assignment

August 16, 2020

[2]:
```
# Necessary imports required for achieving different tasks
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
import scipy.stats as ss
import numpy as np
```

[3]:
```
# Helper functions
def cramers_v(x, y):
    """Returns the correlation between two categorical variables.
       Took this code from the Internet."""
    confusion_matrix = pd.crosstab(x,y)
    chi2 = ss.chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2/n
    r,k = confusion_matrix.shape
    phi2corr = max(0, phi2-((k-1)*(r-1))/(n-1))
    rcorr = r-((r-1)**2)/(n-1)
    kcorr = k-((k-1)**2)/(n-1)
    return np.sqrt(phi2corr/min((kcorr-1),(rcorr-1)))
```

[4]:
```
# Setting the style for the plots which we obtain from the seaborn package's␣
 ↪methods
sns.set(style='whitegrid', palette="deep", font_scale=1.1, rc={"figure.figsize":
 ↪ [8, 5]})
```

[5]:
```
# Importing the dataset with Pandas
data_case_study = pd.read_csv('AS24_Case_Study_Data.csv', sep=';')
```

[6]:
```
data_case_study.head()
```

[6]:
```
   article_id product_tier      make_name  price  first_zip_digit  \
0   350625839        Basic     Mitsubishi  16750                5
1   354412280        Basic  Mercedes-Benz  35950                4
2   349572992        Basic  Mercedes-Benz  11950                3
3   350266763        Basic           Ford   1750                6
```

```
4   355688985        Basic  Mercedes-Benz  26500                3

   first_registration_year created_date deleted_date  search_views  \
0                     2013     24.07.18     24.08.18        3091.0
1                     2015     16.08.18     07.10.18        3283.0
2                     1998     16.07.18     05.09.18        3247.0
3                     2003     20.07.18     29.10.18        1856.0
4                     2014     28.08.18     08.09.18         490.0

   detail_views  stock_days                  ctr
0         123.0          30   0.03780329990294403
1         223.0          52   0.06792567773378008
2         265.0          51    0.0816137973514013
3          26.0         101  0.014008620689655173
4          20.0          12   0.04081632653061224
```

[7]:
```python
# Getting an idea of the number of entries in the dataset
print('The shape of the dataset pandas frame is: {}'.format(data_case_study.
 ↪shape))
```

```
The shape of the dataset pandas frame is: (78321, 12)
```

[8]:
```python
# Observing if there are any null values in any of the coloumns in the dataset
for column in data_case_study.columns:
    print('Number of null values in the {} coloumn are: {}'.format(column,␣
 ↪data_case_study[column].isnull().sum()))
```

```
Number of null values in the article_id coloumn are: 0
Number of null values in the product_tier coloumn are: 0
Number of null values in the make_name coloumn are: 0
Number of null values in the price coloumn are: 0
Number of null values in the first_zip_digit coloumn are: 0
Number of null values in the first_registration_year coloumn are: 0
Number of null values in the created_date coloumn are: 0
Number of null values in the deleted_date coloumn are: 0
Number of null values in the search_views coloumn are: 10
Number of null values in the detail_views coloumn are: 10
Number of null values in the stock_days coloumn are: 0
Number of null values in the ctr coloumn are: 24
```

Okay, so we have 10 null/nan values in the search_views coloumn as well as in the detail_views coloumn. In the ctr coloumn, we have 24 null/nan values out of which 10 are due to the search_views and detail_views null/nan values. We will look at this problem when we explore the distribution of the data in these coloumns.

Firstly, we will perform the exploratory data analysis for getting insights on whether it is possible to predict the "product_tier" of an article from the other coloumns of the dataset.

```
[9]: # Removing the article_id coloumn (no significance)
data_mod = data_case_study.drop(['article_id'], axis=1)
```

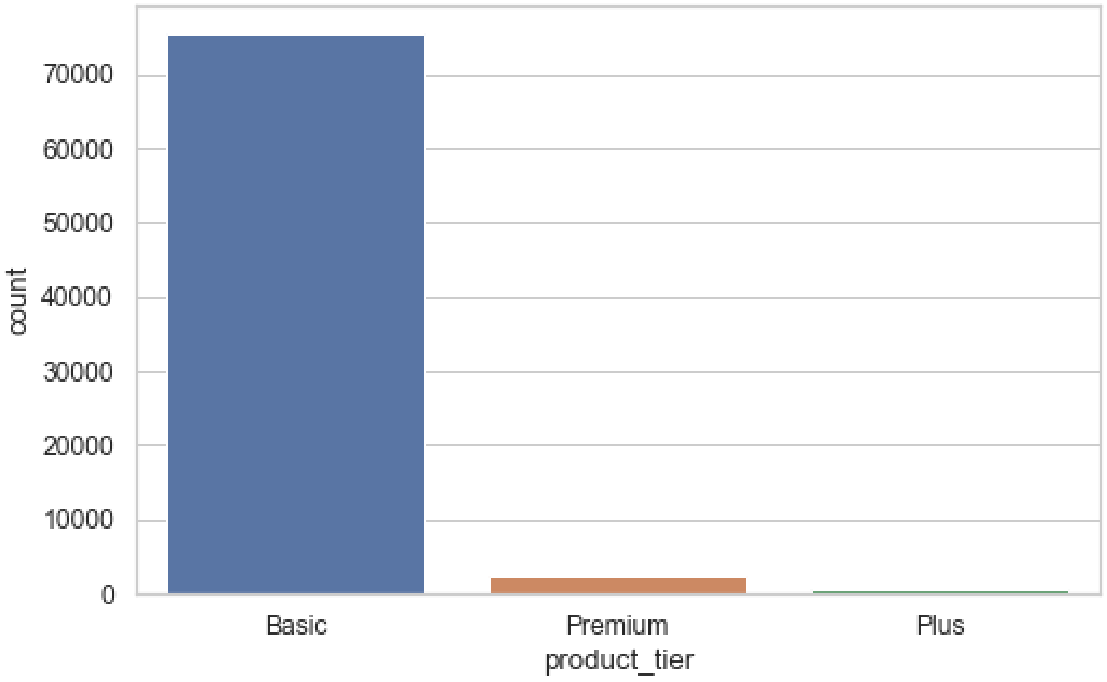
I removed the article_id coloumn from the dataset since it is just a unique identifier which might have been allocated to different car details stored in the database. I think there is no importance of using this variable in our classification task at hand

```
[10]: # Observing the counts of the possible product tiers
data_mod['product_tier'].value_counts()
```

```
[10]: Basic      75421
Premium     2324
Plus         576
Name: product_tier, dtype: int64
```

```
[11]: # Bar plot showing counts for different types of product tiers
sns.countplot(data_mod['product_tier'])
```

```
[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1a23650310>
```



As shown by the value counts and the bar plot above, it is evident that we have a highly misbalanced dataset. So, whatever model we train, it would have a huge bias towards the 'Basic' product tier type as compared to the other product tiers. This problem could be solved by using some over-sampling techniques (SMOTE or just repeating the entries multiple times) for the minority classes (Premium and Plus) or by using some downsampling strategies (Tomek links) for the majority class (Basic). Additionally, we could also assign different weights to these classes while computing the loss. But before doing that, let us observe if the other coloumns could actually even help in predicting these classes or not.

[12]:
```
# Understanding the relationship between the product_tier and make_name␣
↪(categorical variable)
make_tier_rel = pd.crosstab(data_mod.make_name, data_mod.product_tier,␣
↪normalize='index')

print(make_tier_rel)

stack_tier_rel = make_tier_rel.stack().reset_index().rename(columns={0:'value'})
```

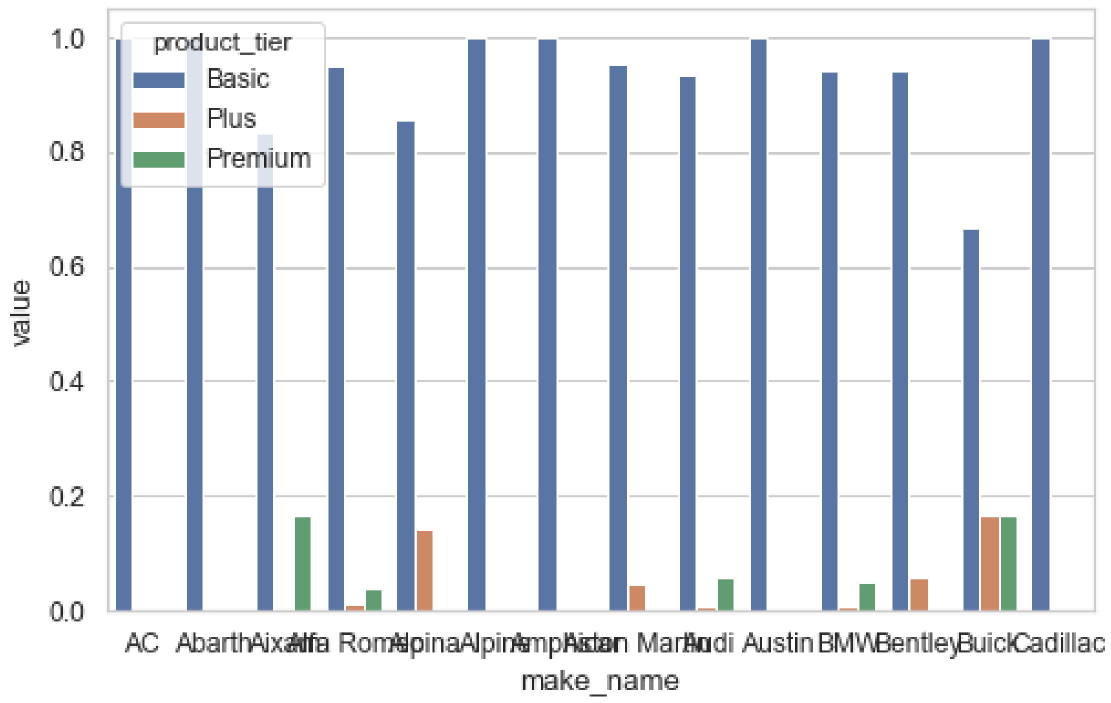
```
product_tier     Basic      Plus   Premium
make_name
AC            1.000000  0.000000  0.000000
Abarth        1.000000  0.000000  0.000000
Aixam         0.833333  0.000000  0.166667
Alfa Romeo    0.949248  0.013158  0.037594
Alpina        0.857143  0.142857  0.000000
…                 …         …         …
Triumph       1.000000  0.000000  0.000000
Trucks-Lkw    1.000000  0.000000  0.000000
Volkswagen    0.959915  0.008165  0.031919
Volvo         0.953931  0.008012  0.038057
smart         0.994536  0.000000  0.005464

[91 rows x 3 columns]
```

By looking at the percentages of some of the categories in the 'make_names' coloumn, we can observe that for most of the car makes, we have almost 100% of them having the basic product tier. But, at the same time if we look at e.g. Aixam, we can observe that ~17% of them have the premium product tier with ~83% having the basic product tier. So, this fact might help our model in making the decisions. In order to have a more comprehensive look, we can observe some plots in subsets as shown below.

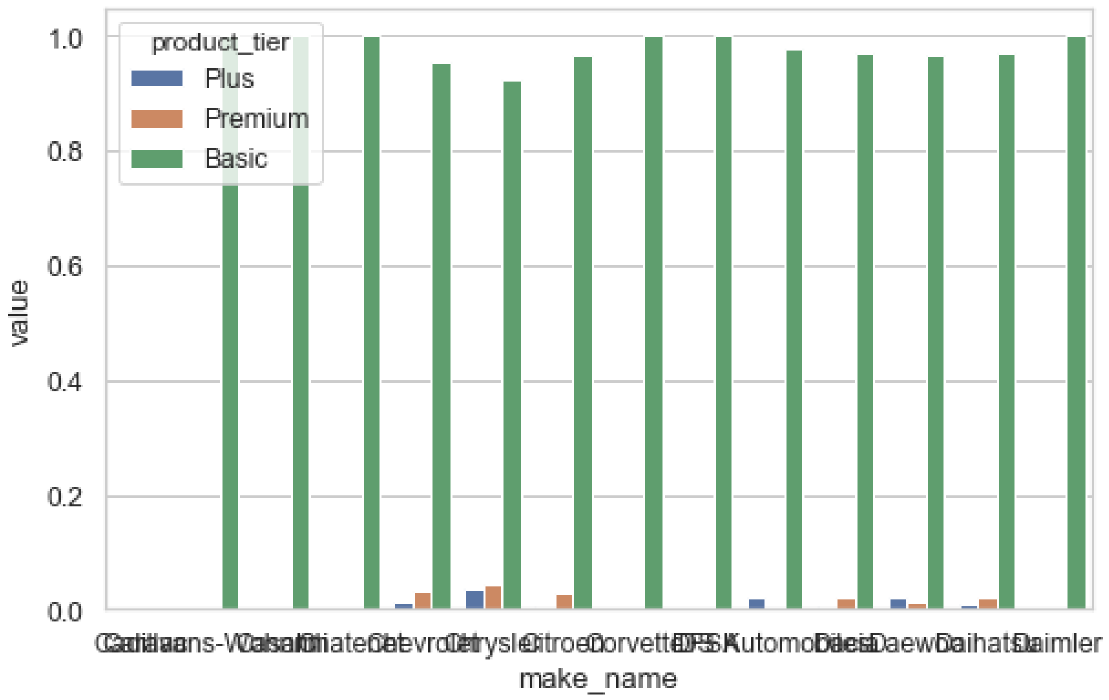
[13]:
```
# Bar plot for the first 40 entries
sns.barplot(x=stack_tier_rel.make_name[:40], y=stack_tier_rel.value[:40],␣
↪hue=stack_tier_rel.product_tier[:40])
```

[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10cb97610>
```



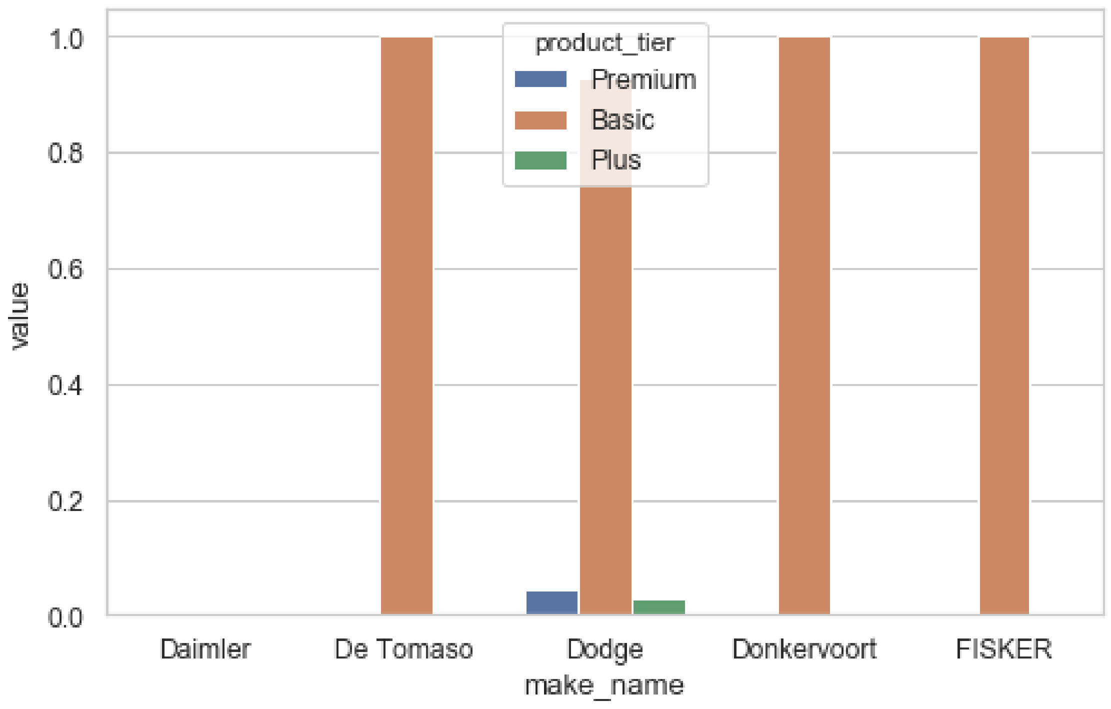
[14]:
```
# Bar plot for the next 40 entries
sns.barplot(x=stack_tier_rel.make_name[40:80], y=stack_tier_rel.value[40:80],␣
↪hue=stack_tier_rel.product_tier[40:80])
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10f829dd0>
```



```
[15]: # Bar plot for the last 11 entries
      sns.barplot(x=stack_tier_rel.make_name[80:91], y=stack_tier_rel.value[80:91],␣
       ↪hue=stack_tier_rel.product_tier[80:91])
```

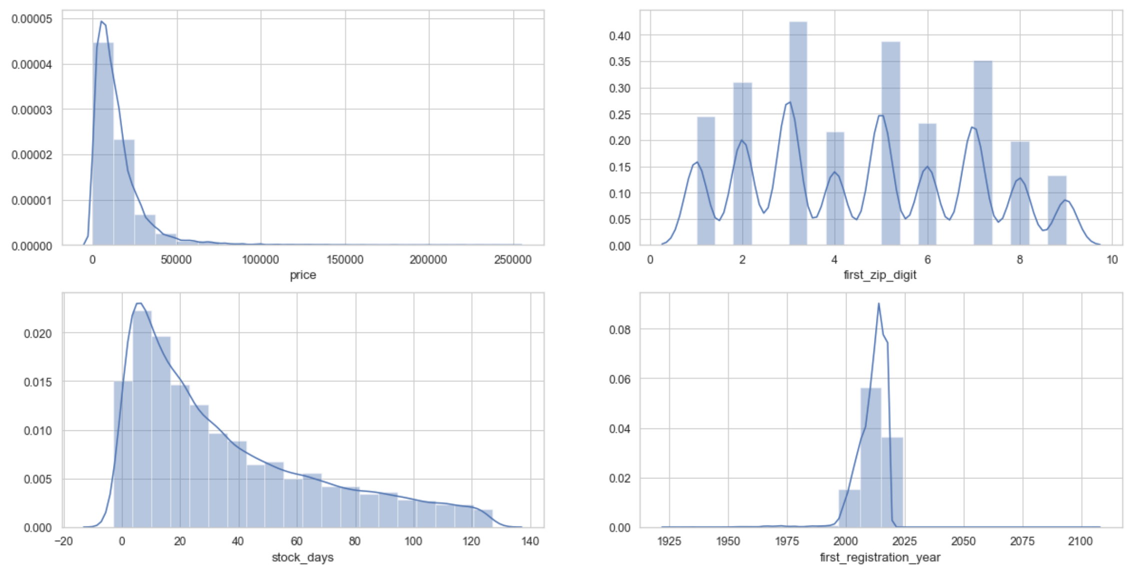
```
[15]: <matplotlib.axes._subplots.AxesSubplot at 0x1a23bd8a90>
```



As we can see from the above plots that there are some car makers for which the product tier 'premium' and 'plus' is a bit significant. So this fact should help our model a bit while deciding the product tier but I suspect that the huge misbalance in the product tiers might not let the model leverage the small variations for some of the car makers.

```
[16]: # Having a look at the distribution of the values in the numerical columns
numerical_var = ['price', 'first_zip_digit', 'stock_days',␣
 ↪'first_registration_year']
fig, ax = plt.subplots(2, 2, figsize=(20, 10))

for var, subplot in zip(numerical_var, ax.flatten()):
    sns.distplot(data_mod[var], norm_hist=False, kde=True, ax=subplot, bins=20,␣
 ↪color='b')
```



From the above plots, we can get some intuitions about the distribution of the values in our numerical variables. As we can see, most of the cars have their prices less than 50000 with only a few having a greater price (skewed distribution). In addition, the variable first_registeration year shows that all of the cars have their first registeration year between 2000 and 2020. Apart from that, the variable first_zip_digit shows a high frequency distribution and the variable stock_days shows a pretty nice distribution which is one of the things which we can keep in our mind for now. Moving on, we know that the stock_days is the difference between the created_date and the deleted_date i.e. a linear combination with the coefficients +1 and -1, we can remove the created_date and the deleted_date coloumn as well instead of getting different representations for both of these dates which might increase the number of variables and could potentially lead to the overfitting problems. I will remove these columns later.

[17]:
```
# Analysis on the ctr variable which is an object and could be a mixture of␣
↪different data types
data_mod['ctr'].apply(type).value_counts()
```

[17]:
```
<class 'str'>      78297
<class 'float'>       24
Name: ctr, dtype: int64
```

So, we can see that most of the values are strings objects however we have some float objects too. Let's try checking these float values

[18]:
```
# Checking NaN values in floats
num_nan = 0
index_list_nan = []

for i, num in enumerate(data_mod['ctr']):
    if isinstance(num, float) and math.isnan(num):
        num_nan += 1
        index_list_nan.append(i)

print('Number of NaN float numbers are: {}'.format(num_nan))
```

```
Number of NaN float numbers are: 24
```

So, all of these float numbers are NaN so we should remove them. We have two options now: 1) Either remove the rows with NaN values or 2) put 0.0 for the ctr values in these rows. Let's see what is the product tier for these NaN values.

[19]:
```
# Checking the product tier for the NaN ctr values
data_mod.loc[index_list_nan, ['product_tier']].value_counts()
```

[19]:
```
product_tier
Basic           24
dtype: int64
```

Since, all of the product tiers corresponding to the NaN ctr values are 'Basic', we can remove them since it is our majority class.

[20]:
```
# Storing the copy of the original data just as a backup and removing the nan␣
↪values from the dataset.
data_mod_back = data_mod.copy()
data_mod = data_mod.drop(index_list_nan)
data_mod = data_mod.reset_index(drop=True)
```

Let's look at the detail_views and search_views coloumn again

[21]:
```
# printing the null values in the detail_views and search_views column
print('Number of null values in the search_views coloums are: {}'.
↪format(data_mod['search_views'].isnull().sum()))
print('Number of null values in the detail_views coloums are: {}'.
↪format(data_mod['detail_views'].isnull().sum()))
```

```
Number of null values in the search_views coloums are: 0
Number of null values in the detail_views coloums are: 0
```

Our search_views and detail_views columns are clean now!

Now let's look at the strings in the ctr coloumn. While trying to convert them into float, I discovered that there are some noisy strings inside where we have more than one decimal points. Let's look how many of them do we have.

[22]:
```
# Counting noisy string ctr values
noisy_str = 0
index_noisy_str = []

for i, str_num in enumerate(data_mod['ctr']):
    if len(str_num.split('.')) > 2:
        print(str_num)
        noisy_str += 1
        index_noisy_str.append(i)

print('Number of noisy ctr string values are: {}'.format(noisy_str))
```

```
27.624.309.392.265.100
4.086.021.505.376.340
30.066.815.144.766.100
5.126.118.795.768.910
1.485.148.514.851.480
49.729.729.729.729.700
3.561.643.835.616.430
19.169.329.073.482.400
43.058.350.100.603.600
16.423.082.718.095.500
17.298.937.784.522.000
23.692.810.457.516.300
18.787.878.787.878.700
25.096.525.096.525.000
21.399.456.521.739.100
16.666.666.666.666.600
29.850.746.268.656.700
5.551.948.051.948.050
41.534.988.713.318.200
11.155.913.978.494.600
2.287.390.029.325.510
10.317.460.317.460.300
3.834.416.223.666.070
22.727.272.727.272.700
5.369.565.217.391.300
9.677.419.354.838.700
57.017.543.859.649.100
2.608.503.100.088.570
2.271.006.813.020.430
5.202.312.138.728.320
52.424.545.844.657.400
2.867.570.385.818.560
5.101.851.851.851.850
5.429.417.571.569.590
11.794.500.723.589.000
18.333.333.333.333.300
3.078.556.263.269.630
13.796.849.538.294.400
5.523.429.710.867.390
30.805.687.203.791.400
6.001.690.617.075.230
```

```
3.838.771.593.090.210
33.738.191.632.928.400
3.245.686.113.393.590
3.703.703.703.703.700
6.719.056.974.459.720
4.545.454.545.454.540
4.953.429.297.205.750
31.758.056.758.056.700
4.973.221.117.061.970
5.272.013.460.459.890
45.243.128.964.059.100
4.292.929.292.929.290
11.920.529.801.324.500
10.508.474.576.271.100
18.905.253.901.956.400
7.886.435.331.230.280
16.949.152.542.372.800
6.493.506.493.506.490
2.010.487.059.417.320
52.287.581.699.346.400
14.511.873.350.923.400
57.926.829.268.292.600
17.543.859.649.122.800
33.898.305.084.745.700
25.333.333.333.333.300
1.485.148.514.851.480
18.569.087.930.092.800
8.695.652.173.913.040
21.978.021.978.021.900
24.081.115.335.868.100
14.957.264.957.264.900
16.938.039.558.518.100
4.291.845.493.562.230
6.785.714.285.714.280
7.547.169.811.320.750
7.574.884.792.626.720
23.525.280.898.876.400
2.883.720.930.232.550
4.199.855.177.407.670
34.981.905.910.735.800
Number of noisy ctr string values are: 81
```

[23]:
```
# Checking the product_tier at noisy indices
data_mod.loc[index_noisy_str, 'product_tier'].value_counts()
```

[23]:
```
Basic      77
Plus        2
```

```
Premium      2
Name: product_tier, dtype: int64
```

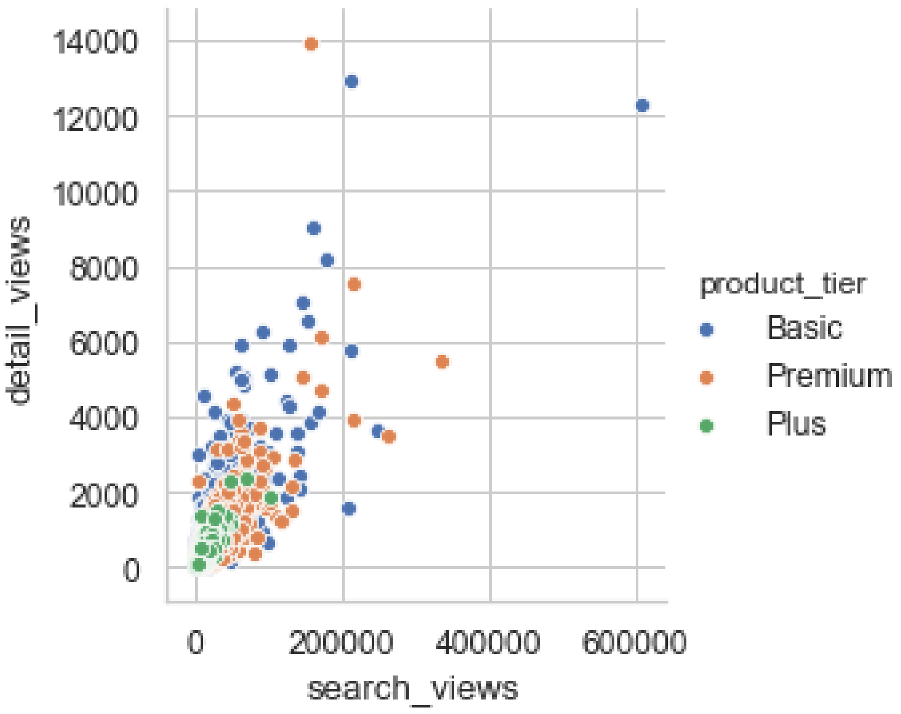
For an easy workaround, I will just remove this ctr coloumn since it is computed from the search_views and detail_views - the coloumns which we have already cleaned. The coloumn ctr is quite noisy. Even if I calculate the above noisy ctr's myself, there are still some weird entries like '01. Apr' which would require more time. So, let's just remove this ctr coloumn and proceed.

Removing the rows with the noisy values would remove some premium and plus product tiers which are already in minority.

[24]:
```
# Removing the ctr coloumn as well as the deleted_date and search_date coloumns␣
↪from the dataset
data_mod_back = data_mod.copy()
data_mod_updated = data_mod.drop(['ctr','created_date', 'deleted_date'], axis=1)
```

[25]:
```
# Conditional plot based on the product tier - search_views vs detail_views
sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
↪scatterplot, 'search_views', 'detail_views').add_legend()
```

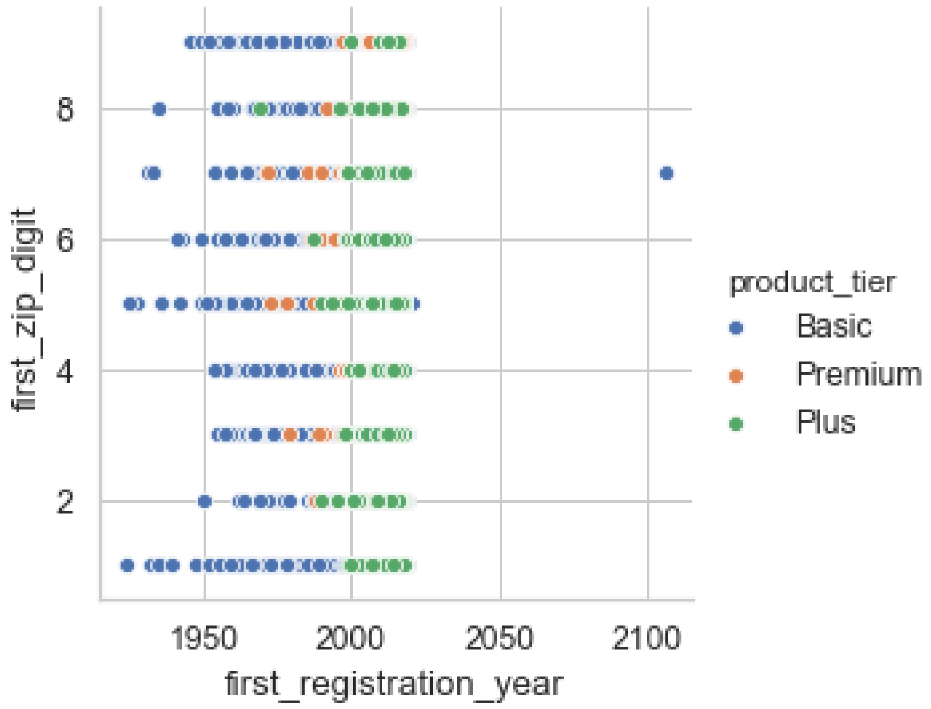
[25]:
```
<seaborn.axisgrid.FacetGrid at 0x1a25647690>
```



[26]:
```
# Conditional plot based on the product tier - first_registeration_year vs␣
↪first_zip_digit
```

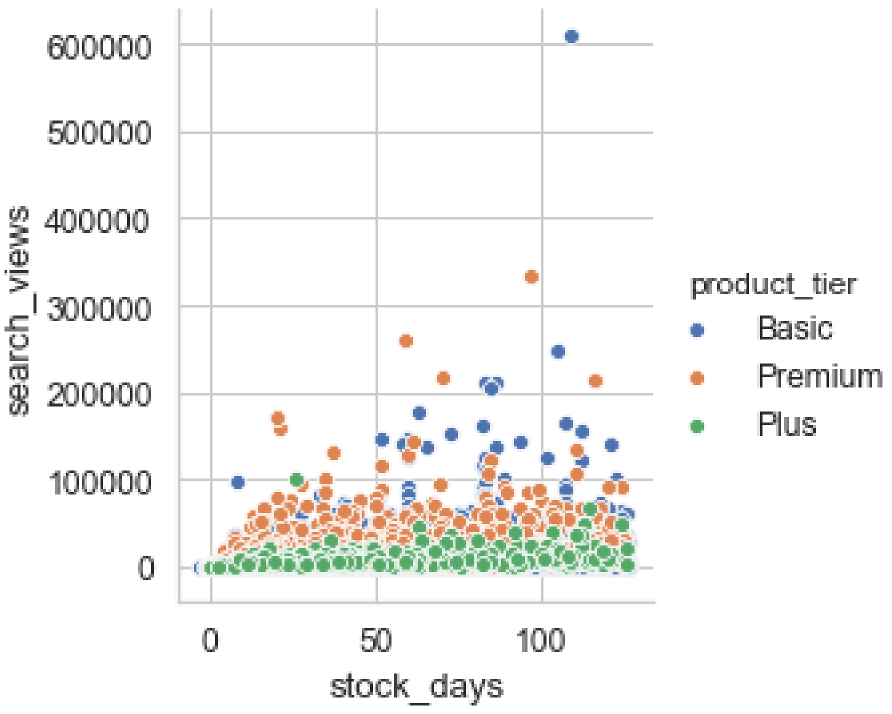
```
sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
↪scatterplot, 'first_registration_year', 'first_zip_digit').add_legend()
```

```
[26]: <seaborn.axisgrid.FacetGrid at 0x1a2770aed0>
```



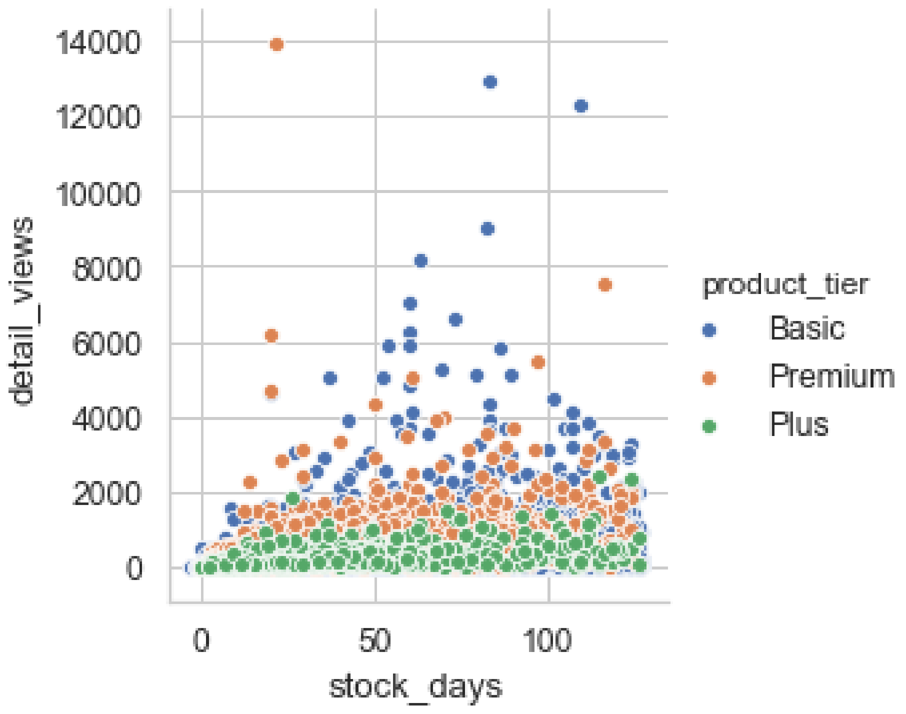
```
[27]: # Conditional plot based on the product tier - stock_days vs search_views
sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
↪scatterplot, 'stock_days', 'search_views').add_legend()
```

```
[27]: <seaborn.axisgrid.FacetGrid at 0x1a2a5b8710>
```



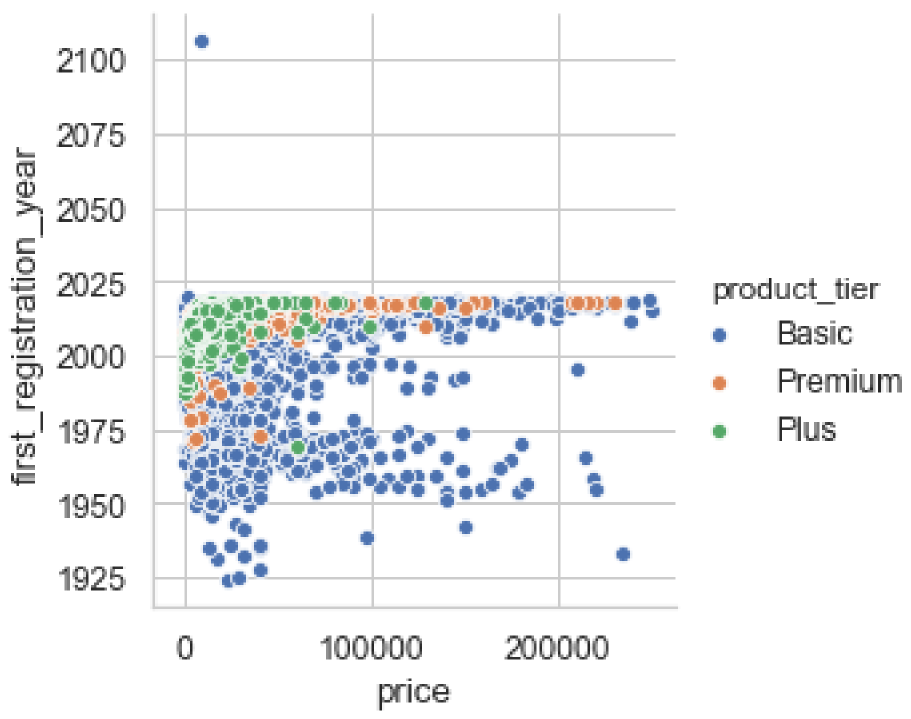
```
[28]: # Conditional plot based on the product tier - stock_days vs detail_views
      sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
       ↪scatterplot, 'stock_days', 'detail_views').add_legend()
```

```
[28]: <seaborn.axisgrid.FacetGrid at 0x1a27850450>
```



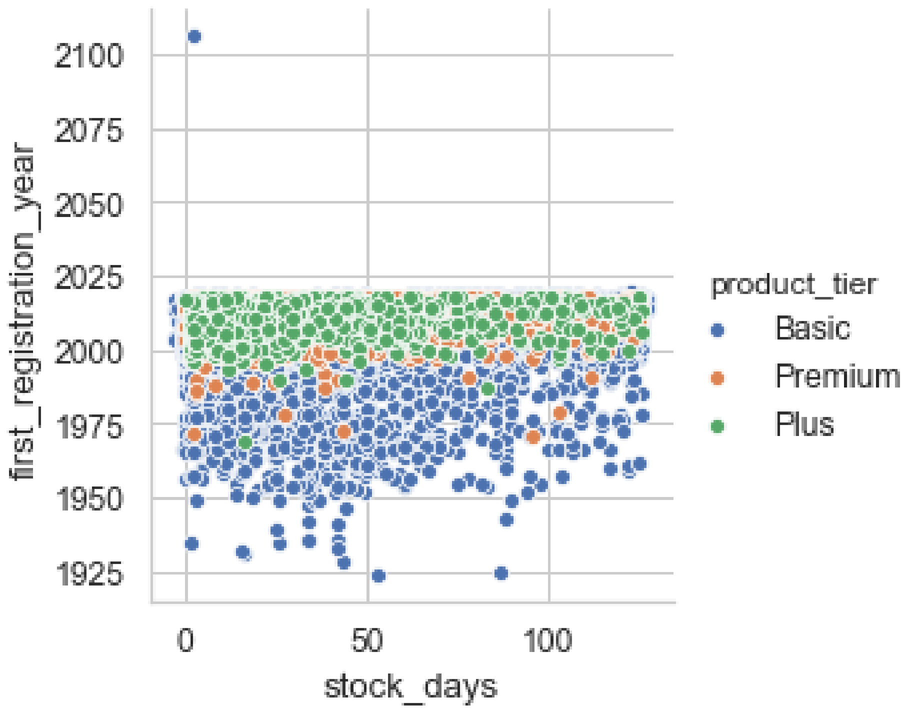
```
[29]: # Conditional plot based on the product tier - first_registration_year vs price
      sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
       ↪scatterplot, 'price', 'first_registration_year').add_legend()
```

```
[29]: <seaborn.axisgrid.FacetGrid at 0x1a27c85e90>
```



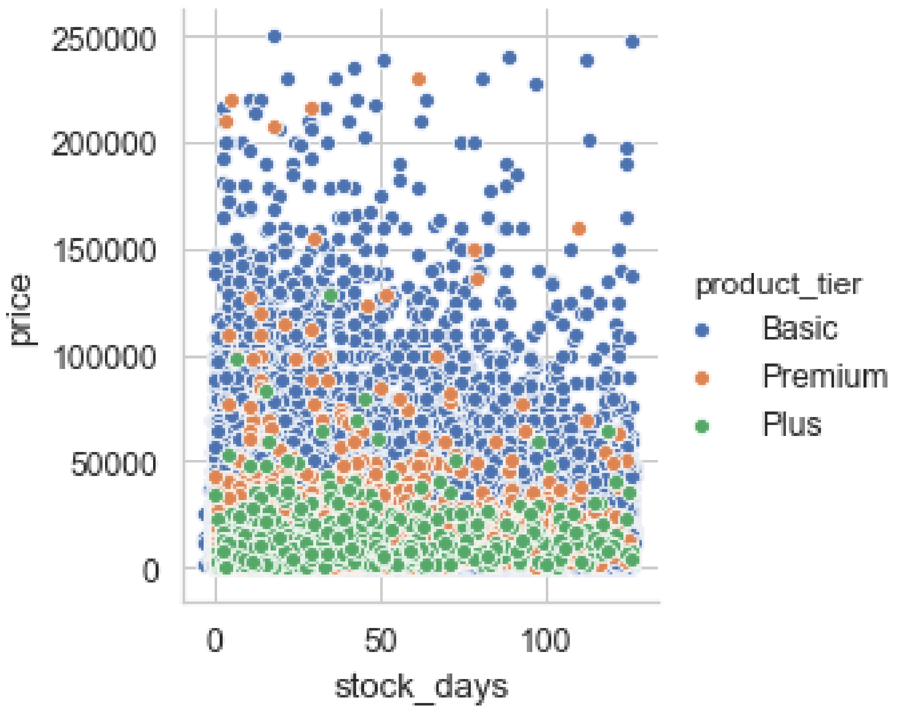
```
[30]: # Conditional plot based on the product tier - stock_days vs␣
       ↪first_registration_year
      sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
       ↪scatterplot, 'stock_days', 'first_registration_year').add_legend()
```

```
[30]: <seaborn.axisgrid.FacetGrid at 0x1a2585a6d0>
```



```
[31]: # Conditional plot based on the product tier - stock_days vs price
      sns.FacetGrid(data_mod_updated, hue='product_tier', height=4).map(sns.
       ↪scatterplot, 'stock_days', 'price').add_legend()
```

```
[31]: <seaborn.axisgrid.FacetGrid at 0x1a25a0fdd0>
```



From the conditional plots given above, we can observe that the three product tier types are overlapping. The three classes do not seem to be easily separable. In addition, due to a huge misbalance in the data, even if we use strong algorithm like Neural networks for solving this task, we would still have problems due to the misbalance.

Additionally, their are some outliers in the different variables which we can either remove or just cap it within the 95th percentile. Let's cap it for now.

Note: I could have plotted pair plots for looking at all the 2D combinations of features but it was taking an infinite amount of time on my machine. So, I just plotted scatter plots for some of the features combinations.

[113]:
```
# Capping the values in the numerical coloumns
num_data_col = ['price', 'first_registration_year', 'detail_views',␣
 ↪'search_views', 'stock_days', 'first_zip_digit']

for col in num_data_col:
    upper_lim = data_mod_updated[col].quantile(.95)
    lower_lim = data_mod_updated[col].quantile(.05)
    data_mod_updated.loc[(data_mod_updated[col] > upper_lim),col] = upper_lim
    data_mod_updated.loc[(data_mod_updated[col] < lower_lim),col] = lower_lim
```

[114]:
```
# Correlation between make_name and product_tier
corr_tier_make = cramers_v(data_mod_updated.make_name, data_mod_updated.
 ↪product_tier)
```

```
print('The value of the correlation between make_name and product_tier is: {}'.
↪format(corr_tier_make))
```

```
The value of the correlation between make_name and product_tier is:
0.05460957919605956
```

Seems like the correlation between the make_name variable and our target product_tier is very low.

Based on the 2D plots above, it seems that it is difficult to predict the product_tiers from the other variables. Atleast from these 2D plots, we can observe that the classes can not be separated linearly thus if we have to make try on checking whether this classification can work well or not, we should go with a non-linear algorithm and use models like LogisticRegression, non-linear SVMs or MLP. Additionally, we should also try a Boosting algorithm like GradientBoosting which helps with misbalanced dataset. We could give a try on the RandomForests as well but it seems like the underlying decision trees might not have that strong representation for this classification task and would easily overfit.

I think it would be better to remove the variable 'first_zip_digit' as well since it has a very high frequency distribution

[215]:
```
# Necessary import for modelling
from sklearn.preprocessing import LabelEncoder, StandardScaler, OneHotEncoder
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import GradientBoostingClassifier
```

[191]:
```
# Separating the data into the predictor and target variables
y = data_mod_updated['product_tier']
X = data_mod_updated.drop(['product_tier', 'first_zip_digit'], axis=1)
```

[192]:
```
# Encoding the categorical labels
# 0 - Basic, 1 - Plus, 2 - Premium
enc_label = LabelEncoder()
y = enc_label.fit_transform(y)
```

[193]:
```
# Making a coloumn transformer
num_data_col_all = ['price', 'first_registration_year', 'detail_views',␣
↪'search_views', 'stock_days']
column_trans_all = make_column_transformer((OneHotEncoder(), ['make_name']),␣
↪(StandardScaler(), num_data_col_all))
```

[233]:
```
# Creating different models
modellog = LogisticRegression()
modelsvc = SVC()
modelboost = GradientBoostingClassifier()
```

[195]:
```
# Making pipelines for different models
pipelog = make_pipeline(column_trans_all, modellog)
pipesvc = make_pipeline(column_trans_all, modelsvc)
```

[234]:
```
# Making pipeline for gradient boosting
pipeboost = make_pipeline(column_trans_all, modelboost)
```

I will do the training as well as the evaluation on whole of the dataset instead of splitting using train_test_split, because we are just interested in observing whether we can predict the product_tier or not

[197]:
```
# Fitting the pipeline for LogisticRegression
pipelog.fit(X, y)
```

[197]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('onehotencoder',
                                                  OneHotEncoder(),
                                                  ['make_name']),
                                                 ('standardscaler',
                                                  StandardScaler(),
                                                  ['price',
                                                   'first_registration_year',
                                                   'detail_views',
                                                   'search_views',
                                                   'stock_days'])])),
                ('logisticregression', LogisticRegression())])
```

[198]:
```
# Fitting the pipeline for SVM
pipesvc.fit(X, y)
```

[198]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('onehotencoder',
                                                  OneHotEncoder(),
                                                  ['make_name']),
                                                 ('standardscaler',
                                                  StandardScaler(),
                                                  ['price',
                                                   'first_registration_year',
                                                   'detail_views',
                                                   'search_views',
                                                   'stock_days'])])),
                ('svc', SVC())])
```

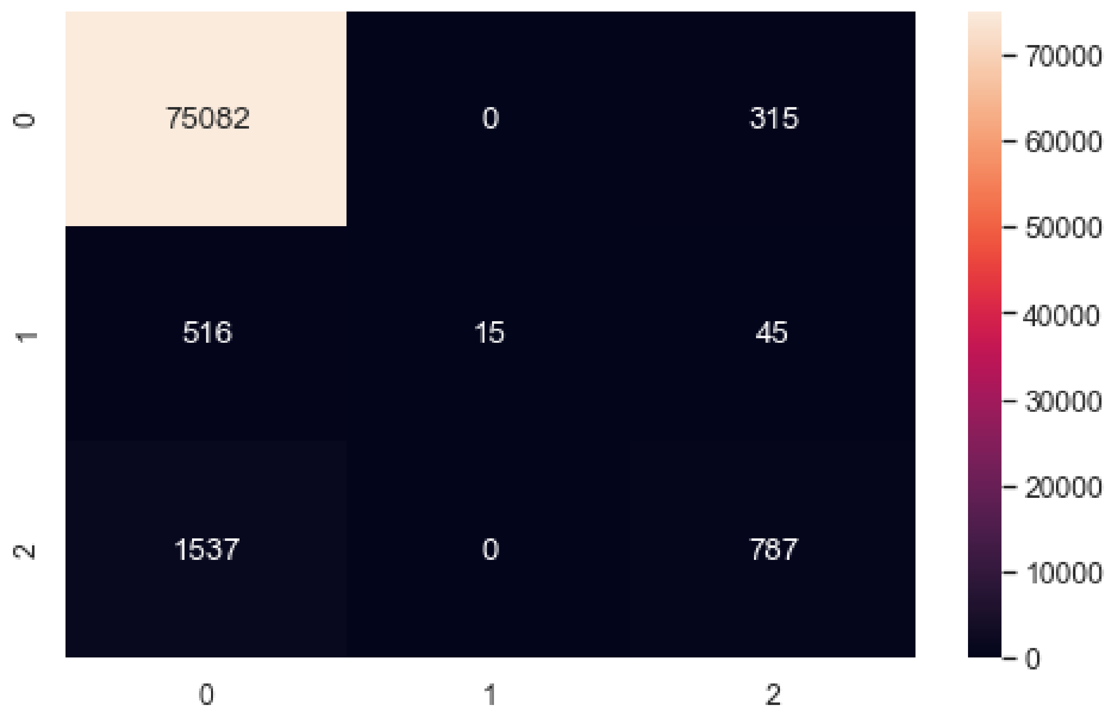
[235]:
```
pipeboost.fit(X, y)
```

[235]:
```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('onehotencoder',
                                                  OneHotEncoder(),
                                                  ['make_name']),
                                                 ('standardscaler',
                                                  StandardScaler(),
                                                  ['price',
                                                   'first_registration_year',
                                                   'detail_views',
                                                   'search_views',
                                                   'stock_days'])])),
                ('gradientboostingclassifier', GradientBoostingClassifier())])
```

[236]:
```
# Getting the predictions on the test set
predlog = pipelog.predict(X)
predsvc = pipesvc.predict(X)
predboost = pipeboost.predict(X)
```

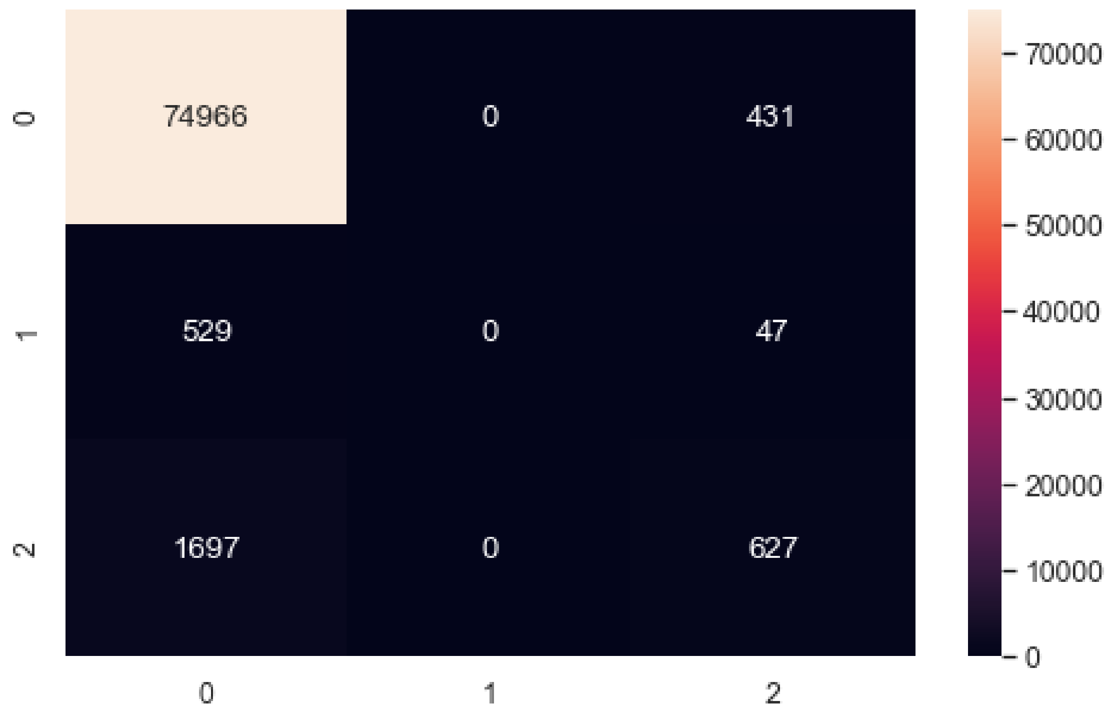
[237]:
```
# Confusion matrix obtained for gradient boosting
sns.heatmap(confusion_matrix(y, predboost), annot=True, fmt='g')
```

[237]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a3c4e50d0>
```



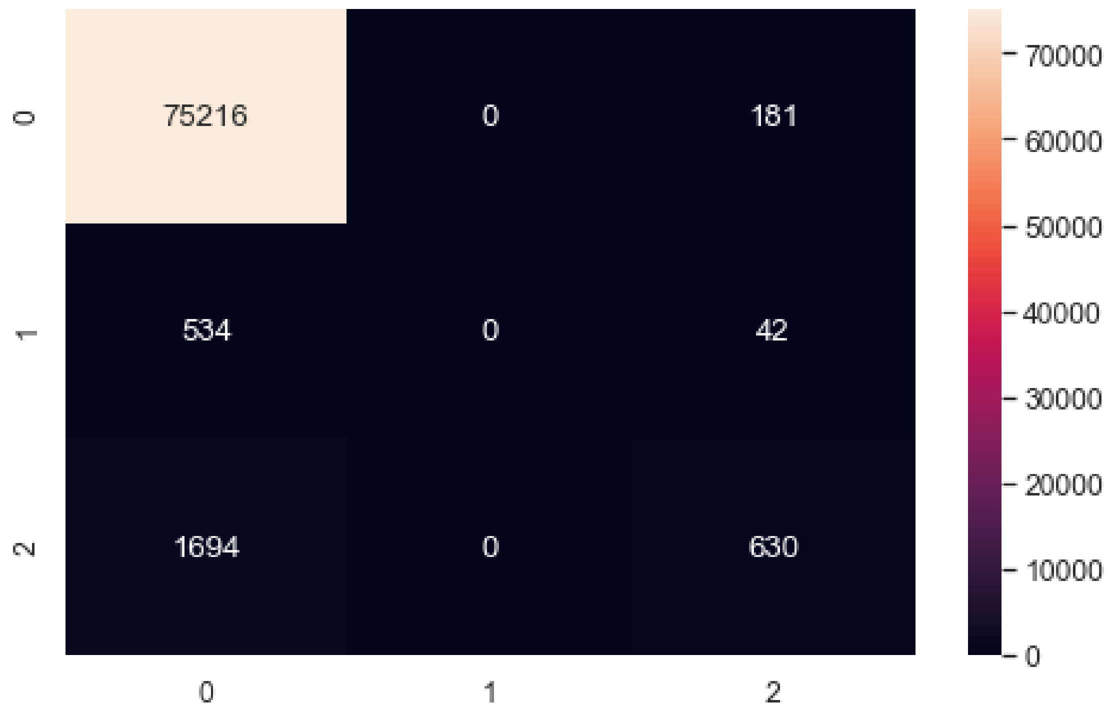
```
[205]: # Confusion matrix obtained for LogisticRegression
       sns.heatmap(confusion_matrix(y, predlog), annot=True, fmt='g')
```

```
[205]: <matplotlib.axes._subplots.AxesSubplot at 0x1a397b7f10>
```



```
[206]: # Confusion matrix obtained for SVM
       sns.heatmap(confusion_matrix(y, predsvc), annot=True, fmt='g')
```

```
[206]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3979bb50>
```



0 - Basic, 1 - Plus, 2 - Premium

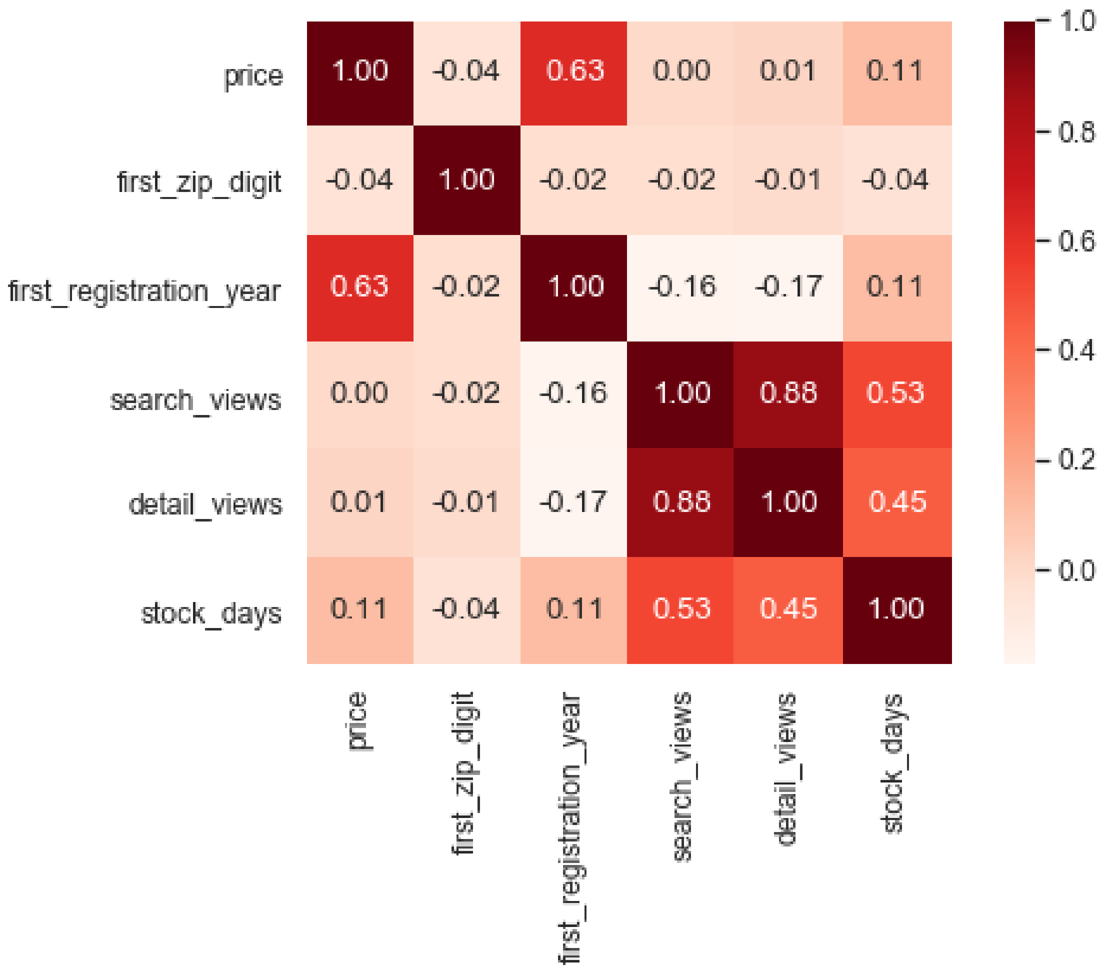
From the above three confusion matrices obtained with LogisticRegression, nonlinear SVM and GradientBoosting respectively, it is clear that it is difficult to predict the product_tier. The main reason for that is the misbalanced dataset. As observed from the plots above, we do have variables that follow a nice distribution and could help with this prediction task, only if we can collect more dataset. I believe that we should see better results if 1) we downsample the basic class and make it balanced with the other classes and 2) if we collect more data.

Gradient Boosting performs the best in terms of the two classes which the other algorithms are able to identify as well!

Let's move to the regression task now and look if it is possible to predict the detail_views variable.

```
[207]: # Creating the correlation matrix for the numerical variable
       dataCorr = data_mod_updated.drop(['product_tier', 'make_name'], axis=1).corr()
       sns.heatmap(dataCorr, annot=True, fmt='.2f', square=True, cmap = 'Reds')
```

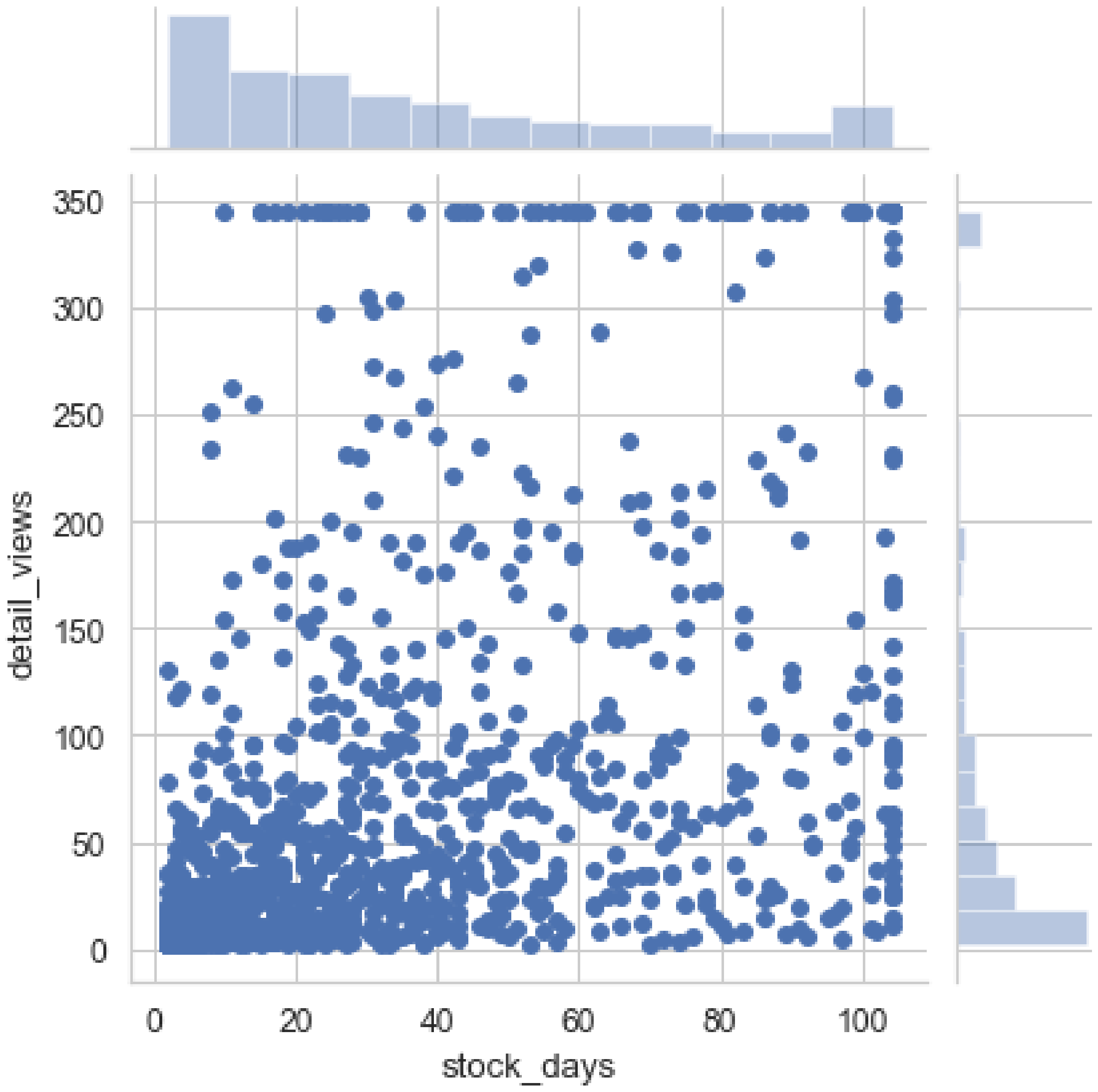
```
[207]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3010e890>
```



From the above matrix, it seems like the stock_days, search_views are highly correlated with our target variable i.e. detail views. But the variables stock_days and search_views are itself highly correlated as well. So it would be better to remove one of them. I think I would go with removing stock_days since if we look at other predictor variables like e.g. Price, we see more correlation with stock_days as compared to search_views. We could also remove first_registration_year since it offers pretty high negative correlation. Let's try plotting both of these variables now.

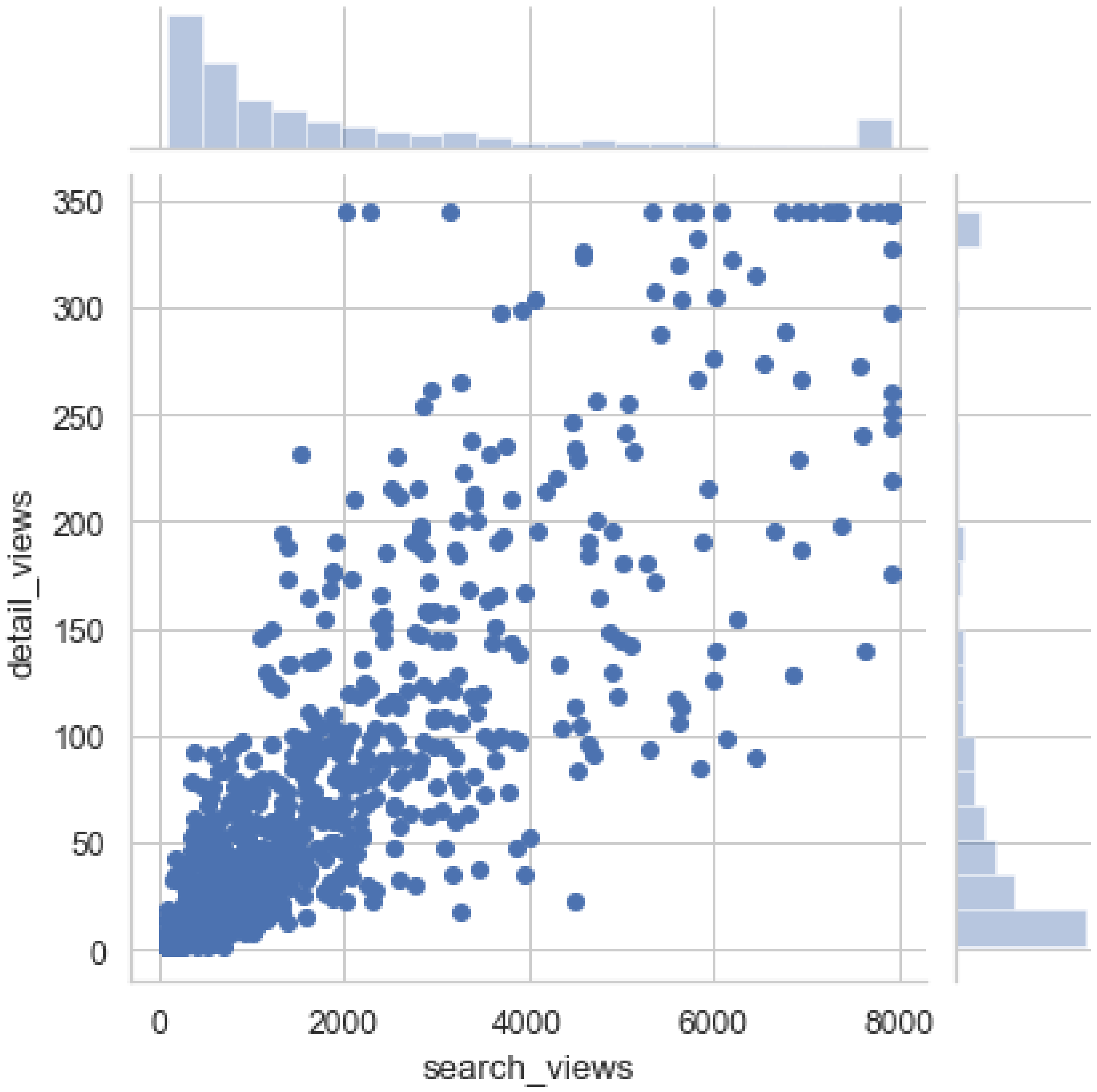
```
[211]: # Detail views vs Stock days
       sns.jointplot(x=data_mod_updated['stock_days'][:1000],␣
        ↪y=data_mod_updated['detail_views'][:1000])
```

```
[211]: <seaborn.axisgrid.JointGrid at 0x1a3759af50>
```



```
[212]: # Detail views vs Search views
       sns.jointplot(x=data_mod_updated['search_views'][:1000],␣
        ↪y=data_mod_updated['detail_views'][:1000])
```

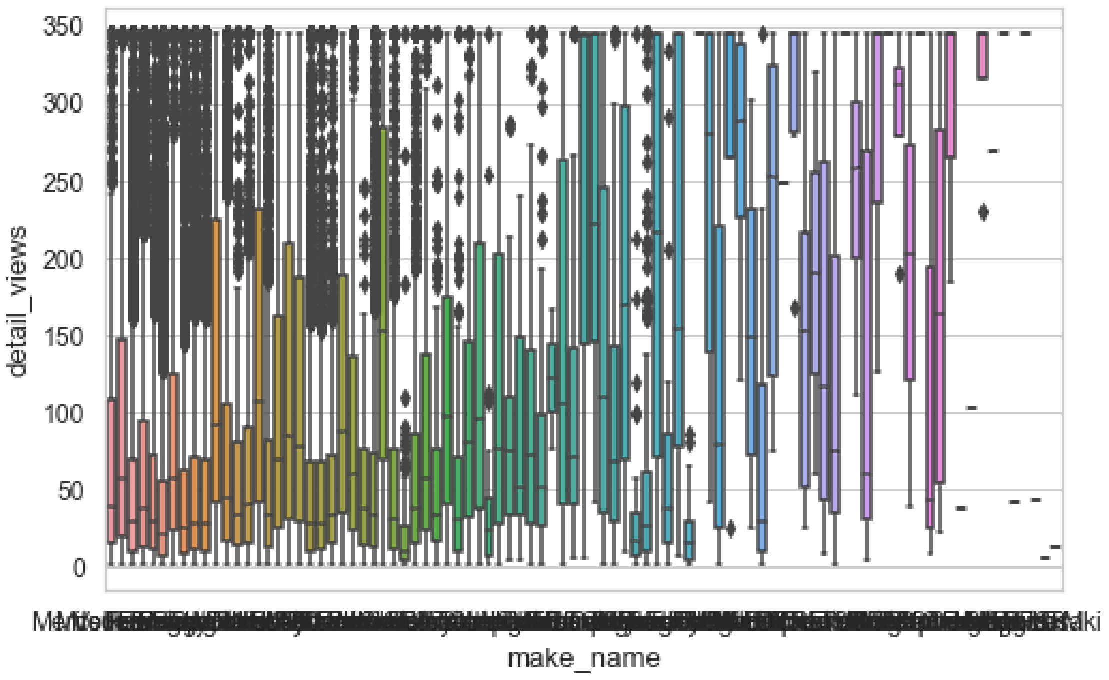
```
[212]: <seaborn.axisgrid.JointGrid at 0x1a37575590>
```



Clearly, search_views seems to be better than stock_days!

```
[213]: sns.boxplot(x='make_name', y='detail_views', data=data_mod_updated)
```

```
[213]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3ad43450>
```



From the above plot we can see that we have a pretty high variance for different make_names and their associated detail_views. I think it should be better to remove this column for our analysis.

Additionally, the variable first zip digit could be removed as well because it shows a bit of negative correlation too.

Based on the analysis above, we have only strong predictor variable which is search_views. The price variable has very less correlation with the detail_views. One might be thinking of the ctr variable right now but I suspect that it would have a very high correlation with the search_views too. So, we couldn't use it anyway.

I think, it is difficult to predict the detail_views from only one strong predictor variable. But, for trying purposes, we can use algorithms like LinearRegression, GradientBoostingRegressor and SupportVectorRegression.

I will show the results with LinearRegression and GradientBoosting only because SupportVector-Regression was causing hanging issues on my machine. I could have trained it on Google co-lab however, I was not sure if it is okay to upload the data online.

[322]:
```python
# Necessary sklearn imports
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import GradientBoostingRegressor
```

[323]:
```python
# Preparing the data
y_reg = data_mod_updated['detail_views']
X_reg = data_mod_updated.drop(['make_name', 'stock_days',␣
 ↪'first_registration_year', 'detail_views', 'first_zip_digit'], axis=1)
```

[289]:
```
# Column transformer
num_col_reg = ['search_views', 'price']
column_trans_reg = make_column_transformer((StandardScaler(), num_col_reg))
```

[329]:
```
# Model definitions
modelreg = LinearRegression()
modelboost = GradientBoostingRegressor()
```

[330]:
```
# Preparing the pipelines
pipereg = make_pipeline(column_trans_reg, modelreg)
pipeboost = make_pipeline(column_trans_reg, modelboost)
```

[331]:
```
# Cross validating the model for getting average RMSE
rmsereg = math.sqrt(abs(cross_val_score(pipereg, X_reg, y_reg, cv=4,␣
 ↪scoring='neg_mean_squared_error').mean()))
```

[327]:
```
# Cross validating the model for getting average RMSE
rmseboosting = math.sqrt(abs(cross_val_score(pipeboost, X_reg, y_reg, cv=4,␣
 ↪scoring='neg_mean_squared_error').mean()))
```

[332]:
```
print('RMSE with Linear regression is: {}'.format(rmsereg))
print('RMSE with Gradient Boosting is: {}'.format(rmseboosting))
```

```
RMSE with Linear regression is: 43.134995852276575
RMSE with Gradient Boosting is: 42.59528739020323
```

We can see that we are getting a very high error after cross validating the model which basically agree with our initial belief that this regression task might not work well. Predicting the target variable with only one strongly correlating variable always poses challenges for data with difficult patterns.

Final conclusion:

Predicting product tier: Difficult because of misbalances Predicting detail views: Difficult because of the lack of strong predictor variables.