Middle East Technical University Department of Computer Engineering

CENG 499

Intro to ML

Fall 2024-2025

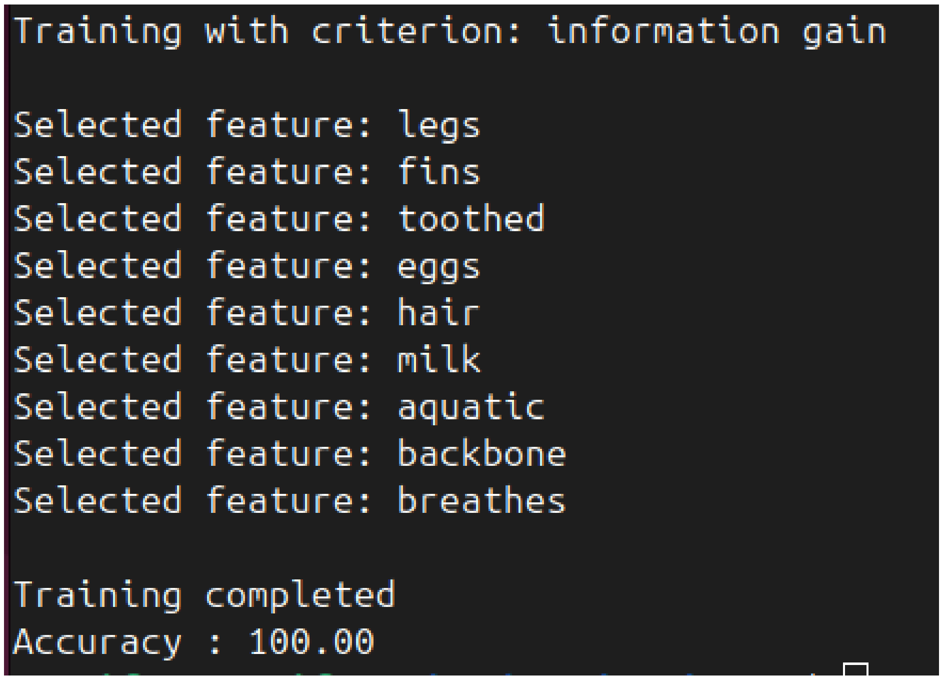
Homework 3

Name SURNAME: Aly Asad GILANI
Student ID: 2547875

Part 1

For this part, I have extensively commented almost every line in my code so it is clear what I did. In general, the helper functions were not too difficult as their calculation formulas have been given in the "DecTrees.pdf" slides, and they could be implemented with just numpy operations. The ID3 function is similar, its pseudocode was given in the slides and I mostly followed that. The predict function was extremely simple, I just had to iteratively follow the tree's path according to the features of the given data sample.

The results of my code on the dataset for "Information Gain" are:



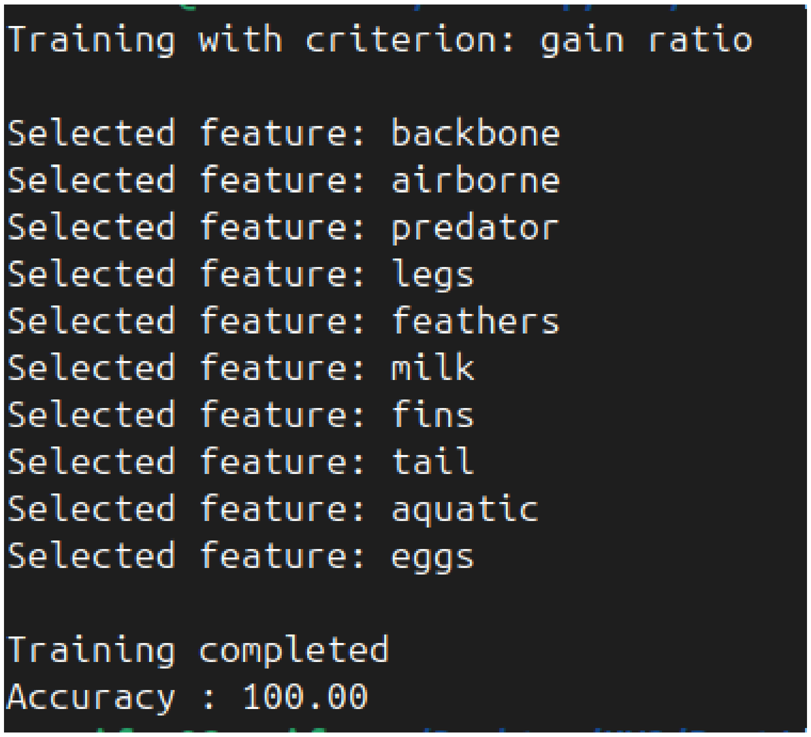
```
Training with criterion: information gain

Selected feature: legs
Selected feature: fins
Selected feature: toothed
Selected feature: eggs
Selected feature: hair
Selected feature: milk
Selected feature: aquatic
Selected feature: backbone
Selected feature: breathes

Training completed
Accuracy : 100.00
```

Figure 1: Results with Information Gain

We can see that the training accuracy of the constructed tree is 100%, this means that there are no contradictions in the training dataset of two datapoints having the same features but different labels. However, we cannot tell if our tree is overfitted or not without a testing dataset.

Additionally, we see that the first picked feature is ”legs”, which means it is the most influential feature to distinguish the animals, and it has the most information gain from all other features in the training dataset. Similarly, ”fins” is second, ”toothed” is third, and so on. We also see that the tree only selected 9 features from the 16 in the dataset for its structure, and discarded the rest. This means the rest of the features contributed basically no information to classify and label different animals.

The results of my code on the dataset for ”Gain Ratio” are:



```
Training with criterion: gain ratio

Selected feature: backbone
Selected feature: airborne
Selected feature: predator
Selected feature: legs
Selected feature: feathers
Selected feature: milk
Selected feature: fins
Selected feature: tail
Selected feature: aquatic
Selected feature: eggs

Training completed
Accuracy : 100.00
```

Figure 2: Results with Gain Ratio

We can see that the training accuracy is 100% for gain ratio as well. This criterion selected features in a different order from information gain. It gave the most importance to ”backbone”, and then ”airborne”, then ”predator” and so on. Also differently from the first criterion, gain ratio made a deeper decision tree with 10 features instead of 9, so the chances of it overfitting are slightly greater, and it might also take slightly more time to classify new data.

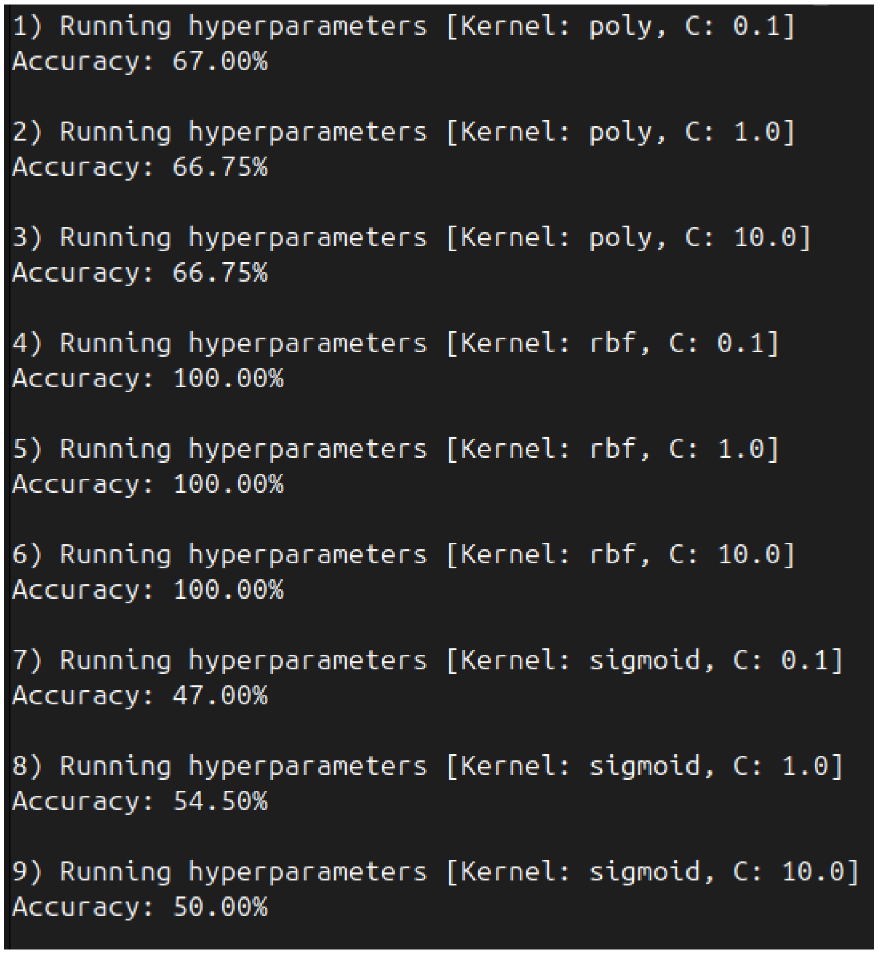
Part 2

2.1: Dataset 1

For the first dataset, even though the minimum number of configurations needed is 4, I tried 9 different configurations to robustly test the possible values. They are:

Kernels = [Polynomial, RBF, Sigmoid]
C = [0.1, 1.0, 10.0]

Where the degree for the Polynomial function is default (degree = 3) and RBF is the Radial Basis Function. The code is not too long and I have written comments to explain what I did, but basically I defined the possible hyperparameters, looped over them for grid-search, trained the SVM model for each configuration, printed its accuracy, and finally plotted the produced boundaries. The accuracy results are:

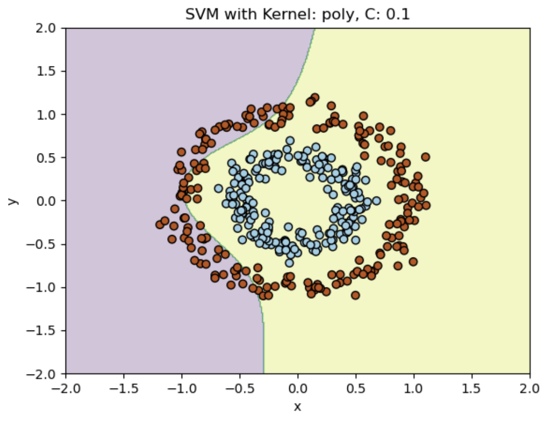


```
1) Running hyperparameters [Kernel: poly, C: 0.1]
Accuracy: 67.00%

2) Running hyperparameters [Kernel: poly, C: 1.0]
Accuracy: 66.75%

3) Running hyperparameters [Kernel: poly, C: 10.0]
Accuracy: 66.75%

4) Running hyperparameters [Kernel: rbf, C: 0.1]
Accuracy: 100.00%

5) Running hyperparameters [Kernel: rbf, C: 1.0]
Accuracy: 100.00%

6) Running hyperparameters [Kernel: rbf, C: 10.0]
Accuracy: 100.00%

7) Running hyperparameters [Kernel: sigmoid, C: 0.1]
Accuracy: 47.00%

8) Running hyperparameters [Kernel: sigmoid, C: 1.0]
Accuracy: 54.50%

9) Running hyperparameters [Kernel: sigmoid, C: 10.0]
Accuracy: 50.00%
```

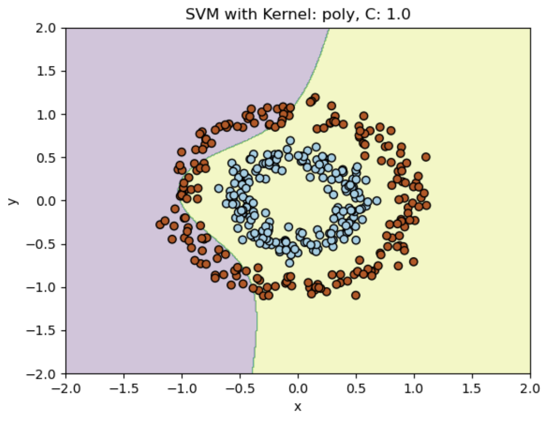
Figure 3: Results of SVM over 9 hyperparameter configurations

From a glance, we can see that the RBF kernel perfectly fit and classified the test data. The Polynomial kernel came second with around 66.83% accuracy, and the Sigmoid kernel performed the worst. We can also see that changing the "C" values did not affect the accuracy much. In terms of speed, the RBF kernel ran instantly, the Polynomial kernel gave its results after around 2 seconds, and the Sigmoid kernel took the most time at around 10 seconds. Now, lets see the boundary plots for each configuration.

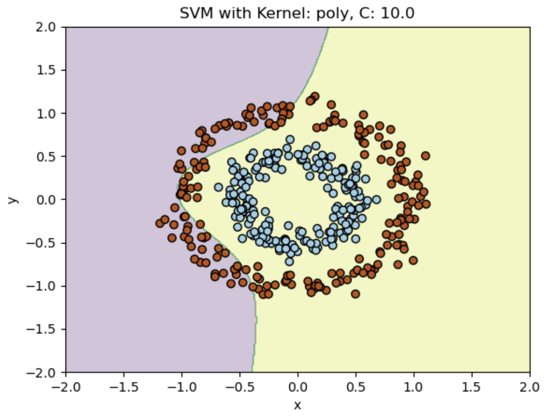
Boundary plots for Polynomial kernel:



(a) Polynomial with C = 0.1



(b) Polynomial with C = 1.0



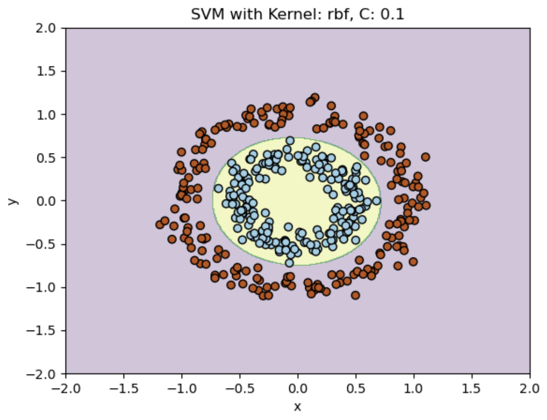
(c) Polynomial with C = 10.0

Figure 4: SVM boundary plots with Polynomial kernel

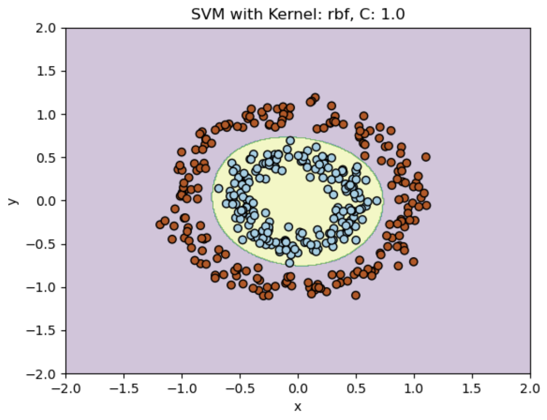
As we can see, no matter the value of C, the Polynomial kernel was not able to very accurately define the boundaries for the two distinct classes, and therefore its accuracy was not the best. We could maybe improve the boundaries and make them more circular if we used a greater degree as the model's input (for example, degree = 50), or maybe we could compare even and odd degrees (for example comparing degree = 2 and degree = 3). We could test the degrees by searching over them as well in the hyperparameter grid search. However, as we already had 9 different configurations, I decided not to search over the degrees as well, as it would greatly increase the running time as well as the number of graphs we get.

As for the performance, it averaged at around 66.83% accuracy, and took around 2 seconds for each configuration to run.

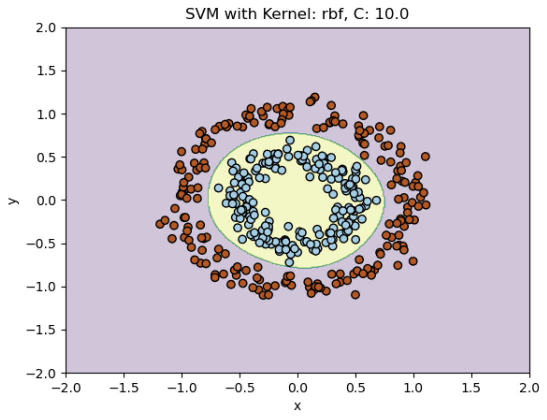
Boundary plots for RBF kernel:



(a) RBF with C = 0.1



(b) RBF with C = 1.0



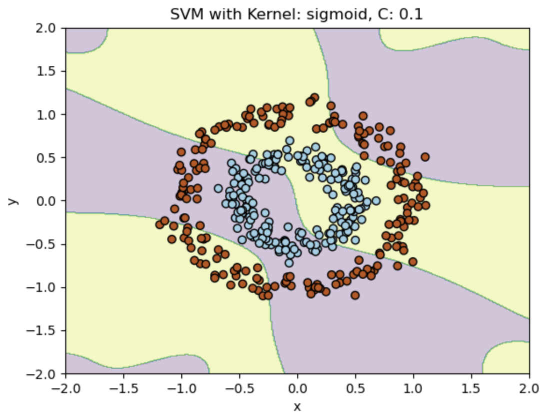
(c) RBF with C = 10.0

Figure 5: SVM boundary plots with RBF kernel

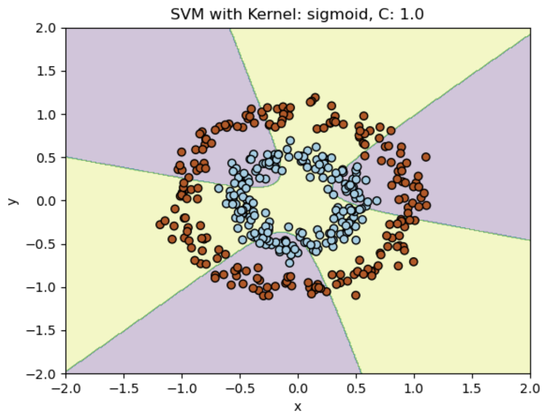
As we can see, all three plots created almost identical boundaries, even though their C values were different. RBF performed the best as it properly identified and classified the radial shape of the dataset's separation.

As for the performance, all three configurations received 100% accuracy score and each configuration took less than a second to run.

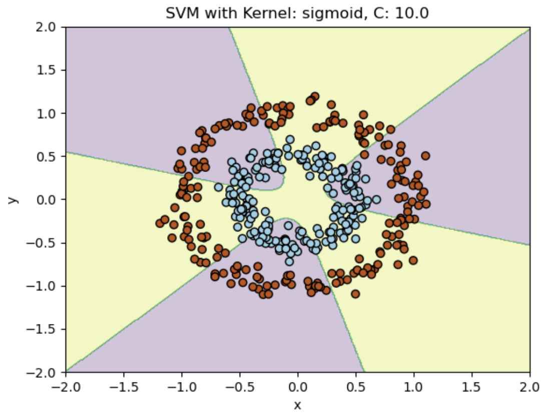
Boundary plots for Sigmoid kernel:



(a) Sigmoid with C = 0.1



(b) Sigmoid with C = 1.0



(c) Sigmoid with C = 10.0

Figure 6: SVM boundary plots with Sigmoid kernel

As we can see, the Sigmoid kernel performed the worst out of the others, and did not even come close to identifying the proper boundaries of the two classes in the dataset. Increasing the C value from 0.1 to 1.0 changed the shape of the boundaries a bit, but further increasing the C value did not result in any changes.

As for the performance, it averaged at around 50.5% accuracy, and took around 10 seconds for each configuration to run.

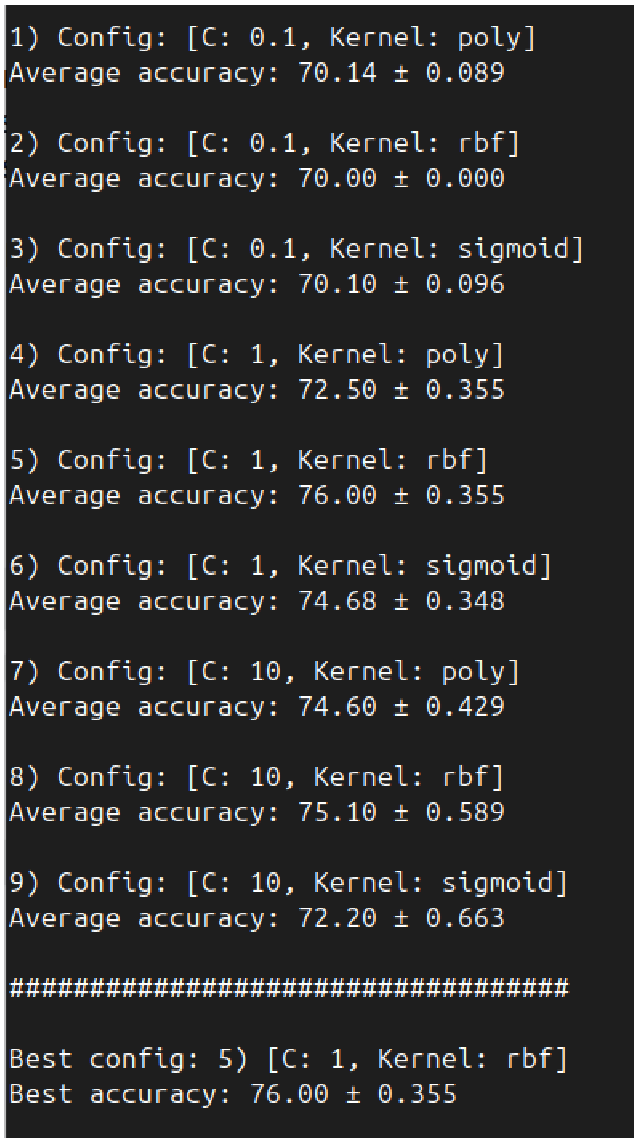
In conclusion, the RBF kernel performed the best, most probably due to the radial nature of the two classes in the training dataset. The polynomial kernel could most probably be greatly improved by experimenting with the ”degree” parameter. The sigmoid kernel performed the worst.

2.2: Dataset 2

For the second dataset, I considered **accuracy** to test and evaluate the hyperparameter configurations. Similar to the first dataset, even though the minimum number of configurations needed is 4, I tried 9 different configurations to robustly test the possible values. They are the same:

Kernels = [Polynomial, RBF, Sigmoid]
C = [0.1, 1.0, 10.0]

Where the degree for the Polynomial function is default (degree = 3) and RBF is the Radial Basis Function. In the code, I made a main loop for 5 times for cross-validation. In it, I randomized the dataset, preprocessed it using Standard Scaler, then did grid search on it using the GridSearchCV class. Then, for each hyperparameter configuration on this shuffled data, I noted the mean accuracy. In the end, I had 5 mean accuracies for each config due to running the cross-validation 5 times. I found the average mean accuracy for each hyperparameter configuration and its confidence interval, and printed it, as well as the configuration that performed the best. I have added comments to the code as well for further clarification. The results are:



```
1) Config: [C: 0.1, Kernel: poly]
Average accuracy: 70.14 ± 0.089

2) Config: [C: 0.1, Kernel: rbf]
Average accuracy: 70.00 ± 0.000

3) Config: [C: 0.1, Kernel: sigmoid]
Average accuracy: 70.10 ± 0.096

4) Config: [C: 1, Kernel: poly]
Average accuracy: 72.50 ± 0.355

5) Config: [C: 1, Kernel: rbf]
Average accuracy: 76.00 ± 0.355

6) Config: [C: 1, Kernel: sigmoid]
Average accuracy: 74.68 ± 0.348

7) Config: [C: 10, Kernel: poly]
Average accuracy: 74.60 ± 0.429

8) Config: [C: 10, Kernel: rbf]
Average accuracy: 75.10 ± 0.589

9) Config: [C: 10, Kernel: sigmoid]
Average accuracy: 72.20 ± 0.663

###################################

Best config: 5) [C: 1, Kernel: rbf]
Best accuracy: 76.00 ± 0.355
```

Figure 7: Results of Cross-Validation

I have also converted the results into a LaTeX table:

Config #	**C**	**Kernel**	**Accuracy**	**Confidence**
1	0.1	Polynomial	70.14	0.089
2	0.1	RBF	70.00	0.000
3	0.1	Sigmoid	70.10	0.096
4	1	Polynomial	72.50	0.355
5	1	RBF	76.00	0.355
6	1	Sigmoid	74.68	0.348
7	10	Polynomial	74.60	0.429
8	10	RBF	75.10	0.589
9	10	Sigmoid	72.20	0.663

Table 1: Hyperparameter Configurations and Cross-Validation Results

As we can see, the accuracy for each hyperparameter configuration is around 70-76%. The majority improvement came from increasing the value of C from 0.1 to 1.0. Further increasing the C value did not improve the accuracy. As for the kernels, the RBF kernel performs the best among the three with slightly higher overall accuracies, while the other two kernels are on par with each other.

The confidence deviation increases as C increases, which means that repeated runs on the same hyperparameters gives a wider spread of accuracies as C increases. The best accuracy is obtained from using the RBF kernel with C = 1, with an accuracy of 76.00%.

Part 3

For this part, I will first explain the general code I wrote, then the classifers and their hyperparameters that I chose, and the results obtained.

For the code, I started by defining the outer and inner folds using RepeatedStratifiedKFold as described in the HW pdf. Then, I created a list of classifiers (total of 6) that I will evaluate, as well as the possible hyperparameter configurations to test. I created two helper functions, one for stochastic models and one for non-stochastic ones, so I can run the stochastic models multiple times to get statistically significant results. Even though the questions states we should run them at least 5 times, I decided to run them for 10 times for better and more meaningful results.

Then, I created a loop over the classifiers, since we are testing 6 classifiers, this loop will run 6 times. In this loop, I printed the name of the classifier we are evaluating currently and started a timer to note how much time it took to train/predict and find the best F1 score for that classifier (I will mention each classifier's time taken separately). Then, I set up the pipeline so the data can be scaled using MinMaxScaler and then setup the GridSearchCV class to perform the inner cross-validation loop.

Next, I ran the outer cross-validation loop myself, in which I performed grid search on the inner cross-validation loop. I also printed the hyperparameters results for 1 run of the outer cross-validation loop (so we can get a general idea of how the different hyperparameter configurations are performing). Next, I ran the testing data on the hyperparameter configuration that performed the best from the inner cross-validation loop, and saved its F1 score. I repeated this step in total 15 times (because the outer loop has 3 splits, 5 repetitions). Then, I found the final mean F1 score and its confidence interval for the current classifier (for example KNN) using the 15 F1 scores of the best hyperparameter configurations.

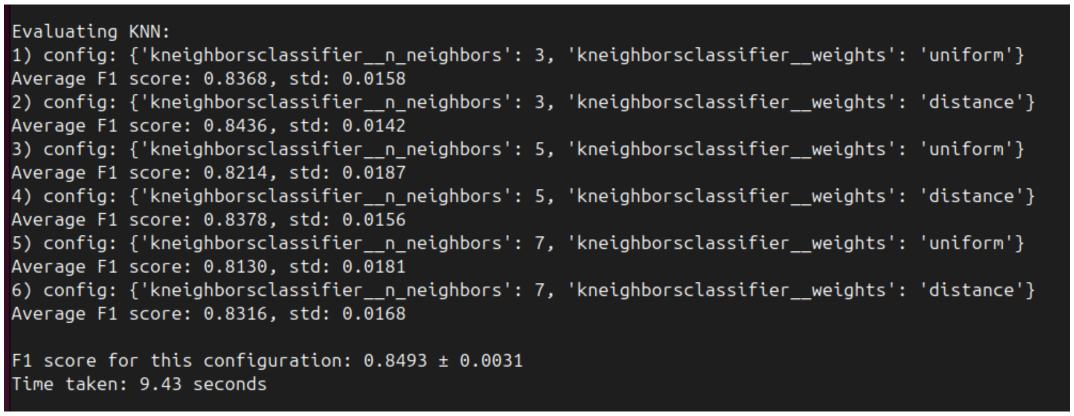
In the end, I printed the mean F1 scores and their confidence interval for each classifier, and the time taken to evaluate it. From the next page, I will post the results obtained and comment about the findings for each classifier. Even though in the question it is stated that we should test at least 2 hyperparameter configurations, for most of the classifiers I decided to test more than 2 to get more reliable results.

3.1: KNN

Starting with the KNN classifier, I chose the following hyperparameters to be tested (total of 6 configurations):

n_neighbours = [3, 5, 7]
weights = [uniform, distance]

The results are (std: Standard Deviation):



```
Evaluating KNN:
1) config: {'kneighborsclassifier__n_neighbors': 3, 'kneighborsclassifier__weights': 'uniform'}
Average F1 score: 0.8368, std: 0.0158
2) config: {'kneighborsclassifier__n_neighbors': 3, 'kneighborsclassifier__weights': 'distance'}
Average F1 score: 0.8436, std: 0.0142
3) config: {'kneighborsclassifier__n_neighbors': 5, 'kneighborsclassifier__weights': 'uniform'}
Average F1 score: 0.8214, std: 0.0187
4) config: {'kneighborsclassifier__n_neighbors': 5, 'kneighborsclassifier__weights': 'distance'}
Average F1 score: 0.8378, std: 0.0156
5) config: {'kneighborsclassifier__n_neighbors': 7, 'kneighborsclassifier__weights': 'uniform'}
Average F1 score: 0.8130, std: 0.0181
6) config: {'kneighborsclassifier__n_neighbors': 7, 'kneighborsclassifier__weights': 'distance'}
Average F1 score: 0.8316, std: 0.0168

F1 score for this configuration: 0.8493 ± 0.0031
Time taken: 9.43 seconds
```

Figure 8: Results of KNN

For clarity, I have converted the results into a LaTeX table:

Config #	n_neighbours	weights	F1 score	Std
1	3	uniform	0.8368	0.0158
2	3	distance	0.8436	0.0142
3	5	uniform	0.8214	0.0187
4	5	distance	0.8378	0.0156
5	7	uniform	0.8130	0.0181
6	7	distance	0.8316	0.0168

Table 2: Results of KNN

We can see that no matter the hyperparameter configuration, the F1 scores are very similar to each other, at around 0.81 to 0.84 with small standard deviations.

From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.8493 with a confidence interval of ± 0.0031 for the KNN classifier.

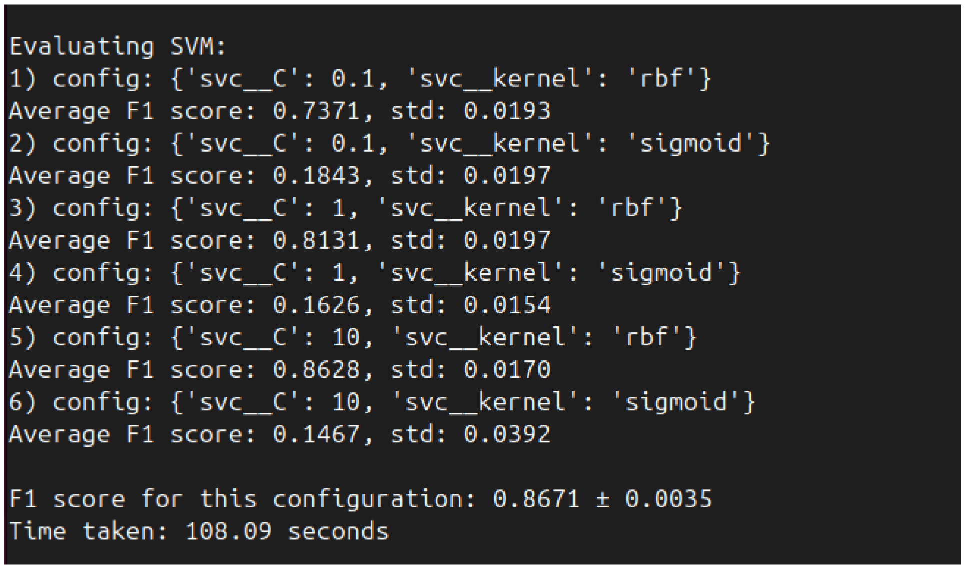
Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for KNN classifier is 9.43 seconds, which is quite fast when compared to the other classifiers.

3.2: SVM

For the SVM classifier, I chose the following hyperparameters to be tested (total of 6 configurations):

C = [0.1, 1, 10]
kernel = [rbf, sigmoid]

The results are:



```
Evaluating SVM:
1) config: {'svc__C': 0.1, 'svc__kernel': 'rbf'}
Average F1 score: 0.7371, std: 0.0193
2) config: {'svc__C': 0.1, 'svc__kernel': 'sigmoid'}
Average F1 score: 0.1843, std: 0.0197
3) config: {'svc__C': 1, 'svc__kernel': 'rbf'}
Average F1 score: 0.8131, std: 0.0197
4) config: {'svc__C': 1, 'svc__kernel': 'sigmoid'}
Average F1 score: 0.1626, std: 0.0154
5) config: {'svc__C': 10, 'svc__kernel': 'rbf'}
Average F1 score: 0.8628, std: 0.0170
6) config: {'svc__C': 10, 'svc__kernel': 'sigmoid'}
Average F1 score: 0.1467, std: 0.0392

F1 score for this configuration: 0.8671 ± 0.0035
Time taken: 108.09 seconds
```

Figure 9: Results of SVM

For clarity, I have converted the results into a LaTeX table:

Config #	C	kernel	F1 score	Std
1	0.1	rbf	0.7371	0.0193
2	0.1	sigmoid	0.1843	0.0197
3	1	rbf	0.8131	0.0197
4	1	sigmoid	0.1626	0.0154
5	10	rbf	0.8628	0.0170
6	10	sigmoid	0.1467	0.0392

Table 3: Results of SVM

We can see that when we choose RBF (radial basis function) as the kernel, we get significantly better results than choosing sigmoid as the kernel. This most probably means that the sigmoid kernel is not well suited for this dataset. As for the C value, the general trend is that increasing it increases the F1 score as well, but this might lead to overfitting.

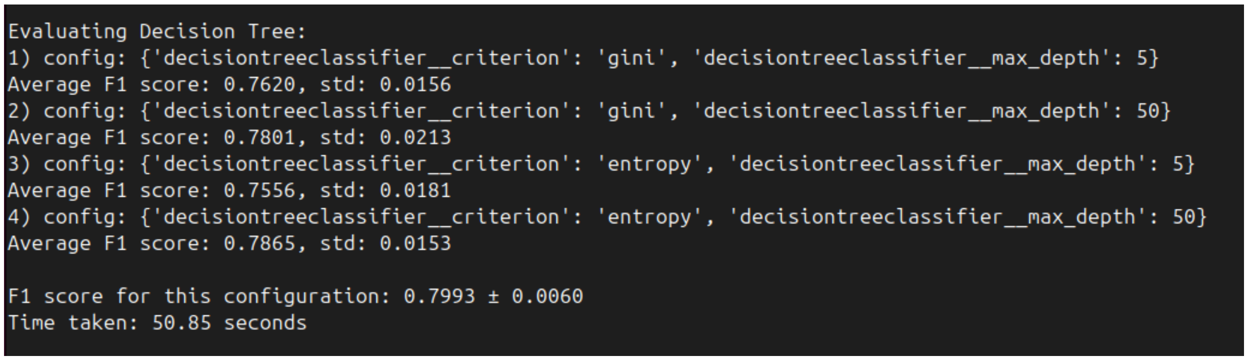
From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.8671 with a confidence interval of ± 0.0035 for the SVM classifier.

Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for SVM classifier is 108.09 seconds, which is almost two minutes.

3.3: Decision Tree
For the Decision Tree classifier, I chose the following hyperparameters to be tested (total of 4 configurations):

criterion = [gini, entropy]
max_depth = [5, 50]

The results are:



```
Evaluating Decision Tree:
1) config: {'decisiontreeclassifier__criterion': 'gini', 'decisiontreeclassifier__max_depth': 5}
Average F1 score: 0.7620, std: 0.0156
2) config: {'decisiontreeclassifier__criterion': 'gini', 'decisiontreeclassifier__max_depth': 50}
Average F1 score: 0.7801, std: 0.0213
3) config: {'decisiontreeclassifier__criterion': 'entropy', 'decisiontreeclassifier__max_depth': 5}
Average F1 score: 0.7556, std: 0.0181
4) config: {'decisiontreeclassifier__criterion': 'entropy', 'decisiontreeclassifier__max_depth': 50}
Average F1 score: 0.7865, std: 0.0153

F1 score for this configuration: 0.7993 ± 0.0060
Time taken: 50.85 seconds
```

Figure 10: Results of Decision Tree

For clarity, I have converted the results into a LaTeX table:

Config #	**criterion**	**max_depth**	**F1 score**	**Std**
1	gini	5	0.7620	0.0156
2	gini	50	0.7801	0.0213
3	entropy	5	0.7556	0.0181
4	entropy	50	0.7865	0.0153

Table 4: Results of Decision Tree

We can see that the choice of criterion does not affect the F1 score much. However, if we increase the max depth, there is a slight improvement on the F1 scores for both criterions.

From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.7993 with a confidence interval of ± 0.0060 for the Decision Tree classifier.

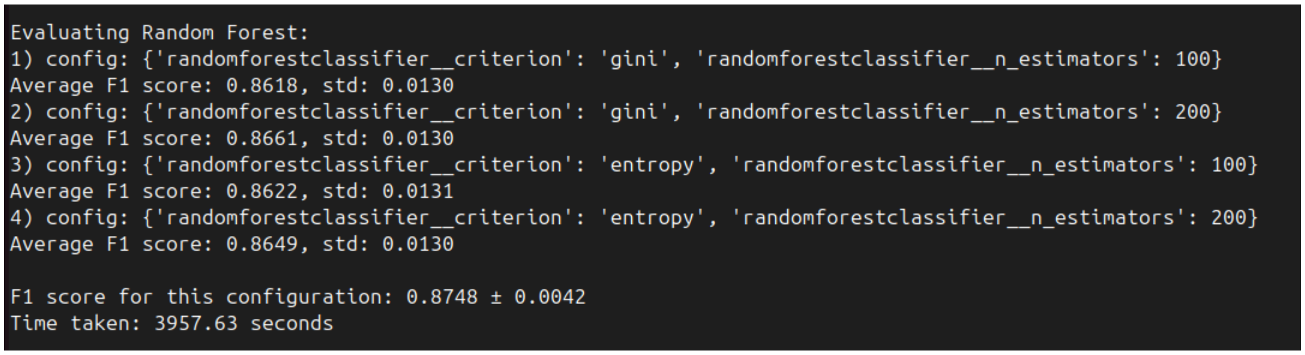
Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for Decision Tree classifier is 50.85 seconds, which is almost one minute.

3.4: Random Forest

Now, we start with the stochastic classifiers, which will take much more running time because of the additional loop. For the Random Forest classifier, I chose the following hyperparameters to be tested (total of 4 configurations):

criterion = [gini, entropy]
n_estimators = [100, 200]

The results are:



```
Evaluating Random Forest:
1) config: {'randomforestclassifier__criterion': 'gini', 'randomforestclassifier__n_estimators': 100}
Average F1 score: 0.8618, std: 0.0130
2) config: {'randomforestclassifier__criterion': 'gini', 'randomforestclassifier__n_estimators': 200}
Average F1 score: 0.8661, std: 0.0130
3) config: {'randomforestclassifier__criterion': 'entropy', 'randomforestclassifier__n_estimators': 100}
Average F1 score: 0.8622, std: 0.0131
4) config: {'randomforestclassifier__criterion': 'entropy', 'randomforestclassifier__n_estimators': 200}
Average F1 score: 0.8649, std: 0.0130

F1 score for this configuration: 0.8748 ± 0.0042
Time taken: 3957.63 seconds
```

Figure 11: Results of Random Forest

For clarity, I have converted the results into a LaTeX table:

Config #	**criterion**	**n_estimators**	**F1 score**	**Std**
1	gini	100	0.8618	0.0130
2	gini	200	0.8661	0.0130
3	entropy	100	0.8622	0.0131
4	entropy	200	0.8649	0.0130

Table 5: Results of Random Forest

We can see that the choice of criterion or n_estimators does not really affect the F1 score much, as the mean and standard deviation of the scores are nearly the same.

From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.8748 with a confidence interval of ± 0.0042 for the Random Forest classifier.

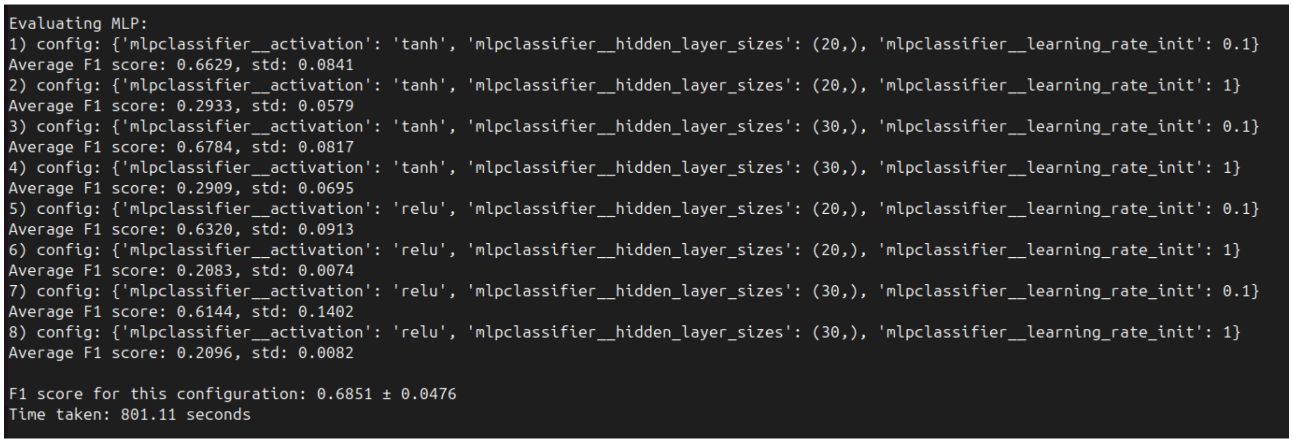
Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for Random Forest classifier is 3957.63 seconds, which is 65.96 minutes, a significant increase in running time when compared to the non-stochastic classifiers.

3.5: MLP

For the MLP classifier, I chose the following hyperparameters to be tested (total of 8 configurations):

activation_function = [tanh, relu]
hidden_layer_size = [20, 30]
learning_rate = [0.1, 1]

The results are:



```
Evaluating MLP:
1) config: {'mlpclassifier__activation': 'tanh', 'mlpclassifier__hidden_layer_sizes': (20,), 'mlpclassifier__learning_rate_init': 0.1}
Average F1 score: 0.6629, std: 0.0841
2) config: {'mlpclassifier__activation': 'tanh', 'mlpclassifier__hidden_layer_sizes': (20,), 'mlpclassifier__learning_rate_init': 1}
Average F1 score: 0.2933, std: 0.0579
3) config: {'mlpclassifier__activation': 'tanh', 'mlpclassifier__hidden_layer_sizes': (30,), 'mlpclassifier__learning_rate_init': 0.1}
Average F1 score: 0.6784, std: 0.0817
4) config: {'mlpclassifier__activation': 'tanh', 'mlpclassifier__hidden_layer_sizes': (30,), 'mlpclassifier__learning_rate_init': 1}
Average F1 score: 0.2909, std: 0.0695
5) config: {'mlpclassifier__activation': 'relu', 'mlpclassifier__hidden_layer_sizes': (20,), 'mlpclassifier__learning_rate_init': 0.1}
Average F1 score: 0.6320, std: 0.0913
6) config: {'mlpclassifier__activation': 'relu', 'mlpclassifier__hidden_layer_sizes': (20,), 'mlpclassifier__learning_rate_init': 1}
Average F1 score: 0.2083, std: 0.0074
7) config: {'mlpclassifier__activation': 'relu', 'mlpclassifier__hidden_layer_sizes': (30,), 'mlpclassifier__learning_rate_init': 0.1}
Average F1 score: 0.6144, std: 0.1402
8) config: {'mlpclassifier__activation': 'relu', 'mlpclassifier__hidden_layer_sizes': (30,), 'mlpclassifier__learning_rate_init': 1}
Average F1 score: 0.2096, std: 0.0082

F1 score for this configuration: 0.6851 ± 0.0476
Time taken: 801.11 seconds
```

Figure 12: Results of MLP

For clarity, I have converted the results into a LaTeX table:

Config #	**Activation**	**HL nodes**	**Learning Rate**	**F1 score**	**Std**
1	tanh	20	0.1	0.6629	0.0841
2	tanh	20	1	0.2933	0.0579
3	tanh	30	0.1	0.6784	0.0817
4	tanh	30	1	0.2909	0.0695
5	relu	20	0.1	0.6320	0.0913
6	relu	20	1	0.2083	0.0074
7	relu	30	0.1	0.6144	0.1402
8	relu	30	1	0.2096	0.0082

Table 6: Results of MLP

We can see that the choice of the learning rate greatly affects the F1 score. A lower learning rate means a higher score, and this might be because it is difficult for the higher learning rate to converge properly. The choice for hidden-layer nodes do not affect the F1 score by any significant amount. Using tanh as the activation function slightly increases the F1 score in all cases when compared to relu.

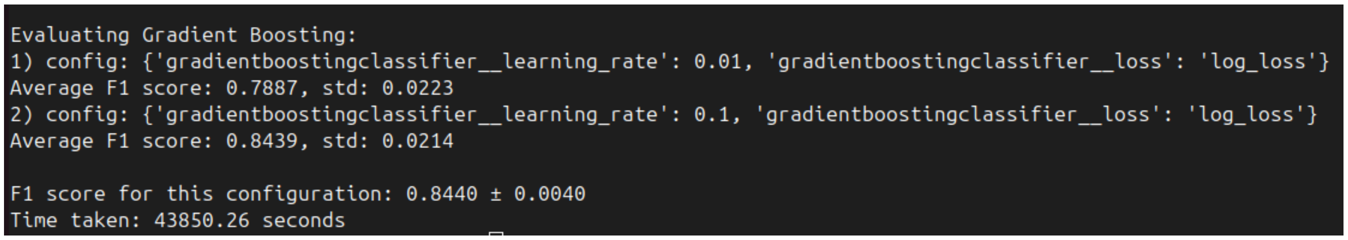
From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.6851 with a confidence interval of ± 0.0476 for the MLP classifier.

Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for MLP classifier is 801.11 seconds, which is 13.35 minutes.

3.6: Gradient Boosting
For the Gradient Boosting classifier, I chose the following hyperparameters to be tested (total of 2 configurations):

learning rate = [0.01, 0.1]
loss function = [log_loss]

Initially, I had planned to use more, but even two configurations took too much time to run so I did not increase them. The results are:



```
Evaluating Gradient Boosting:
1) config: {'gradientboostingclassifier__learning_rate': 0.01, 'gradientboostingclassifier__loss': 'log_loss'}
Average F1 score: 0.7887, std: 0.0223
2) config: {'gradientboostingclassifier__learning_rate': 0.1, 'gradientboostingclassifier__loss': 'log_loss'}
Average F1 score: 0.8439, std: 0.0214

F1 score for this configuration: 0.8440 ± 0.0040
Time taken: 43850.26 seconds
```

Figure 13: Results of Gradient Boosting

For clarity, I have converted the results into a LaTeX table:

Config #	**Learning Rate**	**Loss**	**F1 score**	**Std**
1	0.01	log_loss	0.7887	0.0223
2	0.1	log_loss	0.8439	0.0214

Table 7: Results of Gradient Boosting

We can see that the choice of the learning rate visibly affects the F1 score. An increased learning rate resulted in increased performance on the score.

From the best hyperparameter configs, when performing the outer cross-validation tests, we get an average F1 score of 0.8440 with a confidence interval of ± 0.0040 for the Gradient Boosting classifier.

Lastly, the time taken to completely run the inner and outer cross-validation loops and get the final results for Gradient Boosting classifier is 43850.26 seconds, which is 12.18 hours! This means I had to keep my laptop open and running the program overnight. This evaluation took an extremely long amount of time, even with only two hyperparameter configurations to test.

Now, I will talk about the final results of the classifiers. I have created a table with the outer cross-validation results from each classifier:

#	Classifier	F1 score	Conf int.	Time taken (s)
1	KNN	0.8493	0.0031	9.43
2	SVM	0.8671	0.0035	108.09
3	Decision Tree	0.7993	0.0060	50.85
4	Random Forest	0.8748	0.0042	3957.63
5	MLP	0.6851	0.0476	801.11
6	Gradient Boosting	0.8440	0.0040	43850.26

Table 8: Final results of each classifier

We can see that the Random Forest classifier gave the highest F1 score when compared to the others, with SVM coming in at a close second. As for the worst performance, MLP achieved an F1 score much lower than any other classifier. This might be due to the hyperparameters tested, and testing more hyperparameter configurations might give better results.

MLP also has the most wide confidence interval, this means that the results of MLP depend heavily on its initial conditions and it can make quite different models for each run.

As for the time taken, KNN took the least amount of time, which is several magnitudes lesser than the time taken by Gradient Boosting, even though they gave similar results in their F1 scores.

In the end, the decision for the best classifier is subjective. If the goal is to attain the highest F1 score, Random Forest might be the best choice. However, if we want a good F1 score but also want to keep efficiency and speed in consideration, we can go for the non-stochastic classifiers like KNN and SVM for a major improvement in speed for a slight sacrifice in F1 score.