Using Machine Learning Algorithms to Predict Life Expectancy

Increasing the life expectancy of its citizens is a major goal that many countries strive to achieve. Knowing exactly what decisions to make in order to do this is a very important problem that many developing and developed countries face every single day. This life expectancy dataset from Kaggle can help determine if one can predict the life expectancy of citizens within a country using factors that are believed to contribute the most to longer lifespans. The dataset contains The World Health Organization's data gathered on factors influencing Life Expectancy for 193 countries from the years 2000-2015. Here is the link to the dataset.
https://www.kaggle.com/kumarajarshi/life-expectancy-who

Data Preprocessing

The life expectancy dataset originally has 22 features and 2,938 samples. The 22 features are Country, Year, Status, Life expectancy, Adult Mortality, infant deaths, Alcohol, percentage expenditure, Hepatitis B, Measles, BMI, under-five deaths, Polio, Total expenditure, Diphtheria, HIV/AIDS, GDP, Population, thinness 1-19 years, thinness 5-9 years, Income composition of resources, and Schooling. The dataset was originally a CSV file, so I converted it into a multidimensional numpy array so that it would be easier for me to perform analysis.

I decided to remove the rows containing missing values, denoted in the multidimensional numpy array by empty strings, since I believed this was the best method for dealing with missing values in an unbiased manner. After removing the rows with missing values, I was left with 1,649 samples and data from 133 countries, which was still a decent proportion of the dataset.

The Country and Status features in the dataset are categorical, so I had to convert their categorical values to integers unique to each category in that column. For the Countries column I made it so that each Country was represented by its index in another array that contained each unique Country in the dataset. The Status feature contains only two categories, Developed and Developing, which indicate if the country for this sample is either a developed or developing country. Since there are only two categories, samples that were assigned the Developed category for their Status column were given a value of 0 in that column and samples that were assigned the Developing category for their Status column were given a value of 1 in that column. After removing missing values and handling the categorical columns, I was able to convert all of the data, which were originally strings that represented numbers, to float values.

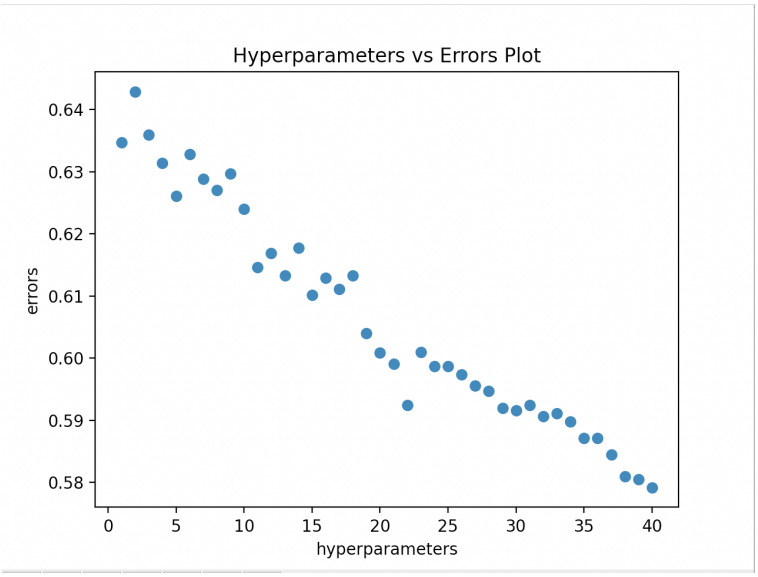
Finally, since I was planning to use the dataset to predict life expectancy, I removed the column that had the life expectancy for each sample in the dataset and then created a new one-dimensional numpy array filled with classification labels for each sample. I decided that I would have three different values for the classification labels: a value denoting high life expectancies, a value denoting around average life expectancies, and a value denoting low life expectancies. The samples with life expectancies that were greater than or equal to the 75% quartile denoted high life expectancies and would have a value of 1 in this one-dimensional numpy array. The samples with life expectancies that fell between the 25% quartile and 75% quartile denoted average life expectancies and would have a value of 0 in this one-dimensional numpy array. The samples with life expectancies that were less than or equal to the 25% quartile denoted low life expectancies and would have a value of -1 in this one-dimensional numpy array.

In the end, I returned three arrays in the function for data preprocessing: the numerically transformed data, the column names for each column in the transformed data, and a one-dimensional array with the classification labels for each sample in the transformed data. The classification algorithms I perform later on in this report aim to predict the label, which denotes a high, average, or low life expectancy, that should be assigned to a certain sample based on the features in the transformed data.

K-Nearest Neighbors Algorithm

I began testing the K-Nearest Neighbors Algorithm on the dataset with training/validation/testing. I ensured that the training/validation/testing sets each contained approximately 1/3 of the samples in the dataset and approximately equal shares of the positive samples (samples with the label 1), zero samples (samples with the label 0), and negative samples (samples with the label -1). I tried tuning the hyperparameter (k-value) by testing the algorithm with k-values 1 to 40(floor of the square root of the amount of samples in the data) to see which k-value produced the lowest error on the validation set and then its final resultant error on the test set. Instructions on how to execute the code for this can be seen in instructions.txt. Below on the left side is a screenshot of the output which shows the error produced on the validation set for each k-value and the best k-value that yielded the lowest error on the validation set and the error it produced on the test set. The image below on the right side shows a graph of each k-value I tested and its error on the validation set. Instructions on how to create this graph can also be seen in instructions.txt.

```
k=1, validation set error=0.6346776859504132
k=2, validation set error=0.6428892561983471
k=3, validation set error=0.6359636363636364
k=4, validation set error=0.6314479338842975
k=5, validation set error=0.6261421487603306
k=6, validation set error=0.6327867768595041
k=7, validation set error=0.6288264462809917
k=8, validation set error=0.6270049586776859
k=9, validation set error=0.6297223140495868
k=10, validation set error=0.6239669421487604
k=11, validation set error=0.614601652892562
k=12, validation set error=0.6168628099173554
k=13, validation set error=0.6133355371900826
k=14, validation set error=0.6177487603305785
k=15, validation set error=0.6102148760330578
k=16, validation set error=0.6128859504132231
k=17, validation set error=0.6111107438016529
k=18, validation set error=0.6133289256198348
k=19, validation set error=0.604
k=20, validation set error=0.6008859504132231
k=21, validation set error=0.5991074380165289
k=22, validation set error=0.5924561983471074
k=23, validation set error=0.6009355371900826
k=24, validation set error=0.598704132231405
k=25, validation set error=0.598694214876033
k=26, validation set error=0.5973619834710744
k=27, validation set error=0.5955834710743801
k=28, validation set error=0.594700826446281
k=29, validation set error=0.5920132231404959
k=30, validation set error=0.5915735537190082
k=31, validation set error=0.5924429752066116
k=32, validation set error=0.5906479338842975
k=33, validation set error=0.5910942148760331
k=34, validation set error=0.5897785123966942
k=35, validation set error=0.5871239669421487
k=36, validation set error=0.5871636363636363
k=37, validation set error=0.5845057851239669
k=38, validation set error=0.5809586776859504
k=39, validation set error=0.5805289256198347
k=40, validation set error=0.5792033057851239
best_k=40
final best_k test set error=0.5522050322627395
```

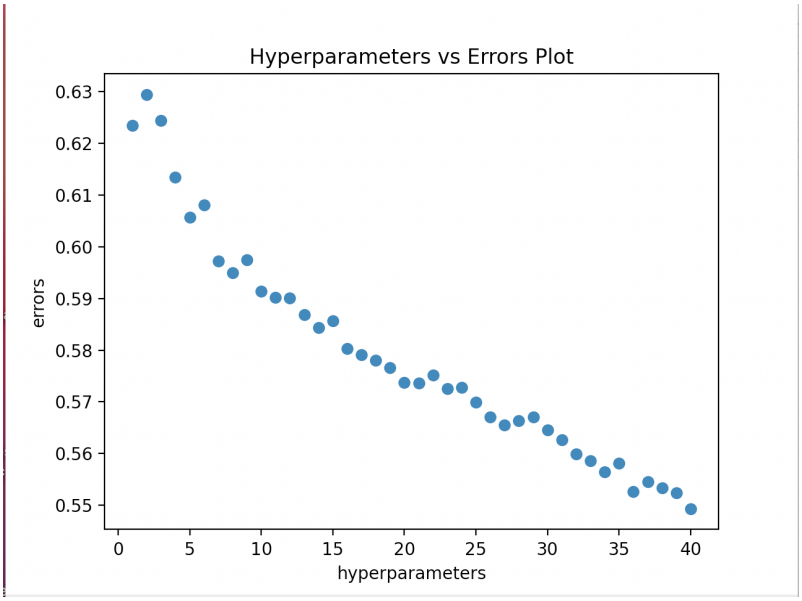


Generally, an elbow in the "Hyperparameters vs Errors Plot" on the right indicates an optimal value of k. However, there's no clear elbow in the graph since the errors tend to decrease at a more linear rate as the k-values increase so it's difficult to pick an optimal k-value with information from the graph.

I then started testing the K-Nearest Neighbors Algorithm on the dataset with bootstrapping. I decided to carry out bootstrapping with $\beta = 50$ so that I could get a good perspective of the overall error in case the errors produced for each iteration of the bootstrapping algorithm have a lot of variance. Also, for each iteration in the bootstrapping algorithm from $\beta = 1$ to 50, I made the training sets include around 25% of the samples in the data to allow for more generalization. Unlike training/validation/testing, the samples in the training sets I used for bootstrapping were picked randomly among the total samples so it isn't guaranteed that there will be an equal share of positive, zero, and negative samples in the training sets. To tune the hyperparameter (k-value), I carried out the bootstrapping algorithm with k-values 1 to 40 (floor of the square root of the amount of samples in the data) to see which k-value produced the lowest average error on all 50 of its bootstrapping iterations. Instructions on how to execute the code for this can be seen in instructions.txt. Below on the left side is a screenshot of the output which shows the average error of all 50 bootstrapping iterations for each k-value and the best k-value that yielded the lowest average error for all 50 of its bootstrapping iterations. The best k-value that I got or the k-value that produces the lowest average error for all 50 of its bootstrapping iterations

tends to change a lot but it still stays in the above 35 range. The image below on the right side corresponds to the data from the screenshot on the left side and shows a graph of each k-value I tested along with its average error for all of its 50 bootstrapping iterations. Instructions on how to create this graph can also be seen in instructions.txt.

```
k=1, avg error of all bootstraps=[0.62345659]
k=2, avg error of all bootstraps=[0.62941238]
k=3, avg error of all bootstraps=[0.62437099]
k=4, avg error of all bootstraps=[0.61341699]
k=5, avg error of all bootstraps=[0.60566541]
k=6, avg error of all bootstraps=[0.60809052]
k=7, avg error of all bootstraps=[0.59720062]
k=8, avg error of all bootstraps=[0.59498178]
k=9, avg error of all bootstraps=[0.59743495]
k=10, avg error of all bootstraps=[0.59140481]
k=11, avg error of all bootstraps=[0.59018227]
k=12, avg error of all bootstraps=[0.59004857]
k=13, avg error of all bootstraps=[0.5868076]
k=14, avg error of all bootstraps=[0.5843947]
k=15, avg error of all bootstraps=[0.5856879]
k=16, avg error of all bootstraps=[0.58030375]
k=17, avg error of all bootstraps=[0.57909334]
k=18, avg error of all bootstraps=[0.57807866]
k=19, avg error of all bootstraps=[0.57660487]
k=20, avg error of all bootstraps=[0.57377324]
k=21, avg error of all bootstraps=[0.57363413]
k=22, avg error of all bootstraps=[0.57516579]
k=23, avg error of all bootstraps=[0.57260202]
k=24, avg error of all bootstraps=[0.57282674]
k=25, avg error of all bootstraps=[0.5699825]
k=26, avg error of all bootstraps=[0.56711816]
k=27, avg error of all bootstraps=[0.56552594]
k=28, avg error of all bootstraps=[0.56642408]
k=29, avg error of all bootstraps=[0.56705389]
k=30, avg error of all bootstraps=[0.56455072]
k=31, avg error of all bootstraps=[0.56269855]
k=32, avg error of all bootstraps=[0.55996871]
k=33, avg error of all bootstraps=[0.55859136]
k=34, avg error of all bootstraps=[0.55643154]
k=35, avg error of all bootstraps=[0.55812383]
k=36, avg error of all bootstraps=[0.55270257]
k=37, avg error of all bootstraps=[0.55453571]
k=38, avg error of all bootstraps=[0.55341004]
k=39, avg error of all bootstraps=[0.5524167]
k=40, avg error of all bootstraps=[0.54935224]
best_k=40
```



Although it still isn't too clear, there is somewhat of an elbow at around k = 10 in the "Hyperparameters vs Errors Plot" on the right since the approximate slope of the points is different before and after k = 10. Still, the errors decrease a substantial amount after k = 10. So this execution of bootstrapping indicates that k = 10 may be an optimal value for k but it's still difficult to confirm that since there is a large decrease that can't be ignored in the errors shown for k-values after k = 10.

Naive-Bayes Algorithm

I started testing the Naive-Bayes Algorithm on the dataset with training/validation/testing with the same general process that I used for the K-Nearest Neighbors algorithm. The only differences were the models I was creating and the hyperparameter I was tuning. Usually, there are no hyperparameters for the Naive-Bayes algorithm. However, I wasn't given any information about the distribution of the features in the dataset. Depending on what I assume the distribution of each feature to be, there are different types of Naive-Bayes algorithms that will produce different results. The different types of Naive-Bayes algorithms that I thought made sense to utilize from the scikit-learn package are: Gaussian Naive-Bayes, Multinomial Naive-Bayes, Complement Naive-Bayes, and Bernoulli Naive-Bayes. The different distributions that each of these algorithms assumes for each feature acts as a hyperparameter in this case. I tried tuning the hyperparameter (type of distribution the Naive-Bayes algorithm assumes for each feature) by testing the different Naive-Bayes algorithms I previously listed to see which one produces the lowest error on the validation set and then its final resultant error on the test set. Instructions on how to execute the code for this can be seen in instructions.txt. Below is a screenshot of the output which shows the error produced on the validation set for each Naive-Bayes algorithm and the best Naive-Bayes algorithm that yielded the lowest error on the validation set and the error it produced on the test set.

```
Algorithm=Gaussian Naive-Bayes validation set error=0.7001851239669421
Algorithm=Multinomial Naive-Bayes validation set error=0.7166479338842975
Algorithm=Complement Naive-Bayes validation set error=0.6556396694214877
Algorithm=Bernoulli Naive-Bayes validation set error=0.686509090909091
best algorithm = Complement Naive-Bayes
final Complement Naive-Bayes test set error= 0.6651954374326831
```

Even though all the Naive-Bayes algorithms I tested generate relatively high errors, the Complement Naive-Bayes algorithm resulted in the lowest validation set error. Thus, it seems that out of the 4 different Naive-Bayes algorithms, the distribution of the features in the dataset may be closest to that which the Complement Naive-Bayes Algorithm assumes for each feature. But the error of the Complement Naive-Bayes algorithm is still extremely high for the validation set and test set so one shouldn't take this result too seriously.

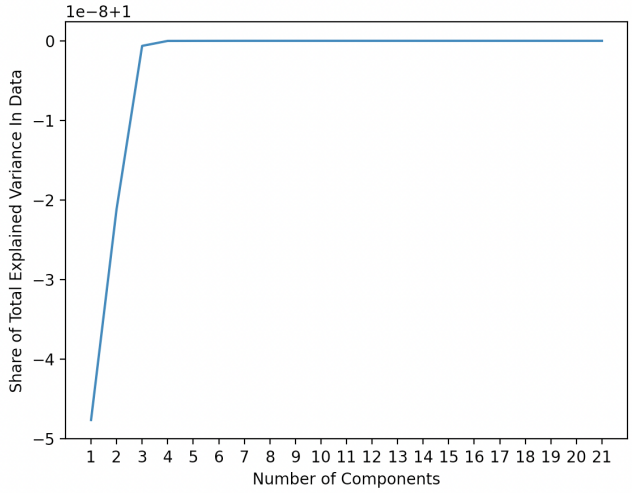
I then started testing the Naive-Bayes Algorithm on the dataset with bootstrapping with the same methods that I used for the K-Nearest Neighbors algorithm. The only differences again were the models I was creating and the hyperparameter I was tuning. So I carried out the bootstrapping algorithm with $\beta = 50$ on each of the 4 types of Naive-Bayes algorithms to see which algorithm produced the lowest average error on all its 50 bootstrapping iterations. Instructions on how to execute the code for this can be seen in instructions.txt. Below is a screenshot of the output which shows the average error of all 50 bootstrapping iterations for each Naive-Bayes algorithm and the best Naive-Bayes algorithm that yielded the lowest average error for all 50 of its bootstrapping iterations.

```
Algorithm=Gaussian Naive-Bayes, avg error of all bootstraps=[0.65135297]
Algorithm=Multinomial Naive-Bayes, avg error of all bootstraps=[0.70274408]
Algorithm=Complement Naive-Bayes, avg error of all bootstraps=[0.68857625]
Algorithm=Bernoulli Naive-Bayes, avg error of all bootstraps=[0.65241243]
best algorithm = Gaussian Naive-Bayes
```

The best algorithm that I get or the Naive-Bayes algorithm that produces the lowest average error for all 50 of its bootstrapping iterations is split fifty-fifty between the Gaussian Naive-Bayes algorithm and Bernoulli Naive-Bayes algorithm. So this bootstrapping method tends to equally choose either the Gaussian Naive-Bayes algorithm or the Bernoulli Naive-Bayes algorithm as the best algorithm. So I don't really have any significant evidence favoring one optimal algorithm and I still have to consider that the average errors of the bootstrapping iterations for all the Naive-Bayes algorithms I tested are relatively high. Thus I shouldn't take these results too seriously either.

Feature Selection

After seeing the results produced from both the K-Nearest Neighbors and Naive-Bayes classification algorithms on the dataset, I decided to investigate the results I would get when performing the same analysis on a smaller subset of the total features in the dataset. To begin the investigation I graphed the share of the total variance explained by all the different numbers of components that the dataset can produce. The images below display this graph, the x-values that were plotted for the different number of components, and the y-values that correspond to those x-values. Instructions on how to create this graph can be seen in instructions.txt.




```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]

[0.99999995 0.99999998 1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.        ]
```

The scaling of the y-values in the graph looks really weird since one feature in the data has a variance that is significantly larger than the variance of any other feature in the dataset. As one can probably notice from the image directly below the graph, the share of total explained variance for one component itself is 99.999995%. Technically, only when the Number of Components is equal to 21, should the share of the total explained variance be equal to 1 since I am taking every single feature into account in that case. Although most of the shares of total explained variance for x-values greater than 1 aren't exactly equal to 1, they are too close to 1 to be deemed any different by python. Anyways, according to this graph and output, one component explains almost all of the variance in the data. After investigating the features more closely, I found out that this component was the "Population" feature. It makes sense why almost all the variation in the dataset was influenced by this feature since the difference in population between certain countries can be extremely large. The minimum population of a country in the data is 34 and the maximum population of a country in the data is 1,293,859,294. Since dimensionality reduction methods like PCA transform datasets by maximizing variance independent of information about the class labels, if one feature's variance is significantly larger than the variance for the other features as shown here, that feature can account for an extremely large amount of explained variance in the dataset. Still, let's see if reducing the data down to just the Population feature will improve classification performance with training/validation/testing. Instructions on how to execute the code for this can be seen in instructions.txt. The images below show the output produced by the training/validation/testing methods for both the K-Nearest Neighbors and Naive-Bayes algorithm on the reduced version of the dataset that contains only the Population feature.

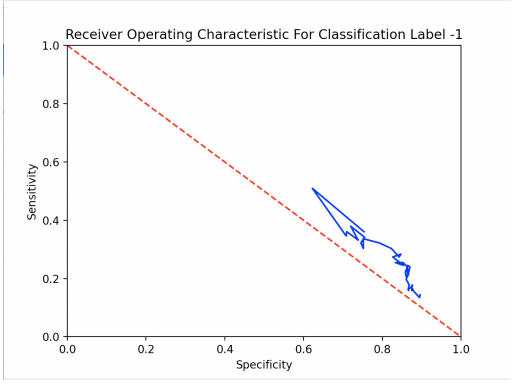
```
k=1, validation set error=0.6342644628099173
k=2, validation set error=0.6473388429752066
k=3, validation set error=0.6341785123966942
k=4, validation set error=0.6327867768595041
k=5, validation set error=0.6279504132231405
k=6, validation set error=0.6350280991735537
k=7, validation set error=0.6288826446280992
k=8, validation set error=0.6292528925619835
k=9, validation set error=0.6261785123966942
k=10, validation set error=0.6213157024793389
k=11, validation set error=0.613295867768595
k=12, validation set error=0.6155570247933885
k=13, validation set error=0.6129123966942148
k=14, validation set error=0.6208892561983471
k=15, validation set error=0.6120231404958678
k=16, validation set error=0.6133619834710744
k=17, validation set error=0.6120330578512396
k=18, validation set error=0.6138016528925619
k=19, validation set error=0.6075768595041322
k=20, validation set error=0.6044595041322314
k=21, validation set error=0.6053553719008264
k=22, validation set error=0.6000495867768595
k=23, validation set error=0.6049553719008265
k=24, validation set error=0.6004892561983471
k=25, validation set error=0.6004826446280992
k=26, validation set error=0.5987074380165289
k=27, validation set error=0.5960396694214876
k=28, validation set error=0.5902380165289256
k=29, validation set error=0.5880066115702479
k=30, validation set error=0.5871107438016528
k=31, validation set error=0.5888727272727273
k=32, validation set error=0.5884165289256198
k=33, validation set error=0.5897553719008265
k=34, validation set error=0.5902280991735537
k=35, validation set error=0.5880198347107438
k=36, validation set error=0.5885090909090909
k=37, validation set error=0.5862909090909091
k=38, validation set error=0.5809619834710744
k=39, validation set error=0.5805289256198347
k=40, validation set error=0.5792066115702479
best_k=40
final best_k test set error=0.5530877698031298
```

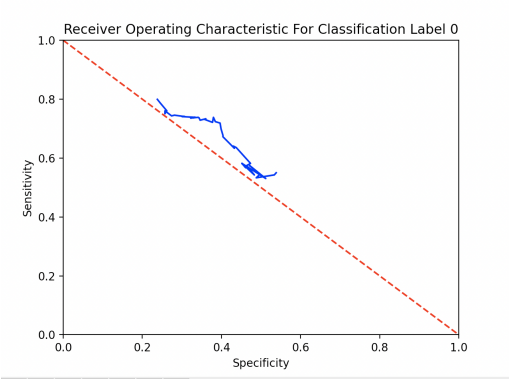
```
Algorithm=Gaussian Naive-Bayes validation set error=0.6865851239669422
Algorithm=Multinomial Naive-Bayes validation set error=0.5036363636363637
Algorithm=Complement Naive-Bayes validation set error=0.7472727272727273
Algorithm=Bernoulli Naive-Bayes validation set error=0.5036363636363637
best algorithm = Multinomial Naive-Bayes
final Multinomial Naive-Bayes test set error= 0.5045372050816697
```

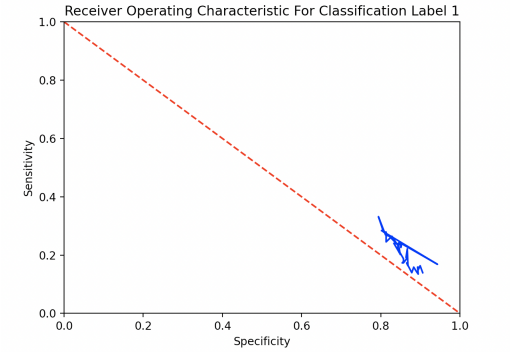
For both K-Nearest Neighbors and Naive-Bayes, the final error produced on the test set by the optimal hyperparameter was slightly better than what I produced before when executing these algorithms with the entire dataset. Originally for training/validation/testing of the complete dataset I produced a final error of 0.5522 on the test set for k=40 in the K-Nearest Neighbors algorithm and a final error of 0.6652 on the test set for the Complement Naive-Bayes algorithm. Now for training/validation/testing of the reduced dataset I am producing a final error of 0.5531 on the test set for k = 40 in the in the K-Nearest Neighbors algorithm and a final error of 0.5045 on the test set for the Multinomial Naive-Bayes algorithm. So reducing the dataset down to just the Population feature actually produced an almost similar error for the K-Nearest Neighbors algorithm and a much smaller error for the Naive-Bayes algorithm, at least in the training/validation/testing cross-validation method. This signals that taking just the Population feature into account in classification may produce similar or sometimes even better results than taking the whole dataset into account during classification with the K-Nearest Neighbors algorithm and the Naive-Bayes algorithm.

ROC Curves

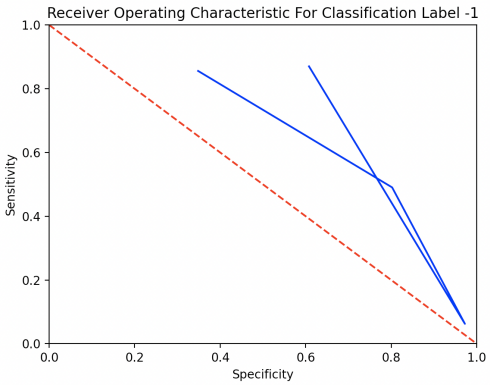
Since my classification system isn't binary, the process through which I generated the ROC curves is a little bit more complex. Instead of having just one graph with two axes indicating the Specificity and Sensitivity, I created three ROC curve plots, one for each type of classifier in my classification system. The different classifiers in this classification system are -1, 0, and 1. As a result, I had one graph indicating the specificities and sensitivities that corresponds to classifier -1, one graph indicating the specificities and sensitivities that corresponds to classifier 0, and one graph indicating the specificities and sensitivities that corresponds to classifier 1. Below are images showing an ROC curve for the K-Nearest Neighbors algorithm with various specificities and sensitivities obtained from executing the algorithm with k-values ranging from 1 to 40(floor of the square root of the amount of samples in the data) for each classifier in my classification system. Instructions on how to create these graphs can be seen in instructions.txt.

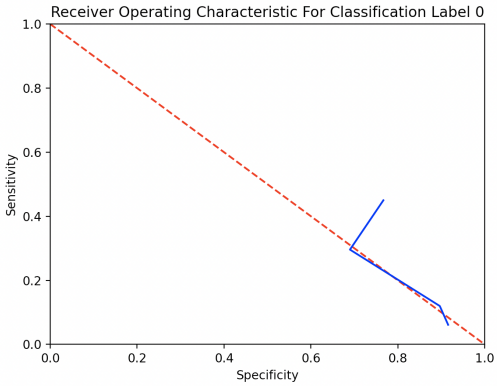


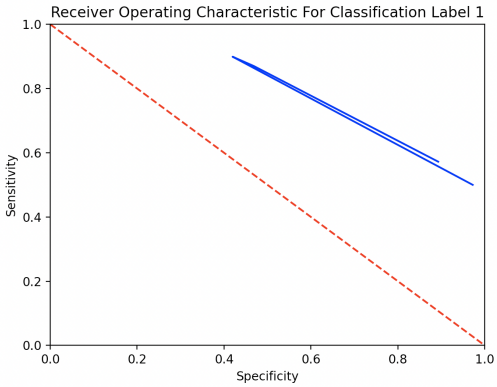




The reason why the ROC curves I produced don't span over the entire graph is because I only graphed sensitivities and specificities generated by values of k ranging from 1 to 40. If I graphed the sensitivities and specificities generated by many more k-values then the curve probably would have covered a much larger portion of the graph. In general for ROC curves, if the area under the curve is close to 0.5 then the classification algorithm is considered poor for that classifier and if the area under the curve is close to 1 then the classification algorithm is considered really good for that classifier. Since all the ROC curves that I generated above are relatively close to the red line, whose area under the curve is exactly 0.5, the K-Nearest Neighbors algorithm is a poor classification algorithm for all the classifiers(-1, 0, 1). I basically carried out the same process to generate the ROC curves for the Naive-Bayes algorithm. However, the hyperparameter that I was changing for the Naive-Bayes algorithm was the prior distributions that the algorithm assumes for each feature in the data. Below are images showing the ROC curves for the Naive-Bayes algorithm with various specificities and sensitivities obtained from the different types of Naive-Bayes algorithms that assume different distributions for the features of the data. Instructions on how to create these graphs can be seen in instructions.txt.







The reason why it doesn't look like many points were plotted is because I only tested 4 specific hyperparameters for the Naive-Bayes algorithm since there were only 4 different types of Naive-Bayes algorithms that I thought made sense to utilize from the scikit-learn package. As one can obviously see from the curves shown above, the ROC curves for classification label 1 and classification label -1 maintain a decent amount of distance above the red dotted line but the ROC curve for classification label 0 doesn't maintain much distance above the red dotted line at all. So the ROC curves show that the Naive-Bayes algorithm performs okay for classification labels 1 and -1, but performs poorly for classification label 0. Overall, the ROC curves still show that the Naive-Bayes algorithm performs much better than the K-Nearest Neighbors algorithm as the ROC curves for the Naive-Bayes algorithm are farther above the red dotted line for the most part.