MSIN0097: Predictive Analytics

Individual Assignment

Predicting NFT Prices

Word count: 2021

Google Colab link for the Notebook file:
https://colab.research.google.com/drive/1WwqqTN6ppQvAIr9Vjk6qPqvIIwZEQMTy?usp=sharing

Dataset source: https://www.kaggle.com/vepnar/nft-art-dataset

Table of Contents

Report

Notebook Setup

In [3]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [6]:
```
#Essentials
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import pandas as pd
from pandas import Series, DataFrame
from pandas.api.types import CategoricalDtype
pd.options.display.max_columns = None
import numpy as np; np.random.seed(2022)
import random

#Image creation
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import matplotlib.patches as mpatches
from matplotlib import pyplot
import plotly.express as px
import plotly.graph_objects as go

#Image display
import cv2
from google.colab.patches import cv2_imshow
from IPython.display import Image as image
from IPython.display import display

#Preprocessing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.compose import make_column_transformer
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline

#Models
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.svm import SVC
from sklearn import svm
from sklearn.naive_bayes import GaussianNB
from sklearn.base import clone
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import BaggingClassifier
```

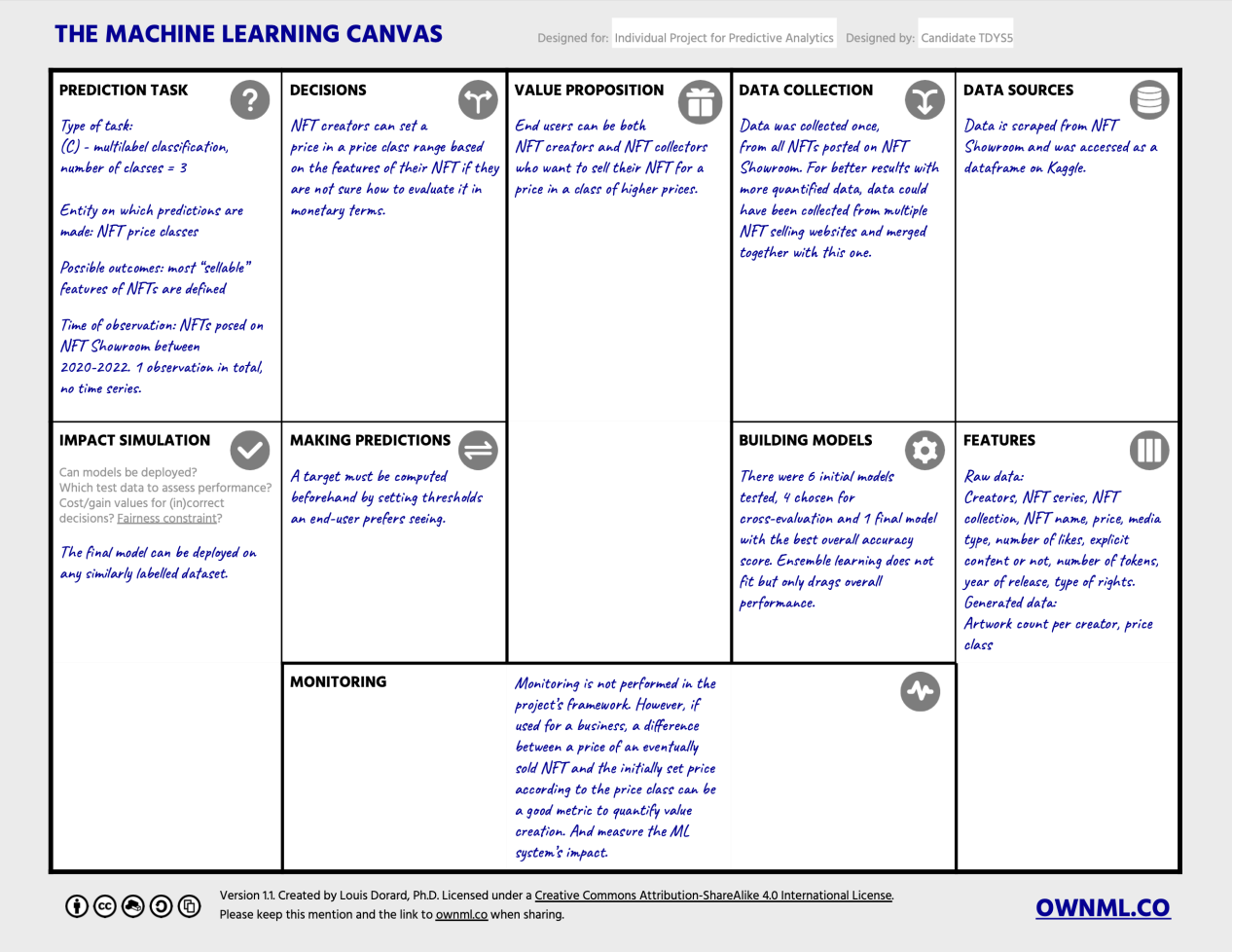
```
#Metrics of accuracy
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score, precision_score, recall_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from numpy import mean
from numpy import std

#Other
import itertools as it
import io
import os
os.sys.path
import sys
import glob
import cv2
import concurrent.futures
from __future__ import print_function
import binascii
import struct
from PIL import Image
import scipy
import scipy.misc
import scipy.cluster
```

Chapter 1: Framing the problem

NFTs are unique cryptographic tokens that exist on a blockchain and cannot be replicated. Most frequently, they represent artwork, being a big opportunity for digital artists in particular. (Sharma, 2021)

The business problem I am trying to solve is the volatility and current unjustification of NFT prices. We will explore patterns between price and other features and predict a price class into which a certain NFT will fall. Although the dataset has continuous variables for price per each NFT, I framed the problem to be a classification problem exactly due to this uncertainty around the newly emerging market. Before setting a price for an item, you need to know the range - which digital artists struggle to identify, based on their artwork. That is the business problem this project tries to solve. More details are seen in Machine Learning Canvas below.

In [ ]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```



The report will cover the following process:

- importing and cleaning the Kaggle data which was aggregated from NFT Showroom;
- EDA before (chapter 2) and after dealing with outliers (chapter 4);
- preprocessing and splitting data;
- building and comparing models;
- fine-tuning hyperparameters on the best one;
- experimenting with ensemble learning.

Chapter 2: Exploratory data analysis

Let's convert the data to a format we can easily manipulate (without changing the data itself yet). We'll lean the dataset leaving only relevant columns for faster perception before visualising relationships between variables we are most interested in.

In [8]:
```
#full_df = pd.read_csv('dataset.csv') #local or Faculty
full_df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Predictive/da
full_df.head()
```

Out[8]:

	title	name	creator	art_series	price	symbol
0	30 min Drawings	Giant Frog	kristyglas	kristyglas_30-min-drawings_giant-frog	50.0	SWAP.HIVE
1	Experimental Video	Biospecimens	juliakponsford	juliakponsford_experimental-video_biospecimens	500.0	SWAP.HIVE
2	Sexy Art	long legs	badsexy	badsexy_sexy-art_long-legs	10.0	SWAP.HIVE
3	Dream World	A Guide in my Dreams	yoslehz	yoslehz_dream-world_a-guide-in-my-dreams	20.0	SWAP.HIVE
4	Dream World	Silent Observer	yoslehz	yoslehz_dream-world_silent-observer	20.0	SWAP.HIVE

In [9]:
```
full_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4189 entries, 0 to 4188
Data columns (total 15 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   title       4189 non-null   object 
 1   name        4189 non-null   object 
 2   creator     4189 non-null   object 
 3   art_series  4189 non-null   object 
 4   price       4189 non-null   float64
 5   symbol      4189 non-null   object 
 6   type        4189 non-null   object 
 7   likes       4189 non-null   int64  
 8   nsfw        4189 non-null   bool   
 9   tokens      4189 non-null   int64  
 10  year        4189 non-null   int64  
 11  rights      4189 non-null   int64  
 12  royalty     4189 non-null   int64  
 13  cid         4189 non-null   object 
 14  path        4189 non-null   object 
dtypes: bool(1), float64(1), int64(5), object(8)
memory usage: 462.4+ KB
```

In [10]:
```
print(full_df.dtypes.astype(str).value_counts())
```

```
object     8
int64      5
float64    1
bool       1
dtype: int64
```

Based on the variable description data below, I found the following relationships most interesting to explore:

- price vs tokens (assumption: 1 token = 1 edition, can be sold or available, i.e. 1 token ≠ 1 sale)
- year vs price
- NFT types by media
- NFT types by explicit content (nsfw)

In [11]:
```
# Iniitial data observation
variable_description_table = pd.read_excel('/content/drive/MyDrive/Colab No
variable_description_table
```

Out[11]:

	variable	type of data	description
0	title	qualitative: nominal	collection name
1	name	qualitative: nominal	artwork name
2	creator	qualitative: nominal	creator's tag on NFT Showroom
3	price	quantitative: continuous	price of NFT in Hives set by creator
4	type	qualitative: nominal	media type (image / gif / video)
5	likes	quantitative: discrete	number of likes on NFT Showroom
6	nsfw	quantitative: binary	not safe for work, i.e. explicit content or not
7	tokens	quantitative: discrete	pegged units to purchase one edition of art
8	year	quantitative: discrete	year of NFT release set by creator
9	rights	quantitative: binary	type of rights: "Private" or "Limited Reproduc...

In [12]:
```
full_df.describe()
```

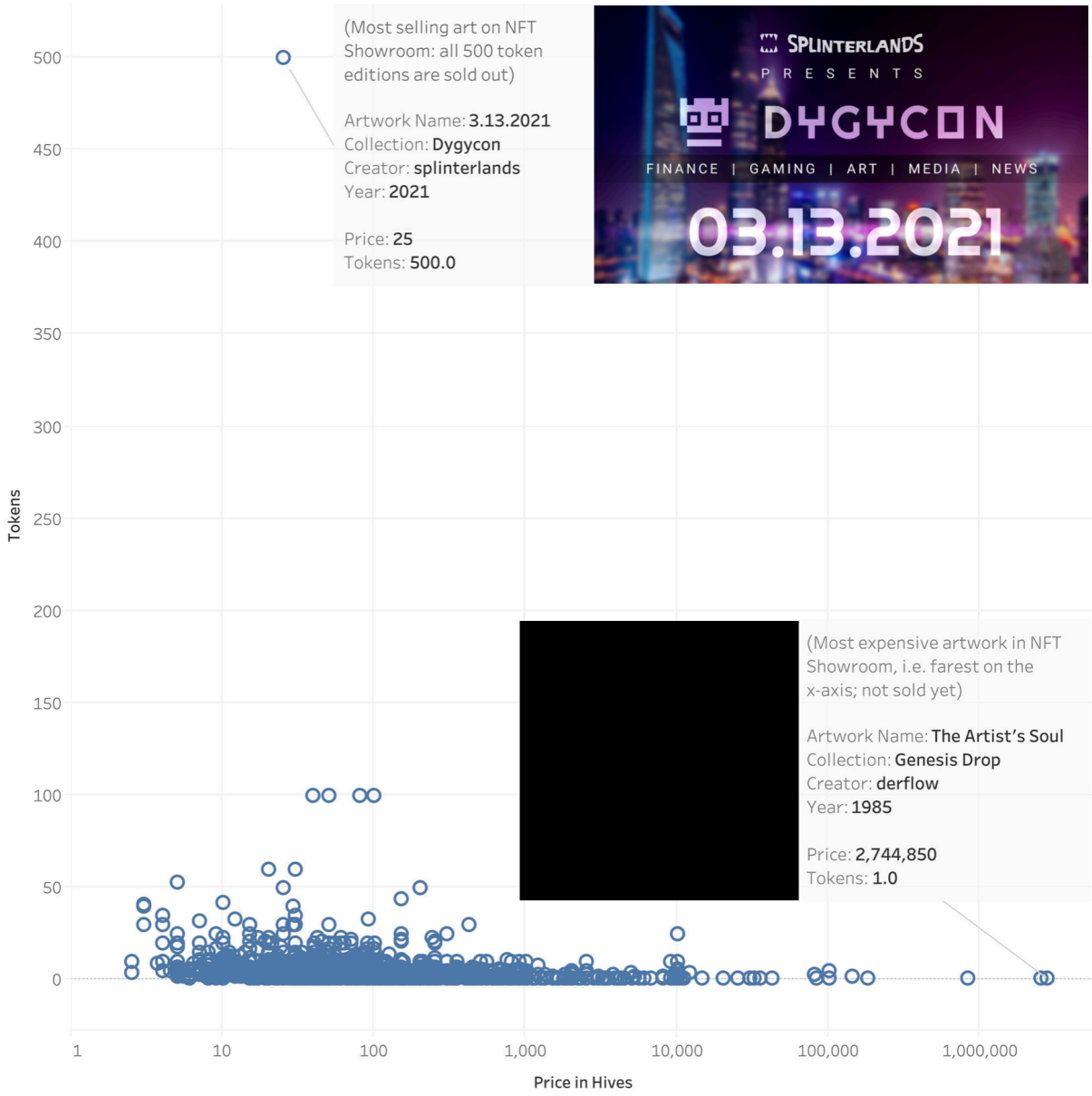
Out[12]:

	price	likes	tokens	year	rights	royalty
count	4.189000e+03	4189.000000	4189.000000	4189.000000	4189.000000	4189.0
mean	2.006903e+03	0.480783	4.162091	2018.009071	1.610169	0.0
std	5.898135e+04	0.935313	9.319016	60.146922	0.920997	0.0
min	2.500000e+00	0.000000	1.000000	1.000000	1.000000	0.0
25%	3.000000e+01	0.000000	1.000000	2020.000000	1.000000	0.0
50%	6.000000e+01	0.000000	3.000000	2020.000000	1.000000	0.0
75%	1.500000e+02	1.000000	5.000000	2021.000000	3.000000	0.0
max	2.744850e+06	10.000000	500.000000	2578.000000	3.000000	0.0

As we can see from the summary table above, there are no null values, as the count throughout all columns is 4189. However, there is a big gap between the minimum and maximum prices, distorting its general distribution pattern. However, we can still see that after manually hiding just 3 outliers in Tableau, prices are close to a Poisson distribution, skewed to the axis' origin.

In [13]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```

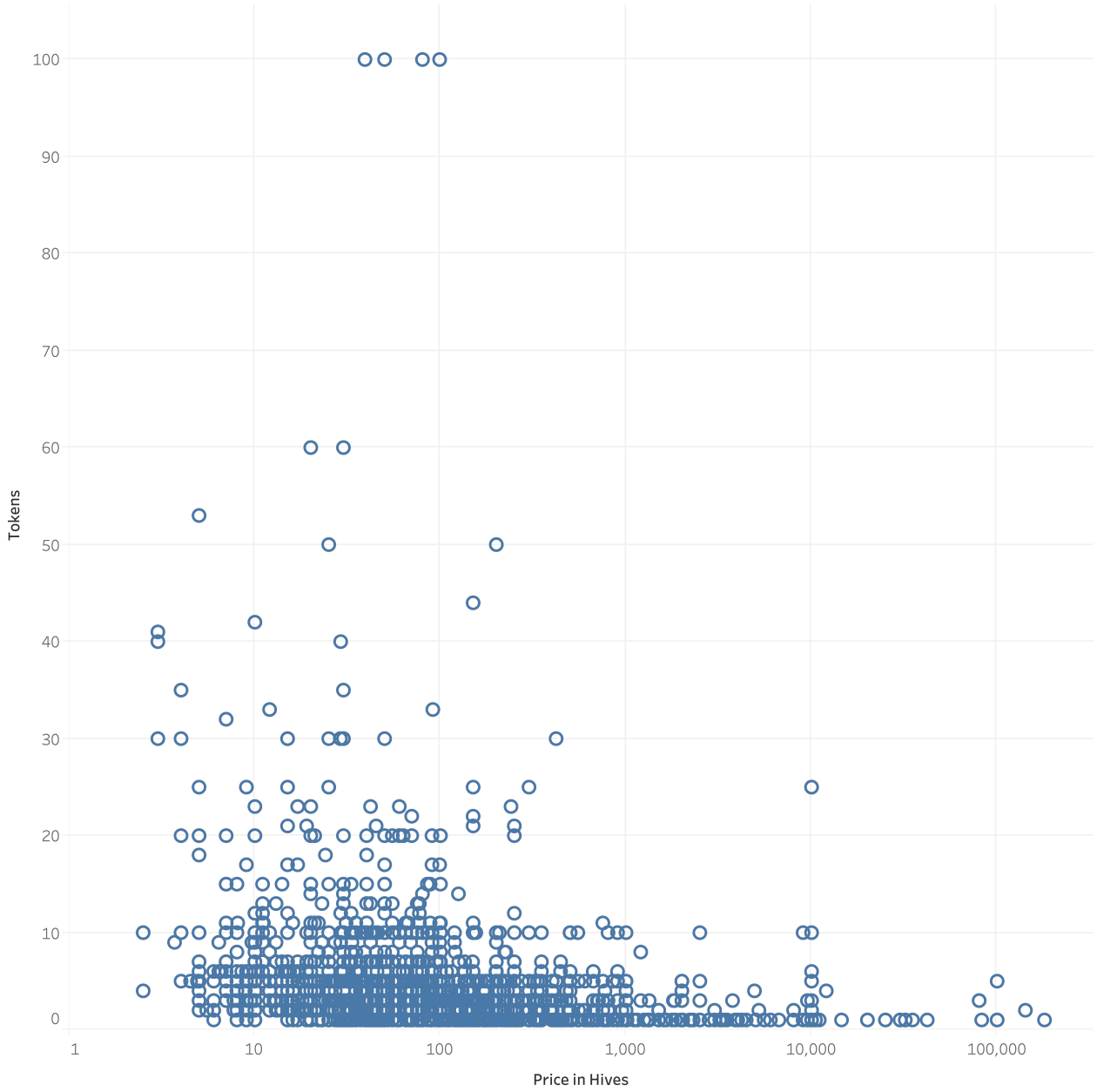
Fig. 1.1: Relationship between tokens and logarithmic scale of prices (all)



Average of Price vs. average of Tokens. Details are shown for various dimensions.

In [14]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```

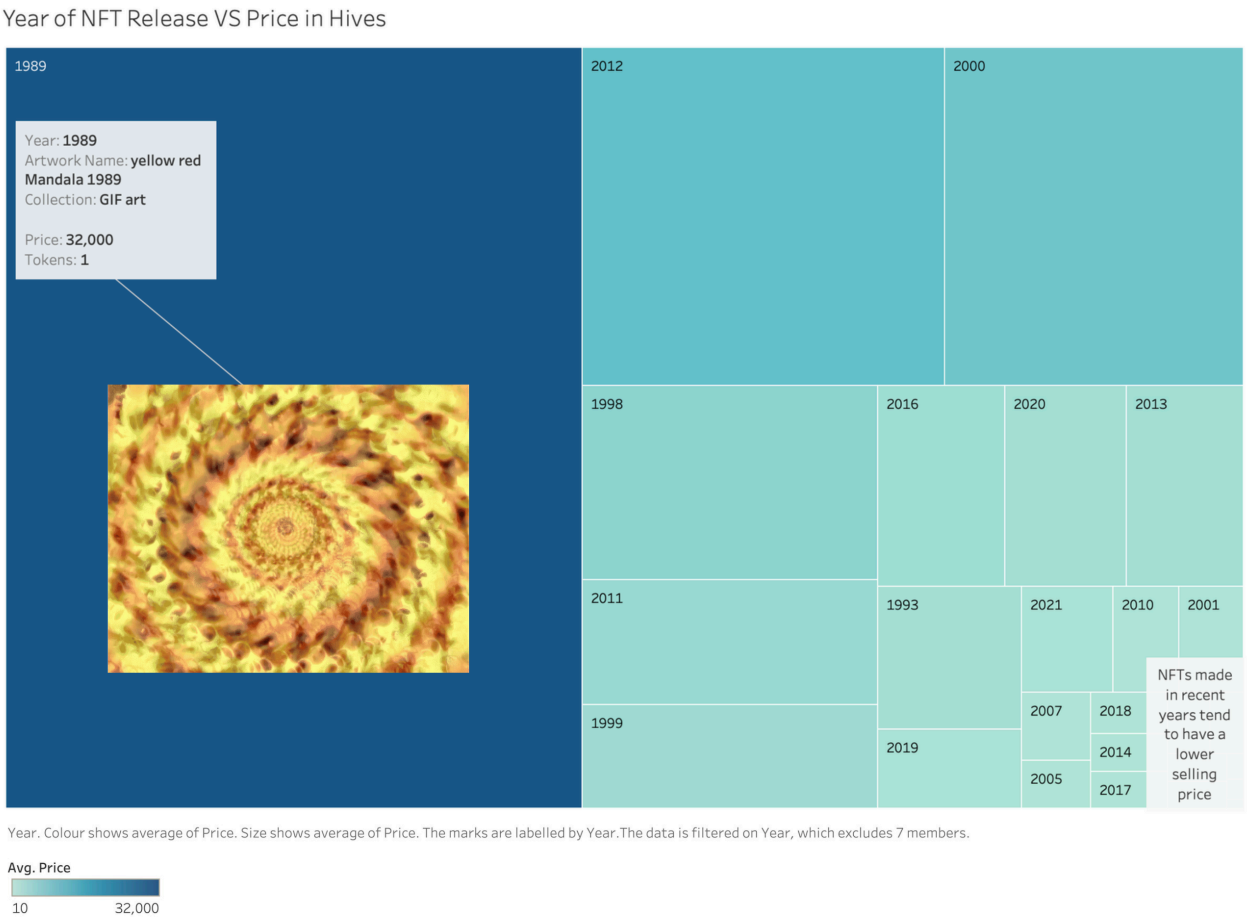
Fig. 1.2: Relationship between tokens and logarithmic scale of prices **(no outliers)**



Average of Price vs. average of Tokens. Details are shown for various dimensions.The view is filtered on Exclusions (Artwork Name,Collection,Creator,Year), which keeps 4,116 members.

A treemap below shows a seemingly reverse relationship between *year* and *price*: the lower the year, the higher price artists set. It mirrors traditional art: the older the painting, the more it is valued by collectors.

In [15]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```



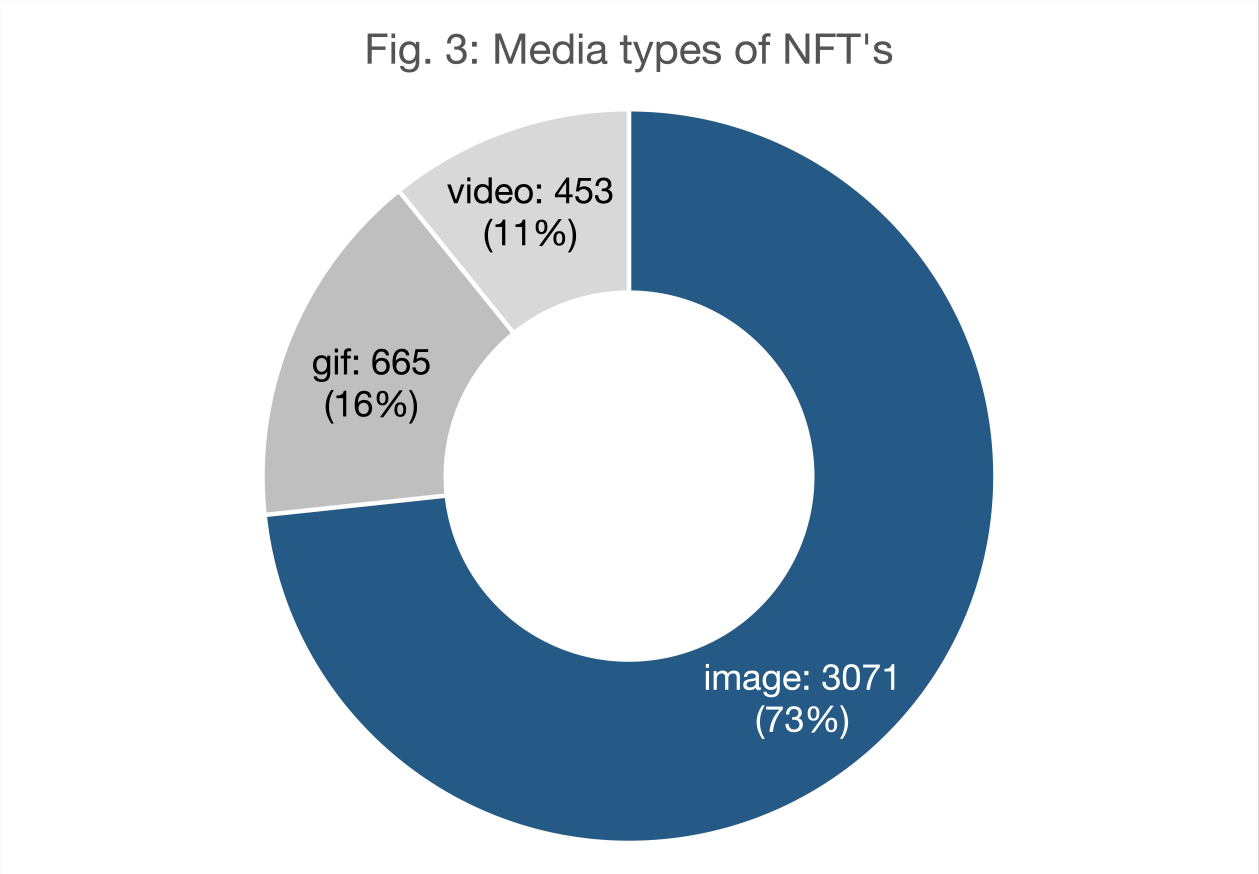
One may think that the trendier an item is, the higher price it should have. In the figure below, numbers represent worldwide search interest relative to the highest point on the chart. A value of 100 is the peak popularity for the term “NFT” and “nft”. A score of 0 means that there was not enough data for this term. (Google Trends, 2021) Interestingly, the first interest peak was in March 2021 - right when the world shifted even more to the digital ecosystem due to Covid-19 restrictions. The global peak of the graph is in January 2022, projected to have a relative fall by the end of February. Hence, higher NFT prices are not much influenced by the recent popularity in the topic.

Screenshot from Google Trends, showing interest in a Google search over time

In [16]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```

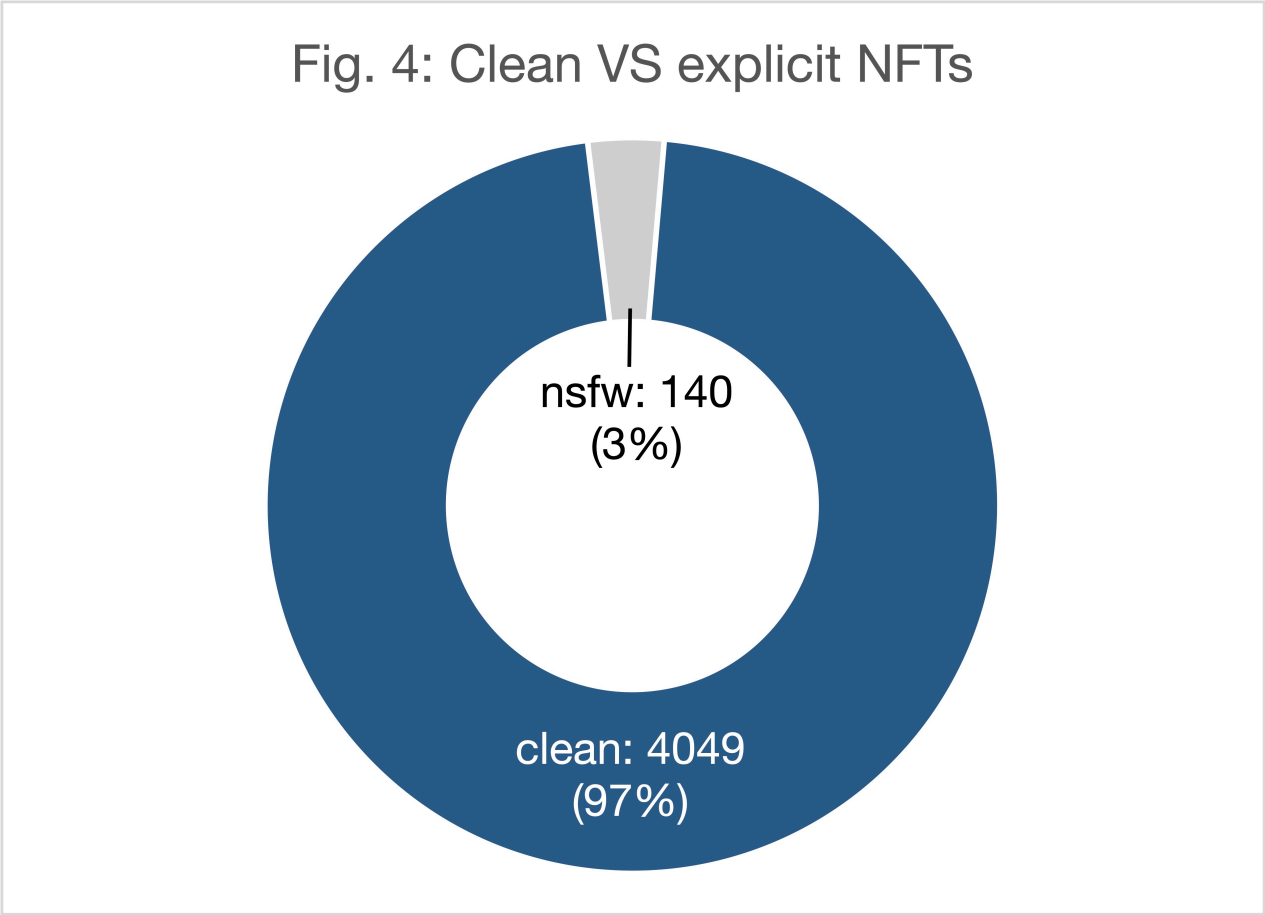
Image accounts for the majority of NFTs' media type.

In [17]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```



97% of NFTs in the dataset are suitable for work (i.e. are not labelled as explicit by the creator). It is unlikely that with such a disproportion between 2 types an algorithm can pick up the relationship between explicit content and price, which one would expect to be proportional, i.e. NFTs with “True” in the “nsfw” column to be of a higher price, rather than one with “False”.

In [18]:
```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```



To conclude this chapter, the main feature properties we can extract are “year” and “tokens”. We will come back to the latter in chapter 4 after excluding outliers, so that we have a more explanatory correlation matrix: without doing so, the matrix is pitch-black, having no correlated variables. Although it fits the real-world inconsistency and vague nature of NFTs at present, for the purpose of the project, we will do data cleaning before exploring feature properties further.

Chapter 3: Data cleaning

3.1 Dataframe sorting

Let’s prepare the dataframe for easier navigation.

In [19]:
```
# Drop categorical columns that are with the same value throughtout the dat
df = full_df.drop(columns = ['cid', 'symbol', 'royalty'])
df.head()
```

Out[19]:

	title	name	creator	art_series	price	type	like
0	30 min Drawings	Giant Frog	kristyglas	kristyglas_30-min-drawings_giant-frog	50.0	PHOTO	2
1	Experimental Video	Biospecimens	juliakponsford	juliakponsford_experimental-video_biospecimens	500.0	VIDEO	0
2	Sexy Art	long legs	badsexy	badsexy_sexy-art_long-legs	10.0	PHOTO	0
3	Dream World	A Guide in my Dreams	yoslehz	yoslehz_dream-world_a-guide-in-my-dreams	20.0	PHOTO	1
4	Dream World	Silent Observer	yoslehz	yoslehz_dream-world_silent-observer	20.0	GIF	0

In [20]:

```
# Quantify columns “type” and “nsfw” for easier data synthesis
df['nsfw'].replace([True, False],[1,0],inplace=True)
df['type'].replace(['PHOTO', 'GIF', 'VIDEO'],[1, 2, 3],inplace=True)
```

In [21]:

```
# Rename columns that may be interchangeably confusing
df = df.rename(columns={'title':'collection', 'name':'artwork_name', 'type'

# Changing order for the first 3 columns
df = df.reindex(['creator', 'artwork_name', 'collection', 'art_series', 'pr

# Sort by ascending creator name to see whether there are more than 1 NFT s
df.sort_values(by=['creator'], inplace = True)
```

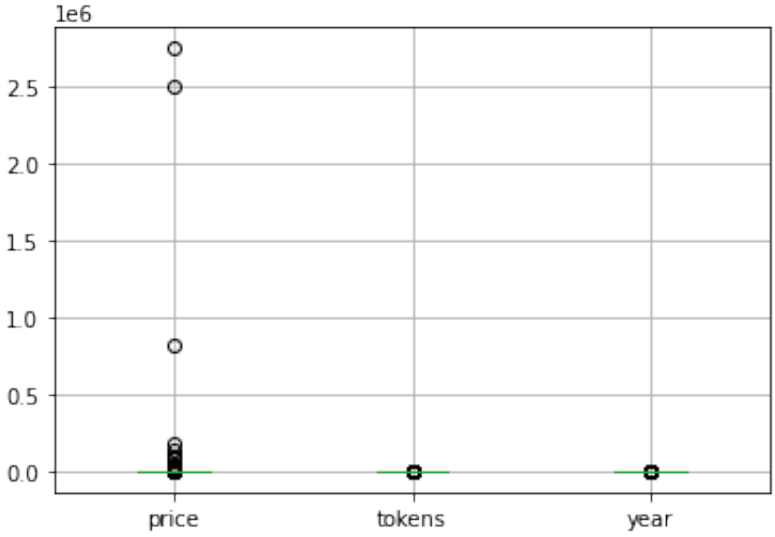
3.2 Dealing with outliers

Now we will have to deal with those outliers found in the previous chapter and substitute them with mean values.

In [22]:

```
numeric_col = ['price', 'tokens', 'year'] # I exclude 'media', 'nsfw', 'rig
df.boxplot(numeric_col)
```

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x7f98a0e45950>



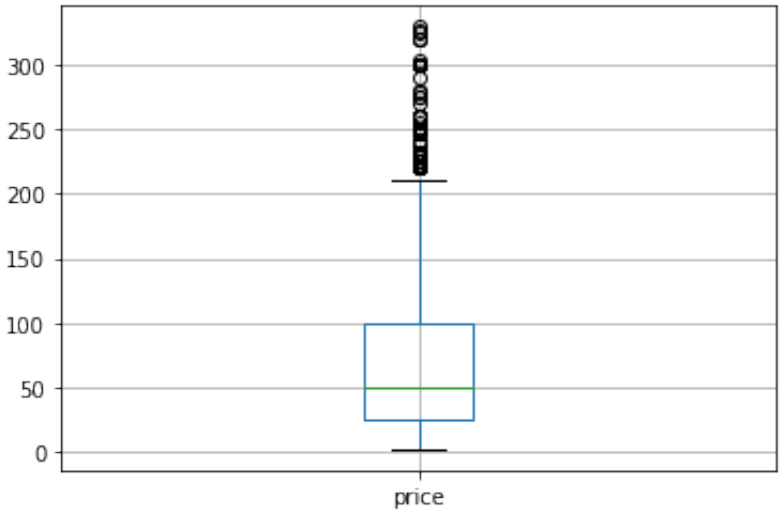
In [23]:
```
for x in ['price']:
    q75,q25 = np.percentile(df.loc[:,x],[75,25])
    intr_qr = q75-q25

    max = q75+(1.5*intr_qr)
    min = q25-(1.5*intr_qr)

    df.loc[df[x] < min,x] = np.nan

    df.loc[df[x] > max,x] = np.nan
```

In [24]:
```
round(df['price'].mean(), 2)
```

Out[24]: 76.63

In [25]:
```
df.boxplot('price')  #green line corresponds to the mean (76.63)
plt.savefig('/content/drive/MyDrive/Colab Notebooks/Predictive/images/price
```

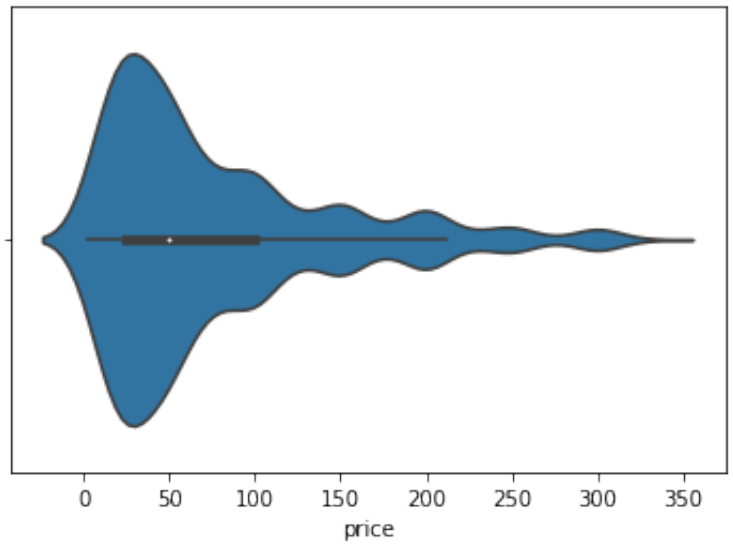
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x7f989e003590>



We have a left-tailed distribution of price, which can also be seen by looking at a violin plot.

In [26]:
```
sns.violinplot(data=df, x='price')
```

Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x7f989df878d0>



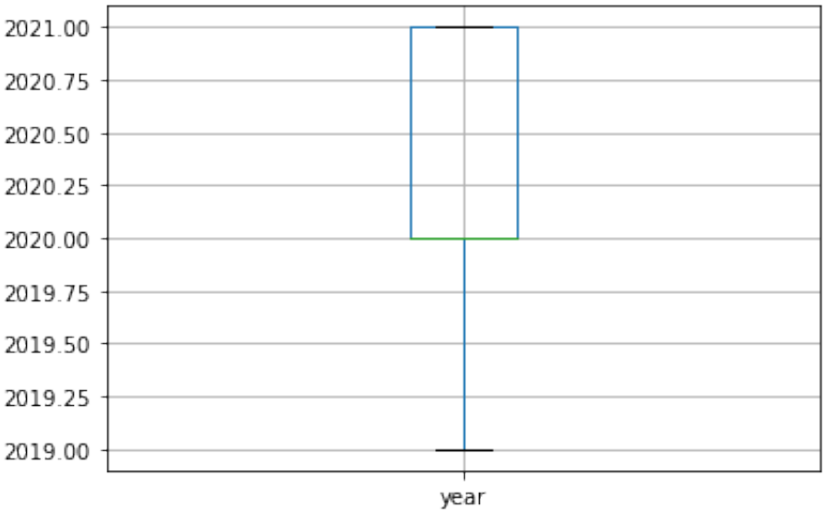
In [27]:
```
for x in ['year']:
    q75,q25 = np.percentile(df.loc[:,x],[75,25])
    intr_qr = q75-q25

    max = q75+(1.5*intr_qr)
    min = q25-(1.5*intr_qr)

    df.loc[df[x] < min,x] = np.nan
    df.loc[df[x] > max,x] = np.nan
```

In [28]:
```
int(df['year'].mean())
```

Out[28]: 2020

In [29]:
```
df.boxplot('year')  #green line corresponds to the mean (2020)
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x7f989bef7550>



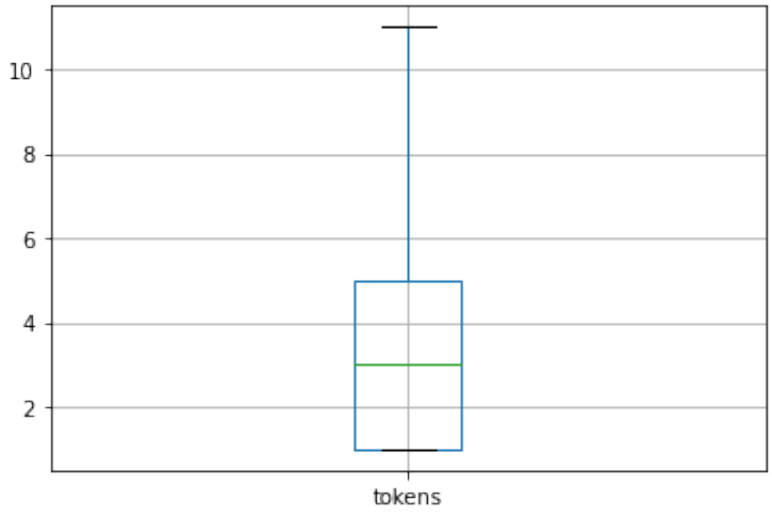
In [30]:
```
for x in ['tokens']:
    q75,q25 = np.percentile(df.loc[:,x],[75,25])
    intr_qr = q75-q25

    max = q75+(1.5*intr_qr)
    min = q25-(1.5*intr_qr)

    df.loc[df[x] < min,x] = np.nan
    df.loc[df[x] > max,x] = np.nan
```

In [31]:
```
int(df['tokens'].mean())
```

Out[31]: 3

In [32]:
```
df.boxplot('tokens') #green line corresponds to the mean (3)
```

Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7f989be79490>



In [33]:
```
df.isnull().sum()
```

Out[33]:
```
creator           0
artwork_name      0
collection        0
art_series        0
price           499
media             0
likes             0
nsfw              0
tokens          142
year            394
rights            0
path              0
dtype: int64
```

In [34]:
```
# using mean to impute the missing values
missing_price = ['price']
for p in missing_price:
    df.loc[df.loc[:,p].isnull(),p]=df.loc[:,p].mean()

missing_tokens = ['tokens']
for t in missing_tokens:
    df.loc[df.loc[:,t].isnull(),t]=int(df.loc[:,t].mean())

missing_year = ['year']
for y in missing_year:
    df.loc[df.loc[:,y].isnull(),y]=int(df.loc[:,y].mean())
```

In [35]:
```
#verify the changes
df.isnull().sum()
```

Out[35]:
```
creator          0
artwork_name     0
collection       0
art_series       0
price            0
media            0
likes            0
nsfw             0
tokens           0
year             0
rights           0
path             0
dtype: int64
```

3.3 Label encoding

Now let's create a new dataframe for all categorical variables only. Since we will not deal with the actual NFTs, we'll drop the “path” column.

In [36]:
```
encoded_df = df.copy()

le = LabelEncoder()
encoded_df['creator'] = le.fit_transform(encoded_df['creator'])
encoded_df['artwork_name'] = le.fit_transform(encoded_df['artwork_name'])
encoded_df['collection'] = le.fit_transform(encoded_df['collection'])
encoded_df['art_series'] = le.fit_transform(encoded_df['art_series'])
encoded_df = encoded_df.drop(columns = ['path'])
encoded_df.head()
```

Out[36]:

	creator	artwork_name	collection	art_series	price	media	likes	nsfw	tokens
1290	0	2923	86	0	76.63127	2	0	0	1.0
1235	0	1316	644	1	100.00000	1	2	0	1.0
873	0	90	935	2	300.00000	1	0	0	1.0
2754	1	978	664	6	58.00000	1	0	0	5.0
3993	1	453	1447	14	110.00000	1	0	0	1.0

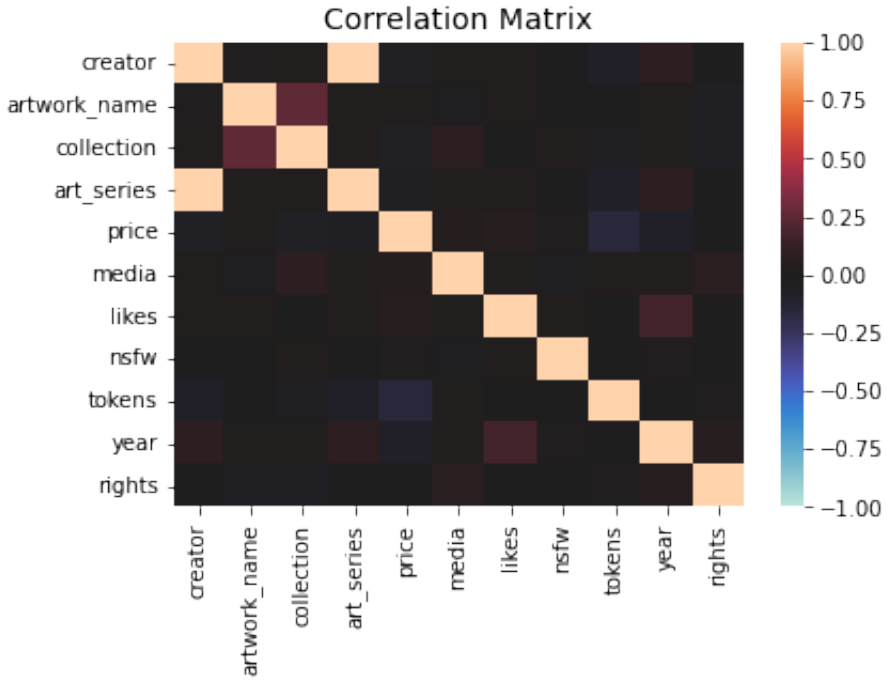
In [37]:
```
# Download clean dataframe for EDA
encoded_df.to_csv('clean_data.csv', index = False)
```

Chapter 4: Correlation between attributes

In [38]:
```
CorrVars = encoded_df.columns
correlation_df = encoded_df[CorrVars].corr().apply(lambda x:round(x,3))
```

In [39]:
```
# Correlation matrix

ax = sns.heatmap(correlation_df,annot=False, vmin=-1, vmax=1, center=0)
ax.set_title("Correlation Matrix", fontsize = 14)

_ = ax.set_yticklabels(
    ax.get_yticklabels(),
    rotation=0,
    horizontalalignment='right')

plt.savefig('correlation_matrix.png')
```

Out[39]: Text(0.5, 1.0, 'Correlation Matrix')

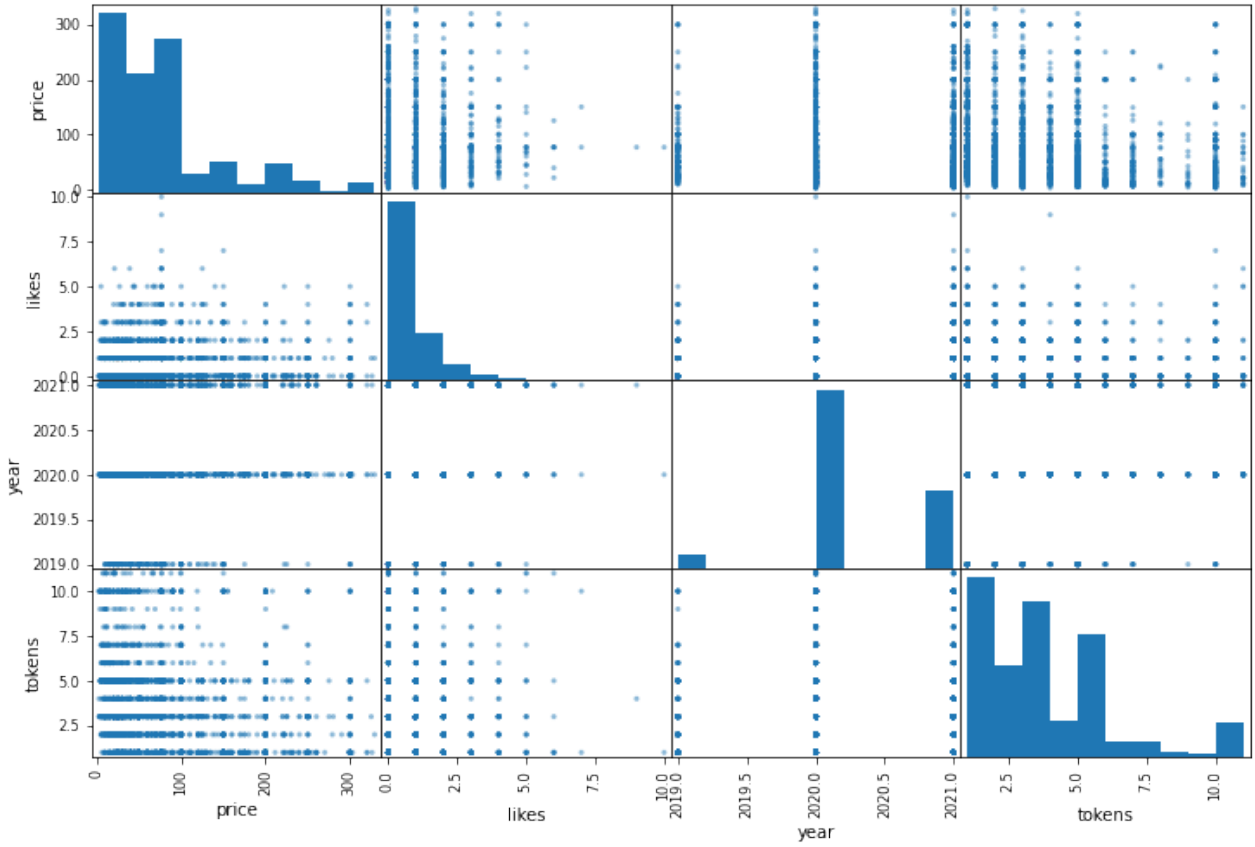


In [40]:
```
correlation_df['price'].sort_values(ascending=False)
```

Out[40]:
```
price           1.000
likes           0.055
media           0.045
artwork_name    0.017
nsfw            0.013
rights         -0.005
art_series     -0.052
creator        -0.055
collection     -0.058
year           -0.073
tokens         -0.150
Name: price, dtype: float64
```

In [41]:
```
from pandas.plotting import scatter_matrix

num_attributes = ['price', 'likes', 'year', 'tokens']
scatter_matrix(df[num_attributes], figsize=(12, 8))

plt.savefig('num_scatter_matrix.png')
```

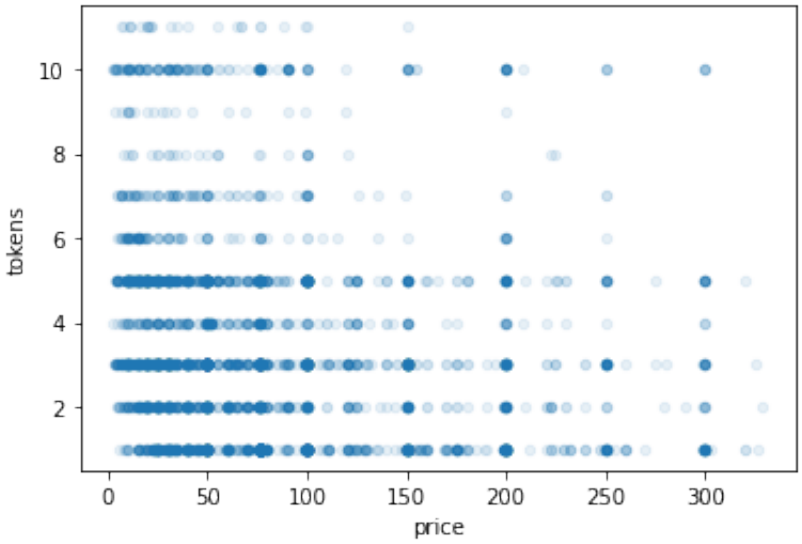
Out[41]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f989bee7ad0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f989bd5e710>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f98994d1d10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f9899491350>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f989944a290>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f98993fd790>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f98993b3d10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f9899371190>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f98993711d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f98993a5790>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f9899319110>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f989e4b5c50>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f989be97910>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f989fe57350>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f989bdc2c50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f989be0b490>]],
      dtype=object)
```



The most promising attribute to predict the NFT value among originally numeric features is tokens, so let's zoom in on their correlation scatterplot.

In [42]:
```
df.plot(kind="scatter", x="price", y="tokens", alpha=0.1)
```

Out[42]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9898fff090>



If we could draw a line over the area of most blue dots, we can notice a flat exponential distribution curve. It also makes sense in real life: intuitively, the fewer tokens (i.e. editions) an NFT has, the more value it should have, as it is rarer - again, just like traditional paintings. Although, reproduction of an NFT is arguably easier than the reproduction of a painting.

Chapter 5: Preprocessing

5.1 Generating artwork counts

Below, I created a new variable that simply tells how many NFTs each artist has. I thought of it as NFT price may be linked to frequency of creator's final artistic output.

In [43]:
```
artwork_counts = {}

for creator in encoded_df.creator:
  if creator in artwork_counts:
    artwork_counts[creator] = artwork_counts[creator] + 1
  else:
    artwork_counts[creator] = 1
```

In [44]:
```
artwork_counts_check = encoded_df['creator'].value_counts().to_dict()
artwork_counts_check == artwork_counts
```

Out[44]: True

In [45]:
```
df_temporary = pd.DataFrame(
    [{"creator": creator, "artwork_counts": artwork_counts} for (creator, a
df_temporary.head()
len(df_temporary)
```

Out[45]:

	creator	artwork_counts
0	0	3
1	1	15
2	2	1
3	3	9
4	4	1

Out[45]: 515

In [46]:
```
encoded_df = encoded_df.merge(df_temporary,how='left', left_on='creator', r
encoded_df.head()
len(encoded_df)
```

Out[46]:

	creator	artwork_name	collection	art_series	price	media	likes	nsfw	tokens	
0	0	2923	86	0	76.63127	2	0	0	1.0	2
1	0	1316	644	1	100.00000	1	2	0	1.0	2(
2	0	90	935	2	300.00000	1	0	0	1.0	2(
3	1	978	664	6	58.00000	1	0	0	5.0	2(
4	1	453	1447	14	110.00000	1	0	0	1.0	2(

Out[46]: 4189

5.2 Feature scaling

"To ensure that the gradient descent moves smoothly towards the minima and that the steps for gradient descent are updated at the same rate for all the features, we scale the data before feeding it to the model." (Analytics Vidhya, 2020) In our case, normalisation works better than standardisation, as outliers have been handled, and data does not seem to follow a normal distribution.

In [47]:
```
from sklearn import preprocessing
d = preprocessing.normalize(encoded_df)
normal_df = pd.DataFrame(d, columns = encoded_df.columns)
normal_df.head()
len(normal_df)
```

Out[47]:

	creator	artwork_name	collection	art_series	price	media	likes	nsfw	
0	0.000000	0.822104	0.024188	0.000000	0.021553	0.000563	0.000000	0.0	0
1	0.000000	0.526948	0.257868	0.000400	0.040042	0.000400	0.000801	0.0	0.
2	0.000000	0.040039	0.415957	0.000890	0.133462	0.000445	0.000000	0.0	0.
3	0.000427	0.417726	0.283610	0.002563	0.024773	0.000427	0.000000	0.0	0
4	0.000396	0.179177	0.572337	0.005537	0.043509	0.000396	0.000000	0.0	0.

Out[47]:
```
4189
```

5.3 Generating price classes

In [48]:
```
all_prices = pd.DataFrame(normal_df, columns = ['price'])
all_prices.head()
```

Out[48]:

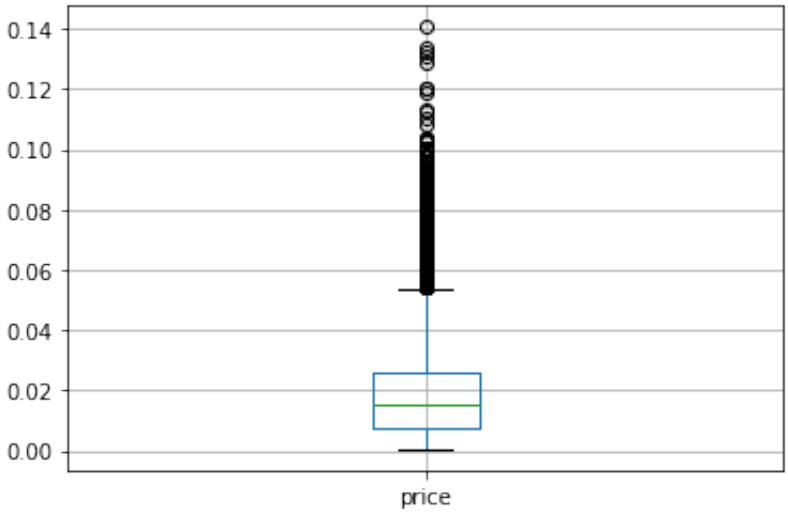
	price
0	0.021553
1	0.040042
2	0.133462
3	0.024773
4	0.043509

In [49]:
```
normal_df['price'].mean() #must be same as mean price of 76.63 in encoded_
```

Out[49]:
```
0.020851333525412057
```

In [50]:
```
normal_df.boxplot('price')  #green line corresponds to the abovementioned m
```

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x7f9898f56f50>



Now we can create 3 price targets based on the mean price as our threshold.

In [51]:
```
price_class = {}
for price in normal_df.price:
  if price < 0.020851333525412057:
    price_class[price] = 0 #cheap NFTs
  elif price > 0.05:
    price_class[price] = 2 #luxury NFTs
  else:
    price_class[price] = 1 #average NFTs
```

In [52]:
```
price_class_df = pd.DataFrame(
    [{"price": price, "price_class": price_class} for (price, price_class)
price_class_df.head()
len(price_class_df)
```

Out[52]:

	price	price_class
0	0.021553	1
1	0.040042	1
2	0.133462	2
3	0.024773	1
4	0.043509	1

Out[52]: 4120

In [53]:
```
average_full_1 = (all_prices >= 0.020851333525412057)
average_full_2 = (all_prices <= 0.05)
average_full_prep = pd.concat([average_full_1, average_full_2], axis = 1) #
average_nft = pd.DataFrame(average_full_prep.all(axis=1))
average_nft.rename(columns={0:"price"},inplace=True)

cheap_nft = (all_prices < 0.020851333525412057)

luxury_nft = (all_prices > 0.05)
```

In [54]:
```
all_labels = pd.concat([cheap_nft, average_nft, luxury_nft], axis = 1)*1 #a
all_labels
```

Out[54]:

	price	price	price
0	0	1	0
1	0	1	0
2	0	0	1
3	0	1	0
4	0	1	0
...	...	...	...
4184	1	0	0
4185	1	0	0
4186	0	1	0
4187	0	1	0
4188	1	0	0

4189 rows × 3 columns

In [55]:
```
prep_classes = all_labels['price'].sum().reset_index()
prep_classes.rename(columns={0:'NFT_count'},inplace=True)
prep_classes.rename(columns={'index': 'category'}, inplace=True)
prep_classes
```

Out[55]:

	category	NFT_count
0	price	2736
1	price	1086
2	price	367

In [56]:
```
price_classes = pd.DataFrame({'categories': ['cheap', 'average', 'luxury'],
                              'NFT_count': [2389, 1593, 207]})
price_classes
```

Out[56]:

	categories	NFT_count
0	cheap	2389
1	average	1593
2	luxury	207

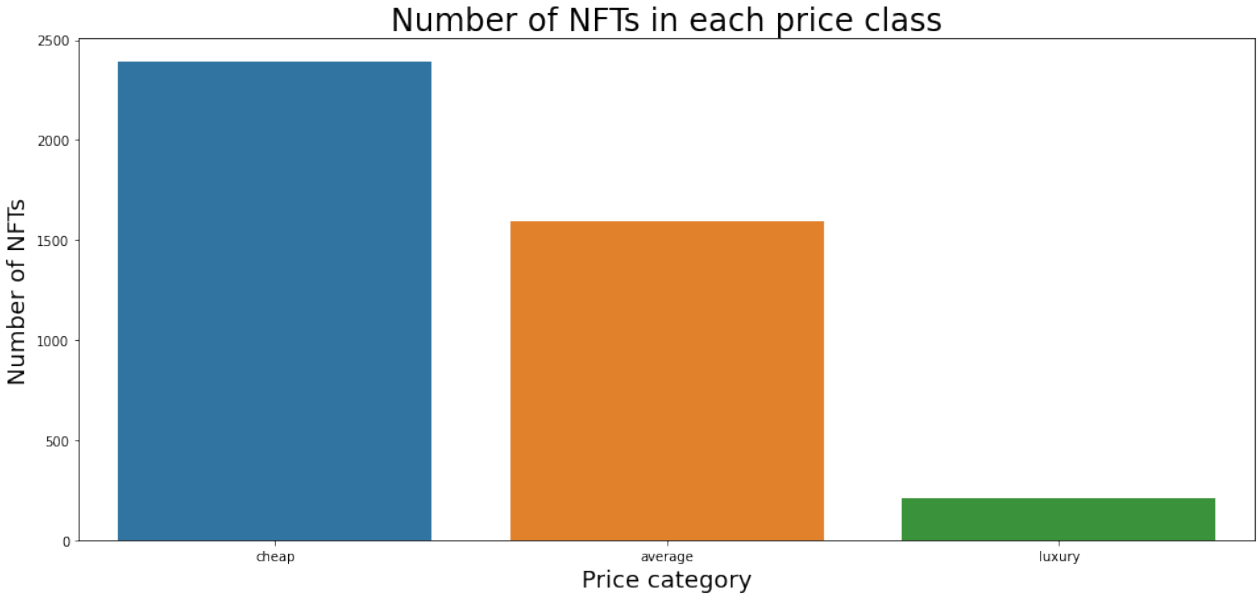
In [57]:
```
# Seaborn Visualisation
plt.figure(figsize=(16,7))
ax = sns.barplot(x = 'categories', y = 'NFT_count', data = price_classes)
plt.title('Number of NFTs in each price class', fontsize=24)
plt.ylabel('Number of NFTs', fontsize=18)
plt.xlabel('Price category', fontsize=18)
```

Out[57]:
```
<Figure size 1152x504 with 0 Axes>
```

Out[57]:
```
Text(0.5, 1.0, 'Number of NFTs in each price class')
```

Out[57]:
```
Text(0, 0.5, 'Number of NFTs')
```

Out[57]:
```
Text(0.5, 0, 'Price category')
```



The figure above reveals 2 things. Firstly, the threshold prices have been split realistically: there is normally an abundance of cheap products and limited supply of luxury ones. However, due to such an uneven number of instances per class, we have an imbalanced classification.

In [58]:
```
full_df = normal_df.merge(price_class_df,how='left', left_on='price', right
full_df.head()
len(full_df)
```

Out[58]:

	creator	artwork_name	collection	art_series	price	media	likes	nsfw	
0	0.000000	0.822104	0.024188	0.000000	0.021553	0.000563	0.000000	0.0	0
1	0.000000	0.526948	0.257868	0.000400	0.040042	0.000400	0.000801	0.0	0.
2	0.000000	0.040039	0.415957	0.000890	0.133462	0.000445	0.000000	0.0	0.
3	0.000427	0.417726	0.283610	0.002563	0.024773	0.000427	0.000000	0.0	0
4	0.000396	0.179177	0.572337	0.005537	0.043509	0.000396	0.000000	0.0	0.

Out[58]: 4189

5.4 Data splitting

Before delving into algorithm training, I will split the dataset into train and test sets to avoid over- and underfitting of our future models.

In [59]:
```
# Split dataset into features and labels
X = full_df[['creator', 'artwork_name', 'collection',
             'art_series', 'media', 'likes', 'nsfw',
             'tokens','year', 'rights', 'artwork_counts']]  # Removed origina
y = full_df['price_class']

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra
print(f"No. of training data: {X_train.shape[0]}")
print(f"No. of training targets: {y_train.shape[0]}")
print(f"No. of testing data: {X_test.shape[0]}")
print(f"No. of testing targets: {y_test.shape[0]}")
```

```
No. of training data: 3351
No. of training targets: 3351
No. of testing data: 838
No. of testing targets: 838
```

Chapter 6: Model selection and training

Let's consider the nature of the problem more closely. With the new column "price class" generated as a multilabel classification's target data with a coherent structure, it is a supervised learning task, predicting the best-fitting price class for the rest of the labelled features.

For precision scores, we will use "weighted" for the "average" parameter, which alters "macro" to account for label imbalance.

6.1 Logistic Regression

In [ ]:
```
#URL: https://www.datacamp.com/community/tutorials/understanding-logistic-r

# Instantiate the model
random.seed(2022)
lr = LogisticRegression()

#Train the model using the training sets
lr.fit(X_train,y_train)

#Predict the response for test dataset
lr_y_pred=lr.predict(X_test)

# Accuracy measures
print("Accuracy score of LR: " + str(round(metrics.accuracy_score(y_test, l
print("Precision score of LR: " + str(round(metrics.precision_score(y_test,
print("Recall score of LR: " + str(round(metrics.recall_score(y_test, lr_y_
print("F1 of LR: " + str(round(metrics.f1_score(y_test, lr_y_pred, average=
```

Out[ ]:
```
LogisticRegression()

Accuracy score of LR: 61.46%
Precision score of LR: 58.39%
Recall score of LR: 61.46%
F1 of LR: 50.78%
```

61.46% accuracy score is a relatively weak start but not too bad. The harmonic mean of the precision and recall, is quite bad though, making the prediction 50/50. Logistic regression performed the worst overall.

6.2 KNeighbors Classifier

In [ ]:
```
#Create KNN Classifier
random.seed(2022)
knn_3 = KNeighborsClassifier(n_neighbors = 3)

#Train the model using the training sets
knn_3.fit(X_train, y_train)

#Predict the response for test dataset
knn_3_y_pred = knn_3.predict(X_test)

# Accuracy measures
print("Accuracy score of KNN-3: " + str(round(metrics.accuracy_score(y_test
print("Precision score of KNN-3: " + str(round(metrics.precision_score(y_te
print("Recall score of KNN-3 " + str(round(metrics.recall_score(y_test, knn
print("F1 of KNN-3: " + str(round(metrics.f1_score(y_test, knn_3_y_pred, av
```

Out[ ]:
```
KNeighborsClassifier(n_neighbors=3)
```

```
Accuracy score of KNN-3: 62.41%
Precision score of KNN-3: 58.57%
Recall score of KNN-3 62.41%
F1 of KNN-3: 59.209999999999994%
```

KNN performed better than logit. For further evaluation, let's create a model for a different number of neighbours.

In [ ]:
```
random.seed(2022)
knn_7 = KNeighborsClassifier(n_neighbors=7)
knn_7.fit(X_train, y_train)
knn_7_y_pred = knn_7.predict(X_test)

print("Accuracy score of KNN-7: " + str(round(metrics.accuracy_score(y_test
print("Precision score of KNN-7: " + str(round(metrics.precision_score(y_te
print("Recall score of KNN-7 " + str(round(metrics.recall_score(y_test, knn
print("F1 of KNN-7: " + str(round(metrics.f1_score(y_test, knn_7_y_pred, av
```

Out[ ]:
```
KNeighborsClassifier(n_neighbors=7)
```

```
Accuracy score of KNN-7: 63.6%
Precision score of KNN-7: 58.48%
Recall score of KNN-7 63.6%
F1 of KNN-7: 58.86%
```

KNN with 7 neighbours performed 1.19% better than with 3 neighbours, however, with a slightly lower precision score and thus, harmonic mean.

6.3 Decision Tree

In [ ]:
```
# Create Decision Tree Classifer object
random.seed(2022)
dtc = DecisionTreeClassifier()

# Train Decision Tree Classifer
dtc = dtc.fit(X_train,y_train)

#Predict the response for test dataset
dtc_y_pred = dtc.predict(X_test)

# Accuracy measures
print("Accuracy score of DTC: " + str(round(metrics.accuracy_score(y_test,
print("Precision score of DTC: " + str(round(metrics.precision_score(y_test
print("Recall score of DTC: " + str(round(metrics.recall_score(y_test, dtc_
print("F1 of DTC: " + str(round(metrics.f1_score(y_test, dtc_y_pred, averag
```

```
Accuracy score of DTC: 62.17%
Precision score of DTC: 61.63999999999999%
Recall score of DTC: 62.17%
F1 of DTC: 61.88%
```

Decision tree performed consistently across all performance metrics, making it the most predictable model.

6.4 Random Forest Classifier

In [ ]:
```
#Create a Gaussian Classifier
random.seed(2022)
rfc = RandomForestClassifier(random_state = 2022)

#Train the model using the training sets y_pred=clf.predict(X_test)
rfc.fit(X_train,y_train)

# prediction on test set
rfc_y_pred=rfc.predict(X_test)

# Accuracy measures
print("Accuracy score of RFC: " + str(round(metrics.accuracy_score(y_test,
print("Precision score of RFC: " + str(round(metrics.precision_score(y_test
print("Recall score of RFC: " + str(round(metrics.recall_score(y_test, rfc_
print("F1 of RFC: " + str(round(metrics.f1_score(y_test, rfc_y_pred, averag
```

Out[ ]:
```
RandomForestClassifier(random_state=2022)
```

```
Accuracy score of RFC: 68.85%
Precision score of RFC: 66.44%
Recall score of RFC: 68.85%
F1 of RFC: 65.51%
```

Random forest performed the best so far, and it is the only model that scored nearly 70% in accuracy.

6.5 XGBoost

In [ ]:
```
#Create a XGBoost model
random.seed(2022)
xgb = XGBClassifier(n_estimators=100, learning_rate=0.05, booster='gbtree',

#Train the model using the training sets y_pred=clf.predict(X_test)
xgb.fit(X_train,y_train)

# Prediction on test set
xgb_y_pred=xgb.predict(X_test)

# Accuracy measures
print("Accuracy score of XGB: " + str(round(metrics.accuracy_score(y_test,
print("Precision score of XGB: " + str(round(metrics.precision_score(y_test
print("Recall score of XGB: " + str(round(metrics.recall_score(y_test, xgb_
print("F1 of XGB: " + str(round(metrics.f1_score(y_test, xgb_y_pred, averag
```

Out[ ]:
```
XGBClassifier(learning_rate=0.05, objective='multi:softprob', random_state=
2022)
Accuracy score of XGB: 66.11%
Precision score of XGB: 60.089999999999996%
Recall score of XGB: 66.11%
F1 of XGB: 58.47%
```

Surprisingly, XGB's harmonic mean is in the 50s, whilst its accuracy and recall scores are good.

6.6 Naive Bayes

In [ ]:
```
#Import Gaussian Naive Bayes model
from sklearn.naive_bayes import GaussianNB

#Create a Gaussian Classifier
gnb = GaussianNB()

#Train the model using the training sets
gnb.fit(X_train, y_train)

#Predict the response for test dataset
gnb_y_pred = gnb.predict(X_test)

# Accuracy measures
print("Accuracy score of GNB: " + str(round(metrics.accuracy_score(y_test,
print("Precision score of GNB: " + str(round(metrics.precision_score(y_test
print("Recall score of GNB: " + str(round(metrics.recall_score(y_test, gnb_
print("F1 of GNB: " + str(round(metrics.f1_score(y_test, gnb_y_pred, averag
```

Out[ ]:
```
GaussianNB()
```

```
Accuracy score of GNB: 57.52%
Precision score of GNB: 52.5%
Recall score of GNB: 57.52%
F1 of GNB: 53.82%
```

Naive Bayes is one of the weaker models which we will not consider further.

6.7 Models comparison

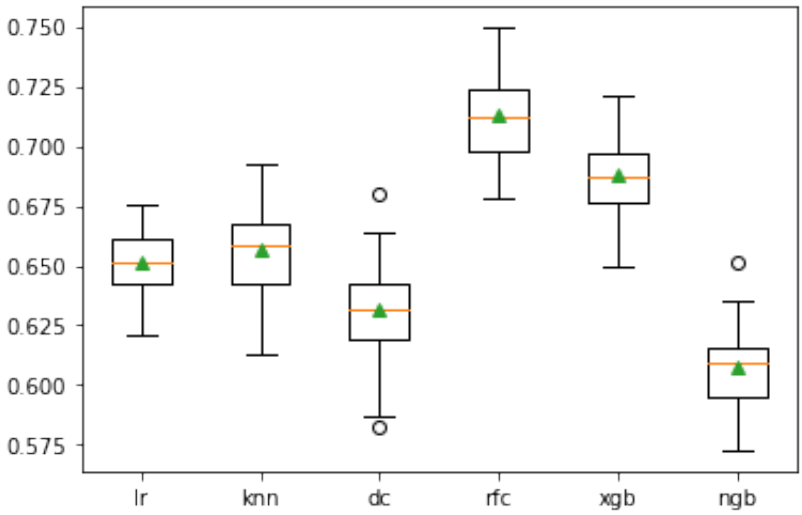
Finally, we have 6 different algorithms that perform okay, in different ways on this dataset.

In [ ]:

```
#URL: https://machinelearningmastery.com/stacking-ensemble-machine-learning

# get a list of models to evaluate
def get_models():
        models = dict()
        models['lr'] = LogisticRegression()
        models['knn'] = KNeighborsClassifier()
        models['dc'] = DecisionTreeClassifier()
        models['rfc'] = RandomForestClassifier()
        models['xgb'] = XGBClassifier()
        models['ngb'] = GaussianNB()
        return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_
        return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
        scores = evaluate_model(model, X, y)
        results.append(scores)
        names.append(name)
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>lr 0.651 (0.012)
>knn 0.656 (0.017)
>dc 0.632 (0.021)
>rfc 0.713 (0.019)
>xgb 0.688 (0.016)
>ngb 0.607 (0.017)
```

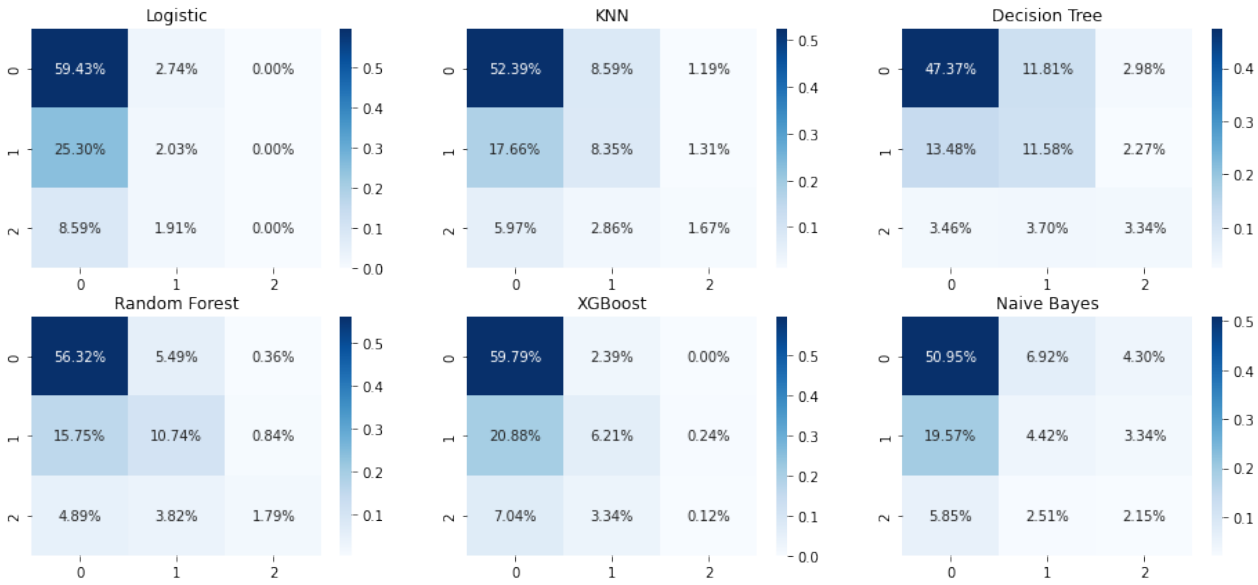
Out[ ]:
```
{'boxes': [<matplotlib.lines.Line2D at 0x7f0c45e12190>,
  <matplotlib.lines.Line2D at 0x7f0c45e1dcd0>,
  <matplotlib.lines.Line2D at 0x7f0c45db0790>,
  <matplotlib.lines.Line2D at 0x7f0c45dc8310>,
  <matplotlib.lines.Line2D at 0x7f0c45e5ef10>,
  <matplotlib.lines.Line2D at 0x7f0c45de75d0>],
 'caps': [<matplotlib.lines.Line2D at 0x7f0c45e12e50>,
  <matplotlib.lines.Line2D at 0x7f0c45e167d0>,
  <matplotlib.lines.Line2D at 0x7f0c45e23dd0>,
  <matplotlib.lines.Line2D at 0x7f0c45e23f50>,
  <matplotlib.lines.Line2D at 0x7f0c45db7850>,
  <matplotlib.lines.Line2D at 0x7f0c45db7dd0>,
  <matplotlib.lines.Line2D at 0x7f0c45dc8f10>,
  <matplotlib.lines.Line2D at 0x7f0c45dd0890>,
  <matplotlib.lines.Line2D at 0x7f0c45e4fc10>,
  <matplotlib.lines.Line2D at 0x7f0c45dd0c50>,
  <matplotlib.lines.Line2D at 0x7f0c45d2e650>,
  <matplotlib.lines.Line2D at 0x7f0c45d2eb90>],
 'fliers': [<matplotlib.lines.Line2D at 0x7f0c45e16ed0>,
  <matplotlib.lines.Line2D at 0x7f0c45e2b990>,
  <matplotlib.lines.Line2D at 0x7f0c45dc0dd0>,
  <matplotlib.lines.Line2D at 0x7f0c467c9bd0>,
  <matplotlib.lines.Line2D at 0x7f0c45ddf710>,
  <matplotlib.lines.Line2D at 0x7f0c45d37b90>],
 'means': [<matplotlib.lines.Line2D at 0x7f0c45e1d2d0>,
  <matplotlib.lines.Line2D at 0x7f0c45e2bd50>,
  <matplotlib.lines.Line2D at 0x7f0c45dc0890>,
  <matplotlib.lines.Line2D at 0x7f0c45e3e7d0>,
  <matplotlib.lines.Line2D at 0x7f0c45ddfb10>,
  <matplotlib.lines.Line2D at 0x7f0c45d37650>],
 'medians': [<matplotlib.lines.Line2D at 0x7f0c45e16d50>,
  <matplotlib.lines.Line2D at 0x7f0c45e2b810>,
  <matplotlib.lines.Line2D at 0x7f0c45dc0350>,
  <matplotlib.lines.Line2D at 0x7f0c45d96650>,
  <matplotlib.lines.Line2D at 0x7f0c45ddf590>,
  <matplotlib.lines.Line2D at 0x7f0c45d37110>],
 'whiskers': [<matplotlib.lines.Line2D at 0x7f0c45e127d0>,
  <matplotlib.lines.Line2D at 0x7f0c45e12d10>,
  <matplotlib.lines.Line2D at 0x7f0c45e1de50>,
  <matplotlib.lines.Line2D at 0x7f0c45e237d0>,
  <matplotlib.lines.Line2D at 0x7f0c45db0d50>,
  <matplotlib.lines.Line2D at 0x7f0c45db0e90>,
  <matplotlib.lines.Line2D at 0x7f0c45dc8890>,
  <matplotlib.lines.Line2D at 0x7f0c45dc8dd0>,
  <matplotlib.lines.Line2D at 0x7f0c45e5e4d0>,
  <matplotlib.lines.Line2D at 0x7f0c45e54050>,
  <matplotlib.lines.Line2D at 0x7f0c45de7b90>,
  <matplotlib.lines.Line2D at 0x7f0c45de7cd0>]}
```



According to F1 scores, the top 3 models are random forest, decision tree, and KNN with 3 neighbours. Now, let's compare all those models and their confusion rates, first rerunning KNN-3 cell.

In [ ]:

```
plt.figure(figsize=(16,7))

plt.subplot(2,3,1)
sns.heatmap(confusion_matrix(y_test,lr_y_pred)/np.sum(confusion_matrix(y_te
plt.title("Logistic")

plt.subplot(2,3,2)
sns.heatmap(confusion_matrix(y_test,knn_3_y_pred)/np.sum(confusion_matrix(y
plt.title("KNN")

plt.subplot(2,3,3)
sns.heatmap(confusion_matrix(y_test,dtc_y_pred)/np.sum(confusion_matrix(y_t
plt.title("Decision Tree")

plt.subplot(2,3,4)
sns.heatmap(confusion_matrix(y_test,rfc_y_pred)/np.sum(confusion_matrix(y_t
plt.title("Random Forest")

plt.subplot(2,3,5)
sns.heatmap(confusion_matrix(y_test,xgb_y_pred)/np.sum(confusion_matrix(y_t
plt.title("XGBoost")

plt.subplot(2,3,6)
sns.heatmap(confusion_matrix(y_test,gnb_y_pred)/np.sum(confusion_matrix(y_t
plt.title("Naive Bayes")

plt.show();
```



From the confusion matrix below, we can see that the coverage of error made by the models identified different biases in different models. We want the diagonal from the top left corner to the bottom right corner to be as high as possible, as these are the true values.

In [ ]:
```
rfc_cv = cross_val_score(rfc, X_train, y_train, cv=10, scoring="accuracy")
print('RFC accuracy score is ' + str(round(rfc_cv[1],3)*100) + '%')

knn_cv = cross_val_score(knn_3, X_train, y_train, cv=10, scoring="accuracy"
print('KNN accuracy score is ' + str(round(knn_cv[1],3)*100) + '%')

dtc_cv = cross_val_score(dtc, X_train, y_train, cv=10, scoring="accuracy")
print('DTC accuracy score is ' + str(round(dtc_cv[1],3)*100) + '%')

xgb_cv = cross_val_score(xgb, X_train, y_train, cv=10, scoring="accuracy")
print('XGB accuracy score is ' + str(round(xgb_cv[1],3)*100) + '%')
```

```
RFC accuracy score is 72.2%
KNN accuracy score is 64.2%
DTC accuracy score is 62.7%
XGB accuracy score is 69.89999999999999%
```

Random Forest is still the best.

Chapter 7: Fine-tuning

7.1 Grid search

Let's instantiate the random search and fine-tune hyperparameters towards a higher recall of the best model - random forest - using 5-fold cross validation, searching across 100 different combinations.

In [ ]:
```
from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(rfc.get_params())
```

```
Parameters currently in use:

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 2022,
 'verbose': 0,
 'warm_start': False}
```

In [ ]:
```
# URL: https://towardsdatascience.com/hyperparameter-tuning-the-random-fore

from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 100, stop = 2000, num =
# Number of features to consider at every split
max_features = ['auto','sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]

# Create the random grid
random_grid = {'n_estimators': n_estimators,
                       'max_features': max_features,
                       'max_depth': max_depth,
                       'min_samples_split': min_samples_split,
                        'min_samples_leaf': min_samples_leaf,
                        'bootstrap': bootstrap}
pprint(random_grid)
```

```
{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
```

```
 'min_samples_split': [2, 5, 10],
 'n_estimators': [100,
                  119,
                  138,
                  157,
                  176,
                  195,
                  215,
                  234,
                  253,
                  272,
                  291,
                  311,
                  330,
                  349,
                  368,
                  387,
                  407,
                  426,
                  445,
                  464,
                  483,
                  503,
                  522,
                  541,
                  560,
                  579,
                  598,
                  618,
                  637,
                  656,
                  675,
                  694,
                  714,
                  733,
                  752,
                  771,
                  790,
                  810,
                  829,
                  848,
                  867,
                  886,
                  906,
                  925,
                  944,
                  963,
                  982,
                  1002,
                  1021,
                  1040,
                  1059,
                  1078,
                  1097,
                  1117,
                  1136,
                  1155,
                  1174,
                  1193,
                  1213,
                  1232,
```

```
                    1251,
                    1270,
                    1289,
                    1309,
                    1328,
                    1347,
                    1366,
                    1385,
                    1405,
                    1424,
                    1443,
                    1462,
                    1481,
                    1501,
                    1520,
                    1539,
                    1558,
                    1577,
                    1596,
                    1616,
                    1635,
                    1654,
                    1673,
                    1692,
                    1712,
                    1731,
                    1750,
                    1769,
                    1788,
                    1808,
                    1827,
                    1846,
                    1865,
                    1884,
                    1904,
                    1923,
                    1942,
                    1961,
                    1980,
                    2000]}
```

In [ ]:

```
rfc_tuned = RandomizedSearchCV(estimator = rfc,
                               param_distributions = random_grid,
                               n_iter = 100,
                               cv = 6,
                               verbose = 2,
                               random_state = 2022,
                               n_jobs = -1)
# Fit the random search model
rfc_tuned.fit(X_train, y_train)
```

```
Fitting 6 folds for each of 100 candidates, totalling 600 fits
```

Out[ ]:
```
RandomizedSearchCV(cv=6, estimator=RandomForestClassifier(random_state=2022
),
                   n_iter=100, n_jobs=-1,
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [10, 20, 30, 40, 50, 6
0,
                                                      70, 80, 90, 100, 110,
                                                      None],
                                        'max_features': ['auto', 'sqrt'],
                                        'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [100, 119, 138, 157
,
                                                         176, 195, 215, 234
,
                                                         253, 272, 291, 311
,
                                                         330, 349, 368, 387
,
                                                         407, 426, 445, 464
,
                                                         483, 503, 522, 541
,
                                                         560, 579, 598, 618
,
                                                         637, 656, ...]},
                   random_state=2022, verbose=2)
```

In [ ]:
```
# Showing the best parameteres after randomized grid search
rfc_tuned.best_params_
```

Out[ ]:
```
{'bootstrap': False,
 'max_depth': 110,
 'max_features': 'sqrt',
 'min_samples_leaf': 4,
 'min_samples_split': 2,
 'n_estimators': 234}
```

Now, let's evaluate random search.

In [ ]:
```
# Base model results
random.seed(2022)
rfc = RandomForestClassifier(random_state = 2022)
rfc.fit(X_train,y_train)
base_y_pred = rfc.predict(X_test)
base_accuracy = round(metrics.accuracy_score(y_test, base_y_pred), 3)*100
print('Accuracy of base RFC is ' + str(base_accuracy)+'%')
```

Out[ ]:
```
RandomForestClassifier(random_state=2022)
```

```
Accuracy of base RFC is 68.89999999999999%
```

In [ ]:
```
# Tuned model results
rfc_tuned = rfc_tuned.best_estimator_
rfc_tuned.fit(X_train, y_train)
tuned_y_pred = rfc_tuned.predict(X_test)
tuned_accuracy = round(metrics.accuracy_score(y_test, tuned_y_pred), 3)*100
print('Accuracy of tuned RFC is ' + str(tuned_accuracy)+'%')
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, max_depth=110, max_features='sqrt',
                       min_samples_leaf=4, n_estimators=234, random_state=2
022)
Accuracy of tuned RFC is 69.8%
```

In [ ]:
```
print('Improvement of {:0.2f}%'.format(100 * (tuned_accuracy - base_accurac
```

```
Improvement of 1.31%
```

Although model performance improved only by 1.31% suggesting there might not be much improvement left at all, one more grid search iteration is applied to check if the performance can be enhanced further.

In [ ]:
```
# GridSearch with Cross Validation
from sklearn.model_selection import GridSearchCV
# Creating the parameter grid
param_grid = {'bootstrap': [True],
              'max_depth': [70, 80, 90, 100],
              'max_features': [2, 3],
              'min_samples_leaf': [1, 2, 3],
              'min_samples_split': [3, 5, 8],
              'n_estimators': [600, 700, 800, 1500]}
# The grid search model
grid_search = GridSearchCV(estimator = rfc, param_grid = param_grid,
                           cv = 4, n_jobs = -1, verbose = 2)
# Fitting the grid search to the data
grid_search.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 288 candidates, totalling 1152 fits
```

In [ ]:
```
# Showing the best parameteres after grid search
grid_search.best_params_
```

Out[ ]:
```
{'bootstrap': True,
 'max_depth': 70,
 'max_features': 2,
 'min_samples_leaf': 2,
 'min_samples_split': 5,
 'n_estimators': 1500}
```

In [ ]:
```
# Best model results
rfc_best = grid_search.best_estimator_
rfc_best.fit(X_train, y_train)
best_y_pred = rfc_best.predict(X_test)
best_accuracy = round(metrics.accuracy_score(y_test, best_y_pred), 3)*100
print('Accuracy of RFC is ' + str(best_accuracy)+'%')
```

Out[ ]:
```
RandomForestClassifier(max_depth=70, max_features=2, min_samples_leaf=2,
                       min_samples_split=5, n_estimators=1500,
                       random_state=2022)
Accuracy of RFC is 70.19999999999999%
```

In [ ]:
```
# Showing the accuracy improvement from rfc_tuned
print('Improvement of {:0.2f}%'.format( 100 * (best_accuracy - tuned_accura
```

```
Improvement of -0.28%
```

Albeit runtime of 1 hour 23 minutes for *rfc_best*, it gave the same accuracy result and sometimes, even slightly worse.

7.2 Feature importance

In [ ]:
```
feature_imp = pd.Series(rfc_tuned.feature_importances_, index=X.columns).so
feature_imp
```

Out[ ]:
```
year              0.131835
artwork_counts    0.125489
tokens            0.123521
media             0.108854
collection        0.104098
rights            0.094078
creator           0.093931
art_series        0.092470
artwork_name      0.081245
likes             0.038369
nsfw              0.006110
dtype: float64
```

In [ ]:
```
# Create a new DataFrame for feature importance
rfc_tuned.feature_names = normal_df.drop("price", axis = 1).columns
rfc_tuned_feature_importance = pd.DataFrame({"Feature": rfc_tuned.feature_n
rfc_tuned_feature_importance = rfc_tuned_feature_importance.sort_values(by
```

In [ ]:
```
# Plotting a bar plot for feature importance
%matplotlib inline

plt.figure(figsize = (14,7))
sns.barplot(rfc_tuned_feature_importance["Feature"], rfc_tuned_feature_impo
plt.title("Feature Importance")
plt.xlabel("Features")
plt.ylabel("Feature Importance Score")
plt.xticks(rotation = "vertical")
plt.legend()
plt.show()
```

Out[ ]:
```
<Figure size 1008x504 with 0 Axes>

/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variables as keyword args: x, y. From version 0.12
, the only valid positional argument will be `data`, and passing other argu
ments without an explicit keyword will result in an error or misinterpretat
ion.
  FutureWarning
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa640ce1c50>
```

Out[ ]:
```
Text(0.5, 1.0, 'Feature Importance')
```

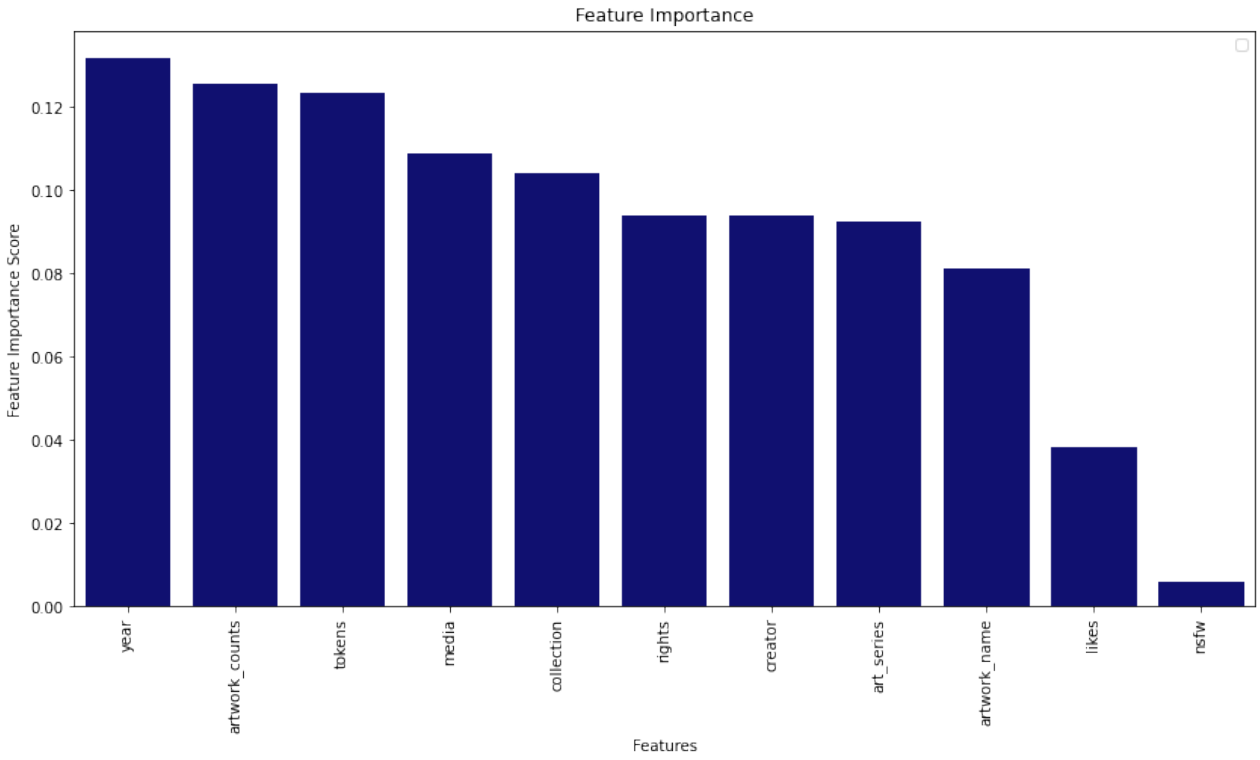
Out[ ]:
```
Text(0.5, 0, 'Features')
```

Out[ ]:
```
Text(0, 0.5, 'Feature Importance Score')
```

Out[ ]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
 <a list of 11 Text major ticklabel objects>)
```

```
No handles with labels found to put in legend.
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7fa6467f9e50>
```



In [ ]:
```
# Generating the model on all features

X = full_df[['year', 'artwork_counts', 'tokens',
             'media', 'collection',  'rights',
             'creator', 'art_series', 'artwork_name', 'likes', 'nsfw']]
y = full_df['price_class']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

random.seed(2022)
rfc_tuned.fit(X_train,y_train)
y_pred=rfc_tuned.predict(X_test)
print('Accuracy of RFC with all features is ' + str(round(metrics.accuracy_
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, max_depth=110, max_features='sqrt',
                       min_samples_leaf=4, n_estimators=234, random_state=2
022)
Accuracy of RFC with all features is 69.89999999999999%
```

In [ ]:
```
# 1 feature dropped
X = full_df[['year', 'artwork_counts', 'tokens',
             'media', 'collection',  'rights',
             'creator', 'art_series', 'artwork_name', 'likes']]
y = full_df['price_class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

rfc_tuned.fit(X_train,y_train)
y_pred=rfc_tuned.predict(X_test)
print('Accuracy of tuned RFC with 1 feature dropped is ' + str(round(metric
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, max_depth=110, max_features='sqrt',
                       min_samples_leaf=4, n_estimators=234, random_state=2
022)
Accuracy of tuned RFC with 1 feature dropped is 69.89999999999999%
```

In [ ]:
```
# 2 features dropped
X = full_df[['tokens', 'artwork_counts', 'year',
             'media', 'collection',  'rights',
             'creator', 'art_series', 'artwork_name']]
y = full_df['price_class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

rfc_tuned.fit(X_train,y_train)
y_pred=rfc_tuned.predict(X_test)
print('Accuracy of tuned RFC with 2 features dropped is ' + str(round(metri
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, max_depth=110, max_features='sqrt',
                       min_samples_leaf=4, n_estimators=234, random_state=2
022)
Accuracy of tuned RFC with 2 features dropped is 69.5%
```

Dropping even 1 feature made model's accuracy lower, causing diminishing returns. Hence, we're going to keep all of them, rerunning the penultimate cell, as it seems to maximise the accuracy.

7.3 Ensemble learning

7.3.1 Voting classifier

In [ ]:
```
from sklearn.ensemble import VotingClassifier

rfc = rfc_tuned
dtc = DecisionTreeClassifier()
xgb = XGBClassifier()

voting_clf = VotingClassifier(
    estimators=[('rfc', rfc_tuned), ('dtc', dtc), ('xgb', xgb)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Out[ ]:
```
VotingClassifier(estimators=[('rfc',
                              RandomForestClassifier(bootstrap=False,
                                                     max_depth=110,
                                                     max_features='sqrt',
                                                     min_samples_leaf=4,
                                                     n_estimators=234,
                                                     random_state=2022)),
                             ('dtc', DecisionTreeClassifier()),
                             ('xgb', XGBClassifier())])
```

In [ ]:
```
from sklearn.metrics import accuracy_score
for clf in (rfc_tuned, dtc, xgb, voting_clf):
  clf.fit(X_train, y_train)
  y_pred = clf.predict(X_test)
  print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, max_depth=110, max_features='sqrt',
                       min_samples_leaf=4, n_estimators=234, random_state=2
022)
RandomForestClassifier 0.6945107398568019
```

Out[ ]:
```
DecisionTreeClassifier()

DecisionTreeClassifier 0.630071599045346
```

Out[ ]:
```
XGBClassifier(objective='multi:softprob')

XGBClassifier 0.6682577565632458
```

Out[ ]:
```
VotingClassifier(estimators=[('rfc',
                              RandomForestClassifier(bootstrap=False,
                                                     max_depth=110,
                                                     max_features='sqrt',
                                                     min_samples_leaf=4,
                                                     n_estimators=234,
                                                     random_state=2022)),
                             ('dtc', DecisionTreeClassifier()),
                             ('xgb',
                              XGBClassifier(objective='multi:softprob'))])
VotingClassifier 0.698090692124105
```

7.3.2 Stacking

Let's try to combine our initial models into a single one using stacking. We'll set random forest to learn how to best combine the predictions from each of the separate 6 models.

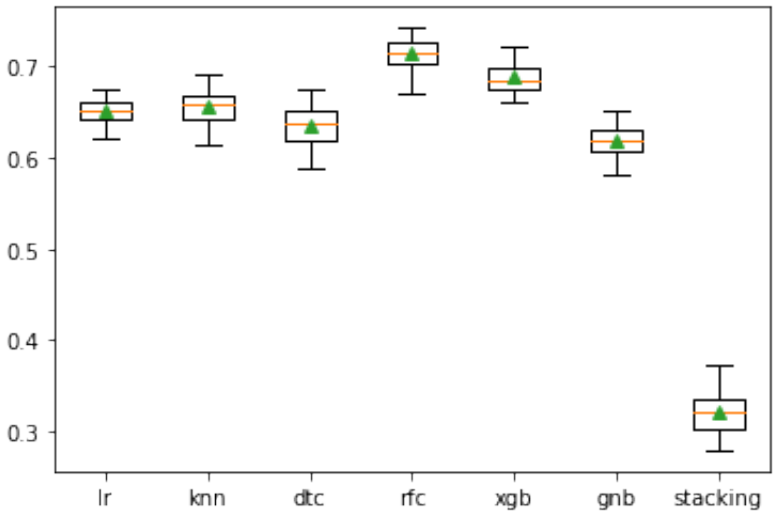
In [ ]:

```
#URL: https://machinelearningmastery.com/stacking-ensemble-machine-learning

# get a stacking ensemble of models
def get_stacking():
        # define the base models
        level0 = list()
        level0.append(('lr', LogisticRegression()))
        level0.append(('knn', KNeighborsClassifier()))
        level0.append(('dtc', DecisionTreeClassifier()))
        level0.append(('rfc', rfc_tuned))
        level0.append(('xgb', XGBClassifier()))
        level0.append(('gnb', GaussianNB()))

        # define meta learner model
        level1 = rfc_tuned
        # define the stacking ensemble
        model = StackingClassifier(estimators=level0, final_estimator=level
        return model

# get a list of models to evaluate
def get_models():
        models = dict()
        models['lr'] = LogisticRegression()
        models['knn'] = KNeighborsClassifier()
        models['dtc'] = DecisionTreeClassifier()
        models['rfc'] = rfc_tuned
        models['xgb'] = XGBClassifier()
        models['gnb'] = GaussianNB()
        models['stacking'] = get_stacking()
        return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
        cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state
        scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_
        return scores

# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
        scores = evaluate_model(model, X, y)
        results.append(scores)
        names.append(name)
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>lr 0.651 (0.012)
>knn 0.656 (0.017)
>dtc 0.634 (0.021)
>rfc 0.714 (0.017)
>xgb 0.688 (0.016)
>gnb 0.619 (0.017)
```

```
>stacking 0.322 (0.023)
```

Out[ ]:
```
{'boxes': [<matplotlib.lines.Line2D at 0x7fa62ee84890>,
  <matplotlib.lines.Line2D at 0x7fa62ee97390>,
  <matplotlib.lines.Line2D at 0x7fa62eea5dd0>,
  <matplotlib.lines.Line2D at 0x7fa62ee3e910>,
  <matplotlib.lines.Line2D at 0x7fa62ee58410>,
  <matplotlib.lines.Line2D at 0x7fa62ef4fb10>,
  <matplotlib.lines.Line2D at 0x7fa62efd6f10>],
 'caps': [<matplotlib.lines.Line2D at 0x7fa62ee878d0>,
  <matplotlib.lines.Line2D at 0x7fa62ee87e10>,
  <matplotlib.lines.Line2D at 0x7fa62ee9e3d0>,
  <matplotlib.lines.Line2D at 0x7fa62ee9e910>,
  <matplotlib.lines.Line2D at 0x7fa62eeaee90>,
  <matplotlib.lines.Line2D at 0x7fa62eeb7450>,
  <matplotlib.lines.Line2D at 0x7fa62ee47990>,
  <matplotlib.lines.Line2D at 0x7fa62ee47ed0>,
  <matplotlib.lines.Line2D at 0x7fa62ee62490>,
  <matplotlib.lines.Line2D at 0x7fa62ee62990>,
  <matplotlib.lines.Line2D at 0x7fa62ef2bf90>,
  <matplotlib.lines.Line2D at 0x7fa62ef2b690>,
  <matplotlib.lines.Line2D at 0x7fa62ee7d5d0>,
  <matplotlib.lines.Line2D at 0x7fa62ee7db10>],
 'fliers': [<matplotlib.lines.Line2D at 0x7fa62ee8fe50>,
  <matplotlib.lines.Line2D at 0x7fa62eea5910>,
  <matplotlib.lines.Line2D at 0x7fa62ee3e450>,
  <matplotlib.lines.Line2D at 0x7fa62ee4fed0>,
  <matplotlib.lines.Line2D at 0x7fa62eec6490>,
  <matplotlib.lines.Line2D at 0x7fa62ef6cb10>,
  <matplotlib.lines.Line2D at 0x7fa62ee05b10>],
 'means': [<matplotlib.lines.Line2D at 0x7fa62ee8f910>,
  <matplotlib.lines.Line2D at 0x7fa62eea53d0>,
  <matplotlib.lines.Line2D at 0x7fa62eeb7ed0>,
  <matplotlib.lines.Line2D at 0x7fa62ee4f990>,
  <matplotlib.lines.Line2D at 0x7fa62ee6a490>,
  <matplotlib.lines.Line2D at 0x7fa6400ff290>,
  <matplotlib.lines.Line2D at 0x7fa62ee055d0>],
 'medians': [<matplotlib.lines.Line2D at 0x7fa62ee8f3d0>,
  <matplotlib.lines.Line2D at 0x7fa62ee9ee50>,
  <matplotlib.lines.Line2D at 0x7fa62eeb7990>,
  <matplotlib.lines.Line2D at 0x7fa62ee4f450>,
  <matplotlib.lines.Line2D at 0x7fa62ee62ed0>,
  <matplotlib.lines.Line2D at 0x7fa62ef1ff10>,
  <matplotlib.lines.Line2D at 0x7fa62ee05090>],
 'whiskers': [<matplotlib.lines.Line2D at 0x7fa62ee84e10>,
  <matplotlib.lines.Line2D at 0x7fa62ee87390>,
  <matplotlib.lines.Line2D at 0x7fa62ee97910>,
  <matplotlib.lines.Line2D at 0x7fa62ee97e50>,
  <matplotlib.lines.Line2D at 0x7fa62eeae3d0>,
  <matplotlib.lines.Line2D at 0x7fa62eeae950>,
  <matplotlib.lines.Line2D at 0x7fa62ee3eed0>,
  <matplotlib.lines.Line2D at 0x7fa62ee47450>,
  <matplotlib.lines.Line2D at 0x7fa62ee58a10>,
  <matplotlib.lines.Line2D at 0x7fa62ee58f10>,
  <matplotlib.lines.Line2D at 0x7fa64020f9d0>,
  <matplotlib.lines.Line2D at 0x7fa62ef37a10>,
  <matplotlib.lines.Line2D at 0x7fa62ee6ab10>,
  <matplotlib.lines.Line2D at 0x7fa62ee7d090>]}
```



Doesn't look good - our overall stacking accuracy is dragged down. If we chose a stacking ensemble as our final model with only 2 best models, the cell below demonstrates how we would use it on our dataset. However, this is not our final model due to the low accuracy score.

In [ ]:

```
# define the base models
level0 = list()
level0.append(('rfc', rfc_tuned))
level0.append(('xgb', XGBClassifier()))

# define meta learner model
level1 = rfc_tuned

# define the stacking ensemble
model = StackingClassifier(estimators=level0, final_estimator=level1, cv=5)

# fit the model on all available data
model.fit(X, y)

# make a prediction for one example
random_NFT = X.sample(n=1)
yhat = model.predict(random_NFT)
print('Predicted Class: %d' % (yhat))
```

Out[ ]:
```
StackingClassifier(cv=5,
                   estimators=[('rfc',
                                RandomForestClassifier(bootstrap=False,
                                                       max_depth=110,
                                                       max_features='sqrt',
                                                       min_samples_leaf=4,
                                                       n_estimators=234,
                                                       random_state=2022)),
                               ('xgb', XGBClassifier())],
                   final_estimator=RandomForestClassifier(bootstrap=False,
                                                          max_depth=110,
                                                          max_features='sqr
t',
                                                          min_samples_leaf=
4,
                                                          n_estimators=234,
                                                          random_state=2022
))
Predicted Class: 1
```

Ensemble learning did not quite work for the dataset, making us stop with hypertuned random forest classifier with a final accuracy score of 71.4%.

7.4 ROC: evaluation of the final model

"ROC curves are typically used in binary classification to study the output of a classifier. In order to extend ROC curve and ROC area to multi-label classification, it is necessary to binarize the output." (scikit-learn, n.d.)

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle

from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import roc_auc_score

# Refresh data that is ultimately used
X = full_df[['year', 'artwork_counts', 'tokens',
             'media', 'collection',  'rights',
             'creator', 'art_series', 'artwork_name', 'likes']]
y = full_df['price_class']

# Shuffle and split training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ra

# Binarize the output
y = label_binarize(y, classes=[0, 1, 2])
n_classes = y.shape[1]

# Learn to predict each class against the other
RFC = RandomForestClassifier(100, random_state = 2022)
OVRC = OneVsRestClassifier(RFC)
OVRC.fit(X_train, y_train)
```

Out[ ]:
```
OneVsRestClassifier(estimator=RandomForestClassifier(random_state=2022))
```

Individual predictions for the 3 classifiers can be get as following:

In [ ]:
```
#URL: https://stackoverflow.com/questions/57962736/can-onevsrestclassifier-

print(OVRC.estimators_[0].predict_proba(X_test[0:5])) # label 0 vs the rest
print(OVRC.estimators_[1].predict_proba(X_test[0:5])) # label 1 vs the rest
print(OVRC.estimators_[2].predict_proba(X_test[0:5])) # label 2 vs the rest
```

```
[[0.36 0.64]
 [0.41 0.59]
 [0.44 0.56]
 [0.32 0.68]
 [0.47 0.53]]
[[0.8  0.2 ]
 [0.63 0.37]
 [0.71 0.29]
 [0.72 0.28]
 [0.62 0.38]]
[[0.98 0.02]
 [1.   0.  ]
 [0.96 0.04]
 [0.87 0.13]
 [0.85 0.15]]
```

Let's compute macro-average ROC curve and ROC area.

In [ ]:
```
#URL: https://stackoverflow.com/questions/45332410/roc-for-multiclass-class

from sklearn.metrics import roc_curve, auc
from sklearn import datasets
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

y = label_binarize(y, classes=[0,1,2])
n_classes = 3

# shuffle and split training and test sets
X_train, X_test, y_train, y_test =\
    train_test_split(X, y, test_size=0.2, random_state=2022)

# classifier
random.seed(2022)
y_score = rfc_tuned.fit(X_train, y_train).predict(X_test)

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot of a ROC curve for a specific class
for i in range(n_classes):
    plt.figure()
    plt.plot(fpr[i], tpr[i], label='ROC curve (area = %0.2f)' % roc_auc[i])
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```

Out[ ]:
```
<Figure size 432x288 with 0 Axes>
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fa62e962810>]
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fa62e962ed0>]
```

Out[ ]:
```
(0.0, 1.0)
```

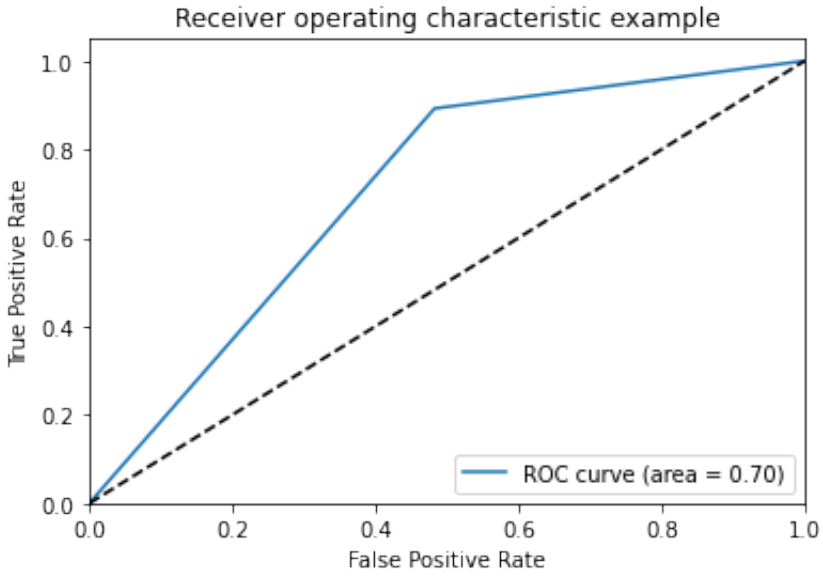
Out[ ]:
```
(0.0, 1.05)
```

Out[ ]:
```
Text(0.5, 0, 'False Positive Rate')
```

Out[ ]: Text(0, 0.5, 'True Positive Rate')

Out[ ]: Text(0.5, 1.0, 'Receiver operating characteristic example')

Out[ ]: <matplotlib.legend.Legend at 0x7fa62e94a950>



Out[ ]: <Figure size 432x288 with 0 Axes>

Out[ ]: [<matplotlib.lines.Line2D at 0x7fa62e92b910>]

Out[ ]: [<matplotlib.lines.Line2D at 0x7fa62e8d1550>]

Out[ ]: (0.0, 1.0)

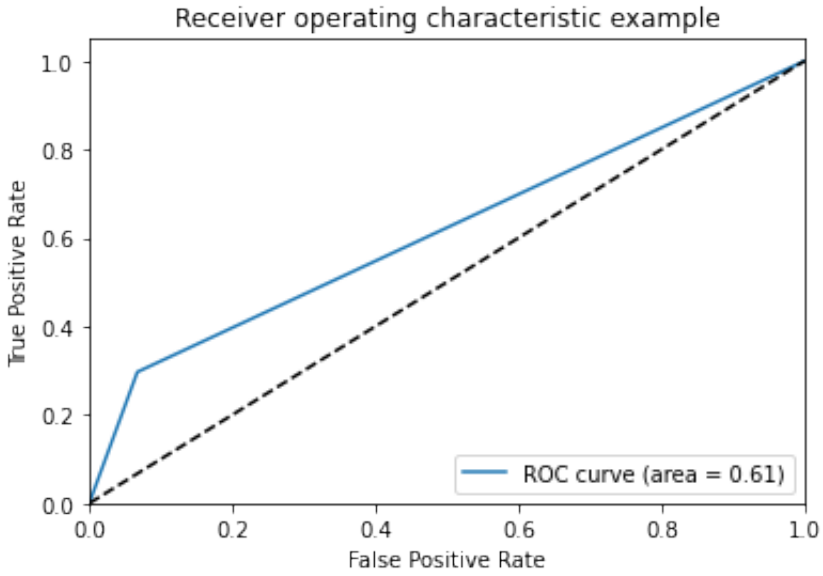
Out[ ]: (0.0, 1.05)

Out[ ]: Text(0.5, 0, 'False Positive Rate')

Out[ ]: Text(0, 0.5, 'True Positive Rate')

Out[ ]: Text(0.5, 1.0, 'Receiver operating characteristic example')

Out[ ]: <matplotlib.legend.Legend at 0x7fa62e934e90>



Out[ ]: <Figure size 432x288 with 0 Axes>

Out[ ]: [<matplotlib.lines.Line2D at 0x7fa62e87e7d0>]

Out[ ]: [<matplotlib.lines.Line2D at 0x7fa62e846b10>]

Out[ ]: (0.0, 1.0)

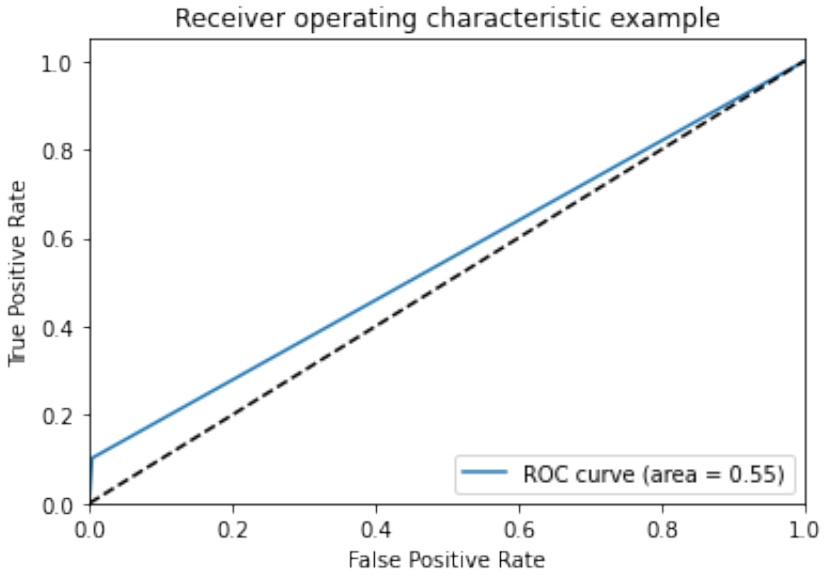
Out[ ]: (0.0, 1.05)

Out[ ]: Text(0.5, 0, 'False Positive Rate')

Out[ ]: Text(0, 0.5, 'True Positive Rate')

Out[ ]: Text(0.5, 1.0, 'Receiver operating characteristic example')

Out[ ]: <matplotlib.legend.Legend at 0x7fa62e896910>



Classifiers that give curves closer to the top-left corner indicate a better performance. As we can see, the curve for class 0 (cheap NFTs) VS all comes the closest to the ideal clinical discriminator, in comparison to the other 2 lines.

Class 1 (average NFTs) VS all has a moderate accuracy result, with AUC equal to 0.68 which feels like an average of our all trained models in chapter 6.

With the ROC curve being the closest o the 45-degree diagonal, class 3 (expensice NFTs) is the least accurate test, unfortunately, having nearly no predictive value.

Chapter 8: Further work

8.1 Segmenation clustering on colours

Given more time and resources like bigger Google Drive memory and faster GPU, I would perform clustering on the dataset's images. The way I'd start approaching it is in the code below. I would create a separate column with the dominant colour of the NFT, and cluster them into RGB channels. A model which also contains the channel would be more informative both for collectors that want to spiculate the next hottest NFT in Hives value and give more autonomy to creators who would hope to set a price in a higher class ("2").

In [ ]:
```
#define values
values = [2, 3]

#drop rows that contain any value in the list
df = df[df.media.isin(values) == False]
```

In [ ]:
```
# Changing 'path' name to work in Google Collab.
# Don't run this cell if viewing in Faculty or locally.

df['path'] = df['path'].str[1:]
df['path'] ='/content/drive/MyDrive/Colab Notebooks/Predictive' + df['path'
df['path'].sample(n=1).iloc[0]
```

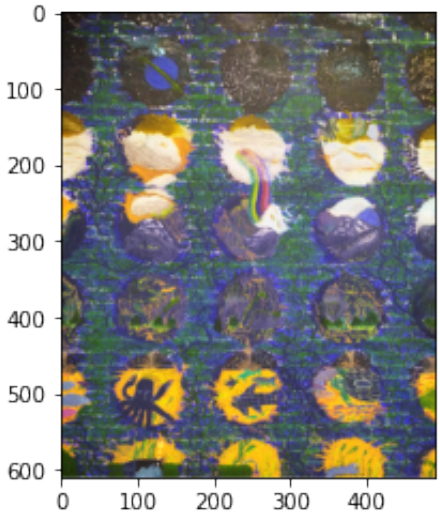
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4: SettingWith
CopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs
/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  after removing the cwd from sys.path.
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: SettingWith
CopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs
/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
```

Out[ ]:
```
'/content/drive/MyDrive/Colab Notebooks/Predictive/dataset/image/Qmavzf1ZPt
7isXpRZGXrxPU3Rso9F3Fra293UVoJ8yWaos.png'
```

In [ ]:
```
images = []
for x in df.path[:10]:
    x = cv2.imread(x)
    images.append(x)
```

In [ ]:
```
plt.imshow(images[3])
```

Out[ ]: <matplotlib.image.AxesImage at 0x7f0c46ca1890>



In [ ]:
```
pip install opencv-python
```

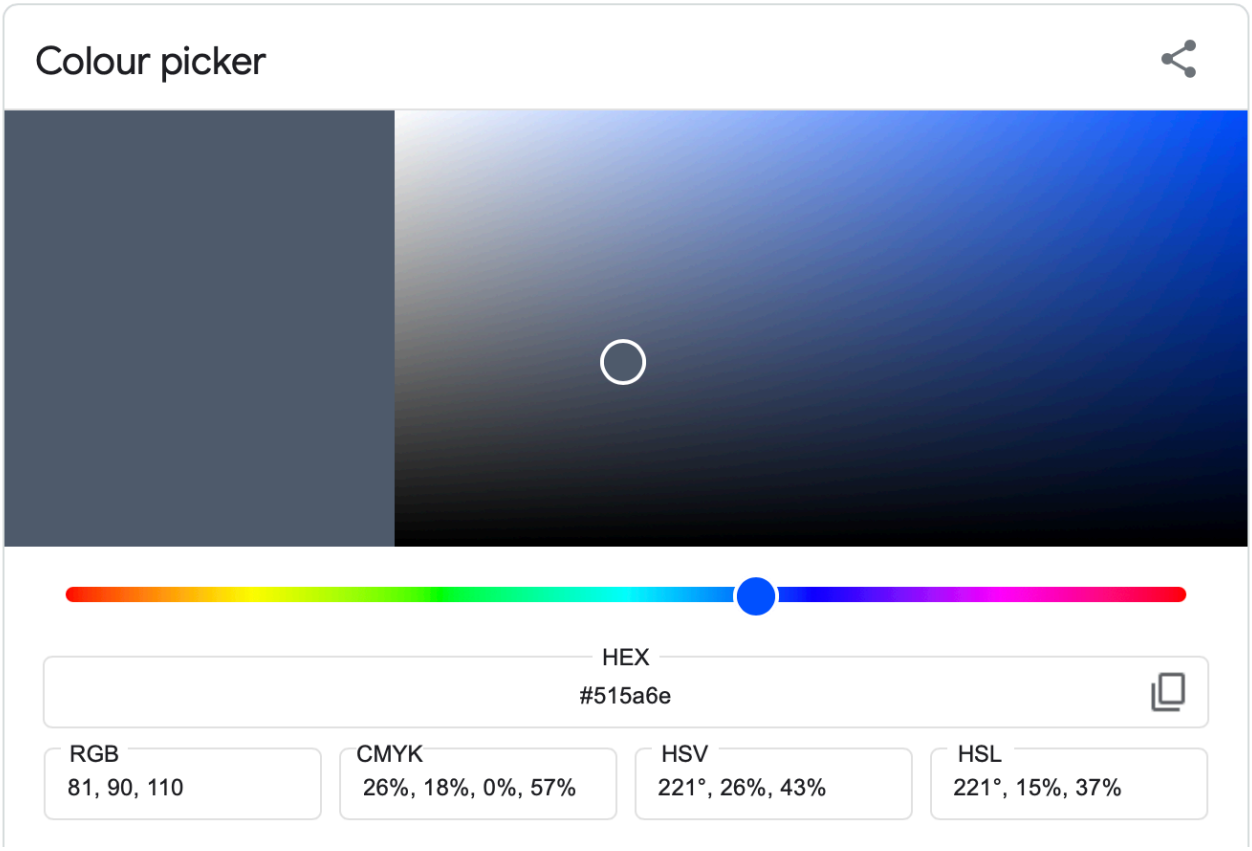
```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.7/di
st-packages (4.1.2.30)
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/di
st-packages (from opencv-python) (1.21.5)
```

In [ ]:
```
NUM_CLUSTERS = 5

print('reading image')
im = images[3]
#im = im.resize((600, 600)) # optional, to reduce time
ar = np.asarray(im)
shape = ar.shape
ar = ar.reshape(scipy.product(shape[:2]), shape[2]).astype(float)

print('finding clusters')
codes, dist = scipy.cluster.vq.kmeans(ar, NUM_CLUSTERS)
print('cluster centres:\n', codes)

vecs, dist = scipy.cluster.vq.vq(ar, codes)         # assign codes
counts, bins = scipy.histogram(vecs, len(codes))    # count occurrences

index_max = scipy.argmax(counts)                    # find most frequent
peak = codes[index_max]
colour = binascii.hexlify(bytearray(int(c) for c in peak)).decode('ascii')
print('most frequent is %s (#%s)' % (peak, colour))
```

```
reading image
finding clusters
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:8: Deprecation
Warning: scipy.product is deprecated and will be removed in SciPy 2.0.0, us
e numpy.product instead

cluster centres:
 [[ 81.1648292   90.29676796 110.56592341]
 [220.13537412 171.43418637  45.02169433]
 [122.43624067 123.01486255 150.00568567]
 [ 64.1374965   71.56656035  63.02931006]
 [229.05858305 219.22492558 202.79075213]]
most frequent is [ 81.1648292   90.29676796 110.56592341] (#515a6e)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:15: Deprecatio
nWarning: scipy.histogram is deprecated and will be removed in SciPy 2.0.0,
use numpy.histogram instead
  from ipykernel import kernelapp as app
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:17: Deprecatio
nWarning: scipy.argmax is deprecated and will be removed in SciPy 2.0.0, us
e numpy.argmax instead
```

In [ ]:

```
display(image(filename='/content/drive/MyDrive/Colab Notebooks/Predictive/i
```



8.2 Other improvements

Transformation pipeline would also be useful to do all at once: data cleaning, adding new features, transforming categorical data by encoding, processing numerical and categorical data separately. Additionally, the performance could have been improved through higher *n_iter* parameter in grid search iterations but the main trade-off is runtime.

Chapter 9: Conclusion

This project used a secondary dataset to investigate features that digital artists of NFT Showroom should consider the most when defining a price of their NFT. We gathered insights about the features through data visualizations first, making initial observations in relationships, which were not as accurate to the naked eye. For example, we would have expected a year of NFT release and number of its editions (tokens) to be very deterministic for a price. Even after label encoding of categorical columns, feature importance revealed those to be 1st and 3rd order of importance respectfully.

Six machine learning models are tested and random forest classifier is chosen. The fine-tuned model can be used both by data-savvy NFT creators and collectors willing to invest into highly valued digital artworks. It is suggested that apart from unique names, artist's final output on the NFT market (*artwork_count*) is also a very important consideration when determining a price.

Since there were not many explanatory variables, the model has suffered from the reduced precision of price class predictions. More quantitative rather than categorical data would give more informative insights into the problem, which might be improved with a more consistent NFT selling process in the future.

Chapter 10: References

Analytics Vidhya. (2020). *Feature Scaling | Standardization Vs Normalization*. [online] Available at: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/ [Accessed 10 Mar. 2022].

Brownlee, J. (2020). *Bagging and Random Forest for Imbalanced Classification*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/ [Accessed 16 Mar. 2022].

CoinGecko (n.d.). *Hive Price Today, Chart, Market Cap & News*. [online] CoinGecko. Available at: https://www.coingecko.com/en/coins/hive [Accessed 12 Feb. 2022].

De Haan, A. (2021). *NFT Art Collection 2021*. [online] kaggle.com. Available at: https://www.kaggle.com/vepnar/nft-art-dataset.

Géron, A. (2019). *Classification - Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition* [Book]. [online] www.oreilly.com. Available at: https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch03.html [Accessed 11 Mar. 2022].

Google Trends (n.d.). *Google Trends - NFT*. [online] Google Trends. Available at: https://trends.google.com/trends/explore?date=all&q=nft [Accessed 21 Feb. 2022].

Koehrsen, W. (2018). *Hyperparameter Tuning the Random Forest in Python*. [online] Medium. Available at: https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74 [Accessed 17 Mar. 2022].

MonkeyLearn Blog. (2020). *Classification Algorithms in Machine Learning: How They Work*. [online] Available at: https://monkeylearn.com/blog/classification-algorithms/.

scikit-learn. (n.d.). *Receiver Operating Characteristic (ROC)*. [online] Available at: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html [Accessed 16 Mar. 2022].

Sharma, R. (2021). *Non-Fungible Token Definition: Understanding NFTs*. [online] Investopedia. Available at: https://www.investopedia.com/non-fungible-tokens-nft-5115211 [Accessed 21 Feb. 2022].

MSIN0097_Predictive_Analytics_Individual_Assignment

GRADEMARK REPORT

FINAL GRADE

70 /100

GENERAL COMMENTS

Instructor

Good work on this assignment. It wa good to see a machine learning canvas in your introduction.. Good work on providing narrative on your work. Adding more arguments towards your chosen topic and going in-depth in each of the areas would have enchanced the quality of your work. Your visualisations are good but it would be great to see more interesting insights. Finally, it would be great if your code is cleaned a bit and good work on providing evidence in support of your arguments.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15