Part 1 - GA for Knapsack Problem

My genetic algorithm that I implemented to solve the knapsack problem follows the pseudocode provided in lectures.

Because it is being applied to different files representing different knapsack problems, the parameters for the GA need to be appropriate for each one. Before my GA runs, my program checks which file it is being applied to and sets the parameters to have suitable values.

An initial population is created by generating bit strings, where each bit has an equal chance of being 1 or 0, and represents whether an item is selected or not. Bit strings are an appropriate representation for individuals as they easily communicate which items are included and are also easy to manipulate during GA. Tasks like crossover and mutation are easily performed on bit strings because of their simple nature. All of these bit strings are grouped together in the population which the GA then accesses to evolve.

The termination criteria for my GA is simply whether it has reached the predefined number of generations to run for. I chose these generations after developing my GA and manually testing how many times it needs to run to achieve good results. For files 10_269, 23_10000, and 100_1000, they run for 20, 40, and 150 generations, respectively. The convergence graphs show that the best solutions either reach the optimal value (shown in red), or come very close. Therefore, these generation numbers are appropriate for these knapsack problems.

At the beginning of every generation, each individual is assessed on its performance using the fitness function. I implemented my fitness function following the penalty method as described in lectures. The overall fitness is returned as the value of items that the individual includes, with a penalty applied if the weight exceeds the bag capacity. The penalty value includes an alpha multiple that decides how harsh the penalty is. For the first two files, a relatively small alpha value of 5 is used. The nature of the last file, 100_1000, means that a higher alpha value of 100 should be used. In this file, there are many items that have a weight that almost meets the bag capacity by themselves. Therefore, the GA needs to be harsh on individuals that exceed the weight because it is so easy to do so with item weights being very high. This makes sure that the highest value individuals are never exceeding the bag capacity. Overall, this fitness function is appropriate for the knapsack problem because individuals are first and foremost valued by the items they contain, but penalised if they violate the domain rule of exceeding the bag capacity.

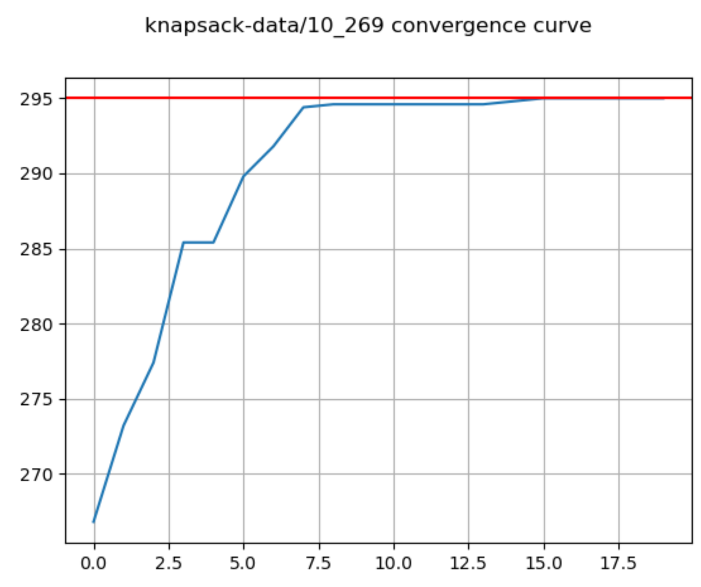
Next, a new empty population is created. For the three datasets, I set the population size to be 50, 100, and 150. The first individuals to be included in this are the top performing individuals from the previous generation. This is where I implemented elitism which ensures that a small number of the most valuable individuals are kept in the population. For files 10_269 and 23_10000, I had the 5 best individuals copied which consisted of 10% and 5% of the population, respectively. File 100_1000 had 25 individuals copied over which was also 5% of the population. Roulette wheel selection is used to choose the remaining individuals that will be included in the next generation. I took advantage of a thread on Stack Overflow for details on how to implement this selection process. I needed to make my fitness values positive, so I added the absolute value of the worst individual to each of them. As a result, my GA is choosing individuals with a probability proportionate to their value. Pairs of these selected individuals undergo reproduction through 1-point crossover and bit-flip mutation. 1-point crossover has two individuals swap parts of their bit strings at a random point. Flipping a bit in the individual's bit string to the opposite value provides the mutation functionality of the GA. For all three files, the mutation rate is set at 0.2, or 20%.

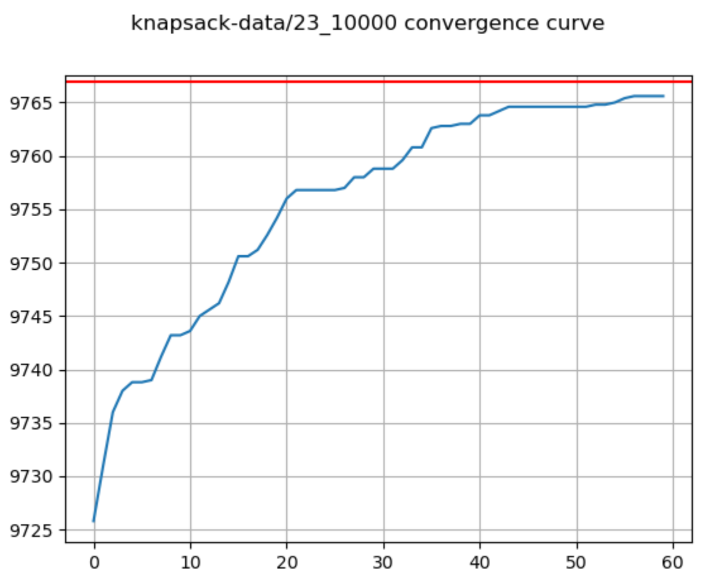
This new collection of evolved individuals is assigned as the current population, and the GA continues until the stopping criteria is met.

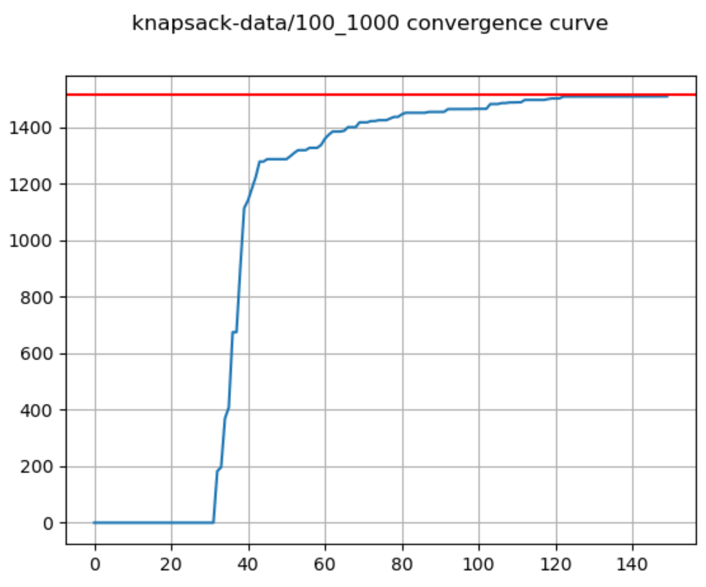
The means, standard deviations, and convergence curves of the different files over 5 runs are shown below. My GA is shown to be extremely effective for file 10_269 as the mean is the same as the optimal value, indicating that it found the perfect solution each time. The standard deviation of 0 for this file matches up with the mean results as well. The convergence curve shows a linear improvement rate until the 7th generation where it almost reaches the optimal value. From there, it needs a handful of generations to achieve the optimal value. My GA also performs very well on file 23_10000 as the mean is only 0.014% off the optimal value. It also found similar solutions as the standard deviation is quite small at 2.8. The convergence curve for this file shows a logarithmic growth rate, improving quickly at the start but slowing down over time. The last file, 100_1000, was more challenging for my GA as the mean value was slightly off the optimal value. The standard deviation was also greater at 7.3, indicating that it wasn't finding consistent solutions. I believe that these results are reflected from the nature of the file. As mentioned earlier, there are many items that take up a significant portion of the bag capacity, and having so many items in general allows for much higher solution variation. An interesting pattern to note on the convergence graph is the GA's inability to find any viable solutions until the 30th generation. Once it does find some viable solutions, their value drastically increases in just a dozen generations. However, it does then take another hundred generations to come close to the optimal value.

```
Mean values of best solutions across 5 runs:
knapsack-data/10_269 = 295.0        Optimal value = 295
knapsack-data/23_10000 = 9765.6        Optimal value = 9767
knapsack-data/100_1000 = 1508.2        Optimal value = 1514

Standard deviation of best solutions across 5 runs:
knapsack-data/10_269 = 0.0
knapsack-data/23_10000 = 2.8000000000000003
knapsack-data/100_1000 = 7.277362159464101
```


knapsack-data/10_269 convergence curve
295
290
285
280
275
270
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


knapsack-data/23_10000 convergence curve
9765
9760
9755
9750
9745
9740
9735
9730
9725
0
10
20
30
40
50
60


knapsack-data/100_1000 convergence curve
1400
1200
1000
800
600
400
200
0
0
20
40
60
80
100
120
140

Part 2 - GA for Feature Selection

My GA for feature selection is structured very similarly to my GA for the knapsack problem. It features the same individual representation, population initialisation method, stopping criteria, elitism method, roulette wheel selection, crossover, and mutation. To accommodate the two different types of feature selection, I created different fitness functions for wrapper and filter based approaches. Overall, it follows the same flow of processes with the core difference being the fitness functions.

Just like the knapsack problem, the best representation for individuals are bit strings as they easily communicate which features are chosen and are also easy to manipulate during evolution. Bit string integers correspond to features in the dataset and 1 represents the individual as including it, and 0 represents the individual as ignoring it. Crossover and mutation are easily performed on bit strings because of how easily manipulated they are.

As mentioned before, I developed a unique fitness function for the wrapper based GA and filter based GA. The wrapper based fitness function takes an individual and modifies the dataset to only include features specified in the bit string. Next, it splits the data for training and testing, maintaining a consistent random state to keep different runs fair. A Naive Bayes classifier is fitted to the training data, and the fitness of the individual is returned as the classification accuracy on the test data. Overall, the fitness function is assessing how well the features that the individual has chosen perform. Feature choice will be reflected in the classification accuracy which is the value used for an individual's fitness.

The filter based fitness function also modifies the data to only include selected features. However, this function uses mutual information to determine an individual’s fitness. For assessing mutual information, continuous variables need to be discretised. Their patterns and trends will remain, but features will be grouped together in bins representing different ranges of values. I used a discretiser method from Scikit-learn to do this upon parsing the data. Mutual information can be calculated to provide an insight into the relationship between two variables. Specifically, it measures “the reduction in uncertainty for one variable given a known value of the other variable”[1]. Therefore, mutual information is a useful statistical measure for my filter based function.

I recorded the means and standard deviations of the computational execution time and classification accuracies for both datasets using both types of fitness function across 5 runs. The classification accuracies were found using a Naive Bayes classifier. For comparison, I also recorded a default accuracy of the classifier using all features. This would give me a baseline to compare the wrapper and filter based methods to.

In both datasets, the wrapper based fitness function is significantly faster than its filter based counterpart. I didn’t expect this result as wrapper based methods are generally slower than filter based functions. As I don’t know exactly how the library I used for calculating mutual

[1] https://machinelearningmastery.com/information-gain-and-mutual-information/

information in the filter based function works, it isn't a fair environment to draw any major conclusions. The standard deviations of the wrapper based GA are fairly small compared to the filter based method, showing the different spreads of the results.

The classification accuracies show interesting patterns. The mean classification accuracies of both methods are almost equivalent to the default accuracies using every feature. This shows that the selected features are able to achieve the same classification performance as when all of the features are used. Reaching the same performance with less data shows that applying feature selection has saved computational cost and could mean less data needs to be collected in the future for this particular application. All of the standard deviations are very small showing very little spread in the results.

Default WBCD classification accuracy: 94.39%
Default Sonar classification accuracy: 69.52%

File/GA type	Wrapper computational time	Filter computational time
WBCD	Mean: 11.96 seconds STD: 0.30	Mean: 63.50 seconds STD: 3.67
Sonar	Mean: 8.21 seconds STD: 0.05	Mean: 77.02 seconds STD: 1.82

File/GA type	Wrapper classification accuracy	Filter classification accuracy
WBCD	Mean: 96.28% STD: 0.0079	Mean: 92.87% STD: 0.0214
Sonar	Mean: 71.43% STD: 0.0702	Mean: 66.98% STD: 0.0407

Part 3 - GP for Symbolic Regression

For my genetic program, I included basic arithmetic operations such as addition, subtraction, multiplication, and division along with trigonometric, exponential, and boolean operations to my function set to ensure sufficiency. As the expression we are wanting to find has an equation for two specific ranges of *x*, I made my GP strongly typed to accommodate my boolean operators. This enabled checking for the value of *x*, and containing different subtrees for whether that is true or not. The square and trigonometric sine functions are required by the expression so including them in the function set was necessary. As for terminals, I included a range of whole numbers from 1 to 6, as well as true and false boolean values.

When assigning fitnesses in my fitness function, I used an average of the mean squared errors between the expected results and the individual's expression results. Mean squared errors were calculated on all of the fitness cases which I specified to be a range of whole numbers from -100 to 100. An average of the results was then returned as the individual's fitness. I initially had some issues with my GA because my fitness range wasn't large enough and was too concentrated around 0. Because the results for fitness cases around this value were so small, issues in individuals would only be noticed when inputting more extreme values. This problem was resolved when I applied fitness cases with a much larger range. After running my GA, this fitness case range of -100 to 100 is sufficient for finding strong individuals.

I used a large population size of 3000 to include a wide range of individuals with a high level of variation. I found 100 generations to be effective for evolving the population to create a high-performing individual. 10% elitism was used which ensured that there would be an appropriate number of elite individuals to perform crossover with. The mutation probability was also set at 10% and crossover at 50%. As recommended in lectures and further research, I set the max depth to be 17. For generating the initial population, I used ramp half-and-half to provide a good mix of individuals.

Running my GP shows consistent strong results. The best genetic programs for my 3 runs are below. Only 10 of the fitness cases are shown to avoid having too much content in the report. However, their patterns and general performance can still be seen.

1. Structure:
   if_then_else(is_positive(sub(ARG0, 2)), sin(ARG0), add(1, if_then_else(is_positive(ARG0), protected_div(sin(sin(protected_div(6, add(ARG0, 3)))), ARG0), add(1, if_then_else(is_positive(ARG0), ARG0, square(add(1, ARG0)))))))

   Performance & Results:
   ```
   -10 ) Expected: 83.0      GP Result: 83
   -8 ) Expected: 51.0      GP Result: 51
   -6 ) Expected: 27.0      GP Result: 27
   -4 ) Expected: 11.0      GP Result: 11
   -2 ) Expected: 3.0      GP Result: 3
   0 ) Expected: 3.0      GP Result: 3
   2 ) Expected: 1.4092974268256817      GP Result: 1.4014186542036084
   4 ) Expected: -0.5068024953079282      GP Result: -0.7568024953079282
   6 ) Expected: -0.1127488315322592      GP Result: -0.27941549819892586
   8 ) Expected: 1.114358246623382      GP Result: 0.9893582466233818
   10 ) Expected: -0.4440211108893698      GP Result: -0.5440211108893698
   ```

2. Structure:
   if_then_else(is_positive(ARG0), sin(ARG0), square(add(ARG0, sin(add(sin(sin(sin(add(sin(sin(sin(add(sin(sin(sin(sin(6)))), if_then_else(is_positive(1), protected_div(5, mul(add(ARG0, 6), add(5, 1))), sin(6)))))), if_then_else(is_positive(1), protected_div(5, mul(add(ARG0, 5), add(4, protected_div(if_then_else(0, 5, 6), mul(ARG0, 1))))), sin(6)))))), protected_div(5, 3))))))

   Performance & Results:
   ```
   -10 ) Expected: 83.0      GP Result: 83.05244988728387
   -8 ) Expected: 51.0      GP Result: 51.47290636779102
   ```

```
-6 ) Expected: 27.0      GP Result: 26.794935715210187
-4 ) Expected: 11.0      GP Result: 10.786291560189463
-2 ) Expected: 3.0      GP Result: 1.7781766250720519
0 ) Expected: 3.0      GP Result: 0.9724633680306494
2 ) Expected: 1.4092974268256817      GP Result: 0.9092974268256817
4 ) Expected: -0.5068024953079282      GP Result: -0.7568024953079282
6 ) Expected: -0.1127488315322592      GP Result: -0.27941549819892586
8 ) Expected: 1.114358246623382      GP Result: 0.9893582466233818
10 ) Expected: -0.4440211108893698      GP Result: -0.5440211108893698
```

3. Structure:
   if_then_else(is_positive(mul(5, ARG0)), add(sin(ARG0), protected_div(1, ARG0)), add(add(3, ARG0), add(if_then_else(is_postive(ARG0), ARG0, ARG0), square(ARG0))))

   Performance & Results:

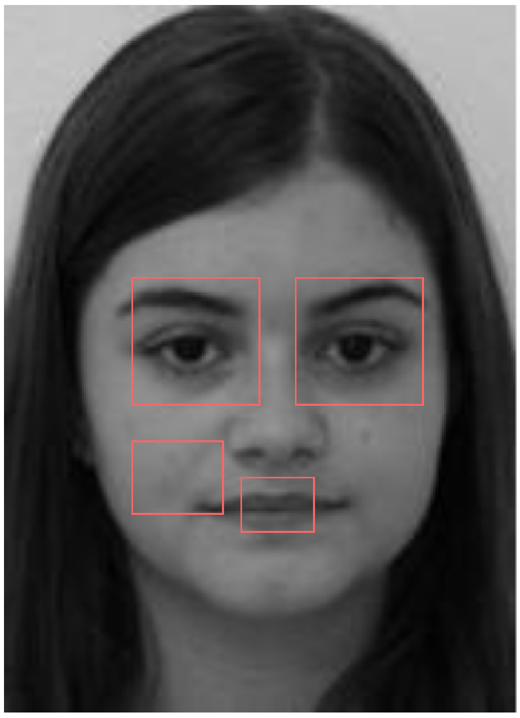
```
-10 ) Expected: 83.0      GP Result: 83
-8 ) Expected: 51.0      GP Result: 51
-6 ) Expected: 27.0      GP Result: 27
-4 ) Expected: 11.0      GP Result: 11
-2 ) Expected: 3.0      GP Result: 3
0 ) Expected: 3.0      GP Result: 3
2 ) Expected: 1.4092974268256817      GP Result: 1.4092974268256817
4 ) Expected: -0.5068024953079282     GP Result: -0.5068024953079282
6 ) Expected: -0.1127488315322592     GP Result: -0.1127488315322592
8 ) Expected: 1.114358246623382     GP Result: 1.114358246623382
```

Overall, these performances show very good results. The last run managed to find an expression which perfectly matches the results of the target expression. I was quite impressed with this result and it was satisfying seeing the decisions I made for my GP pay off. The structure of the best individual is shorter than the others and makes clear use of my boolean operators. However, my GP was able to identify appropriate formulas for each range of 0 and consistently give strong results overall.

Part 4 - GP for Image Classification

4.1 Manual Feature Extraction

For the task of classifying images of whether they are smiling or not smiling, I focused on regions that would change the most between the different behaviours. The first two local regions I thought to include were the eyes and mouth. Eyes show a strong difference in images of both types, with smiling people “closing” their eyes more. The mouth is also a clear indicator of if someone is smiling. Most individuals were smiling with their teeth showing, but I noticed that there were many who weren’t. To classify these images effectively, I chose to also include cheeks as a region for feature selection. My reasoning is that even if someone is smiling without their teeth showing, their cheeks still compress and show a difference along the crease. Among these local regions that I included is the whole image as a global region. I was somewhat generous with the pixel locations I specified for the region methods so that the program would have a better chance of capturing the same information across a range of many different people. For all of these regions, I extracted features using the SIFT method which proved to be effective.



4.2 Creating Tabular Datasets

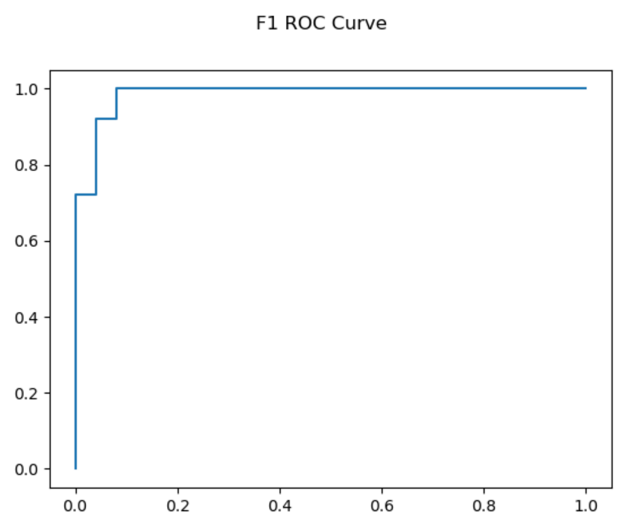
The tabular datasets are included in my submission under appropriate names.

4.3 Image Classification Using Manually Designed Features and Performance Evaluation

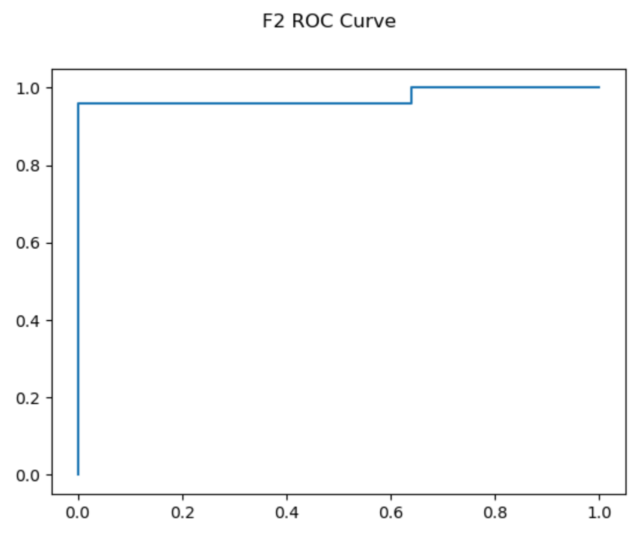
Note: the code I wrote for this section is at the bottom of part_four.py

For testing my GP model's performance I used the classification accuracy. I found this to be a useful method of measuring performance because it clearly shows how well the GP tree works on a given data set. The results of testing my evolved GP tree on the training and testing instances on the two datasets can be seen below, along with the corresponding ROC curves.

Data set: F1
Train data accuracy: 95%
Test data accuracy: 94%



Data set: F2
Train data accuracy: 95%
Test data accuracy: 96%



Given that I can be considered a domain expert for this smiling classification task and I manually chose the regions of interest, I expected the accuracy results to be quite high. This was because I knew what facial features changed when smiling, so I simply needed to specify regions surrounding those parts of the images. With this domain knowledge providing valuable regions, high quality features should be able to be extracted. Fortunately the real accuracy values reflect this at around 95%. This means that using the features extracted from the regions I designed, a developed GP tree was very effective in learning the difference between the two types of images.

As for the features themselves, I expect the quality of them to be quite high because of the application of SIFT. General SIFT advantages include robustness to occlusion, distinctive features, high feature quantity, and strong invariance. The qualities will extend to the features extracted from the regions I manually located, therefore giving them a high quality as well.

4.4 Automatic Feature Extraction through GP

Note: the code I wrote for this section and 4.5 are in the IDGP_main.py file

F1 Dataset
Best individual structure:
FeaCon2(Local_Histogram(Region_R(Image0, 78, 83, 50, 24)), FeaCon2(FeaCon3(Local_DIF(Region_S(Image0, 41, 39, 20)), Global_HOG(Image0), Local_uLBP(Region_R(Image0, 6, 54, 41, 23))), Local_DIF(Region_S(Image0, 35, 17, 23))))

Test results 94.0
Train time 225.828125
Test time 10.78125

F2 Dataset
Best individual FeaCon3(FeaCon2(Local_HOG(Region_S(Image0, 142, 86, 47)), Local_SIFT(Region_S(Image0, 103, 60, 42))), Global_SIFT(Image0), Global_uLBP(Image0))
Test results 92.0
Train time 396.125
Test time 20.484375

From these results between the two datasets, the best feature extractors appear to be Local_Histogram, Local_DIF, Global_HOG, Local_uLBP for the first dataset and Local_HOG, Local_SIFT, Global_SIFT, and Global_uLBP for the second dataset. Common feature extractors between the two include HOG and uLBP methods, indicating that those are the best image feature extractors. These features will be taking advantage of the changes in brightness and other patterns within the regions supplied to them.

4.5 - Image Classification Using Features Extracted by GP and Performance Comparison

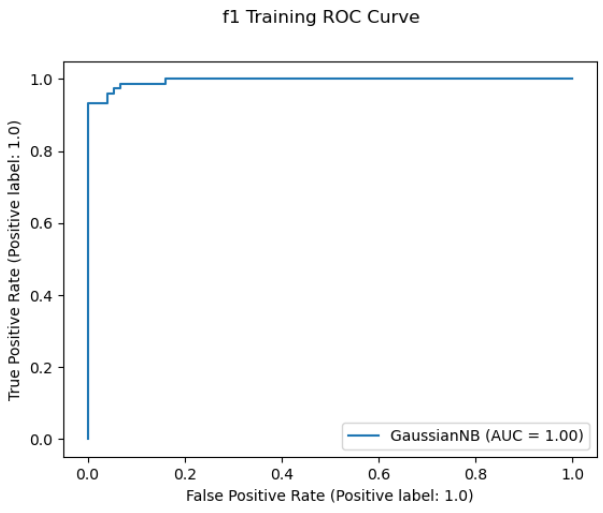
Classification Accuracies on Dataset F1
Automatic Feature Selection - Training Instances: 96%
Automatic Feature Selection - Testing Instances: 92%
Manual Feature Selection - Testing Instances: 92%

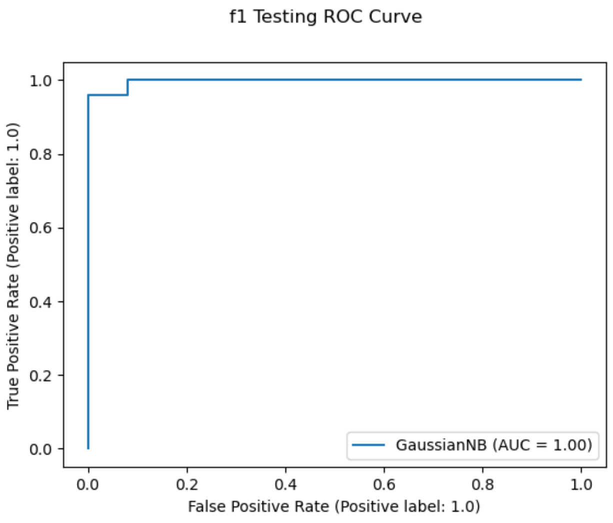
Classification Accuracies on Dataset F2
Automatic Feature Selection - Training Instances: 95%
Automatic Feature Selection - Testing Instances: 96%
Manual Feature Selection - Testing Instances: 94%

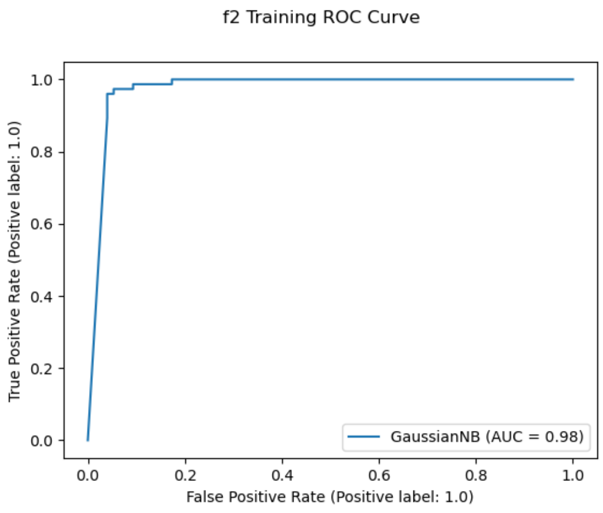
These classification results show very strong performance from both methods of feature selection on the different groups of instances. With no major differences between any classification accuracies, these feature selection methods can be considered to be performing equally well.

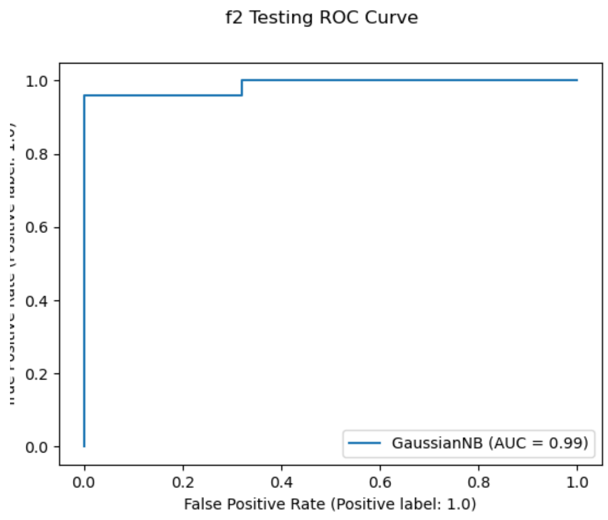
The performance of the automatic feature selection was very impressive as it was able to match the performance of manually selected features from regions specified by a domain expert. This shows that the automatic feature selection methods didn't need the domain knowledge required by the manual method to score so high, but instead could rely on its own processes and results to achieve a high classification accuracy.

The ROC curves below reflect the quality of the classification accuracy results. These ROC curves indicate greater discrimination capacity within the classifier. The true positive rate and false positive rate are also visualised below, further communicating the successfulness of the automatic feature selection methods.


f1 Training ROC Curve
True Positive Rate (Positive label: 1.0)
False Positive Rate (Positive label: 1.0)
GaussianNB (AUC = 1.00)


f1 Testing ROC Curve
True Positive Rate (Positive label: 1.0)
False Positive Rate (Positive label: 1.0)
GaussianNB (AUC = 1.00)


f2 Training ROC Curve
True Positive Rate (Positive label: 1.0)
False Positive Rate (Positive label: 1.0)
GaussianNB (AUC = 0.98)


f2 Testing ROC Curve
False Positive Rate (Positive label: 1.0)
GaussianNB (AUC = 0.99)