': 0.3}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 150, 'learning_rate': 0.1}
{'n_estimators': 10, 'learning_rate': 0.1}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 150, 'learning_rate': 0.1}
{'n_estimators': 100, 'learning_rate': 0.1}
{'n_estimators': 10, 'learning_rate': 0.3}

A slight (.5%) increase in testing accuracy with a tradeoff of about the same decrease in CV accuracy, which isn't too shabby. It is obvious that there is an inverse relationship between number of estimators and the learning rate -> less estimators means that the algorithm needs to be much more aggressive over fewer models. This algorithm performed very similarly to the single decision tree. It appears that in this ensemble method, the first tree is already very accurate, so additional methods do not help it out immensely.

Neural Network:





Initial analysis of the non-tuned, baseline learning curves reveal similar values for the training, testing, and CV accuracies. It appears that the train/test split that maximizes the CV accuracy is .65, so this is the value I will use for hyper parameter optimization.

INITIAL (over 10 replications):
Testing accuracy (average across 10 runs): **95.67%**
Standard deviation of 10 runs for testing accuracy: **.009**
CV accuracy (average across 10 runs): **96.78%**
Standard deviation of 10 runs for CV accuracy: **.005**
Average Standard Deviation of k folds(across 100 runs): **.026**
Standard deviation of the average stdevs for k-folds: **.006**

FINAL (over 10 replication)**:**
Testing accuracy (average across 10 runs): **96%**
Standard deviation of 10 runs for testing accuracy: **.009**
CV accuracy (average across 10 runs): **96.4%**
Standard deviation of 10 runs for CV accuracy: **.006**
Average Standard Deviation of k-10 folds(across 10 runs): **.024**
Standard deviation of the average stdevs for k-10-folds: **.003**

Additionally, the following parameters were optimal over 10 iterations:
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.9}
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'invscaling', 'momentum': 0.5}
{'learning_rate': 'invscaling', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.9}
{'learning_rate': 'constant', 'momentum': 0.7}
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.5}
It appears that adjusting the parameters did not have a large effect on the accuracy. The optimizer appears to have favored a constant learning rate with a 0.5 momentum. This lower momentum parameter may have allowed to not overshoot a local minima, thus increasing our accuracy.

A summary of the final accuracy for each algorithm:
Knn: 96.08%
SVM: **96.42%**
Decision Tree: 95.77%
Adaboost: 95.6%
NN: 96%

SVM has the highest accuracy out of the 5 algorithms. I think the fact that there were 5 dimensions in the dataset after feature selection gave the SVM a slight advantage due to its ability to deal with higher dimensions more effectively. Ultimately though, all algorithms performed very similarly. I think the ease of the UCI dataset helped a lot with the high accuracies.

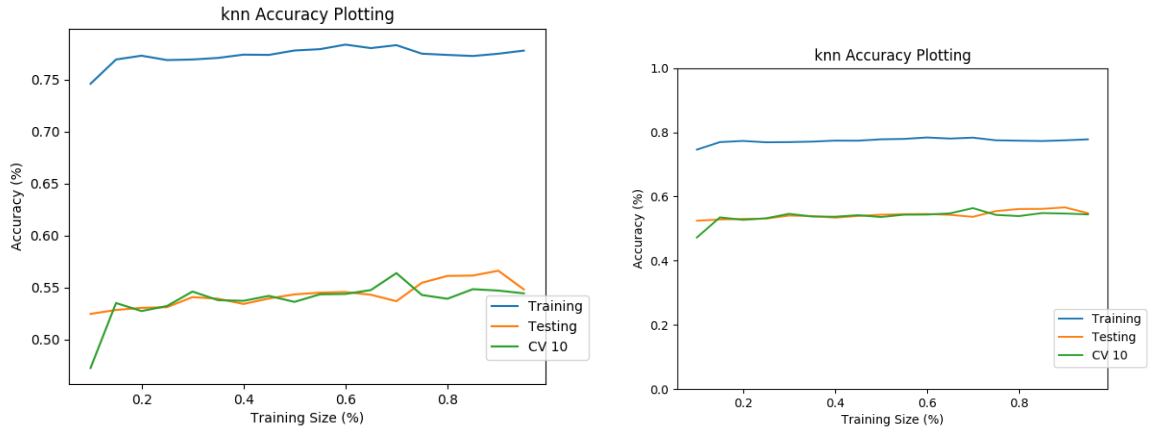
Basketball dataset:

Preprocessing:

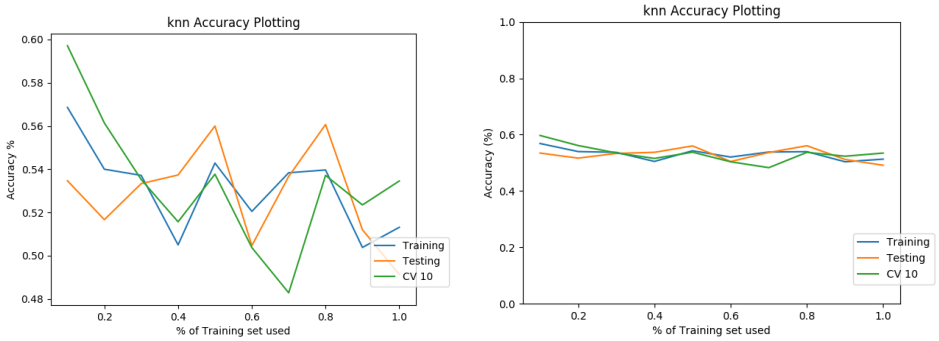
I consistently use random resampling without replication across all of my algorithms so as to avoid large compile times, thus creating a subset of 5000 records. I kept the random seed generator consistent as well so all algorithms would be using the same dataset (I did however, use random shuffling for the training/test sets). I also used scaling ([0-1]) so as to not favor large scale attributes, and used a feature selection method removing low variance variables (threshold of 0.1 ➔ features with variances less than that were removed) as they lacked any predictive power. As I understand it, they do not provide additional information because their values remain consistent even as we change the values of other variables. This reduced my feature count to 6, down from 14 originally.

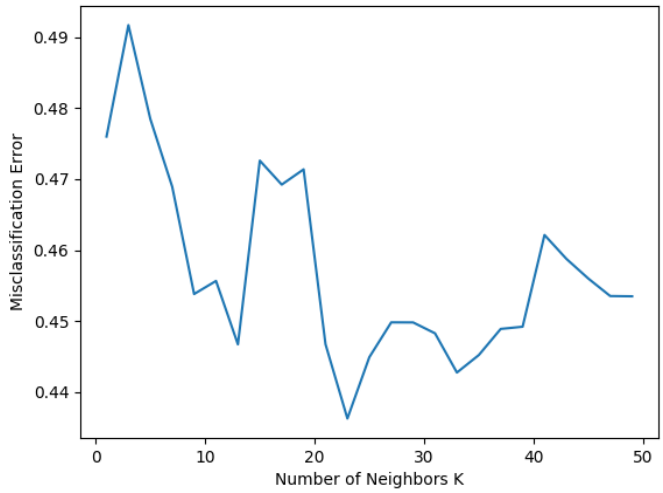
KNN:

Initial learning curves of training splits appear as follows:



CV estimates look to maximize accuracy around 0.7, so I'll split there and check our %training used learning curves as well as use it for hyper parameter tuning. At a first glance the training set accuracy appears to have a much, much higher accuracy than our training and testing set (20% - this seems drastic). This was also consistent across multiple replications. However, after some investigating I discovered that k was quite small during these runs at 3, so it was very possible that the training data was querying against itself, creating a skewed training accuracy. This seems to be normalized after finding an optimal and larger value for k of 23.





Parameter tuning produces a minima for k at k =23, as the graph above illustrates. Using this hyper parameter setting I obtained the following metrics (after 100 replications):

INITIAL(random, feature selection, scaling, cv folds = 10**,** knn = 10)
Testing accuracy (average across 100 runs): **53.9%**
Standard deviation of 100 runs for testing accuracy: **.019**
CV accuracy (average across 100 runs): **54.08%**
Standard deviation of 100 runs for CV accuracy: **.012**
Average Standard Deviation of k folds (across 100 runs): **.0227**
Standard deviation of the average stdevs for k-folds: **.006**

FINAL (K = 23)
Testing accuracy (average across 100 runs): **53.8%**
Standard deviation of 100 runs for testing accuracy: **.0179**
CV accuracy (average across 100 runs): **54.07%**
Standard deviation of 100 runs for CV accuracy: **.011**
Average Standard Deviation of k folds (across 100 runs): **.0242**
Standard deviation of the average stdevs for k-folds: **.006**

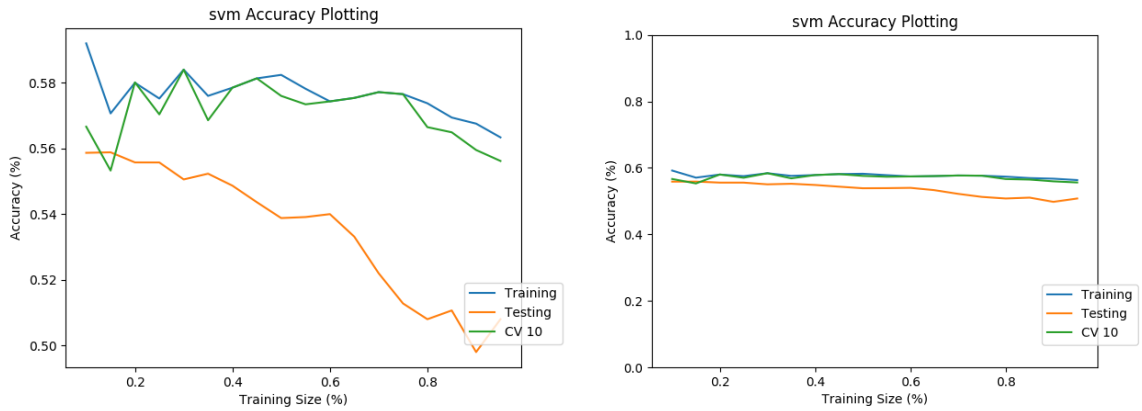
Curious again as to why I was not getting results, I instead used the optimizer function over 15 replications with k values ranging from 1 to 200 and got a much better result! I've realized this was because I optimized the value of k with each iteration of the randomly generated training set, instead of using a non-tuned k for each random set. Also the dataset was much larger, initially my values of k only reached 50.

FINAL (K optimized for each iteration**)**
Testing accuracy (average across 100 runs): **55.19%**
Standard deviation of 100 runs for testing accuracy: **.017**

CV accuracy (average across 100 runs): **55.1%**
Standard deviation of 100 runs for CV accuracy: **.011**
Average Standard Deviation of k folds (across 100 runs): **.019**
Standard deviation of the average stdevs for k-folds: **.007**

These were the k values I obtained over the final 10 runs:
{'n_neighbors': 166}
{'n_neighbors': 196}
{'n_neighbors': 196}
{'n_neighbors': 76}
{'n_neighbors': 106}
{'n_neighbors': 61}
{'n_neighbors': 76}
{'n_neighbors': 61}
{'n_neighbors': 151}
{'n_neighbors': 76}
All above 50 easily. With this new tuning I saw an increase of about 1.4%! I will attribute the large values of k to low variance among the features like the knn from dataset 1.

SVM:

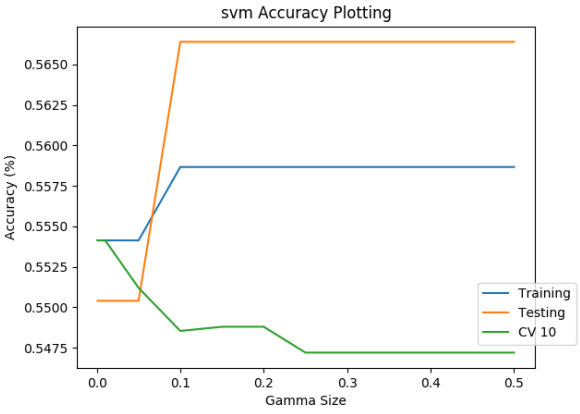


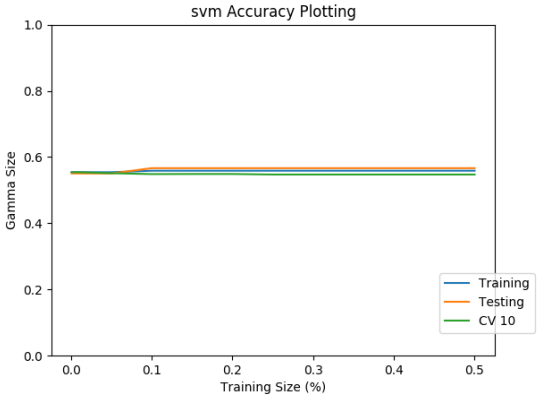
Our first analysis of the train/split size vs accuracy reveals an optimal split at about .75 for CV. I use this to tune the hyper parameters gamma, kernel type, and C for the SVM.

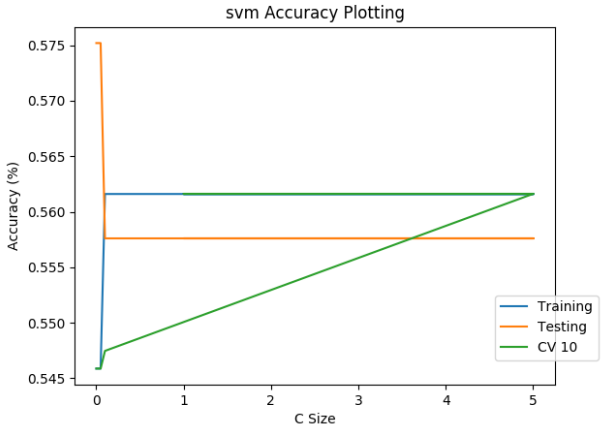
INITIAL (split at .75, random, scaling, feature selection):
Testing accuracy (average across 10 runs): **56.36%**
Standard deviation of 10 runs for testing accuracy: **.009**
CV accuracy (average across 10 runs): **55.82%**
Standard deviation of 10 runs for CV accuracy: **.005**
Average Standard Deviation of k folds(across 100 runs): **.0207**
Standard deviation of the average stdevs for k-folds: **.003**

FINAL:
Testing accuracy (average across 10 runs): **54.86%**
Standard deviation of 10 runs for testing accuracy: **.008**
CV accuracy (average across 10 runs): **55.9%**
Standard deviation of 10 runs for CV accuracy: **.006**
Average Standard Deviation of k-10 folds(across 10 runs): **.018**
Standard deviation of the average stdevs for k-10-folds: **.005**

By far the slowest to run. 10 iterations with parameter ranges at the following for parameter tuning: {'kernel':('linear', 'rbf'), 'C':[1,2,3,4,5,6,7,8,9,10], 'gamma':
[0.01,0.02,0.03,0.04,0.05,0.10,0.2,0.3,0.4,0.5]}
This resulted in a bout a 2 hour compile time. I obtained the following optimal hyper parameters for each of the 10 iterations:
{'kernel': 'rbf', 'C': 1, 'gamma': 0.2}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.05}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.05}
{'kernel': 'rbf', 'C': 9, 'gamma': 0.05}
{'kernel': 'rbf', 'C': 2, 'gamma': 0.05}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.05}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}







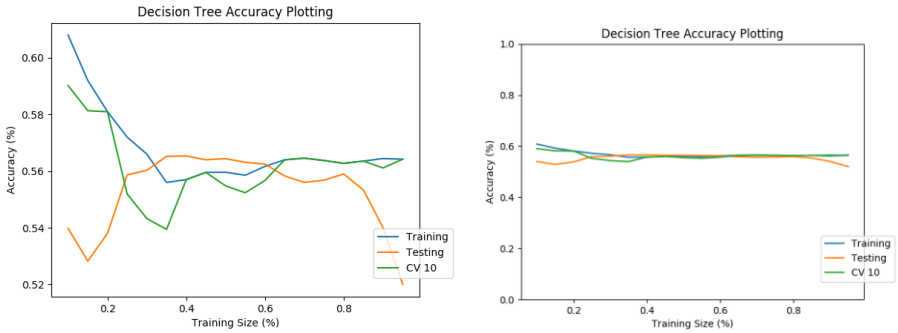
Disappointed by the reduction in accuracy I looked into graphing the parameter sizes vs accuracy. I noticed how the C value for all of the optimal values were 1, which was my lowest value set in the parameter options for the optimizer. This likely means that my data had values far away from the hyper plane which were not being taken into consideration, and potentially overfitting on the training set. I included smaller values for the optimizer and obtained the following over 10 new replications:

{'kernel': 'rbf', 'C': 0.1, 'gamma': 0.5}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'rbf', 'C': 0.1, 'gamma': 0.5}
{'kernel': 'rbf', 'C': 0.1, 'gamma': 0.5}
{'kernel': 'linear', 'C': 0.0001, 'gamma': 0.01}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.5}
{'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
{'kernel': 'linear', 'C': 0.0001, 'gamma': 0.01}
{'kernel': 'linear', 'C': 0.0001, 'gamma': 0.01}
{'kernel': 'rbf', 'C': 0.1, 'gamma': 0.5}

NEW FINAL:
Testing accuracy (average across 10 runs): **55.45%**
Standard deviation of 10 runs for testing accuracy: **.01**
CV accuracy (average across 10 runs): **55.7%**
Standard deviation of 10 runs for CV accuracy: **.006**
Average Standard Deviation of k-10 folds(across 10 runs): **.019**
Standard deviation of the average stdevs for k-10-folds: **.006**

With C reduced to a smaller value, my accuracy from the past tuning increased by ~.7%.

Decision Tree:



:

Initial graphs of this decision tree yet again appear to have relatively low accuracies like with SVM and Knn. Training size split seems to max CV accuracy around 0.75, so we will use this for hyper parameter tuning.

INITIAL (over 5 replications):
Testing accuracy (average across 10 runs): **55.1%**
Standard deviation of 10 runs for testing accuracy: **.013**
CV accuracy (average across 10 runs): **54.1%**
Standard deviation of 10 runs for CV accuracy: **.005**
Average Standard Deviation of k folds(across 100 runs): **.025**
Standard deviation of the average stdevs for k-folds: **.007**

FINAL (incredibly slow runtime, approximately 30 minutes for 5 iterations, likely due to hyper parameter optimization)**:**
Testing accuracy (average across 10 runs): **60.05%**
Standard deviation of 10 runs for testing accuracy: **.010**
CV accuracy (average across 10 runs): **59.69%**
Standard deviation of 10 runs for CV accuracy: **.003**
Average Standard Deviation of k-10 folds(across 10 runs): **.026**
Standard deviation of the average stdevs for k-10-folds: **.008**

They hyper parameter ranges I provided to the GraphCV function were as follows:

parameter ranges as the following: {'criterion':('gini', 'entropy'), 'max_depth':[1,2,3,4,5,6,7,8],
'min_samples_split':[2,3,4,5,6,7,8,9,10,11,12], 'splitter':('best','random')}

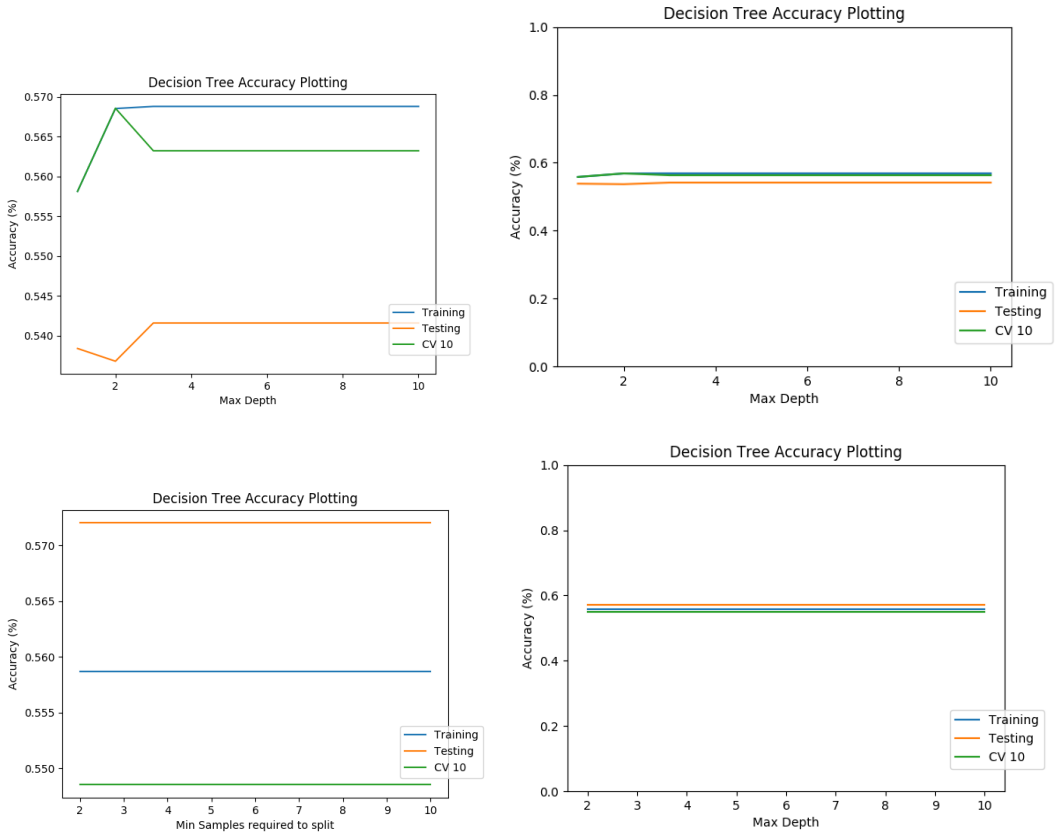
Additionally these are the optimal hyper parameters obtained per iteration:

{'min_samples_split': 2, 'splitter': 'best', 'criterion': 'gini', 'max_depth': 1}
{'min_samples_split': 8, 'splitter': 'random', 'criterion': 'entropy', 'max_depth': 2}
{'min_samples_split': 2, 'splitter': 'best', 'criterion': 'gini', 'max_depth': 3}
{'min_samples_split': 6, 'splitter': 'random', 'criterion': 'entropy', 'max_depth': 4}

{'min_samples_split': 3, 'splitter': 'random', 'criterion': 'entropy', 'max_depth': 5}

Ideal hyper parameters based on modes: {min samples split -2, splitter – random, criterion -- entropy, max_depth -- 3

With these settings I observed an increase of 5% with hyper parameter tuning! This was quite exciting as the other algorithms thus far seemed to have been doing poorly and did not improve significantly after tuning. I can theorize the reason it performed so well after pruning was because of the low number of features. Since they already had (somewhat) low variance to begin with (even after thresholding – the remaining six basketball threshold variables all have variances below .25 -➔ checked this using trial and error on the thresholding function), I would imagine that pruning the tree would remove any unnecessary noise from the prediction algorithm, just leaving the high information gain variables to complete most of the predicting process.

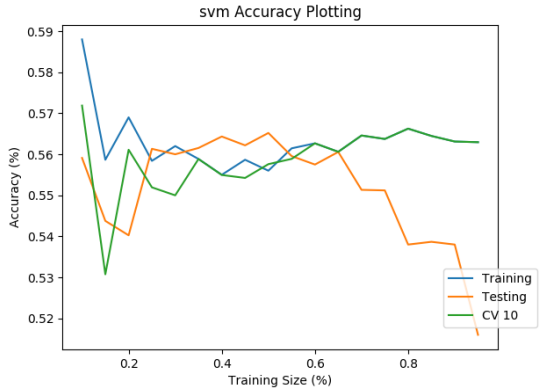


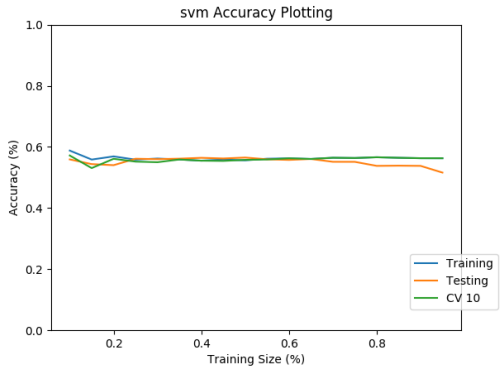
Interestingly enough, minimum samples required to split had no effect on the performance at all. I even tested it with high values (up to 125) as I suspected the size of the dataset might affect it, but still straight lines across the board. Even max depth became stagnant after 2. I would attribute this to low variance in the features ***if*** I was getting high accuracy overall, but this is obviously not the case. If anything, with training data at the very least max depth should scale with accuracy because as I understand it, if there is a leaf for every sample in the data, the training accuracy should be 100%. Regardless, the hyper parameter optimization produces

good results, so I'm guess that may be a combination of these two parameters which increased accuracy.

Adaboost:

Initial graphs to find train test split looks as follows:





CV is maximized at 0.8, so I used that for tuning as per standard:

Hyper parameter tuning:
INITIAL (over 10 replications):
Testing accuracy (average across 10 runs): **56%**
Standard deviation of 10 runs for testing accuracy: **.019**
CV accuracy (average across 10 runs): **56%**
Standard deviation of 10 runs for CV accuracy: **.005**
Average Standard Deviation of k folds(across 100 runs): **.022**
Standard deviation of the average stdevs for k-folds: **.00**

FINAL (over 10 replication)**:**
Testing accuracy (average across 10 runs): **55.36%**
Standard deviation of 10 runs for testing accuracy: **.016**
CV accuracy (average across 10 runs): **56.11%**
Standard deviation of 10 runs for CV accuracy: **.004**
Average Standard Deviation of k-10 folds(across 10 runs): **.023**
Standard deviation of the average stdevs for k-10-folds: **.006**

Final:
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 50, 'learning_rate': 0.1}
{'n_estimators': 150, 'learning_rate': 0.01}
{'n_estimators': 200, 'learning_rate': 0.01}
{'n_estimators': 50, 'learning_rate': 0.1}

{'n_estimators': 10, 'learning_rate': 0.0001}
{'n_estimators': 50, 'learning_rate': 0.1}

Unfortunately, a slight decrease here in accuracy, however a decrease in standard deviation balances that out somewhat. The mode of max estimators was 50, which is the default. This method did not come close to my decision tree, but performed on par with the rest of the algorithms.

Neural Network

Initial analysis of the train split graphs calls for a .65 split, and I use this to tune my hyper parameters.

INITIAL (over 10 replications):
Testing accuracy (average across 10 runs): **55.54%**
Standard deviation of 10 runs for testing accuracy: **.01**
CV accuracy (average across 10 runs): **55.8%**
Standard deviation of 10 runs for CV accuracy: **.004**
Average Standard Deviation of k folds(across 100 runs): **.019**
Standard deviation of the average stdevs for k-folds: **.004**

FINAL (over 10 replication)**:**
Testing accuracy (average across 10 runs): **55.66%**
Standard deviation of 10 runs for testing accuracy: **.008**
CV accuracy (average across 10 runs): **56.1%**
Standard deviation of 10 runs for CV accuracy: **.006**
Average Standard Deviation of k-10 folds(across 10 runs): **.023**
Standard deviation of the average stdevs for k-10-folds: **.006**

{'learning_rate': 'invscaling', 'momentum': 0.7}
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.9}
{'learning_rate': 'constant', 'momentum': 0.5}

{'learning_rate': 'constant', 'momentum': 0.5}
{'learning_rate': 'constant', 'momentum': 0.7}
{'learning_rate': 'adaptive', 'momentum': 0.9}
{'learning_rate': 'constant', 'momentum': 0.7}
{'learning_rate': 'constant', 'momentum': 0.9}

Parameter tuning did not affect this neural network's performance much, similar to the neural network from the first dataset. A constant learning rate was the mode, but momentum seemed to have a scattered range, so it likely depended on the randomly selected training set.

Summary of accuracy for each algorithm:
Knn: 55.19%
SVM: 55.45%
Decision Tree: 60.05%
Adaboost: 55.36%
NN: 55.66%

The decision tree heavily outperformed the other algorithms by 4-5%. My hypothesis for this is the tremendous increase it experienced after hyper-parameter tuning. Each iteration had very different hyper parameters, so the tree seemed to adjust much better for each run. However, all of the algorithms generally performed poorly. I describe why below:

Additional Thoughts about Results for the Basketball Dataset
I had fun with this dataset, but looking back I realized that this may have not been the best dataset for a classification problem. I framed the data in such a way to attempt to try and predict if a shot is made based on distance to the basketball, defender distance, etc, but I really should've taken players into account. The player plays such a large role in shot accuracy, and it certainly holds high predictive power. I still think it was ok to omit player based on the way I framed the problem, but an even better problem would've been to predict if a particular player will make a shot based on the metrics lined out in the dataset. Another thought is how I resampled the data. Since this was only 5% of the overall dataset, it could've been an outlier and not representative of the data. The ideal way would to randomly resample portions of the data and find averages over several iterations, but in my case this would simply take too much computing power and too much time. Regardless, it was interesting to compare the results of the algorithms, even if they all ran poorly.

Sources:
https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)
https://www.kaggle.com/dansbecker/nba-shot-logs
http://scikit-learn.org/stable/modules/preprocessing.html
https://tgmstat.wordpress.com/2014/03/06/near-zero-variance-predictors/
https://stackoverflow.com/questions/36965866/maximum-depth-for-a-random-tree
https://machinelearningmastery.com/parametric-and-nonparametric-machine-learning-algorithms/

http://scikit-learn.org/stable/modules/feature_selection.html#feature-selection
http://scikit-learn.org/stable/modules/tree.html
http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing
https://stackoverflow.com/questions/17034442/scaling-inputs-data-to-neural-network
https://stats.stackexchange.com/questions/9357/why-only-three-partitions-training-validation-test
https://stats.stackexchange.com/questions/19048/what-is-the-difference-between-test-set-and-validation-set
http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html
https://stats.stackexchange.com/questions/19048/what-is-the-difference-between-test-set-and-validation-set
http://blog.hackerearth.com/simple-tutorial-svm-parameter-tuning-python-r
https://stackoverflow.com/questions/30102973/how-to-get-best-estimator-on-gridsearchcv-random-forest-classifier-scikit
http://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#sphx-glr-auto-examples-neighbors-plot-classification-py
https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/
http://scikit-learn.org/stable/modules/neighbors.html
http://scikit-learn.org/stable/modules/svm.html
http://scikit-learn.org/stable/modules/neural_networks_supervised.html
http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier.fit
http://benalexkeen.com/decision-tree-classifier-in-python-using-scikit-learn/
https://stackoverflow.com/questions/39002230/possible-to-modify-prune-learned-trees-in-scikit-learn