Churn prediction with XGboost

1. Split dataset
   A. $wX = y$
   B. Training
   C. Validation
2. Modelling
   A. Xgboost
3. Model evaluation
   A. Accuracy, Precision, Recall
   B. ROC-AUC
   C. Stratified K-fold validation
4. Model finetuning
   A. Random search with cross validation
5. Understanding the model
   A. Feature importance
   B. Partial dependence plot (PDP)
   C. SHAP

---

0. Import packages

Load the necessary packages for this exercise

In [1]:
```
import datetime
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import pickle
import seaborn as sns
import shap
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
import xgboost as xgb
```

In [2]:
```
# Show plots in jupyter notebook
%matplotlib inline
```

In [3]:
```
# Set plot style
sns.set(color_codes=True)
```

In [4]:
```
# Set maximum number of columns to be displayed
pd.set_option('display.max_columns', 100)
```

In [5]:
```
# load JS visualization code to notebook
shap.initjs()
```

---

0. Loading data (pickle)

Data directory

Explicitly show how paths are indicated

In [6]:
```
DATA_DIR = os.path.join("..", "processed_data")
TRAINING_DATA = os.path.join(DATA_DIR, "train_data.pkl")
HISTORY_DATA = os.path.join(DATA_DIR, "history_data.pkl")
```

Load data into a dataframe

In [7]:
```
train_data = pd.read_pickle(TRAINING_DATA)
history_data = pd.read_pickle(HISTORY_DATA)
```

Load data into a dataframe

In [8]:
```
train = pd.merge(train_data, history_data, on="id")
```

Let's check we have all the data

Note: We've transformed the output to a dataframe to facilitate visualization

In [9]:
```
pd.DataFrame({"Dataframe columns": train.columns})
```

Out[9]:

	Dataframe columns
0	id
1	cons_12m
2	cons_gas_12m
3	cons_last_month
4	forecast_cons_12m
5	forecast_discount_energy
6	forecast_meter_rent_12m
7	forecast_price_energy_p1
8	forecast_price_energy_p2
9	forecast_price_pow_p1
10	has_gas
11	imp_cons
12	margin_gross_pow_ele
13	margin_net_pow_ele
14	nb_prod_act
15	net_margin
16	pow_max
17	churn
18	tenure
19	months_activ
20	months_to_end
21	months_modif_prod
22	months_renewal
23	channel_epu
24	channel_ewp
25	channel_fix
26	channel_foo
27	channel_lmk
28	channel_sdd
29	channel_usi
30	origin_ewx
31	origin_kam
32	origin_ldk
33	origin_lxi
34	origin_usa
35	activity_apd
36	activity_ckf
37	activity_clu
38	activity_cwo
39	activity_fmw
40	activity_kkk
41	activity_kwu
42	activity_sfi
43	activity_wxe
44	mean_year_price_p1_var
45	mean_year_price_p2_var
46	mean_year_price_p3_var
47	mean_year_price_p1_fix

	Dataframe columns
48	mean_year_price_p2_fix
49	mean_year_price_p3_fix
50	mean_year_price_p1
51	mean_year_price_p2
52	mean_year_price_p3

1. Splitting data

First of all we will split the data into the variable that we are trying to predict `y` (churn) and those variables that we will use to predict churn `X` (the rest)

In [10]:
```
y = train["churn"]
X = train.drop(labels = ["id","churn"],axis = 1)
```

Next we will split the data into `training` and `validation` data. The percentages of each test can be changed but a `75%-25%` is a good ratio. We also use a random state generator in order to split it randomly.

In [11]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=18)
```

2. Modelling

In [12]:
```
model = xgb.XGBClassifier(learning_rate=0.1,max_depth=6,n_estimators=500,n_jobs=-1)
result = model.fit(X_train,y_train)
```

3. Model evaluation

Accuracy, Precision, Recall

We are going to evaluate our Logistic Regression model on our test data (which we did not use for training) using the evalution metrics of:

> Accuracy : The most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations.

> Precision : The ratio of correctly predicted positive observations to the total predicted positive observations

> Recall (Sensitivity): The ratio of correctly predicted positive observations to the all observations in actual class

In [13]:
```
def evaluate(model_, X_test_, y_test_):
    """
    Evaluate the accuracy, precision and recall of a model
    """

    # Get the model predictions
    prediction_test_ = model_.predict(X_test_)

    # Print the evaluation metrics as pandas dataframe
    results = pd.DataFrame({"Accuracy" : [metrics.accuracy_score(y_test_, prediction_test_)],
                            "Precision" : [metrics.precision_score(y_test_, prediction_test_)],
                            "Recall" : [metrics.recall_score(y_test_, prediction_test_)]})

    # For a more detailed report
    #print(metrics.classification_report(y_test_, prediction_test_))
    return results
```

In [14]:
```
evaluate(model, X_test, y_test)
```

Out[14]:

	Accuracy	Precision	Recall
0	0.907306	0.769231	0.144578

ROC-AUC

Receiver Operating Characteristic(ROC) curve is a plot of the true positive rate against the false positive rate. It shows the tradeoff between sensitivity and specificity.

In a nutshell, **it tells how much model is capable of distinguishing between classes.**

In [15]:
```
def calculate_roc_auc(model_, X_test_, y_test_):
    """
    Evaluate the roc-auc score
    """

    # Get the model predictions
    # Note that we are using the prediction for the class 1 -> churn
    prediction_test_ = model_.predict_proba(X_test_)[:,1]

    # Compute roc-auc
    fpr, tpr, thresholds = metrics.roc_curve(y_test_, prediction_test_)

    # Print the evaluation metrics as pandas dataframe
    score = pd.DataFrame({"ROC-AUC" : [metrics.auc(fpr, tpr)]})

    return fpr, tpr, score

def plot_roc_auc(fpr,tpr):
    """
    Plot the Receiver Operating Characteristic from a list
    of true positive rates and false positive rates.
    """

    # Initialize plot
    f, ax = plt.subplots(figsize=(14,8))

    # Plot ROC
    roc_auc = metrics.auc(fpr, tpr)
    ax.plot(fpr, tpr, lw=2, alpha=0.3,
                label="AUC = %0.2f" % (roc_auc))

    # Plot the random line.
    plt.plot([0, 1], [0, 1], linestyle='--', lw=3, color='r',
             label="Random", alpha=.8)

    # Fine tune and show the plot.
    ax.set_xlim([-0.05, 1.05])
    ax.set_ylim([-0.05, 1.05])
    ax.set_xlabel("False Positive Rate (FPR)")
    ax.set_ylabel("True Positive Rate (TPR)")
    ax.set_title("ROC-AUC")
    ax.legend(loc="lower right")
    plt.show()
```

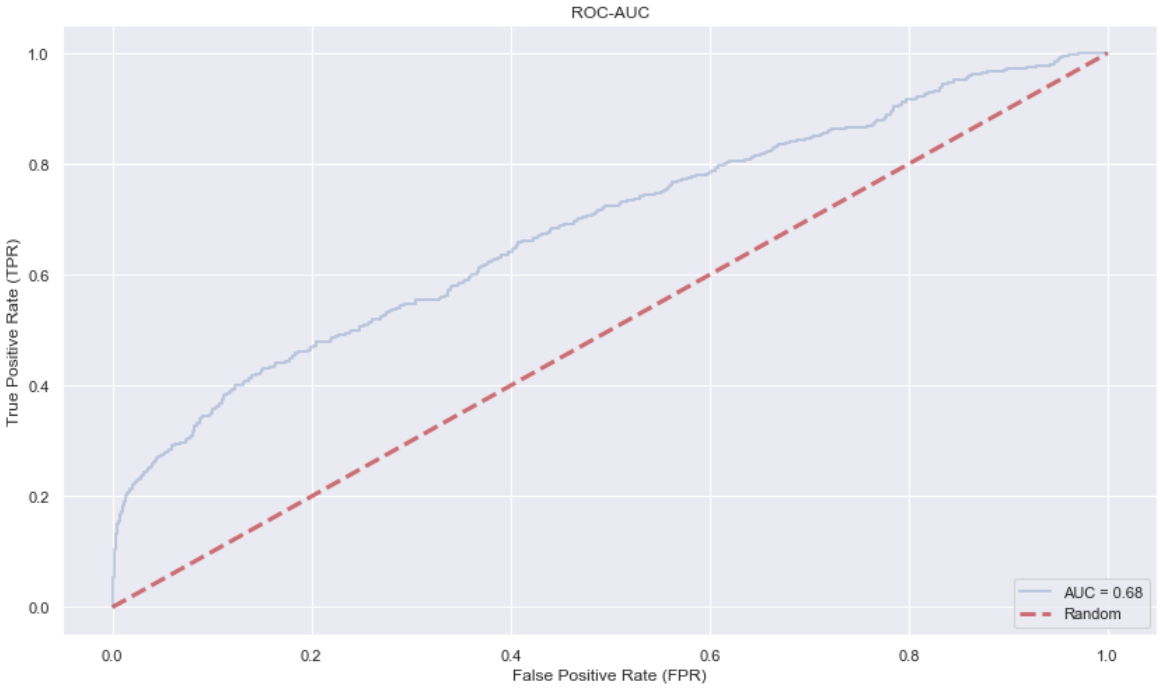
In [16]:
```
fpr, tpr, auc_score = calculate_roc_auc(model, X_test, y_test)
```

In [17]:
```
auc_score
```

Out[17]:

	ROC-AUC
0	0.684637

In [18]:
```
plot_roc_auc(fpr, tpr)
plt.show()
```



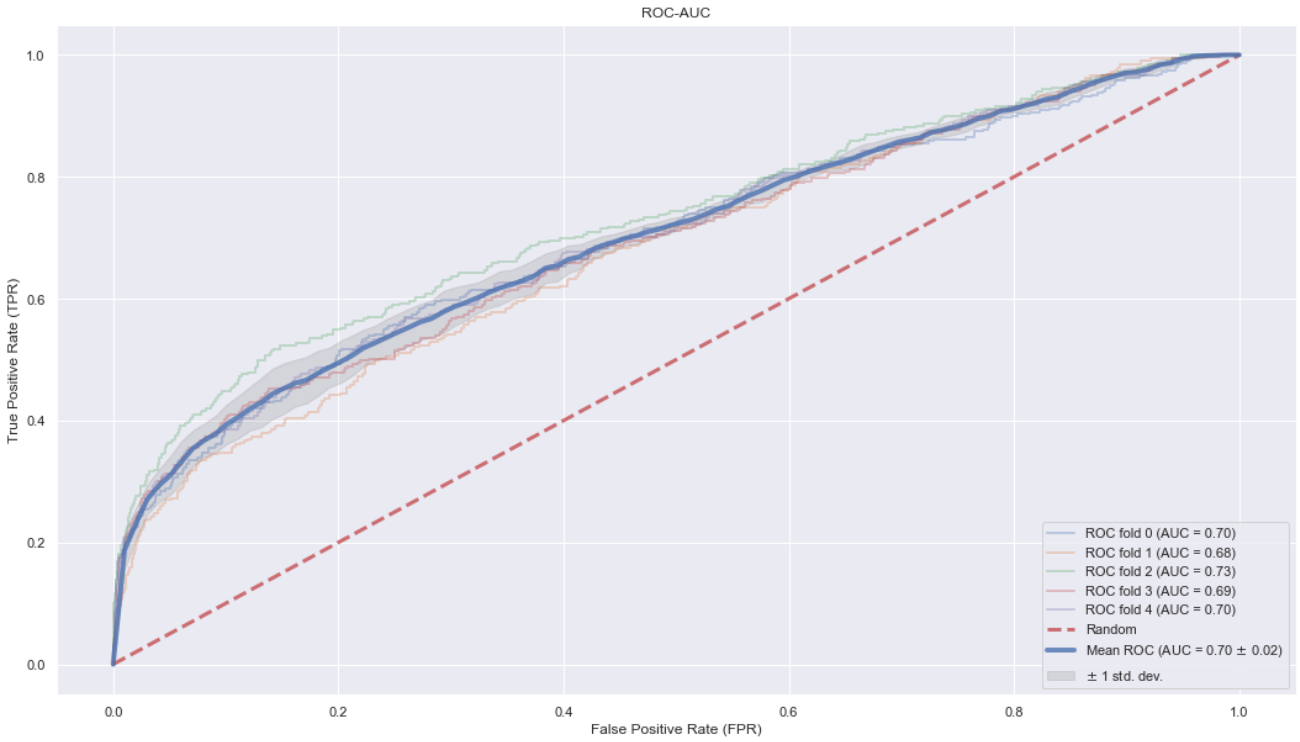
Stratified K-fold validation

In [19]:
```
def plot_roc_curve(fprs, tprs):
    """
    Plot the Receiver Operating Characteristic from a list
    of true positive rates and false positive rates.
    """

    # Initialize useful lists + the plot axes.
    tprs_interp = []
    aucs = []
    mean_fpr = np.linspace(0, 1, 100)
    f, ax = plt.subplots(figsize=(18,10))

    # Plot ROC for each K-Fold + compute AUC scores.
    for i, (fpr, tpr) in enumerate(zip(fprs, tprs)):
        tprs_interp.append(np.interp(mean_fpr, fpr, tpr))
        tprs_interp[-1][0] = 0.0
        roc_auc = metrics.auc(fpr, tpr)
        aucs.append(roc_auc)
        ax.plot(fpr, tpr, lw=2, alpha=0.3,
                label="ROC fold %d (AUC = %0.2f)" % (i, roc_auc))

    # Plot the luck line.
    plt.plot([0, 1], [0, 1], linestyle='--', lw=3, color='r',
             label="Random", alpha=.8)

    # Plot the mean ROC.
    mean_tpr = np.mean(tprs_interp, axis=0)
    mean_tpr[-1] = 1.0
    mean_auc = metrics.auc(mean_fpr, mean_tpr)
    std_auc = np.std(aucs)
    ax.plot(mean_fpr, mean_tpr, color='b',
             label=r"Mean ROC (AUC = %0.2f $\pm$ %0.2f)" % (mean_auc, std_auc),
             lw=4, alpha=.8)

    # Plot the standard deviation around the mean ROC.
    std_tpr = np.std(tprs_interp, axis=0)
    tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
    tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
    ax.fill_between(mean_fpr, tprs_lower, tprs_upper, color="grey", alpha=.2,
                    label=r"$\pm$ 1 std. dev.")

    # Fine tune and show the plot.
    ax.set_xlim([-0.05, 1.05])
    ax.set_ylim([-0.05, 1.05])
    ax.set_xlabel("False Positive Rate (FPR)")
    ax.set_ylabel("True Positive Rate (TPR)")
    ax.set_title("ROC-AUC")
    ax.legend(loc="lower right")
    plt.show()
    return (f, ax)

def compute_roc_auc(model_, index):
    y_predict = model_.predict_proba(X.iloc[index])[:,1]
    fpr, tpr, thresholds = metrics.roc_curve(y.iloc[index], y_predict)
    auc_score = metrics.auc(fpr, tpr)
    return fpr, tpr, auc_score
```

In [20]:
```
cv = StratifiedKFold(n_splits=5, random_state=13, shuffle=True)
fprs, tprs, scores = [], [], []
```

In [21]:
```
for (train, test), i in zip(cv.split(X, y), range(5)):
    model.fit(X.iloc[train], y.iloc[train])
    _, _, auc_score_train = compute_roc_auc(model, train)
    fpr, tpr, auc_score = compute_roc_auc(model, test)
    scores.append((auc_score_train, auc_score))
    fprs.append(fpr)
    tprs.append(tpr)
```

In [22]:
```
plot_roc_curve(fprs, tprs)
plt.show()
```



4. Model finetuning

Random Search Cross Validation

In [23]:
```
from sklearn.model_selection import RandomizedSearchCV
```

In [24]:
```
# Create the random grid
params = {
        'min_child_weight': [i for i in np.arange(1,15,1)],
        'gamma': [i for i in np.arange(0,6,0.5)],
        'subsample': [i for i in np.arange(0,1.1,0.1)],
        'colsample_bytree':  [i for i in np.arange(0,1.1,0.1)],
        'max_depth': [i for i in np.arange(1,15,1)],
        'scale_pos_weight':[i for i in np.arange(1,15,1)],
        'learning_rate': [i for i in np.arange(0,0.15,0.01)],
        'n_estimators' : [i for i in np.arange(0,2000,100)]
        }
```

We will create a new base model

In [25]:
```
# Create model
xg = xgb.XGBClassifier(objective='binary:logistic',
                       silent=True, nthread=1)
```

In [26]:
```
# Random search of parameters, using 5
xg_random = RandomizedSearchCV(xg, param_distributions=params,
                                   n_iter=1, scoring= "roc_auc",
                                   n_jobs=4, cv=5, verbose=3, random_state=1001)

# Fit the random search model
xg_random.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done   2 out of   5 | elapsed:   14.4s remaining:   21.6s
[Parallel(n_jobs=4)]: Done   5 out of   5 | elapsed:   18.6s finished
```

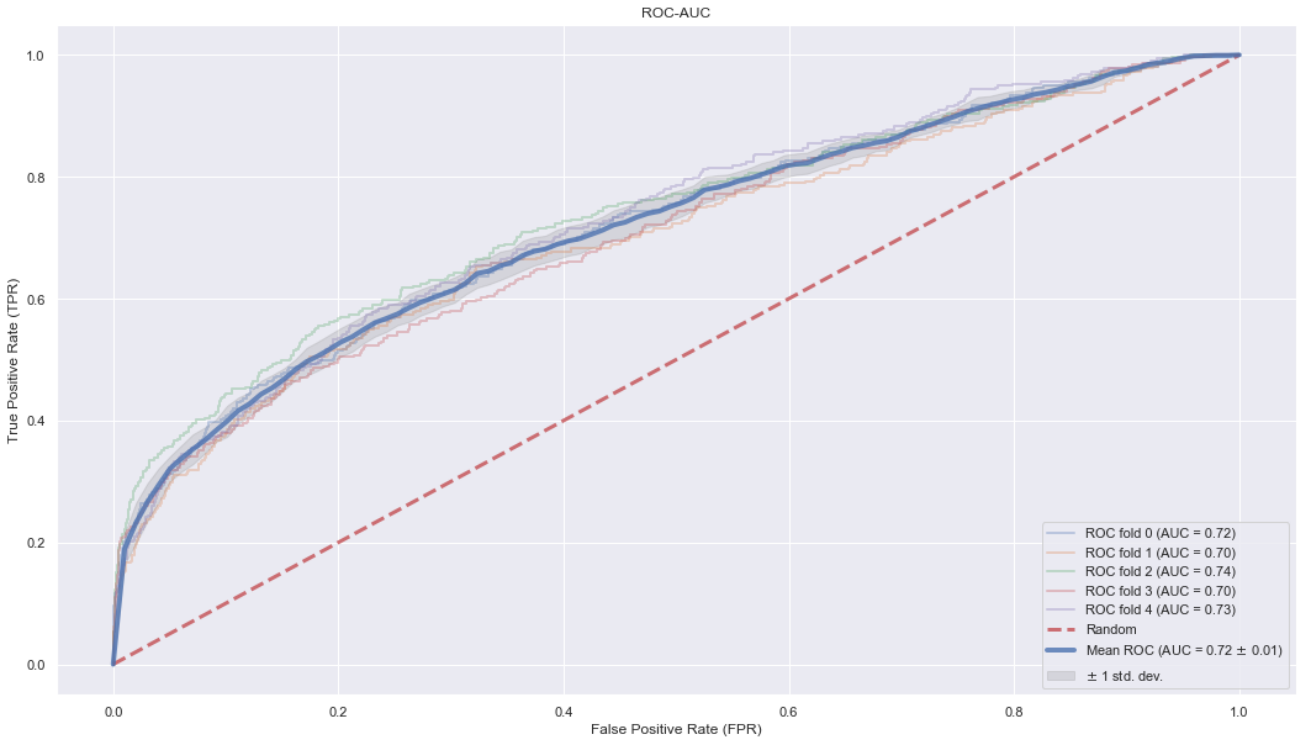
Out[26]:
```
RandomizedSearchCV(cv=5, error_score='raise-deprecating',
                   estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                           colsample_bylevel=1,
                                           colsample_bytree=1, gamma=0,
                                           learning_rate=0.1, max_delta_step=0,
                                           max_depth=3, min_child_weight=1,
                                           missing=None, n_estimators=100,
                                           n_jobs=1, nthread=1,
                                           objective='binary:logistic',
                                           random_state=0, reg_alpha=0,
                                           reg_lambda=1, scale_po...
                                          'n_estimators': [0, 100, 200, 300, 400,
                                                           500, 600, 700, 800,
                                                           900, 1000, 1100, 1200,
                                                           1300, 1400, 1500, 1600,
                                                           1700, 1800, 1900],
                                          'scale_pos_weight': [1, 2, 3, 4, 5, 6,
                                                               7, 8, 9, 10, 11,
                                                               12, 13, 14],
                                          'subsample': [0.0, 0.1, 0.2,
                                                        0.30000000000000004, 0.4,
                                                        0.5, 0.6000000000000001,
                                                        0.7000000000000001, 0.8,
                                                        0.9, 1.0]},
                   pre_dispatch='2*n_jobs', random_state=1001, refit=True,
                   return_train_score=False, scoring='roc_auc', verbose=3)
```

In [27]:
```
best_random = xg_random.best_params_
best_random = {'subsample': 0.8,
 'scale_pos_weight': 1,
 'n_estimators': 1100,
 'min_child_weight': 1,
 'max_depth': 12,
 'learning_rate': 0.01,
 'gamma': 4.0,
 'colsample_bytree': 0.60}
```

In [28]:
```
# Create a model with the parameters found
model_random = xgb.XGBClassifier(objective='binary:logistic',
                      silent=True, nthread=1, **best_random)
fprs, tprs, scores = [], [], []
```

In [29]:
```
for (train, test), i in zip(cv.split(X, y), range(5)):
    model_random.fit(X.iloc[train], y.iloc[train])
    _, _, auc_score_train = compute_roc_auc(model_random, train)
    fpr, tpr, auc_score = compute_roc_auc(model_random, test)
    scores.append((auc_score_train, auc_score))
    fprs.append(fpr)
    tprs.append(tpr)
```

In [30]:
```
plot_roc_curve(fprs, tprs)
plt.show()
```



Grid search with cross validation (calculating over weekend, then make smaller)

In [31]:
```
from sklearn.model_selection import GridSearchCV
```

In [32]:
```
# Create the parameter grid based on the results of random search
param_grid =   {'subsample': [0.7],
                'scale_pos_weight': [1],
                'n_estimators': [1100],
                'min_child_weight': [1],
                'max_depth': [12, 13, 14],
                'learning_rate': [0.005, 0.01],
                'gamma': [4.0],
                'colsample_bytree': [0.6]}
```

In [33]:
```
# Create model
xg = xgb.XGBClassifier(objective='binary:logistic',
                       silent=True, nthread=1)
```

In [34]:
```
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = xg, param_grid = param_grid,
                           cv = 5, n_jobs = -1, verbose = 2, scoring = "roc_auc")
```

In [35]:
```
# Fit the grid search to the data
grid_search.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed: 20.7min finished
```

Out[35]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                     colsample_bylevel=1, colsample_bytree=1,
                                     gamma=0, learning_rate=0.1,
                                     max_delta_step=0, max_depth=3,
                                     min_child_weight=1, missing=None,
                                     n_estimators=100, n_jobs=1, nthread=1,
                                     objective='binary:logistic',
                                     random_state=0, reg_alpha=0, reg_lambda=1,
                                     scale_pos_weight=1, seed=None, silent=True,
                                     subsample=1),
             iid='warn', n_jobs=-1,
             param_grid={'colsample_bytree': [0.6], 'gamma': [4.0],
                         'learning_rate': [0.005, 0.01],
                         'max_depth': [12, 13, 14], 'min_child_weight': [1],
                         'n_estimators': [1100], 'scale_pos_weight': [1],
                         'subsample': [0.7]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='roc_auc', verbose=2)
```

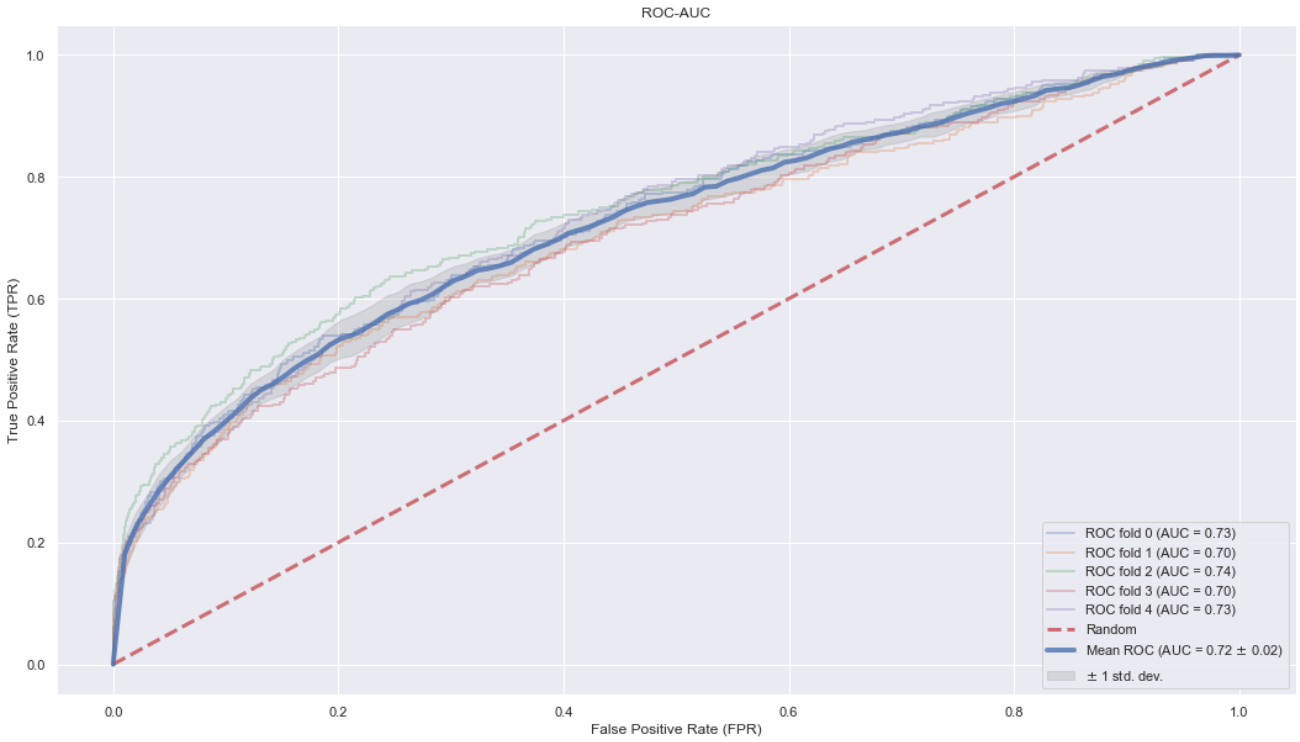
In [36]:
```
best_grid = grid_search.best_params_
best_grid
```

Out[36]:
```
{'colsample_bytree': 0.6,
 'gamma': 4.0,
 'learning_rate': 0.005,
 'max_depth': 14,
 'min_child_weight': 1,
 'n_estimators': 1100,
 'scale_pos_weight': 1,
 'subsample': 0.7}
```

In [37]:
```
# Create a model with the parameters found
model_grid = xgb.XGBClassifier(objective='binary:logistic',
                       silent=True, nthread=1, **best_grid)
fprs, tprs, scores = [], [], []
```

In [38]:
```
for (train, test), i in zip(cv.split(X, y), range(5)):
    model_grid.fit(X.iloc[train], y.iloc[train])
    _, _, auc_score_train = compute_roc_auc(model_grid, train)
    fpr, tpr, auc_score = compute_roc_auc(model_grid, test)
    scores.append((auc_score_train, auc_score))
    fprs.append(fpr)
    tprs.append(tpr)
```

In [39]:
```
plot_roc_curve(fprs, tprs)
plt.show()
```



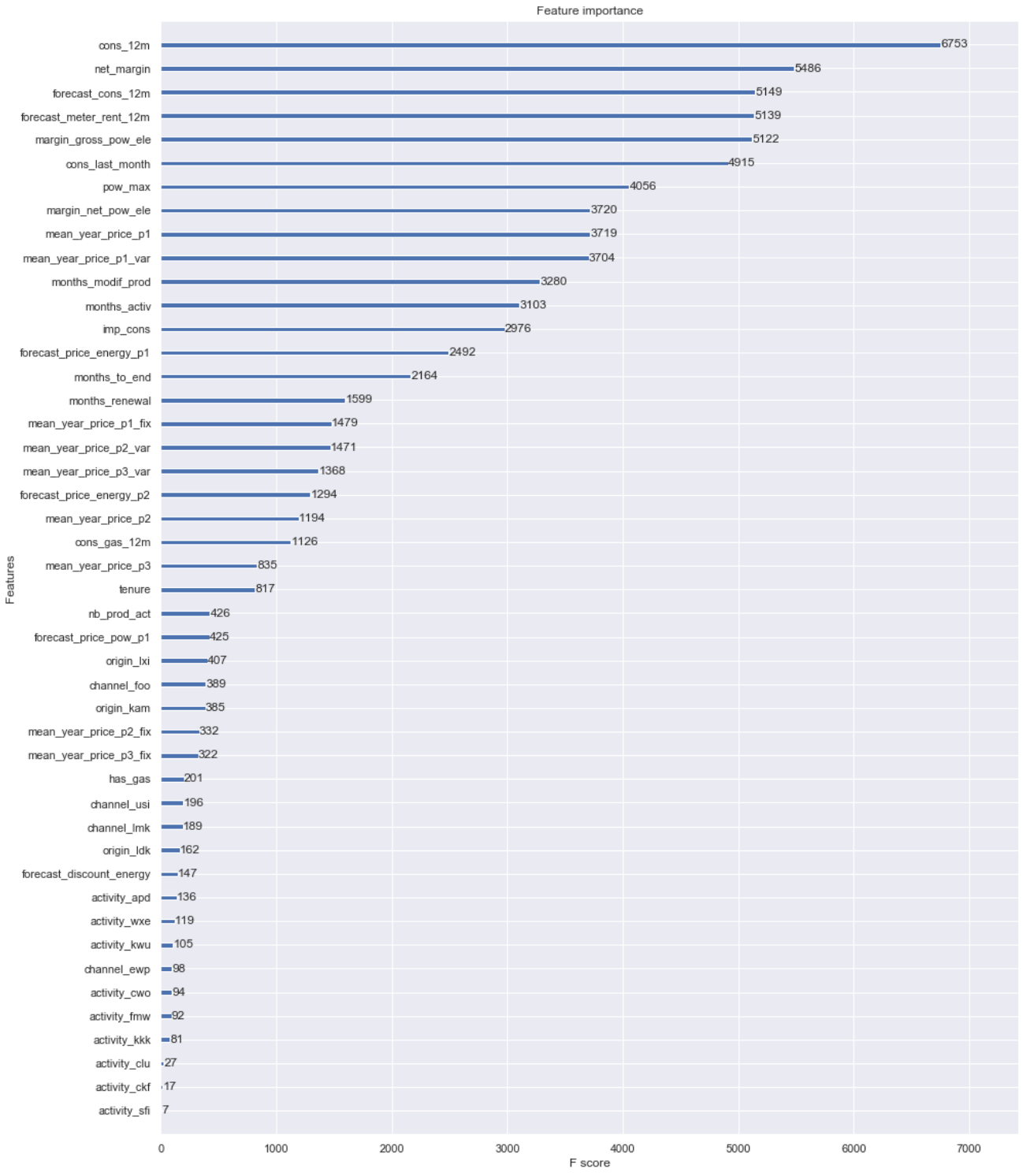
5. Understanding the model

Feature importance

One simple way of boserving the feature importance is through counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear

In [40]:
```
fig, ax = plt.subplots(figsize=(15,20))
xgb.plot_importance(model_grid, ax=ax)
```

Out[40]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2700ccc6f98>
```



In the `feature importance` graph above we can see that `cons_12m` and `net_margin` are the features that appear the most in our model and we could infere that these two features have a significant importnace in our model

Partial dependence plot

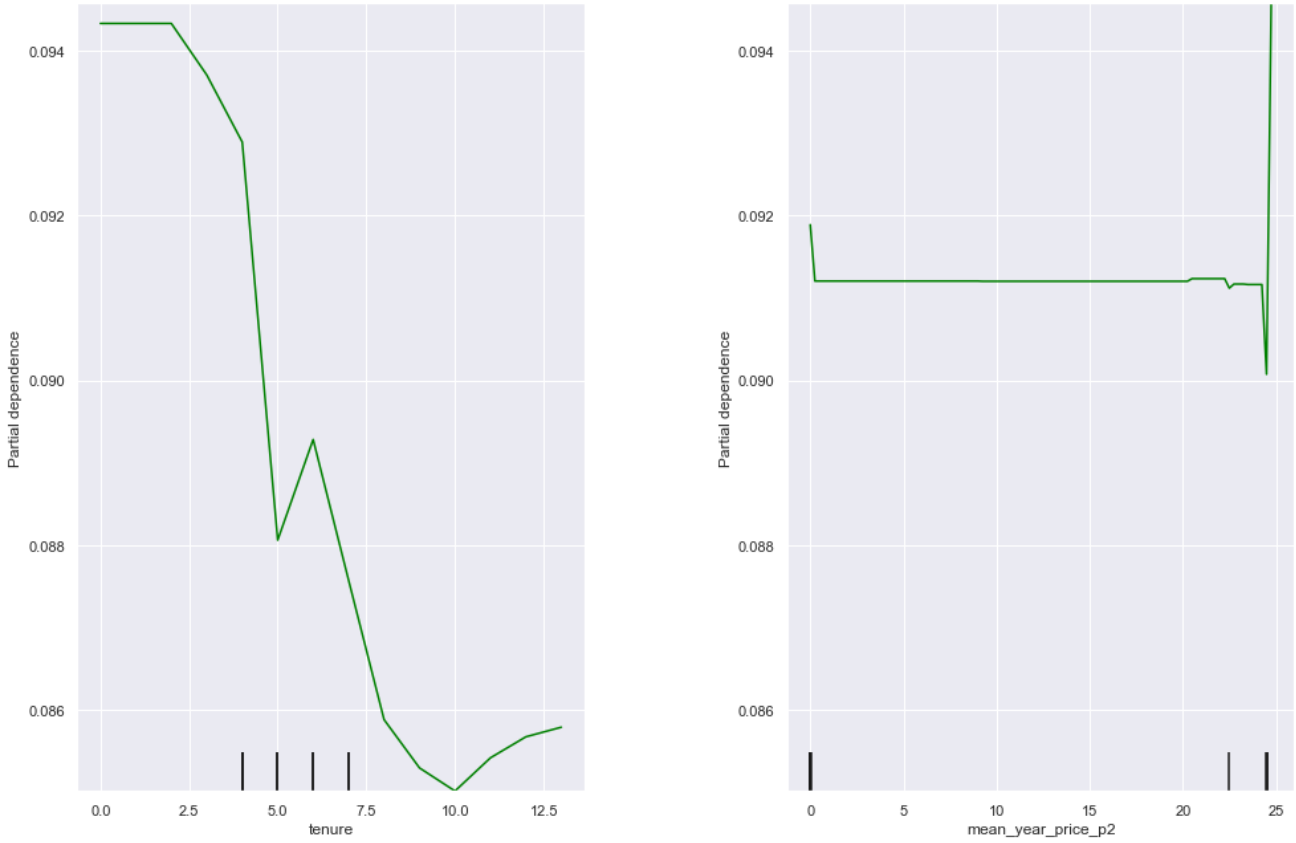
In [41]:
```
from sklearn.inspection import plot_partial_dependence
```

Because currently there is a bug that does not allow us to use our trained model with pandas dataframes, we will create a replica and train it using numpy arrays

In [42]:
```
# Create a model with the parameters found
model_grid_v2 = xgb.XGBClassifier(objective='binary:logistic',
                      silent=True, nthread=1, **best_grid)
model_grid_v2.fit(X_train.values,y_train.values)
```

Out[42]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bytree=0.6, gamma=4.0, learning_rate=0.005,
              max_delta_step=0, max_depth=14, min_child_weight=1, missing=None,
              n_estimators=1100, n_jobs=1, nthread=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=1, seed=None, silent=True,
              subsample=0.7)
```

In [43]:
```
fig = plt.figure(figsize=(15,15))
plot_partial_dependence(model_grid_v2, X_test.values,
                        features=[16, 49],
                        feature_names=X_test.columns.tolist(),
                        fig=fig)
```



Comparing the PDP plots with respect to our previous models, we can see how they are slightly different. `tenure`

The overall trend is unchaged as compared to our previous models. We can see the trend spikes at slighly different times of the tenure (`6y`) but then it goes down again and bottoms around 10 years. Then, it starts recovering a bit.

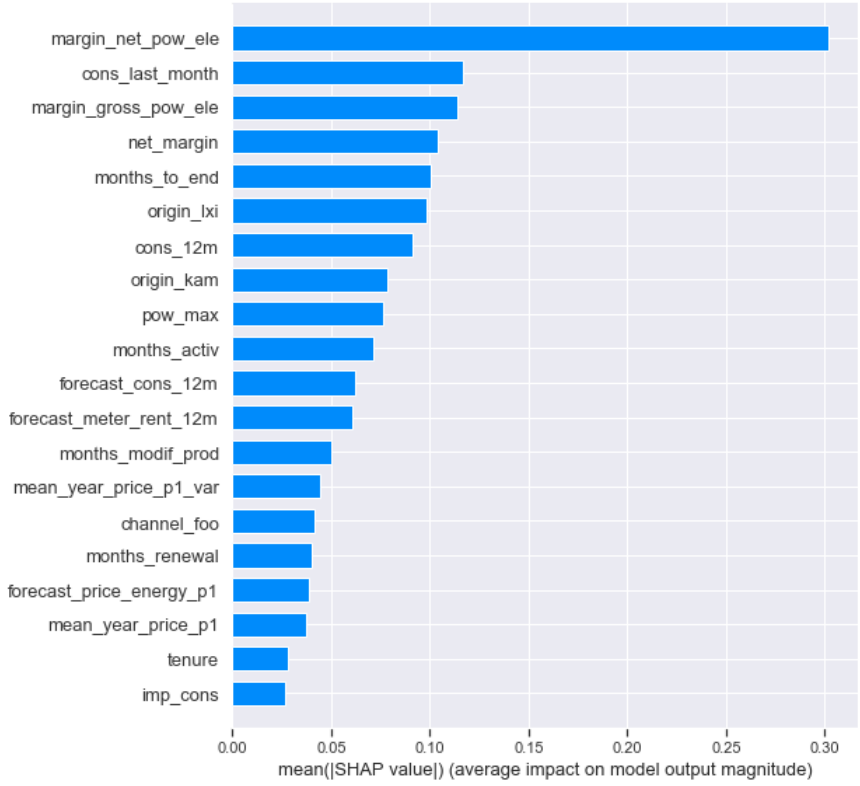
`mean_year_price_p2`

In our previous models, we saw a sort of "stairshape", in this case we see the pdp is almost flat with some spikes on the extreme values, which hints us that the variable `mean_year_price_p2` is not very relevant in this model.

SHAP - Feature importance

In [44]:
```
explainer = shap.TreeExplainer(model_grid)
shap_values = explainer.shap_values(X_test)
```

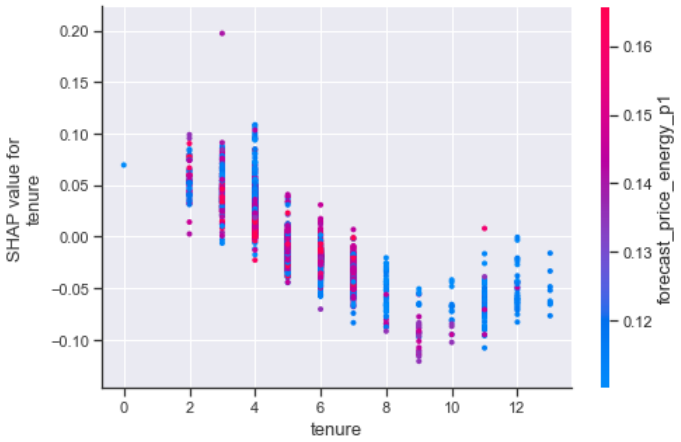
In [45]:
```
# Feature importance for class 1 - churn
shap.summary_plot(shap_values, X_test, plot_type='bar')
```



As expected the `margin_net_pow_ele` is the most important feature by far. It is interesting to compare how much important the top feature becomes in contrast with the other models we created `Random forest` and `Logistic Regression`

SHAP - Partial dependence plot

In [46]:
```
shap.dependence_plot("tenure", shap_values, X_test)#, interaction_index="origin_lxi")
```

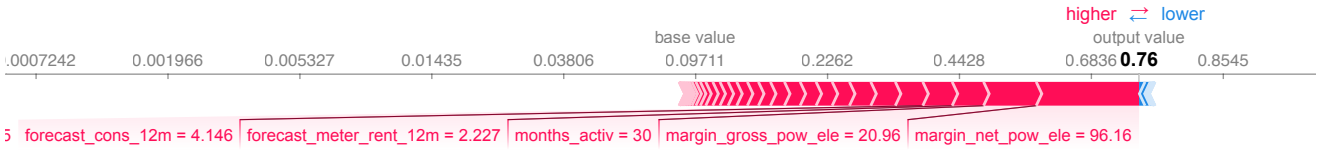


In this case we see a much clearer pattern, in which the longer the tenure the less likely the company is, sort of decreasing linearly until it bottoms around `9y` of tenure. From year 10 of tenure, the churn increases again.

SHAP - Single prediction

In [47]:
```
shap.force_plot(explainer.expected_value, shap_values[4023], X_test.iloc[4023,:], link="logit")
```

Out[47]:



We can see in here how the different features interacted to result in the above prediction. In this case, the high values of the `margin_net_pow_ele` and `margin_gross_pow_ele` pushed the likely of churning way above the baseline.

In [ ]:
```
```