УНИВЕРСИТЕТ ИТМО

Практика 9. Генетические алгоритмы: отбор признаков и оптимизация гиперпараметров

Михаил А. Каканов[1] Олег А. Евстафьев[1]

[1] Факультет систем управления и робототехники, Университет ИТМО
{makakanov,oaevstafev}@itmo.ru

Ноябрь 2021
Курс «Прикладной искусственный интеллект»

Повестка дня

1. Подготовка среды
2. Разработка модели
3. Валидация
4. Оптимизация модели
5. Анализ результатов

Содержание

Подготовка среды I

В этом блокноте будет использоваться набор данных ’Santander Customer Satisfaction’, поскольку он имеет высокую размерность (300+) и подходит для тестирования процесса оптимизации.

Подготовим среду для построения оптимизационной модели и поиска хорошей прогностической модели (признаки и гиперпараметры) для данного набора данных. В данной работе используются следующие пакеты:

- Numpy
- Pandas
- Scipy
- Sklearn

Источник: Genetic Algo: Feature Selection Hyperparameters

```python
import numpy as np
import pandas as pd
import numpy.random as rnd
from scipy import spatial

from sklearn.model_selection import train_test_split, cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.feature_selection import VarianceThreshold
from sklearn import metrics

from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

from sklearn.utils.testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning

import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import seaborn as sns
```

```python
df_train = pd.read_csv("../input/santander-customer-satisfaction/train.csv", index_col="ID")
df_test = pd.read_csv("../input/santander-customer-satisfaction/test.csv", index_col="ID")

print("{} rows and {} columns".format(*df_train.shape))

df_train.TARGET.value_counts(normalize=True)
```

```
76020 rows and 370 columns

0    0.960431
1    0.039569
Name: TARGET, dtype: float64
```

Подготовка среды IV

УНИВЕРСИТЕТ ИТМО

```python
display(df_train.dtypes.value_counts())

print("Number of Missing entries: " + str(df_train.isnull().sum().sum()))

df_train.var3.describe()
```

```
int64      259
float64    111
dtype: int64
Number of Missing entries: 0

count      76020.000000
mean       -1523.199277
std        39033.462364
min      -999999.000000
25%            2.000000
50%            2.000000
75%            2.000000
max          238.000000
Name: var3, dtype: float64
```

```python
display(df_train['var3'].value_counts(normalize=True)[0:3])

df_train['var3'] = df_train['var3'].replace(-999999, 2)
```

```
 2          0.975599
 8          0.001815
-999999     0.001526
Name: var3, dtype: float64
```

В процессе оптимизации к новым обучающим данным будет применена 5-кратная перекрестная валидация, которая поможет получить стабильную оценку того, какие характеристики и гиперпараметры следует включить в окончательную модель.

```
df_train, df_validation = train_test_split(df_train,
                                           test_size=0.2,
                                           random_state=1989,
                                           stratify=df_train.TARGET,
                                           shuffle=True)

kfold = KFold(n_splits=5, random_state=1989, shuffle=True)
```

Предобработка данных I

```
# Identify columns
fts_num = df_train.drop(axis=1,columns=['TARGET']).select_dtypes(np.number).columns

# Numerical Transformer StandardScaler
trans_num = Pipeline(steps = [('Standarise', StandardScaler()), ('MinMax', MinMaxScaler())])

# Create a single Preprocessing step for predictors
preprocessor_preds = ColumnTransformer(
    transformers=[
        ('num', trans_num, fts_num) # Centre and scale and constrain range
    ])

# Apply the transformations to train
df_train2 = pd.DataFrame(preprocessor_preds.fit_transform(df_train))
df_train2.columns = fts_num

# Apply the transformations to validation
df_validation2 = pd.DataFrame(preprocessor_preds.fit_transform(df_validation))
df_validation2.columns = fts_num

# Apply the transformations to test
df_test2 = pd.DataFrame(preprocessor_preds.fit_transform(df_test))
df_test2.columns = fts_num

# Create preprocessed training data
df_train = pd.concat([df_train2,
                      df_train.drop(axis=1,columns=fts_num).reset_index().drop(axis=1,columns=['ID'])],
```

Предобработка данных II


```
                    axis=1)

# Create preprocessed validation data
df_validation = pd.concat([df_validation2,
                           df_validation.drop(axis=1,columns=fts_num).reset_index().drop(axis=1,columns=['ID'])],
                          axis=1)

# Create preprocessed test data
df_test = pd.concat([df_test2,
                     df_test.drop(axis=1,columns=fts_num).reset_index().drop(axis=1,columns=['ID'])],
                    axis=1)

# Clear objects
del df_train2, df_validation2, df_test2, fts_num, trans_num, preprocessor_preds
```

Содержание

Разработка оптимизационной модели I

Функция ниже генерирует начальные решения как для характеристик, так и для гиперпараметров.

Выбор начальных характеристик:

- Для каждого решения в популяции случайным образом выбирается процент в диапазоне 10-91%, который будет применен к общему количеству признаков.
- Для каждого решения случайным образом выбираются признаки до тех пор, пока не будет получен процент столбцов от общего числа столбцов.

Выбор начальных гиперпараметров:

- Для каждого решения генерируйте случайное значение для каждого гиперпараметра между заданными пользователем min и max

Разработка оптимизационной модели II

УНИВЕРСИТЕТ ИТМО

```python
# Randomly generate candidates
def f_random_candidates(features_name, population, hyperparams, output_type, df_pop=False):
    '''create an initial population'''

    # Create solution for features
    if output_type == 'feature':

        # Initial population will have between 10-91% of features
        feature_size = rnd.choice(a=range(10,91),size=population, replace=True)
        feature_size = [np.round(pct / 100 * len(features_name)) for pct in feature_size]

        # Create a list of feature positions for each candidate
        selection = [rnd.choice(a=range(0,len(features_name)-1), replace=False, size=cols.astype('int')) \
                     for cols in feature_size]

        selection = [list(selection[i]) for i in range(len(selection))]

        return selection

    # Create solution for hyperparameters
    elif (output_type == 'hyperparams') & (hyperparams != False):

        # Generate random numbers in range for each hyperparameter
        random_hyperparams = []
        for j in range(len(hyperparams['names'])):
            temp = (np.random.uniform(hyperparams['min_value'][j],
```

Разработка оптимизационной модели III


```
                                  hyperparams['max_value'][j],
                                  population))
        random_hyperparams.append(temp)

    # Get length of features
    n_features = df_pop['features'].apply(len).tolist()

    # Store hyperparameters in diction
    hyperparam_vals = []
    for i in range(population):
        val = {'name':[],'value':[]}
        for j in range(len(hyperparams['names'])):
            val['name'].append(hyperparams['names'][j])
            temp = random_hyperparams[j][i]
            if hyperparams['type'][j] == 'int':
                temp = np.int64(round(temp))
            if hyperparams['names'][j] == 'max_features':
                temp = min(temp, n_features[i])
            val['value'].append(temp)

        hyperparam_vals.append(val)
        del val

    return hyperparam_vals
```

Скрещивание признаков:

- Случайным образом генерируется целое число, которое представляет собой точку пересечения между первой и последней характеристикой (в конечном счете, индекс столбца).
- Взвешенная выборка решений предыдущего поколения и выбор двух родителей
- Создание дочернего решения, которое имеет все признаки до точки пересечения (индекс столбца) от первого родителя и все признаки от второго родителя после точки пересечения.

Скрещивание гиперпараметров:

- Взвешенная выборка решений предыдущего поколения размером в число гиперпараметров
- Случайный выбор гиперпараметров из родительского решения

- Создать дочерние гиперпараметры из выбранного родительского решения

```python
# Crossover function
def f_gen_child_crossover(df, features_name, hyperparams, output_type):
    '''Mutate 2 parents to create a child'''

    # Crossover features
    if output_type == 'feature':

        # Create an integer list of features
        l_features = list(range(0,len(features_name)))

        # Identify a random cross over point
        cross_point = np.int(rnd.randint(low=0, high=len(features_name), size=1))

        # Extract Two Parents
        selection = np.random.choice(df.features,
                                     size=2,
                                     replace=False,
                                     p=df.probability)
        par1 = list(selection[0])
        par2 = list(selection[1])

        # Convert to Boolean
```

```
        par1 = [item in par1 for item in l_features]
        par2 = [item in par2 for item in l_features]

        # Single point cross over and convert to indices
        child = par1[0:cross_point] + par2[cross_point:]
        child = [i for i,x in enumerate(child) if x == True]

        # Return
        return child

    # Crossover hyperparameters
    elif (output_type == 'hyperparams') & (hyperparams != False):

        # Identify the number of parameters
        n_hyperparameters = len(hyperparams['min_value'])

        # Extract n Parents
        selection = np.random.choice(df.hyperparameters,
                                     size=n_hyperparameters,
                                     replace=False,
                                     p=df.probability)

        # Randomly choose which parent to select each parameter from
        parent_choice = list(np.random.choice(range(n_hyperparameters),
                                              size = n_hyperparameters,
                                              replace=False))

```

```
        # Copy the parent as the child
        child = selection[0]

        # Update child vector with choosen parent
        for i in range(n_hyperparameters):
            child['value'][i] = selection[parent_choice[i]]['value'][i]

        # Return
        return child
```

Разработка оптимизационной модели VIII

Мутация признаков:

- Для каждого признака в кадре данных сгенерируйте случайное число от 0 до 1.
- Если сгенерированная вероятность ниже указанной пользователем скорости мутации, то поменяйте местами переключатели для этого столбца (т.е. если функция включена, то удалите ее и наоборот).

Мутация гиперпараметров:

- Для каждого гиперпараметра в выбранной модели сгенерируйте случайное число от 0 до 1.
- Если случайное число ниже указанной пользователем скорости мутации, сгенерируйте случайное число в указанном диапазоне.

- Наконец, проверьте, не находится ли гиперпараметр за пределами диапазона min-max, и при необходимости уменьшите его до этого диапазона.

```python
# Mutate function
def f_gen_child_mutate(candidate, features_name, p_mutate,
                       hyperparams, output_type,
                       hyperparams_increment):
    '''Mutate 2 parents to create a child'''

    # Mutate Features
    if output_type == 'feature':

        # Create an integer list of features
        l_features = list(range(0,len(features_name)))

        # Convert feature into boolean vector
        candidate = [item in candidate for item in l_features]

        # Conditionally mutate features in chromosome (reverse binary flag)
        candidate_new = []
        for item in candidate:
            if rnd.rand() <= p_mutate:
                candidate_new.append(not item)
```

```python
            else:
                candidate_new.append(item)

        # Convert to indicies
        candidate_new = [i for i,x in enumerate(candidate_new) if x == True]

        # Return
        return candidate_new

    # Mutate hyperparameters
    elif (output_type == 'hyperparams') & (hyperparams != False):

        # Identify size of mutation
        v_mutate = (np.random.uniform((1-hyperparams_increment),
                                      (1+hyperparams_increment), 1)).item()

        # Identify Min and Max for parameters
        l_min =  hyperparams['min_value']
        l_max =  hyperparams['max_value']

        # Identify the number of parameters
        n_hyperparameters = len(l_min)

        # Probabilistically mutate certain parameters
        candidate_new = []
        for i in range(n_hyperparameters):
            if rnd.rand() <= p_mutate:
```

Разработка оптимизационной модели XI


```python
                temp = candidate['value'][i] * v_mutate
                if hyperparams['type'][i] == 'int':
                    temp = np.int64(round(temp))
                candidate_new.append(temp)
            else:
                candidate_new.append(candidate['value'][i])

        # Ensure that value is between ranges
        for i in range(n_hyperparameters):
            if (candidate_new[i] < l_min[i]):
                candidate_new[i] = l_min[i]
            elif (candidate_new[i] > l_max[i]):
                candidate_new[i] = l_max[i]

        # Update values
        candidate['value'] = candidate_new

        # return
        return candidate
```

Разработка оптимизационной модели XII

УНИВЕРСИТЕТ ИТМО

```python
# Function to generate a population of candidates
def f_generate_population(inital_flag, population, features_name,
                          p_crossover, p_mutate,
                          hyperparams, hyperparams_increment,
                          hyperparams_multiple,
                          df=False, generation=0, initalise=False):
    '''Generates all candidates in population'''

    # Create initial population
    if inital_flag == True:

        # Check if there is an initial solution & reduce
        # population by one if there is
        if initalise != False:
            population = population - 1

        # generate random features
        df_pop = pd.DataFrame({'generation':generation,
                               'candidate':range(0,population),
                               'features':f_random_candidates(features_name,
                                                              population,
                                                              hyperparams,
                                                              output_type = 'feature')})

        # Duplicate rows for population range
        df_pop = df_pop.loc[df_pop.index.repeat(hyperparams_multiple)]
```

Разработка оптимизационной модели XIII


```python

        # Generate population
        df_pop['hyperparameters'] = \
            f_random_candidates(features_name=features_name,
                                population = population * hyperparams_multiple,
                                hyperparams=hyperparams,
                                output_type = 'hyperparams',
                                df_pop=df_pop)

        # If Initial solution then add in
        if initialise != False:
            df = pd.DataFrame({'generation':generation,
                               'candidate':range(population, population + 1),
                               'features':[initalise['features']],
                               'hyperparameters':[initalise['hyperparameters']]},
                             index=[population])

            df_pop = df_pop.append(df)


        # Reset Index
        df_pop.index = range(0, population * hyperparams_multiple)

        # Return
        return df_pop
    else:
        # Distribute the population
```

Разработка оптимизационной модели XIV

```python
        population_crossover = round(population * p_crossover)
        population_remainder = population-population_crossover

        # ----- Create crossover candidates -----

        # Create crossover populate for feature selection
        df_pop = pd.DataFrame({'generation':generation,
                               'candidate':range(0,population_crossover)})
        df_pop['features'] = [f_gen_child_crossover(df=df,
                                                    features_name=features_name,
                                                    hyperparams=hyperparams,
                                                    output_type = 'feature') \
                              for _ in range(population_crossover)]

        # Duplicate rows for population range
        df_pop = df_pop.loc[df_pop.index.repeat(hyperparams_multiple)]

        # Create crossover population for hyperparameters
        df_pop['hyperparameters'] = \
            [f_gen_child_crossover(df=df,
                                   features_name=features_name,
                                   hyperparams=hyperparams,
                                   output_type = 'hyperparams') \
             for _ in range(population_crossover * hyperparams_multiple)]

        # Reset Index
        df_pop.index = range(0, population_crossover * hyperparams_multiple)
```

```python

        # ----- Create Randomly Selected candidates -----

        # Initialise population
        df_temp = pd.DataFrame({'generation':generation,
                                'candidate':range(population_crossover,
                                                  population)})
        # Randomly select candidates
        selected_index = \
            df.sample(n=population_remainder,
                      replace = False,
                      weights=df.probability).candidate.tolist()

        # Extract hyperparameters
        selected_features = df.iloc[selected_index,:].features.tolist()
        selected_params = df.iloc[selected_index,:].hyperparameters.tolist()

        # Update temp dataframe
        df_temp['features'] = [selected_features[i]
                               for i in range(len(selected_features))]
        df_temp['hyperparameters'] = [selected_params[i]
                               for i in range(len(selected_params))]

        # Duplicate rows for population range
        df_temp = df_temp.loc[df_temp.index.repeat(population_remainder)]

        # Append to population dataframe
```

```
df_pop = df_pop.append(df_temp,ignore_index=True)

# Clear up
del selected_features, selected_params, df_temp

# ----- Mutate Population -----

# Mutate existing candidate features
df_pop['features'] = \
    df_pop.features.apply(f_gen_child_mutate,
                          features_name=features_name,
                          p_mutate=p_mutate,
                          hyperparams=hyperparams,
                          output_type = 'feature',
                          hyperparams_increment=hyperparams_increment)

# Mutate existing candidate hyperparameters
df_pop['hyperparameters'] = \
    df_pop.hyperparameters.apply(f_gen_child_mutate,
                                 features_name=features_name,
                                 p_mutate=p_mutate,
                                 hyperparams=hyperparams,
                                 output_type = 'hyperparams',
                                 hyperparams_increment=
                                     hyperparams_increment)

# ----- Hyperparameter fix -----
```

Разработка оптимизационной модели XVII


```python
        if hyperparams != False:

            # Get length of features
            n_features = df_pop['features'].apply(len).tolist()

            # Hyperparameter fix
            for i in range(population):
                for j in range(len(hyperparams['names'])):
                    if hyperparams['names'][j] == 'max_features':
                        if df_pop.hyperparameters[i]['value'][j] > n_features[i] :
                            df_pop.hyperparameters[i]['value'][j] = \
                                n_features[i]

        # Return
        return df_pop
```

Содержание

```
# Evaluate solution fitness
@ignore_warnings(category=ConvergenceWarning)
def f_fitness(model, eval_metric, features, target,
              feature_idx, kfold, hyperparams):
    '''Evaluates fitness of proposed solution'''

    # Extract the hyperparameters
    n_hyperparams = len(hyperparams['name'])
    hyperparameters = {hyperparams['name'][0]:hyperparams['value'][0]}
    if n_hyperparams > 1:
        for i in range(n_hyperparams):
            tempparameters = {hyperparams['name'][i]:hyperparams['value'][i]}
            hyperparameters = {**hyperparameters, **tempparameters}

    # Determine CV strategy
    if kfold == False:
        kfold = 5
    else:
        kfold = kfold

    # Apply cross validation to the modells
    results = cross_val_score(model.set_params(**hyperparameters),
                              features.iloc[:,feature_idx],
                              target,
                              cv=kfold,
                              scoring=eval_metric)

```

```
    # Replace NA's with 0
    results[np.isnan(results)] = 0

    return results
```

```
# Apply evaluation score to current population
def f_evaluation_score(df, features, target, eval_metric, model,
                       kfold, hyperparams):
    '''Apply f_fitness to each candidate'''

    # Calculate the evaluation metric
    evaluation_score = []
    for val in range(0, len(df)):
        eval_score = f_fitness(model=model,
                               eval_metric=eval_metric,
                               features = features,
                               target=target,
                               feature_idx=df['features'][val],
                               kfold=kfold,
                               hyperparams=df['hyperparameters'][val])

        # Average evaluation metric across folds
        evaluation_score.append(eval_score.mean())
```

Валидация III


```python

        # Clear object
        del eval_score

    # Clear object
    del val

    # return evaluation score
    return evaluation_score
```

```python
# Calculate jaccard similarity
def f_j_sim(list1 , list2):
    s1 = set(list1)
    s2 = set(list2)
    return float(len(s1.intersection(s2)) / len(s1.union(s2)))

# Calculate cosine similarity
def f_c_sim(l_other, l_best_score):

    # Extract hyperparameter values
    l_other = l_other['value']

    # calculate similarity
```

Валидация IV


```python
    sim = 1 - spatial.distance.cosine(l_best_score, l_other)

    # return
    return sim

# Calculate similarity between candidates and probability for next gen selection
def f_sim_n_prob(df):

    # Calculate similarity of solutions with best solutions - Features
    l_best_score = df.features[df['fitness_score'].idxmax()]
    df['similarity_features'] = df['features'].apply(f_j_sim, list2=l_best_score)
    del l_best_score

    # Calculate similarity of solutions with best solutions
    l_best_score = df.hyperparameters[df['fitness_score'].idxmax()]['value']
    df['similarity_hyperparameters'] = df.hyperparameters.apply(f_c_sim, l_best_score=l_best_score)
    del l_best_score

    # Calculate cumulative probability for future stages
    df['probability'] = (df['fitness_score'] / sum(df['fitness_score']))

    # return
    return df
```

Валидация V


```python
# Function to populate attributes of candidates
def f_population_features(df, features, target, desiriability,
                          eval_metric, model, kfold, hyperparams):
    '''Get features of all candidates in population'''

    # Calculate feature size for candidates
    df['feature_size'] = df['features'].apply(len)

    # Calculate evaluation score for candidates
    df['evaluation_score'] = f_evaluation_score(df,
                                                features,
                                                target,
                                                eval_metric,
                                                model,
                                                kfold,
                                                hyperparams)

    # Conditionally create desirability fitness score
    if desiriability != False:

        # Create scalars - Features
        v_lb_features = desiriability['lb'][1]
        v_ub_features = desiriability['ub'][1]
        v_s_features = desiriability['s'][1]

        # Create scalars - Evaluation Metric
```

Валидация VI

```
        v_lb_eval = desiriability['lb'][0]
        v_ub_eval = desiriability['ub'][0]
        v_s_eval = desiriability['s'][0]

        # Calculate desirability for features
        df['desire_features']  = [0 if x > v_ub_features else 1
                                  if x < v_lb_features else
                                      ((x-v_ub_features)/
                                       (v_lb_features-v_ub_features))**
                                      v_s_features
                                  for x in df['feature_size']]

        # Calculate desirability for evaluation metric
        df['desire_eval']  = [0 if x < v_lb_eval else 1
                              if x > v_ub_eval else
                                      ((x-v_lb_eval)/
                                       (v_ub_eval-v_lb_eval))**
                                      v_s_eval
                              for x in df['evaluation_score']]

        # calculate fitness score
        df['fitness_score'] = (df['desire_features'] * df['desire_eval'])**0.5

        # Drop fields
        df = df.drop(columns=['desire_features', 'desire_eval'])

    else:
```

```
        # calculate fitness score
        df['fitness_score'] = df['evaluation_score']

    # Return
    return df
```

```
# Main Optimisation Function
def f_model_optimisation(df,
                         target_var,
                         generations,
                         population,
                         eval_metric,
                         model,
                         kfold=False,
                         hyperparams_multiple = 3,
                         hyperparams = False,
                         desiriability=False,
                         p_crossover=0.8,
                         p_mutate=0.01,
                         hyperparams_increment = 0.1,
                         elitism=False,
                         gens_no_improve = False,
                         initalise = False):
```

Валидация VIII


```
    '''Function uses GA's to choose features and tune hyperparameters'''

    # Print Model Stats
    print('Model Initialisation')

    # --------- Split features and target ---------
    features = df.drop(target_var,axis=1)
    features_name = features.columns
    target = df[target_var]

    # --------- First Generation ---------

    # Generate inital candidate features solutions
    df_pop_cur = f_generate_population(inital_flag=True,
                                       population=population,
                                       features_name=features_name,
                                       p_crossover=p_crossover,
                                       p_mutate=p_mutate,
                                       hyperparams=hyperparams,
                                       hyperparams_increment=hyperparams_increment,
                                       hyperparams_multiple=hyperparams_multiple,
                                       initalise=initalise)

    # Enrich candidate solutions with features
    df_pop_cur = f_population_features(df=df_pop_cur,
                                       features=features,
                                       target=target,
```

Валидация IX


```python
                                    desiriability=desiriability,
                                    eval_metric=eval_metric,
                                    model=model,
                                    kfold=kfold,
                                    hyperparams=hyperparams
                                    )

    # Extract best score for each candidate
    df_pop_cur = df_pop_cur.loc[df_pop_cur.reset_index().\
                               groupby(['candidate'])['fitness_score'].\
                                  idxmax()]

    # Enrich candidate solutions with similarity & probability
    df_pop_cur = f_sim_n_prob(df_pop_cur)

    # ---------- Create search storage ----------
    df_output = df_pop_cur.copy()

    # Print Model Stats
    print('Gen: 00' +
          ' - Generation Mean:' + str(round(df_output.fitness_score.mean(), 4)).zfill(4) +
          ' - Generation Best:' + str(round(df_output.fitness_score.max(), 4)).zfill(4) +
          ' - Global Best:' + str(round(df_output.fitness_score.max(), 4)).zfill(4)
          )

    # Track best solution
    if gens_no_improve != False:
```

Валидация X

```
    count = 0
    v_best = df_output.fitness_score.max()

# --------- Run additional generations ---------

# Loop for additional generations
for gen in range(1, generations):

    # --------- Elitism ---------
    if elitism > 0:

        # Create a dataframe with elite candidates
        df_elite = df_output.nlargest(columns='fitness_score', n=elitism)
        df_elite['candidate'] = population - 1
        df_elite['generation'] = gen
        df_elite = df_elite.drop(columns=['similarity_features',
                                          'similarity_hyperparameters', 'probability'])
    # --------- New Population ---------

    # Generate next candidate solutions
    df_pop_cur = f_generate_population(inital_flag=False,
                                       generation = gen,
                                       population=(population-elitism),
                                       features_name=features_name,
                                       df=df_pop_cur,
                                       p_crossover=p_crossover,
                                       p_mutate=p_mutate,
```

```
                                       hyperparams=hyperparams,
                                       hyperparams_increment=hyperparams_increment,
                                       hyperparams_multiple=hyperparams_multiple
                                       )

    # Enrich candidate solutions with features
    df_pop_cur = f_population_features(df=df_pop_cur,
                                       features=features,
                                       target=target,
                                       desiriability=desiriability,
                                       eval_metric=eval_metric,
                                       model=model,
                                       kfold=kfold,
                                       hyperparams=hyperparams)

    # Add elite
    if elitism > 0:
        df_pop_cur = pd.concat([df_pop_cur, df_elite]).reset_index().drop(columns=['index'])
        del df_elite

    # Extract best score for each candidate
    df_pop_cur = df_pop_cur.loc[df_pop_cur.reset_index().\
                                groupby(['candidate'])['fitness_score'].\
                                idxmax()]

    # Enrich candidate solutions with similarity & probability
    df_pop_cur = f_sim_n_prob(df=df_pop_cur)
```

Валидация XII


```python

        # Update Output
        df_output = df_output.append(df_pop_cur,ignore_index=True)

        # Print Model Stats
        print('Gen: ' + str(gen).zfill(2) +
              ' - Generation Mean:' + str(round(df_output[df_output.generation == gen].fitness_score.mean(), 4)).zfill(4) +
              ' - Generation Best:' + str(round(df_output[df_output.generation == gen].fitness_score.max(), 4)).zfill(4) +
              ' - Global Best:' + str(round(df_output.fitness_score.max(), 4)).zfill(4)
              )

        # Track number of generations with no improvement
        if gens_no_improve != False:
            if df_output.fitness_score.max() > v_best:
                count = 0
                v_best = df_output.fitness_score.max()
            else:
                count += 1

            # Conditionally break loop
            if count == gens_no_improve:
                break

    # Return df
    return df_output
```

Валидация XIII

Содержание

Оптимизация модели I

Применим следующие модели к нашему процессу оптимизации:

- ElasticNet
- XGBoost

ElasticNet I


```
# Run Optimisation - Optimise for AUC
df_ENet_AUC = f_model_optimisation(df=df_train,
                                   target_var='TARGET',
                                   generations=7,
                                   population=20,
                                   p_crossover=0.8,
                                   p_mutate=0.02,
                                   hyperparams_increment=0.01,
                                   hyperparams_multiple = 5,
                                   eval_metric='roc_auc',
                                   kfold=False,
                                   model=ElasticNet(),
                                   hyperparams = {'names':['alpha', 'l1_ratio'],
                                                  'min_value': [0, 0],
                                                  'max_value': [0.01, 1],
                                                  'type':['float', 'float']}
                                  )
```

ElasticNet II

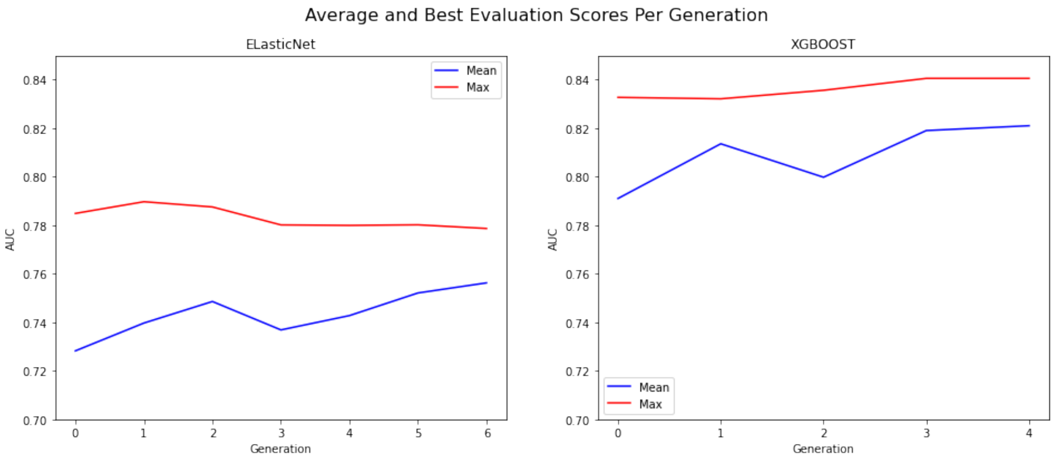

```
Model Initialisation
Gen: 00 - Generation Mean:0.7282 - Generation Best:0.7849 - Global Best:0.7849
Gen: 01 - Generation Mean:0.7397 - Generation Best:0.7897 - Global Best:0.7897
Gen: 02 - Generation Mean:0.7486 - Generation Best:0.7875 - Global Best:0.7897
Gen: 03 - Generation Mean:0.7369 - Generation Best:0.7801 - Global Best:0.7897
Gen: 04 - Generation Mean:0.7428 - Generation Best:0.7799 - Global Best:0.7897
Gen: 05 - Generation Mean:0.7521 - Generation Best:0.7802 - Global Best:0.7897
Gen: 06 - Generation Mean:0.7563 - Generation Best:0.7787 - Global Best:0.7897
```

```
# Run Optimisation - Optimise for AUC
df_xgb_AUC = f_model_optimisation(df=df_train,
                                  target_var='TARGET',
                                  generations=5,
                                  population=20,
                                  p_crossover=0.8,
                                  p_mutate=0.02,
                                  hyperparams_increment=0.01,
                                  hyperparams_multiple = 5,
                                  eval_metric='roc_auc',
                                  kfold=False,
                                  model=XGBClassifier(objective="binary:logistic", scale_pos_weight = 25),
                                  hyperparams = {'names':['learning_rate', 'max_depth',
                                                          'min_child_weight', 'gamma', 'colsample_bytree'],
                                                 'min_value': [0.03, 2,  1, 0,   0.3],
                                                 'max_value': [0.3,  15, 7, 0.5, 0.7],
                                                 'type':['float', 'int', 'int', 'float', 'float']}
                                 )
```

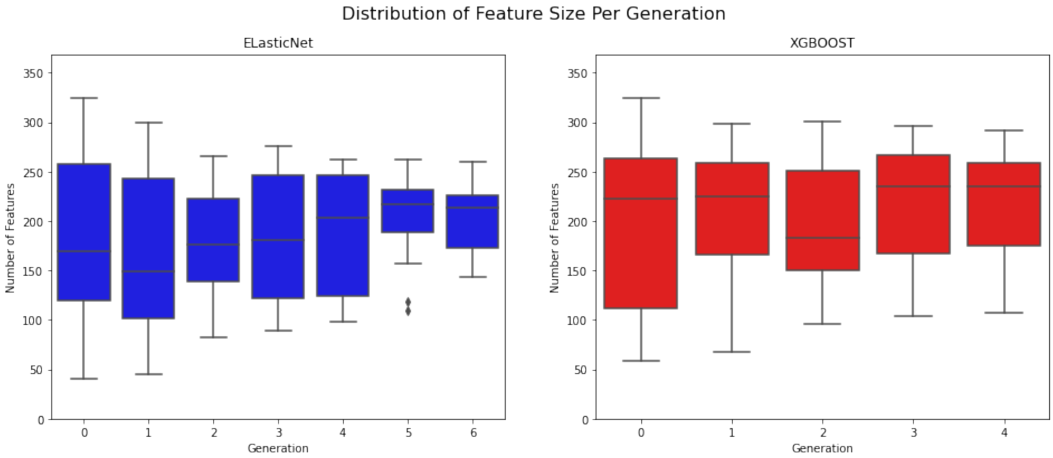
```
Model Initialisation
Gen: 00 - Generation Mean:0.791 - Generation Best:0.8327 - Global Best:0.8327
Gen: 01 - Generation Mean:0.8136 - Generation Best:0.8321 - Global Best:0.8327
Gen: 02 - Generation Mean:0.7998 - Generation Best:0.8356 - Global Best:0.8356
Gen: 03 - Generation Mean:0.819 - Generation Best:0.8405 - Global Best:0.8405
Gen: 04 - Generation Mean:0.821 - Generation Best:0.8405 - Global Best:0.8405
```

Содержание

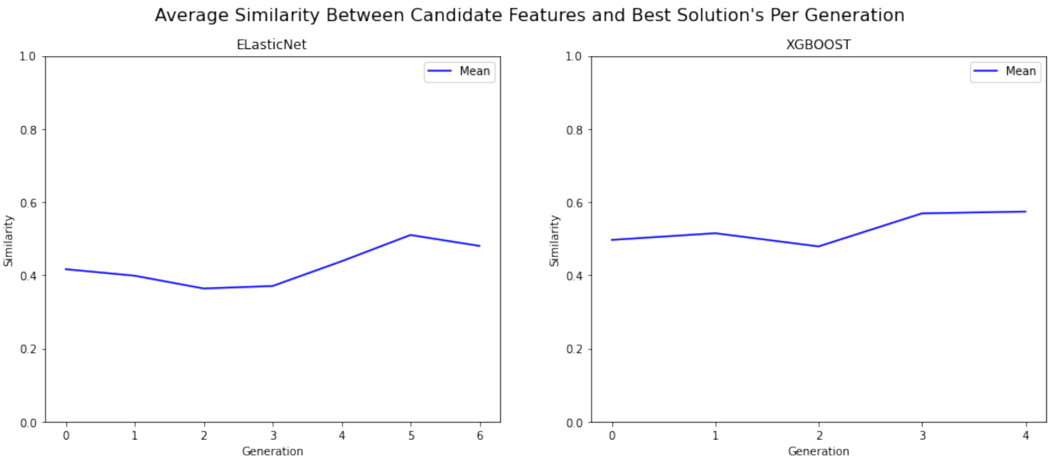
Анализ результатов I



Анализ результатов II



Анализ результатов III



Анализ результатов IV