Experiment 7: Genetic Algorithm

Name - Pravesh P. Ganwani
T.E. I.T.
Batch B
UID – 2018140021

Aim:
To use genetic algorithm for hyperparameter tuning

Problem Statement:
Choose a classification dataset of your choice from any of the following Repository Links, download it:

1. Kaggle: https://www.kaggle.com/
2. UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/index.php

Perform Linear Regression on the chosen dataset.

Your notebook should contain:
1. Basic EDA

[***Hint***: Follow the steps in Titanic notebook uploaded on moodle under Expt 3 reference material]

Tool/Language:
Programming language: Python
Libraries: numpy, pandas, sklearn, matplotlib, seaborn

Code with visualization graphs:
1) **Dataset Chosen:** Musk Dataset
2) **Dataset Description:** It contains a set of 102 molecules, out of which 39 are identified by humans as having odour that can be used in perfumery and 69 not having the desired odour. The dataset contains 6,590 low-energy conformations of these molecules, containing 166 features.
3) **Code:**

```
# Importing the Libraries

import numpy as np
import pandas as pd
import random
import xgboost as xgb
import matplotlib.pyplot as plt
import io
import seaborn as sns
plt.style.use('ggplot')

"""Dataset Chosen - https://archive.ics.uci.edu/ml/machine-learning-
databases/musk/
```

```
It contains a set of 102 molecules, out of which 39 are identified by humans as h
aving odor that can be used in perfumery and 69 not having the desired odor. The
dataset contains 6,590 low-
energy conformations of these molecules, containing 166 features.
"""

# Importing Dataset to Google Colab
from google.colab import files
uploaded = files.upload()

"""# **Basic EDA**"""

# Reading the Dataset
df = pd.read_csv(io.BytesIO(uploaded['musk_csv.csv']))
df.head()
```

	ID	molecule_name	conformation_name	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15	f16	f17	f18	f19	f20	f21	f22	f23	f24	f25	f26	f27	f28
0	1	MUSK-211	211_1+1	46	-108	-60	-69	-117	49	38	-161	-8	5	-323	-220	-113	-299	-283	-307	-31	-106	-227	-42	-59	-22	-67	189	81	17	-27	-89
1	2	MUSK-211	211_1+10	41	-188	-145	22	-117	-6	57	-171	-39	-100	-319	-111	-228	-281	-281	-300	54	-149	-98	-196	-27	-22	2	75	49	-34	45	-91
2	3	MUSK-211	211_1+11	46	-194	-145	28	-117	73	57	-168	-39	-22	-319	-111	-104	-283	-282	-303	52	-152	-97	-225	-28	-22	2	179	49	-33	46	-88
3	4	MUSK-211	211_1+12	41	-188	-145	22	-117	-7	57	-170	-39	-99	-319	-111	-228	-282	-281	-301	54	-150	-98	-196	-28	-22	2	77	48	-34	46	-91
4	5	MUSK-211	211_1+13	41	-188	-145	22	-117	-7	57	-170	-39	-99	-319	-111	-228	-282	-281	-301	54	-150	-98	-196	-28	-22	2	78	48	-34	46	-91

5 rows × 170 columns

```
# Looking for Datatype and No. of Null Entries

df.info()
```

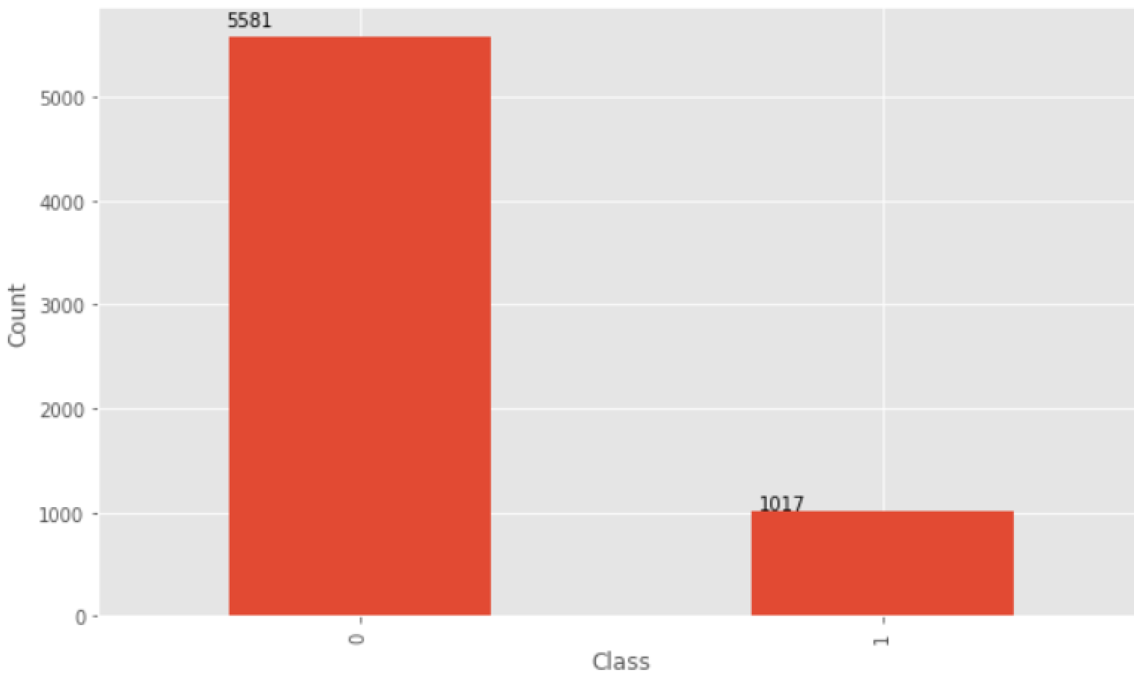
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6598 entries, 0 to 6597
Columns: 170 entries, ID to class
dtypes: int64(168), object(2)
memory usage: 8.6+ MB
```

```
df.describe()
```

	ID	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13
count	6598.00000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000	6598.000000
mean	3299.50000	58.945135	-119.128524	-73.146560	-0.628372	-103.533495	18.359806	-14.108821	-1.858290	-86.003031	-44.495756	-119.456502	-84.929221	-61.911185
std	1904.82287	53.249007	90.813375	67.956235	80.444617	64.387559	80.593655	115.315673	90.372537	108.326676	72.088903	108.911397	79.541410	61.444281
min	1.00000	-31.000000	-199.000000	-167.000000	-114.000000	-118.000000	-183.000000	-171.000000	-225.000000	-245.000000	-286.000000	-328.000000	-321.000000	-305.000000
25%	1650.25000	37.000000	-193.000000	-137.000000	-70.000000	-117.000000	-28.000000	-159.000000	-85.000000	-217.000000	-96.750000	-207.000000	-114.000000	-85.000000
50%	3299.50000	44.000000	-149.000000	-99.000000	-25.000000	-117.000000	33.000000	27.000000	19.000000	-40.000000	-29.000000	-83.000000	-86.000000	-66.000000
75%	4948.75000	53.000000	-95.000000	-19.000000	42.000000	-116.000000	74.000000	57.000000	61.000000	-21.000000	4.000000	-46.000000	-35.000000	-45.000000
max	6598.00000	292.000000	95.000000	81.000000	161.000000	325.000000	200.000000	220.000000	320.000000	147.000000	231.000000	176.000000	184.000000	195.000000

8 rows × 168 columns

```
fig = plt.figure(figsize=(10,6))
musk_counts = df['class'].value_counts()
ax = musk_counts.plot.bar()
ax.set_xlabel('Class')
ax.set_ylabel('Count')
for p in ax.patches:
    ax.annotate(str(p.get_height()), (p.get_x() * 1.02, p.get_height() * 1.02))
```




```
"""# **Data Preprocessing**"""

X = df.iloc[:, 3:169].values # Feature Classes
y = df.iloc[:, 169].values # Target Class

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, rando
m_state = 97)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# XGboost Classifier
```

```
xgDMatrix = xgb.DMatrix(X_train, y_train)
xgbDMatrixTest = xgb.DMatrix(X_test, y_test)

"""# **Initialization**"""

random.seed(723)
np.random.seed(723)

def initialize_poplulation(numberOfParents):
    learningRate = np.empty([numberOfParents, 1])
    nEstimators = np.empty([numberOfParents, 1], dtype = np.uint8)
    maxDepth = np.empty([numberOfParents, 1], dtype = np.uint8)
    minChildWeight = np.empty([numberOfParents, 1])
    gammaValue = np.empty([numberOfParents, 1])
    subSample = np.empty([numberOfParents, 1])
    colSampleByTree =  np.empty([numberOfParents, 1])

    for i in range(numberOfParents):
        print(i)
        learningRate[i] = round(random.uniform(0.01, 1), 2)
        nEstimators[i] = random.randrange(10, 1500, step = 25)
        maxDepth[i] = int(random.randrange(1, 10, step= 1))
        minChildWeight[i] = round(random.uniform(0.01, 10.0), 2)
        gammaValue[i] = round(random.uniform(0.01, 10.0), 2)
        subSample[i] = round(random.uniform(0.01, 1.0), 2)
        colSampleByTree[i] = round(random.uniform(0.01, 1.0), 2)

    population = np.concatenate((learningRate, nEstimators, maxDepth, minChildWei
ght, gammaValue, subSample, colSampleByTree), axis= 1)
    return population

numberOfParents = 8 # No. of Parents to start
numberOfParentsMating = 4 # No. of Parents that will mate
numberOfParameters = 7 # No. of Parameters that will be optimized
numberOfGenerations = 4 # No. of Generation that will be created

# Population Size

populationSize = (numberOfParents, numberOfParameters)

# Initialize the population with randomly generated parameters
population = initialize_poplulation(numberOfParents)

# Array to store the Fitness Hitory
```

```
fitnessHistory = np.empty([numberOfGenerations+1, numberOfParents])

# Array to store the Value of each Parameter for Each Parent and Generation
populationHistory = np.empty([(numberOfGenerations+1)*numberOfParents, numberOfPa
rameters])

# Initial Parameters to History
populationHistory[0:numberOfParents, :] = population

"""# **Parent Selection (Survival of the Fittest)**"""

from sklearn.metrics import f1_score

# Function to Predict F1_score
def fitness_f1score(y_true, y_pred):
    fitness = round((f1_score(y_true, y_pred, average='weighted')), 4)
    return fitness

# Train the data and find Fitness Score
def train_population(population, dMatrixTrain, dMatrixtest, y_test):
    fScore = []
    for i in range(population.shape[0]):
        param = { 'objective':'binary:logistic',
              'learning_rate': population[i][0],
              'n_estimators': population[i][1],
              'max_depth': int(population[i][2]),
              'min_child_weight': population[i][3],
              'gamma': population[i][4],
              'subsample': population[i][5],
              'colsample_bytree': population[i][6],
              'seed': 24}
        num_round = 100
        xgbT = xgb.train(param, dMatrixTrain, num_round)
        preds = xgbT.predict(dMatrixtest)
        preds = preds>0.5
        fScore.append(fitness_f1score(y_test, preds))
    return fScore

# Selecting Parents for Mating
def new_parents_selection(population, fitness, numParents):
    selectedParents = np.empty((numParents, population.shape[1])) #create an arra
y to store fittest parents

    #find the top best performing parents
    for parentId in range(numParents):
```

```
        bestFitnessId = np.where(fitness == np.max(fitness))
        bestFitnessId  = bestFitnessId[0][0]
        selectedParents[parentId, :] = population[bestFitnessId, :]
        fitness[bestFitnessId] = -1 #set this value to negative, in case of F1-
score, so this parent is not selected again
    return selectedParents

"""# **Crossover**"""

'''
Mate these parents to create chilren having parameters from these parents (we are
 using uniform crossover method)
'''
def crossover_uniform(parents, childrenSize):

    crossoverPointIndex = np.arange(0, np.uint8(childrenSize[1]), 1, dtype= np.ui
nt8) #get all the index
    crossoverPointIndex1 = np.random.randint(0, np.uint8(childrenSize[1]), np.uin
t8(childrenSize[1]/2)) # select half  of the indexes randomly
    crossoverPointIndex2 = np.array(list(set(crossoverPointIndex) - set(crossover
PointIndex1))) #select leftover indexes

    children = np.empty(childrenSize)

    '''
    Create child by choosing parameters from two paraents selected using new_pare
nt_selection function. The parameter values
    will be picked from the indexes, which were randomly selected above.
    '''
    for i in range(childrenSize[0]):

        #find parent 1 index
        parent1_index = i%parents.shape[0]
        #find parent 2 index
        parent2_index = (i+1)%parents.shape[0]
        #insert parameters based on random selected indexes in parent 1
        children[i, crossoverPointIndex1] = parents[parent1_index, crossoverPoint
Index1]
        #insert parameters based on random selected indexes in parent 1
        children[i, crossoverPointIndex2] = parents[parent2_index, crossoverPoint
Index2]
    return children

"""# **Mutation**"""
```

```
'''
Introduce some mutation in the children. In case of XGboost we will introdcue mut
ation randomly on each parameter one at a time,
based on which parameter is selected at random. Initially, we will define the max
imum/minimum value that is allowed for the parameter, to prevent the
out the range error during runtime. Subsequently, we will generate mutation value
 and add it to the parameter, and return the mutated offspring!!!
'''
def mutation(crossover, numberOfParameters):
    #Define minimum and maximum values allowed for each parameter

    minMaxValue = np.zeros((numberOfParameters, 2))

    minMaxValue[0:] = [0.01, 1.0] #min/max learning rate
    minMaxValue[1, :] = [10, 2000] #min/max n_estimator
    minMaxValue[2, :] = [1, 15] #min/max depth
    minMaxValue[3, :] = [0, 10.0] #min/max child_weight
    minMaxValue[4, :] = [0.01, 10.0] #min/max gamma
    minMaxValue[5, :] = [0.01, 1.0] #min/maxsubsample
    minMaxValue[6, :] = [0.01, 1.0] #min/maxcolsample_bytree

    # Mutation changes a single gene in each offspring randomly.
    mutationValue = 0
    parameterSelect = np.random.randint(0, 7, 1)
    print(parameterSelect)
    if parameterSelect == 0: #learning_rate
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)
    if parameterSelect == 1: #n_estimators
        mutationValue = np.random.randint(-200, 200, 1)
    if parameterSelect == 2: #max_depth
        mutationValue = np.random.randint(-5, 5, 1)
    if parameterSelect == 3: #min_child_weight
        mutationValue = round(np.random.uniform(5, 5), 2)
    if parameterSelect == 4: #gamma
        mutationValue = round(np.random.uniform(-2, 2), 2)
    if parameterSelect == 5: #subsample
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)
    if parameterSelect == 6: #colsample
        mutationValue = round(np.random.uniform(-0.5, 0.5), 2)

    #indtroduce mutation by changing one parameter, and set to max or min if it g
oes out of range
    for idx in range(crossover.shape[0]):
        crossover[idx, parameterSelect] = crossover[idx, parameterSelect] + mutat
ionValue
```

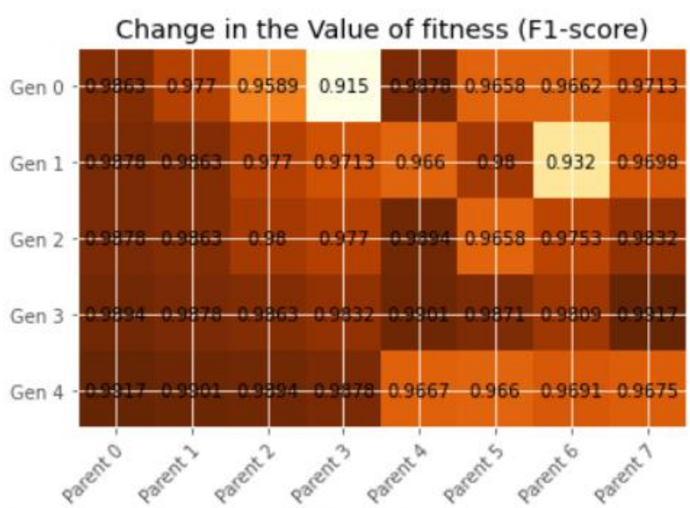
```
        if(crossover[idx, parameterSelect] > minMaxValue[parameterSelect, 1]):
            crossover[idx, parameterSelect] = minMaxValue[parameterSelect, 1]
        if(crossover[idx, parameterSelect] < minMaxValue[parameterSelect, 0]):
            crossover[idx, parameterSelect] = minMaxValue[parameterSelect, 0]
    return crossover

"""# **Implementation**"""

for generation in range(numberOfGenerations):
    print("This is No. %s Generation" % (generation))

    # Train the dataset and obtain Fitness
    fitnessValue = train_population(population=population, dMatrixTrain=xgDMatrix
, dMatrixtest=xgbDMatrixTest, y_test=y_test)
    fitnessHistory[generation, :] = fitnessValue

    # Best Score in the current Iteration
    print('Best F1 score in the this Iteration = {}'.format(np.max(fitnessHistory
[generation, :])))

    # Survival of the Fittest
    parents = new_parents_selection(population=population, fitness=fitnessValue,
numParents=numberOfParentsMating)

    # Mating
    children = crossover_uniform(parents=parents, childrenSize=(populationSize[0]
 - parents.shape[0], numberOfParameters))

    # Adding Mutation to create Genetic Diversity
    children_mutated = mutation(children, numberOfParameters)

    '''
    We will create new population, which will contain parents that where selected
 previously based on the
    fitness score and rest of them  will be children
    '''
    population[0:parents.shape[0], :] = parents # Fittest Parents
    population[parents.shape[0]:, :] = children_mutated # Children

    populationHistory[(generation+1)*numberOfParents : (generation+1)*numberOfPar
ents+ numberOfParents , :] = population


# Best solution from the final Iteration
```

```
fitness = train_population(population=population, dMatrixTrain=xgDMatrix, dMatrix
test=xgbDMatrixTest, y_test=y_test)
fitnessHistory[generation+1, :] = fitness

bestFitnessIndex = np.where(fitness == np.max(fitness))[0][0]

# Best Fitness
print("Best Fitness is =", fitness[bestFitnessIndex])

# Best Parameters
print("Best Parameters are - ")
print('learning_rate', population[bestFitnessIndex][0])
print('n_estimators', population[bestFitnessIndex][1])
print('max_depth', int(population[bestFitnessIndex][2]))
print('min_child_weight', population[bestFitnessIndex][3])
print('gamma', population[bestFitnessIndex][4])
print('subsample', population[bestFitnessIndex][5])
print('colsample_bytree', population[bestFitnessIndex][6])
```

```
This is No. 0 Generation
Best F1 score in the this Iteration = 0.9878
[2]
This is No. 1 Generation
Best F1 score in the this Iteration = 0.9878
[2]
This is No. 2 Generation
Best F1 score in the this Iteration = 0.9894
[1]
This is No. 3 Generation
Best F1 score in the this Iteration = 0.9917
[2]
Best Fitness is = 0.9917
Best Parameters are -
learning_rate 0.55
n_estimators 10.0
max_depth 3
min_child_weight 2.09
gamma 0.27
subsample 0.77
colsample_bytree 0.61
```

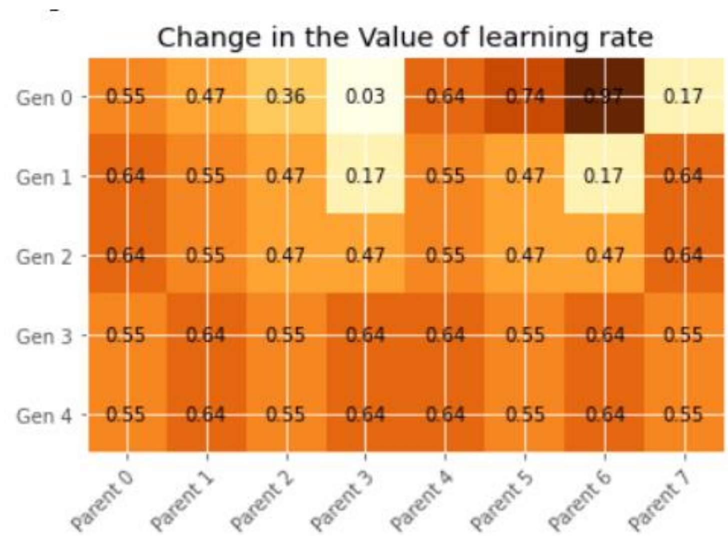
```

"""# **Visualisation**"""

'''
This function will allow us to genrate the heatmap for various parameters and fit
ness to visualize
```

```
how each parameter and fitness changes with each generation
'''
def plot_parameters(numberOfGenerations, numberOfParents, parameter, parameterNam
e):
    generationList = ["Gen {}".format(i) for i in range(numberOfGenerations+1)]
    populationList = ["Parent {}".format(i) for i in range(numberOfParents)]
    fig, ax = plt.subplots()
    im = ax.imshow(parameter, cmap=plt.get_cmap('YlOrBr'))

    ax.set_xticks(np.arange(len(populationList)))
    ax.set_yticks(np.arange(len(generationList)))

    ax.set_xticklabels(populationList)
    ax.set_yticklabels(generationList)

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor
")

    for i in range(len(generationList)):
        for j in range(len(populationList)):
            text = ax.text(j, i, parameter[i, j], ha="center", va="center", color
="k")

    ax.set_title("Change in the Value of " + parameterName)
    fig = plt.figure(figsize=(20,10))
    fig.tight_layout()
    plt.show()

plot_parameters(numberOfGenerations, numberOfParents, fitnessHistory, "fitness (F
1-score)")
```



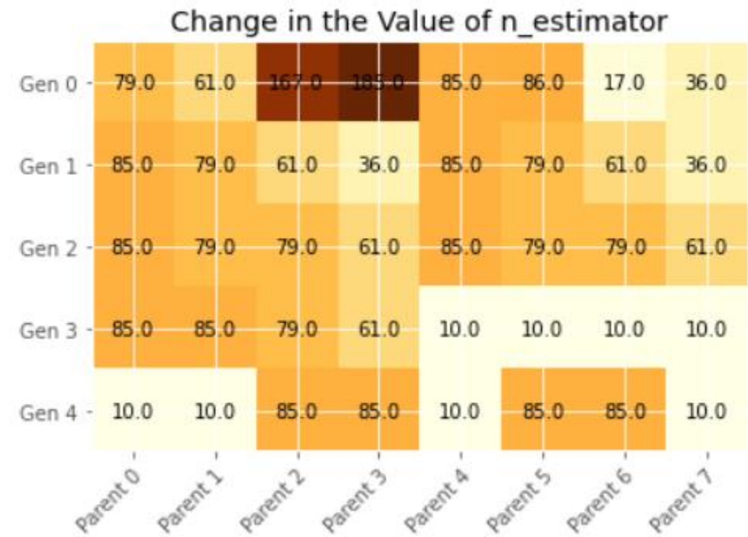

```
<Figure size 1440x720 with 0 Axes>
```

```
# Look at individual parameters change with generation

learnigRateHistory = populationHistory[:, 0].reshape([numberOfGenerations+1, numberOfParents])
nEstimatorHistory = populationHistory[:, 1].reshape([numberOfGenerations+1, numberOfParents])
maxdepthHistory = populationHistory[:, 2].reshape([numberOfGenerations+1, numberOfParents])
minChildWeightHistory = populationHistory[:, 3].reshape([numberOfGenerations+1, numberOfParents])
gammaHistory = populationHistory[:, 4].reshape([numberOfGenerations+1, numberOfParents])
subsampleHistory = populationHistory[:, 5].reshape([numberOfGenerations+1, numberOfParents])
colsampleByTreeHistory = populationHistory[:, 6].reshape([numberOfGenerations+1, numberOfParents])

# Generate Heatmap for each parameter

plot_parameters(numberOfGenerations, numberOfParents, learnigRateHistory, "learning rate")
```



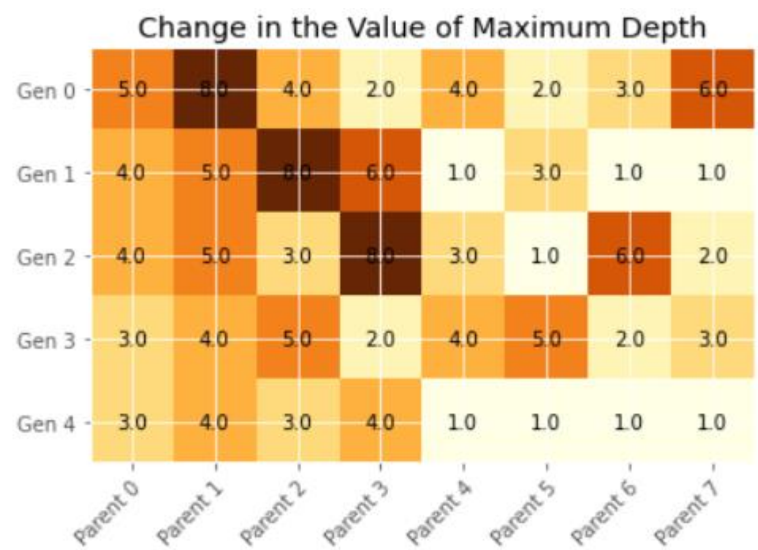

```
<Figure size 1440x720 with 0 Axes>
```

```
plot_parameters(numberOfGenerations, numberOfParents, nEstimatorHistory, "n_estimator")
```



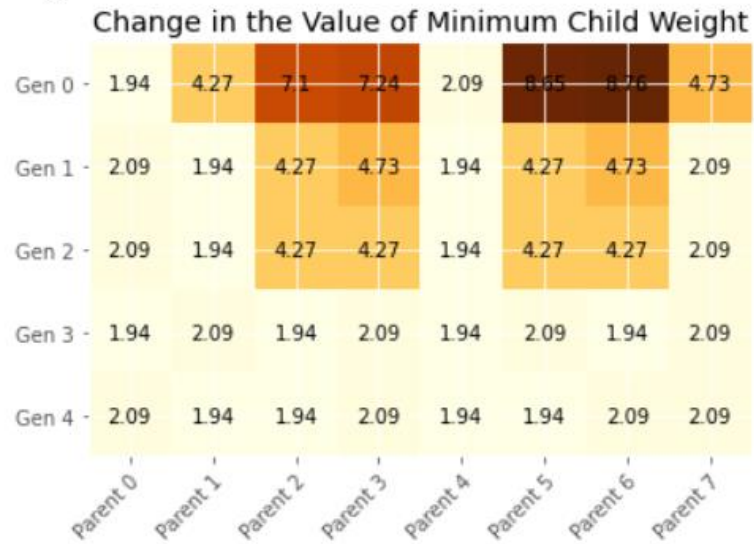

```
<Figure size 1440x720 with 0 Axes>
```

```python
plot_parameters(numberOfGenerations, numberOfParents, maxdepthHistory, "Maximum D
epth")
```



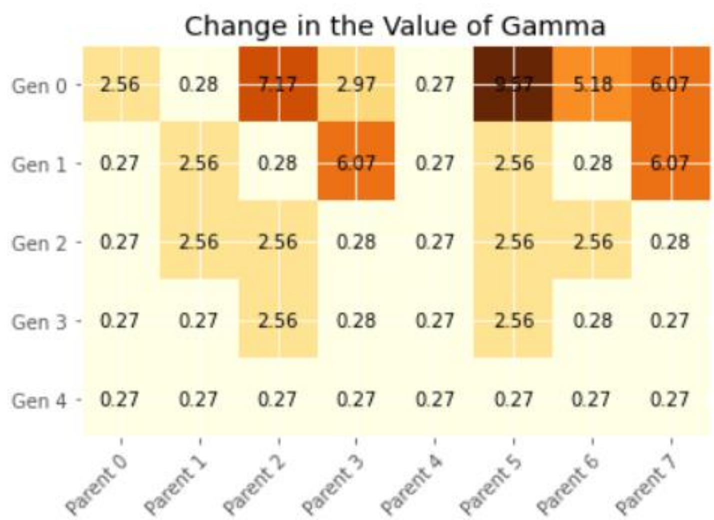

```
<Figure size 1440x720 with 0 Axes>
```

```python
plot_parameters(numberOfGenerations, numberOfParents, minChildWeightHistory, "Min
imum Child Weight")
```



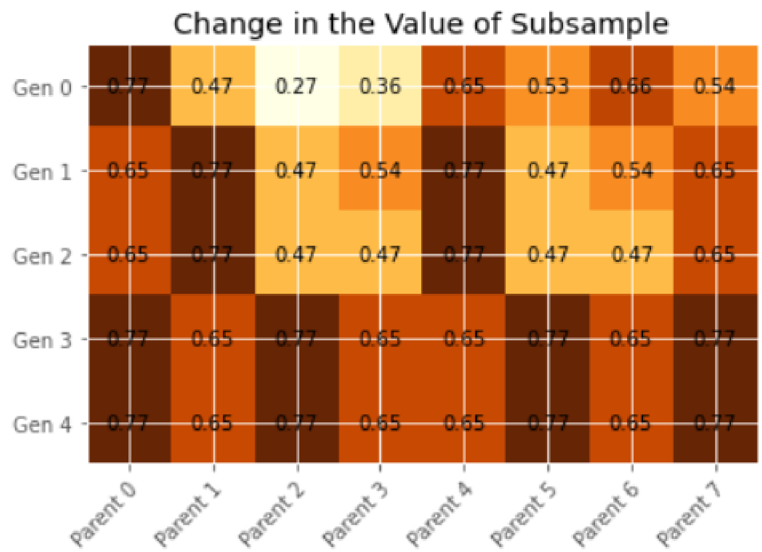

```
<Figure size 1440x720 with 0 Axes>
```

```
plot_parameters(numberOfGenerations, numberOfParents, gammaHistory, "Gamma")
```



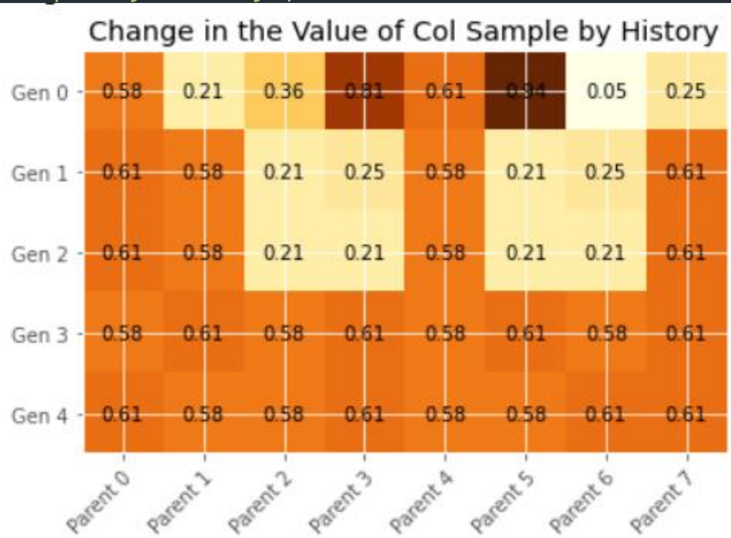

```
<Figure size 1440x720 with 0 Axes>
```

```
plot_parameters(numberOfGenerations, numberOfParents, subsampleHistory, "Subsampl
e")
```




```
<Figure size 1440x720 with 0 Axes>
```

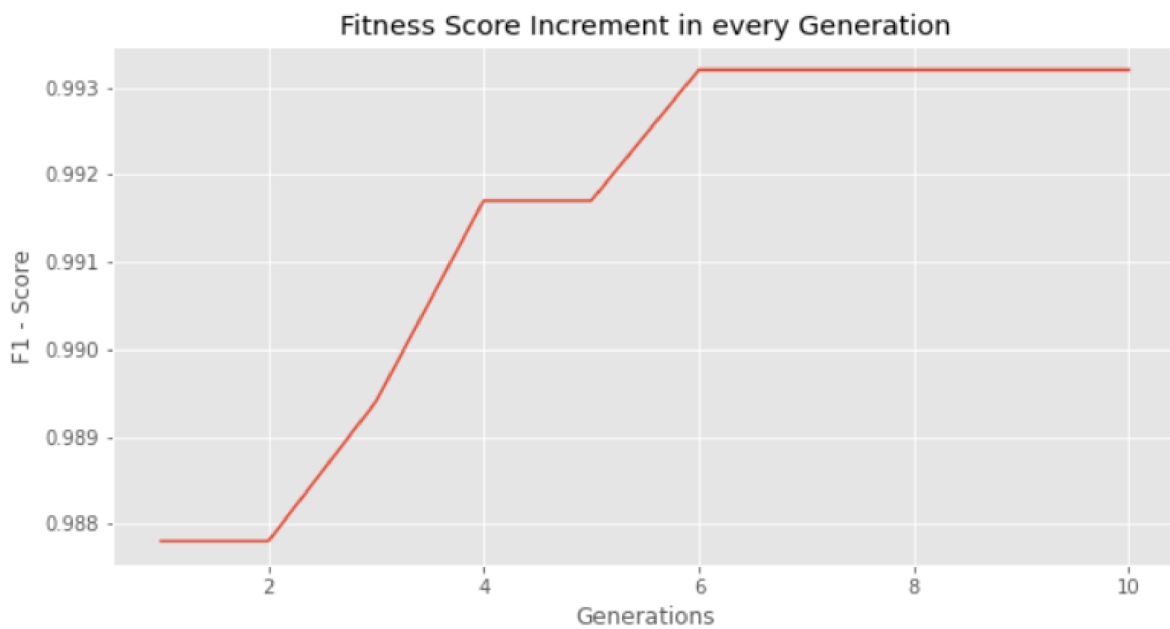
```
plot_parameters(numberOfGenerations, numberOfParents, colsampleByTreeHistory, "Col Sample by History")
```




```
<Figure size 1440x720 with 0 Axes>
```

```
y = best_f1_score_list
x = [1,2,3,4,5,6,7,8,9,10]

# plotting the points
plt.figure(figsize=(10, 5))
plt.plot(x, y)

# naming the x axis
```

```
plt.xlabel('Generations')
# naming the y axis
plt.ylabel('F1 - Score')

# giving a title to my graph
plt.title('Fitness Score Increment in every Generation')

# function to show the plot
plt.show()
```



Conclusion:

While we already started with high F1-score (~0.98), in two of the parents, in the randomly generated initial population, we were able to improve it further in the final generation. The lowest F1-score was 0.9143 for one parent in the initial population and the best score was 0.9947 for one of the parents in the final generation. This demonstrate that we can improve the performance metric in XGBoost, by simple implementation of genetic algorithm.