Nishant Roy
CS 4641 – Homework 1

Analysis of Supervised Learning Algorithms

Datasets Chosen

OptDigits: This dataset is comprised of 8x8 pixel matrices generated from normalized bitmaps of handwritten digits, where the goal is to predict the digit (0-9) represented.

The data is complete. There are 64 attributes (excluding class) ranging from 0-16 each, 10 possible values for the class (0-9) and 5620 total instances.

Training computers to identify handwritten figures is an interesting problem within the field of computer vision, because of the tremendous variations in handwriting, as well as how similar some digits look (8 looks much like a 6 or 9, 5 and 6 look similar, etc.), leading to noise in the data. Additionally, considering the number of attributes, there is a very small number of total instances, especially since there are 10 possible outcomes. The curse of dimensionality, therefore, means this dataset should be hard to get very accurate predictions for. The data is very evenly distributed, which should help fairly train the prediction models.

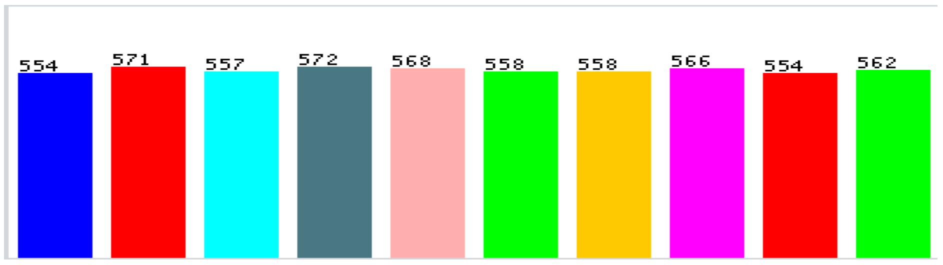


Figure 1: OptDigits data distribution. Starting from the right: 0, 1, 2, ..., 9, 10

Phishing Websites: This dataset is comprised of features that research shows to be useful in predicting phishing websites, such as domain length, number of redirects, presence of popup windows, etc., where the goal is to classify a website as suspicious or legitimate.

The data is complete, however, the authors note that it is hard to identify all possible features required for this classification problem because "there is no agreement in literature on the definitive features that characterize phishing webpages." There are 30 attributes (excluding class), 2 possible values for the class (legitimate or phishing), and 11055 total instances.

With hackers getting smarter by the day, and the use of the Internet getting increasingly prevalent, identifying phishing websites is an important and interesting problem. For a binary classification problem with 30 attributes,

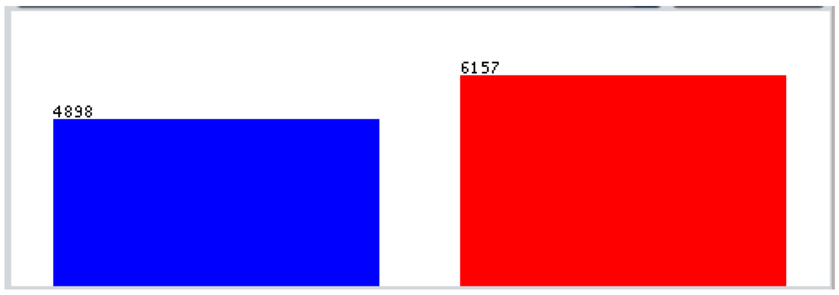
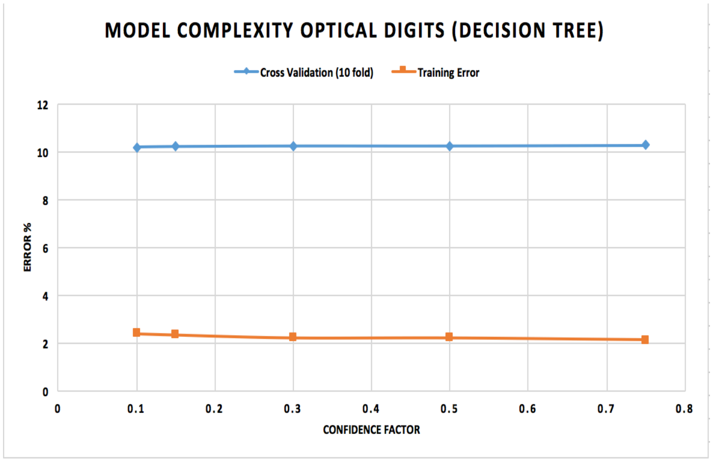


Figure 2: Phishing Websites data distribution. Legitimate on the left, Phishing on the right

the number of instances should be enough to lower the noise in the data, and ensure a high accuracy. The data is fairly even, with slightly more instances of phishing websites than legitimate ones, which should ensure a fairly trained prediction model.

Decision Trees



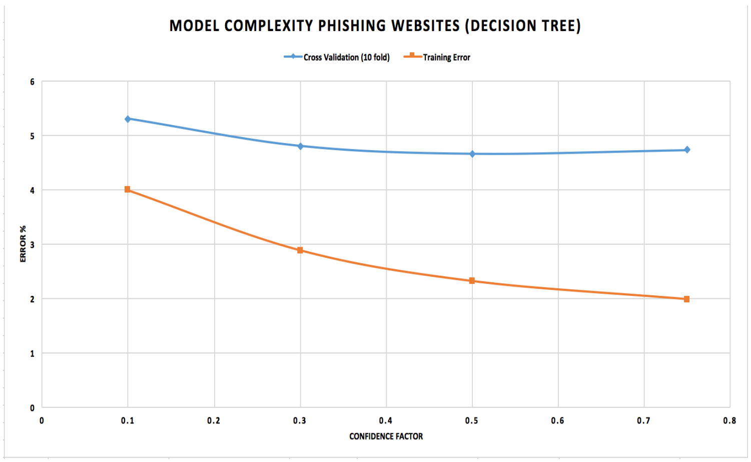


Figure 3: Model Complexity Curves for Decision Tree Algorithm (J48)

Looking at the Model Complexity (MC) curve for **OptDigits**, it is very hard to determine a distinction point however, as the confidence factor increases (pruning decreases), the gap seems to increase, which would indicate that more aggressive pruning would likely bring the curves closer to the trade-off point. If I could redo the experiments, I would try lower confidence factors with this dataset to test my hypothesis. The decision tree generated exhibits overfitting, with the large gap between the Cross-Validation and Training curves indicating low bias and high variance.

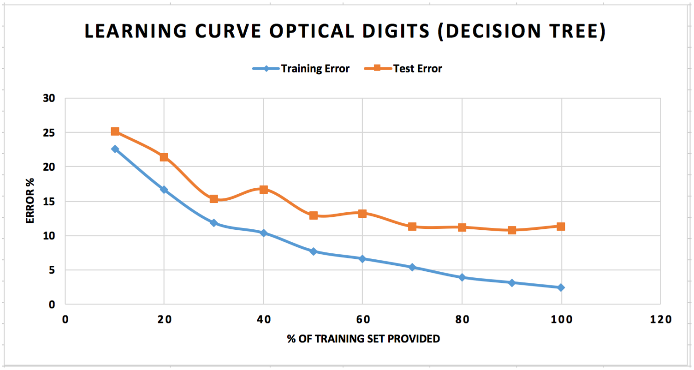
The MC curve for **Phishing Websites** starts to diverge somewhere between confidence factor values of 0.2 and 0.3, so the bias/variance trade-off point lies between those values, which could be determined with further experiments focused on those values. The model exhibits overfitting – the Cross-Validation and Training curves are diverging sizably – which suggests that the data has low bias and high variance. In both cases, the J48 algorithm generated over-fitted models, apparent from the training error decreasing as the confidence factor increases, and the cross-validation error increasing.

Pruned?	Confidence Factor	Leaves	Tree Size	Cross Validation (%)	Training Error (%)	Training time (s)
			Phishing Websites			
Yes	0.1	85	151	5.2985	3.9933	0.06
Yes	0.3	154	273	4.8074	2.8819	0.06
Yes	0.5	208	368	4.6653	2.3133	0.08
Yes	0.75	270	475	4.7299	1.9773	38.61
No	n.a.	292	516	4.6394	1.9902	0.04
			Optical Digits			
Yes	0.1	159	317	10.1932	2.3894	0.3
Yes	0.15	161	321	10.2186	2.3386	0.34
Yes	0.3	166	331	10.244	2.2115	0.22
Yes	0.5	166	331	10.244	2.2115	0.01
Yes	0.75	173	345	10.2694	2.1352	23.5
No	n.a.	174	347	10.422	2.1352	0.21

Figure 4: Data for Decision Tree Algorithm (J48)

From the data table, we can see that the tree for the **OptDigits** dataset did not get pruned very much; the tree remains shallow and wide (more than the half the tree is leaves). This corroborates my hypothesis that more aggressive pruning would likely help build a more accurate model for this dataset. For **Phishing Websites**, most of the tree is still leaves, because there are so many combinations of features to classify a website as suspicious.

However, the tree size drops considerably as pruning increases, and the cross-validation error is much lower for this dataset than **OptDigits**, suggesting that it generalized the training data better. This dataset shows strange behavior though, with the Cross-Validation error *rising* as pruning increases. The decision tree algorithm is much more accurate for the **Phishing Websites** dataset (> 90%), likely because of the large number of instances, allowing it to train better. The **OptDigits** dataset is quite noisy, with a *lot* of attributes and possible outcomes, and not enough instances to classify them accurately.



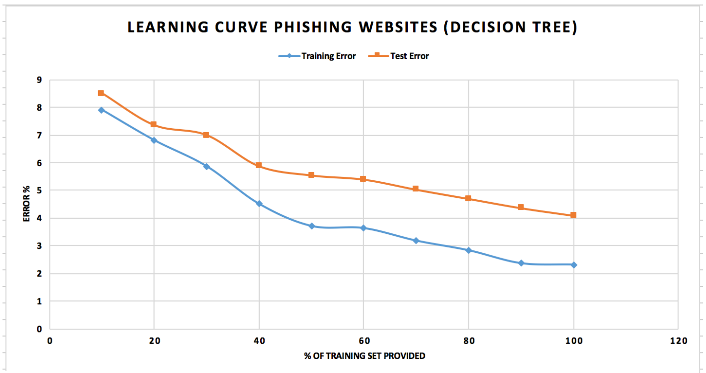
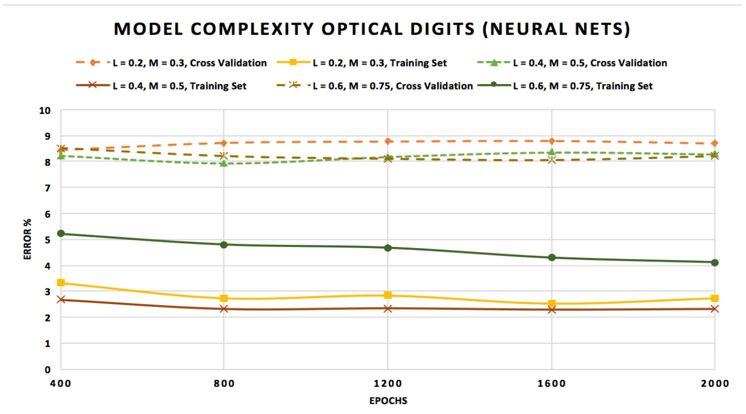


Figure 5: Learning Curves for Decision Tree Algorithm (J48)

For **OptDigits**, I used a confidence factor of 0.1, since that provided the lower cross-validation error. As more training data is provided, the test-error rate drops initially, and eventually plateaus, suggesting that our model has some bias, which cannot be overcome just by obtaining more training data. For **Phishing Websites**, I used a confidence factor of 0.25, since that is the trade-off point observed from the MC curve. The test-error continues to drop as we provide more training data, therefore, our decision tree exhibits more variance than bias, so a larger training set would help obtain a more accurate prediction.

Neural Nets



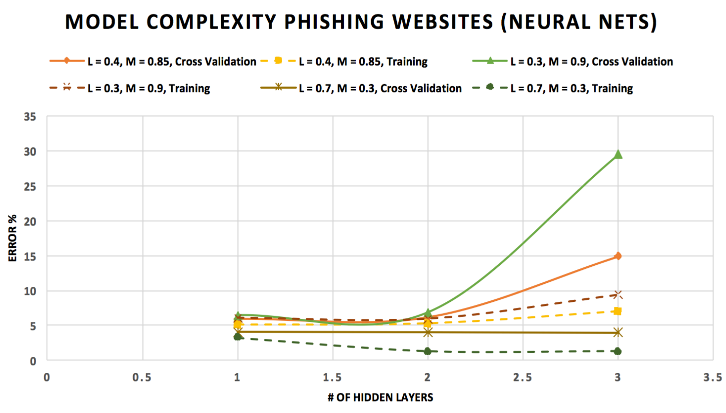


Figure 6: Model Complexity Curves for Neural Networks

For the **OptDigits** dataset, I varied the training time with different learning rates and momentums to find the best model, while for **Phishing Websites**, I varied the number of hidden layers.

Increasing the number of epochs did not lower the cross-validation or even the training error much. Lowering learning rate reduced the error, indicating that the model needed to be fine-tuned – smaller corrections were required. This may be explained by the curse of dimensionality. There are 64 attributes and only ~550 instances for each possible value of class, so there is insufficient training data for the network to generalize and improve its performance. The curves do not diverge much, so I did not find the best trade-off point for the model.

Nishant Roy

CS 4641 – Homework 1

Neural nets are able to emulate any function and hence tend to produce over-fitted models. This is apparent in the MC curve for **Phishing Websites**; we see the cross-validation error spike dramatically as the number of hidden layers rises. A high momentum had a very adverse effect on the accuracy of the model. Momentum is used to keep the neural net from settling in local minima, but also to ensure it does not incorrectly identify the absolute minimum as a local minimum. As exhibited by the MC curve, a very high momentum increases the error, likely due to the model not settling in the absolute minimum. The combination of a high learning rate and low momentum produced the best model, enabling the algorithm to make large changes to the bias and edge weights, while not rolling out of minima very easily. This suggests that there weren't very many local minima, or the model may have incorrectly settled, and the bias must be low, since changing it in large increments did not affect results.

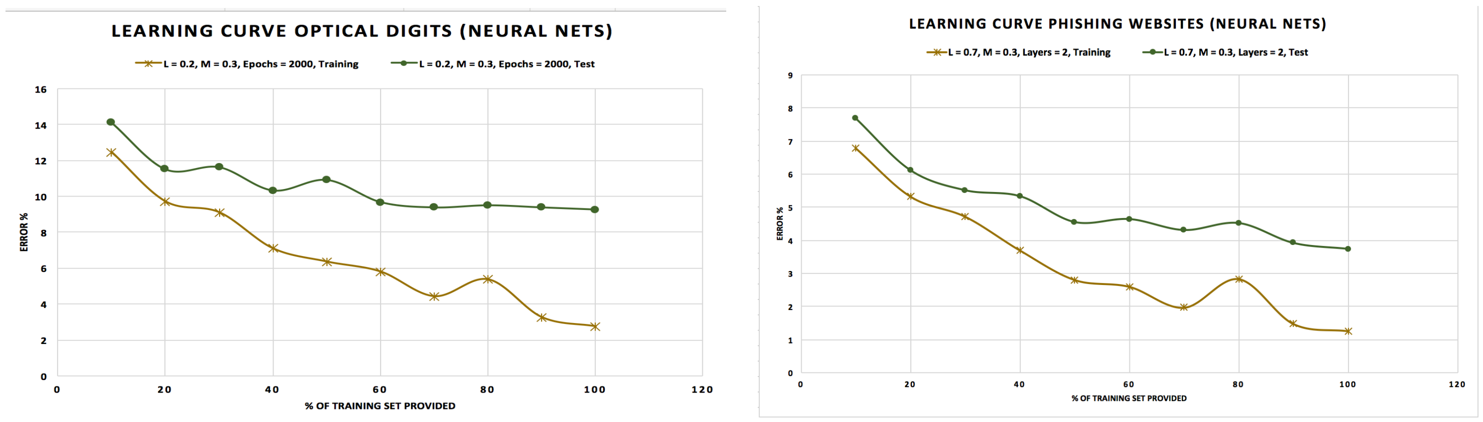


Figure 7: Learning Curves for Neural Networks

For **OptDigits**, the error eventually levels out despite increasing the size of the training set, indicating that there is some inherent bias our model cannot overcome even with more training data. One potential solution is to increase the momentum so that our neural net is not stuck in a local minimum. Another could be to increase the learning rate to see if a larger rate of change to weights and bias helps fit the function better. Finally, it might be better to increase the number of nodes per layer for **OptDigits** to better model the function comprising of 64 attributes. For **Phishing Websites**, the error decreases steadily as we provided more training data so there is likely high variance in the data, and if we can gather sufficient training data, we can improve the accuracy of our model. I also used a higher number of nodes per layer for this dataset, which would have allowed it to properly model the 30 attributes, therefore providing a higher accuracy.

Neural Nets (a nodes/layer, # epochs = 500)

Learning Rate	Momentum	# hidden layers	Cross Validation (10 fold)	Training Error	Training time (s)
0.4	0.85	1	5.9964	5.0659	76.57
0.4	0.85	2	6.2161	5.2457	148.62
0.4	0.85	3	14.8746	7.0432	277.04
0.3	0.9	1	6.4228	6.061	147.93
0.3	0.9	2	6.8364	5.9188	351.89
0.3	0.9	3	29.4908	9.421	639.62
0.7	0.3	1	4.045	3.2308	208.04
0.7	0.3	2	4.0062	1.2536	555.79
0.7	0.3	3	3.9674	1.2794	764.7

Figure 8: Data for Neural Networks (Phishing Websites)

Nishant Roy
CS 4641 – Homework 1

Neural Nets (5 nodes/layer, 1 layer)

Learning Rate	Momentum	# Epochs	Cross Validation (10 fold)	Training Error	Training time (s)
0.2	0.3	400	8.4392	3.3299	19.52
0.2	0.3	800	8.7189	2.7453	39.23
0.2	0.3	1200	8.7697	2.847	75.27
0.2	0.3	1600	8.7951	2.5419	80.33
0.2	0.3	2000	8.6934	2.7453	102.84
0.4	0.5	400	8.2359	2.669	10.82
0.4	0.5	800	7.9309	2.3132	30.77
0.4	0.5	1200	8.1851	2.3386	70.17
0.4	0.5	1600	8.363	2.2877	45.57
0.4	0.5	2000	8.2867	2.3132	115.05
0.6	0.75	400	8.5155	5.211	22.85
0.6	0.75	800	8.2105	4.8043	35.08
0.6	0.75	1200	8.1088	4.6772	55.27
0.6	0.75	1600	8.058	4.2959	75.75
0.6	0.75	2000	8.2105	4.1179	101.15

Figure 9: Data for Neural Networks (OptDigits)

Neural networks are an example of an eager learner – one that attempts to build a model to generalize the training data before any queries are received. Comparatively, lazy learners store the training data they are given and wait till a test query is received. Eager learners tend to take much longer to build their classification model as compared to lazy learners. They decide on a certain hypothesis from the complete hypothesis space and stick to it, using it to classify future test queries. This was apparent in comparing the training times of Neural Nets to k-Nearest Neighbors (example of lazy learner). k-NN had a training time of 0 seconds, since it simply stores all the training data, while Neural Nets took as long as 10 minutes to build their model in some cases. However, k-NN may take longer to test (as long as 6 seconds), while Neural Nets ran through test cases almost instantaneously.

Boosting (Adaboost - J48 Decision Trees)

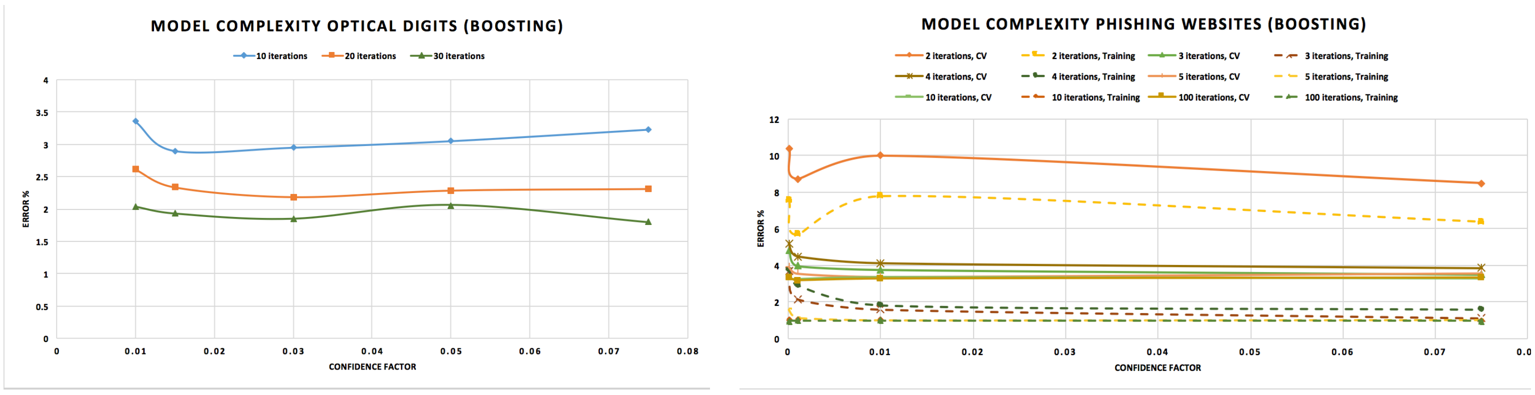
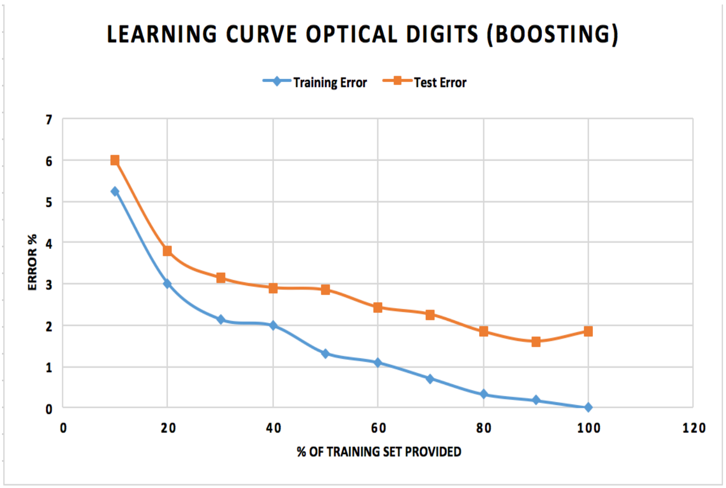


Figure 10: Model Complexity Curves for Boosting Algorithm (Adaboost – J48)

The training error obtained when boosting the decision tree for **OptDigits** was 0, so it isn't shown in the graph. Very aggressive pruning (10x as aggressive as before) was used while boosting, which brought the error rates down tremendously. Without boosting, J48 gave around 80% accuracy, and even with just 10 iterations, the accuracy shot up to over 95%. As the number of iterations rose, the accuracy did as well, which is expected, since we continue to retrain on the "bad" (wrongly classified) examples at the end of each iteration. The confidence factor did not impact our accuracy too much, likely because they were all extremely low (largest CF used was 0.075).

Nishant Roy
CS 4641 – Homework 1
The training error shot down rapidly as the number of iterations rose for **Phishing Websites**, indicating that we were fitting the model more heavily to the provided data. The testing error also showed a downward trend as we boosted the model more. For fewer iterations, less pruning provided better accuracy, probably because very aggressive pruning was causing too *much* generalization and disregarding too many attributes, which was corrected when more iterations were performed. The MC curve shows that after a point, increasing the number of iterations had a very marginal effect on the error. The difference in errors between 4 iterations and 100 iterations was negligible (< 3%), for a very large jump in training time (~0.3 seconds vs. ~6 seconds). This is an important tradeoff to consider as the size of our dataset increases and the number of attributes rise.



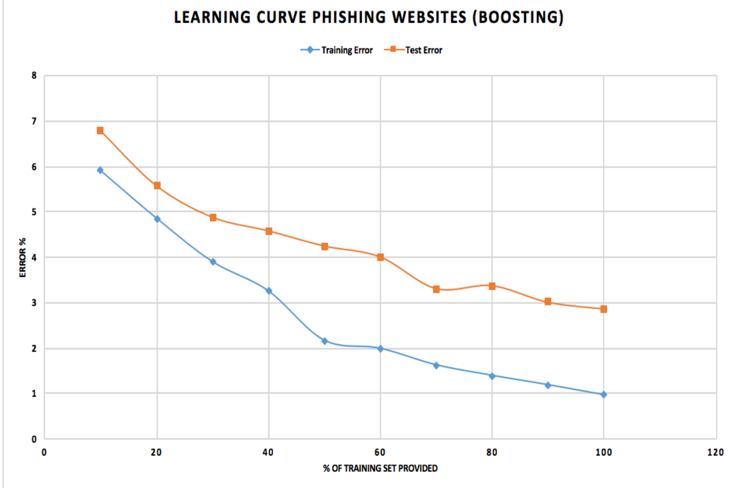


Figure 11: Learning Curves for Boosting (Adaboost - J48)

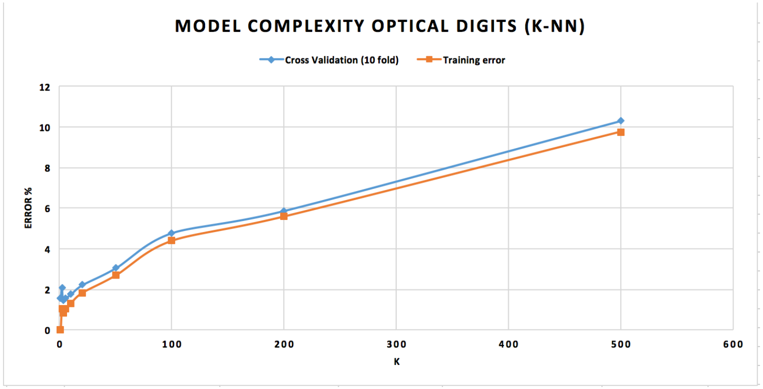
For **OptDigits**, I used 45 iterations with a CF of 0.075, and for **Phishing Websites**, I used 100 iterations with a CF of 0.001. In retrospect, it would have been better to use a lower number of iterations for Phishing Websites, because of the principle of Occam's Razor – picking the less complex assumption. As mentioned above, going from 4 iterations to 100 iterations did not cause a very large drop in error rate, but it did cause a significant rise in training time. If I had followed Occam's Razor, I would have gotten a much simpler model that predicted future outcomes almost as well as the more complex model.

Phishing Websites had more instances, which allowed for more aggressive pruning than **OptDigits**, because over-generalization was eventually corrected as the model retrained on the bad examples several times. With **OptDigits**, pruning too aggressively increased the Cross-Validation error because there simply wasn't enough data to overcome the over-generalization.

For both datasets, the error rates continue to drop as we provide more training data, more so for **Phishing Websites** than for **OptDigits**. Our boosted decision trees exhibit high variance, which we could overcome by gathering more training data. After about 30% of the training set is provided, the test and training error curves start to diverge more, indicating that we start to overfit around this point. The learning curve for **OptDigits** showed a similar pattern for J48 without boosting, leveling out after a certain % of training data is provided. However, the learning curve with boosting shows less plateauing and at a lower level (~2% vs. ~10%), showing that the increased iterations allowed the Decision Tree algorithm to generalize the model more, reducing the effect of the bias.

Nishant Roy
CS 4641 – Homework 1

k-Nearest Neighbors



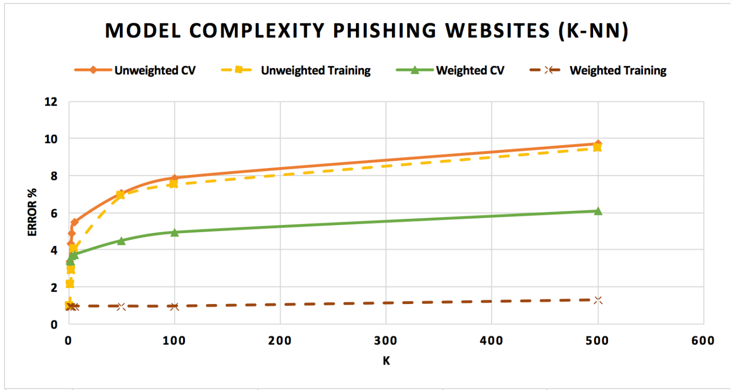


Figure 12: Model Complexity Curves for k-Nearest Neighbors Algorithm

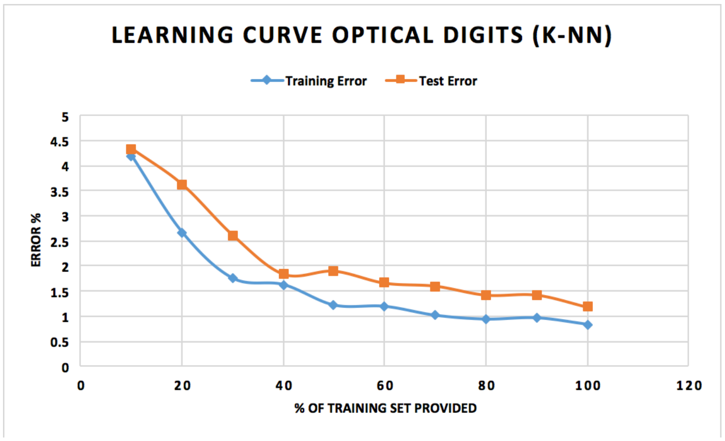
For **OptDigits**, I used the unweighted average, while for **Phishing Websites**, I used unweighted as well as a weight of $\frac{1}{distance}$. For both datasets, as *k* rises, both Cross-Validation error and Training error rise steadily. This indicates that there is variance in the datasets, because taking more neighbors into account should give a better prediction. There are two explanations for this. Firstly, the curse of dimensionality plays a huge role. We have a very large number of attributes in both test cases (60 for **OptDigits**, 30 for **Phishing Websites**). Secondly, for **OptDigits**, we treat all attributes as equally important (weight = 1), when a certain set of attributes (group of pixels in this case) may be particularly indicative of what the digit is. Furthermore, for **OptDigits**, we have a significantly higher number of possible classifications (10). Therefore, using the nearest-neighbors algorithm to make a prediction becomes increasingly less accurate, as our total sample space for all possible points of data rises exponentially.

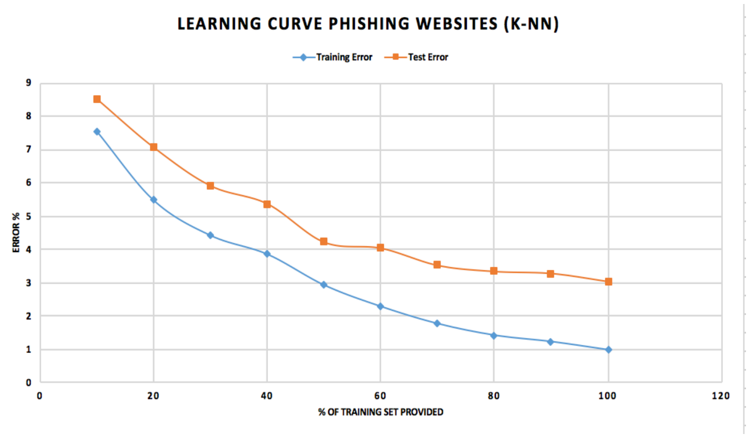
k-NN			
k	**Weight**	**Cross Validation (10 fold)**	**Training error**
Phishing Websites			
1	1/distance	3.4376	0.9692
2	1/distance	3.373	0.9692
3	1/distance	3.6831	0.9692
5	1/distance	3.7477	0.9692
50	1/distance	4.4973	0.9692
100	1/distance	4.9367	0.9692
500	1/distance	6.0739	1.2923
1	1	3.373	0.9692
2	1	4.3293	2.1323
3	1	4.8721	2.869
5	1	5.4665	4.0062
50	1	7.0302	6.901
100	1	7.8573	7.5213
500	1	9.6924	9.4727
OptDigits			
1	1	1.5506	0
2	1	2.0844	1.0168
3	1	1.4743	0.8388
5	1	1.576	1.0422
10	1	1.7539	1.3218
20	1	2.2115	1.8302
50	1	3.0503	2.6945
100	1	4.7789	4.3976
200	1	5.8719	5.5923
500	1	10.2949	9.7611

Figure 13: Data for k-Nearest Neighbors

Using the weighted average for **Phishing Websites** resulted in a notably lower test error, and almost 0 training error. With the weighted average, the test and training error diverge. This indicates that the trade-off point for the trade-off point is closer to 1 than the larger values used in the experiments. Till the value of *k* rises tremendously, the training error remains constant, and then rises, indicating that after a certain point, we start to under-fit.

The k-Nearest Neighbors algorithm is excellent at tuning out noise in data, because depending on the value of *k*, it doesn't take into account all the data to make a prediction. When we take weighted averages rather than valuing everything equally, this is very apparent. By taking the inverse of the distance, we tune out points that lie very far out of the instance space we are considering.



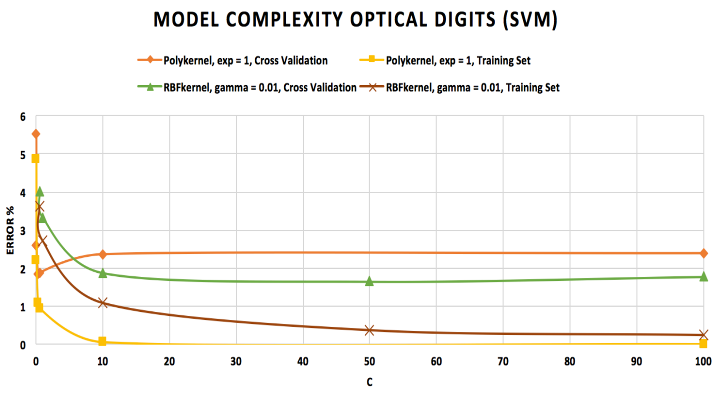


For both datasets, the learning curve tends to 0 as we provide more training data. This is expected, since k-NN is a lazy learner, using the training data it saved to estimate a value for every test query. Surprisingly, the training error did not reach absolute 0 for **OptDigits** even when the entire training set was provided. One reason for this may be that I did not use the weighted average for OptDigits, and a value of 3 for *k*, so it did not match up exactly. The training error for **Phishing Websites** on the other hand did reach 0, likely because the weighted average led it to tune out noisy data. I also used a value of 2 for *k*, which meant that it was very focused on the points that were absolutely next to it. This would cause overfitting, however, the test error also decreased, indicating that perhaps the testing data is similar to the training data, and hence, the model was extremely accurate. After about 60% of the training set is provided in Phishing Websites, however, the curves start to diverge a little, indicating high variance in our data.

k-NN is an example of a lazy learner, i.e., it stores all the training data and waits until a test query is given. By using a number of linear functions to form a hypothesis rather than settling on a single hypothesis for the entire instance space like eager learners do, they can sometimes be more accurate. While the training time is 0, the space complexity is notably higher, since it must store all the training data. Additionally, the testing time is higher than eager learners (like Neural Nets), because it waits till the test queries are provided, then with the help of local linear functions, builds a global approximation to make its prediction. So, there is a clear trade-off between space and time between lazy and eager learners. k-NN may be a more acceptable algorithm for a smaller dataset with more noise, while neural networks may be better suited to large datasets with less noise that require rapid testing.

Nishant Roy
CS 4641 – Homework 1

Support Vector Machines



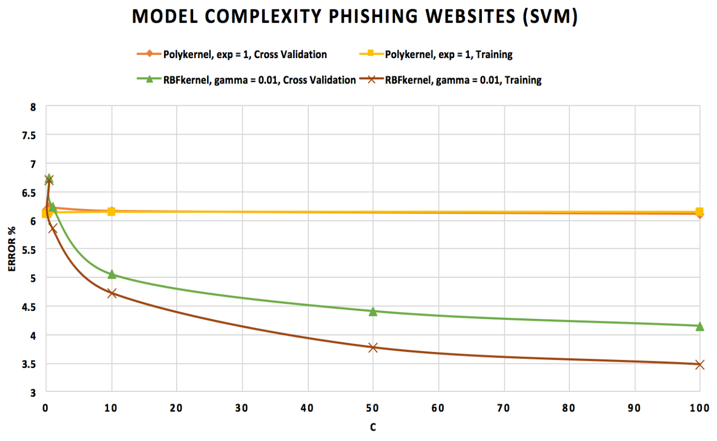


Figure 14: Model Complexity Curves for Support Vector Machine

The Support Vector Machine algorithm shows one of the best accuracies for the **OptDigits** dataset, which is expected, since SVMs are known to be excellent at recognizing handwritten characters. This is because SVMs belong to the class of algorithms known as kernel methods, which are used for pattern analysis. I used two kernels for my experiments – Polykernel (which attempts to find polynomial patterns of the given degree in the data) and RBFkernel (which uses an approximation of the Gaussian function to find patterns in the data).

I was interested in studying the impact that the soft margin parameter, *C*, has on the performance of the SVM, so I kept the degree/gamma values for the kernels constant, and changed the value of *C*. *C* is used to determine the extent to which you want to avoid incorrect classifications (high *C* means you are more concerned about misclassifying data). Even though SVMs look for the hyperplane with the largest margin, a large value for C tells the algorithm to choose a smaller margin hyperplane if it provides more accurate classification. Naturally, this should display longer training times.

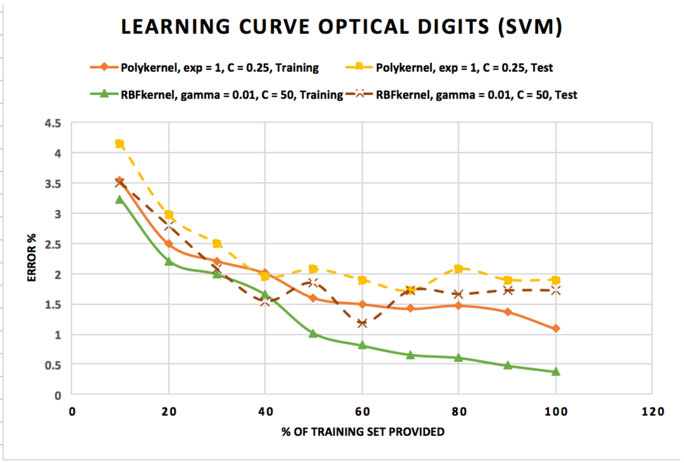
SVM					
Kernel	**Exponent/Gamma**	**C**	**Cross-Validation**	**Training Error**	**Training Time**
		Phishing Websites			
PolyKernel	1	0.005	7.0949	6.9398	0.22
PolyKernel	1	0.05	6.1773	6.0998	0.37
PolyKernel	1	0.25	6.1644	6.1127	2.37
PolyKernel	1	0.5	6.229	6.1385	4.31
PolyKernel	1	10	6.1644	6.1515	63.83
PolyKernel	1	100	6.1127	6.1515	546.31
RBFKernel	0.01	0.5	6.7459	6.6942	35.53
RBFKernel	0.01	1	6.229	5.8671	43.46
RBFKernel	0.01	10	5.053	4.7299	16.09
RBFKernel	0.01	50	4.4068	3.7736	21.42
RBFKernel	0.01	100	4.1484	3.4764	65.81
		OptDigits			
PolyKernel	1	0.005	5.5414	4.8551	0.35
PolyKernel	1	0.05	2.5928	2.2115	0.31
PolyKernel	1	0.25	1.8556	1.093	0.37
PolyKernel	1	0.5	1.881	0.9405	0.4
PolyKernel	1	10	2.364	0.0508	0.49
PolyKernel	1	100	2.3894	0	0.52
RBFKernel	0.01	0.5	4.0163	3.635	4.02
RBFKernel	0.01	1	3.3299	2.7199	3.19
RBFKernel	0.01	10	1.881	1.093	1.19
RBFKernel	0.01	50	1.6523	0.3813	1.6
RBFKernel	0.01	100	1.7794	0.2542	1.43

Figure 15: Data for Support Vector Machine

The training times did go up as expected as *C* increased, except when using the RBFkernel for **OptDigits**. This may be due to the fact that this dataset might have a Gaussian function that approximates it well, allowing for quick training, testing and model selection. However, the Polykernel also performed reasonably well with

Nishant Roy
CS 4641 – Homework 1

OptDigits. The time increased, but the error rates on the training set, in particular, tends to 0, so, perhaps there is also a linear function that we can use to build a model, even though this may lead to over-fitting. In most cases, increasing *C* also reduced error, because we search more carefully for a model that fits well. However, this is an example of over-fitting, as apparent from the data that shows (in most cases), that error first falls and then rises again, after we are trusting our training data too much.



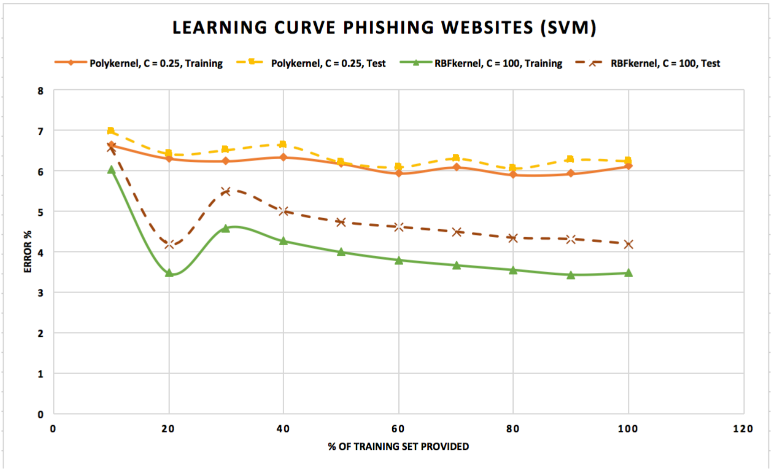


Figure 16: Learning Curves for Support Vector Machines

For **OptDigits**, the RBFkernel performed better on both training and test sets, but only marginally on the test set. When 60% of data was provided, the RBFkernel provided the least error on the test set. The lowest error for the Polykernel came with 70% of training data. This indicates that after about 60-70% training data is provided, we begin to overfit, and the curves begin to diverge, with the training error continuing to fall, while the test error rises. Even though SVMs are known to be less susceptible to overfitting, it is possible. The SVM algorithm was incredibly accurate on this dataset, with more than ~95% accuracy.

For **Phishing Websites**, the RBFkernel performed better on both datasets by a notable margin. Interestingly, it also took significantly less time to find a model than Polykernel with a high soft margin (~1 minute vs ~10 minutes). No matter how much training data was provided, the Polykernel was only able to marginally improve its accuracy. This means that the model built with the Polykernel has some inherent bias and/or irreducible error that cannot be rectified by adding more training data.

Although the accuracy of SVM improved as the value of the soft margin increased, it is a trade-off between marginal accuracy gains and sizable training time increases. By the principle of Occam's Razor, therefore, it is better to pick the lower value of *C*, which, if I were to run the experiments again, I would do for RBFkernel, since I chose *C* = 100, the largest value I tested.

In comparison, the RBFkernel improved dramatically going from 10% to 20%, then rose, and then continued to fall. This seems like a false minimum, and perhaps adding more training data would improve the accuracy of the RBFkernel with the **Phishing Websites** dataset. The only way to test this hypothesis is to gather more data about features that indicate suspicious websites.