BankCustomerChurnPrediction

January 14, 2022

1 Bank Customer Churn Prediction

1.1 First step: problem framing, knowing what problem we are trying to solve–identify customers who are likely to close their account in the future, and predict the probability that a customer is about to churn.

Motivation: Take actions to keep the customers that are about to churn. For example, send discount, make new products and send emails, push app

In this project, we use supervised learning models to identify customers who are likely to churn in the future. Furthermore, we will analyze top factors that influence user retention.

Data resource: https://www.kaggle.com/adammaus/predicting-churn-for-bank-customers

1.2 Contents

- Part 1: Data Exploration
- Part 2: Feature Preprocessing
- Part 3: Model Training and Results Evaluation

2 Part 0: Setup Google Drive Environment / Data Collection

reference: https://colab.research.google.com/notebooks/io.ipynb

autherize drive to get data

[67]:
```
# install pydrive to load data
!pip install -U -q PyDrive

from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

auth.authenticate_user()
gauth = GoogleAuth()
```

```
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

tell drive where to get the data, load data and set data a new name

[68]:
```
id = "1um4-JcKVHitHeQeOOQIYofywgv1Bi8up"
file = drive.CreateFile({'id':id})
file.GetContentFile('bank_churn.csv')
```

3 Part 1: EDA

3.0.1 Part 1.1: Understand the Raw Dataset

[69]:
```
import pandas as pd
import numpy as np

churn_df = pd.read_csv('bank_churn.csv')
```

[70]:
```
churn_df.head()
```

[70]:
```
   RowNumber  CustomerId   Surname  …  IsActiveMember EstimatedSalary Exited
0          1    15634602  Hargrave  …               1       101348.88      1
1          2    15647311      Hill  …               1       112542.58      0
2          3    15619304      Onio  …               0       113931.57      1
3          4    15701354      Boni  …               0        93826.63      0
4          5    15737888  Mitchell  …               1        79084.10      0

[5 rows x 14 columns]
```

Tenure: IsActiveMember how to tell?

[71]:
```
# check data info
churn_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   RowNumber        10000 non-null  int64
 1   CustomerId       10000 non-null  int64
 2   Surname          10000 non-null  object
 3   CreditScore      10000 non-null  int64
 4   Geography        10000 non-null  object
 5   Gender           10000 non-null  object
 6   Age              10000 non-null  int64
```

```
 7   Tenure           10000 non-null  int64
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64
 10  HasCrCard        10000 non-null  int64
 11  IsActiveMember   10000 non-null  int64
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

no missing data

[72]:
```python
# check the unique values for each column
churn_df.nunique()
```

[72]:
```
RowNumber          10000
CustomerId         10000
Surname             2932
CreditScore          460
Geography              3
Gender                 2
Age                   70
Tenure                11
Balance             6382
NumOfProducts          4
HasCrCard              2
IsActiveMember         2
EstimatedSalary     9999
Exited                 2
dtype: int64
```

RowNumber == #CustomerId indicates that there's no duplicates; These two are irrelavant variables; Surname(last name) doesn't have an impact on bank behavior in common sense; 3 countries; Balance: over 3k people have same balance(0?)

[73]:
```python
# Get label
y = churn_df['Exited']
```

3.0.2 Part 1.2: Understand the features

[74]:
```python
# check missing values
churn_df.isnull().sum()
```

[74]:
```
RowNumber          0
CustomerId         0
Surname            0
CreditScore        0
```

```
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

[75]:
```python
# understand Numerical feature
# discrete/continuous
# 'CreditScore', 'Age', 'Tenure', 'NumberOfProducts'
# 'Balance', 'EstimatedSalary'
churn_df[['CreditScore', 'Age', 'Tenure', 'NumOfProducts','Balance',␣
↪'EstimatedSalary']].describe()
```

[75]:

	CreditScore	Age	…	Balance	EstimatedSalary
count	10000.000000	10000.000000	…	10000.000000	10000.000000
mean	650.528800	38.921800	…	76485.889288	100090.239881
std	96.653299	10.487806	…	62397.405202	57510.492818
min	350.000000	18.000000	…	0.000000	11.580000
25%	584.000000	32.000000	…	0.000000	51002.110000
50%	652.000000	37.000000	…	97198.540000	100193.915000
75%	718.000000	44.000000	…	127644.240000	149388.247500
max	850.000000	92.000000	…	250898.090000	199992.480000

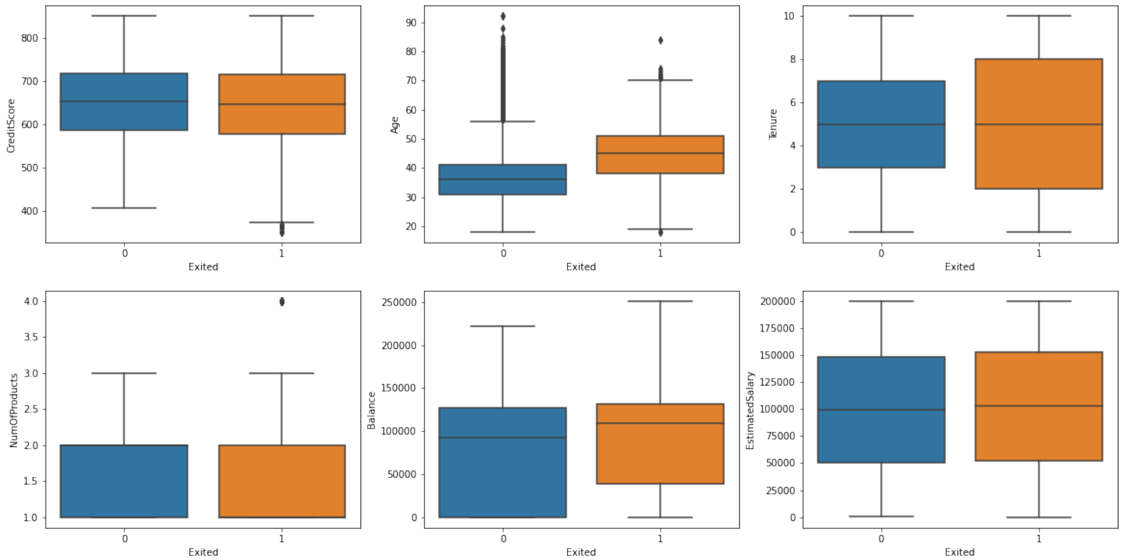
[8 rows x 6 columns]

Minimum estimate salary: 11.580000? Is it an outlier?

[76]:
```python
# check the feature distribution
# pandas.DataFrame.describe()
# boxplot, distplot, countplot
import matplotlib.pyplot as plt
import seaborn as sns
```

[77]:
```python
# boxplot for numerical feature
_,axss = plt.subplots(2,3, figsize=[20,10])
sns.boxplot(x='Exited', y ='CreditScore', data=churn_df, ax=axss[0][0])
sns.boxplot(x='Exited', y ='Age', data=churn_df, ax=axss[0][1])
sns.boxplot(x='Exited', y ='Tenure', data=churn_df, ax=axss[0][2])
sns.boxplot(x='Exited', y ='NumOfProducts', data=churn_df, ax=axss[1][0])
sns.boxplot(x='Exited', y ='Balance', data=churn_df, ax=axss[1][1])
sns.boxplot(x='Exited', y ='EstimatedSalary', data=churn_df, ax=axss[1][2])
```

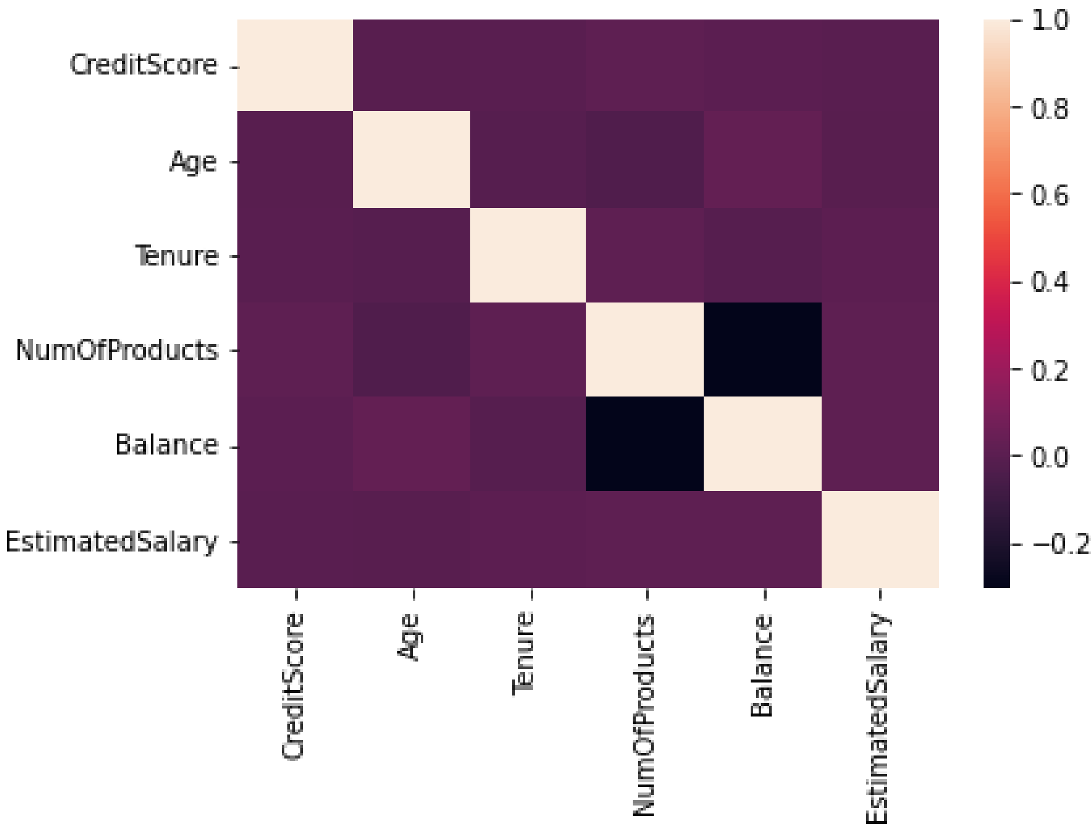
[77]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa63c155cd0>
```



People tend to churn: order, more tenure, more balance

[78]:
```
# correlations between features
corr_score = churn_df[['CreditScore', 'Age', 'Tenure',␣
↪'NumOfProducts','Balance', 'EstimatedSalary']].corr()

# show heapmap of correlations
sns.heatmap(corr_score)
```

[78]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa63c02bf50>
```



[79]:
```
# check the actual values of correlations
corr_score
```

[79]:

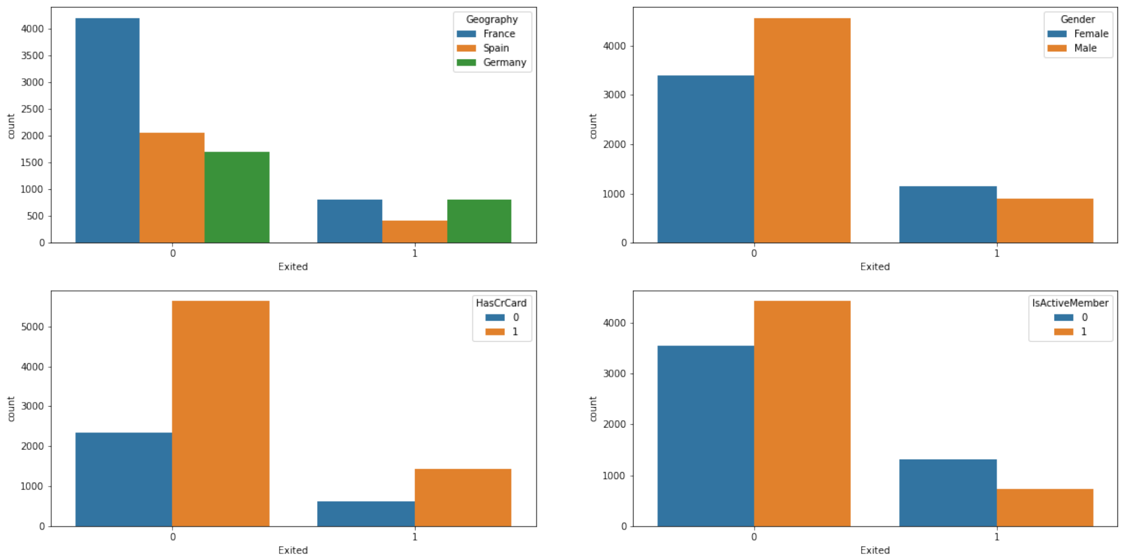
	CreditScore	Age	…	Balance	EstimatedSalary
CreditScore	1.000000	-0.003965	…	0.006268	-0.001384
Age	-0.003965	1.000000	…	0.028308	-0.007201
Tenure	0.000842	-0.009997	…	-0.012254	0.007784
NumOfProducts	0.012238	-0.030680	…	-0.304180	0.014204
Balance	0.006268	0.028308	…	1.000000	0.012797
EstimatedSalary	-0.001384	-0.007201	…	0.012797	1.000000

[6 rows x 6 columns]

[80]:
```
# understand categorical feature
# 'Geography', 'Gender'
# 'HasCrCard', 'IsActiveMember'
_,axss = plt.subplots(2,2, figsize=[20,10])
sns.countplot(x='Exited', hue='Geography', data=churn_df, ax=axss[0][0])
sns.countplot(x='Exited', hue='Gender', data=churn_df, ax=axss[0][1])
sns.countplot(x='Exited', hue='HasCrCard', data=churn_df, ax=axss[1][0])
```

```python
sns.countplot(x='Exited', hue='IsActiveMember', data=churn_df, ax=axss[1][1])
```

[80]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa63c7f4a90>



Ratio of Germany user that churns is the highest might related to culture) Female tends to churn; Non-active users tend to churn

4 Part 2: Feature Preprocessing

[81]:
```python
# drop useless feature
to_drop = ['RowNumber','CustomerId','Surname','Exited']
X = churn_df.drop(to_drop, axis=1)
```

[82]:
```python
X.head()
```

[82]:
```
   CreditScore Geography  Gender  …  HasCrCard  IsActiveMember
EstimatedSalary
0          619    France  Female  …          1               1
101348.88
1          608     Spain  Female  …          0               1
112542.58
2          502    France  Female  …          1               0
113931.57
3          699    France  Female  …          0               0
93826.63
4          850     Spain  Female  …          1               1
79084.10
```

```
[5 rows x 10 columns]
```

[83]:
```
X.dtypes
```

[83]:
```
CreditScore          int64
Geography           object
Gender              object
Age                  int64
Tenure               int64
Balance            float64
NumOfProducts        int64
HasCrCard            int64
IsActiveMember       int64
EstimatedSalary    float64
dtype: object
```

[84]:
```
cat_cols = X.columns[X.dtypes == 'O']
num_cols = X.columns[(X.dtypes == 'float64') | (X.dtypes == 'int64')]
```

[85]:
```
num_cols
```

[85]:
```
Index(['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard',
       'IsActiveMember', 'EstimatedSalary'],
      dtype='object')
```

[86]:
```
cat_cols
```

[86]:
```
Index(['Geography', 'Gender'], dtype='object')
```

Split dataset before deal with categorical variables, since we don't want to change testing set

[87]:
```
from sklearn import model_selection

#train: test = 3:1 # use stratified sampling to balance the ratio of people␣
↪churn and stay (especially when data size is small)
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,␣
↪test_size=0.25, stratify = y, random_state=888)
print('training data has ' + str(X_train.shape[0]) + ' observation with ' +␣
↪str(X_train.shape[1]) + ' features')
print('test data has ' + str(X_test.shape[0]) + ' observation with ' +␣
↪str(X_test.shape[1]) + ' features')
```

```
training data has 7500 observation with 10 features
test data has 2500 observation with 10 features
```

- 10000 -> 8000 '0' + 2000 '1'
-

4.1 25% test 75% training

without stratified sampling: • extreme case: —

1. testing: 2000 ‘1’ + 500 ‘0’
2.

4.2 training: 7500 ‘0’ (can not build model on only churn people)

with stratified sampling:

3. testing: 2000 ‘0’ + 500 ‘1’
4. training: 6000 ‘0’ + 1500 ‘1’

[88]:
```
X_train.head()
```

[88]:
```
      CreditScore Geography  Gender  …  HasCrCard  IsActiveMember
EstimatedSalary
4719          566   Germany  Female  …          1               0
66245.44
3591          769    France    Male  …          1               0
84872.66
2393          850   Germany    Male  …          1               0
60708.72
6733          668    France    Male  …          1               0
193018.71
2091          661    France  Female  …          1               0
81102.81

[5 rows x 10 columns]
```

A set of scikit-learn-style transformers for encoding categorical variables into numeric with different techniques: http://contrib.scikit-learn.org/category_encoders/

[89]:
```
# apply One-hot encoding to geography
from sklearn.preprocessing import OneHotEncoder

def OneHotEncoding(df, enc, categories):
  transformed = pd.DataFrame(enc.transform(df[categories]).toarray(), 
 ↪columns=enc.get_feature_names(categories))
  return pd.concat([df.reset_index(drop=True), transformed], axis=1).
 ↪drop(categories, axis=1)

categories = ['Geography']
enc_ohe = OneHotEncoder()
#use training set to fit one hot
enc_ohe.fit(X_train[categories])
```

```
X_train = OneHotEncoding(X_train, enc_ohe, categories)
X_test = OneHotEncoding(X_test, enc_ohe, categories)#apply enc_ohe to testing␣
 ↪set
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

[25]:
```
X_train.head()
```

[25]:
```
   CreditScore  Gender  …  Geography_Germany  Geography_Spain
0          566  Female  …                1.0              0.0
1          769    Male  …                0.0              0.0
2          850    Male  …                1.0              0.0
3          668    Male  …                0.0              0.0
4          661  Female  …                0.0              0.0

[5 rows x 12 columns]
```

[90]:
```
# Ordinal encoding (one-hot is better, since ordinal gives a distance of␣
 ↪categorical variables)
# from sklearn.preprocessing import OrdinalEncoder

categories1 = ['Gender']
# enc_oe = OrdinalEncoder()
# enc_oe.fit(X_train[categories])

# X_train[categories] = enc_oe.transform(X_train[categories])
# X_test[categories] = enc_oe.transform(X_test[categories])
enc_ohe1 = OneHotEncoder()
enc_ohe1.fit(X_train[categories1])

X_train = OneHotEncoding(X_train, enc_ohe1, categories1)
X_test = OneHotEncoding(X_test, enc_ohe1, categories1)#apply enc_ohe to testing␣
 ↪set
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
```

```
instead.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

[27]:
```
X_train.head()
```

[27]:

	CreditScore	Age	Tenure	…	Geography_Spain	Gender_Female	Gender_Male
0	566	35	1	…	0.0	1.0	0.0
1	769	29	2	…	0.0	0.0	1.0
2	850	28	4	…	0.0	0.0	1.0
3	668	28	4	…	0.0	0.0	1.0
4	661	37	5	…	0.0	1.0	0.0

[5 rows x 13 columns]

Standardize/Normalize Data: without this step, algorithms based on distance would be influenced, like linear regression, logistic regression and knn

[28]:
```
# Scale the data, using standardization
# standardization (x-mean)/std
# normalization (x-x_min)/(x_max-x_min) ->[0,1]

# 1. speed up gradient descent
# 2. same scale
# 3. algorithm requirments

# for example, use training data to train the standardscaler to get mean and␣
↪std
# apply mean and std to both training and testing data.
# fit_transform does the training and applying, transform only does applying.
# Because we can't use any info from test, and we need to do the same␣
↪modification
# to testing data as well as training data

# https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.
↪html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py
# https://scikit-learn.org/stable/modules/preprocessing.html

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train[num_cols])
X_train[num_cols] = scaler.transform(X_train[num_cols])
X_test[num_cols] = scaler.transform(X_test[num_cols])
```

[29]:
```
X_train.head()
```

[29]:
```
   CreditScore       Age   Tenure  …  Geography_Spain  Gender_Female
Gender_Male
0    -0.877394 -0.375981 -1.393024  …              0.0            1.0
0.0
1     1.223120 -0.950309 -1.048159  …              0.0            0.0
1.0
2     2.061257 -1.046031 -0.358429  …              0.0            0.0
1.0
3     0.178037 -1.046031 -0.358429  …              0.0            0.0
1.0
4     0.105605 -0.184538 -0.013565  …              0.0            1.0
0.0

[5 rows x 13 columns]
```

5 Part 3: Model Training and Result Evaluation

5.0.1 Part 3.1: Model Training

Prediction are categorical discrete numbers, so we use Logistic Regression.

KNN is the mostly wide used supervised classifier, which classifies the data by finding the type of its k nearest points.

[30]:
```
#@title build models
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

# Logistic Regression
classifier_logistic = LogisticRegression()

# K Nearest Neighbors
classifier_KNN = KNeighborsClassifier()

# Random Forest
classifier_RF = RandomForestClassifier()
```

[31]:
```
# Train the model on training set
classifier_logistic.fit(X_train, y_train)
```

[31]:
```
LogisticRegression()
```

[32]:
```
# Prediction of test data
classifier_logistic.predict(X_test)
```

[32]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[33]:
```
# Accuracy of test data
classifier_logistic.score(X_test, y_test)
```

[33]:
```
0.81
```

[34]:
```
# Use 5-fold Cross Validation to get the accuracy for different models
model_names = ['Logistic Regression','KNN','Random Forest']
model_list = [classifier_logistic, classifier_KNN, classifier_RF]
count = 0

for classifier in model_list:
    cv_score = model_selection.cross_val_score(classifier, X_train, y_train,␣
 ↪cv=5)
    print(cv_score)
    print('Model accuracy of ' + model_names[count] + ' is ' + str(cv_score.
 ↪mean()))
    count += 1
```

```
[0.80466667 0.806      0.81933333 0.81466667 0.80733333]
Model accuracy of Logistic Regression is 0.8104000000000001
[0.82866667 0.832      0.834      0.836      0.82533333]
Model accuracy of KNN is 0.8311999999999999
[0.858      0.86066667 0.866      0.86133333 0.858     ]
Model accuracy of Random Forest is 0.8607999999999999
```

Random Forest gives the best performance.

5.0.2 (Optional) Part 3.2: Use Grid Search to Find Optimal Hyperparameters

alternative: random search

[35]:
```
#Loss/cost function --> (wx + b - y) ^2 +   * |w| --> lambda is a hyperparameter
```

[36]:
```
from sklearn.model_selection import GridSearchCV

# helper function for printing out best grid search results
def print_grid_search_metrics(gs):
    print ("Best score: " + str(gs.best_score_))
    print ("Best parameters set:")
    best_parameters = gs.best_params_
    for param_name in sorted(best_parameters.keys()):
        print(param_name + ':' + str(best_parameters[param_name]))
```

Part 3.2.1: Find Optimal Hyperparameters - LogisticRegression

[37]:
```python
# Possible hyperparamter options for Logistic Regression Regularization
# Penalty is choosed from L1 or L2
# C is the 1/lambda value(weight) for L1 and L2
# solver: algorithm to find the weights that minimize the cost function

# ('l1', 0.01)('l1', 0.05) ('l1', 0.1) ('l1', 0.2)('l1', 1)
# ('l2', 0.01)('l2', 0.05) ('l2', 0.1) ('l2', 0.2)('l2', 1)
parameters = {
    'penalty':('l1', 'l2'),
    'C':(0.01, 0.05, 0.1, 0.2, 1)
}
#The ‘liblinear’ solver supports both L1 and L2 regularization, with a dual␣
↪formulation only for the L2 penalty.
#For small datasets, ‘liblinear’ is a good choice.
Grid_LR = GridSearchCV(LogisticRegression(solver='liblinear'),parameters, cv=5)
#train model
Grid_LR.fit(X_train, y_train)
```

[37]:
```
GridSearchCV(cv=5, estimator=LogisticRegression(solver='liblinear'),
             param_grid={'C': (0.01, 0.05, 0.1, 0.2, 1),
                         'penalty': ('l1', 'l2')})
```

[38]:
```python
# the best hyperparameter combination
# C = 1/lambda
print_grid_search_metrics(Grid_LR)
```

```
Best score: 0.8121333333333333
Best parameters set:
C:0.05
penalty:l2
```

[39]:
```python
# best model
best_LR_model = Grid_LR.best_estimator_
```

[40]:
```python
best_LR_model.predict(X_test)
```

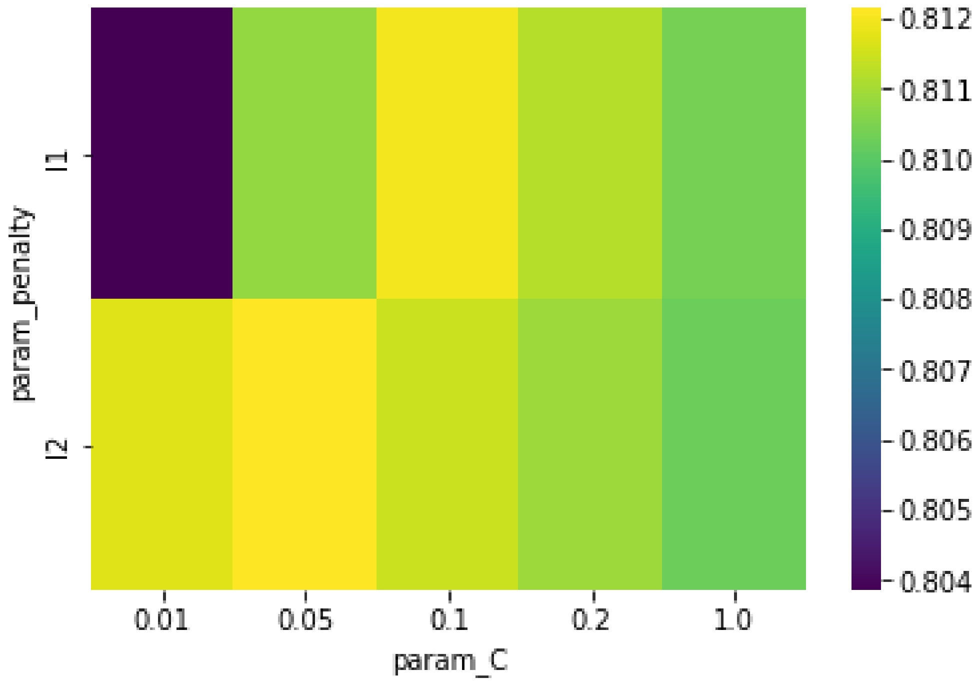
[40]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[41]:
```python
best_LR_model.score(X_test, y_test)
```

[41]:
```
0.8104
```

[42]:
```python
LR_models = pd.DataFrame(Grid_LR.cv_results_)
res = (LR_models.pivot(index='param_penalty', columns='param_C',␣
↪values='mean_test_score')
            )
```

```
_ = sns.heatmap(res, cmap='viridis')
```



Part 3.2.2: Find Optimal Hyperparameters: KNN

[43]:
```
# Possible hyperparamter options for KNN
# Choose k
parameters = {
    'n_neighbors':[5,7,9, 13, 15, 19]
}
Grid_KNN = GridSearchCV(KNeighborsClassifier(),parameters, cv=5)
Grid_KNN.fit(X_train, y_train)
```

[43]:
```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [5, 7, 9, 13, 15, 19]})
```

[44]:
```
# best k
print_grid_search_metrics(Grid_KNN)
```

```
Best score: 0.8402666666666667
Best parameters set:
n_neighbors:15
```

[45]:
```
best_KNN_model = Grid_KNN.best_estimator_
```

Part 3.2.3: Find Optimal Hyperparameters: Random Forest

```
[46]: # Possible hyperparamter options for Random Forest
      # Choose the number of trees
      parameters = {
          'n_estimators' : [60,80,100], #number of decision tree
          'max_depth': [1,5,10]
      }
      Grid_RF = GridSearchCV(RandomForestClassifier(),parameters, cv=5)
      Grid_RF.fit(X_train, y_train)
```

```
[46]: GridSearchCV(cv=5, estimator=RandomForestClassifier(),
                   param_grid={'max_depth': [1, 5, 10],
                               'n_estimators': [60, 80, 100]})
```

```
[47]: # best number of tress
      print_grid_search_metrics(Grid_RF)
```

```
Best score: 0.8618666666666668
Best parameters set:
max_depth:10
n_estimators:80
```

```
[48]: # best random forest
      best_RF_model = Grid_RF.best_estimator_
```

```
[49]: best_RF_model
```

```
[49]: RandomForestClassifier(max_depth=10, n_estimators=80)
```

improve performance: use more complicated models, get rid of some irrelavant features, get more useful features, etc

####Part 3.3: Model Evaluation - Confusion Matrix (Precision, Recall, Accuracy)

class of interest as positive

TP: correctly labeled real churn

Precision(PPV, positive predictive value): tp / (tp + fp); Total number of true predictive churn divided by the total number of predictive churn; High Precision means low fp, not many return users were predicted as churn users.

Recall(sensitivity, hit rate, true positive rate): tp / (tp + fn) Predict most postive or churn user correctly. High recall means low fn, not many churn users were predicted as return users.

```
[50]: from sklearn.metrics import confusion_matrix
      from sklearn.metrics import classification_report
      from sklearn.metrics import precision_score
      from sklearn.metrics import recall_score
```

```python
# calculate accuracy, precision and recall, [[tn, fp],[]]
def cal_evaluation(classifier, cm):
    tn = cm[0][0]
    fp = cm[0][1]
    fn = cm[1][0]
    tp = cm[1][1]
    accuracy  = (tp + tn) / (tp + fp + fn + tn + 0.0)
    precision = tp / (tp + fp + 0.0)
    recall = tp / (tp + fn + 0.0)
    print (classifier)
    print ("Accuracy is: " + str(accuracy))
    print ("precision is: " + str(precision))
    print ("recall is: " + str(recall))
    print ()

# print out confusion matrices
def draw_confusion_matrices(confusion_matricies):
    class_names = ['Not','Churn']
    for cm in confusion_matrices:
        classifier, cm = cm[0], cm[1]
        cal_evaluation(classifier, cm)
```

[51]:
```python
# Confusion matrix, accuracy, precison and recall for random forest and␣
 ↪logistic regression
confusion_matrices = [
    ("Random Forest", confusion_matrix(y_test,best_RF_model.predict(X_test))),
    ("Logistic Regression", confusion_matrix(y_test,best_LR_model.
 ↪predict(X_test))),
    ("K nearest neighbor", confusion_matrix(y_test, best_KNN_model.
 ↪predict(X_test)))
]

draw_confusion_matrices(confusion_matrices)
```

```
Random Forest
Accuracy is: 0.8664
precision is: 0.8458498023715415
recall is: 0.4204322200392927

Logistic Regression
Accuracy is: 0.8104
precision is: 0.6023391812865497
recall is: 0.20235756385068762

K nearest neighbor
Accuracy is: 0.8432
precision is: 0.7695852534562212
```

```
recall is: 0.3280943025540275
```

Random Forest performs the best in terms of all 3 metrics.

What cases use precision and recall? Check if someone actually had a disease.

5.0.3 Part 3.4: Model Evaluation - ROC & AUC

RandomForestClassifier, KNeighborsClassifier and LogisticRegression have predict_prob() function

Part 3.4.1: ROC of RF Model ROC curve: true positive versus false positive

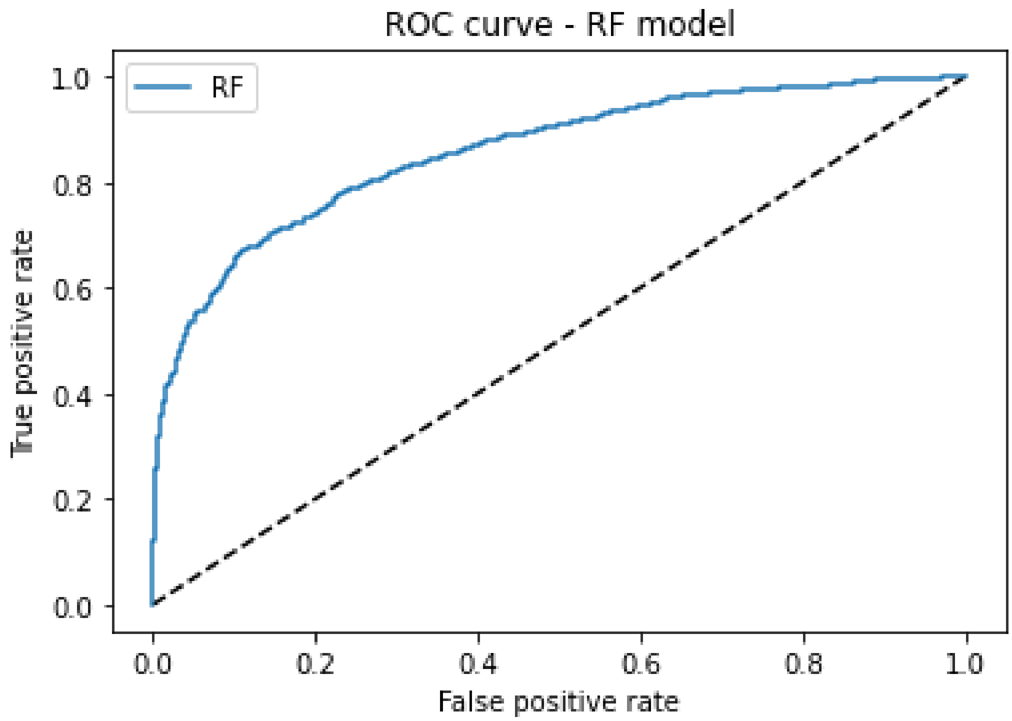
[52]:
```python
from sklearn.metrics import roc_curve
from sklearn import metrics

# Use predict_proba to get the probability results of Random Forest
y_pred_rf = best_RF_model.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_rf)
```

[53]:
```python
best_RF_model.predict_proba(X_test)
```

[53]:
```
array([[0.93668555, 0.06331445],
       [0.93918736, 0.06081264],
       [0.85817542, 0.14182458],
       …,
       [0.89646081, 0.10353919],
       [0.91602177, 0.08397823],
       [0.92983914, 0.07016086]])
```

Second column represents the probability of churning

[54]:
```python
# ROC curve of Random Forest result
import matplotlib.pyplot as plt
plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve - RF model')
plt.legend(loc='best')
plt.show()
```



[55]:
```python
from sklearn import metrics

# AUC score
metrics.auc(fpr_rf,tpr_rf)
```

[55]: 0.8573512041909614

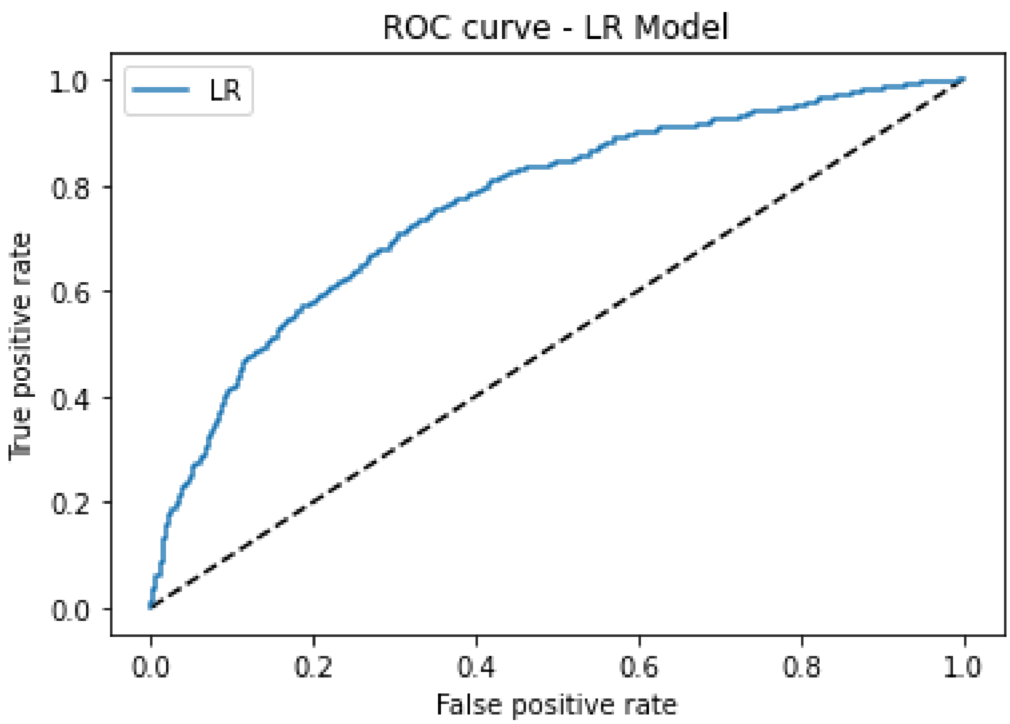
Part 3.4.1: ROC of LR Model

[56]:
```python
# Use predict_proba to get the probability results of Logistic Regression
y_pred_lr = best_LR_model.predict_proba(X_test)[:, 1]
fpr_lr, tpr_lr, thresh = roc_curve(y_test, y_pred_lr)
```

[57]:
```python
best_LR_model.predict_proba(X_test)
```

[57]:
```
array([[0.90951146, 0.09048854],
       [0.82908881, 0.17091119],
       [0.56192019, 0.43807981],
       …,
       [0.78023494, 0.21976506],
       [0.61709981, 0.38290019],
       [0.97126664, 0.02873336]])
```

[64]:
```
# ROC Curve
plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_lr, tpr_lr, label='LR')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve - LR Model')
plt.legend(loc='best')
plt.show()
```



[59]:
```
# AUC score
metrics.auc(fpr_lr,tpr_lr)
```

[59]:
```
0.7625898073748371
```

Part 3.4.1: ROC of KNN Model

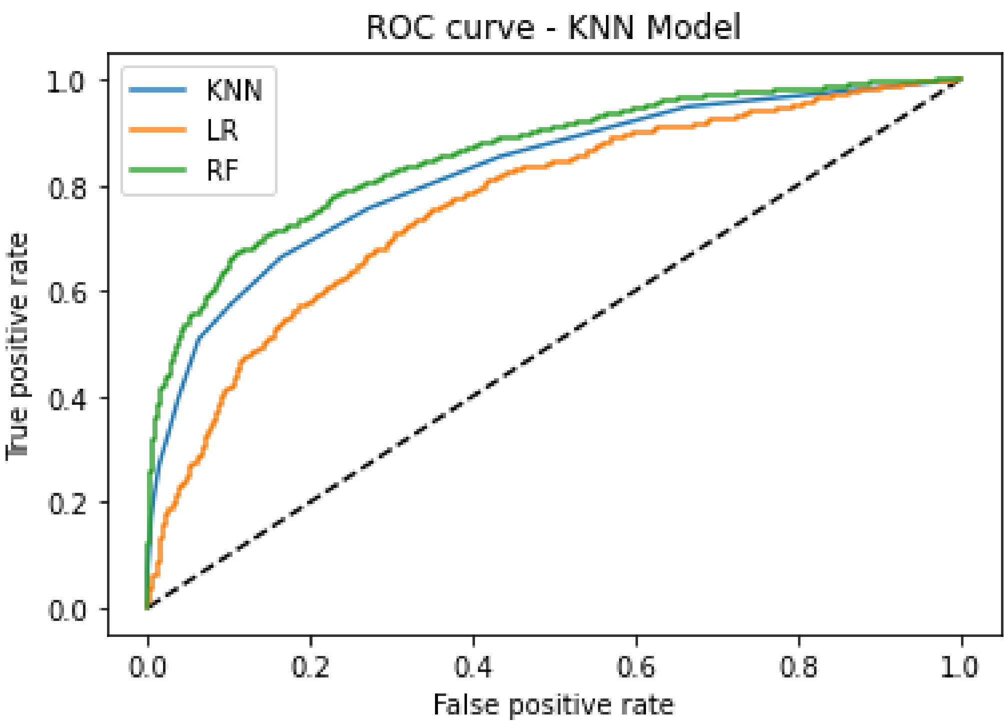
[60]:
```
y_pred_knn = best_KNN_model.predict_proba(X_test)[:, 1]
fpr_knn, tpr_knn, thresh = roc_curve(y_test, y_pred_knn)
best_KNN_model.predict_proba(X_test)
```

[60]:
```
array([[1.        , 0.        ],
       [1.        , 0.        ],
```

```
        [0.8       , 0.2       ],
        …,
        [1.        , 0.        ],
        [0.86666667, 0.13333333],
        [1.        , 0.        ]])
```

[63]:

```
plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_knn, tpr_knn, label='KNN')
plt.plot(fpr_lr, tpr_lr, label='LR')
plt.plot(fpr_rf, tpr_rf, label='RF')
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve - KNN Model')
plt.legend(loc='best')
plt.show()
```



[62]:

```
metrics.auc(fpr_knn,tpr_knn)
```

[62]: 0.8234955137016378

According to the ROC curve and AUC score, RF still works the best.

6 Part 4: Model Extra Functionality

6.0.1 Part 4.1: Logistic Regression Model

The corelated features that we are interested in

[93]:
```
X.head()
```

[93]:
```
   CreditScore Geography  Gender  …  HasCrCard  IsActiveMember
EstimatedSalary
0          619    France  Female  …          1               1
101348.88
1          608     Spain  Female  …          0               1
112542.58
2          502    France  Female  …          1               0
113931.57
3          699    France  Female  …          0               0
93826.63
4          850     Spain  Female  …          1               1
79084.10

[5 rows x 10 columns]
```

[98]:
```
X_with_corr = X.copy()

X_with_corr = OneHotEncoding(X_with_corr, enc_ohe, ['Geography'])
X_with_corr = OneHotEncoding(X_with_corr, enc_ohe1, ['Gender'])
#X_with_corr['Gender'] = enc_ohe1.transform(X_with_corr[['Gender']])
#add a new feature which is highly correlated to an existed feature
X_with_corr['SalaryInRMB'] = X_with_corr['EstimatedSalary'] * 6.4
X_with_corr.head()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

[98]:
```
   CreditScore  Age  Tenure  …  Gender_Female  Gender_Male  SalaryInRMB
0          619   42       2  …            1.0          0.0   648632.832
1          608   41       1  …            1.0          0.0   720272.512
2          502   42       8  …            1.0          0.0   729162.048
3          699   39       1  …            1.0          0.0   600490.432
```

```
4          850   43       2  …             1.0           0.0   506138.240

[5 rows x 14 columns]
```

[99]:
```python
# add L1 regularization to logistic regression
# check the coef for feature selection
scaler = StandardScaler()
X_l1 = scaler.fit_transform(X_with_corr)
LRmodel_l1 = LogisticRegression(penalty="l1", C = 0.04, solver='liblinear')
LRmodel_l1.fit(X_l1, y)

indices = np.argsort(abs(LRmodel_l1.coef_[0]))[::-1]

print ("Logistic Regression (L1) Coefficients")
for ind in range(X_with_corr.shape[1]):
  print ("{0} : {1}".format(X_with_corr.columns[indices[ind]],round(LRmodel_l1.
 ↪coef_[0][indices[ind]], 4)))
```

```
Logistic Regression (L1) Coefficients
Age : 0.7307
IsActiveMember : -0.5046
Geography_Germany : 0.3121
Gender_Female : 0.2173
Balance : 0.1509
CreditScore : -0.0457
NumOfProducts : -0.0439
Tenure : -0.0271
Gender_Male : -0.0236
EstimatedSalary : 0.0069
Geography_France : -0.0043
SalaryInRMB : 0.0023
HasCrCard : -0.0022
Geography_Spain : 0.0
```

L1: not stable, not good for features correlated; Therefore, L2 is used generally

[100]:
```python
# add L2 regularization to logistic regression
# check the coef for feature selection
np.random.seed()
scaler = StandardScaler()
X_l2 = scaler.fit_transform(X_with_corr)
LRmodel_l2 = LogisticRegression(penalty="l2", C = 0.1, solver='liblinear',␣
 ↪random_state=42)
LRmodel_l2.fit(X_l2, y)
LRmodel_l2.coef_[0]

indices = np.argsort(abs(LRmodel_l2.coef_[0]))[::-1]
```

```
print ("Logistic Regression (L2) Coefficients")
for ind in range(X_with_corr.shape[1]):
  print ("{0} : {1}".format(X_with_corr.columns[indices[ind]],round(LRmodel_l2.
↪coef_[0][indices[ind]], 4)))
```

```
Logistic Regression (L2) Coefficients
Age : 0.751
IsActiveMember : -0.5272
Geography_Germany : 0.2279
Balance : 0.162
Gender_Male : -0.13
Gender_Female : 0.13
Geography_France : -0.1207
Geography_Spain : -0.089
CreditScore : -0.0637
NumOfProducts : -0.0586
Tenure : -0.0452
HasCrCard : -0.0199
SalaryInRMB : 0.0137
EstimatedSalary : 0.0137
```

Using L2, coefficients don't change, and the highly correlated features have the same coefficients; Therefore, L2 could take care of highly multicollinearity

6.0.2 Part 4.2: Random Forest Model - Feature Importance Discussion

[102]:
```
X_RF = X.copy()

X_RF = OneHotEncoding(X_RF, enc_ohe, ['Geography'])
X_RF = OneHotEncoding(X_RF, enc_ohe1, ['Gender'])
#X_RF['Gender'] = enc_oe.transform(X_RF[['Gender']])

X_RF.head()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87:
FutureWarning: Function get_feature_names is deprecated; get_feature_names is
deprecated in 1.0 and will be removed in 1.2. Please use get_feature_names_out
instead.
  warnings.warn(msg, category=FutureWarning)
```

[102]:

	CreditScore	Age	Tenure	…	Geography_Spain	Gender_Female	Gender_Male
0	619	42	2	…	0.0	1.0	0.0

1	608	41	1	…	1.0	1.0	0.0
2	502	42	8	…	0.0	1.0	0.0
3	699	39	1	…	0.0	1.0	0.0
4	850	43	2	…	1.0	1.0	0.0

[5 rows x 13 columns]

[103]:
```
# check feature importance of random forest for feature selection
forest = RandomForestClassifier()
forest.fit(X_RF, y)

#The higher, the more important the feature. The importance of a feature is␣
↪computed as the (normalized) total reduction of the criterion brought
#by that feature. It is also known as the Gini importance.
importances = forest.feature_importances_

indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature importance ranking by Random Forest Model:")
for ind in range(X.shape[1]):
  print ("{0} : {1}".format(X_RF.
↪columns[indices[ind]],round(importances[indices[ind]], 4)))
```

```
Feature importance ranking by Random Forest Model:
Age : 0.2365
EstimatedSalary : 0.148
CreditScore : 0.1435
Balance : 0.1414
NumOfProducts : 0.1316
Tenure : 0.0829
IsActiveMember : 0.0411
Geography_Germany : 0.0211
HasCrCard : 0.0181
Geography_France : 0.0103
```