Isabella Silva
CS4641
Machine Learning

Assignment 1: Supervised Learning

For this assignment I will be analyzing the following supervised learning algorithms: Decision Trees, Neural Nets, K-Nearest Neighbor (KNN), Boosting, and Support Vector Machine (SVM). I will be using this algorithms on two datasets. Both datasets are taken from Kaggle, the first of which explores exoplanet stars and the second of which determines country happiness.
For this project I used the Scikit package to train and test my data.
https://www.kaggle.com/keplersmachines/kepler-labelled-time-series-data
https://www.kaggle.com/unsdsn/world-happiness

DATASET AND CLASSIFICATION

Kepler [Labelled Time Series] Data

I chose the first dataset because I think that the idea of applying machine learning to space exploration is very creative and interesting. This dataset classification uses the flux of a star to determine if it is a host to a planet. Using data to determine if a star is exoplanetary is a very binary problem, but the logic behind the data can be applied to other areas of space exploration. I think it would be incredible to apply this same type of data to determine if there is correlation between planetary attributes and the existence of liquid water on the planet. This is similar to the current classification that determines if there exist planets nearby each star, similar to the way that our solar system is constructed.

While running each learning algorithm with this dataset, I originally would only use a few hundred rows rather than the full 3,197 while still getting the highest accuracy I could. I did this because the amount of data overwhelmed my computer and while attempting to run an algorithm like the Neural Net, it would run for over an hour before I forced it to terminate. While this does cut down on accuracy, it allowed me to get a good idea of the ratios of accuracy for the functions and learning algorithms I was using. This is because I kept the minimized dataset number constant.

This data set has 3,197 attributes that all relate to the luminosity of the stars. This dataset has 319,800 data points.

World Happiness Data

I chose the second dataset because I think that it would be interesting to be able to determine the ‘happiness’ of a country based purely on data. While this data is from 2015, I think that the factors themselves are still relevant and valid. One of the reasons that I chose this dataset is because it has the potential to be a binary dataset.

The happiness measure was originally a scale from 1.00 to 10.00, with ten being the highest possible happiness and 1 being the lowest. I altered the data to represent ‘happiness’ as binary where a score of below 5 was ‘not happy’ (0), and a score of above 5 was ‘happy’ (1). This allowed me to parse my data more easily because the list of outputs became binary. I also

removed the column that specified the region that the country being evaluated was in. I did this because I did not feel that this attribute was pertinent to the rest of the data or the output.

This dataset has 10 linearly valued attributes and 2 nominal attributes. This dataset has 158 data points.

DECISION TREES

Hypothesis: I believe that the data accuracy will not be very high, but will become higher after pruning for the Kepler *and* Happiness data sets.

Decision tree algorithms use data attributes to create paths to a specific outcome. For training the two datasets I used entropy *and* the gini index to calculate my model's attribute split from the scikit learn package. These functions are used to determine how the decision tree splits data as each decision passes. For the Happiness dataset I decided to use the same attribute split because there was a higher accuracy prediction when I used entropy over the gini index. This is because entropy minimizes log loss for probabilities that distinguish between binary values.

Sklearn.model_selection contains a train_test_split function that I used to train my test data. The train_test_split function splits up the given data into two sets randomly so that one portion of the data is used for training and the other is used for testing. The data is split up repeatedly and randomly to unsure that the data is not being tested on the same data that it is trained on and that it has sufficient information to learn from.

Pruning is an algorithm that strips away attributes of the data that are not relevant to the final output. For a small dataset like the World Happiness Scale, pruning is less effective and can result in pruning branches of the decision tree that are necessary. However with the Kepler Data, which has 5 rows and over 1,000 columns, pruning is important to compensate for the possibility of overfitting. Another advantage of pruning is that it reduces the time complexity of the classifier.

Happiness Decision Tree with Pruning Accuracy:

Training: 98%

Testing: 71%

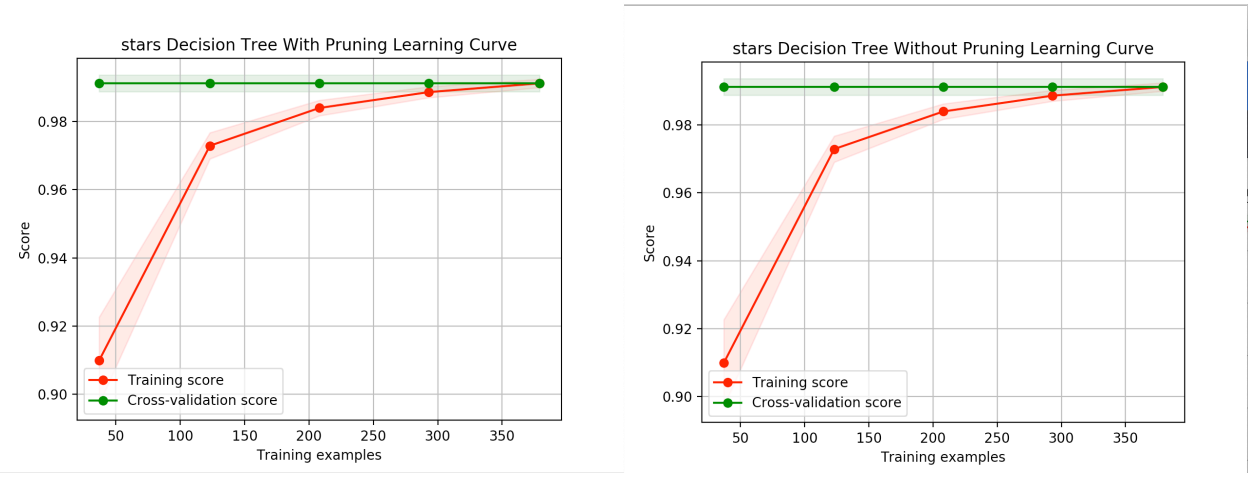
Happiness Decision Tree without Pruning Accuracy:

Training: 100%

Testing: 98%

As can be seen from the graphs above, the decision tree predictions are *more* accurate when the tree has not been pruned. This is a phenomena that occurs when pruning causes the dataset inputs to become over generalized and decrease accuracy the more data is introduced. This is because the inputs are mapping to outputs too often without proper consistency. The cross validation score of 0.5 supports this hypothesis as it demonstrates under-fitting.

Kepler Design Tree with Pruning Accuracy:
Training: 98%
Testing: 97%
Kepler Design Tree without Pruning Accuracy:
Training: 99%
Testing: 98%



The graph for the Kepler graph is interesting because there is virtually no difference between the pruned tree and the unpruned tree. This implies that some of the attributes were inconsequential and when removed they had little effect on the algorithm's learning. Most likely, this is due to the lack of diversity between the attributes. However, the training model was very accurate as the training examples *strongly* effected the learning score of the model. The cross validation score remained at a constant 100% which suggests the tree size is too small for the size of the dataset. This would ideally be corrected after the tree is pruned, however the cross validation score remained constant. The size of the dataset is likely responsible for this even after pruning.

These two datasets reacted differently to pruning due to the difference in their sizes. The much larger dataset did not lose necessary attributes after being pruned, whereas the smaller dataset suffered when key attributes were pruned away.

BOOSTING

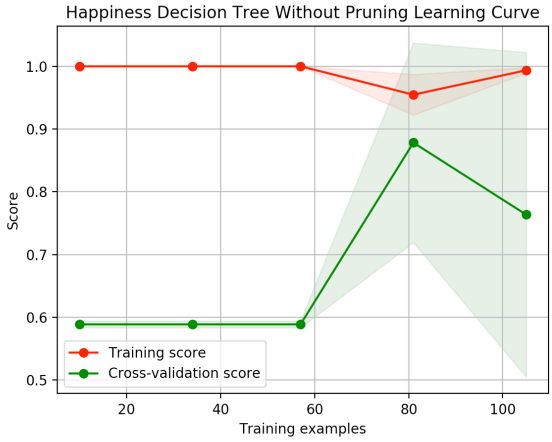
Hypothesis: Boosting will have a higher accuracy and a longer run time than decision trees.

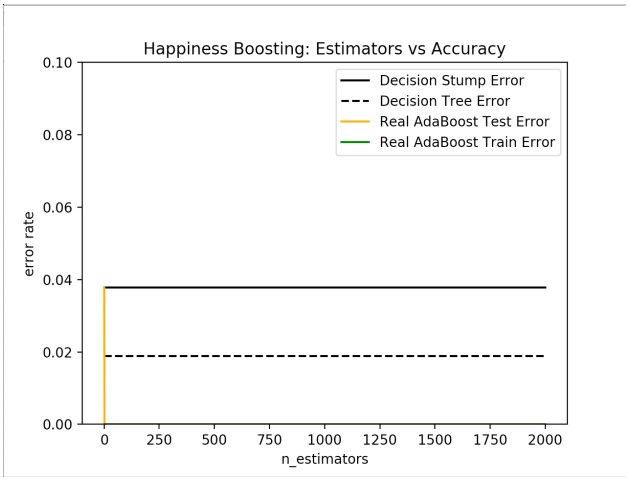
Boosting is similar to decision trees in the way that it analyses data, except that in theory it should be more accurate because it uses multiple calculations and takes the mean. The function that I applied to my decision tree is Python's Adaboost algorithm. This form of boosting uses the Scikit Decision Tree classifier to create multiple decision trees. This algorithm then takes the highest training score and uses the decision tree that is produced to optimize learning. For my boosting implementation I used the same entropy and Gini index functions as I did for my Decision Tree. However, specified my number of estimators hyper-parameter as opposed to the max-depth parameter for the decision tree. This is because for this algorithm I do not want to test the robustness of the algorithm itself, rather the effect of the estimators on the accuracy. To compensate for my issue of over-pruning, I ran my World Happiness Data without being pruned as to avoid losing necessary attributes.

Happiness Boosting Accuracy:

Training: 99%

Testing: 99.2%



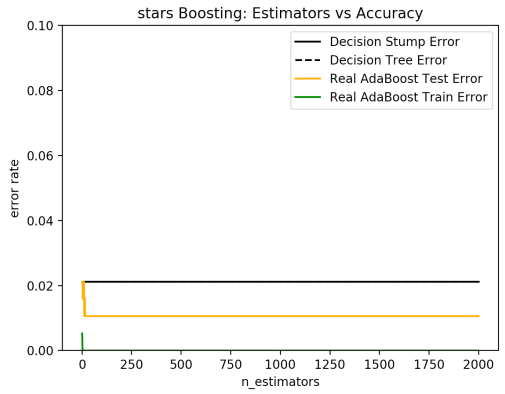


As we can see from the graph on the left the error rate is significantly larger than it had been for the decision tree alone. We can then compare this to the graph on the right which shows the the decision stump, or a decision tree with a max depth of 1, has a *higher* error rate than a larger decision tree. This is supported by our data from the decision tree which shows that as the depth of the tree increases and approaches the given max hyper parameter so does the accuracy of the output. For the boosting algorithm the max depth of the tree was set to be 9 which decreased the error rate by 50%.

Kepler Boosting Accuracy:

Training: 100%

Testing: 97.8%



For the Kepler dataset, in relation to the decision trees we can see that boosting results in fewer errors, but the number of estimators as little effect on the error rate for a decision tree with a max-depth of 1, also known as a decision stump. With this implementation I used pruning as well because of the higher potential for accuracy demonstrated with the singular decision tree algorithm. The hyper-parameters that I altered for the Kepler dataset were the Gini coefficient and the n_estimators. I experimented with setting the n_estimators to various values before settling on 100 which allowed me to reach the lowest error rate. It makes sense that as the n_estimators increases so would the accuracy because the n_estimators specifies the number of sequential trees to be modeled per iteration. Each iteration would then provide the model with a higher chance of success at obtaining the correct output.

As predicted, Boosting was more accurate than the Decision Tree implementation for the smaller dataset and took a longer time to run for both datasets. This algorithm improved accuracy by 1% for the World Happiness dataset but was 1% *less* accurate for the Kepler Dataset.

NEURAL NET

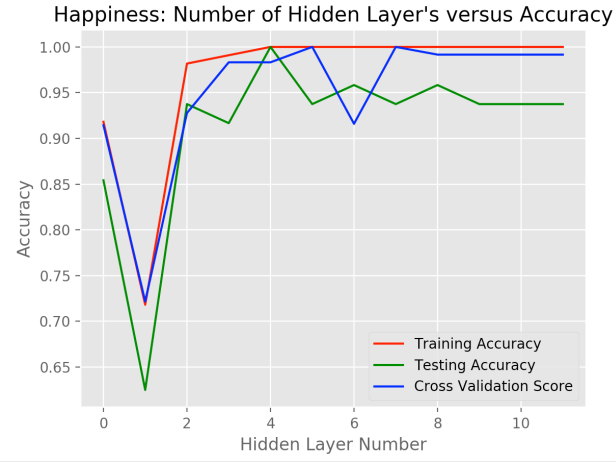
Hypothesis: Given my previous knowledge I expect the Neural Net algorithm to have the highest accuracy with the most efficient runtime complexity.

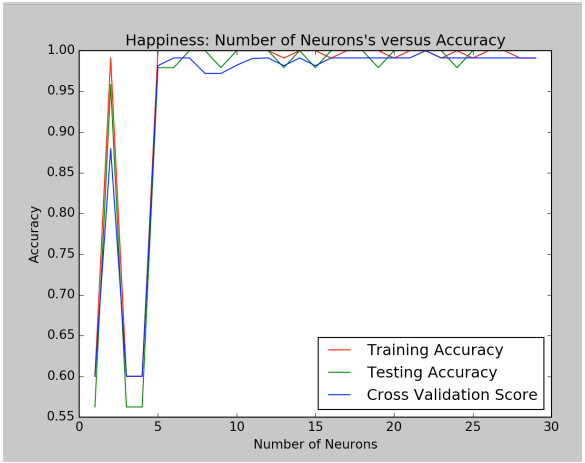
Neural nets mimic biology in the way that they use nodes and weights to piece together attributes and arrive at a singular output. I used the Multilayer Perceptron functions from Scikit to implement this algorithm. I also used the 'Adam' solver for each of my runs as opposed to stochastic gradient descent due to how consistently Adam outperformed SGD. This learning algorithm ran consistently each time with a very high accuracy rate. The hyper parameters that affected the accuracy of these training sets were the number of neurons per iteration and the number of hidden layers. I specified the number of hidden layers to scale relatively to the number of neurons.

Happiness Neural Net Accuracy:

Training: 100%

Testing: 99.2%



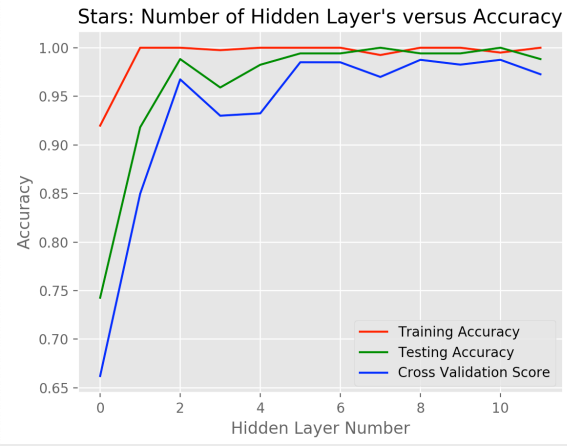


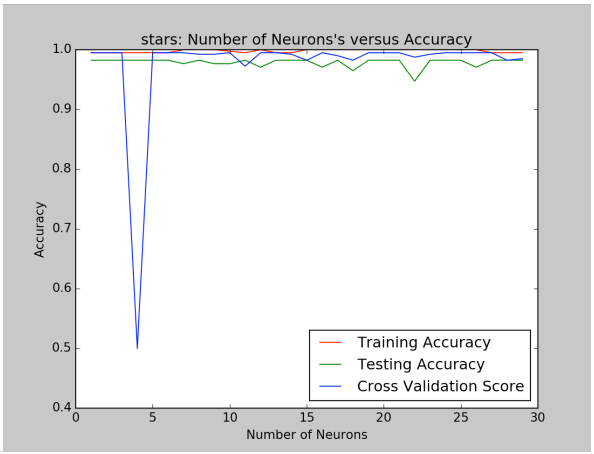
The primary hyper parameters that I studied with this algorithm were the number of neurons and hidden layers. For the World Happiness dataset there was a clear effect of the number of neurons on the accuracy of the data. As the number of hidden layers and neural net expanded, the accuracy of the data increased dramatically. Unfortunately there is a large amount of noise, which can be seen as a result of the attributes that are not relevant to the output, but are still taken into account. Within a neural net, neurons have the role of holding pieces of the data to then be matched up over time. These matches then create a singular path to an output, similar to a decision tree, through multiple hidden layers. Unfortunately, large numbers of attributes that have little import can create the noise that is visible in the graph to the right. For the graph on the left, there is no clear relationships between the number of hidden layers and the accuracy of the algorithm. I find this interesting because a larger number of neurons would imply more hidden layers as well which would in turn imply a higher accuracy.

Kepler Neural Net Accuracy:

Training: 100%

Testing: 99.1%





The Kepler Data reacted differently to the Neural Network algorithm with the given hyper parameters specified. The number of neurons increased consistently with the accuracy of

the data. This relationship between the accuracy and the number of neurons was consistent across the datasets, however the number of hidden layers had a very clear effect on the Kepler Dataset. For both hyper parameters, as the quantity increased so did the accuracy of the dataset. The outlier for this relationships was the cross validation score which had a significant dip towards the smaller amount of neurons and hidden layers. This is likely due to an error when shuffling the data randomly that resulted in a cluster of neurons with little useful data. At such a small number of neurons this is more likely to occur because there is little room for data abstraction.

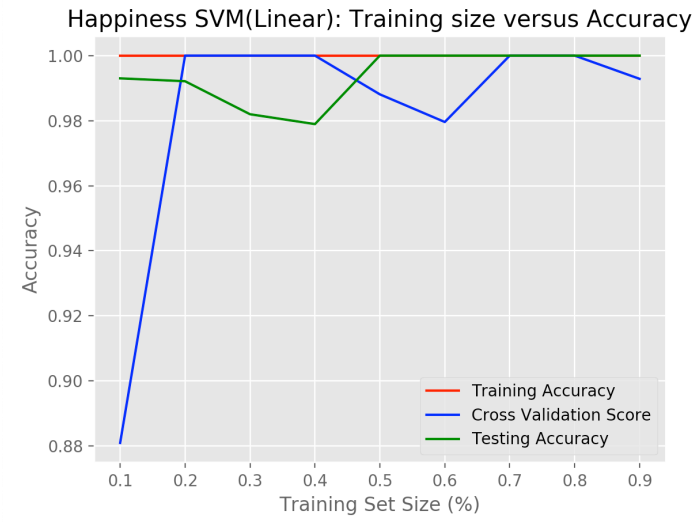
Over all, the neural network was a very effective learning algorithm in terms of accuracy for both datasets.

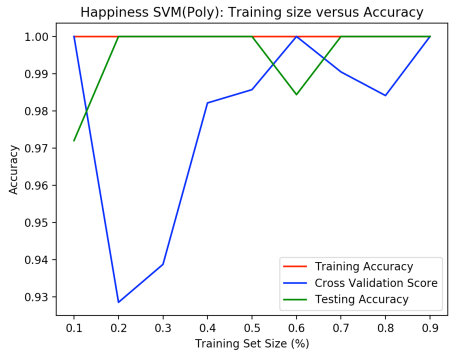
Support Vector Machines

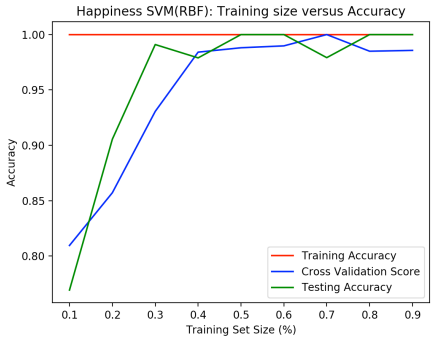
Hypothesis: I think that the Support Vector Machine algorithm will not be as necessary to implement due to the way that my data is structured; there is already a clear relationship between the inputs and the output which would render abstracting the data obsolete.

Support Vector Machines are a supervised learning algorithm that work by taking data and abstracting it into a higher dimension. This allows the algorithm to find a split among the data that is large enough to consider the data bisected. The algorithm then finds a line of best fit between the split data that is equidistant to the two closest data points. This allows the algorithm to create a mesh grid and determine appropriate data classifications. Along with the KNN algorithm, I found the SVM algorithm to be a very interesting supervised learning algorithm. My primary interest in the SVM algorithm is the way that it takes data and abstracts it. I think that the concept of Support Vector Machines is incredibly innovative in the way that it takes a singular data set, in my case a 1-dimensional array, and reformats it to a higher dimension. The way that this allows for relationship analysis from a different angle is something that seems trivial, but is incredibly powerful when implemented.

For this implementation of the SVM algorithm I looked at the three different kernel functions: Radial Basis Function (rbf), Linear Function, and Polynomial Function. I looked at each of these to determine how well each separate kernel function performed in relation to the others.



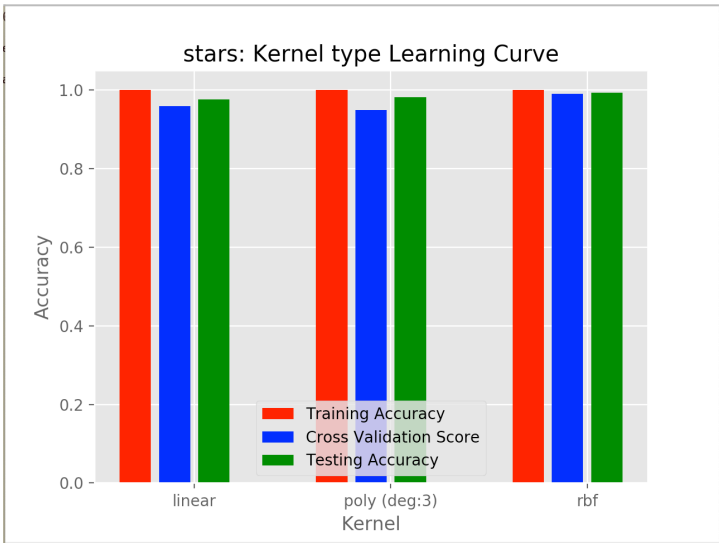




Happiness SVM Accuracy:
Training: 98.32%
Testing: 96.85%

For the Happiness dataset, each of the different SVM functions performed with the same accuracy. I found the Linear Function for the SVM implementation to have the highest accuracy when tested in relation to the training set size. While each function had a similar endpoint (reaching higher accuracy as the training set size got larger), the linear function learned the *fastest* as well as the most consistently when run. One issue that I encountered while working with the SVM algorithm was an inconsistency in learning depending on how the data was split into training and testing data. In some circumstances there was so much noise in my data the graphs became useless. In other situations, particularly while using the Poly and RBF functions, the algorithm had good accuracy, but it was not correlated with the training set size.

Kepler SVM Accuracy:
Training: 99%
Testing: 97.54%



When tested against the Kepler Dataset the SVM functions did *not* perform with the same accuracy, and the difference was actually quite obvious. While I used the same implementation for the SVM algorithm for both datasets, the Radial Basis Function (rbf) had the highest testing

accuracy and cross validation score. This is likely due to the structure of my data. For the World Happiness Dataset the linear function had the highest testing accuracy because I was able to start from a simple hypothesis space and divide my data linearly. However, the Kepler Dataset proved to be more complex and required a non-linear split which was performed by the Radial Basis Function. We can see that the testing accuracy increases as the hypothesis space becomes more complex with the Polynomial function having the next highest accuracy over the Linear function. This is also due to the structure of the data when it is abstracted onto a higher dimension. While the Linear function is able to define a split in the data, it is simply not as accurate as the other two functions.

K- Nearest Neighbor

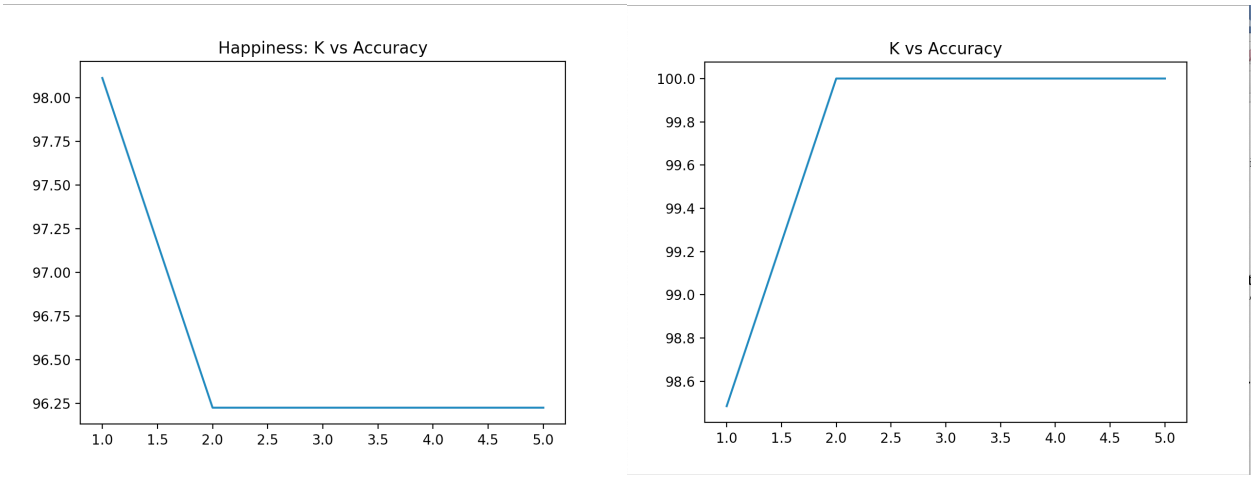
Hypothesis:

The K-Nearest Neighbor algorithm, or KNN, takes data points and graphs them. It then uses the proximity of data points to determine the correct classification for that data point. I personally found the KNN algorithm to be one of the more interesting algorithms because of how many different implementations I found for it. The primary different between each implementation being how to determine 'tiebreakers' when a datapoint fell within the scheme of multiple classifications. I used the K-neighbors classifier of the Scikit package. I also used a for loop to determine the changes in accuracy as the number of neighbors counted, k, iterated from 0 to 10.

Another interesting part of the KNN algorithm is that it is non-parametric. This means that it does not take data distribution and input probability into account when determining the data classification. This allows for more robustness from the algorithm, however the downside is that it is more likely to result in a higher error rate than a parametric algorithm.

Happiness KNN Accuracy:
Training: 100%
Testing: 99.8%

Kepler KNN Accuracy:
Training: 100%
Testing: 99.87%



The graph above on the left shows a drastic decrease in accuracy as the number or neighbors increases. This is to be expected as a larger number of neighbors leaves more chances for the algorithm to choose an incorrect output. For my implementation I used the Scikit Learn

KNeighborsClassifier which will classify the data point depending on the closer classifier. This will occur in the case that the *k* neighbors are equally split into two different classifications. However, in the case that the *k* neighbors are evenly split, *and are equidistant*, the KNeighborsClassifier will actually determine the data classification by choosing the classification depending on the ordering within the training data. For my data this was especially more likely occur because my data points were unweighted.

The graph on the right for the Kepler data gets more accurate as the number of neighbors increases which is not as difficult to understand because there are far fewer outputs for the Kepler dataset that map onto many inputs. Due to the KNN algorithm being non-parametric, it is a far more successful algorithm on datasets with binary outputs. The Kepler dataset has a very clear relationship between the inputs and the outputs which allows for higher accuracy and a faster runtime relative to the size of the data. However, this runtime was still similar to the runtime of the World Happiness Dataset because of how much larger the Kepler dataset is.

IMPROVEMENTS AND COMPARISONS

Decision Trees

For my decision tree implementation I was not satisfied with the over all accuracy. I think that my implementation of the decision tree could have been further optimized with regards to the pruning function. I had a large issue with over- and under-fitting my data with my small and large datasets respectively.

Boosting

One common issue with boosting is that it is a 'weak' learning algorithm that tends to fail as problems get harder. This is due to its high bias which becomes more obvious as the decision trees get deeper. The way that I dealt with this for these particular datasets was by keeping my outputs strictly binary and setting a max_depth for each tree that was generated by the boosting algorithm.

Neural Net

While the neural net was very accurate, my largest concern with this algorithm was the runtime complexity. Even with small datasets, creating each perceptron layer and weighted neuron took a great deal of time and processing power. Given the opportunity to improve this I think it would make sense to either have a smaller dataset or to further limit the maximum number of layers. The issue with this solution is that it trades a more efficient runtime for a less accurate algorithm. This can be solved in a similar way to fitting decision trees if a minimum *and* maximum number of nodes could be found that balances runtime with accuracy.

Support Vector Machines

One error that I encountered while running my SVM implementation was a Memory Error. This occurred when the built in python library (Scikit) attempted to copy over a list of my data while creating a mesh grid. There was an issue because of just how large the Kepler data set was. I managed to get around this error by improving some of my other functions so that the functions themselves were in place rather than copied and deleted, however this made me realize just how much memory an SVM requires.

K-Nearest Neighbor

If I were to improve my KNN algorithm I would increase the specificity of the distance changes that I made. For each iteration, rather than increasing the distance that I searched, I increased the minimum number of nodes to include within my search radius. While this was effective enough, I think that I could have improved accuracy further by having smaller increments and using distance rather than nodes. A drawback to this approach is that it will use more memory as it creates multiple arrays for the training data. Further improvements are weighed voting (which would help to break ties) or data rescaling (to make the distance metric more meaningful).

Happiness Scores for Different Learning Algorithms

Learning Algorithm	Highest Test Accuracy (%)	Model Precision	Clock Time (seconds)
Decision Tree	86.79	89.98	1548208509.18
Boosting	96.22	96.54	1548206077.05
Neural Net	100	100	1548211201.94
KNN	99.98	99.99	1548212857.17
SVM	96.85	97.66	1548208419.91

Kepler Scores for Different Learning Algorithms

Learning Algorithm	Highest Test Accuracy	Model Precision	Clock Time (seconds)
Decision Tree	98.94	97.84	1548211731.03
Boosting	97.88	97.88	1548207132.10
Neural Net	100	100	1548372304.56
KNN	99.87	99.95	1548212378.01
SVM	97.54	96.99	1558108735.91

CONCLUSION

Over all I found each of these learning algorithms to be very interesting and informative. One thing that I appreciated about each of these algorithms was their consistency in their values. When run multiple times, there was a very clear trend to the accuracy of the algorithm. Further, the different algorithms demonstrated different strengths but were all able to handle datasets of very different sizes. While some of the algorithms took significantly more time to train and test on the larger dataset, there were no major problems that arose from working with such a large mass of numbers. Each algorithm showed trends that were very clearly effected by their hyper parameters, which made this assignment very interesting as I could clearly test the effects of

n_estimators, Gini coefficient, max_depth, and more. This helped me visualize the inner workings of each of the algorithms more easily.

Finally, I would consider the Neural Net Algorithm to be the ‘best’ supervised learning algorithm. I chose the Neural Net because the first thing that I looked at for my definition of ‘best’ was accuracy. It is clear that my Neural Net was the most accurate algorithm that I ran. The next variable that I took into account was efficiency via running time. The Neural Net was not the fastest algorithm, especially as the number of hidden layers and neurons increased. It was however, the most efficient in terms of accuracy per amount of time taken. I also found that the Neural Net provided me personally with the most insight into how it was processing the data in relation to its hyper parameters.