CSE381: Introduction to Machine Learning

Final Project Phase 2

Spring 2023

Shaimaa Mohamed Ahmed 19P7484

Aya Ahmed Mohamed Gamal 19P1689

Rana Ahmed Abd elhalim 19P2468

Contents

Table of figures

Introduction

In this project we have data that will help us identify whether a person is going to recover from coronavirus symptoms or not based on some pre-defined standard symptoms. These symptoms are based on guidelines given by the World Health Organization (WHO).

Also we have 14 major variables which are our features that will be having an impact on whether someone has recovered or not, we will use these features to train our model and predict the labeled output which is the result, as the 0 value means person is recovered from corona and 1 means that he is died.

Our goal is to tune the hyper parameters that we will pass to different classifiers like decision trees and Multi-layer perceptron .By tuning the hyper parameters we can reach the best version of the model which is corresponding to the best results, we will measure the results through different metrics like the precision, recall, F1-score, and ROC/AUC curves.

1. Decision tree

Concept

It works by creating a tree-like model of decisions and their possible consequences. At the root of the tree, the algorithm considers the entire dataset and selects the feature that provides the best split. It then splits the data based on this feature, creating two or more branches. Each branch represents a decision based on the value of the selected feature. The algorithm continues this process recursively, selecting the best feature and splitting the data until it reaches a stopping condition.

The stopping condition may be a certain depth of the tree, a minimum number of instances in a leaf node, or another criterion. At the leaf nodes, the algorithm assigns a class label, or a prediction based on the majority class or the average value of the instances in that node.

Hyper parameter

1) **Maximum depth**: To limits the depth of the decision tree. Because when the maximum depth is high this cause overfitting and if the value is small this cause underfitting.
2) **Minimum samples for a split**: To set the minimum number of samples required to split a node. When the value is too high this lead to make tree so simple and when the value s too low this cause overfitting.
3) **Minimum samples for a leaf**: To set the minimum number of samples required to be in a leaf node. When the value is too high this lead to make tree so simple and when the value is too low this cause overfitting.
4) **Maximum leaf nodes**: To set the maximum number of leaf nodes in the tree. When its value is high this cause overfitting and if the value is small this cause underfitting
5) **Split criterion:** To set the criterion used to evaluate the quality of a split. The two most used criteria are Gini impurity and entropy.
6) **Maximum features**: To set the maximum number of features to consider when looking for the best split. When the value is high this cause overfitting and if the value is small this cause underfitting.

Steps

- We will apply the same steps as before and train our model by fitting the features on the x-axis and output on y-axis.
- We will try to predict using the cross-validation dataset and calculate the accuracy between the expected output and the original output by tuning the hyper parameter.
- Test on the testing data

Results before tunning parameters

First, we create an instance of the DecisionTreeClassifier class from scikit-learn to build a decision tree model. Then trains the decision tree model using the training data x_train and y_train. After that the fit method of the classifier is called and it recursively splits the data based on the features, using the criteria specified in the hyperparameters of the classifier. Finally uses the trained decision tree model to predict the class labels for the test data x_cv. The predict method of the classifier is called and it takes in the test data and returns the predicted labels for each instance. We calculate the accuracy then and the result is 97% and also we calculated using some other metric as roc curve and f1 score.

```
clf10 = DecisionTreeClassifier()
clf10 = clf10.fit(x_train,y_train)
#Predict the response for test dataset
y_pred10 = clf10.predict(x_cv)

#Accuracy
print("Accuracy:",metrics.accuracy_score(y_cv, y_pred10))

print(confusion_matrix(y_cv, y_pred10))
print("roc score",metrics.roc_auc_score(y_cv,y_pred10))
print(classification_report(y_cv, y_pred10))
```

[21] ✓ 0.0s

```
Accuracy: 0.9767441860465116
[[115   1]
 [  2  11]]
roc score 0.918766578249337
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       116
           1       0.92      0.85      0.88        13

    accuracy                           0.98       129
   macro avg       0.95      0.92      0.93       129
weighted avg       0.98      0.98      0.98       129
```

Results after tunning parameters manually

1) First, we change the criterion to use entropy and see the result, but the accuracy decreases to 96.8%. and tree to visualize tree using graphiz.

```
clf = DecisionTreeClassifier(criterion='entropy')
clf = clf.fit(x_train,y_train)
#Predict the response for test dataset
y_pred = clf.predict(x_cv)
```

```
#Accuracy
print("Accuracy:",metrics.accuracy_score(y_cv, y_pred))
```

```
Accuracy: 0.9689922480620154
```

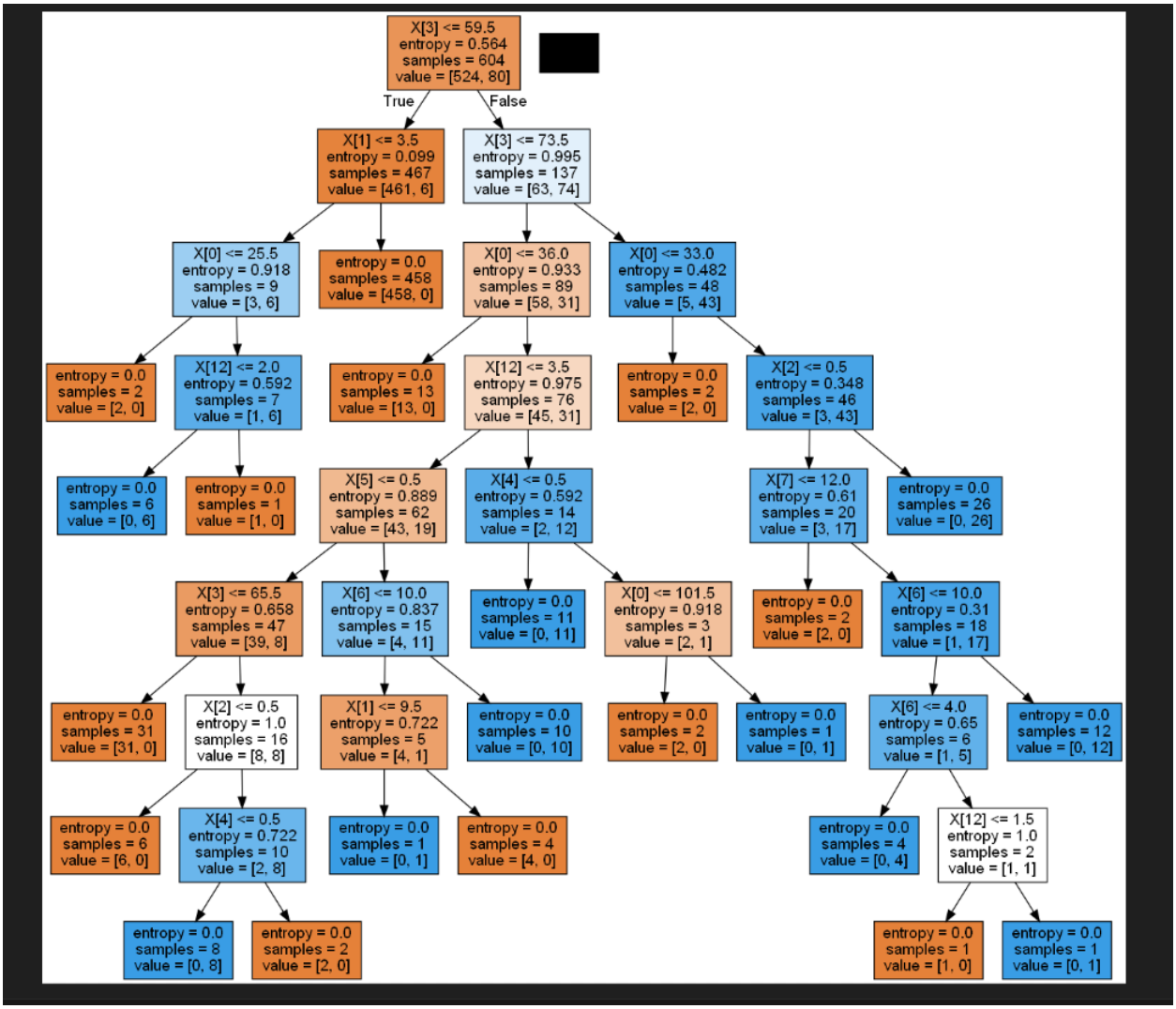
```
print(confusion_matrix(y_cv, y_pred))
```

```
[[115   1]
 [  3  10]]
```

```
print("roc score",metrics.roc_auc_score(y_cv,y_pred))
```

```
roc score 0.8803050397877984
```

```
from six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(clf3, out_file=dot_data,
                filled=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

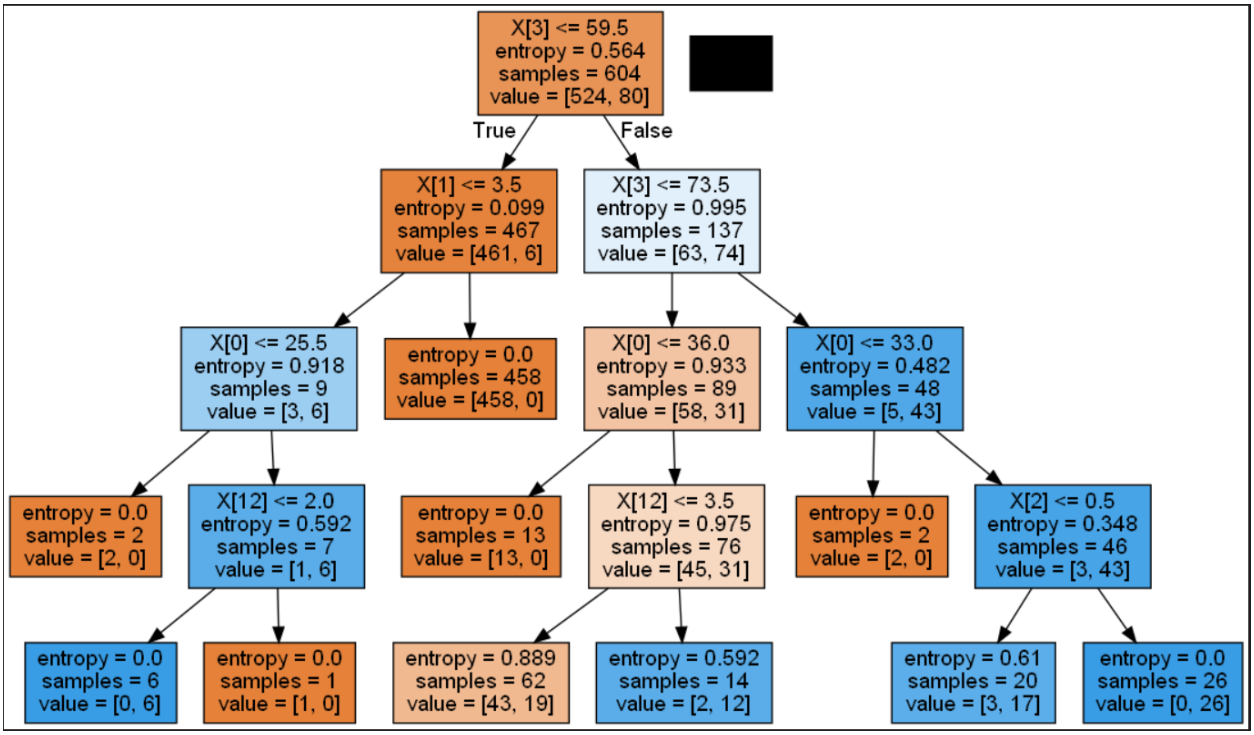


2) Second, we change the default value of maxdepth to 4 and we predict that the accuracy decreases as it goes to under fitting and the results was as we expected. The accuracy decreases to 93 % and value of roc decreases to 79%.

```
clf2 = DecisionTreeClassifier(criterion='entropy',max_depth=4)
clf2 = clf2.fit(x_train,y_train)
#Predict the response for test dataset
y_pred2 = clf2.predict(x_cv)
```

```
[10]
    #Accuracy
    print("Accuracy:",metrics.accuracy_score(y_cv, y_pred2))
    print("roc score",metrics.roc_auc_score(y_cv,y_pred2))
    print(classification_report(y_cv, y_pred2))
```

```
Accuracy: 0.937984496124031
roc score 0.7947612732095491
              precision    recall  f1-score   support

           0       0.96      0.97      0.97       116
           1       0.73      0.62      0.67        13

    accuracy                           0.94       129
   macro avg       0.84      0.79      0.82       129
weighted avg       0.93      0.94      0.94       129
```



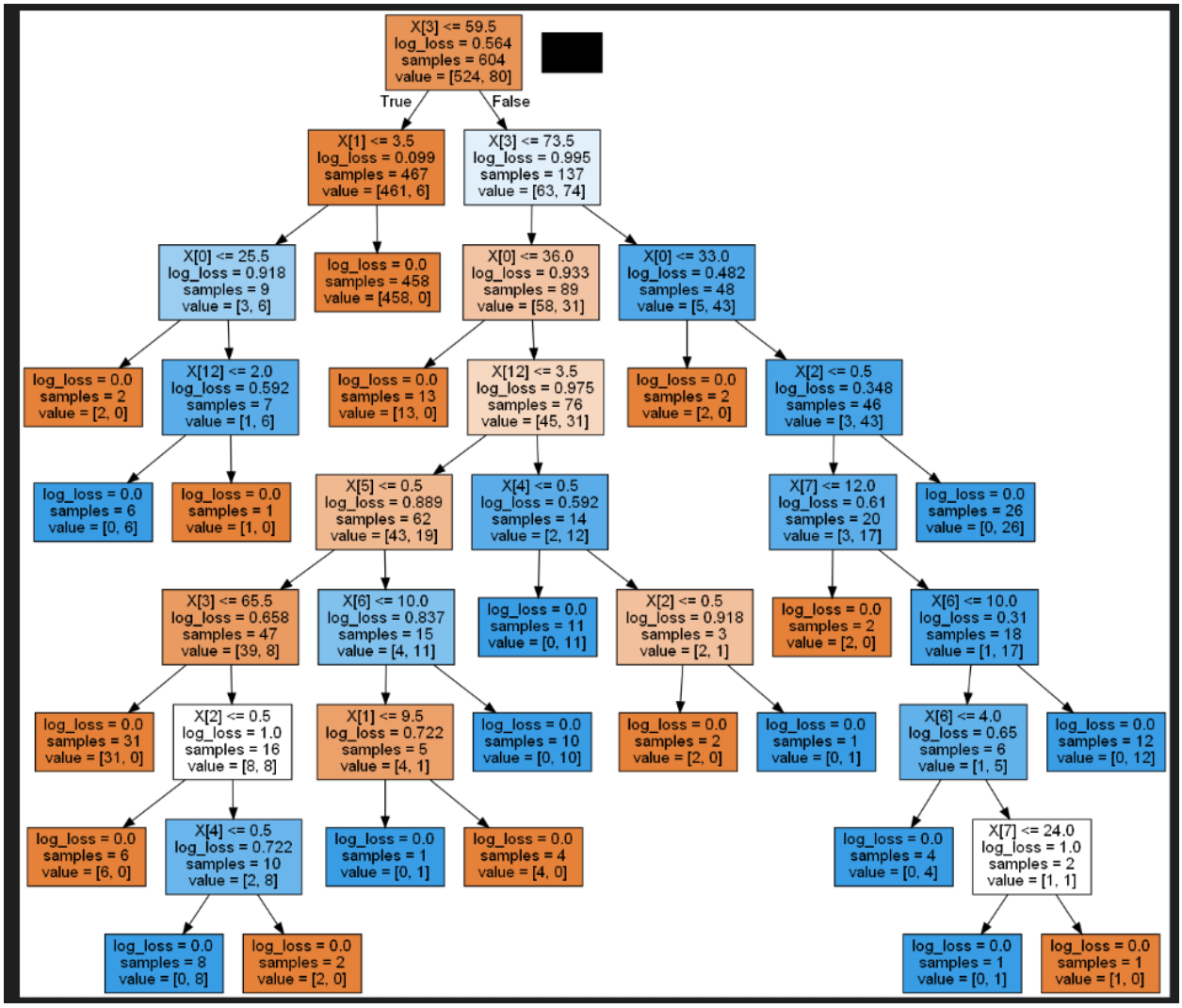
3) We changed this time the critration from entropy to logloss and print the accuracy, but it was the same as entropy also after visualizing the tree the are barely the same

```
[12]
clf3 = DecisionTreeClassifier(criterion='log_loss')
#log loss has higher accuracy
clf3 = clf3.fit(x_train,y_train)
#Predict the response for test dataset
y_pred3 = clf3.predict(x_cv)
```

```
[13]
#Accuracy
print("Accuracy:",metrics.accuracy_score(y_cv, y_pred3))
print("roc score",metrics.roc_auc_score(y_cv,y_pred3))
print(classification_report(y_cv, y_pred3))
```

```
Accuracy: 0.9689922480620154
roc score 0.8803050397877984
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       116
           1       0.91      0.77      0.83        13

    accuracy                           0.97       129
   macro avg       0.94      0.88      0.91       129
weighted avg       0.97      0.97      0.97       129
```

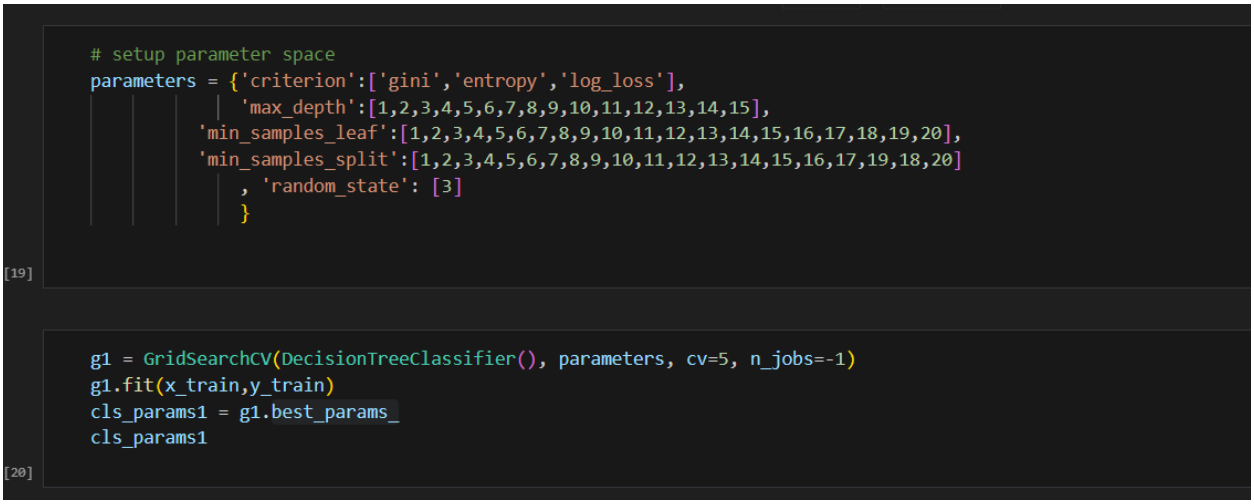
```
from six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(clf3, out_file=dot_data,
                filled=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```



4) There are more trials by adjusting max-features or change the critration used but they all perform results near to each other.

Results after tunning parameters using grid search

First, we define a dictionary called parameters that specifies the hyperparameter space to search over. Here, we have specified the values to try for the criterion, max_depth, min_samples_leaf, and min_samples_split hyperparameters. We have also set the random_state parameter to 3 which ensures reproducibility of the results. Then we create an instance of the GridSearchCV that performs an exhaustive search over the hyperparameter space specified in parameters to find the best combination of hyperparameters that results in the highest cross-validated performance. The first argument to GridSearchCV is the estimator, which in this case is the DecisionTreeClassifier. The cv argument specifies the number of cross-validation folds to use, and n_jobs specifies the number of parallel jobs to run. Finaly using best_params retrieves the best hyperparameters found by the GridSearchCV object and assigns them to the cls_params1 variable.



```
[19]
# setup parameter space
parameters = {'criterion':['gini','entropy','log_loss'],
              'max_depth':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
             'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],
             'min_samples_split':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,18,20]
              , 'random_state': [3]
              }
```

```
[20]
g1 = GridSearchCV(DecisionTreeClassifier(), parameters, cv=5, n_jobs=-1)
g1.fit(x_train,y_train)
cls_params1 = g1.best_params_
cls_params1
```

And here is the best combination of hyper parameters to obtain the best results for our data. And this combination gives accuracy 95% and roc value 79.9%. but reached a better results when we tun hyper parameters manually

```
{'criterion': 'entropy',
 'max_depth': 8,
 'min_samples_leaf': 3,
 'min_samples_split': 14,
 'random_state': 3}
```

```
[21]
    model = g1.best_estimator_
    y_pred6 = model.predict(x_cv)
    print('accuracy score: %.2f' % metrics.accuracy_score(y_cv,y_pred6))
    print('precision score: %.2f' % metrics.precision_score(y_cv,y_pred6))
    print('recall score: %.2f' % metrics.recall_score(y_cv,y_pred6))
    print('f1 score: %.2f' % metrics.f1_score(y_cv,y_pred6))
    print("roc score",metrics.roc_auc_score(y_cv,y_pred6))
    # print('computation time: %.2f' % duration)

...  accuracy score: 0.95
     precision score: 0.80
     recall score: 0.62
     f1 score: 0.70
     roc score 0.7990716180371352
```

Results after tunning parameters using bayes search.

First, we define a dictionary called parameters that specifies the hyperparameter space to search over. Here, we have specified the values to try for the criterion, max_depth, min_samples_leaf, min_samples_split, random_state, and max_leaf_nodes hyperparameters. Then, create an instance of the BayesSearchCV to perform a hyperparameter search using Bayesian optimization, which is a probabilistic approach to finding the optimal hyperparameters. The first argument to BayesSearchCV is the estimator, which in this case is the DecisionTreeClassifier. The cv argument specifies the number of cross-validation folds to use, and n_iter specifies the number of iterations to perform the search. The random_state argument is used for reproducibility, and n_jobs specify the number of parallel jobs to run. Then print the best combination of parameters to get best results.

This time the results is higher than using grid search as the accuracy is 96% and roc value is 87% but still tunning parameters manually is the best results

```
from skopt import BayesSearchCV
parameters = {
    'criterion':['gini','entropy','log_loss'],
                'max_depth':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
            'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],
            'min_samples_split':[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,18,20]
                , 'random_state': [3],
                'max_leaf_nodes': [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,18,20]}

# create an instance of the bayesian search object
b1 = BayesSearchCV(DecisionTreeClassifier(), parameters, cv=5, n_iter=30, random_state=42, n_jobs=-1)

# conduct randomized search over the parameter space
b1.fit(x_train,y_train)


# show best parameter configuration found for classifier
cls_params3 = b1.best_params_

cls_params3
```

[23]

```
OrderedDict([('criterion', 'log_loss'),
             ('max_depth', 14),
             ('max_leaf_nodes', 18),
             ('min_samples_leaf', 2),
             ('min_samples_split', 6),
             ('random_state', 3)])
```

```
model = b1.best_estimator_
y_pred7 = model.predict(x_cv)
print('accuracy score: %.2f' % metrics.accuracy_score(y_cv,y_pred7))
print('precision score: %.2f' % metrics.precision_score(y_cv,y_pred7))
print('recall score: %.2f' % metrics.recall_score(y_cv,y_pred7))
print('f1 score: %.2f' % metrics.f1_score(y_cv,y_pred7))
print("roc score",metrics.roc_auc_score(y_cv,y_pred7))
```

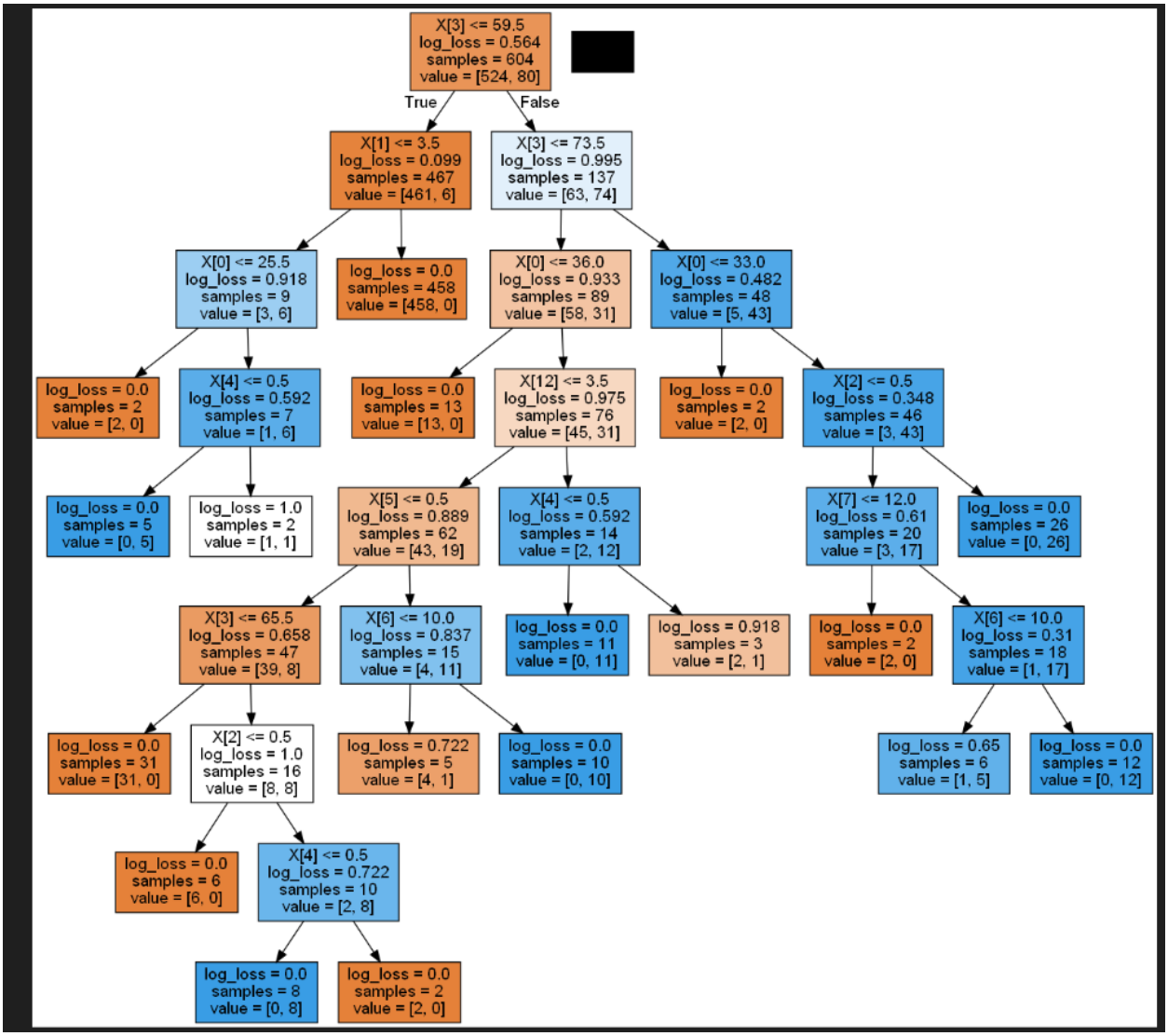
24]

```
accuracy score: 0.96
precision score: 0.83
recall score: 0.77
f1 score: 0.80
roc score 0.8759946949602121
```

```
from six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

5]

Visualizing the tree after using bayes search and choosing the best hyper parameters



2.0 Multilayer Perceptron

2.1 Concept

A multilayer perceptron (MLP) is a type of artificial neural network (ANN) that consists of multiple layers of nodes (neurons). It is a feed forward neural network, meaning the information flows in one direction from the input layer to the output layer. MLPs are commonly used for various machine learning tasks, including classification, regression, and pattern recognition.

The structure of an MLP typically consists of an input layer, one or more hidden layers, and an output layer. Each layer is composed of a set of neurons or nodes. The neurons in the input layer receive the input data, and the neurons in the output layer produce the final output or prediction. The hidden layers are intermediary layers that perform computations and introduce non-linear transformations to the input data.

The nodes in each layer are interconnected with weighted connections. Each node receives inputs from the previous layer, applies an activation function to the weighted sum of these inputs, and produces an output. The activation function introduces non-linearity and enables the MLP to learn complex patterns in the data.

During training, an MLP adjusts the weights of its connections to minimize the difference between the predicted output and the actual output, using a process called back propagation, also there is forward propagation. Back propagation calculates the gradients of the loss function with respect to the weights and updates the weights using an optimization algorithm (e.g., gradient descent) to iteratively improve the model's performance.

2.2 Hyper parameters using sklearn

Hyper-Parameters:

- **Number of layers:** The choice of the number of hidden layers depends on the complexity of the problem and the amount of data available. Increasing the number of hidden layers allows the MLP to learn more complex representations but, may also increase the risk of overfitting.
- **Number of neurons per layer:** Each layer consists of a certain number of neurons. The number of neurons determines the capacity and expressive power of the MLP. Increasing the number of neurons allows the MLP to learn more intricate patterns.
- **Activation Function:** Activation functions introduce non-linearity into the MLP and determine the output of each neuron. Common activation functions used in MLPs include sigmoid, tanh, and rectified linear units (ReLU).
- **Optimizer:** The optimizer is responsible for updating the weights of the MLP based on the computed gradients. Common optimization algorithms used in MLPs include Adam & Adagrad.
- **Learning rate :** It determines how the learning rate changes during training. Options include 'constant' (the learning rate remains fixed), 'invscaling' (the learning rate decreases gradually over time), and 'adaptive' (the learning rate is adjusted based on the validation score). The default value is 'constant'.

These hyper parameters need to be carefully tuned to achieve good performance on a given Classification task.

2.3 Steps

1. Make the necessary library imports that we will use and read the data.csv file
2. Make list of the features dropping the output column.

   feature_cols= ['location', 'country' ,'gender' ,'age' ,'vis_wuhan', 'from_wuhan', 'symptom1' ,'symptom2' ,'symptom3' ,'symptom4' ,'symptom5' ,'symptom6', 'diff_sym_hos']
3. x=train_data[feature_cols] , and y=train_data.result which is the column of output
4. Split the train data into 70 % training data, 15 % validation data , and 15 % testing.

5. Train the MLP classifier model using this code and test on validation data

```
clf_cv = MLPClassifier(random_state=1)
#Train the model using the training sets
clf_cv.fit(x_train, y_train)
#Predict the response on validation data
y_pred_cv = clf_cv.predict(x_validate)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_validate, y_pred_cv))
print("Precision:",metrics.precision_score(y_validate, y_pred_cv))
print("F1 score:",metrics.f1_score(y_validate, y_pred_cv))
print("Recall:",metrics.recall_score(y_validate, y_pred_cv))
print("ROC score:",metrics.roc_auc_score(y_validate, y_pred_cv))
```

✓ 0.3s Python

```
MLP classifier results:
Accuracy: 0.8992248062015504
Precision: 0.6818181818181818
F1 score: 0.6976744186046512
Recall: 0.7142857142857143
ROC score: 0.8247354497354499
```

Figure 1 Results before hyper parameter tuning.

1. Start tuning the hyper parameters instead of the default values given by sklearn

   Default values
   (hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001,)

   First we will tune the hidden_layer_sizes

   hidden_layer_sizes: This parameter specifies the number of neurons in each hidden layer. In the example above, we have two hidden layers with 5 and 5 neurons, respectively. You can adjust these values based on your specific problem.

```
clf1_cv = MLPClassifier(hidden_layer_sizes=(5,5), max_iter=200,random_state=42)
#Train the model using the training sets
clf1_cv.fit(x_train, y_train)
#Predict the response on validation data
y_pred_cv1 = clf1_cv.predict(x_validate)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_validate, y_pred_cv1))
print("Precision:",metrics.precision_score(y_validate, y_pred_cv1))
print("F1 score:",metrics.f1_score(y_validate, y_pred_cv1))
print("Recall:",metrics.recall_score(y_validate, y_pred_cv1))
print("ROC score:",metrics.roc_auc_score(y_validate, y_pred_cv1))
```

[59] ✓ 0.3s Python

```
MLP classifier results:
Accuracy: 0.8372093023255814
Precision: 0.0
F1 score: 0.0
Recall: 0.0
ROC score: 0.5
```

When we tried tuning two hidden_layers with 5 and 5 neurons results were as shown above Accuracy is 0.837 and roc_score is 0.5 which are small values and not achieving nor good results nor performance.

2. Increase value of neurons in the layers which gives the best result

Reason for choosing the best results

The accuracy here is 0.97 which is an amazing result more over ROC score is 0.89

```
clf2 = MLPClassifier(batch_size=10,hidden_layer_sizes=(15,15), max_iter=200  ,random_state=42)
#Train the model using the training set
clf2.fit(x_train, y_train)
#Predict the response for test dataset
y_pred2 = clf2.predict(x_test)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred2))
print("Precision:",metrics.precision_score(y_test, y_pred2))
print("F1 score:",metrics.f1_score(y_test, y_pred2))
print("Recall:",metrics.recall_score(y_test, y_pred2))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred2))
```

✓ 3.2s Pytho

```
MLP classifier results:
Accuracy: 0.9769230769230769
Precision: 1.0
F1 score: 0.88
Recall: 0.7857142857142857
ROC score: 0.8928571428571428
```

3. When we tried to increase layers with relatively large number of neurons

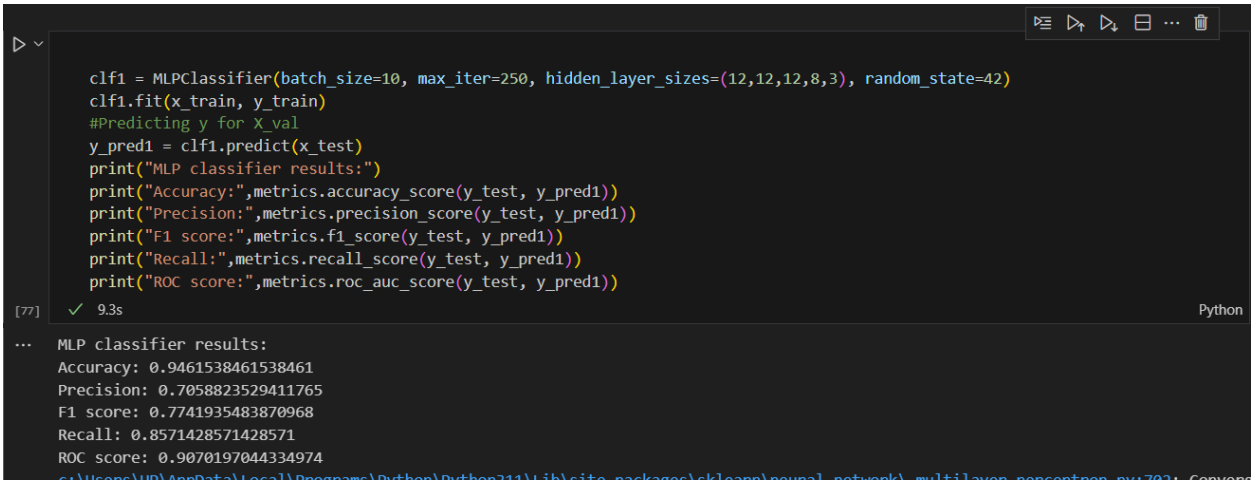
```
clf = MLPClassifier(hidden_layer_sizes=(20,20,10,5), max_iter=200,random_state=42)
#Train the model using the training sets
clf.fit(x_train, y_train)
#Predict the response for test dataset
y_pred = clf.predict(x_test)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred))
print("F1 score:",metrics.f1_score(y_test, y_pred))
print("Recall:",metrics.recall_score(y_test, y_pred))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred))
```

[70] ✓ 0.3s Python

```
MLP classifier results:
Accuracy: 0.9307692307692308
Precision: 0.7272727272727273
F1 score: 0.64
Recall: 0.5714285714285714
ROC score: 0.7727832512315271
```

All results are enhanced as the hidden layers become 4 layers with values 20,10,10,5 and here we noticed that accuracy increased to 93 % and roc score to 77 % which is a great difference.

4. we increased another layer and making batch size and more iterations



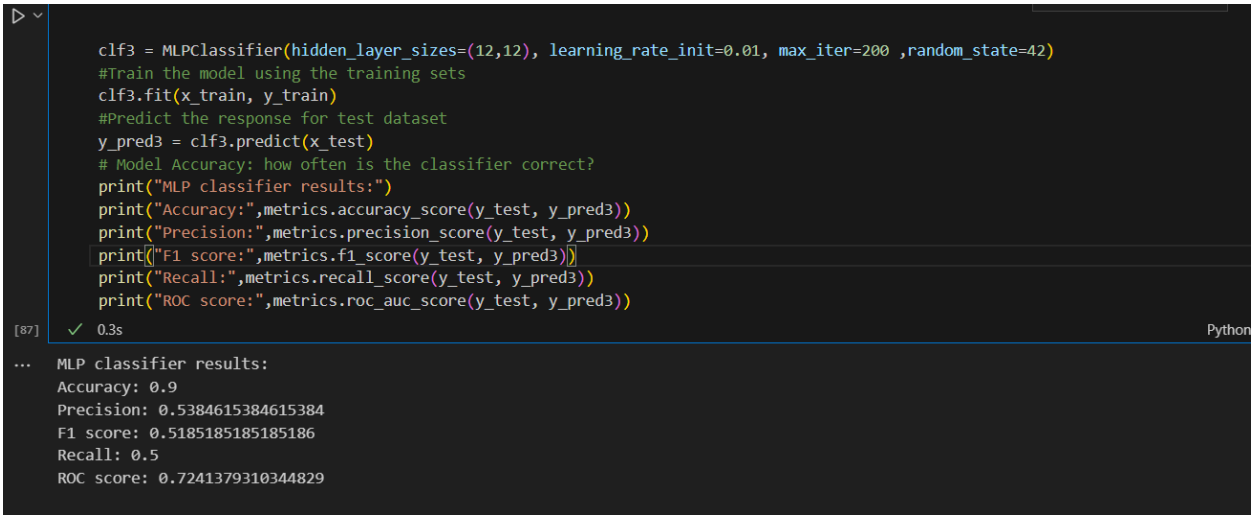
```
clf1 = MLPClassifier(batch_size=10, max_iter=250, hidden_layer_sizes=(12,12,12,8,3), random_state=42)
clf1.fit(x_train, y_train)
#Predicting y for X_val
y_pred1 = clf1.predict(x_test)
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred1))
print("Precision:",metrics.precision_score(y_test, y_pred1))
print("F1 score:",metrics.f1_score(y_test, y_pred1))
print("Recall:",metrics.recall_score(y_test, y_pred1))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred1))
```

[77] ✓ 9.3s Python

```
MLP classifier results:
Accuracy: 0.9461538461538461
Precision: 0.7058823529411765
F1 score: 0.7741935483870968
Recall: 0.8571428571428571
ROC score: 0.9070197044334974
```

In conclusion, by increasing the number of layers , or making small number of layers with large neurons the accuracy of model and roc score increase.

Tuning the Learning rate_init



```
clf3 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.01, max_iter=200 ,random_state=42)
#Train the model using the training sets
clf3.fit(x_train, y_train)
#Predict the response for test dataset
y_pred3 = clf3.predict(x_test)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred3))
print("Precision:",metrics.precision_score(y_test, y_pred3))
print("F1 score:",metrics.f1_score(y_test, y_pred3))
print("Recall:",metrics.recall_score(y_test, y_pred3))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred3))
```

[87] ✓ 0.3s Python

```
MLP classifier results:
Accuracy: 0.9
Precision: 0.5384615384615384
F1 score: 0.5185185185185186
Recall: 0.5
ROC score: 0.7241379310344829
```

The learning_rate_init parameter in the MLPClassifier of scikit-learn is used to set the initial learning rate for weight updates during training. It specifies the step size at the beginning of the optimization process.

During the training of an MLP, the weights and biases are updated iteratively to minimize the loss function. The learning rate determines the magnitude of these updates.

Increasing the learning_rate_init

```
clf3 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.1 ,random_state=42)
#Train the model using the training sets
clf3.fit(x_train, y_train)
#Predict the response for test dataset
y_pred3 = clf3.predict(x_test)
# Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred3))
print("Precision:",metrics.precision_score(y_test, y_pred3))
print("F1 score:",metrics.f1_score(y_test, y_pred3))
print("Recall:",metrics.recall_score(y_test, y_pred3))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred3))
```

✓ 0.1s

```
MLP classifier results:
Accuracy: 0.8923076923076924
Precision: 0.0
F1 score: 0.0
Recall: 0.0
ROC score: 0.5
```

A higher learning rate allows for larger weight updates, potentially leading to faster convergence but also increasing the risk of overshooting the optimal solution. Conversely, a lower learning rate results in smaller weight updates, which may slow down convergence but could provide more precise fine-tuning.

Change in the activation function instead of default (relu)

```
clf4 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.01, max_iter=250 , activation='relu',random_state=42)

clf4.fit(x_train, y_train)
#Predict the response for test dataset
y_pred4 = clf4.predict(x_test)
 # Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred4))
print("Precision:",metrics.precision_score(y_test, y_pred4))
print("F1 score:",metrics.f1_score(y_test, y_pred4))
print("Recall:",metrics.recall_score(y_test, y_pred4))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred4))
```

✓ 0.2s Python

```
MLP classifier results:
Accuracy: 0.9
Precision: 0.5384615384615384
F1 score: 0.5185185185185186
Recall: 0.5
ROC score: 0.7241379310344829
```

```
clf5 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.01, max_iter=200 , activation='logistic',random_state=42)
 #Train the model using the training sets
clf5.fit(x_train, y_train)
#Predict the response for test dataset
y_pred5 = clf5.predict(x_test)
 # Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred5))
print("Precision:",metrics.precision_score(y_test, y_pred5))
print("F1 score:",metrics.f1_score(y_test, y_pred5))
print("Recall:",metrics.recall_score(y_test, y_pred5))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred5))
```

✓ 0.2s Python

```
MLP classifier results:
Accuracy: 0.9230769230769231
Precision: 0.6428571428571429
F1 score: 0.6428571428571429
Recall: 0.6428571428571429
ROC score: 0.7998768472906403
```

The logistic activation gives good results with making other parameters constant like 2 layers each one is 12 neurons , learning_rate_init =0.01 and max_iterations 200.

Accuracy is 92 % and ROC_score is 79% .

Tuning using tanh as an activation function

```
clf6 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.01, max_iter=200 , activation='tanh',random_state=42)
 #Train the model using the training sets
clf6.fit(x_train, y_train)
 #Predict the response for test dataset
y_pred6 = clf6.predict(x_test)
 # Model Accuracy: how often is the classifier correct?
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred6))
print("Precision:",metrics.precision_score(y_test, y_pred6))
print("F1 score:",metrics.f1_score(y_test, y_pred6))
print("Recall:",metrics.recall_score(y_test, y_pred6))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred6))
```

✓ 0.1s Python

```
MLP classifier results:
Accuracy: 0.8923076923076924
Precision: 0.5
F1 score: 0.22222222222222224
Recall: 0.14285714285714285
ROC score: 0.562807881773399
```

Tune using learning_rate to be adaptive

learning_rate: It determines how the learning rate changes during training. Options include 'constant' (the learning rate remains fixed), 'invscaling' (the learning rate decreases gradually over time), and 'adaptive' (the learning rate is adjusted based on the validation score). The default value is 'constant'.

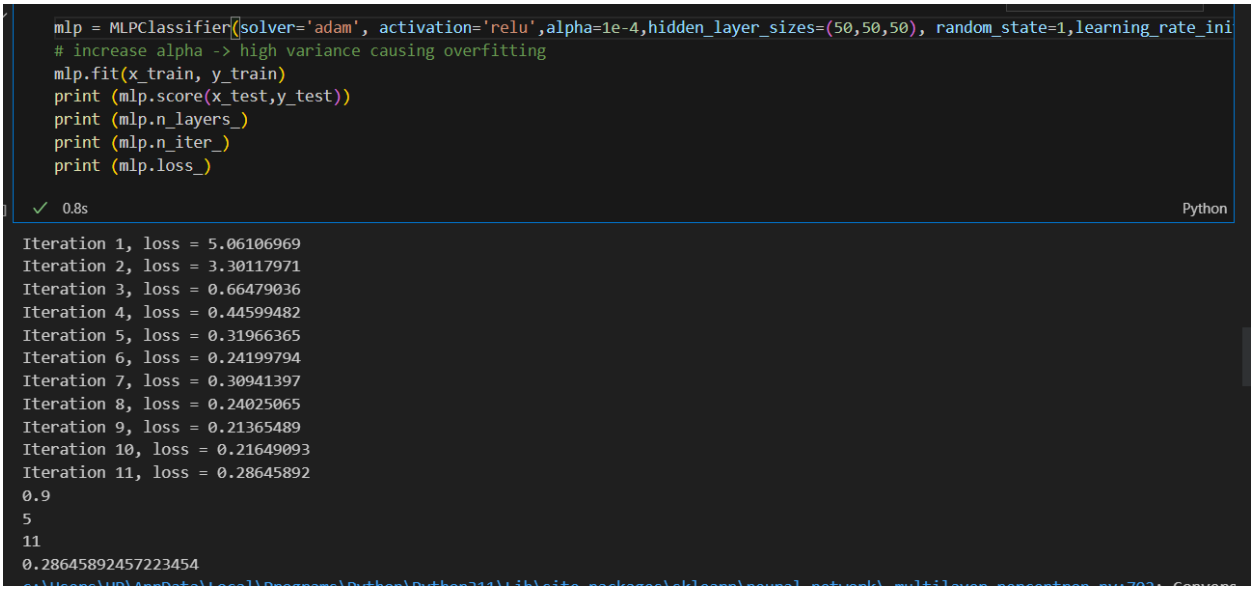
```
clf7 = MLPClassifier(hidden_layer_sizes=(12,12), learning_rate_init=0.01, max_iter=200 , activation='logistic',learning_rate='a
clf7.fit(x_train, y_train)
y_pred7 = clf7.predict(x_test)
print("MLP classifier results:")
print("Accuracy:",metrics.accuracy_score(y_test, y_pred7))
print("Precision:",metrics.precision_score(y_test, y_pred7))
print("F1 score:",metrics.f1_score(y_test, y_pred7))
print("Recall:",metrics.recall_score(y_test, y_pred7))
print("ROC score:",metrics.roc_auc_score(y_test, y_pred7))
```

[115] ✓ 0.2s Python

```
MLP classifier results:
Accuracy: 0.9230769230769231
Precision: 0.6428571428571429
F1 score: 0.6428571428571429
Recall: 0.6428571428571429
ROC score: 0.7998768472906403
```

Adding more hyperparameters like alpha , and optimization solver to be adam , and Making the learning_rate_init as before 0.01

The alpha parameter represents the regularization strength and takes a positive float value. It is also referred to as the L2 penalty or weight decay. The higher the value of alpha, the stronger the regularization effect.



The solver parameter is used to specify the algorithm used for weight optimization during the training of the neural network. The solver determines how the MLP updates its weights and biases to minimize the loss function.

adam': This solver uses an adaptive learning rate and is inspired by the Adam optimization algorithm. It is known for its efficiency and good performance on various types of data.

'adam' is often a safe choice as it generally works well in many scenarios.

The optimal value for alpha depends on your specific dataset and problem, so it is often necessary to experiment with different values or perform hyper parameter tuning to find the best setting. But when it gets smaller, it gives us better results.

Here we find that the loss is 0.28 which is an outstanding results , means that this tuning is done in an efficient way and enhancing the performance of the model.

Trying GridSearchCV () for tuning

```
from sklearn.neural_network import MLPClassifier
mlp1 = MLPClassifier(max_iter=100)
parameter_space = {
    'hidden_layer_sizes': [(50,50,50), (50,100,50), (100,),(10,10),(15,15)],
    'activation': ['tanh', 'relu','logistic'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant','adaptive'],
    'learning_rate_init':[ 0.01,0.1,0.001,0.0001 ]
}
```

✓ 0.2s Python

```
gridsearch = GridSearchCV(mlp1, parameter_space, n_jobs=-1, cv=3)
gridsearch.fit(x_train, y_train)
# Best paramete set
print('Best parameters found:\n', gridsearch.best_params_)
```

✓ 2m 38.5s Python

```
Best parameters found:
 {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'learning_rate_init': 0.0
```

We will that the hyper parameters among this parameter space are

{'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 50, 50), 'learning_rate': 'adaptive', 'learning_rate_init': 0.001, 'solver': 'adam'}

```
y_true, y_precept = y_test , gridsearch.predict(x_test)
from sklearn.metrics import classification_report
print('Results on the test set:')
print(classification_report(y_true, y_precept))
```

✓ 0.0s Python

```
Results on the test set:
              precision    recall  f1-score   support

           0       0.95      0.94      0.94       116
           1       0.53      0.57      0.55        14

    accuracy                           0.90       130
   macro avg       0.74      0.76      0.75       130
weighted avg       0.90      0.90      0.90       130
```

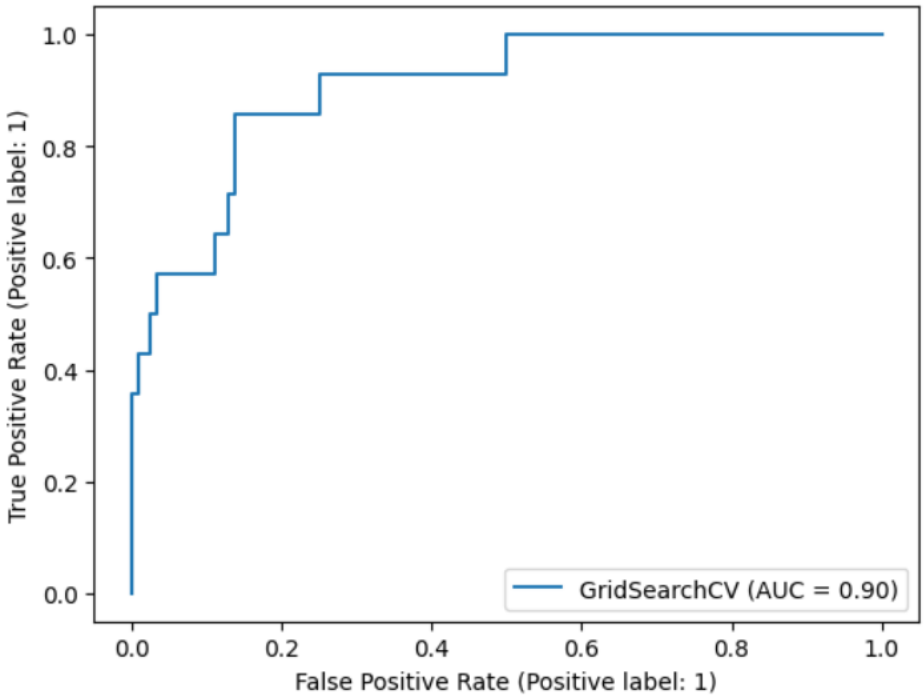
```
print("Accuracy:",metrics.accuracy_score(y_test, y_precept))
print("ROC_Score:",metrics.roc_auc_score(y_test, y_precept))
```

✓ 0.0s Python

```
Accuracy: 0.9
ROC_Score: 0.7555418719211823
```

The gridSearch results are good trying all possible combinations from these hyper parameters giving the final results for accuracy to be 90 % and roc score to be 75 %.

Plotting roc curve



Showing the confusion Matrix

```
print(confusion_matrix(y_test, y_precept))
pd.crosstab(y_test, y_precept, rownames=['True'], colnames=['Predicted'], margins=True)
```

✓ 0.7s

```
[[109   7]
 [  6   8]]
```

Predicted	0	1	All
True			
0	109	7	116
1	6	8	14
All	115	15	130

True positive is 109 out of 116, and false negative are 6 out of 14 which is a good result to this algorithm, as we are concerned with the false negatives and the result is amazing as only 6 where predicted to be not ill while actually they are ill.

Using RandomizedSearchCv()

RandomizedSearchCV
estimator: MLPClassifier
MLPClassifier

```
print('Best parameters found:\n', randomSearch.best_params_)
```

✓ 0.0s Python

```
Best parameters found:
 {'solver': 'adam', 'learning_rate_init': 0.1, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100,), 'alpha': 0.05, 'activatic
```

```
y_true1, y_precept1 = y_test , randomSearch.predict(x_test)
from sklearn.metrics import classification_report
print('Results on the test set:')
print(classification_report(y_true1, y_precept1))
```

✓ 0.0s Python

```
Results on the test set:
              precision    recall  f1-score   support

           0       0.90      0.97      0.94       116
           1       0.40      0.14      0.21        14

    accuracy                           0.88       130
   macro avg       0.65      0.56      0.57       130
weighted avg       0.85      0.88      0.86       130
```

Best parameters are

{'solver': 'adam', 'learning_rate_init': 0.1, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100,), 'alpha': 0.05, 'activation': 'relu'}

Printing the accuracy and ROC

```
print("Accuracy:",metrics.accuracy_score(y_test, y_precept1))
print("ROC CURVE :",metrics.roc_auc_score(y_test, y_precept1))
```

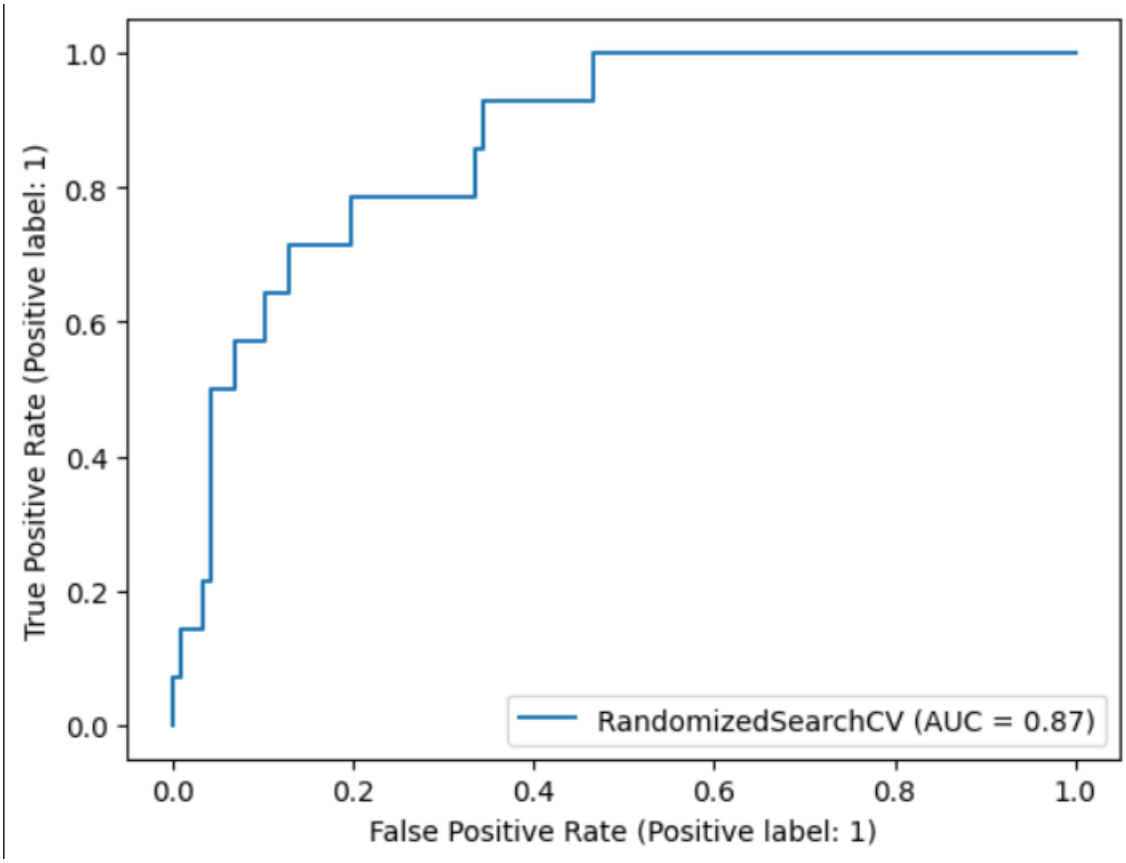
✓ 0.0s Python

```
Accuracy: 0.8846153846153846
ROC CURVE : 0.5584975369458128
```

Accuracy decreased in comparison to the GridSearchCV() algorithm and also Roc .

Their values are 88 % accuracy and approximately 56 % for roc score.

Plotting ROC_auc curve



Showing Confusion matrix

```
print(confusion_matrix(y_test, y_precept1))
pd.crosstab(y_test, y_precept1, rownames=['True'], colnames=['Predicted'], margins=True)
```

✓ 0.2s

```
[[113   3]
 [ 12   2]]
```

Predicted	0	1	All
True			
0	113	3	116
1	12	2	14
All	125	5	130

True positive is 113 out of 116, and false negative are 12 out of 14 which is not a good result to this algorithm, as we are concerned with the false negatives and the result is 12 where predicted to be not ill while actually they are ill , this results need more learning to get enhanced.

Compare gridSearchCV vs RandomizedSearchCV results on test data

Algorithm used for tuning	**gridSearchCV**	**RandomizedSearchCV**
Accuracy	**0.9**	**0.88**
ROC	**0.755**	**0.558**
precision	**0.53**	**0.4**
F1-score	**0.55**	**0.21**
Recall	**0.57**	**0.14**
AUC	**0.90**	**0.87**
Hyperparameters	{'solver': 'adam', 'learning_rate_init': 0.1, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100,), 'alpha': 0.05, 'activation': 'relu'}	{'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (50, 100, 50), 'learning_rate': 'constant', 'learning_rate_init': 0.0001, 'solver': 'adam'}

Reason for choosing the final results

- GridSearch is the best algorithm to be used in tuning as it gives best values comparing it to randomizedSearchCV , it gives accuracy 90 % , and roc_score 0.755 which are outstanding results.

2. Multi-layer perceptron using tensor flow library

1. Build the model with 3 layers in which the layers consist of 256 neurons, 128 neurons and 10 neurons respectively. Their activation function is ‘sigmoid’.

```
model = Sequential([
    # dense layer 1
    Dense(256, activation='sigmoid'),

    # dense layer 2
    Dense(128, activation='sigmoid'),

    # output layer
    Dense(10, activation='sigmoid'),
])
```

Figure 2: Building Neural Network model

2. Compile the model to try Adam’s Algorithm that automatically increases alpha to take bigger steps to get the minimum loss function & if gradient descent oscillates the algorithm decreases alpha automatically.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

✓ 0.1s Python

Figure 3: Compile the model

3. Train the model.

```
model.fit(x_train, y_train, epochs=10,
          batch_size=2000, validation_split=0.2)
```

✓ 3.7s

```
Epoch 1/10
1/1 [==============================] - 2s 2s/step - loss: 2.3339 - accuracy: 0.0000e+00 - val_loss: 1.9669 - val_accuracy: 0.5620
Epoch 2/10
1/1 [==============================] - 0s 98ms/step - loss: 1.8874 - accuracy: 0.5342 - val_loss: 1.5921 - val_accuracy: 0.8430
Epoch 3/10
1/1 [==============================] - 0s 141ms/step - loss: 1.4977 - accuracy: 0.8882 - val_loss: 1.2884 - val_accuracy: 0.8430
Epoch 4/10
1/1 [==============================] - 0s 140ms/step - loss: 1.1797 - accuracy: 0.8882 - val_loss: 1.0619 - val_accuracy: 0.8430
Epoch 5/10
1/1 [==============================] - 0s 102ms/step - loss: 0.9394 - accuracy: 0.8882 - val_loss: 0.9067 - val_accuracy: 0.8430
Epoch 6/10
1/1 [==============================] - 0s 114ms/step - loss: 0.7717 - accuracy: 0.8882 - val_loss: 0.8080 - val_accuracy: 0.8430
Epoch 7/10
1/1 [==============================] - 0s 132ms/step - loss: 0.6621 - accuracy: 0.8882 - val_loss: 0.7481 - val_accuracy: 0.8430
Epoch 8/10
1/1 [==============================] - 0s 135ms/step - loss: 0.5930 - accuracy: 0.8882 - val_loss: 0.7118 - val_accuracy: 0.8430
Epoch 9/10
1/1 [==============================] - 0s 129ms/step - loss: 0.5497 - accuracy: 0.8882 - val_loss: 0.6887 - val_accuracy: 0.8430
Epoch 10/10
1/1 [==============================] - 0s 140ms/step - loss: 0.5216 - accuracy: 0.8882 - val_loss: 0.6721 - val_accuracy: 0.8430
```

Figure 4: Trying 10 iterations

4. Test on test data.

```
results = model.evaluate(x_test,  y_test, verbose = 0)
print('test loss, test acc:', results)
```

✓ 0.3s

```
test loss, test acc: [0.4941493570804596, 0.892307698726654]
```

Figure 5: Results on test data

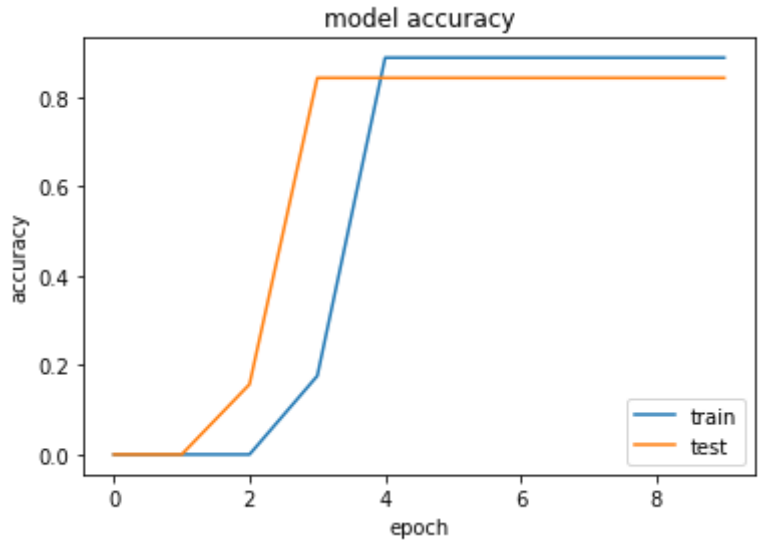
After evaluating on test data, the accuracy of the Model is 89.2% and the loss is approximately 0.5.

5. Here's a plot for the model's accuracy in each epoch, which shows that the accuracy increases until it reaches a settle point.

```
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history[ 'accuracy' ])
plt.plot(history.history[ 'val_accuracy' ])
plt.title( 'model accuracy' )
plt.ylabel( 'accuracy' )
plt.xlabel( 'epoch' )
plt.legend([ 'train' , 'test' ], loc= 'lower right' )
plt.show()
```

✓ 0.2s

.

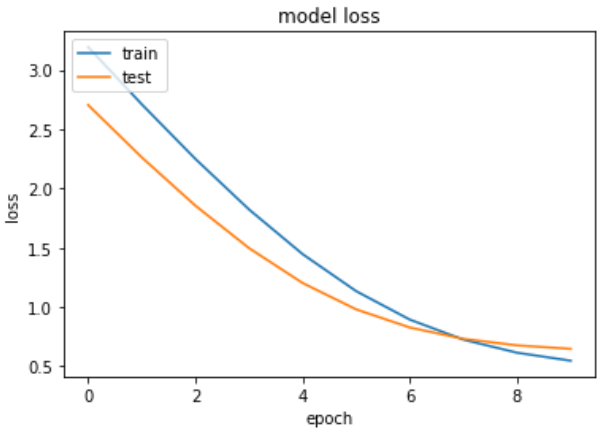


Also we tried to show the loss graph and that it decreases in each epoch

```
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history[ 'loss' ])
plt.plot(history.history[ 'val_loss' ])
plt.title( 'model loss' )
plt.ylabel( 'loss' )
plt.xlabel( 'epoch' )
plt.legend([ 'train' , 'test' ], loc= 'upper left' )
plt.show()
```

✓ 0.1s

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```



Tuning hyper-parameters:

A. Changing the number of neurons and activation function.

1. Adding more neurons in the first layer to be 300 neurons, and the second to be 150 neurons.
   Also changed the activation function of the first layer is changed to be ‘relu’.

Adding more units

```
model1 = Sequential([
    # dense layer 1
    Dense(300, activation='relu'),

    # dense layer 2
    Dense(150, activation='sigmoid'),

    # output layer
    Dense(10, activation='sigmoid'),
])
```

Figure 6: tuning the number of neurons and one of the activation functions.

2. Compile and test after tuning the parameters.

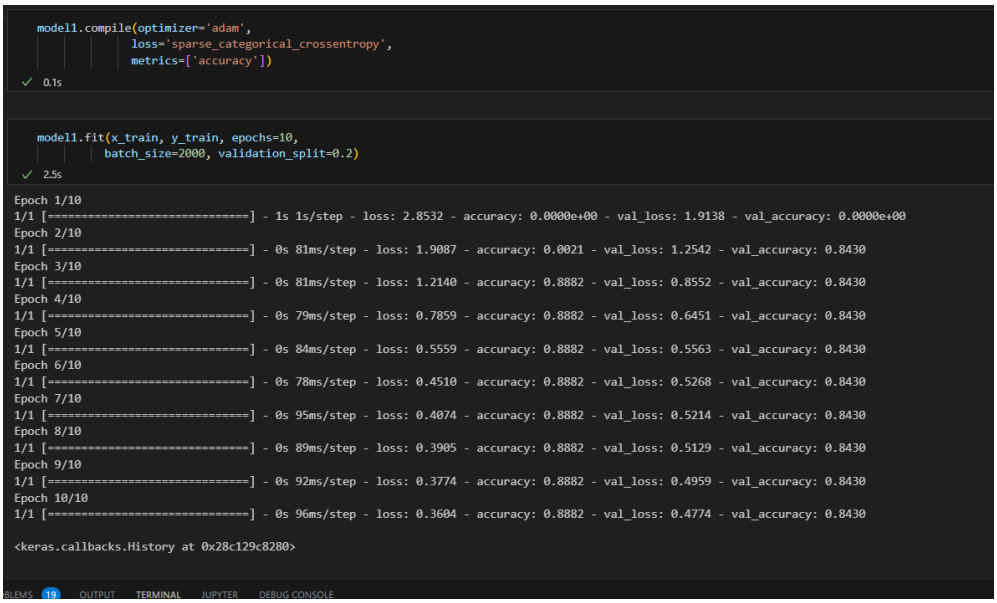


Figure 7: compile and fit

3. Calculate the accuracy and loss after tuning some parameters.

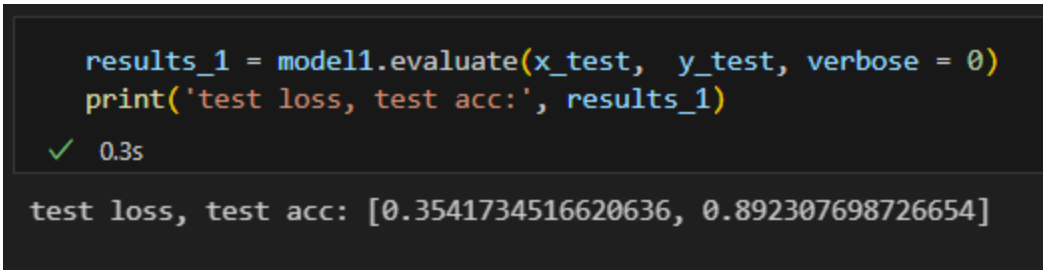


Figure 8: Results

4. After evaluating on test data, the accuracy of the Model is 89.2% which is relatively the same as without tuning the parameters but the loss decreased became 0.35.

B. Adding an additional layer.

1. Build the model with 4 layers in which the layers consist of 300 neurons, 150 neurons, 90 neurons, and 10 neurons respectively. Their activation function is ‘sigmoid’.

```
model2 = Sequential([
    # dense layer 1
    Dense(300, activation='sigmoid'),

    # dense layer 2
    Dense(150, activation='sigmoid'),

    # dense layer 3
    Dense(90, activation='sigmoid'),

     # output layer
    Dense(10, activation='sigmoid')
])
✓ 0.1s
```

Figure 9: Adding an additional layer

2. Compile and train the model

```
model2.compile(optimizer='Adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
✓ 0.1s

model2.fit(x_train, y_train, epochs=10,
           batch_size=2000, validation_split=0.2)
✓ 2.7s

Epoch 1/10
1/1 [==============================] - 2s 2s/step - loss: 2.7078 - accuracy: 0.0000e+00 - val_loss: 2.3866 - val_accuracy: 0.0000e+00
Epoch 2/10
1/1 [==============================] - 0s 81ms/step - loss: 2.3849 - accuracy: 0.0000e+00 - val_loss: 2.0923 - val_accuracy: 0.0000e+00
Epoch 3/10
1/1 [==============================] - 0s 79ms/step - loss: 2.0766 - accuracy: 0.0000e+00 - val_loss: 1.8179 - val_accuracy: 0.0000e+00
Epoch 4/10
1/1 [==============================] - 0s 72ms/step - loss: 1.7887 - accuracy: 0.0000e+00 - val_loss: 1.5691 - val_accuracy: 0.8430
Epoch 5/10
1/1 [==============================] - 0s 80ms/step - loss: 1.5269 - accuracy: 0.8882 - val_loss: 1.3509 - val_accuracy: 0.8430
Epoch 6/10
1/1 [==============================] - 0s 85ms/step - loss: 1.2962 - accuracy: 0.8882 - val_loss: 1.1664 - val_accuracy: 0.8430
Epoch 7/10
1/1 [==============================] - 0s 90ms/step - loss: 1.0997 - accuracy: 0.8882 - val_loss: 1.0165 - val_accuracy: 0.8430
Epoch 8/10
1/1 [==============================] - 0s 82ms/step - loss: 0.9385 - accuracy: 0.8882 - val_loss: 0.8995 - val_accuracy: 0.8430
Epoch 9/10
1/1 [==============================] - 0s 80ms/step - loss: 0.8110 - accuracy: 0.8882 - val_loss: 0.8115 - val_accuracy: 0.8430
Epoch 10/10
1/1 [==============================] - 0s 71ms/step - loss: 0.7133 - accuracy: 0.8882 - val_loss: 0.7475 - val_accuracy: 0.8430

<keras.callbacks.History at 0x28c13b0dd90>
```

Figure 10: Compile and fit the model with the additional layer.

3. Trying to test on test data.

```
results_2 = model2.evaluate(x_test,  y_test, verbose = 0)
print('test loss, test acc:', results_2)
```

✓ 0.3s

```
test loss, test acc: [0.6326569318771362, 0.892307698726654]
```

Figure 11: Loss and test accuracy

The accuracy is relatively the same, but the loss increased became 0.6 which means that after adding an additional layer the model became worse.

C. Trying Linear activation function in the first layer.

1. Building the model.

Change in the activation function , make the first layer linear activation function

```
model3 = Sequential([
    # dense layer 1
    Dense(300, activation='linear'),

    # dense layer 2
    Dense(150, activation='sigmoid'),


     # output layer
    Dense(10, activation='sigmoid')
])
```

✓ 0.1s

Figure 12: Layer 1 activation function is linear.

2. Compile and train.

```
model3.compile(optimizer='Adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
```
✓ 0.1s

```
model3.fit(x_train, y_train, epochs=10,
           batch_size=2000, validation_split=0.2)
```
✓ 2.5s

```
Epoch 1/10
1/1 [==============================] - 1s 1s/step - loss: 3.1600 - accuracy: 0.0000e+00 - val_loss: 1.9663 - val_accuracy: 0.1157
Epoch 2/10
1/1 [==============================] - 0s 85ms/step - loss: 1.9102 - accuracy: 0.1429 - val_loss: 1.1839 - val_accuracy: 0.8430
Epoch 3/10
1/1 [==============================] - 0s 86ms/step - loss: 1.0768 - accuracy: 0.8882 - val_loss: 0.7977 - val_accuracy: 0.8430
Epoch 4/10
1/1 [==============================] - 0s 120ms/step - loss: 0.6713 - accuracy: 0.8882 - val_loss: 0.6799 - val_accuracy: 0.8430
Epoch 5/10
1/1 [==============================] - 0s 128ms/step - loss: 0.5386 - accuracy: 0.8882 - val_loss: 0.6459 - val_accuracy: 0.8430
Epoch 6/10
1/1 [==============================] - 0s 110ms/step - loss: 0.4957 - accuracy: 0.8882 - val_loss: 0.6272 - val_accuracy: 0.8430
Epoch 7/10
1/1 [==============================] - 0s 102ms/step - loss: 0.4744 - accuracy: 0.8882 - val_loss: 0.6069 - val_accuracy: 0.8430
Epoch 8/10
1/1 [==============================] - 0s 121ms/step - loss: 0.4550 - accuracy: 0.8882 - val_loss: 0.5844 - val_accuracy: 0.8430
Epoch 9/10
1/1 [==============================] - 0s 100ms/step - loss: 0.4333 - accuracy: 0.8882 - val_loss: 0.5621 - val_accuracy: 0.8430
Epoch 10/10
1/1 [==============================] - 0s 127ms/step - loss: 0.4117 - accuracy: 0.8882 - val_loss: 0.5452 - val_accuracy: 0.8430

<keras.callbacks.History at 0x28c14ccaa90>
```

Figure 13: loss decreases in each epoch

3. Results:

```
results_3 = model3.evaluate(x_test,  y_test, verbose = 0)
print('test loss, test acc:', results_3)
```
✓ 0.2s

```
test loss, test acc: [0.4147969186306, 0.892307698726654]
```

Figure 14: [loss, accuracy]

When the activation function of the first layer became linear, the loss decreased and became 0.415 but the accuracy is the same.

D. Trying to tune the optimizer to be “Adagrad”

1. Building the same model as before.

```
model4 = Sequential([
    # dense layer 1
    Dense(300, activation='linear'),

    # dense layer 2
    Dense(150, activation='sigmoid'),


     # output layer
    Dense(10, activation='sigmoid')
])
```

2. Change the optimizer and fit the data.

```
model4.compile(optimizer='Adagrad',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
```

```
model4.fit(x_train, y_train, epochs=10,
           batch_size=2000, validation_split=0.2)
```

```
Epoch 1/10
1/1 [==============================] - 1s 1s/step - loss: 2.8727 - accuracy: 0.0000e+00 - val_loss: 2.3994 - val_accuracy: 0.0000e+00
Epoch 2/10
1/1 [==============================] - 0s 76ms/step - loss: 2.3626 - accuracy: 0.0062 - val_loss: 2.0309 - val_accuracy: 0.0826
Epoch 3/10
1/1 [==============================] - 0s 76ms/step - loss: 1.9839 - accuracy: 0.0911 - val_loss: 1.7656 - val_accuracy: 0.3802
Epoch 4/10
1/1 [==============================] - 0s 80ms/step - loss: 1.7147 - accuracy: 0.4617 - val_loss: 1.5785 - val_accuracy: 0.7438
Epoch 5/10
1/1 [==============================] - 0s 81ms/step - loss: 1.5250 - accuracy: 0.7847 - val_loss: 1.4474 - val_accuracy: 0.8017
Epoch 6/10
1/1 [==============================] - 0s 89ms/step - loss: 1.3897 - accuracy: 0.8489 - val_loss: 1.3472 - val_accuracy: 0.8264
Epoch 7/10
1/1 [==============================] - 0s 83ms/step - loss: 1.2846 - accuracy: 0.8696 - val_loss: 1.2630 - val_accuracy: 0.8430
Epoch 8/10
1/1 [==============================] - 0s 78ms/step - loss: 1.1958 - accuracy: 0.8778 - val_loss: 1.1882 - val_accuracy: 0.8430
Epoch 9/10
1/1 [==============================] - 0s 76ms/step - loss: 1.1175 - accuracy: 0.8861 - val_loss: 1.1233 - val_accuracy: 0.8430
Epoch 10/10
1/1 [==============================] - 0s 84ms/step - loss: 1.0500 - accuracy: 0.8882 - val_loss: 1.0671 - val_accuracy: 0.8430
```

3. Results of evaluation:

```
results_4 = model4.evaluate(x_test,  y_test, verbose = 0)
print('test loss, test acc:', results_4)
```

✓ 0.0s

```
test loss, test acc: [0.9793413281440735, 0.892307698726654]
```

The loss increased this time to be 0.98 which is not good, and this means that this optimizer doesn’t cope with our data.

Link of github repository

https://github.com/Aya-gamal2211/MachineLearning