## Importing Relevant Libraries

In [3]:

```
!pip3 install -r requirements.txt
```

```
Requirement already satisfied: prettytable==3.3.0 in /root/venv/lib/pyth
3.7/site-packages (from -r requirements.txt (line 1)) (3.3.0)
Requirement already satisfied: statsmodels==0.13.2 in /root/venv/lib/pytho
n3.7/site-packages (from -r requirements.txt (line 2)) (0.13.2)
Requirement already satisfied: scikit-plot in /root/venv/lib/python3.7/sit
e-packages (from -r requirements.txt (line 3)) (0.3.7)
Requirement already satisfied: xgboost==1.6.1 in /root/venv/lib/python3.7/
site-packages (from -r requirements.txt (line 4)) (1.6.1)
Collecting pydotplus==2.0.2
  Downloading pydotplus-2.0.2.tar.gz (278 kB)
     |████████████████████████████████| 278 kB 7.7 MB/s
Requirement already satisfied: wcwidth in /shared-libs/python3.7/py-core/l
ib/python3.7/site-packages (from prettytable==3.3.0->-r requirements.txt
(line 1)) (0.2.5)
Requirement already satisfied: importlib-metadata; python_version < "3.8"
in /shared-libs/python3.7/py-core/lib/python3.7/site-packages (from pretty
table==3.3.0->-r requirements.txt (line 1)) (4.11.4)
Requirement already satisfied: numpy>=1.17 in /shared-libs/python3.7/py/li
b/python3.7/site-packages (from statsmodels==0.13.2->-r requirements.txt
(line 2)) (1.21.6)
Requirement already satisfied: pandas>=0.25 in /shared-libs/python3.7/py/l
ib/python3.7/site-packages (from statsmodels==0.13.2->-r requirements.txt
(line 2)) (1.2.5)
Requirement already satisfied: scipy>=1.3 in /shared-libs/python3.7/py/li
b/python3.7/site-packages (from statsmodels==0.13.2->-r requirements.txt
(line 2)) (1.7.3)
Requirement already satisfied: packaging>=21.3 in /shared-libs/python3.7/p
y/lib/python3.7/site-packages (from statsmodels==0.13.2->-r requirements.t
xt (line 2)) (21.3)
Requirement already satisfied: patsy>=0.5.2 in /root/venv/lib/python3.7/si
te-packages (from statsmodels==0.13.2->-r requirements.txt (line 2)) (0.5.
2)
Requirement already satisfied: scikit-learn>=0.18 in /shared-libs/python3.
7/py/lib/python3.7/site-packages (from scikit-plot->-r requirements.txt (l
ine 3)) (1.0.2)
Requirement already satisfied: joblib>=0.10 in /shared-libs/python3.7/py/l
ib/python3.7/site-packages (from scikit-plot->-r requirements.txt (line
3)) (1.1.0)
Requirement already satisfied: matplotlib>=1.4.0 in /shared-libs/python3.
7/py/lib/python3.7/site-packages (from scikit-plot->-r requirements.txt (l
ine 3)) (3.5.2)
Requirement already satisfied: pyparsing>=2.0.1 in /shared-libs/python3.7/
py/lib/python3.7/site-packages (from pydotplus==2.0.2->-r requirements.txt
(line 5)) (3.0.9)
Requirement already satisfied: zipp>=0.5 in /shared-libs/python3.7/py-cor
e/lib/python3.7/site-packages (from importlib-metadata; python_version <
"3.8"->prettytable==3.3.0->-r requirements.txt (line 1)) (3.8.0)
Requirement already satisfied: typing-extensions>=3.6.4; python_version <
"3.8" in /shared-libs/python3.7/py-core/lib/python3.7/site-packages (from
importlib-metadata; python_version < "3.8"->prettytable==3.3.0->-r require
ments.txt (line 1)) (4.2.0)
Requirement already satisfied: python-dateutil>=2.7.3 in /shared-libs/pyth
on3.7/py-core/lib/python3.7/site-packages (from pandas>=0.25->statsmodels=
=0.13.2->-r requirements.txt (line 2)) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /shared-libs/python3.7/py/l
ib/python3.7/site-packages (from pandas>=0.25->statsmodels==0.13.2->-r req
uirements.txt (line 2)) (2022.1)
```

```
Requirement already satisfied: six in /shared-libs/python3.7/py-core/lib/p
ython3.7/site-packages (from patsy>=0.5.2->statsmodels==0.13.2->-r require
ments.txt (line 2)) (1.16.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /shared-libs/python
3.7/py/lib/python3.7/site-packages (from scikit-learn>=0.18->scikit-plot->
-r requirements.txt (line 3)) (3.1.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /shared-libs/python3.
7/py/lib/python3.7/site-packages (from matplotlib>=1.4.0->scikit-plot->-r
requirements.txt (line 3)) (1.4.2)
Requirement already satisfied: pillow>=6.2.0 in /shared-libs/python3.7/py/
lib/python3.7/site-packages (from matplotlib>=1.4.0->scikit-plot->-r requi
rements.txt (line 3)) (9.1.1)
Requirement already satisfied: fonttools>=4.22.0 in /shared-libs/python3.
7/py/lib/python3.7/site-packages (from matplotlib>=1.4.0->scikit-plot->-r
requirements.txt (line 3)) (4.33.3)
Requirement already satisfied: cycler>=0.10 in /shared-libs/python3.7/py/l
ib/python3.7/site-packages (from matplotlib>=1.4.0->scikit-plot->-r requir
ements.txt (line 3)) (0.11.0)
Building wheels for collected packages: pydotplus
  Building wheel for pydotplus (setup.py) ... done
  Created wheel for pydotplus: filename=pydotplus-2.0.2-py3-none-any.whl s
ize=24566 sha256=3a63176aa6199b2787d2506e1b5feb311ce36875121b1be60eda53751
4a5a01a
  Stored in directory: /root/.cache/pip/wheels/1e/7b/04/7387cf6cc9e48b4a96
e361b0be812f0708b394b821bf8c9c50
Successfully built pydotplus
Installing collected packages: pydotplus
Successfully installed pydotplus-2.0.2
WARNING: You are using pip version 20.1.1; however, version 22.1.2 is avai
lable.
You should consider upgrading via the '/root/venv/bin/python -m pip instal
l --upgrade pip' command.
```

In [4]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
from sklearn.preprocessing import OneHotEncoder,LabelEncoder,StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel, SelectKBest,mutual_info_classif,chi2
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split,GridSearchCV,RandomizedSearchCV
from keras.wrappers.scikit_learn import KerasClassifier
from keras.models import Sequential
from keras.layers import Dense
import pydotplus
import keras
from keras.utils import plot_model
keras.utils.vis_utils.pydot = pydotplus
import tensorflow as tf
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score,confusion_matrix,roc_auc_score,recall_score,prec
from statsmodels.stats.outliers_influence import variance_inflation_factor
import xgboost
import scikitplot as skplt
from prettytable import PrettyTable
import warnings
import datetime
warnings.filterwarnings('ignore')
```

## (1) Data Pre-processing and exploratory analysis

In [5]:

```
data = pd.read_csv("data.csv")
```

In [6]:

```
data.head()
```

Out[6]:

| | R_fighter | B_fighter | Referee | date | location | Winner | title_bout | weight_class | B_av |
|---|---|---|---|---|---|---|---|---|---|
| **0** | Adrian Yanez | Gustavo Lopez | Chris Tognoni | 2021-03-20 | Las Vegas, Nevada, USA | Red | False | Bantamweight | |
| **1** | Trevin Giles | Roman Dolidze | Herb Dean | 2021-03-20 | Las Vegas, Nevada, USA | Red | False | Middleweight | |
| **2** | Tai Tuivasa | Harry Hunsucker | Herb Dean | 2021-03-20 | Las Vegas, Nevada, USA | Red | False | Heavyweight | |
| **3** | Cheyanne Buys | Montserrat Conejo | Mark Smith | 2021-03-20 | Las Vegas, Nevada, USA | Blue | False | WomenStrawweight | |
| **4** | Marion Reneau | Macy Chiasson | Mark Smith | 2021-03-20 | Las Vegas, Nevada, USA | Blue | False | WomenBantamweight | |

5 rows × 144 columns

## (1.1) Removing columns that are unique identifiers

We can clearly see that columns which are unique identifiers such as the R_figher, B_fighter,referee, date and location should be removed for generalisability of the model since they are just identifiers.

In [7]:

```
columns_to_drop = ["R_fighter","B_fighter","Referee","date","location"]
data = data.drop(columns=columns_to_drop)
```

## (1.2) Dealing with NaN values

We can also see that there are some NaN values present.
We are going to see what percentage of the dataset contains these NaN values

In [8]:

```
number_of_nan_values = data.isna().any(axis=1).sum()
print(f"The number of rows containing a NaN value is: {number_of_nan_values}")
number_of_datapoints = np.shape(data)[0]
print(f"The number of rows in total is: {number_of_datapoints}")
percentage_of_data_missing_values = number_of_nan_values/number_of_datapoints
print(f"The percentage of the dataset that contains null values is: {percentage_of_data_mis
```

```
The number of rows containing a NaN value is: 2098
The number of rows in total is: 6012
The percentage of the dataset that contains null values is: 34.8968729208250
2 %
```

## Finding all the columns which contain NaN values

In [9]:

```
for column in data.columns:
    if data[column].isnull().sum()!=0:
        print(f"Number of NaN's in {column}: {data[column].isnull().sum()}")
```

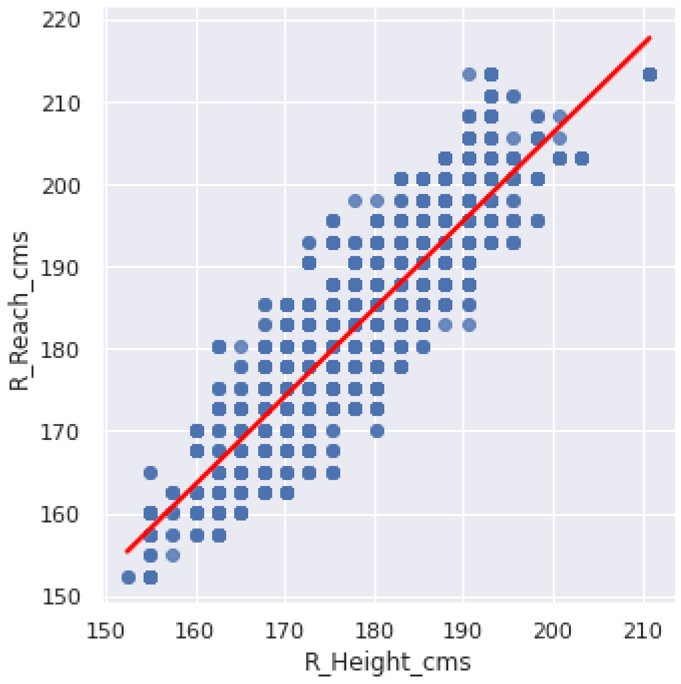
```
Number of NaN's in R_avg_opp_TOTAL_STR_landed: 712
Number of NaN's in R_avg_TD_att: 712
Number of NaN's in R_avg_TD_landed: 712
Number of NaN's in R_avg_opp_TD_att: 712
Number of NaN's in R_avg_opp_TD_landed: 712
Number of NaN's in R_avg_HEAD_att: 712
Number of NaN's in R_avg_HEAD_landed: 712
Number of NaN's in R_avg_opp_HEAD_att: 712
Number of NaN's in R_avg_opp_HEAD_landed: 712
Number of NaN's in R_avg_BODY_att: 712
Number of NaN's in R_avg_BODY_landed: 712
Number of NaN's in R_avg_opp_BODY_att: 712
Number of NaN's in R_avg_opp_BODY_landed: 712
Number of NaN's in R_avg_LEG_att: 712
Number of NaN's in R_avg_LEG_landed: 712
Number of NaN's in R_avg_opp_LEG_att: 712
Number of NaN's in R_avg_opp_LEG_landed: 712
Number of NaN's in R_avg_DISTANCE_att: 712
Number of NaN's in R_avg_DISTANCE_landed: 712
```

## Looking at the relationship between reach and height in order to fill in these missing values

### Red Corner

In [10]:

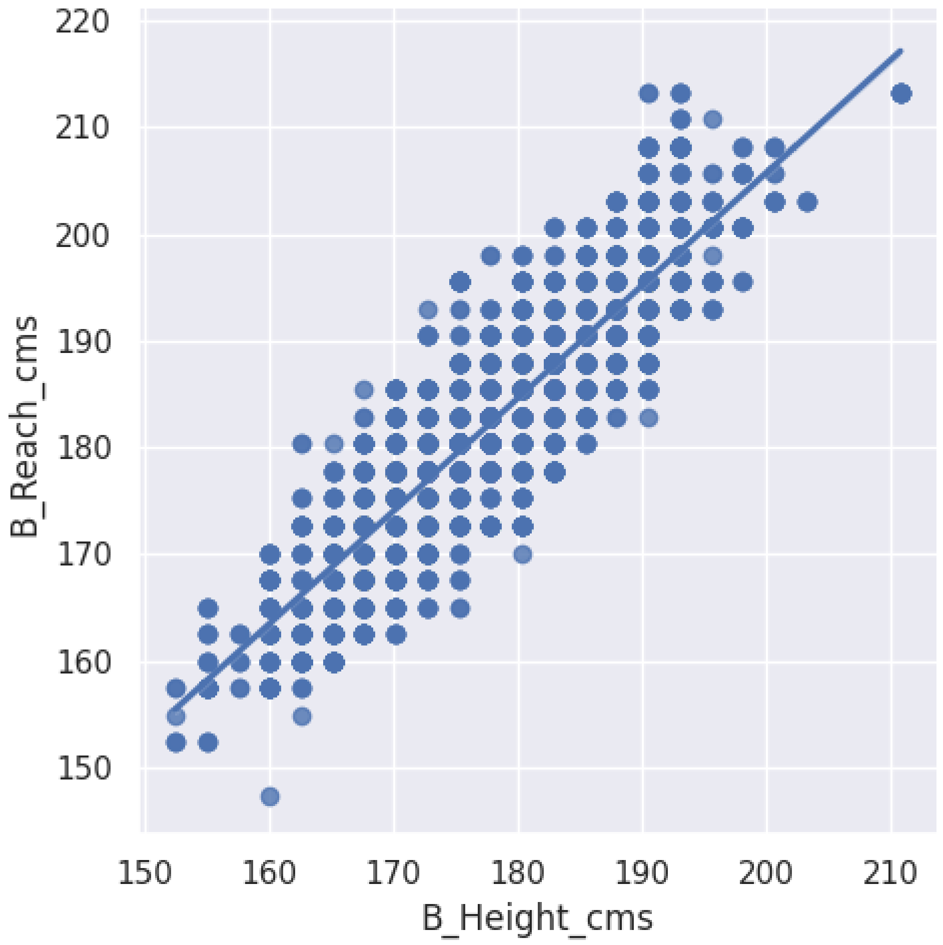
```
sns.lmplot(x = 'R_Height_cms', y = 'R_Reach_cms', data=data, fit_reg=True,line_kws={'color'
plt.show()
```



**Blue Corner**

In [11]:

```
sns.lmplot(x = 'B_Height_cms', y = 'B_Reach_cms', data=data, fit_reg=True)
plt.show()
```



- We see the correlation between a fighters height and their reach, so we can use the height to fill in the missing values in the reach column.

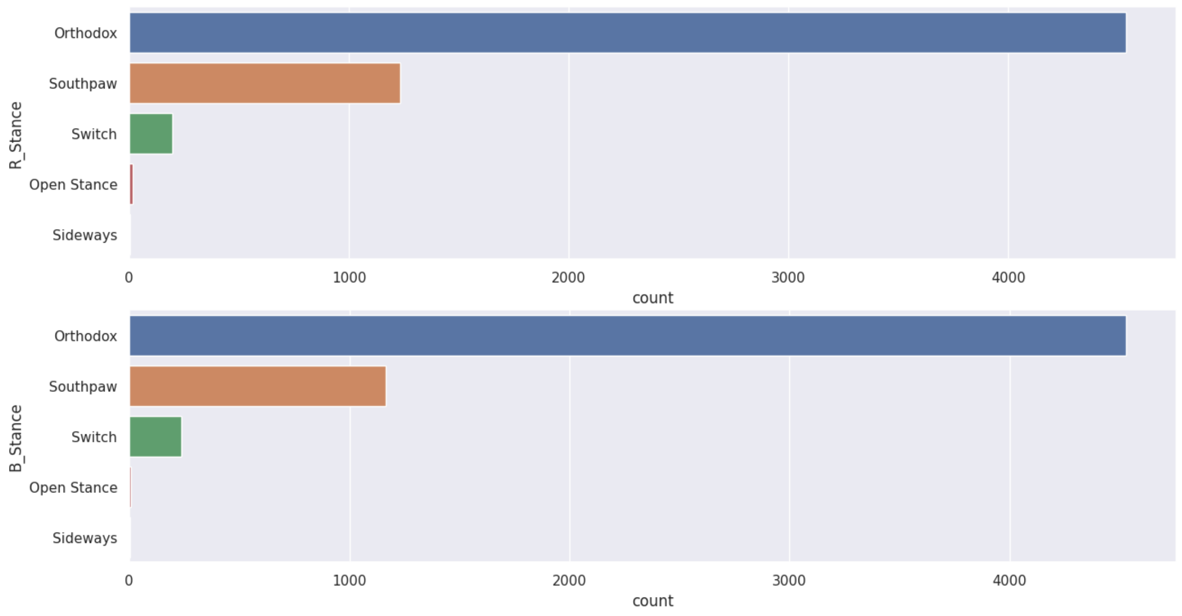
In [12]:

```
data['R_Reach_cms'].fillna(data['R_Height_cms'], inplace=True)
data['B_Reach_cms'].fillna(data['B_Height_cms'], inplace=True)
```

## Looking at the stance columns, any missing values will be replaced by simply using the most common stance (mode).

In [13]:

```
fig,ax = plt.subplots(nrows = 2, figsize=(15,8))
sns.countplot(y=data['R_Stance'],ax=ax[0])
sns.countplot(y=data['B_Stance'],ax=ax[1])

sns.set()
sns.set(style="white")
plt.show()
```



We are using the mode which will be Orthodox for both columns

In [14]:

```
data['R_Stance'].fillna(data['R_Stance'].mode()[0], inplace=True)
data['B_Stance'].fillna(data['B_Stance'].mode()[0], inplace=True)
```

In [15]:

```
data.dropna(inplace=True)
data.reset_index()
data.shape
```

Out[15]:

```
(4260, 139)
```

## (1.3) Performing One-Hot Encoding

Looking at the columns which contain features that will need encoding

In [16]:

```
print("Weight Class unique values: ",len(data["weight_class"].unique()))
print("Red Corner Stance unique values: ",len(data["R_Stance"].unique()))
print("Blue Corner Stance values: ",len(data["B_Stance"].unique()))
```

```
Weight Class unique values:  14
Red Corner Stance unique values:  4
Blue Corner Stance values:  4
```

We can see that the above columns contain features which we need to encode in order to utilise in any ML algorithm.
We cannot use label encoding since that assumes some order and will impact the performance of the ML algorithms.
Instead we are going to use one-hot encoding to encode these columns. This will however introduce more features, but we will narrow down the feature space later on.

In [17]:

```
enc = OneHotEncoder(categories='auto')
encoded_features = enc.fit_transform(data[["weight_class","R_Stance","B_Stance","Winner"]])
labels = enc.categories_
feature_labels = []
for label in labels:
    for l in label:
        feature_labels.append(l)

features = pd.DataFrame(encoded_features, columns=feature_labels)
features.head()
```

Out[17]:

| | Bantamweight | CatchWeight | Featherweight | Flyweight | Heavyweight | LightHeavyweight | Light |
|---|---|---|---|---|---|---|---|
| **0** | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **1** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **2** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **3** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **4** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |

5 rows × 25 columns

Now that we have these features encoded, we can add them to the original data and drop the columns from which these encoded features stem.

In [18]:

```
for i in range(len(feature_labels)):
    data.insert(i,feature_labels[i],encoded_features[:,i],allow_duplicates=True)

cols = ["weight_class","R_Stance","B_Stance"]
data = data.drop(columns=cols)

data.rename({'Blue':'Winner Blue', 'Red':'Winner Red'},axis=1,inplace=True)
data.head()
```

Out[18]:

| | Bantamweight | CatchWeight | Featherweight | Flyweight | Heavyweight | LightHeavyweight | Light |
|---|---|---|---|---|---|---|---|
| **0** | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **1** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **4** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **5** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| **6** | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |

5 rows × 161 columns

## (1.4) Performing Label Encoding

Now we can focus on the columns which require us to use label encoding

In [19]:

```
data['title_bout'] = data['title_bout'].apply(lambda x: 1 if x else 0)
```

## (1.5) Splitting the data into targets and features

In [20]:

```
targs = ["Winner Blue","Winner Red","Draw"]
targets = data[targs]
targetsForImportances = data["Winner"]
targs.extend(["Winner"])
features = data.drop(targs,axis=1)

print("The shape of the targets is: ",np.shape(targets))
print("The shape of the features is: ",np.shape(features))
```

```
The shape of the targets is:  (4260, 3)
The shape of the features is:  (4260, 157)
```

### Looking at the colinearity between variables and removing predictors that are highly correlated using the variance inflation factor (VIF)

- VIF is a number that determines whether a variable has multicollinearity or not. That number also represents how much a variable is inflated because of the linear dependence with other variables.

- The VIF value starts from 1, and it has no upper limit. If the number gets larger, it means the variable has huge multicollinearity on it.
- For calculating the VIF, we will do a linear regression process for each variable, where that variable will become a target variable. After we do that process, we calculate the R square from it. And finally, we calculate the VIF value with this formula: $VIF = \frac{1}{(1-R^2)}$ .
- We consider any feature with a VIF > 10 as highly correlated with other features and remove it.
- We remove multicollinearity since it will affect the statistical analysis of the importances of features which are conducted below.

In [21]:

```
vif_info = pd.DataFrame()
vif_info['VIF'] = [variance_inflation_factor(features.values, i) for i in range(features.sh
vif_info['Column'] = features.columns
vif_info.sort_values('VIF', ascending=False)
```

Out[21]:

| | VIF | Column |
|---|---|---|
| **0** | inf | Bantamweight |
| **49** | inf | B_avg_BODY_att |
| **56** | inf | B_avg_opp_LEG_landed |
| **55** | inf | B_avg_opp_LEG_att |
| **54** | inf | B_avg_LEG_landed |
| **...** | ... | ... |
| **30** | 1.299078 | B_avg_opp_SUB_ATT |
| **90** | 1.279007 | R_avg_opp_KD |
| **24** | 1.247823 | B_avg_opp_KD |
| **79** | NaN | B_draw |
| **145** | NaN | R_draw |

157 rows × 2 columns

In [22]:

```
cols = np.array(vif_info["Column"])
ind = np.where(vif_info["VIF"]>10)
ColumnsWithHighCollinearity = cols[ind]

features.drop(columns = ColumnsWithHighCollinearity,axis=1,inplace=True)
# data.drop(columns = ColumnsWithHighCollinearity,axis=1,inplace=True)
print(f"The total number of features after dropping highly correlated features is: {feature
```

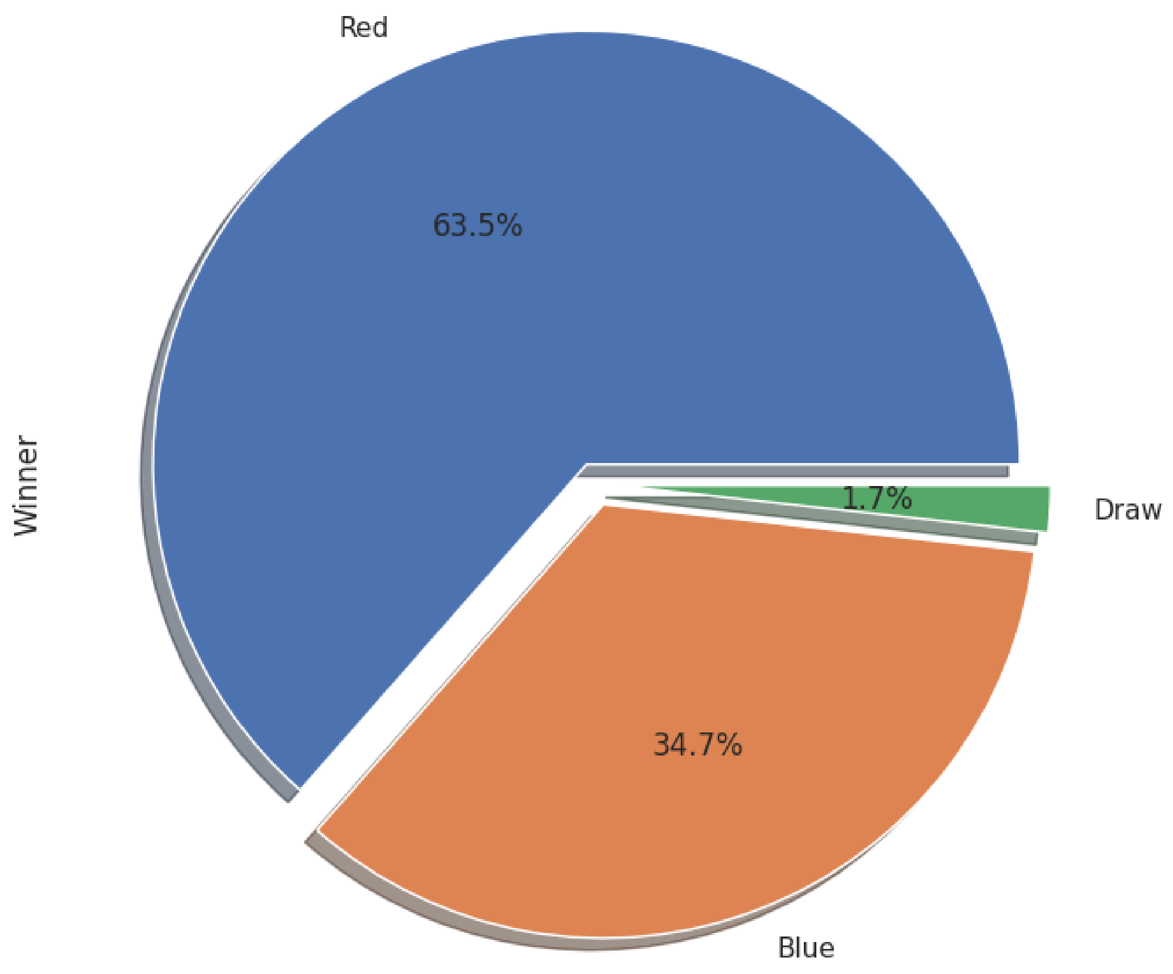
```
The total number of features after dropping highly correlated features is:
3
```

## (1.6) Data Visualizations

### Looking at the frequency each corner wins with

In [23]:

```
plt.figure(figsize=(15,8))
targetsForImportances.value_counts()[:10].plot.pie(explode=[0.05,0.05,0.05],autopct='%1.1f%
plt.show()
```

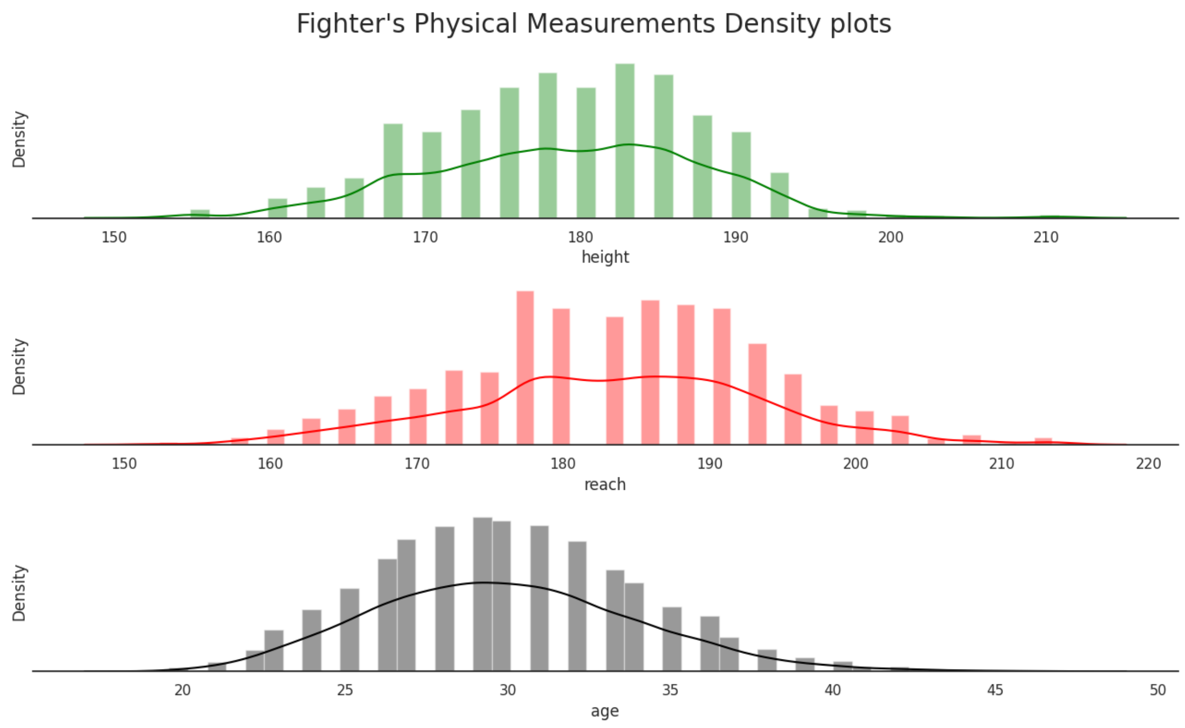


- We can see that majority of the wins occur in the red corner and so a model should be able to predict this class better than the others.
- Draws are very few and thus might be the results we predict incorrectly since there are too few data instances to learn from.

### Looking at the distribution of the fighters height, reach and age

In [24]:

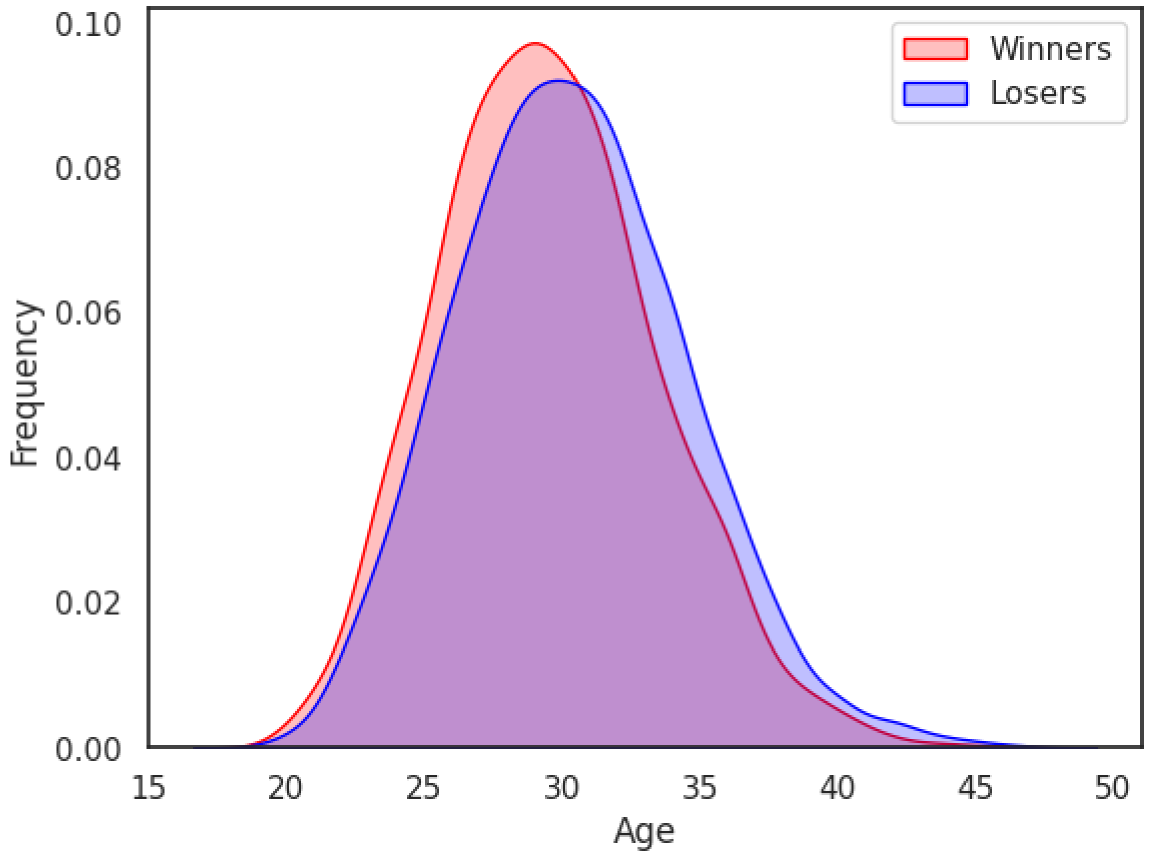
```
# combine measures of both blue and red fighter
height = pd.concat([data["B_Height_cms"],data["R_Height_cms"]],axis=0)
reach = pd.concat([data["B_Reach_cms"],data["R_Reach_cms"]],axis=0)
age = pd.concat([data["B_age"],data["R_age"]],axis=0)
fighter_measures = pd.DataFrame({"height":height, "reach":reach, "age":age}).reset_index(dr

# subplots:
fig4, axes = plt.subplots(3, 1, figsize=(13, 8), sharex=False) # Grid
fig4.suptitle("Fighter's Physical Measurements Density plots",fontsize=20) # Main title
sns.despine(left=True,right=True) # remove axis border

sns.distplot(fighter_measures["height"],color="green",
              ax=axes[0] # position in grid
            )
sns.distplot(fighter_measures["reach"],color="red",ax=axes[1])
sns.distplot(fighter_measures["age"],color="black",ax=axes[2])

plt.setp(axes, yticks=[]) # y-ticks range disabled
plt.tight_layout() #auto-adjusts subplots to the figure area
```



## Exploring age against winning and losing

In [25]:

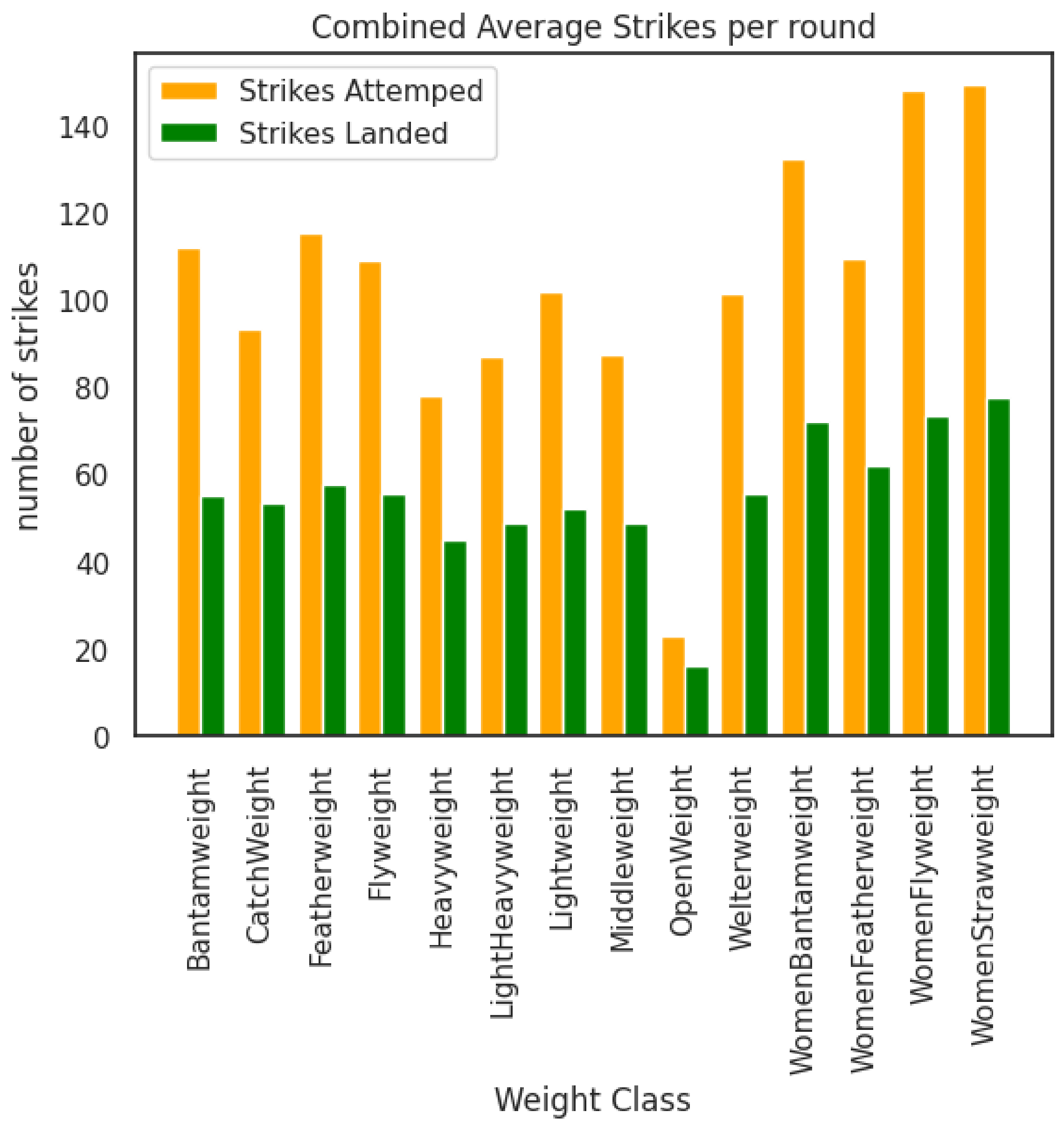
```
# Explore Age distibution
winners_age_combinded = pd.concat([data["B_age"][(data["Winner"] == "Blue") ], # age of blu
                                   data["R_age"][(data["Winner"] == "Red") ] # age of red w
                                  ]).reset_index(drop=True)

losers_age_combinded = pd.concat([data["B_age"][(data["Winner"] == "Red") ], # when winner
                                   data["R_age"][(data["Winner"] == "Blue") ] # same logic
                                  ]).reset_index(drop=True)
g = sns.kdeplot(winners_age_combinded, color="Red", shade = True)
g = sns.kdeplot(losers_age_combinded, color="Blue", shade = True, ax=g)


g.set_xlabel("Age")
g.set_ylabel("Frequency")
g = g.legend(["Winners","Losers"])
```



### Which weight-classes tend to utilize more striking?

In [26]:

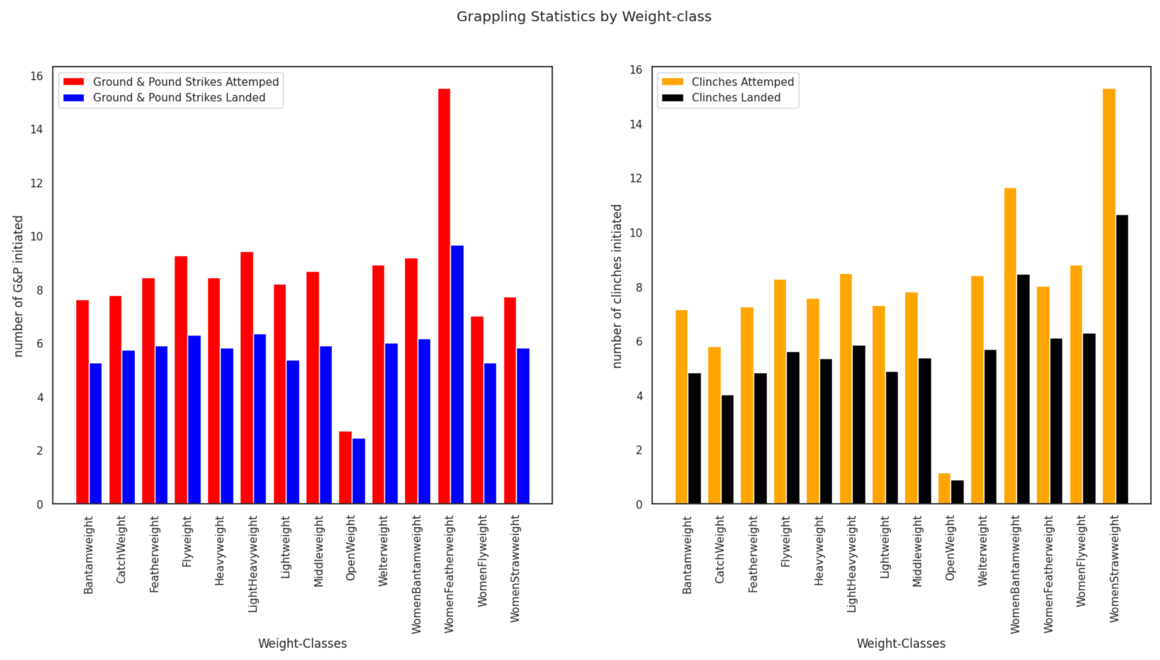
```
#strike_app & strike_landed

weight_classes = ['Bantamweight' ,'CatchWeight', 'Featherweight', 'Flyweight', 'Heavyweight
blue_strikes = ['B_avg_TOTAL_STR_att','B_avg_TOTAL_STR_landed'] #avg red corner strikes in
red_strikes = ['R_avg_TOTAL_STR_att','R_avg_TOTAL_STR_landed'] #avg blue corner strikes in
stats = []

for w_class in weight_classes:
    current_data = data.loc[data[w_class] == 1]
    current_data = current_data[[blue_strikes[0],blue_strikes[1],red_strikes[0],red_strikes
    Total_avg = current_data.mean(axis=0)
    stats.append({'avg_TOTAL_STR_att': (Total_avg[blue_strikes[0]]+Total_avg[red_strikes[0]


strikes_landed = []
strikes_attempted = []
for i in stats:
    strikes_landed.append(i['avg_TOTAL_STR_landed'])
    strikes_attempted.append(i['avg_TOTAL_STR_att'])
X_axis = np.arange(len(weight_classes))
plt.xticks(X_axis,weight_classes, rotation=90)
plt.xlabel("Weight Class")
plt.ylabel("number of strikes")

plt.bar(X_axis - 0.2, strikes_attempted, 0.4, label = 'Strikes Attemped',color='orange')
plt.bar(X_axis + 0.2, strikes_landed, 0.4, label = 'Strikes Landed',color='green')
plt.title("Combined Average Strikes per round")
plt.legend()
plt.show()
```



## Grappling statistics by weight-class

In [27]:

```
blue_clinch = ['B_avg_CLINCH_att', 'B_avg_CLINCH_landed']
red_clinch = ['R_avg_CLINCH_att', 'R_avg_CLINCH_landed']
blue_ground = ['B_avg_GROUND_att', 'B_avg_GROUND_landed']
red_ground = ['R_avg_GROUND_att', 'R_avg_GROUND_landed']


ground = []
clinch = []

for w_class in weight_classes:
    current_data = data.loc[data[w_class] == 1]
    current_data = current_data[[blue_clinch[0],blue_clinch[1],red_clinch[0],red_clinch[1],
    Total_avg = current_data.mean(axis=0)
    ground.append({'avg_TOTAL_GRD_att': (Total_avg[blue_ground[0]]+Total_avg[red_ground[0]]
    clinch.append({'avg_TOTAL_CLN_att': (Total_avg[blue_clinch[0]]+Total_avg[red_clinch[0]]


ground_landed = []
ground_attempted = []
clinch_landed = []
clinch_attempted = []

for i in range(len(ground)):
    ground_landed.append(ground[i]['avg_TOTAL_GRD_landed'])
    ground_attempted.append(ground[i]['avg_TOTAL_GRD_att'])
    clinch_landed.append(clinch[i]['avg_TOTAL_CLN_landed'])
    clinch_attempted.append(clinch[i]['avg_TOTAL_CLN_att'])

fig, ax = plt.subplots(1, 2, figsize=(20,8))
ax[0].set_xticks(X_axis,weight_classes, rotation=90)
ax[1].set_xticks(X_axis,weight_classes, rotation=90)

fig.suptitle('Grappling Statistics by Weight-class')
ax[0].set_ylabel("number of G&P initiated")
ax[0].set_xlabel("Weight-Classes")

ax[0].bar(X_axis - 0.2, ground_attempted, 0.4, label = 'Ground & Pound Strikes Attemped',co
ax[0].bar(X_axis + 0.2, ground_landed, 0.4, label = 'Ground & Pound Strikes Landed',color='

ax[1].set_ylabel("number of clinches initiated")
ax[1].set_xlabel("Weight-Classes")

ax[1].bar(X_axis - 0.2, clinch_attempted, 0.4, label = 'Clinches Attemped',color='orange')
ax[1].bar(X_axis + 0.2, clinch_landed, 0.4, label = 'Clinches Landed',color='black')

ax[0].legend()
ax[1].legend()
plt.show()
```



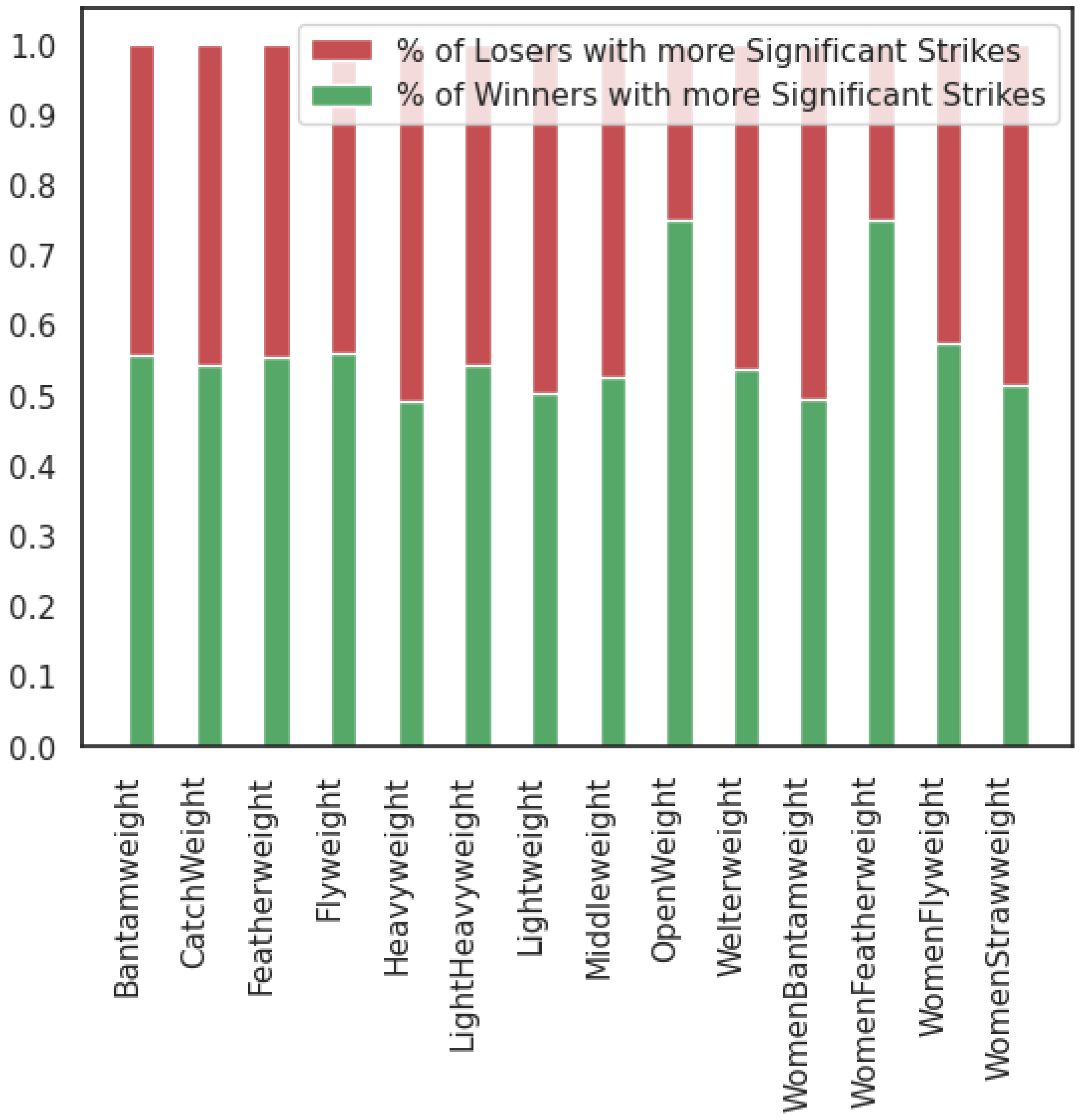
Looking at the previous data - some trends become obvious.

- The Openweight Category can be treated as an outlier; looking at the history UFC - this was a short-lived category that was featured in the first 6 UFC tournaments and allowed fighters of any weights to go against each other. These fights were done for the sole purpose of curiosity and entertainment.
- Striking: We can see a general trend with more average strikes occuring in the lighter weight classes and lessening going up. This could be due to the fact that generally lighter people have a lot energy and fatigue less easily than heavier people. The lighter one is, the weaker their punches tend to be as well and so can unload a lot more punches in a fight. With a stronger punch, comes a lot more knockout-power and increase the chance of cutting a fight short by knocking-out an opponent, so will have less average punches overrall.
- Grappling: In the grappling statistics we see a similar trend amongst all the weight-classes with similar attempts and succesful actions. We would assume this would be due to the fact that initiating grappling is quite different to striking. Often times, in a fight in a specific round a fighter will initiate a grappling technique, land it succesfully, and then control them in the position till the end of the round. There are fighters out there who do not grapple at all (such as current Middleweight champion Israel Adesanya), so the data associated with grappling will have a lot more 0 values.
- Women's statistics: The women's division was introduced into the UFC in 2013 and still to this day does not have as large a roster or frequent fights as the Mens division, so the data is skewed. However despite being skewed, we still see the same trend amongst the women's divisions in terms of lighter divisions have more average strikes thrown per fight.

## Do winners generally have more significant strikes ?

In [28]:

```
win = []
for w_class in weight_classes:
    current = data.loc[data[w_class] == 1]
    winners = current.loc[((current['Winner Blue'] == 1) & (current['B_avg_SIG_STR_landed']
    percentage = winners.shape[0]/current.shape[0]
    win.append(percentage)

loss = [1 for i in win]

plt.yticks(np.arange(0, 1.0001, step=0.1))
plt.xticks(X_axis,weight_classes, rotation=90)
plt.bar(weight_classes, loss, align='edge', width=0.4, color='r', label='% of Losers with m
plt.bar(weight_classes, win, align='edge', width=0.4, color='g',label='% of Winners with mo
plt.legend()
plt.show()
```



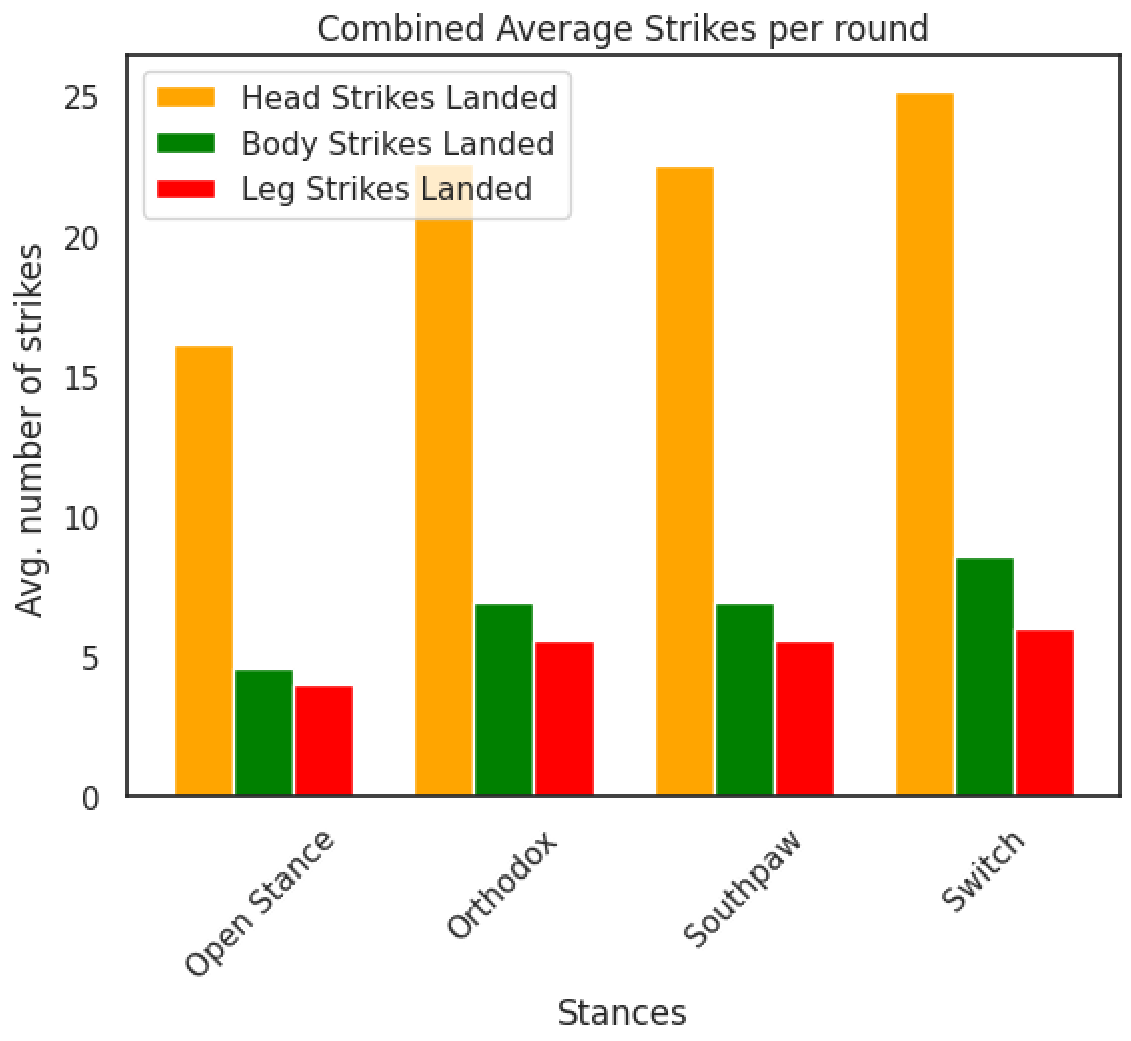
## Each stances preferred strike

In [29]:

```
def df_column_uniquify(df):
    df_columns = df.columns
    new_columns = []
    for item in df_columns:
        counter = 0
        newitem = item
        while newitem in new_columns:
            counter += 1
            newitem = "{}_{}".format(item, counter)
        new_columns.append(newitem)
    df.columns = new_columns
    return df
```

In [30]:

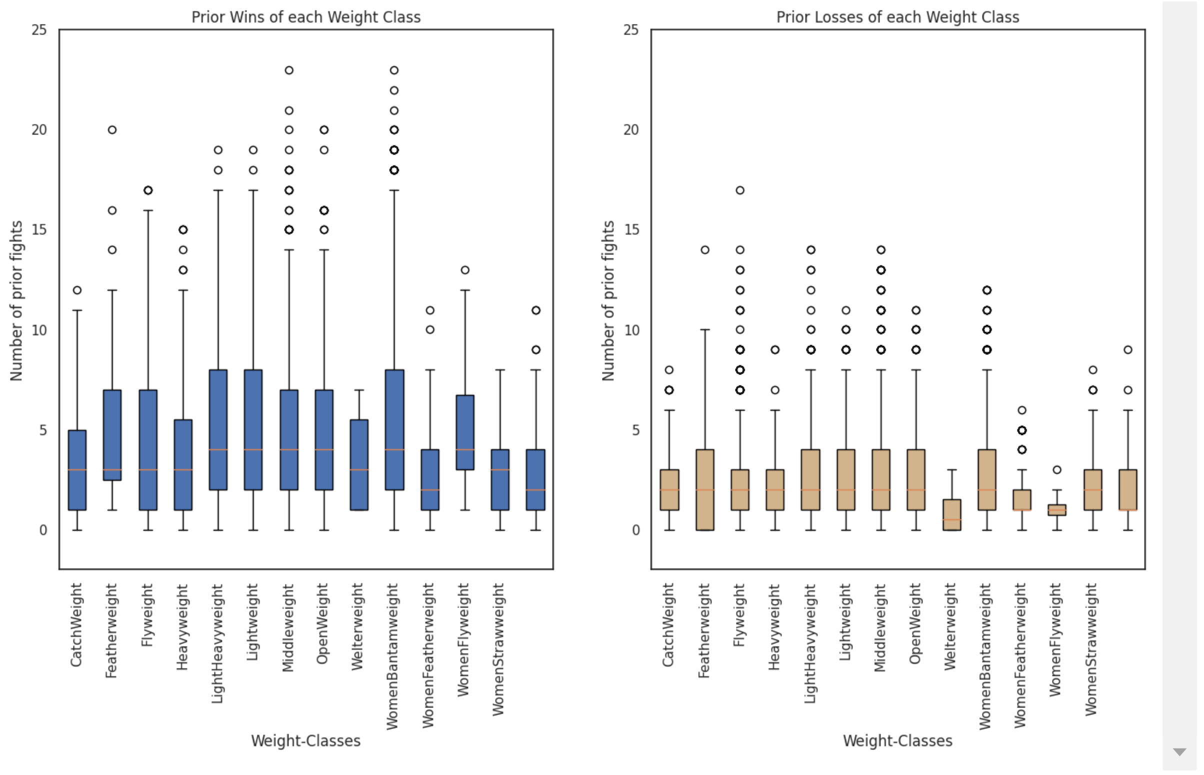
```
stances = [('Open Stance','Open Stance_1'),('Orthodox','Orthodox_1'),('Southpaw','Southpaw_



strikes_b = ['B_avg_HEAD_landed','B_avg_BODY_landed', 'B_avg_LEG_landed']
strikes_r = ['R_avg_HEAD_landed','R_avg_BODY_landed', 'R_avg_LEG_landed']

temp_data = data #so to not mess up anything
temp_data = df_column_uniquify(temp_data)
only_stance = [stance[0] for stance in stances]

striking = {}
for stance in stances:
    #_1 corresponds to red, non to blue
    current_b = temp_data.loc[(temp_data[stance[0]] == 1)]
    current_b = current_b[strikes_b].mean()

    current_r = temp_data.loc[(temp_data[stance[1]] == 1)]
    current_r = current_r[strikes_r].mean()

    Total = []
    for i in range(len(current_b)):
        Total.append((current_b[i] + current_r[i])/2)

    striking[stance[0]] = Total

head_strikes = [strike[0] for strike in striking.values()]
body_strikes = [strike[1] for strike in striking.values()]
leg_strikes = [strike[2] for strike in striking.values()]


N = 4
ind = np.arange(N)
width = 0.25

plt.xticks(ind+width, list(striking.keys()), rotation=45)
plt.xlabel("Stances")
plt.ylabel("Avg. number of strikes")

plt.bar(ind, head_strikes, width, label = 'Head Strikes Landed',color='orange')
plt.bar(ind+width, body_strikes, width, label = 'Body Strikes Landed',color='green')
plt.bar(ind+width*2, leg_strikes, width, label = 'Leg Strikes Landed',color='red')

plt.title("Combined Average Strikes per round")
plt.legend()
plt.show()
```



### Wins and Losses Per Weight class

In [31]:

```python
red_record = ['R_wins', 'R_losses']
blue_record = ['B_wins', 'B_losses']
red_record.extend(blue_record)

wins_losses = []

for weight in weight_classes:
    current = data.loc[data[weight] == 1]
    current = current[red_record]
    wins = current['R_wins'].to_numpy()
    np.concatenate((wins, current['B_wins'].to_numpy()),axis=None)

    losses = current['R_losses'].to_numpy()
    np.concatenate((losses, current['B_losses'].to_numpy()),axis=None)

    wins_losses.append((wins,losses))

wins = [data[0] for data in wins_losses]
loss = [data[1] for data in wins_losses]


fig, ax = plt.subplots(1, 2, figsize=(16,8))
ax[0].set_xticks(X_axis,weight_classes, rotation=90)

# Creating axes instance
# Creating plot
ax[0].boxplot(wins, labels=weight_classes, patch_artist=True)
ax[0].set_ylim(-2,25)
ax[0].set_ylabel("Number of prior fights")
ax[0].set_xlabel("Weight-Classes")
ax[0].set_title("Prior Wins of each Weight Class")
# show plot

ax[1].set_xticks(X_axis,weight_classes, rotation=90)

# Creating axes instance
# Creating plot
box_loss = ax[1].boxplot(loss, labels=weight_classes, patch_artist=True)

ax[1].set_xlabel("Weight-Classes")
ax[1].set_ylabel("Number of prior fights")
ax[1].set_title("Prior Losses of each Weight Class")
ax[1].set_ylim(-2,25)


colors = []
for i in range(len(weight_classes)):
    colors.append('tan')
for patch, color in zip(box_loss['boxes'], colors):
    patch.set_facecolor(color)
# show plot
plt.show()
```



## (1.7) Scaling the data

In [32]:

```
sc = StandardScaler()
cols = features.columns
features = pd.DataFrame(sc.fit_transform(features),columns=cols)
```

# (2) Finding the importances of the features so we can only choose the most important ones for our model

## (2.1) Using embedded methods to find optimal features

- Embedded methods are iterative in a sense that takes care of each iteration of the model training process and carefully extract those features which contribute the most to the training for a particular iteration. Regularization methods are the most commonly used embedded methods which penalize a feature given a coefficient threshold.
- Embedded methods are iterative in a sense that takes care of each iteration of the model training process and carefully extract those features which contribute the most to the training for a particular iteration. Regularization methods are the most commonly used embedded methods which penalize a feature given a coefficient threshold.

### (2.1.1) Using Logistic Regression with L1 regularisation

- Regularisation consists in adding a penalty to the different parameters of the machine learning model to reduce the freedom of the model and in other words to avoid overfitting. In linear model regularisation, the penalty is applied over the coefficients that multiply each of the predictors. From the different types of regularisation, Lasso or l1 has the property that is able to shrink some of the coefficients to zero. Therefore, that feature can be removed from the model.

In [ ]:

```
logistic = LogisticRegression(C=0.03, penalty="l1", solver='liblinear', random_state=7)
model_Logistic = SelectFromModel(logistic)  # this chooses the best features for us!
model_Logistic.fit(features, targetsForImportances)

selected_feat_lr = features.columns[(model_Logistic.get_support())]
print('total features: {}'.format((features.shape[1])))
print('selected features: {}'.format(len(selected_feat_lr)))
```

```
total features: 53
selected features: 35
```

## (2.1.2) Using a Linear SVC with L1 regularization

In [ ]:

```
lsvc = LinearSVC(C=0.01, penalty="l1", dual=False).fit(features, targetsForImportances)
model_svc = SelectFromModel(lsvc)
model_svc.fit(features,targetsForImportances)
selected_feat_svc = features.columns[(model_svc.get_support())]
print('total features: {}'.format((features.shape[1])))
print('selected features: {}'.format(len(selected_feat_svc)))
```

```
total features: 53
selected features: 38
```

## (2.1.3) Using a Random Forest

- Random forests are one the most popular machine learning algorithms. They are so successful because they provide in general a good predictive performance, low overfitting and easy interpretability. This interpretability is given by the fact that it is straightforward to derive the importance of each variable on the tree decision. In other words, it is easy to compute how much each variable is contributing to the decision.
- Random forests consist of 4-12 hundred decision trees, each of them built over a random extraction of the observations from the dataset and a random extraction of the features. Not every tree sees all the features or all the observations, and this guarantees that the trees are de-correlated and therefore less prone to over-fitting. Each tree is also a sequence of yes-no questions based on a single or combination of features. At each node (this is at each question), the three divides the dataset into 2 buckets, each of them hosting observations that are more similar among themselves and different from the ones in the other bucket. Therefore, the importance of each feature is derived by how "pure" each of the buckets is.
- For classification, the measure of impurity is either the Gini impurity or the information gain/entropy. For regression the measure of impurity is variance. Therefore, when training a tree, it is possible to compute how much each feature decreases the impurity. The more a feature decreases the impurity, the more important the feature is. In random forests, the impurity decrease from each feature can be averaged across trees to determine the final importance of the variable.
- To give a better intuition, features that are selected at the top of the trees are in general more important than features that are selected at the end nodes of the trees, as generally the top splits lead to bigger information gains.

In [ ]:

```
model = RandomForestClassifier(n_estimators=250, random_state=42, max_depth=10)
# fit the model to start training.
model.fit(features,targetsForImportances)
# get the importance of the resulting features.
importances = model.feature_importances_
# create a data frame for visualization.
final_df = pd.DataFrame({ "Features": pd.DataFrame(features).columns, "Importances": import
final_df.set_index('Importances')
# sort in ascending order to better visualization.
final_df = final_df.sort_values('Importances',ascending= False)
random_forest = final_df[:53].to_numpy()
random_forest = list(random_forest[:,0])
```

## (2.2) Using filter methods to find optimal features

- Filter methods are generally used as a preprocessing step. The selection of features is independent of any machine learning algorithms. Instead, features are selected on the basis of their scores in various statistical tests for their correlation with the outcome variable. The characteristics of these methods are as follows:

  1. These methods rely on the characteristics of the data (feature characteristics)
  2. They do not use machine learning algorithms.
  3. These are model agnostic.
  4. They tend to be less computationally expensive.
  5. They usually give lower prediction performance than wrapper methods.
  6. They are very well suited for a quick screen and removal of irrelevant features.

### (2.2.1) Using Information Gain

- Information gain or mutual information measures how much information the presence/absence of a feature contributes to making the correct prediction on the target.
- It estimates mutual information for a discrete target variable.
- Mutual information (MI) between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

In [ ]:

```
information_gain = mutual_info_classif(features,targetsForImportances,random_state=7)

info_gain_importances  = pd.DataFrame(information_gain,features.columns,columns =["Importan
info_gain_importances = info_gain_importances.sort_values("Importances",ascending=False)

important_feats_indices = np.where(info_gain_importances.to_numpy() != 0)
labels = np.array(features.columns)
ig_feats = []
for i in important_feats_indices[0]:
    ig_feats.append(labels[i])

print('total features: {}'.format((features.shape[1])))
print('selected features: {}'.format(len(ig_feats)))
```

```
total features: 53
selected features: 29
```

## (2.3) Wrapper Methods

- In wrapper methods, we try to use a subset of features and train a model using them. Based on the inferences that we draw from the previous model, we decide to add or remove features from the subset. The problem is essentially reduced to a search problem. These methods are usually computationally very expensive.
- The algorithm starts from all features and greedily eliminates the least important feature. Once all features are removed, the algorithm returns the subset which gives the best performance.

### (2.3.1) Using Recursive feature elimination with cross-validation

- Recursive Feature Elimination with Cross-Validated (RFECV) feature selection technique selects the best subset of features for the estimator by removing 0 to N features iteratively using recursive feature elimination.
- Then it selects the best subset based on the accuracy or cross-validation score or roc-auc of the model. Recursive feature elimination technique eliminates n features from a model by fitting the model multiple times and at each step, removing the weakest features.

In [ ]:

```
model_for_RFEWCV = RandomForestClassifier(n_estimators=250, random_state=42, max_depth=10)
rfe = RFECV(model_for_RFEWCV,
            step=1,
            cv=2,
            scoring= 'accuracy',
           verbose= 1)
rfe = rfe.fit(features, targetsForImportances)
selected_feat_rfe = features.columns[(rfe.get_support())]
print('total features: {}'.format((features.shape[1])))
print('selected features: {}'.format(len(selected_feat_rfe)))
```

```
Fitting estimator with 53 features.
Fitting estimator with 52 features.
Fitting estimator with 51 features.
Fitting estimator with 50 features.
Fitting estimator with 49 features.
Fitting estimator with 48 features.
Fitting estimator with 47 features.
Fitting estimator with 46 features.
Fitting estimator with 45 features.
Fitting estimator with 44 features.
Fitting estimator with 43 features.
Fitting estimator with 42 features.
Fitting estimator with 41 features.
Fitting estimator with 40 features.
Fitting estimator with 39 features.
Fitting estimator with 38 features.
Fitting estimator with 37 features.
Fitting estimator with 36 features.
Fitting estimator with 35 features.
```

## (2.4) Selecting the best features based on the common features returned by all the above processes

In [ ]:

```
Features = list(set(random_forest).intersection(selected_feat_svc)
                .intersection(selected_feat_lr)
                .intersection(ig_feats)
                .intersection(selected_feat_rfe))
print(f"The initial number of features is: {features.shape[1]}")
print(f"The number of features selected is: {len(Features)}")
print(f"The number of features removed is: {features.shape[1] - len(Features)}")
```

```
The initial number of features is: 53
The number of features selected is: 16
The number of features removed is: 37
```

In [ ]:

```
SelectedFeatures = features[Features]
```

## (2.5) Splitting into training,validation and testing sets

In [ ]:

```
Train_Features, Test_Features, Train_Targets, Test_Targets = train_test_split(SelectedFeatu
```

## Helper Functions

A function that will take in one-hot encoded targets or the predictions of a network and convert it into a list of single predictions. These predictions either being Red or Blue as winner or a draw.

In [ ]:

```
def Convert_Targets(Targets):
    test = list()
    for i in range(len(np.asarray(Targets))):
            test.append(np.argmax(np.asarray(Targets)[i]))

    return test
```

A function that will plot the loss during training as well as validation if we implement early stopping

In [ ]:

```
def plot_metric(history, metric,earlyStopping):
    if(earlyStopping):
        train_metrics = history.history[metric]
        val_metrics = history.history['val_'+metric]
        epochs = range(1, len(train_metrics) + 1)
        plt.plot(epochs, train_metrics)
        plt.plot(epochs, val_metrics)
        plt.title('Training and validation '+ metric)
        plt.xlabel("Epochs")
        plt.ylabel(metric)
        plt.legend(["train_"+metric, 'val_'+metric])
        plt.show()
    else:
        train_metrics = history.history[metric]
        epochs = range(1, len(train_metrics) + 1)
        plt.plot(epochs, train_metrics)
        plt.title('Training '+ metric)
        plt.xlabel("Epochs")
        plt.ylabel(metric)
        plt.show()
```

A method that will take in the one-hot encoded actual targets and the probabilities predicted by the Neural Network, convert them into single targets and then calculate the accuracy score and provide a confusion matrix.

In [ ]:

```
def getAccuracy(predicted,actual,alreadyFiltered):
    if(not alreadyFiltered):
        true_targets = Convert_Targets(actual)
        model_predictions = Convert_Targets(predicted)
        acc = accuracy_score(true_targets,model_predictions)
        print(f"The accuracy is: {np.round(acc*100,4)}")
        mat = confusion_matrix(true_targets,model_predictions)
        sns.heatmap(mat,fmt='g',annot=True)
        plt.xlabel("Predicted ")
        plt.ylabel("True Values")
        plt.show()
        return acc
    else:
        acc = accuracy_score(actual,predicted)
        print(f"The accuracy is: {np.round(acc*100,4)}")
        mat = confusion_matrix(actual,predicted)
        sns.heatmap(mat,fmt='g',annot=True)
        plt.xlabel("Predicted ")
        plt.ylabel("True Values")
        plt.show()
        return acc
```

# (3) Machine Learning Models

## (3.1) Extreme Gradient Boosting (XGBoost)

### Explaining XGBoost

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework.
When it comes to small-to-medium structured/tabular data, decision tree based algorithms are considered best-in-class right now.

### 1. Decision Trees:

- Decision trees are supervised machine learning algorithm that is used for both classification and regression tasks.
- Decision Trees are a tree-like model that can be used to predict the class/value of a target variable and Decision trees handle non-linear data effectively.
- In the decision trees, at each node, the split is done based on the feature. If the feature is a continuous variable, the splitting condition is made on a certain threshold. If the feature is a categorical variable, the splitting condition is made on all the values. Since we are doing classification, we will focus on that.
- The criteria used to calculate the best split are:
    1. Classification Error: Split is done based on each feature. For each feature, the change in classification error is calculated and the feature which gives the maximum classification error change is chosen for the split. However since it is possible to end up with a 0 change in error, the method is not preferred.
    2. Entropy: Entropy is used to measure the homogeneity of a node in the decision tree. If the node is homogenous, entropy is 0. Entropy refers to impurity. Reduction in entropy (impurity) is known as Information Gain. If the information gain is more for a particular split, that split condition is executed

first.Reduction in entropy (impurity) is known as Information Gain. If the information gain is more for a particular split, that split condition is executed first.
3. Gini-index: Gini-index is another measure of impurity.

- Decision trees tend to overfit and are a high variance model. Thus it is incapable of generalizing well.
- Besides utilising pruning, bagging is used to reduce the variance of the model.

## Ensemble Modelling

- Ensemble learning offers a systematic solution to combine the predictive power of multiple learners. The resultant is a single model that gives the aggregated output from several models.
- The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms.
- bagging and boosting are two highly praised ensemble learners

### 2. Bagging :

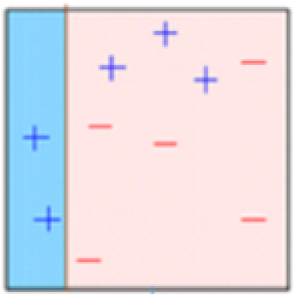
- Bagging means building different models using sample subset and then aggregating the predictions of the different models to reduce variance.
- Bagging means Bootstrap Aggregation.
- Bootstrapping helps to create multiple subsets from the training data. Then we can build multiple trees on the bootstrapped samples. Bootstrapping will randomly select repeated data points from the training set and create multiple subsets — sampling with replacement.
- Now we have different bootstrap samples and have built 'B' decision trees for each 'B' bootstrap sample. The next step is aggregating the output.
- Now we have different bootstrap samples and have built 'B' decision trees for each 'B' bootstrap sample. The next step is aggregating the output.
- Another approach is to get the probability of the class from the 'B' trees, and the final class is classified from the average of the probability.

### 3. Boosting :

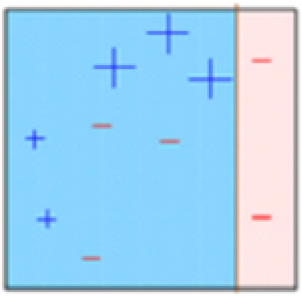
- Boosting algorithms seek to improve the prediction power by training a sequence of weak models, each compensating the weaknesses of its predecessors.
- As a reminder, a weak learner classifies data slightly better than random guessing.
- Different boosting algorithms:
    1. Adaptive boosting: This method operates iteratively, identifying misclassified data points and adjusting their weights to minimize the training error. The model continues optimize in a sequential fashion until it yields the strongest predictor.
    2. Gradient boosting: works by sequentially adding predictors to an ensemble with each one correcting for the errors of its predecessor. However, instead of changing weights of data points like AdaBoost, the gradient boosting trains on the residual errors of the previous predictor.

**The general boosting procdure is as follows:**

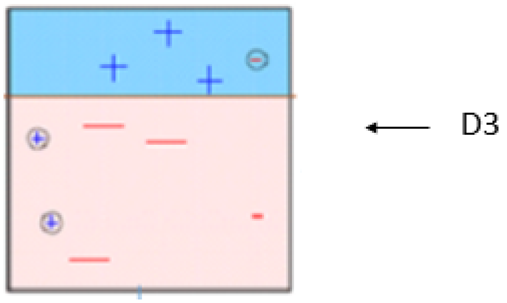
- The first weak learner is created and used to classify the data. The results are shown in the image below.



We can clearly see that the boundary is incorrect. There are 3 positives that are incorrectly classified. The residual of this model is calculated, and the next predictor will focus on minimising this residual by learning to classify these incorrectly classified points correctly.

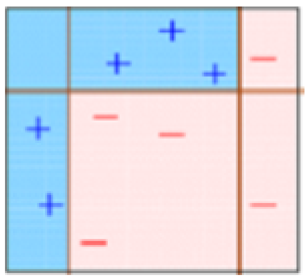


- The next weak learner is created and used to classify the data, but the previously misclassified points are focussed on. We can now see that those previously misclassified points are classified correctly, but there are 3 negative values that are classified incorrectly. Consequently, the next predictor will focus on minimising this predictors residual by learning to classify these incorrectly classified points correctly.



- The next weak learner is created and used to classify the data, but the previously misclassified points are focussed on. We can now see that those previously misclassified negative points are classified correctly, but there are 2 negative values that are classified incorrecly. However, those points are classified correctly by the first learner.

If we combine the above weak learners, we end up with a strong learner denoted below:

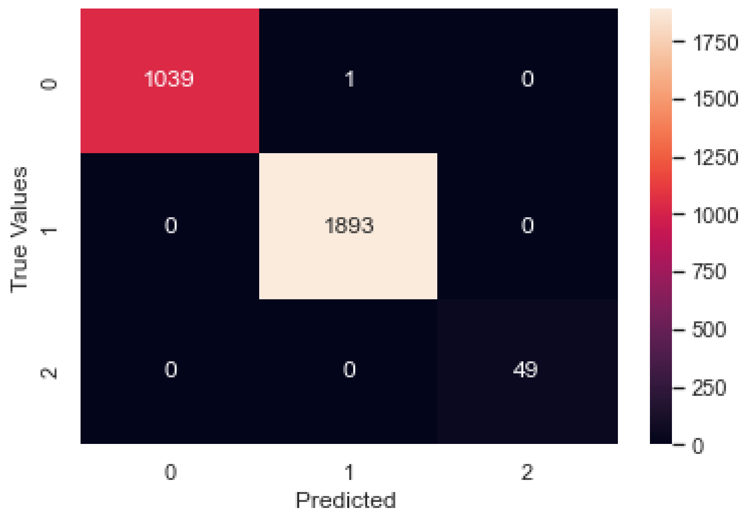


## (3.1.1) Basic XGBoost Model

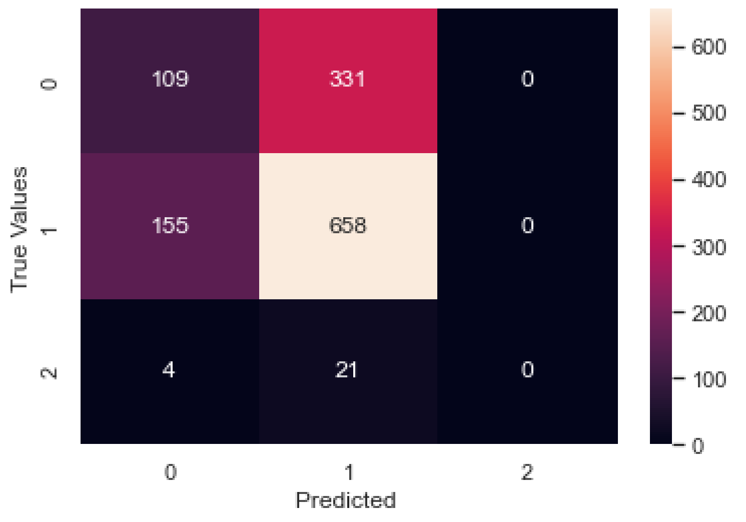
In [ ]:

```
xgb_basic = xgboost.XGBClassifier(n_estimators=100,
                                  eval_metric = "mlogloss")

xgb_basic.fit(Train_Features, Convert_Targets(Train_Targets))

prediction_basic = xgb_basic.predict(Test_Features)

simple_xg_table = PrettyTable(["Training Accuracy","Testing Accuracy"])

print("The training results: ")
train_acc = getAccuracy(xgb_basic.predict(Train_Features),Convert_Targets(Train_Targets),Tr

print("The testing results: ")
test_acc = getAccuracy(prediction_basic,Convert_Targets(Test_Targets),True)
simple_xg_table.add_row([train_acc*100,test_acc*100])
```

```
The training results:
The accuracy is: 99.9665
```



```
The testing results:
The accuracy is: 60.0156
```

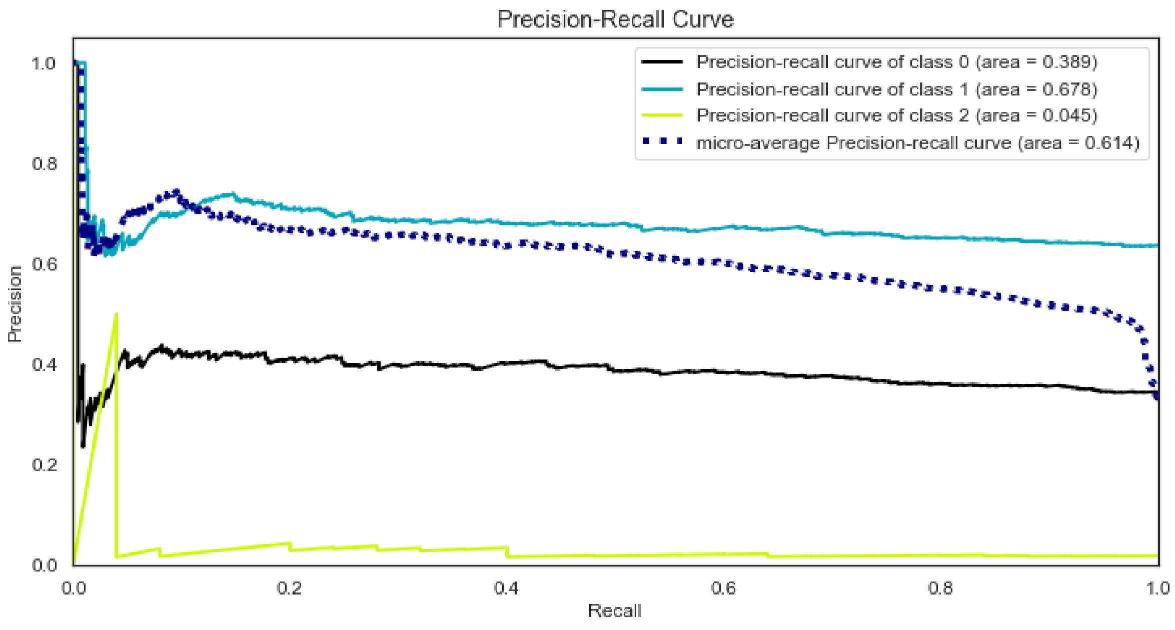


## (3.1.2) Analysing the simple models results

### Precision-Recall curve for the testing data

In [ ]:

```
skplt.metrics.plot_precision_recall_curve(Convert_Targets(Test_Targets), xgb_basic.predict_
                        title="Precision-Recall Curve", figsize=(12,6));
```

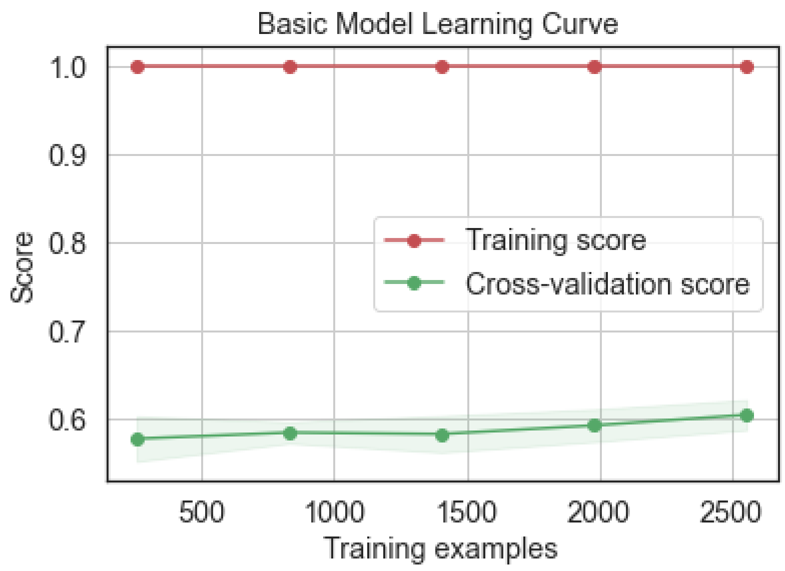


- The area under the each line of the precision-recall curve to be high since this will indicate the model is capable of predicting the correct classes.
- However, we can clearly see that the class 1 achieves the best area under the curve, which indicates the model is good at predicting this class and predicts the other classes poorly.
- This is due to the fact that our targets are imbalanced, and 1 which represents the Red corner, which won majority of the fights.
- The lowest area is represented by class 2 which represents a draw and was the smallest represented outcome.

### Basic Model Cross Validation Performance Plot

In [ ]:

```
skplt.estimators.plot_learning_curve(xgb_basic, Train_Features, Convert_Targets(Train_Targe
                                     cv=7, shuffle=True, scoring="accuracy", n_jobs=-1,
                                     figsize=(6,4), title_fontsize="large", text_fontsize="
                                     title="Basic Model Learning Curve ")
plt.show()
```



In [ ]:

```
print(simple_xg_table)
```

```
+-------------------+-------------------+
| Training Accuracy |  Testing Accuracy |
+-------------------+-------------------+
|  99.9664654594232 | 60.01564945226917 |
+-------------------+-------------------+
```

- We can see that our testing accuracy is significantly lower than our training accuracy.
- This clearly implies that we have overfit.
- Consequently we have a high variance model.

## (3.2.1) Optimizing the XGBoost parameters

## Parameters:

**1. eta/learning rate:**

- Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.

**2. max_depth:**

- Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree.

**3. min_child_weight:**

- It defines the minimum sum of weights of all observations required in a child.
- It is used to control over-fitting.
- Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
- Too high values can lead to under-fitting.

**4. subsample:**

- It denotes the fraction of observations to be randomly samples for each tree.
- Subsample ratio of the training instances.
- Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. - This will prevent overfitting.
- Subsampling will occur once in every boosting iteration.
- Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.

**5. lambda:**

- L2 regularization term on weights (analogous to Ridge regression).
- This is used to handle the regularization part of XGBoost.
- Increasing this value will make model more conservative.

**6. alpha:**

- L1 regularization term on weights (analogous to Lasso regression).
- It can be used in case of very high dimensionality so that the algorithm runs faster when implemented.
- Increasing this value will make model more conservative.

**7. scale_pos_weight :**

- It controls the balance of positive and negative weights,
- It is useful for imbalanced classes.
- A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.
- A typical value to consider: sum(negative instances) / sum(positive instances).

**8. N_estimators:**

- It determines the number of trees which we use in the ensemble.

In [ ]:

```
params = { 'max_depth': [3, 5, 6, 10, 15, 20],
           'min_child_weight': [1,5,10,20],
           'learning_rate': [0.01, 0.1, 0.2, 0.3],
           'subsample': np.arange(0.5, 1.0, 0.1),
           'n_estimators': [100, 500, 1000],
           'lambda': [1,3,7,15],
           'alpha': [1,3,7,15],
          'scale_pos_weight': [1,3,7,10]
         }
xgbr = xgboost.XGBClassifier(seed = 20,
                             eval_metric = "mlogloss" )

clf = RandomizedSearchCV(estimator=xgbr,
                         param_distributions=params,
                         scoring='accuracy',
                         n_iter=50,
                         cv = 2,
                         verbose=0,
                        random_state=42)

clf.fit(Train_Features, Convert_Targets(Train_Targets))
```

```
[15:44:19] WARNING: ..\src\learner.cc:576:
Parameters: { "scale_pos_weight" } might not be used.

  This could be a false alarm, with some parameters getting used by langua
ge bindings but
  then being mistakenly passed down to XGBoost core, or some parameter act
ually being used
  but getting flagged wrongly here. Please open an issue if you find any s
uch cases.


[15:44:19] WARNING: ..\src\learner.cc:576:
Parameters: { "scale_pos_weight" } might not be used.

  This could be a false alarm, with some parameters getting used by langua
ge bindings but
  then being mistakenly passed down to XGBoost core, or some parameter act
ually being used
  but getting flagged wrongly here. Please open an issue if you find any s
```

In [ ]:

```
xgb_df = pd.DataFrame(clf.cv_results_)
print("Best parameters:", clf.best_params_)
print("Best accuracy: ", clf.best_score_*100)
```

```
Best parameters: {'subsample': 0.7999999999999999, 'scale_pos_weight': 7, '
_estimators': 100, 'min_child_weight': 1, 'max_depth': 20, 'learning_rate':
0.01, 'lambda': 7, 'alpha': 3}
Best accuracy:  63.816230717639165
```

## Optimized XGBoost Hyperparameters:

#### 1. eta/learning rate: 0.01

**2. max_depth: 20**

**3. min_child_weight: 1**

**4. subsample: 0.79**

**5. lambda: 7**

**6. alpha: 3**

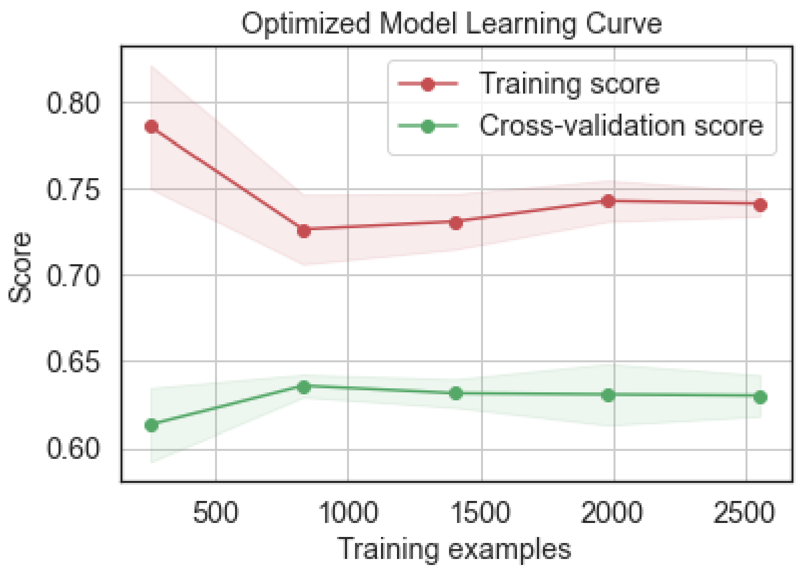
**7. scale_pos_weight : 7**

**8. N_estimators: 100**

### Optimized XGBoost Model Cross Validation Performance Plot

In [ ]:

```
skplt.estimators.plot_learning_curve(clf.best_estimator_, Train_Features, Convert_Targets(T
                                     cv=7, shuffle=True, scoring="accuracy", n_jobs=-1,
                                     figsize=(6,4), title_fontsize="large", text_fontsize="
                                     title="Optimized Model Learning Curve ")
plt.show()
```



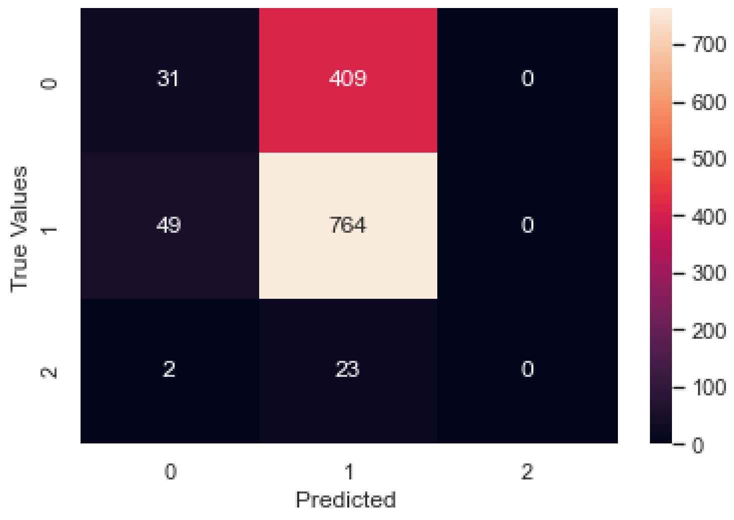
### (3.2.2) Analysing Optimized XGBoost Model Results

### Optimized XGBoost Test Accuracy

In [ ]:

```
test_acc = getAccuracy(clf.best_estimator_.predict(Test_Features),Convert_Targets(Test_Targ
```

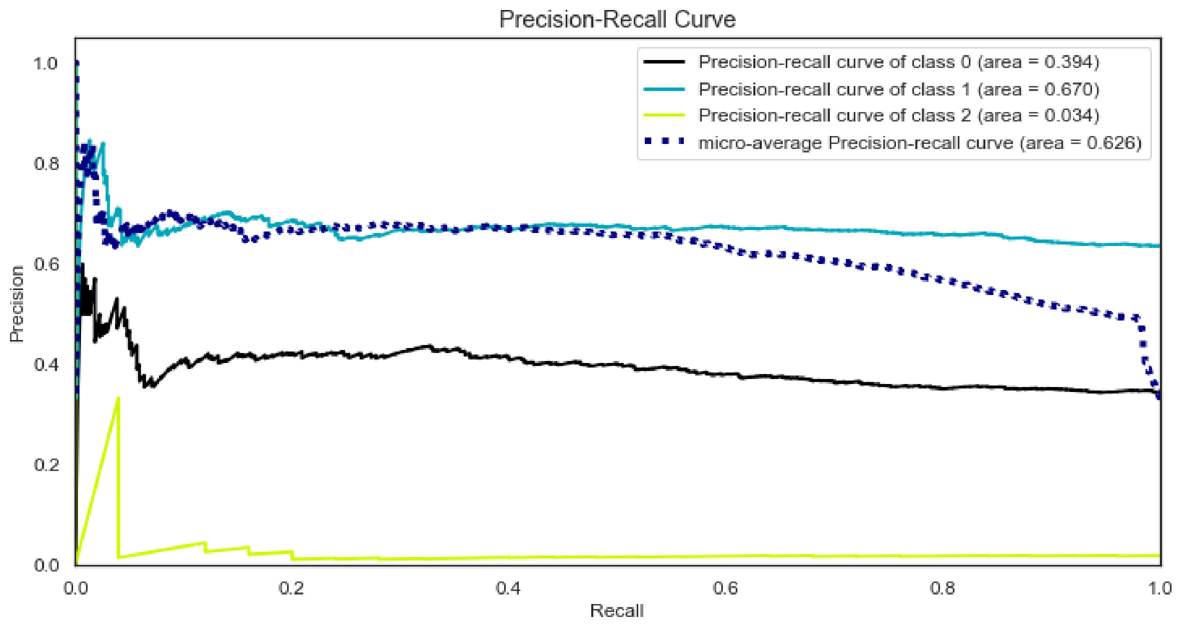
```
The accuracy is: 62.2066
```



## Optimized XGBoost Precision-Recall curve for the testing data

In [ ]:

```
skplt.metrics.plot_precision_recall_curve(Convert_Targets(Test_Targets), clf.best_estimator
                                          title="Precision-Recall Curve", figsize=(12,6));
```



## Table showing the areas under the Precision-Recall curve for the basic and

### optimized model

In [ ]:

```
xg_area_table = PrettyTable(["Model","Class 0","Class 1","Class 2"])
xg_area_table.add_row(["Basic",0.289,0.678,0.045])
xg_area_table.add_row(["Optimized",0.394,0.670,0.034])
print(xg_area_table)
```

```
+-----------+---------+---------+---------+
|   Model   | Class 0 | Class 1 | Class 2 |
+-----------+---------+---------+---------+
|   Basic   |  0.289  |  0.678  |  0.045  |
| Optimized |  0.394  |   0.67  |  0.034  |
+-----------+---------+---------+---------+
```

- The optimized model was able to achieve a 2% increase in accuracy.
- We also see that the optimized model was able to imporve its predictive capability on Class 0, however it dropped on predicting Class 2.
- However, it would be difficult for any model to learn Class 2 since there are extremely few occurences of Draws.

## (3.2) Neural Network

### Building the Neural Network

- We use this function to build a simple netwokr with 1 hidden layer and 10 neurons in the hidden layer.
- The default optimizer is Stochastic Gradient Descent and initializer is uniform.
- The loss function used is categorical loss entropy since we have a multiclass classification problem.

In [ ]:

```
def Create_Model(initalizer = 'random_normal',
                 optimizer = 'sgd',
                 hidden_layers = 1,
                 activation = "sigmoid",
                 neurons = 10,
                 Train_Features = Train_Features):

    NN = Sequential()
    NN.add(Dense(neurons,input_dim=Train_Features.shape[1],activation = activation,kernel_i

    if(hidden_layers>1):
        for i in range(hidden_layers):
            NN.add(Dense(neurons,activation=activation, kernel_initializer = initalizer))

    ## We always want the final layer to have 3 neurons with a softmax activation
    NN.add(Dense(3,activation='softmax',kernel_initializer=initalizer))


    NN.compile(loss='categorical_crossentropy',
           optimizer= optimizer,
           metrics=['accuracy'])

    return NN
```

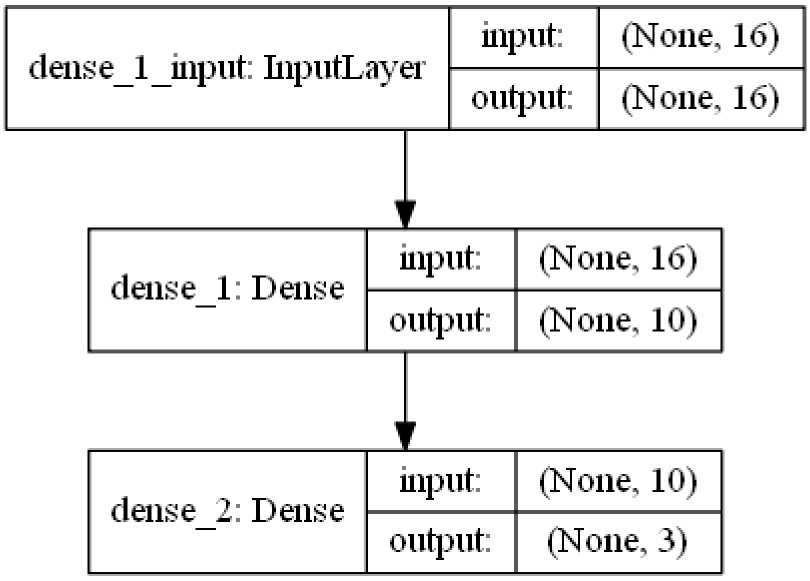
## (3.2.1) A simple Neural Network

- Using the basic setup we already have defined above

In [ ]:

```
simple_model = Create_Model()
plot_model(simple_model, to_file='model.png', show_shapes=True,show_layer_names=True)
```

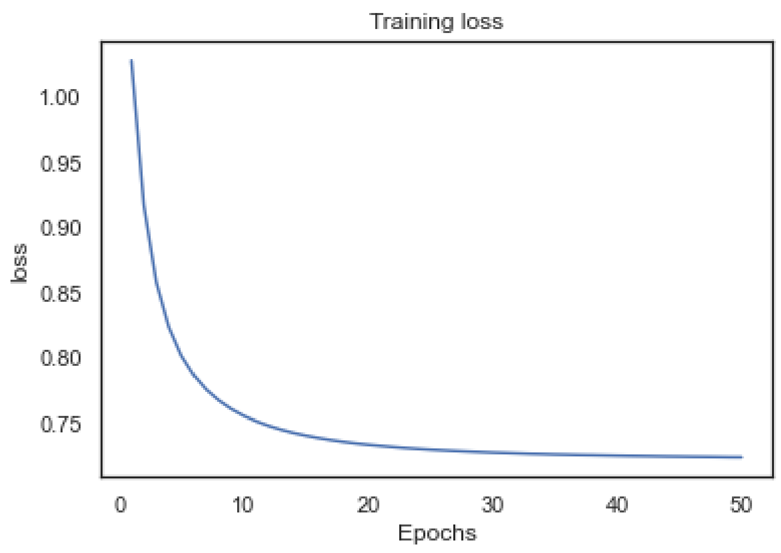
Out[48]:

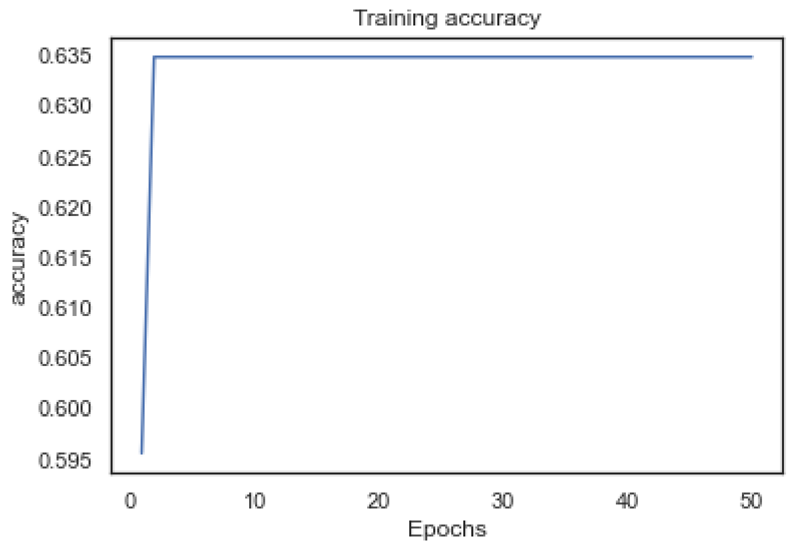


In [ ]:

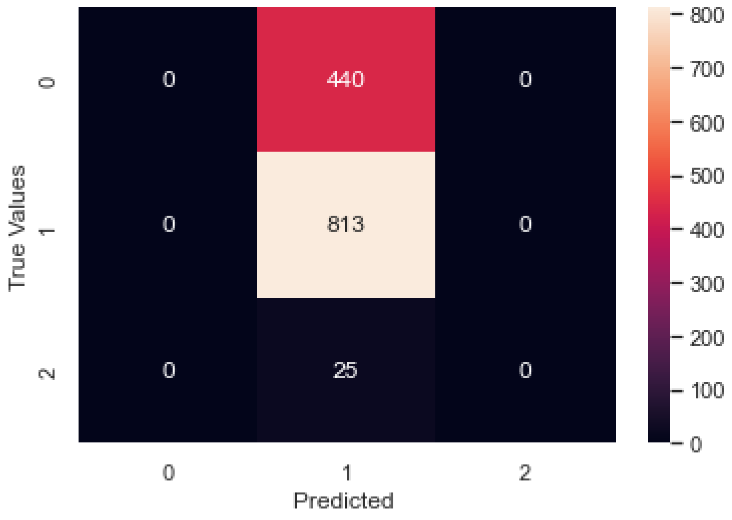
```
history = simple_model.fit(Train_Features,
                Train_Targets,
                epochs = 50,
                batch_size = 100,
                verbose = 0)

print("The training Error and Accuracy graphs: ")
plot_metric(history, "loss",False)
plot_metric(history, "accuracy",False)
train_nn_acc = accuracy_score(Convert_Targets(Train_Targets),Convert_Targets(simple_model.p


print("The Testing Accuracy: ")
test_pred = simple_model.predict(Test_Features)
test_nn_acc = getAccuracy(test_pred,Test_Targets,False)

nn_basic_table = PrettyTable(["Training Accuracy","Testing Accuracy"])
nn_basic_table.add_row([train_nn_acc*100,test_nn_acc*100])
```

```
WARNING:tensorflow:From C:\Users\yasee\.conda\envs\tensorflow\lib\site-pack
ges\keras\backend\tensorflow_backend.py:422: The name tf.global_variables is
deprecated. Please use tf.compat.v1.global_variables instead.

The training Error and Accuracy graphs:
```





```
The Testing Accuracy:
The accuracy is: 63.615
```



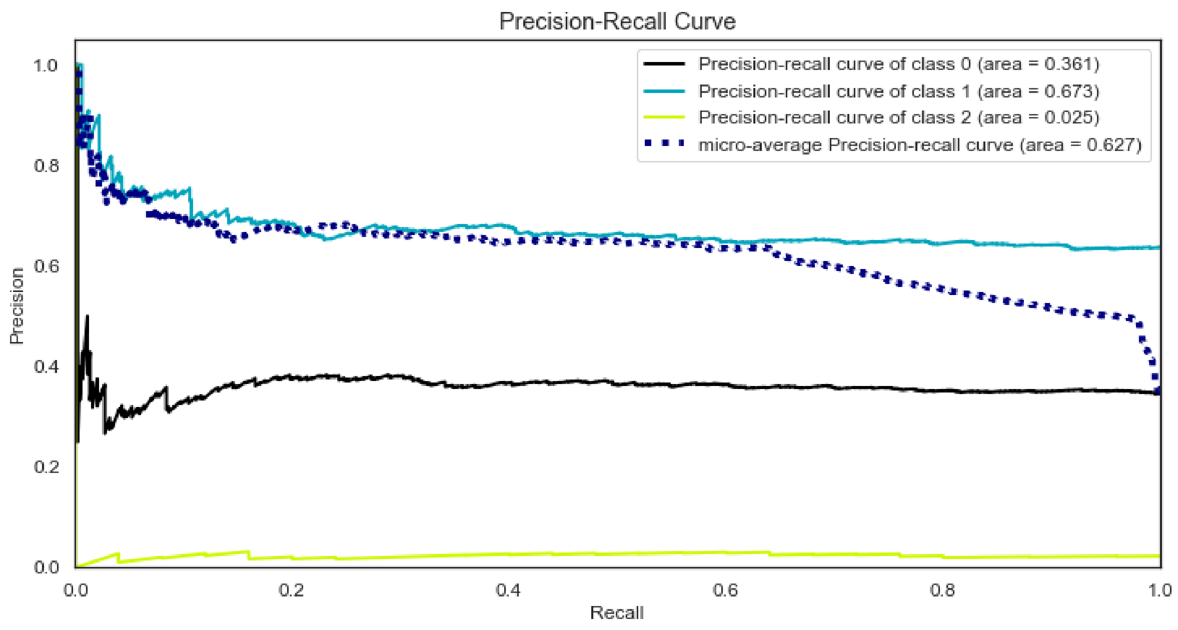
- From the confusion matrix we see that the model simply predicted class 1 for every instance, and since class 1 is the majority class, it achieves a decent accuracy. The fact that the model can only predict a single class is indicative that learning stopped early.
- Looking at the accuracy graph, we see that the model reaches an accuracy of approximately 63.5% and thereafter stops improving. From the loss graph we see that the loss function stops decreasing once it reaches a value of approximately 0.75. Both of these indicate that the model eventually stopped learning. Considering we utilised a sigmoid function and just randomly initialised the weights, learning could have stopped due to vanishing gradients. Since we utilise gradients in order to update our weights, when the gradients become extremely small such that they are approximately 0, no weight adjustment occurs and thus learning stops. This is the problem of vanishing gradients.

## (3.2.1.1) Analysing the basic models results

## Precision-Recall curve for the testing data

In [ ]:

```
skplt.metrics.plot_precision_recall_curve(Convert_Targets(Test_Targets), test_pred,
                                          title="Precision-Recall Curve", figsize=(12,6));
```



- The area under the each line of the precision-recall curve to be high since this will indicate the model is capable of predicting the correct classes.
- However, we can clearly see that the class 1 achieves the best area under the curve, which indicates the model is good at predicting this class and predicts the other classes poorly.
- This is due to the fact that our targets are imbalanced, and 1 which represents the Red corner, which won majority of the fights.
- The lowest area is represented by class 2 which represents a draw and was the smallest represented outcome.
- The basic XGBoost classifier also has similar areas than the basic NN and achieves similar accuracies.

In [ ]:

```
print(nn_basic_table)
```

```
+-------------------+------------------+
| Training Accuracy | Testing Accuracy |
+-------------------+------------------+
| 63.480885311871226 | 63.6150234741784 |
+-------------------+------------------+
```

## (3.2.2) Optimizing the Neural Network

- As described by Manuel Fokam, the algoirthm parameters define how you learn a model.
- In the case of Neural Networks, there a various parameters we would like to optimize:
  1. The weights initializer: The way the weight matrices are initialized.
  2. The optimizer: The algorithm used to minimize the loss function.

3. The batch size: This decides how many data points are propagated through the network before an update is made to the weights.
4. Epochs : The number of iterations we allow training to go on for.
5. The number of hidden layers
6. The number of neurons per hidden layer
7. The activation functions to be used within the hidden layers.

## Weights Initializers Explained:

### 1. Glorot initialization:

- This is the preferred method of initialization when the activation function is a Tanh or Sigmoid.
- The weights are sampled using a uniform distribution or a normal distribution.
- Using a uniform distribution, the weights are generated using the following formula: $W = U[-\sqrt{\frac{6}{N_{in}+N_{out}}}, \sqrt{\frac{6}{N_{in}+N_{out}}}]$, where $N_{in}$ represents the number of neurons from the current layer and $N_{out}$ represents the number of neurons in the layer the weights connect to.
- Using a noram distribution, the weights are generated using the following formula: $W = G(0, \sqrt{\frac{6}{N_{in}+N_{out}}})$.
- Note that $\sqrt{\frac{6}{N_{in}+N_{out}}}$ represents the Variance.
- This ensures that the variance between weights is not large and relatively similar throughout the layers.
- By keeping the variance the same across the various layers, we are able to combat the problem of vanishing and exploding gradients which hinders learning.

### 2. He initialization:

- This is the preferred method of initialization when the activation function is a ReLU.
- The formulae is exactly the same as Glorot, the only different is that we multiply the variance by 2.
- The reason we do this is because the ReLU turns half of the Z-values (the negative ones) into zeros, effectively removing about half of the variance. So, we need to double the variance of the weights to compensate for it.

All the above information can be found here: https://towardsdatascience.com/hyper-parameters-in-action-part-ii-weight-initializers-35aee1a28404 (https://towardsdatascience.com/hyper-parameters-in-action-part-ii-weight-initializers-35aee1a28404) .

## Optimizers Explained:

### 1. Gradient Descent:

- Gradient Descent is the most basic but most used optimization algorithm. It's used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.
- Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.
- It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized.

### 2. Stochastic Gradient Descent:

- It’s a variant of Gradient Descent that tries to update the model’s parameters more frequently by making an update after each point is propagated through the network.
- As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities. Thus we take steps in non-optimal directions. This leads to high variance amongst the model parameters which leads to a non-uniform convergence path to the minima.
- However since we update the weights more frequently, convergence is faster. But the caveat is that we may overshoot the minima and diverge.
- This method is preferred if the dataset is large since we don't require the entire dataset to be in RAM.
- The better alternative is called **Mini-Batch Gradient Descent**. Which propagates a smaller batch through the network before making an update to the weights.
- This combines the speed of SGD with the Uniformity of general GD.

**3. Adagrad:**

- In most other optimizers, the learning rate is kept constant for all the weights at every epoch.
- This optimizer changes the learning rate. It changes the learning rate ‘η’ for each parameter and at every time step ‘t’.
- It works on the derivative of an error function and also works using the second order derivative.
- η is a learning rate which is modified for given parameter θ(i) at a given time based on previous gradients calculated for given parameter θ(i).
- The update formula is as follow: $\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,i}+\epsilon}} . g_{t,i}$ , where $g_{t,i} = \nabla_\theta J(\theta_t, i)$.
- We store the sum of the squares of the gradients w.r.t. θ(i) up to time step t, while ϵ is a smoothing term that avoids division by zero.
- It makes big updates for less frequent parameters and a small step for frequent parameters.
- A problem with AdaGrad is that it can slow the search down too much, resulting in very small learning rates for each parameter or dimension of the search by the end of the run. This has the effect of stopping the search too soon, before the minimal can be located.

**4. RMSprop:**

- This is an extension of Adagrad.
- In AdaGrad we take the cumulative summation of squared gradients but, in RMSP we take the 'exponential average'.
- An optimizer which uses a decaying average or moving average of the partial derivative in the calculation of the learning rate for each parameter.
- Using a decaying moving average of the partial derivative allows the search to forget early partial derivative values and focus on the most recently seen shape of the search space.
- The update formula is as follow: $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{V_t+\epsilon}} . g_t$, where $V_t = \beta * V_{t-1} + (1 - \beta) * g_t^2$, where $g_t = \nabla_\theta J(\theta_t)$.
- Uses a moving average of squared gradients to normalize the gradient which is denoted by $g_t^2$.
- This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding and increasing the step for small gradients to avoid vanishing.

**5. Adam:**

- This optimizer is a combination of Gradient Descent with momentum and RMSprop.
- Momentum is used to accelerate the gradient descent algorithm by taking into consideration the ‘exponentially weighted average’ of the gradients. Using averages makes the algorithm converge towards the minima in a faster pace.
- By combining these two optimizations, Adam is able to converge much faster but is computationally expensive.

- The update formula is given by: $\theta_{t+1} = \theta_t - \frac{\eta * M_t}{\sqrt{V_t + \epsilon}} \cdot g_t$, where $V_t = \beta_2 * V_{t-1} + (1 - \beta_2) * g_t^2$ and $M_t = \beta_1 * M_{t-1} + (1 - \beta_1) * g_t$, where $g_t = \nabla_\theta J(\theta_t)$.

**We will now perform a random search to find the optimal values for the following:**

1. The initialization method to use.
2. The optimization technique to use.
3. The number of epochs to use.
4. The best batch-size to use.
5. The best activation function for the hidden layer.

Note: That the architecture for this problem is kept constant and is the simple architecture used before. The simple architecture is as follows:

- A single hidden layer with 10 neurons.
- The output layer has 3 nodes and a softmax activation function.

In [ ]:

```
# fix random seed for reproducibility (this might work or might not work
# depending on each library's implenentation)
seed = 7
np.random.seed(seed)
# create the sklearn model for the network
model_NN = KerasClassifier(build_fn=Create_Model,
                                            verbose=0)

## A list of initializers we would like to choose from
init_mode = ['glorot_uniform', 'he_uniform']
## The batch sizes we want to consider
batches = [50,128, 512,1000]
## The number of epochs we want to allow
epochs = [10, 20,50,100]
## The optimizers we want to investigate
optimizer = ["sgd","adagrad","rmsprop","adam"]
## The set of activation functions
activations = [tf.nn.leaky_relu,"sigmoid","tanh","relu"]

# grid search for initializer, batch size , number of epochs, optimizer and activation func
param_grid = dict(epochs=epochs, batch_size=batches, initalizer=init_mode, optimizer = opti


search_nn = RandomizedSearchCV(estimator=model_NN,
                         param_distributions=param_grid,
                         scoring='accuracy',
                         n_iter=25,
                         cv = 2,
                         verbose=0,
                          random_state=42)

search_nn.fit(Train_Features, Convert_Targets(Train_Targets))
```

Out[52]:

```
RandomizedSearchCV(cv=2,
                   estimator=<keras.wrappers.scikit_learn.KerasClassifier ob
ject at 0x000001E0A1653348>,
                   n_iter=25,
                   param_distributions={'activation': [<function leaky_relu
at 0x000001E099EA8318>,
                                                       'sigmoid', 'tanh',
                                                       'relu'],
                                        'batch_size': [50, 128, 512, 1000],
                                        'epochs': [10, 20, 50, 100],
                                        'initalizer': ['glorot_uniform',
                                                       'he_uniform'],
                                        'optimizer': ['sgd', 'adagrad',
                                                      'rmsprop', 'adam']},
                   random_state=42, scoring='accuracy')
```

In [ ]:

```
NN_df = pd.DataFrame(search_nn.cv_results_)
NN_df = NN_df.dropna()
print("Best parameters:", search_nn.best_params_)
print("Best accuracy: ", search_nn.best_score_*100)
```

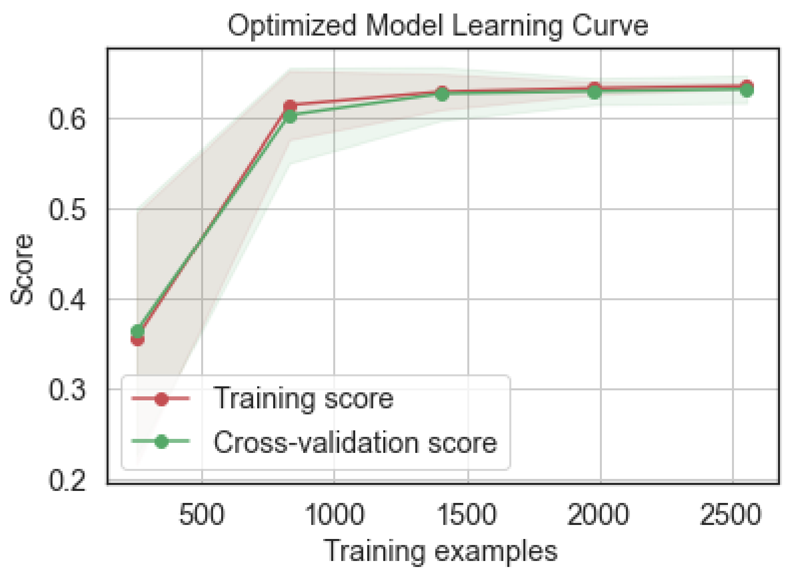
```
Best parameters: {'optimizer': 'adam', 'initalizer': 'glorot_uniform', 'epo
hs': 10, 'batch_size': 50, 'activation': 'sigmoid'}
Best accuracy:  63.581488933601605
```

- We can see that the optimal initializer is glorot_uniform and the optimal activation is a sigmoid. This is expected since glorot initialization was based on having the sigmoid or tanh as the activation function.
- The optimal optimizer is adam which was also expected since it converges the fastest and is currently the best optimizer since it combines the intuition of other optimizers.

## Cross Validation Performance Plot for Partially optimized Network

In [ ]:

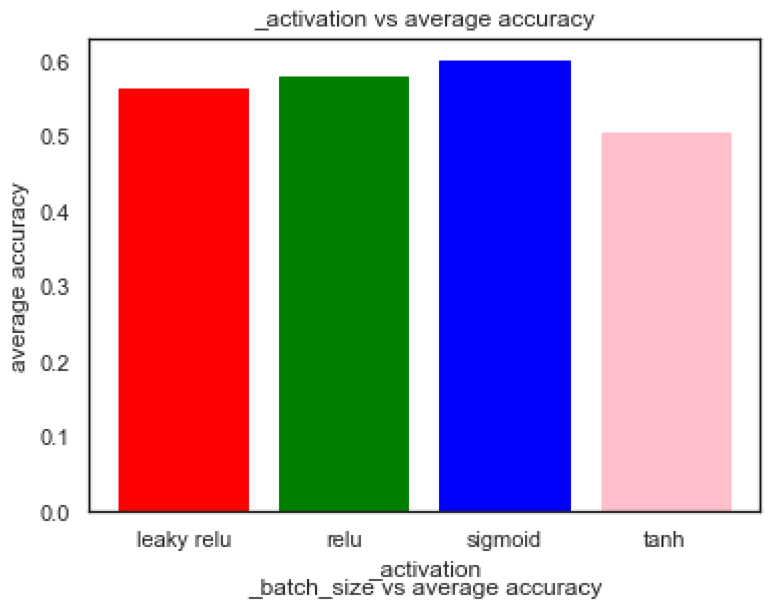
```
skplt.estimators.plot_learning_curve(search_nn.best_estimator_, Train_Features, Convert_Tar
                                     cv=7, shuffle=True, scoring="accuracy", n_jobs=-1,
                                     figsize=(6,4), title_fontsize="large", text_fontsize="
                                     title="Optimized Model Learning Curve ")
plt.show()
```

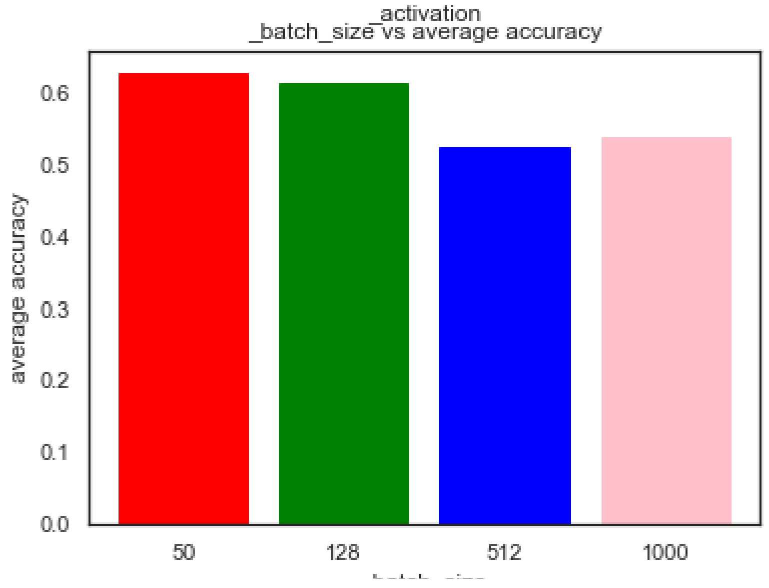


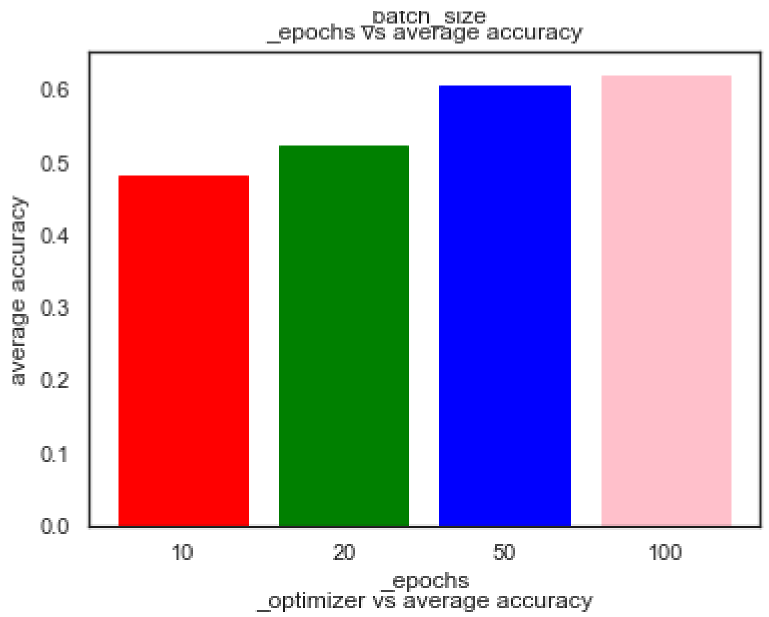
## Bar plot of the hyperparameters and their average accuracies

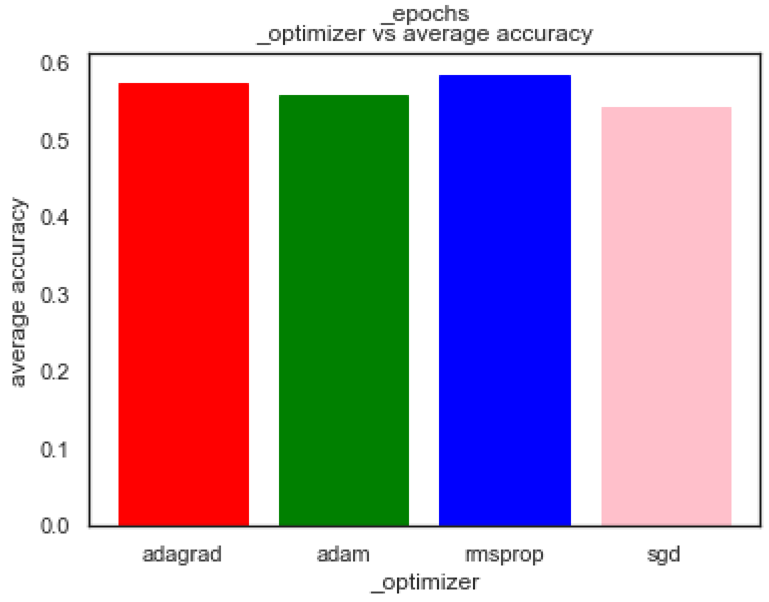
In [ ]:

```
params = ["param_activation","param_batch_size","param_epochs","param_optimizer"]
fig,ax = plt.subplots(nrows= 4, ncols=1, figsize=(6,20))
for i in range(len(params)):
    g = NN_df.groupby(params[i])["mean_test_score"]
    param_means = []
    keys = list(g.groups.keys())
    for a in keys:
        param_means.append(g.get_group(a).mean())
    if(i==0):
        keys[0] = "leaky relu"
    if(i==1 or i==2):
        keys = list(map(str, keys))

    bars = ax[i].bar(keys,param_means)
    colors = ["Red","Green","Blue","pink"]
    for j in range(len(colors)):
        bars[j].set_color(colors[j])

    ax[i].set_ylabel("average accuracy")
    name = params[i].replace("param",'')
    ax[i].set_xlabel(name)
    ax[i].set_title(f"{name} vs average accuracy")

plt.show()
```









- We can see that the leaky ReLU and the Sigmoid activation functions achieve the best accuracies with the sigmoid being slightly better. Even though the sigmoid usually suffers from vanishing gradients, the utilisation of glorot initialisation is the reason for its improved performance.
- The optimal batch size is 50 since this will enable more batches to pass through the network and more frequent weight updates.
- Having higher epochs allows the model to train longer and capture the trends of the data. Usually this might lead to overfitting but this is not the case for our data.

**We now have the optimal values for the activation function, epochs, batch size, initializer and optimizer.**

In [ ]:

```
def Create_Model_2(initalizer = 'glorot_uniform',
                 optimizer = 'adam',
                 hidden_layers = 1,
                 activation = "sigmoid",
                 neurons = 10,
                 Train_Features = Train_Features):

    NN = Sequential()
    if(activation == "sigmoid" or activation == "tanh"):
        NN.add(Dense(neurons,input_dim=Train_Features.shape[1],activation = activation,kern
    else:
        NN.add(Dense(neurons,input_dim=Train_Features.shape[1],activation = activation,kern

    if(hidden_layers>1):
        for i in range(hidden_layers):
            if(activation == "sigmoid" or activation == "tanh"):
                NN.add(Dense(neurons,activation=activation, kernel_initializer = "glorot_un
            else:
                NN.add(Dense(neurons,activation=activation, kernel_initializer = "he_unifor

    ## We always want the final layer to have 3 neurons with a softmax activation
    NN.add(Dense(3,activation='softmax',kernel_initializer=initalizer))


    NN.compile(loss='categorical_crossentropy',
           optimizer= optimizer,
           metrics=['accuracy'])

    return NN
```

## Optimizing the architecture of the Network

- We are going to another search by varying the number of hidden layers, their number of neurons and testing the activation functions again with the new architectures.

In [ ]:

```
params_architecture = {
    "hidden_layers": [1,3,5,10],
    "neurons": [10,20,30,50],
    "activation" : [tf.nn.leaky_relu,"sigmoid","tanh","relu"]
}

model_NN_architecture = KerasClassifier(build_fn=Create_Model_2,
                                        verbose=0)

search_nn_architecture = RandomizedSearchCV(estimator=model_NN_architecture,
                         param_distributions=params_architecture,
                         scoring='accuracy',
                         n_iter=25,
                         cv = 2,
                         verbose=0,
                        random_state= 42)

search_nn_architecture.fit(Train_Features,
                           Convert_Targets(Train_Targets),
                           epochs =search_nn.best_params_["epochs"],
                           batch_size = search_nn.best_params_["batch_size"],
                           )
```

Out[57]:

```
RandomizedSearchCV(cv=2,
                   estimator=<keras.wrappers.scikit_learn.KerasClassifier ob
ject at 0x000001E0C65227C8>,
                   n_iter=25,
                   param_distributions={'activation': [<function leaky_relu
at 0x000001E099EA8318>,
                                                        'sigmoid', 'tanh',
                                                        'relu'],
                                         'hidden_layers': [1, 3, 5, 10],
                                         'neurons': [10, 20, 30, 50]},
                   random_state=42, scoring='accuracy')
```

In [ ]:

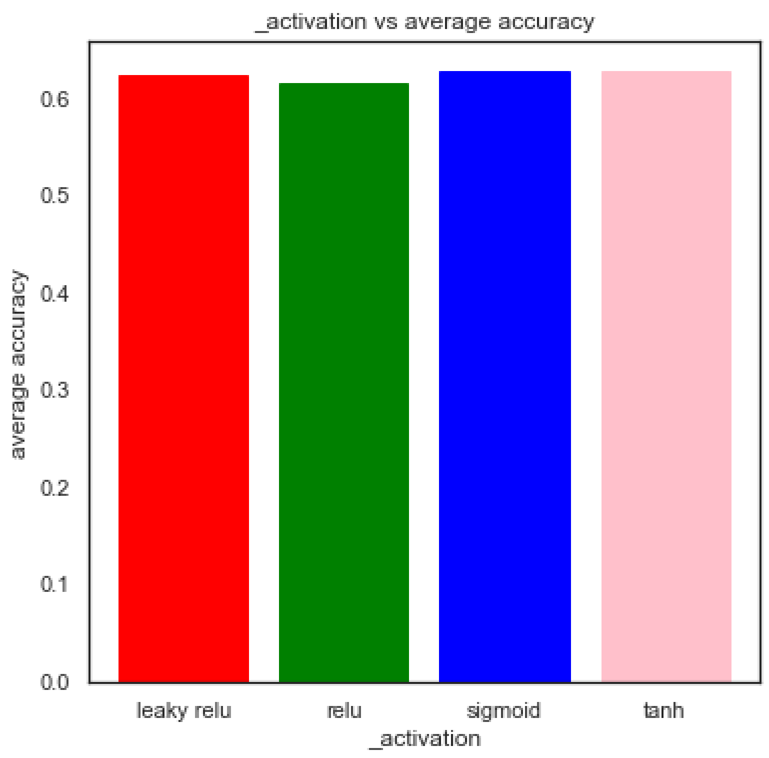
```
NN_df_architecture = pd.DataFrame(search_nn_architecture.cv_results_)
NN_df_architecture = NN_df_architecture.dropna()
print("Best parameters:", search_nn_architecture.best_params_)
print("Best accuracy: ", search_nn_architecture.best_score_*100)
```

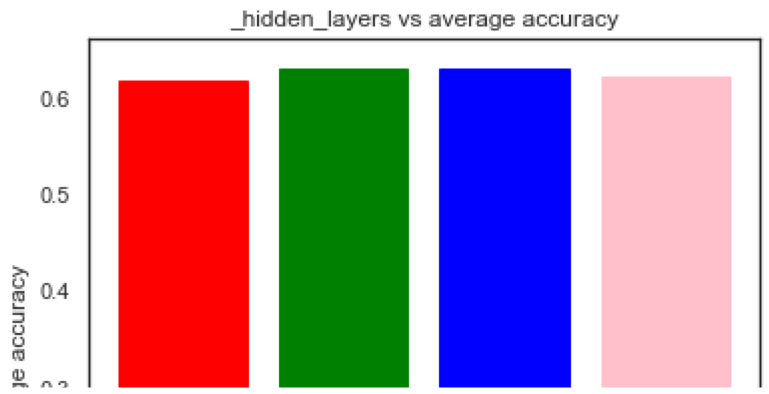
```
Best parameters: {'neurons': 10, 'hidden_layers': 3, 'activation': 'tanh'}
Best accuracy:  63.84976525821596
```

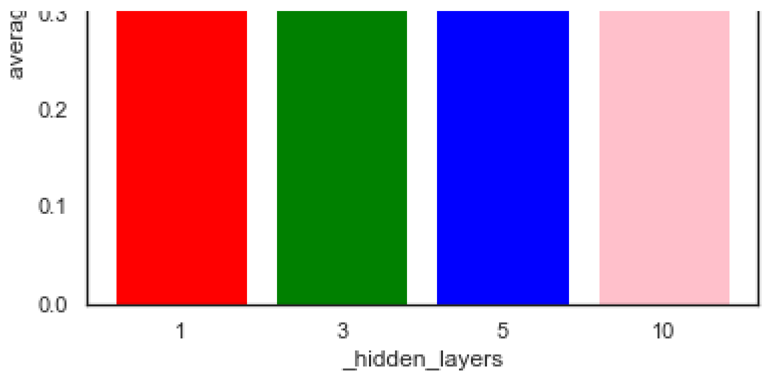
### Bar plot of the architecture hyperparameters and their average accuracies

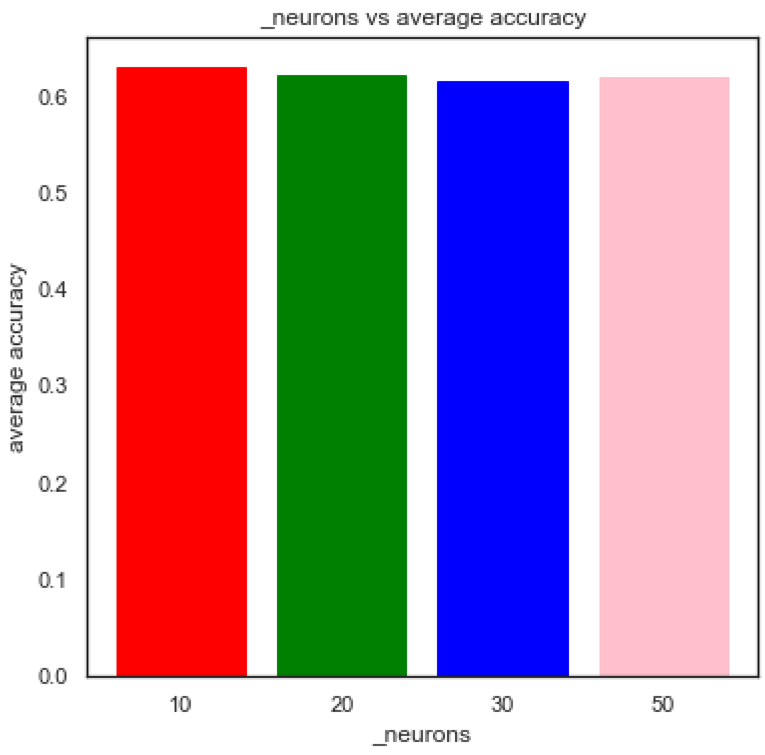
In [ ]:

```
params = ["param_activation","param_hidden_layers","param_neurons"]
fig,ax = plt.subplots(nrows= 3, ncols=1, figsize=(6,20))
for i in range(len(params)):
    g = NN_df_architecture.groupby(params[i])["mean_test_score"]
    param_means = []
    keys = list(g.groups.keys())
    for a in keys:
        param_means.append(g.get_group(a).mean())
    if(i==0):
        keys[0] = "leaky relu"
    if(i==1 or i==2):
        keys = list(map(str, keys))

    bars = ax[i].bar(keys,param_means)
    colors = ["Red","Green","Blue","pink"]
    for j in range(len(colors)):
        bars[j].set_color(colors[j])

    ax[i].set_ylabel("average accuracy")
    name = params[i].replace("param",'')
    ax[i].set_xlabel(name)
    ax[i].set_title(f"{name} vs average accuracy")

plt.show()
```







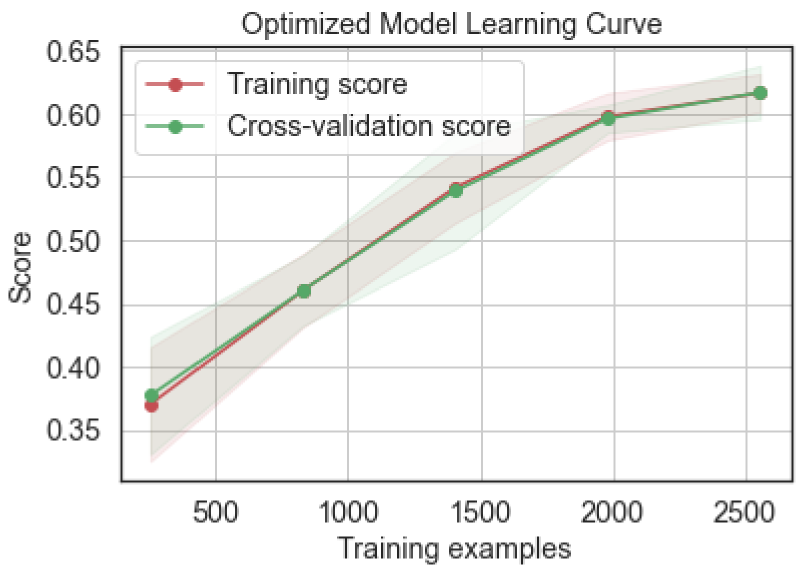


- We can see that all the activations perform similarly. This is due to the fact that we have ensured then when a sigmoid or tanh is used, we utilise the glorot initialization and when the ReLU or leaky ReLU are used, we use the He initialization. This enables us to reduce the effects of vanishing or exploding gradients.
- We can also see that there is no improved performance from 10 neurons or more. The performance drops since the model tends to overfit and thus performs poorly on the cross-validation data.
- We also see that an increase in performance as we increase the hidden layers. However, once we surpass 5 hidden layers we drop in our cross-validation performace since we tend to overfit.

## Cross Validation Performance Plot for optimized model.

In [ ]:

```
skplt.estimators.plot_learning_curve(search_nn_architecture.best_estimator_, Train_Features
                                     cv=7, shuffle=True, scoring="accuracy", n_jobs=-1,
                                     figsize=(6,4), title_fontsize="large", text_fontsize="
                                     title="Optimized Model Learning Curve ")
plt.show()
```



## The optimal hyperparameters are:

**1. The weights initializer: glorot_uniform.**

**2. The optimizer: adam.**

**3. The batch size: 50**

**4. The Epochs : 10**

**5. The number of hidden layers: 3**

**6. The number of neurons per hidden layer: 10**

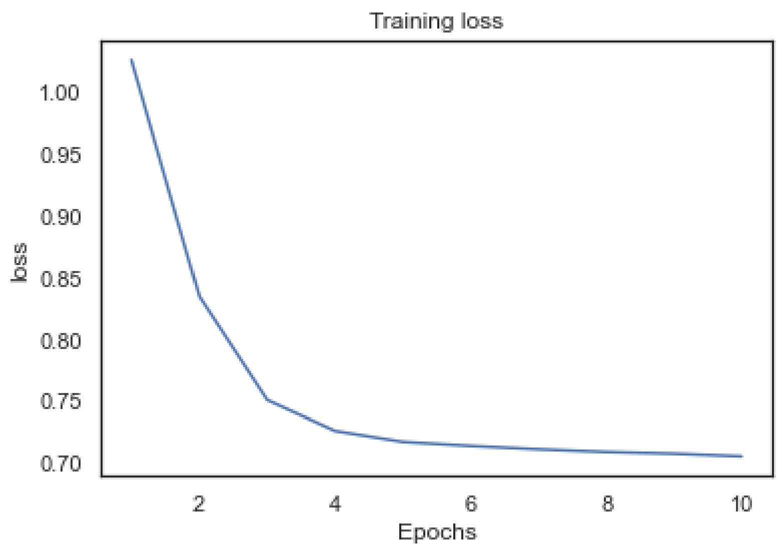
**7. The activation functions to be used within the hidden layers.: Tanh**

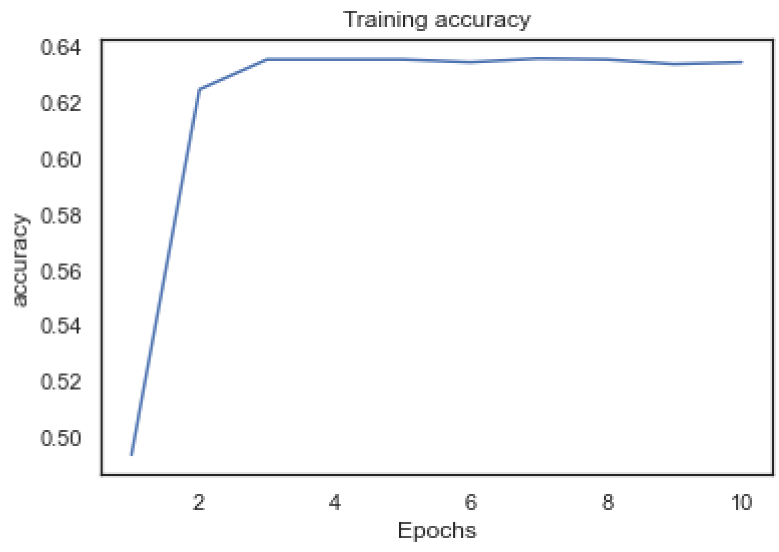
## Building the optimal Neural Network

In [ ]:

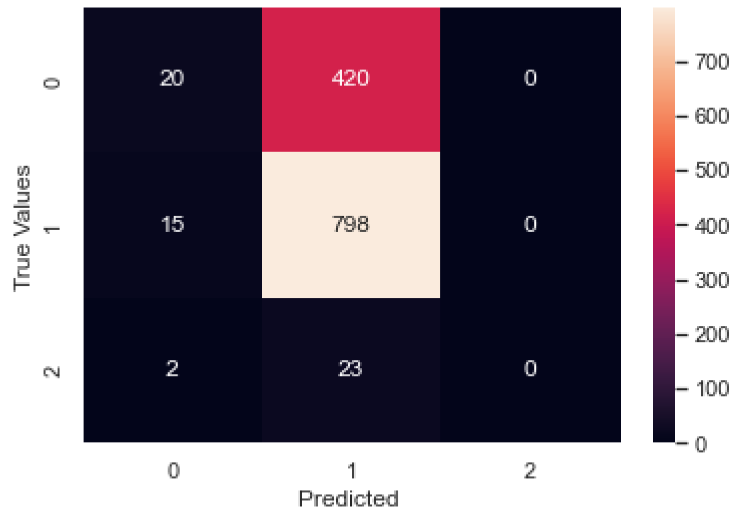
```
optimal_nn = Create_Model_2(initalizer = search_nn.best_params_["initalizer"],
                 optimizer = search_nn.best_params_["optimizer"],
                 hidden_layers = search_nn_architecture.best_params_["hidden_layers"],
                 activation =  search_nn_architecture.best_params_["activation"],
                 neurons =  search_nn_architecture.best_params_["neurons"])

optimal_history = optimal_nn.fit(Train_Features,
                                 Train_Targets,
                                 epochs = search_nn.best_params_["epochs"],
                                 batch_size = search_nn.best_params_["batch_size"])

print("The training Error and Accuracy graphs: ")
plot_metric(optimal_history, "loss",False)
plot_metric(optimal_history, "accuracy",False)
train_nn__optimal_acc = accuracy_score(Convert_Targets(Train_Targets),Convert_Targets(optim


print("The Testing Accuracy: ")
test_pred = optimal_nn.predict(Test_Features)
test_nn_optimal_acc = getAccuracy(test_pred,Test_Targets,False)

nn_optimal_table = PrettyTable(["Training Accuracy","Testing Accuracy"])
nn_optimal_table.add_row([train_nn__optimal_acc*100,test_nn_optimal_acc*100])
```

```
Epoch 1/10
2982/2982 [==============================] - 1s 292us/step - loss: 1.0264 -
accuracy: 0.4940
Epoch 2/10
2982/2982 [==============================] - 0s 63us/step - loss: 0.8349 - a
ccuracy: 0.6244
Epoch 3/10
2982/2982 [==============================] - 0s 57us/step - loss: 0.7510 - a
ccuracy: 0.6351
Epoch 4/10
2982/2982 [==============================] - 0s 59us/step - loss: 0.7261 - a
ccuracy: 0.6351
Epoch 5/10
2982/2982 [==============================] - 0s 54us/step - loss: 0.7173 - a
ccuracy: 0.6351
Epoch 6/10
2982/2982 [==============================] - 0s 57us/step - loss: 0.7141 - a
ccuracy: 0.6341
Epoch 7/10
2982/2982 [==============================] - 0s 55us/step - loss: 0.7114 - a
ccuracy: 0.6355
Epoch 8/10
2982/2982 [==============================] - 0s 58us/step - loss: 0.7091 - a
ccuracy: 0.6351
Epoch 9/10
2982/2982 [==============================] - 0s 55us/step - loss: 0.7078 - a
ccuracy: 0.6335
Epoch 10/10
2982/2982 [==============================] - 0s 54us/step - loss: 0.7056 - a
ccuracy: 0.6341
The training Error and Accuracy graphs:
```





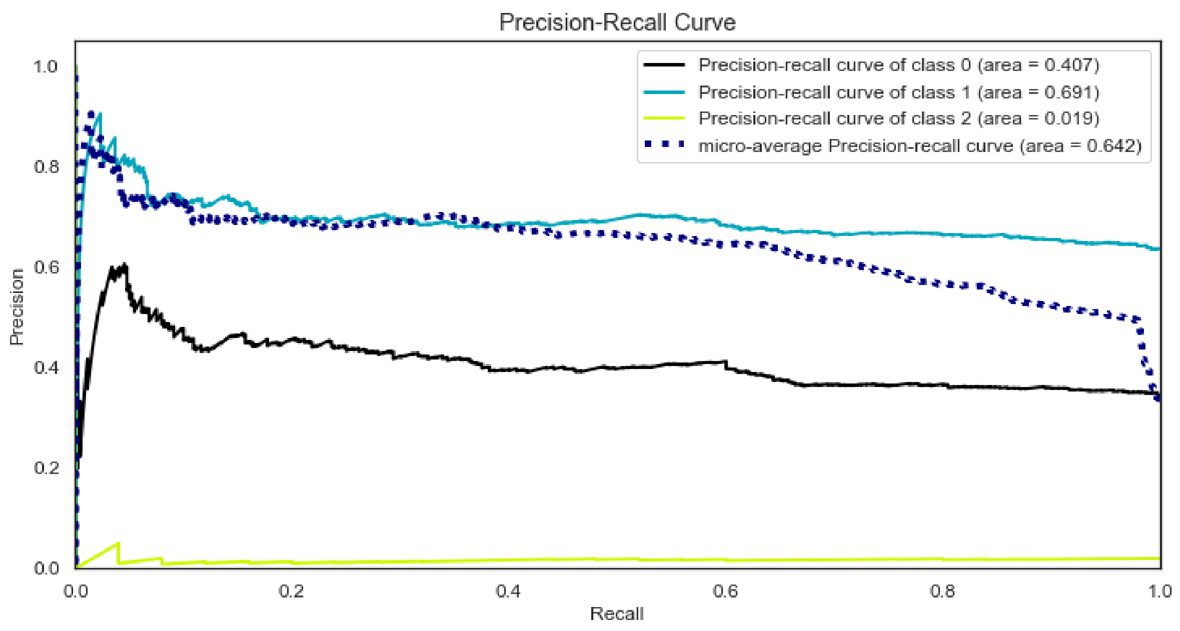
```
The Testing Accuracy:
The accuracy is: 64.0063
```



## Precision-Recall curve for the testing data

In [ ]:

```
skplt.metrics.plot_precision_recall_curve(Convert_Targets(Test_Targets), test_pred,
                                          title="Precision-Recall Curve", figsize=(12,6));
```

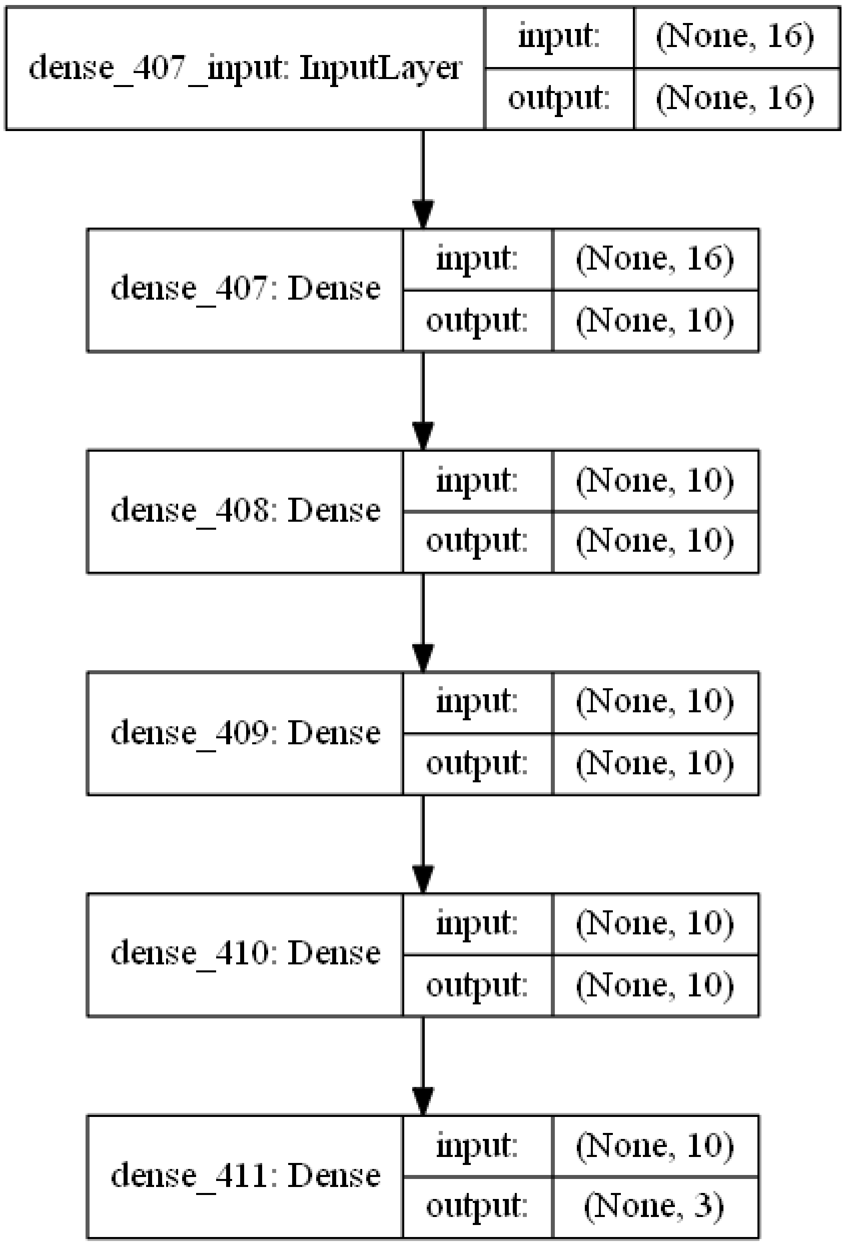


## Optimal Architecture

In [ ]:

```
plot_model(optimal_nn, to_file='model.png', show_shapes=True,show_layer_names=True)
```

Out[71]:



dense_407_input: InputLayer | input: | (None, 16) | output: | (None, 16)
dense_407: Dense | input: | (None, 16) | output: | (None, 10)
dense_408: Dense | input: | (None, 10) | output: | (None, 10)
dense_409: Dense | input: | (None, 10) | output: | (None, 10)
dense_410: Dense | input: | (None, 10) | output: | (None, 10)
dense_411: Dense | input: | (None, 10) | output: | (None, 3)

### (3.2.2.1) Analysing the optimal models results

In [ ]:

```
nn_area_table = PrettyTable(["Model","Class 0","Class 1","Class 2"])
nn_area_table.add_row(["Basic",0.361,0.673,0.025])
nn_area_table.add_row(["Optimized",0.407,0.691,0.019])
print(nn_area_table)
```

```
+-----------+---------+---------+---------+
|   Model   | Class 0 | Class 1 | Class 2 |
+-----------+---------+---------+---------+
|   Basic   |  0.361  |  0.673  |  0.025  |
| Optimized |  0.407  |  0.691  |  0.019  |
+-----------+---------+---------+---------+
```

- We see that the model was able to achieve a higher test accuracy but only a 0.4% increase.
- We see that the optimized model has improved in its ability to predict the Classes 0 and 1, however it decreased in it's power to predict the minority class correctly.
- However, the accuracies cannot be compared since the basic model simply predicted class 1 and since it is a majority class, it achieved a high accuracy score.
- The inability of the neural network to increase its performance is due to the fact the data is imbalanced in it's classes. Consequently, it is a strong predictor of the majority class, Class 1, relatively well on the second largest class and poorly on the least ocurring class.

## Conclusion

- Both the neural network and gradient boosting model achieved similar results.
- This however is not due to the models, but rather the limitations within the dataset.
- The major limitation within the data is that of imbalanced classes.
- Another possible limitation for neural network could be the amount of data. We were only able to use approximately 4000 points for both training and testing. This might not be sufficient for the neural network.

Created in **Deepnote**(https://deepnote.com?utm_source=created-in-deepnote-cell&projectId=664729b2-75bf-4289-a3fe-9e10aaaaa8be)