**CI512 Open Ended Project**



[[1](#_References)]

Contents

[Evidence of Understanding 3](#_Toc93631387)

[What is the functionality of your system? 3](#_Toc93631388)

[What are the basic inputs and outputs? 3](#_Toc93631389)

[What real-world problem does it solve? 3](#_Toc93631390)

[Algorithm Explanation 4](#_Toc93631391)

[Obtaining the data 4](#_Toc93631392)

[Analysis and pre-processing 4](#_Toc93631393)

[Using various models and measuring their performance 6](#_Toc93631394)

[Short-listing promising models and fine-tuning the system 7](#_Toc93631395)

[Evaluation and results 10](#_Toc93631396)

[References 11](#_Toc93631397)

# Evidence of Understanding

## What is the functionality of your system?

The sole functionality of my system of algorithms/models is to accurately predict whether a person has heart disease or not. This is based on various types of data collected from hospitals.

# What are the basic inputs and outputs?

My chosen dataset is comprised of 5 separate heart datasets from various locations across the world, making it the largest publicly-available dataset for researching heart disease. It contains 12 types/columns of clinical data from 918 different patients **[**[**2**](#_References)**]**, including ‘HeartDisease’ which is this dataset’s target. I.e., the deciding factor/column of whether a patient has heart disease or not.

**Clinical data types:**

|  |  |
| --- | --- |
| * Age * Sex * Serum cholesterol * Chest pain type * Maximum heartrate achieved * Resting blood pressure | * Fasting blood sugar * Resting electrocardiogram results * Exercise-induced angina * Oldpeak (numerical ST equivalent) * Slope of the peak exercise ST segment * HeartDisease (target) |

These 12 columns are the basic inputs of the system, since each model utilizes them in order to form conclusions. This is done by splitting the data randomly into two separate datasets, one for training the models and the other for testing the models. This will be talked about further in the [Analysis and preprocessing](#_Analysis_and_preprocessing) section in the algorithm explanation. Once the predictions are evaluated, the result (output of the system) is a printed statement in a terminal, giving an accuracy and precision score ranging from 0 to 1 for each algorithm.

# What real-world problem does it solve?

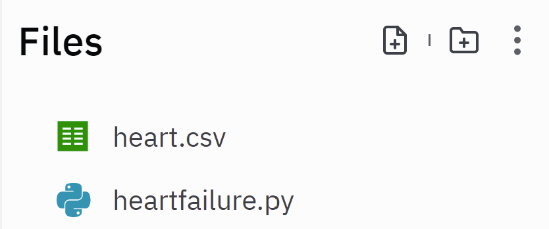
Ideally, the objective is to get the percentage of correct predictions to be as high as possible, as lives could potentially be at risk from an incorrect one. This is due to the real-world usage, which (if the accuracy was almost 100%) would be to implement the system in hospitals across the globe to aid in early detections of heart disease. Implementing this would ultimately solve the problem of doctors needing to analyze extrapolated data from hundreds of tests for many hours in order to come to a conclusion. It solves this problem because the system of algorithms can analyze the data and come to conclusions of similar accuracy in a matter of seconds/minutes. In other words, machine learning techniques are useful not-only for faster detections of heart disease, but the automation of this itself removes the need for hours or even days of manual analysis from healthcare professionals.

# Algorithm Explanation

This section will cover how I obtained the heart disease dataset, prepared and cleaned it, and trained a wide variety of classification models to understand their performances on the dataset. This is so that I could shortlist the highest-performing models, and fine-tune them.

## Obtaining the data

First, I had to get a fairly large dataset with a real-world use, that I could train various algorithms on. I was recommended a great website called ‘Kaggle’, which has a massive online community of machine-learning enthusiasts, and an enormous collection of free-to-use datasets. After a long period of searching, I stumbled upon the ‘Heart Failure Prediction Dataset’, which had over 450 code examples and a few discussions. Once I ensured that this was a viable option, I downloaded the dataset as a ‘.csv’ file, and uploaded it to my workspace on ‘Replit’. This site is a browser-based IDE that allows users to run python projects in a console or shell terminal. I also created a new python file for the project.



## Analysis and pre-processing

Once I imported the dataset and created the python file, I first had to read the ‘.csv’ file using ‘pandas’, which is python’s standard library for data analysis and manipulation **[**[**3**](#_References_2)**]**. I then had to analyze and clean/preprocess the data so that the models can interpret and use all attributes provided. In order to do this, I had to check whether the target is balanced or unbalanced. This is because algorithms that are trained on unbalanced datasets tend to have poorer accuracy when classifying unseen observations **[**[**4**](#_References)**]**. I checked this using the following code:

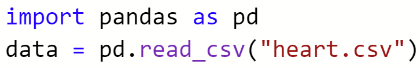


Figure 1 - Importing pandas to read the csv file:

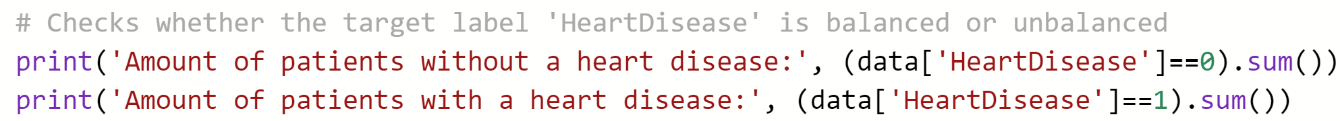


Figure 2 - Input:

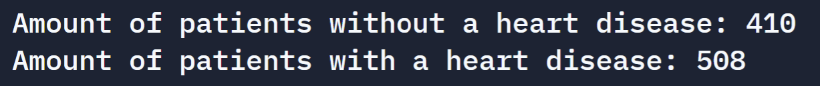


Figure 3 - Output:

The difference in observations is not significant, therefore we can assume that the target variable is balanced and will not affect algorithm performance.

Next, I had to figure out which columns/attributes were non-numerical, so that I could convert them into numerical data. This is necessary because each algorithm can only interpret numerical data. I filtered this using the following code:

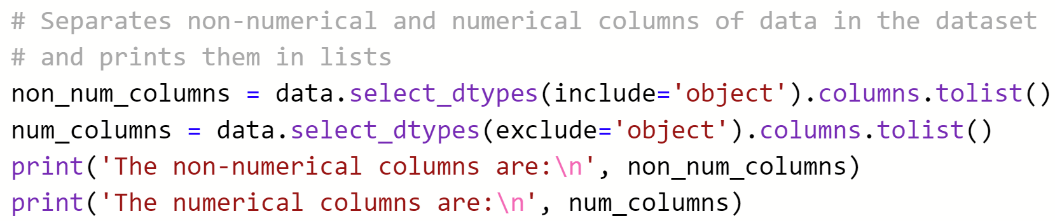


Figure 4 - Input:

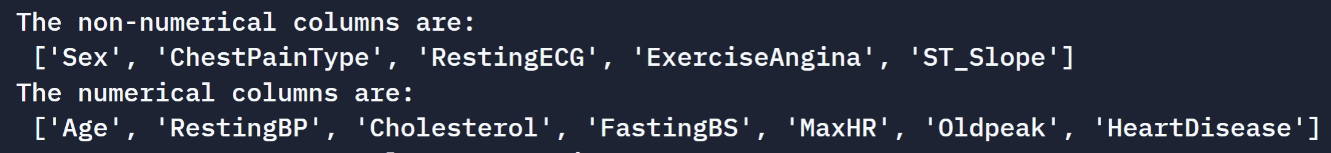


Figure 5 - Output:

Once this was done, I converted the non-numerical data to numerical using the ‘.replace()’ method:

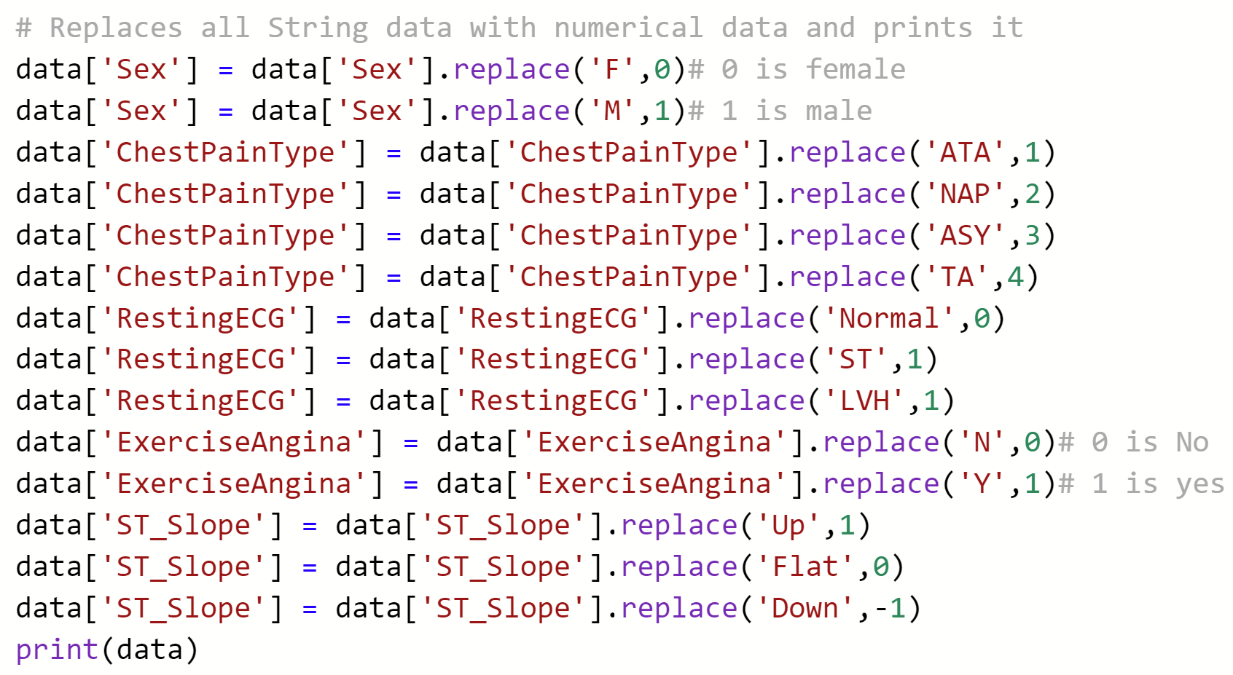


Figure 6 – Pre-processing the data:

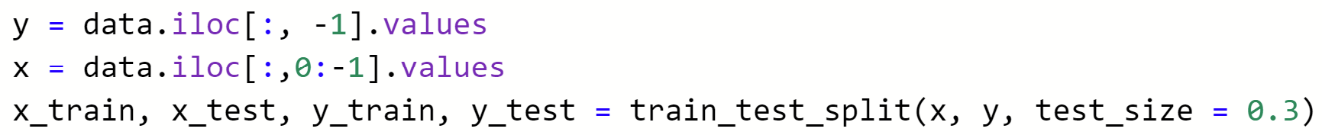
After this, as stated previously, the data is divided into two subsets; one for training and the other for testing the algorithms, using the ‘train\_test\_split’ procedure. One of the parameters in this procedure is ’test\_size’, which I specify as 0.3. This allocates 30% of the data split for testing and the other 70% for training.

Figure - Applying the train/test split:

## Using various models and measuring their performance

Once the dataset was cleaned and the train\_test\_split was specified, this allowed me to quickly evaluate pre-existing models that we have used in the labs. This included:

* KNeighborsClassifier
* DecisionTreeClassifier
* RandomForestClassifier
* MLPClassifier

I also independently researched a model called ‘SVC’ (Support Vector Classifier) which is based on SVM (Support Vector Machine). This is a classification algorithm that commonly performs well in datasets with relatively few samples **[**[**5**](#_References_2)**]**. Thus, I chose to include it in the first evaluation.

I imported each classifier from its specific library, and stored them in separate variables. I then used the ‘.fit()’ method to adjust the classifiers’ weights to best fit the training data given in each split. This then allows class predictions to be made from unseen data instances in the testing data, using the ‘.predict()’ method **[**[**6**](#_References_2)**]**. Finally, results are obtained using the metrics ‘accuracy\_score’ and ‘precision\_score’. Although these metrics are somewhat related, accuracy refers to the ratio of correctly predicted classes relative to all classes, whereas precision refers to the ratio of correctly predicted positive classes to the total predicted positive classes **[**[**7**](#_References)**]**. These metrics give a decimal score representing the percentage of correctly predicted classes. Below is an example of this, using DecisionTreeClassifier.

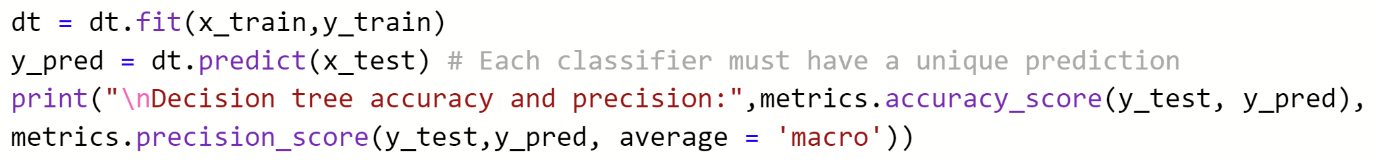


Figure - Input:

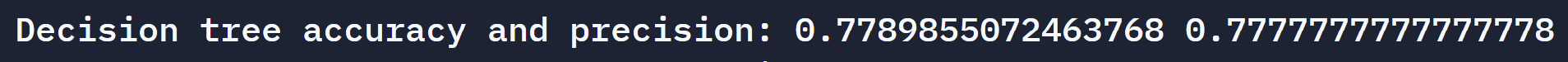


Figure - Output:

Since I had no clue as to how these models would perform on the heart dataset, I decided to first test them all using their default hyperparameters. This is done by simply leaving the models parentheses blank before the learning process commences, as seen in the first image in Figure 8. In order to get a more accurate depiction of each model’s accuracy and precision, I ran them 5 times each, recorded the results, and calculated the mean accuracy and precision.

From these results, it is obvious that MLP and RandomForest are the best-performing models, with a mean accuracy and precision of over **84%**. It is also worth noting that in this dataset, the differentiation between these scores is negligible, as DecisionTree and SVC’s mean scores were equal, and the other classifiers’ mean precision scores were only slightly higher than their mean accuracies. Since all models achieved a relatively high score with default parameters, we can assume that the dataset was thoroughly processed and sorted before it was published.

## Short-listing promising models and fine-tuning the system

Based on the results from the first set of tests, I made the decision to shortlist the RandomForest and MLP classifiers due to their astounding performance. One technique for boosting performance is the usage of ensemble models. These combine separate models and account for each of their predictions, often resulting in higher overall performance, once combined. Common examples of ensemble models that apply this technique include ‘VotingClassifier’ and ‘StackingClassifier’, as they require ‘estimators’ i.e., models to use as predictors. After further research, I decided that ‘GradientBoostingClassifier’ would also be a good advocate, as this model applies a similar strategy to RandomForest by combining multiple weaker prediction-models, typically decision trees. However, it does this in a forward stage-wise fashion, allowing the arbitrary differentiable loss functions of the weaker models to be optimized **[**[**8**](#_References)**]**. This often leads to it performing better than RandomForest, so I used it along with the other ensemble models.

Furthermore, in an attempt to further boost performance, I tweaked the hyperparameters of a few of the models. For example, GradientBoostingClassifier contains one called ‘n\_estimators’, which represents the number of boosting stages to perform **[**[**9**](#_References)**]**. According to the scikit-learn page, the default value of this parameter is 100, and increasing it usually results in better performance. Thus, I changed this by adding ‘n\_estimators=200’ inside the parentheses of the classifier. The changes made to each model’s hyperparameters can be seen in Figure 10.

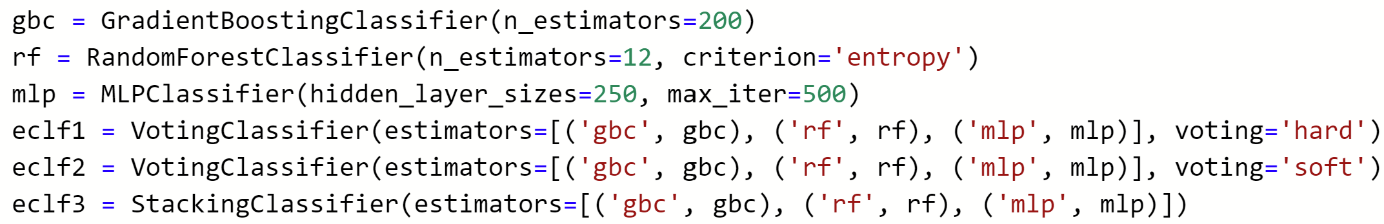


Figure 10 - Custom parameters and ensemble models:

Just like the first set of tests, I ran each classifier 5 times and recorded the results in a graph, then calculated their mean accuracy and precision score.

**Note:** There are 2 voting classifiers, (one being hard voting, the other being soft). This is because hard voting ensemble models classify data instances based on the majority vote of all models involved, i.e., the mode prediction. On the other hand, soft voting models classify data based on the weights (importance) of each model involved. These weights are assigned according to the probability of all predictions made **[**[**10**](#_References)**]**.

Based on this set of results, StackingClassifier is the best-performing ensemble model, with a mean accuracy and precision score of **87%**. The hard and soft VotingClassifiers had very similar performance, only differentiating slightly in accuracy. GradientBoostingClassifier also performed well, attaining the same accuracy as the soft VotingClassifier. However, it turned out that the way that I changed the hyperparameters of the individual models noticeably worsened their performances. Therefore, I removed the changes from the parentheses for the third and final test of performance. The last thing to note from these results at face-value is that the movement of each model’s performance is somewhat related. This will be explained further in the next section.

The last set of tests that I performed on each model had the sole purpose of gauging their accuracy and precision even better. I did this using k-fold cross validation, where **k** is the number of folds/sections that the input data is split into. Each section is used as a testing set at a certain stage, and each time, the other sections serve as training sets. In my case, I decided to use 5-fold cross validation. The difference in this sets of tests is that I only ran the models 3 times. This is because 5-fold cross validation returns 5 different accuracy and precision scores for each classifier, and the mean of these scores can be printed below using the ‘.mean()’ method. The following graphs show the recorded mean of each models performance across the three instances.

These final results were quite surprising, due to the fact that 4 of the 6 models had a mean accuracy score of 0.86 in the first test, and the StackingClassifier and MLPClassifier’s accuracy scores did not change at all. Moreover, the hard and soft VotingClassifiers behaved very differently to before. This is because the soft voting model’s mean accuracies were consistently higher than the hard voting model’s, despite following the same trend. Also, the hard voting’s mean accuracy and precision scores move in opposite directions, however this could be a coincidence/anomaly.

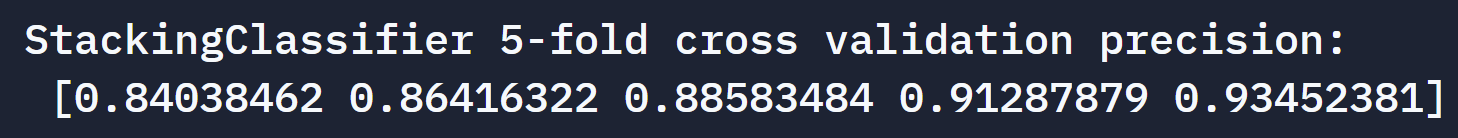
# Evaluation and results

Since each model determines whether a patient has heart disease **or not** based on them having specific characteristics of the data, this is a binary classification problem. Thus, the correct performance metrics to use are ‘accuracy\_score’ and ‘precision\_score’. As previously stated, accuracy is the fraction of correctly classified predictions from a given model, whereas precision quantifies the amount of correct positive predictions made, using the formula: where *tp* represents the number of true positives, and *fp* represents the number of false positives **[**[**11**](#_Evaluation_and_results)**]**.

Conclusively, I can confidently say that some of the predictions that I made were correct, and others were incorrect. For example, researching and utilizing the GradientBoostingClassifier proved to be a very good decision. This is because (just as predicted) the algorithm outperformed RandomForest. This can be seen in the ensemble model accuracy and precision graphs. If inspected closely, it can be seen that in the first test, both models have an equal accuracy score of **0.88**, and an equal precision score of **0.85**. RandomForest’s accuracy then dips to **0.86** in the second test, and **0.84** in the third. GBC’s accuracy also dips to **0.86** in test 2, and remains at **0.86** in test 3. As for RandomForest’s precision score, this stagnates at **0.85** for the first four tests. However, something interesting happens in the fourth test. GBC’s scores dive, in a way where the scores become **0.85** for both model’s accuracy and precision. On the fifth and final test, both models increase in accuracy and precision, however GBC’s increase is larger. What we can derive from this is that GradientBoostingClassifier outperforms RandomForest with a **1.2%** higher mean accuracy, and **1.4%** higher mean precision, although it does not consistently outperform RandomForest, but more-so ‘bounces off of it’.

On the contrary, some things did not go according to plan. For example, editing the hyperparameters of RandomForest and MLPClassifier seemed to slightly worsen their performance as opposed to improving it. This can be seen when comparing the first set of tests with default parameters to the second set of tests with custom parameters. In retrospect, I also should have tested GradientBoostingClassifier with default parameters to begin with, as I am unsure as to whether increasing ‘n\_estimators’ increased or decreased its performance.

Finally, performing 5-fold cross validation on the shortlisted models three times and calculating the mean result gave me a clearer perception of each model’s true performance. I was surprised to see that the soft VotingClassifier dominated accuracy, and that MLP still had a lower accuracy and performance, despite removing the tweaked hyperparameters. Overall, I have measured that the system is successful by performing various types of tests on each classification model, recording results, and determining the best model to use based on the results. In this case, considering all factors, I would say that the preferred model to use would be StackingClassifier, simply because it attained the highest mean accuracy and precision of all models. It is also worth noting that in the third and final cross validation test, the fourth and fifth fold gave precision scores of **0.91** and **0.93**. This means that if system was fine-tuned correctly, the StackingClassifier could achieve precision and possibly accuracy scores of over **90%**.



# References

1. Pin on Future of Artificial Intelligence [WWW Document], n.d. URL <https://www.pinterest.co.uk/pin/794111346782106915/> (accessed 1.17.22) – **Title page cover image**.

2. Predicting Heart Failure with KNN 90% Accuracy [WWW Document], n.d. URL <https://kaggle.com/victordonjuan/predicting-heart-failure-with-knn-90-accuracy> (accessed 1.17.22) – **Facts about my specific dataset**.

3. What Is Pandas in Python? Everything You Need to Know, n.d. . ActiveState. URL <https://www.activestate.com/resources/quick-reads/what-is-pandas-in-python-everything-you-need-to-know/> (accessed 1.19.22) – **Python ‘pandas’ definition**.

4. Lahera, G., 2019. Unbalanced Datasets & What To Do. Strands Tech Corner. URL <https://medium.com/strands-tech-corner/unbalanced-datasets-what-to-do-144e0552d9cd> (accessed 1.19.22) – **Why unbalanced datasets can be problematic**.

5. sklearn.svm.SVC [WWW Document], n.d. . scikit-learn. URL <https://scikit-learn/stable/modules/generated/sklearn.svm.SVC.html> (accessed 1.19.22) – **SVC information**.

6. Brownlee, J., 2018. How to Make Predictions with scikit-learn. Machine Learning Mastery. URL <https://machinelearningmastery.com/make-predictions-scikit-learn/> (accessed 1.19.22) – **predict() method function/definition**.

7. Accuracy, Precision, Recall & F1 Score: Interpretation of Performance Measures [WWW Document], 2016. . Exsilio Blog. URL <https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/> (accessed 1.19.22) – **Accuracy and precision definitions**.

8. Gradient boosting. Wikipedia URL <https://en.wikipedia.org/wiki/Gradient_boosting> (accessed 1.19.22) - **Gradient Boosting research.**

9. sklearn.ensemble.GradientBoostingClassifier [WWW Document], n.d. . scikit-learn. URL <https://scikit-learn/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html> (accessed 1.21.22) - **n\_estimators definition.**

10. Kumar, A., 2020. Hard vs Soft Voting Classifier Python Example. Data Analytics. URL <https://vitalflux.com/hard-vs-soft-voting-classifier-python-example/> (accessed 1.21.22) – **Hard and Soft voting definitions**.

11. sklearn.metrics.precision\_score [WWW Document], n.d. . scikit-learn. URL <https://scikit-learn/stable/modules/generated/sklearn.metrics.precision_score.html> (accessed 1.18.22) - **Precision equation**.