Final Project Submission

- Student Name: Deztany Jackson
- Student Pace: Self Pace
- Scheduled Project Review: February 24, 2023
- Instructor Name: Morgan Jones
- Blog Post URL: http://dmvinedata.com/sysofinterest/

Classification of Paris Real Estate Property - Luxury vs Basic

 Paris

Business Understanding

Certain Paris Real Estate (RE) investment agencies are looking to invest in luxury properties to lease out to Hotels, VIP guests of the city and Companies for their events, employees and clients. There are no rules for foreigners on owning property in France. As one of the most expensive real estate cities in the world picking prime real estate to buy at the right time is a safe investment.

Classification models will be built to correctly identify "Luxury" property and the best data attributes (features) that help make the best predictions.

It is more important to correctly identify "Luxury" property as best as possible and minimize incorrectly identifying "Basic" property. This will cost having a long list of "Luxury" property but help the RE Investment agencies feel confident in the smaller set of listings.

Due to data imbalance and business goal our main metrics will be : Precision and F1

F1 Score Metric, Joos Kortanje, 2021

Data Understanding

This project uses dataset from kaggle and used for Educational Purposes "ParisHousing.csv". The intial dataset starts with 10,000 Rows of data and 18 Features (including "Category" the target value).

Paris Kaggle Dataset, 2021

Import Libraries & Packages

In [1]:
```
#Import packages

import pandas as pd
import numpy as np
```

```
from matplotlib import pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_sc
from sklearn.dummy import DummyClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import plot_confusion_matrix,classification_report,recall_s
from sklearn.metrics import precision_score, f1_score, confusion_matrix, plot_ro
from sklearn.metrics import precision_recall_curve,average_precision_score, make
from sklearn.metrics import plot_precision_recall_curve
from sklearn import tree

import warnings
warnings.filterwarnings('ignore')
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImPipeline
```

Importing the data and checking the counts of the target variable

In [2]:
```
# Import csv file
paris_df = pd.read_csv("data/ParisHousingClass.csv")
paris_df.head()
```

Out[2]:

	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	cityPartRange	numPrevOw
0	75523	3	0	1	63	9373	3	
1	80771	39	1	1	98	39381	8	
2	55712	58	0	1	19	34457	6	
3	32316	47	0	0	6	27939	10	
4	70429	19	1	1	90	38045	3	

Exploring Paris Dataset

Loan Classification,Kamal Khumar, 2021

(0; 1; 2; 3+)(Number)

Feature	Description	Data Type
Category	Classification of Real Estate (Basic or Luxury)	Nominal Number
Price	Cost of RE (Units unknown)	Cardinal Number
Building_Size(m)	Size of the building (meters)	Cardinal Number
Has_Yard	Has yard (0:No,1:Yes)	Nominal Number
Has_Pool	Has Pool (0:No,1:Yes)	Nominal Number

Feature	Description	Data Type
Num_of_Floors	Number of Floors	Nominal Number
Num_of_Rooms	Number of Rooms (0; 1; 2; 3+)	Nominal Number
City_Part_Range	Thehigher the range, the more exclusive the neighbourhood is	Cardinal Number
City_Code	Zipcode	Nominal Number
Num_Previous_Owners	Number of Previous Ownder	Cardinal Number
Garage_Size(m)	Size of the garage (meters)	Cardinal Number
Attic_Size(m)	Size of the attic (meters)	Cardinal Number
Has_Storm_Protector	Has Storm Protector (0:No,1:Yes)	Nominal Number
Basement_Size(m)	Size of the basement (meters)	Cardinal Number
Year_Built	Year built	Nominal Number
Num_of_Guest_Rooms	Number of Guest Rooms (0; 1; 2; 3+)	Cardinal Number
Has_Storage_Room	Has Storage Room (0:No,1:Yes)	Nominal Number

In [3]:
```
#Shape of dataset
paris_df.shape
```

Out[3]:
```
(10000, 18)
```

In [4]:
```
#Value count of dataset
paris_df["category"].value_counts()
```

Out[4]:
```
Basic     8735
Luxury    1265
Name: category, dtype: int64
```

In [5]:
```
#Value count of dataset
paris_df["cityCode"].value_counts()
```

Out[5]:
```
36929    3
56356    3
37363    3
92628    3
16401    3
        ..
94771    1
9011     1
84338    1
21039    1
33301    1
Name: cityCode, Length: 9509, dtype: int64
```

This dataset has 10000 rows and 18 columns (including the target)

The main categorical values are "Basic" and "Luxury"

It looks a bit imbalanced from value count

The describe method helps to understand descrptive statistics. This is a look at the data as a whole and not broken down by category yet.

In [6]:
```
#Describe the dataset
paris_df.describe()
```

Out[6]:

	squareMeters	numberOfRooms	hasYard	hasPool	floors	city
count	10000.00000	10000.000000	10000.000000	10000.000000	10000.000000	10000.00
mean	49870.13120	50.358400	0.508700	0.496800	50.276300	50225.48
std	28774.37535	28.816696	0.499949	0.500015	28.889171	29006.67
min	89.00000	1.000000	0.000000	0.000000	1.000000	3.00
25%	25098.50000	25.000000	0.000000	0.000000	25.000000	24693.75
50%	50105.50000	50.000000	1.000000	0.000000	50.000000	50693.00
75%	74609.75000	75.000000	1.000000	1.000000	76.000000	75683.25
max	99999.00000	100.000000	1.000000	1.000000	100.000000	99953.00

- Based on the features decribing size , rooms and number of floors. Its is pretty clear that most of the properties are large buildings. There are a few outliers that may describe an apartment or room.
- These buildings are throughout the city based on the city code.
- About 50% of the buildings have a yard and pool. Having a yard for a building is a great ammenity.
- These buildings have an average of over 5 owners.
- For an old city the buildings are less than 35 years old with some as recents 2021.
- The average price of these buildings is ~5 Million Units, 10 Mill max. It is unclear if the price is in US Dollars or French Francs. Even if US Dollars 5 million USD may be cheap for large buildins.

Checking for Null values & Datatypes

In [7]:
```
#Information on dataset
paris_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 18 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   squareMeters       10000 non-null  int64
 1   numberOfRooms      10000 non-null  int64
 2   hasYard            10000 non-null  int64
 3   hasPool            10000 non-null  int64
 4   floors             10000 non-null  int64
 5   cityCode           10000 non-null  int64
 6   cityPartRange      10000 non-null  int64
 7   numPrevOwners      10000 non-null  int64
 8   made               10000 non-null  int64
```

```
 9   isNewBuilt          10000 non-null  int64
 10  hasStormProtector   10000 non-null  int64
 11  basement            10000 non-null  int64
 12  attic               10000 non-null  int64
 13  garage              10000 non-null  int64
 14  hasStorageRoom      10000 non-null  int64
 15  hasGuestRoom        10000 non-null  int64
 16  price               10000 non-null  float64
 17  category            10000 non-null  object
dtypes: float64(1), int64(16), object(1)
memory usage: 1.4+ MB
```

There are no textual values except for the target variable although some are categorical even though they are numerical. No missing rows.

Dropping column that doesn't add value to the current dataset. The "isNewBuilt" column tells similar information to the "made" column which describes year built.

In [8]:
```
#drop columns that don't seem to add value
paris_df = paris_df.drop(columns="isNewBuilt",axis =1)
```

In [9]:
```
#checking shape after deletion
paris_df.shape
```

Out[9]:
```
(10000, 17)
```

Checking for Null and Duplicate values. We would want to remove or impute certain values for Null, depending on where or what they are. We would want to remove any duplicate values.

In [10]:
```
#Any null values? - No NULL Values
paris_df.isnull().sum()
#No Null values
```

Out[10]:
```
squareMeters          0
numberOfRooms         0
hasYard               0
hasPool               0
floors                0
cityCode              0
cityPartRange         0
numPrevOwners         0
made                  0
hasStormProtector     0
basement              0
attic                 0
garage                0
hasStorageRoom        0
hasGuestRoom          0
price                 0
category              0
dtype: int64
```

In [11]:
```
#Checking for duplicates - None
paris_df.duplicated().sum()
```

Out[11]:
```
0
```

Renaming Colums to make understanding feature names more intuitive and for better

formatting

In [12]:
```
#Renaming columns for better clarity
old_cols = []
new_col = ['Building_Size(m)','Num_of_Rooms','Has_Yard','Has_Pool','Num_of_Floor
           'Num_Previous_Owners', 'Year_Built','Has_Storm_Protector',
           'Basement_Size(m)','Attic_Size(m)', 'Garage_Size(m)', 'Has_Storage_Ro
col_dict = {}
for ind, col in enumerate(paris_df.columns):
    col_dict[col] = new_col[ind]
print(col_dict)
paris_df.rename(columns = col_dict, inplace= True)
```

```
{'squareMeters': 'Building_Size(m)', 'numberOfRooms': 'Num_of_Rooms', 'hasYard':
'Has_Yard', 'hasPool': 'Has_Pool', 'floors': 'Num_of_Floors', 'cityCode': 'City_
Code', 'cityPartRange': 'City_Part_Range', 'numPrevOwners': 'Num_Previous_Owner
s', 'made': 'Year_Built', 'hasStormProtector': 'Has_Storm_Protector', 'basemen
t': 'Basement_Size(m)', 'attic': 'Attic_Size(m)', 'garage': 'Garage_Size(m)', 'h
asStorageRoom': 'Has_Storage_Room', 'hasGuestRoom': 'Num_of_Guest_Rooms', 'pric
e': 'Price', 'category': 'Category'}
```

In [13]:
```
paris_df.columns
```

Out[13]:
```
Index(['Building_Size(m)', 'Num_of_Rooms', 'Has_Yard', 'Has_Pool',
       'Num_of_Floors', 'City_Code', 'City_Part_Range', 'Num_Previous_Owners',
       'Year_Built', 'Has_Storm_Protector', 'Basement_Size(m)',
       'Attic_Size(m)', 'Garage_Size(m)', 'Has_Storage_Room',
       'Num_of_Guest_Rooms', 'Price', 'Category'],
      dtype='object')
```

All features except for the category feature is numerical. There are no null values and textual placeholders. The describe outcome showed a few outliers but nothing that resembles placeholders.

- Every Luxury property Has a pool and Has a Yard. This will most likely be most significant features.

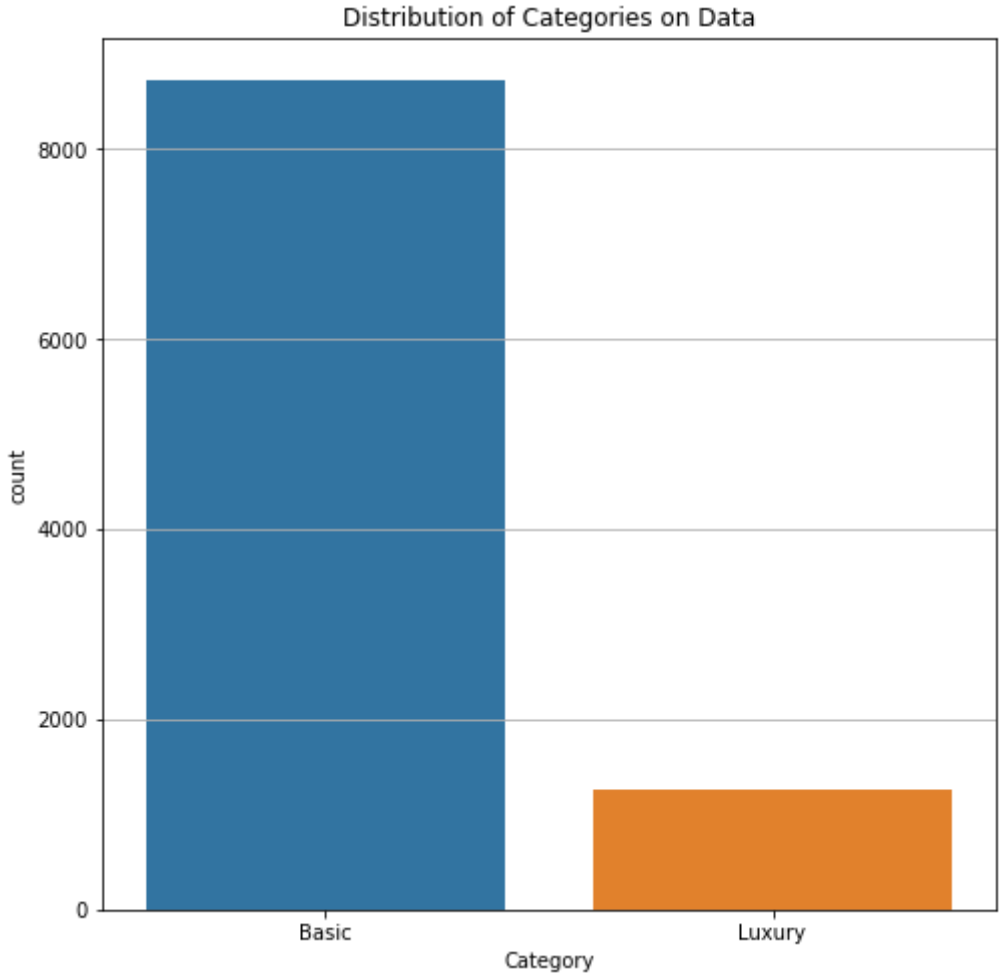
- The average price may not be a determining factor in if a property is Luxury. This is usually a big factor that takes into account the features.

- Other than the "Pool" and "Yard" feature, applying the mean on this data by category doesn't give explicit incite into what features will be significant in predicting the correct category.

Exploring Data Features

Our intial exploration is visually seeing the count distribution of the "Category" column

In [14]:
```
#Distribution of Category Variables
fig,ax=plt.subplots(figsize=(8,8), facecolor = "white")
sns.countplot(x = "Category", data=paris_df)
plt.title('Distribution of Categories on Data')
plt.grid(axis = "y")
plt.show()
fig.savefig("images/dist.png", dpi=150)
```



There is a significant amount of "Basic" instances vs. "Luxury"

Changing the values of "Basic" and "Luxury" to 0 and 1 respectively. This will allow the values to be used in some visualizations and plots.

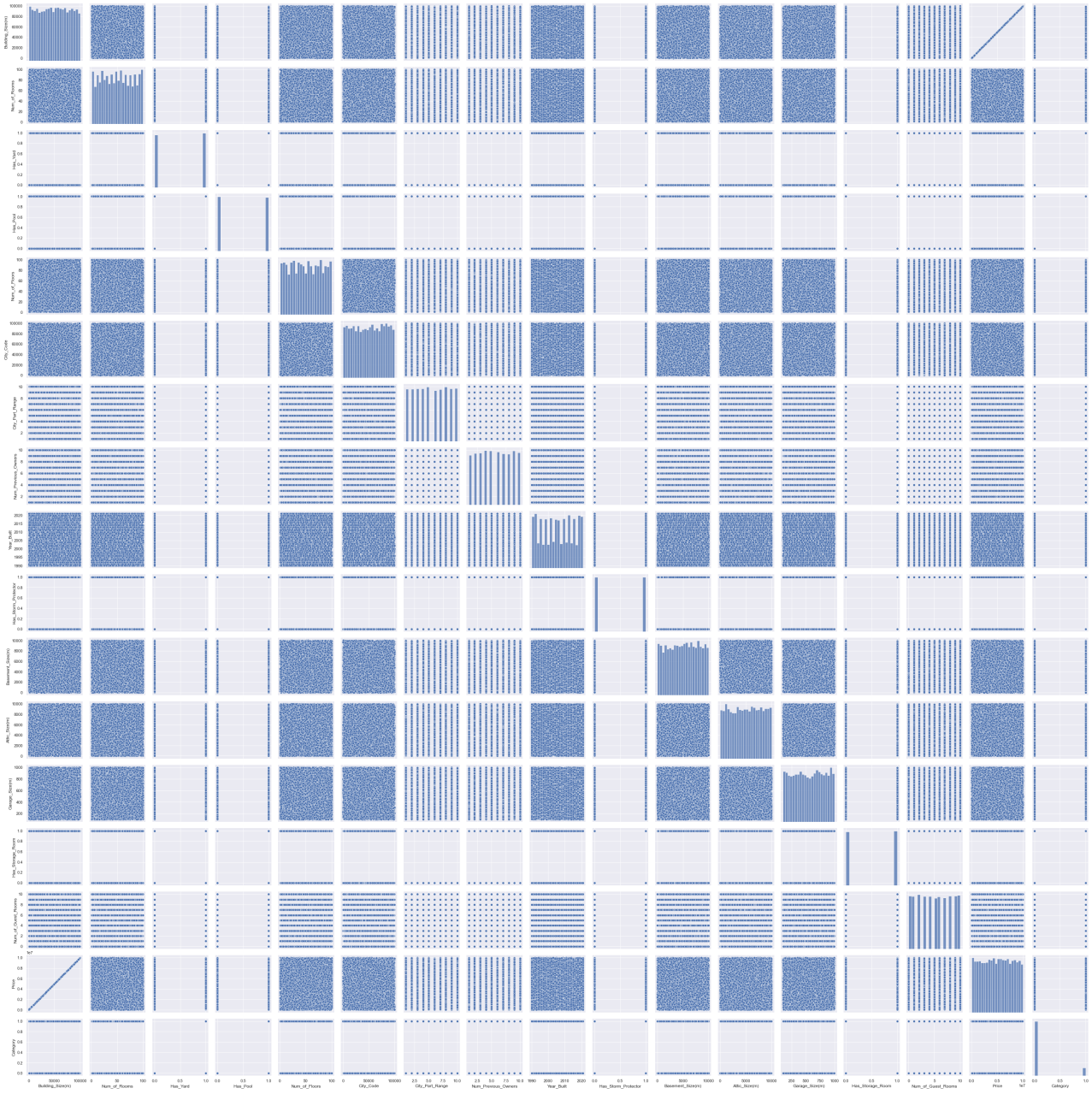
In [15]:
```
pdf = paris_df.copy()
pdf["Category"] = paris_df["Category"].map({"Basic":0, "Luxury":1})
pdf["Category"].value_counts()
```

Out[15]:
```
0    8735
1    1265
Name: Category, dtype: int64
```

Using a pairplot to understand the the entire dataset's distrubition and linearity.

In [16]:
```
#Pair plot for quick view of the datasets distribution
sns.set(rc={'figure.figsize':(20,15)})
sns.pairplot(pdf)
```

Out[16]:
```
<seaborn.axisgrid.PairGrid at 0x7fc4c9d53250>
```



With so many features, it isn't easy to understand at a specific level all of the features. From this highlevel perspective, most are evenly distributed patterns. It does look like property in the dataset fluctuated heavility up and down with the year built. It also looks like the only linear relationship is the "Price" and "Building_Size(m)"

Correlation and Heat Map

Understanding the correlation of the features to each other. This supports multicollinearity understanding and feature importance.

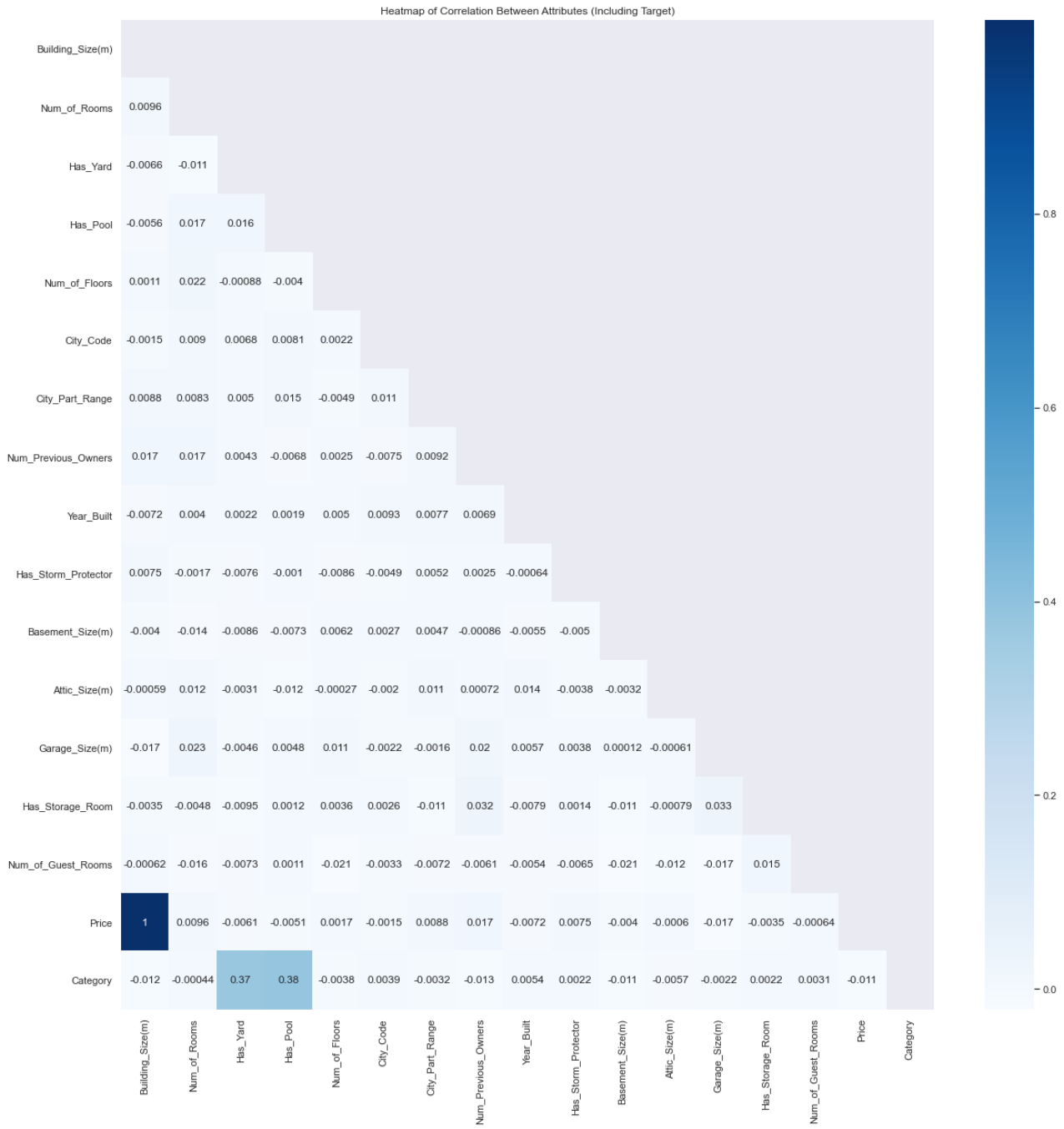
In [17]:
```
#Print out correlation values in dataframe
corr = pdf.corr()
corr
```

Out[17]:

	Building_Size(m)	Num_of_Rooms	Has_Yard	Has_Pool	Num_of_Floors
Building_Size(m)	1.000000	0.009573	-0.006650	-0.005594	0.001109

	Building_Size(m)	Num_of_Rooms	Has_Yard	Has_Pool	Num_of_Floors
Num_of_Rooms	0.009573	1.000000	-0.011240	0.017015	0.022244
Has_Yard	-0.006650	-0.011240	1.000000	0.015514	-0.000883
Has_Pool	-0.005594	0.017015	0.015514	1.000000	-0.004006
Num_of_Floors	0.001109	0.022244	-0.000883	-0.004006	1.000000
City_Code	-0.001541	0.009040	0.006760	0.008072	0.002207
City_Part_Range	0.008758	0.008340	0.005023	0.014613	-0.004921
Num_Previous_Owners	0.016619	0.016766	0.004279	-0.006848	0.002463
Year_Built	-0.007207	0.003978	0.002214	0.001894	0.005022
Has_Storm_Protector	0.007480	-0.001656	-0.007598	-0.001001	-0.008566
Basement_Size(m)	-0.003960	-0.013990	-0.008558	-0.007268	0.006228
Attic_Size(m)	-0.000588	0.012061	-0.003085	-0.011901	-0.000270
Garage_Size(m)	-0.017246	0.023188	-0.004626	0.004832	0.011303
Has_Storage_Room	-0.003486	-0.004760	-0.009506	0.001238	0.003616
Num_of_Guest_Rooms	-0.000623	-0.015529	-0.007276	0.001123	-0.021155
Price	0.999999	0.009591	-0.006119	-0.005070	0.001654
Category	-0.011800	-0.000442	0.373987	0.382995	-0.003827

In [18]:
```
# Set up figure and axes
fig, ax = plt.subplots(figsize=(20,20),facecolor = "white")

# Plot a heatmap of the correlation matrix, with both
# numbers and colors indicating the correlations
sns.heatmap(
    # Specifies the data to be plotted
    data=corr,
    # The mask means we only show half the values,
    # instead of showing duplicates. It's optional.
    mask=np.triu(np.ones_like(corr, dtype=bool)),
    # Specifies that we should use the existing axes
    ax=ax,
    #Color of the heatmap
    cmap="Blues",
    # Specifies that we want labels, not just colors
    annot=True,

)

# Customize the plot appearance
ax.set_title("Heatmap of Correlation Between Attributes (Including Target)");
```

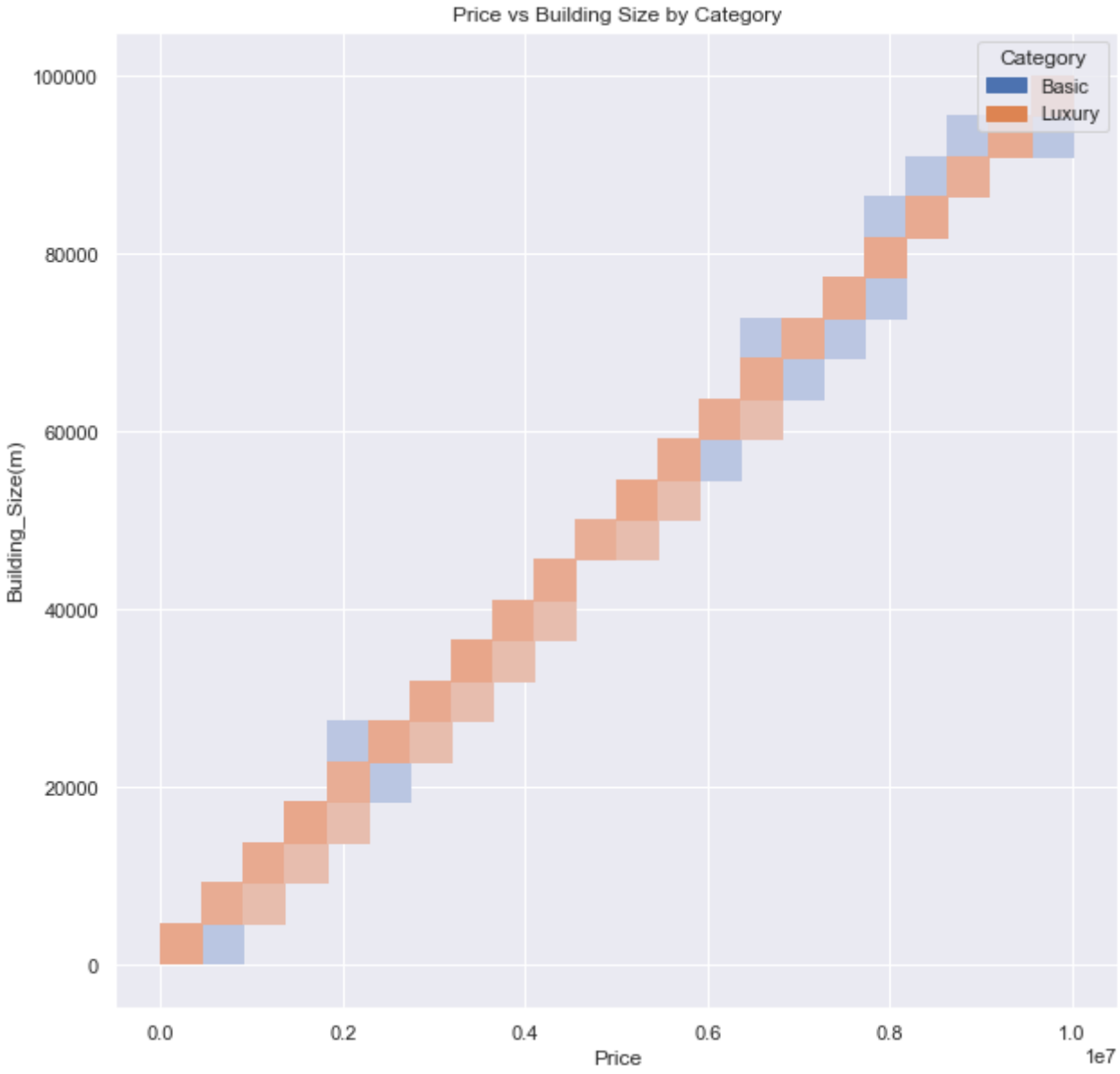


- **Currently "Price" is 100% correlated to the "Building_Size(m)". Because we are a doing classification problem, multicollinearity doesn't have a great impact.**
- **Having a yard and pool have the highest correlation to the target, "Category"**
- **The other values have less than .005%**

Displaying other Visuals to Support Data Understanding

Plotting the Price Vs Building Size to understand where our target values are. "Basic" is blue "Luxury" is orange, and overlap is light orange.

In [19]:
```
#Histogram plot with
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
plot = sns.histplot(y = "Building_Size(m)", x ="Price" , hue = "Category", multi
                    data = paris_df, bins= "auto").set(title= "Price vs Building
```



The overlap of the "Basic" and "Luxury" categories is the light orange.

- **Both "Basic" and "Luxury"have prices and building size linearly distributed across the min and max values of the data.**
- **At the higher prices and building size, the "Basic" property has more variation than the "Luxury.**
- **I would have thought the "Basic" property would not cost as much, but that is not true**
- **I would have also thought the "Basic" property would be smaller in size**

Checking the count distribution of Year_Built between category values.

In [20]:
```
pd.crosstab(paris_df["Category"],paris_df["Year_Built"])
```

Out[20]:

Year_Built	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	...	2012	2013	201
Category														
Basic	272	282	327	267	268	248	280	260	282	254	...	270	314	27
Luxury	45	42	29	53	44	37	47	36	36	39	...	35	38	3

2 rows × 32 columns

The distribution amount of values are evenly distributed between each year respective category values.

Viewing the distribution of Having a yard vs Price.

In [21]:
```
#Cross tab of Category Values and Has_Yard Feature
pd.crosstab(paris_df["Category"],paris_df["Has_Yard"])
```

Out[21]:

Has_Yard	0	1
Category		
Basic	4913	3822
Luxury	0	1265

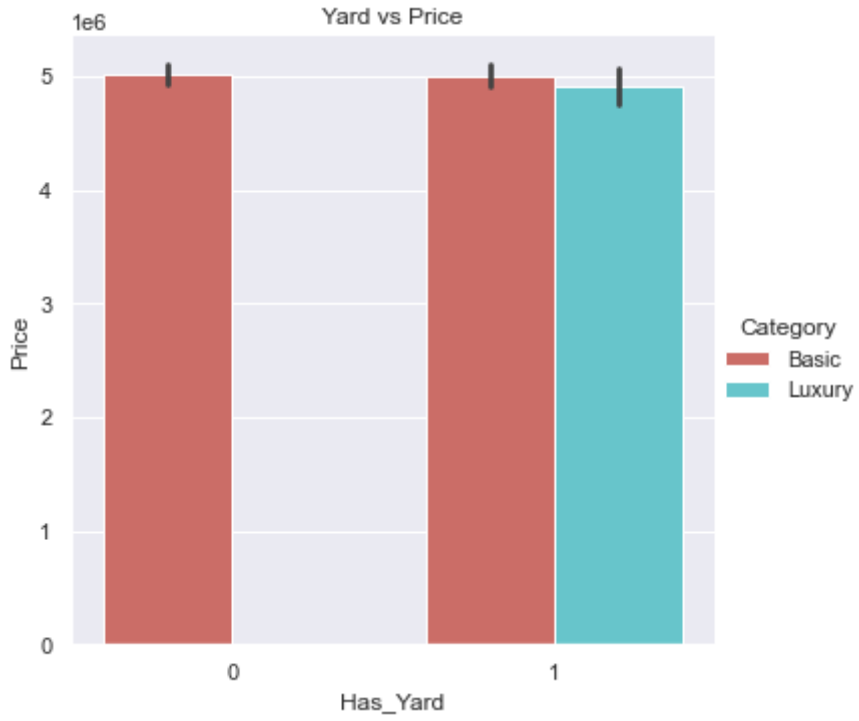
In [22]:
```
#View of the Has_Yard Vs Price
paris_df.groupby(["Has_Yard", 'Category'])['Price'].mean().unstack().fillna(0)
```

Out[22]:

Category	Basic	Luxury
Has_Yard		
0	5.011363e+06	0.000000e+00
1	4.998944e+06	4.907260e+06

In [23]:
```
# Create barplot of Has_Yard Vs Price
sns.factorplot(x="Has_Yard", y='Price', data=paris_df, hue = "Category", kind =
               palette= "hls").set(title= "Yard vs Price")
```

Out[23]: <seaborn.axisgrid.FacetGrid at 0x7fc4c8028e80>



All "Luxury" properties have a yard 100% of the time.
The "Basic" properties have a yard 44% of the time.
All "Basic" property cost an average of ~5 Million with and without a yard. The "Basic" property w/o a yard is slightly more.
The "Basic" property average cost of a property with a yard is more than a "Luxury" property's average.

Viewing the distribution of Having a pool vs Price.

In [24]:
```
#Cross tab of Category Values and Has_Pool Feature
pd.crosstab(paris_df["Category"],paris_df["Has_Pool"])
```

Out[24]:

Has_Pool	0	1
Category		
Basic	5032	3703
Luxury	0	1265

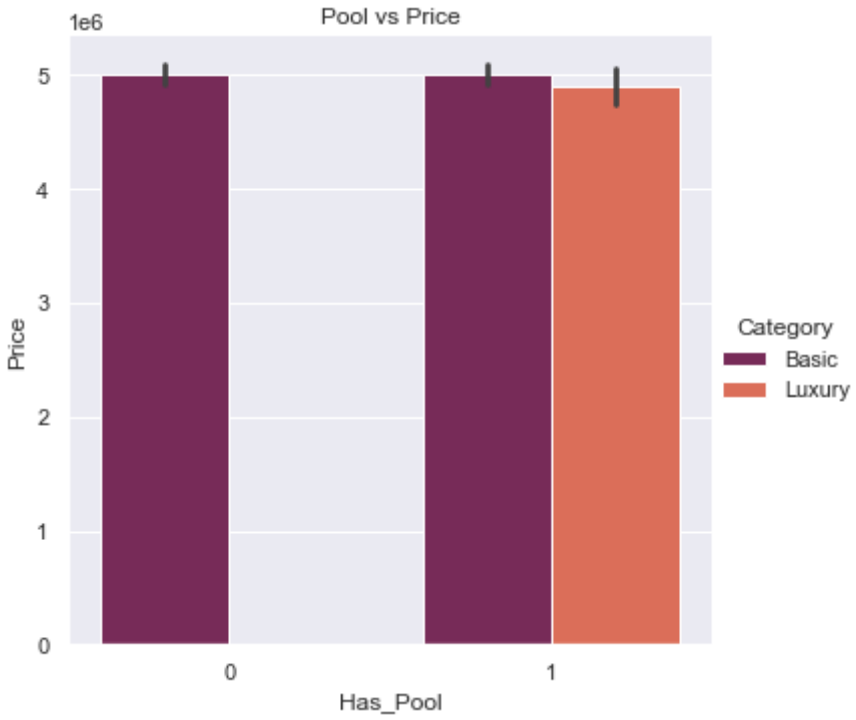
In [25]:
```
#View of the Has_Yard Vs Price
paris_df.groupby(["Has_Pool", 'Category'])['Price'].mean().unstack().fillna(0)
```

Out[25]:

Category	Basic	Luxury
Has_Pool		
0	5.007943e+06	0.000000e+00
1	5.003192e+06	4.907260e+06

In [26]:
```
# Create barplot of Has Pool vs Price
sns.factorplot(x="Has_Pool", y='Price', data=paris_df, hue = "Category",kind = "
               palette= "rocket").set(title = "Pool vs Price")
```

Out[26]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc50c353bb0>
```



All "Luxury" properties have a pool 100% of the time.
The "Basic" properties have a pool 42% of the time.
All "Basic" property cost an average of ~5 Million with and without a pool.
The "Basic" property average cost of a property with a pool is more than the "Luxury" property's average.

Viewing the distribution of Having a Storm Protector

In [27]:
```
#Cross tab of Distribution of Has Storm Protector
pd.crosstab(paris_df["Category"],paris_df["Has_Storm_Protector"])
```

Out[27]:

Has_Storm_Protector	0	1
Category		
Basic	4372	4363
Luxury	629	636

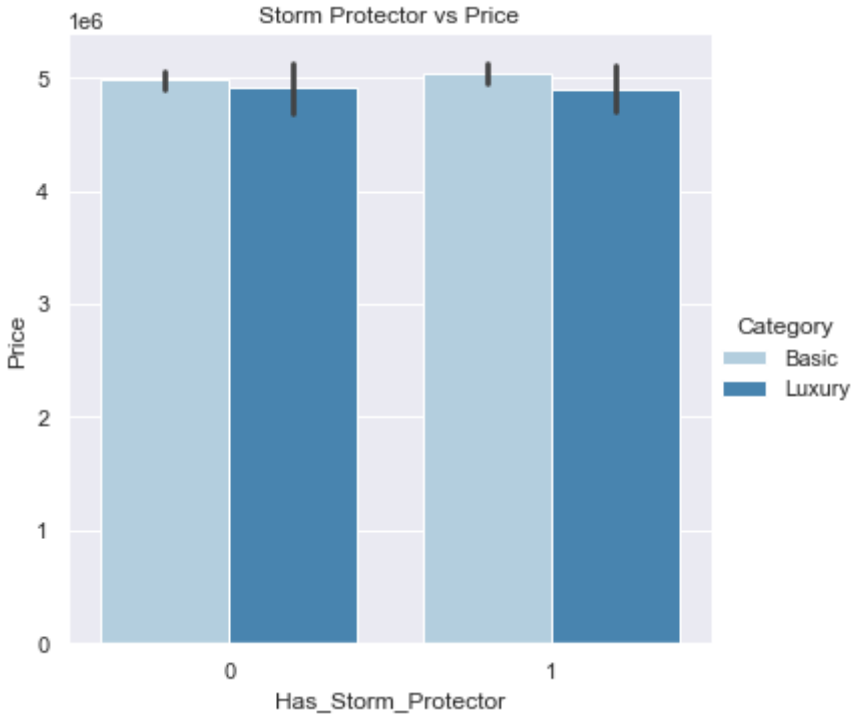
In [28]:
```
#View of the Has_Yard Vs Price
paris_df.groupby(["Has_Storm_Protector", 'Category'])['Price'].mean().unstack().
```

Out[28]:

Category	Basic	Luxury
Has_Storm_Protector		
0	4.979922e+06	4.916014e+06
1	5.031990e+06	4.898603e+06

In [29]:
```
# Create barplot of Storm Protector
sns.factorplot(x="Has_Storm_Protector", y='Price', data=paris_df, hue = "Categor
               palette= "Blues").set(title = "Storm Protector vs Price", )
```

Out[29]: <seaborn.axisgrid.FacetGrid at 0x7fc4aa086790>



All "Luxury" properties have a storm protector ~50% of the time. "Luxury" property w/ a storm protector is slightly more.
The "Basic" properties have a storm protector ~50% of the time.
All "Basic" property cost an average of ~5 Million with and without a storm protector.
All "Luxury" property cost an average of ~4.9 Million with and without a storm protector.
The "Basic" property average cost of a property with a pool is more than the "Luxury" property's average.

Kaggle EDA Ref,ABO-ELKHAIR 2022

Modeling

Modeling will be done interatively. As new information is learned new models, parameters and transformation techniques will be applied. Because of a data imbalance and goal for the stakeholder our main metrics will be : Precision and F1 respectively. Recall and Accuracy will be calculated for understanding but not as a decision driver.

- **1. Dummy Classifier for intial "Random" prediction comparison**
- **1. Simple Modeling (Default Parameters): Logistic Regression, Decision Tree and Random Forerest**
- **1. Apply Transformations: SMOTE and Standard Scaling Logistic Regression, Decision Tree and Random Forerest**
- **1. Gridsearch w/ Transformation: Logistic Regression, Decision Tree and Random Forerest**
- **1. Choose Best Estimator**

Perform a Train and Test split of the data to prepare for modeling.

In [30]:
```
#Train Test Split of the Data for Modeling
X = paris_df.drop("Category", axis = 1)
y = paris_df["Category"]
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state = 42, stra
```

In [31]:
```
#Checking to see if the value counts are stratified for imbalance
print(y_test.value_counts())
print(y_train.value_counts())
```

```
Basic     2184
Luxury     316
Name: Category, dtype: int64
Basic     6551
Luxury     949
Name: Category, dtype: int64
```

The train and test split have a similar categorical distribution

Model 1 - Dummy Classifier Model

Dummy Classifier- Abiheet Sahoo, 2020

The initial model will be a dummy model. This is a classifier predicting based on"most frequent" values and not specific patterns in the data. Therefore this should show an accuracy rate similar to someone manually/randomly choosing a classification.

In [32]:
```
#Instantiating and fitting the model
dummy_paris_df = DummyClassifier(strategy="most_frequent") # Using Most Frequent
dummy_paris_df.fit(X_train, y_train)
```

Out[32]:
```
DummyClassifier(strategy='most_frequent')
```

In [33]:
```
#Cross Val results on training data
cv_results = cross_val_score(dummy_paris_df, X_train, y_train, cv=5)
cv_results.mean()
```

Out[33]:
```
0.8734666666666666
```

In [34]:
```
cv_results = cross_val_score(dummy_paris_df, X_test, y_test, cv=5)
cv_results.mean()
```

Out[34]:
```
0.8736
```

In [35]:
```
#Getting the  predicted y scores for train and test
dummy_y_train = dummy_paris_df.predict(X_train)
dummy_y_test = dummy_paris_df.predict(X_test)
```

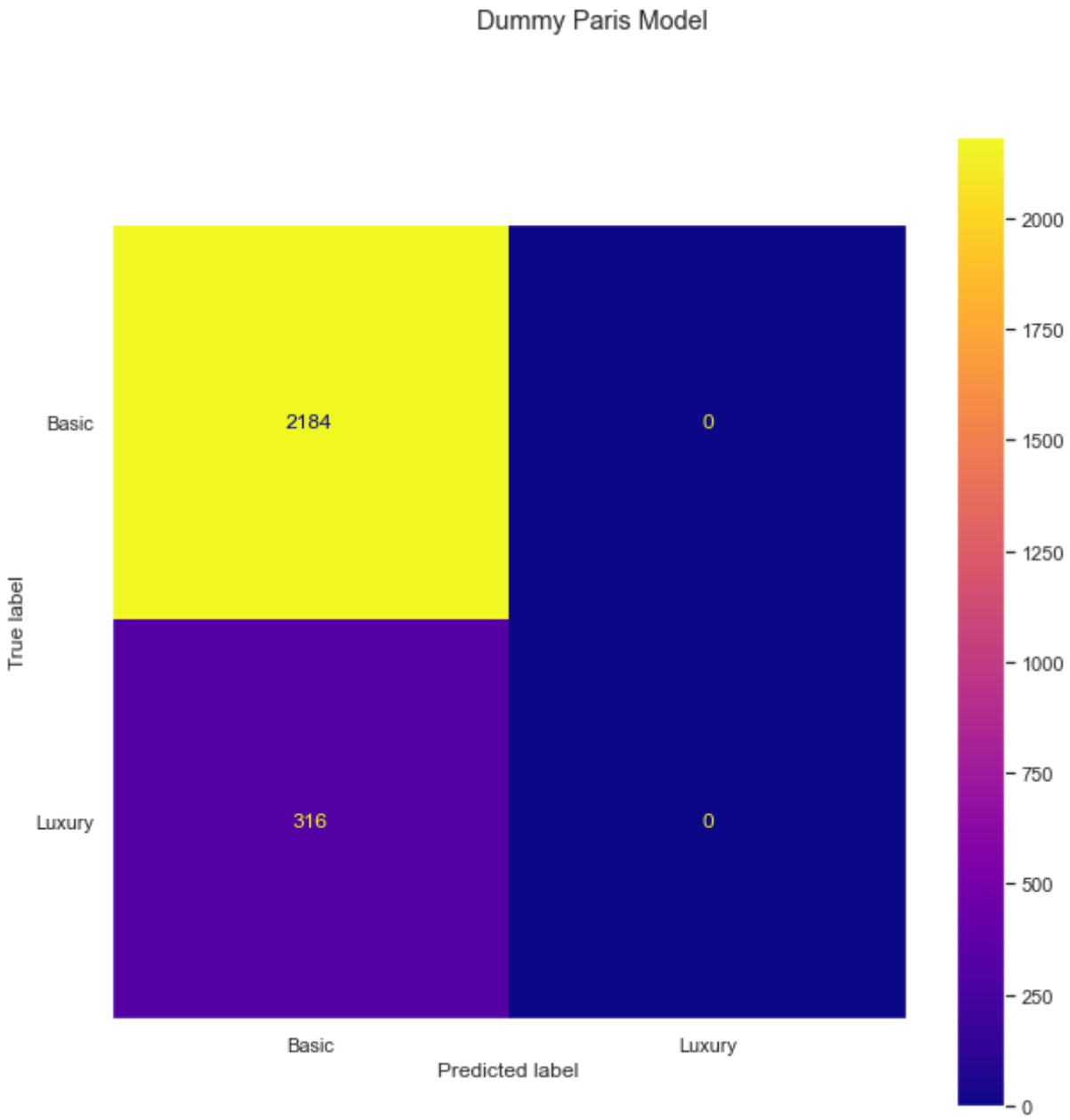
The training and testing cv scores (accuracy) for choosing "Basic" are both ~ 873%. The model won't be used, but it is good fit based on training and test data.

In [36]:
```
#Score of guessing Luxury
print("Percent of guessing Luxury Accuracy= ", round(1-cv_results.mean(),2), "%"
```

```
Percent of guessing Luxury Accuracy=  0.13 %
```

The data is imbalanced, therefore accuracy will not be the best metric. It will always learn in favor of the most frequent value. The cross matrix below shows the True predictions (True Negative, True Posiive) as well as those with Type I (False Negative) and Type II error (False Positive).

In [37]:
```
#Confusion Matrix using Test Data
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Dummy Paris Model")
plot_confusion_matrix(dummy_paris_df, X_test, y_test, ax=ax, cmap="plasma", )
plt.grid(False)
```



With such a strong data imbalance, "Luxury" was never predicted at all, even no FP guesses. The model predicted all "Basic" property. We want to maximize correctly predicting "Luxury" and minimize incorrectly predictuing "Luxury".

In [38]:
```
#Printing out Test data classification report with scores
print("Train Classificaiton Report:\n", classification_report(y_train, dummy_y_t
```

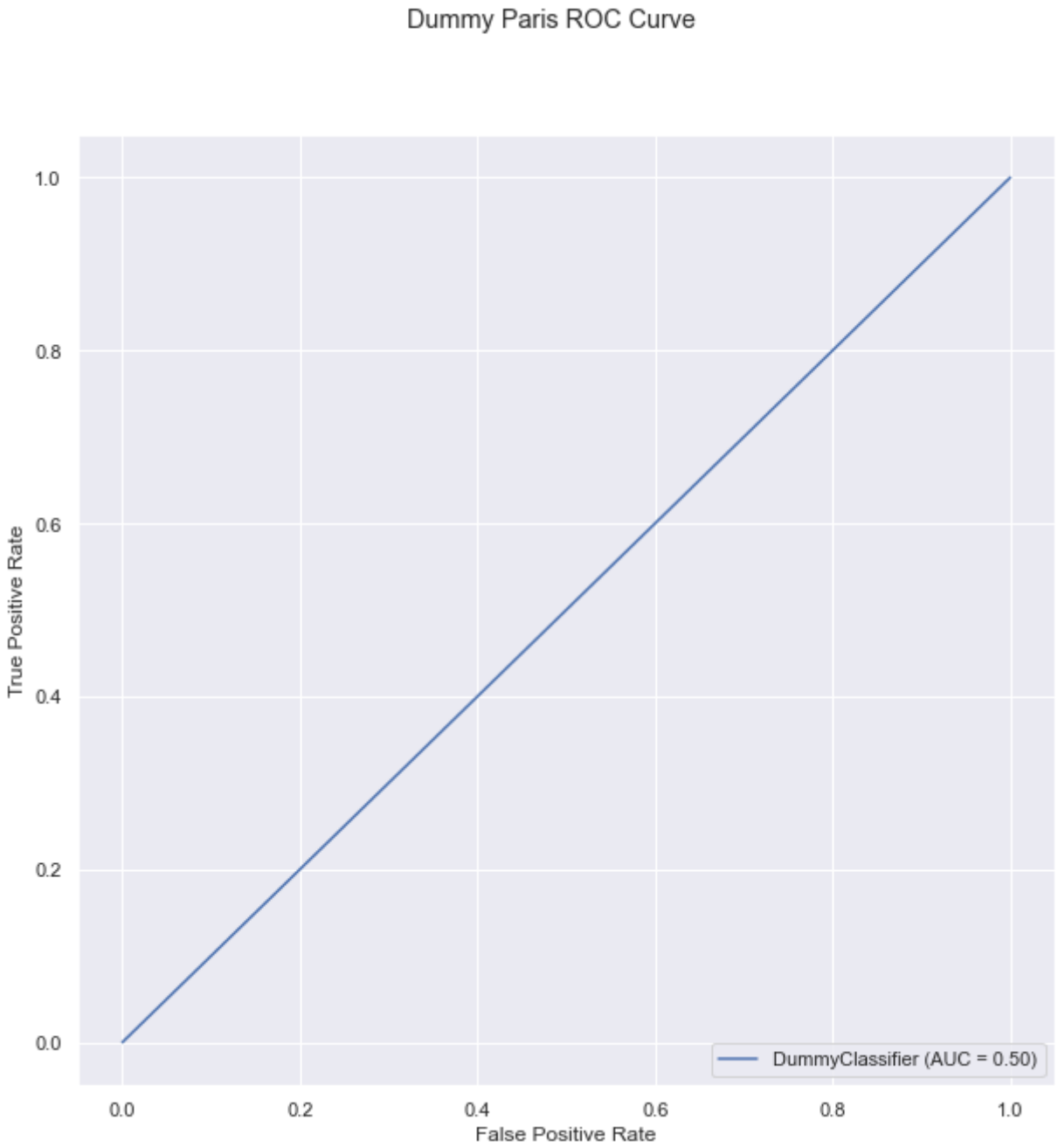
```
print("Test Classificaiton Report:\n", classification_report(y_test, dummy_y_tes
```

```
Train Classificaiton Report:
              precision    recall  f1-score   support

       Basic       0.87      1.00      0.93      6551
      Luxury       0.00      0.00      0.00       949

    accuracy                           0.87      7500
   macro avg       0.44      0.50      0.47      7500
weighted avg       0.76      0.87      0.81      7500

Test Classificaiton Report:
              precision    recall  f1-score   support

       Basic       0.87      1.00      0.93      2184
      Luxury       0.00      0.00      0.00       316

    accuracy                           0.87      2500
   macro avg       0.44      0.50      0.47      2500
weighted avg       0.76      0.87      0.81      2500
```

Plotting ROC Curve: Dummy Classifier

In [39]:
```
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Dummy Paris ROC Curve")
plot_roc_curve(dummy_paris_df, X_test, y_test, ax=ax)
#Increase the font size and labels
```

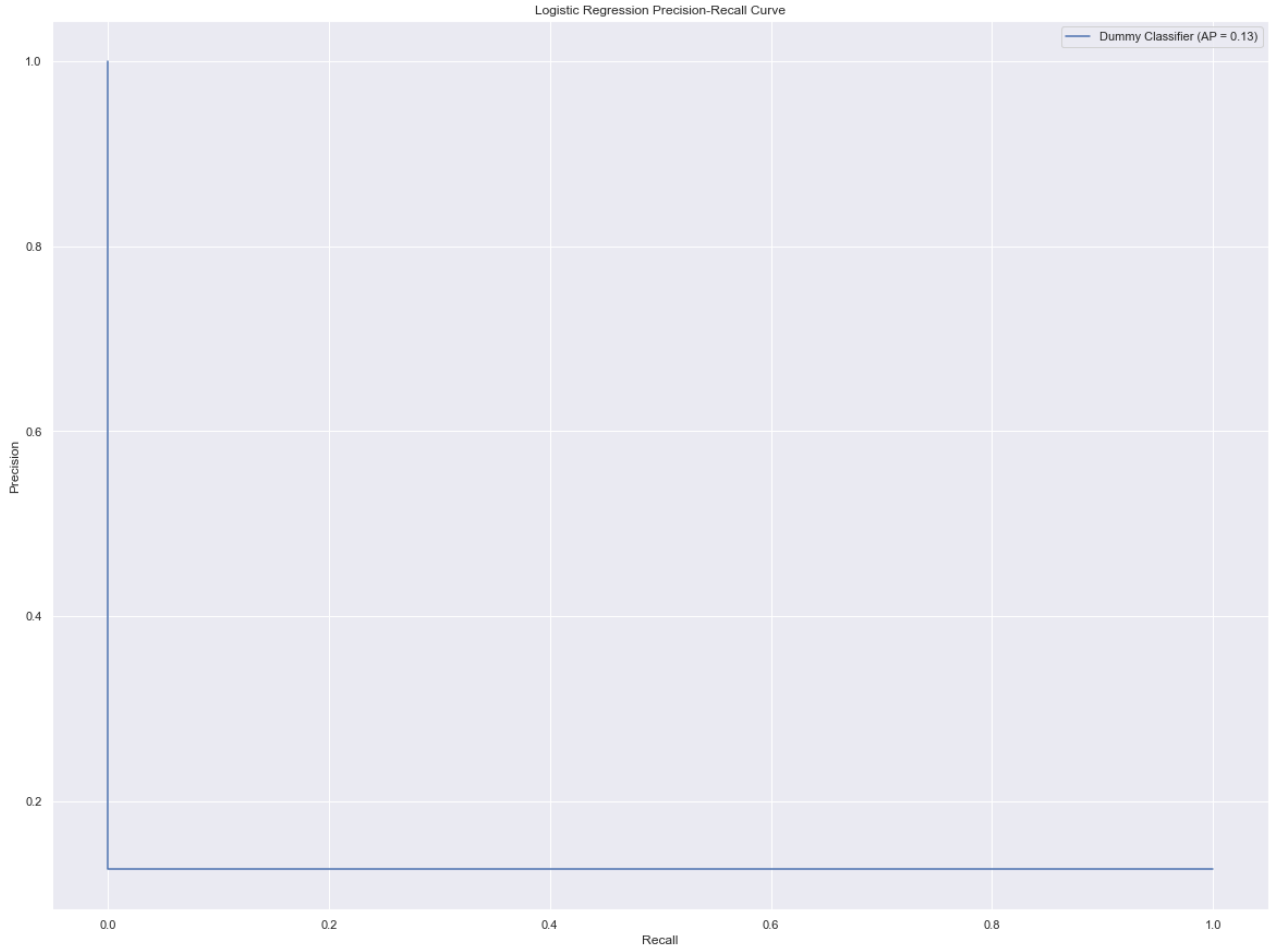
Out[39]:
```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fc4d82a90a0>
```



Precision Recall Curve

In [40]:
```
plot_precision_recall_curve(dummy_paris_df, X_test, y_test, name = "Dummy Classi
#axis labels
plt.title("Logistic Regression Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
#Legend
plt.legend()
```

Out[40]:
```
<matplotlib.legend.Legend at 0x7fc4ba52f7c0>
```



The AUC looks to be .5 with the test set. ROC-AUC curve will not be the best for this imblanced dataset. It doesn't adequately explain the nuances of the dataset.

The AUC-PR/Average Precision for the Precision Recall is a better model for the imbalanced data. This counts for only .13. This is aligned to the accuracy for randomly guessing "Luxury calculation above.

For future iterations the main metrics will be Precision and F1 . For future plots a Precision-Recall Curve will be sued.

Chosen Metrics:

For future iterations Precision and F1 will be metrics in focus.
Precision: The is to correctly identity "Luxury" and minimize incorrectly identifying "Basic". This will help understand this.
F1: The dataset is imbalanced and accuracy doesn't take that into account. The F1 score balances out the Precision and Recall score to give a harmonic mean, supporting the above goal.

Model 2.1 Simple Modeling

For these next stage of modeling, three classificaiton models will be chosen to used on the current dataset. Logistic Regression, Decision Tree, and Random Forest will be chosen as more sophisticated models compared to the Dummy classifier. The Random Forest classifier is more advanced than the decision tree, it is made up of decision trees. For these models, the default parameters will be used and scored. These will establish a more credible set of baseline models.

Each model will scored on training and test data. The primary metrics to view and evaluate models by are: Precision and f1. Recall will be calculated too, but secondary for evaluation and used support the f1 score.

Each model will have:

- **Cross Validation Score (Average Accuracy)**
- **Precision***
- **Recall**
- **F1***
- **Classification Report**
- **Cross Matrix**
- **Precision Recall (Curve Precision Average Score)**

***Primary Evaluation Metrics in Bold**

Model 2.1 - Logistic Regression Model (Simple Model)

Logistic regression uses a sigmoid function which helps to plot an "s"-like curve that enables a linear function to act as a binary classifier. This works well for our binary classification problem.

Creating functions to implement the repetive code throughout the modeling

In [41]:
```
#Function to predict from fitted model
def simple_model_predict(fit_model,X):

    y_pred = fit_model.predict(X)
    y_pred_prob = fit_model.predict_proba(X)
    return y_pred, y_pred_prob
```

In [42]:
```
#Calculating precision, recall and f1
def precision_recall(y_true,y_pred,pos_label = "Luxury"):

    precision = precision_score(y_true, y_pred, pos_label=pos_label)
    recall = np.mean(recall_score(y_true, y_pred, pos_label=pos_label))
    f1 = f1_score(y_true, y_pred, pos_label=pos_label)

    print ("The Precision score is: {}".format(precision))
    print ("The Recall score is: {}".format(recall))
    print ("The F1 score is: {}".format(f1))
    print("\n")
    return precision, recall, f1
```

In [43]:
```
def cv_score(model, X, y):
    scores = cross_val_score(model, X, y, cv=5 )
    return  print("Accuracy of Model:",round(scores.mean(),3))
```

Instatiating, fitting and predicting the logistic regression model. The predicting is done on the trained and testing data.

In [44]:
```
#Instantiate
lr_simple = LogisticRegression(random_state = 42,penalty= "none")
#Fit training data
lr_simple.fit(X_train,y_train)
#Predict with Train
lr_y_train_pred,lr_y_train_pred_prob  = simple_model_predict(lr_simple,X_train)
#Predict with Train
lr_y_test_pred,lr_y_test_pred_prob  = simple_model_predict(lr_simple,X_test)
```

Model 2.1 - Logistic Regression Model (Simple Model) Evaluation

Training Data Evaluation

In [45]:
```
### Train
cv_score(lr_simple, X_train, y_train)
```

```
Accuracy of Model: 0.873
```

In [46]:
```
#Precision, Recall and F1
#Train Score
print("Luxury Train Scores: \n")
precision_recall(y_train, lr_y_train_pred,"Luxury")
```

```
Luxury Train Scores:

The Precision score is: 0.0
The Recall score is: 0.0
The F1 score is: 0.0
```

Out[46]:
```
(0.0, 0.0, 0.0)
```

In [47]:
```
#Printing out report with scores for Train data
print(classification_report(y_train, lr_y_train_pred))
```

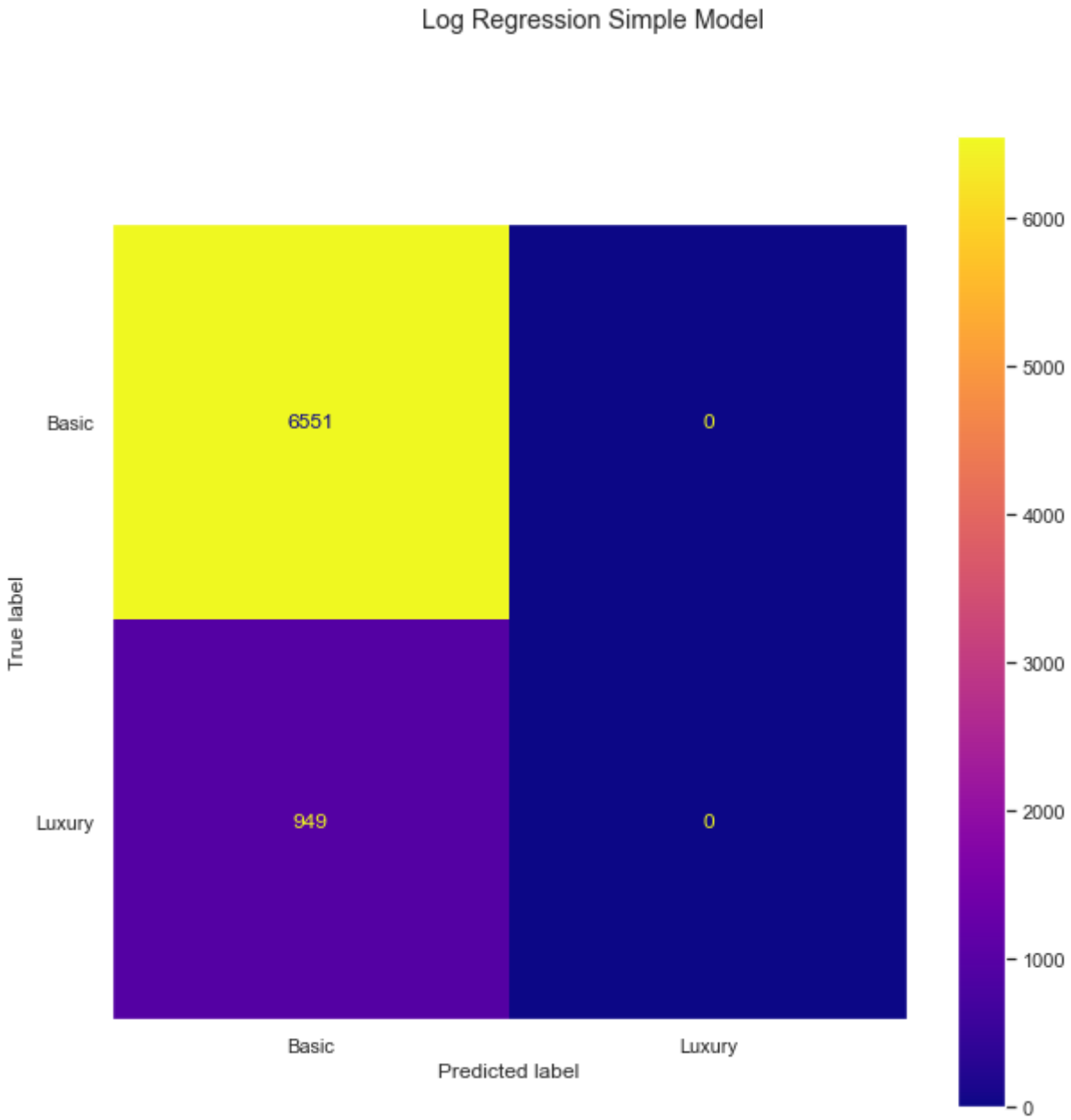
```
              precision    recall  f1-score   support

       Basic       0.87      1.00      0.93      6551
      Luxury       0.00      0.00      0.00       949

    accuracy                           0.87      7500
   macro avg       0.44      0.50      0.47      7500
weighted avg       0.76      0.87      0.81      7500
```

The precision, recall and f1-score of of the training data for Luxury value are all 0. This is no better than the Dummy classifier metrics.

In [48]:
```
#Print out array confusion matrix
print(confusion_matrix(y_train, lr_y_train_pred, labels = ["Basic","Luxury"]))
```

```
[[6551    0]
 [ 949    0]]
```

In [49]:
```
#Plotting Confusion Matrix
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Log Regression Simple Model")
plot_confusion_matrix(lr_simple, X_train, y_train, ax=ax, cmap="plasma")
plt.grid(False)
```



Both this training set and the Dummy classifier test set did not correctly predict any "Luxury" values. It is not a fair comparison because it is training and test data, but they still are insufficient.

Test Data Evaluation

In [50]:
```
#CV Accuracy
cv_score(lr_simple, X_test, y_test)
```

```
Accuracy of Model: 0.874
```

In [51]:
```
#Precision, Recall and F1
#Test Score
print("Luxury Test Scores: \n")
precision_recall(y_test, lr_y_test_pred,"Luxury")
```

```
Luxury Test Scores:

The Precision score is: 0.0
The Recall score is: 0.0
The F1 score is: 0.0
```

Out[51]:
```
(0.0, 0.0, 0.0)
```

In [52]:
```
#Printing out report with scores -
print(classification_report(y_test, lr_y_test_pred))
```

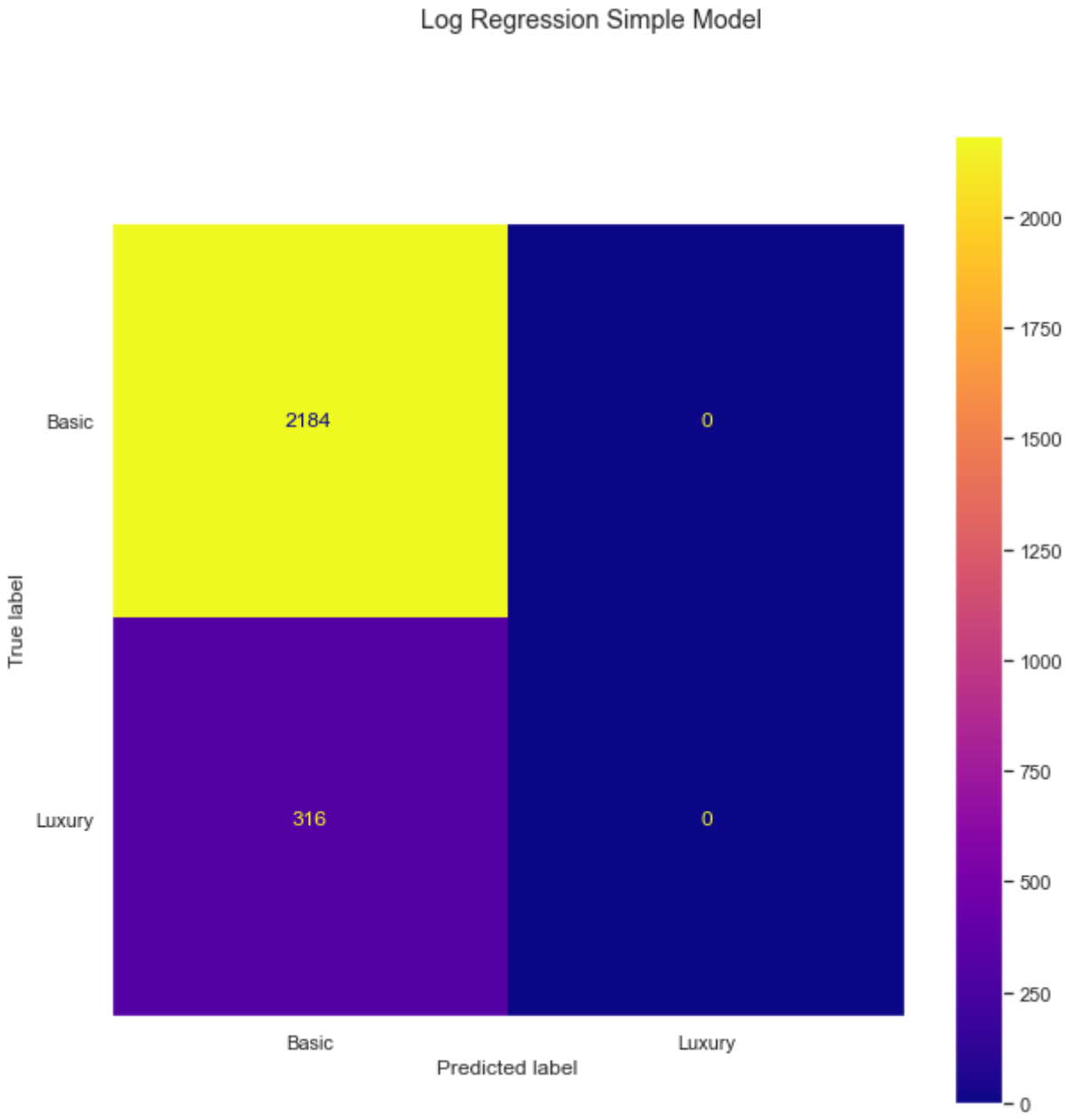
```
              precision    recall  f1-score   support

       Basic       0.87      1.00      0.93      2184
      Luxury       0.00      0.00      0.00       316

    accuracy                           0.87      2500
   macro avg       0.44      0.50      0.47      2500
weighted avg       0.76      0.87      0.81      2500
```

All the metrics for the prediction, recall and f1 score for "Luxury" values are the same for the Dummy Classifier and LogReg training data. They are all insufficient.

In [53]:
```
#Print out array confusion matrix
print(confusion_matrix(y_test, lr_y_test_pred, labels = ["Basic","Luxury"]))
```

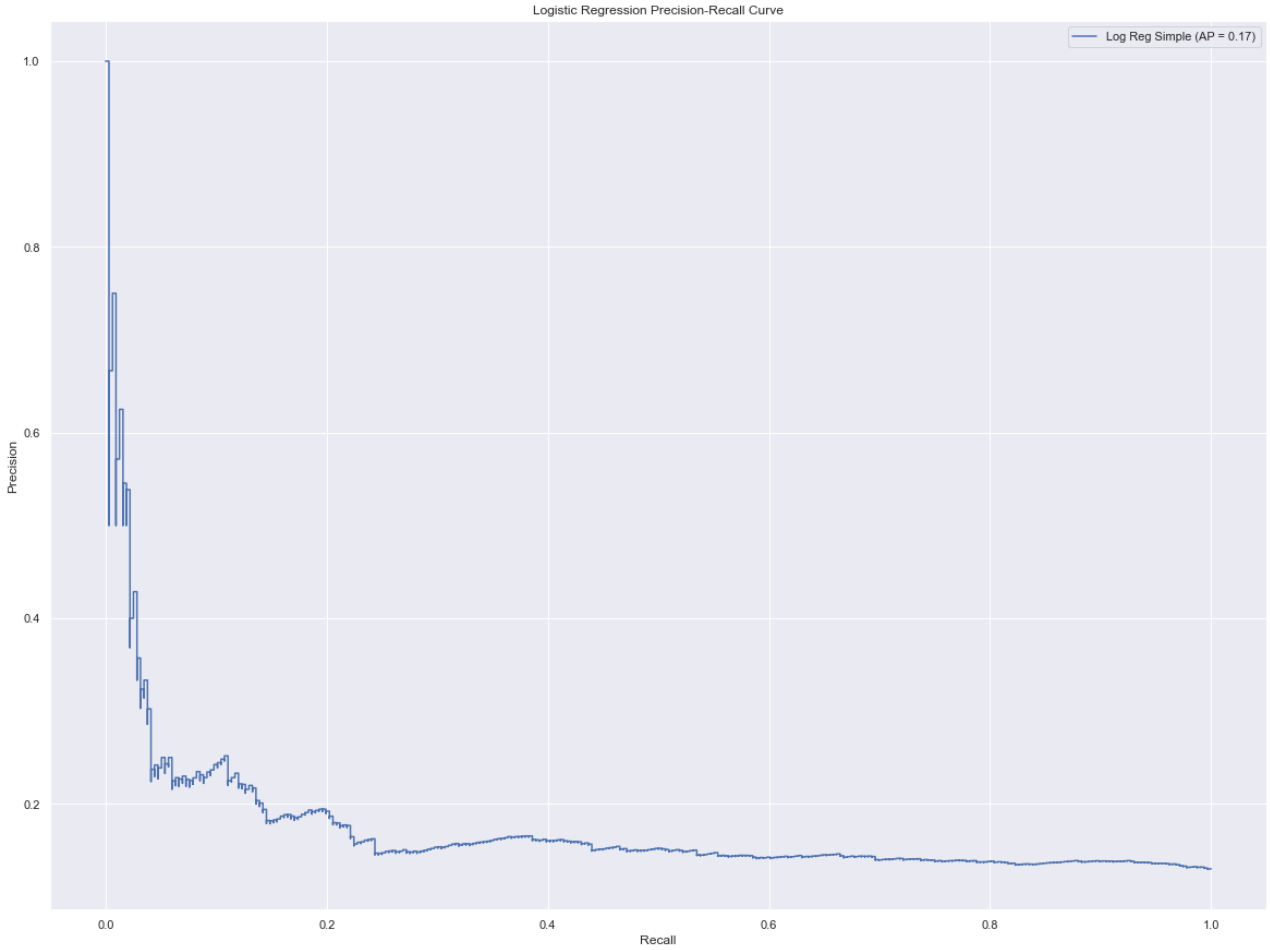
```
[[2184    0]
 [ 316    0]]
```

In [54]:
```
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Log Regression Simple Model")
plot_confusion_matrix(lr_simple, X_test, y_test, ax=ax, cmap="plasma");
plt.grid(False)
```



In [55]:
```
plot_precision_recall_curve(lr_simple, X_test, y_test, name = "Log Reg Simple")
#axis labels
plt.title("Logistic Regression Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
#Legend
plt.legend()
```

Out[55]:
```
<matplotlib.legend.Legend at 0x7fc4cb1c0580>
```



Currently the Dummy Classifier and the Logistic Regression metrics for the "Luxury" value are all insufficient. None of the models guess any value of "Luxury" correct.

The AUC-PR of Average Precision for the Precision Recall score is .17. This is an imporvement on the dummy classifier. So this provides a better chance at randomly guessing.

Model 2.2 - Decision Tree Model (Simple Model)

The Decision Tree model will classify the data in various splits based on "True" or "False" down through decomposition. It is a an explainable model, but is susceptible to overfitting.

In [56]:
```
#Instantiate
dt_simple = DecisionTreeClassifier(random_state=42, criterion="entropy")
#Fit training data
dt_simple.fit(X_train,y_train)
#Predict with test
dt_y_train_pred = dt_simple.predict(X_train)
dt_y_test_pred = dt_simple.predict(X_test)
dt_y_test_prob = dt_simple.predict_proba(X_test)
```

Train Data Evaluation

In [57]:
```
# Decision Tree CV score
cv_score(dt_simple, X_train, y_train)
```

```
Accuracy of Model: 0.872
```

In [58]:
```
print("Luxury Train Scores: \n")
precision_recall(y_train, dt_y_train_pred,"Luxury")
```

```
Luxury Train Scores:

The Precision score is: 1.0
The Recall score is: 1.0
The F1 score is: 1.0
```

Out[58]:
```
(1.0, 1.0, 1.0)
```

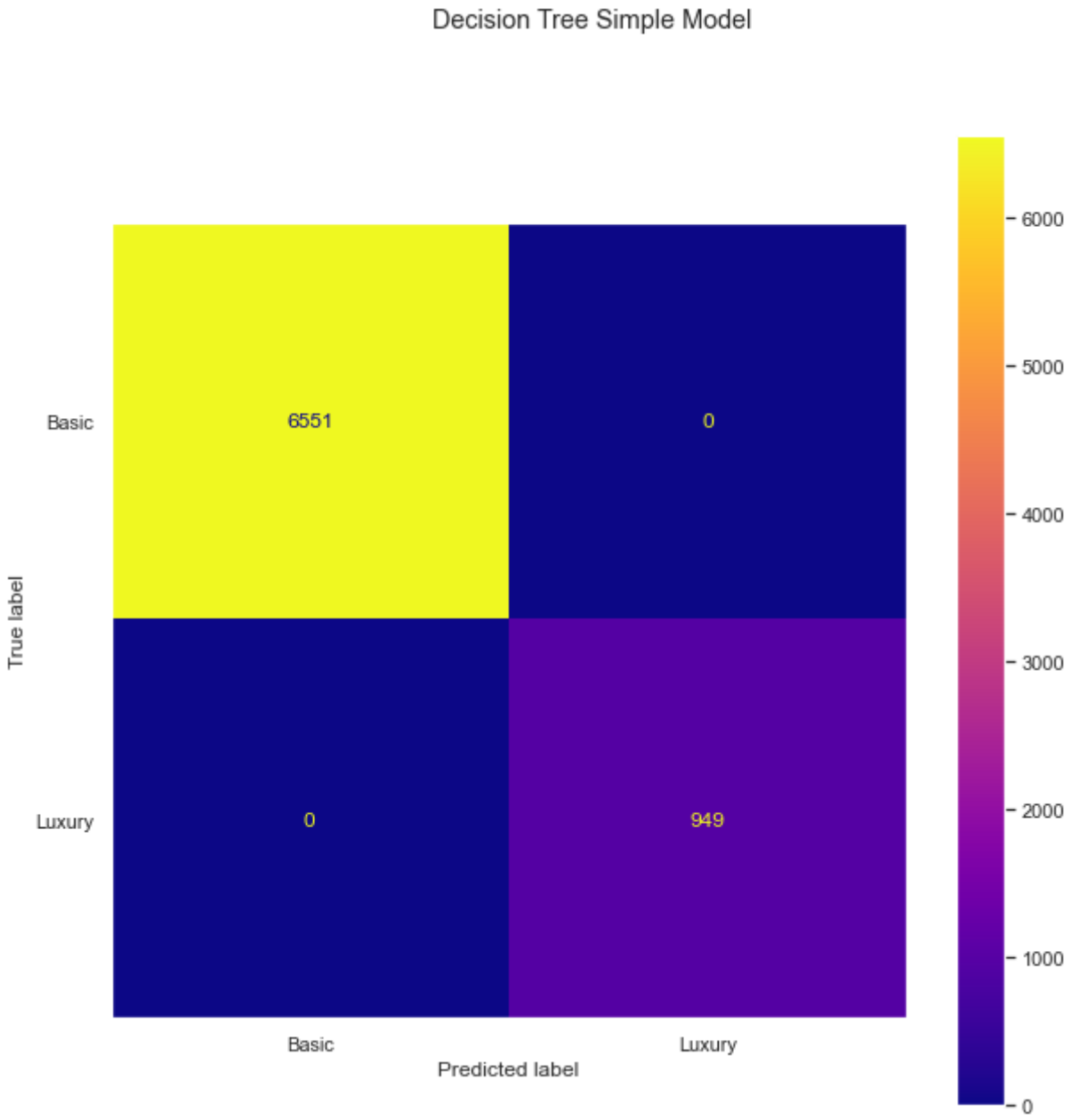
In [59]:
```
#Printing out report with scores -
print(classification_report(y_train, dt_y_train_pred))
```

```
              precision    recall  f1-score   support

       Basic       1.00      1.00      1.00      6551
      Luxury       1.00      1.00      1.00       949

    accuracy                           1.00      7500
   macro avg       1.00      1.00      1.00      7500
weighted avg       1.00      1.00      1.00      7500
```

In [60]:
```
print(confusion_matrix(y_train, dt_y_train_pred, labels = ["Basic","Luxury"]))
```

```
[[6551    0]
 [   0  949]]
```

In [61]:
```
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Decision Tree Simple Model")
plot_confusion_matrix(dt_simple, X_train, y_train, ax=ax, cmap="plasma");
plt.grid(False)
```



Our primary metrics are "perfect". Because this is training data, the tree is learning from the perfect data on how to make decisions.

Test Data Evaluation

In [62]:
```
cv_score(dt_simple, X_test, y_test)
```

```
Accuracy of Model: 0.87
```

In [63]:
```
print("Luxury Test Scores: \n")
precision_recall(y_test, dt_y_test_pred,"Luxury")
```

```
Luxury Test Scores:

The Precision score is: 0.4634920634920635
The Recall score is: 0.4620253164556962
The F1 score is: 0.46275752773375595
```

Out[63]:
```
(0.4634920634920635, 0.4620253164556962, 0.46275752773375595)
```

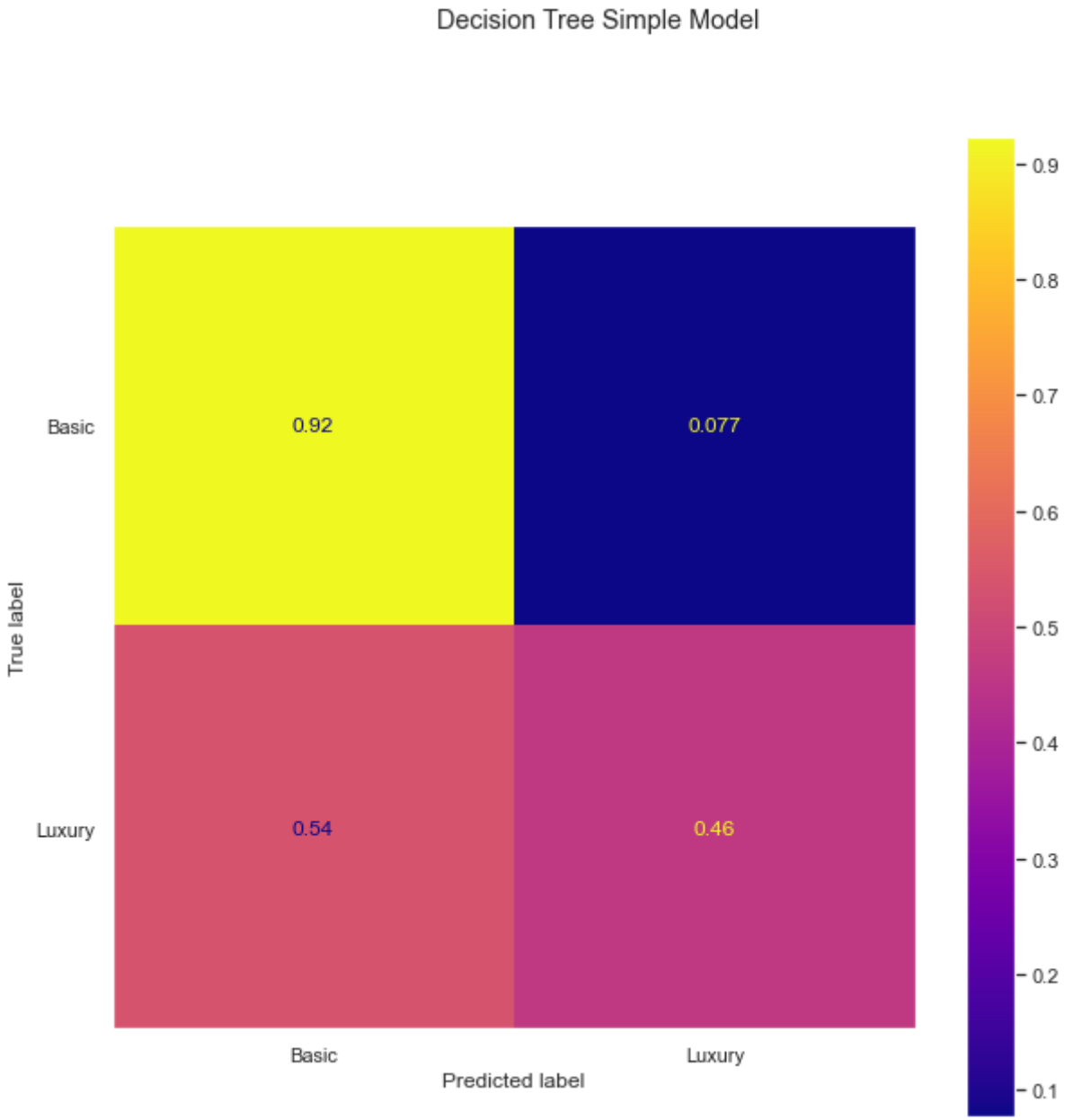
In [64]:
```
print(confusion_matrix(y_test, dt_y_test_pred, labels = ["Basic","Luxury"]))
```

```
[[2015  169]
 [ 170  146]]
```

In [65]:
```
print(classification_report(y_test, dt_y_test_pred))
```

```
              precision    recall  f1-score   support

       Basic       0.92      0.92      0.92      2184
      Luxury       0.46      0.46      0.46       316

    accuracy                           0.86      2500
   macro avg       0.69      0.69      0.69      2500
weighted avg       0.86      0.86      0.86      2500
```

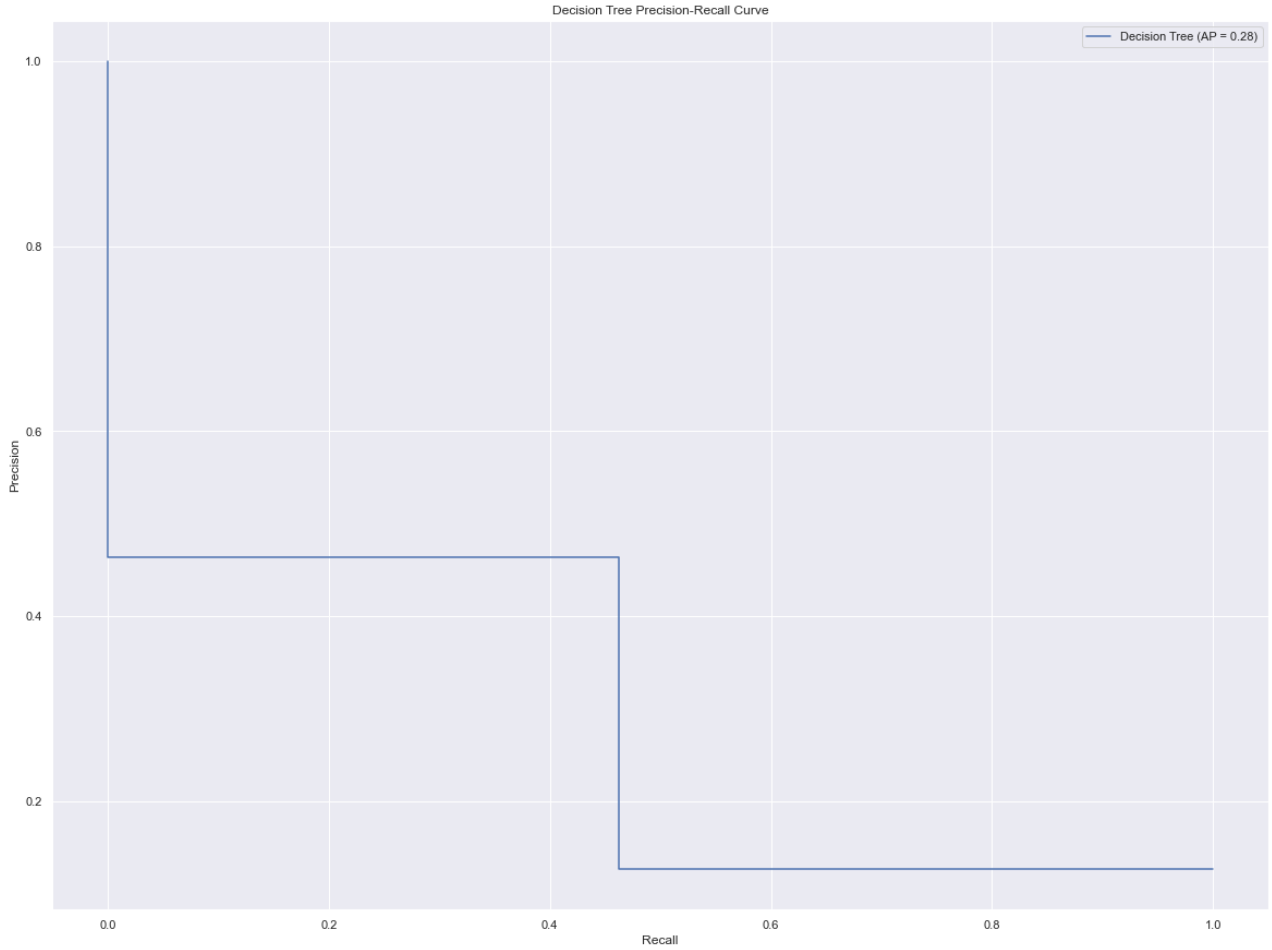
In [66]:
```
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Decision Tree Simple Model")
plot_confusion_matrix(dt_simple, X_test, y_test, ax=ax,normalize = "true", cmap=
plt.grid(False)
```



Explaination of Precsion/Recall Curve Use

In [67]:
```
plot_precision_recall_curve(dt_simple, X_test, y_test, name = 'Decision Tree')

#plt.plot(dt_recall, dt_precision, marker='.', label='Luxury')
#axis labels
plt.title("Decision Tree Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
#Legend
plt.legend()
```

Out[67]:
```
<matplotlib.legend.Legend at 0x7fc4f81627c0>
```



The precision and f1 score (both 46%) are not stellar but in comparision to the Log Reg testing results. The decision tree model greatly improves on the results.

The confusion matrix shows the TP guessing 146 Luxury instances correctly (46% of Luxury instances) but guessing 169 incorrectly (.08% of Basic instances). The previous models were not useful for sufficient comparison.

The Precision-Recall curve shows .28. This is a 65% improvement on the AUC-PC/AP score than the Logistic Regression.

Model 2.3 - Random Forest Model (Simple Model)

The Random Forest model is a make up of several decision trees. It is not as easily explainable as the decision tree but provides more rubustness generally leading to better accuracy and precision.

The decision tree provided better results than the logisic regression model. The Random Forest model should potentially lead to an increase in precision from the decision tree.

In [68]:
```
#Instantiate
rf_simple = RandomForestClassifier(random_state = 42)
#Fit training data
rf_simple.fit(X_train,y_train)
#Predict with test
rf_y_train_pred = rf_simple.predict(X_train)
```

```
rf_y_test_pred = rf_simple.predict(X_test)
rf_y_test_prob = rf_simple.predict_proba(X_test)
```

Train Data Evaluation

In [69]:
```
#CV Scores
cv_score(rf_simple, X_train, y_train)
```

```
Accuracy of Model: 0.866
```

In [70]:
```
print("Luxury Train Scores: \n")
precision_recall(y_train, rf_y_train_pred,"Luxury")
```

```
Luxury Train Scores:

The Precision score is: 1.0
The Recall score is: 1.0
The F1 score is: 1.0
```

Out[70]:
```
(1.0, 1.0, 1.0)
```

In [71]:
```
#Confusion Matrix
print(confusion_matrix(y_train, rf_y_train_pred, labels = ["Basic","Luxury"]))
```

```
[[6551    0]
 [   0  949]]
```

In [72]:
```
#Classification Report
print(classification_report(y_train, rf_y_train_pred))
```

```
              precision    recall  f1-score   support

       Basic       1.00      1.00      1.00      6551
      Luxury       1.00      1.00      1.00       949

    accuracy                           1.00      7500
   macro avg       1.00      1.00      1.00      7500
weighted avg       1.00      1.00      1.00      7500
```

Just like the decision tree, the training data leads to "perfect" score in the classification report. These are not sufficient to use.

Test Data Evaluation

In [73]:
```
#CV Scores
cv_score(rf_simple, X_test, y_test)
```

```
Accuracy of Model: 0.872
```

In [74]:
```
print("Luxury Test Scores: \n")
precision_recall(y_test, rf_y_test_pred,"Luxury")
```

```
Luxury Test Scores:

The Precision score is: 0.5185185185185185
The Recall score is: 0.35443037974683544
The F1 score is: 0.42105263157894735
```

Out[74]:
```
(0.5185185185185185, 0.35443037974683544, 0.42105263157894735)
```

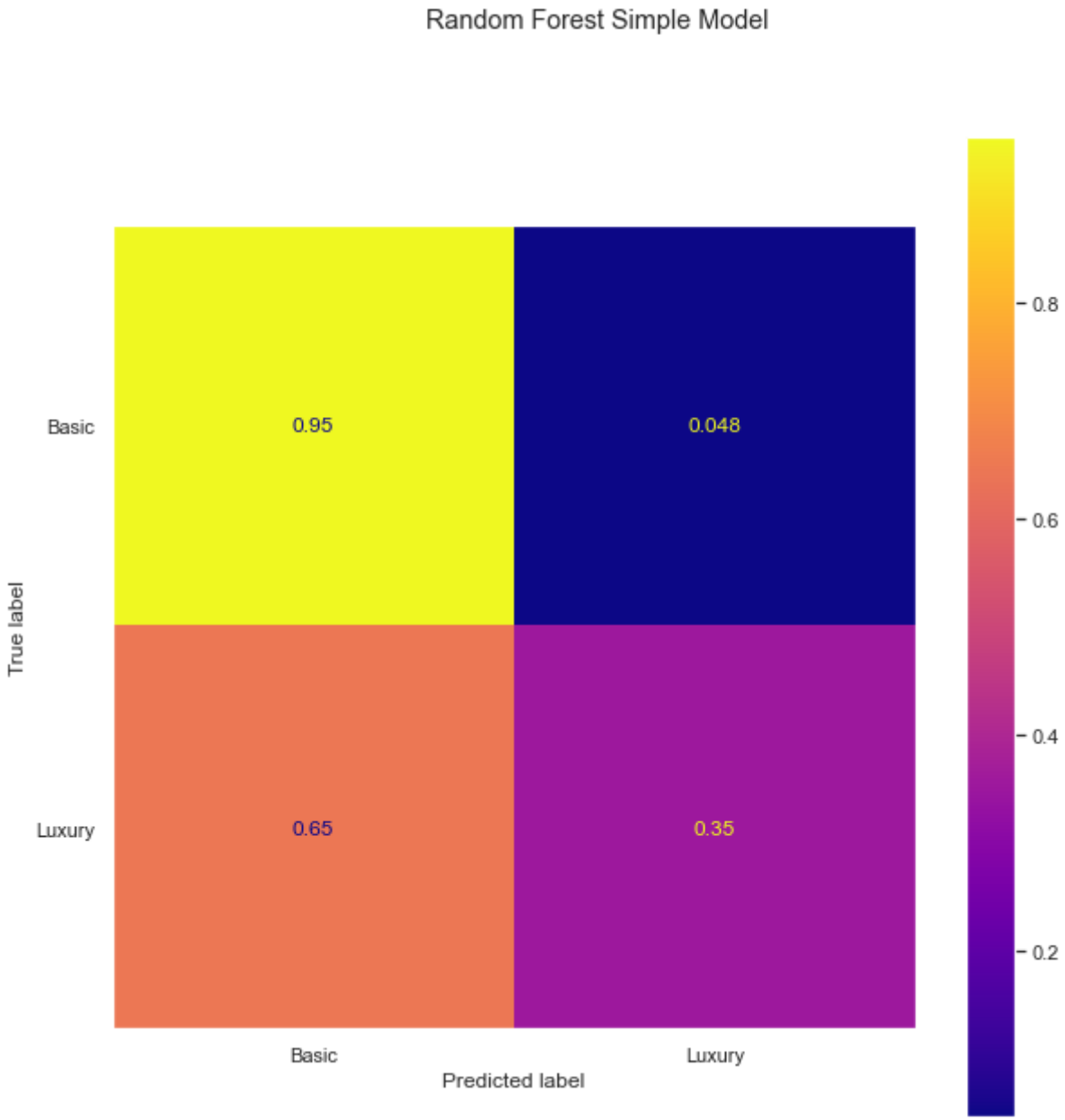
In [75]:
```
print(confusion_matrix(y_test, rf_y_test_pred, labels = ["Basic","Luxury"]))
```

```
[[2080  104]
 [ 204  112]]
```

In [76]:
```
print(classification_report(y_test, rf_y_test_pred))
```

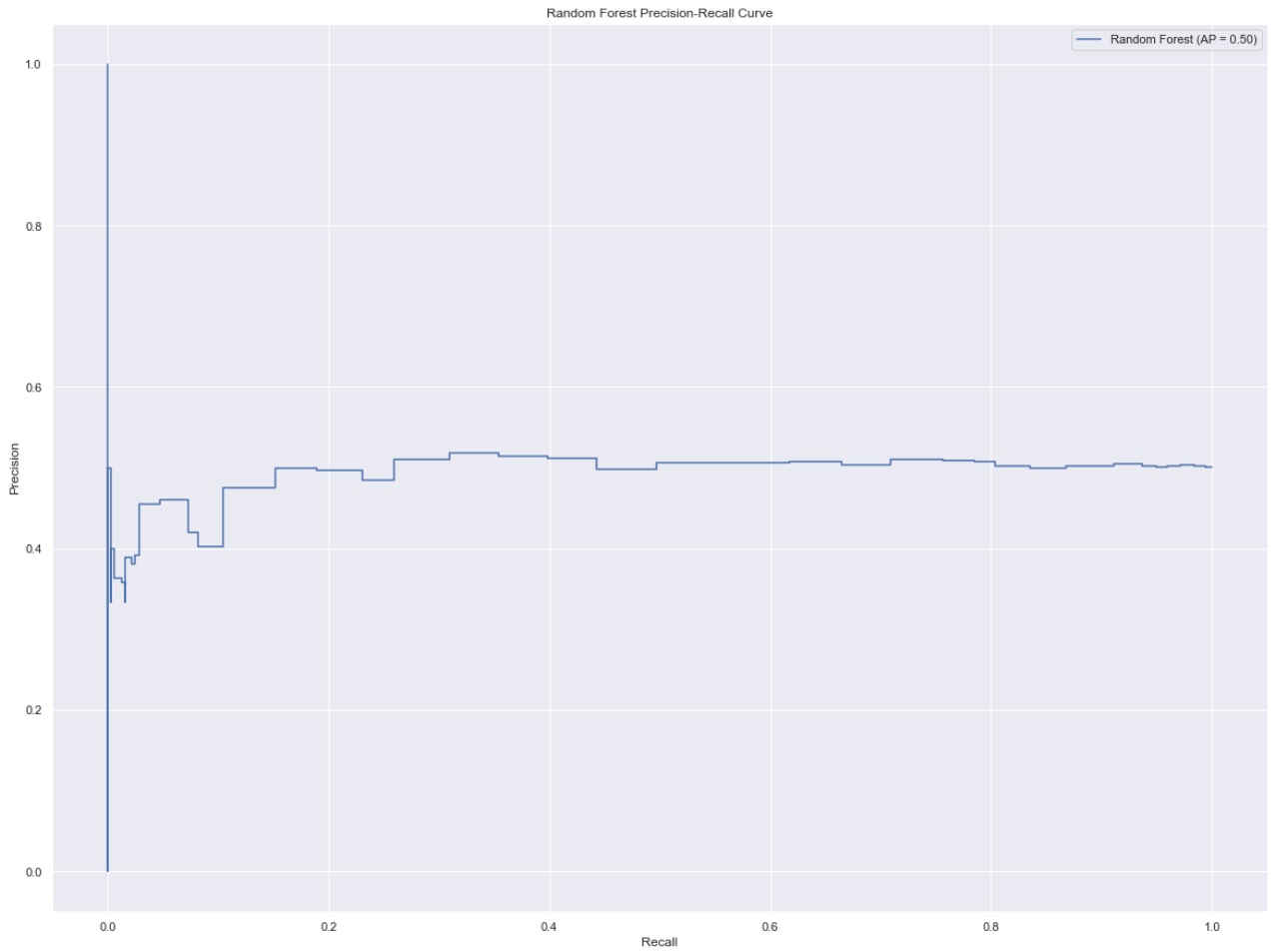
```
              precision    recall  f1-score   support

       Basic       0.91      0.95      0.93      2184
      Luxury       0.52      0.35      0.42       316

    accuracy                           0.88      2500
   macro avg       0.71      0.65      0.68      2500
weighted avg       0.86      0.88      0.87      2500
```

In [77]:
```
fig, ax = plt.subplots(figsize = (10,10), facecolor = "white")
fig.suptitle("Random Forest Simple Model")
plot_confusion_matrix(rf_simple, X_test, y_test, ax=ax,normalize = "true", cmap=
plt.grid(False)
```



In [78]:
```
#Plotting the precision recall curve
plot_precision_recall_curve(rf_simple, X_test, y_test, name = 'Random Forest')
#axis labels
plt.title("Random Forest Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
#Legend
plt.legend()
```

Out[78]:
```
<matplotlib.legend.Legend at 0x7fc4cb66c0d0>
```



The "Luxury" value had 52% precision, this is an (~12%) increase from the Decision Tree's "Luxury" value of 48%.

The Precision-Recall curve shows .50. This is over a 100% improvement on the AUC-PC/AP score than the Decision Tree's classifier and almost a 200% increase from the Logistic Regression model.

The Random Forest classifier seems to be the best choice, even without any hypertuning.

3 Data Prep - SMOTE (Data Imbalance), Scaling & Gridsearch

Logistic Regression Cumulative Lab

The next steps will be to try to apply SMOTE and scaling to the data help with the imbalance. The scaling will best help with the Logistic Regression classifier. Scaling will not help or hurt the Decsion Tree and Random Forest classifiers.

The intial application of these transformations will be applied using a loop with a pipeline.

Afterwards we will apply gridsearch to the same pipeline.

3.1 SMOTE and Scaling : Log Regression, Decsion Tree & Random Forest Classifier

Applying SMOTE and Scaling to Classifier

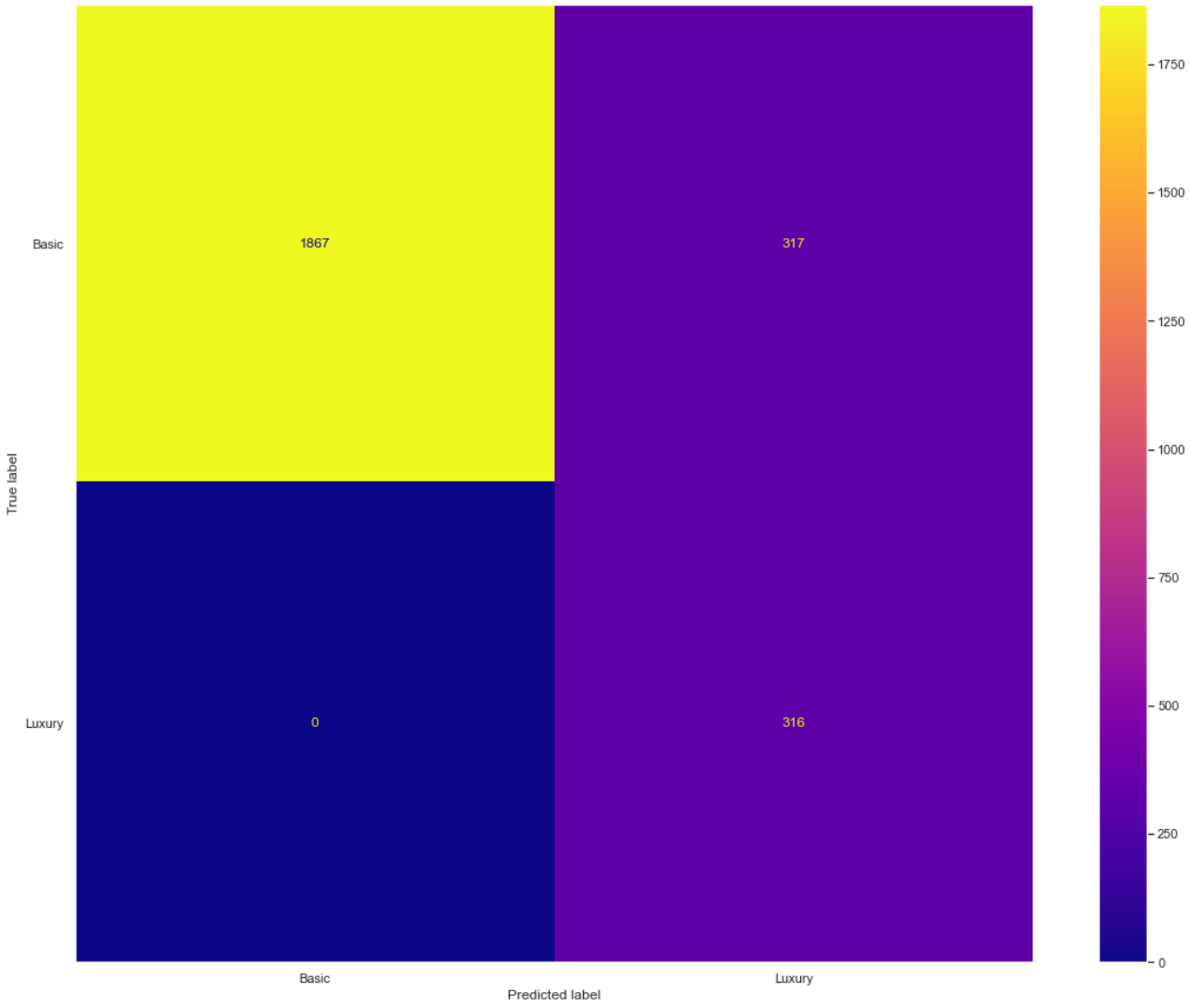
In [79]:
```
#Create a function, run loop for each type of classifier
#Log Regression, Decision Tree, Random Forest

#Create Pipeline for each classifier
#Create dict of classifiers
clf_dict = {"lgr":LogisticRegression(random_state=42), "dt":DecisionTreeClassifi
pipe_dict = {} # Create a dictionary of the classifiers in the pipeline

for name,clf in clf_dict.items():

    #Name of Classifier
    print("{} Classifier".format(clf))
    print('---' * 20)
    print("\n")


    #Create pipelines with Scaling, SMOTE
    imb_pipe = ImPipeline(steps=[('ct', StandardScaler()),
                                 ('sm', SMOTE( random_state=42)),
                                 (name, clf)])

    print(imb_pipe)
    print("\n")

    #fitting the pipeline on the training data
    imb_pipe.fit(X_train, y_train)



    #Training Accuracy score
    print("Training Accuracy score:{0:0.2f}".format(imb_pipe.score(X_train, y_tr


    #Testing Accuracy score
    print("Testing Accuracy score:{0:0.2f}".format(imb_pipe.score(X_test, y_test
    print("\n")

    #Creating a dictionary of the different pipelines
    pipe_dict[name] = imb_pipe

    #Predicting the new data
    y_test_pred = imb_pipe.predict(X_test)

    #Predicting the new data
    y_test_prob = imb_pipe.predict_proba(X_test)
    y_test_prob = y_test_prob[:,1]

    #Classification Report
    print("Classification Report:\n",classification_report(y_test, y_test_pred))
    print("\n")

    #Confusion Matrix
    cf_matrix = confusion_matrix(y_test, y_test_pred, labels = ["Basic","Luxury"
```

```
    print("Confusion Matrix: \n", cf_matrix)
    print("\n")


    #Plot Confusion Map Matrix
    fig = plt.figure(figsize = (5,5), facecolor = "white")
    plot_confusion_matrix(imb_pipe, X_test, y_test, cmap="plasma")
    plt.grid(False)
    print("\n")

    #Plotting Precision-Recall Curve
    plot_precision_recall_curve(imb_pipe, X_test, y_test, name = clf )

    #axis labels
    plt.title("Precision-Recall Curve: {}".format(clf))
    plt.xlabel("Recall")
    plt.ylabel("Precision")

    #Legend
    plt.legend()
    print('---' * 20)
    print("\n")
```

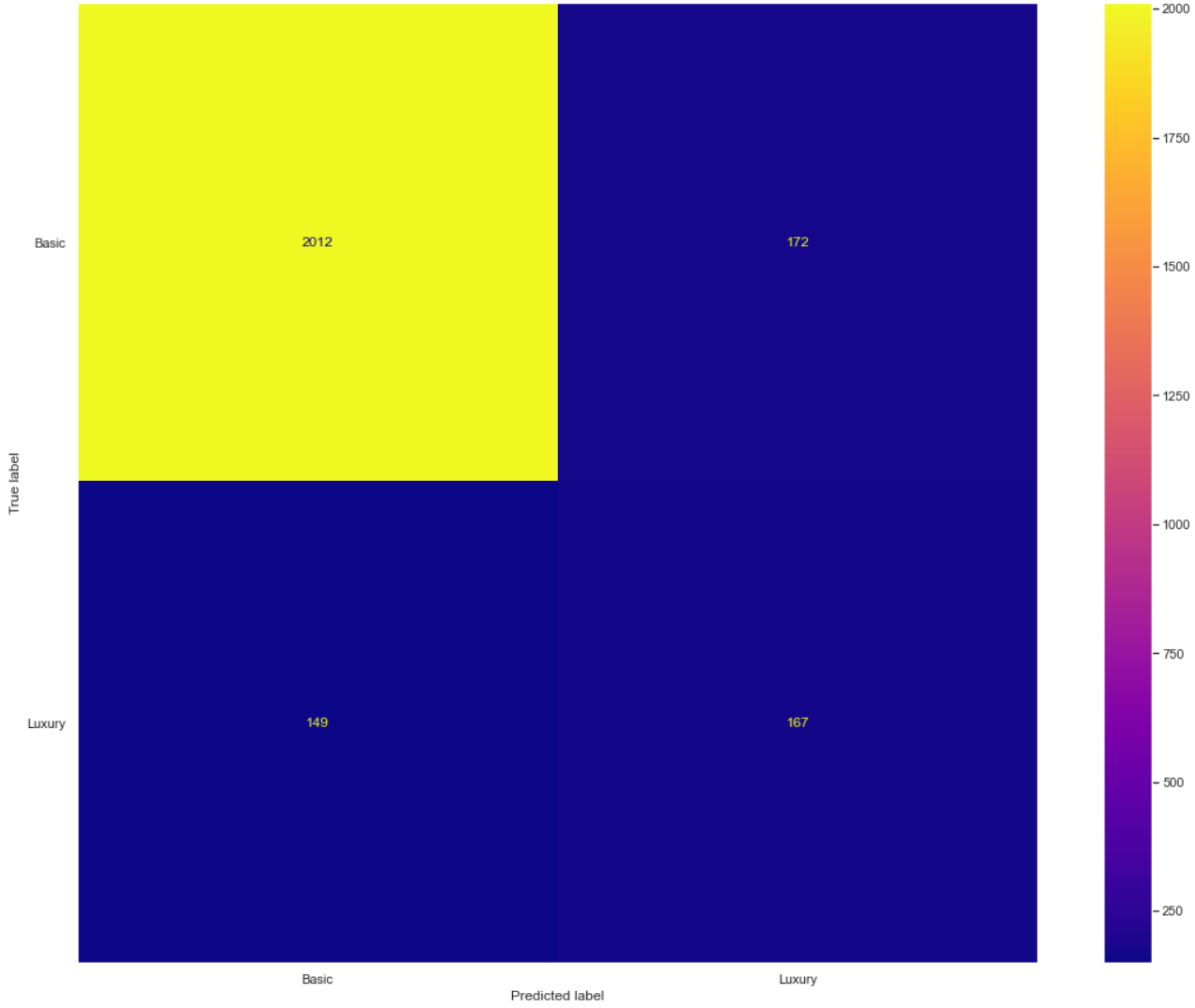
```
LogisticRegression(random_state=42) Classifier
------------------------------------------------------------


Pipeline(steps=[('ct', StandardScaler()), ('sm', SMOTE(random_state=42)),
                ('lgr', LogisticRegression(random_state=42))])


Training Accuracy score:0.87
Testing Accuracy score:0.87


Classification Report:
              precision    recall  f1-score   support

       Basic       1.00      0.85      0.92      2184
      Luxury       0.50      1.00      0.67       316

    accuracy                           0.87      2500
   macro avg       0.75      0.93      0.79      2500
weighted avg       0.94      0.87      0.89      2500



Confusion Matrix:
 [[1867  317]
 [   0  316]]



------------------------------------------------------------


DecisionTreeClassifier(random_state=42) Classifier
```

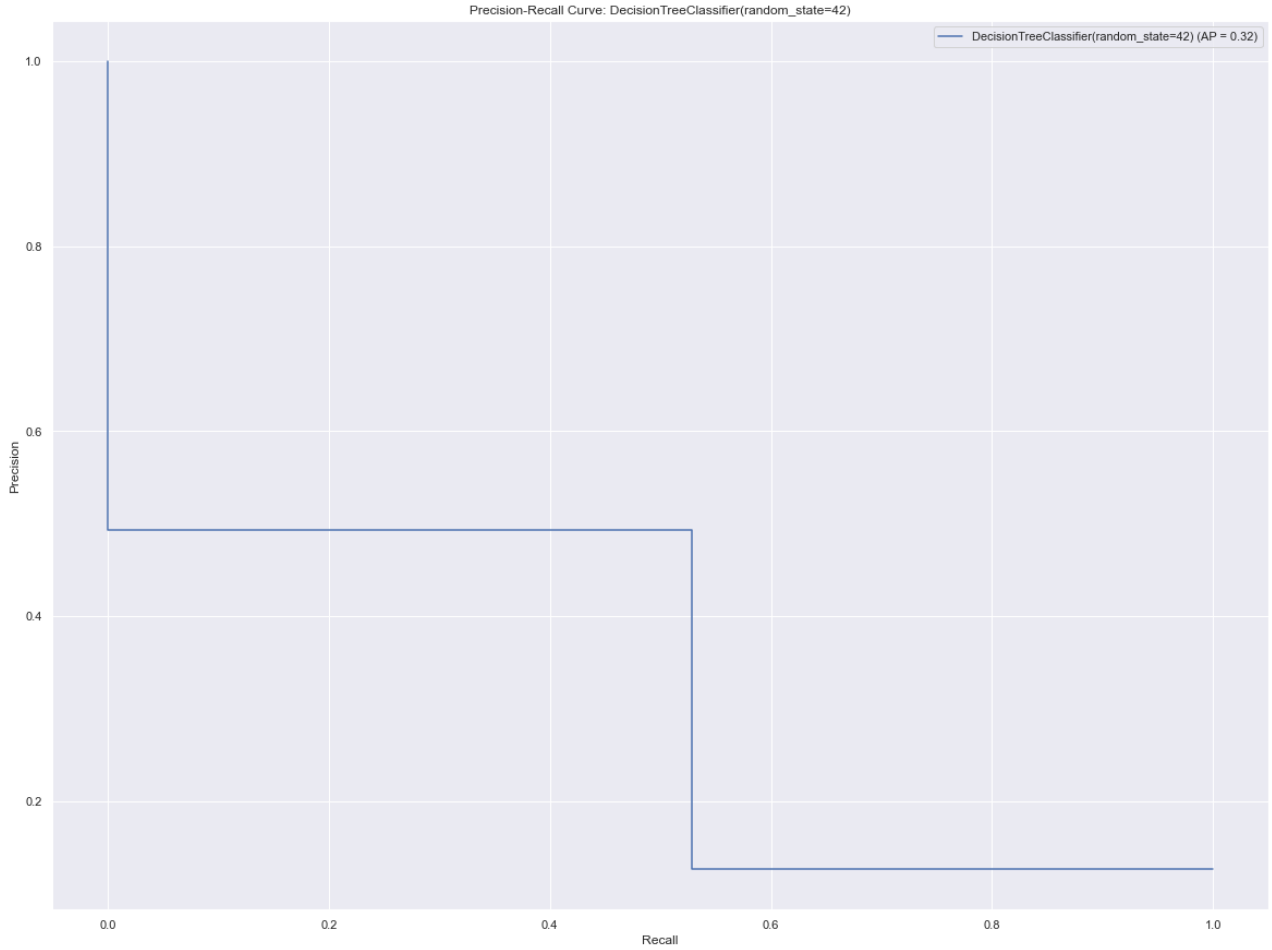
```
------------------------------------------------------------


Pipeline(steps=[('ct', StandardScaler()), ('sm', SMOTE(random_state=42)),
                ('dt', DecisionTreeClassifier(random_state=42))])


Training Accuracy score:1.00
Testing Accuracy score:0.87


Classification Report:
               precision    recall  f1-score   support

       Basic       0.93      0.92      0.93      2184
      Luxury       0.49      0.53      0.51       316

    accuracy                           0.87      2500
   macro avg       0.71      0.72      0.72      2500
weighted avg       0.88      0.87      0.87      2500



Confusion Matrix:
 [[2012  172]
 [ 149  167]]




------------------------------------------------------------


RandomForestClassifier(random_state=42) Classifier
------------------------------------------------------------


Pipeline(steps=[('ct', StandardScaler()), ('sm', SMOTE(random_state=42)),
                ('rf', RandomForestClassifier(random_state=42))])


Training Accuracy score:1.00
Testing Accuracy score:0.88


Classification Report:
               precision    recall  f1-score   support

       Basic       0.96      0.89      0.93      2184
      Luxury       0.51      0.76      0.61       316

    accuracy                           0.88      2500
   macro avg       0.73      0.83      0.77      2500
weighted avg       0.91      0.88      0.89      2500



Confusion Matrix:
 [[1948  236]
 [  75  241]]




------------------------------------------------------------
```

```
<Figure size 360x360 with 0 Axes>
```

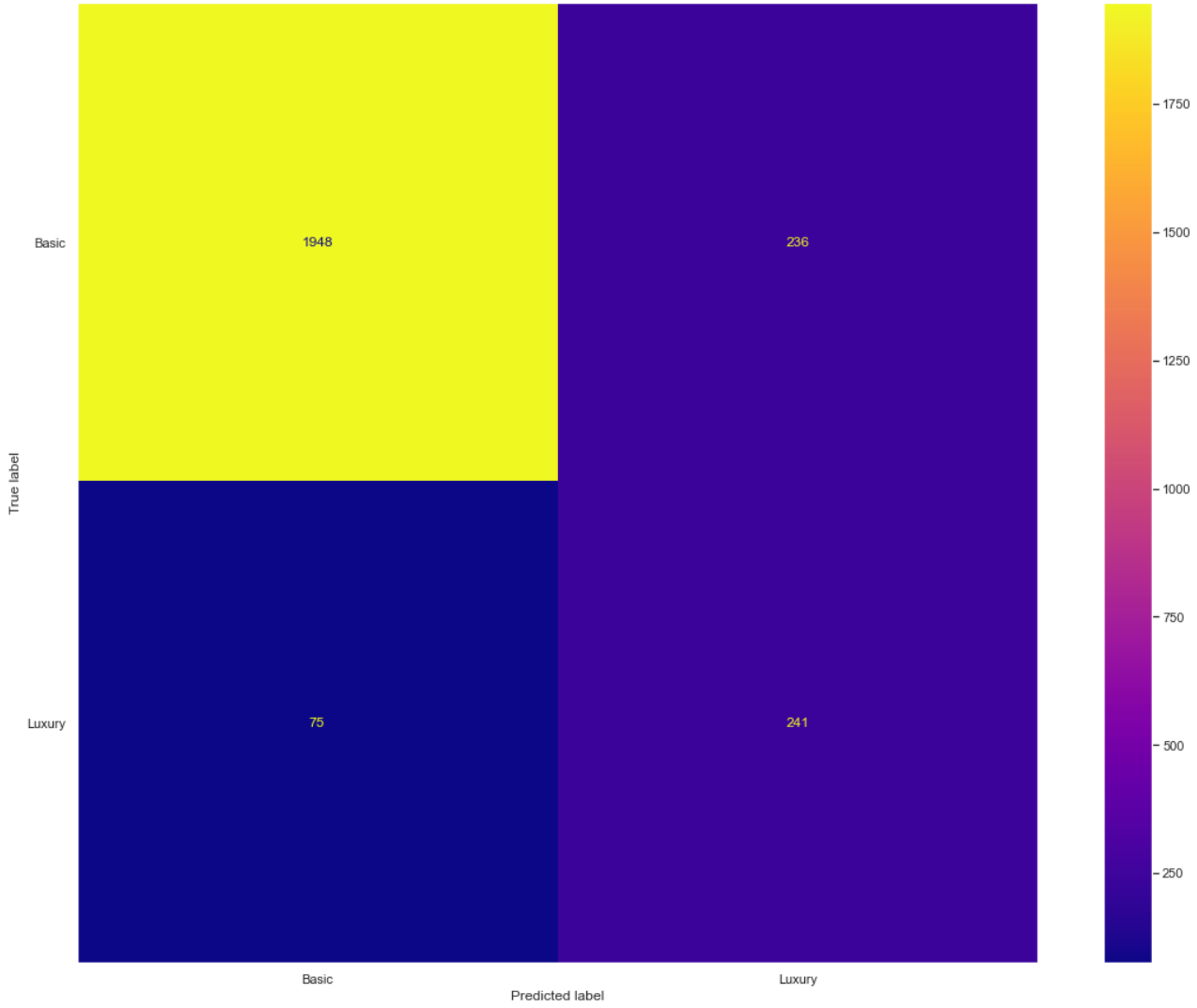


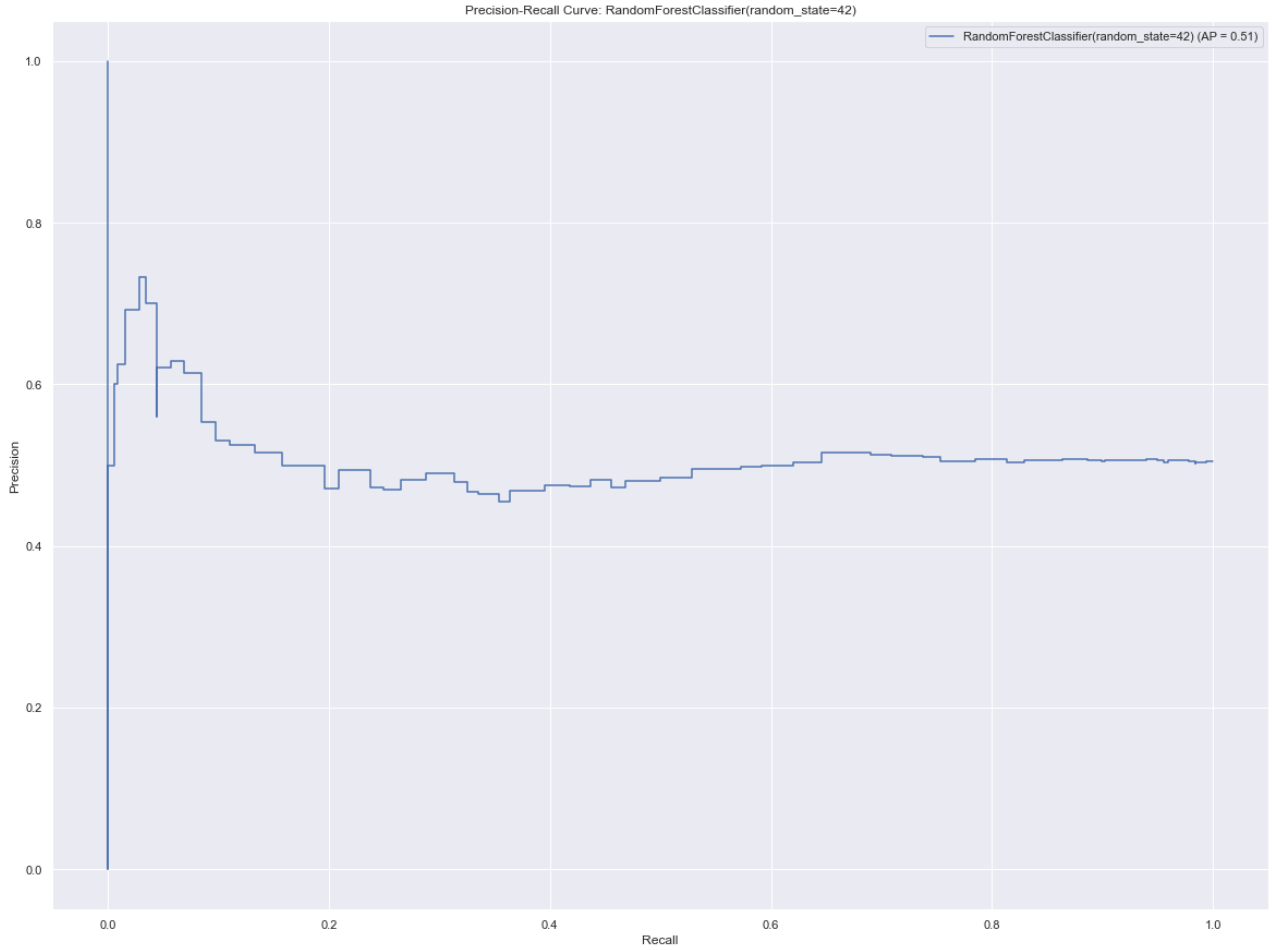

```
<Figure size 360x360 with 0 Axes>
```


Basic
Luxury
True label
2012
172
149
167
Basic
Luxury
Predicted label
2000
1750
1500
1250
1000
750
500
250




```
<Figure size 360x360 with 0 Axes>
```


Basic
Luxury
True label
1948
236
75
241
Basic
Luxury
Predicted label
1750
1500
1250
1000
750
500
250



With the SMOTE and scaling applied:

- **Logisitic Regression: Luxury Precision - .5, Luxury f1_score - .67, AUC-PR/Average Precison -.52**
- **Decision Tree: Luxury Precision - .49 Luxury f1_score - .51, AUC-PR/Average Precison - .32**
- **Random Forest: Luxury Precision - .51, Luxury f1_score - .61, AUC-PR/Average Precison -.51**

Statistically Logistic Regression comes out as a slightly better classifier than the Random Forest. These classifiers still have default parameters with no hypertuning.

Looking at the Confusion Matrix:

- **Logisitic Regression classifier correctly guesses 100% of the Luxury instances. However, there were 15% (FP = 317) of the "Basic" Instances incorrectly guessed.**
- **Random Forest classifier correctly guesses 76% of the Luxury instances. However, there were 11% (FP = 236) of the "Basic" Instances incorrectly guessed.**

Looking at the these values supports my original goal of guessing "Luxury" instances but minimizing False Positives.

3.2 Adding Gridsearch to the Pipeline

Adding gridsearch to the previous pipeline to search for the best hyperparameters. Gridsearch allows running various combinations of hyperparameters and viewing their performance. The gridsearch parameters were manually chosen. Because of resource limitations a small amount of parameter and hyperparameter options were chosen. There were attempts for hyperparamters options chosen on a spectrum.

In [80]:
```
#Function used to print out specific results fromm gridsearch

def gridsearch_score(grid_name):

    #Getting prediction value
    gs_y_pred_train = grid_name.predict(X_train)
    #Getting prediction value
    gs_y_pred_test = grid_name.predict(X_test)

    print("Best Estimator:", grid_name.best_estimator_)
    print("\n")
    print("Best Paremeters:", grid_name.best_params_)
    print("\n")
    print("Best Precision Score:", grid_name.best_score_)
    print("\n")


    #Checking our metrics to see how well our model performed
    print("Best Train Estimator Score:", grid_name.best_estimator_.score(X_train
    print("Train Classification Report:\n", classification_report(y_train, gs_y_
    print("\n")
    print("Best Test Estimator Score:", grid_name.best_estimator_.score(X_test,
    print("Test Classification Report:\n", classification_report(y_test, gs_y_pr
    print('---' * 20)
    print("\n")
```

Making a scoring dictionary parameter. The score parameter dictionary can consist of precision, recall, f1_score and accuracy.

In [81]:
```
#Creation of of scoring dictionary for the gridsearch scoring multimetrics
cus_rec = make_scorer(recall_score,pos_label="Luxury")
cus_prec = make_scorer(precision_score,pos_label="Luxury")
cus_f1 =  make_scorer(f1_score,pos_label="Luxury")
cus_accur =  make_scorer(accuracy_score)
scoring = {"Precision":cus_prec,"Rec": cus_rec,"F1":cus_f1,"Accuracy":cus_accur}
```

3.2.1 LogisticRegression GridsearchCV

- SMOTE Paramters:
  - Sampling Strategy: Wanted to sample to increase the values for the minority class ("Luxury"). Chose various values for the hyperparameters that support this.
- Logistic Regression:
  - Logistic Parameters Paramters Chosen:
    - Penalty: Chose all the values of the penalty to try, including the default
    - Sovler: Chose all the values of the solver to try, including the default

- GridSearchCV Parameters Precion:
  - n_jobs: The local computer running on 64GB of RAM, it was enough to run multiple jobs in parallel. This is part of the reason multiple hyperparameters and values were chosen.

In [82]:
```
#Logistic Regression parameters and GridSearch instantiation
parameters = {"sm__sampling_strategy":["minority",.2,.6],
                      'lgr__penalty': ["none","l1"'l2', 'elasticnet'],
                      'lgr__solver': ["lbfgs", "liblinear", "saga"]}

gs_lgr = GridSearchCV(estimator= pipe_dict["lgr"], param_grid=parameters,
n_jobs= 3, refit = "Precision", scoring=scoring, return_train_score= True)
```

In [83]:
```
#Fitting the classifier to Training Data
gs_lgr.fit(X_train, y_train)
```

Out[83]:
```
GridSearchCV(estimator=Pipeline(steps=[('ct', StandardScaler()),
                                       ('sm', SMOTE(random_state=42)),
                                       ('lgr',
                                        LogisticRegression(random_state=42))]),
             n_jobs=3,
             param_grid={'lgr__penalty': ['none', 'l112', 'elasticnet'],
                         'lgr__solver': ['lbfgs', 'liblinear', 'saga'],
                         'sm__sampling_strategy': ['minority', 0.2, 0.6]},
             refit='Precision', return_train_score=True,
             scoring={'Accuracy': make_scorer(accuracy_score),
                      'F1': make_scorer(f1_score, pos_label=Luxury),
                      'Precision': make_scorer(precision_score, pos_label=Luxur
y),
                      'Rec': make_scorer(recall_score, pos_label=Luxury)})
```

Printing out the Gridsearch's best estimator, parameters, precision score and results for training and testing

In [84]:
```
#Running the function to print best estimator, scores and results for training a
gridsearch_score(gs_lgr)
```

```
Best Estimator: Pipeline(steps=[('ct', StandardScaler()),
                ('sm', SMOTE(random_state=42, sampling_strategy='minority')),
                ('lgr', LogisticRegression(penalty='none', random_state=42))])


Best Paremeters: {'lgr__penalty': 'none', 'lgr__solver': 'lbfgs', 'sm__sampling_
strategy': 'minority'}


Best Precision Score: 0.491081404082648


Best Train Estimator Score: 1.0
Train Classification Report:
              precision    recall  f1-score   support

       Basic       1.00      0.85      0.92      6551
      Luxury       0.49      1.00      0.66       949

    accuracy                           0.87      7500
   macro avg       0.75      0.92      0.79      7500
weighted avg       0.94      0.87      0.89      7500
```

```
Best Test Estimator Score: 1.0
Test Classification Report:
              precision    recall  f1-score   support

       Basic       1.00      0.85      0.92      2184
      Luxury       0.50      1.00      0.67       316

    accuracy                           0.87      2500
   macro avg       0.75      0.93      0.79      2500
weighted avg       0.94      0.87      0.89      2500

------------------------------------------------------------
```

3.2.2 Decision Tree GridsearchCV

- SMOTE Paramters:
  - Sampling Strategy: Wanted to sample to increase the values for the minority class ("Luxury"). Chose various values for the hyperparameters that support this.
- Decision Tree:
  - Decision Tree Parameters Chosen:
    - Criterion: Chose the default and "Entropy"
    - Max Depth: Chose the default and two values across a wide range.
    - Min Samples Split:Chose several values for minimum samples.
    - Min Samples Leaf: Chose several small values for minimum leaf node
- GridSearchCV Parameters Precion:
  - n_jobs: The local computer running on 64GB of RAM, it was enough to run multiple jobs in parallel. This is part of the reason multiple hyperparameters and values were chosen.

In [85]:
```
#Decision Tree parameters and GridSearch instantiation
parameters = {"sm__sampling_strategy":["minority",.2,.6],
                      'dt__criterion': ["gini", "entropy"],
                      'dt__max_depth': [None, 5,100],
                      'dt__min_samples_split': [2,7,9],
                      'dt__min_samples_leaf': [2,5,7]}

gs_dt = GridSearchCV(estimator= pipe_dict["dt"], param_grid=parameters,
n_jobs= 3, refit = "Precision", scoring=scoring, return_train_score= True)
```

In [86]:
```
#Fit GridSearch to Training Data
gs_dt.fit(X_train, y_train)
```

Out[86]:
```
GridSearchCV(estimator=Pipeline(steps=[('ct', StandardScaler()),
                                       ('sm', SMOTE(random_state=42)),
                                       ('dt',
                                        DecisionTreeClassifier(random_state=4
2))]),
             n_jobs=3,
             param_grid={'dt__criterion': ['gini', 'entropy'],
                         'dt__max_depth': [None, 5, 100],
                         'dt__min_samples_leaf': [2, 5, 7],
                         'dt__min_samples_split': [2, 7, 9],
```

```
                        'sm__sampling_strategy': ['minority', 0.2, 0.6]},
              refit='Precision', return_train_score=True,
              scoring={'Accuracy': make_scorer(accuracy_score),
                       'F1': make_scorer(f1_score, pos_label=Luxury),
                       'Precision': make_scorer(precision_score, pos_label=Luxur
y),
                       'Rec': make_scorer(recall_score, pos_label=Luxury)})
```

In [87]:
```
#Running the function to print best estimator, scores and results for training a
gridsearch_score(gs_dt)
```

```
Best Estimator: Pipeline(steps=[('ct', StandardScaler()),
                ('sm', SMOTE(random_state=42, sampling_strategy='minority')),
                ('dt',
                 DecisionTreeClassifier(min_samples_leaf=7, random_state=42))])


Best Paremeters: {'dt__criterion': 'gini', 'dt__max_depth': None, 'dt__min_sampl
es_leaf': 7, 'dt__min_samples_split': 2, 'sm__sampling_strategy': 'minority'}


Best Precision Score: 0.5104866800724549


Best Train Estimator Score: 1.0
Train Classification Report:
              precision    recall  f1-score   support

       Basic       0.98      0.96      0.97      6551
      Luxury       0.76      0.85      0.80       949

    accuracy                           0.95      7500
   macro avg       0.87      0.91      0.89      7500
weighted avg       0.95      0.95      0.95      7500



Best Test Estimator Score: 1.0
Test Classification Report:
              precision    recall  f1-score   support

       Basic       0.95      0.92      0.93      2184
      Luxury       0.53      0.66      0.59       316

    accuracy                           0.88      2500
   macro avg       0.74      0.79      0.76      2500
weighted avg       0.90      0.88      0.89      2500

------------------------------------------------------------
```

3.2.3 Random Forest GridsearchCV

- SMOTE Parameters:
  - Sampling Strategy: Wanted to sample to increase the values for the minority class ("Luxury"). Chose various values for the hyperparameters that support this.
- Random Forest:
  - Decision Tree Parameters Chosen:
    - n_estimators: Chose several amounts across a spectrum. Resources permitted.

    - Criterion: Chose the default and "Entropy"
    - Max Depth: Chose the default and two values across a wide range.
    - Min Samples Leaf: Chose several small values for minimum leaf node
- GridSearchCV Parameters Precion:
    - n_jobs: The local computer running on 64GB of RAM, it was enough to run multiple jobs in parallel. This is part of the reason multiple hyperparameters and values were chosen.

In [88]:
```
#Random Forest parameters and GridSearch instantiation
parameters = {"sm__sampling_strategy":["minority",.2,.6],
                      'rf__n_estimators': [10,50,75],
                      'rf__criterion': ["gini", "entropy"],
                      'rf__max_depth': [None, 5,15,25,50],
                      'rf__min_samples_leaf': [2,5,7]}

gs_rf = GridSearchCV(estimator= pipe_dict["rf"], param_grid=parameters,
n_jobs= 3, refit = "Precision", scoring=scoring, return_train_score= True)
```

In [89]:
```
#Fit GridSearch to Training Data
gs_rf.fit(X_train, y_train)
```

Out[89]:
```
GridSearchCV(estimator=Pipeline(steps=[('ct', StandardScaler()),
                                       ('sm', SMOTE(random_state=42)),
                                       ('rf',
                                        RandomForestClassifier(random_state=4
2))]),
             n_jobs=3,
             param_grid={'rf__criterion': ['gini', 'entropy'],
                         'rf__max_depth': [None, 5, 15, 25, 50],
                         'rf__min_samples_leaf': [2, 5, 7],
                         'rf__n_estimators': [10, 50, 75],
                         'sm__sampling_strategy': ['minority', 0.2, 0.6]},
             refit='Precision', return_train_score=True,
             scoring={'Accuracy': make_scorer(accuracy_score),
                      'F1': make_scorer(f1_score, pos_label=Luxury),
                      'Precision': make_scorer(precision_score, pos_label=Luxur
y),
                      'Rec': make_scorer(recall_score, pos_label=Luxury)})
```

In [90]:
```
#Running the function to print best estimator, scores and results for training a
gridsearch_score(gs_rf)
```

```
Best Estimator: Pipeline(steps=[('ct', StandardScaler()),
                ('sm', SMOTE(random_state=42, sampling_strategy=0.2)),
                ('rf',
                 RandomForestClassifier(criterion='entropy', max_depth=5,
                                        min_samples_leaf=7, n_estimators=10,
                                        random_state=42))])


Best Paremeters: {'rf__criterion': 'entropy', 'rf__max_depth': 5, 'rf__min_sampl
es_leaf': 7, 'rf__n_estimators': 10, 'sm__sampling_strategy': 0.2}


Best Precision Score: 0.557936507936508


Best Train Estimator Score: 1.0
Train Classification Report:
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

       Basic       0.89      0.99      0.94      6551
      Luxury       0.64      0.12      0.21       949

    accuracy                           0.88      7500
   macro avg       0.77      0.56      0.57      7500
weighted avg       0.86      0.88      0.84      7500


Best Test Estimator Score: 1.0
Test Classification Report:
              precision    recall  f1-score   support

       Basic       0.88      0.99      0.93      2184
      Luxury       0.55      0.08      0.13       316

    accuracy                           0.88      2500
   macro avg       0.71      0.53      0.53      2500
weighted avg       0.84      0.88      0.83      2500

------------------------------------------------------------
```

The Best Precision Score comes from the Random Forest model on the "Luxury" class. Best Precision Score: 0.55.

4 Choose Best Estimator

Find the best estimator, parameters and hyperparameters from the GridSearchCV.

In [91]:
```
#Display best estimator
gs_rf.best_estimator_
```

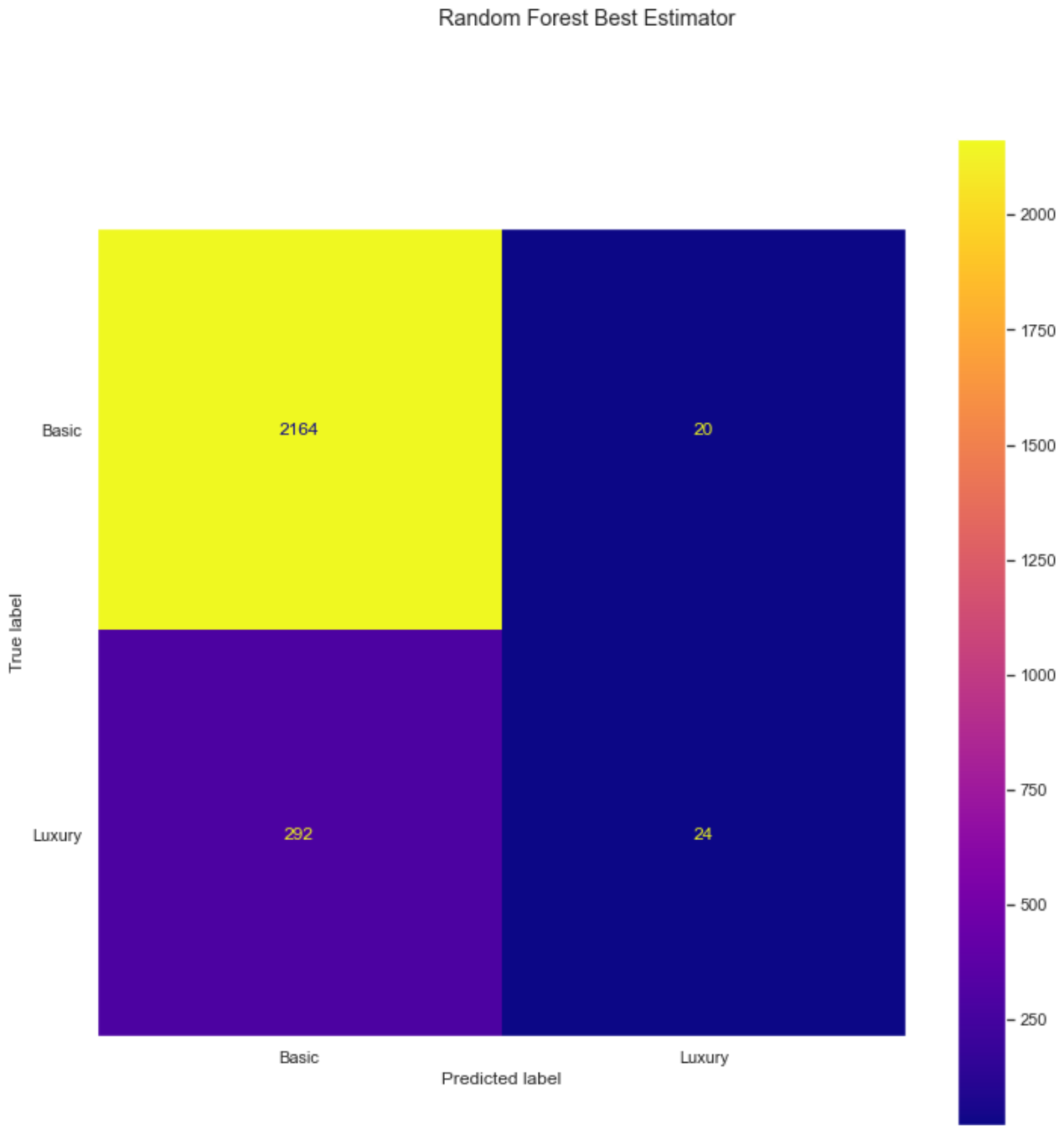
Out[91]:
```
Pipeline(steps=[('ct', StandardScaler()),
                ('sm', SMOTE(random_state=42, sampling_strategy=0.2)),
                ('rf',
                 RandomForestClassifier(criterion='entropy', max_depth=5,
                                        min_samples_leaf=7, n_estimators=10,
                                        random_state=42))])
```

The best estimator includes the SMOTE, Scaler transformation and the Random Forest Classifier with the specific parameters.

In [92]:
```
#Confusion Matrix for Best Estimator
y_gs_rf_pred = gs_rf.best_estimator_.predict(X_test)
confusion_matrix(y_test,y_gs_rf_pred)
```

Out[92]:
```
array([[2164,   20],
       [ 292,   24]])
```

In [93]:
```
#Map Confusion Matrix
fig, ax = plt.subplots(figsize = (12,12), facecolor = "white")
fig.suptitle("Random Forest Best Estimator")
plot_confusion_matrix(gs_rf.best_estimator_, X_test, y_test,ax=ax, cmap="plasma"
plt.grid(False)
fig.savefig("images/BestEst.png")
```



The confusion matrix from the best estimator was dsiplayed.

The TP was a low 24, ~.1%, "Luxury".

The FP was also low 20, ~.1%.

The best estimator was optimized for the best precision score.

In [94]:
```
#Precision Recall on best estimator
y_gs_rf_prob = gs_rf.best_estimator_.predict_proba(X_test)
gs_rf_precision,gs_rf_recall, _ = precision_recall_curve(y_test, y_gs_rf_prob[:,
rf_auc =  auc(gs_rf_recall, gs_rf_precision)
gs_f1 = f1_score( y_test, y_gs_rf_pred, pos_label= "Luxury")
```

In [95]:
```
#Checking the Average Precision Mean and F1 Score
print("Average Metric Score (w/o cv:)")
print ("\n")
print ("Best Estimator Random Forest Average Precision Mean:", round(gs_rf_preci
print ("Best Estimator Random Forest Average F1_Score Mean:", round(gs_f1.mean()
```

```
Average Metric Score (w/o cv:)


Best Estimator Random Forest Average Precision Mean: 0.52
Best Estimator Random Forest Average F1_Score Mean: 0.13
```

The precision score is similar to the best Simple models and models ran with the SMOTE & Scaled transformation from the Logistic Regression and Random Forest models classifier.

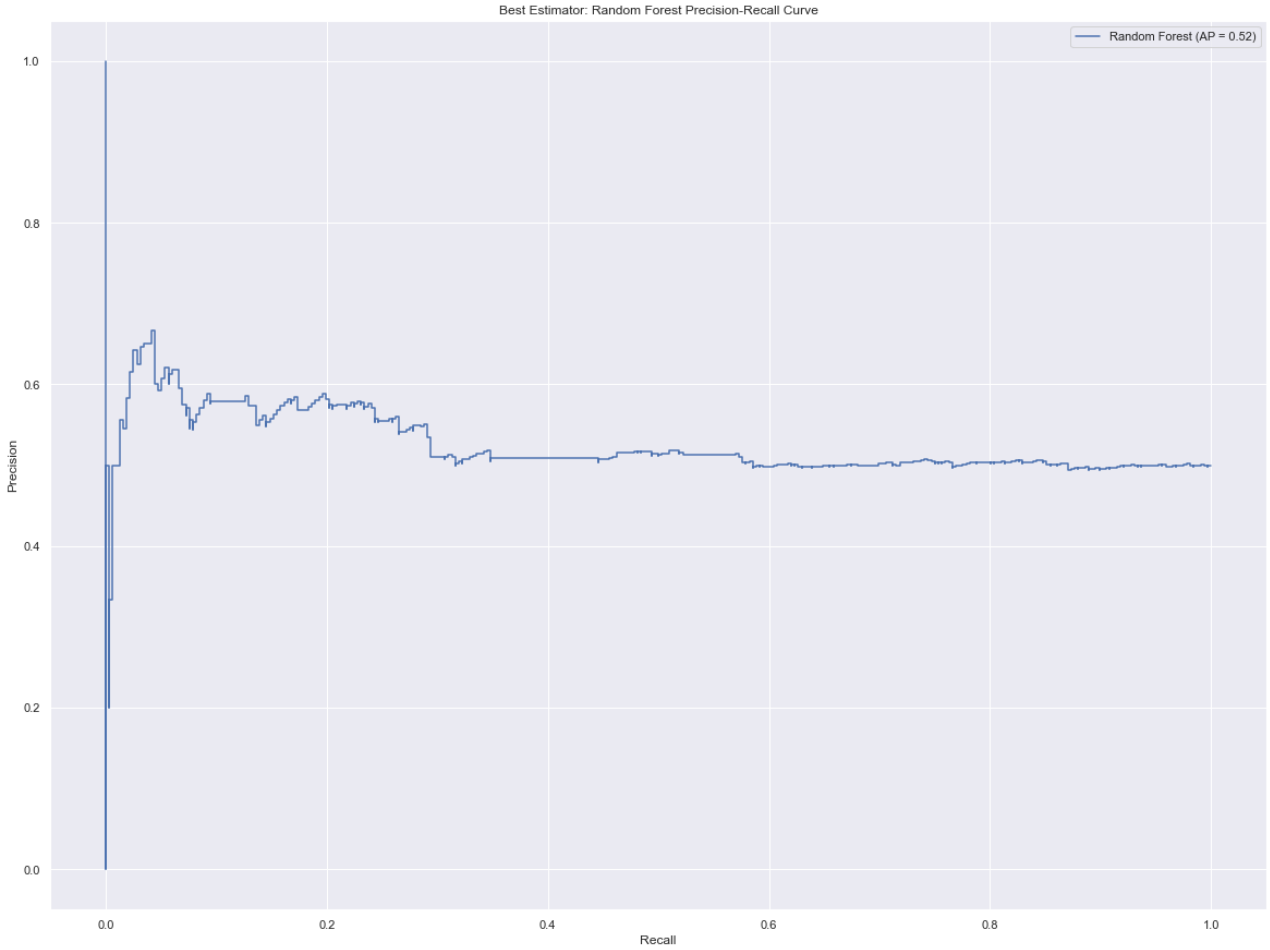
The f1_score drastically dropped around 300-400% compared the best models.

A general review of the statstics would not have made it clear GridSearch "Best Estimator" to be the best model. However, it gives a low TP 24 (.1%), and also gives the lowest FP 20 (.1%) from the all the models. Others have higher TP and FP combined. The stakeholder has a larger "luxury" list to choose from, but introduces more chances of getting "Basic" property accidently. Therefore, this model still meets the expectations fo meeting the orginal goal.

Plotting the Precision Recall Curve for the Best Estimator from the GridSearchCV: Random Forest

In [96]:
```
#Plotting the Precision Recall Curve for the Best Estimator
plot_precision_recall_curve(gs_rf.best_estimator_, X_test, y_test, name = 'Rando
plt.title("Best Estimator: Random Forest Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
#Legend
plt.legend()
```

Out[96]:
```
<matplotlib.legend.Legend at 0x7fc4aa189760>
```



The AUC-PR/Precision Average is about .52. For an imbalanced dataset we are doing 400% better than the initial dummy classifier. This is about 5-10% more than the Random Forest model with the default values.

Feature Importance Selection

Feature Importance shows us what were the most impactful features in the data for the specific classifer. Before the start of modeling the data was correlated (including the target value) and evaluated to see the most correlated features. The is notebook will just get to understand the most important features without further modeling.

In [97]:
```
#Get the best features from the best  model
best_model = gs_rf.best_estimator_
feat_importance = best_model.steps[2][1].feature_importances_
```

In [98]:
```
#Getting the current feature names
columns = X_train.columns
```

Creating a sorted dataframe of the most important features by nama eand importance level. Sorting the data allows for easier plotting.

In [99]:
```
#Creating a dataframe for feature importance
rf_imp_features  = pd.DataFrame({"Features": columns,"Importance_Level": feat_im
```

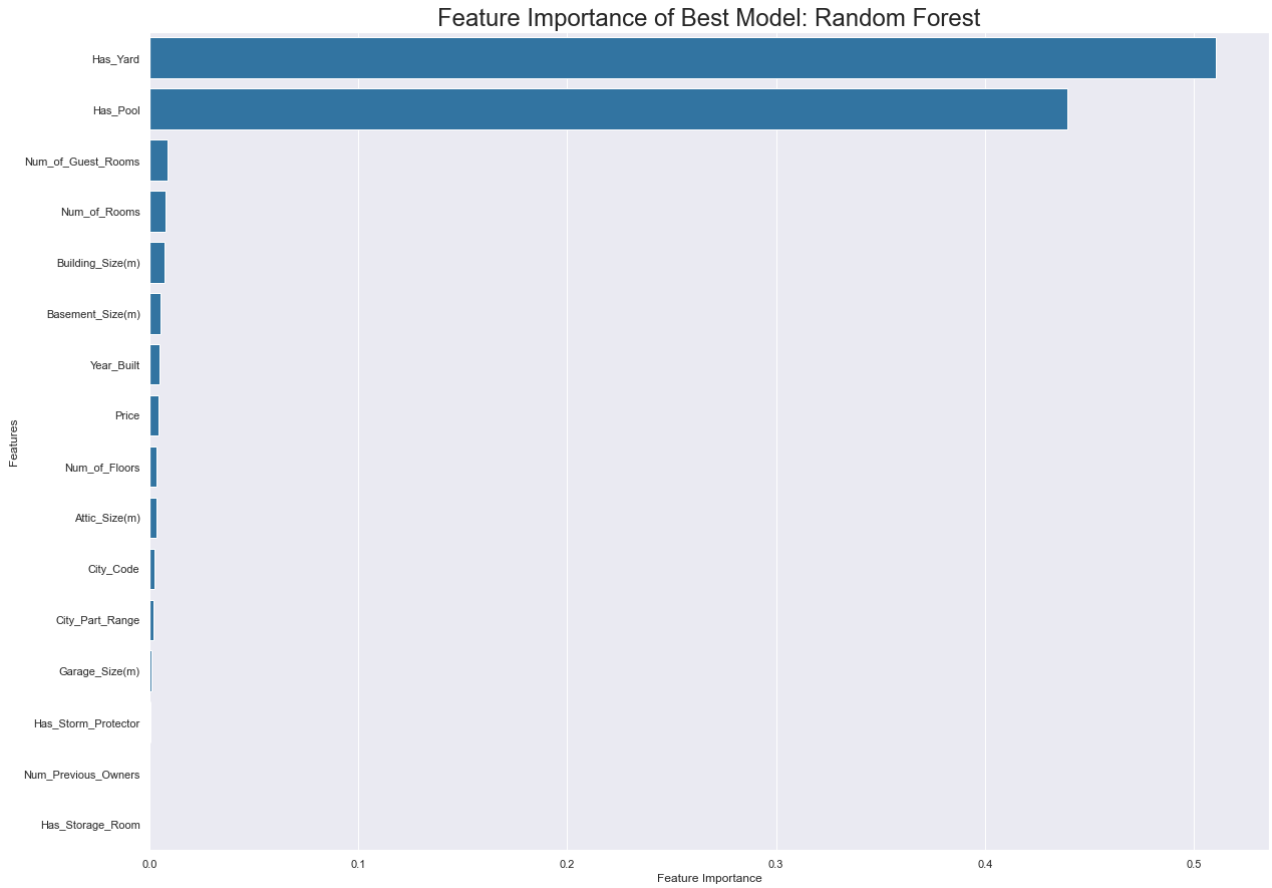
Out[99]:

	Features	Importance_Level
0	Has_Yard	0.510498
1	Has_Pool	0.439397
2	Num_of_Guest_Rooms	0.008769
3	Num_of_Rooms	0.007498
4	Building_Size(m)	0.007118
5	Basement_Size(m)	0.005478
6	Year_Built	0.004575
7	Price	0.004121
8	Num_of_Floors	0.003369
9	Attic_Size(m)	0.003365
10	City_Code	0.002209
11	City_Part_Range	0.001940
12	Garage_Size(m)	0.001148
13	Has_Storm_Protector	0.000316
14	Num_Previous_Owners	0.000198
15	Has_Storage_Room	0.000000

Plotting the feature importance

In [100…

```
#Plotting Importance from Best Model
sns.barplot(x = feat_importance,
            y = columns,
            data = rf_imp_features,
            color = 'tab:blue',
            order=rf_imp_features["Features"],
            orient = 'h')
sns.set(rc={'figure.figsize':(30,30)},font_scale= 2)
plt.title("Feature Importance of Best Model: Random Forest",)
plt.xlabel('Feature Importance')
plt.ylabel('Features')
plt.show()
plt.savefig("images/feat_imp.png")
#change size of labels
```




```
<Figure size 2160x2160 with 0 Axes>
```

Choosing the top 5 features from the sorted feature importance list

In [101…

```
#Top 5 Features
rf_imp_features[0:5]
```

Out[101…

	Features	Importance_Level
0	Has_Yard	0.510498
1	Has_Pool	0.439397
2	Num_of_Guest_Rooms	0.008769
3	Num_of_Rooms	0.007498
4	Building_Size(m)	0.007118

The top two features are still "Has_Yard" and "Has_Pool". These were the same top two that were identified in the previous correlation matrix pre-modeling. "Has_Yard" is first importance here, but was second highest correlated before. This is expected since every "Luxury" property has both a yard and pool.

The third through fifth important features are many factors below the top two. Only "Num_of_Rooms" was among the top 5 of the initially correlated features.

It is interesting Price is ranked 8th in importance.

5 Conclusion

Recommendations

- **Recommend using the "best" model if satisfied with a small list of "Luxury" property listings, which includes a small error chance**
- **Recommend looking into predicted "Basic" and "Luxury" property that has a yard, pool and increased guess rooms.**
- **Recommend not using the chosen model to base final searches on, better for initial. The distribution is too small.**
  - **It minimizes error, but minimizes accurate choices.**
- **Recommend identifying criteria requirements for top features and increasing the chances/error of a "Basic" property being chosen.**
  - **Therefore chosen property isn't just dependent on"Label" but important features that the stakeholder would like**
- **Recommend changing classification from binary of "Basic and Luxury" to multi-classification with more than 3 gradients**
  - **Other than the yard and pool there were was more overlap of the feature's values between the "Basic" and "Luxury" than expected. Adding more classes may help with characterizing the property better.**

Limitations

- **Imbalanced data between "Basic" and "Luxury" limits accurate and useful metrics**
- **With data amount and imbalance, synthetic data was created. It only is a limited representation of the real data.**
- **Limited computing resources, limit parameter and model combination computations**
- **Limited knowledge of stakeholders specific requerements**
- **No knowledge on the condition and grade of the proper. Especially since "Basic" and "Luxury" property has many overlapping similarilys**