Heart Failure Classifier

Data Minor

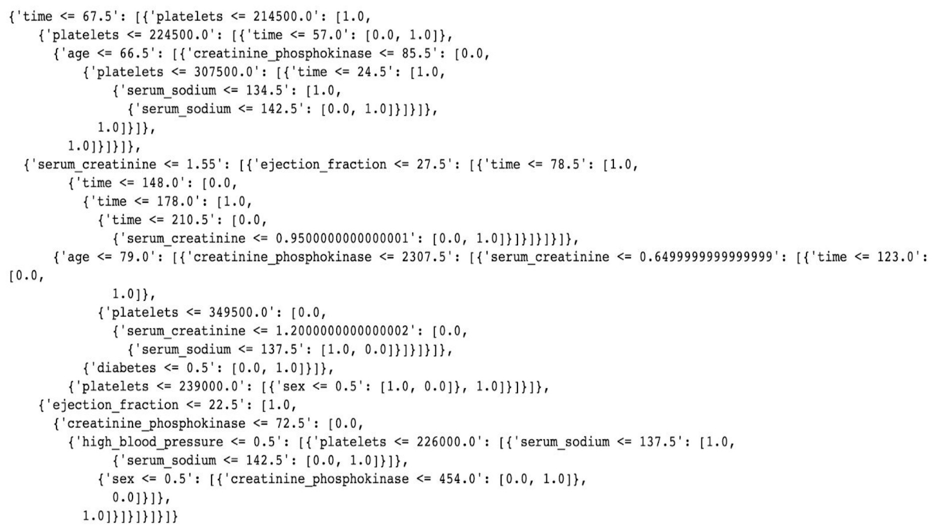
Members:
Sang Choi
Jose Cruz

Abstract:

Our group used the Heart Failure dataset from Kaggle to build three different types of classifiers. We built a K-nearest Neighbor classifier, a Naive Bayes classifier, and a Decision Tree classifier and compared them to Sci-kit learn's library of classifiers. Our algorithms follow the basic design of the original algorithms to achieve the desired classification when predicting unseen data. We found that for the most part, our built algorithms matched or we close in accuracy to the algorithms provided by Sci-kit learn. The biggest difference we saw was in the high execution time to learn and predict unseen data from our own built algorithms. This could be attributed to the lack of parallelization in our native implementation and inefficiency.

Design:

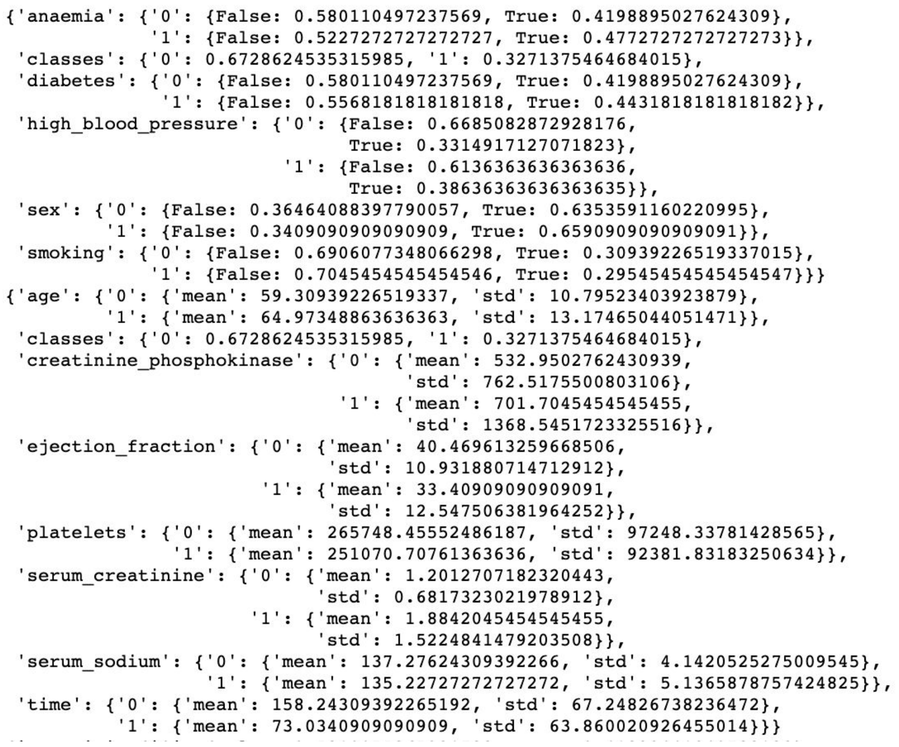
The project was designed to compare the performance of our own built algorithms with those provided by Scikit learn's implementation. Each algorithm was built following the original designs intended for their purpose of classifying datasets by training a model and seeing how well it can learn and test it against unseen data.



```
{'time <= 67.5': [{'platelets <= 214500.0': [1.0,
    {'platelets <= 224500.0': [{'time <= 57.0': [0.0, 1.0]},
      {'age <= 66.5': [{'creatinine_phosphokinase <= 85.5': [0.0,
          {'platelets <= 307500.0': [{'time <= 24.5': [1.0,
              {'serum_sodium <= 134.5': [1.0,
                {'serum_sodium <= 142.5': [0.0, 1.0]}]}]},
            1.0]}]},
        1.0]}]}]},
  {'serum_creatinine <= 1.55': [{'ejection_fraction <= 27.5': [{'time <= 78.5': [1.0,
        {'time <= 148.0': [0.0,
          {'time <= 178.0': [1.0,
            {'time <= 210.5': [0.0,
              {'serum_creatinine <= 0.9500000000000001': [0.0, 1.0]}]}]}]}]},
      {'age <= 79.0': [{'creatinine_phosphokinase <= 2307.5': [{'serum_creatinine <= 0.6499999999999999': [{'time <= 123.0':
[0.0,
              1.0]},
            {'platelets <= 349500.0': [0.0,
              {'serum_creatinine <= 1.2000000000000002': [0.0,
                {'serum_sodium <= 137.5': [1.0, 0.0]}]}]}]},
          {'diabetes <= 0.5': [0.0, 1.0]}]},
        {'platelets <= 239000.0': [{'sex <= 0.5': [1.0, 0.0]}, 1.0]}]}]},
    {'ejection_fraction <= 22.5': [1.0,
      {'creatinine_phosphokinase <= 72.5': [0.0,
        {'high_blood_pressure <= 0.5': [{'platelets <= 226000.0': [{'serum_sodium <= 137.5': [1.0,
              {'serum_sodium <= 142.5': [0.0, 1.0]}]}]},
            {'sex <= 0.5': [{'creatinine_phosphokinase <= 454.0': [0.0, 1.0]},
              0.0]}]},
          1.0]}]}]}]}]}
```

Figure 1: Example of Decision Tree

For the decision tree, we first select the best attribute to split the data by calculating its entropy. Next, we make that attribute a decision node and break down the dataset into smaller subtrees. This process is repeated as such building a tree. The process terminates when tuples in the same node reach a purity of 1, or when there are no more remaining attributes, or there are no more instances. This can be further limited by the minimum number of tuples per leaf or the max depth in a node. Our implementation stores the tree in a python dictionary object as seen in Figure 1 with key being the split condition.



```
{'anaemia': {'0': {False: 0.580110497237569, True: 0.4198895027624309},
             '1': {False: 0.5227272727272727, True: 0.4772727272727273}},
 'classes': {'0': 0.6728624535315985, '1': 0.3271375464684015},
 'diabetes': {'0': {False: 0.580110497237569, True: 0.4198895027624309},
              '1': {False: 0.5568181818181818, True: 0.4431818181818182}},
 'high_blood_pressure': {'0': {False: 0.6685082872928176,
                               True: 0.3314917127071823},
                         '1': {False: 0.6136363636363636,
                               True: 0.38636363636363635}},
 'sex': {'0': {False: 0.36464088397790057, True: 0.6353591160220995},
         '1': {False: 0.3409090909090909, True: 0.6590909090909091}},
 'smoking': {'0': {False: 0.6906077348066298, True: 0.30939226519337015},
             '1': {False: 0.7045454545454546, True: 0.29545454545454547}}}
{'age': {'0': {'mean': 59.30939226519337, 'std': 10.79523403923879},
         '1': {'mean': 64.97348863636363, 'std': 13.17465044051471}},
 'classes': {'0': 0.6728624535315985, '1': 0.3271375464684015},
 'creatinine_phosphokinase': {'0': {'mean': 532.9502762430939,
                                    'std': 762.5175500803106},
                              '1': {'mean': 701.7045454545455,
                                    'std': 1368.5451723325516}},
 'ejection_fraction': {'0': {'mean': 40.469613259668506,
                             'std': 10.931880714712912},
                       '1': {'mean': 33.40909090909091,
                             'std': 12.547506381964252}},
 'platelets': {'0': {'mean': 265748.45552486187, 'std': 97248.33781428565},
               '1': {'mean': 251070.70761363636, 'std': 92381.83183250634}},
 'serum_creatinine': {'0': {'mean': 1.2012707182320443,
                            'std': 0.6817323021978912},
                      '1': {'mean': 1.8842045454545455,
                            'std': 1.5224841479203508}},
 'serum_sodium': {'0': {'mean': 137.27624309392266, 'std': 4.1420525275009545},
                  '1': {'mean': 135.22727272727272, 'std': 5.1365878757424825}},
 'time': {'0': {'mean': 158.24309392265192, 'std': 67.24826738236472},
          '1': {'mean': 73.0340909090909, 'std': 63.860020926455014}}}
```

Figure 2: Naive Bayes Likelihood Table

For the Naive Bayes algorithm, we created a dictionary of the dataset to calculate the apriori probabilities. From there we can classify data by means and standard deviations for numerical data. For categorical data, a class distribution was used with corresponding probabilities. Our implementation builds a likelihood table as seen in Figure 2, which is used to calculate posterior probability for each class. The algorithm selects whichever class that has the highest posterior probability as its prediction for the tuple.

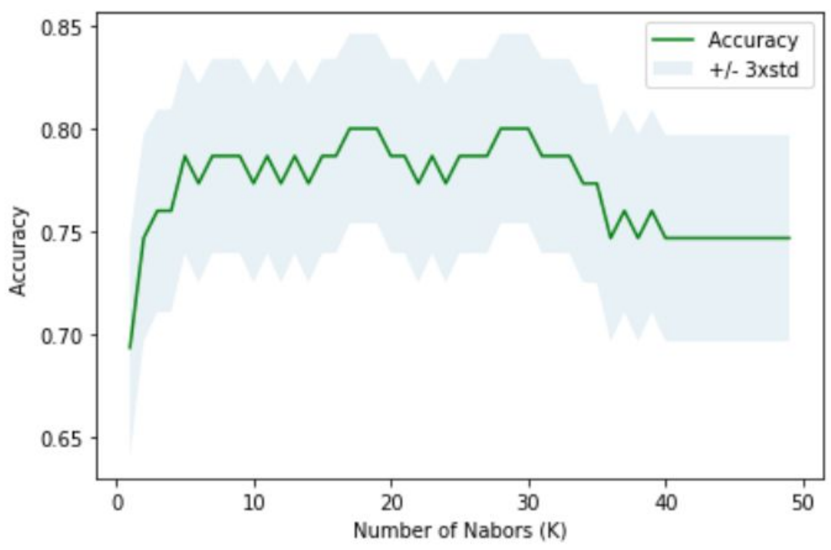


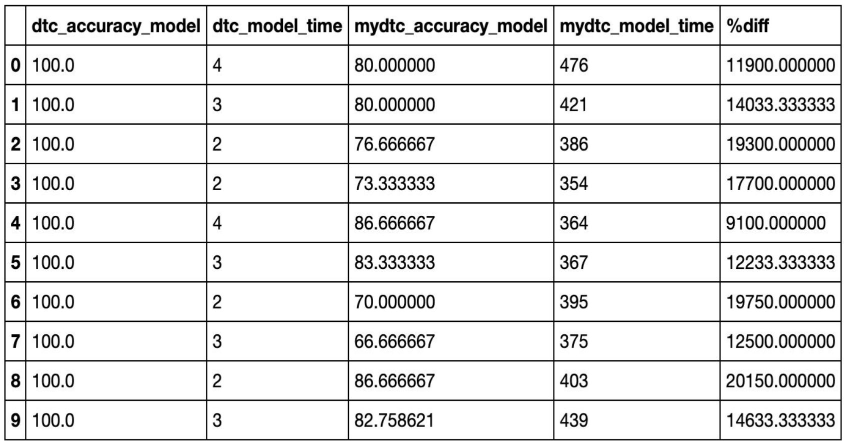
Figure 3: K selection in KNN algorithm

For K-nearest Neighbor, we clustered the nearest neighbors to a predefined k value. We implemented a loop to find the best k value by running a range and testing its accuracy to find the best fit as seen in figure 3. To find the nearest neighbors, we implemented the scaler preprocessing because the data scale was not uniform in the dataset and then used the euclidean distance algorithm to locate the nearest neighbors.

Implementation:

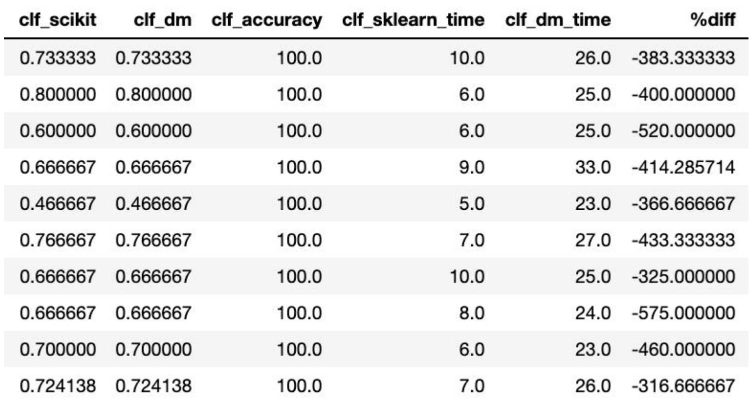
All three algorithms were implemented equally to test their performance. We split the dataset between training and testing tuples. Next, we used the k-fold splitting algorithm on the training part of the dataset. We then used a timer to test the time performance of the algorithm and we used the accuracy score to see how well the models properly classified the unseen data. This was done in a loop to ensure the same subset of the k-fold was used by the Sci-kit learn's model and our own to make direct performance comparisons. The data was then put into a data frame for record-keeping.

Testing:



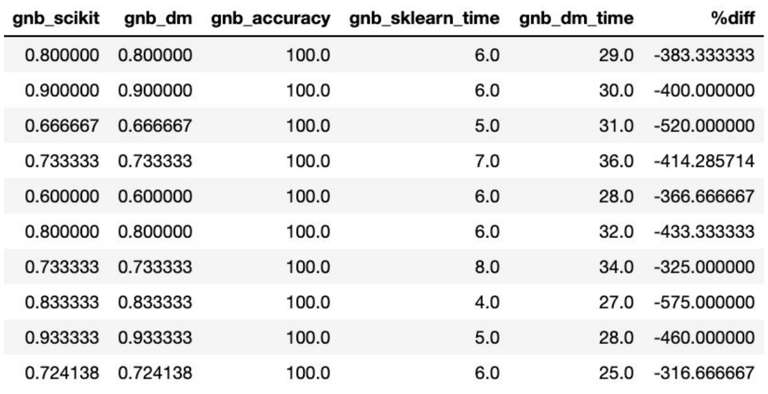
	dtc_accuracy_model	dtc_model_time	mydtc_accuracy_model	mydtc_model_time	%diff
0	100.0	4	80.000000	476	11900.000000
1	100.0	3	80.000000	421	14033.333333
2	100.0	2	76.666667	386	19300.000000
3	100.0	2	73.333333	354	17700.000000
4	100.0	4	86.666667	364	9100.000000
5	100.0	3	83.333333	367	12233.333333
6	100.0	2	70.000000	395	19750.000000
7	100.0	3	66.666667	375	12500.000000
8	100.0	2	86.666667	403	20150.000000
9	100.0	3	82.758621	439	14633.333333

Table 1: Decision Tree - Accuracy and Performance Comparison



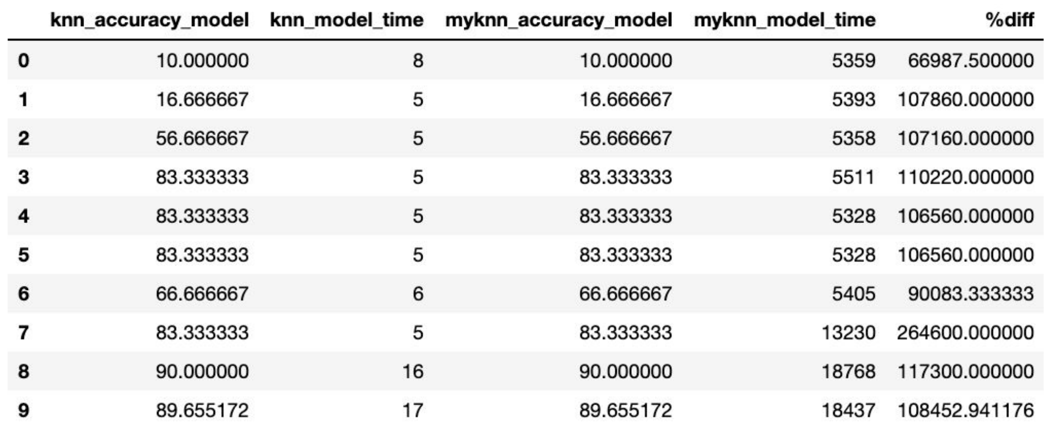
clf_scikit	clf_dm	clf_accuracy	clf_sklearn_time	clf_dm_time	%diff
0.733333	0.733333	100.0	10.0	26.0	-383.333333
0.800000	0.800000	100.0	6.0	25.0	-400.000000
0.600000	0.600000	100.0	6.0	25.0	-520.000000
0.666667	0.666667	100.0	9.0	33.0	-414.285714
0.466667	0.466667	100.0	5.0	23.0	-366.666667
0.766667	0.766667	100.0	7.0	27.0	-433.333333
0.666667	0.666667	100.0	10.0	25.0	-325.000000
0.666667	0.666667	100.0	8.0	24.0	-575.000000
0.700000	0.700000	100.0	6.0	23.0	-460.000000
0.724138	0.724138	100.0	7.0	26.0	-316.666667

Table 2: Naive Bayes(Categorical) - Accuracy and Performance Comparison



gnb_scikit	gnb_dm	gnb_accuracy	gnb_sklearn_time	gnb_dm_time	%diff
0.800000	0.800000	100.0	6.0	29.0	-383.333333
0.900000	0.900000	100.0	6.0	30.0	-400.000000
0.666667	0.666667	100.0	5.0	31.0	-520.000000
0.733333	0.733333	100.0	7.0	36.0	-414.285714
0.600000	0.600000	100.0	6.0	28.0	-366.666667
0.800000	0.800000	100.0	6.0	32.0	-433.333333
0.733333	0.733333	100.0	8.0	34.0	-325.000000
0.833333	0.833333	100.0	4.0	27.0	-575.000000
0.933333	0.933333	100.0	5.0	28.0	-460.000000
0.724138	0.724138	100.0	6.0	25.0	-316.666667

Table 3: Naive Bayes(Numerical) - Accuracy and Performance Comparison



	knn_accuracy_model	knn_model_time	myknn_accuracy_model	myknn_model_time	%diff
0	10.000000	8	10.000000	5359	66987.500000
1	16.666667	5	16.666667	5393	107860.000000
2	56.666667	5	56.666667	5358	107160.000000
3	83.333333	5	83.333333	5511	110220.000000
4	83.333333	5	83.333333	5328	106560.000000
5	83.333333	5	83.333333	5328	106560.000000
6	66.666667	6	66.666667	5405	90083.333333
7	83.333333	5	83.333333	13230	264600.000000
8	90.000000	16	90.000000	18768	117300.000000
9	89.655172	17	89.655172	18437	108452.941176

Table 4: KNN - Accuracy and Performance Comparison

As you can see from the results above, we were able to match or get close to the accuracy of the Sci-kit learn's performance. What we were not able to achieve was to reduce the time our models took to train, learn, and predict accurately the dataset. We feel that the lack of implementing parallelization within the models to perform tasks concurrently and efficiently is to blame for our results.

User Manual:

All three notebooks can be run simply by loading onto a notebook and running alongside the dataset. There are no special instructions for running them.

Future:

We could continue on this journey of increasing efficiency and to improve the performance of the algorithms. It would be great to achieve what Sci-kit learns has been able to do.

Team Contribution and comments:

It was nice to build our own models and compare to algorithms that have been professionally done. The experience of building the models from the scratch gave us a deeper understanding of developing a classifier.

Jose:

- Decision Tree classifier
- Knn classifier
- Final Report
- Progress Report
- Proposal

Sang:

- Naive Bayes Categorical classifier
- Gaussian Naive Bayes
- Presentation
- Progress Report
- Proposal

References:

F. Meng, Z. Zhang, et al. “Machine Learning Can Predict Survival of Patients with Heart Failure from Serum Creatinine and Ejection Fraction Alone.” BMC Medical Informatics and Decision Making, BioMed Central, 1 Jan. 1970, bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-020-1023-5.

Kamber, Micheline, and Jiawei Han. Data Mining: Concepts and Techniques 3rd Edition Ed. 3. Morgan Kaufmann Publishers, 2011.

Larxel. “Heart Failure Prediction.” Kaggle, 20 June 2020, www.kaggle.com/andrewmvd/heart-failure-clinical-data.