Feature Engineering & Data Modeling

6. Import packages

```
import pandas as pd
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

7. Load data

```
df = pd.read_csv('./clean_data_after_eda.csv')
df["date_activ"] = pd.to_datetime(df["date_activ"], format='%Y-%m-%d')
df["date_end"] = pd.to_datetime(df["date_end"], format='%Y-%m-%d')
df["date_modif_prod"] = pd.to_datetime(df["date_modif_prod"], format='%Y-%m-%d')
df["date_renewal"] = pd.to_datetime(df["date_renewal"], format='%Y-%m-%d')
df = df.drop(columns='Unnamed: 0')
```

```
df.head()
```

	id	channel_sales	cons_12m	cons_
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	
3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	
4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	

5 rows × 53 columns

8. Feature engineering

8.1 Difference between off-peak prices in December and preceding January

```
price_df = pd.read_csv('price_data.csv')
price_df["price_date"] = pd.to_datetime(price_df["price_date"],
                                        format='%Y-%m-%d')
price_df.head()
```

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

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

# 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
diff1 = pd.merge(dec_prices.rename(columns={'price_off_peak_var': 'dec_1',
                                            'price_off_peak_fix': 'dec_2'}),
                 jan_prices.drop(columns='price_date'), on='id')
diff1['offpeak_diff_dec_january_energy'] = diff1['dec_1'] - \
 diff1['price_off_peak_var']
diff1['offpeak_diff_dec_january_power'] = diff1['dec_2'] - \
 diff1['price_off_peak_fix']
diff1 = diff1[['id', 'offpeak_diff_dec_january_energy',
               'offpeak_diff_dec_january_power']]
diff1.head()
```

	id	offpeak_diff_dec_january_energy	offpeak_diff_d
0	0002203ffbb812588b632b9e628cc38d	-0.006192	
1	0004351ebdd665e6ee664792efc4fd13	-0.004104	

2	0010bcc39e42b3c2131ed2ce55246e3c	0.050443
3	0010ee3855fdea87602a5b7aba8e42de	-0.010018
4	00114d74e963e47177db89bc70108537	-0.003994

8.2 Difference between peak prices in December and preceding January

```
# Group off-peak prices by companies and month
monthly_price_2_by_id = (price_df.groupby(['id', 'price_date']).agg({
    'price_peak_var': 'mean', 'price_peak_fix': 'mean'}).reset_index())

# Get january and december prices
jan_prices_2 = monthly_price_2_by_id.groupby('id').first().reset_index()
dec_prices_2 = monthly_price_2_by_id.groupby('id').last().reset_index()

# Calculate the difference
diff2 = pd.merge(dec_prices_2.rename(columns={'price_peak_var': 'dec_1',
                                              'price_peak_fix': 'dec_2'}),
                 jan_prices_2.drop(columns='price_date'), on='id')
diff2['peak_diff_dec_january_energy'] = diff2['dec_1'] - diff2['price_peak_var']
diff2['peak_diff_dec_january_power'] = diff2['dec_2'] - diff2['price_peak_fix']
diff2 = diff2[['id', 'peak_diff_dec_january_energy',
               'peak_diff_dec_january_power']]
diff2.head()
```

	id	peak_diff_dec_january_energy	peak_diff_dec_jan
0	0002203ffbb812588b632b9e628cc38d	-0.002302	
1	0004351ebdd665e6ee664792efc4fd13	0.000000	
2	0010bcc39e42b3c2131ed2ce55246e3c	0.000000	
3	0010ee3855fdea87602a5b7aba8e42de	-0.005120	
4	00114d74e963e47177db89bc70108537	0.000000	

8.3 Difference between mid prices in December and preceding January

```
# Group off-peak prices by companies and month
monthly_price_3_by_id = (price_df.groupby(['id', 'price_date']).agg({
    'price_mid_peak_var': 'mean', 'price_mid_peak_fix': 'mean'}).reset_index())

# Get january and december prices
jan_prices_3 = monthly_price_3_by_id.groupby('id').first().reset_index()
```

```
jan_prices_3 = monthly_price_3_by_id.groupby('id').first().reset_index()
dec_prices_3 = monthly_price_3_by_id.groupby('id').last().reset_index()

# Calculate the difference
diff3 = pd.merge(dec_prices_3.rename(columns={'price_mid_peak_var': 'dec_1',
                                              'price_mid_peak_fix': 'dec_2'}),
                 jan_prices_3.drop(columns='price_date'), on='id')
diff3['midpeak_diff_dec_january_energy'] = diff3['dec_1'] - \
 diff3['price_mid_peak_var']
diff3['midpeak_diff_dec_january_power'] = diff3['dec_2'] - \
diff3['price_mid_peak_fix']
diff3 = diff3[['id', 'midpeak_diff_dec_january_energy',
               'midpeak_diff_dec_january_power']]
diff3.head()
```

	id	midpeak_diff_dec_january_energy	midpeak_diff_d
0	0002203ffbb812588b632b9e628cc38d	0.003487	
1	0004351ebdd665e6ee664792efc4fd13	0.000000	
2	0010bcc39e42b3c2131ed2ce55246e3c	0.000000	
3	0010ee3855fdea87602a5b7aba8e42de	0.000763	
4	00114d74e963e47177db89bc70108537	0.000000	

```
from google.colab import drive
drive.mount('/content/drive')
```

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

```
# Merging the three tables with price differences plus df dataframe except
# the column 'churn'

diff_price = pd.merge(diff1, diff2, on= 'id')
diff_price = pd.merge(diff_price, diff3, on='id')

df = pd.merge(df, diff_price, on='id')

df.head()
```

	id	channel_sales	cons_12m	cons_
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	

3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema		1584
4	149d57cf92fc41cf94415803a877cb4b		MISSING	4425

5 rows × 59 columns

8.4 Maximum price changes across periods and months

```
#Aggregate average prices per period by company
mean_prices_by_months = price_df.groupby(['id', 'price_date']).agg({
    'price_off_peak_var': 'mean',
    'price_peak_var': 'mean',
    'price_mid_peak_var': 'mean',
    'price_off_peak_fix': 'mean',
    'price_peak_fix': 'mean',
    'price_mid_peak_fix': 'mean'
}).reset_index()


#Calculate the mean difference between consecutive periods
mean_prices_by_months['off_peak_peak_var_mean_diff'] = \
  mean_prices_by_months['price_off_peak_var'] - \
  mean_prices_by_months['price_peak_var']
mean_prices_by_months['peak_mid_peak_var_mean_diff'] = \
  mean_prices_by_months['price_peak_var'] - \
  mean_prices_by_months['price_mid_peak_var']
mean_prices_by_months['off_peak_mid_peak_var_mean_diff'] = \
  mean_prices_by_months['price_off_peak_var'] - \
  mean_prices_by_months['price_mid_peak_var']
mean_prices_by_months['off_peak_peak_fix_mean_diff'] = \
  mean_prices_by_months['price_off_peak_fix'] - \
  mean_prices_by_months['price_peak_fix']
mean_prices_by_months['peak_mid_peak_fix_mean_diff'] = \
  mean_prices_by_months['price_peak_fix'] - \
  mean_prices_by_months['price_mid_peak_fix']
mean_prices_by_months['off_peak_mid_peak_fix_mean_diff'] = \
  mean_prices_by_months['price_off_peak_fix'] - \
  mean_prices_by_months['price_mid_peak_fix']



#Calculate the maximum monthly difference across time periods
max_diff_acrross_periods_months = mean_prices_by_months.groupby(['id']).agg({
    'off_peak_peak_var_mean_diff': 'max',
    'peak_mid_peak_var_mean_diff': 'max',
    'off_peak_mid_peak_var_mean_diff': 'max',
    'off_peak_peak_fix_mean_diff': 'max',
    'peak_mid_peak_fix_mean_diff': 'max',
```

```
    'off_peak_mid_peak_fix_mean_diff': 'max'
}).reset_index().rename(
    columns={
    'off_peak_peak_var_mean_diff': 'off_peak_var_max_monthly_diff',
    'peak_mid_peak_var_mean_diff': 'peak_mid_peak_var_max_monthly_diff',
    'off_peak_mid_peak_var_mean_diff': 'off_peak_mid_peak_var_max_monthly_diff',
    'off_peak_peak_fix_mean_diff': 'off_peak_peak_fix_max_monthly_diff',
    'peak_mid_peak_fix_mean_diff': 'peak_mid_peak_fix_max_monthly_diff',
    'off_peak_mid_peak_fix_mean_diff': 'off_peak_mid_peak_fix_max_monthly_diff'
    }
)


columns = [
    'id',
    'off_peak_var_max_monthly_diff',
    'peak_mid_peak_var_max_monthly_diff',
    'off_peak_mid_peak_var_max_monthly_diff',
    'off_peak_peak_fix_max_monthly_diff',
    'peak_mid_peak_fix_max_monthly_diff',
    'off_peak_mid_peak_fix_max_monthly_diff'
]

df = pd.merge(df, max_diff_acrross_periods_months[columns], on='id')
df.head()
```

	id	channel_sales	cons_12m	cons_
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	
3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	
4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	

5 rows × 65 columns

8.5 Other features - Data augmentation - Data transformation

We can add the tenure or, in other words, the years a company has been a client of PowerCo. Moreover, it is interesting to convert the dates in months, transforming boolean and categorical data into numerical data and standardize some variables.

Tenure

How long a company has been a client of PowerCo

How long a company has been a client of PowerCo.

```
import numpy as np

df['tenure'] = (((df['date_end'] - df['date_activ'])/ np.timedelta64(1, 'Y'))
.astype(int))

df.groupby(['tenure']).agg({'churn': 'mean'}).sort_values(by='churn',
                                                          ascending=False)
```

tenure	churn
3	0.143836
2	0.133080
4	0.125756
13	0.095238
5	0.085425
12	0.083333
6	0.080713
7	0.073394
11	0.063584
8	0.048000
9	0.024096
10	0.020000

We can observe that it is almost directly proportional the years of antiquity with the fidelity.

Transforming dates into months

- 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)

```
def convert_months(reference_date, df, column):
```

```
    """
    Input a column with timedeltas and return months
    """
    time_delta = reference_date - df[column]
    months = (time_delta / np.timedelta64(1, 'M')).astype(int)
    return months


from datetime import date, time, datetime

# Create reference date
reference_date = datetime(2016, 1, 1)

# Create columns
df['months_activ'] = convert_months(reference_date, df, 'date_activ')
df['months_to_end'] = -convert_months(reference_date, df, 'date_end')
df['months_modif_prod'] = convert_months(reference_date, df, 'date_modif_prod')
df['months_renewal'] = convert_months(reference_date, df, 'date_renewal')


# We no longer need the datetime columns that we used for feature engineering,
# so we can drop them
remove = [
    'date_activ',
    'date_end',
    'date_modif_prod',
    'date_renewal'
]

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

	id	channel_sales	cons_12m	cons_
0	24011ae4ebbe3035111d65fa7c15bc57	foosdfpfkusacimwkcsosbicdxkicaua	0	
1	d29c2c54acc38ff3c0614d0a653813dd	MISSING	4660	
2	764c75f661154dac3a6c254cd082ea7d	foosdfpfkusacimwkcsosbicdxkicaua	544	
3	bba03439a292a1e166f80264c16191cb	lmkebamcaaclubfxadlmueccxoimlema	1584	
4	149d57cf92fc41cf94415803a877cb4b	MISSING	4425	

5 rows × 66 columns

Transforming Boolean data

We simply have to transform the column `has_gas` from being categorical to being a binary flag

```
df['has_gas'] = df['has_gas'].replace(['t', 'f'], [1, 0])
df.groupby(['has_gas']).agg({'churn': 'mean'})
```

has_gas	churn
0	0.100544
1	0.081887

If a customer also buys gas from PowerCo, it shows that they have multiple products and are a loyal customer to the brand. Hence, it is no surprise that customers who do not buy gas are almost 2% more likely to churn than customers who also buy gas from PowerCo. Hence, this is a useful feature.

Transforming categorical data

A predictive model cannot accept categorical or string values, hence as a data scientist you need to encode categorical features into numerical representations in the most compact and discriminative way possible.

For encoding categorical features we will use dummy variables or one hot encoding. This create a new feature for every unique value of a categorical column, and fills this column with either a 1 or a 0 to indicate that this company does or does not belong to this category.

channel_sales

```
# Transform into categorical type
df['channel_sales'] = df['channel_sales'].astype('category')

# Let's see how many categories are within this column
df['channel_sales'].value_counts()
```

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

We have 8 categories, so we will create 8 dummy variables from this column. However, as you

can see the last 3 categories in the output above, show that they only have 11, 3 and 2 occurrences respectively. Considering that our dataset has about 14000 rows, this means that these dummy variables will be almost entirely 0 and so will not add much predictive power to the model at all (since they're almost entirely a constant value and provide very little).

For this reason, we will drop these 3 dummy variables.

```
df = pd.get_dummies(df, columns=['channel_sales'], prefix='channel')
df = df.drop(columns=['channel_sddiedcslfslkckwlfkdpoeeailfpeds',
                      'channel_epumfxlbckeskwekxbiuasklxalciiuu',
                      'channel_fixdbufsefwooaasfcxdxadsiekoceaa'])
df.head()
```

	id	cons_12m	cons_gas_12m	cons_last_month	forecas
0	24011ae4ebbe3035111d65fa7c15bc57	0	54946	0	
1	d29c2c54acc38ff3c0614d0a653813dd	4660	0	0	
2	764c75f661154dac3a6c254cd082ea7d	544	0	0	
3	bba03439a292a1e166f80264c16191cb	1584	0	0	
4	149d57cf92fc41cf94415803a877cb4b	4425	0	526	

5 rows × 70 columns

origin_up

```
# Transform into categorical type
df['origin_up'] = df['origin_up'].astype('category')

# Let's see how many categories are within this column
df['origin_up'].value_counts()
```

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

Similar to `channel_sales` the last 2 categories in the output above show very low frequency, so we will remove these from the features after creating dummy variables.

```
df = pd.get_dummies(df, columns=['origin_up'], prefix='origin_up')
```

```
df = df.drop(columns=['origin_up_usapbepcfoloekilkwsdiboslwaxobdp',
                      'origin_up_ewxeelcelemmiwuafmddpobolfuxioce'])
df.head()
```

	id	cons_12m	cons_gas_12m	cons_last_month	forecas
0	24011ae4ebbe3035111d65fa7c15bc57	0	54946	0	
1	d29c2c54acc38ff3c0614d0a653813dd	4660	0	0	
2	764c75f661154dac3a6c254cd082ea7d	544	0	0	
3	bba03439a292a1e166f80264c16191cb	1584	0	0	
4	149d57cf92fc41cf94415803a877cb4b	4425	0	526	

5 rows × 73 columns

Transforming numerical data

In the EDA we saw that some variables were highly skewed. The reason why we need to treat skewness is because some predictive models have inherent assumptions about the distribution of the features that are being supplied to it. Such models are called parametric models, and they typically assume that all variables are both independent and normally distributed.

Skewness isn't always a bad thing, but as a rule of thumb it is always good practice to treat highly skewed variables because of the reason stated above, but also as it can improve the speed at which predictive models are able to converge to its best solution.

There are many ways that you can treat skewed variables. You can apply transformations such as:

- Square root
- Cubic root
- Logarithm

to a continuous numeric column and you will notice the distribution changes. For this use case we will use the 'Logarithm' transformation for the positively skewed features.

Note: We cannot apply log to a value of 0, so we will add a constant of 1 to all the values

First we should see the statistics of the skewed features, so that we can compare before and after transformation

```
skewed = [
    'cons_12m',
    'cons_gas_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'
]

df[skewed].describe()
```

	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_cons_
count	1.460500e+04	1.460500e+04	14605.000000	14605.000000	14605.00
mean	1.592303e+05	2.809108e+04	16091.371448	1868.638618	1399.85
std	5.734836e+05	1.629786e+05	64366.262314	2387.651549	3247.87
min	0.000000e+00	0.000000e+00	0.000000	0.000000	0.00
25%	5.674000e+03	0.000000e+00	0.000000	494.980000	0.00
50%	1.411600e+04	0.000000e+00	793.000000	1112.610000	314.00
75%	4.076400e+04	0.000000e+00	3383.000000	2402.270000	1746.00
max	6.207104e+06	4.154590e+06	771203.000000	82902.830000	175375.00

Huge standard deviation in all cases. We are going to solve this inconvenience applying logarithmical transformation for all the values.

```
# Apply log10 transformation
df["cons_12m"] = np.log10(df["cons_12m"] + 1)
df["cons_gas_12m"] = np.log10(df["cons_gas_12m"] + 1)
df["cons_last_month"] = np.log10(df["cons_last_month"] + 1)
df["forecast_cons_12m"] = np.log10(df["forecast_cons_12m"] + 1)
df["forecast_cons_year"] = np.log10(df["forecast_cons_year"] + 1)
df["forecast_meter_rent_12m"] = np.log10(df["forecast_meter_rent_12m"] + 1)
df["imp_cons"] = np.log10(df["imp_cons"] + 1)

df[skewed].describe()
```

	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_cons_
count	14605.000000	14605.000000	14605.000000	14605.000000	14605.00
mean	4.223945	0.778978	2.264801	2.962162	1.78

std	0.884545	1.716828	1.769266	0.683612	1.58
min	0.000000	0.000000	0.000000	0.000000	0.00
25%	3.753966	0.000000	0.000000	2.695464	0.00
50%	4.149742	0.000000	2.899821	3.046733	2.49
75%	4.610287	0.000000	3.529430	3.380803	3.24
max	6.792889	6.618528	5.887169	4.918575	5.24

Now, the standard deviation for almost all the variables is much lower after apply the logarithm.

This shows that these features are more stable and predictable now.

8.6 Correlations

In terms of creating new features and transforming existing ones, it is very much a trial and error situation that requires iteration. Once we train a predictive model we can see which features work and don't work, we will also know how predictive this set of features is. Based on this, we can come back to feature engineering to enhance our model.

For now, we will leave feature engineering at this point. Another thing that is always useful to look at is how correlated all of the features are within your dataset.

This is important because it 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. One of the assumptions of any parametric predictive model (as stated earlier) is that all features must be independent.

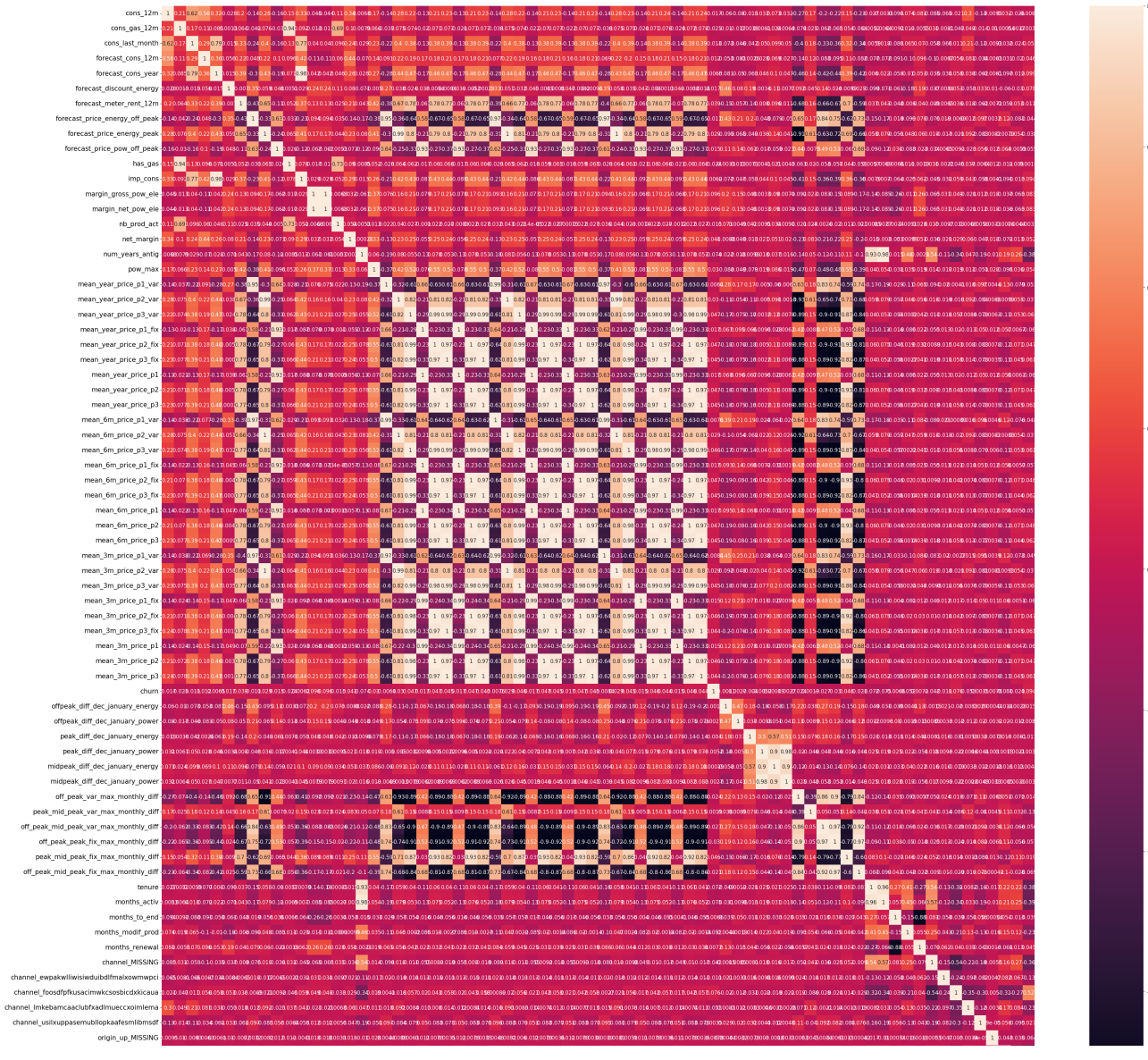
For features to be independent, this means that each feature must have absolutely no dependence on any other feature. If two features are highly correlated and share similar information, this breaks this assumption.

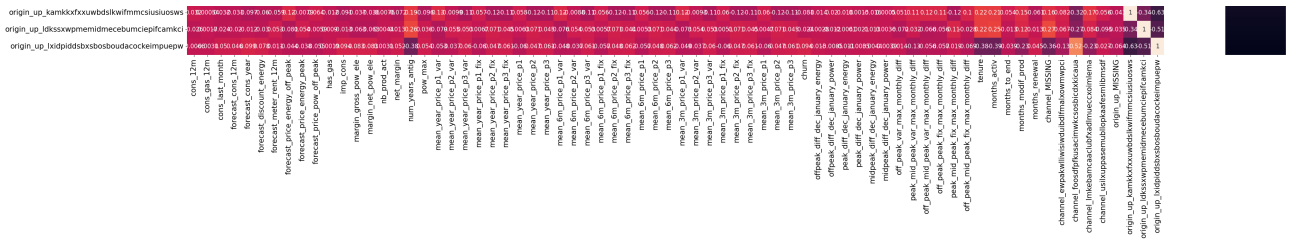
Ideally, you want a set of features that have 0 correlation with all of the independent variables (all features except our target variable) and a high correlation with the target variable (churn). However, this is very rarely the case and it is common to have a small degree of correlation between independent features.

So now let's look at how all the features within the model are correlated.

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


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





We will remove ten variables which exhibit a high correlation with other independent features.

```
df = df.drop(columns=['margin_gross_pow_ele', 'margin_net_pow_ele',
                      'mean_year_price_p1_fix', 'mean_year_price_p2_fix',
                      'mean_year_price_p3_fix', 'mean_3m_price_p1_fix',
                      'mean_3m_price_p2_fix', 'mean_3m_price_p1_fix',
                      'num_years_antig', 'forecast_cons_year'])
df.head()
```

	id	cons_12m	cons_gas_12m	cons_last_month	forecas
0	24011ae4ebbe3035111d65fa7c15bc57	0.000000	4.739944	0.000000	
1	d29c2c54acc38ff3c0614d0a653813dd	3.668479	0.000000	0.000000	

2	764c75f661154dac3a6c254cd082ea7d	2.736397	0.000000	0.000000
3	bba03439a292a1e166f80264c16191cb	3.200029	0.000000	0.000000
4	149d57cf92fc41cf94415803a877cb4b	3.646011	0.000000	2.721811

5 rows × 64 columns

---

9. Data Modeling

We are using a Random Forest classifier at that extent. A Random Forest sits within the category of ensemble algorithms because internally the Forest refers to a collection of Decision Trees which are tree-based learning algorithms. As the data scientist, you can control how large the forest is (that is, how many decision trees you want to include).

The reason why an ensemble algorithm is powerful is because of the laws of averaging, weak learners and the central limit theorem. If we take a single decision tree and give it a sample of data and some parameters, it will learn patterns from the data. It may be overfit or it may be underfit, but that is now our only hope, that single algorithm.

With ensemble methods, instead of banking on 1 single trained model, we can train 1000's of decision trees, all using different splits of the data and learning different patterns. It would be like asking 1000 people to all learn how to code. You would end up with 1000 people with different answers, methods and styles! The weak learner notion applies here too, it has been found that if you train your learners not to overfit, but to learn weak patterns within the data and you have a lot of these weak learners, together they come together to form a highly predictive pool of knowledge! This is a real life application of many brains are better than 1.

Now instead of relying on 1 single decision tree for prediction, the random forest puts it to the overall views of the entire collection of decision trees. Some ensemble algorithms using a voting approach to decide which prediction is best, others using averaging.

As we increase the number of learners, the idea is that the random forest's performance should converge to its best possible solution.

Some additional advantages of the random forest classifier include:

1. The random forest uses a rule-based approach instead of a distance calculation and so features do not need to be scaled
2. It is able to handle non-linear parameters better than linear based models

On the flip side, some disadvantages of the random forest classifier include:

1. The computational power needed to train a random forest on a large dataset is high, since we need to build a whole ensemble of estimators.
2. Training time can be longer due to the increased complexity and size of thee ensemble

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

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

#Splitting randomly data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

#Creating Random Forest Object
random_forest = RandomForestClassifier()

#Defining hyperparameters range
param_grid = {
    'max_depth':[2,4,6,8,10,12,14,16,18,20],
    'min_samples_leaf': [1,2,3,4,5,6,7,8,9,10],
    'criterion': ['gini', 'entropy'],
    'random_state': [0,42]
}

#Creating GridSearch object
random_grid_tree=GridSearchCV(random_forest, param_grid, cv=5,
                              scoring='roc_auc', n_jobs=-1)

#Fitting the GridSearch object to the training set
random_grid_tree.fit(X_train, y_train)

#Printing the best hyperparameters for Random Forest
print("Best parameters are: ", random_grid_tree.best_params_)
```

```
Best parameters are:  {'criterion': 'entropy', 'max_depth': 12, 'min_samples_leaf': 3
```

```
rfc=RandomForestClassifier(criterion='entropy', max_depth= 12,
                           min_samples_leaf= 1, random_state= 42)
rfc
```

```
▾                          RandomForestClassifier
RandomForestClassifier(criterion='entropy', max_depth=12, random_state=42)
```

Training the model

```
#Training the model
random_forest.fit(X_train, y_train)

#Training accuracy
random_forest.score(X_train, y_train)
```

```
0.999743238616912
```

Predicting Churn from test set

```
yhat = random_forest.predict(X_test)
yhat[:10]
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

---

10. Model Evaluation

10.1 Evaluation

Now let's evaluate how well this trained model is able to predict the values of the test dataset.

We are going to use 3 metrics to evaluate performance:

- *Accuracy = the ratio of correctly predicted observations to the total observations*
- *Precision = the ability of the classifier to not label a negative sample as positive*
- *Recall = the ability of the classifier to find all the positive samples*

The reason why we are using these three metrics is because a simple accuracy is not always a good measure to use. To give an example, let's say you're predicting heart failures with patients in a hospital and there were 100 patients out of 1000 that did have a heart failure.

If the prediction is 80 out of 100 (80%) of the patients that did have a heart failure correctly, you might think that you've done well! However, this also means that you predicted 20 wrong and what may the implications of predicting these remaining 20 patients wrong? Maybe they miss out on getting vital treatment to save their lives.

As well as this, what about the impact of predicting negative cases as positive (people not having heart failure being predicted that they did)? Maybe a high number of false positives means that resources get used up on thee wrong people and a lot of time is wasted when they could have been helping the real heart failure sufferers.

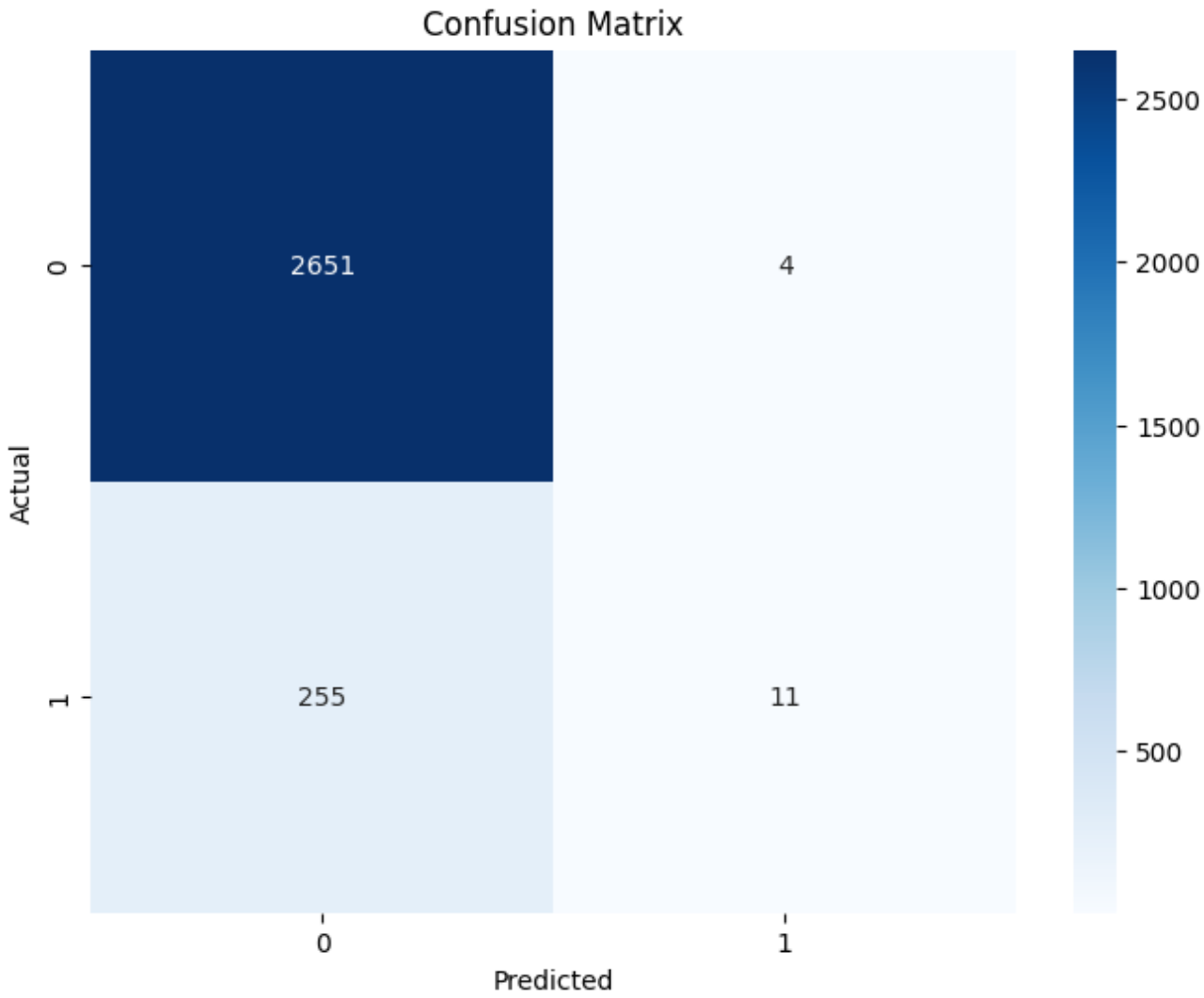
This is just an example, but it illustrates why other performance metrics are necessary such Precision and Recall, which are good measures to use in a classification scenario.

10.2 Confusion Matrix

```
#Confusion matrix heatmap

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8,6))
sns.heatmap(confusion_matrix(y_test,yhat),annot=True,fmt='d',cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```



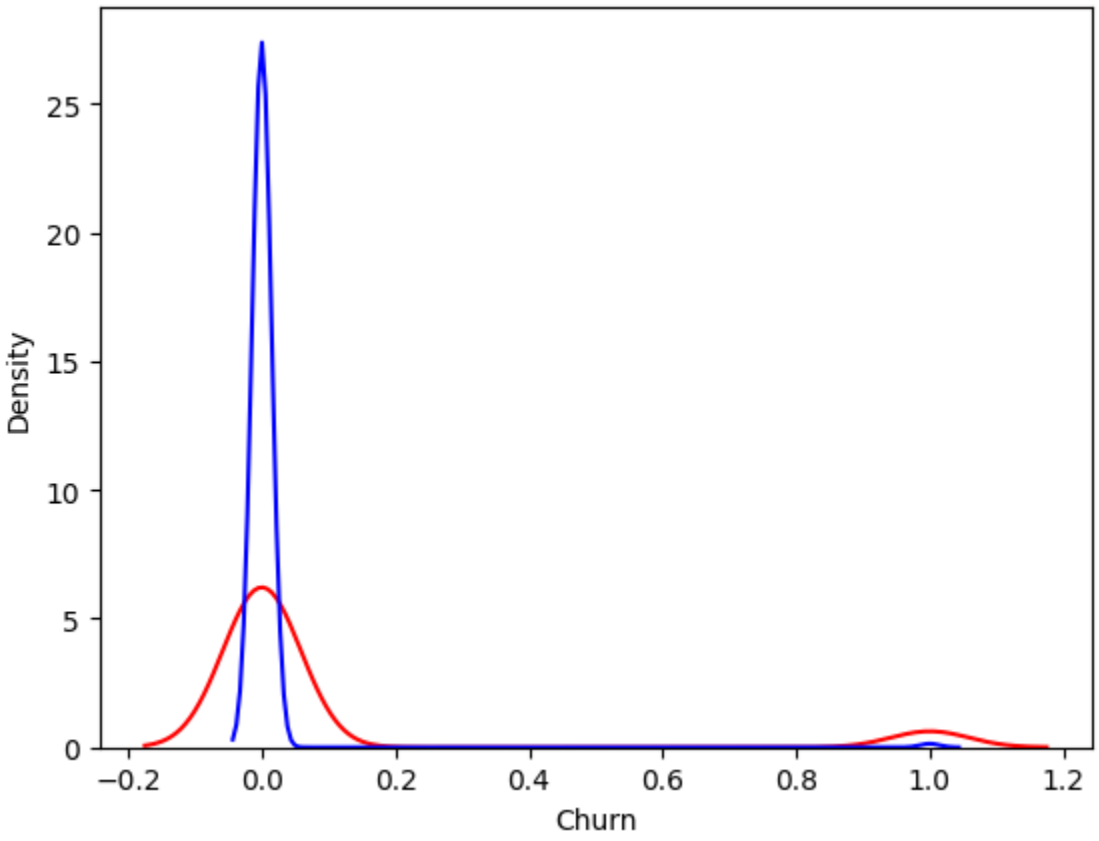
The True Positive (correct prediction of churn) shows the count of correctly classified data point whereas the False Positive elements are those which are missclassified by the model. Results show a very accurate prediction of True Negative and a poor prediction of True Positive (correct prediction of not churn). TN = 2651 TP = 11 FN = 255 FP = 4

10.3 Distribution Plot

```
import warnings
warnings.filterwarnings("ignore", category=UserWarning)

ax = sns.distplot(y_test, hist=False, color='r', axlabel="Churn")
sns.distplot(yhat, hist=False, color='b', axlabel="Churn", ax=ax)
```

```
<Axes: xlabel='Churn', ylabel='Density'>
```



Not a great overlapping, so the model doesn't fit really well.

10.4 Classification Report

```
from sklearn.metrics import classification_report
print(classification_report(y_test, yhat))
```

```
                  precision    recall  f1-score   support

               0       0.91      1.00      0.95      2655
               1       0.73      0.04      0.08       266

        accuracy                           0.91      2921
       macro avg       0.82      0.52      0.52      2921
    weighted avg       0.90      0.91      0.87      2921
```

Because of a very bad prediction in True Negative, we have a recall close to 0

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

print("Accuracy Score: ",accuracy_score(y_test, yhat))
print("Mean Squared Error (MSE): ", mean_squared_error(y_test, yhat))
print("R2 Score: ", r2_score(y_test, yhat))
```

```
Accuracy Score:  0.9113317357069497
Mean Squared Error (MSE):  0.08866826429305033
R2 Score:  -0.07123599960352833
```

Looking at these results there are a few things to point out:

1. Within the test set about 10% of the rows are churners (churn = 1).
2. Looking at the true negatives, we have 2651 out of 2655. This means that out of all the negative cases (churn = 0), we predicted 2651 as negative. This is really good!
3. 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 quite high at 255, we want to get the false negatives to as close to 0 as we can, so this would need to be addressed when improving the model.
4. Looking at false positives, this is where we have predicted a client to churn when they actually didn't churn. For this value we can see there are 4 cases, a fantastic prediction.
5. With the true positives, we can see that in total we have 266 clients that churned in the test dataset. However, we are only able to correctly identify 11 of those 266, which is very poor.
6. 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.
7. Looking at the precision score, this shows us a score of 0.91 which is good.
8. However, the recall of value 0.04 shows us that the classifier has a very poor ability to

identify positive samples. This would be the main concern for improving this model!

9. A R2 negative is not a mathematical impossibility or the sign of a computer bug. It simply means that the chosen model (with its constraints) fits the data really poorly.

So overall, we're able to very accurately identify clients that do not churn, but we are not able to predict cases where clients do churn. What we are seeing is that a high % of clients are being identified as not churning when they should be identified as churning. This in turn tells us that the current set of features are not discriminative enough to clearly distinguish between churners and non-churners.

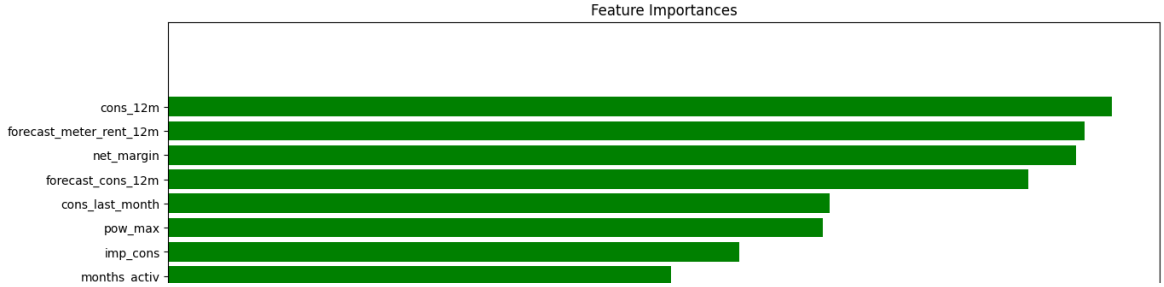
For now, lets dive into understanding the model a little more.

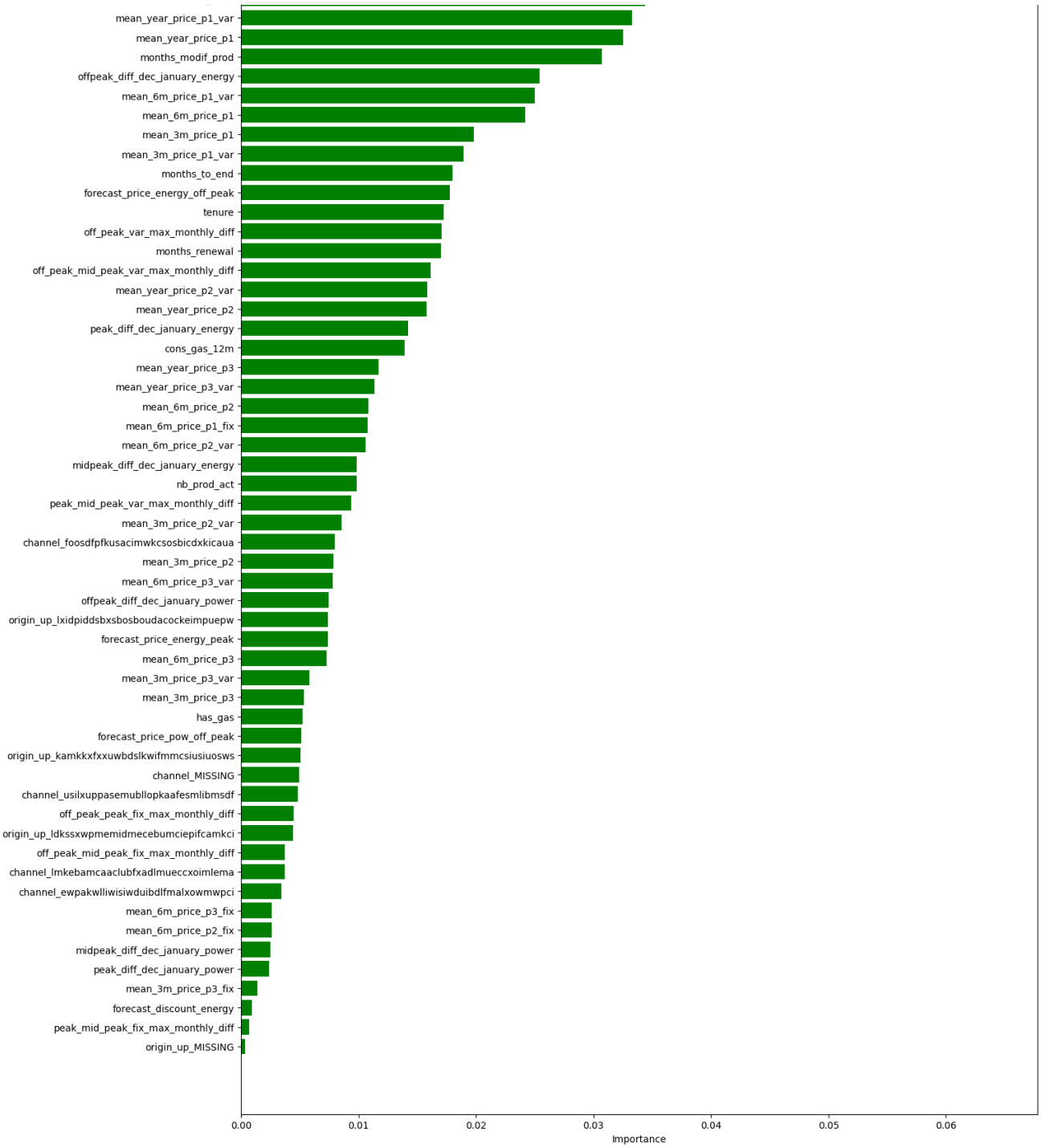
10.5 Model understanding

A simple way of understanding the results of a model is to look at feature importances. Feature importances indicate the importance of a feature within the predictive model, there are several ways to calculate feature importance, but with the Random Forest classifier, we're able to extract feature importances using the built-in method on the trained model. In the Random Forest case, the feature importance represents the number of times each feature is used for splitting across all trees

```
feature_importances = pd.DataFrame({
    'features': X_train.columns,
    'importance': random_forest.feature_importances_
}).sort_values(by='importance', ascending=True).reset_index()
```

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





11. Conclusions

From this chart, we can observe the following points:

1. Consumption over 12 months, forecasted bill of meter rental for the next 12 months and net margin and are top driver for churn in this model
2. Forecast cons 12 also is an influential driver
3. Time seems to be an influential factor, especially the number of months they have been active, their tenure and the number of months since they updated their contract
4. Our price sensitivity features are scattered around but are not the main driver for a customer churning

The last observation is important because this relates back to our original hypothesis:

> Is churn driven by the customers' price sensitivity?

Based on the output of the feature importances, it is not a main driver but it is a weak contributor.

```
proba_predictions = random_forest.predict_proba(X_test)
probabilities = proba_predictions[:, 1]
```

```
X_test = X_test.reset_index()
X_test.drop(columns='index', inplace=True)
```

```
X_test['churn'] = yhat.tolist()
X_test['churn_probability'] = probabilities.tolist()
X_test.to_csv('out_of_sample_data_with_predictions.csv')
X_test.head()
```

	cons_12m	cons_gas_12m	cons_last_month	forecast_cons_12m	forecast_discount_ener
0	4.084934	0.000000	3.173478	3.264197	
1	4.416690	0.000000	3.678518	3.585982	
2	3.389875	0.000000	0.000000	2.570858	
3	4.106972	3.899711	3.068557	3.139123	
4	2.733999	0.000000	0.000000	1.914872	

5 rows × 64 columns