Zane Motiwala

Machine Learning Fall 2018

Assignment 1 - Supervised Learning

Classification Problem 1: Using the breast cancer dataset, classify whether the given observations of data represent cancer that is MALIGNANT or BENIGN.

Why is it Interesting?

This problem is interesting to me for 2 reasons. Firstly, I have always been interested in a career where I could work towards improving healthcare and making a positive impact on society. My last job was in healthcare, but I wasn't in a department where improving care was a priority. I would love to work in healthcare again, but with a focus on curing disease using data analysis or something like that. The second reason is that just a couple months ago my Aunt was diagnosed with breast cancer and I have been mentoring my younger cousins who were shocked by the news. Therefore, this dataset resonated right away and it was hard for me to not explore it further. From an ML perspective, I think this is a great example of where Machine Learning can be so useful for our future. There is an incredible amount of information out there and harnessing the power of computing to compliment human intuition and research already used can make for remarkable improvements in healthcare and curing diseases.

If you are using python, this dataset can be loaded directly from the scikit-learn datasets using the commands below. Otherwise it can be downloaded from the UCI ML Repository https://archive.ics.uci.edu/ml/datasets.html.

```
import sklearn
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

Classification Problem number 2:

Using the image segmentation dataset, classify which of seven categories each image belongs to. The seven categories are (BRICKFACE, SKY, FOLIAGE, CEMENT, WINDOW, PATH, GRASS).

Why is it Interesting?

I was interested in this dataset for multiple reasons as well. I took a computational photography course last semester and was hoping we could get into the content of images and being able to figure out what is in pictures using machine learning, but we never did. However, I use Google Photos app and they have search functionality where you can type things like GRASS and they can find which of your photos have that. I am fascinated by this type of technology and the benefits it could bring. I also thought it would be a good contrast to the first dataset since there are multiple classification options and completely different types of features to model with. From an ML perspective, as we move to more and more digital content, being able to quickly and accurately determine what is in digital content can help in many fields (healthcare, legal issues, forensics, national security). The data can be downloaded from the UCI ML Repository https://archive.ics.uci.edu/ml/datasets.html.

Decision Trees:

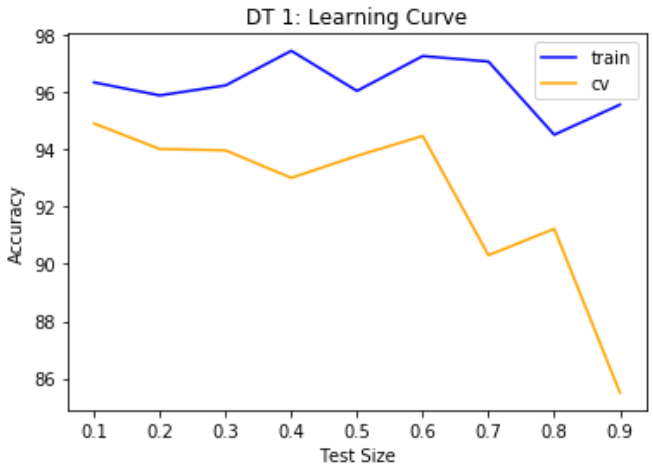
I started the analysis with decision trees as that was the only algorithm I had previously worked with. I watched tutorials online to learn how to run them in Python using Scikit-learn. In the top of the iPython Notebooks I start with a url from the videos where I learned the code from. This goes for each of the notebooks I have for all of the models so I won't repeat this in the below sections.

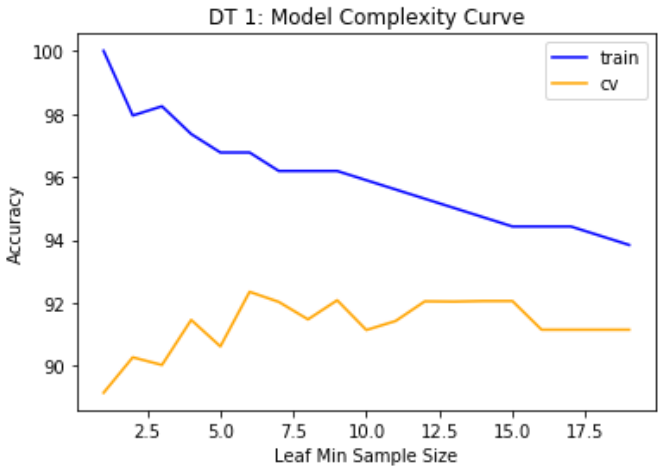
Parameters: The main 2 parameters I played around with and used for pruning were max_depth and min_samples_leaf. I iterated through lists and charted how the training and CV accuracy change as these parameters change. As max_depth increases the training accuracy increases, but cv decreases as the model gets more complex (hence overfits). The number of leafs was the opposite. If there is only 1 leaf then the model overfits and as the leaf sizes increase the model better represents the data. However, after a certain point it begins to underfit the data so too little or too much pruning are both bad.

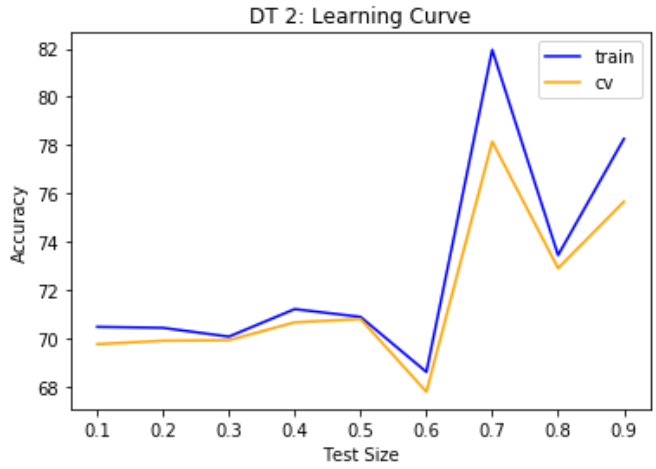
Analysis of algorithm: I was surprised how well decision tree did. I thought it would not work well with the image segmentation algorithm (and it didn't for a while), but then after a lot of tweaking of the parameters I was able to get pretty high accuracy score. After analyzing which features were the most important it looked like only a few features were very important. This probably helped the Decision Tree

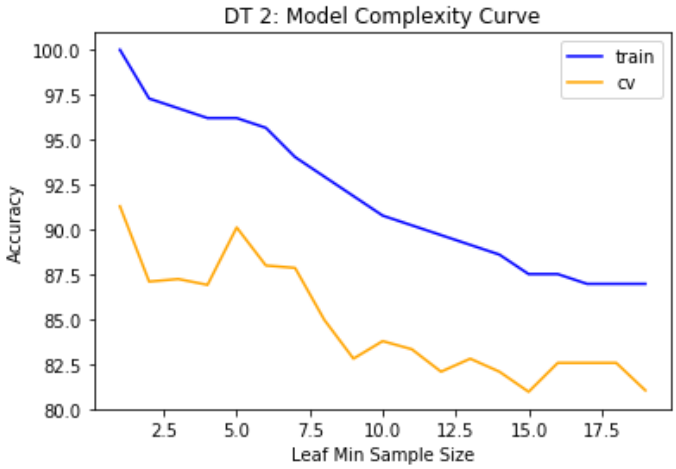
algorithm be able to perform well. One of the downsides of Decision Trees is that once you split subsequent decisions and payoffs may be based primarily on expectations and those may not be right.

I didn't see any higher scores on test data from using entropy to split the branches/features so stuck with the default of GINI.









Boosting:

I performed boosting next to see how this algorithm would improve the previous decision tree. I had a little experience before learning about random forests and bagging so was really interested to see how

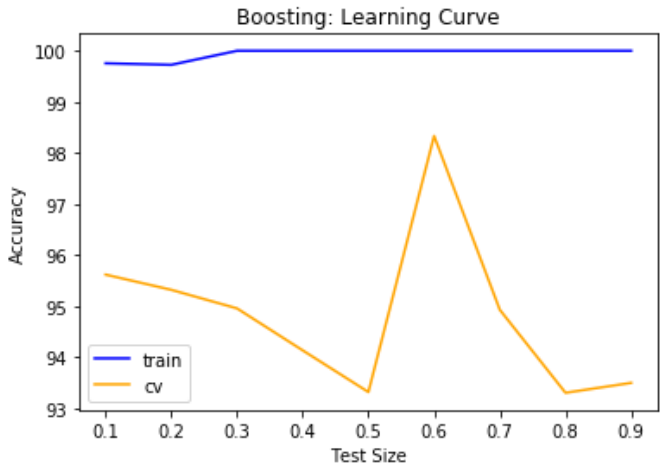
boosting would work. The first thing I did after getting boosting to work was to check the confusion matrix of the decision tree on the breast cancer data and compare it to boosting. I could already tell just a few seconds after running the GradientBoostingClassifier with 20 learners that boosting would improve my results. The decision tree had an accuracy of 90% vs this was already at 94% (before tweaking any parameters.

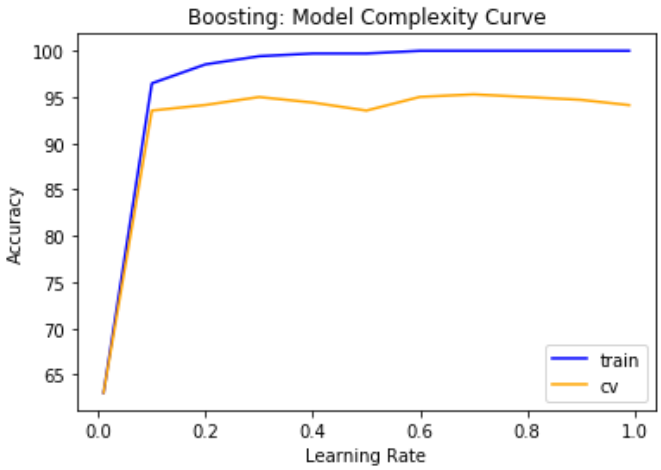
Parameters: The 2 parameters that I modified were N_estimators and the learning_rate. The n_estimators is essentially the number of trees or learners that are used. I thought that as I increased the number of estimators I would start to see a decline in the results. However, at a certain point the testing and training just stop getting better. There isn't a decline as you use with other algorithms due to over-complexity or over fitting. After about 50 n_estimators (up to 1,000) I got the same accuracy on both training and cross-validation. The learning_rate is known to be an effective way to slow down the learning between trees. Otherwise, boosting can quickly begin to overfit and make the model to complex. Typically, there are lower ranges for the learning rate, but in my model I found 0.5 to get the best accuracy result.

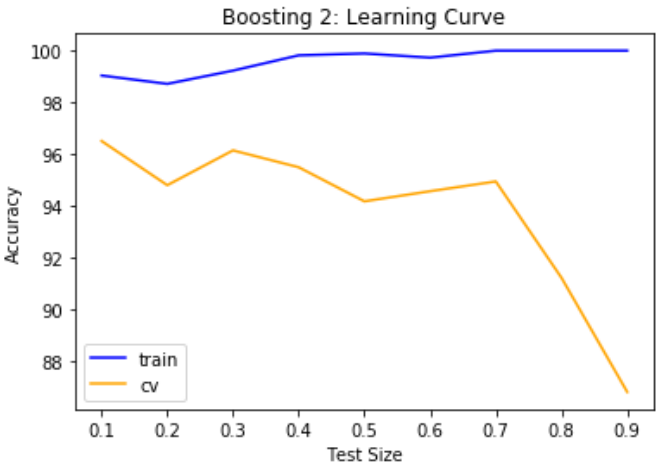
Analysis of algorithm: Boosting is a very quick and powerful way to enhance your algorithms performance. In my research, I read it is a typical way for many of the top performers on Kaggle competitions to get good results. It is based on a simple concept of combining weak learners to continuously iterate and improve your algorithm. I was able to get better results as well using boosting. Interestingly, as training error was really high no matter what the size of the testing set was. There are a few things to be careful for:

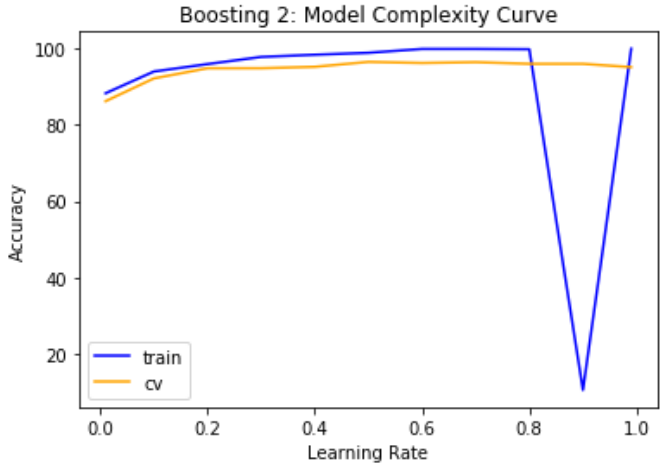
1. Decision trees were easier to interpret as you could print out the actual tree. In this case there are several learners/trees so you can't really see it.

2. It takes longer to run and can quickly begin to overfit the data or make it too complex so it is important to control for that and tune the parameters. I found that it didn't take too much more time though as you can be more aggressive with pruning the max depth to only 2 or 3 and the leaf sample size to larger numbers and still get better results. Pretty cool stuff!







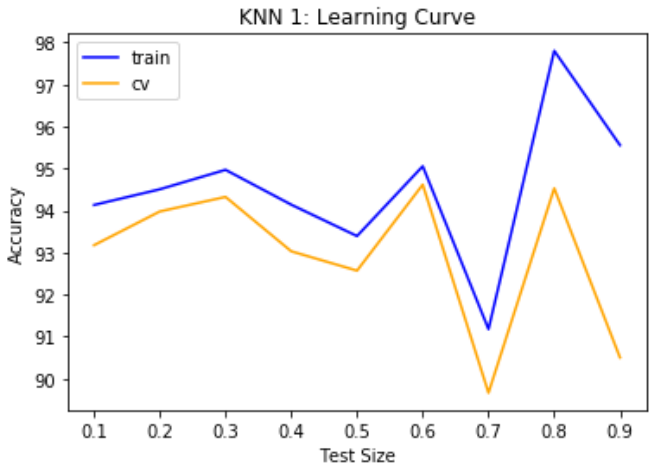


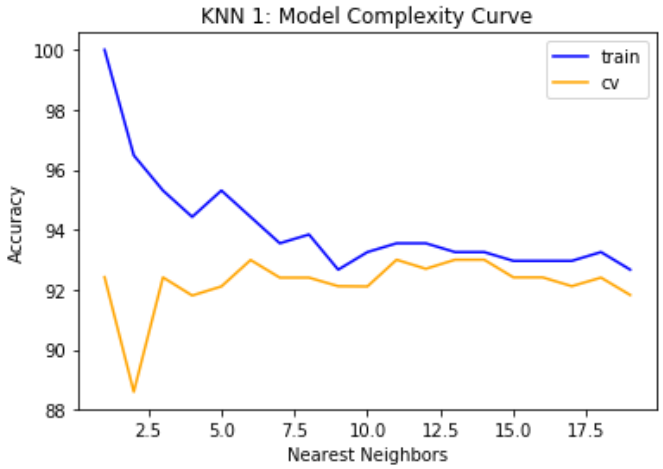
KNN:

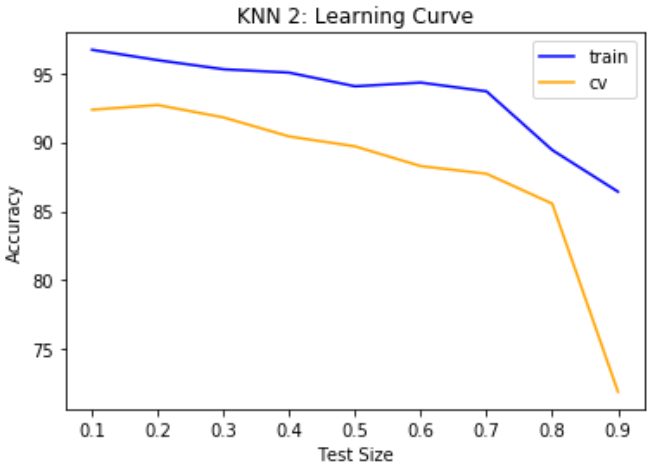
From the lectures, KNN seemed to be the most straightforward type of model. I was surprised to see it discussed with the other models as it seemed too simple to be true. It reminded me of how any person would go about analyzing data. Try and find similar historical examples and estimate based off of that. Therefore, I was interested in seeing how well this model worked in comparison with the others. Was it really just as good?

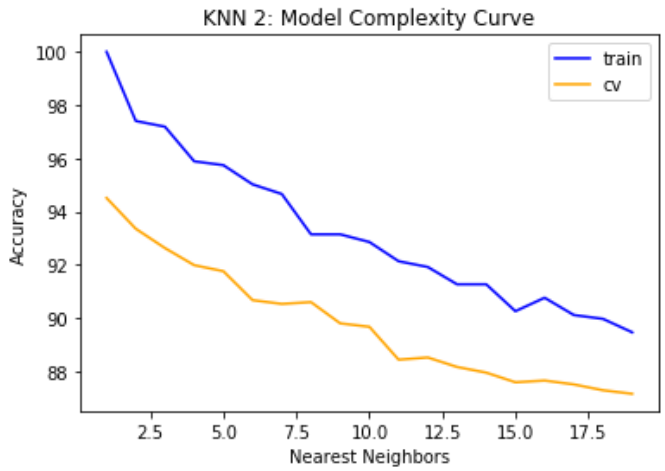
Parameters: In the KNN model the only parameter I tweaked and optimized was K (the number of neighbors). At K=1 the model overfits and after K=15 it seems the model generalizes a bit too much to make accurate predictions. The optimal number was somewhere between 5-15 and I assume that is probably the case for many models.

Analysis of algorithm: The KNN algorithm performed pretty well in comparison the other models. K is definitely a huge part of optimizing the algorithm and can vary from problem to problem. It must be understood and analyzed and optimized otherwise the algorithm will not produce good results. It also will not work well for problems where different classification outcomes share similar features/attributes. I think there needs to be a distance function to separate the values or some form of transformation like SVM to make the points distinct. I was expecting KNN to take long to query, since it just stores the data until querying time, but it ran very fast (as fast as decision trees) for me at about 1-2 seconds. Larger test sizes resulted in lower accuracy.









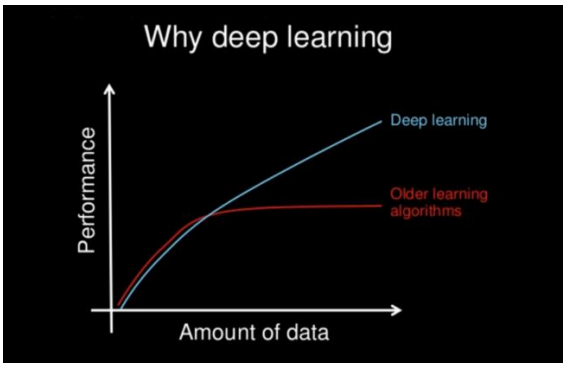
Neural Network:

The next algorithm I worked on was the neural network. It has been the algorithm I have heard most about in the news, but knew nothing about. I still don't understand it fully, but watching how it is able to learn and classify the MNIST data was amazing. I thought this algorithm would be difficult to implement and would perform the best.

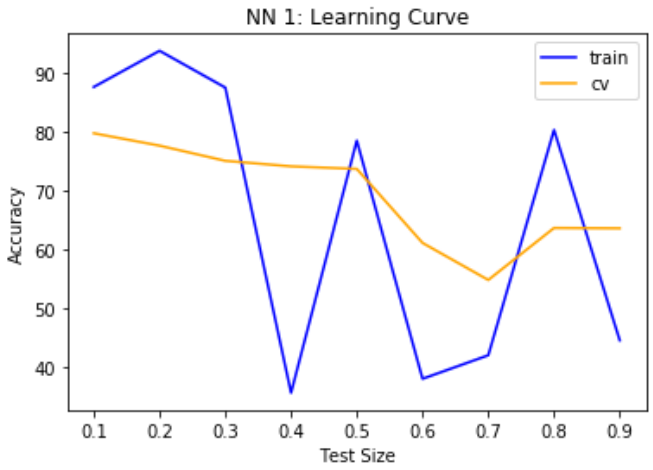
Parameters: The parameters that I modified and analyzed were the number of hidden layers, the number of units and the alpha (penalty) rate. Surprisingly, it was not difficult to implement the algorithm or tweak these parameters using scikit-learn. I thought figuring out the hidden layer and nodes would be a challenge, but they were just like the parameters in any other algorithm. For my classification problems 2 hidden layers seemed to be the optimal amount with a hundred or so units. Having a low alpha helped combat overfitting by constraining the size of the weights. Increasing alpha can fix high variance (overfitting) and lowering it can fix high bias (underfitting), as these change the complexity of the decision boundary.

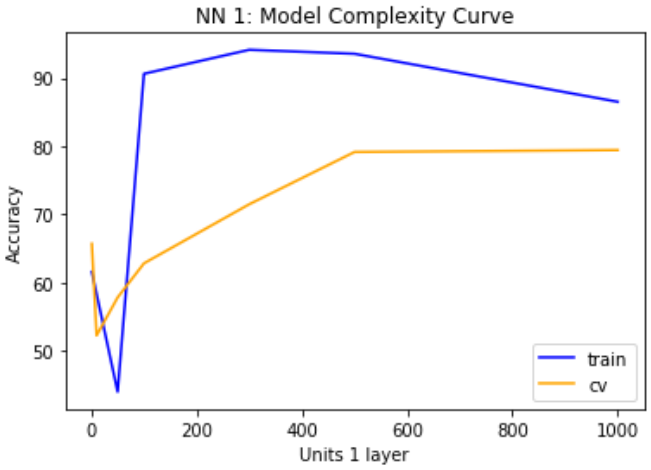
Analysis of algorithm: Although it's still a bit ambiguous exactly what is happening within each of the hidden layers and weights as the model is tweaked. It is pretty amazing to think that the algorithm still figures out how to interpret the data it receives and is able to work similar to how our brains work. Surprisingly, I didn't get the greatest accuracy results using this algorithm as I had originally thought. I thought I just wasn't tweaking the parameters and units and layers correctly, but I did. I think that is probably one of the challenges with this algorithm for many people is that since it's not easy to understand and interpret exactly what is going on under the hood, so it is very challenging to know what to change to get the optimal result. It is also hard to explain to others why the algorithm decided what it did.

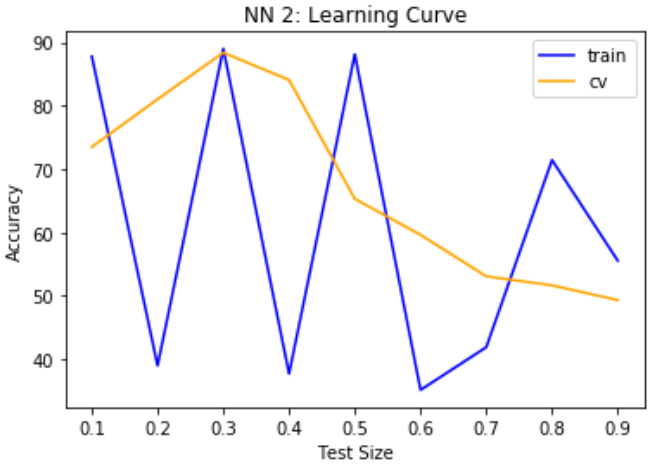
I think the reason that my results were not as good as I had thought is because I didn't have enough data. In researching I saw that this algorithms requires tons of data (thousands and thousands of observations). I found this illustration below useful. I think this can also be seen in the learning curve. There is a high variance in the results achieved as the test set size is modified unlike any of the other models.

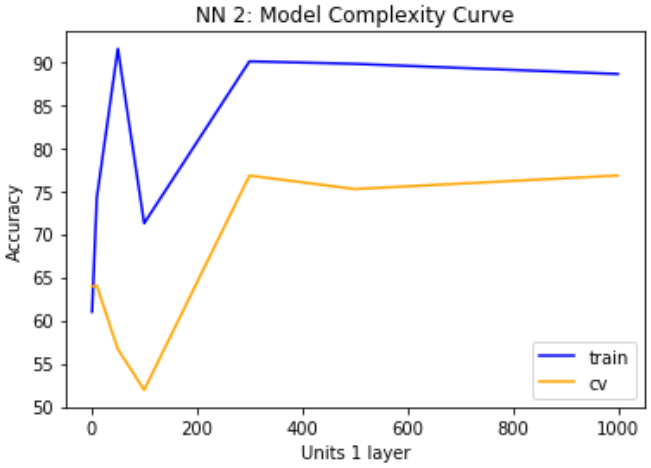


Source: https://towardsdatascience.com/hype-disadvantages-of-neural-networks-6af04904ba5b







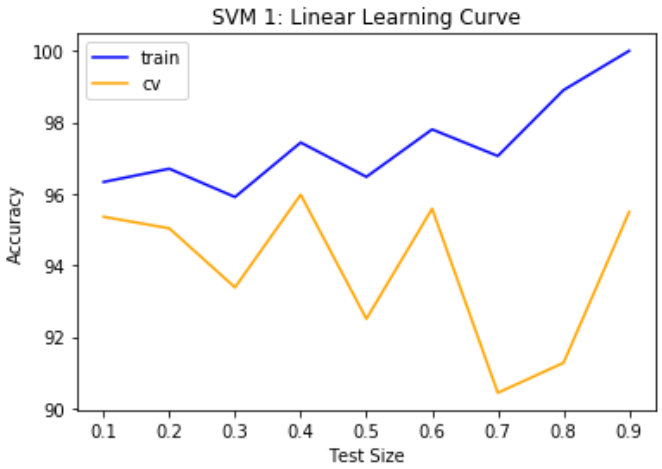


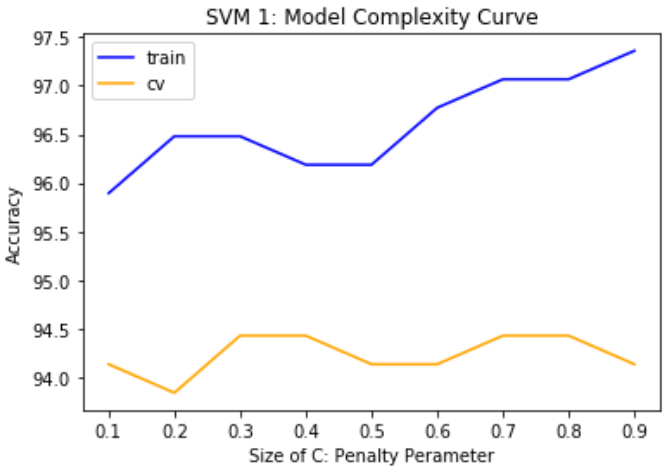
SVM:

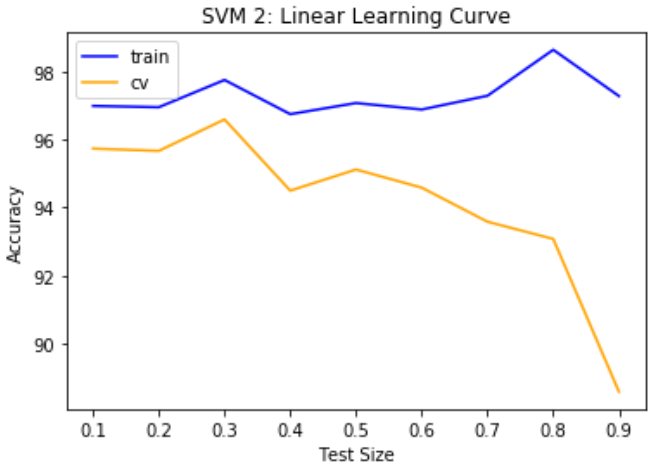
I had never even heard of SVM before taking this class. To me it felt like I was doing linear regression until I saw the example of how data was in circle and kernel trick transformed the data into a plane that could be linearly separable. It was pretty nifty so I was hoping to see that when learning this algorithm. Unfortunately, I didn't get to see that transformation as a lot of the work is done under the hood and you just change the parameter to change the Kernel.

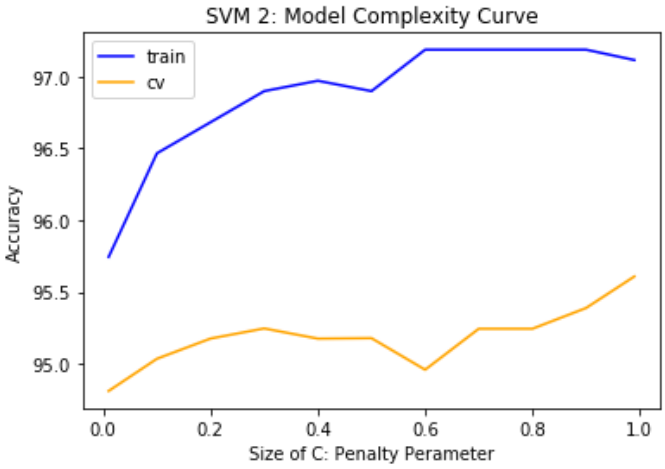
Parameters: The 2 parameters that I modified were the kernels themselves and value of C (penalty parameter of the error term). I kernels I tried were linear, polynomial (degree 8), Gaussian, and sigmoid. The linear and polynomial worked better with on these classification problems. Running this algorithm took the longest when iterating over it many times. The polynomial took the longest and once didn't even finish over several hours. Both linear and polynomial had really high accuracy around 95%. Similar to regression models it seems like this algorithm takes a while to train model, but then is quicker to query when the model is called. When analyzing multiple levels of C, smaller levels of c (up to about 0.6 had good results, but higher than that seem to overfit the data as training accuracy rose sharply while cv accuracy declined.

Analysis of algorithm: The ability for this algorithm and particularly the kernel trick to transform the data makes this a great algorithm that can be used in a broad array of datasets, whether the data is linearly separable or not. You can also expect that the results will be really accurate as it has the ability to find the global minimum instead of just local. The main issue I ran into is understanding the differences between the kernels and being able to pick the right C value. Therefore, you need to test all these out and pick the model that works best. Another problem was the amount of time it took to train the model and my datasets are not that big.









Overall Analysis / Comparison of Algorithms

Overall, for me the best algorithm was boosting with decision trees. It took a little more time than some of the other algorithms, but waiting 20 seconds wasn't a big deal in completing this assignment and boosting had the highest accuracy and over-fitting was controlled. There are lots of parameters that can be tuned to optimize the model. If you need to explain what features the model is likely using you can still perform the decision tree model and analyze which features are more important. I was expecting Neural Networks to outperform, but my datasets were not large enough. If I did the assignment again I would pick something large like MNIST data set and see how the algorithms compare. Boosting might start to take really long to converge on something like that and neural networks might go from last to first.

Timing Comparison:

On my classification problems, decision trees and KNN were very fast. They just took a second to run. Boosting and Neural Networks took a little longer taking about 20-30 seconds. This is actually expected since there are multiple learners in boosting. SVM took the longest. Polynomial 8 degrees when running iteratively witch different C values took several hours and never finished. If I just ran the query 1 time though, not iteratively, then it would take about 40-50 seconds.

Ease of Analysis/Interpretation:

Decision trees and KNN were the easiest to interpret and to be able to explain to others. You can see what features were split on in DTs and what neighbors were used in predicting via KNN. If you work in a position where you need to explain why a decision is being made these would be easier to use. If you don't really need to do that then I think boosting or SVM should be used. If you have a ton of data then try Neural Networks as well and see which has best accuracy.

Accuracy Comparison:

Below is the accuracy when tested the final trained models on a 20% holdout test set. I did not use this in training or in the cross validations above. Overall, Boosting and SVM had the highest accuracy rates and performed well on both classification problems. Decision trees did average on both datasets. KNN did better on the Image Segmentation, but not as well on the breast cancer dataset. Neural Networks had the lowest scores on both datasets. Which was surprising until I researched and found out you typically need a lot more data in order to see the superior performance of neural networks.

Test Set Holdout Accuracy

Classification	Null	DT	Boosting	KNN	SVM	NN
1 (Breast Cancer)	0.627	0.965	0.965	0.931	0.965	0.912
2 (Image Segment)	0.143	0.955	0.981	0.952	0.974	0.921