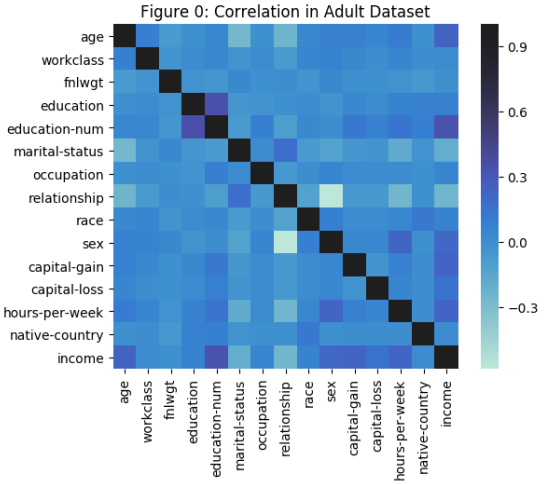
SUPERVISED LEARNING REPORT

DATASET #1

Census Income Data Set. This is a dataset taken from the UCI Repository of a 1994 census data to predict whether the income of an adult exceeds $50k/year based on 14 different attributes such as age, work class, education, occupation, sex, native country, and others. This dataset comes with 48842 different instances which can be used for the classification task.

WHY IS IT AN INTERESTING DATASET?

Figure 0: Correlation in Adult Dataset



The census dataset, also referred as the Adult dataset, is interesting for its practical implications as well as its application in the classification supervised learning task. Estimating the income being above or under a threshold of an adult based on statistical data from census can open up to a bunch of possibilities, for example, banks can predict if a person can be a potential customer by their income. With respect to machine learning it will be interesting to see how supervised learning can classify the income by analyzing various attributes, some which have many categorical features such as 30+ countries, 10+ occupation types, and so on. Another reason this dataset is interesting is because it contains 48842 different instances! So, we have a lot of data to work with, and hence we can see how the algorithms manage to generalize in the presence of so much data. Another reason why this is interesting is because there is class imbalance. About ~75% of the people in the data earn <=50k and ~25% earn >50k. In this situation accuracy alone is not enough to compare the algorithms.

I first analyzed the dataset. I discovered that there were some incomplete rows filled with "?", so I cleaned up the dataset and kept only the clean records. Later I plotted a correlation graph in Figure 0 and realized that `education-num` and `education` were very correlated. This was because one contained the number of years studied, and the other had highest level of study achieved in words. So, I got rid of the one in words. Finally, I had to preprocess my dataset to get better classification results by performing Feature Scaling. This is because to achieve better accuracy, the training needed to be conducted with all features uniformly evaluated. The range of values of `age` (<80), capital-gain (in thousands), and `hours-per-week` are all different and if we use the Euclidean distance metric (as in kNN for instance), since `capital-gain` is much larger than `age` and `hours-per-week`, it would not play any role because it is several orders larger than the other features. I normalized the features to a scale between (0,1), so they contribute equally. I also found that for Neural Networks and SVM, encoding categorical features as binary attributes (for example: Male 0, Female 1) helped achieve better results.

DATASET #2

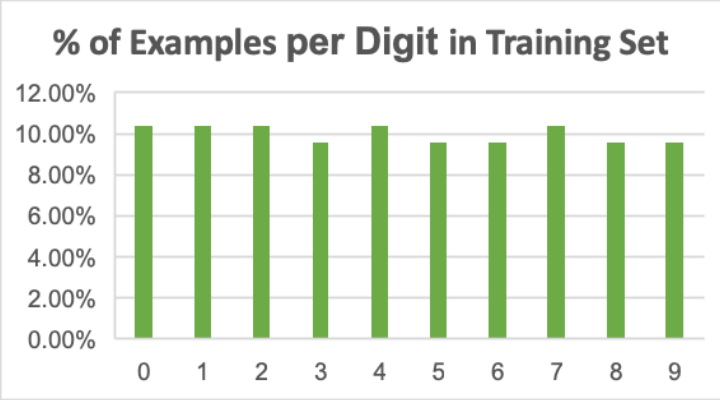
Pen-Based Recognition of Handwritten Digits Data Set. This is a dataset taken from the UCI Repository of handwritten digit samples from 44 writers. This is a multiclass dataset with no missing values, 16 attributes and 10992 instances which can be used for the classification task.

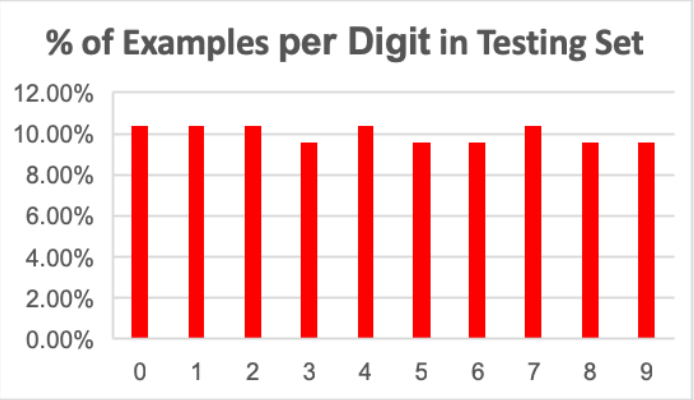
WHY IS IT AN INTERESTING DATASET?

The digit dataset is interesting for its practical implications as well as its application for optical character recognition, which is a computer vision problem applied to images of handwritten documents. My first dataset

was a binary classification, so for my second dataset I wanted to experiment with training machine learning algorithms to create models to tackle this task, which is a multiclass classification problem. It will be interesting to see how supervised learning is used to classify 10 different digits in this dataset. Another thing that is interesting about this dataset is that the database is created by collecting from 44 writers, and dataset is separated such that samples written by 30 writers are used for training and cross-validation, and the digits written by the other 14 writers are used for testing. This is interesting because the data used to train the model should generalize enough the be able to recognize handwriting of totally different people that make up the test set.

A thing to note in this dataset is that the class distribution is even between the training data and test data for all the digits, and we don't have the class imbalance problem as in the first dataset.





Decision Trees

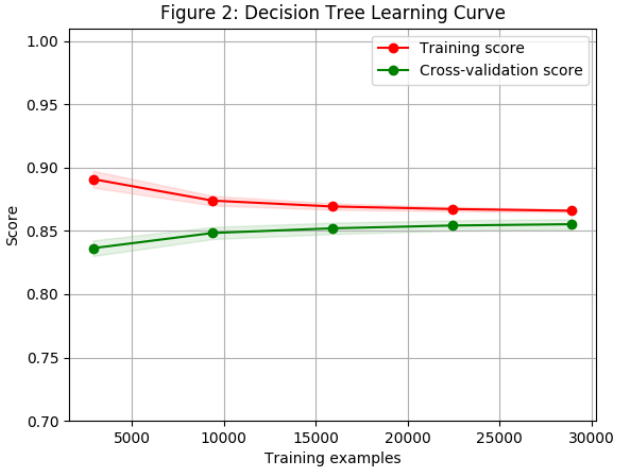
For building the decision trees I used the Scikit Library. After preprocessing, initially I ran the Decision Tree classifier on the Adult dataset with the default parameters and no pruning, splitting in 80/20 for testing and training. However, I noticed that initially I was getting zero error in my training set, and high error for the test set. I figured that this was occurring because my tree is overfitting the training data. Hence, I had to somehow prune my tree. One way to do this was to tune my tree max depth hyperparameter. To determine the best depth for my tree, I plotted my Cross-Validation error and my Training error vs the Tree Depth (Figure 1) and saw that a huge tree with a depth of 50 will fit perfectly the training data but perform poorly in the validation data. On the other hand, if I used a very small tree, both the training and validation sets will perform bad because the tree will underfit. By analyzing the plot, I realized that a depth of about 10 was ideal and any other value higher or lower would either overfit or underfit. I also tried tuning different values of max leaf size but trying out a range of values did not seem to help very much in the prediction outcome. I also experimented splitting criteria with Entropy, explained in lecture, and Gini Index to split attributes, which measures how a random element would be incorrectly classified if it was randomly classified according to the distribution of each label in the subset. I ended up using Gini Index which marginally outperformed Entropy with a higher accuracy.

Figure 1: Decision Tree Error vs Tree Depth

I then trained my model with a tree depth of 10 and plotted the Learning Curve (Figure 2) for visualizing the training and cross-validation curves with respect to the training set size. To get the Learning Curve, I performed cross validation with 100 iterations to get smoother mean test and train score curves, each time with 20% data

randomly selected as a validation set. In the learning curve we observe that the validation curve has almost converged toward the training curve. The gap between them is quite narrow meaning that there is low variance. After using our model with training and test data, we can observe that the accuracy for both validation and test set is about 85%. This means that our classifier is doing a decent job. It is remarkable to note that this was one the fastest algorithm to train.

Figure 2: Decision Tree Learning Curve

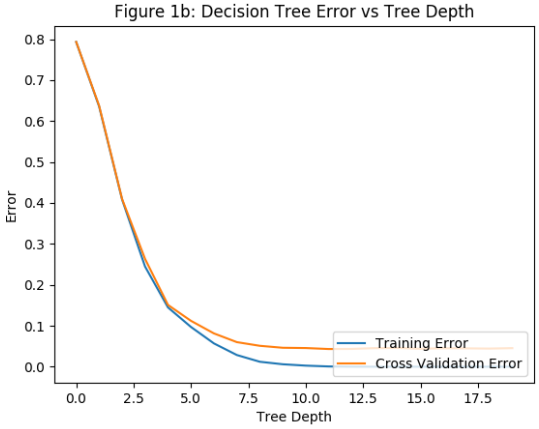



```
Confusion Matrix:
[[6444  363]
 [ 977 1261]]
              precision    recall  f1-score   support

       <=50K       0.87      0.95      0.91      6807
        >50K       0.78      0.56      0.65      2238

   micro avg       0.85      0.85      0.85      9045
   macro avg       0.82      0.76      0.78      9045
weighted avg       0.85      0.85      0.84      9045

Accuracy   : 85.1851851852
Train Score:  0.864278408934
CV Score   :  [ 0.85189485  0.85438262  0.84973878]
Test  Score:  0.851851851852
```

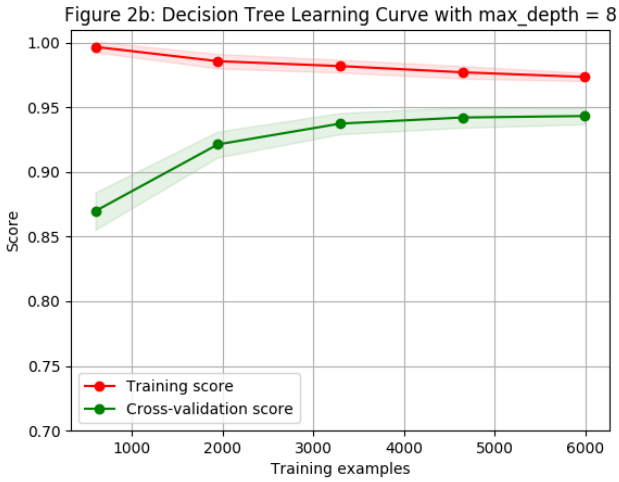
- **Splitting Criteria:** Gini Index (which performed marginally better than Entropy with higher accuracy)
- **Max Depth:** Values ranging from 1-50. The value of 10 did not overfit or underfit the training set.
- **Max Leaf Size**: Values ranging from 2-200. This did not help improve significantly the final accuracy.

With the second dataset, we had 68.2% of our data for training and 31.8% for testing. I first experimented with the splitting criteria. Entropy gave a higher accuracy than Gini Gain on this dataset. Then I wanted to prune my tree to not overfit my training set and get better accuracy on my cross validation and test set. So, I made a plot of the decision tree training error and cross validation as the number of max-depth increased. In Figure 1b, we can observe, that about after a depth of 8, the train error and validation error do not get any better. Hence a depth of 8 is ideal for training our decision tree. I also experimented with the max_leaf_size parameter with values ranging between 2 to 200 and found that a value of 100 gave similar accuracy to max depth of 8, hence I just max depth with produced a much smaller tree giving similar accuracy. I then trained my decision tree with the tuned hyperparameters and performed cross validation with 100 iterations each time with 20% data randomly selected as a validation set and generated the learning curve in Figure 2b. As we can observe we are getting about 94% cross validation accuracy and 89% training accuracy. In the confusion matrix we can observe that the diagonal contains most of the correctly classified values. There is a small gap between the training and validation curves, but no matter how much we tuned the hyperparameters, the accuracy for a single decision tree did not go beyond 89% in the test set.

Figure 1b: Decision Tree Error vs Tree Depth



- **Splitting Criteria:** Entropy (which performed marginally better than Gini Gain with higher accuracy)
- **Max Depth:** Values ranging from 1-50. The value of 8 gave better accuracy.
- **Max Leaf Size**: Values ranging from 2-200. The value of 100 gave similar accuracy to max depth=8.




```
Accuracy   : 89.5368782161
Train Score:  0.971443821724
CV Score   :  [ 0.9332      0.94397759  0.94348697]
Test  Score:  0.895368782161
```

```
Confusion Matrix:
[[353   4   2   0   0   0   0   4   0]
 [  0 294  29  21   2   0   0   2   1  15]
 [  0  19 342   0   0   2   0   1   0   0]
 [  1   3   1 326   0   3   0   1   0   1]
 [  0   1   4   0 329   7   0   2   0  21]
 [  0   0   0  21   8 277   0   1   4  24]
 [  2   8   5   0   1   3 308   0   6   3]
 [  0  40  30   0   4   0   0 284   6   0]
 [  7   3   5   0   0   7   1   5 308   0]
 [  0   5   1   3   2  11   0   3   0 311]]
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       363
           1       0.78      0.81      0.79       364
           2       0.82      0.94      0.87       364
           3       0.88      0.97      0.92       336
           4       0.95      0.90      0.93       364
           5       0.89      0.83      0.86       335
           6       1.00      0.92      0.96       336
           7       0.95      0.78      0.86       364
           8       0.94      0.92      0.93       336
           9       0.83      0.93      0.87       336

   micro avg       0.90      0.90      0.90      3498
   macro avg       0.90      0.90      0.90      3498
weighted avg       0.90      0.90      0.90      3498
```

NEURAL NETWORKS

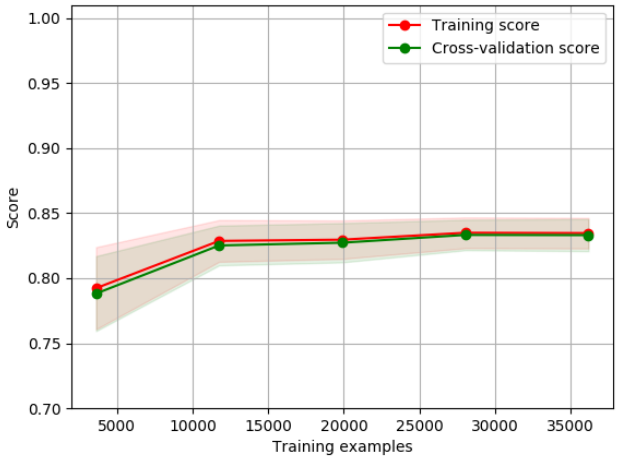
This was possibly one of the most challenging ones to implement with the Adult dataset. The large size of the dataset, and the numerous amounts of hyperparameters to tune (number of layers, number of nodes per layer, activation function, solver, and alpha values) lead to very long training times. I used the Multilayer Perceptron implementation for classification from Scikit.

I run a couple of experiments, altering one hyperparameter at a time, with the dataset to obtain a set of parameters that would give a good result for the dataset. I altered the number of layers and then the number of nodes per layer. This was very tricky because the results would take time (hours) to generate. I experimented initially with one layer and two layers and a combination 5, 10, 25, 50, and 100 nodes. I realized that adding more layers and more nodes per layer were overfitting the data and giving gaps between the training and validation curve (high variance). The best one was (5, 2) nodes in 2 layers. The next thing I tuned was the activation function. I tried the `logistic`, `relu`, `tanh`, and `identity` activation functions. From all of those, `logistic` marginally outperformed. Then for the neural network solver, I did some research and discovered that `adam` solver works well with very large datasets with more than thousands of samples, so I used that. Finally, I tried tuning the alpha value which is the penalty parameter and used 0.0001 which performed better than 1, 0.01, 0.001, 0.0001 and 0.00001.

The results I obtained with the tuned hyperparameters are summarized in the Learning Curve (Figure 3) obtained by performing cross validation with 100 iterations to get smoother mean test and train score curves, each time with 20% data randomly selected as a validation set. We can see in the training accuracy and the cross-validation accuracy curves there is barely any gap, and both are moving close together, meaning that there is no variance/bias problem. We can also note that as the number of training samples increases, the accuracy increases as well, and after 35000 samples it seems that adding more samples will not improve accuracy any further. This implementation manages to get an accuracy of about 85% in the test set.

- **Hidden Layer Sizes**: 2 layers with 5 nodes in the first and 2 in the second.
- **Activation Function:** `Logistic` (which performed marginally better than `relu`, `tanh`, and `identity`)
- **Solver:** `adam` which performs well on large datasets.
- **Alpha**: Penalty value of 0.0001 (which outperforms 1, 0.01, 0.001, 0.0001 and 0.00001).

Figure 3: Neural Network Learning Curve

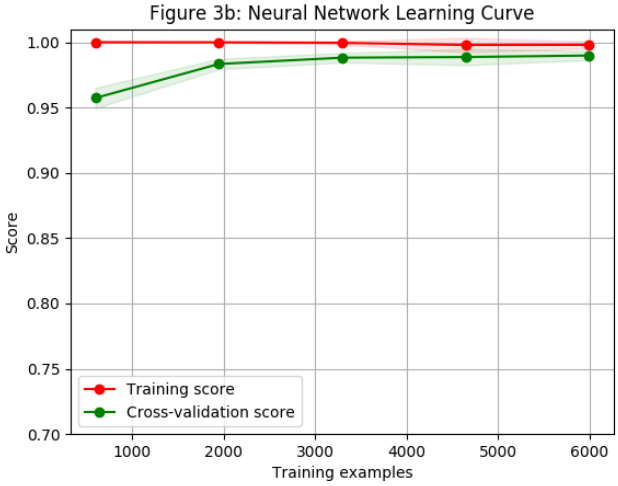



```
Confusion Matrix:
[[6328  476]
 [ 876 1365]]
              precision    recall  f1-score   support

       <=50K       0.88      0.93      0.90      6804
        >50K       0.74      0.61      0.67      2241

   micro avg       0.85      0.85      0.85      9045
   macro avg       0.81      0.77      0.79      9045
weighted avg       0.84      0.85      0.85      9045

Accuracy   : 85.0525152018
Train Score: 0.853746855737
CV Score   : [ 0.85031926  0.84534373  0.84401692]
Test  Score: 0.850525152018
```

For the digits dataset, neural networks got a much better performance than decision trees. With experimentation on the number of hidden layers and nodes per layer I saw that to do well, I needed to increase number of hidden units to 100, 50, 100 (3 layers). The more the number, the more features were extracted by my network. However adding too many features made it overfit and didn't generalize on new data from the testing set. After tuning the remaining hyperparameters such as `adam` solver since this was a big dataset, and the `relu` rectified linear unit activation function gave a good accuracy. I also realized that the more iterations I gave for weight updated, the more accurate was the model's accuracy. After performing all this tuning, I ran my model and performed cross validation with 100 iterations, each time with 20% data randomly selected as a validation set and generated my learning curve in Figure 3b. As we can observe, as the number of samples increase, the training and cross validation accuracy is about 99%. There is no variance or bias problem either, so we aren't overfitting nor underfitting our train data. When using the Neural Net on our test data, an accuracy of about 97.7% was obtained!

- **Hidden Layer Sizes**: 100, 50, 100 (3 layers).
- **Activation Function:** `relu` (which performed better than `logistic`, `tanh`, and `identity`)
- **Solver:** `adam` which performs well on large datasets.
- **Number of Iterations**: 1000

Figure 3b: Neural Network Learning Curve




```
Accuracy   : 97.7129788451
Train Score:  0.998532159061
CV Score   :  [ 0.986       0.99039616  0.98917836]
Test  Score:  0.977129788451
```

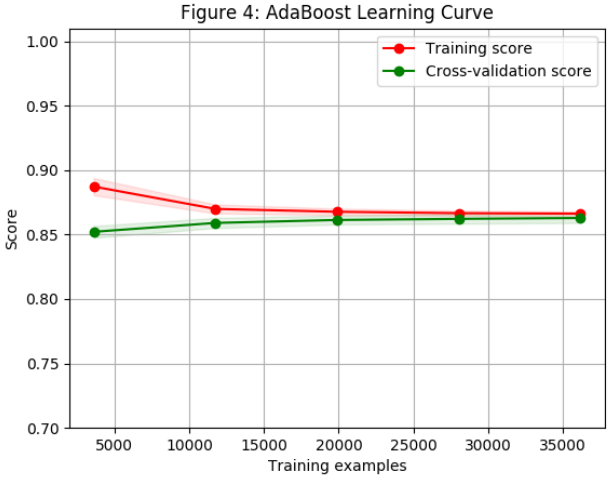
```
Confusion Matrix:
[[348   0   0   0   0   0   6   0   9   0]
 [  0 356   1   0   2   0   0   5   0   0]
 [  0   1 362   0   0   0   0   1   0   0]
 [  0   1   0 332   0   1   0   0   0   2]
 [  0   0   0   0 354   9   0   0   0   1]
 [  0   0   0   3   0 330   0   0   1   1]
 [  0   0   0   0   0   0 335   0   1   0]
 [  0   4   2   0  19   0   0 339   0   0]
 [  2   0   0   0   0   1   0   0 333   0]
 [  0   0   0   0   0   2   0   4   1 329]]
              precision    recall  f1-score   support

           0       0.99      0.96      0.98       363
           1       0.98      0.98      0.98       364
           2       0.99      0.99      0.99       364
           3       0.99      0.99      0.99       336
           4       0.94      0.97      0.96       364
           5       0.96      0.99      0.97       335
           6       0.98      1.00      0.99       336
           7       0.97      0.93      0.95       364
           8       0.97      0.99      0.98       336
           9       0.99      0.98      0.98       336

   micro avg       0.98      0.98      0.98      3498
   macro avg       0.98      0.98      0.98      3498
weighted avg       0.98      0.98      0.98      3498
```

BOOSTING (ADABOOST)

For boosting, I used the AdaBoost implementation from Scikit with the same decision tree used before. The main difference is that I could be more aggressive about pruning, so in the Adult dataset instead of limiting the depth of my tree to 10, I reduced the depth limit to 5 after experimenting with a range of values between 1 and 10. Compared to just using decision trees alone, since AdaBoost adjusts the weights of incorrectly classified samples, such that subsequent classifiers focus more on difficult cases, it does an overall better work with the ensemble of weak learners (shorter decision trees in this case). As we can see from the Learning Curve in Figure 4, generated by performing cross validation with 100 iterations to get smoother mean test and train score curves, each time with 20% data randomly selected as a validation set, the gap becomes more and more narrow as the training set size increases. Since the gap is very narrow, we can safely conclude that the variance is low, and the curves converge to an irreducible error. To achieve this result also I tuned the number of estimators to 50 which performed best between 10, 50, and 100.

- **Decision Tree max depth pruning**: 5 which performed best in the range [1, 10].
- **Number of estimators:** 100 (best between 10, 50, 100)

Figure 4: AdaBoost Learning Curve

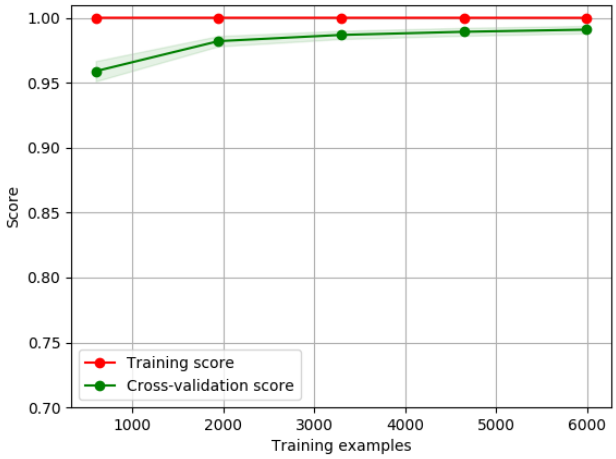



```
Confusion Matrix:
[[6279  523]
 [ 712 1531]]
              precision    recall  f1-score   support

       <=50K       0.90      0.92      0.91      6802
        >50K       0.75      0.68      0.71      2243

   micro avg       0.86      0.86      0.86      9045
   macro avg       0.82      0.80      0.81      9045
weighted avg       0.86      0.86      0.86      9045

Accuracy   : 86.3460475401
Train Score:  0.864250767062
CV Score   :  [ 0.86119403  0.85985571  0.85942942]
Test  Score:  0.863460475401
```

In the digits dataset, I used the same decision tree used from the previous section but I pruned even further my tree to a depth of 5 from the previous depth of 8. I also set my number of estimators to 200, which was the best among 10, 50, 100, and 200. This means more the number of weak learners, the higher is the accuracy. I generated my Learning Curve in Figure 4b performing cross validation, each time with 20% data randomly selected as a validation set. As we can see, the learning curve is similar to the one obtained for Neural Networks on digit dataset. However, the main difference is that here the gap is a slightly bigger, so the accuracy is slightly lower than the Neural Networks for this same dataset in the validation set. For the training set, an accuracy of 96.9% was obtained.

- **Decision Tree max depth pruning**: 5 which performed best in the range [1, 10].
- **Number of estimators:** 200 (best among 10, 50, 100, and 200)

Figure 4b: AdaBoost Learning Curve




```
Accuracy    : 96.9696969697
Train Score:  0.999733119829
CV Score    :  [ 0.9908      0.99239696  0.99038076]
Test  Score:  0.969696969697
```

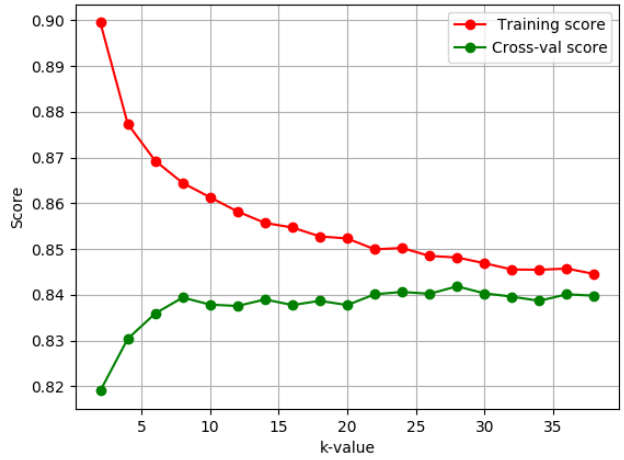
```
Confusion Matrix:
[[343   1   1   0   0   0   5   0  13   0]
 [  0 341  21   0   0   0   0   2   0   0]
 [  0   2 362   0   0   0   0   0   0   0]
 [  0   2   0 332   0   0   0   1   0   1]
 [  0   0   0   0 362   1   1   0   0   0]
 [  0   0   0   6   0 326   0   0   1   2]
 [  0   0   0   0   0   0 336   0   0   0]
 [  0  27   4   0   0   0   0 327   1   5]
 [  0   0   0   0   0   0   0   0 336   0]
 [  0   3   0   2   0   1   0   2   1 327]]
              precision    recall  f1-score   support

           0       1.00      0.94      0.97       363
           1       0.91      0.94      0.92       364
           2       0.93      0.99      0.96       364
           3       0.98      0.99      0.98       336
           4       1.00      0.99      1.00       364
           5       0.99      0.97      0.98       335
           6       0.98      1.00      0.99       336
           7       0.98      0.90      0.94       364
           8       0.95      1.00      0.98       336
           9       0.98      0.97      0.97       336

   micro avg       0.97      0.97      0.97      3498
   macro avg       0.97      0.97      0.97      3498
weighted avg       0.97      0.97      0.97      3498
```

K-Nearest Neighbors

So, in general, the way kNN works is by taking the majority vote of the k-nearest neighbors. Therefore, a low k means that our model will overfit and get most of the training instances right (hence low error). As the value of k increases, we would take the majority vote of many neighbors, hence our model will underfit. To see if this is actually happening with our Adult dataset, I decided to use different values of k to train my kNN Classifier using Scikit and decided to plot the Training Score and 10-fold Validation Score. As we can see from the plot in Figure 5, with a k value of 1, we have a very high training accuracy, but a low testing accuracy. As we increase the value of k, our training accuracy decreases, and our test accuracy increases till a certain point. This means that as we increase k, we are reducing overfitting. From this plot we can also see that after a k value of 30, our training score reaches a plateau, and it does not get any better as the value of k increases. Therefore, a value of k=30 should be a good value for training this dataset. I trained my model with k=30 and performed cross validation thrice with each time with 20% data randomly selected as a validation set. The result obtained is that training, validation, and test accuracy are all about the same. In the test set I obtained an accuracy of about 84.0%.

- **k value:** 30 (best among the range 1 to 40+)
- **Distance Metric:** Euclidean worked better than Mahalanobis Distance

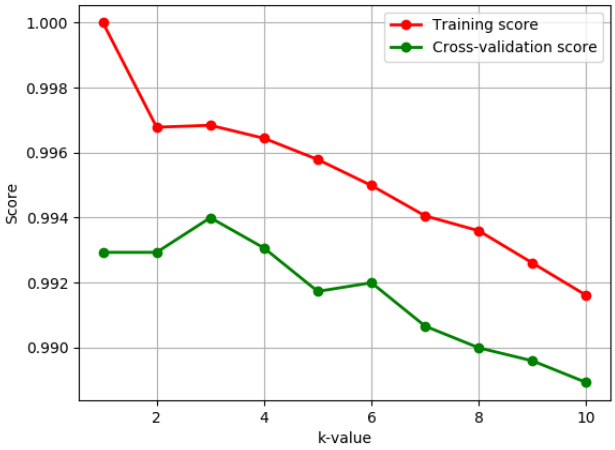
Figure 5: KNN Accuracy vs k-value




```
Confusion Matrix
[[6929  550]
 [1010 1280]]
              precision    recall  f1-score   support

           0       0.87      0.93      0.90      7479
           1       0.70      0.56      0.62      2290

   micro avg       0.84      0.84      0.84      9769
   macro avg       0.79      0.74      0.76      9769
weighted avg       0.83      0.84      0.83      9769

Accuracy    : 84.0311188453
Train Score:  0.846901952755
CV Score    :  [ 0.83147793  0.84160012  0.83614865]
Test  Score:  0.840311188453
```

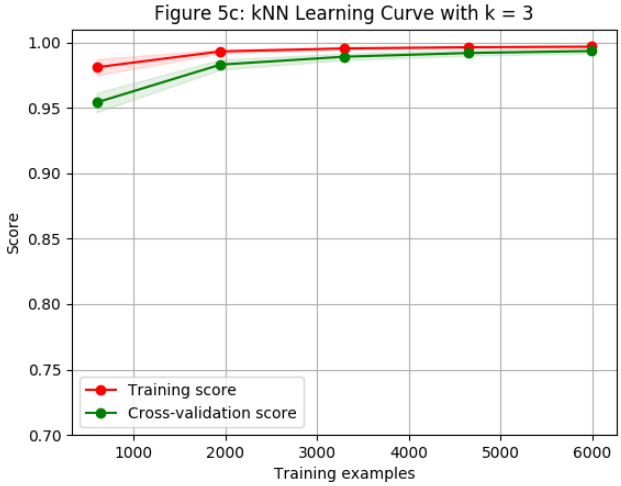
Similarly, for the digits dataset I wanted to plot the training and test error against the cross validation error to determine which should be the best k-value for training my dataset. Comparing both the Mahalanobis and Euclidean distance, Euclidean gave a better accuracy. I got the validation curve in Figure 5b by performing 10-fold CV. As we can see from the figure, the k-value of 3 gave the highest accuracy, and anything before or after gave lower accuracy on the validation set. I then trained my model using Euclidean Distance metric and k=3 and got the Learning Curve in Figure 5c, by performing cross validation with 100 iterations, each time with 20% data randomly selected as a validation set. The final accuracy on the test set was 97.5%. It makes sense that kNN gets good results for this dataset because the closest neighbors are those that have similar digit characteristics to the current digit being analyzed.

Figure 5b: KNN Accuracy vs k-value



- **k value:** 3 (best among the range 1 to 10)
- **Distance Metric:** Euclidean worked better than Mahalanobis Distance

Figure 5c: kNN Learning Curve with k = 3




```
Accuracy    : 97.5986277873
Train Score:  0.996930878036
CV Score    :  [ 0.992       0.99279712  0.99318637]
Test  Score:  0.975986277873
```

```
Confusion Matrix:
[[351   0   1   0   0   0   4   0   0   7]
 [  0 351  10   0   1   0   0   2   0   0]
 [  0   2 362   0   0   0   0   0   0   0]
 [  0   1   0 333   0   0   0   0   0   2]
 [  0   0   0   0 354   9   1   0   0   0]
 [  0   0   0   6   0 327   0   0   0   2]
 [  0   0   0   0   0   0 336   0   0   0]
 [  0  10   1   0   0   0   2 347   4   0]
 [  1   0   0   0   0   1   0   0 334   0]
 [  0   3   0   7   1   1   0   4   1 319]]
              precision    recall  f1-score   support

           0       1.00      0.97      0.98       363
           1       0.96      0.96      0.96       364
           2       0.97      0.99      0.98       364
           3       0.96      0.99      0.98       336
           4       0.99      0.97      0.98       364
           5       0.97      0.98      0.97       335
           6       0.98      1.00      0.99       336
           7       0.98      0.95      0.97       364
           8       0.99      0.99      0.99       336
           9       0.97      0.95      0.96       336

   micro avg       0.98      0.98      0.98      3498
   macro avg       0.98      0.98      0.98      3498
weighted avg       0.98      0.98      0.98      3498
```

Support Vector Machines

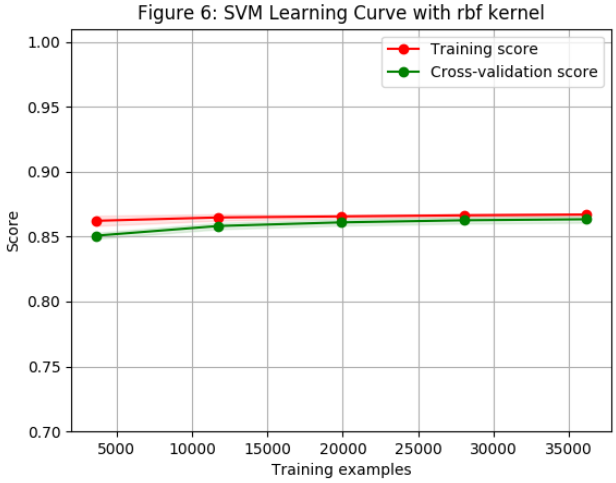
For the Support Vector Machine Classifier, I experimented with different kernels from the Scikit library, as well as the C value which is the error penalty term and the tolerance for stopping criteria. The good thing about SVMs is if our data is not linearly separable in a single dimension, SVM will find a dimension that can separate the data. However, this led to huge runtimes. In fact, it took much longer than Neural Networks for certain kernels in the Adult dataset.

The kernels I experimented were `rbf` which marginally outperformed the `linear`, `poly` and `sigmoid` kernel in that order. In fact, apart from `rbf` giving the best accuracy, it was the fastest to give cross validation & learning curve results in under 2 hours. For the penalty term C, we tell how much we want to avoid misclassifying each training example. A larger C would give smaller margin, and small C will get a larger margin. On that note, I experimented with C values of 0.1, 0.5, 1, and 10 and C = 1 was best amongst them. The final hyperparameter

I tuned was the tolerance term which is used as a stopping criterion. This tells Scikit to stop searching once a certain tolerance is achieved. I used 0.001 which performed better than 1, 0.01, 0.001, 0.0001 and 0.00001. In Figure 6 we can observe the SVM learning curve with the `rbf` kernel, which managed to achieve an accuracy of about 84% in our test set.

- **Kernel**: `rbf` which performed better than `linear`, `poly` and `sigmoid` in that order.
- **C (error penalty term):** 1 (best between 0.1, 0.5, 1, and 10)
- **Tolerance**: 0.001 (better between 1, 0.01, 0.001, 0.0001 and 0.00001)

Figure 6: SVM Learning Curve with rbf kernel

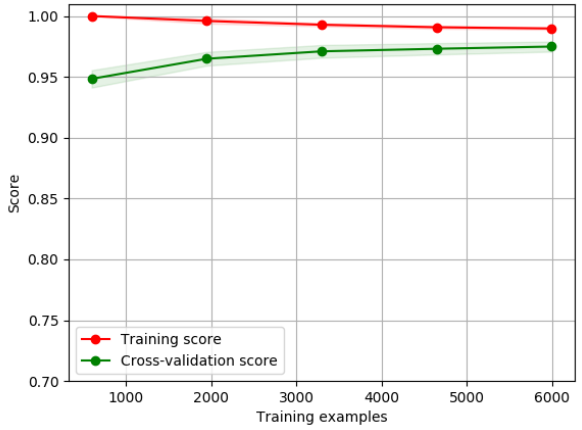



```
Confusion Matrix:
[[6319  516]
 [ 878 1332]]
              precision    recall  f1-score   support

       <=50K       0.88      0.92      0.90      6835
        >50K       0.72      0.60      0.66      2210

   micro avg       0.85      0.85      0.85      9045
   macro avg       0.80      0.76      0.78      9045
weighted avg       0.84      0.85      0.84      9045

Accuracy   : 84.5881702598
Train Score:  0.85291759958
CV Score   :  [ 0.84046434  0.84476325  0.84275999]
Test  Score:  0.845881702598
```

For the digits dataset, I experimented with the `linear` and the `rbf` kernel. For the both kernels, and mostly for the `rbf` kernel, apart from the `C` penalty term, and the `tolerance`, I had to tune a parameter called `gamma`, that defines how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'. This parameter with a value of 0.001 had most impact in increasing the accuracy in the validation and test sets. I plotted the Learning Curve performing cross validation with 100 iterations, each time with 20% data randomly selected as a validation set in Figure 6b for the `linear` kernel, and Figure 6c for the `rbf` kernel. Among both, the `rbf` kernel had higher accuracy in the test set of 95.5%.

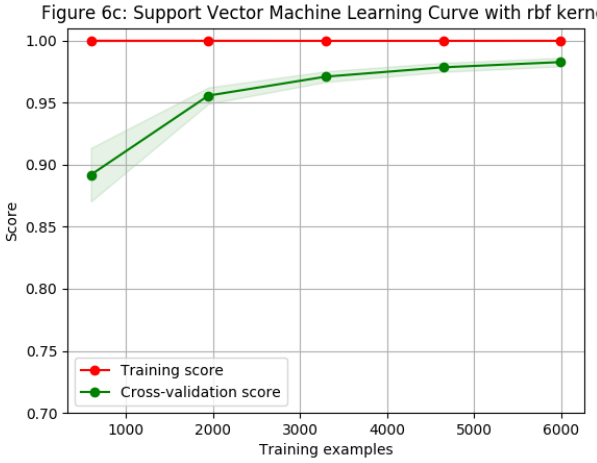
- **Kernel**: `rbf` which performed better than `linear`.
- **C (error penalty term):** 1 (best between 0.1, 0.5, 1, and 10)
- **Tolerance**: 0.001 (better between 1, 0.01, 0.001, 0.0001 and 0.00001)
- **Gamma**: 0.001

Figure 6b: Support Vector Machine Learning Curve with linear kernel




```
Accuracy   : 93.9393939394
Train Score:  0.988924472912
CV Score   :  [ 0.9712      0.97358944  0.97755511]
Test  Score:  0.939393939394
```

Figure 6c: Support Vector Machine Learning Curve with rbf kernel




```
Accuracy    : 95.5974842767
Train Score:  0.999733119829
CV Score    :  [ 0.9804      0.97759104  0.97995992]
Test  Score:  0.955974842767
```

```
Confusion Matrix:
[[337   0   0   0   0   0   0  26   0]
 [  0 358   2   0   0   0   0   1   3   0]
 [  0   1 359   0   0   0   0   0   4   0]
 [  0   1   0 328   0   0   0   0   7   0]
 [  0   0   0   0 342   0   0   0  21   1]
 [  0   0   0   4   0 313   0   0   5  13]
 [  0   0   0   0   0   0 318   0  18   0]
 [  0   9   0   0   0   0   0 330  25   0]
 [  0   0   0   0   0   0   0   0 336   0]
 [  0   1   0   0   0   0   0   3   9 323]]
              precision    recall  f1-score   support

           0       1.00      0.93      0.96       363
           1       0.97      0.98      0.98       364
           2       0.99      0.99      0.99       364
           3       0.99      0.98      0.98       336
           4       1.00      0.94      0.97       364
           5       1.00      0.93      0.97       335
           6       1.00      0.95      0.97       336
           7       0.99      0.91      0.95       364
           8       0.74      1.00      0.85       336
           9       0.96      0.96      0.96       336

   micro avg       0.96      0.96      0.96      3498
   macro avg       0.96      0.96      0.96      3498
weighted avg       0.96      0.96      0.96      3498
```

Analysis of Results

Comparison of the different algorithms. Why did I get the results I did? Which algorithm performed best?

For the Adult dataset, as I described earlier, we have class imbalance, so accuracy isn't a very helpful metric to compare our algorithms. We care about getting more correct predictions for the >50K case. For instance, if we use our dataset in the tax evasion case, if a fraudulent person earning more than 50k (Actual Positive) is predicted as non-fraudulent (Predicted Negative), the consequence can be very bad. Hence comparing using Recall might be useful for this dataset. Combining the performance statistics of all the models implemented above, we can see that AdaBoost has the highest F1-score (0.71), Recall (0.68), and Accuracy (0.86). I also plotted the ROC curves for all the algorithms above in Figure 7. From the amplified version of the top left corner in Figure 8, we can see that the ROC curve of the AdaBoost model has the highest lift and is closest to the top left corner, and also has the highest area under the curve (AUC = 0.911). AdaBoost's curve clearly separates itself from the other models' curves. Hence for this dataset AdaBoost would be the preferred model, not only because it has the highest Accuracy, Recall and F1-score, because it also simple to implement, and because it is made up of weak learners it's less susceptible to overfitting. However, it is sensitive to noisy data and outliers. There is always a tradeoff between choosing the best algorithm for a dataset. There is no free lunch!

Figure 7: Receiver operating characteristic (ROC)

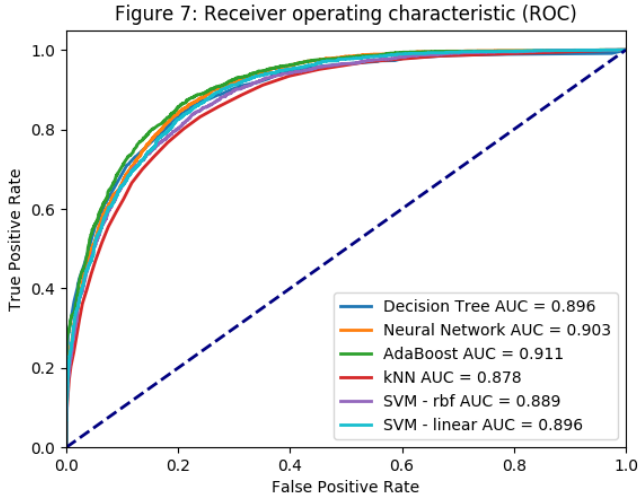
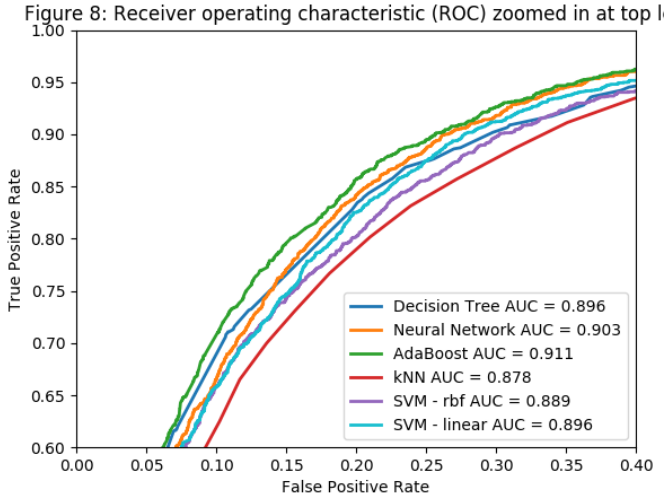
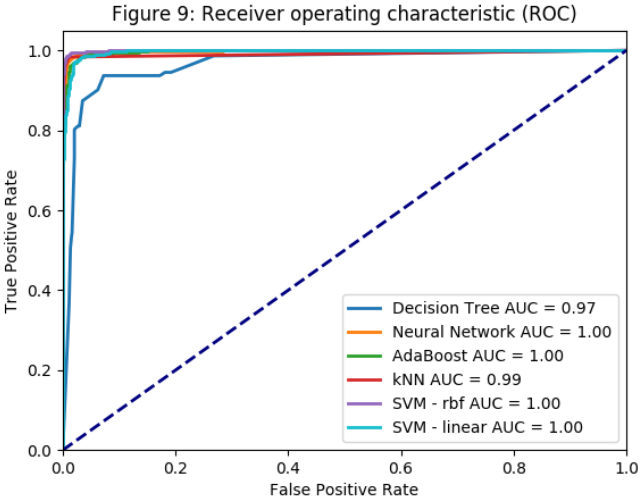


Figure 8: Receiver operating characteristic (ROC) zoomed in at top left



For the digit dataset dataset, since our classes are balanced, the intuitive measure to use to compare the algorithms would be Accuracy. From all the algorithms, the one with the highest accuracy was Neural Networks with an accuracy of 97.7% followed by a close call with kNN with 97.5%. The rest also had very high accuracies, and the lowest one was Decision Trees with 89.5% accuracy. I also plot in Figure 9 the ROC curves for all algorithms in the second dataset. As we can observe the AUC for Neural Networks, SVMs (with linear and rbf kernel), and AdaBoost are almost 1.0 followed by kNN with an AUC of 0.99. These are those that are concentrated in the top left of the plot. The one with lowest AUC is decision tree which got the lowest training set accuracy as well.

Figure 9: Receiver operating characteristic (ROC)



Due to the nature of the dataset, mostly all algorithms managed to generalize well and get good test set accuracy on unknown digit samples. In this case all algorithms did a good job but which one should be the best one to choose? Neural Nets / SVMs take long time to train. kNN takes no time to train, but does take some time to query. AdaBoost doesn't take long to train, is easy to implement, and gives a similar accuracy. It will boil down to which one gives you the best accuracy and takes the lowest time to train. In my opinion AdaBoost is great if we are not willing to spend a lot of time and effort tuning Neural Networks / or SVMs and training them to get a same result than if we can just implement something simple as AdaBoost.

What sort of changes did I make to each of those algorithms to improve performance?

This is described in detail for each algorithm above. In summary:

- **Decision trees**: adjusting the tree depth and number of leaves helped reduce overfitting.
- **Neural networks**: finding a good activation function, and the right number of layers and nodes per layer.
- **AdaBoost**: getting the right estimator number and a good set of weak learners (pruned decision trees).
- **kNN**: getting the best k-value and distance metric.
- **SVM**: getting the right kernel, penalty term, tolerance, and gamma value.

How fast were the algorithms?

For the Adult dataset, decision tree training was the fastest taking less than 10 minutes. Followed by AdaBoost, which had a combination of decision trees, that took about 20 minutes to train. Querying was relatively quick for both. kNN was fast for training but took more than an hour for cross validation since it is slow for querying because it needs to calculate distances every time. Finally, SVM and Neural Networks are tied by taking several hours to generate results. In these times I am factoring in the time to perform cross validation which helped to tune our hyperparameters. For the Digits dataset, Decision Trees and Adaboost were very quick, giving results between 3-5 minutes. kNN on the other hand took about 7 minutes. Neural Networks and SVMs took about 10 minutes.

Did cross validation help?

Cross validation was indeed very helpful. Especially when I used it to generate learning curves, I could easily visualize the plot and check if there was a gap meaning I had a variance problem and was overfitting my training data, or I had high error, meaning there was a bias problem and I was underfitting. This helped tuned hyperparameters and get better performance.

How much performance was due to the problems you chose?

The nature of the Adult dataset problem is such in that the data we have, we cannot generalize with a lot of accuracy. As seen above, we have tuned the algorithms to perform good given the data we have, however we couldn't get an accuracy higher than 86% with all the algorithms. This can be because firstly we have class imbalance, and secondly, we have more samples of a particular type. For instance, about 90% our data is of adults from the US and about 10% of the data is of people from 30+ countries. Ideally if we had a lot of data of every kind of person then we would be able to make better predictions, but given we have a certain type/amount of data, we are limited in what we can learn from it.

On the other hand, the digits dataset had a lot of samples, and all of them were equally distributed across all the classes (every digit). The other detail was that there was almost no noise, and every feature had only relevant information, so mostly all the algorithms were able to crack the dataset and get good performance.

FINAL THOUGHTS

On a final note after doing this assignment, I would like to mention that no algorithm is superior to another. The algorithm that works best for a problem is specific to the particular dataset and the kind of data we get. There is no free lunch. Always looking first at the data, analyzing it, and then running algorithms, deciding what we want to optimize for. For instance:

- Type I error: to falsely infer the existence of something that is not there, such as having a non-spam email being classified as spam, or
- Type II error, to falsely infer the absence of something that is present, such as making sure that a disease does not go undetected)

Then we can perform the tuning hyperparameters accordingly. Also, Cross validation and Learning Curves are very helpful to detect bias or variance problems, to then tune hyper parameters to accordingly reduce underfitting or overfitting. Then the outcome of the tuned model can help us determine which algorithm will perform best for a particular task.