Feature Engineering and Modelling

1. Import packages
2. Load data
3. Feature engineering
4. Build Predictive Models

1. Import packages

In [1]:
```
!pip3 install imblearn
```

```
Requirement already satisfied: imblearn in c:\users\lenovo\anaconda3\lib\site-packages (0.0)
Requirement already satisfied: imbalanced-learn in c:\users\lenovo\anaconda3\lib\site-packages (fr
om imblearn) (0.10.1)
Requirement already satisfied: joblib>=1.1.1 in c:\users\lenovo\anaconda3\lib\site-packages (from
imbalanced-learn->imblearn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\lenovo\anaconda3\lib\site-packages
(from imbalanced-learn->imblearn) (2.2.0)
Requirement already satisfied: numpy>=1.17.3 in c:\users\lenovo\anaconda3\lib\site-packages (from
imbalanced-learn->imblearn) (1.21.5)
Requirement already satisfied: scipy>=1.3.2 in c:\users\lenovo\anaconda3\lib\site-packages (from i
mbalanced-learn->imblearn) (1.9.1)
Requirement already satisfied: scikit-learn>=1.0.2 in c:\users\lenovo\anaconda3\lib\site-packages
(from imbalanced-learn->imblearn) (1.1.2)
```

In [2]:
```
#import the necessary libraries
import pandas as pd #data processing
import numpy as np # linear algebra
import matplotlib.pyplot as plt #data plot
%matplotlib inline
import seaborn as sns #data plot
import warnings
warnings.filterwarnings("ignore") #ignore warning
```

2. Load data

In [3]:
```
## Loading the cleaned data
date_cols=['date_activ','date_end','date_modif_prod','date_renewal']
df = pd.read_csv(r"C:\Users\lenovo\Desktop\BCG Internship\Task 2\clean_data_modeling.csv",parse_date
df.head()
```

Out[3]:

	Unnamed: 0	id	channel_sales	cons_12m	cons_gas_12m	cons_last_r
0	0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	54946	
1	1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	0	
2	2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	0	
3	3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	0	
4	4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	0	

5 rows × 54 columns

This data consists of PowerCo's churn information which mean customer will switch to another provider or not. Scroll to the right if necessary, and note that the final column in the dataset (Churn) contains the value **0** for customer who doesn't churn, and **1** for patients who churns. This is the label that we will train our model to predict; other columns are the features we will use to predict the Churn label.

As usual, I'll train a **binary classifier** to predict whether or not a customer should be churn based on some historical data. But according to task requirements, a **Random Forest classifier** will used to predict customer churn and evaluate the performance of the model with suitable evaluation metrics.

For tree-based models, there is no need to perform data scaling, unlike other machine learning models. And dont need to apply one-hot encoding to processing categorical variables, just use label-encoding to convert categorical variables to numeric variables.

In [4]:
```
df.drop(columns = 'Unnamed: 0', inplace=True) #drop the Unnamed column
```

In [5]:
```
df.shape #check the shape of the dataset
```

Out[5]:
```
(14605, 53)
```

There are 53 columns and 14605 rows.

In [6]:
```
df.dtypes
```

Out[6]:
```
id                                 object
channel_sales                      object
cons_12m                            int64
cons_gas_12m                        int64
cons_last_month                     int64
date_activ                 datetime64[ns]
date_end                   datetime64[ns]
date_modif_prod            datetime64[ns]
date_renewal               datetime64[ns]
forecast_cons_12m                 float64
forecast_cons_year                  int64
forecast_discount_energy          float64
forecast_meter_rent_12m           float64
forecast_price_energy_off_peak    float64
forecast_price_energy_peak        float64
forecast_price_pow_off_peak       float64
has_gas                            object
imp_cons                          float64
margin_gross_pow_ele              float64
margin_net_pow_ele                float64
nb_prod_act                         int64
net_margin                        float64
num_years_antig                     int64
origin_up                          object
pow_max                           float64
mean_year_price_off_peak_var      float64
mean_year_price_peak_var          float64
mean_year_price_mid_peak_var      float64
mean_year_price_off_peak_fix      float64
mean_year_price_peak_fix          float64
mean_year_price_mid_peak_fix      float64
mean_year_price_off_peak          float64
mean_year_price_peak              float64
mean_year_price_med_peak          float64
mean_6m_price_off_peak_var        float64
mean_6m_price_peak_var            float64
mean_6m_price_mid_peak_var        float64
mean_6m_price_off_peak_fix        float64
mean_6m_price_peak_fix            float64
mean_6m_price_mid_peak_fix        float64
mean_6m_price_off_peak            float64
mean_6m_price_peak                float64
mean_6m_price_med_peak            float64
mean_3m_price_off_peak_var        float64
mean_3m_price_peak_var            float64
mean_3m_price_mid_peak_var        float64
mean_3m_price_off_peak_fix        float64
mean_3m_price_peak_fix            float64
mean_3m_price_mid_peak_fix        float64
mean_3m_price_off_peak            float64
mean_3m_price_peak                float64
mean_3m_price_med_peak            float64
churn                               int64
dtype: object
```

3. Feature engineering

Difference between off-peak prices in December and preceding January

Below is the code created by your colleague to calculate the feature described above. Use this code to re-create this feature and then think about ways to build on this feature to create features with a higher predictive power.

In [7]:
```
price_df = pd.read_csv(r"C:\Users\lenovo\Desktop\BCG Internship\Task 2\Resources\4.2 Price Data.csv'
price_df["price_date"] = pd.to_datetime(price_df["price_date"], format='%Y-%m-%d')
price_df.head()
```

Out[7]:

	id	price_date	price_off_peak_var	price_peak_var	price_mid_peak_var	price_off_peak_fi
0	038af19179925da21a25619c5a24b745	2015-01-01	0.151367	0.0	0.0	44.26693
1	038af19179925da21a25619c5a24b745	2015-02-01	0.151367	0.0	0.0	44.26693
2	038af19179925da21a25619c5a24b745	2015-03-01	0.151367	0.0	0.0	44.26693
3	038af19179925da21a25619c5a24b745	2015-04-01	0.149626	0.0	0.0	44.26693
4	038af19179925da21a25619c5a24b745	2015-05-01	0.149626	0.0	0.0	44.26693

In [8]:
```
# Group off-peak prices by companies and month
monthly_price_by_id = price_df.groupby(['id', 'price_date']).agg({'price_off_peak_var': 'mean', 'pr

# Get january and december prices
jan_prices = monthly_price_by_id.groupby('id').first().reset_index()
dec_prices = monthly_price_by_id.groupby('id').last().reset_index()

# Calculate the difference
diff = pd.merge(dec_prices.rename(columns={'price_off_peak_var': 'dec_1', 'price_off_peak_fix': 'de
diff['offpeak_diff_dec_january_energy'] = diff['dec_1'] - diff['price_off_peak_var']
diff['offpeak_diff_dec_january_power'] = diff['dec_2'] - diff['price_off_peak_fix']
diff = diff[['id', 'offpeak_diff_dec_january_energy','offpeak_diff_dec_january_power']]
diff.head()
```

Out[8]:

	id	offpeak_diff_dec_january_energy	offpeak_diff_dec_january_power
0	0002203ffbb812588b632b9e628cc38d	-0.006192	0.162916
1	0004351ebdd665e6ee664792efc4fd13	-0.004104	0.177779
2	0010bcc39e42b3c2131ed2ce55246e3c	0.050443	1.500000
3	0010ee3855fdea87602a5b7aba8e42de	-0.010018	0.162916
4	00114d74e963e47177db89bc70108537	-0.003994	-0.000001

In [9]:
```
monthly_price_by_id.head()
```

Out[9]:

	id	price_date	price_off_peak_var	price_off_peak_fix
0	0002203ffbb812588b632b9e628cc38d	2015-01-01	0.126098	40.565969
1	0002203ffbb812588b632b9e628cc38d	2015-02-01	0.126098	40.565969
2	0002203ffbb812588b632b9e628cc38d	2015-03-01	0.128067	40.728885
3	0002203ffbb812588b632b9e628cc38d	2015-04-01	0.128067	40.728885
4	0002203ffbb812588b632b9e628cc38d	2015-05-01	0.128067	40.728885

In [10]:
```
jan_prices.head()
```

Out[10]:

	id	price_date	price_off_peak_var	price_off_peak_fix
0	0002203ffbb812588b632b9e628cc38d	2015-01-01	0.126098	40.565969
1	0004351ebdd665e6ee664792efc4fd13	2015-01-01	0.148047	44.266931
2	0010bcc39e42b3c2131ed2ce55246e3c	2015-01-01	0.150837	44.444710
3	0010ee3855fdea87602a5b7aba8e42de	2015-01-01	0.123086	40.565969
4	00114d74e963e47177db89bc70108537	2015-01-01	0.149434	44.266931

In [11]:
```
df = pd.merge(df, diff, on='id')
df.head()
```

Out[11]:

	id	channel_sales	cons_12m	cons_gas_12m	cons_last_month	date
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	54946	0	20
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	0	0	20
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	0	0	20
3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	0	0	20
4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	0	526	20

5 rows × 55 columns

In the dataset, we have some datetime features:

- date_activ = date of activation of the contract
- date_end = registered date of the end of the contract
- date_modif_prod = date of the last modification of the product
- date_renewal = date of the next contract renewal From these features, we can create new columns called:
- tenure: the time customer uses service of PowerCo
- months_activ = Number of months active until reference date (Jan 2016)
- months_to_end = Number of months of the contract left until reference date (Jan 2016)
- months_modif_prod = Number of months since last modification until reference date (Jan 2016)
- months_renewal = Number of months since last renewal until reference date (Jan 2016)

In [12]:
```
# We no longer need the datetime columns so we can drop them

remove = ['date_activ', 'date_end', 'date_modif_prod', 'date_renewal']

df = df.drop(columns=remove)
df.head()
```

Out[12]:

	id	channel_sales	cons_12m	cons_gas_12m	cons_last_month	forec
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	54946	0	
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	0	0	
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	0	0	
3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	0	0	
4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	0	526	

5 rows × 51 columns

Transform Categorical Data

We have 3 categorical variables including: has_gas, channel_sales, origin_up.

In [13]:
```
#For the column has_gas, replace t for 1 and f for 0
df['has_gas']=df['has_gas'].replace(['t','f'],[1,0])
```

Let's check values of the two others.

In [14]:
```
df['channel_sales'].value_counts()
```

Out[14]:
```
foosdfpfkusacimwkcsosbicdxkicaua    6753
MISSING                             3725
lmkebamcaaclubfxadlmueccxoimlema    1843
usilxuppasemubllopkaafesmlibmsdf    1375
ewpakwlliwisiwduibdlfmalxowmwpci     893
sddiedcslfslkckwlfkdpoeeailfpeds      11
epumfxlbckeskwekxbiuasklxalciiuu       3
fixdbufsefwooaasfcxdxadsiekoceaa       2
Name: channel_sales, dtype: int64
```

In [15]:
```
df['origin_up'].value_counts()
```

Out[15]:
```
lxidpiddsbxsbosboudacockeimpuepw    7096
kamkkxfxxuwbdslkwifmmcsiusiuosws    4294
ldkssxwpmemidmecebumciepifcamkci    3148
MISSING                               64
usapbepcfoloekilkwsdiboslwaxobdp       2
ewxeelcelemmiwuafmddpobolfuxioce       1
Name: origin_up, dtype: int64
```

There is imbalance in the number of values for channel_sales and origin_up so we will remove the values with very little occurences comparing with others.

In [16]:
```
#One hot encoding using get_dummies()
df = pd.get_dummies(df, columns = ['channel_sales', 'origin_up'])
```

In [17]:
```
df.head()
```

Out[17]:

	id	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_cons_year
0	24011ae4ebbe3035111d65fa7c15bc57	0	54946	0	0.00	0
1	d29c2c54acc38ff3c0614d0a653813dd	4660	0	0	189.95	0
2	764c75f661154dac3a6c254cd082ea7d	544	0	0	47.96	0
3	bba03439a292a1e166f80264c16191cb	1584	0	0	240.04	0
4	149d57cf92fc41cf94415803a877cb4b	4425	0	526	445.75	526

5 rows × 63 columns

In [18]:
```
#remove values with very little occurences
df = df.drop(columns=['channel_sales_sddiedcslfslkckwlfkdpoeeailfpeds', 'channel_sales_epumfxlbckesl
                      'origin_up_MISSING', 'origin_up_usapbepcfoloekilkwsdiboslwaxobdp', 'origin_up_e
```

In [19]:
```
df.head()
```

Out[19]:

	id	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_cons_year
0	24011ae4ebbe3035111d65fa7c15bc57	0	54946	0	0.00	0
1	d29c2c54acc38ff3c0614d0a653813dd	4660	0	0	189.95	0
2	764c75f661154dac3a6c254cd082ea7d	544	0	0	47.96	0
3	bba03439a292a1e166f80264c16191cb	1584	0	0	240.04	0
4	149d57cf92fc41cf94415803a877cb4b	4425	0	526	445.75	526

5 rows × 57 columns

Transform Numerical Data

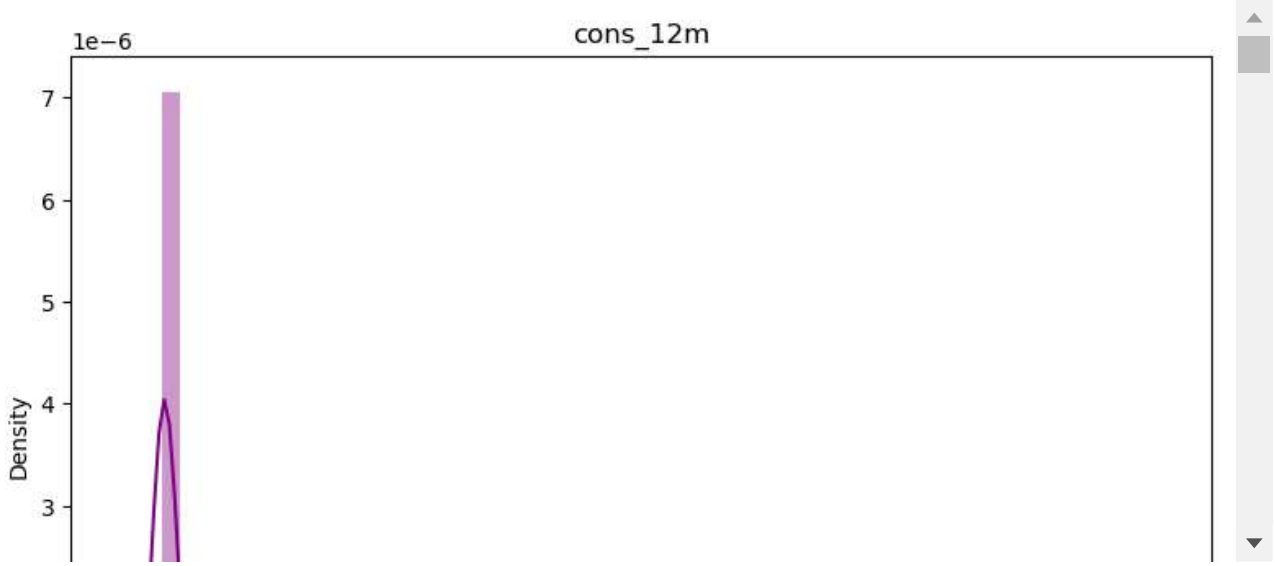
From the previous EDA we can see that some features are highly skewed, we need to transform the distribution to normal-like distribution. We will use log transfomation via Numpy by calling the log() function on the desired column. First, we will check skewed features, so that we can compare before and after transformation

In [20]:
```
numeric_features = df[[
    'cons_12m',
    'cons_gas_12m',
    'cons_last_month',
    'forecast_cons_12m',
    'forecast_cons_year',
    'forecast_discount_energy',
    'forecast_meter_rent_12m',
    'forecast_price_energy_off_peak',
    'forecast_price_energy_peak',
    'forecast_price_pow_off_peak'
]]
numeric_features.describe().T
```

Out[20]:

	count	mean	std	min	25%	50%	75%	
cons_12m	14605.0	159230.267032	573483.629064	0.0	5674.000000	14116.000000	40764.000000	6.20
cons_gas_12m	14605.0	28091.082506	162978.563803	0.0	0.000000	0.000000	0.000000	4.15
cons_last_month	14605.0	16091.371448	64366.262314	0.0	0.000000	793.000000	3383.000000	7.71
forecast_cons_12m	14605.0	1868.638618	2387.651549	0.0	494.980000	1112.610000	2402.270000	8.29
forecast_cons_year	14605.0	1399.858747	3247.876793	0.0	0.000000	314.000000	1746.000000	1.75
forecast_discount_energy	14605.0	0.966450	5.108355	0.0	0.000000	0.000000	0.000000	3.00
forecast_meter_rent_12m	14605.0	63.090448	66.166636	0.0	16.180000	18.800000	131.030000	5.99
forecast_price_energy_off_peak	14605.0	0.137282	0.024623	0.0	0.116340	0.143166	0.146348	2.73
forecast_price_energy_peak	14605.0	0.050488	0.049037	0.0	0.000000	0.084138	0.098837	1.95
forecast_price_pow_off_peak	14605.0	43.130085	4.486140	0.0	40.606701	44.311378	44.311378	5.92

In [21]:
```
# Plot a histogram for each numeric feature
for col in numeric_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    feature = df[col]
    sns.distplot(feature, ax = ax, color="purple")
    ax.set_title(col)
plt.show()
```

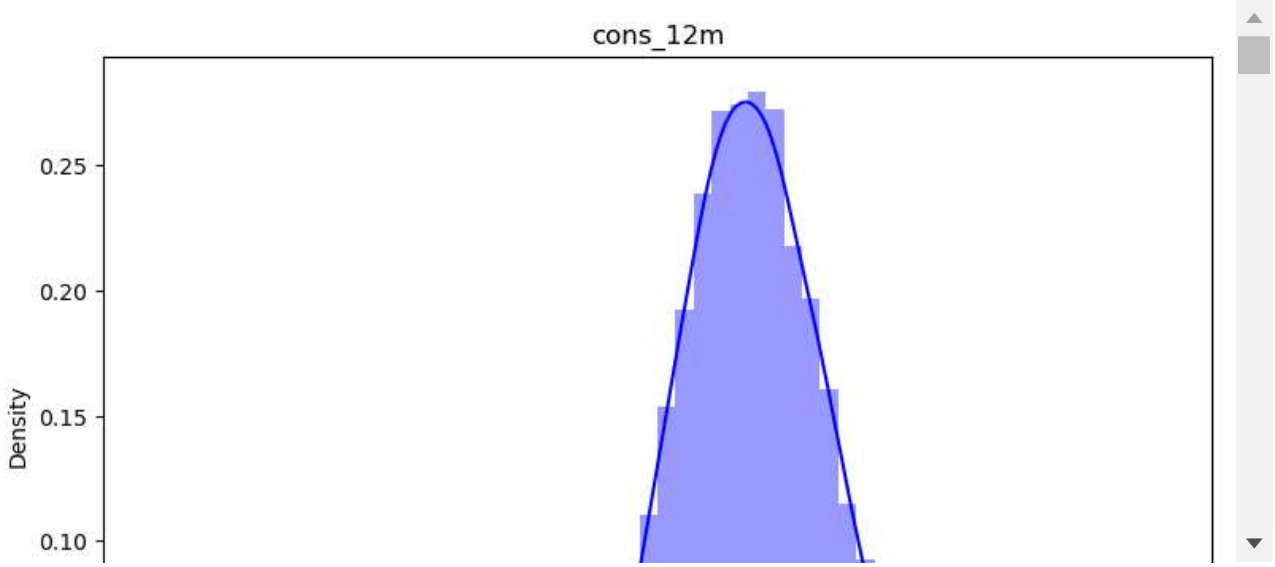


We can see that the standard deviation for most of these features is quite high. Note that we cannot apply log to a value of 0, so we will add a constant of 1 to all the values.

In [22]:
```
for i in numeric_features:
    df[i]=df[i].apply(lambda x:np.log(1+x))
```

Let's quickly check the distributions after log transformation.

In [23]:
```
# Plot a histogram for each numeric feature
for col in numeric_features:
    fig = plt.figure(figsize=(9, 6))
    ax = fig.gca()
    feature = df[col]
    sns.distplot(feature, ax = ax, color="blue")
    ax.set_title(col)
plt.show()
```



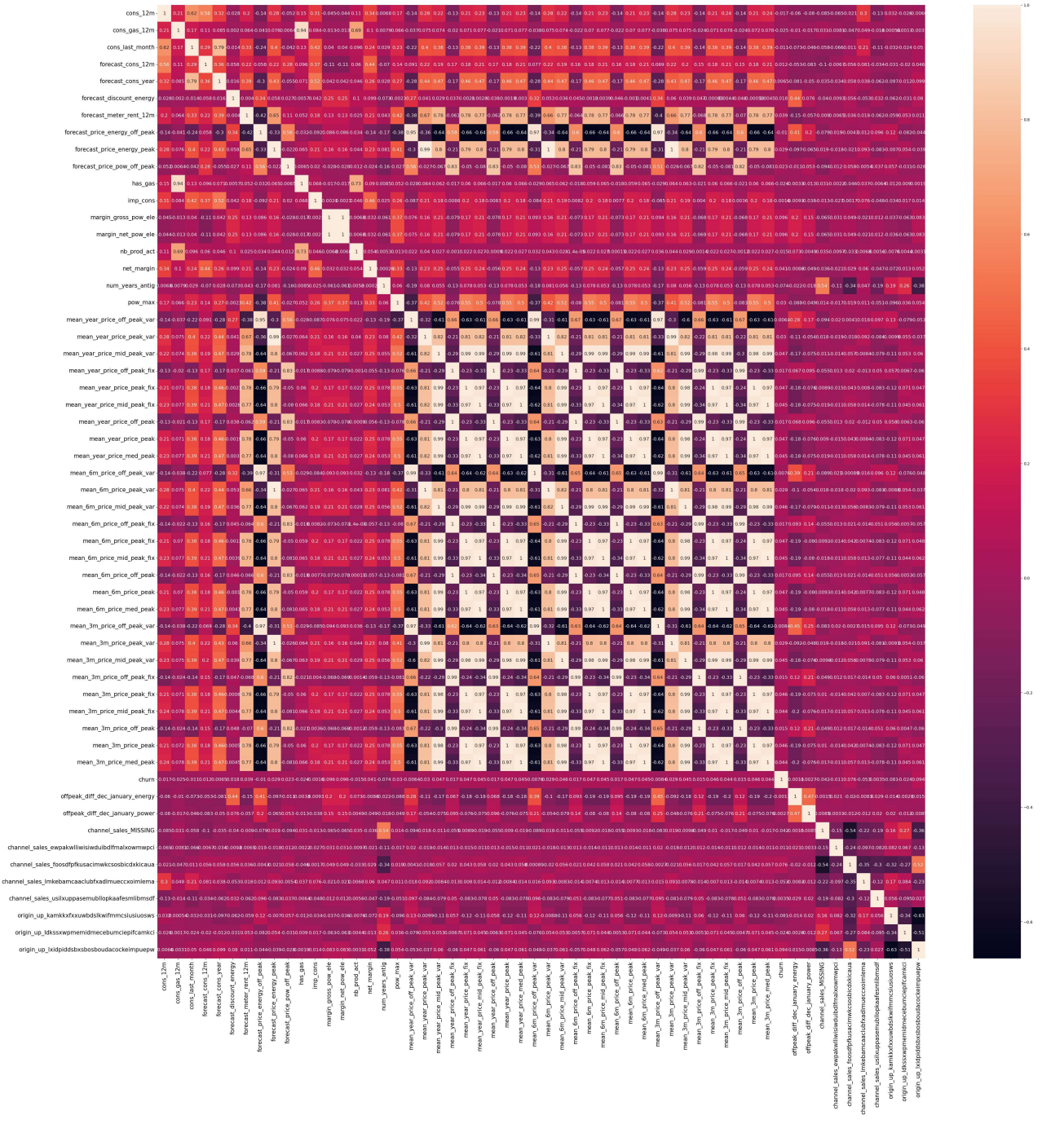
Correlations

Correlation reveals the linear relationships between features. We want features to correlate with churn, as this will indicate that they are good predictors of it. However features that have a very high correlation can sometimes be suspicious. This is because 2 columns that have high correlation indicates that they may share a lot of the same information.

For features to be independent, this means that each feature must have absolutely no dependence on any other feature. If two features are highly correlated and share similar information, we can remove them.

In [24]:
```
correlation = df.corr()
```

In [25]:
```
# Plot correlation
plt.figure(figsize=(45, 45))
sns.heatmap(
    correlation,
    xticklabels=correlation.columns.values,
    yticklabels=correlation.columns.values,
    annot=True,
    annot_kws={'size': 12}
)
# Axis ticks size
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.show()
```



I will remove some variables which exhibit a high correlation with other independent features.

In [26]:
```
df = df.drop(columns=['mean_year_price_mid_peak_var', 'mean_year_price_peak_fix', 'mean_year_price_
                      'mean_year_price_peak', 'mean_year_price_med_peak', 'mean_6m_price_mid_peak_va
                      'mean_6m_price_peak_fix', 'mean_6m_price_mid_peak_fix', 'mean_6m_price_peak',
                      'mean_6m_price_med_peak', 'mean_3m_price_mid_peak_var', 'mean_3m_price_off_pea
                      'mean_3m_price_peak_fix', 'mean_3m_price_mid_peak_fix', 'mean_3m_price_peak',
df.head()
```

Out[26]:

	id	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_cons_year
0	24011ae4ebbe3035111d65fa7c15bc57	0.000000	10.914124	0.000000	0.000000	0.000000
1	d29c2c54acc38ff3c0614d0a653813dd	8.446985	0.000000	0.000000	5.252012	0.000000
2	764c75f661154dac3a6c254cd082ea7d	6.300786	0.000000	0.000000	3.891004	0.000000
3	bba03439a292a1e166f80264c16191cb	7.368340	0.000000	0.000000	5.484963	0.000000
4	149d57cf92fc41cf94415803a877cb4b	8.395252	0.000000	6.267201	6.101999	6.267201

5 rows × 41 columns

4. Building Prediction Models

Split the Data

We need to split the dataset into training dataset and testing dataset. We will use train_test_slit from sklearn package to split the data into 70% for training and hold back 30% for testing.

In [27]:
```
# Separate target variable from independent variables
X = df.drop(columns=['id', 'churn'])
y = df['churn']
```

In [28]:
```
from sklearn.model_selection import train_test_split

# Split data 70%-30% into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=0)

print ('Training cases: %d\nTest cases: %d' % (X_train.shape[0], X_test.shape[0]))
```

```
Training cases: 10223
Test cases: 4382
```

In [29]:
```
y.value_counts()
```

Out[29]:
```
0    13186
1     1419
Name: churn, dtype: int64
```

We can see that the number of churn and not churn is imbalanced. Here we will use imblearn's SMOTE or Synthetic Minority Oversampling Technique to solve the this problem

In [30]:
```
from imblearn.over_sampling import SMOTE
X, y = SMOTE().fit_resample(X, y)
```

In [31]:
```
#check the number of target variable after oversampling using SMOTE
from collections import Counter
print('Resample dataset shape', Counter(y))
```

```
Resample dataset shape Counter({1: 13186, 0: 13186})
```

It can be seen that the distrubtion of Churn status are now balanced.

In [32]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(10223, 39)
(10223,)
(4382, 39)
(4382,)
```

Logistic Regression

In [33]:
```
# Train the model
from sklearn.linear_model import LogisticRegression

# Set regularization rate
reg = 0.01

# train a logistic regression model on the training set
model = LogisticRegression(C=1/reg, solver="liblinear").fit(X_train, y_train)
print (model)
```

```
LogisticRegression(C=100.0, solver='liblinear')
```

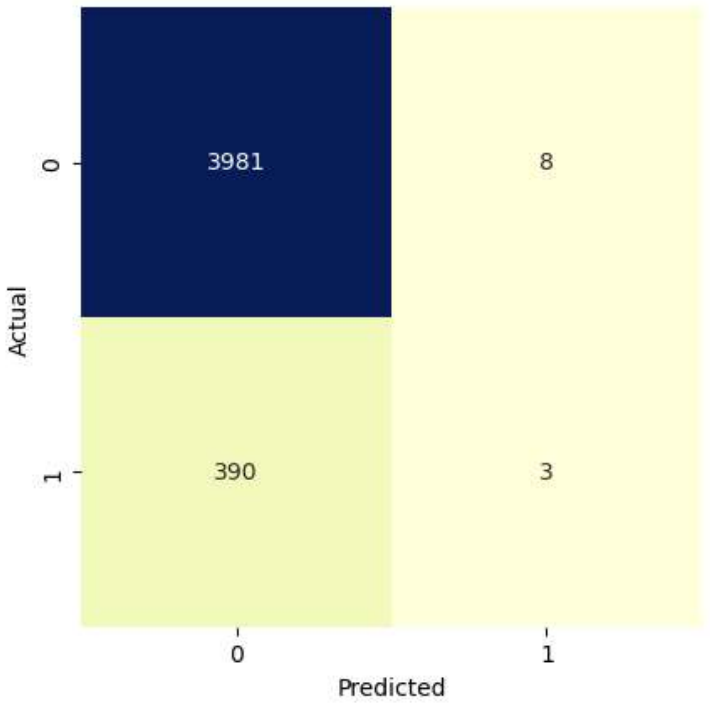
In [34]:
```
y_pred = model.predict(X_test)
```

In [35]:
```
#Let's check the accuracy of the predictions
from sklearn.metrics import accuracy_score

print('Accuracy: ', accuracy_score(y_test, y_pred))
```

```
Accuracy:  0.9091738931994523
```

In [36]:
```
#create confusion matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, square=True, annot=True, cbar=False,  cmap="YlGnBu" ,fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
```

Out[36]: Text(113.9222222222222, 0.5, 'Actual')



In [37]:
```
from sklearn import metrics
tn, fp, fn, tp = metrics.confusion_matrix(y_test, y_pred).ravel()
print(f"True positives: {tp}")
print(f"False positives: {fp}")
print(f"True negatives: {tn}")
print(f"False negatives: {fn}\n")
```

```
True positives: 3
False positives: 8
True negatives: 3981
False negatives: 390
```

In [38]:
```
from sklearn. metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.91      1.00      0.95      3989
           1       0.27      0.01      0.01       393

    accuracy                           0.91      4382
   macro avg       0.59      0.50      0.48      4382
weighted avg       0.85      0.91      0.87      4382
```

In [39]:
```
# retrieve the precision_score and recall_score metrics
from sklearn.metrics import precision_score, recall_score

print("Overall Precision:",precision_score(y_test, y_pred))
print("Overall Recall:",recall_score(y_test, y_pred))
```

```
Overall Precision: 0.2727272727272727
Overall Recall: 0.007633587786259542
```

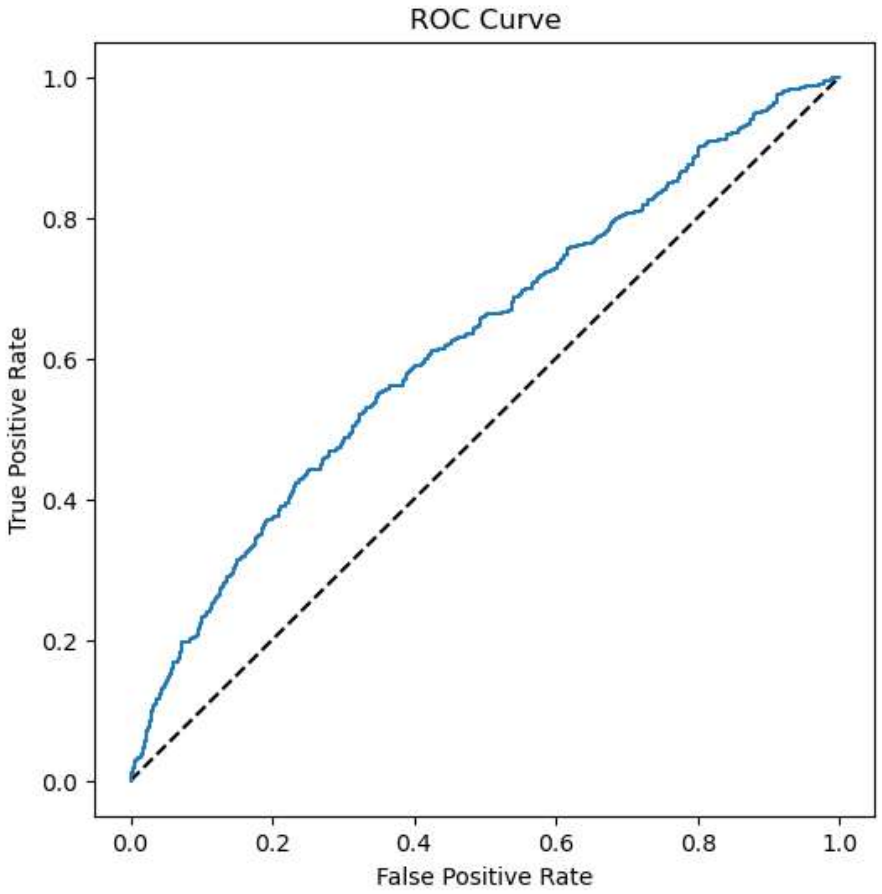
Until now, we've considered the predictions from the model as being either 1 or 0 class labels. Actually, things are a little more complex than that. Statistical machine learning algorithms, like logistic regression, are based on probability; so what actually gets predicted by a binary classifier is the probability that the label is true (P(y)) and the probability that the label is false (1 - P(y)). A threshold value of 0.5 is used to decide whether the predicted label is a 1 (P(y) > 0.5) or a 0 (P(y) <= 0.5). You can use the predict_proba method to see the probability pairs for each case:

In [40]:
```
y_scores = model.predict_proba(X_test)
print(y_scores[:10])
```

```
[[0.86235116 0.13764884]
 [0.92930697 0.07069303]
 [0.9174114  0.0825886 ]
 [0.91644396 0.08355604]
 [0.75506493 0.24493507]
 [0.88050745 0.11949255]
 [0.94409377 0.05590623]
 [0.93636063 0.06363937]
 [0.9141664  0.0858336 ]
 [0.90965688 0.09034312]]
```

The decision to score a prediction as a 1 or a 0 depends on the threshold to which the predicted probabilities are compared. If we were to change the threshold, it would affect the predictions; and therefore change the metrics in the confusion matrix. A common way to evaluate a classifier is to examine the true positive rate (which is another name for recall) and the false positive rate for a range of possible thresholds. These rates are then plotted against all possible thresholds to form a chart known as a received operator characteristic (ROC) chart, like this:

In [41]:
```
from sklearn.metrics import roc_curve
# calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores[:,1])

# plot ROC curve
fig = plt.figure(figsize=(6, 6))
# Plot the diagonal 50% line
plt.plot([0, 1], [0, 1], 'k--')
# Plot the FPR and TPR achieved by our model
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```



The ROC chart shows the curve of the true and false positive rates for different threshold values between 0 and 1. A perfect classifier would have a curve that goes straight up the left side and straight across the top. The diagonal line across the chart represents the probability of predicting correctly with a 50/50 random prediction; so you obviously want the curve to be higher than that (or your model is no better than simply guessing!).

The area under the curve (AUC) is a value between 0 and 1 that quantifies the overall performance of the model. The closer to 1 this value is, the better the model. Once again, scikit-Learn includes a function to calculate this metric.

In [42]:
```
from sklearn.metrics import roc_auc_score

auc = roc_auc_score(y_test,y_scores[:,1])
print('AUC: ' + str(auc))
```

```
AUC: 0.6231710996589221
```

- Looking at the true negatives, we have 3981 out of 3989. This means that out of all the negative cases (churn = 0), we predicted 3981 as negative (hence the name True negative). This is great!

- Looking at the false negatives, this is where we have predicted a client to not churn (churn = 0) when in fact they did churn (churn = 1). This number is too high at 390, but we want to get the false negatives to as close to 0 as we can.
- Looking at false positives, this is where we have predicted a client to churn when they actually didnt churn. For this value we can see there are 8, quite good.
- With the true positives, we can see that in total we have 393 clients that churned in the test dataset. However, we are only able to correctly identify 3 of those 393, which is very poor.
- Looking at the accuracy score, this is very misleading! Hence the use of precision and recall is important. The accuracy score is high, but it does not tell us the whole story.
- Looking at the precision score and recall score, this shows us a score of 0.27 and 0.02 which are very bad. This is not a good model.

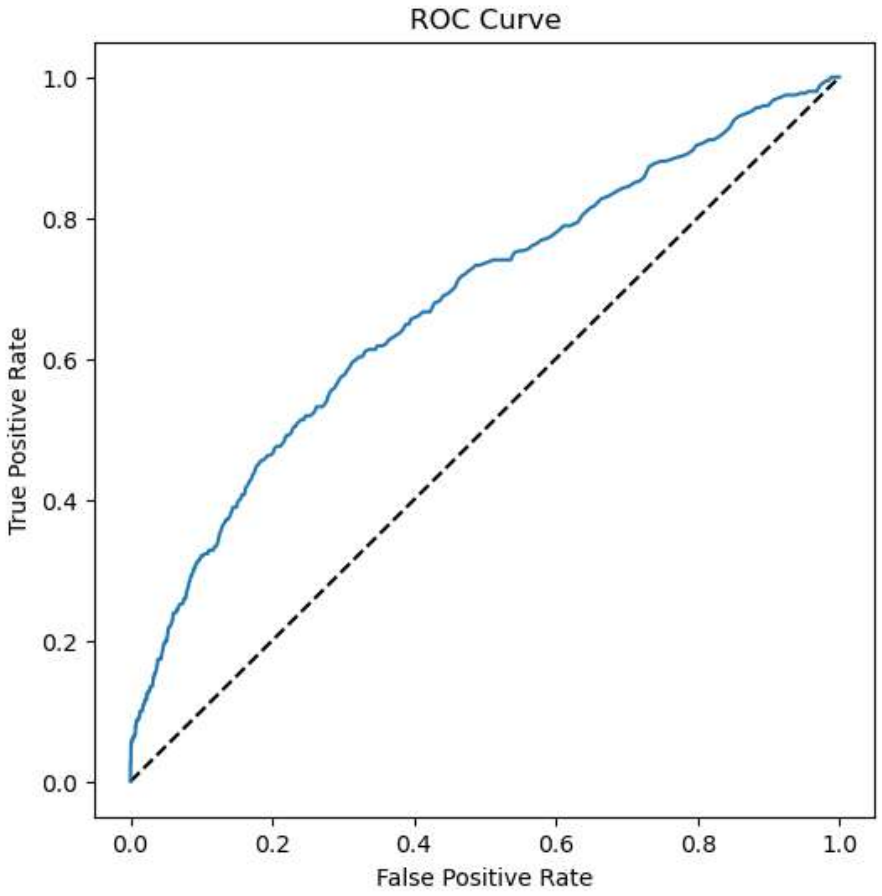
Random Forest

In [43]:
```
from sklearn.ensemble import RandomForestClassifier

model_RF = RandomForestClassifier(n_estimators = 1000).fit(X_train, (y_train))
print (model_RF)
```

```
RandomForestClassifier(n_estimators=1000)
```

In [44]:
```
y_pred_RF = model_RF.predict(X_test)
y_scores_RF = model_RF.predict_proba(X_test)

tn, fp, fn, tp = metrics.confusion_matrix(y_test, y_pred_RF).ravel()
print(f"True positives: {tp}")
print(f"False positives: {fp}")
print(f"True negatives: {tn}")
print(f"False negatives: {fn}\n")

print(classification_report(y_test, y_pred_RF))

print('Accuracy:', accuracy_score(y_test, y_pred_RF))
print("Overall Precision:",precision_score(y_test, y_pred_RF))
print("Overall Recall:",recall_score(y_test, y_pred_RF))
auc = roc_auc_score(y_test,y_scores_RF[:,1])
print('\nAUC: ' + str(auc))

# calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_scores_RF[:,1])

# plot ROC curve
fig = plt.figure(figsize=(6, 6))
# Plot the diagonal 50% line
plt.plot([0, 1], [0, 1], 'k--')
# Plot the FPR and TPR achieved by our model
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()
```

```
True positives: 21
False positives: 7
True negatives: 3982
False negatives: 372

              precision    recall  f1-score   support

           0       0.91      1.00      0.95      3989
           1       0.75      0.05      0.10       393

    accuracy                           0.91      4382
   macro avg       0.83      0.53      0.53      4382
weighted avg       0.90      0.91      0.88      4382

Accuracy: 0.9135098128708352
Overall Precision: 0.75
Overall Recall: 0.05343511450381679

AUC: 0.6778807113965442
```



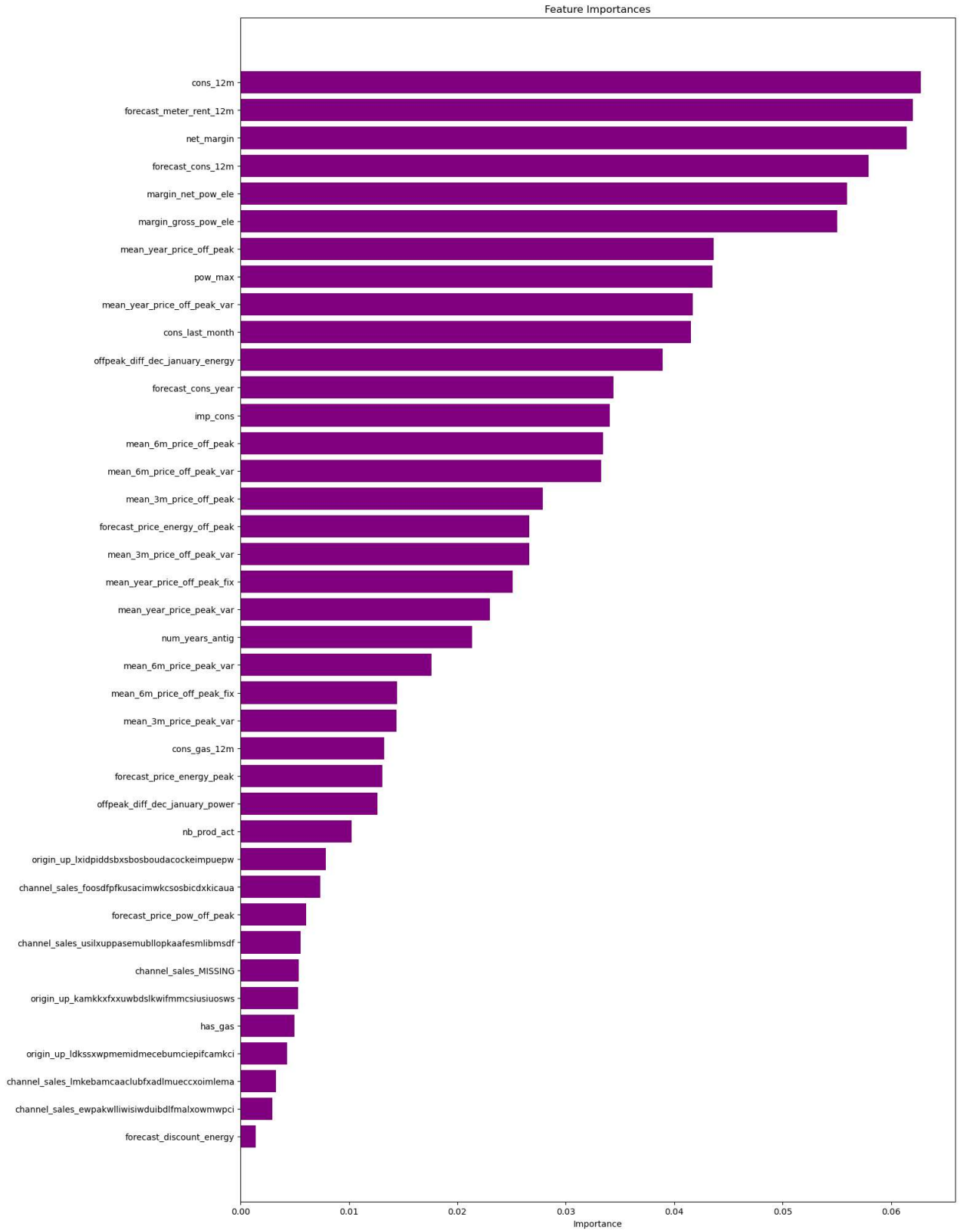
- Looking at the true negatives, we have 3984 out of 3989. This is great!
- Looking at the false negatives, 371 is too high but better than Logistic Model.
- Looking at false positives, we can see there are only , quite good.
- With the true positives, we correctly identify 22 of those 393, which is very poor.
- Looking at the accuracy score, this is very misleading, looking at the precision score and recall score, this shows us a score of 0.81 which is not bad, but could be improved. However, the recall shows us that the classifier has a very poor ability to identify positive samples. This would be the main concern for improving this model!
- The AUC is better a little bit comparing with Logistic Regression.

Model Understanding

Feature importances indicate the importance of a feature within the predictive model.

In [45]:
```
feature_importances = pd.DataFrame({
    'features': X_train.columns,
    'importance': model_RF.feature_importances_
}).sort_values(by='importance', ascending=True).reset_index()
```

In [46]:
```
plt.figure(figsize=(15, 25))
plt.title('Feature Importances')
plt.barh(range(len(feature_importances)), feature_importances['importance'], color='purple', align=
plt.yticks(range(len(feature_importances)), feature_importances['features'])
plt.xlabel('Importance')
plt.show()
```



From this chart, we can observe the following points:

- Consumption over 12 months and forecast meter rent 12 month are a top driver for churn in this model
- Net margin also plays an important role in the predictive model.
- Our price sensitivity features are scattered around but are not the main driver for a customer churning

In [47]:
```
X_test = X_test.reset_index()
X_test.drop(columns='index', inplace=True)
```

In [48]:
```
y_scores = model_RF.predict_proba(X_test)
probabilities = y_scores[:, 1]
```

In [49]:
```
X_test['churn'] = y_pred_RF.tolist()
X_test['churn_probability'] = probabilities.tolist()
X_test.to_csv('data_with_predictions.csv')
```

Now let's save our trained model so we can use it again later.

In [50]:
```
import joblib

# Save the model as a pickle file
filename = 'powerco_churn_model.pkl'
joblib.dump(model_RF, filename)
```

Out[50]:
```
['powerco_churn_model.pkl']
```

- - - - - - - - X X X X X X X X - - - - - - - -