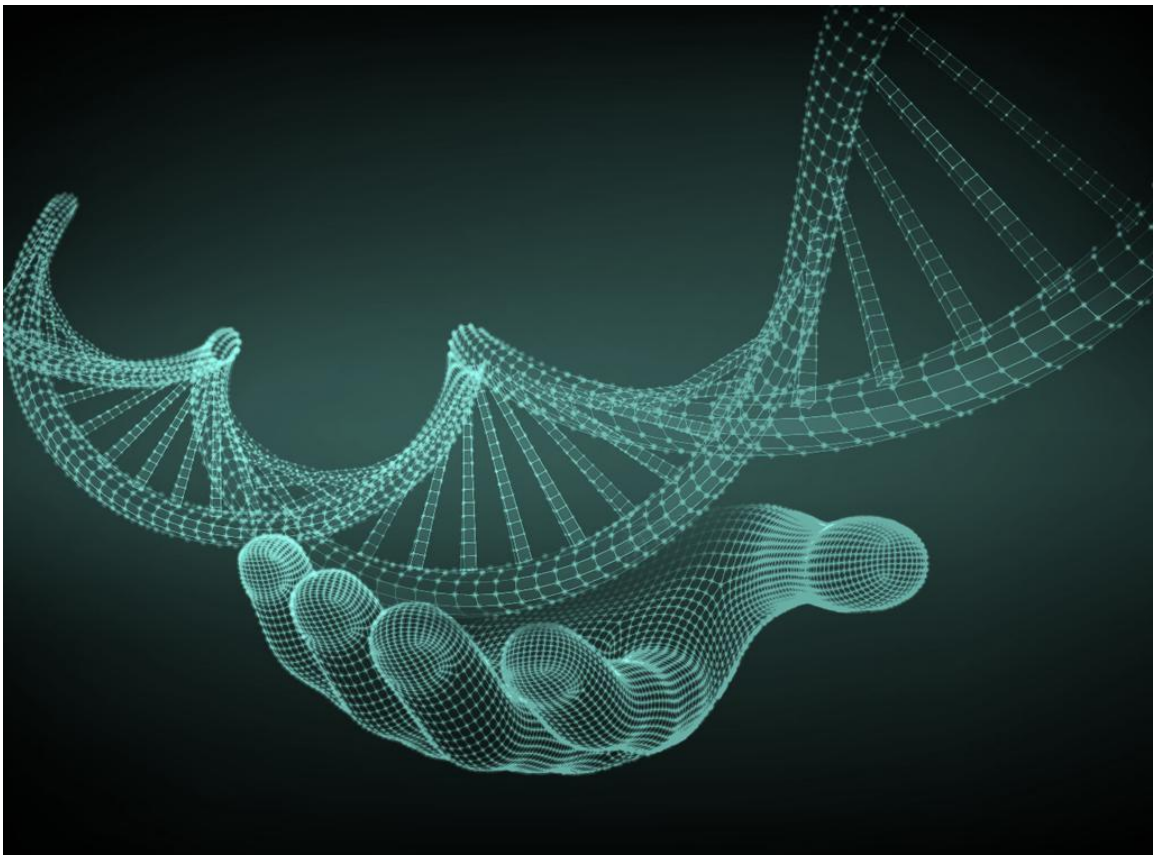


A Comparative Study

COMPARISON BETWEEN CONVENTIONAL CLASSIFIERS AND GENETIC ALGORITHMS BASED MULTI-CLASS CLASSIFIER

Keerti P. Charantimath | 19MA20059

Genetic Algorithms in Engineering Process Modelling

15-04-2021

Index

Abstract

This project aims to use an evolutionary algorithm-based approach to build a classifier that includes major steps like random population creation, calculation of fitness values, selection, crossover, and mutation to build a classifier for the given dataset after its preprocessing. Other methods like descent-based mathematical algorithms and decision tree-based methods were also applied to the same preprocessed dataset and a comparative study was done on these approaches. An attempt was also made to select features for the conventional methods of classification using the Genetic Algorithm methodology.

Problem Statement

The problem statement can be into three major parts as follows:

- Build classifiers to classify songs according to their popularity as “very low”, “low”, “average”, “high” and “very high”.

- Enhance conventional classification algorithms using Genetic Algorithms for feature selection.
- Compare the accuracies and time of convergence of various methods used.

Related works

- A Similar work conducted in Stanford University cited in a paper by the name of 'Predicting Song Popularity' used several different feature selection algorithms, and were able to identify the most influential features in their dataset by taking the intersection among the feature selection algorithms, namely artist familiarity, loudness, year, and a number of genre tags.
- Precision, recall, and F1 score were used by the authors to capture how well their model did in the task of classification. Precision measured the portion of examples that were classified as popular that were truly popular while recall measured the portion of examples that were truly popular that their model classified as popular.

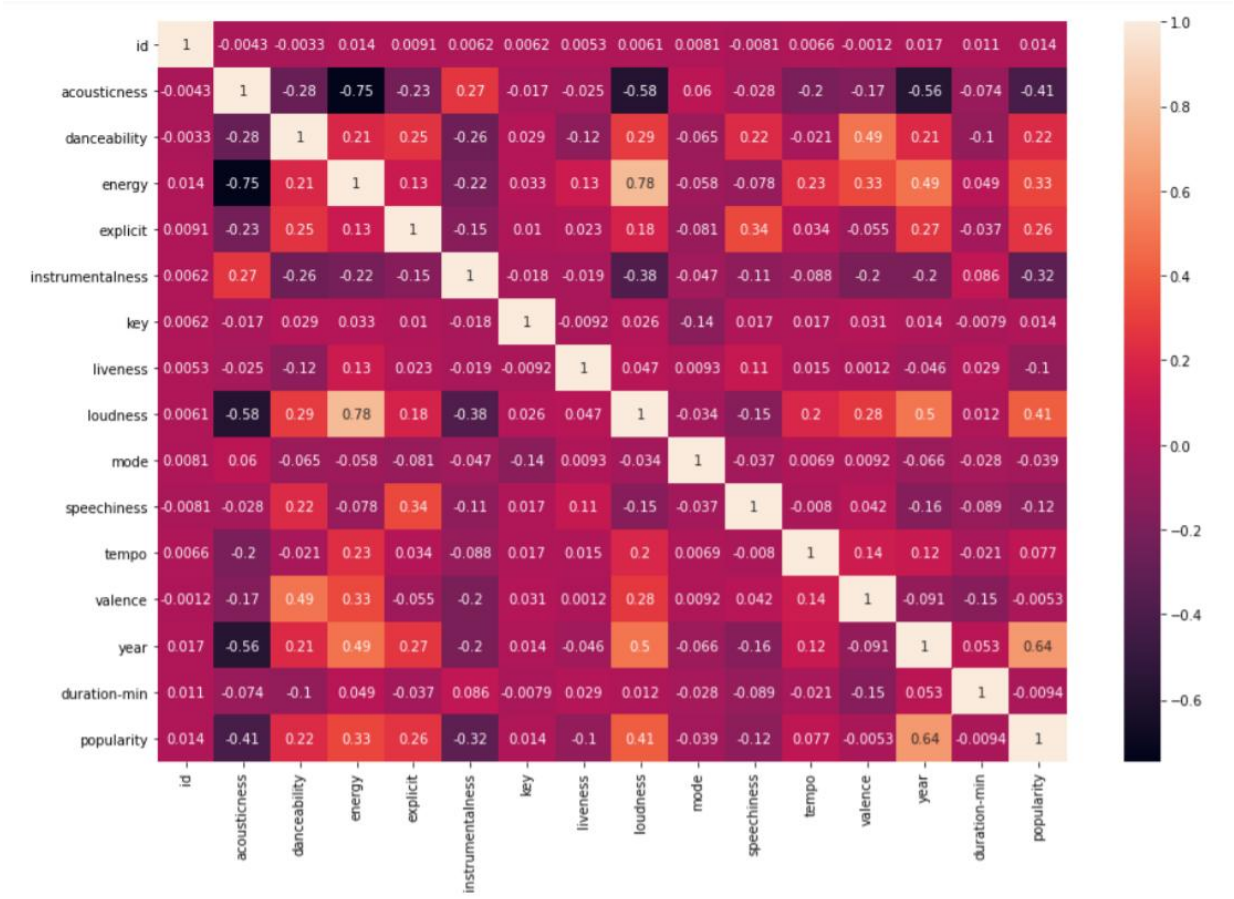
Dataset Summary

- The dataset contained 14 features + Id.
- The target response "popularity" is of categorical type with five categories viz. "very low", "low", "average", "high" and "very high".
- There were 12227 instances of data which was further split into testing and training data.
- There are twelve quantitative features which are "acousticness", "danceability", "energy", "instrumentalness", "liveness", "loudness", "key", "release_date", "speechiness", "tempo", "valence", "year" and "duration-min".
- There are only two qualitative features viz. "explicit" and "mode".
- There was no data missing in the dataset.

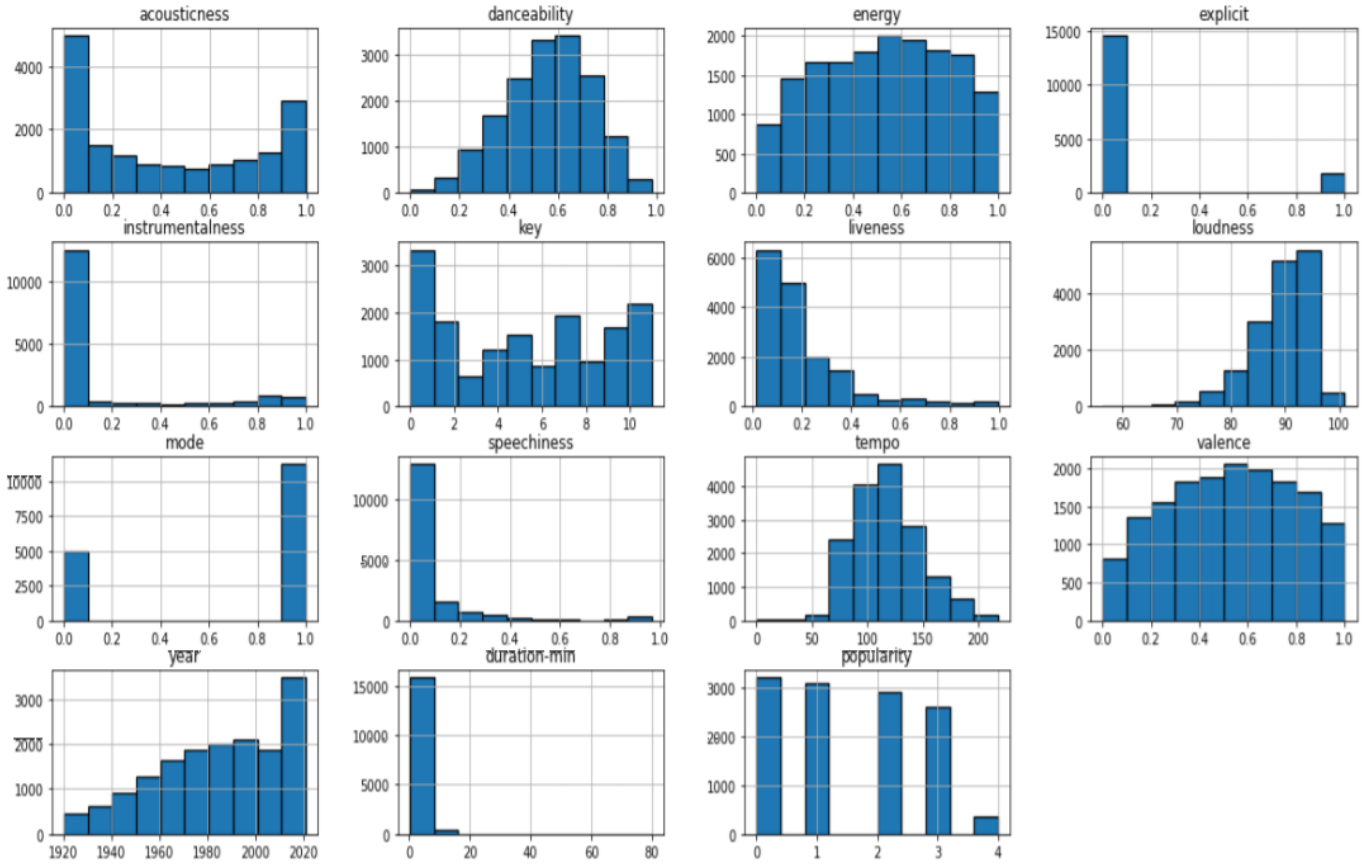
QUALITATIVE DESCRIPTION OF THE DATASET

	acousticness	danceability	energy	instrumentalness	key	liveness	loudness	speechiness	tempo	valence	year	duration-min
count	12227	12227	12227	12227	12227	12227	12227	12227	12227	12227	12227	12227
mean	0.430578	0.556353	0.522129	0.149321	5.205202	0.201365	-10.6686	0.09768	118.167495	0.5253	1984.517298	3.888133
std	0.366893	0.175373	0.262482	0.297954	3.526954	0.173987	5.506888	0.155895	30.200064	0.258205	25.911998	2.383133
min	0.000001	0	0.00002	0	0	0.0147	-43.738	0	0	0	1920	0.2
25%	0.05895	0.438	0.303	0	2	0.0962	-13.656	0.0347	95.0505	0.321	1966	2.9
50%	0.354	0.569	0.534	0.000115	5	0.132	-9.584	0.0456	116.915	0.532	1987	3.6
75%	0.805	0.685	0.739	0.05565	8	0.252	-6.5715	0.0789	136.1085	0.737	2008	4.4
max	0.996	0.98	1	1	11	0.997	1.006	0.968	216.843	1	2021	72.8

HEAT MAP



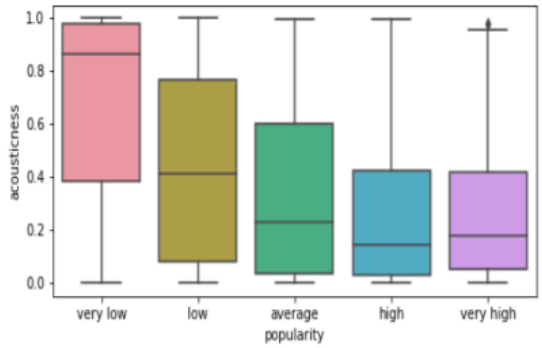
HISTOGRAM PLOTS

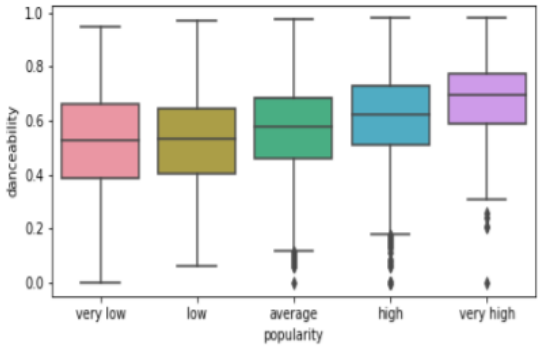


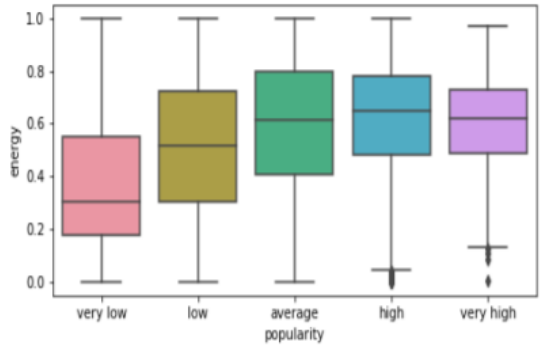
From the distribution of the features it can be observed that the distributions are:

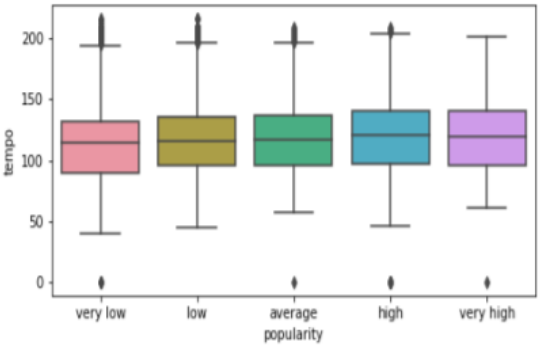
- **acousticness:** inverse normal
- **danceability:** normal
- **energy:** uniform
- **liveness:** lognormal
- **loudness:** skewed normal
- **valence:** uniform
- **popularity:** imbalanced dataset

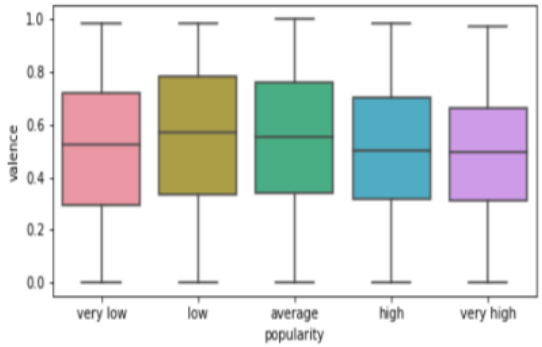
BOX PLOTS

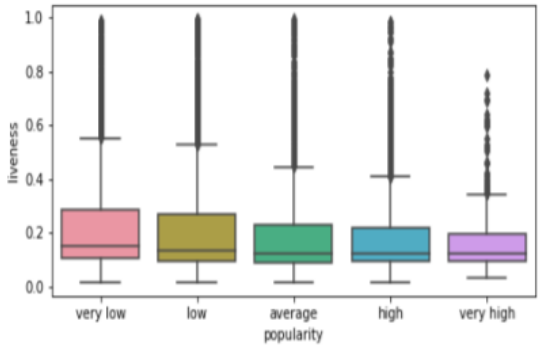












Dataset Preprocessing

MISSING DATA AND DATA CLEANING

- No value in the dataset was found missing.
- In the feature “duration-min”, there was an outlier of 72 minutes which was making the dataset highly skewed. Limiting “duration-min” to 5 minutes increased the correlation of this feature with popularity by 0.14.

ENCODING

- The features “popularity”, “explicit” and “mode” were assigned labels using Label encoding.
- Target “popularity” was encoded as-
  average: 0, high: 1, low: 2, very high: 3, very low: 4
- label_encoder from the SciKit Learn package of Python was used to fit the label encoder and return the encoded labels.
- The feature “key” was converted to key1, key2, etc. using OneHotEncoder of the SciKit Learn package of Python.

DATA TRANSFORMATION

- The feature “duration-min”, and loudness have been log transformed.
- Applied quantile transformation on variables that were highly skewed and far from normal distribution. But this didn't seem to help improve the accuracy of the model much.
- Applied Power Transform to “tempo”, “liveness” and “danceability” to reduce the skewness.
- Oversampling of “very high popularity” data was done to address the imbalance in the dataset using sklearn.utils.resample.
- Normalized the features to get their means to be approximately “0” and their standard deviation to be around “1”. This increased the accuracy of our model by about 3%

DROPPED FEATURES

- The feature “release_date” was dropped as most of the dates included 1st of january and year was already a feature present in the dataset.
- The features “valence”, “tempo”, “liveness” and “key” were dropped initially as their covariance was less when compared with popularity. But this did not seem to improve the model hence these features were used.

NORMALIZATION

- Standard scaling was used to normalize the data which increased the accuracy of the models by ~2%.
- The mean and variance of all the features were brought to be about '0' and '1' respectively.

```
scaler = preprocessing.StandardScaler().fit(X)
X = scaler.transform(X)
```

Code to normalize the dataset using StandardScalar() by SciKitLearn

DATA IMBALANCE

- Approximate percentages of data containing different "popularity" labels are as follows:
  average: 24.3%, high: 21.9%, low: 26.2%, very high: 3.0%, very low: 24.5%
- Imbalanced dataset was fixed using SMOTE (Synthetic Minority Over-sampling Technique), which creates synthetic datapoints for minority class "very high".

ADDITIONAL FEATURES

- Added a feature called "Feature1" defined by loudness × energy x -100.
- Added another feature called "Feature2" defined by speechiness + explicit.
- We have a total of 16 features after preprocessing of the dataset

```
[ ] df['Feature1'] = pd.Series(np.random.randn(len(df['id'])), index=df.index)
    df['Feature1']=df['loudness']*df['energy']*(-100)

[ ] df['Feature2'] = pd.Series(np.random.randn(len(df['id'])), index=df.index)
    df['Feature2']=df['speechiness']+df['explicit']
```

Code to exhibiting creating features "Feature1" and "Feature2"

Part I
Building Classifiers

LOGISTIC REGRESSION

Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable.

Used two approaches to classify data using Logistic Regression. They are:

- **LogisticRegression () by SciKit Learn**

  HYPERPARAMETERS
  1. Solver: Limited-memory Broyden–Fletcher–Goldfarb–Shanno (lbfgs)
  [lbfgs approximates the second derivative matrix updates with gradient evaluations for multiclass classification]

  2.Maximum iterations: 2000

  (other hyperparameters had default values)

```
#training a logistics regression model
logmodel = LogisticRegression(solver='lbfgs', max_iter=2000)
logmodel.fit(X_train,y_train)
predictions = logmodel.predict(X_test)
```

Code to execute logistic regression using SciKitLearn

- **Made from scratch Logistic Regression**

  HYPERPARAMETERS
  1. Learning Rate: 0.1
  [Size of the step the function will take during optimization]

  2.Maximum iterations: 2000

FUNCTIONS

```
def linearPredict(featureMat, weights):
    len_=len(featureMat)
    logitScores = rand(len_,5) # creating empty(garbage value) array for each feature set
    for i in range(featureMat.shape[0]): # iterating through each feature set
      logitScores[i] = (weights.dot(featureMat[i].reshape(-1,1)) ).reshape(-1)
                #calculates logit score for each feature set then flattens the logit vector
    return logitScores
```

- This is the linear predictor function that gives the logit score for each possible outcome.
- Input-
  featureMat- A numpy array of features
  weights- A numpy array of weights for our model
  biases- A numpy array of biases for our model
- Returns-
  logitScores-
  logit scores for each possible outcome of the target variable for each feature set in the feature matrix

```
def softmaxNormalizer(logitMatrix):
    len_=len(logitMatrix)
    probabilities = rand(len_,5)
    for i in range(logitMatrix.shape[0]):
        exp = np.exp(logitMatrix[i]+0.005) # exponentiates each element of the logit array
        sumOfArr = np.sum(exp) # adds up all the values in the exponentiated array
        probabilities[i] = exp/(sumOfArr+0.005) # logit scores to probability values
    return probabilities
```

- This is the **softmaxNormalizer** that converts logit scores for each possible outcome to probability values.
- Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector.

- Input-
  logitMatrix -This is the output of our logitPredict function; consists of logit scores for each feature set
- Returns-
  probabilities - Probability value of each outcome for each feature set

```
def multinomialLogReg(features, weights):
    logitScores = linearPredict(features, weights)
    probabilities = softmaxNormalizer(logitScores)
    predictions = np.array([np.argmax(i) for i in probabilities]) #returns the outcome with m
ax probability
    return probabilities, predictions
```

- Performs logistic regression on a given feature set.
- Input-
  features- Numpy array of features(standardized)
  weights- A numpy array of weights for our model
  biases- A numpy array of biases for our model
- Returns-
  probabilities- Probability values for each possible outcome for each feature set in the feature matrix
  predictions- Outcome with max probability for each feature set

```
def accuracy(predictions, target):
    correctPred = 0
    for i in range(len(predictions)):
        if predictions[i] == target[i]:
            correctPred += 1
    accuracy = correctPred/len(predictions)*100
    return accuracy
```

- Calculates total accuracy for the model
- Input-
  predictions- Predicted target outcomes
  target- Actual target values

- Returns-
  accuracy- Accuracy percentage of the model

```
def crossEntropyLoss(probabilities, target):
    n_samples = len(target)
    CELoss = 0
    for sample, i in zip(probabilities, target):
        CELoss += -np.log(sample[i]+0.05)
        CELoss /= (n_samples+0.05)
    return CELoss
```

- Calculates cross-entropy loss for a set of predictions and actual targets.
- Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1.
- Input-
  predictions- Probability predictions, as returned by multinomialLogReg function
  target- Actual target values
- Returns-
  CELoss- Average cross-entropy loss

```
def stochGradDes(learning_rate, epochs, target, features, weights, biases):
    target = target.astype(int)
    loss_list = np.array([]) #initiating an empty array

    for i in range(epochs):
        probabilities, _ = multinomialLogReg(features, weights, biases) # Calculates
                                                probabilities for each possible outcome
        CELoss = crossEntropyLoss(probabilities, target) # Calculates cross entropy loss for
                                                actual target and predictions
        loss_list = np.append(loss_list, CELoss) # Adds the CELoss value for the epoch to
                                                loss_list
        probabilities[np.arange(features.shape[0]),target] = 1 # Substract 1 from the scores
                                                of the correct outcome
        grad_weight = probabilities.T.dot(features) # gradient of loss w.r.t. weights
```

```
    grad_biases = np.sum(probabilities, axis = 0).reshape(-1,1)
                                                    # gradient of loss w.r.t. biases
    #updating weights and biases
    weights -= (learning_rate * grad_weight)
    biases -= (learning_rate * grad_biases)

return weights, biases, loss_list
```

- Performs **stochastic gradient descent optimization** on the model.
- Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or subdifferentiable).
- Input-
  learning_rate- Size of the step the function will take during optimization
  epochs- No. of iterations the function will run for on the model
  target- Numpy array containing actual target values
  features- Numpy array of independent variables
  weights- Numpy array containing weights associated with each feature
  biases- Array containing model biases
- Returns-
  weights- Latest weight calculated (Numpy array)
  bias- Latest bias calculated (Numpy array)
  loss_list- Array containing list of losses observed after each epoch

RUNNING THE BUILT MODEL

```
updatedWeights, updatedBiases, loss_list = stochGradDes(0.1, 2000, target, features, weights, biases)
testProbabilities, testPredictions = multinomialLogReg(test_features, updatedWeights, updatedBiases)
```

Code to execute multi-class classification using the built model

RANDOM FOREST CLASSIFIER

Random forests or random decision forests are an ensemble learning method for classification, regression, and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees.

Used RandomForestClassifier of ScikitLearn package to classify the data.

HYPERPARAMETERS:

- bootstrap: True
- criterion: gini
- min_weight_fraction_leaf=0.2
- max_features: auto

Other hyper parameters had default value

```
[ ]  from sklearn.ensemble import RandomForestClassifier

[ ]  clf = RandomForestClassifier(bootstrap: True,criterion: gini,min_weight_fraction_leaf=0.2,max_features: auto)
     clf.fit(X_train, y_train)
```

Code to run random forest classifier using SciKitLearn

GENETIC CLASSIFIER

The following approaches were used to build a Genetic Classifier:

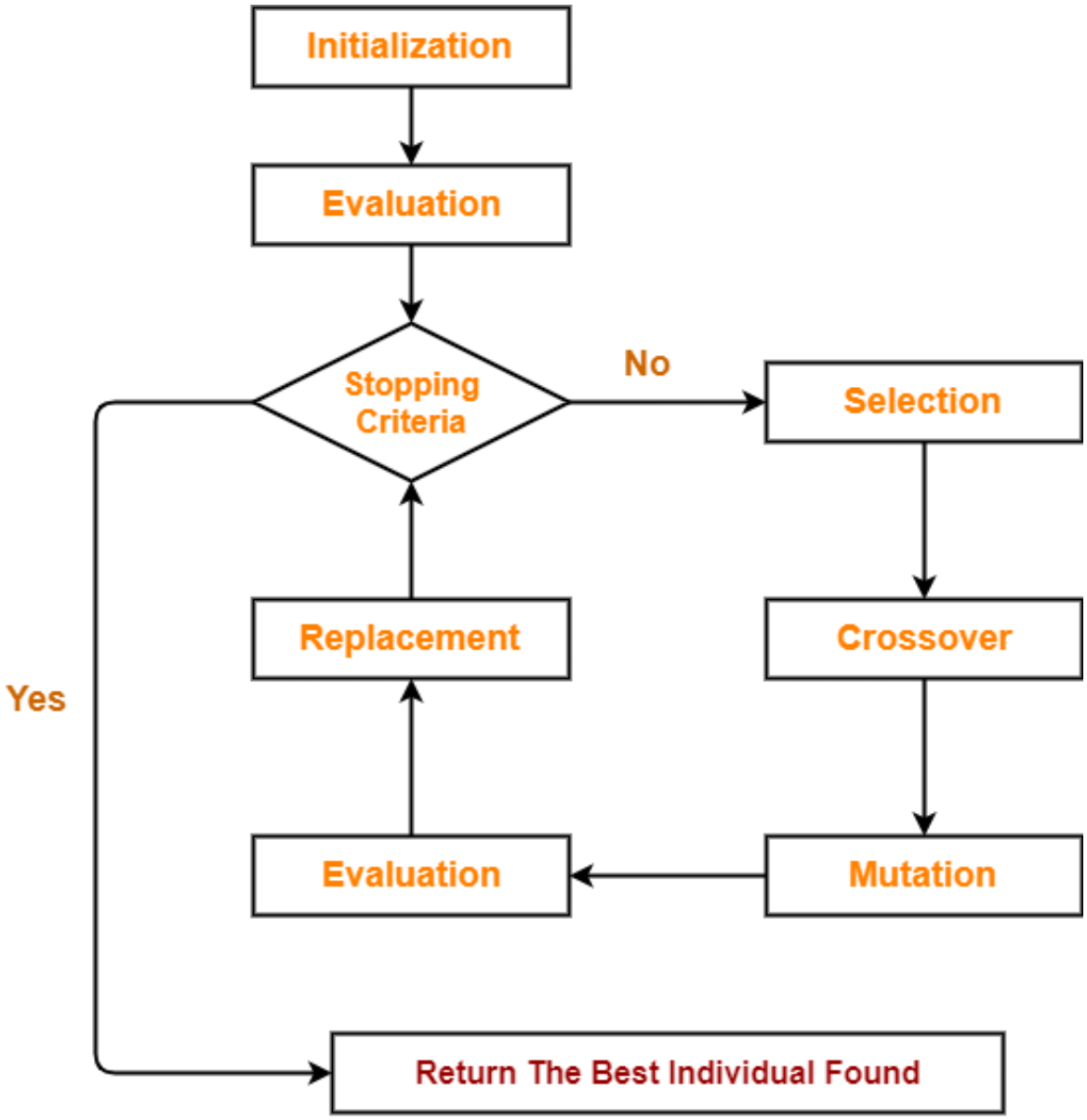
- Built from scratch genetic classifier.

Using python library **PyGAD**.

Built from scratch Genetic Classifier

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve.

The following algorithm was used in this project:



FUNCTIONS

```
#initialize the population
pop=list()
for i in range(n_pop):
    biases, weights = rand(5,1), rand(5, 16)
    updatedWeights = weights.reshape(1, 80)
    updatedBiases = biases.reshape(1, 5)
    individual = np.concatenate((updatedBiases, updatedWeights),axis=1)
    pop.append(individual)
```

Initialization is the first step in the Genetic Algorithm Process. Population P is defined as a set of chromosomes. The initial population P(0), which is the first generation is created randomly using the numpy.random.rand() function.
The output was a numpy array of desired size containing random float values between 0 and 1.

The **population size** used was **100.** The stopping criteria was running the program for 250 **generations.**

```
# objective/fitness function
def objective(vari):
  biases=vari[0][0:5]
  biases = biases.reshape(5, 1)
  weights=vari[0][5:85]
  weights = weights.reshape(5, 16)
  probabilities, predictions = multinomialLogReg(features, weights, biases)
  acc = accuracy(predictions, target)
  return(-acc)
```

Evaluation is the step where the fitness of each individual is calculated. For the given problem, accuracy of the individual set of weights is used. As this is a **minimization problem,** (**-accuracy**) is used as the fitness score.

```
# tournament selection of k=3
def selection(pop, scores, k=3):
  selection_ix = randint(len(pop))      # first random selection
  for ix in randint(0, len(pop), k-1):
    if scores[ix] < scores[selection_ix]: # check if better (e.g. perform a tournament)
      selection_ix = ix
  return pop[selection_ix]
```

Selection is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding according to the fitness values.

In tournament selection with tournament size k, we select k-individuals and run a tournament among them. Only the fittest candidate amongst those selected candidates is chosen and is passed on to the next generation.

```
# Uniform crossover two parents to create two children
def crossover(p1, p2, r_cross):
  c1=p1 # children are copies of parents by default
  c2=p2
  for i in range(len(p1[0])):
    if rand() < r_cross:       # check for a crossover
      c1[0][i]=p2[0][i]
      c2[0][i]=p1[0][i]
  return [c1, c2]
```

Crossover, also called recombination, is a genetic operator used to combine the genetic information of two parents to generate new offspring.

In **uniform crossover**, each value is chosen from either parent with equal probability. Crossover probability of **0.5** is used in the project.

```
# uniform/random mutation operator
def mutation(bitstring, r_mut):
  for i in range(len(bitstring[0])):
    if rand() < r_mut:       # check for a mutation
      bitstring[0][i] += random.uniform(-0.2,0.2) #real mutation
```

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. Mutation alters one or more gene values in a chromosome from its initial state.

Uniform/random mutation replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene. Mutation probability of **0.1** is used in the project.

PyGAD Classifier

PyGAD is an open-source Python library for building the genetic algorithm and optimizing machine learning algorithms.

PyGAD supports different types of crossover, mutation, and parent selection operators. PyGAD allows different types of problems to be optimized using the genetic algorithm by customizing the fitness function.

FUNCTIONS

```
def fitness_func1(solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_idx],
                                                                    data_inputs=data_inputs)
    ans=f1_score(data_outputs,predictions,average="weighted")
    solution_fitness=np.sum(ans)
    return solution_fitness
```

This function calculates the fitness for each solution set. The fitness score used in this case is the **weighted F1 score which is almost equal to the accuracy of the model.** This is a **maximization function**.

```
def callback_generation(ga_instance):
    global GANN_instance

    population_matrices = pygad.gann.population_as_matrices(population_networks=
                    GANN_instance.population_networks, population_vectors=ga_instance.population)

    GANN_instance.update_population_trained_weights(population_trained_weights=
                                                                        population_matrices)

    print("Generation = {generation}".format(generation=ga_instance.generations_completed))
    print("Accuracy   = {fitness}".format(fitness=ga_instance.best_solution()[1]))
```

This is the function for updating the population of weights after each generation/iteration.

```
GANN_instance = pygad.gann.GANN(num_solutions=6, num_neurons_input=16,
                                num_neurons_output=5,
                                num_neurons_hidden_layers=[],
                                output_activation="softmax")
population_vectors = pygad.gann.population_as_vectors(population_networks=
                                                       GANN_instance.population_networks)
ga_instance = pygad.GA(num_generations=1000, num_parents_mating=2,
                       initial_population=population_vectors.copy(),
                       fitness_func=fitness_func1,
                       crossover_probability=0.5,
                       mutation_probability=0.9,
                       mutation_type="random",
                       crossover_type="uniform",
                       on_generation=callback_generation,
                       parent_selection_type="tournament",
                       K_tournament=3, keep_parents=1,
                       allow_duplicate_genes=False)
```

This function initializes all the population vectors which are sets of random weights for the classifier. It also sets all the parameters for the genetic algorithm as well as the classifier which are as follows:

PARAMETERS FOR NEURAL NETWORK

Input features=16,
Hidden layers=0,
Number of outputs=5,
Output Activation=”softmax”

PARAMETERS FOR GENETIC ALGORITHM

Population size=6,
Number of generations=1000,
Number of parent mating=2,
Mutation type="random" with probability 0.9 (as the population size was less, high mutation probability was required for convergence),
Crossover type="uniform" with probability 0.5(as the population size was less, high crossover probability would result in destruction of good individuals),
Selection algorithm="tournament selection" with tournament size 3,
Number of parents to keep after mating=1,
Allow duplicate genes?=No

Part II
Enhancing Classifiers using Genetic Algorithms

FEATURE SELECTION

Feature Selection is the process where you automatically or manually select those features that contribute most to your prediction variable or output in which you are interested. Having irrelevant features in data can decrease the accuracy of the models and make your model learn based on irrelevant features.

The algorithm used for the genetic algorithm classifier was used for the genetic algorithm feature selector.

FUNCTIONS (for logistic regressor feature selection)

```
def initialization_of_population(size,n_feat):
    population = []
    for i in range(size):
        chromosome = np.ones(n_feat,dtype=np.bool)
        chromosome[:int(0.3*n_feat)]=False
        np.random.shuffle(chromosome)
        population.append(chromosome)
    return population
```

A random binary population of **size 200** was initialized. The code was run for **38 generations.**

```
def fitness_score(population):
    scores = []
    for chromosome in population:
        logmodel.fit(X_train.iloc[:,chromosome],y_train)
        predictions = logmodel.predict(X_train.iloc[:,chromosome])
        ans=f1_score(y_train,predictions,average="weighted")
        solution_fitness=np.sum(ans)
        scores.append(solution_fitness)
    scores, population = np.array(scores), np.array(population)
    inds = np.argsort(scores)
    return list(scores[inds][::-1]), list(population[inds,:][::-1])
```

The fitness function used was **weighted average of F1 score**. This is a maximization function.

```
def selection(pop_after_fit,n_parents):
    population_nextgen = []
    for i in range(n_parents):
        population_nextgen.append(pop_after_fit[i])
    return population_nextgen
```

The selection function used in this case was simply passing such individuals to the next generation which have the highest accuracy. The number of parents participating in selection was set to **100**

```
def crossover(pop_after_sel,cross_prob):
    population_nextgen=pop_after_sel
    for i in range(len(pop_after_sel)):
        child=pop_after_sel[i]
        if(rand()<=cross_prob)
          a=randint(1,len(pop_after_sel)-2)
          b=randint(a,len(pop_after_sel)-1)
          child[a:b]=pop_after_sel[(i+1)%len(pop_after_sel)][a:b]
        population_nextgen.append(child)
    return population_nextgen
```

The crossover function used in this case was **Two-point crossover** with crossover probability **0.9**.

```
def mutation(pop_after_cross,mutation_rate):
    population_nextgen = []
    for i in range(0,len(pop_after_cross)):
        chromosome = pop_after_cross[i]
        for j in range(len(chromosome)):
            if random.random() < mutation_rate:
                chromosome[j]= not chromosome[j]
        population_nextgen.append(chromosome)
    return population_nextgen
```

Mutation used in this problem was **Random mutation** with mutation probability **0.1.**

FUNCTIONS (for random forest feature selection)

The functions built for logistic regressor feature selection were used for random forest feature selection.

The parameter values used were:

Population size=100,
Number of parents for selection =50,
Mutation rate=0.10,
Crossover rate=0.80,
Number of generations=25

The only function which was different was the fitness function

```
def fitness_score_rf(population):
    scores = []
    for chromosome in population:
        clf.fit(X_train.iloc[:,chromosome],y_train)
        predictions = clf.predict(X_val.iloc[:,chromosome])
        ans=f1_score(y_val,predictions,average="weighted")
        solution_fitness=np.sum(ans)
        scores.append(solution_fitness)
        #scores.append(accuracy_score(y_val,predictions))
    scores, population = np.array(scores), np.array(population)
    inds = np.argsort(scores)
    return list(scores[inds][::-1]), list(population[inds,:][::-1])
```

This fitness function used the **weighted F1 score** of **Validation set** as the training accuracy was 100.

Result

CLASSIFIER ACCURACY

1. LOGISTIC REGRESSOR BY SCIKIT LEARN

```
#train accuracy logistic regression
predictions = logmodel.predict(X_train)
print(classification_report(y_train, predictions))
```

	precision	recall	f1-score	support
0	0.44	0.33	0.38	2030
1	0.50	0.68	0.58	1826
2	0.51	0.52	0.52	2159
3	0.80	0.01	0.03	269
4	0.62	0.65	0.64	2049
accuracy			0.53	8333
macro avg	0.58	0.44	0.43	8333
weighted avg	0.53	0.53	0.51	8333

```
[110] #test_accuracy logistic regression
      print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0	0.46	0.33	0.38	873
1	0.53	0.72	0.61	772
2	0.51	0.54	0.52	939
3	0.00	0.00	0.00	99
4	0.63	0.66	0.65	889
accuracy			0.54	3572
macro avg	0.43	0.45	0.43	3572
weighted avg	0.52	0.54	0.52	3572

Train Accuracy:53%
Test Accuracy: 54%
Time to train: 0 seconds

2. BUILT FROM SCRATCH LOGISTIC REGRESSOR

```
#train_accuracy self built logistic regression
probabilities, predictions = multinomialLogReg(features,weights,biases)
acc = accuracy(predictions, target)
print("Accuracy logistic regression = "acc)))
```

```
Accuracy logistic regression = 0.5251410056402256
```

```
[25] #test_accuracy self built logistic regression
     probabilities, predictions = multinomialLogReg(features_test,weights,biases)
     acc = accuracy(predictions, target_test)
     print("Accuracy logistic regression = "acc)))
```

```
Accuracy logistic regression = 0.5405935050391937
```

Train accuracy: 52.5%
Test accuracy: 54.0%
Time to train: 1 second

3. RANDOM FOREST CLASSIFIER

```
#train accuracy for random forest
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2030
           1       1.00      1.00      1.00      1826
           2       1.00      1.00      1.00      2159
           3       1.00      1.00      1.00       269
           4       1.00      1.00      1.00      2049

    accuracy                           1.00      8333
   macro avg       1.00      1.00      1.00      8333
weighted avg       1.00      1.00      1.00      8333
```

```
#test accuracy for random forest
precision    recall  f1-score   support

           0       0.50      0.45      0.47       427
           1       0.63      0.58      0.60       378
           2       0.56      0.66      0.61       496
           3       0.45      0.21      0.29        42
           4       0.76      0.77      0.77       443

    accuracy                           0.61      1786
   macro avg       0.58      0.53      0.55      1786
weighted avg       0.61      0.61      0.61      1786
```

Train Accuracy:100%
Test Accuracy: 61%
Time to train: 10 seconds

4. BUILT FROM SCRATCH GENETIC CLASSIFIER

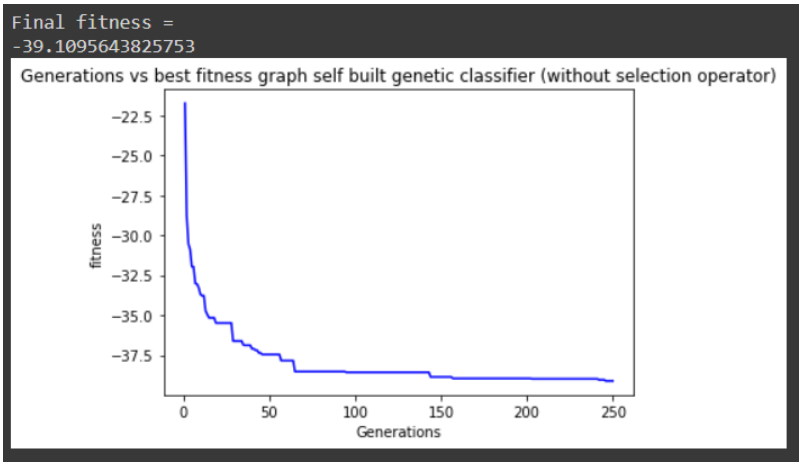
```
[140] #train accuracy of self made logistic regressor
      biases=parameters[0][0:5]
      biases = biases.reshape(5, 1)
      weights=parameters[0][5:85]
      weights = weights.reshape(5, 16)
      probabilities, predictions = multinomialLogReg(features, weights, biases)
      acc = accuracy(predictions, target)
      print("Train accuracy of genetic classifier is:")
      print(acc)

      Train accuracy of genetic classifier is:
      56.506886151446054
```

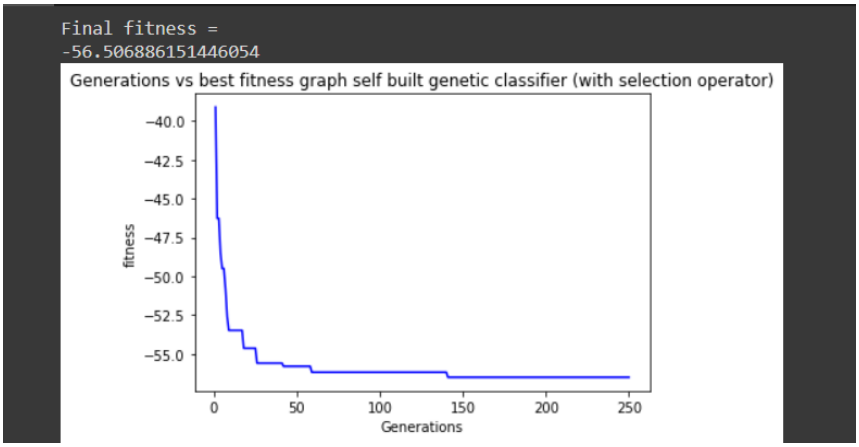
```
[141] #test accuracy of self made logistic regressor
      biases=parameters[0][0:5]
      biases = biases.reshape(5, 1)
      weights=parameters[0][5:85]
      weights = weights.reshape(5, 16)
      probabilities, predictions = multinomialLogReg(features_test, weights, biases)
      acc = accuracy(predictions, target_test)
      print("Test accuracy of genetic classifier is:")
      print(acc)

      Test accuracy of genetic classifier is:
      56.4198557400340479
```

Train accuracy: 56.5%
Test accuracy: 56.4%
Total time to converge: 1 hour 25 minutes 23 seconds



Graph of genetic algorithm run for 250 generations without selection operator
Time to converge: 36 minutes 27 seconds



Graph of genetic algorithm run for 250 generations with selection operator.
(The final population of the previous run was used as initial population of this run)
Time to train: 48 minutes 56 seconds

5. PYGAD CLASSIFIER

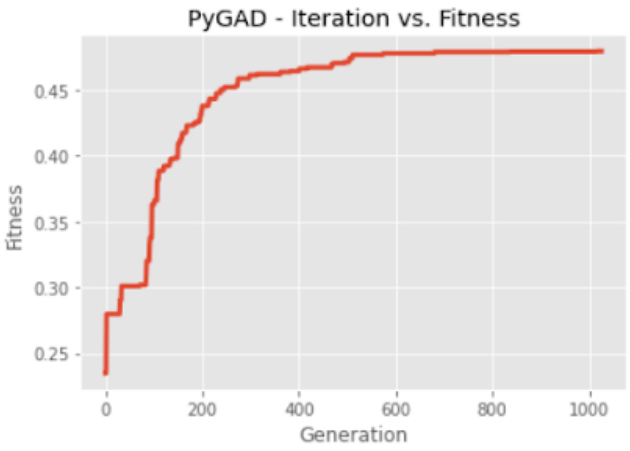
```
[83] #calculate accuracy for test data
     from sklearn.metrics import classification_report, confusion_matrix
     print(classification_report(data_test_y, predictions))
```

	precision	recall	f1-score	support
0	0.31	0.27	0.29	450
1	0.51	0.58	0.54	410
2	0.41	0.43	0.42	458
3	0.04	0.03	0.04	58
4	0.67	0.66	0.66	410
accuracy			0.46	1786
macro avg	0.39	0.39	0.39	1786
weighted avg	0.46	0.46	0.46	1786

```
[80] #calculate accuracy for train data
     from sklearn.metrics import classification_report, confusion_matrix
     print(classification_report(data_outputs, predictions))
```

	precision	recall	f1-score	support
0	0.33	0.28	0.30	2014
1	0.48	0.60	0.54	1820
2	0.45	0.44	0.44	2155
3	0.03	0.03	0.03	254
4	0.71	0.67	0.69	2090
accuracy			0.48	8333
macro avg	0.40	0.41	0.40	8333
weighted avg	0.48	0.48	0.48	8333

Train accuracy: 48%
Test accuracy: 46%
Time to converge: 44 minutes 52 seconds



Final fitness after 1000 iterations: 0.4790

CLASSIFIER ACCURACY AFTER FEATURE SELECTION

1. LOGISTIC REGRESSOR

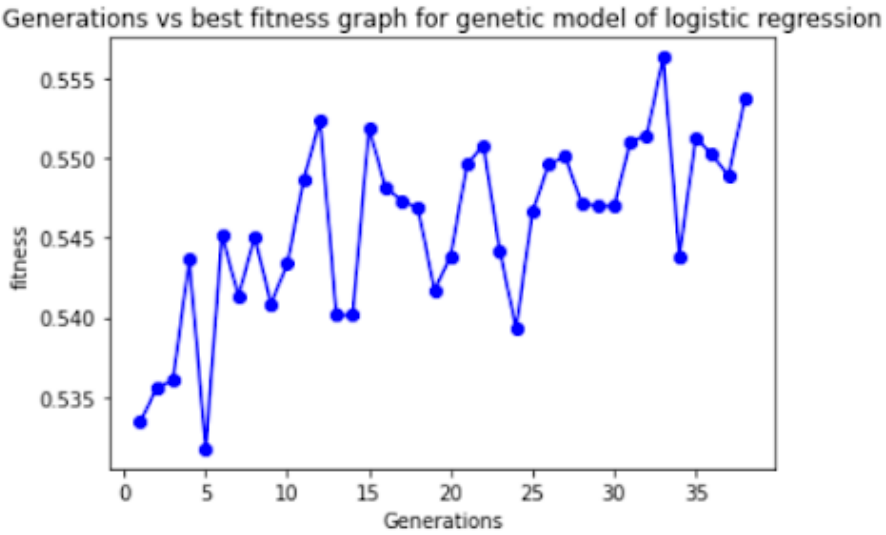
```
#test_accuracy genetic model
print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0	0.49	0.33	0.39	873
1	0.50	0.72	0.59	772
2	0.55	0.54	0.54	939
3	0.00	0.00	0.00	99
4	0.66	0.72	0.68	889
accuracy			0.55	3572
macro avg	0.44	0.46	0.44	3572
weighted avg	0.54	0.55	0.54	3572

```
#train accuracy genetic model
predictions = logmodel.predict(X_train.iloc[:,chromo[-1]])
print(classification_report(y_train, predictions))
```

	precision	recall	f1-score	support
0	0.48	0.33	0.39	2030
1	0.47	0.69	0.56	1826
2	0.56	0.55	0.55	2159
3	0.00	0.00	0.00	269
4	0.68	0.70	0.69	2049
accuracy			0.55	8333
macro avg	0.44	0.45	0.44	8333
weighted avg	0.53	0.55	0.53	8333

Train accuracy: 55% (increased by 2%)
Test accuracy: 55% (increased by 1%)



Final fitness after 38 iterations: 0.5532

Time to converge: 22 minutes 36 seconds

RANDOM FOREST CLASSIFIER

```
#Test set accuracy genetic model of random forest
              precision    recall  f1-score   support

           0       0.52      0.46      0.49       427
           1       0.64      0.58      0.61       378
           2       0.56      0.69      0.62       496
           3       0.50      0.21      0.30        42
           4       0.78      0.78      0.78       443

    accuracy                           0.62      1786
   macro avg       0.60      0.54      0.56      1786
weighted avg       0.62      0.62      0.62      1786
```

```
#train set accuracy genetic model of random forest
precision    recall  f1-score   support

           0       1.00      1.00      1.00      2030
           1       1.00      1.00      1.00      1826
           2       1.00      1.00      1.00      2159
           3       1.00      1.00      1.00       269
           4       1.00      1.00      1.00      2049

    accuracy                           1.00      8333
   macro avg       1.00      1.00      1.00      8333
weighted avg       1.00      1.00      1.00      8333
```

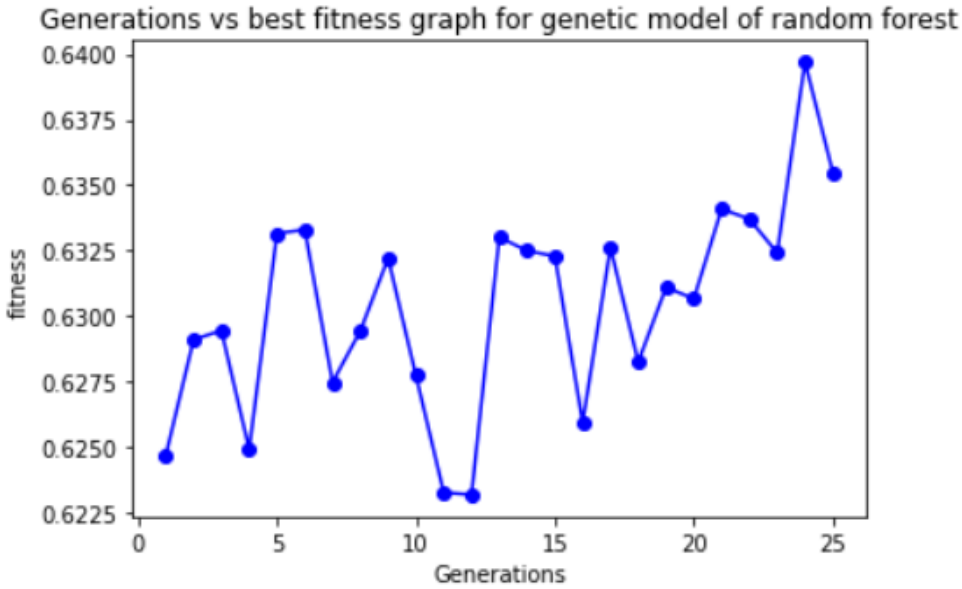
```
#validation set accuracy genetic model of random forest

              precision    recall  f1-score   support

           0       0.49      0.44      0.46       446
           1       0.62      0.60      0.61       394
           2       0.55      0.65      0.60       443
           3       0.60      0.16      0.25        57
           4       0.80      0.83      0.81       446

    accuracy                           0.62      1786
   macro avg       0.61      0.54      0.55      1786
weighted avg       0.63      0.62      0.63      1786
```

Train accuracy: 100%
Test accuracy: 62% (increased by 1%)
Validation set accuracy: 62%



Final fitness after 25 iterations: 0.6351
Time to converge: 1 hour 11 minutes 47 seconds

Observations

- Neural network was applied on the dataset using PyGAD but the convergence time was a lot with about 2 minutes 16 seconds per generation. The neural network had 5 neurons in the hidden layer and was overfitting the data. Hence the accuracy was always around 20% even after 20 generations. Thus, simple logistic regression was used.

- When the feature selection algorithm was run with a population size of 25, there was a lot of variation in the fitness values in consecutive generations. This resulted in a noisy generation vs. fitness graph. Due to less population size, the algorithm was not converging.

- When the Genetic Classifier was run for 250 generations without selection function, the classifier increased its accuracy from 21.8% to 39.1% over 250 generations but it could not converge further.

Conclusion

METHOD	TIME TO CONVERGENCE	TRAIN ACCURACY	TEST ACCURACY
SciKit Logistic Regressor	0 seconds	53%	54%
Built from scratch Logistic Regressor	1 second	52.5%	54%
Random Forest Classifier	10 seconds	100%	61%
Built from scratch Genetic Classifier	1 hour 25 minutes 23 seconds	56.5%	56.4%
PyGAD Genetic Classifier	44 minutes 52 seconds	48%	46%
Logistic Regressor after Feature selection	22 minutes 36 seconds	55%	55%
Random Forest after Feature selection	1 hour 11 minutes 47 seconds	100%	62%

Comparison table of various methods to compare their test accuracy, train accuracy, and their time to convergence

From the observations made through graphs and comparison table, we can conclude that:

- Genetic Algorithm built from scratch performs better than PyGAD Classifier and all the different Logistic Regressors in terms of Test Accuracy.
- Genetic Algorithm is also successful in selecting the right features to enhance the accuracy of Logistic Regressor and Random Forest Classifier.
- The main draw-back of Genetic based codes that we see through this project is the huge time to convergence. Also, the Genetic Classifier fails to surpass the accuracy of Random forest Classifier.
- Further, if we succeed to make these Genetic Algorithms faster, we can use them to enhance the existing classifiers to attain better accuracies.

References

- Elyan, Eyad and Gaber, Mohamed - *"A Genetic Algorithm Approach to Optimising Random Forests Applied to Class Engineered Data"*
- PyGAD documentation: https://pygad.readthedocs.io/en/latest/
- Genetic Algorithm Implementation in Python by Ahmed Gad.