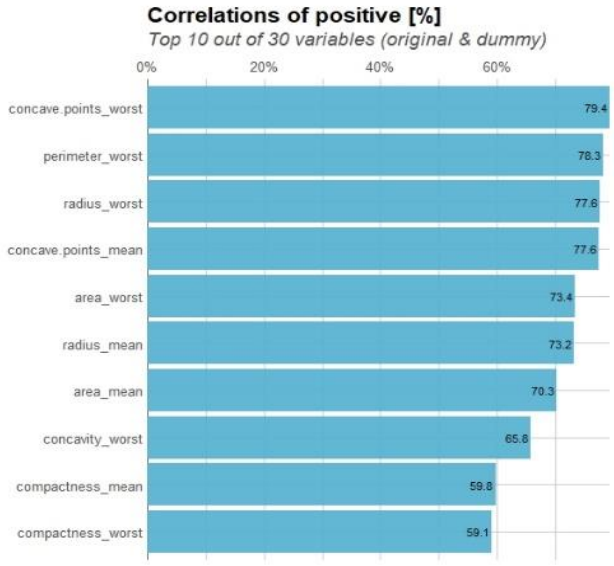
Sahil Desai
Ebhrahim Mortaz
MGT 388
5/9/2020

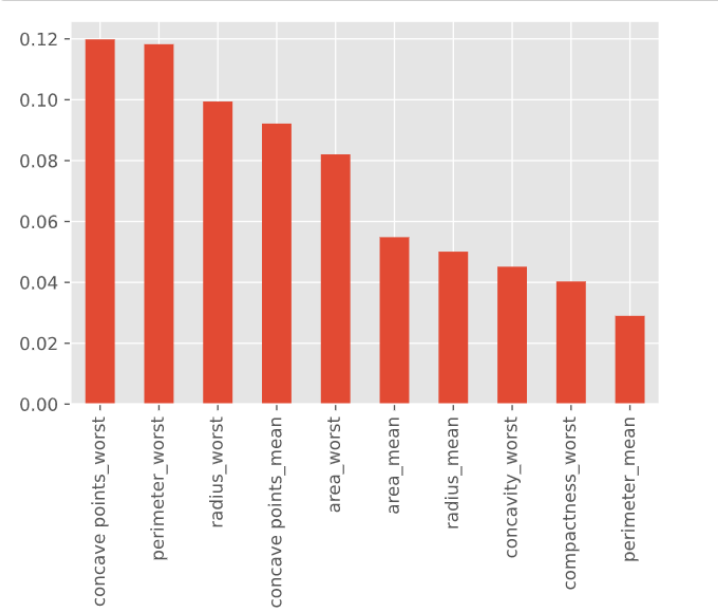
Final Project Report

This project was a great introduction to testing different machine learning algorithms. I did run into some problems we discussed in class such as overfitting (which I will discuss later in the report) which helped me gain deeper understanding on how to tune different algorithms. This dataset was mainly numerical, with one target variable being the classifier for cancerous and noncancerous. I will start by discussing my original plan with the EDA analysis.

To begin, I found that there were a lot of numerical variables to explore. First, I dropped the ID column since that yields no value. Then, I wanted to find the variables that were most highly correlated with the diagnosis column. Since this was the beginning of the project, I went straight to R and created a plot that pits diagnosis with its most highly correlated variables (do not worry! This is the only time I used R in this project since it was the beginning). This was created using the Lares package.



Alternatively, there is a feature selection technique from sklearn that plots a similar graph



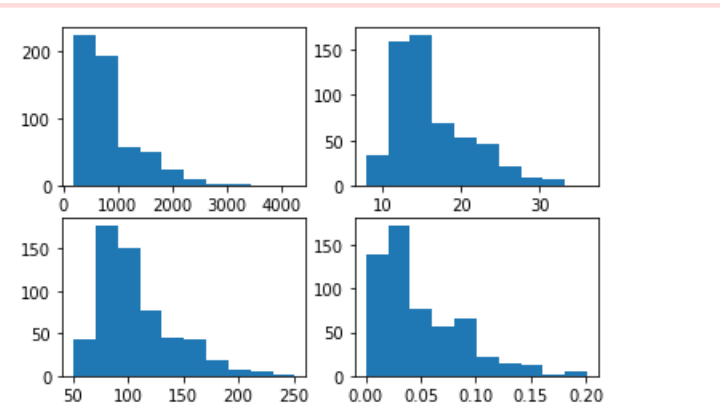
From here, it was simple EDA such as using the describe function to see how the data for each column was distributed. The count for benign (B) and malignant (M) tumors was also calculated with B having 357 counts and M having 212 counts. In addition, I also found median and variance of the top correlated columns and incorporated it into an easy to read data frame.

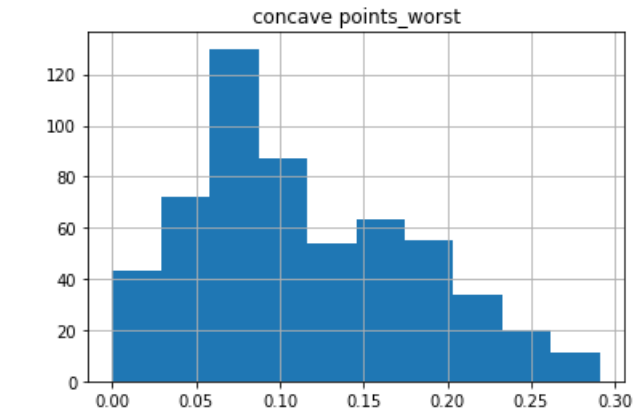
	median	var
radius_mean	13.375000	12.435853
area_mean	548.750000	124715.175511
compactness_mean	0.092525	0.002794
concave points_mean	0.033500	0.001504
radius_worst	14.970000	23.426565
perimeter_worst	97.660000	1129.130847
area_worst	686.500000	324167.385102
compactness_worst	0.211000	0.024755
concavity_worst	0.226550	0.043274
concave points_worst	0.099930	0.004321

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean
count	568.000000	569.000000	569.000000	568.000000	566.000000	568.000000	567.000000	567.000000	567.000000
mean	14.130245	19.289649	105.047821	654.229225	0.096248	0.104354	61.817244	0.048854	0.710289
std	3.526450	4.301036	312.218444	353.150358	0.013971	0.052858	1469.858104	0.038787	12.596692
min	6.981000	9.710000	43.790000	0.800000	0.052630	0.019380	0.000000	0.000000	0.106000
25%	11.697500	16.170000	75.210000	419.900000	0.086130	0.064815	0.029570	0.020310	0.162000
50%	13.375000	18.840000	86.340000	548.750000	0.095825	0.092525	0.061550	0.033500	0.179300
75%	15.797500	21.800000	104.300000	784.150000	0.105250	0.130425	0.132000	0.073820	0.195800
max	28.110000	39.280000	7517.000000	2501.000000	0.163400	0.345400	35000.000000	0.201200	300.130000

I did a quick checkup on the NA values in the dataset, and surprisingly there were not many listed among the features (under 30 NA values total). I decided it would okay to completely remove rows with NA's for the machine learning portion as this would not result in much data loss.

To check for data distribution, I decided to use simple histograms over the only the top 5 most correlated values. Below are four of those plots, with my Python program which one is which. The last plot is inherently the most correlated feature of the bunch



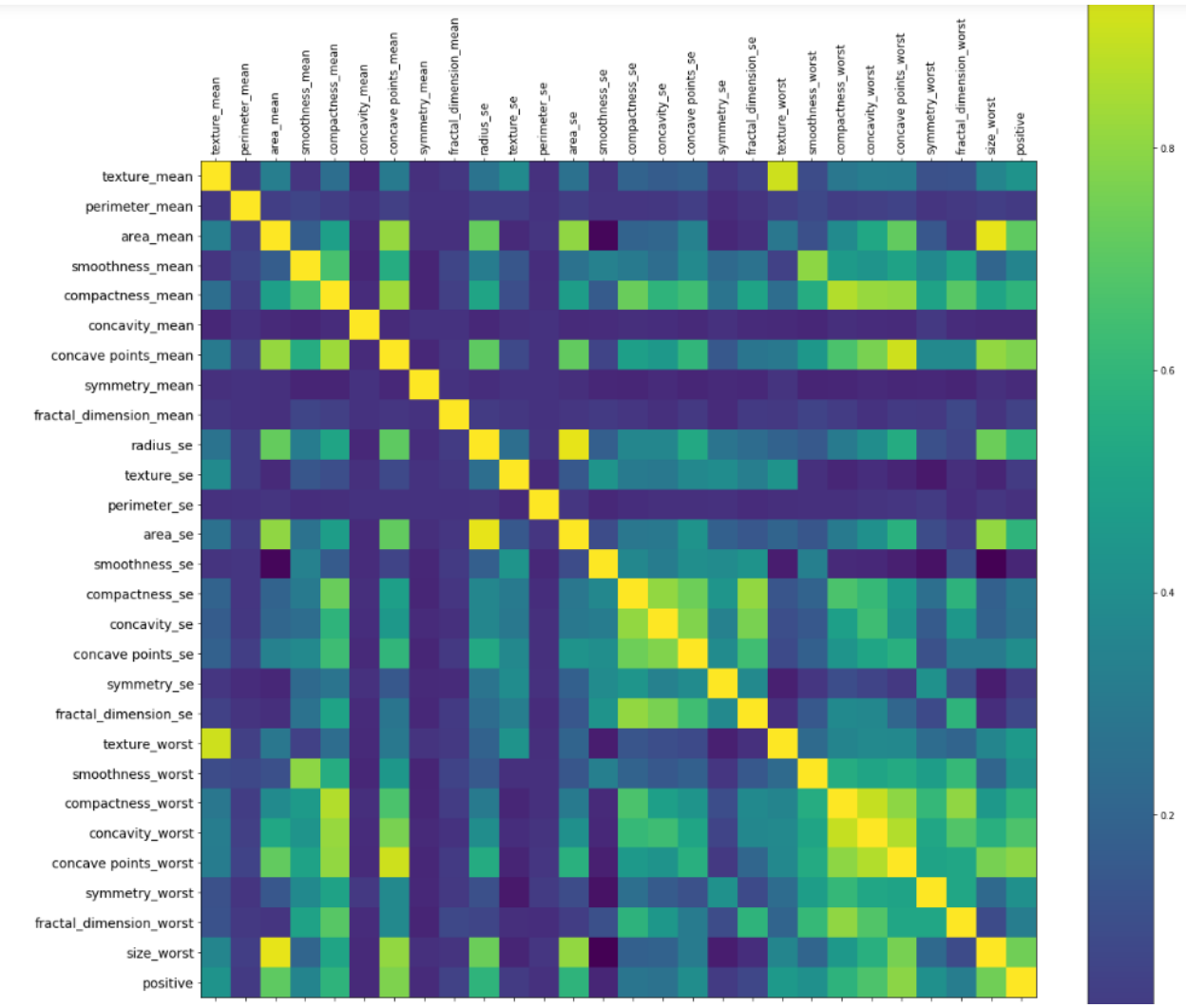


As for identifying outliers, I used a quantile function and initially wanted to filter out values that were above and below the 95% intervals. However, this resulted in too much data loss, and I found that the outliers in the dataset were more extreme. I filtered out the values above and below the 99.99% intervals among the feature columns. We can see from this output some very glaring outliers that would negatively impact the machine learning models

```
        radius_mean  texture_mean  perimeter_mean   area_mean  \
0.0001     7.021257      9.748056       44.024584     8.89109
0.9999    28.070877     38.969304     7100.741200  2500.88660

        smoothness_mean  compactness_mean  concavity_mean  \
0.0001         0.053188          0.019610        0.000000
0.9999         0.162343          0.343472    33019.024157

        concave points_mean  symmetry_mean  fractal_dimension_mean  ...  \
0.0001              0.00000       0.106606                0.049976  ...
0.9999              0.20064     283.159848             5141.933825  ...

        radius_worst  texture_worst  perimeter_worst  area_worst  \
0.0001      7.972337      12.046649        50.641744   187.38112
0.9999     35.875294      49.405054       249.956080  4207.31040

        smoothness_worst  compactness_worst  concavity_worst  \
0.0001          0.071743           0.027689         0.000000
0.9999          0.222361           1.051178         1.247351

        concave points_worst  symmetry_worst  fractal_dimension_worst
0.0001               0.00000        0.156506                  0.05505
0.9999               0.29096        0.658892                  0.20554
```

After I removed these, I decided to engineer a new variable called ‘size_worst’ which is a culmination of adding area_worst, radius_worst, and perimeter_worst columns. From here, I decided to drop these three as I felt it would have been redundant data that would not benefit my models at all. To finish off the EDA, I added a dummy variable for the diagnosis column so my machine learning models can have a target variable. B would be coded as 0 while M would be coded as 1 for each observation. I then dropped the diagnosis column as it was no longer needed and created a heat map correlations matrix to finish this part of the project. To begin the machine learning portion, I ran the info function to double-check I had the columns I wanted.

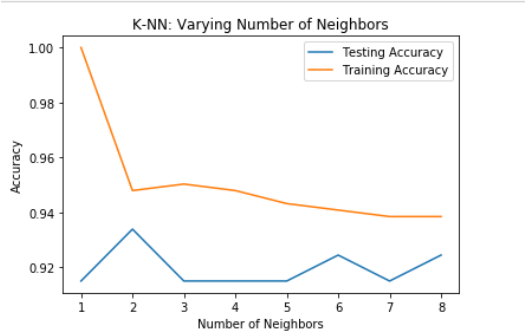
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 551 entries, 0 to 567
Data columns (total 29 columns):
radius_mean                550 non-null float64
texture_mean               551 non-null float64
perimeter_mean             551 non-null float64
area_mean                  551 non-null float64
smoothness_mean            548 non-null float64
compactness_mean           550 non-null float64
concavity_mean             549 non-null float64
concave points_mean        549 non-null float64
symmetry_mean              549 non-null float64
fractal_dimension_mean     550 non-null float64
radius_se                  551 non-null float64
texture_se                 551 non-null float64
perimeter_se               549 non-null float64
area_se                    549 non-null float64
smoothness_se              551 non-null float64
compactness_se             551 non-null float64
concavity_se               551 non-null float64
concave points_se          549 non-null float64
symmetry_se                550 non-null float64
fractal_dimension_se       550 non-null float64
texture_worst              550 non-null float64
smoothness_worst           551 non-null float64
compactness_worst          551 non-null float64
concavity_worst            551 non-null float64
concave points_worst       551 non-null float64
symmetry_worst             551 non-null float64
fractal_dimension_worst    551 non-null float64
size_worst                 549 non-null float64
positive                   551 non-null int64
dtypes: float64(28), int64(1)
```



Machine Learning

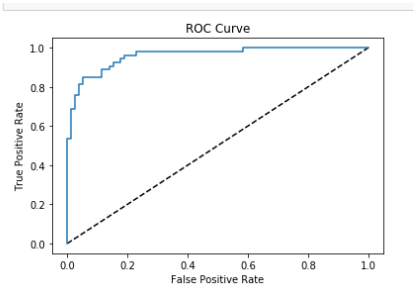
*****Before we start the machine learning algorithms, I filtered out all NA's values from the data set.

KNN was the first method to be implemented. I created a code where I was able to get each neighbor from one through eights' accuracy score using KNN. My training data consisted of 80% of the data and testing was 20%. I also stratified so the target variable was proportional through the split. The most accurate ending up being where the neighbors parameter was set to two. Below is the graph displaying test and train accuracy.



One thing to note is that my testing accuracy was stagnant until increasing the number of neighbors past six. After that is started to decline.

Logistic Regression was the next method to be utilized. I used a test size of 25% and train size of 75%. This time, I decided to normalize the data since some of my features had different ranges than others. When I implemented the logistic regression model without normalization, I found overfitting to be a problem. That is probably because logistic regression is not going to make assumptions on the distributions of my data. After implementing this, I came with a lower accuracy score that I could tune. Below is the confusion matrix, classification report, and ROC curve.

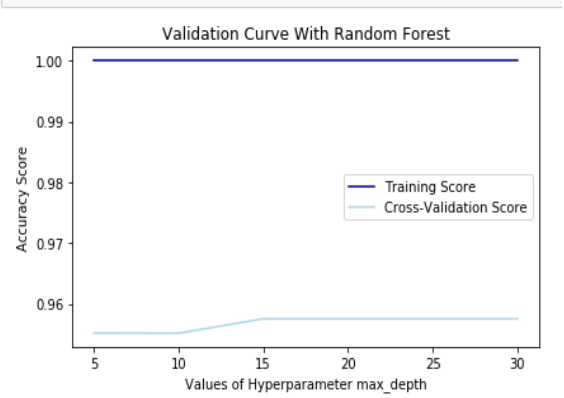


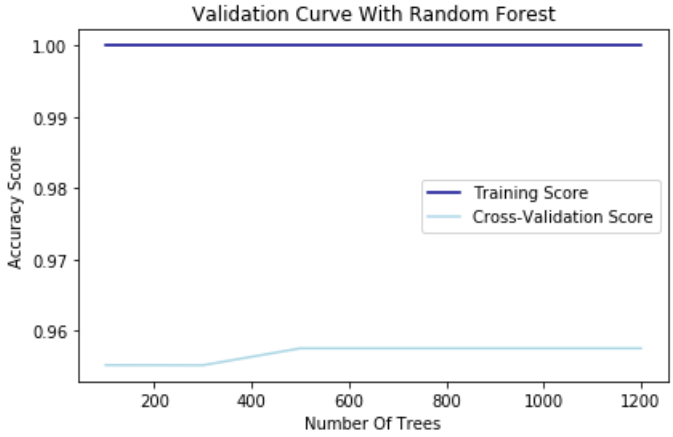

```
[[79  0]
 [54  0]]
              precision    recall  f1-score   support

           0       0.59      1.00      0.75        79
           1       0.00      0.00      0.00        54

    accuracy                           0.59       133
   macro avg       0.30      0.50      0.37       133
weighted avg       0.35      0.59      0.44       133
```

From here I used cross-fold validation (cv=5) which increased by accuracy immensely.

The next algorithm I incorporated was a *decision tree*. Undoing the normalization from the previous model was the first step. I went back to making my test size 20% and my train size 80%. I ended up coming up with 92% accuracy on the test score on the initial run. From here, I instilled different accuracy scoring methods (gini and entropy). To improve my accuracy, I did a randomizedsearchCV (cv =5) with suggestions to parameters such as max depth, criterion, max features, and min samples leaf. This contributed to a slight raise in accuracy score from 92% to 93%.

My next algorithm was the implementing *random forest*. My training data was 20% and my testing data was 80%. I came away with an accuracy score of 93%. Using GridSearchCV for tuning ended up being computationally expensive, but I was able to find the best hyper parameters. I also plotted validation curves to get a better look on how the CV score changes with different parameters. I was able to increase my accuracy. Below are the validation curves.





The final technique I used was gradient boosting. My training data was 80% and testing data was 20%. I came away with 92% accuracy off the default. Using GridSearchCV (cv = 3), I instilled two hyperparameters (learning rate, n estimators) that I wanted to see the best values for. After finding them out from the grid search, I tuned them into my model which raised my accuracy to 94%.