Data Analysis In “Game of Thrones”

Leyi Wang	001493704
Xiao Liu	001472735
Yan Lyu	001493900
Yitong Liu	001494304

1.Introduction

In this project, we are going to analysis the story of Game of Throne -- a popular TV series around the world and finding some interesting conclusions in python using machine learning. With data preprocessing/ different models (including linear regression , Random Forest, Decision Tree, KNN, SVC, DNB, GradientBoostin and AdaBoostClassifier) training and comparing the accuracy score of each analysis result, finally, we will figure out who acted as important characters in this series and the relationship between the character death and battles.

2.Background

"Game of Thrones" is a wonderful TV series adapted from the novel "Song of Ice and Fire" of George RR Marti. These TV series focus on the struggles of the nine families, with its complex political landscape, a large number of characters and deaths of characters. "Game of Thrones" contains thousands of characters with different performances and personalities.

However, due to cultural, geographical and linguistic differences, the audience who first watched Game of Thrones (such as us) often could not figure out different characters and could not understand the background of the story. After analyzing the dataset of Game of Thrones in Python and coming to some general conclusions, the new audience can have a better understanding of this TV series.

Moreover, as a big fan, we can also get a deeper understanding of the play from the following analysis. For example, in order to establish different models to predict the death of the characters, we can have an experience as a scriptwriter; in order to analyze different battles according to the built model, we can further understand the motivation of the characters in the play. In this way, completing such a project is a perfect combination of interest and learning.

3. Objectives

Analyze the battles in “Game of Thornes”

Demonstrate the character relationship of the characters in “Game of Thornes.

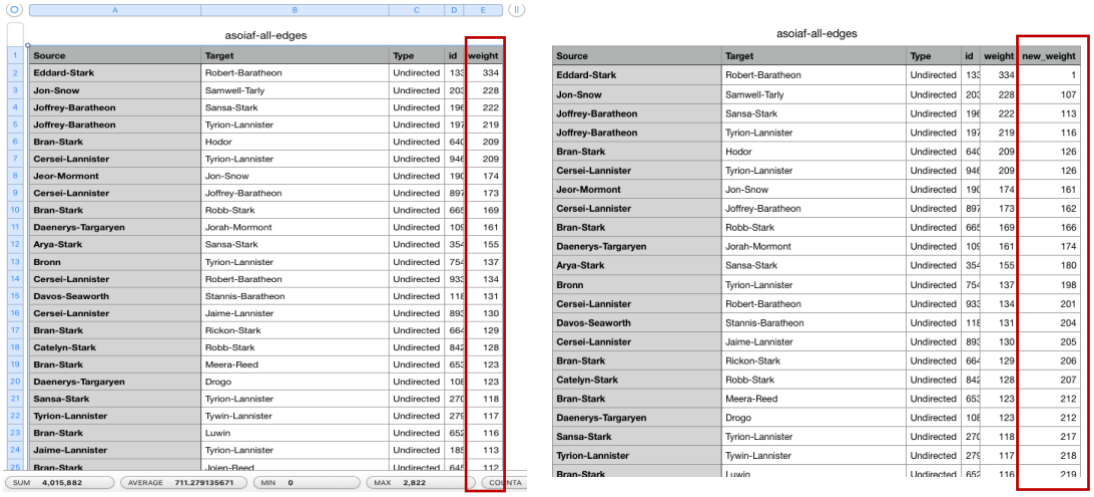
Predict the death of a character in “Game of Thornes”

4. Methodologies

4.1 Data Preprocessing

We use data mainly for analyzing the battles and characters in the series and predict the death of a character. For the first analysis part, we firstly analyze the battles using battle.csv file without data preprocessing and for the second analysis part, we need to use the betweenness centrality idea to study the importance of each character. The basis for calculating the betweenness centrality is the shortest path, which means that the shorter path is, the more frequent communication between the two roles is. If we need to use the new weight to represent this idea, we have to preprocess the data in asoiaf-all-edges.csv from book1 to book5.

$$\text{New\_weight} = \text{Max\_Weight} + 1 - \text{weight}$$



asoiaf-all-edges

Source	Target	Type	id	weight
Eddard-Stark	Robert-Baratheon	Undirected	133	334
Jon-Snow	Samwell-Tarly	Undirected	203	228
Joffrey-Baratheon	Sansa-Stark	Undirected	196	222
Joffrey-Baratheon	Tyrion-Lannister	Undirected	197	219
Bran-Stark	Hodor	Undirected	640	209
Cersei-Lannister	Tyrion-Lannister	Undirected	946	209
Jeor-Mormont	Jon-Snow	Undirected	190	174
Cersei-Lannister	Joffrey-Baratheon	Undirected	897	173
Bran-Stark	Robb-Stark	Undirected	665	169
Daenerys-Targaryen	Jorah-Mormont	Undirected	109	161
Arya-Stark	Sansa-Stark	Undirected	354	155
Bronn	Tyrion-Lannister	Undirected	754	137
Cersei-Lannister	Robert-Baratheon	Undirected	933	134
Davos-Seaworth	Stannis-Baratheon	Undirected	118	131
Cersei-Lannister	Jaime-Lannister	Undirected	893	130
Bran-Stark	Rickon-Stark	Undirected	664	129
Catelyn-Stark	Robb-Stark	Undirected	842	128
Bran-Stark	Meera-Reed	Undirected	653	123
Daenerys-Targaryen	Drogo	Undirected	108	123
Sansa-Stark	Tyrion-Lannister	Undirected	270	118
Tyrion-Lannister	Tywin-Lannister	Undirected	279	117
Bran-Stark	Luwin	Undirected	652	116
Jaime-Lannister	Tyrion-Lannister	Undirected	185	113
Bran-Stark	Jojen-Reed	Undirected	645	112

SUM 4,015,882 AVERAGE 711.279135671 MIN 0 MAX 2,822 COUNTA

asoiaf-all-edges

Source	Target	Type	id	weight	new_weight
Eddard-Stark	Robert-Baratheon	Undirected	133	334	1
Jon-Snow	Samwell-Tarly	Undirected	203	228	107
Joffrey-Baratheon	Sansa-Stark	Undirected	196	222	113
Joffrey-Baratheon	Tyrion-Lannister	Undirected	197	219	116
Bran-Stark	Hodor	Undirected	640	209	126
Cersei-Lannister	Tyrion-Lannister	Undirected	946	209	126
Jeor-Mormont	Jon-Snow	Undirected	190	174	161
Cersei-Lannister	Joffrey-Baratheon	Undirected	897	173	162
Bran-Stark	Robb-Stark	Undirected	665	169	166
Daenerys-Targaryen	Jorah-Mormont	Undirected	109	161	174
Arya-Stark	Sansa-Stark	Undirected	354	155	180
Bronn	Tyrion-Lannister	Undirected	754	137	198
Cersei-Lannister	Robert-Baratheon	Undirected	933	134	201
Davos-Seaworth	Stannis-Baratheon	Undirected	118	131	204
Cersei-Lannister	Jaime-Lannister	Undirected	893	130	205
Bran-Stark	Rickon-Stark	Undirected	664	129	206
Catelyn-Stark	Robb-Stark	Undirected	842	128	207
Bran-Stark	Meera-Reed	Undirected	653	123	212
Daenerys-Targaryen	Drogo	Undirected	108	123	212
Sansa-Stark	Tyrion-Lannister	Undirected	270	118	217
Tyrion-Lannister	Tywin-Lannister	Undirected	279	117	218
Bran-Stark	Luwin	Undirected	652	116	219

Figure 1. Preprocessed data in asoiaf-all-edges.csv

Data preprocessing for battles

Data source: battles.csv

The original data has a total of 39 rows and 25 columns. The processing of the raw data will now be described.

We firstly remove duplicate columns and columns that don't make sense. And we fill in the missing values.

```
missing_df = dataset_battles.isnull().sum(axis=0).reset_index()
missing_df.columns = ['variable', 'missing values']
missing_df['filling percentage']=(dataset_battles.shape[0]-missing_df['missing values'])/dataset_battles.
missing_df.sort_values('filling percentage').reset_index(drop = True)
```

Figure 2. Coding to fill in the missing values

The results as below are arranged in descending order of missing values (filling rate from small to large). The figure shows: ‘attacker_4’, ‘defender_2’, ‘attacker_3’ has the most missing values, and the filling rate is less than 10%. We must choose the appropriate strategy to fill in the missing values and it is not advisable to directly delete samples with missing values.

	variable	missing values	filling percentage
0	attacker_4	36	5.263158
1	defender_2	36	5.263158
2	attacker_3	35	7.894737
3	note	33	13.157895

Figure 3. Result of filling in missing values

Secondly, we change to another strategy. When filling in missing values, string type data is filled with "no", and numeric type data is filled with "0". After the fill is complete, there are no more missing values in the dataset:

12	defender_1	0	100.0
13	attacker_4	0	100.0
14	attacker_3	0	100.0
15	attacker_2	0	100.0
16	attacker_1	0	100.0
17	defender_king	0	100.0
18	attacker_king	0	100.0

Figure 4. Result of another strategy

In the current dataset, the "attacker_outcome" attribute indicates whether the character is still alive now, which is what we are going to predict and is declared as Y. In addition to this column of data, the other columns are divided into X.

Then, we classify the data. The purpose of classifying data is to convert data that is not a numeric type into numbers. The basic strategy is to divide the data of each column into different types and use the numbers to represent each type. sklearn.preprocessing.LabelEncoder is used to achieve it.

```
from sklearn.preprocessing import LabelEncoder,OneHotEncoder
labelEncoder_X = LabelEncoder()
X[:,0] = labelEncoder_X.fit_transform(X[:,0])
X[:,3] = labelEncoder_X.fit_transform(X[:,3])
X[:,4] = labelEncoder_X.fit_transform(X[:,4])
X[:,5] = labelEncoder_X.fit_transform(X[:,5])
X[:,6] = labelEncoder_X.fit_transform(X[:,6])
X[:,7] = labelEncoder_X.fit_transform(X[:,7])
X[:,8] = labelEncoder_X.fit_transform(X[:,8])
X[:,9] = labelEncoder_X.fit_transform(X[:,9])
X[:,10] = labelEncoder_X.fit_transform(X[:,10])
X[:,11] = labelEncoder_X.fit_transform(X[:,11])
X[:,12] = labelEncoder_X.fit_transform(X[:,12])
X[:,17] = labelEncoder_X.fit_transform(X[:,17])
X[:,18] = labelEncoder_X.fit_transform(X[:,18])
X[:,20] = labelEncoder_X.fit_transform(X[:,20])
X[:,21] = labelEncoder_X.fit_transform(X[:,21])
```

Figure 5. Classify data

From the *Figure 5*, we can find that some features have values ranging between 0 and 1, and some features have values ranging from 0 to 1500, which is very disadvantageous for establishing an effective model.

For the fourth step, we do a power-of-square operation, features with larger values will have a greater impact on the results, while features with smaller values will have negligible effects on the results. However, in real life, the factors that influence the outcome should be the feature and its value, not just the value. Therefore, we must scale each feature to the same interval. We used sklearn.preprocessing.StandardScaler object to achieve it.

```
array([[ 1.75505122e+00, -1.34030115e-01,  5.49461687e-01,
        -3.20408485e-01,  8.03913879e-01,  1.05021006e-01,
        -2.93987366e+00,  3.12347524e-01,  2.67261242e-01,
         8.00105132e-01,  2.52645576e-01,  4.34524095e-01,
        -1.26687088e+00,  1.22474487e+00,  1.52752523e+00,
        -1.98905380e-01,  9.46914484e-02,  2.05897064e+00,
        -1.88232905e-01,  7.60885910e-01,  1.31748897e+00,
         4.40652649e-01],
       [-1.10845340e+00, -1.47433127e+00, -1.38486772e+00,
        -3.20408485e-01,  8.03913879e-01,  6.30126038e-01,
         5.67930139e-01,  3.12347524e-01,  2.67261242e-01,
         1.10394252e+00,  2.52645576e-01,  4.34524095e-01,
         4.36852028e-02, -8.16496581e-01,  1.52752523e+00,
         4.34393401e-01,  1.43612497e+00, -6.21576042e-01,
        -1.05700016e+00,  7.60885910e-01,  3.07273785e-01,
         4.40652649e-01],
       [-9.23711167e-02,  1.20627104e+00,  1.05406936e+00,
        -1.38843677e+00, -8.78696566e-01,  3.67573522e-01,
         5.67930139e-01,  3.12347524e-01,  2.67261242e-01,
         1.40777992e+00,  2.52645576e-01,  4.34524095e-01,
         4.36852028e-02, -8.16496581e-01, -6.54653671e-01,
        -3.91648487e-01, -6.27618910e-01, -1.08775807e+00,
         1.11491797e+00, -1.31425748e+00,  9.38658275e-01,
        -2.20326325e-01],
```

Figure 5. Result of feather scaling

Finally, we used the sklearn.model_selection.train_test_split object to split training set and test set. Choose 20% of them as the training set and 80% as the test set.

Data preprocessing for predicting the death of characters

Data source: character-predictions.csv

The original data has a total of 1946 rows and 33 columns. The processing of the raw data will now be described.

First of all, we remove duplicate columns and columns that don't make sense and fill in the missing values.

```
missing_df = dataset_death.isnull().sum(axis=0).reset_index()
missing_df.columns = ['variable', 'missing values']
missing_df['filling percentage']=(dataset_death.shape[0]-missing_df['missing values'])/dataset_death.shape[0]*100
missing_df.sort_values('filling percentage').reset_index(drop = True)
```

	variable	missing values	filling percentage
0	mother	1925	1.079137
1	isAliveMother	1925	1.079137
2	heir	1923	1.181912
3	isAliveHeir	1923	1.181912
4	isAliveFather	1920	1.336074
5	father	1920	1.336074
6	isAliveSpouse	1670	14.182939
7	spouse	1670	14.182939
8	age	1513	22.250771
9	culture	1269	34.789311
10	title	1008	48.201439
11	house	427	78.057554
12	isNoble	0	100.000000
13	isMarried	0	100.000000
14	isPopular	0	100.000000

Figure 6. Result of filling the missing values

Figure 6 shows that results are arranged in descending order of missing values (filling rate from small to large). The figure shows: ‘mother’, ‘isAliveMother’ has the most missing values, and the filling rate is only 1%. There are five features with a filling rate of 10% or less, three features with a filling rate of 10% to 40%, and three features with a filling rate of 30% to 99%, for a total of 11 features. Considering that our data set has only 1946 samples, we must choose the appropriate strategy to fill in the missing values and it is not advisable to directly delete samples with missing values.

Secondly, we change another strategy to fill in missing values. String type data is filled with "no", and numeric type data is filled with "-1". After the fill is complete, there are no more missing values in the dataset:

	variable	missing values	filling percentage
0	title	0	100.0
1	isPopular	0	100.0
2	boolDeadRelations	0	100.0
3	numDeadRelations	0	100.0
4	age	0	100.0

Figure 7. Result of another strategy

In the current dataset, the "isAlive" attribute indicates whether the character is still alive now, which is what we are going to predict and is declared as Y. In addition to this column of data, the other columns are divided into X.

Then, we classify the data. The purpose of classifying data is to convert data that is not a numeric type into numbers. The basic strategy is to divide the data of each column into different types and use the numbers to represent each type. sklearn.preprocessing.LabelEncoder object is used to achieve it.

	title	male	culture	mother	father	heir	house	spouse
0	259	1	62	12	8	0	346	254
1	151	1	41	17	20	22	116	184
2	217	1	62	17	20	22	264	254
3	194	0	62	17	20	22	23	241
4	68	0	10	17	20	22	237	74

Figure 8. Result of classifying data

From *figure 8*, we can find that some features have values ranging between 0 and 1, and some features have values ranging from 0 to 350, which is very disadvantageous for establishing an effective model.

For the fourth step, we do the operations such as a power-of-square operation, features with larger values will have a greater impact on the results, while features with smaller values will have negligible effects on the results. However, in real life, the factors that influence the outcome should be the feature and its value, not just the value. Therefore, we must scale each feature to the same interval.

sklearn.preprocessing.StandardScaler is used to achieve it.

```
1 X
```

```
array([[ 0.65897578,  0.78417967,  0.63274963, ...,  3.52429988,
         3.9902054 ,  3.21295684],
       [-0.91664649,  0.78417967, -0.48061727, ...,  3.52429988,
         3.9902054 ,  5.02554666],
       [ 0.04623379,  0.78417967,  0.63274963, ..., -0.2837443 ,
        -0.25061367,  1.10868589],
```

Figure 9. Result of feature scaling

Finally, we used the sklearn.model_selection.train_test_split object to split training set and test set. Choose 25% of them as training set and 75% as test set.

Here is result in the novel (*Figure 10*), a total of 25.44% of the characters died, accounting for a quarter of the total number of characters. However, the death of a character affects the reader's love for the novel. Therefore, it is important to be able to predict whether a character is dead.

```
([<matplotlib.patches.Wedge at 0x1a203e8518>,
  <matplotlib.patches.Wedge at 0x1a203e8eb8>],
 [Text(-0.8600916514117876, -0.8368048465273993, 'alive'),
  Text(0.788417347127472, 0.7670711093167825, 'dead')],
 [Text(-0.5017201299902093, -0.4881361604743162, '74.56%'),
  Text(0.43004582570589384, 0.41840242326369953, '25.44%')])
```

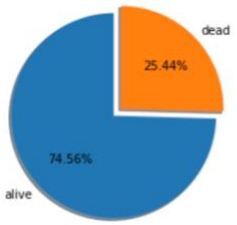


Figure 10. The percentage of character death

4.2 Data Analysis

Before we formally build the model, we need to have a general understanding of what the data is. We take two parts to analyze the dataset.

For the first part, we analyze dataset battles.csv. We mainly around the topic: region and death, battles and death, culture and death, attacker king and defender king and the greatest commanders.

For the second part, we analyze the important characters by calculating the betweenness centrality in 5 books and figure out the top 10 important characters.

4.2.1 The analysis of battles

To predict the death of characters, we guess many factors, such as region, culture and so on. And battles may be an important factor for the characters. To test this idea and have a good understanding of background in "Game of Thrones", we start to have completed analysis of battles.

First of all, we import some useful libraries like Numpy, Pandas, Seaborn, Matplotlib.

Then, we read data from "battles.csv" files that have downloaded. According to the column name "definder_1", "definder_2", "definder_3", "definder_4", we get the sum of defenders of each battle. Also, we get the sum of attackers, the sum of attacker-commanders and the sum of defender-commanders.

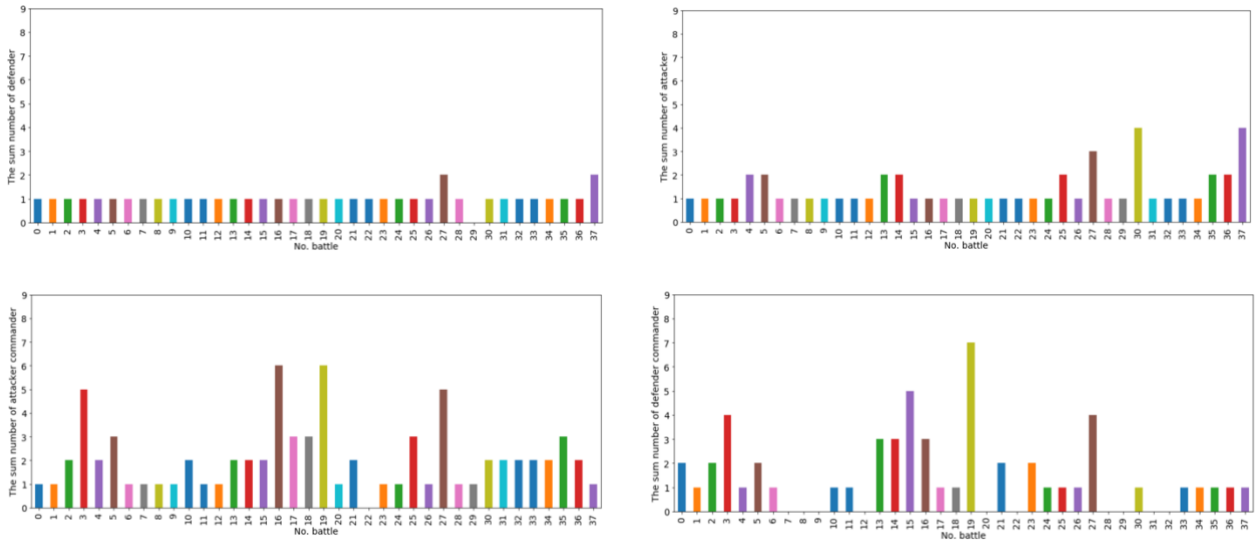


Figure 11. Number of attacker and defender for each battle.

Death and region

We group the major death or the major capture by year. Here are the death and capture information of Year 298, Year 299 and Year 300.

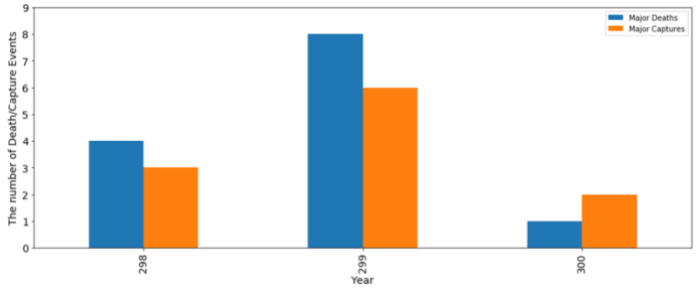


Figure 12. Number of major death or major captures of three years

And we study the regions where battles took place by the "region" column. As the figure shows, the Riverlands area had the largest number of battles.

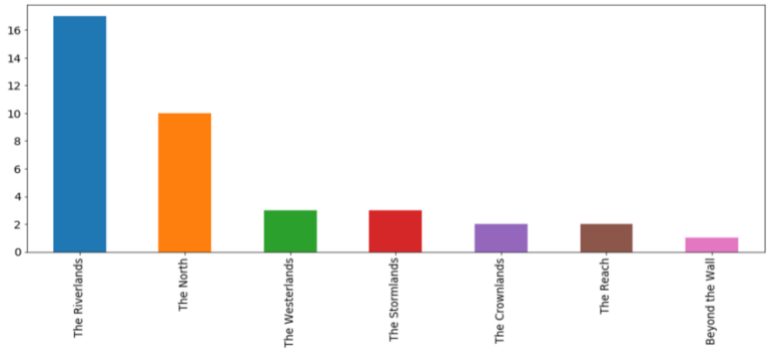


Figure 13. Number of deaths in each region

Battles and death

We get the sum of major death and major captures of each region group by "region". We found that the region, which had a larger number of battles, had a larger number of death.

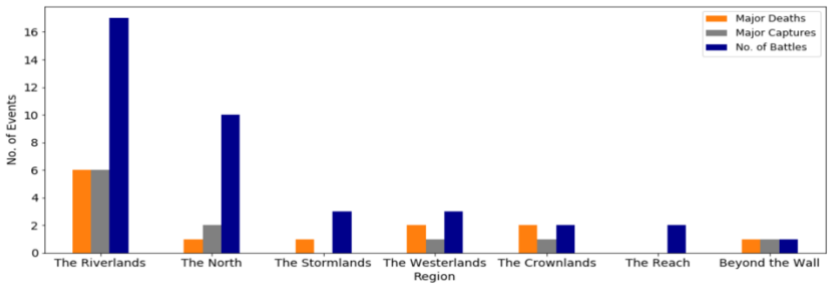


Figure 14. Number of battles in each region

Culture and death

We renamed cultures using domain knowledge as many of the culture values maps to a single culture. Then we got the "culture" column from character_predictions.csv and found the characters whose culture had these keywords. We count the death and survival of these characters within one culture. According to the figure below, we guess that the culture has a strong relationship with the death of characters.

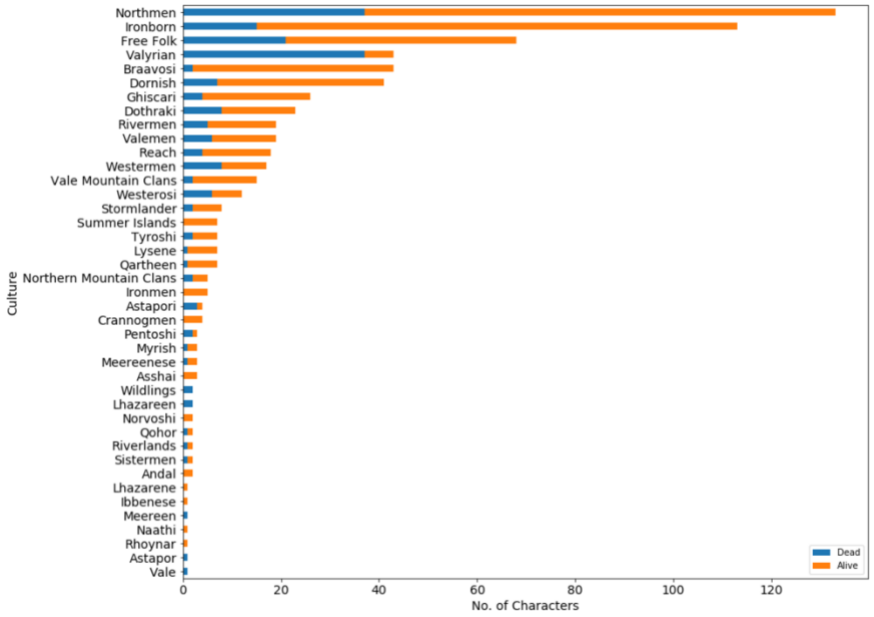


Figure 15. Number of deaths within each culture

King and battles

We selected "battle_type", "attacker_king", "defender_king" and"name" columns and fill the NA value. For each attacker_king, we group the battle by "battle_type" and count the number by "name". And for each "defender_king", we group the battle by "battle_type" and count the number by "name".

```
[['Pitched Battle', 'Ambush', 'Siege', 'Razing', 'Unknown'],
 ('As Attacker King',
  [[6, 3, 5, 0, 0],
   [3, 5, 2, 0, 0],
   [2, 2, 2, 1, 0],
   [2, 0, 2, 0, 1],
   [1, 0, 0, 1, 0],
   [0, 0, 0, 0, 0],
   [0, 0, 0, 0, 0]]),
 ('As Defender King',
  [[4, 5, 2, 1, 1],
   [6, 5, 3, 0, 0],
   [2, 0, 2, 0, 0],
   [0, 0, 2, 0, 0],
   [2, 0, 0, 1, 0],
   [0, 0, 1, 0, 0],
   [0, 0, 1, 0, 0]])]
```

Joffrey/Tommen
Robb Stark
Balon/ Euron Greyjoy
Stannis Baratheon
Unknown
Renly Baratheon
Mance Rayder
Joffrey/Tommen
Robb Stark
Balon/ Euron Greyjoy
Stannis Baratheon
Unknown
Renly Baratheon
Mance Rayder

Figure 16. The number of different type of battles king attended

The data contains four types of battles of which Pitched Battle is the most frequent and Razing the least. Joffrey/ Tommen Baratheon's favorite battle type is "Pitched Battle" and Robb Stark's favorite type of attack seems to be "Ambush". All five of his "Ambush" type battles are against Joffrey/Tommen Baratheon.

Army size and battles

For each battle, we calculate the difference of army size by using "attacker_size" minus "defender_size". According to the figure below, we guess that the battle result has little correlation with the size of army.

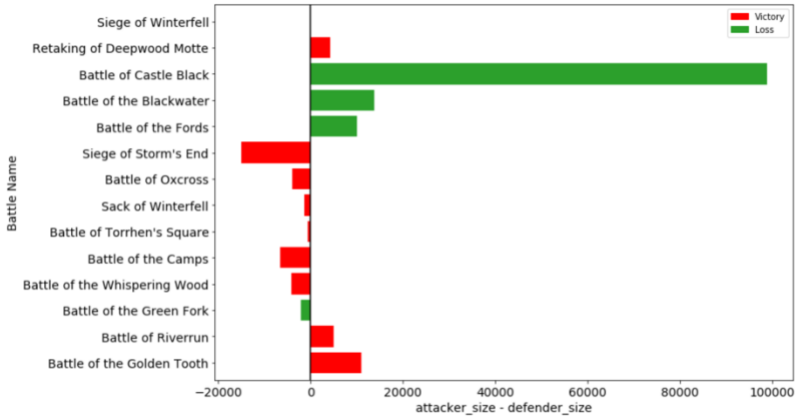


Figure 17. Battles result with different army size

Attacker king and defender king

We analyze the attacker king and defender king of each battles by the "attacker_king" column and "defender_king" column. Joffrey/Tommen Baratheon attacked other kings the most and Robb Stark was attacked by other kings the most.

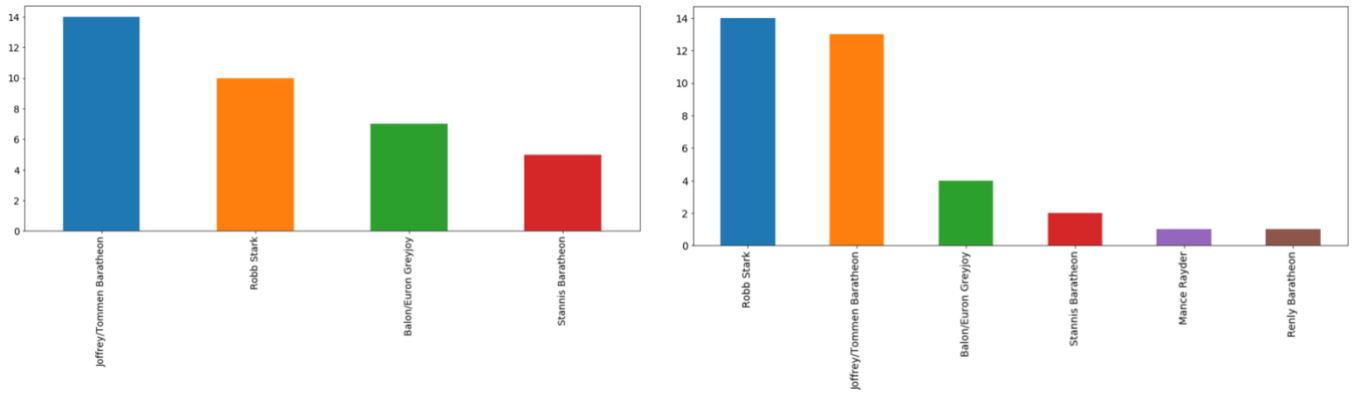


Figure 18. The number of attacker kings and defender kings

Commanders in the battles

We firstly create four score lists: attack_win, attack_loss, defend_win, and defend_lost. For each commander, if he is "attacker_commander" and the "attacker_outcome" is "win", we add him into the attack_win list. If he is "attacker_commander" but the "attacker_outcome" is "loss", we add him into the attack_loss list. It is the same idea for the "defender_commander".

Secondly, we create a dictionary of these four score lists and created a DataFrame from that dictionary, indexed by the commander's name.

- Most Active Commanders:

For each commander, we compare the total number of battles commander participated in by calculating his number in attack_win+attack_loss+defend_win+defend_loss lists. Then, sort the value and get 5 most active commanders.

```
battle['total_battles'] = battle['attack_win'] + battle['attack_loss'] + battle['defend_win'] + battle['defend_loss']
battle.sort_values('total_battles', ascending=False).head(5)['total_battles']
```

```
Gregor Clegane       8
Stannis Baratheon    5
Roose Bolton         4
Tywin Lannister      4
Robb Stark           4
Name: total_battles, dtype: int64
```

Figure 19. Five most active commanders

- Commanders who had most victories:

For each commander, we compare the total number of victorious battles commander participated in by calculating the sum in the attack_win list and defend_win list. Then, sort the value and get 5 commanders who have most victories.

```
battle['total_wins'] = battle['attack_win'] + battle['defend_win']
battle.sort_values('total_wins', ascending=False).head(5)['total_wins']
```

```
Gregor Clegane     6
Jaime Lannister    4
Robb Stark         4
Walder Frey        3
Theon Greyjoy      3
Name: total_wins, dtype: int64
```

Figure 20. Five commanders who has most victories

- The win percentage of commander;

We got the win percentage by using the formula: win percentage=#total_wins/ #total_battles and sorted it.

```
battle['win_percentage'] = battle['total_wins'] / battle['total_battles'] * 100
battle.sort_values('win_percentage', ascending=False).head(75)['win_percentage'][battle['win_percentage'] != float('Inf')]
```

```
Karyl Vance          100.000000
Sandor Clegane       100.000000
Kevan Lannister      100.000000
Rorge                100.000000
Theon Greyjoy        100.000000
Smalljon Umber       100.000000
Jonos Bracken        100.000000
Mathis Rowan         100.000000
Robett Glover        100.000000
Rodrik Cassel        100.000000
Loras Tyrell         100.000000
Vance                100.000000
Tytos Blackwood      100.000000
Euron Greyjoy        100.000000
Cley Cerwyn          100.000000
Robb Stark           100.000000
Asha Greyjoy         100.000000
Mace Tyrell          100.000000
```

Figure 21. Win percentage of each commander

- Greatest commanders in the battles:

Firstly, we scored each commander by counting the number of victories and loss. If he appeared in the "attack_win" and "defend_win" lists, then he earned 1 point for each, and if he appeared in the "attack_loss" and "defend_loss" lists, he would lose 1 point for each. Then, we sort the score and showed the top 3 commanders.

```
battle['score'] = (battle['attack_win'] + battle['defend_win']) - (battle['attack_loss'] + battle['defend_loss'])
battle.sort_values('score', ascending=False).head(3)['score']
```

```
Gregor Clegane     4
Jaime Lannister    4
Robb Stark         4
Name: score, dtype: int64
```

Figure 22. Top three commanders with 4 score

As the result showed above, these three commanders had the same score. To figure out one greatest commander, the army size was considered as a factor that values the greatest commanders. When he acted as the attacker commander and won the battle, the relative army size means each one people in attacker army had to fight with how many people in the defender army. The larger relative army size is, the commander has a greater value in the battle. It can be calculated by using defender size/attacker size. We created a DataFrame to show the relative army size of four battles these commanders participated in. Finally, we use the mean( ) to get the average. As the figure shows, Gregor Clegane is the greatest commander.

```
top3_ratio = pd.DataFrame()
top3_ratio['gregor'] = gregor_ratio
top3_ratio['jaime'] = jaime_ratio
top3_ratio['robb'] = robb_ratio
top3_ratio
```

	gregor	jaime	robb
0	NaN	0.266667	1.111111
1	2.104167	0.666667	3.200000
2	NaN	NaN	NaN
3	3.688525	NaN	NaN

```
print('Gregor Clegane ', top3_ratio['gregor'].mean())
print('Jaime Lannister', top3_ratio['jaime'].mean() )
print('Robb Stark     ',top3_ratio['robb'].mean())
```

```
Gregor Clegane  2.8963456284153004
Jaime Lannister 0.4666666666666667
Robb Stark      2.155555555555556
```

Figure 23. Greatest commander after considering relative army size

Conclusion of the first part

The Riverlands area had the largest number of battles.

The number of death and captures are related to the regions and battles.

The culture has a strong relationship with the death of characters.

The most common type of battles was pitched battle. Joffrey/ Tommen Baratheon's favorite battle type is "Pitched Battle" and Robb Stark's favorite type of attack seems to be "Ambush". All five of his "Ambush" type battles are against Joffrey/Tommen Baratheon.

Gregor Clegane is the greatest commander.

4.2.2 The analysis of characters

Firstly, we read the .csv files to data Frame.

[5]:
```
# Importing modules
import pandas as pd
import matplotlib.pyplot as plt

# Reading in datasets/book1.csv
book1 = pd.read_csv("asoiaf-book1-edges.csv")
book1.head()
```

[5]:

	Source	Target	Type	weight	book	new_weight
0	Addam-Marbrand	Jaime-Lannister	Undirected	3	1	289
1	Addam-Marbrand	Tywin-Lannister	Undirected	6	1	286
2	Aegon-I-Targaryen	Daenerys-Targaryen	Undirected	5	1	287
3	Aegon-I-Targaryen	Eddard-Stark	Undirected	4	1	288
4	Aemon-Targaryen-(Maester-Aemon)	Alliser-Thorne	Undirected	4	1	288

Figure 24. DataFrame of book 1

Secondly, we import network package and create a Graph Object. And we iterate the book1 data Frame and add edge to graph object according to the column name.

Calculation of degree centrality

we calculate the degree centrality in book1 and sort it according to their degree centrality value (*Figure 14*).

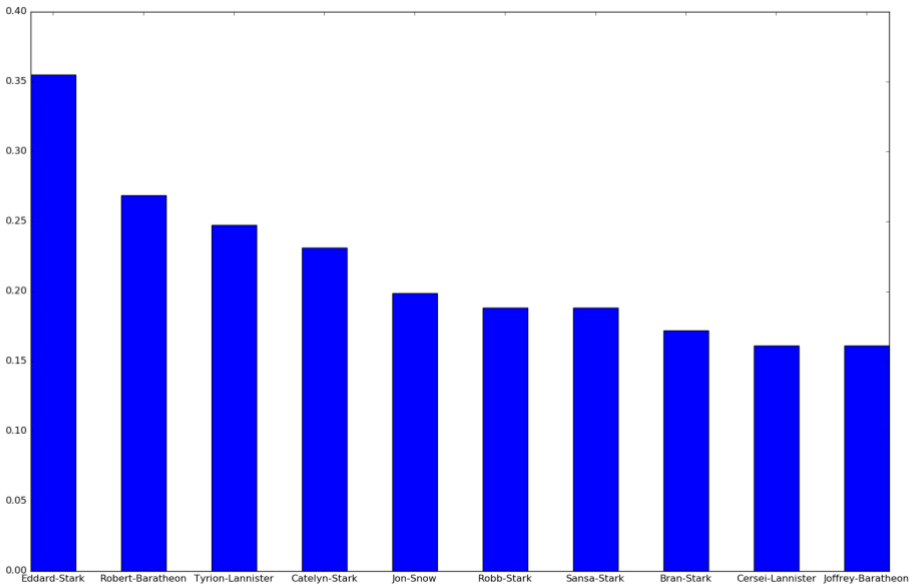


Figure 25. Result for Degree Centrality rank in book 1

Calculation of betweenness centrality

Then, we calculate the betweenness centrality in book1 and sort to get the top 10 characters.

```
#Betweenness centrality of book1
bet_cen_book1 = nx.betweenness_centrality(books[0], weight='new_weight')
```

Figure 26. Calculation of betweenness centrality in book 1

Finally, we calculate the betweenness centrality in all books and sort to get the top 10 characters.

```
#Closeness_Centrality
bet_cen_book_all = nx.closeness_centrality(books[5], distance='new_weight')
sorted_bet_cen_book_all = sorted(bet_cen_book_all.items(), key=lambda x:x[1], reverse=True)[0:10]
print('top10 of ALL Book: ','\n', sorted_bet_cen_book_all, '\n')
```

```
top10 of ALL Book:
 [('Tyrion-Lannister', 0.4763331336129419), ('Robert-Baratheon', 0.4592720970537262), ('Eddard-Stark', 0.455848623853211),
('Cersei-Lannister', 0.45454545454545453), ('Jaime-Lannister', 0.4519613416714042), ('Jon-Snow', 0.44537815126050423), ('Sta
nnis-Baratheon', 0.4446308724832215), ('Robb-Stark', 0.4441340782122905), ('Joffrey-Baratheon', 0.4339519650655022), ('Catel
yn-Stark', 0.4334787350054526)]
```

Figure 27. Calculation of betweenness centrality in all books

Result analysis

Unlike Harry Potter Series, this series is combined with different stories which contains different important characters. For example, we can see how the degree centrality of Eddard-

Stark changes from book 1 to book 5.

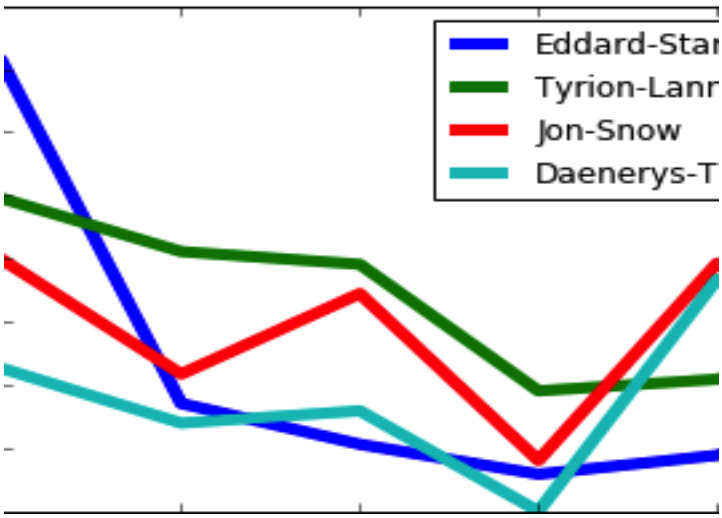


Figure 28. Result

We take an observation of *Figure 28*, there is a sudden drop at the line Eddard-Strak! WHY? Because Eddard was the most important character in book 1, but he dead… And another interesting phenomenon in *Figure 28*, although the story of Ice (John Snow) and Fire (Daenerys) happened in different physical position, their characters seems to have similar development trend of line. In book 5, finally the ice and fire become together, and the winter comes. That interesting trend reflecting the author's control over the story.

Conclusion of second part

Actually, we can know from the result that Tyrion-Lannister has the highest degree centrality for the whole book, which means he is good at communication with others and knows most persons from different culture and families. And for John Snow, he has the highest between centrality. That means he become one of the greatest central bridge (leader) to combine all other characters. Finally, the Closeness Centrality shows the shortest way from one character to another, which means, Tyrion has The highest frequency of appearance with other characters. That is Game of Throne, you never know who's the next king or he is cruel or mercy. Sometimes you are just as good as Eddard, but you are finally framed to death.

So, who will dead next? This is next interesting question we are going to handle.

5. Building models

To predict the death of characters and test our predictions, we build eight models.

5.1 Decision Tree model

Algorithm principle

The decision tree is a tree-like decision graph with additional probability results, and is an intuitive graphical method using statistical probability analysis. We use a decision tree to construct a predictive model that represents a mapping between object properties and object values. Each node in the tree represents the condition of the object's properties, and its branches represent objects that meet the node's criteria. The leaf node of the tree represents the prediction result to which the object belongs.

Process of implementation

- Import the model: Import the DecisionTreeClassifier object in sklearn.tree
- Instantiate an object: Instantiate the DecisionTreeClassifier object. When instantiating the object, we need to assign values to the parameters in the object. The value of the parameter selection will be explained later.
- Fit the data: Call the fit( ) function to fit the model using the training set data. Use training set data to predict if a character is dead
- Score the model: The model is scored using the accuracy_score function that comes with the decision tree object. The score can indicate the accuracy of the model prediction.

Using the default decision tree, the decision tree with default parameter values, the predictive model scored 0.74 points (as shown below). 0.74 indicates that this model has a weak prediction effect and does not have practical application value. Plus, we will optimize the decision tree model.

```
from sklearn.tree import DecisionTreeClassifier
DT=DecisionTreeClassifier()
DT.fit(X_train,y_train)
preds_dt = DT.predict_proba(X_test)
print('DecisionTree Accuracy: (original)\n',accuracy_score(y_test, np.argmax(preds_dt, axis = 1), normali
```

```
DecisionTree Accuracy: (original)
 0.7371663244353183
```

Figure 29. Accuracy of Decision Tree

Algorithm improvement

In the process of optimizing the algorithm, we modified the values of the following parameters-max_features, max_depth and min_samples_split. Finally, the accuracy score of our model was increased to 0.83. As shown in the figure below, when max_features=14, max_depth=16, min_samples_split=10, the model has the highest score, 0.83 points. Compared to the original model, accuracy score increased by 13.7%, and the correct rate of 83% indicates that the model has a good prediction effect.

```
14 16 10
DecisionTree Accuracy: (original)
 0.8275154004106776

: 1 pd.DataFrame(score).max()

: 0    0.827515
dtype: float64
```

Figure 30. Accuracy of Decision Tree after improvement

Analysis of algorithm

To analyze the decision tree algorithm, we plot the confusion matrix. In the picture representing the confusion matrix, it can be seen that most of the data is correctly divided into the classification in which it is located. Most of the data that is misclassified is due to the division of the character that died in the novel to be alive. But for the score of 0.83, it is enough to predict the life and death of the character.

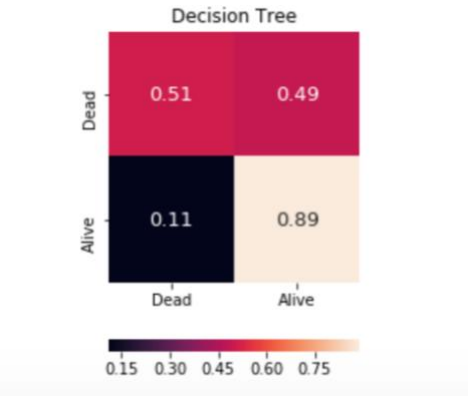


Figure 31. Confusion matrix of Decision Tree

5.2 KNN model

Algorithm principle

This algorithm can be used to find groups within unlabeled data. There is a sample data set, also called a training sample set, and there is a label for each data in the sample set, that is, we know the correspondence between each data in the sample set and its own. After inputting the new data without the label, each feature of the new data is compared with the feature corresponding to the data in the sample set, and then the algorithm extracts the classification label of the most similar data (nearest neighbor) of the feature in the sample set. In general, we only select the K most similar data in the sample data set. This is the source of K in the K-nearest neighbor algorithm. Usually K is an integer not greater than 20. Finally, select the category with the most occurrences among the K most similar data as the classification of new data.

Process of implementation

- Import the model: Import KNeighborsClassifier from sklearn.neighbors.
- Instantiate an object: Define the classifier by inputting the parameter n_neighbors
- Fit the data: Call the fit( ) function to fit the model using the training set data. Use training set data to predict if a character is dead
- Score the model: The model is scored using the accuracy_score function that comes with the KNN object. The score can indicate the accuracy of the model prediction.

Algorithm improvement

Change the parameter n_neighbors of KNNclassifier: it means the number of neighbors to get. Finally, the score is improved to 0.8437821171634121

Analysis of algorithm

Advantages: high precision, insensitivity to outliers, no data input assumptions.

Disadvantages: high time complexity and high space complexity

Applicable data range: numeric and nominal.

5.3 SVC model

Algorithm principle

A support vector machine (SVM) is machine learning algorithm that analyzes data for classification and regression analysis. SVM is a supervised learning method that looks at data and sorts it into one of two categories. An SVM outputs a map of the sorted data with the margins between the two as far apart as possible. SVMs are used in text categorization, image classification, handwriting recognition and in the sciences.

Process of implementation

- Import the model: Import SVC, LinearSVC from sklearn.svm
- Instantiate an object: Define the classifier by setting the parameter probability=True
- Fit the data: Call the fit( ) function to fit the model using the training set data. Use training set data to predict if a character is dead
- Score the model: The model is scored using the accuracy_score function that comes with the SVC object. The score can indicate the accuracy of the model prediction.

Algorithm improvement

Change the parameter C in classifier. The larger C is equivalent to the penalty slack variable, and it is hoped that the slack variable is close to 0, that is, the penalty for misclassification increases, and tends to be fully paired for the training set. Thus, the accuracy of the training set is high, but generalization. Weak ability. The C value is small, the penalty for misclassification is reduced, tolerance is allowed, and they are regarded as noise points, and the generalization ability is strong.

Analysis of algorithm

Advantages: The quality of generalization and ease of training of SVM is far beyond the capacities of these more traditional methods. SVM performs well on data sets that have many attributes, even if there are very few cases on which to train the model. There is no upper limit on the number of attributes; the only constraints are those imposed by hardware. Traditional neural nets do not perform well under these circumstances.

Disadvantages: The disadvantages are that the theory only really covers the determination of the parameters for a given value of the regularization and kernel parameters and choice of kernel. In a way the SVM moves the problem of over-fitting from optimizing the parameters to model selection.

5.4 Gradient Boosting model

Algorithm principle

The basic idea of Boosting is to make each round of base learners pay more attention to the sample of the previous round containing learning errors in the training process.

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. Using a training set $\{(x_1, y), \ldots, (x_n, y_n)\}$of known values of x and corresponding values of y, the goal is to find an approximation $\widehat{F}(x))$ to a function $F(x)$ that minimizes the expected value of some specified loss function $L(y, F(x))$:

$$\widehat{F}(x) = arg_F min \mathrm{E}_{x,y}[L(y, F(x))]$$

The gradient boosting method assumes a real-valued y and seeks an approximation F^(x) in the form of a weighted sum of functions h_i (x) from some class H, called base (or weak) learners:

$$\widehat{F}(x) = \sum_{i=1}^{M} \gamma_i h_i(x) + constant$$

Process of implementation

- Import the model: import GradientBoostingClassifier from sklearn.ensemble
- Instantiate an object: Define the classifier by setting the parameter n_estimators=500,random_state=0,learning_rate=0.1
- Fit the data: Call the fit( ) function to fit the model using the training set data. Use training set data to predict if a character is dead
- Score the model: The model is scored using the accuracy_score function that comes with the Gradient Boosting object. The score can indicate the accuracy of the model prediction.

```
GradientBoosting Accuracy:
 0.7741273100616016
```

Figure 32. Accuracy of Gradient Boosting

Algorithm improvement

Configure the hyperparameter. After changing the value of n_estimator and the value of random_state, the score increased from 0.774 to 0.828.

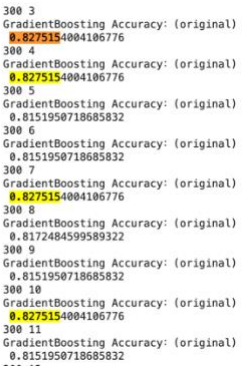


Figure 33. Accuracy of Gradient Boosting after improvement

Analysis of algorithm

Disadvantages: GBM does not regularize and cannot prevent overfitting.

When the GBM algorithm is used for pruning, it will stop splitting directly when the node information gain is negative, so it is a greedy algorithm. And GBM can only test a limited size of value.

5.5 AdaBoostClassifier model

Algorithm principle

The AdaBoostClassifier exists in scikit-learn and is an object for classification. AdaBoostClassifier uses two implementations of the Adaboost classification algorithm, SAMME and SAMME.R.

Process of implementation

- Import the model: Import AdaBoostClassifier object from sklearn.ensemble
- Instantiate an object: Instantiate the AdaBoostClassifier object. While instantiating the object, we need to assign values to the parameters in the object. The parameters that need to be assigned are: learning_rate, algorithm, n_estimators, and base_estimator.
- Fit the data: Call the fit( ) function to fit the model using the training set data. Use training set data to predict if a character is dead
- Score the model: The model is scored using the cross_val_score function. The scoring strategy is to calculate the accuracy of the model, repeat the intersection 8 times, and finally calculate the average value of each score.

Algorithm improvement

There are two ways to optimize AdaBoostClassifier. The first way is to adjust the Adaboost

framework. The second way is to adjust the weak classifier we choose.

Similar to the process of optimizing the decision tree algorithm, we select the parameters of the decision tree algorithm as a base estimator and filter the optimal values of max_features, max_depth and min_samples_leaf. After constructing a three-layer loop, the optimal combination is obtained: max_features=7, max_depth=5 and min_samples_leaf=8. Under this optimal combination, the accuracy of the model reached 0.83. Compared with the conclusion of the previous stage, the model has only a slight improvement.

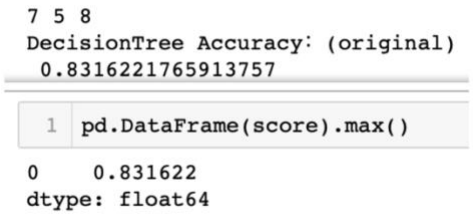


Figure 34. Accuracy of AdaBoostClassifier after improvement

5.6 Linear Regression model

Algorithm principle

Linear regression models are a good starting point for regression tasks. Such models are popular because they can be fit very quickly, and are very interpretable

Process of implementation

- Import model:from sklearn.linear_model import LinearRegression, from sklearn.metrics import accuracy_score.
- Instantiate an object: Instantiate the LinearRegression object.
- Fit the data: Call the fit() function to fit the model using the training set data
- Predict results: Use the built model to predict result using X_test. We assume that value is set as 1 if it's more than or equal to 0.5, as 0 if it's less than 0.5.
- Score the model: The model is scored using the score function.

Algorithm improvement

We call function, statsmodels.formula.api, to get OLS Regression Results to improve the model performance. Among the parameter P> |t|, we get the maximum one to delete. If there is a higher

Adj.R-squared after deleting that parameter, we think the model is improved

5.7 Random Forest model

Algorithm principle

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

Process of implementation

- Import the model: from sklearn.ensemble import RandomForestClassifier
- Instantiate an object: Instantiate the RandomForestClassifier object. When instantiating the object, we need to assign values to the parameters in the object. The parameters that need to be assigned are: n_estimators, max_features, max_depth, bootstrap, oob_score. The value of the parameter selection will be explained later
- Fit the data: Call the fit() function to fit the model using the training set data
- Score the model: The model is scored using the score function.

Algorithm improvement

We assign different values to max_features, max_depth, min_samples between 5 and 16, 5 and 20, between 1 and 20 to get the maximum accuracy score 0.827515.

5.8 Naive Bayes

Algorithm principle

In machine learning, Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with string (naive) independence assumption between the feature.

Process of implementation

- Import the model: from sklearn.naive_bayes import GaussianNB
- Instantiate an object: Instantiate the GaussianNB object.
- Fit the data: Call the fit() function to fit the model using the training set data
- Predict results: Use the built model to predict result using X_test.
- Score the model: The model is scored using the score function.

Algorithm improvement

We assign different values to max_features, max_depth, min_samples between 5 and 16, 5 and 20, between 1 and 20 to get the maximum accuracy score 0.827515.

6. Results

We use 4 pictures to draw the final results.

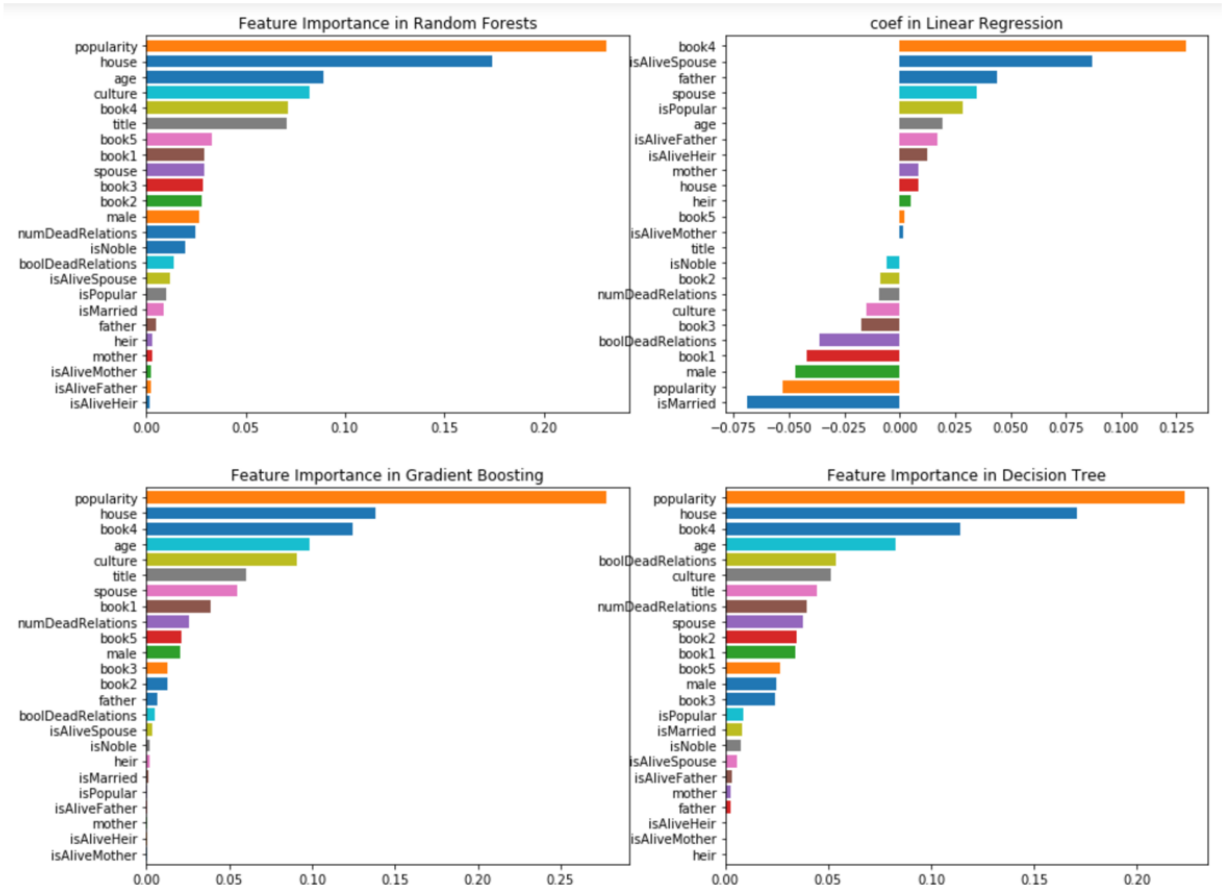


Figure 35. Feature importance in 4 models

Feature importance (*Figure 35*)

Figure 35 shows the importance of each feature in the Random Forest, Decision Tree, and Gradient Boosting algorithms under the current data set, as well as the coefficients of the Linear Regression model.

By observing the following figure, we can see that among the four models, popularity, house and book4 are the three most important factors affecting the results. This conclusion is very easy to understand. For popularity, the author as a novelist is bound to pay attention to the reader's feedback on the characters. If a role is more popular to readers, then he should live longer. For the

house, as we said earlier, the background of the "Song of Ice and Fire" story is the kingdom hegemony. When the white walker invade, wars and conflicts will cause more people to die. For the hose, it stores information about the place where the character lives. The location of the characters is closely related to their situation and the security, so the house attribute will also greatly affect if the character is survive. For book4, the plot is full of conflicts, and the country is full of battles. Therefore, it is very easy to understand that there are a large number of characters died in this book.

Confusion matrix (*Figure 36*)

Figure 36 shows the confusion matrix drawn by the model constructed by Linear Regression, Random Forest, Decision Tree, Knn, SVC and Naive Bayes algorithm. By showing a picture of the confusion matrix, we can see that each model can divide most of the test sets into the correct category. Among them, Random Forest performed best, and Naive Bayes and Linear Regression performed the worst. The reason may be that linear regression is more suitable for classification problems of continuous distribution. Naive Bayes is more suitable for models such as word bag analysis because of its principle of calculation by matching probability.

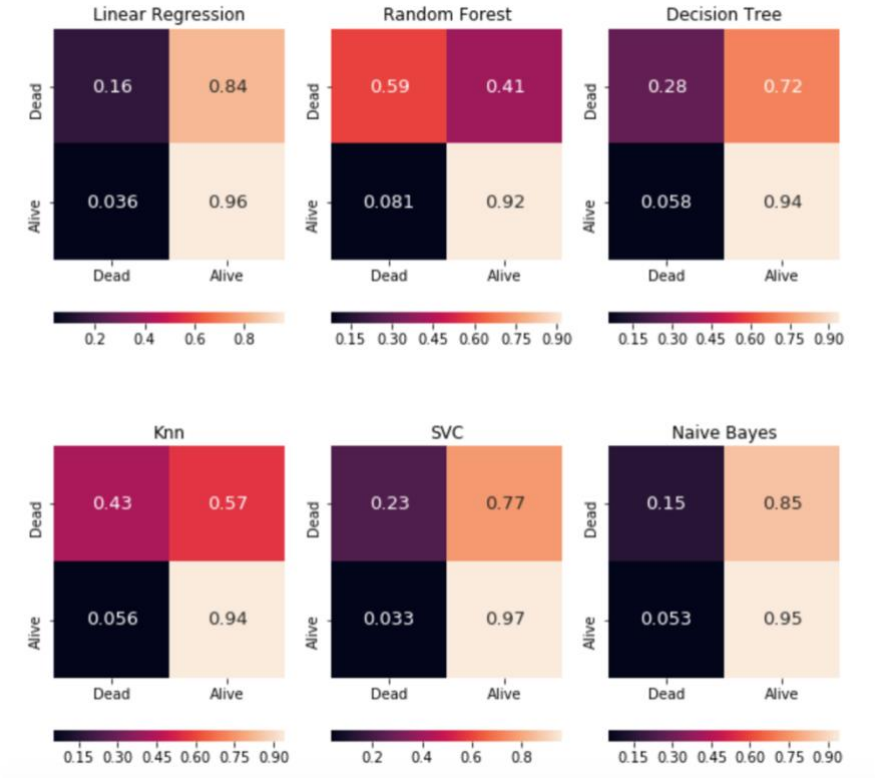


Figure 36. Confusion matrix of 6 models

ROC (Figure 37)

Figure 37 is a ROC plot drawn from six models constructed by the linear regression, random forest, decision tree, knn, svc and naive bayes algorithms. The ROC curve is based on a series of different two-class classifications, with TPR as the ordinate and FPR as the abscissa. The ROC observation model correctly identifies the trade-off between the proportion of the positive case and the ratio at which the model incorrectly identifies the negative case data as a positive example. The area under the ROC curve is a measure of the accuracy of the model, AUC (Area under roccurve). According to the definition of ROC, the closer the curve is to the upper left corner, the better the performance of the model. Therefore, linear regression is the worst performer, and random forest performs best. This is consistent with the conclusion we got from the confusion matrix.

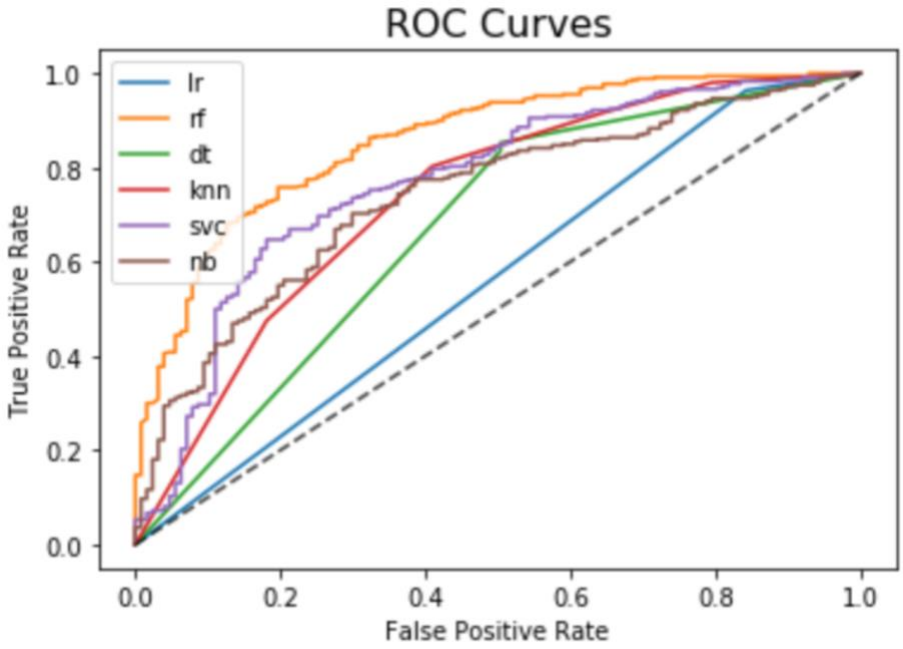


Figure 37. ROC curves of 6 models

Accuracy score of models

Figure 38 is used to represent the model's accuracy score. The classification accuracy score is the correct percentage of all classifications, but it does not tell you the potential distribution of the response, and it does not tell you the type of error the classifier made. We constructed a histogram of the eight models' accuracy scores. All models can achieve more than 74% accuracy, Decision Tree, Random Forest and AdaBoostClassifier have the best accuracy, and

Naive Bayes and Linear Regression have the lowest accuracy.

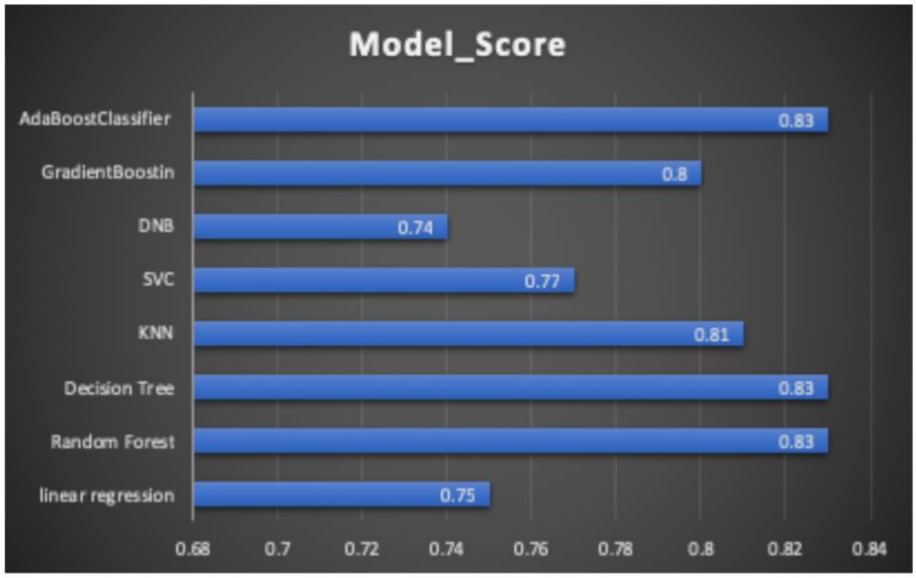


Figure 38. Accuracy score of 8 models

7.Conclusion

From thousands of node and edges, we learned that some characters in this book series really important which have high degrees / betweenness centrality（like Tyrion-Lannister and John Snow）, but no one worth 'The only protagonist'. Because with the book series moves, some popular character may not exist -- that is Game of Throne, you never know who's the next king or he is cruel or mercy. Sometimes you are just as good as Eddard - Stark, but you are finally framed to death.

So, who will dead next? Using 1946 samples, we constructed eight models to predict whether the character in the novel "Song of Ice and Fire" would die. By comparing these eight models, we can conclude that the random forest algorithm performs best and is the most suitable model for predicting whether a character will die in this novel. Its correct rate is 83%. Although there is no way to achieve 100% accuracy, we believe that this result is still very meaningful. Because the work we do is to predict whether the character will die in the novel, the death of the character itself will have more uncertain factors than the sample in real life - the author's will.

Since one of the dramatic factors cannot be avoided, we can only accept imperfections. However, the "accident" death is always a minority, and the death of most characters is in line with the plot development and the environment at the time. Especially for a famous serialized novel, the author is more concerned about whether the character is popular with readers. Therefore, the popularity and role of the area of residence become the most important factor affecting their fate.

In conclusion, our model could always contribute a lot in characters' death prediction.

8.Reference

1. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
2. https://www.kaggle.com/mylesoneill/game-of-thrones
3. https://www.kaggle.com/pierreericgarcia/game-of-thrones-analysis
4. http://carinyperez.com/game-thrones-data-analysis/
5. https://dzone.com/articles/scikit-learn-and-game-of-thrones