Accident Survival Prediction using GA

February 28, 2025

0.1 k-nearest neighbors on accident survival prediction dataset using genetic algorithms

0.1.1 Introduction

Understanding the elements that contribute to survival in traffic accidents, such as wearing a seatbelt, can be crucial for road safety and the prevention of fatalities. Investigating these aspects and potentially predicting survival outcomes based on a range of parameters, such as age, gender, impact speed, and safety measures like wearing a seatbelt and a helmet, are made possible by this dataset. In this research, we will examine whether applying genetic algorithms to enhance our machine learning model's performance is effective.

[1]:
```
## Image Classification with KNNs

# 1. Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics,  preprocessing, model_selection
import time
start_time = time.time()

import keras
from pandas import DataFrame
from sklearn.preprocessing import MinMaxScaler,minmax_scale,LabelEncoder
from keras.preprocessing import sequence
keras.utils.np_utils.to_categorical
from sklearn.metrics import␣
 ↪confusion_matrix,classification_report,accuracy_score,f1_score
from hyperopt import hp,fmin,tpe,STATUS_OK,Trials
from sklearn.model_selection import cross_val_score


import random, statistics
from deap import base, creator, tools, algorithms
from statistics import mean,mode,stdev,median
```

```
import gc
gc.enable()
gc.collect()

pd.options.display.max_columns = 100
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
pd.set_option("display.max_columns", None)
```

The dataset from the accident survivor prediction challenge will be used. Given the features in the dataset, we must predict if a person survived an accident. This dataset relates to a classification problem. In order to continue, we must first load the dataset. However, the dependencies ought to have been loaded first. These dependencies will be necessary. Now that the dependencies have been imported, let's use the Pandas library to load the dataset into a Dataframe object.

[2]:
```
# 2. Load and Preprocess Dataset
path = pd.read_csv('./accident.csv')
print(path.head())
print(path.shape)
plt.show()
```

```
   Age  Gender  Speed_of_Impact Helmet_Used Seatbelt_Used  Survived
0   56  Female             27.0          No            No         1
1   69  Female             46.0          No           Yes         1
2   46    Male             46.0         Yes           Yes         0
3   32    Male            117.0          No           Yes         0
4   60  Female             40.0         Yes           Yes         0
(200, 6)
```

Four different features labeled into the outcomes of 1 and 0, where 1 indicates survival and 0 indicates non-survival. Finding any null values will be the next stage. When attempting to create a machine learning model, many classifiers, such as logistic regression, are unable to handle null data.

To find out these details, we should obtain information about our data set.

[3]:
```
path.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Age              200 non-null    int64
 1   Gender           199 non-null    object
 2   Speed_of_Impact  197 non-null    float64
 3   Helmet_Used      200 non-null    object
 4   Seatbelt_Used    200 non-null    object
 5   Survived         200 non-null    int64
```

```
dtypes: float64(1), int64(2), object(3)
memory usage: 9.5+ KB
```

0.1.2 Data Cleaning, Preprocessing and Analysis

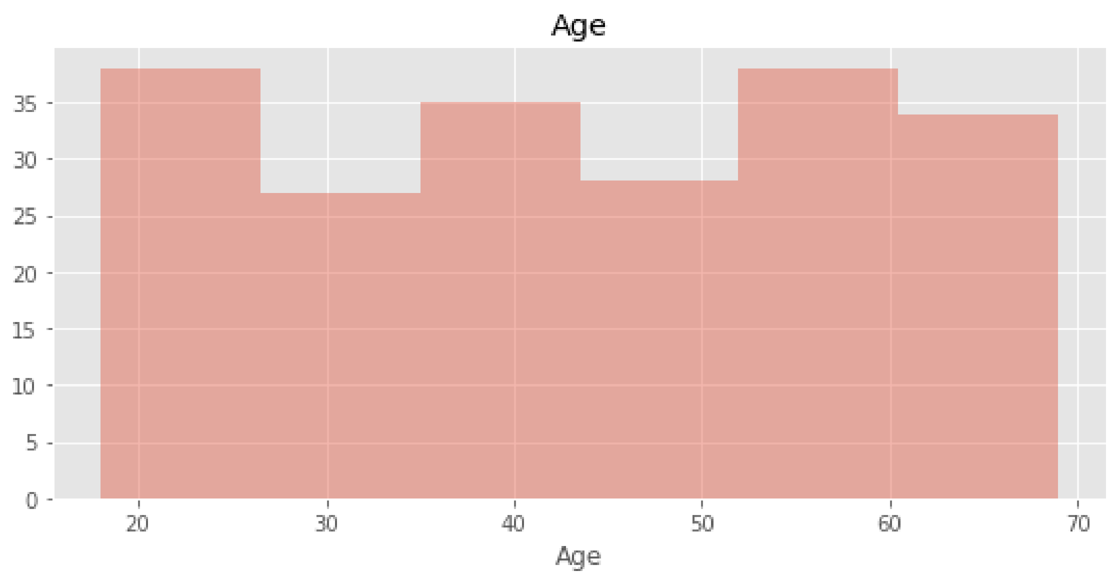
Prior to testing the model and tuning the hyperparameters, we will perform data preprocessing, analysis, and cleaning. This entails dealing with missing values, encoding categorical variables, and making sure all data formats are suitable for graphs and analysis.

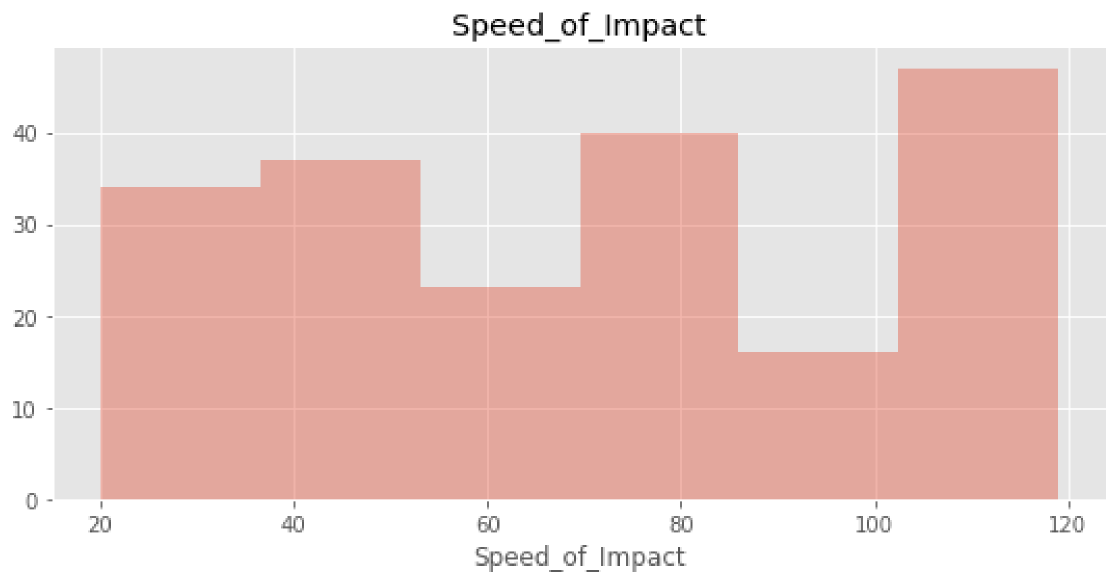
[4]:
```python
missing_values = path.isnull()
print(missing_values)
```

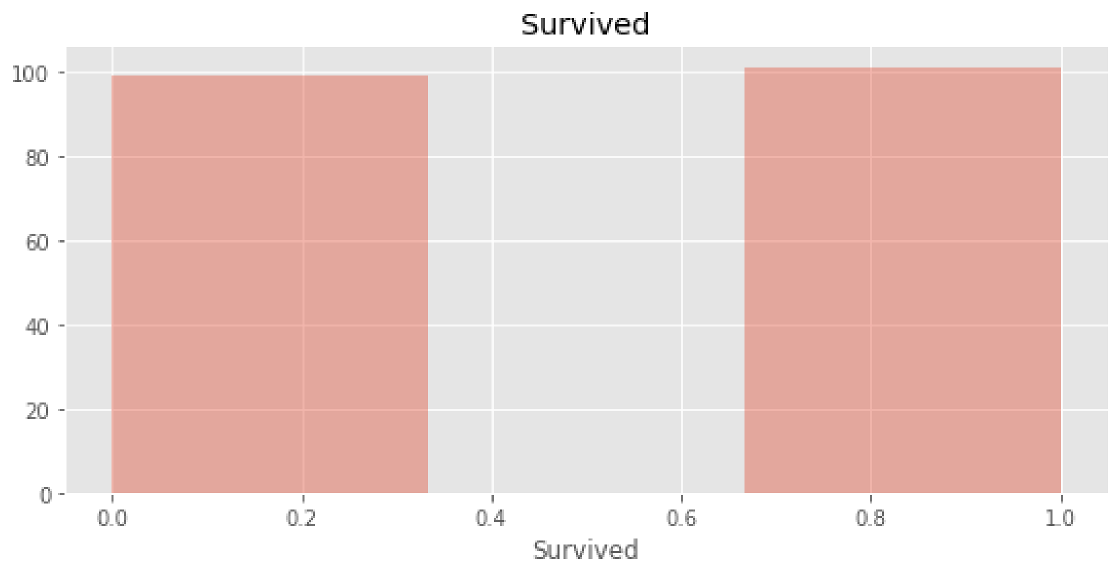
```
       Age  Gender  Speed_of_Impact  Helmet_Used  Seatbelt_Used  Survived
0    False   False            False        False          False     False
1    False   False            False        False          False     False
2    False   False            False        False          False     False
3    False   False            False        False          False     False
4    False   False            False        False          False     False
..     …       …                …            …              …         …
195  False   False            False        False          False     False
196  False   False            False        False          False     False
197  False   False            False        False          False     False
198  False   False            False        False          False     False
199  False   False            False        False          False     False

[200 rows x 6 columns]
```

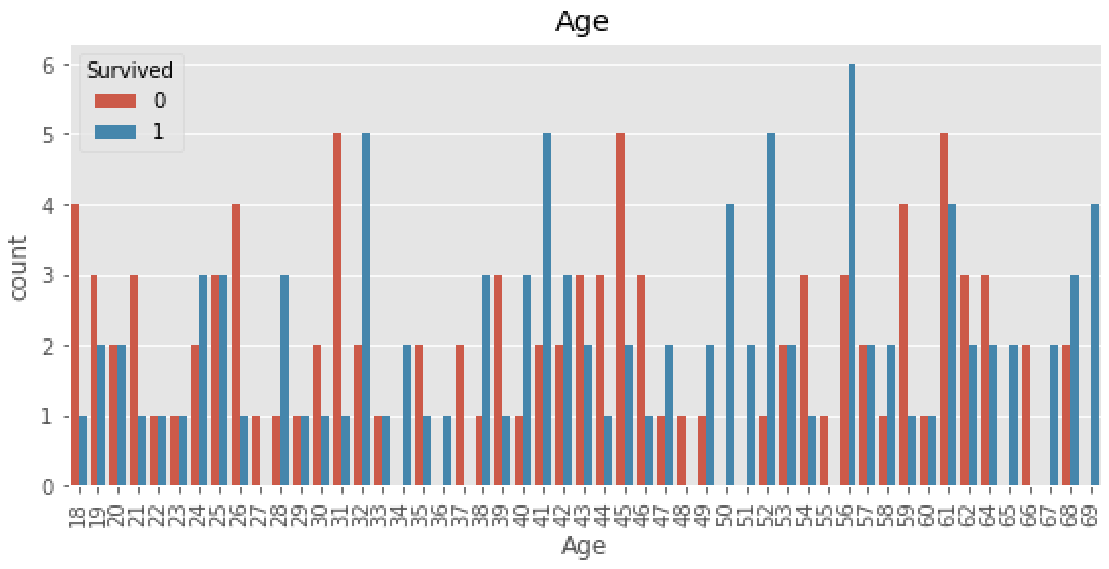
[5]:
```python
# plotting histogram for each numerical variable
plt.style.use("ggplot")
for column in [ 'Age','Speed_of_Impact', 'Survived'] :
    plt.figure(figsize=(20,4))
    plt.subplot(121)
    sns.distplot(path[column], kde=False)
    plt.title(column)
```

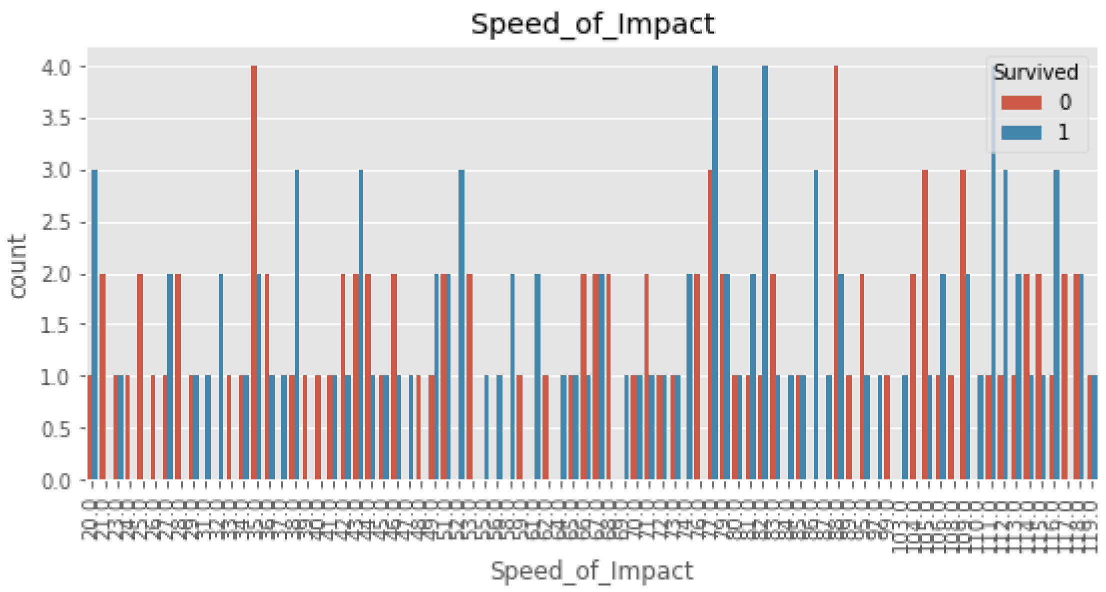

Age
35
30
25
20
15
10
5
0
20
30
40
50
60
70
Age

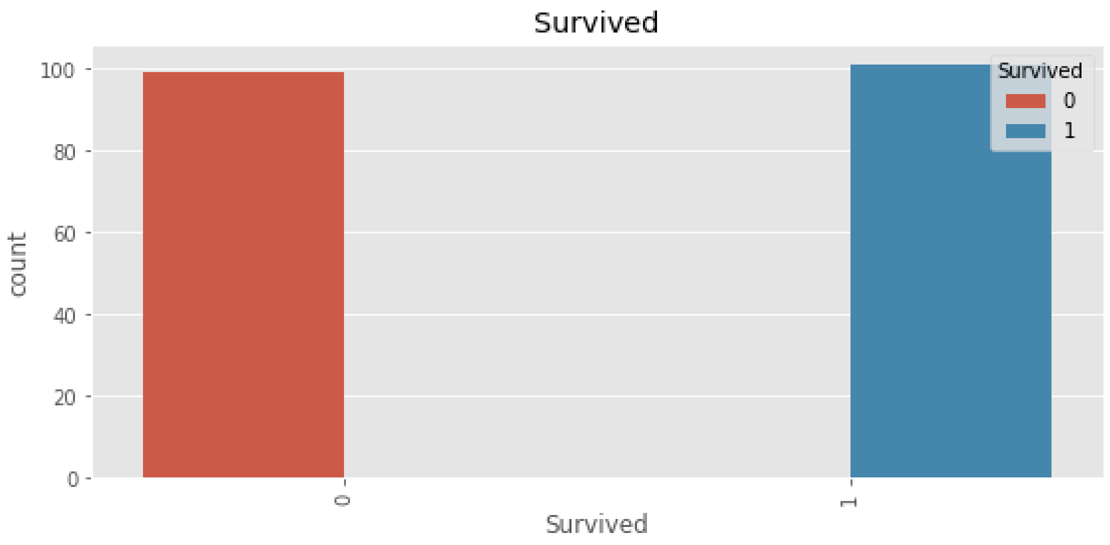

Speed_of_Impact
40
30
20
10
0
20
40
60
80
100
120
Speed_of_Impact



```
[6]: plt.style.use("ggplot")
     for column in [ 'Age','Speed_of_Impact', 'Survived']:
         plt.figure(figsize=(20,4))
         plt.subplot(121)
         sns.countplot(path[column], hue=path["Survived"])
         plt.title(column)
         plt.xticks(rotation=90)
```







[7]:
```
# indentifying the categorical variables
Cata= path.select_dtypes(include= ["object"]).columns
print(Cata)
```

```
Index(['Gender', 'Helmet_Used', 'Seatbelt_Used'], dtype='object')
```

[8]:
```
path.head(3)
```

```
[8]:    Age  Gender  Speed_of_Impact Helmet_Used Seatbelt_Used  Survived
     0   56  Female             27.0          No            No         1
     1   69  Female             46.0          No           Yes         1
     2   46    Male             46.0         Yes           Yes         0
```

```
[9]: # initializing label encoder
     from sklearn.preprocessing import LabelEncoder
     le= LabelEncoder()

     # iterating through each categorical feature and label encoding them
     for feature in Cata:
         path[feature]= le.fit_transform(path[feature])
```

```
[10]: # label encoded dataset
      path.head()
```

```
[10]:    Age  Gender  Speed_of_Impact  Helmet_Used  Seatbelt_Used  Survived
      0   56       0             27.0            0              0         1
      1   69       0             46.0            0              1         1
      2   46       1             46.0            1              1         0
      3   32       1            117.0            0              1         0
      4   60       0             40.0            1              1         0
```

```
[11]: # Convert empty strings to NaN
      path.replace('', np.nan, inplace=True)

      # Replace NaN, inf, and large values
      path.replace([np.nan], 0, inplace=True)

      print(path.head())  # Check results
```

```
   Age  Gender  Speed_of_Impact  Helmet_Used  Seatbelt_Used  Survived
0   56       0             27.0            0              0         1
1   69       0             46.0            0              1         1
2   46       1             46.0            1              1         0
3   32       1            117.0            0              1         0
4   60       0             40.0            1              1         0
```

```
[12]: X=path.drop('Survived',axis=1)
      y=path['Survived']
```

```
[13]: from sklearn.model_selection import train_test_split
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
       ↪random_state=3)
```

```
[14]: trans = MinMaxScaler()
      X_train = trans.fit_transform(X_train)
```

[15]:
```python
X_train = DataFrame(X_train)
X_train.head(4)
```

[15]:
```
          0    1         2    3    4
0  0.647059  0.0  0.705882  1.0  1.0
1  0.549020  1.0  0.638655  1.0  0.0
2  0.803922  0.0  0.663866  0.0  0.0
3  0.176471  0.0  0.226891  1.0  0.0
```

0.1.3 Model Building and Evaluation

A KNN classifier with default hyperparameters will be constructed to predict survival outcomes, and its performance will be assessed using the f1 score.

[16]:
```python
start_time = time.time()
knn = KNeighborsClassifier().fit(X_train, y_train)
y_pred = knn.predict(X_test)
end_time = time.time() - start_time
print("Execution time: " + str(end_time))
f1_score=f1_score(y_test, y_pred, average='macro')
print(f1_score)
```

```
Execution time: 0.018085956573486328
0.431602879878742
```

Although the model's low f1 score suggests that characteristics like age, gender, and safety precautions have some impact on survival outcomes, our goal is to investigate if GA can enhance the results.

By adjusting the classifier's hyperparameters, we hope to enhance the model. We wish to adjust the hyperparameters using a genetic method. Three KNN hyperparameters—k, weights, and metric—are applied in this instance. However, according to (Li Yang and Abdallah Shami), the K hyperparameter is the most crucial one.

- k (n_neigbours) is usually an odd number when dealing with even classes. In this case, we have two classes, hence, we have started by selecting a range of small odd values. Then include the even values to make the technique to choose the values.
- Weights: This hyperparameter have only two possibilities and we have selected both of them, that is 'uniform' and 'distance'.
- Metric: We are going to select most popular the distance metric that is
  - Manhattan distance
  - Euclidean distance
  - Minkowski distance

To ensure fairness, we must establish a grid of hyperparameters that specify the size of the GA that must be run in order to identify the optimal model. You can get the full list of adjustable hyperparameters on Scikit-Learn.

0.1.4 Tuning using genetic algorithms

[17]:
```
from sklearn.model_selection import GridSearchCV
weights = ['uniform','distance']
metric = ['minkowski','euclidean','manhattan']
n_neighbors = [1, 2, 3, 4, 5, 10, 20]
param_grid = dict(n_neighbors=n_neighbors, metric=metric, weights=weights)
print(param_grid)
```

```
{'n_neighbors': [1, 2, 3, 4, 5, 10, 20], 'metric': ['minkowski', 'euclidean',
'manhattan'], 'weights': ['uniform', 'distance']}
```

[18]:
```
Size= len(weights)*len(n_neighbors)*len(metric)
print(Size)
```

```
42
```

Firstly, we create the grid for GA, the list of genes to create chromosomes. The gene is created by randomly choosing from the list

[19]:
```
weights = ['uniform','distance']
metric = ['minkowski','euclidean','manhattan']
lower_n_neighbors, upper_n_neighbors = 1, 20
```

We create a mutation function that randomly selects one of the genes in the individual chromosome, and mutates. How this mutation happens depends on the gene. For example, the one with only two options can simply be swapped around to the other possible value.

[20]:
```
def mutate(individual):
    '''This function randomly selects a gene and randomly generates a new
      value for it based on a set of rules
    '''
    #print(individual[0], individual[1], individual[2])
    gene = random.randint(0,2) #select which parameter to mutate
    if gene == 0:
        if individual[0] == 'uniform':
            individual[0] = 'distance'
        else:
            individual[0] = 'uniform'


    elif gene == 1:
        if individual[1] == 'minkowski':
            individual[1] = random.choice(['euclidean','manhattan'])


        elif individual[1] == 'euclidean':
            individual[1] = random.choice(['minkowski' , 'manhattan'])
        elif individual[1] == 'manhattan':
            individual[1] = random.choice(['euclidean','minkowski'])

    elif gene == 2:
        individual[2] = random.choice(np.arange(1,21))
    #print(individual[0], individual[1], individual[2])

    #print('' '')

    return individual,
```

Using the hyperparameters of an assessed individual chromosome, we are constructing a model and subsequently determining the model's F1-score. The particular chromosome's fitness score is represented by this F1-score.

[21]:
```
def evaluate(individual):
    '''
    build and test a model based on the parameters in an individual and return
    the f1 value
    '''
    #  extract the values of the parameters from the individual chromosome
    weights = individual[0]
    metric = individual[1]
    n_neighbors = individual[2]

    model = KNeighborsClassifier(weights= weights,
                                 metric=metric,
                                 n_neighbors=n_neighbors).fit(X_train, ␣
 ↪y_train.values.ravel())
    y_pred = model.predict(X_test)
    f1 =f1_score(y_test.values.ravel(), y_pred, average='macro')

    return f1,
```

[22]:
```
creator.create("FitnessMax", base.Fitness, weights=(1.0,)) # Maximise the␣
 ↪fitness function value
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
```

[23]:
```
N_CYCLES = 1

#define how each gene will be generated (e.g. weights is a random choice from␣
 ↪the weights list).
toolbox.register("attr_weights",random.choice, weights)
toolbox.register("attr_metric",random.choice, metric)
toolbox.register("attr_n_neighbors", random.randint, lower_n_neighbors,␣
 ↪upper_n_neighbors)
```

[24]:
```
# This is the order in which genes will be combined to create a chromosome
toolbox.register("individual", tools.initCycle, creator.Individual,
                 (toolbox.attr_weights, toolbox.attr_metric, toolbox.
 ↪attr_n_neighbors), n=N_CYCLES)

toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

[25]:
```
num_dic = 1
population_size = 12
gen=Size//population_size//num_dic
crossover_probability = 0.9
num_run = 30
mutation_probability = 0.3
```

We have created two functions (mutate, evaluate) and taken the other two from generic DEAP functions (mate, select). We have decided to use 20% of the population size as the tournament.

[26]:
```
toolbox.register("mate", tools.cxOnePoint)
toolbox.register("mutate",mutate)
toolbox.register("select", tools.selTournament, tournsize=population_size//4)
toolbox.register("evaluate", evaluate)
```

Finally, after the size of the population, crossover probability, mutation probability, and the number of generations are set. We combine these functions to create the GA.

[27]:
```
from sklearn.metrics import f1_score
f1_values = []
Execution_time = []
for loop_ga in range(num_run):
    pop = toolbox.population(n=population_size)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", np.max)

    start_time = time.time()
    pop, log = algorithms.eaSimple(pop, toolbox, cxpb=crossover_probability,␣
 ↪stats = stats,
                               mutpb = mutation_probability, ngen=gen,␣
 ↪halloffame=hof,
                               verbose=False)
    end_time = time.time() - start_time
    best_parameters = hof[0] # save the optimal set of parameters
    print(best_parameters)
    f1_values.append(max(log.select("max")))
    Execution_time.append(end_time)
    print(f1_values[loop_ga])
    Execution_time[loop_ga]
    print("Execution time: " + str((end_time)) + 's')
print(f1_values)
print(np.average(f1_values))
print(stdev(f1_values))
print(Execution_time)
print(np.average(Execution_time))
print(stdev(Execution_time))
print(np.median(f1_values))
print(np.percentile(f1_values, [75]))
print(np.percentile(f1_values, [25]))
```

```
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.6009304523468018s
['uniform', 'manhattan', 8]
0.5165323701028064
Execution time: 0.5152487754821777s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.4141521453857422s
['distance', 'minkowski', 11]
0.5333333333333333
Execution time: 0.5034575462341309s
['distance', 'minkowski', 12]
0.5970695970695972
Execution time: 0.6262428760528564s
['distance', 'euclidean', 12]
0.5970695970695972
Execution time: 0.4667854309082031s
['uniform', 'euclidean', 11]
0.5333333333333333
Execution time: 0.6557726860046387s
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.6775376796722412s
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.7897036075592041s
['uniform', 'euclidean', 11]
0.5333333333333333
Execution time: 0.675039529800415s
['distance', 'minkowski', 11]
0.5333333333333333
Execution time: 0.718010425567627s
['uniform', 'euclidean', 11]
0.5333333333333333
```

```
Execution time: 0.505232572555542s
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.6469993591308594s
['uniform', 'euclidean', 11]
0.5333333333333333
Execution time: 0.5917670726776123s
['uniform', 'euclidean', 11]
0.5333333333333333
Execution time: 0.6886792182922363s
['distance', 'euclidean', 12]
0.5970695970695972
Execution time: 0.6394364833831787s
['distance', 'euclidean', 11]
0.5333333333333333
Execution time: 0.5640068054199219s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.6530416011810303s
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.7696733474731445s
['uniform', 'euclidean', 11]
0.5333333333333333
Execution time: 0.4494762420654297s
['distance', 'minkowski', 11]
0.5333333333333333
Execution time: 0.33391380310058594s
['distance', 'euclidean', 12]
0.5970695970695972
Execution time: 0.5895750522613525s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.5656466484069824s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.5947623252868652s
['uniform', 'minkowski', 11]
0.5333333333333333
Execution time: 0.5762999057769775s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.5698695182800293s
['uniform', 'manhattan', 10]
0.5165323701028064
Execution time: 0.5801780223846436s
['uniform', 'euclidean', 11]
0.5333333333333333
```

```
Execution time: 0.6100821495056152s
['distance', 'minkowski', 12]
0.5970695970695972
Execution time: 0.48270344734191895s
['uniform', 'manhattan', 8]
0.5165323701028064
Execution time: 0.5444169044494629s
[0.5333333333333333, 0.5165323701028064, 0.5165323701028064, 0.5333333333333333,
0.5970695970695972, 0.5970695970695972, 0.5333333333333333, 0.5333333333333333,
0.5333333333333333, 0.5333333333333333, 0.5333333333333333, 0.5333333333333333,
0.5333333333333333, 0.5333333333333333, 0.5333333333333333, 0.5970695970695972,
0.5333333333333333, 0.5165323701028064, 0.5333333333333333, 0.5333333333333333,
0.5333333333333333, 0.5970695970695972, 0.5165323701028064, 0.5165323701028064,
0.5333333333333333, 0.5165323701028064, 0.5165323701028064, 0.5333333333333333,
0.5970695970695972, 0.5165323701028064]
0.5394757870945701
0.02718889438140525
[0.6009304523468018, 0.5152487754821777, 0.4141521453857422, 0.5034575462341309,
0.6262428760528564, 0.4667854309082031, 0.6557726860046387, 0.6775376796722412,
0.7897036075592041, 0.675039529800415, 0.718010425567627, 0.505232572555542,
0.6469993591308594, 0.5917670726776123, 0.6886792182922363, 0.6394364833831787,
0.5640068054199219, 0.6530416011810303, 0.7696733474731445, 0.4494762420654297,
0.33391380310058594, 0.5895750522613525, 0.5656466484069824, 0.5947623252868652,
0.5762999057769775, 0.5698695182800293, 0.5801780223846436, 0.6100821495056152,
0.48270344734191895, 0.5444169044494629]
0.5866213877995808
0.10101871104548861
0.5333333333333333
[0.53333333]
[0.52073261]
```

The author tested a genetic algorithm technique on tuning the KNN hyperparameters. For this purpose, author first built the model using the default hyperparameters on accident survival prediction dataset.

The author have compared the performance of the GA with the model trained with default hyperparameters. The obtained results showed that the GA can generate good performance is better than model trained with default hyperparameters in terms of F1 score.

```
[ ]:
```