CISC 867: Deep Learning

Project #1

Supervisor:
Dr. Hazem Abbas

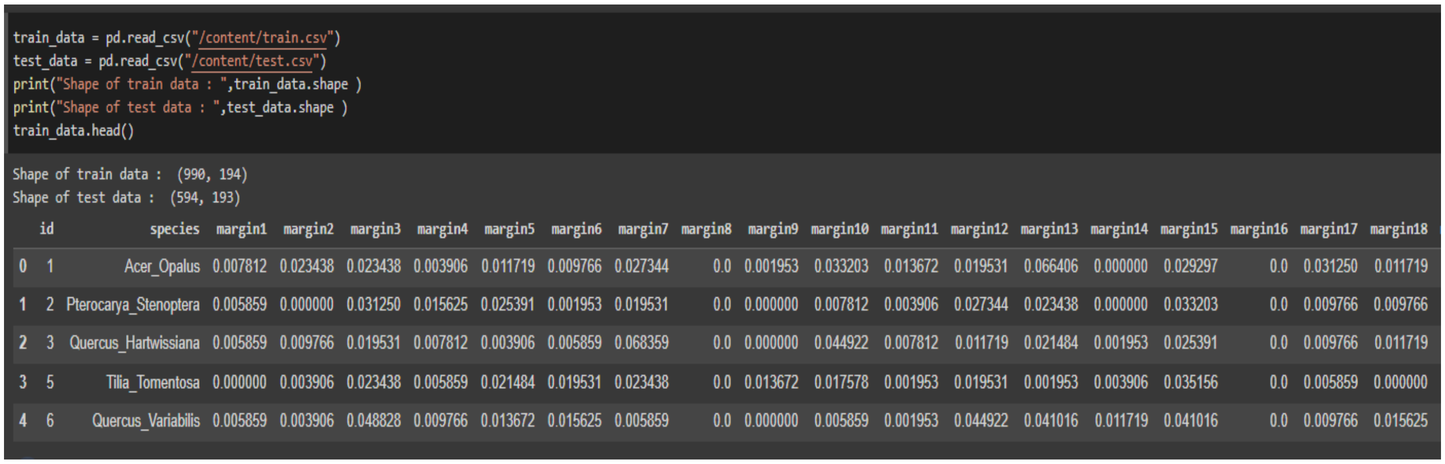
Done By:
Name : Esraa Ahmed Fouad Omar
ID : 20399123

Contents

List of Figures

1 Data

After describing our data, we found that our data contains 3 sets of features are also provided per image: a shape contiguous descriptor, an interior texture histogram, and a fine-scale margin histogram. For each feature, a 64_attribute vector is given per leaf sample.In additions, there is an ID column as features. These features are used to accurately identify 99 species of plant.



```
train_data = pd.read_csv("/content/train.csv")
test_data = pd.read_csv("/content/test.csv")
print("Shape of train data : ",train_data.shape )
print("Shape of test data : ",test_data.shape )
train_data.head()
```

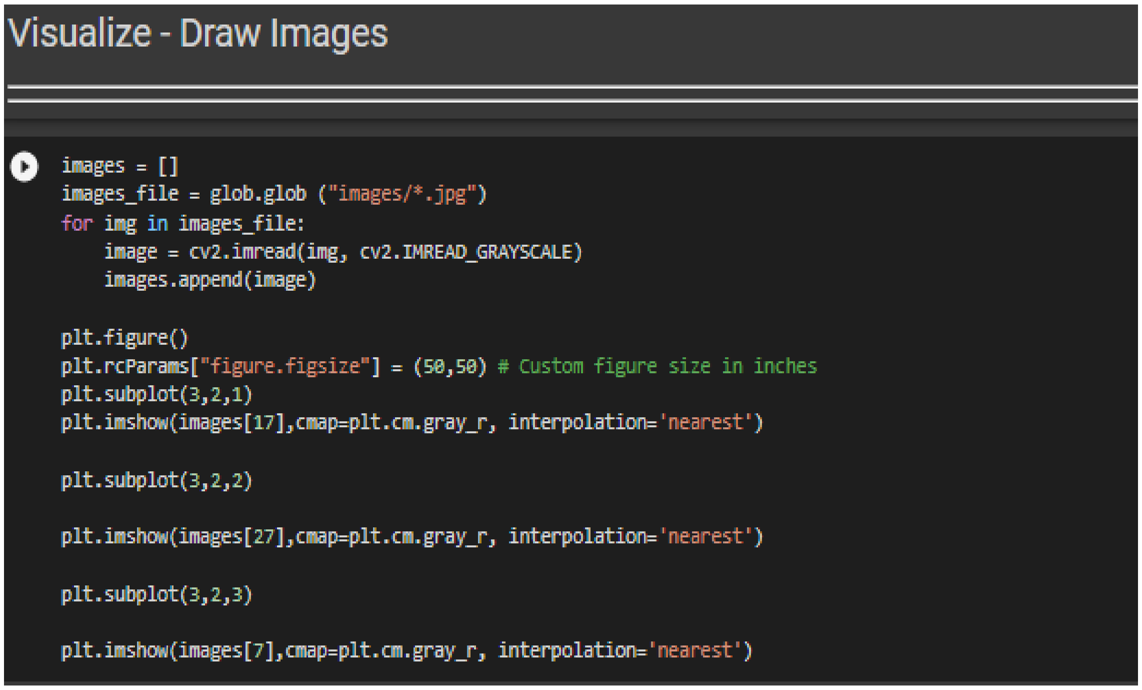
```
Shape of train data :  (990, 194)
Shape of test data :  (594, 193)
```

	id	species	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margin13	margin14	margin15	margin16	margin17	margin18
0	1	Acer_Opalus	0.007812	0.023438	0.023438	0.003906	0.011719	0.009766	0.027344	0.0	0.001953	0.033203	0.013672	0.019531	0.066406	0.000000	0.029297	0.0	0.031250	0.011719
1	2	Pterocarya_Stenoptera	0.005859	0.000000	0.031250	0.015625	0.025391	0.001953	0.019531	0.0	0.000000	0.007812	0.003906	0.027344	0.023438	0.000000	0.033203	0.0	0.009766	0.009766
2	3	Quercus_Hartwissiana	0.005859	0.009766	0.019531	0.007812	0.003906	0.005859	0.068359	0.0	0.000000	0.044922	0.007812	0.011719	0.021484	0.001953	0.025391	0.0	0.009766	0.011719
3	5	Tilia_Tomentosa	0.000000	0.003906	0.023438	0.005859	0.021484	0.019531	0.023438	0.0	0.013672	0.017578	0.001953	0.019531	0.001953	0.003906	0.035156	0.0	0.005859	0.000000
4	6	Quercus_Variabilis	0.005859	0.003906	0.048828	0.009766	0.013672	0.015625	0.005859	0.0	0.000000	0.005859	0.001953	0.044922	0.041016	0.011719	0.041016	0.0	0.009766	0.015625

Figure 1: Sample of data

2 Draw some of the images

Here, we draw some of the images that exists in the images folder.we chose randomly 3 images



Visualize - Draw Images

```
images = []
images_file = glob.glob ("images/*.jpg")
for img in images_file:
    image = cv2.imread(img, cv2.IMREAD_GRAYSCALE)
    images.append(image)

plt.figure()
plt.rcParams["figure.figsize"] = (50,50) # Custom figure size in inches
plt.subplot(3,2,1)
plt.imshow(images[17],cmap=plt.cm.gray_r, interpolation='nearest')

plt.subplot(3,2,2)

plt.imshow(images[27],cmap=plt.cm.gray_r, interpolation='nearest')

plt.subplot(3,2,3)

plt.imshow(images[7],cmap=plt.cm.gray_r, interpolation='nearest')
```

Figure 2: Code to draw images

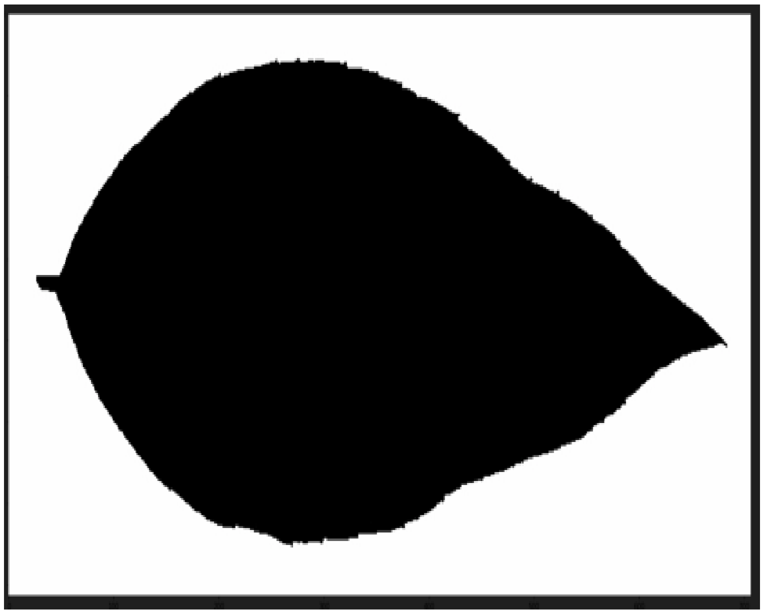


Figure 3: image of index 17

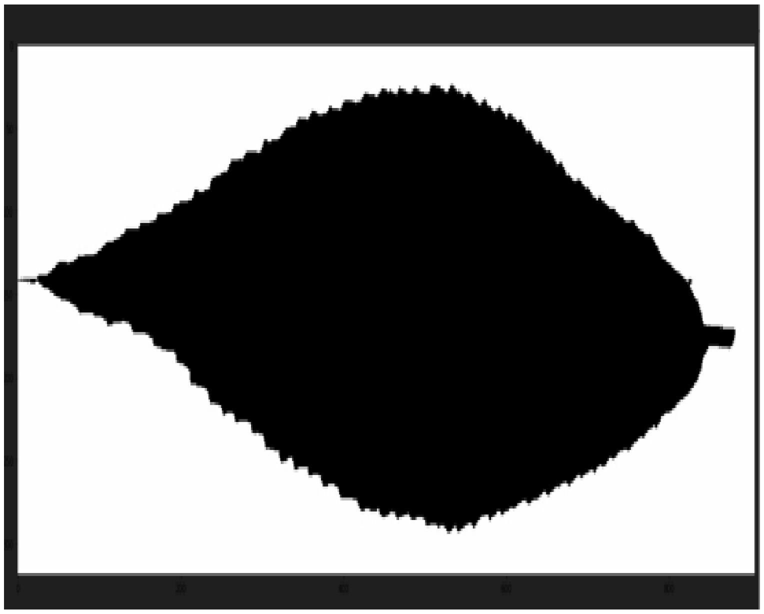


Figure 4: image of index 27

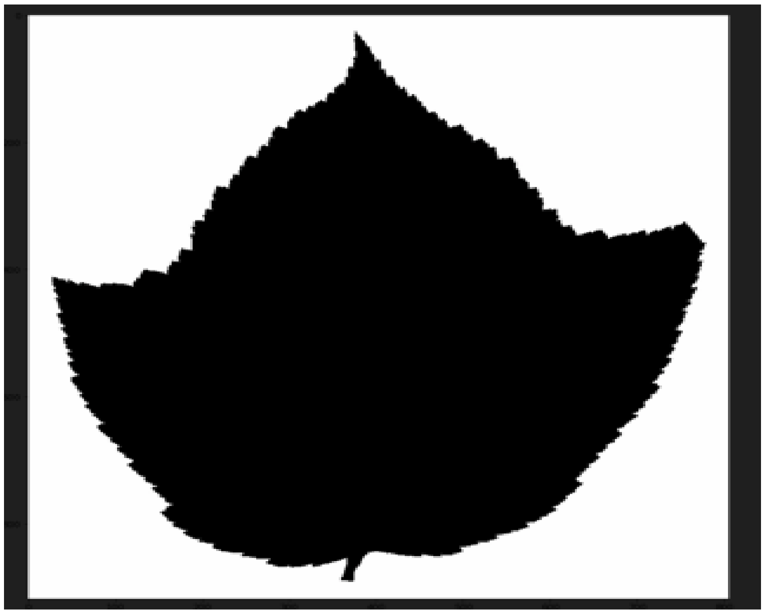
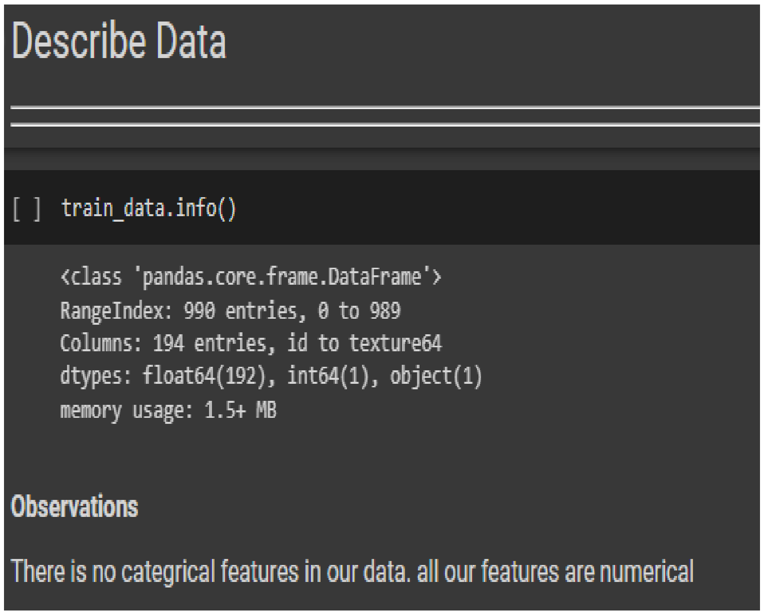


Figure 5: image of index 7

3 Explore Data

1. Check Type of our data : As you can see in figure[6], all of our features are numerical except the target column(Species) is categorical



Describe Data

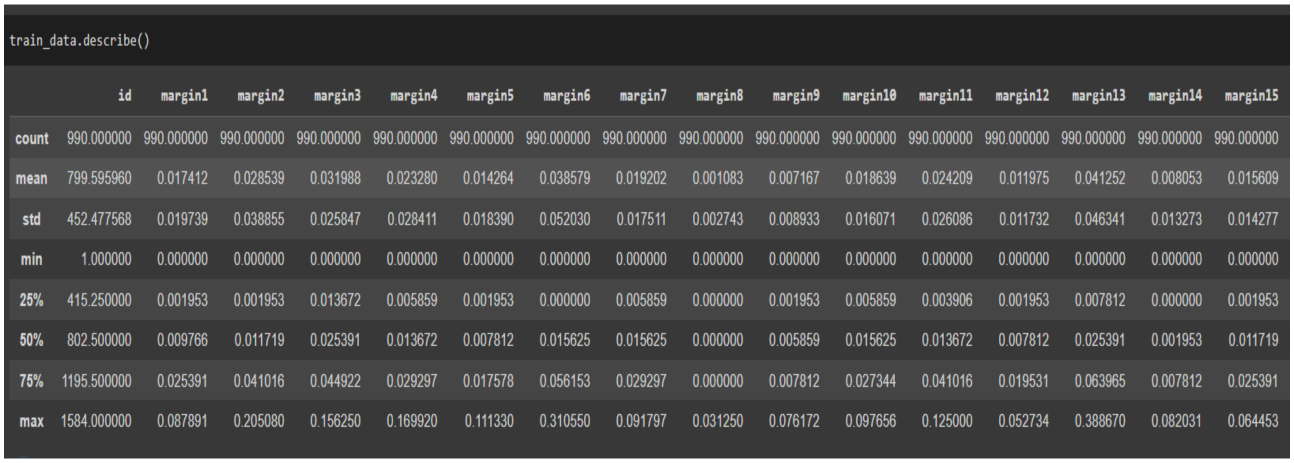
```
[ ] train_data.info()

    <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 990 entries, 0 to 989
    Columns: 194 entries, id to texture64
    dtypes: float64(192), int64(1), object(1)
    memory usage: 1.5+ MB
```

Observations

There is no categrical features in our data. all our features are numerical

Figure 6: Type of our data

2. Check the Distribution of our data : As you can see in figure[7],a statistical summary of the input variables is provided showing that values are numeric and range approximately from 0 to 1.Also, some of features are skewed distribution because mean and median are not equal



```
train_data.describe()
```

	id	margin1	margin2	margin3	margin4	margin5	margin6	margin7	margin8	margin9	margin10	margin11	margin12	margin13	margin14	margin15
count	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000	990.000000
mean	799.595960	0.017412	0.028539	0.031988	0.023280	0.014264	0.038579	0.019202	0.001083	0.007167	0.018639	0.024209	0.011975	0.041252	0.008053	0.015609
std	452.477568	0.019739	0.038855	0.025847	0.028411	0.018390	0.052030	0.017511	0.002743	0.008933	0.016071	0.026086	0.011732	0.046341	0.013273	0.014277
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	415.250000	0.001953	0.001953	0.013672	0.005859	0.001953	0.000000	0.005859	0.000000	0.001953	0.005859	0.003906	0.001953	0.007812	0.000000	0.001953
50%	802.500000	0.009766	0.011719	0.025391	0.013672	0.007812	0.015625	0.015625	0.000000	0.005859	0.015625	0.013672	0.007812	0.025391	0.001953	0.011719
75%	1195.500000	0.025391	0.041016	0.044922	0.029297	0.017578	0.056153	0.029297	0.000000	0.007812	0.027344	0.041016	0.019531	0.063965	0.007812	0.025391
max	1584.000000	0.087891	0.205080	0.156250	0.169920	0.111330	0.310550	0.091797	0.031250	0.076172	0.097656	0.125000	0.052734	0.388670	0.082031	0.064453

Figure 7: Distribution of our data using Describe function

Note that :

The figure [7] doesn't contain all the information about all columns. I couldn't take a snap shoot for all columns. The full image is in the notebook, you can see it

Let's visualize our data to see the distribution in more details.we can see from 8 that many features have a skewed distribution.

The dataset provides a good candidate for using scaler transforms. So, we need to apply a scaler transforms in our data . Later on, we will standardize the data using standerscaler

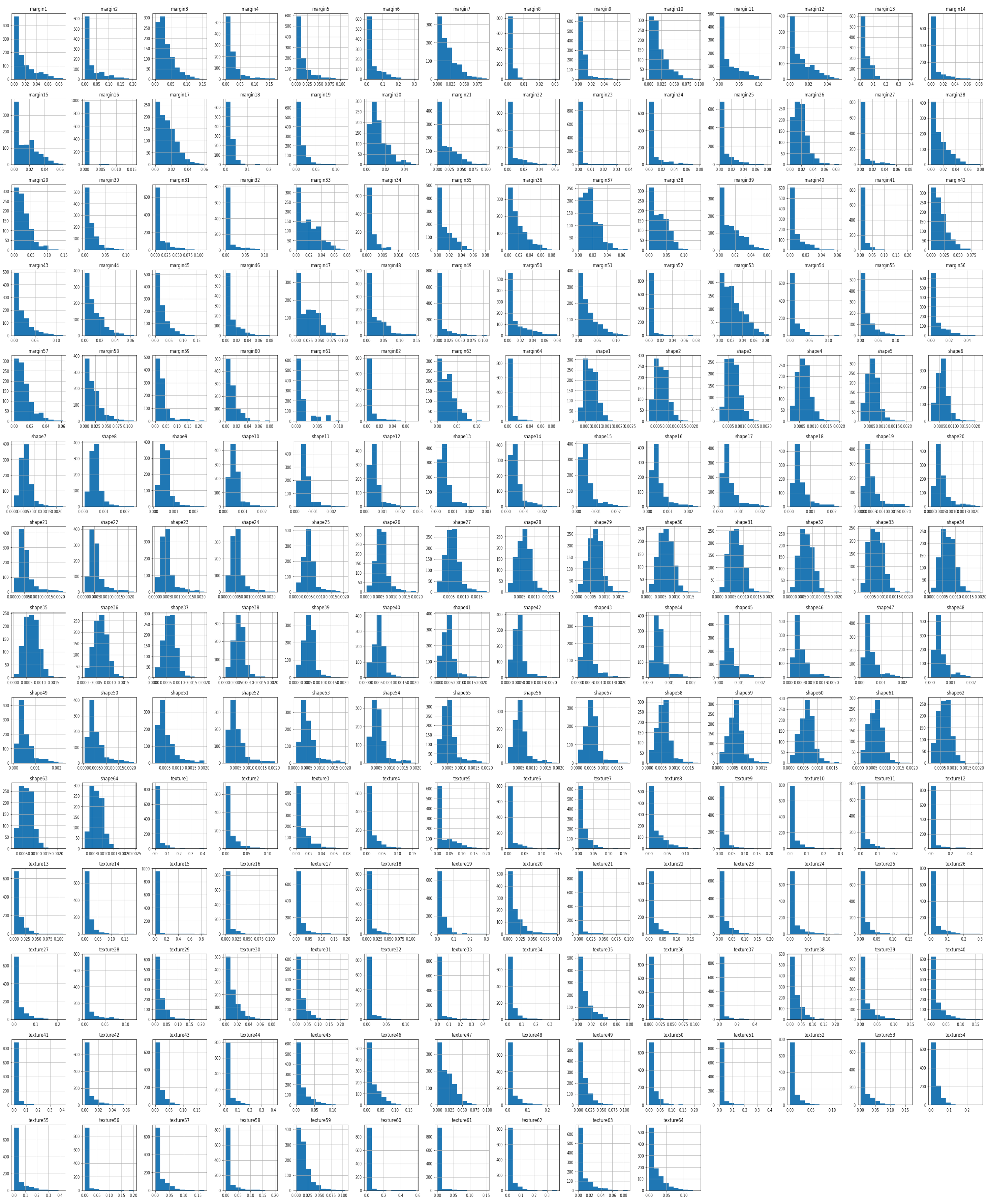
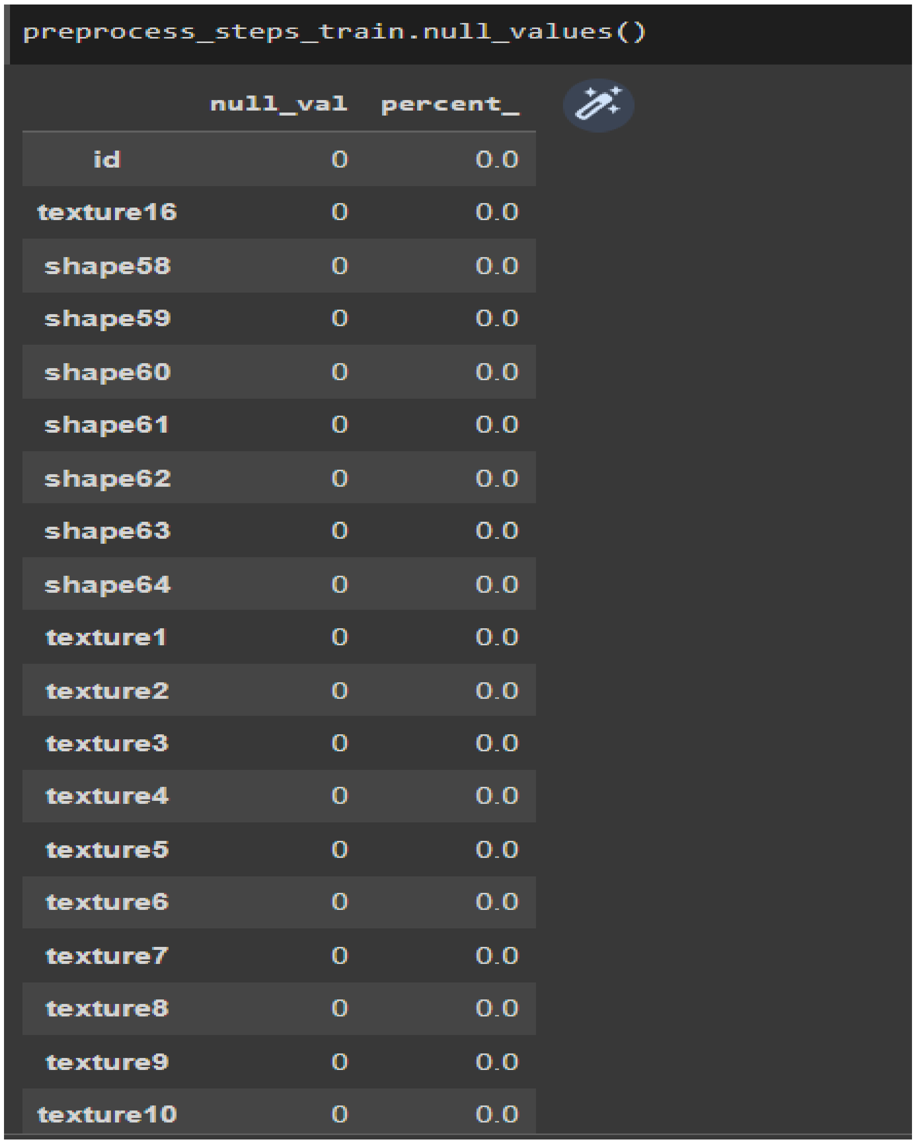


Figure 8: Distribution of our data using Visualization

3. Check Null values: As you can see in figure 9, there is no null values in our data

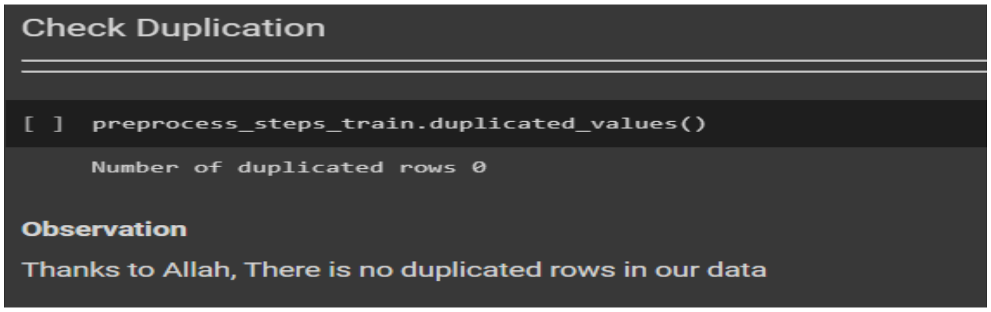


```
preprocess_steps_train.null_values()
```

	null_val	percent_
id	0	0.0
texture16	0	0.0
shape58	0	0.0
shape59	0	0.0
shape60	0	0.0
shape61	0	0.0
shape62	0	0.0
shape63	0	0.0
shape64	0	0.0
texture1	0	0.0
texture2	0	0.0
texture3	0	0.0
texture4	0	0.0
texture5	0	0.0
texture6	0	0.0
texture7	0	0.0
texture8	0	0.0
texture9	0	0.0
texture10	0	0.0

Figure 9: sample of

4. Check Duplication: As you can see in figure 10, there is no duplicated rows in our data



Check Duplication

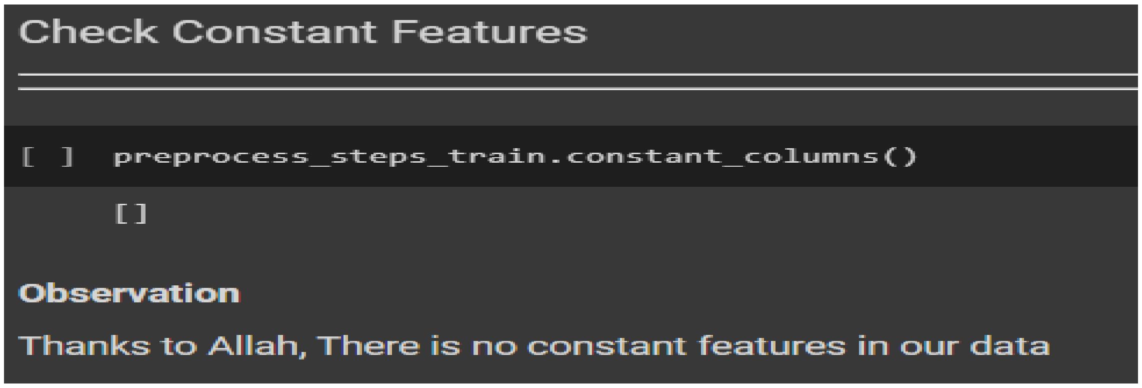
```
[ ] preprocess_steps_train.duplicated_values()

    Number of duplicated rows 0
```

Observation

Thanks to Allah, There is no duplicated rows in our data

Figure 10: Number of duplicated rows in our data

5. Check Constant features : Here, we check if there are columns contain only one value a cross all samples. As you can see in figure 11, there is no constant features in our data



Check Constant Features

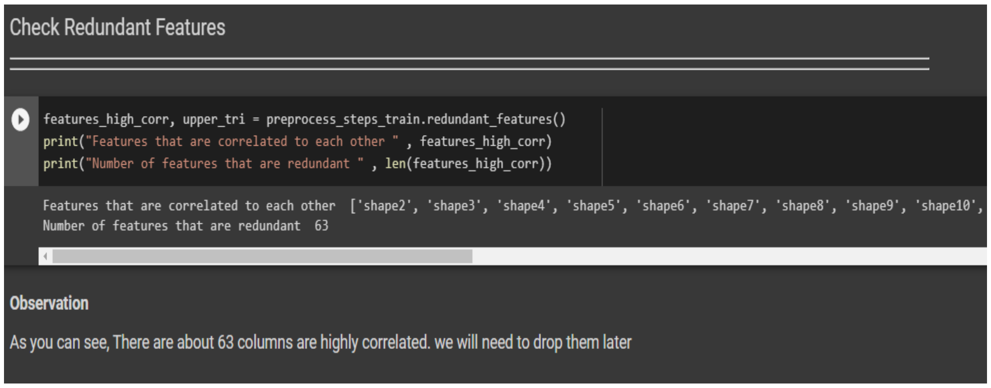
```
[ ]  preprocess_steps_train.constant_columns()

     []
```

Observation

Thanks to Allah, There is no constant features in our data

Figure 11: Number of constant features in our data

6. Check redundant features : Here, we check if there are columns are highly correlated (corr more than 0.95). As you can see in figure 12, there are about 63 columns are correlated. We will drop them in next section



Check Redundant Features

```
features_high_corr, upper_tri = preprocess_steps_train.redundant_features()
print("Features that are correlated to each other " , features_high_corr)
print("Number of features that are redundant " , len(features_high_corr))
```

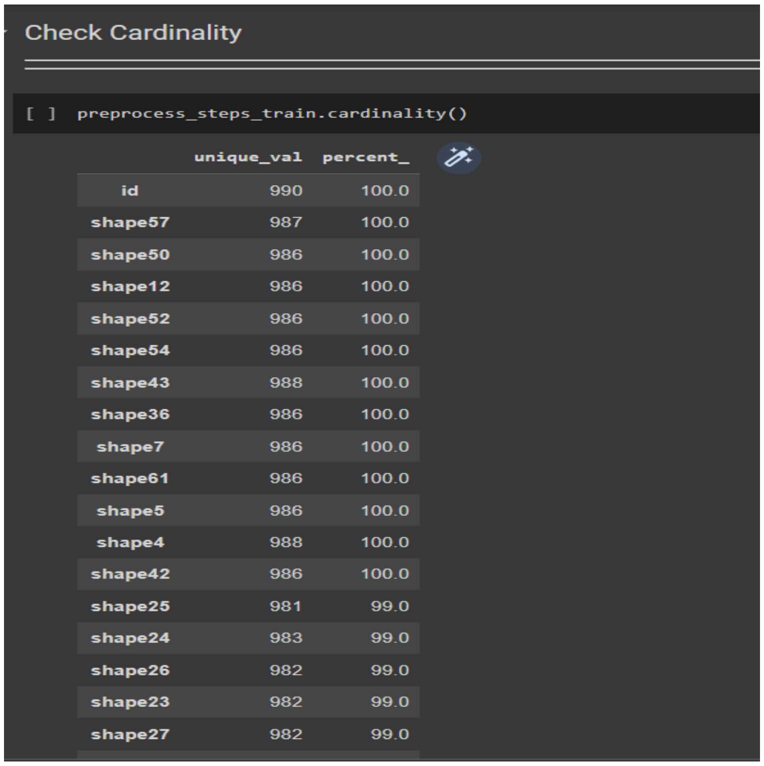
```
Features that are correlated to each other  ['shape2', 'shape3', 'shape4', 'shape5', 'shape6', 'shape7', 'shape8', 'shape9', 'shape10',
Number of features that are redundant  63
```

Observation

As you can see, There are about 63 columns are highly correlated. we will need to drop them later

Figure 12: Number of redundant features in our data

7. Check cardinality: As you can see, There are about 66 columns are highly cardinality in Figure 13 . we will need to drop them (equal or more than 99%) in the next section



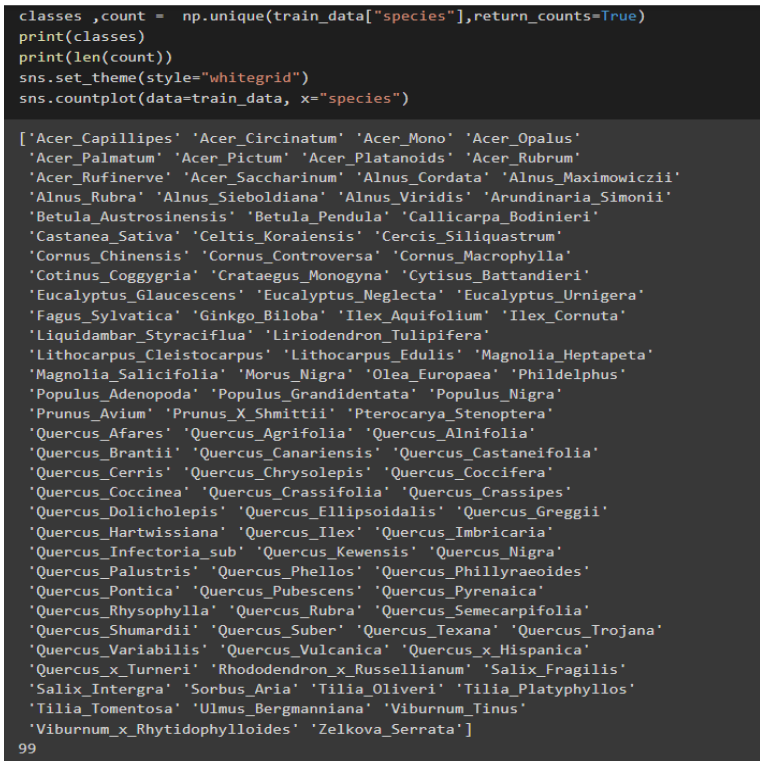
Check Cardinality

```
[ ]  preprocess_steps_train.cardinality()
```

	unique_val	percent_
id	990	100.0
shape57	987	100.0
shape50	986	100.0
shape12	986	100.0
shape52	986	100.0
shape54	986	100.0
shape43	988	100.0
shape36	986	100.0
shape7	986	100.0
shape61	986	100.0
shape5	986	100.0
shape4	988	100.0
shape42	986	100.0
shape25	981	99.0
shape24	983	99.0
shape26	982	99.0
shape23	982	99.0
shape27	982	99.0

Figure 13: sample of columns and its unique values

8. Check Imbalance in classes:



```
classes ,count =  np.unique(train_data["species"],return_counts=True)
print(classes)
print(len(count))
sns.set_theme(style="whitegrid")
sns.countplot(data=train_data, x="species")
```

```
['Acer_Capillipes' 'Acer_Circinatum' 'Acer_Mono' 'Acer_Opalus'
 'Acer_Palmatum' 'Acer_Pictum' 'Acer_Platanoids' 'Acer_Rubrum'
 'Acer_Rufinerve' 'Acer_Saccharinum' 'Alnus_Cordata' 'Alnus_Maximowiczii'
 'Alnus_Rubra' 'Alnus_Sieboldiana' 'Alnus_Viridis' 'Arundinaria_Simonii'
 'Betula_Austrosinensis' 'Betula_Pendula' 'Callicarpa_Bodinieri'
 'Castanea_Sativa' 'Celtis_Koraiensis' 'Cercis_Siliquastrum'
 'Cornus_Chinensis' 'Cornus_Controversa' 'Cornus_Macrophylla'
 'Cotinus_Coggygria' 'Crataegus_Monogyna' 'Cytisus_Battandieri'
 'Eucalyptus_Glaucescens' 'Eucalyptus_Neglecta' 'Eucalyptus_Urnigera'
 'Fagus_Sylvatica' 'Ginkgo_Biloba' 'Ilex_Aquifolium' 'Ilex_Cornuta'
 'Liquidambar_Styraciflua' 'Liriodendron_Tulipifera'
 'Lithocarpus_Cleistocarpus' 'Lithocarpus_Edulis' 'Magnolia_Heptapeta'
 'Magnolia_Salicifolia' 'Morus_Nigra' 'Olea_Europaea' 'Phildelphus'
 'Populus_Adenopoda' 'Populus_Grandidentata' 'Populus_Nigra'
 'Prunus_Avium' 'Prunus_X_Shmittii' 'Pterocarya_Stenoptera'
 'Quercus_Afares' 'Quercus_Agrifolia' 'Quercus_Alnifolia'
 'Quercus_Brantii' 'Quercus_Canariensis' 'Quercus_Castaneifolia'
 'Quercus_Cerris' 'Quercus_Chrysolepis' 'Quercus_Coccifera'
 'Quercus_Coccinea' 'Quercus_Crassifolia' 'Quercus_Crassipes'
 'Quercus_Dolicholepis' 'Quercus_Ellipsoidalis' 'Quercus_Greggii'
 'Quercus_Hartwissiana' 'Quercus_Ilex' 'Quercus_Imbricaria'
 'Quercus_Infectoria_sub' 'Quercus_Kewensis' 'Quercus_Nigra'
 'Quercus_Palustris' 'Quercus_Phellos' 'Quercus_Phillyraeoides'
 'Quercus_Pontica' 'Quercus_Pubescens' 'Quercus_Pyrenaica'
 'Quercus_Rhysophylla' 'Quercus_Rubra' 'Quercus_Semecarpifolia'
 'Quercus_Shumardii' 'Quercus_Suber' 'Quercus_Texana' 'Quercus_Trojana'
 'Quercus_Variabilis' 'Quercus_Vulcanica' 'Quercus_x_Hispanica'
 'Quercus_x_Turneri' 'Rhododendron_x_Russellianum' 'Salix_Fragilis'
 'Salix_Intergra' 'Sorbus_Aria' 'Tilia_Oliveri' 'Tilia_Platyphyllos'
 'Tilia_Tomentosa' 'Ulmus_Bergmanniana' 'Viburnum_Tinus'
 'Viburnum_x_Rhytidophylloides' 'Zelkova_Serrata']
99
```

Figure 14: sample of columns and its unique values

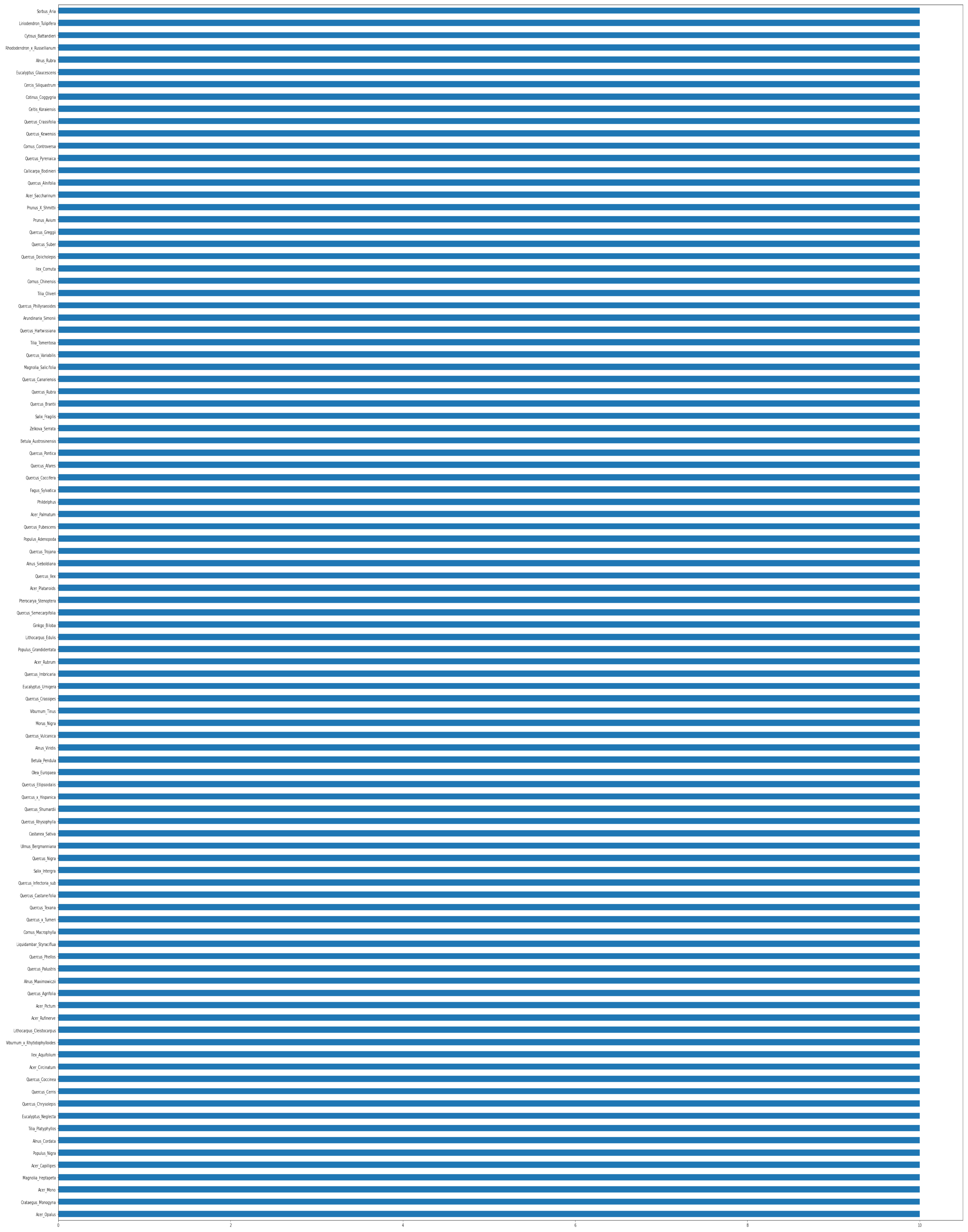
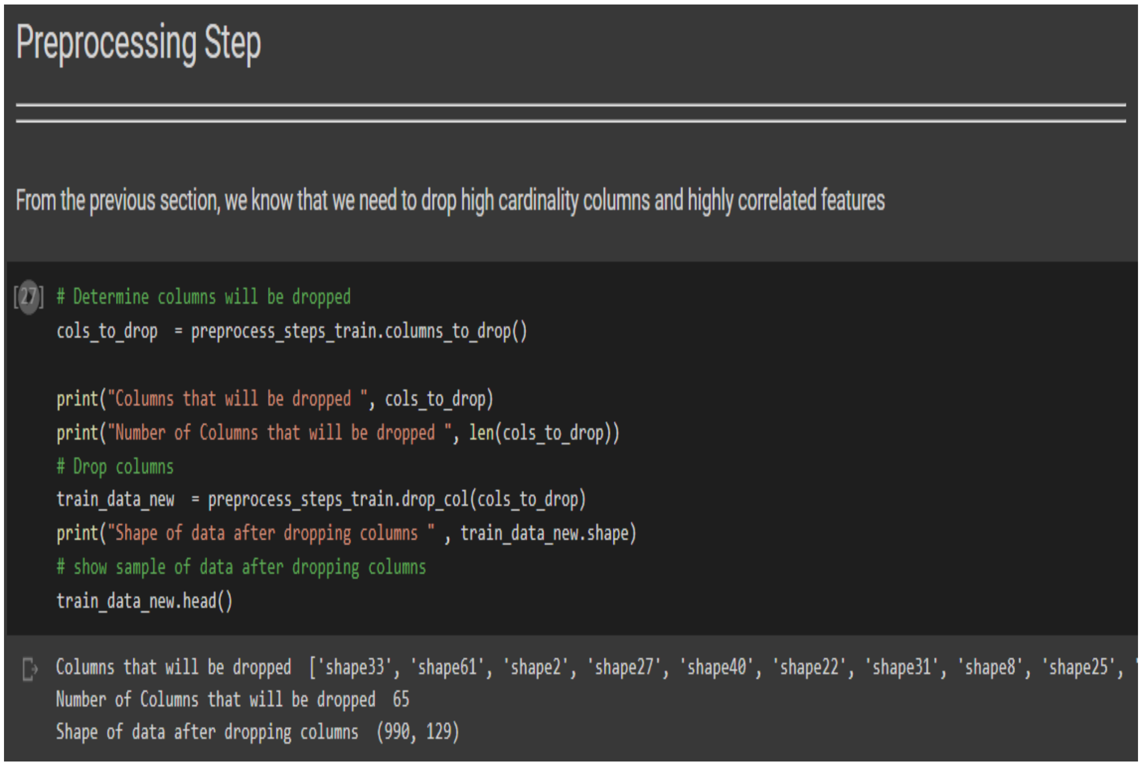


Figure 15:   sample of columns and its unique values

4 Preprocessing Data

From the previous section, we know that we need to drop high cardinality columns and highly correlated features. As you can see, There are about 65 columns dropped



Preprocessing Step

From the previous section, we know that we need to drop high cardinality columns and highly correlated features

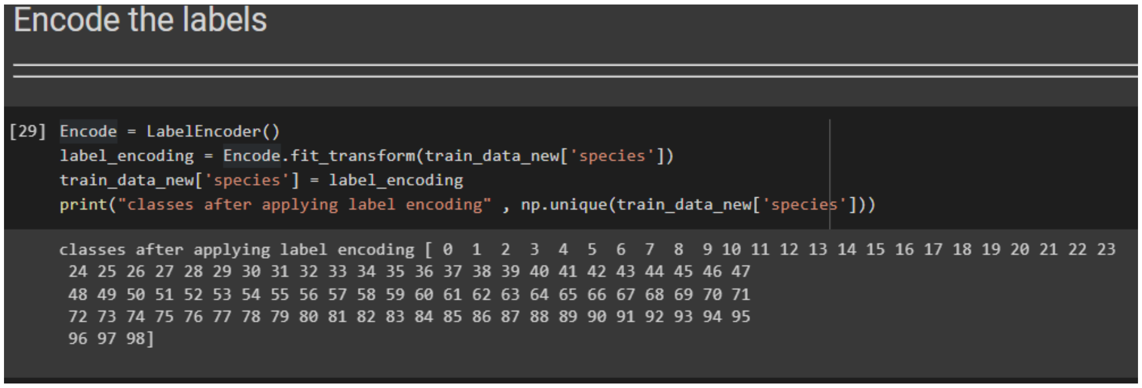
```
[27] # Determine columns will be dropped
     cols_to_drop  = preprocess_steps_train.columns_to_drop()

     print("Columns that will be dropped ", cols_to_drop)
     print("Number of Columns that will be dropped ", len(cols_to_drop))
     # Drop columns
     train_data_new  = preprocess_steps_train.drop_col(cols_to_drop)
     print("Shape of data after dropping columns " , train_data_new.shape)
     # show sample of data after dropping columns
     train_data_new.head()
```

```
Columns that will be dropped  ['shape33', 'shape61', 'shape2', 'shape27', 'shape40', 'shape22', 'shape31', 'shape8', 'shape25', '
Number of Columns that will be dropped  65
Shape of data after dropping columns  (990, 129)
```

Figure 16: shape of data after dropping

5 Encode Labels

As we Knew from the section of Explore Data, the classes is string .So, we need to convert them into a numeric form so as to convert them into the machine-readable form.As shown in figure 14, we successfully converted it.



Encode the labels

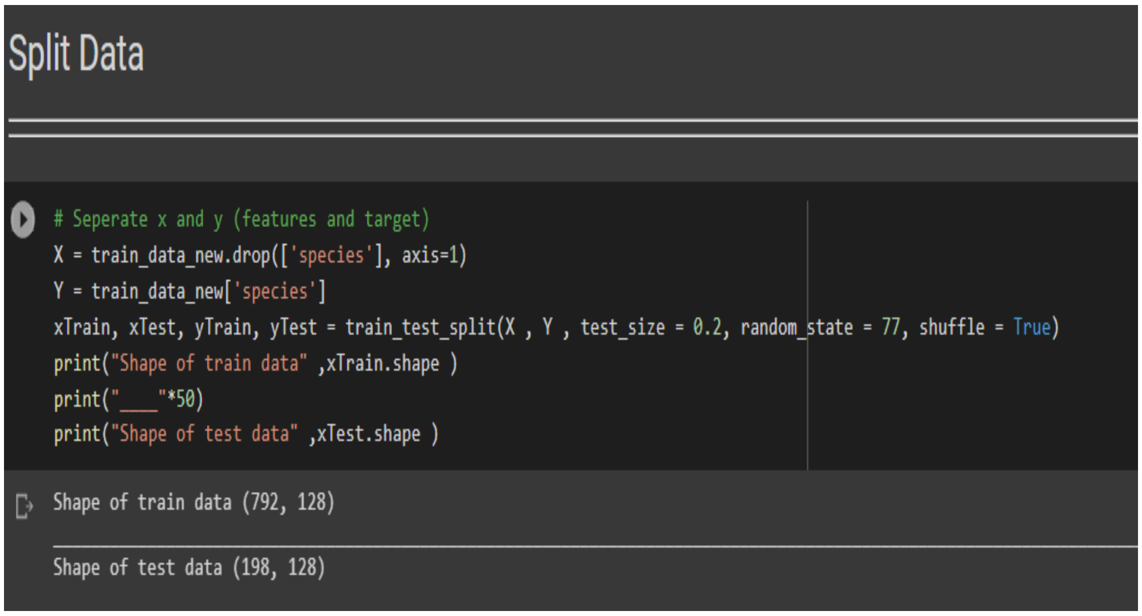
```
[29] Encode = LabelEncoder()
     label_encoding = Encode.fit_transform(train_data_new['species'])
     train_data_new['species'] = label_encoding
     print("classes after applying label encoding" , np.unique(train_data_new['species']))
```

```
classes after applying label encoding [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98]
```

Figure 17: Label encode

6 Split Data

We split data into train and test. The train set represents 80% of actual data and the test set represents 20 % of actual data as shown in figure 15. Then,we split the train into 2 partitions: train set and validation test internally in GridSearchCV to tune optimal hyper parameters.



Split Data

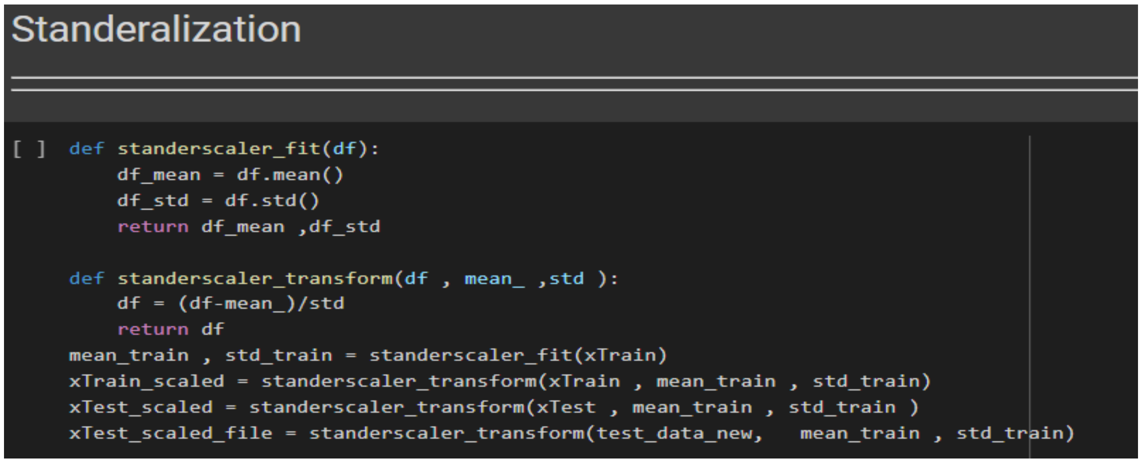
```
# Seperate x and y (features and target)
X = train_data_new.drop(['species'], axis=1)
Y = train_data_new['species']
xTrain, xTest, yTrain, yTest = train_test_split(X , Y , test_size = 0.2, random_state = 77, shuffle = True)
print("Shape of train data" ,xTrain.shape )
print("___"*50)
print("Shape of test data" ,xTest.shape )
```

```
Shape of train data (792, 128)
______________________________________________________________________________________________________________________________________________________
Shape of test data (198, 128)
```

Figure 18: shape of test and train after splitting

7 Standardization

As we knew from Explore Data Section, that most of features are skewed distribution.Here, we apply Standardization method to reduce the effect of skewed distribution on our model. we computed the mean and standard deviation for each feature dimension using the training set only, then subtracting the mean and dividing by the stdev for each feature and each sample for both train and test data.



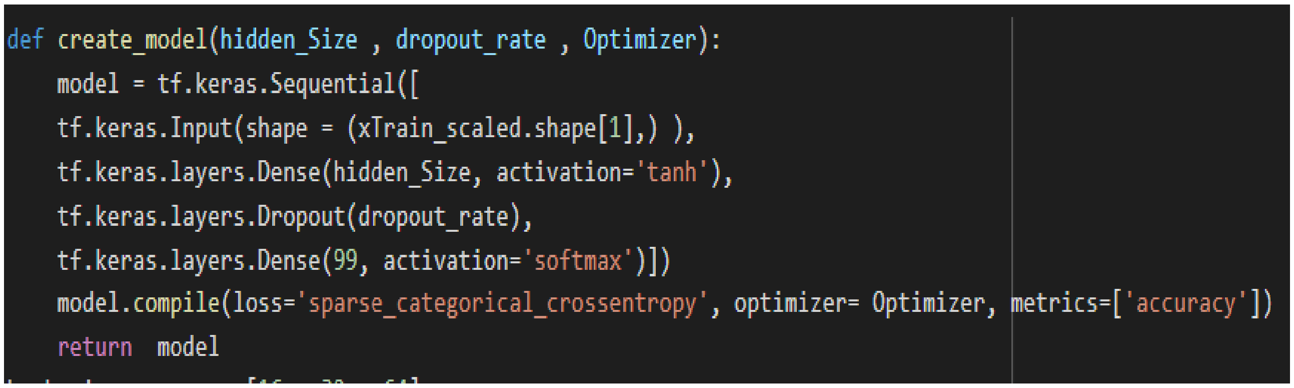
Standeralization

```
[ ]  def standerscaler_fit(df):
         df_mean = df.mean()
         df_std = df.std()
         return df_mean ,df_std

     def standerscaler_transform(df , mean_ ,std ):
         df = (df-mean_)/std
         return df
     mean_train , std_train = standerscaler_fit(xTrain)
     xTrain_scaled = standerscaler_transform(xTrain , mean_train , std_train)
     xTest_scaled = standerscaler_transform(xTest , mean_train , std_train )
     xTest_scaled_file = standerscaler_transform(test_data_new,   mean_train , std_train)
```

Figure 19: scale train and test data

8 Build Model

In this Section, we built our model (one input layer, one hidden layer with tanh activation and one output layer) as shown in figure 20.

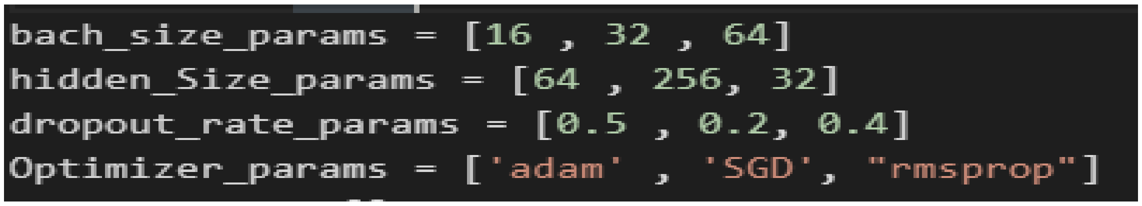


```
def create_model(hidden_Size , dropout_rate , Optimizer):
    model = tf.keras.Sequential([
    tf.keras.Input(shape = (xTrain_scaled.shape[1],) ),
    tf.keras.layers.Dense(hidden_Size, activation='tanh'),
    tf.keras.layers.Dropout(dropout_rate),
    tf.keras.layers.Dense(99, activation='softmax')])
    model.compile(loss='sparse_categorical_crossentropy', optimizer= Optimizer, metrics=['accuracy'])
    return  model
```

Figure 20: Build Model

9 Tune Hyper parameters

In our project, we tried 4 different hyper parameters and chose 3 different values for each hyperparameters as shown in figure 21. we used GridSearchCV to create different combination between these hyper parameters

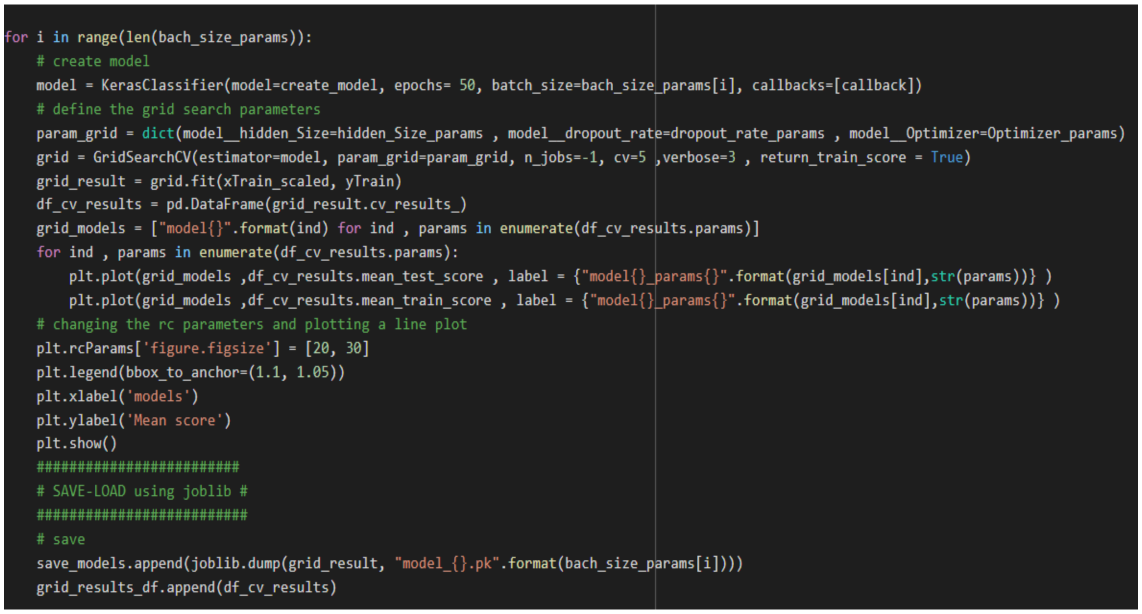


```
bach_size_params = [16 , 32 , 64]
hidden_Size_params = [64 , 256, 32]
dropout_rate_params = [0.5 , 0.2, 0.4]
Optimizer_params = ['adam' , 'SGD', "rmsprop"]
```

Figure 21: values for each hyper parameters

Our approach was that we used GridSearchCV 3 times. Each time, we passed different batch_size value from the above values. we applied GridSearchCV on the 3 other hyper parameters(Hidden size , Dropout , Optimizer)

Figure [22] shows that we looped on the batch size values and each time,GridSearchCV was Fitting 5 folds for each of 27 candidates, totalling 135 fits and plotting accuracy of train and test for each trail. Totally, we tried about 81 trails.Then, save the best 3 trails.



```
for i in range(len(bach_size_params)):
    # create model
    model = KerasClassifier(model=create_model, epochs= 50, batch_size=bach_size_params[i], callbacks=[callback])
    # define the grid search parameters
    param_grid = dict(model__hidden_Size=hidden_Size_params , model__dropout_rate=dropout_rate_params , model__Optimizer=Optimizer_params)
    grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=5 ,verbose=3 , return_train_score = True)
    grid_result = grid.fit(xTrain_scaled, yTrain)
    df_cv_results = pd.DataFrame(grid_result.cv_results_)
    grid_models = ["model{}".format(ind) for ind , params in enumerate(df_cv_results.params)]
    for ind , params in enumerate(df_cv_results.params):
        plt.plot(grid_models ,df_cv_results.mean_test_score , label = {"model{}_params{}".format(grid_models[ind],str(params))} )
        plt.plot(grid_models ,df_cv_results.mean_train_score , label = {"model{}_params{}".format(grid_models[ind],str(params))} )
    # changing the rc parameters and plotting a line plot
    plt.rcParams['figure.figsize'] = [20, 30]
    plt.legend(bbox_to_anchor=(1.1, 1.05))
    plt.xlabel('models')
    plt.ylabel('Mean score')
    plt.show()
    ########################
    # SAVE-LOAD using joblib #
    #########################
    # save
    save_models.append(joblib.dump(grid_result, "model_{}.pk".format(bach_size_params[i])))
    grid_results_df.append(df_cv_results)
```

Figure 22: Tune Hyper-parameters

1. **1st iteration (Fixing batch size to 16) :**

   Figure [23] shows that :

   - **Changing hidden size and fixing optimizer to adam and dropout to 0.5 :**
     From model 0 to 2 , we noticed that increasing hidden size reduced overfit. As shown in Figure [23] that model 1 with hidden size equals 256 which is largest one has low difference between train and test accuracy. Also, model2 ,which has lowest hidden size equals 32, has large difference between train and test accuracy.
   - **Changing hidden size and fixing optimizer to adam and dropout to 0.4 :**
     From model 3 to 5, we noticed that There is no significant difference between these trails and previous 3 trails. As shown in Figure [23], the only difference is the difference between train and test accuracy(in model 5 with hidden size 32) is increased than in model 2 which has the same hidden size.That means the overfitting is increasing. Increasing Dropout rate reduces overfitting.
   - **Changing hidden size and fixing optimizer to adam and dropout to 0.2 :**
     From model 5 to 8, we noticed that There is no significant difference between these trails and previous trails.
   - **Changing hidden size and fixing optimizer to SGD and dropout to 0.5 :**
     From model 9 to 11, we noticed that SGD it just got worse. The accuracy in both train and test decreased significantly. Also, The overfitting increased. Model 10 (HIGH HIDDEN SIZEN) is fairly good.Increasing hidden size improves the performance
   - **Changing hidden size and fixing optimizer to SGD and dropout to 0.4 :**
     From model 12 to 14, we noticed that there is a little increasing in overfitting rather than choosing drop out rate with 0.5 in SGD trails.
   - **Changing hidden size and fixing optimizer to SGD and dropout to 0.2 :**
     From model 15 to 17, we noticed that there is a little increasing in overfitting rather than

choosing drop out rate with 0.4 in SGD trails.

Increasing hidden size and Increasing dropout rate improve the performance. Adam optimizer is better than SGD optimizer.

- **Changing hidden size and fixing optimizer to rmsprop and dropout to 0.5 :**
  From model 18 to 20 , we noticed that there is a significant improvement in performance in using rmsprop compared to SGD and Adam. Increasing hidden size reduced overfit and increased accuracy. As shown in Figure [23] that model 19 with hidden size equals 256 which is largest one has low difference between train and test accuracy. Also, model20 ,which has lowest hidden size equals 32, has large difference between train and test accuracy. As shown in Figure [23], Model2 (adam hidden size = 32) is better than model20(rmsprop and hidden size = 32). RMSPROP is more sensitive to hidden size and works well with large hidden size. In general, RMSPORP is the best compared to adam and SGD
- **Changing hidden size and fixing optimizer to rmsprop and dropout to 0.4 :**
  From model 21 to 23, we noticed that the difference between train and test accuracy(in model 22 with hidden size 32) is increased than in model 19 which has the same hidden size.That means the overfitting is increasing. the strange thing that the accuracy improved in model 21 and 23 compared to model 18 and 20. Increasing Dropout rate is good with high hidden size.
- **Changing hidden size and fixing optimizer to rmsprop and dropout to 0.2 :**
  From model 24 to 26, we noticed that there is an improvement in model 25 compared to model22.but still model 19 is best compared to model 25 and 22. Dropout rate improves the performance.

  RMSPROP is the best optimizer .Increasing hidden size and dropout rate improve the performance. Adam is also good. SGD is the bad optimizer.

  At the end of this iteration,GridSearchCV will save the best model with best hyper parameters.

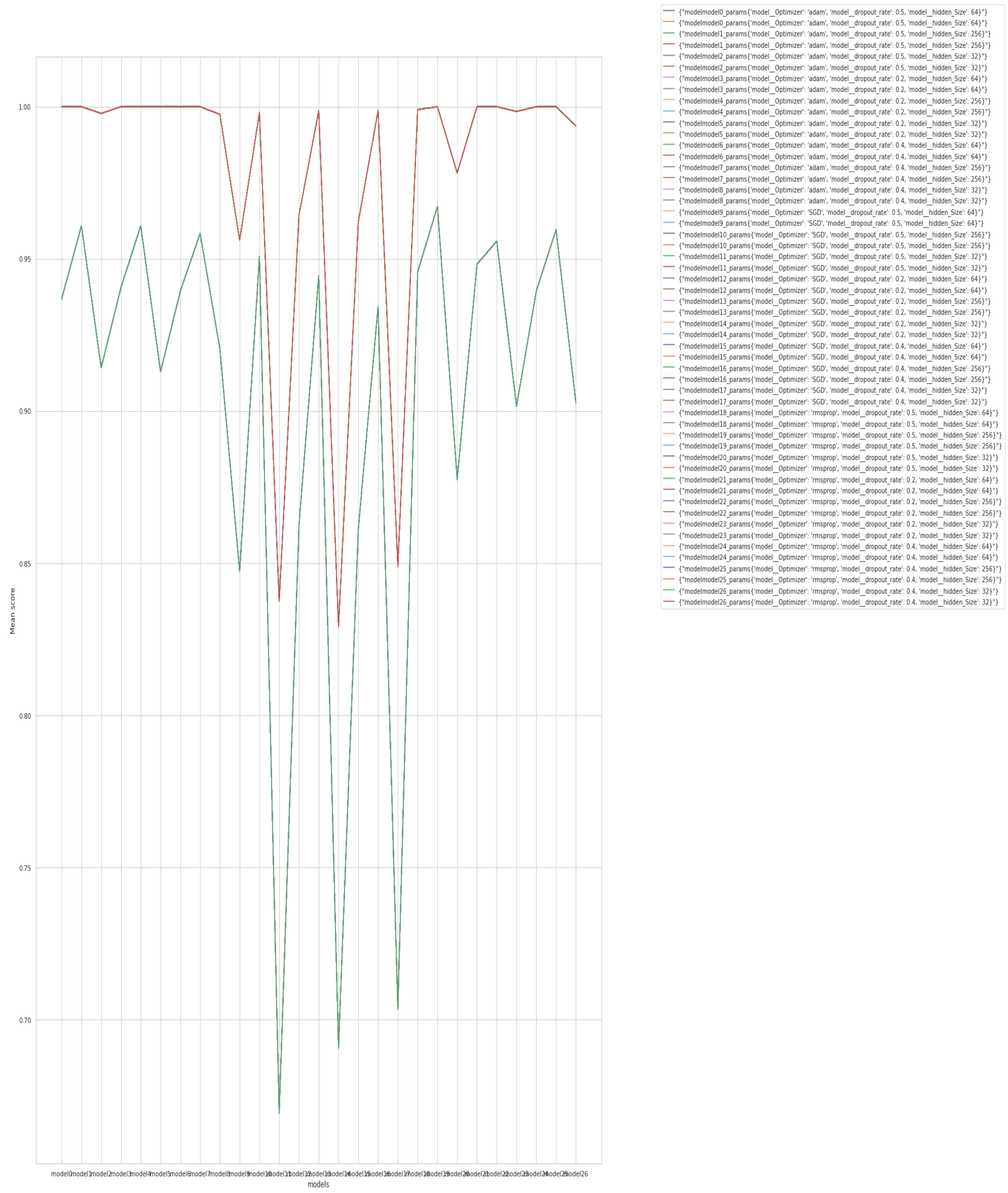


Figure 23: Tune Hyper-parameters

2. **2nd iteration (Fixing batch size to 32) :**

   Figure [24] shows that :

   - From the previous iteration, we found that the largest hidden size achieved the best performance. As shown Figure [24],the largest hidden size achieved the best performance.So, we will compare optimizer and dropout rate in hidden size 256
   - From Figure [24], Increasing batch size improved SGD.There is a significant improvement in decreasing overfit compared to SGD in figure [23].
   - In adam, the model 7 (droprate = 0 .4 and hidden size = 256)is the best model compared to model 1(droprate = 0.5 and hidden size = 256) and 4 ((droprate = 0.2 and hidden size = 256)
   - In SGD, the model 13 (droprate = 0 .2 and hidden size = 256)is the best model compared to model 10(droprate = 0.5 and hidden size = 256) and 16 ((droprate = 0.4 and hidden size = 256)
   - In RMSPROP, the model 22 (droprate = 0.2 and hidden size = 256)is the best model compared to model 19(droprate = 0.5 and hidden size = 256) and 25 ((droprate = 0.4 and hidden size = 256)

     In general, RMSPROP is still the best compared to adam and SGD. We can't say that the dropout rate is the best in all cases. Increasing batch size and hidden size improve the performance.

   At the end of this iteration,GridSearchCV will save the best model with best hyper parameters.

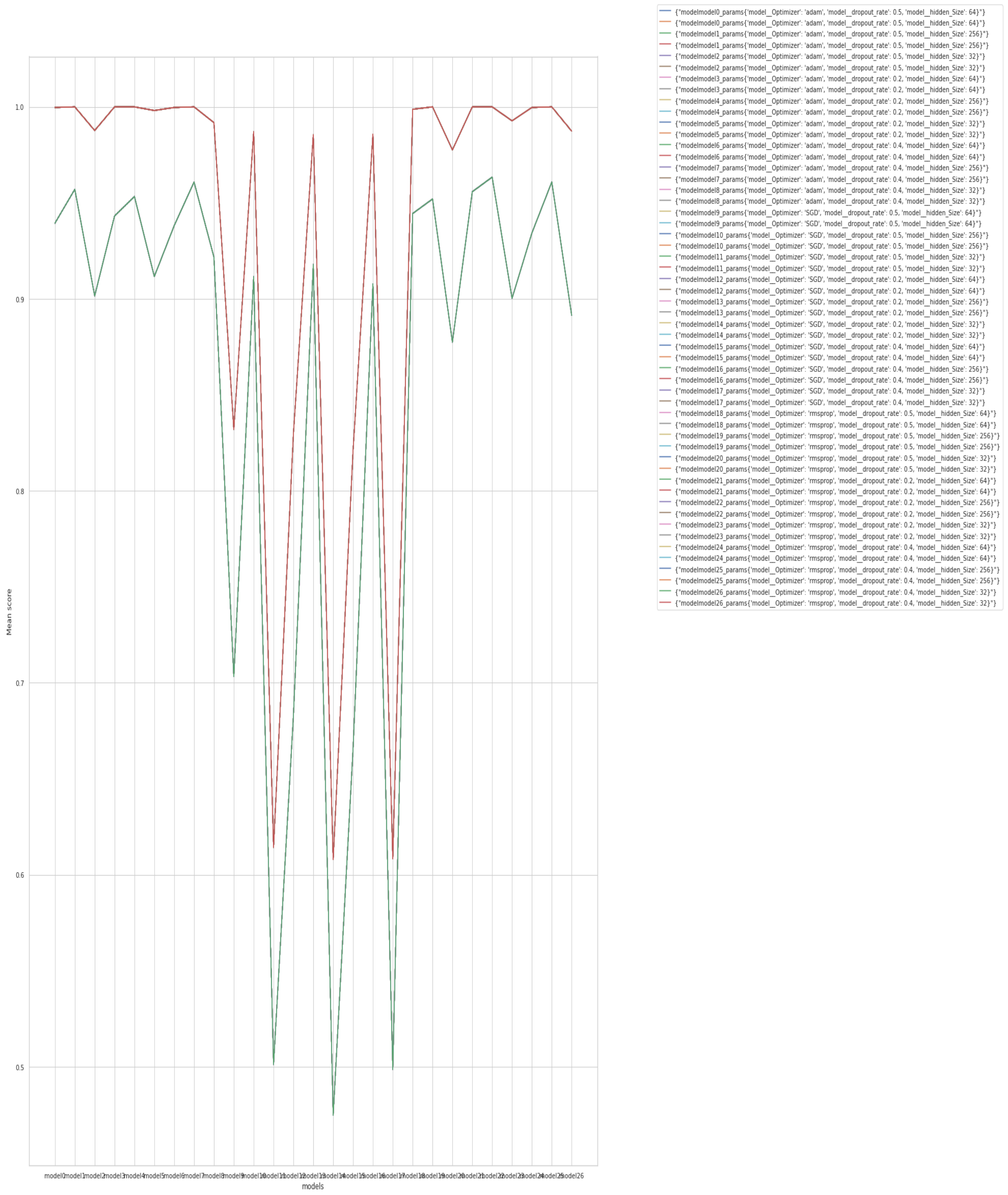


Figure 24: Tune Hyper-parameters

3. **3rd iteration (Fixing batch size to 64) :**

   Figure [24] shows that :

   - From the previous iteration, we found that the largest hidden size achieved the best performance. As shown Figure [24],the largest hidden size achieved the best performance.So, we will compare optimizer and dropout rate in hidden size 256
   - From Figure [24], Increasing batch size improved SGD.There is a significant improvement in decreasing overfit compared to SGD in figure [23].
   - In adam, the model 7 (droprate = 0.4 and hidden size = 256), model 1(droprate = 0.5 and hidden size = 256), model 4 (droprate = 0.2 and hidden size = 256) are almost the same.
   - In SGD, the model 13 (droprate = 0.2 and hidden size = 256)is the best model compared to model 10(droprate = 0.5 and hidden size = 256) and 16 (droprate = 0.4 and hidden size = 256)
   - In RMSPROP, the model 25(droprate = 0.4 and hidden size = 256)is the best model compared to model 19(droprate = 0.5 and hidden size = 256) and 22 ((droprate = 0.2 and hidden size = 256)

     In general, RMSPROP is still the best compared to adam and SGD. We can't say that the dropout rate is the best in all cases. Increasing hidden size improve the performance. Increasing batch size doesn't achieve significant difference.

   At the end of this iteration,GridSearchCV will save the best model with best hyper parameters.

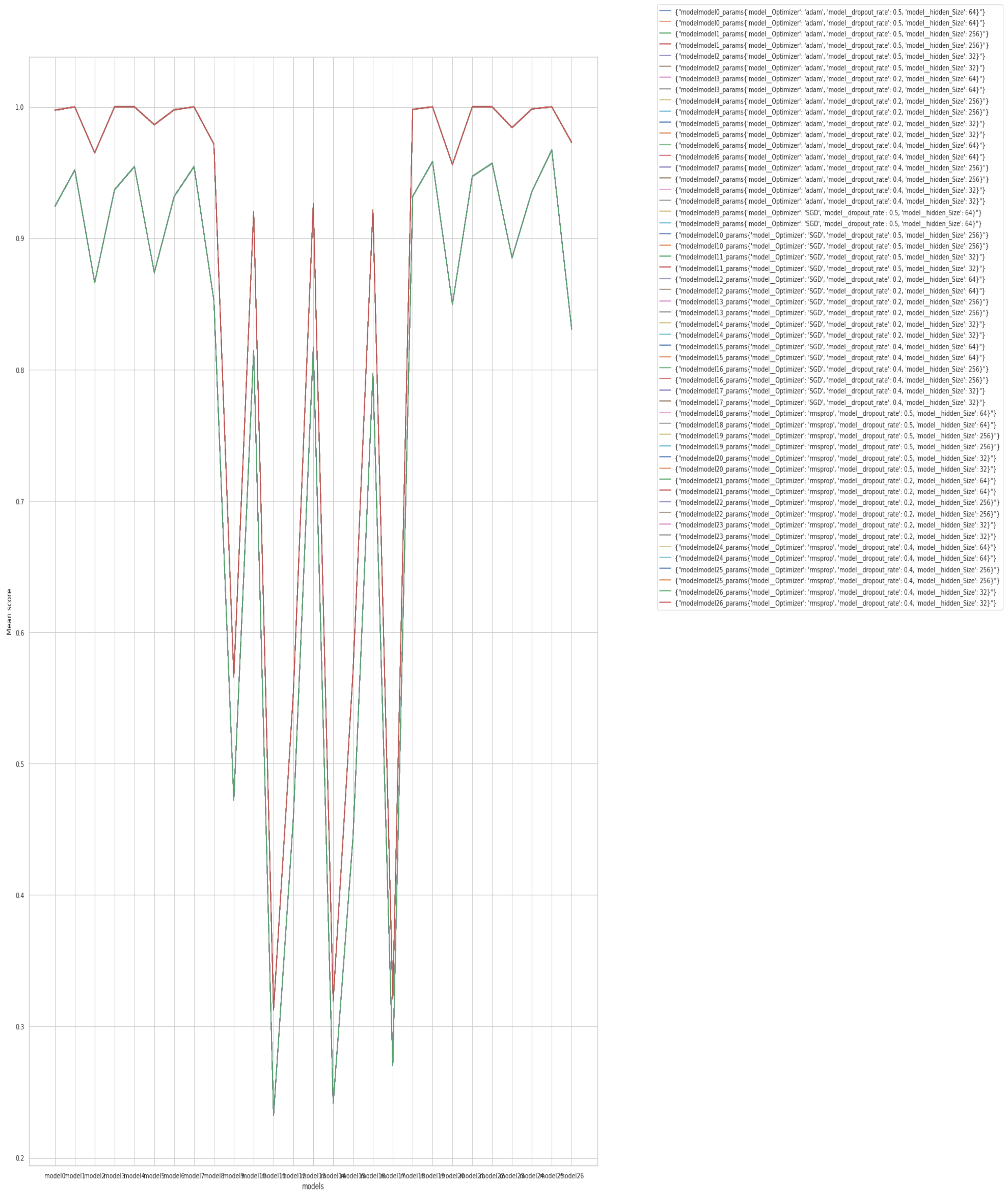
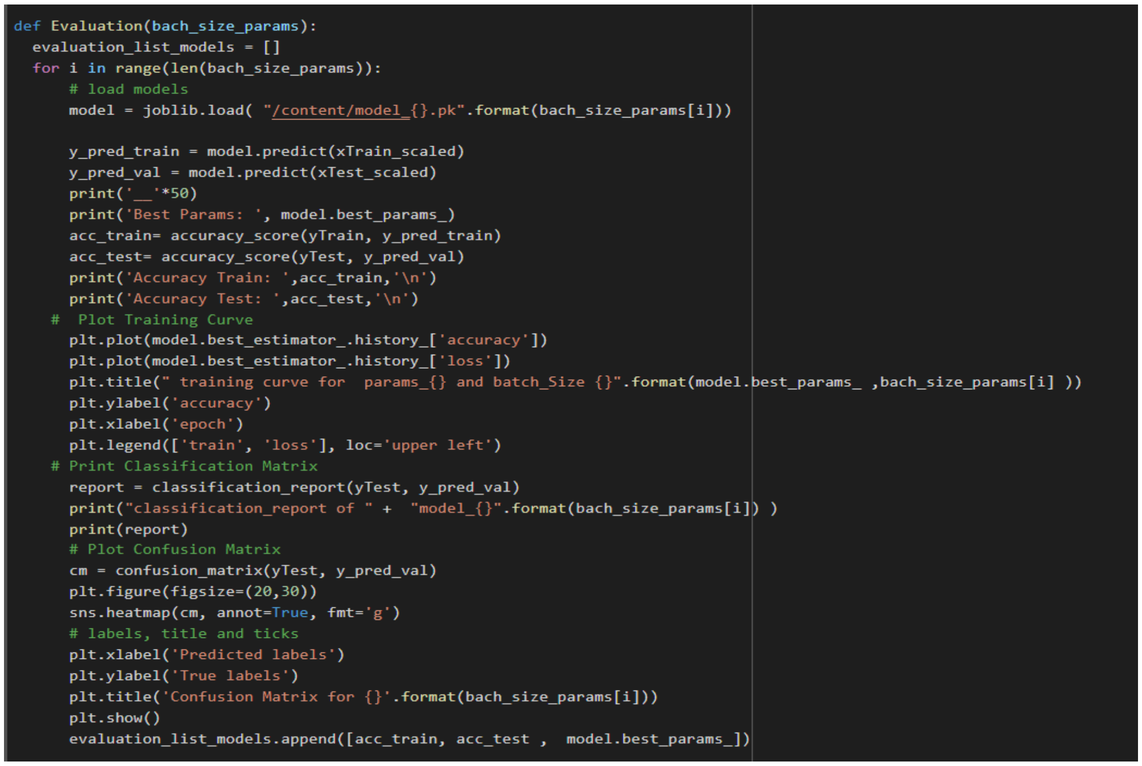


Figure 25: Tune Hyper-parameters

10 Evaluation

In this section, we evaluated the best 3 models that we saved from the previous step.

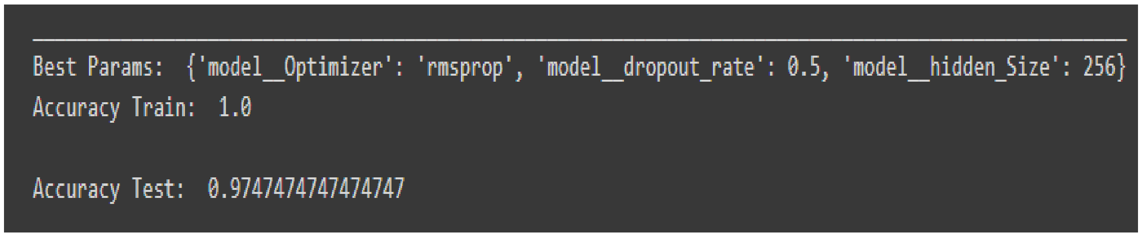
we implemented the function below figure [26].This function load the model and evaluate its performance on train/test set.Also, it plot training accuracy and loss curves for each model. For more evaluation, the function also print classification report and confusion matrix.



```
def Evaluation(bach_size_params):
  evaluation_list_models = []
  for i in range(len(bach_size_params)):
      # load models
      model = joblib.load( "/content/model_{}.pk".format(bach_size_params[i]))

      y_pred_train = model.predict(xTrain_scaled)
      y_pred_val = model.predict(xTest_scaled)
      print('__'*50)
      print('Best Params: ', model.best_params_)
      acc_train= accuracy_score(yTrain, y_pred_train)
      acc_test= accuracy_score(yTest, y_pred_val)
      print('Accuracy Train: ',acc_train,'\n')
      print('Accuracy Test: ',acc_test,'\n')
    #  Plot Training Curve
      plt.plot(model.best_estimator_.history_['accuracy'])
      plt.plot(model.best_estimator_.history_['loss'])
      plt.title(" training curve for  params_{} and batch_Size {}".format(model.best_params_ ,bach_size_params[i] ))
      plt.ylabel('accuracy')
      plt.xlabel('epoch')
      plt.legend(['train', 'loss'], loc='upper left')
    # Print Classification Matrix
      report = classification_report(yTest, y_pred_val)
      print("classification_report of " +  "model_{}".format(bach_size_params[i]) )
      print(report)
      # Plot Confusion Matrix
      cm = confusion_matrix(yTest, y_pred_val)
      plt.figure(figsize=(20,30))
      sns.heatmap(cm, annot=True, fmt='g')
      # labels, title and ticks
      plt.xlabel('Predicted labels')
      plt.ylabel('True labels')
      plt.title('Confusion Matrix for {}'.format(bach_size_params[i]))
      plt.show()
      evaluation_list_models.append([acc_train, acc_test ,  model.best_params_])
```

Figure 26: Function to evaluate model

1. **Accuracy of model(1st iteration with batch size = 16)**
   As you can see in Figure [27],the best hyper parameters are the same as we got in the previous section.



```
____________________________________________________________________________________________________
Best Params:  {'model__Optimizer': 'rmsprop', 'model__dropout_rate': 0.5, 'model__hidden_Size': 256}
Accuracy Train:  1.0

Accuracy Test:  0.9747474747474747
```

Figure 27: Best hyper parameters and Accuracy in train and test for model 1

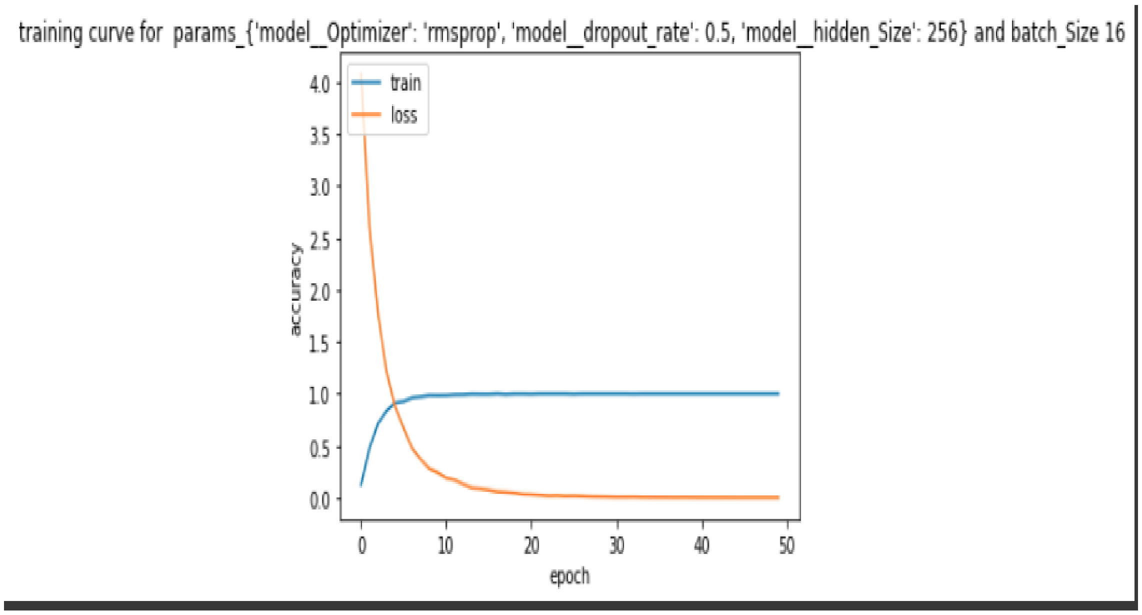
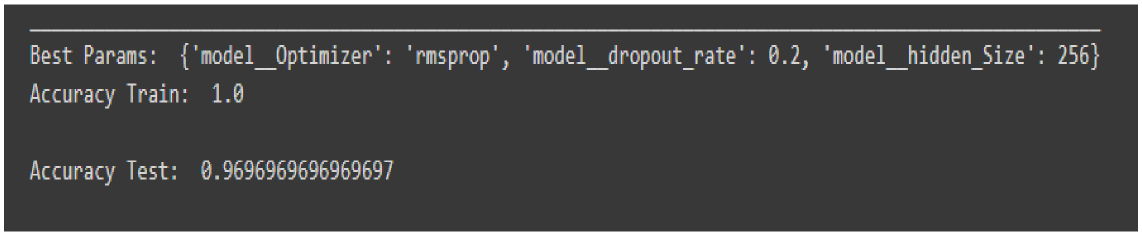


Figure 28: Training loss / accuracy curve for model 1

2. **Accuracy of model(2nd iteration with batch size = 32)**
   As you can see in Figure [29],the best hyper parameters are the same as we got in the previous section.



```
Best Params:  {'model__Optimizer': 'rmsprop', 'model__dropout_rate': 0.2, 'model__hidden_Size': 256}
Accuracy Train:  1.0

Accuracy Test:  0.9696969696969697
```

Figure 29: Best hyper parameters and Accuracy in train and test for model 2

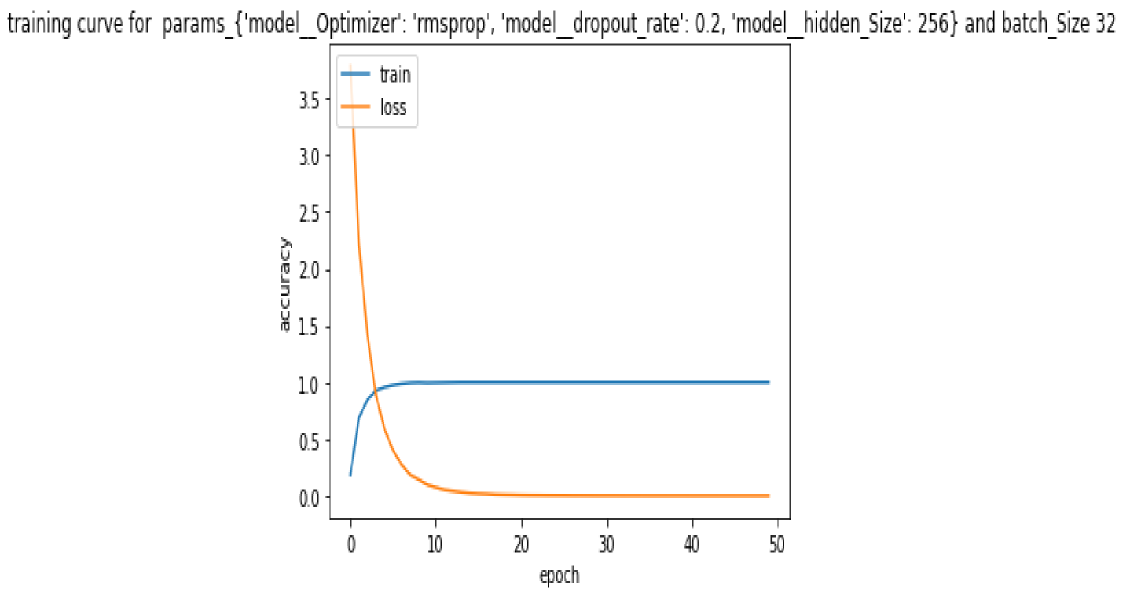
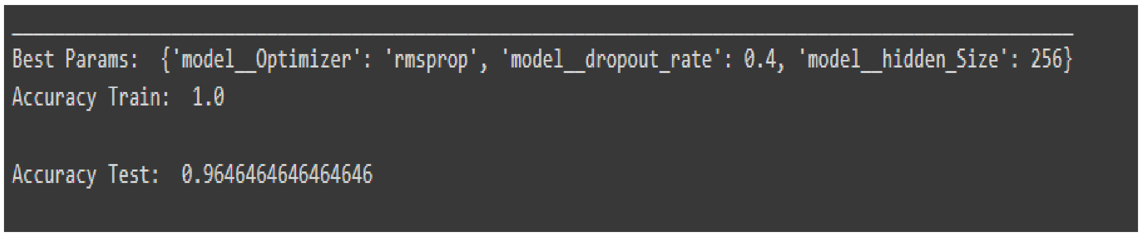


Figure 30: Training loss / accuracy curve for model 2

3. **Accuracy of model(3rd iteration with batch size = 64)**
   As you can see in Figure [31],the best hyper parameters are the same as we got in the previous section.



```
Best Params:  {'model__Optimizer': 'rmsprop', 'model__dropout_rate': 0.4, 'model__hidden_Size': 256}
Accuracy Train:  1.0

Accuracy Test:  0.9646464646464646
```

Figure 31: Best hyper parameters and Accuracy in train and test for model 3

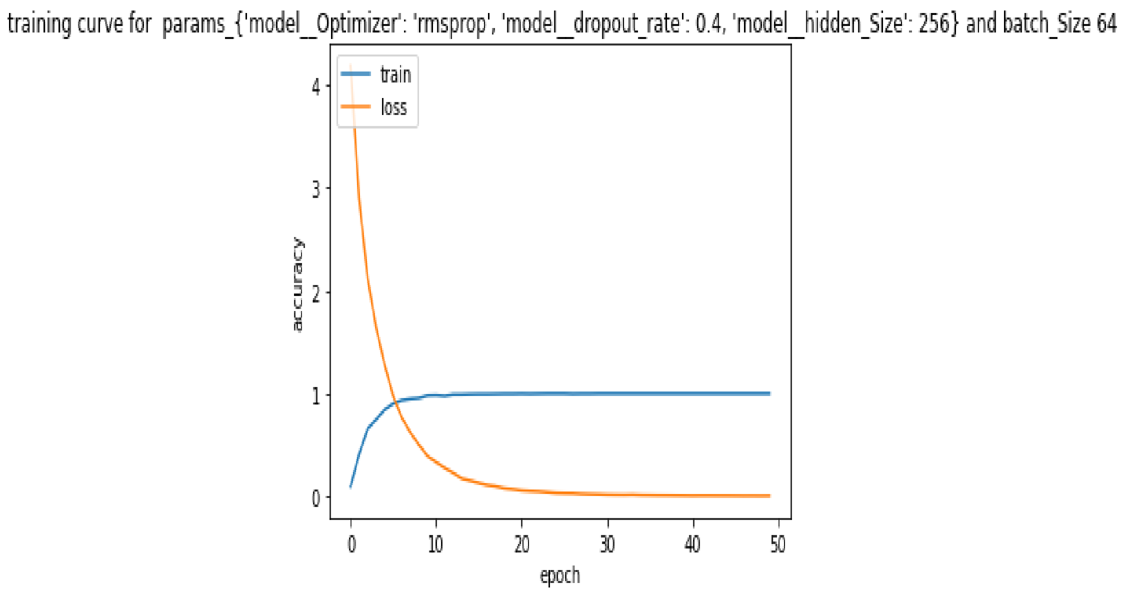


Figure 32: Training loss / accuracy curve for model 3

From Figure [27],[28],[29][30],[31] and [32], we can notice that all the models performs well. There a little difference between them.All of these models used rmsprop as optimizer.

Let’s put them in one plot to get

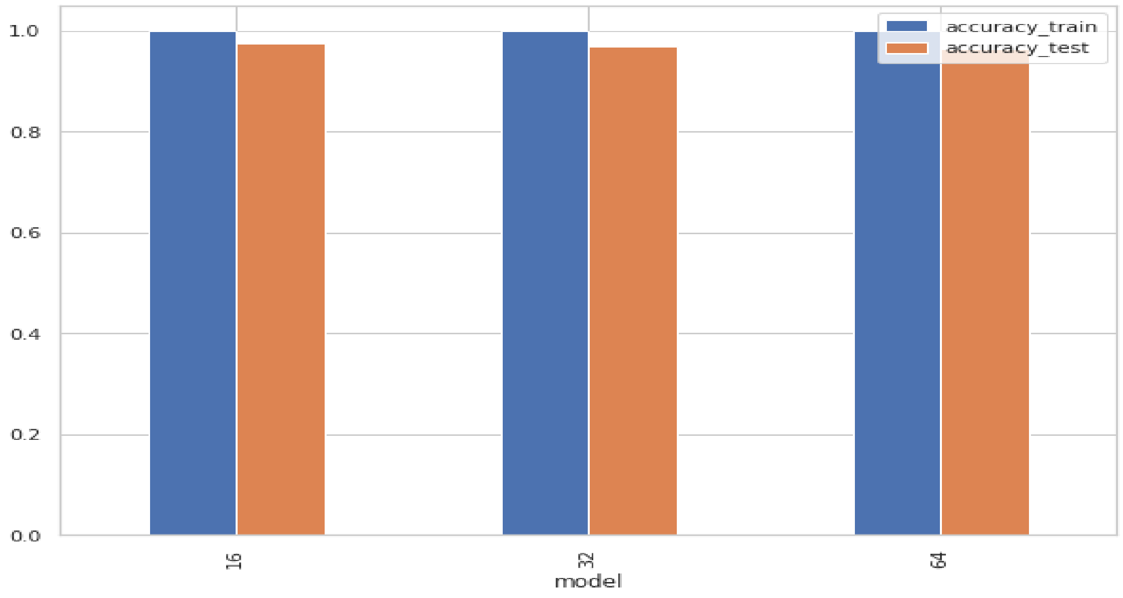
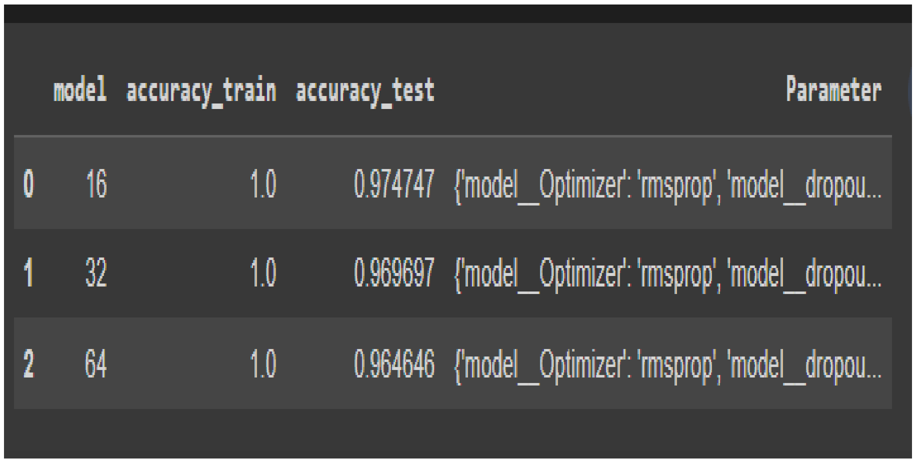


Figure 33: The best 3 models



	model	accuracy_train	accuracy_test	Parameter
0	16	1.0	0.974747	{'model__Optimizer': 'rmsprop', 'model__dropou...
1	32	1.0	0.969697	{'model__Optimizer': 'rmsprop', 'model__dropou...
2	64	1.0	0.964646	{'model__Optimizer': 'rmsprop', 'model__dropou...

Figure 34: The table of best 3 models

As we can see the best model among 3 models is model 1 with the following hyper parameters (model__Optimizer: rmsprop, model__dropout_rate: 0.5, model__hidden_Size: 256)

11 Conclusion

1. RMSPROP and adam are good optimizerS compared to SGD
2. RMSPROP is the best compared to adam and SGD.
3. Increasing hidden size improve the performance as we see in the Hyper tune Section.
4. We can't say that the dropout rate is the best in all cases. as you can see the best 3 models have different dropout rate. Although, the best among them was model with high dropout rate (0.5). But the difference between them wasn't large as shown in figure 33 and 34
5. Increasing batch size doesn't achieve significant difference in improving performance as shown in figure 33 and 34

.