UFC Project Final

July 29, 2020

1 Predicting the Type of Finish in UFC Matches

The goal of this project is to investigate whether UFC match outcomes can be predicted accurately using three different classification algorithms. More precisely, the goal is to determine whether the algorithms can correctly predict an outcome of a match being:

1. A submission ('SUB');
2. A unanimous decision ('U-DEC');
3. A split decision ('S-DEC');
4. A KO/TKO ('KO/TKO')

The algorithms used will be:

1. A Random Forest Classifier (model named RF);
2. A K-Nearest Neighbors Classifier (model named KNN);
3. A Support Vector Machine Classifier (model named SVM).

The process of selecting/engineering the features and then using them in the algorithms is iterative in order to improve classifier score.

[422]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

Reading in the data to a DataFrame called df

[423]:
```
df = pd.read_csv('ufc-master.csv')
```

[424]:
```
df.head()
```

[424]:

	R_fighter	B_fighter	R_odds	B_odds	R_ev	\
0	Deiveson Figueiredo	Joseph Benavidez	-225	180	44.444444	
1	Jack Hermansson	Kelvin Gastelum	-112	-112	89.285714	

```
2        Marc Diakiese     Rafael Fiziev    -167     135  59.880240
3        Ariane Lipski    Luana Carolina    -125     100  80.000000
4    Alexandre Pantoja     Askar Askarov    -210     165  47.619048

         B_ev       date                                    location  \
0  180.000000  7/18/2020  Abu Dhabi, Abu Dhabi, United Arab Emirates
1   89.285714  7/18/2020  Abu Dhabi, Abu Dhabi, United Arab Emirates
2  135.000000  7/18/2020  Abu Dhabi, Abu Dhabi, United Arab Emirates
3  100.000000  7/18/2020  Abu Dhabi, Abu Dhabi, United Arab Emirates
4  165.000000  7/18/2020  Abu Dhabi, Abu Dhabi, United Arab Emirates

                country Winner  …  B_Featherweight_rank B_Bantamweight_rank  \
0  United Arab Emirates    Red  …                   NaN                 NaN
1  United Arab Emirates    Red  …                   NaN                 NaN
2  United Arab Emirates   Blue  …                   NaN                 NaN
3  United Arab Emirates    Red  …                   NaN                 NaN
4  United Arab Emirates   Blue  …                   NaN                 NaN

  B_Flyweight_rank  B_Pound-for-Pound_rank  better_rank  finish  \
0              2.0                     NaN          Red     SUB
1              NaN                     NaN          Red     SUB
2              NaN                     NaN      neither   U-DEC
3              NaN                     NaN      neither     SUB
4              NaN                     NaN      neither   U-DEC

     finish_details  finish_round  finish_round_time  total_fight_time_secs
0  Rear Naked Choke           1.0              04:48                  288.0
1         Heel Hook           1.0              01:18                   78.0
2               NaN           3.0              05:00                  900.0
3           Kneebar           1.0              01:28                   88.0
4               NaN           3.0              05:00                  900.0

[5 rows x 113 columns]
```

[425]:
```
df.shape
```

[425]:
```
(4292, 113)
```

1.1 So, it can be seen that there are many columns in the dataset.

The features chosen to train and test the classification algorithms will have to be selected.

[426]:
```
df.columns
```

[426]:
```
Index(['R_fighter', 'B_fighter', 'R_odds', 'B_odds', 'R_ev', 'B_ev', 'date',
       'location', 'country', 'Winner',
       …
```

```
       'B_Featherweight_rank', 'B_Bantamweight_rank', 'B_Flyweight_rank',
       'B_Pound-for-Pound_rank', 'better_rank', 'finish', 'finish_details',
       'finish_round', 'finish_round_time', 'total_fight_time_secs'],
      dtype='object', length=113)
```

[427]:
```
df.dtypes
```

[427]:
```
R_fighter                object
B_fighter                object
R_odds                    int64
B_odds                    int64
R_ev                    float64
                         ...
finish                   object
finish_details           object
finish_round            float64
finish_round_time        object
total_fight_time_secs   float64
Length: 113, dtype: object
```

As can be seen above, there are a mixture of datatypes in the DataFrame. There are also columns with negative numbers. This will have to be considered when selecting/using features.

2 Choosing features

This is an extremely important step in the process of building a classification model.

The features should have some relationship to the dependent variable, and ideally be independent of one another.

Multicollinearity (when features are dependent on each other) should be avoided where possible to reduce interference in the classification process.

It may also be necessary to engineer the features to some extent

The inital features chosen are:

- The fighter odds
- The difference in number of rounds each fighter has fought
- The difference in number of KO wins each fighter has
- The difference in average significant strikes per match for each fighter
- The difference in submission attempts made by each fighter
- The difference in takedowns made by each fighter

However, there are some issues with using these features as-is, so the features must be engineered to suit the models.

This process will add new columns to df. The goal will be to remove negative values and create columns which have values of the differential stats between the two fighters.

[428]:
```
df['odds_dif'] = abs(df['R_odds'] - df['B_odds'])
```

[429]:
```
df.columns
```

[429]:
```
Index(['R_fighter', 'B_fighter', 'R_odds', 'B_odds', 'R_ev', 'B_ev', 'date',
       'location', 'country', 'Winner',
       …
       'B_Bantamweight_rank', 'B_Flyweight_rank', 'B_Pound-for-Pound_rank',
       'better_rank', 'finish', 'finish_details', 'finish_round',
       'finish_round_time', 'total_fight_time_secs', 'odds_dif'],
      dtype='object', length=114)
```

[430]:
```
df['odds_dif']
```

[430]:
```
0       405
1         0
2       302
3       225
4       375
       …
4287    290
4288    385
4289    480
4290    755
4291    300
Name: odds_dif, Length: 4292, dtype: int64
```

Above, it can be seen that the new column has been added. The methodology here is that hopefully a large differential in odds will help the algorithms in deciding which way the match will end.

Below, this will be extended to the other features chosen.

[431]:
```
df['abs_round_dif'] = abs(df['total_round_dif'])
```

[432]:
```
df['abs_ko_dif'] = abs(df['ko_dif'])
```

[433]:
```
df['abs_sig_str_dif'] = abs(df['sig_str_dif'])
```

[434]:
```
df['abs_sub_att_dif'] = abs(df['avg_sub_att_dif'])
```

[435]:
```
df['abs_td_dif'] = abs(df['avg_td_dif'])
```

[436]:
```
df.columns
```

[436]:
```
Index(['R_fighter', 'B_fighter', 'R_odds', 'B_odds', 'R_ev', 'B_ev', 'date',
       'location', 'country', 'Winner',
       …
       'finish_details', 'finish_round', 'finish_round_time',
```

```
       'total_fight_time_secs', 'odds_dif', 'abs_round_dif', 'abs_ko_dif',
       'abs_sig_str_dif', 'abs_sub_att_dif', 'abs_td_dif'],
      dtype='object', length=119)
```

[437]:
```
df.shape
```

[437]:
```
(4292, 119)
```

We can see above that we have added the 6 new columns.

There is no need to .pop() or remove any of the original features at this stage

[438]:
```
df['finish']
```

[438]:
```
0          SUB
1          SUB
2        U-DEC
3          SUB
4        U-DEC
         ...
4287    KO/TKO
4288    KO/TKO
4289    KO/TKO
4290     U-DEC
4291     U-DEC
Name: finish, Length: 4292, dtype: object
```

The "finish" column has non-numerical values, so we will map them to numbers.

[439]:
```
def finish_type(finish):
    if finish == 'SUB':
        return 1
    elif finish == 'U-DEC':
        return 2
    elif finish == 'S-DEC':
        return 3
    elif finish == 'KO/TKO':
        return 4
    else:
        return 0
```

[440]:
```
df['finish'] = df.finish.apply(finish_type)
```

[441]:
```
df['finish']
```

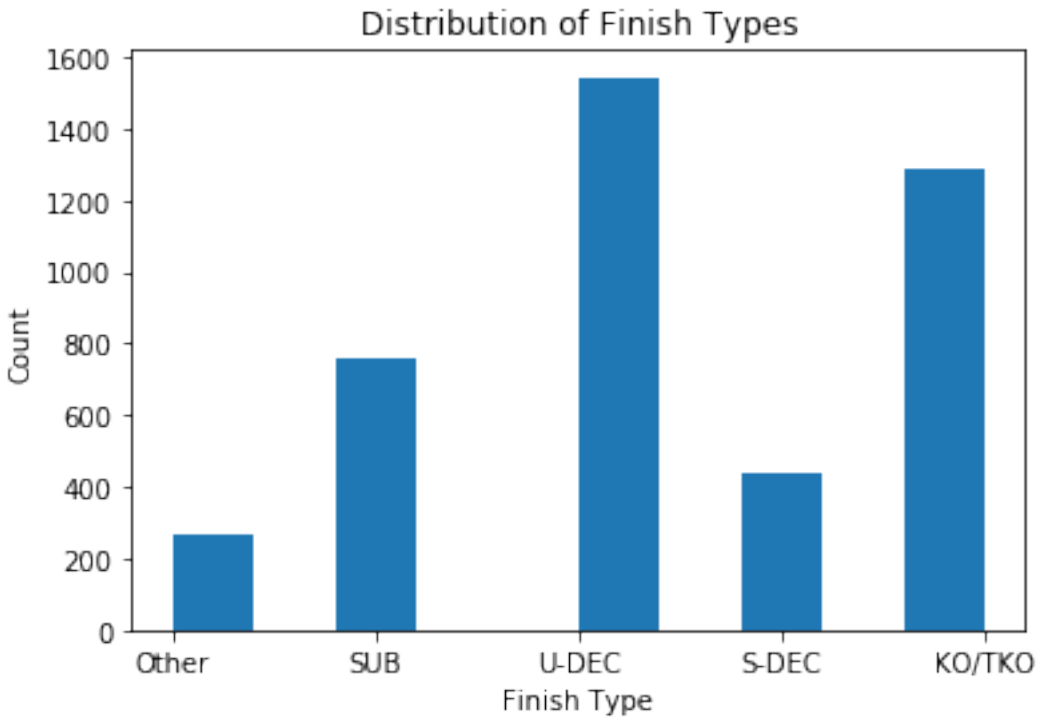
[441]:
```
0       1
1       1
2       2
```

```
3       1
4       2
       ..
4287    4
4288    4
4289    4
4290    2
4291    2
Name: finish, Length: 4292, dtype: int64
```

3 Visualising the Finish Types

A histogram will show us the distribution of finish types, and a pie chart can further provide insight into the proportions of the fights when end in each way.

[442]:
```
plt.hist(df['finish'], bins = 10)
plt.title('Distribution of Finish Types')
plt.xlabel('Finish Type')
plt.ylabel('Count')
plt.xticks(np.arange(5), ('Other', 'SUB', 'U-DEC', 'S-DEC', 'KO/TKO'))

plt.show()
```

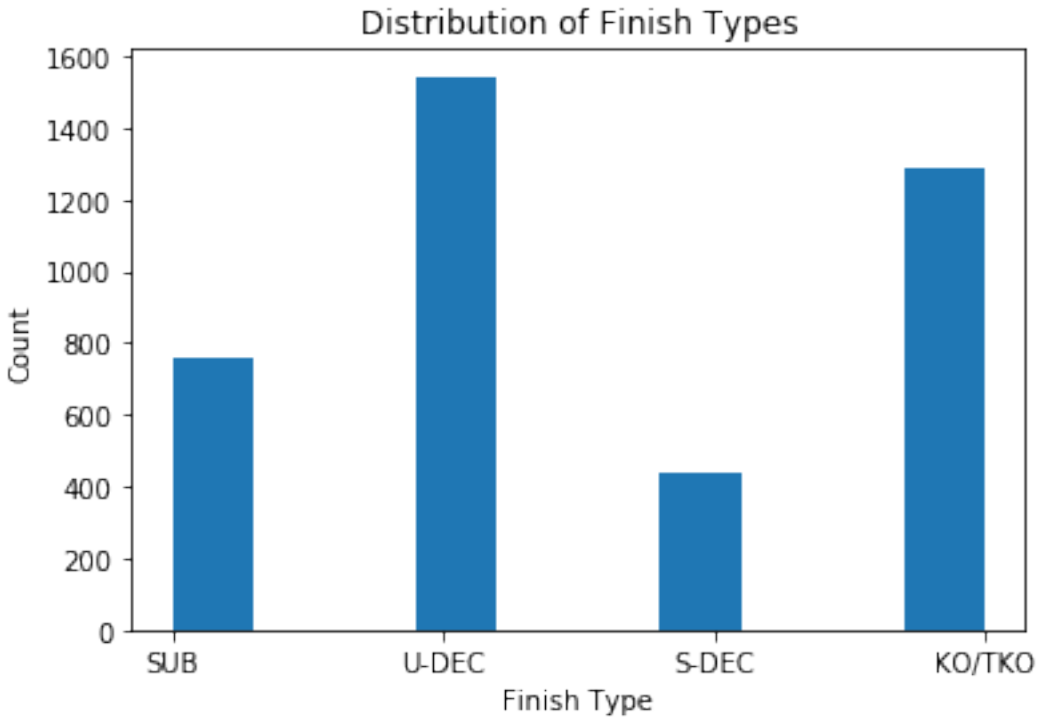


As can be seen, there is another category of finish. Upon inspecting the data, this is mostly where '0' has been put into rows of the DataFrame where there is a missing value due to the fights not taking place and there being no finish.

These rows will be dropped from df.

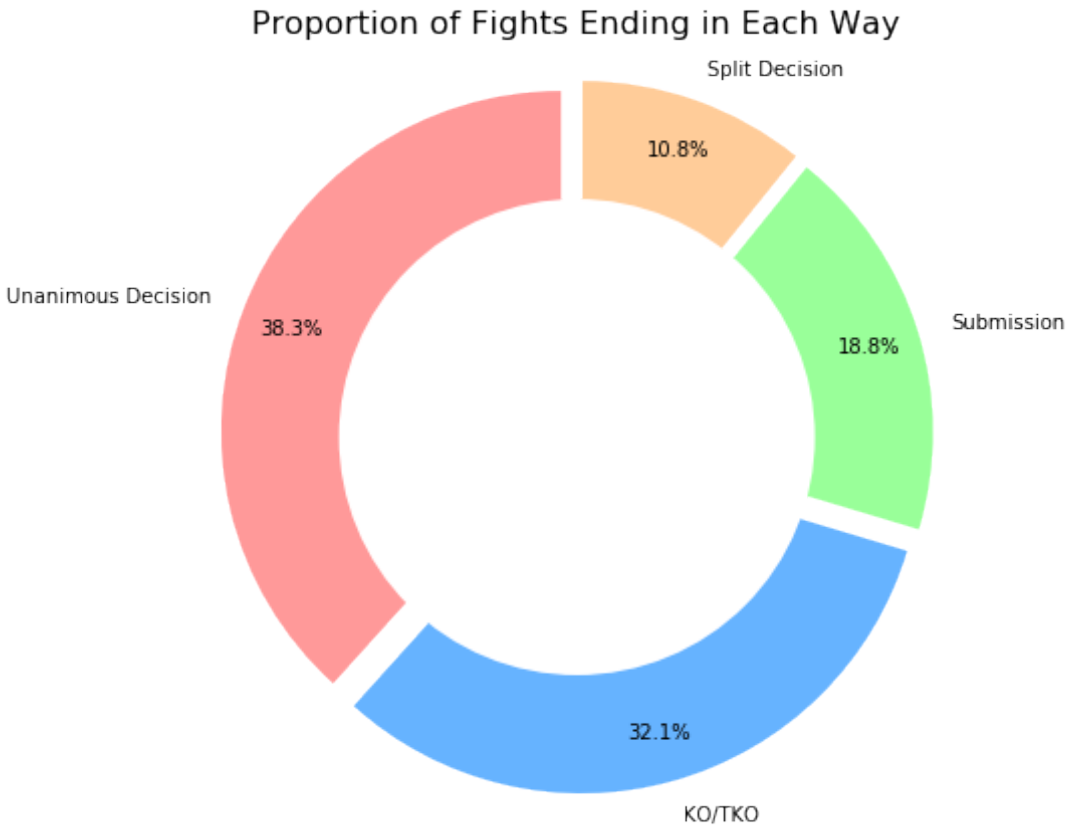
[443]:
```
df = df[df['finish'] != 0]
```

[444]:
```
plt.hist(df['finish'], bins=10)
plt.title('Distribution of Finish Types')
plt.xlabel('Finish Type')
plt.ylabel('Count')
plt.xticks(np.arange(1, 5), ('SUB', 'U-DEC', 'S-DEC', 'KO/TKO'))

plt.show()
```



[445]:
```
labels = ['Unanimous Decision', 'KO/TKO', 'Submission', 'Split Decision']
sizes = [15, 30, 45, 10]#colors
colors = ['#ff9999','#66b3ff','#99ff99','#ffcc99']
explode = (0.05,0.05,0.05,0.05)

plt.figure(figsize=(8, 6))

plt.pie(df['finish'].value_counts(),
```

```
        colors = colors,
        labels=labels,
        autopct='%1.1f%%',
        startangle=90,
        pctdistance=0.85,
        explode = explode)

centre_circle = plt.Circle((0,0),0.70,fc='white')
fig = plt.gcf()
fig.gca().add_artist(centre_circle)
plt.title('Proportion of Fights Ending in Each Way', fontsize = 16)
plt.axis('equal')
plt.tight_layout()
plt.show()
```



[446]:
```
df['finish'].value_counts()
```

[446]:
```
2    1543
4    1291
1     758
```

```
3     435
Name: finish, dtype: int64
```

4 Probabilities of Finish Types

As can be seen in the pie chart above, with the data of over 4,000 fights the probabilities of each finish type are:

1. Submission = 18.8%
2. Unanimous Decision = 38.3%
3. Split Decision = 10.8%
4. KO/TKO = 32.1%

[447]:
```
df.shape
```

[447]:
```
(4027, 119)
```

5 First Iteration of Classifiers

5.1 Initial Feature Selection

To begin using the Classifiers, we will select out the features to be used in the classification

[448]:
```
features1 = df[['odds_dif',
                'abs_round_dif',
                'abs_ko_dif',
                'abs_sig_str_dif',
                'abs_sub_att_dif',
                'abs_td_dif']]
```

[449]:
```
target = df['finish']
```

Splitting the data into train and test sets

[450]:
```
X_train, X_test, Y_train, Y_test = train_test_split(features1, target,␣
↪train_size = 0.8, test_size = 0.2)
```

5.2 Implementing the Random Forest Classifier

[451]:
```
RF1 = RandomForestClassifier()
```

[452]:
```
RF1.fit(X_train, Y_train)
```

[452]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
```

```
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[453]:
```
RF1.predict(X_test)
```

[453]:
```
array([2, 2, 2, 2, 4, 4, 2, 4, 2, 4, 1, 2, 2, 2, 2, 2, 2, 4, 2, 4, 1, 2,
       4, 2, 4, 2, 3, 2, 4, 2, 4, 1, 4, 4, 2, 4, 4, 4, 2, 2, 2, 2, 1, 2,
       2, 4, 2, 4, 2, 4, 2, 4, 4, 2, 4, 4, 2, 1, 4, 1, 4, 3, 4, 1, 1, 2,
       4, 4, 2, 2, 2, 2, 2, 2, 4, 4, 1, 3, 2, 2, 1, 4, 2, 2, 1, 4, 2, 2,
       4, 4, 4, 4, 2, 4, 4, 2, 2, 4, 4, 2, 1, 2, 4, 2, 4, 2, 2, 2, 2, 4,
       1, 1, 2, 2, 2, 2, 2, 2, 2, 4, 2, 1, 2, 4, 1, 2, 4, 4, 4, 4, 4, 2,
       4, 2, 2, 2, 4, 4, 4, 2, 2, 4, 2, 2, 1, 2, 2, 2, 2, 4, 2, 2, 4, 2,
       2, 2, 4, 4, 4, 2, 1, 2, 1, 2, 2, 2, 1, 2, 4, 2, 2, 2, 2, 2, 2, 2,
       4, 1, 4, 4, 2, 4, 2, 4, 2, 2, 2, 1, 4, 4, 4, 2, 2, 4, 4, 4, 2, 2,
       4, 4, 1, 1, 2, 2, 4, 2, 2, 2, 2, 4, 2, 2, 4, 2, 2, 2, 1, 2, 4, 4,
       2, 2, 2, 2, 1, 4, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 4, 4,
       4, 3, 4, 2, 2, 4, 2, 4, 2, 4, 2, 4, 4, 4, 4, 4, 4, 2, 4, 4, 3, 4,
       2, 2, 2, 1, 2, 4, 4, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 4, 2,
       4, 4, 2, 4, 4, 4, 2, 4, 2, 2, 4, 4, 2, 2, 2, 2, 4, 2, 4, 1, 2, 4,
       2, 4, 4, 4, 2, 2, 2, 2, 2, 4, 2, 2, 4, 4, 1, 4, 4, 1, 2, 4, 2, 2,
       1, 2, 4, 4, 1, 2, 4, 2, 2, 2, 1, 4, 4, 4, 2, 4, 4, 1, 4, 2, 2, 2,
       2, 2, 1, 2, 4, 2, 4, 2, 2, 4, 2, 1, 4, 1, 4, 4, 1, 1, 4, 2, 2, 2,
       2, 3, 2, 4, 2, 2, 2, 2, 2, 1, 4, 1, 1, 4, 2, 2, 2, 4, 2, 2, 1, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 4, 4, 4, 2, 2, 4, 2, 4, 2, 4, 2, 2,
       2, 4, 4, 2, 2, 2, 2, 1, 2, 1, 1, 3, 2, 2, 4, 2, 4, 4, 3, 4, 2, 2,
       1, 4, 2, 2, 4, 2, 1, 4, 1, 2, 1, 2, 1, 2, 2, 4, 2, 2, 4, 2, 2, 1,
       4, 1, 4, 2, 2, 2, 2, 1, 2, 2, 2, 4, 4, 2, 1, 2, 2, 4, 2, 4, 2, 4,
       2, 2, 4, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 4, 2, 4,
       2, 1, 1, 4, 2, 1, 2, 2, 2, 4, 2, 4, 2, 4, 4, 2, 2, 2, 4, 2, 2, 2,
       4, 2, 2, 2, 2, 2, 3, 4, 4, 4, 4, 2, 4, 4, 4, 2, 4, 4, 2, 4, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 4, 2, 4, 2, 2, 2, 4, 1, 4, 4, 2, 4, 4, 4, 2,
       4, 2, 1, 4, 2, 4, 4, 2, 4, 2, 2, 1, 2, 2, 4, 4, 4, 2, 4, 2, 2, 4,
       2, 2, 2, 2, 4, 2, 2, 4, 4, 2, 2, 4, 2, 4, 2, 4, 2, 2, 2, 4, 2, 2,
       4, 1, 1, 4, 2, 4, 2, 2, 1, 2, 4, 2, 2, 1, 2, 2, 1, 2, 2, 2, 4, 2,
       2, 2, 2, 4, 2, 2, 1, 4, 4, 4, 4, 2, 4, 4, 2, 4, 2, 4, 4, 1, 4, 3,
       4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 4, 4, 2, 2, 4, 4,
       2, 4, 1, 2, 2, 2, 2, 4, 2, 2, 2, 4, 4, 2, 1, 1, 2, 2, 2, 4, 2, 2,
       2, 2, 1, 2, 2, 2, 4, 1, 2, 4, 3, 4, 2, 2, 4, 2, 2, 4, 1, 1, 4, 2,
       1, 2, 2, 2, 2, 4, 4, 2, 4, 2, 2, 2, 2, 1, 2, 1, 4, 2, 2, 2, 2, 2,
       2, 4, 2, 2, 2, 4, 2, 3, 4, 4, 2, 4, 2, 4, 4, 2, 4, 4, 3, 2, 4, 4,
       2, 2, 2, 2, 4, 2, 2, 2, 4, 2, 2, 4, 2, 2, 4, 2, 4, 2, 4, 2, 3, 2,
       2, 2, 2, 1, 1, 1, 2, 4, 2, 4, 2, 2, 2, 2], dtype=int64)
```

[454]:
```
RF1.score(X_train, Y_train)
```

[454]:
```
0.9810617820552623
```

[455]:
```
RF1.score(X_test, Y_test)
```

[455]:
```
0.3225806451612903
```

5.3 Poor Results First Time

Above, we can see that the classifier has a very high score for the training set with an accuracy of 98.1%, but a poor performance on the test set with an accuracy of 32.25%

So, the model has been heavily overfitted on the training data and does not perform well on test data.

To see if this is true for the other two classifiers, we will implement them.

5.4 Implementing the KNN Algorithm

[456]:
```
KNN1 = KNeighborsClassifier()
```

[457]:
```
KNN1.fit(X_train, Y_train)
```

[457]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[458]:
```
KNN1.predict(X_test)
```

[458]:
```
array([2, 2, 2, 2, 1, 4, 4, 1, 4, 2, 4, 3, 2, 4, 2, 2, 2, 1, 2, 4, 2, 3,
       4, 4, 1, 4, 1, 4, 2, 1, 2, 1, 2, 2, 2, 4, 2, 4, 2, 2, 1, 4, 1, 2,
       1, 2, 2, 1, 2, 4, 4, 4, 4, 4, 1, 2, 2, 1, 1, 1, 2, 3, 4, 2, 1, 2,
       4, 4, 2, 1, 2, 4, 4, 4, 2, 2, 1, 2, 2, 2, 4, 2, 2, 2, 2, 4, 4, 4,
       2, 2, 2, 2, 2, 1, 2, 4, 4, 4, 2, 2, 4, 2, 4, 4, 4, 2, 1, 2, 2, 1,
       2, 1, 2, 2, 2, 4, 2, 4, 2, 2, 4, 1, 1, 1, 2, 4, 2, 2, 2, 2, 2, 2,
       4, 2, 2, 2, 1, 2, 2, 4, 2, 2, 2, 1, 4, 2, 1, 3, 4, 1, 2, 2, 2, 2,
       1, 2, 4, 2, 2, 4, 2, 2, 1, 1, 2, 4, 4, 2, 2, 2, 1, 1, 2, 1, 3, 2,
       4, 2, 4, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 1, 2, 2, 2, 2, 3,
       4, 4, 2, 1, 1, 2, 1, 2, 1, 2, 1, 4, 4, 2, 4, 1, 2, 1, 2, 2, 2, 2,
       3, 2, 1, 2, 4, 4, 4, 1, 2, 2, 4, 2, 2, 4, 2, 2, 2, 4, 2, 2, 4, 2,
       2, 2, 4, 4, 2, 2, 1, 4, 4, 2, 1, 1, 2, 4, 2, 4, 4, 2, 2, 2, 2, 1,
       1, 3, 1, 4, 1, 1, 1, 1, 4, 2, 2, 1, 2, 4, 2, 1, 1, 2, 2, 2, 2, 4,
       2, 2, 2, 2, 1, 4, 1, 4, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 4, 4, 2, 2,
       2, 2, 1, 2, 2, 1, 1, 3, 2, 2, 1, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 2,
       1, 2, 2, 1, 4, 2, 4, 1, 2, 2, 1, 2, 2, 4, 2, 3, 2, 1, 1, 2, 2, 2,
       4, 4, 1, 2, 2, 4, 4, 4, 2, 2, 1, 2, 1, 2, 3, 2, 2, 1, 2, 2, 4, 1,
       3, 2, 1, 4, 2, 2, 2, 1, 1, 2, 1, 4, 1, 3, 4, 1, 2, 4, 2, 1, 1, 4,
       4, 2, 4, 4, 2, 3, 2, 2, 2, 2, 1, 2, 4, 1, 2, 4, 4, 2, 1, 2, 2, 2,
       4, 2, 4, 2, 4, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 4, 2,
```

```
       2, 4, 2, 2, 1, 4, 2, 2, 2, 1, 1, 2, 1, 2, 1, 4, 4, 2, 2, 1, 2, 2,
       4, 2, 4, 2, 2, 4, 4, 2, 2, 2, 2, 2, 2, 1, 2, 3, 2, 2, 2, 2, 1, 2,
       3, 2, 2, 1, 4, 2, 1, 2, 2, 2, 1, 4, 4, 2, 2, 2, 2, 2, 1, 4, 2, 2,
       1, 2, 2, 4, 2, 4, 1, 1, 2, 1, 2, 3, 2, 2, 3, 4, 4, 1, 1, 2, 4, 1,
       2, 2, 1, 3, 2, 4, 2, 4, 2, 4, 2, 2, 3, 2, 4, 4, 3, 4, 2, 2, 4, 1,
       1, 2, 2, 4, 3, 2, 2, 3, 4, 2, 2, 4, 4, 3, 4, 1, 1, 4, 2, 2, 4, 2,
       4, 2, 1, 2, 4, 4, 2, 2, 1, 1, 2, 1, 1, 1, 4, 4, 1, 2, 2, 2, 2, 4,
       2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 4, 2, 1, 2, 4, 1, 2, 2, 1, 2, 2,
       4, 2, 2, 3, 2, 1, 2, 2, 2, 1, 4, 2, 3, 1, 2, 2, 1, 2, 2, 2, 2, 2,
       2, 2, 2, 3, 2, 2, 2, 1, 2, 2, 2, 2, 4, 1, 2, 1, 1, 4, 4, 1, 4, 4,
       4, 1, 2, 4, 4, 4, 4, 2, 2, 4, 1, 2, 1, 2, 2, 4, 1, 1, 2, 2, 2, 3,
       2, 4, 1, 2, 2, 4, 2, 2, 2, 2, 2, 1, 4, 2, 1, 2, 1, 2, 1, 2, 4, 4,
       2, 4, 4, 1, 2, 2, 2, 4, 2, 2, 2, 2, 1, 2, 1, 2, 2, 1, 2, 3, 3, 4,
       1, 4, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1, 4, 2, 3, 4, 1, 2, 2, 2,
       2, 2, 2, 1, 1, 2, 4, 3, 4, 1, 4, 2, 2, 4, 4, 2, 2, 4, 4, 3, 2, 2,
       4, 2, 1, 2, 4, 2, 2, 2, 1, 3, 2, 2, 2, 1, 4, 2, 2, 2, 1, 4, 2, 4,
       1, 1, 4, 1, 2, 1, 1, 4, 2, 1, 4, 2, 2, 2], dtype=int64)
```

[459]:
```
KNN1.score(X_train, Y_train)
```

[459]:
```
0.5336851909344924
```

[460]:
```
KNN1.score(X_test, Y_test)
```

[460]:
```
0.30893300248138955
```

So, the KNN algorithm performs even more poorly than the Random Forest.

Lastly, we will implement the Support Vector Machine.

5.5 Implementing the SVM algorithm

[461]:
```
SVM1 = SVC(kernel = 'rbf', gamma = 0.2)
```

[462]:
```
SVM1.fit(X_train, Y_train)
```

[462]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.2, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[463]:
```
SVM1.predict(X_test)
```

[463]:
```
array([2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 1, 2,
       4, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 4, 2, 2, 2, 2, 2,
       2, 2, 2, 4, 2, 2, 2, 2, 4, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 4, 2,
       2, 2, 2, 2, 2, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

```
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 4, 1, 2, 2, 2, 2, 2,
4, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 4, 1, 4, 2, 2, 2,
4, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 1, 4, 2, 2, 2, 2,
2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 4, 2, 2, 2, 2, 2, 4, 2,
2, 2, 2, 2, 4, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2,
2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2,
2, 4, 2, 4, 2, 2, 2, 1, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 2, 2, 1, 2, 2,
2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 4, 2, 2, 1, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 4, 4, 2, 2, 4, 2, 4, 2, 2, 2, 2,
2, 2, 2, 4, 2, 4, 2, 4, 2, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 2, 1, 2, 2,
2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 4, 2, 2, 2, 2,
2, 2, 2, 1, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2,
2, 1, 2, 2, 2, 4, 2, 2, 2, 2, 2, 3, 2, 2, 2, 4, 2, 2, 4, 2, 2, 2,
2, 2, 2, 2, 2, 4, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 3, 2, 2, 4, 4, 2,
2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
4, 2, 2, 2, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 2, 4, 2, 2, 4,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 1, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 4, 4,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 4, 2, 2, 2, 2, 4, 4, 4, 2, 1, 2, 2, 2, 4, 2, 2, 2, 2, 2,
2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 2, 4, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 4, 2, 2, 2,
2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2, 2, 4, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=int64)
```

[464]:
```
SVM1.score(X_train, Y_train)
```

[464]:
```
0.9382179447376591
```

[465]:
```
SVM1.score(X_test, Y_test)
```

[465]:
```
0.36228287841191065
```

So, the SVM algorithm has the best results for the test set with 36.22%, but this is still very poor.

The gamma parameter was adjusted to get the best score on the training set - results ranged from ~30% to 36.22%

6 Second Iteration of Classifiers

6.1 Normalising the features

There may be an improvement in the model performance if we use Scikit's Standard Scaler method to normalise the features before training the models with them.

[466]:
```
scaler = StandardScaler()
```

[467]:
```
normed_X_train = scaler.fit_transform(X_train)

normed_X_test = scaler.fit_transform(X_test)
```

[468]:
```
RF1.fit(normed_X_train, Y_train)
```

[468]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[469]:
```
RF1.score(normed_X_train, Y_train)
```

[469]:
```
0.9810617820552623
```

[470]:
```
RF1.score(normed_X_test, Y_test)
```

[470]:
```
0.3188585607940447
```

[471]:
```
KNN1.fit(normed_X_train, Y_train)
```

[471]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[472]:
```
KNN1.score(normed_X_train, Y_train)
```

[472]:
```
0.5442409189692642
```

[473]:
```
KNN1.score(normed_X_test, Y_test)
```

[473]:
```
0.3200992555831266
```

[474]:
```
SVM1.fit(normed_X_train, Y_train)
```

[474]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.2, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[475]:
```
SVM1.score(normed_X_train, Y_train)
```

[475]:
```
0.4386836386215461
```

[476]:
```
SVM1.score(normed_X_test, Y_test)
```

[476]:
```
0.3759305210918114
```

6.2 Slight Improvement with Normalisation

The improvement in score on the test features is as follows:

1. RandomForest - 0.37%
2. KNN + 1.11%
3. SVM + 1.36%

So, this improvememt is marginal. More will have to be done to improve the model score.

As the net gain in classifier accuracy is 2.1%, when fitting the models with training data from this point on the data will be normalised.

7 Selecting Another Feature

To try and improve the performance of the models, we will select another feature from the DataFrame that will most likely have a large impact on the outcome of the classfication.

A feature that will most likely have a large impact on the classification is the 'finish_round_time' column.

This column has the time in the round at the end of the fight.

If the fight "goes the distance", that is, there is no KO/TKO or submission, the fight time will be '05:00' as this is the duration of a round.

[477]:
```
df['finish_round_time']
```

[477]:
```
0       04:48
1       01:18
2       05:00
3       01:28
4       05:00
        ...
4287    00:44
4288    02:01
4289    00:47
```

```
4290    05:00
4291    05:00
Name: finish_round_time, Length: 4027, dtype: object
```

[478]:
```
df['finish_round_time'].dtype
```

[478]:
```
dtype('O')
```

7.1 Addressing Feature Problems

So, there are a couple of issues to sort before this feature can be used.

Firstly, there are colons in each of the records, which will need to be removed.

Secondly, the datatype is 'object' - this will have to be converted to 'int' for the analysis.

[479]:
```
# Function that will remove colons from the records:

def remove_colon(obj):
    obj = str(obj)
    obj = obj.replace(':', '')
    return obj
```

[480]:
```
# creating a new column in the DataFrame that will be the same as the␣
↪finish_round_time
# but will have no punctuation

df['finish_time'] = df['finish_round_time'].apply(remove_colon)
```

[481]:
```
df['finish_time']
```

[481]:
```
0       0448
1       0118
2       0500
3       0128
4       0500
        …
4287    0044
4288    0201
4289    0047
4290    0500
4291    0500
Name: finish_time, Length: 4027, dtype: object
```

So, we have addressed one issue, but the datatype is still 'object'. This will be addressed next.

[482]:
```
# a lambda function can convert each entry into an integer
```

```
make_int = lambda obj: int(obj)
```

[483]:
```
df['finish_time'] = df['finish_time'].apply(make_int)
```

[484]:
```
df['finish_time']
```

[484]:
```
0       448
1       118
2       500
3       128
4       500
       ...
4287     44
4288    201
4289     47
4290    500
4291    500
Name: finish_time, Length: 4027, dtype: int64
```

[485]:
```
df['finish_time'].dtype
```

[485]:
```
dtype('int64')
```

So, the finish_time column has been converted successfully to integers.

8 Revisiting the Models - Third Iteration of Classifiers

8.1 Adding the new finish_time feature to each classifier

The training and test data will be normalised again as this provided a small increase in accuracy previously

[486]:
```
new_features = df[['odds_dif',
                   'abs_round_dif',
                   'abs_ko_dif',
                   'abs_sig_str_dif',
                   'abs_sub_att_dif',
                   'abs_td_dif',
                   'finish_time']]
```

[487]:
```
X_train, X_test, Y_train, Y_test = train_test_split(new_features, target,␣
↪train_size = 0.8, test_size = 0.2)
```

[488]:
```
new_normed_X_train = scaler.fit_transform(X_train)

new_normed_X_test = scaler.fit_transform(X_test)
```

8.2 Creating a new Random Forest Classifier

This will be a new model to see what improvement has been made on the original model.

[489]:
```
RF2 = RandomForestClassifier()
```

[490]:
```
RF2.fit(new_normed_X_train, Y_train)
```

[490]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[491]:
```
RF2.score(new_normed_X_train, Y_train)
```

[491]:
```
0.9965849115181621
```

[492]:
```
RF2.score(new_normed_X_test, Y_test)
```

[492]:
```
0.6674937965260546
```

8.3 Creating a new KNN Classifier

[493]:
```
KNN2 = KNeighborsClassifier()
```

[494]:
```
KNN2.fit(new_normed_X_train, Y_train)
```

[494]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[495]:
```
KNN2.score(new_normed_X_train, Y_train)
```

[495]:
```
0.721825520024837
```

[496]:
```
KNN2.score(new_normed_X_test, Y_test)
```

[496]:
```
0.5905707196029777
```

8.4 Creating a new SVM Classifier

[497]:
```
SVM2 = SVC(kernel = 'rbf', gamma = 0.1)
```

[498]:
```
SVM2.fit(new_normed_X_train, Y_train)
```

[498]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[499]:
```
SVM2.score(new_normed_X_train, Y_train)
```

[499]:
```
0.6979199006519714
```

[500]:
```
SVM2.score(new_normed_X_test, Y_test)
```

[500]:
```
0.6687344913151365
```

9 Improvements on the Models

Adding this new feature to the model data has had a considerable impact on the model accuracy:

1. Random Forest + 34.86%
2. KNN + 27.05%
3. SVM + 29.28%

This suggests that the models are leaning heavily on this feature to make their prediction

The RandomForestClassifier has a feature_importances_ method which can show how much it is using each feature to form a decision boundary at each node in the trees.

[501]:
```
RF2.feature_importances_
```

[501]:
```
array([0.1125878 , 0.08224081, 0.03688285, 0.11380877, 0.0840908 ,
       0.09387729, 0.47651167])
```

The last float in the above array is the proportion of the importance of the final feature in the data i.e. the finish_time feature.

It can be seen that it is considerably larger than the other floats, and so is more important in classification.

10 Adjusting the Features Further - Fourth Iteration of Classifiers

So, we are almost at 70% classification accuracy with the Random Forest Classifier and the Support Vector Classifier.

To see if keeping these features and adding more will increase the classification accuracy beyond 70%, two more features are selected for input to the algorithms:

1. “R_ev” - The money returned from a successful 100 US Dollar bet on the “Red fighter”
2. “B_ev” - The money returned from a successful 100 US Dollar bet on the “Blue fighter”

[502]:
```
newest_features = df[['odds_dif',
                      'abs_round_dif',
                      'abs_ko_dif',
                      'abs_sig_str_dif',
                      'abs_sub_att_dif',
                      'abs_td_dif',
                      'finish_time',
                      'R_ev',
                      'B_ev']]
```

[503]:
```
X_train, X_test, Y_train, Y_test = train_test_split(newest_features, target,␣
↪train_size = 0.8, test_size = 0.2)
```

[504]:
```
# normalising the features before fitting the model with them

newest_normed_X_train = scaler.fit_transform(X_train)

newest_normed_X_test = scaler.fit_transform(X_test)
```

10.1 Creating A New Random Forest Classifier

[505]:
```
RF3 = RandomForestClassifier()
```

[506]:
```
RF3.fit(newest_normed_X_train, Y_train)
```

[506]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[507]:
```
RF3.score(newest_normed_X_train, Y_train)
```

[507]:
```
0.9981372244644521
```

[508]:
```
RF3.score(newest_normed_X_test, Y_test)
```

[508]:
```
0.6935483870967742
```

10.2 Creating A New KNN Classifier

[509]:
```
KNN3 = KNeighborsClassifier()
```

[510]:
```
KNN3.fit(newest_normed_X_train, Y_train)
```

[510]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

[511]:
```
KNN3.score(newest_normed_X_train, Y_train)
```

[511]:
```
0.6988512884197454
```

[512]:
```
KNN3.score(newest_normed_X_test, Y_test)
```

[512]:
```
0.6129032258064516
```

10.3 Creating A New SVM Classifier

[513]:
```
SVM3 = SVC(kernel= 'rbf', gamma=0.1)
```

[514]:
```
SVM3.fit(newest_normed_X_train, Y_train)
```

[514]:
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[515]:
```
SVM3.score(newest_normed_X_train, Y_train)
```

[515]:
```
0.6945048121701335
```

[516]:
```
SVM3.score(newest_normed_X_test, Y_test)
```

[516]:
```
0.7009925558312655
```

11 Results

There is a decrease in classification accuracy across all three classifier types with the addition of the final two features.

1. Random Forest - 2.61%
2. KNN - 2.24%
3. SVM - 3.22%

The Support Vector algorithm reached the maximum classification accuracy on the fourth iteration with 70.09% accuracy.

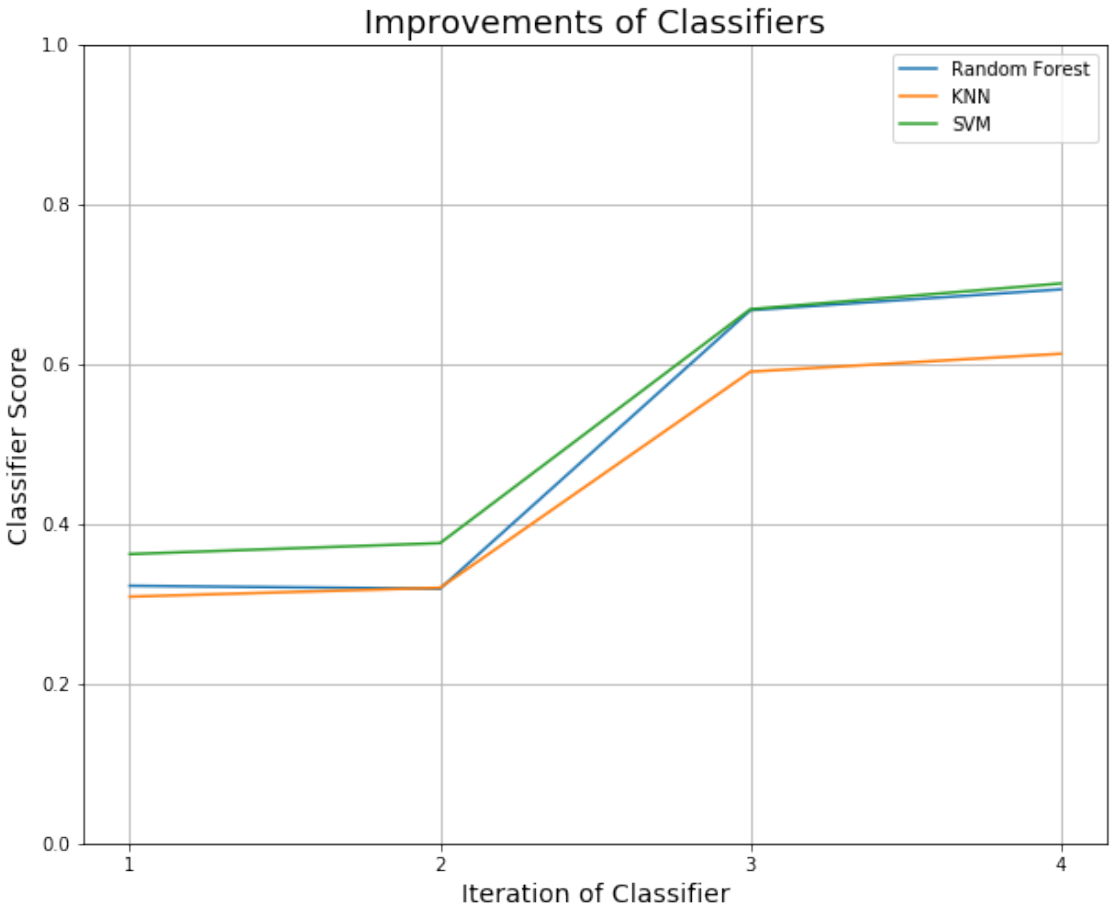
The second highest classification accuracy was the Random Forest algorithm on the fourth iteration with 69.35%.

The third highest classification accuracy was the Support Vector algorithm on the third iteration with 66.87%/

The maximum classification accuracy achieved by the K-Nearest Neighbors algorithm was on the third iteration with 61.29%

11.1 Plotting the delta accuracy with each iteration

[518]:
```
x = [1, 2, 3, 4]

plt.figure(figsize=(10, 8))

plt.grid()

plt.plot(x, [0.3225, 0.3188, 0.6674, 0.6935])
plt.plot(x, [0.3089, 0.32, 0.5905, 0.6129])
plt.plot(x, [0.3622, 0.3759, 0.6687, 0.7009])
plt.ylim(bottom = 0, top=1)

plt.legend(['Random Forest', 'KNN', 'SVM'])
plt.title('Improvements of Classifiers', fontsize=18)
plt.xlabel('Iteration of Classifier', fontsize=14)
plt.ylabel('Classifier Score', fontsize=14)
plt.xticks(np.arange(1, 5, step=1))

plt.show()
```


Improvements of Classifiers
Random Forest
KNN
SVM
Classifier Score
Iteration of Classifier
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4

Predictive Analysis of UFC Matches

Ste-Ro

Abstract

This project uses the “Ultimate UFC Dataset” provided by mdabbert via Kaggle, and has data on all fights from 2010-July 2020.

The investigation aims to find the probability of a fight ending in one of four ways, whether machine learning algorithms can use the data in this dataset to predict with ≥ 70% accuracy which of these four ways a fight will end, and compare the performance of three different classification algorithms when implemented in this classification task.

The methods employed in this project are: feature selection and engineering, algorithm selection, training and optimisation, and performance analysis using Random Forests, K-Nearest Neighbors, and Support Vector Machines.

It is found that the probability of a fight outcome, from Submission, to Unanimous Decision, to Split Decision, to KO/TKO is 18.8%, 38.3%, 10.8%, and 32.1% respectively. It is also possible to predict the outcome of a fight with an accuracy greater than 70% using a Support Vector Machine classifier - and this can be deemed the best suited classifier for this task.

Random Forests, however, are only marginally less effective at classifying the fight outcomes.

K-Nearest Neigbor classifiers consistently perform the worst when attempting this classification task.

Motivation

For any competitive sport, predicting the likelihood of some kind of outcome at the end of the competition would be useful in placing bets on the outcome of the fight. Higher probabilities of one type of outcome given the stats of two opposing fighters could inform betting and improve returns.

Given the data in the dataset and the knowledge gained from DSE200x, can this be done?

Consumers of the sport of MMA, or anyone looking to capitalise on understanding the probabilities of UFC match outcomes, could use such a classification algorithm in order to make money given they have greater insight into how the fight may end.

Dataset

The dataset used in this project is the “Ultimate UFC Dataset”. This Dataset is 1.7MB and is primarily the work of Rajeev Warrier but has been augmented and added to by mdabbert, who has posted it on Kaggle.

There are just under 4,300 records in the dataset and 113 rows. The dataset contains data on all fights from 2010 until July 2020, with the columns mostly containing data on the fighters’ stats (e.g. longest winning streak, strike differential between the fighters in the given match, etc.) but also contains columns with betting odds for each fighter, the way in which each fight ends, etc.

Data Preparation and Cleaning

Forutnately, the dataset is largely complete and clean, although there are some records with null values for the column with the data on the way in which the fight ends. These records are removed in the dataset as they mainly occur when the fight was cancelled.

Some datatype augumentation is also required in the investigation – where one column contains time data stored as an object in the form “xx:xx”. This column must have the colons removed from the records and have each record transformed to integers for use in the classification algorithms.

The fight outcome column also must be augmented. The outcomes are stored as strings and there are four types used in the investigation – “U-DEC”, “S-DEC”, “SUB”, “KO/TKO”. These outcomes are mapped to integers to be used in the analysis, where the algorithms are attempting to classify the fights as being a number from 1 to 4, each representing one of the outcomes.

Also, some columns are added to aggregate/consolidate data from other columns. To reduce the number of features needed for the classifiers, columns are inserted with differential statistics between two fighters in each fight. For example, a column is added and utilised that has the difference between the betting odds for the fighters, as opposed to using the individual odds for each fighter.

Research Questions

- Given the data on all fights from 2010-2020, what is the probability of fights ending in each of the four ways considered in the investigation?
- Is it possible to use data in the “Ultimate UFC Dataset” to predict with an accuracy of 70% or greater the way in which UFC matches will end?
- With the above two questions answered, which of the three algorithms provides the highest classification accuracy for the given task? Which performs the worst?

Methods

This project uses classification algorithms to classify fights as ending as either 1, 2, 3, or 4. Each of these integers maps to a type of finish, as mentioned previously. Therefore, all inputs to the algorithms are ensured to be numeric.

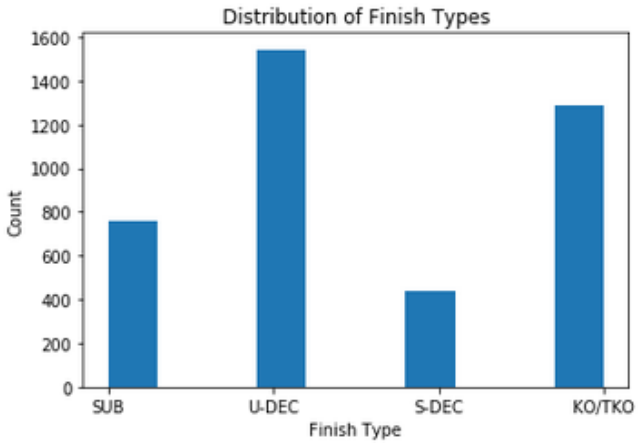
Firstly, the proportion of fights ending in each of the four ways was investigated (after mapping the fight outcomes to numbers) using pandas' .value_counts() method. The results are also visualized. This shows the proportion of fights ending each way, and the probability of a fight ending in each of the four ways.

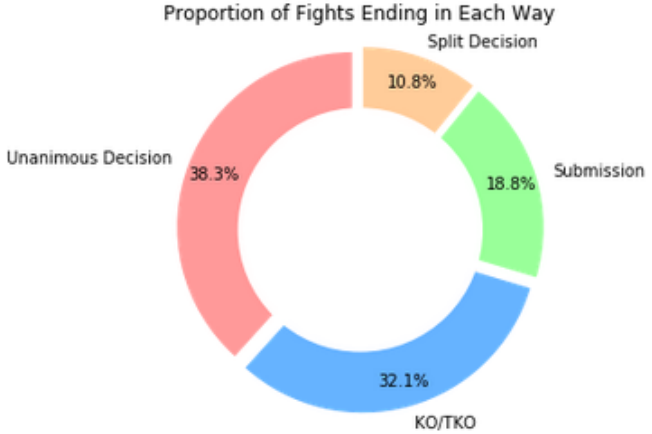
Then, an iterative approach is adopted for dealing with the classification algorithms. This consisted of selecting features, ensuring their usability, implementing them in each of the three algorithms, assessing the performance, and then using this to inform the next iteration. Four iterations are adopted:

Iteration	# Features	Normalized?
1	6	N
2	6	Y
3	7	Y
4	9	Y

The results are then visualized for all four iterations, with each classifier's accuracy plotted on a single line graph for the four iterations. This provides an assessment of whether the goal of the project is achieved, and which algorithm provides the best classification result.

Findings (1/3)



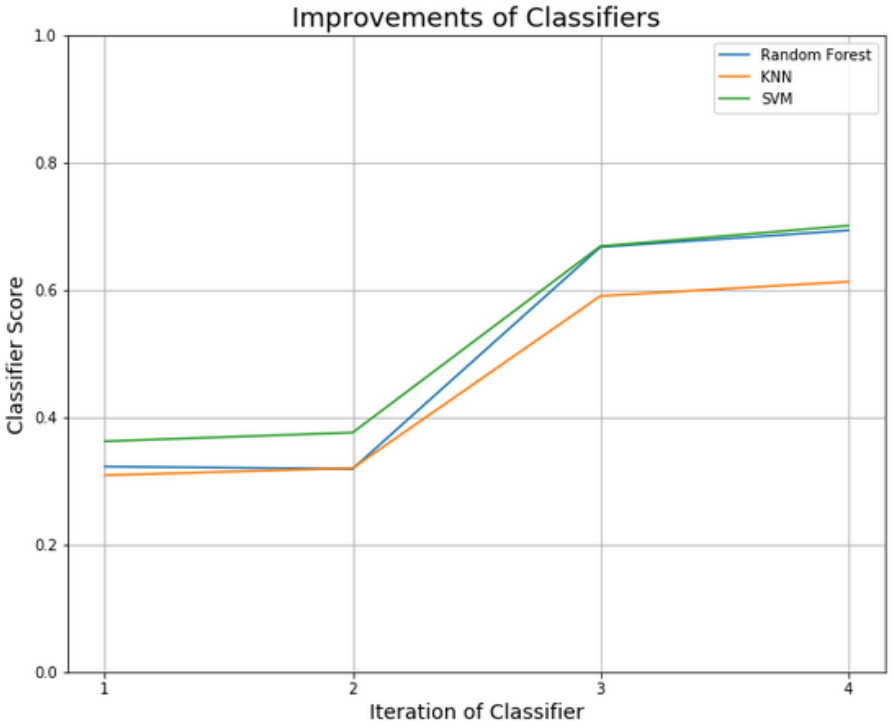


As can be seen, the most common type of finish to a fight is by unanimous decision, followed by KO/TKO.

Both of these types of finish are significantly more likely than either a submission or split decision, and over 70% of all fights will end in one of these two ways.

49.1% of fights end with either a unanimous or split decision, meaning that over half of all fights will end before the full available fight time has elapsed.

Findings (2/3)



Iteration	Test Accuracy	Δ Accuracy
Random Forest Algorithm		
1	32.25%	N/A
2	31.88%	- 0.37%
3	66.74%	+ 34.86%
4	69.35%	+ 2.61%
K-Nearest Neighbours Algorithm		
1	30.89%	N/A
2	32.00%	+ 1.11%
3	59.05%	+ 27.05%
4	61.29%	+ 2.24%
Support Vector Machine Algorithm		
1	36.22%	N/A
2	37.59%	+ 1.37%
3	66.87%	+ 29.28%
4	70.09%	+ 3.22%

Findings (3/3)

The classifiers improve with every iteration, with the exception of a minor reduction in the Random Forest model in the second iteration.

The features are chosen to be normalised after the second iteration as a net gain of 2.1% accuracy is achieved across the models when the features are normalised.

It is possible to achieve a prediction accuracy of ≥ 70% using the chosen dataset.

The SVM algorithm achieves an accuracy of over 70% and so can be considered the best-suited algorithm to this classification task.

The difference between the SVM and RF algorithms is marginal, with the RF model very nearly exceeding an accuracy of 70% on the fourth iteration.

The KNN algorithm is clearly the poorest performing model of the three in this investigation.

Limitations

Very few records existed where the fight outcome was overturned and so the “finish type” column had a null value. These records were therefore removed, but the probabilities of fight outcomes would nonetheless be slightly affected by the addition of this outcome.

Also, a small number of records were incomplete as the fights were cancelled but were still included in the dataset. These would also have slightly impacted the probabilities of fight outcomes, but were removed as there were so few of them.

The train_test_split algorithm is random in the way it splits the data. This means that the results in this investigation are not absolute, and it is possible that if re-run the Random Forest model could perform better than the SVM model simply due to the way the data is split into training and test sets via train_test_split.

The classification algorithms also give a different output each time they are run due to the way they choose to create a decision boundary changing each time. This means that, particularly since the outcomes of the Random Forest and SVM models are so similar, if they were re-run the accuracy of each classifier would be different and perhaps the Random Forest would perform better.

Ultimately, the algorithms use a post-finish feature of the fights in order to accuracy classify how the fight ended. That is, the time at which the fight ended is used in the model. This limits the models as predictive agents.

Conclusions

- The probabilities of a fight ending each way are as follows:

Finish Type	Probability
Submission	18.8%
U-Decision	38.3%
S-Decision	10.8%
KO/TKO	32.1%

- It is possible to predict the way in which a fight ends with an accuracy of 70.09% using a Support Vector Machine classifier.

- The hierarchy for efficacy in predicting the type of outcome of a UFC match in this investigation is as follows:

ML Algorithm	Rank
Support Vector Machine	1
Random Forest	2
K-Nearest Neighbour	3

Acknowledgements

I would like to acknowledge Kaggle and both Rajeev Warrier and mdabbert for their provision of the dataset used in this investigation.

The entirety of the work in this project is my own, and it was not shown to anyone before submission.

References

All of the work in this project is my own.