HW1: Supervised Learning

Name: Jiun-Yu Lee GTID: 903435223

Environment

In this assignment, I used python 3.7.2 with scikit-learn and matplotlib as my coding environment. To reproduce the result, a proper version of python and a least the two packages listed above is required. Please see README.txt file in the submission to get detail information of how to run my code.

Algorithm

- **Decision Tree:** *DecisionTreeClassifier* module in sklearn is used to be my implementation of the decision tree algorithm. It supports multiple pruning methods such as minimum number of samples required at a leaf node or the maximum depth of the tree. Here, the **maximum depth of the tree** is chosen to be my pruning method. It also supports two criterions for splitting the node, gini index, and entropy. We will search both of the methods to see which one is best suitable for the classification problem.
- **Support Vector Machine:** *SVC* module in sklearn is used to train my support vector machine model. It supports various predefined kernel, such as *linear, polynomial, sigmoid, RBF*, etc. To be specific, ***linear, polynomial*, and *RBF*** kernel as well as other parameters such as ***gamma, degree*** of the polynomial kernel, and regularization parameter ***C*** will be explored in my experiment.
- **K-nearest Neighbors:** I used *KNeighborsClassifier* module to do my experiment in this assignment. Different k value will be evaluated, as well as different distance metric (manhattan_distance (**l1**) and euclidean_distance (**l2**), specifically.) Also, in addition to treated all k-nearest neighbors equally important, I try to weight the vote by the inverse of the distance of a node so that a node will contribute more if it is closer to the test case.
- **Boosting:** *AdaBoostClassifier* module is used to implement my boosting version of decision tree. To be able to see the influence of boosting without interfering by the factor of decision tree, and reduce overfit bringing by a not weak enough classifier, I fixed the max depth of the decision tree to one (that is, **decision stump**.) Other parameters like the **number of estimators** and the **learning rate** will be explored to find the best performance for the following classification problem.
- **Neural Network:** Many different structure and framework of neural network are emerging in deep learning field, such as CNN, RNN, GAN, and so on. However, in order to make a fair comparison between Neural Network and the other four algorithms in the above (e.g using CNN on images or RNN on time series data is very much likely to perform better than the other four algorithms, which we already apply more domain knowledge on this particular algorithm), I only explored fully-connected network in this assignment. *MLPClassifier* module is used in our experiment to build the

network. Here, I will explore different **network structure** (various combination layers) **activation functions** (*relu, tanh, logistic*), and **learning rate**. The solver is fixed to Adam optimizer since it is can converge more quickly than others (such as SGD).

Hyper-parameter Optimization:

Before applying each algorithm to classification problems, I will first do some hyper-parameter optimization for those algorithms on each dataset to gain a reasonable performance. My optimization algorithm is mainly based on **grid search**, which I used *GridSearchCV* module in sklearn to achieve it. The parameter search space for each algorithm is predefined, and the GridSearchCV will exhaustedly search through all the parameter combination in the search space, perform cross-validation, and return the best parameter set. For the cross validation, I used *StratifiedKFold* here to ensure the proportion of the classes in each fold is equal to the proportion in the entire dataset (since I actually have an imbalanced data set in the following classification problem) and divided the training data into 5 folds. After doing the grid search through the default parameter space. I will see the result (learning curve, final performance) and fine-tune it manually if necessary. The detail information about how the parameter space is defined can be seen in src/searchParameter.py

Problem 1: Credit Card Fraud Detection

For the first classification problem, I want to analyze a very common problem that may appear in the real-world dataset, imbalanced dataset. I choose the credit card fraud detection dataset on kaggle to analyze since it has about 285K samples but only about 500 of them are label as a fraud. This problem itself is a very interest because we can actual deal with the real world data and predict the potential crime that may happen. Moreover, Imbalanced dataset can happen in not only credit card fraud but also many other classification problem like cancer detection. Compare to balanced dataset, We will care much less about accuracy (we can simply classify all sample as negetive to achieve a high accuracy, but this is meaningless) but much more about the recall (if there is a credit card fraud or cancer, we don't want our classifier to miss it.) Therefore, I want to see how each of the five algorithm will perform on this kind of imbalanced dataset. My expectation is that AdaBoost may have a better result in this case since it has the ability to get the hard sample right by changing their weights, yet SVM or neural network may just treated lots of the positive as noise.

Data preprocessing:

The original dataset have a size of (285k, 30), in which it feature space is already reduced using PCA. Only around 500 of the samples is positive (0.17%), which is a very hard imbalanced data. Apply algorithm directly on this dataset is not likely to perform well anyway. In order to deal with this problem, I create another dataset but keeping all the positive data and randomly sample negative data from the original dataset until it have 5000 samples. With the new (5000, 30) dataset, the positive class proportion is now 10%, still an Imbalanced dataset, but more easily to deal with.

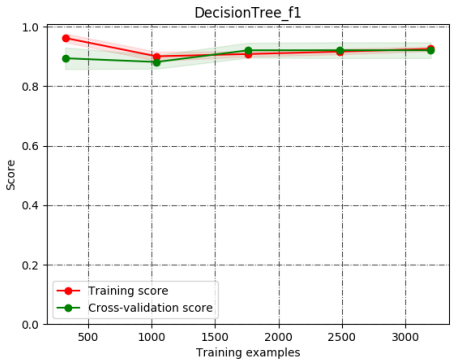
This dataset is then separated to training and testing data with 4:1 ratio. The training data is used to train and validate and the testing data is used only in the final test.

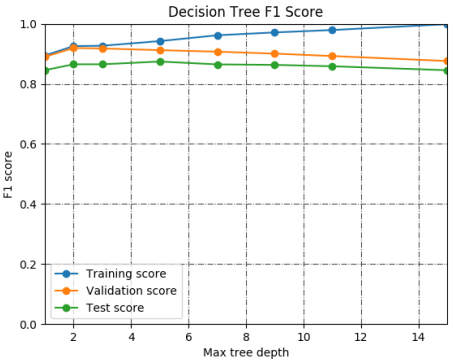
Metrics:

In this problem, I use F1 score to be the main metric to measure the performance of the model. As I mention above, the recall of the results is the most important thing I should consider in this kind of problem since I don't want my problem to misclassify any thing that may be positive. However, a model with a high false positive rate is not a good model, either. In other words, I still want to consider the precision. Therefore, F-1 score become the ideal metric to measure the performance, since it is a harmonic average between precision and recall, which can reflect both of them.

Result & Analysis:

- **Decision Tree:**

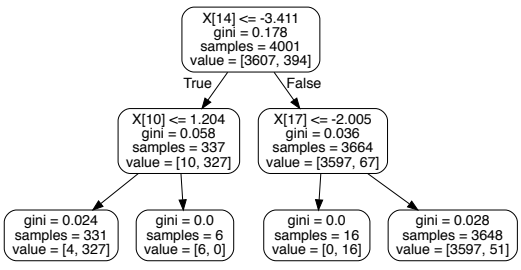




best parameter - criterion: "gini",
max_depth": 2

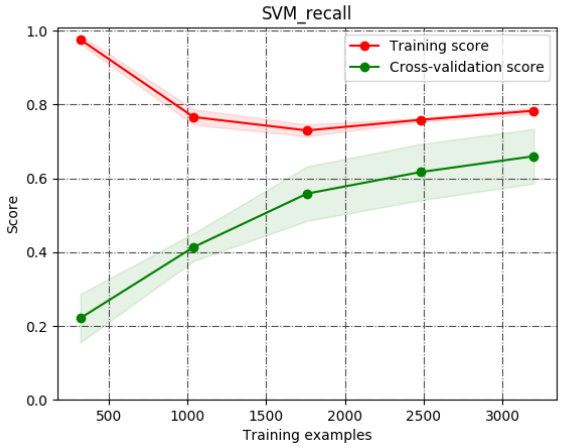
As we can see in the learning curve above, decision tree works relatively well in this dataset. The f1 score of both training and validation is above 0.9, and there are barely any overfitting happening is this setting. I am surprised that a tree with max depth of only 2 achieves the best performance. That means the tree will only have a maximize of three splits, which should normally have high bias and should not do well. However, if you see the rightmost graph which shows the relationship between max_depth and performance, we can found out that overfitting does happening when the max_depth is 3 or above.

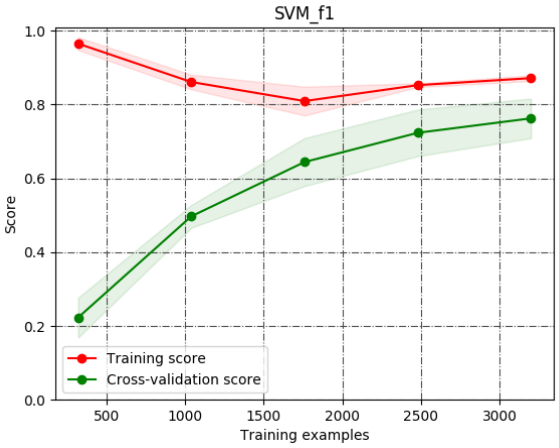
To dig deeper, I visualize the tree with max_depth = 2 as the right figure. It shows that the first split is able to separate lots of positive samples out, and we are able to get 87% of positive samples right after second level splits. However, when I increase the max_depth, what the tree does is just split one to two positive samples out from a node at a time, which is definitely not going to generalize well.



Some ensemble methods like Adaboost or random forest could help to tackle this problem. Since random forest will randomly choose features space for each tree, it is possible that some tree can separate the 51 positive samples in the rightmost leaf at the previous level, and thus produce a correct result after voting.

- **Support Vector Machine:**

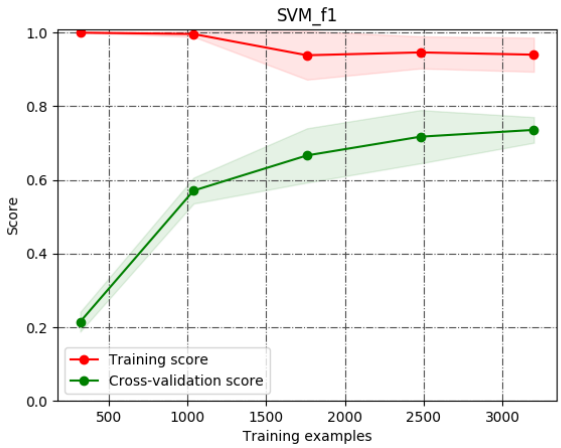




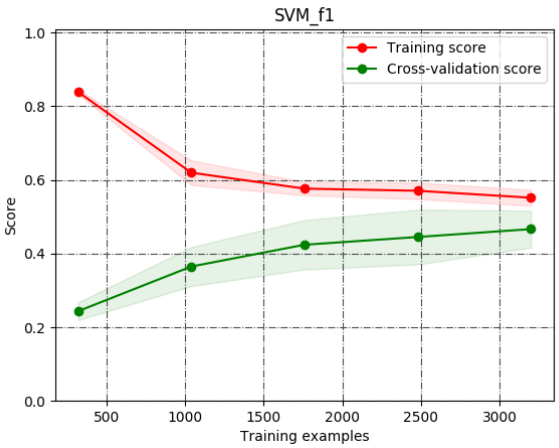
best_parameter - C: 10000,
gamma: 1e-07,
kernel": rbf,
max_iter: 100000

In the result of support vector machine, we can see there are gaps between training and cross-validation score for both F1 and recall, indicating there is overfitting happening. From the recall graph, we can see it only has about 0.8 even in training score. Although this setting gives us the highest validation score among the parameter space, SVM is likely to treat positive samples in an imbalanced dataset as noise, since it is even not able to get the positive data in its training set correct.

One observation is that when we switch to ‘linear’ or ‘poly’, the model performs poorly and are not even going to converge. The reason for this may be that our dataset is not separable in linear or poly space, so it is hard for the model to optimize with this kernel. Another Interesting finding is that the C parameter is very sensitive in this classification problem, and it has a tendency toward a bigger C. When I changed C to 1000, the performance decreased significantly as the following graph shows. C is a regularization parameter for SVM. If we use bigger C, SVM will try to get everything right no matter how small the margin is. On the contrary, SVM will try to find a bigger margin even there will be some classification



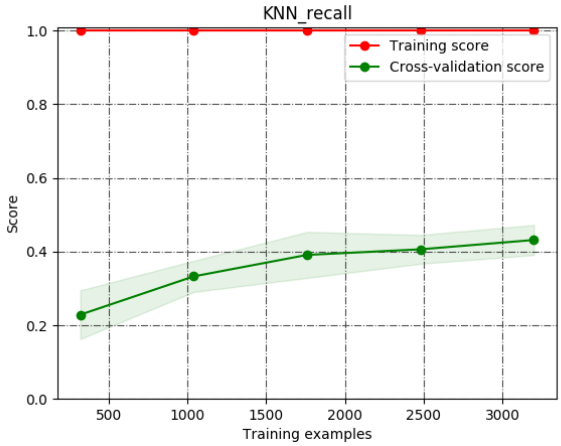
C = 1000000

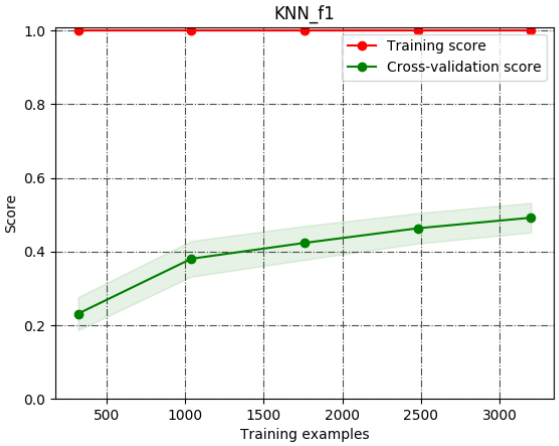


C = 1000

error. Such an Imbalanced dataset like this one will make the positive sample more ‘sparse’ to each other, which require SVM use a bigger C and train a more complex model to get it right. Smaller C will just treat lots of positive sample as noise. However, high C will make the model more likely to overfit.

- **K-nearest Neighbors:**



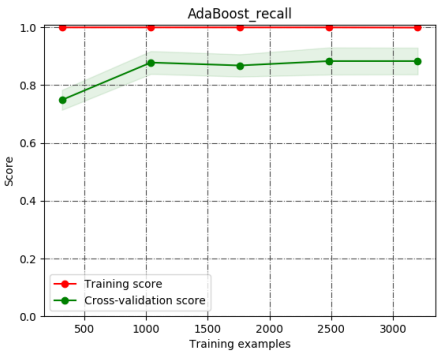


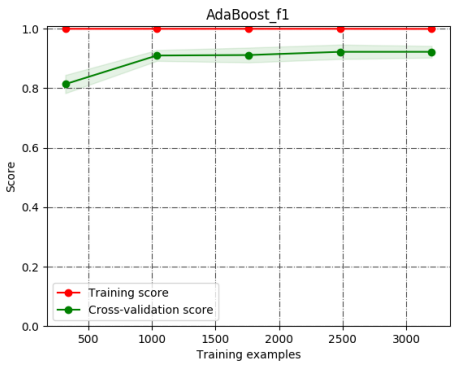
best parameters - n_neighbors: 1,
weights: 'uniform',
p: 1

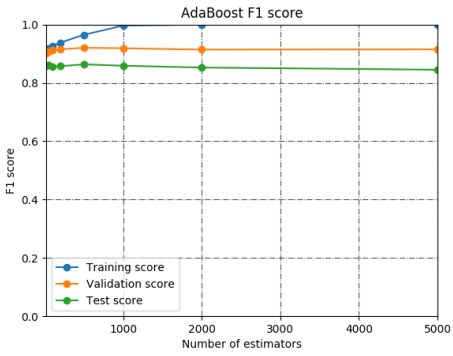
The best parameter I found using grid search is very surprising in the beginning. It just the n_neighbors = 1, which makes KNN entirely overfit to the data. Indeed, no matter how I try to increase the n_neighbors, the f1 validation score is not increasing even though the gap might be smaller. However, it makes sense when you consider the factor that data is imbalanced. Since the positive data are far more away from each other, taking more point into consideration will just let the result become negative (since nearly all your surrounding are negative.) Thus, choosing one will maximize the validation score even though it is still low.

One of the improvements for this case could be that if we see one positive sample in k neighbors, we label it as positive. However, it is also possible to get a low precision since a negative sample will be classified if it is too close to a positive one.

- **Boosting:**







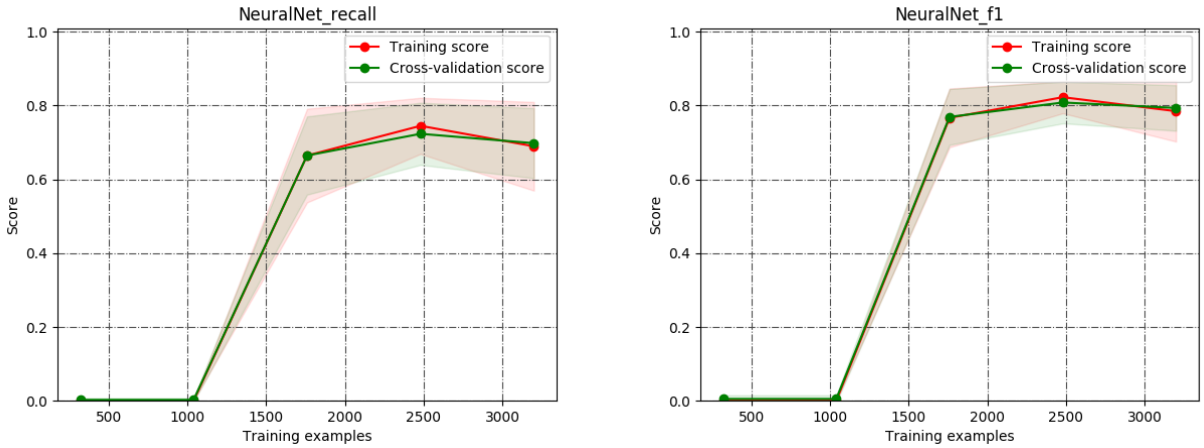
best parameter - learning_rate: 0.1,
n_estimators: 2000

As I expected in the beginning, boosting gave me a high performance in terms of F1 and recall score. The difference between boosting and decision tree is that decision tree still have a little bias (its training still have a gap with 1.0), while Adaboost fixed this problem but introduced overfitting to our model. On the other hand, Adaboost can fit all positive training sample without suffering from significant overfitting comparing with SVM.

My explanation is that since a decision tree with max_depth = 2 is strong enough to get good performance (decision tree visualization in the previous section shows that the first level split can have 80% of positive samples separated), decision stump may not be weak enough for boosting in this case. Therefore, a 'not weak enough' estimator results in overfitting after perform boosting. Nevertheless, we can still try to acquire more data to get a better result.

If you see the rightmost figure, you will found out that the amount of overfitting are barely changing when the model complexity (n_estimators) increase compared with decision tree. Thus, we can say that Adaboost still have some ability to resist overfitting.

- **Neural Network:**



best parameter - "activation": "relu",
"alpha": 0.0001,
"hidden_layer_sizes": [200],
"learning_rate": "adaptive",
"learning_rate_init": 1e-05,
"max_iter": 1000,
"solver": "adam"

The learning curve of the Neural Network is a little bit wired to explain. It shows that it may underfit the train data. However, no matter how change the parameter, including increasing the size and number of layer in the network, and the learning rate, it still give me the similar or even worse result. Nevertheless, if we see the testing result, we will find out that there is nearly no false positive. My interpretation of this situation is that neural network overfits negative samples and

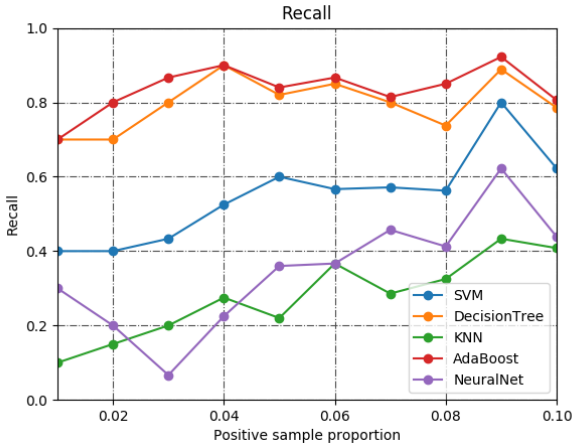
treats lots of the positive sample as noise and therefore even the training recall score are very low.

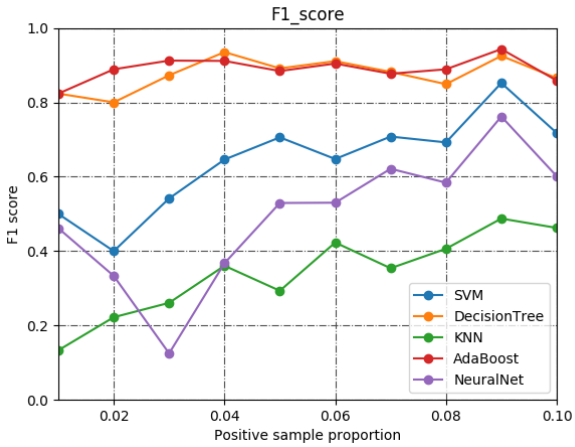
- **Conclusion:**

  the following table shows the overall result of five algorithms on test dataset.

	validation F1 score	Test F1 score	True positive	True negative	False positive	False negative
Decision Tree	0.919	**0.865**	77	898	3	**21**
SVM	0.752	0.718	61	890	11	37
KNN	0.471	0.462	40	866	35	58
AdaBoost	0.921	**0.859**	79	894	7	**19**
Neural Network	0.488	0.601	43	899	2	55

And these two figures are the recall and F1 score for five algorithms running on a fixed size (5000) dataset with different proportion of positive data.





As you can see, Decision tree and Adaboost are more stable on imbalanced data even though the proportion of positive sample goes very low. While Adaboost has a little overfitting in this classification problem. It is possible to add more training data to achieve a better result. On the contrary, SVM, KNN, and Neural Network are more likely to be influenced by imbalanced data and become overfit if we try to get more positive samples correct.

For improving the performance, I will try more aggressive undersampling or oversampling the positive data to create a more balanced dataset. Also, other ensemble methods like bagging or random forest should be explored.

Problem 2: High Dimensional (MNIST) Classification

High dimensional data is another problem that becomes more and more crucial in today's machine learning and data science. For example, a 50x50 grayscale image will have 2500 dimensions. In today's world of big data, it is very common to have high dimensional feature space in our dataset. However, "The Curse of Dimensionality" theory says that when the number of dimension increase, the number of samples needed grow exponentially. MNIST dataset contains digit

images with 784 features which is an ideal dataset for experiment the performance under the curse of dimensionality. In this problem, I want to explore the performance of each algorithm under the dataset with high dimension and limited of data. Also, I will trying dimension reduction using PCA before applying the algorithm and compare the result with original one.

Data preprocessing:

The original MNIST dataset has 10 class (0 ~ 9) with each class around 6000 ~ 7000 samples. To simplify the problem, I select data from 4 and 9, which is relatively hard to distinguish among all pairs of digits, to form a binary classification problem. 1000 instance is randomly sampled from each class to get a total size of (2000, 784) dataset so that the dataset is suffered more from the curse of dimensionality.

Again, this dataset is then separated to training and testing data with a 4:1 ratio. The training data is used to train and validate and the testing data is used only in the final test.

Metrics:

Since it is a balanced binary classification problem, and I care both of the classes equally. I simply choose accuracy as my metric to evaluate the performance.

Result & Analysis:

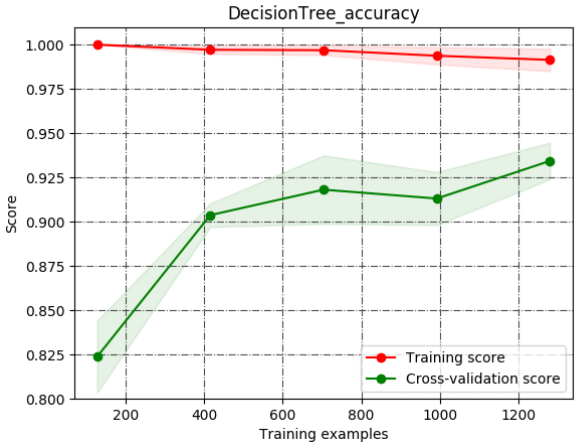
Although the performance of the five algorithms seems to be good on this dataset, we can still see the difference after we dive into the results. I will discuss those results one by one.

- **Decision Tree:**

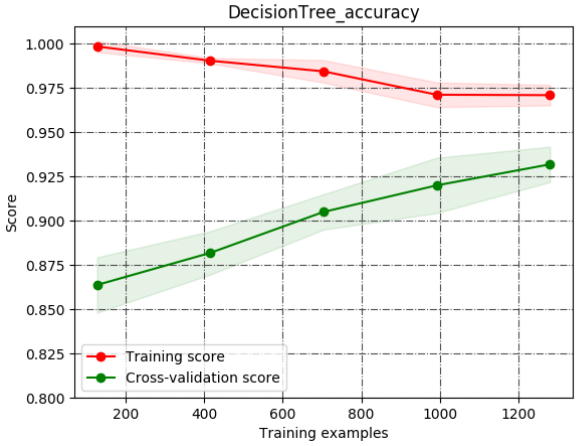
  best parameter -
  criterion: “gini”,
  max_depth: 8

  As we can see in the top learning curve on the right, Decision Tree has a huge gap between the training and validation curve. In fact, the amount of overfitting in Decision Tree is the largest among all the algorithms. If I decrease the max_depth to 5, it will result in the bottom learning curve. Even though the gap between two score lines is decreased, the validation score did not go higher.

  This result demonstrates that Decision Tree is more vulnerable with respect to the high dimensional data. The reason for this is that when dimension



max_depth = 8



max_depth = 5

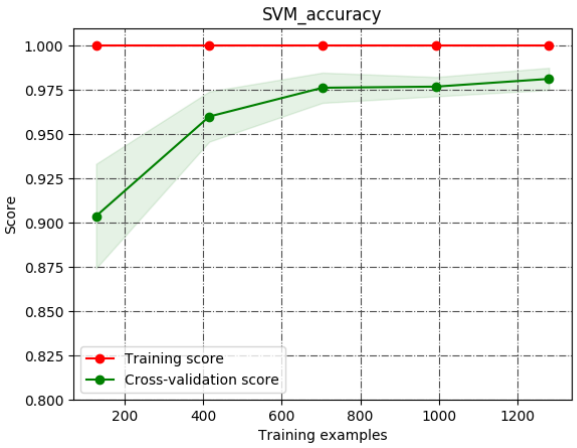
increase, the number of choices we can use for splitting samples increase as well. With limit samples, it is hard for the algorithm to choose one feature over another. Also, as the sample space increases, the distances between data points increases, which makes it much harder to find a "good" split.

One way to tackle this problem is by using Random Forest. Each tree in the random forest uses a subset of features instead of all of them. Therefore, it reduces the space each tree is optimizing and can help to combat with the curse of dimensionality.

- **Support Vector Machine:**

  best parameter -

  kernel: 'poly',
  C: 0.0001,
  degree: 3,
  gamma: 1e-5



C = 0.0001

The top right learning curve shows the performance of SVM on MNIST dataset. In contrast to Decision Tree, the overfitting is less significant.

After searching for why SVM can have the ability to resist the curse of dimensionality, I found that the reason is the excessive regularization it can achieve. In another aspect, SVM model highly depends on the margin we want to optimize, but not very much on the number of feature dimensions. That's why we can use kernel trick to transform our feature to some high dimension (even infinite) space but still achieved good performance. Therefore, SVM should be independent to the dimension of feature space, theoretically. However, in practice, SVM is highly sensitive to regularization parameter C and the kernel we used. For example, change C to 1e-6 will get a very different result as the bottom left figure shows. Using RBF or linear kernel will also perform poorly and the learning curve is meaningless.

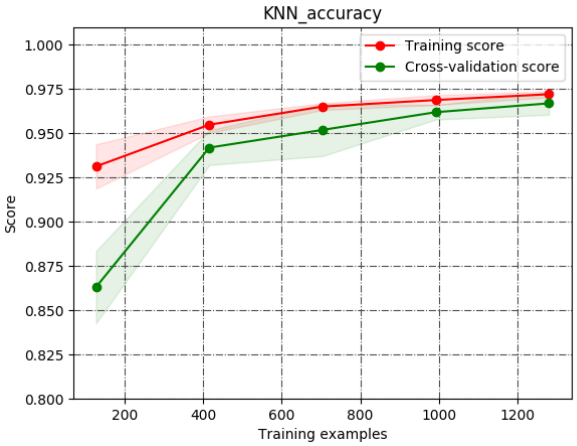
C = 1e-6

- **K-nearest Neighbors:**

  best parameter -

  n_neighbors: 9,
  p: 2,
  weights: "uniform"

In the beginning, I thought the result of K-nearest Neighbor will suffer greatly from the curse of dimensionality since it uses l2 as the distance metric. As the dimension goes up, the distance between each sample increase significantly. Therefore, we need much more data to cover the sample space so that the newly-coming data point will have closer enough points to descript it. That is also the reason that the training score increase as I increase the training data.

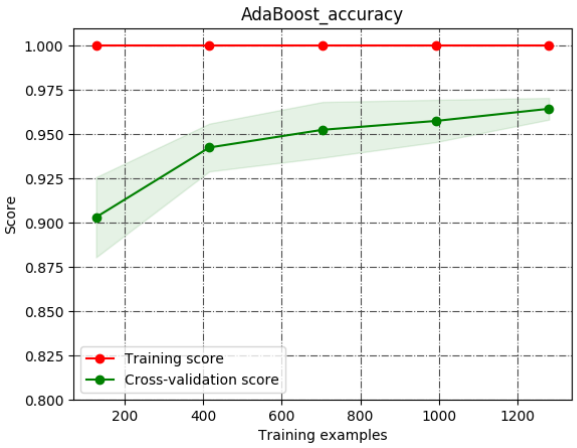


However, KNN performs better as I expect, and is even better than Decision Tree. After diving into the dataset, I found that those pixels who have value will concentrate in the same area (center). In fact, there are 207 features that have only zero values and lots of other features have only 10 to 20 non-zero values. So doing KNN on this dataset may be approximate to KNN on those concentrated areas, which reduce the dimension indirectly.

- **Boosting:**

  best parameter -

  learning_rate: 0.8,
  n_estimators: 500

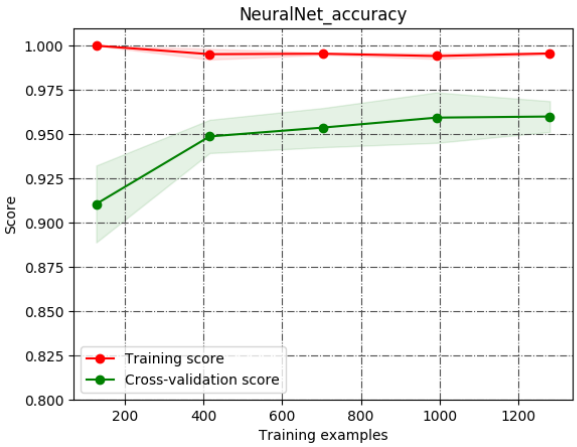


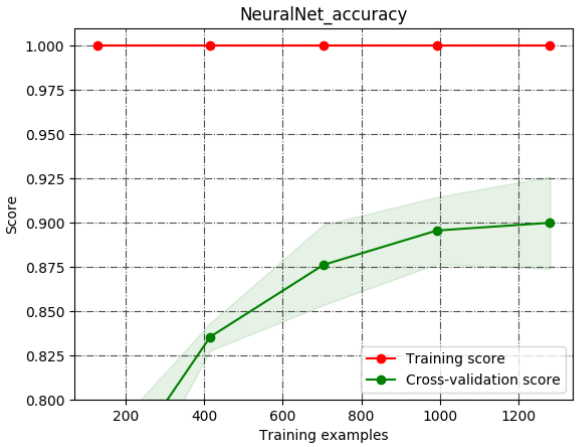
Comparing the Boosting result to Decision Tree, it shows that Boosting does increase the performance a little. Nevertheless, overfitting still exists as the left figure shows. It makes sense that Adaboost will also suffer from high dimensional data, because Adaboost is just a linear combination of multiple Decision Trees, which itself suffer from the curse of dimensionality. However, Boosting can still be a technique to make this effect less significant.

- **Neural Network:**

  best parameter -

  activation: "logistic",
  alpha: 1e-06,
  hidden_layer_sizes: (64, 32, 16),
  learning_rate: "adaptive",
  learning_rate_init: 0.0001,
  max_iter: 1000,
  solver: “adam"





The left figure shows that Neural Network can also suffer from overfitting. The interesting finding is that if I change the activation function to relu, the effect becomes much more significant (as right figure shows.) In Deep Learning field, relu is generally considered to be good for deep Neural Network, since it can reduce gradient vanishing problem and make the network more likely to converge. However, my network here is not a 'deep' one and does not suffer from gradient vanishing. Therefore, use relu here is possible to make the network converge so well that it is overfitting. On the other hand, logistic activation function here could act as some kind of regularization to prevent the network from overfitting.

- **Conclusion:**

The following table shows the result for five algorithms on the testing set:

	Decision Tree	SVM	KNN	AdaBoost	Neural Network
Train acc	0.987	1.000	0.974	1.000	0.996
Val acc	0.934	0.979	0.961	0.958	0.963
Test acc	0.925	0.988	0.958	0.963	0.973
Training time (s)	0.737	2.758	5.276	63.661	89.711

And this table shows the result after using PCA to reduce features space into 115 dimensions, which preserved about 95% of covariance:

	Decision Tree	SVM	KNN	AdaBoost	Neural Network
Train acc	0.978	1.000	0.975	1.000	0.996
Val acc	**0.824**	0.974	0.967	0.955	0.954
Test acc	**0.880**	0.988	0.955	0.960	0.960
Training time (s)	0.565	0.966	0.963	49.508	44.612

Overall, PCA can reduce the training time significantly. The performance of SVM, KNN, Adaboost, and Neural Network are barely changed after doing PCA. This is as expected since the purpose of PCA is

reducing the dimension while keeping as much information as possible and is not aim for significant improvement of performance. Here, we can actually see that the performance of Decision Tree actually decreased. However, after optimizing the hyper-parameters again, I can achieve a *val acc = 0.94* and *test acc = 0.917 that is still close to the original performance.*

In conclusion, SVM is more likely to counter the curse of dimensionality better. Nevertheless, C and kernel function need to be tuned carefully, and searching for optimal hyper-parameter for SVM can be as hard as tackling the overfitting. Non-optimal hyper-parameters for SVM can still lead to overfitting. On the other hand, Decision Tree is an algorithm that works directly on sample space, which is much more likely to suffer from the curse of dimensionality. Adaboost can help to combat this problem, but in this dataset, SVM still perform better than Adaboost. Although the result of KNN did not demonstrate overfitting, I believe that overfitting will happen if we use other high dimension data without so many features contain nearly only zero. As for Neural Network, since it is a relative easy classification problem (lots of people on kaggle can achieve an even better result on classifying all ten classes) and the size of the data is relatively small, it is not an ideal problem for Neural Network to unleash it power. I believe that Neural Network will shine if we can feed it more data.

References:

- scikit-learn https://scikit-learn.org/stable/index.html
- kaggle - Credit Card Fraud Detection https://www.kaggle.com/mlg-ulb/creditcardfraud
- THE MNIST DATABASE of handwritten digit http://yann.lecun.com/exdb/mnist/
- SVM, Overfitting, curse of dimensionality https://stats.stackexchange.com/questions/35276/svm-overfitting-curse-of-dimensionality