Grammatical Evolution for Spaceship Titanic

DARSHAN DINNI
LIN YUAN CHONG

TABLE OF CONTENTS

INTRODUCTION

The Spaceship Titanic is a Machine Learning competition currently running on Kaggle with the participation of more than 2,000 teams.

Problem Definition: Spaceship Titanic

Welcome to the year 2912, where your data science skills are needed to solve a cosmic mystery. We've received a transmission from four lightyears away and things aren't looking good.

The spaceship Titanic was an interstellar passenger liner launched a month ago. With almost 13,000 passengers on board, the vessel set out on its maiden voyage transporting emigrants from our solar system to three newly habitable exoplanets orbiting nearby stars.

While rounding Alpha Centauri en route to its first destination—the torrid 55 Cancri E—the unwary Spaceship Titanic collided with a spacetime anomaly hidden within a dust cloud. Sadly, it met a similar fate as its namesake from 1000 years before. Though the ship stayed intact, almost half of the passengers were transported to an alternate dimension!

To help rescue crews and retrieve the lost passengers, you are challenged to predict which passengers were transported by the anomaly using records recovered from the spaceship’s damaged computer system.

Help save them and change history.

DATA

To help the prediction process, we are provided with a set of personal records recovered from the ship's damaged computer system. Two files are provided to help us work through the project.

1. **spaceshipTitanic_train.csv:**
   The training dataset.
2. **spaceshipTitanic_test.csv:**
   The testing dataset.

A total of 13 different data fields are included in the dataset. In this section, we will be explaining the dataset by describing what each data field indicates.

	PassengerId	HomePlanet	CryoSleep	Destination	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Name	Transported
0	0	Earth	False	55 Cancri e	22	False	0	833	381	0	12	Miranda Pratt	True
1	1	Mars	True	TRAPPIST-1e	61	False	0	0	0	0	0	Isaac Werner	True
2	2	Mars	True	TRAPPIST-1e	5	False	0	0	0	0	0	Elisha Rosario	True
3	3	Earth	False	55 Cancri e	14	False	653	0	4	0	0	Deshawn Hall	False
4	4	Earth	False	PSO J318.5-22	2	False	0	0	0	0	0	Justice Archer	True

Figure 1

1. **PassengerId:**
   A unique Id for each passenger.
2. **HomePlanet:**
   The planet the passenger departed from, is typically their planet of permanent residence.
3. **CryoSleep:**
   Indicates whether the passenger elected to be put into suspended animation for the duration of the voyage.
4. **Destination:**
   The planet the passenger will be debarking to.
5. **Age:**
   The age of the passenger.
6. **VIP:**
   Whether the passenger has paid for special VIP service during the voyage.
7. **RoomService, FoodCourt, ShoppingMall, Spa, VRDeck:**
   The amount the passenger has billed at each of the Spaceship Titanic's many luxury amenities.
8. **Name:**
   The first and last names of the passengers.
9. **Transported:**
   Whether the passenger was transported to another dimension. This is the target, the column you are trying to predict.

APPROACH

From the problem definition, it is clear that we have a classification problem i.e. we have to classify if the passengers were transported or not based on the input data given.

We are going to use grammatical evolution as our approach to solving this problem at hand. Taking a step-by-step approach to data preprocessing, applying grammatical evolution to create and evolve equations based on the training data, and predicting if a passenger is transported or not, on the best individual generated.

For this section, we will be breaking down the training process into two major parts: Preprocessing and the Grammatical Evolution itself.

Preprocessing

Before the training and prediction process, the dataset has to be reviewed and processed to ensure maximum efficiency. In this section, we will be describing the preprocessing methodologies we have adopted. These methodologies are classified into two categories to provide a more accurate explanation: Basic Preprocessing and Advanced Preprocessing.

- ### Basic Preprocessing

  Before we carry out any preprocessing, we ensure that the dataset is complete and without any empty cells. We do so using the isna() and isnull() functions.

```
# Checking training data has NaN values and null values

print('Is NaN count for all columns:\n', data.isna().sum())
print('\n Is Null count for all columns\n', data.isnull().sum())
```

Figure 2

According to the printing results, it appears that the dataset provided is complete, and no cells are filled with either “NaN” or “Null”.

```
Is NaN count for all columns:
 PassengerId     0
HomePlanet       0
CryoSleep        0
Destination      0
Age              0
VIP              0
RoomService      0
FoodCourt        0
ShoppingMall     0
Spa              0
VRDeck           0
Name             0
Transported      0
dtype: int64
```

Figure 3

```
 Is Null count for all columns
 PassengerId     0
HomePlanet       0
CryoSleep        0
Destination      0
Age              0
VIP              0
RoomService      0
FoodCourt        0
ShoppingMall     0
Spa              0
VRDeck           0
Name             0
Transported      0
dtype: int64
```

Figure 4

After checking the dataset, we complete our basic preprocessing by dropping features that we agree to have no insights into helping the training, testing, and prediction process.

```
# Drop the 'PassengerId' and 'Name' feature that does not really give much

data_preprocessing = data.drop(['PassengerId', 'Name'], axis=1)
test_data_preprocessing = test_data.drop(['PassengerId', 'Name'], axis=1)
```

Figure 5

- ## Advanced Preprocessing

After completing the two basic preprocessing steps, we move on to carry out advanced preprocessing on the dataset. This stage involves steps that change the values of data fields. Before performing any advanced preprocessing tasks, we separated the features into two categories: numerical and categorical. Features of each of these categories undergo different preprocessing methods.

The numerical features of the dataset are: “Age”, “RoomService”, “FoodCourt”, “ShoppingMall”, “Spa”, and “VRDeck”. As these values are all non-uniformed and differently scaled, it is best to rescale all of them into the same ratio to maximize training efficiency. We achieved this by using the StandardScaler() method to scale all values from a range from -1 to 1. The new values are then appended to a new dataset in order to avoid overwriting the original dataset. The figure below shows the code of the explained procedures.

```
[12]
# Standardize the numerical features for train and test data

# Creating list of all the numerical features to standardize
numericals_list = ['Age','RoomService', 'FoodCourt', 'ShoppingMall', 'Spa', 'VRDeck']

# Iterating through all the columns to standardize
for column in numericals_list:
    std_scaler = StandardScaler(with_mean=True, with_std=True)
    std_scaler.fit(data_preprocessing[column].values.reshape(-1,1))
    data_preprocessing[column] = std_scaler.transform(data_preprocessing[column].values.reshape(-1,1))
    test_data_preprocessing[column] = std_scaler.transform(test_data_preprocessing[column].values.reshape(-1,1))
```

Figure 6

The next step is to encode categorical features into numbers. This is essential as the training and testing processes function more efficiently with numerical data. These categorical features are: “CryoSleep”, “VIP”, “Transported”, “HomePlanet”, and “Destination”. As most of these categorical data feature only two types of values, converting them to either “0”s or “1”s will be sufficient. This can be done by using LabelEncoder().

```
[14]  1 # Encoding categorical features for train and test data
      2
      3 # Creating list of all the categorical features to encode
      4 categorical_classes_list = ['CryoSleep', 'VIP', 'Transported', 'HomePlanet','Destination']
      5
      6 # Iterating through all the columns to encode different category
      7 for column in categorical_classes_list:
      8     label_encoder = LabelEncoder()
      9     label_encoder.fit(data_preprocessing[column])
     10     data_preprocessing[column] = label_encoder.transform(data_preprocessing[column])
     11     if column != 'Transported':
     12       test_data_preprocessing[column] = label_encoder.transform(test_data_preprocessing[column])
```

Figure 7

However, the two columns “HomePlanet” and “Destination” include more than two categories, using LabelEncoder() as a preprocessing can have a disadvantage i.e. for any machine learning problem higher numbers are marked as highly important which can affect the learning process. So, to avoid this we have used a specific method that is needed to handle values of this type, known as One Hot Encoding. This process converts each type of value of a categorical feature into new binary variables along with a unique integer as a boolean value. The figure below shows One Hot Encoding done on the two features “HomePlanet” and “Destination”.

```
1 # Apply One hot encoding to categorical features for train and test data
2
3 data_preprocessing = pd.get_dummies(data_preprocessing, columns=['HomePlanet','Destination'])
4 test_data_preprocessing = pd.get_dummies(test_data_preprocessing, columns=['HomePlanet','Destination'])
```

Figure 8

The data is then finally fully processed. The figure below displays samples of the processed dataset:

```
[16] 1 data_preprocessing.sample(10)
```

	CryoSleep	Age	VIP	RoomService	FoodCourt	ShoppingMall	Spa	VRDeck	Transported	HomePlanet_0	HomePlanet_1	HomePlanet_2	Destination_0	Destination_1	Destination_2
720	0	-0.174730	0	-0.346747	-0.265996	-0.326312	0.337130	-0.263720	0	1	0	0	0	0	1
1917	0	-0.379851	0	-0.341874	0.028419	-0.253684	-0.276866	-0.251965	1	1	0	0	0	1	0
1557	1	1.466245	0	-0.346747	-0.282447	-0.326312	-0.276866	-0.263720	1	1	0	0	0	0	1
1836	1	-0.858469	0	-0.346747	-0.282447	-0.326312	-0.276866	-0.263720	1	1	0	0	0	0	1
1369	1	-0.926843	0	-0.346747	-0.282447	-0.326312	-0.276866	-0.263720	1	0	1	0	0	0	1
1249	1	-1.405461	0	-0.346747	-0.282447	-0.326312	-0.276866	-0.263720	0	1	0	0	0	0	1
382	0	-0.106356	0	0.049609	-0.282447	0.639451	-0.036607	-0.241190	1	0	0	1	0	0	1
1900	0	-0.516599	0	2.814353	-0.261458	-0.326312	-0.276866	-0.240210	0	1	0	0	1	0	0
1234	1	1.534619	0	-0.346747	-0.282447	-0.326312	-0.276866	-0.263720	1	0	1	0	0	0	1

Figure 9

We then performed train_test_split() on the processed data. X contains everything needed to predict the y label, while y indicates all the "Transported" labels, which are the predicted labels. The dataset is now ready for training and testing.

```
1 processed_data = data_preprocessing.copy()
2 X = processed_data.drop(['Transported'], axis=1).values
3 y = processed_data['Transported'].values
4
5 from sklearn.model_selection import train_test_split
6
7 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

Figure 10

Grammatical Evolution

Grammatical evolution is a grammar-based version of genetic programming. It revolves around the idea of a genotype to phenotype mapping, with an associated grammar, which specifies the syntax of the resulting solution individuals.

These codons are used to choose productions from the given grammar, generating syntactically correct solutions for individuals.

The phenotypes can be evaluated, and their resulting fitness is returned to the search engine.

- ### Initialization

  For initialization, we have used the Sensible initialization (SI) approach. We start by noting the minimum depth required to reach a string consisting of terminal symbols only, starting with its associated non-terminal symbols.

  Once this labeling is done, SI works in the same way as Ramped-Half-and-Half, applying to Grow and Full to create derivation trees from the start symbol, until only terminal symbols exist as leaves, only productions whose minimum depths lead to a branch depth less than or equal to the maximum depth specified are selected.

  - **Grammar**

    The purpose of grammar is to define the structures and syntactic properties of the language used to express solutions to a given problem. Below is the grammar that we used for the Spaceship Titanic problem.

    <log_op> ::= <conditional_branches> | and_(<log_op>,<log_op>) | or_(<log_op>,<log_op>) | not_(<log_op>) | <boolean_feature>
    <c> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    <conditional_branches> ::= less_than_or_equal(<num_op>,<num_op>) | greater_than_or_equal(<num_op>, <num_op>)
    <num_op> ::= add(<num_op>,<num_op>) | sub(<num_op>,<num_op>) | mul(<num_op>,<num_op>) | pdiv(<num_op>,<num_op>) | <nonboolean_feature>
    <boolean_feature> ::= x[0]|x[2]|x[8]|x[9]|x[10]|x[11]|x[12]|x[13]
    <nonboolean_feature> ::= x[1]|x[3]|x[4]|x[5]|x[6]|x[7]

- **Parameters**

  Following are the list of hyperparameters used, which are required for the grammatical evolution.

  - ➢ **population_size**: The size of the population of individuals.
  - ➢ **max_generation:** Maximum number of generations.
  - ➢ **p_crossover**: The probability of crossover.
  - ➢ **p_mutation**: The probability of mutation.
  - ➢ **elite_size**: The number of elite individuals.
  - ➢ **halloffame_size**: Object to register the elite individuals.
  - ➢ **tournament _size:** Size for the tournament selection.
  - ➢ **max_tree_depth**: Specifying the maximum tree depth.
  - ➢ **min_tree_depth**: Specifying the minimum tree depth.
  - ➢ **toolbox**: Object of the module base, where the fitness function, the selection method, the crossover operator, and the mutation operator were registered.
  - ➢ **training data**: Samples of the training set.
  - ➢ **testing data**: Samples of the test set.
  - ➢ **stats**: Object to register the statistics (optional parameter).
  - ➢ **Verbose**: If True, the report will be printed on the screen each generation.

● Crossover and Mutation

We have used a one-point crossover (“ripple crossover”) that exchanges codons between two genotypes. The resulting genotypes are then mapped to their respective phenotypes using a Backus-Naur form of grammar.

The one-point crossover starts with selecting two individuals randomly in a population. Then a crossover point in each of the genotypes is selected randomly and independently for both individuals. Then, the codons on the right side of the crossover points are exchanged between the individuals.

Below is an example of how a one-point crossover is performed.

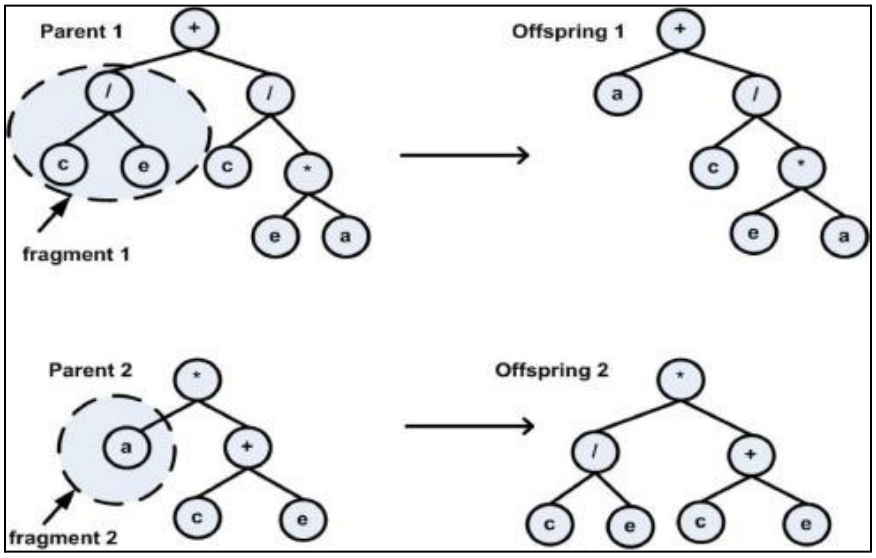

Figure 11

The mutation method implemented here is the mutation_int_flip_per_codon method. This method gives each bit of an individual an equal chance of flipping its value according to the given mutation probability, for this case, 0.05. The figure below is a visualization of how this mutation method works. Multiple bits of the offspring undergo mutation and have their values flipped.

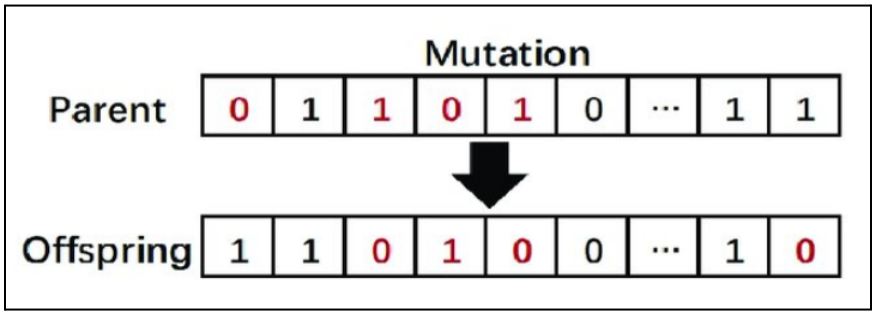

Figure 12

- ## Selection

  Selection is the process in genetic algorithms where individual genomes are chosen from a population for later reproduction.

  For the Spaceship Titanic problem, we are using Tournament Selection as a selection algorithm. Tournament Selection is used for selecting the fittest candidates from the current generation. These selected candidates are then passed on to the next generation.

  In a k-tournament selection, 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. In this way many such tournaments take place and we have our final selection of candidates who move on to the next generation.

  The figure below shows how a single iteration of k-size tournament selection is performed.

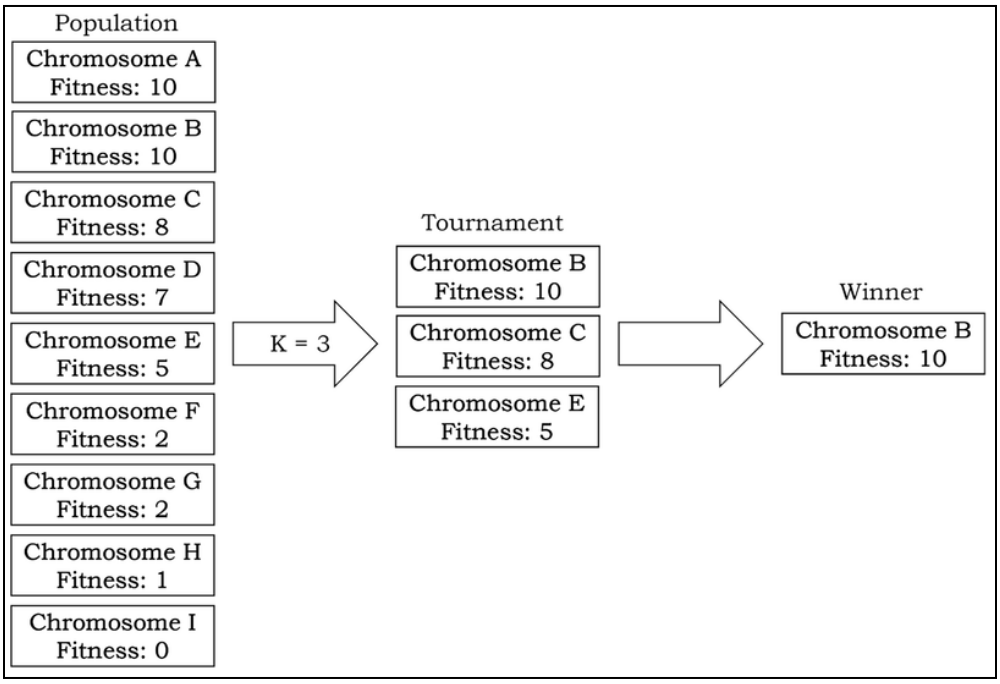

Figure 13

Once the first generation starts the initial population is evaluated with the fitness function, and then the hall of fame object is updated according to the best individuals found in the population for that generation.

Next, the stats and results are calculated and recorded in the logbook respectively. After the initial generation ends the algorithm starts to run the remaining generations.

The first step is selecting parents according to the method registered in the toolbox. There are selected individuals as parents for each generation. These individuals are operated by crossover and mutation, considering their respective probabilities specified.

Regarding the crossover, all individuals are considered pairwise, according to the probability of each pair being operated on or not, and the point to perform the crossover in each individual is chosen at random within the effective length of the genomes. Before finishing the crossover step, the offspring is mapped to get its current effective length before the next operation.

Concerning the mutation, all individuals after the crossover procedure are considered individually. Each codon within the effective length of the genomes is mutated or not, according to the mutation probability. Finally, the offspring are generated and evaluated with the fitness function.

Next, the population is replaced with the offspring, the hall of fame object is updated, and the stats and results are calculated and recorded in the logbook respectively. If the last generation was achieved, the process ends, and the best individual is evaluated with the fitness function. Otherwise, the loop continues.

We take the best individual which is evaluated from the training, and use it to predict the class for the test data.

RESULTS

GE offers a flexible yet powerful framework for automatic program generation. The syntax and the structure of the programs are described using context-free grammar, and their objective is determined by the fitness function. An evolutionary search is performed on the grammar to find the program that minimizes the errors.

By using the grammatical evolution we were able to achieve the best individual and best fitness over population sizes of 1000 over 500 generations. Following are the results achieved.

- **Best individual:**
  not_(or_(not_(greater_than_or_equal(x[4],x[3])),not_(less_than_or_equal(add(x[6],x[7]), x[4]))))
- **Training Fitness:**
  0.20671641791044781
- **Depth:**
  9
- **Length of the genome:**
  86
- **Used portion of the genome:**
  0.22
- **Accuracy on the final test data:**
  0.79338