Bank Customer Churn Prediction

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

Contents

Part 0: Setup Google Drive Environment

```
# 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)


id = "1hToFUitLcAVfQ3OFW18blhSUfSVMYCX5"
file = drive.CreateFile({'id':id})
file.GetContentFile('bank_churn')
```

```
import pandas as pd

df = pd.read_csv('bank_churn')
df.head()
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	B
0	1	15634602	Hargrave	619	France	Female	42	2	
1	2	15647311	Hill	608	Spain	Female	41	1	8
2	3	15619304	Onio	502	France	Female	42	8	15
3	4	15701354	Boni	699	France	Female	39	1	

```
df.Geography
```

```
0        France
1         Spain
2        France
3        France
4         Spain
          ...
9995     France
9996     France
9997     France
9998    Germany
9999     France
Name: Geography, Length: 10000, dtype: object
```

```
df.groupby('Geography')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fdf5759ce80>
```

```
df.groupby('Geography').transform('mean')
```

	RowNumber	CustomerId	CreditScore	Age	Tenure	Balance	NumOfP
0	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
1	4950.667743	1.569192e+07	651.333872	38.890997	5.032297	61818.147763	
2	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
3	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
4	4950.667743	1.569192e+07	651.333872	38.890997	5.032297	61818.147763	
...	...	...	...	...	...	...	
9995	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
9996	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
9997	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	
9998	5000.278996	1.569056e+07	651.453567	39.771622	5.009964	119730.116134	
9999	5025.228560	1.569065e+07	649.668329	38.511767	5.004587	62092.636516	

10000 rows × 11 columns

```
df.groupby('Geography').transform('mean').NumOfProducts
```

```
0       1.530913
1       1.539362
2       1.530913
3       1.530913
4       1.539362
          ...
9995    1.530913
9996    1.530913
9997    1.530913
9998    1.519729
9999    1.530913
Name: NumOfProducts, Length: 10000, dtype: float64
```

Part 1: Data Exploration

Part 1.1: Understand the Raw Dataset

```
import pandas as pd
import numpy as np

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

```
churn_df.head()
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	B
0	1	15634602	Hargrave	619	France	Female	42	2	
1	2	15647311	Hill	608	Spain	Female	41	1	8
2	3	15619304	Onio	502	France	Female	42	8	15
3	4	15701354	Boni	699	France	Female	39	1	
4	5	15737888	Mitchell	850	Spain	Female	43	2	12

```
# 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
```

```
# check the unique values for each column
churn_df.nunique()
```

```
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
```

```
# Get target variable
y = churn_df['Exited']
```

```
# check the propotion of y = 1
# python package: imbalance-learn
print(y.sum() / y.shape[0] * 100)
```

```
20.369999999999997
```

Part 1.2: Understand the features

```
# check missing values
churn_df.isnull().sum()
```

```
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
```

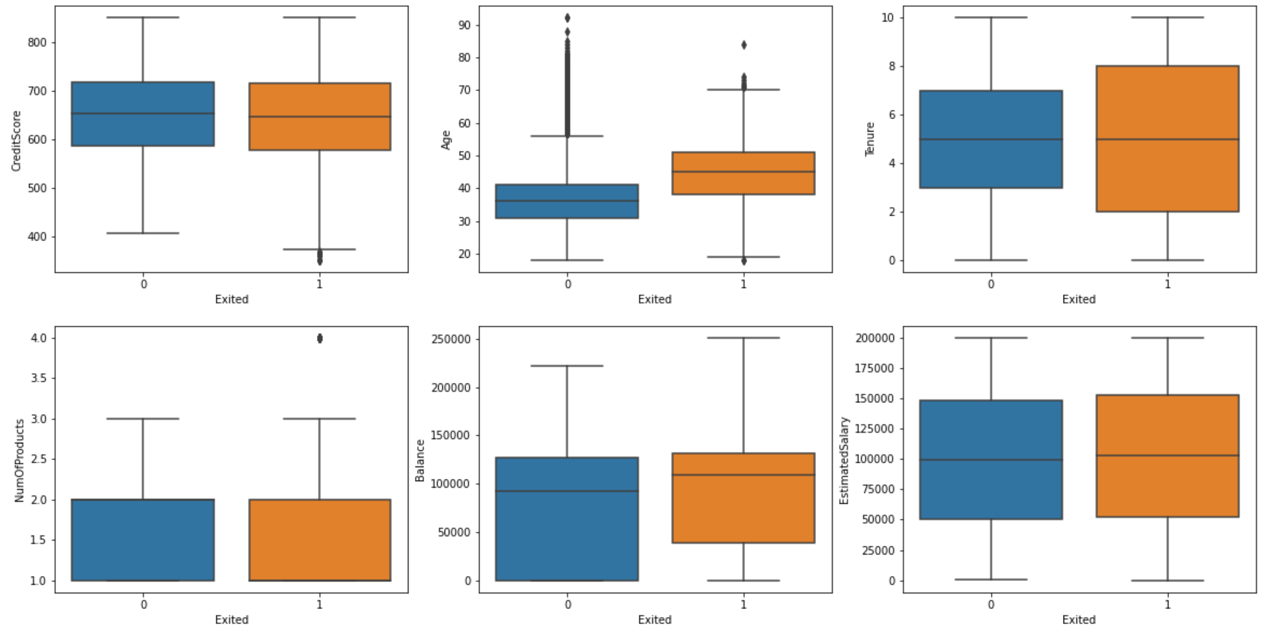
```
# understand Numerical feature
# discrete/continuous
# 'CreditScore', 'Age', 'Tenure', 'NumberOfProducts'
# 'Balance', 'EstimatedSalary'
churn_df[['CreditScore', 'Age', 'Tenure', 'NumOfProducts','Balance', 'EstimatedSalary'
```

	CreditScore	Age	Tenure	NumOfProducts	Balance	Estimate
count	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	1000
mean	650.528800	38.921800	5.012800	1.530200	76485.889288	1000
std	96.653299	10.487806	2.892174	0.581654	62397.405202	575
min	350.000000	18.000000	0.000000	1.000000	0.000000	
25%	584.000000	32.000000	3.000000	1.000000	0.000000	510
50%	652.000000	37.000000	5.000000	1.000000	97198.540000	1001
75%	718.000000	44.000000	7.000000	2.000000	127644.240000	1493
max	850.000000	92.000000	10.000000	4.000000	250898.090000	1999

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

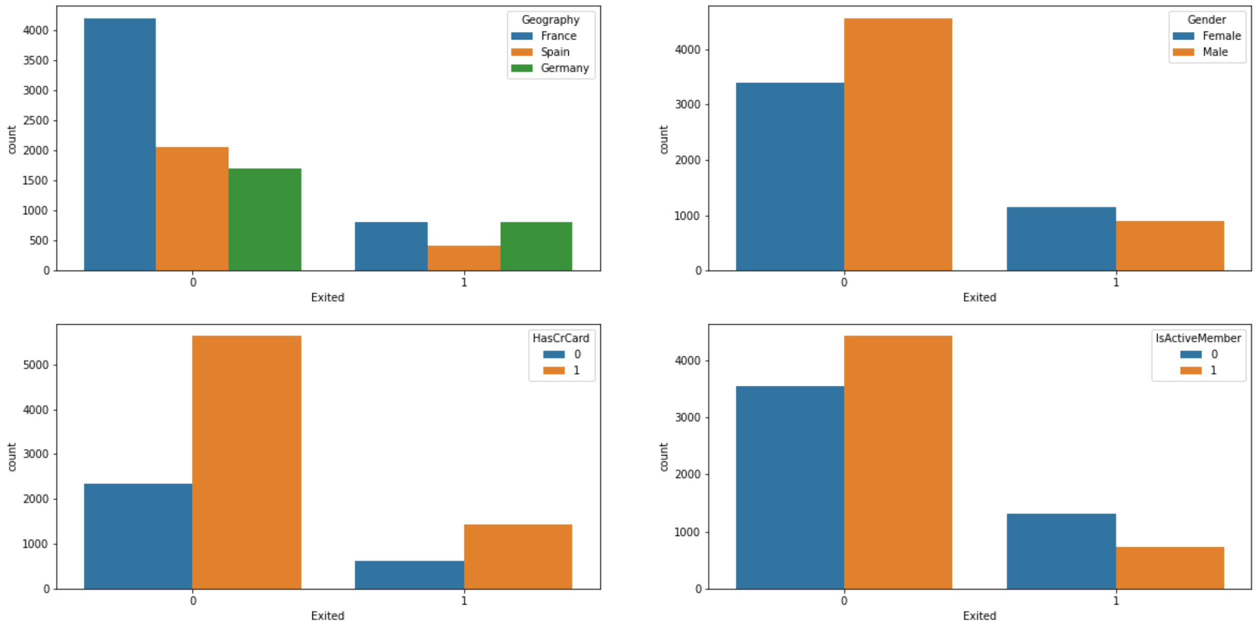
```
# boxplot for numerical feature
_,axss = plt.subplots(2,3, figsize=[20,10])
sns.boxplot(x='Exited', y ='CreditScore', data=churn_df, ax=axss[0][0])
sns.boxplot(x='Exited', y ='Age', data=churn_df, ax=axss[0][1])
sns.boxplot(x='Exited', y ='Tenure', data=churn_df, ax=axss[0][2])
sns.boxplot(x='Exited',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

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fdf5716c320>
```




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

<matplotlib.axes._subplots.AxesSubplot at 0x7fdf57972978>




```
# correlations between features
corr_score = churn_df[['CreditScore', 'Age', 'Tenure', 'NumOfProducts','Balance', 'Est

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

<matplotlib.axes._subplots.AxesSubplot at 0x7fdf5736b780>


```
# check the actual values of correlations
corr_score
```

	CreditScore	Age	Tenure	NumOfProducts	Balance	Estimated
CreditScore	1.000000	-0.003965	0.000842	0.012238	0.006268	-0.
Age	-0.003965	1.000000	-0.009997	-0.030680	0.028308	-0.
Tenure	0.000842	-0.009997	1.000000	0.013444	-0.012254	0.
NumOfProducts	0.012238	-0.030680	0.013444	1.000000	-0.304180	0.
Balance	0.006268	0.028308	-0.012254	-0.304180	1.000000	0.
EstimatedSalary	-0.001384	-0.007201	0.007784	0.014204	0.012797	1.

Part 2: Feature Preprocessing

feature encoding, feature scaling

```
# ordinal encoding
churn_df['Gender'] = churn_df['Gender'] == 'Female'
```

```
churn_df.shape
```

```
(10000, 14)
```

```
# one hot encoding! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
# onehotencoder
churn_df = pd.get_dummies(churn_df, columns=['Geography'])
```

```
churn_df.head(10)
```

	RowNumber	CustomerId	Surname	CreditScore	Gender	Age	Tenure	Balance	Num
0	1	15634602	Hargrave	619	True	42	2	0.00	

```
# Get feature space by dropping useless feature
to_drop = ['RowNumber','CustomerId','Surname','Exited']
X = churn_df.drop(to_drop, axis=1)
```

3	4	15701354	Boni	699	True	39	1	0.00

```
X.head()
```

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
0	619	True	42	2	0.00	1	1	
1	608	True	41	1	83807.86	1	0	
2	502	True	42	8	159660.80	3	1	
3	699	True	39	1	0.00	2	0	
4	850	True	43	2	125510.82	1	1	

Part 3: Model Training and Result Evaluation

Part 3.1: Split dataset

```
# Splite data into training and testing
from sklearn import model_selection

# Reserve 25% for testing
# stratify example:
# 100 -> y: 80 '0', 20 '1' -> 4:1
# 80% training 64: '0', 16:'1' -> 4:1
# 20% testing  16:'0', 4: '1' -> 4:1
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.

print('training data has ' + str(X_train.shape[0]) + ' observation with ' + str(X_trai
print('test data has ' + str(X_test.shape[0]) + ' observation with ' + str(X_test.shap
```

```
training data has 7500 observation with 12 features
test data has 2500 observation with 12 features
```

```
X
```

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActi
0	619	True	42	2	0.00	1	1	
1	608	True	41	1	83807.86	1	0	
2	502	True	42	8	159660.80	3	1	
3	699	True	39	1	0.00	2	0	
4	850	True	43	2	125510.82	1	1	
...	...	...	...	...	...	...	...	
9995	771	False	39	5	0.00	2	1	
9996	516	False	35	10	57369.61	1	1	
9997	709	True	36	7	0.00	1	0	
9998	772	False	42	3	75075.31	2	1	
9999	792	True	28	4	130142.79	1	1	

```
# Scale the data, using standardization
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)  #算min max std
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

```
y
```

```
0       1
1       0
2       1
3       0
4       0
       ..
9995    0
9996    0
9997    1
9998    1
9999    0
Name: Exited, Length: 10000, dtype: int64
```

Part 3.2: Model Training and Selection

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

build models

```
# Logistic Regression
classifier_logistic = LogisticRegression()

# K Nearest Neighbors
classifier_KNN = KNeighborsClassifier()

# Random Forest
classifier_RF = RandomForestClassifier()
```

```
# Train the model
classifier_logistic.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

```
# Prediction of test data
classifier_logistic.predict(X_test)
```

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

```
# Accuracy of test data
classifier_logistic.score(X_test, y_test)
```

```
0.808
```

```
# 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
```

```
[0.81933333 0.80666667 0.80666667 0.80933333 0.82      ]
Model accuracy of Logistic Regression is 0.8124
[0.82533333 0.836      0.814      0.824      0.832     ]
Model accuracy of KNN is 0.8262666666666666
[0.876      0.86       0.85666667 0.85666667 0.862     ]
Model accuracy of Random Forest is 0.8622666666666667
```

Part 3.3: Use Grid Search to Find Optimal Hyperparameters

alternative: random search

```
from sklearn.model_selection import GridSearchCV

# helper function for printing out 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 3.3.1: Find Optimal Hyperparameters - LogisticRegression

```
# Possible hyperparamter options for Logistic Regression Regularization
# Penalty is choosed from L1 or L2
# C is the lambda value(weight) for L1 and L2

# ('l1', 1) ('l1', 5) ('l1', 10) ('l2', 1) ('l2', 5) ('l2', 10)
parameters = {
    'penalty':('l1', 'l2'),
    'C':(0.01, 1, 5, 10)
}
Grid_LR = GridSearchCV(LogisticRegression(solver='liblinear'),parameters, cv=5)
Grid_LR.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='auto',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='liblinear',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': (0.01, 1, 5, 10), 'penalty': ('l1', 'l2')},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
# the best hyperparameter combination
# C = 1/lambda
print_grid_search_metrics(Grid_LR)
```

```
Best score: 0.8124
Best parameters set:
C:1
penalty:l2
```

```
# best model
best_LR_model = Grid_LR.best_estimator_
```

Part 3.3.2: Find Optimal Hyperparameters: KNN

```
# Possible hyperparamter options for KNN
# Choose k
parameters = {
    'n_neighbors':[1,3,5,7,9]
}
Grid_KNN = GridSearchCV(KNeighborsClassifier(),parameters, cv=5)
Grid_KNN.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='deprecated', n_jobs=None,
             param_grid={'n_neighbors': [1, 3, 5, 7, 9]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
# best k
print_grid_search_metrics(Grid_KNN)
```

```
Best score: 0.8322666666666667
Best parameters set:
n_neighbors:9
```

```
best_KNN_model = Grid_KNN.best_estimator_
```

Part 3.3.3: Find Optimal Hyperparameters: Random Forest

```
# Possible hyperparamter options for Random Forest
# Choose the number of trees
parameters = {
    'n_estimators' : [40,60,80]
}
Grid_RF = GridSearchCV(RandomForestClassifier(),parameters, cv=5)
Grid_RF.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
```

```
                                                   min_impurity_decrease=0.0,
                                                   min_impurity_split=None,
                                                   min_samples_leaf=1,
                                                   min_samples_split=2,
                                                   min_weight_fraction_leaf=0.0,
                                                   n_estimators=100, n_jobs=None,
                                                   oob_score=False,
                                                   random_state=None, verbose=0,
                                                   warm_start=False),
                 iid='deprecated', n_jobs=None,
                 param_grid={'n_estimators': [40, 60, 80]}, pre_dispatch='2*n_jobs',
                 refit=True, return_train_score=False, scoring=None, verbose=0)
```

```
# best number of tress
print_grid_search_metrics(Grid_RF)
```

```
Best score: 0.8641333333333334
Best parameters set:
n_estimators:60
```

```
# best random forest
best_RF_model = Grid_RF.best_estimator_
```

```
best_RF_model
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=60,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Part 3.4: 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.

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import precision_score
```

```
from sklearn.metrics import recall_score

# 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)


# Confusion matrix, accuracy, precison and recall for random forest and logistic regre
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: 0.8592
precision is: 0.775438596491228
recall is: 0.43418467583497056

Logistic Regression
Accuracy is: 0.808
precision is: 0.5857988165680473
recall is: 0.1944990176817289

K nearest neighbor
Accuracy is: 0.8336
precision is: 0.6837944664031621
recall is: 0.33988212180746563
```

Part 3.4: Model Evaluation - ROC & AUC

RandomForestClassifier, KNeighborsClassifier and LogisticRegression have predict_prob() function

Part 3.4.1: ROC of RF Model

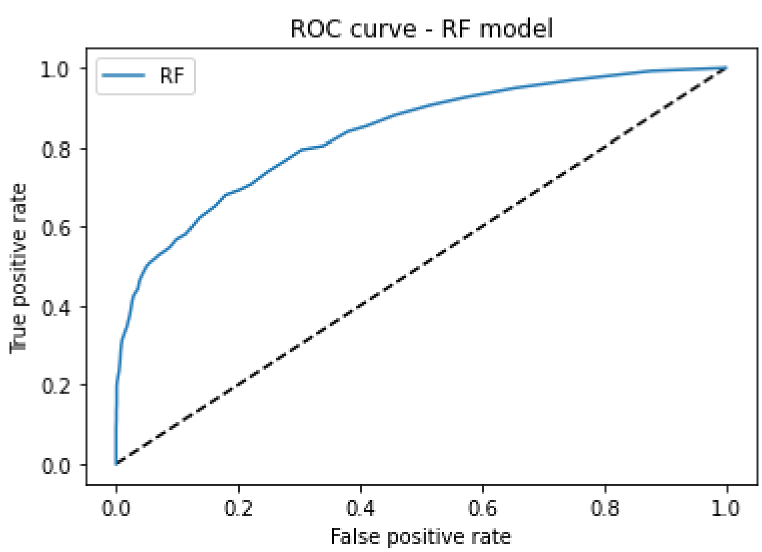
```
from sklearn.metrics import roc_curve
from sklearn import metrics

# 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, thresh = roc_curve(y_test, y_pred_rf)
```

```
best_RF_model.predict_proba(X_test)
```

```
array([[0.76666667, 0.23333333],
       [1.        , 0.        ],
       [0.7       , 0.3       ],
       ...,
       [0.9       , 0.1       ],
       [0.98333333, 0.01666667],
       [0.96666667, 0.03333333]])
```

```
# ROC curve of Random Forest result
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()
```




```
from sklearn import metrics
```

```
from sklearn import metrics

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

```
0.8339995599056264
```

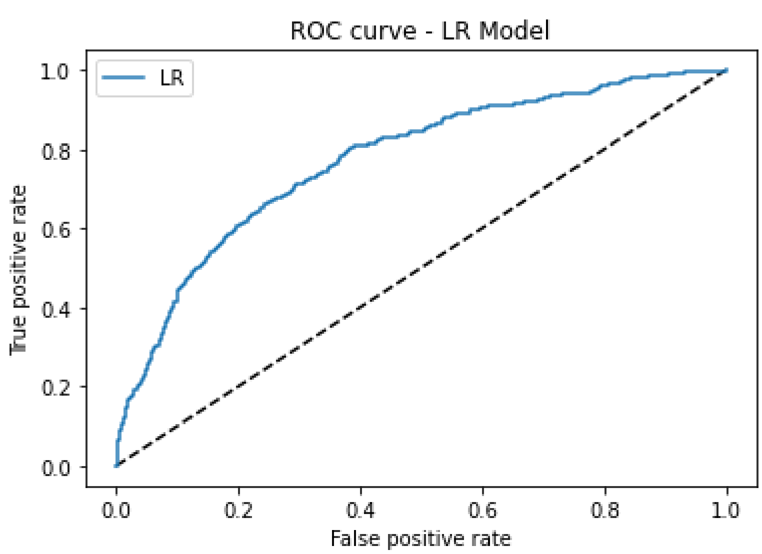
Part 3.4.1: ROC of LR Model

```
# 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)
```

```
best_LR_model.predict_proba(X_test)
```

```
array([[0.82440541, 0.17559459],
       [0.93201435, 0.06798565],
       [0.85485771, 0.14514229],
       ...,
       [0.71400625, 0.28599375],
       [0.89297649, 0.10702351],
       [0.85539214, 0.14460786]])
```

```
# 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()
```




```
# AUC score
```

```
# AUC score
metrics.auc(fpr_lr,tpr_lr)
```

    0.7722314264879581

Part 4: Feature Selection

Part 4.1: Logistic Regression Model - Feature Selection Discussion

The corelated features that we are interested in

```
X_with_corr = X.copy()
X_with_corr['SalaryInRMB'] = X['EstimatedSalary'] * 6.91
X_with_corr.head()
```

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
0	619	True	42	2	0.00	1	1	
1	608	True	41	1	83807.86	1	0	
2	502	True	42	8	159660.80	3	1	
3	699	True	39	1	0.00	2	0	
4	850	True	43	2	125510.82	1	1	

```
# 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 = 0.07, 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_[(
```

```
Logistic Regression (L1) Coefficients
Age : 0.744
IsActiveMember : -0.5184
Geography_Germany : 0.3156
Gender : 0.2503
Balance : 0.1567
CreditScore : -0.0537
```

```
NumOfProducts : -0.0503
Tenure : -0.0351
SalaryInRMB : 0.0165
Geography_France : -0.0099
HasCrCard : -0.0099
EstimatedSalary : 0.0005
Geography_Spain : 0.0
```

```
# 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_stat
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_[(
```

```
Logistic Regression (L2) Coefficients
Age : 0.751
IsActiveMember : -0.5272
Gender : 0.2591
Geography_Germany : 0.2279
Balance : 0.162
Geography_France : -0.1207
Geography_Spain : -0.089
CreditScore : -0.0637
NumOfProducts : -0.0586
Tenure : -0.0452
HasCrCard : -0.0199
EstimatedSalary : 0.0137
SalaryInRMB : 0.0137
```

Part 4.2: Random Forest Model - Feature Importance Discussion

```
# check feature importance of random forest for feature selection
forest = RandomForestClassifier()
forest.fit(X, y)

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.columns[indices[ind]],round(importances[indices[ind]], 4
```

```
Feature importance ranking by Random Forest Model:
Age : 0.2391
EstimatedSalary : 0.1444
Balance : 0.1422
CreditScore : 0.142
NumOfProducts : 0.1339
Tenure : 0.0821
IsActiveMember : 0.039
Geography_Germany : 0.0227
HasCrCard : 0.0183
Gender : 0.0181
Geography_France : 0.0095
Geography_Spain : 0.0088
```