Part 1: Data Exploration

Part 1.1: Understand the Raw Dataset

In [45]:

```
import pandas as pd
import numpy as np

churn_df = pd.read_csv('bank_data.csv')
churn_df.head()
```

Out[45]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balanc
0	1	15634602	Hargrave	619	France	Female	42	2	0.0
1	2	15647311	Hill	608	Spain	Female	41	1	83807.8
2	3	15619304	Onio	502	France	Female	42	8	159660.8
3	4	15701354	Boni	699	France	Female	39	1	0.0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.8

In [46]:

```
# 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
```

In [47]:

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

Out[47]:

```
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
```

Part 1.2: Understand the Features

In [48]:

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

Out[48]:

```
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
```

Part 1.2.1: Understand the Numerical Features

In [49]:

```
churn_df[['CreditScore', 'Age', 'Tenure', 'NumOfProducts','Balance', 'EstimatedSalary']].describe()
```

Out[49]:

	CreditScore	Age	Tenure	NumOfProducts	Balance	EstimatedSala
count	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.0000
mean	650.528800	38.921800	5.012800	1.530200	76485.889288	100090.2398
std	96.653299	10.487806	2.892174	0.581654	62397.405202	57510.4928
min	350.000000	18.000000	0.000000	1.000000	0.000000	11.5800
25%	584.000000	32.000000	3.000000	1.000000	0.000000	51002.1100
50%	652.000000	37.000000	5.000000	1.000000	97198.540000	100193.9150
75%	718.000000	44.000000	7.000000	2.000000	127644.240000	149388.2475
max	850.000000	92.000000	10.000000	4.000000	250898.090000	199992.4800

In [50]:

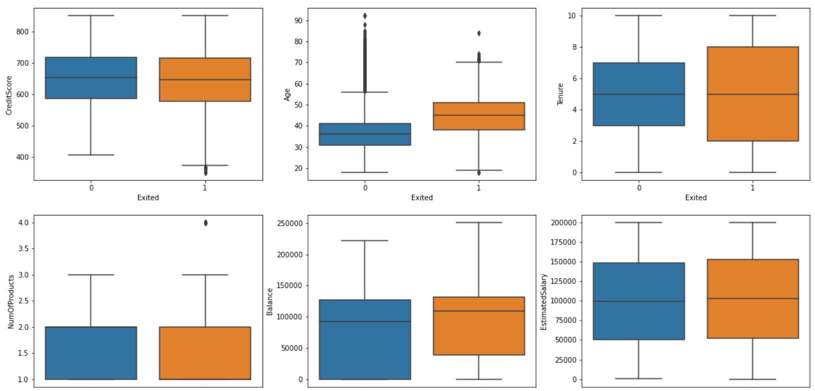
```
# check the features' distribution
# pandas.DataFrame.describe()
# boxplot, distplot, countplot
# the column 'Exited' is our target

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

In [51]:

```
# 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])
```

Out[51]:

<AxesSubplot:xlabel='Exited', ylabel='EstimatedSalary'>



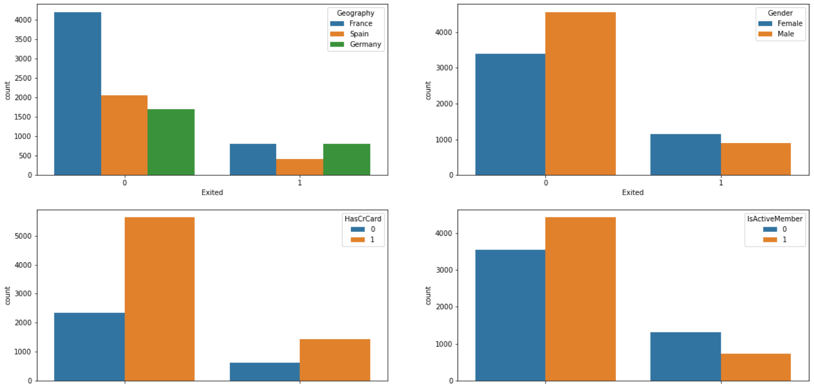
Part 1.2.2: Understand the Categorical Features

In [52]:

```
# countplot is a good choice here
_,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])
```

Out[52]:

```
<AxesSubplot:xlabel='Exited', ylabel='count'>
```



Part 2: Feature Preprocessing

Part 2.1: Extract Features

In [53]:

```
# Get feature space by dropping useless feature
# Obviously, these do not have a logical connection with churn
feature_drop = ['RowNumber','CustomerId','Surname','Exited']
X = churn_df.drop(feature_drop, axis=1)
X.head()
```

Out[53]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsAc
0	619	France	Female	42	2	0.00	1	1	
1	608	Spain	Female	41	1	83807.86	1	0	
2	502	France	Female	42	8	159660.80	3	1	
3	699	France	Female	39	1	0.00	2	0	
4	850	Spain	Female	43	2	125510.82	1	1	

In [54]:

```
# There are two types of our features: categorical / numerical
X.dtypes
```

Out[54]:

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

In [55]:

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

In [56]:

```
cat_cols
```

Out[56]:

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

In [57]:

```
num_cols
```

Out[57]:

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

In [58]:

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

In [59]:

```
# Splite data into training(75%) and testing(25%) using model_selection function in sklearn

from sklearn import model_selection

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=0.25, stratify =
#stratified sampling

print('training data has '+ str(X_train.shape[0])+ ' observation with ' + str(X_train.shape[1]) + '
print('test data has '+ str(X_test.shape[0]) + ' observation with ' + str(X_test.shape[1]) + ' featu
# Show the size of both training of test data
```

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

In [60]:

```
# stratified sampling is used here to prevent extreme cases

X_train.head()
```

Out[60]:

	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	Is
7971	633	Spain	Male	42	10	0.00	1	0	
9152	708	Germany	Female	23	4	71433.08	1	1	
6732	548	France	Female	37	9	0.00	2	0	
902	645	France	Female	48	7	90612.34	1	1	
2996	729	Spain	Female	45	7	91091.06	2	1	

Part 2.2: Encoding for Categorical Data

In [61]:

```
# One hot encoding
# Transform the categorical data of 'Geography' to numerical

from sklearn.preprocessing import OneHotEncoder

def OneHotEncoding(df, enc, categories):
  transformed = pd.DataFrame(enc.transform(df[categories]).toarray(), columns=enc.get_feature_names(
  return pd.concat([df.reset_index(drop=True), transformed], axis=1).drop(categories, axis=1)
# step1: Define a function to transform the result of OneHotCoding to dataframe
# step2: Substitute old column with new column

categories_1 = ['Geography']
enc_ohe = OneHotEncoder()
enc_ohe.fit(X_train[categories_1])
# Set up the One hot encoding size for 'Geography'

X_train = OneHotEncoding(X_train, enc_ohe, categories_1)
X_test = OneHotEncoding(X_test, enc_ohe, categories_1)
# Apply function to specific columns in both train and testing dataset
```

In [62]:

```
X_train.head()
```

Out[62]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember
0	633	Male	42	10	0.00	1	0	1
1	708	Female	23	4	71433.08	1	1	0
2	548	Female	37	9	0.00	2	0	0
3	645	Female	48	7	90612.34	1	1	1
4	729	Female	45	7	91091.06	2	1	0

In [63]:

```
# Ordinal encoding
from sklearn.preprocessing import OrdinalEncoder

categories = ['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])
# Encoding 'Gender'
```

In [64]:

```
X_train.head()
```

Out[64]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember
0	633	1.0	42	10	0.00	1	0	1
1	708	0.0	23	4	71433.08	1	1	0
2	548	0.0	37	9	0.00	2	0	0
3	645	0.0	48	7	90612.34	1	1	1
4	729	0.0	45	7	91091.06	2	1	0

Part 2.3: Standardize/Normalize Data

In [65]:

```
# Scale the data, using standardization
# Advantages:
# 1. speed up gradient descent
# 2. same scale
# 3. algorithm requirments

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])
```

In [66]:

```
X_train.head()
```

Out[66]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveM
0	-0.172985	1.0	0.289202	1.731199	-1.218916	-0.912769	-1.542199	0.9
1	0.602407	0.0	-1.509319	-0.341156	-0.076977	-0.912769	0.648425	-1.0
2	-1.051762	0.0	-0.184093	1.385806	-1.218916	0.796109	-1.542199	-1.0
3	-0.048922	0.0	0.857156	0.695022	0.229625	-0.912769	0.648425	0.9
4	0.819517	0.0	0.573179	0.695022	0.237278	0.796109	0.648425	-1.0

Part 3: Model Training and Result Evaluation

Part 3.1: Model Training

In [67]:

```
# 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()
```

In [92]:

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

Out[92]:

LogisticRegression()

In [93]:

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

Out[93]:

array([0, 0, 0, ..., 0, 0, 0], dtype=int64)

In [94]:

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

Out[94]:

0.8088

In [71]:

```
# 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
[0.84133333 0.84066667 0.83       0.83066667 0.84      ]
Model accuracy of KNN is 0.8365333333333334
[0.87666667 0.86266667 0.85466667 0.85866667 0.86266667]
Model accuracy of Random Forest is 0.8630666666666666
```

Part 3.2: Use Grid Search to Find Optimal Hyperparameters

In [72]:

```
# Loss/cost function --> (wx + b - y) ^2 + λ * |w| --> λ is a hyperparameter
```

In [95]:

```
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 3.2.1: Find Optimal Hyperparameters - LogisticRegression-lambda

In [96]:

```
# 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)
}
Grid_LR = GridSearchCV(LogisticRegression(solver='liblinear'),parameters, cv=5)
Grid_LR.fit(X_train, y_train)
```

Out[96]:

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

In [105]:

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

```
Best score: 0.8125333333333333
Best parameters set:
C:1
penalty:l1
```

In [106]:

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

Out[106]:

```
LogisticRegression(C=1, penalty='l1', solver='liblinear')
```

In [107]:

```
best_LR_model.predict(X_test)
```

Out[107]:

```
array([0, 0, 0, ..., 0, 0, 0], dtype=int64)
```

In [108]:

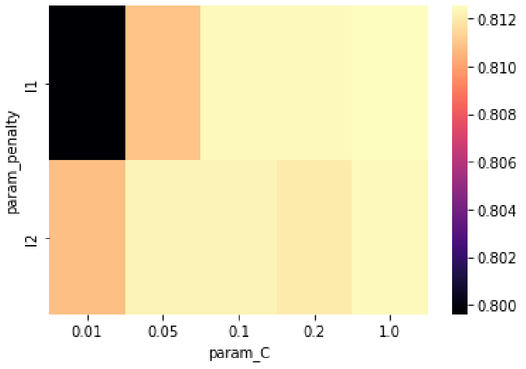
```
best_LR_model.score(X_test, y_test)
```

Out[108]:

```
0.8092
```

In [109]:

```
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='magma')
```



Part 3.2.2: Find Optimal Hyperparameters: KNN-C

In [110]:

```
# 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)
```

Out[110]:

```
GridSearchCV(cv=5, estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [1, 3, 5, 7, 9]})
```

In [111]:

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

```
Best score: 0.8433333333333334
Best parameters set:
n_neighbors:9
```

In [112]:

```
best_KNN_model = Grid_KNN.best_estimator_
```

Part 3.2.3: Find Optimal Hyperparameters: Random Forest-depth

In [117]:

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

Out[117]:

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

In [114]:

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

```
Best score: 0.8664000000000002
Best parameters set:
max_depth:10
n_estimators:80
```

In [115]:

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

Out[115]:

```
RandomForestClassifier(max_depth=10, n_estimators=80)
```

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

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.

In [118]:

```
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)
```

In [119]:

```
# 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: 0.8604
precision is: 0.8076923076923077
recall is: 0.4125736738703340

Logistic Regression
Accuracy is: 0.8092
precision is: 0.5963855421686747
recall is: 0.1944990176817289

K nearest neighbor
Accuracy is: 0.8428
precision is: 0.7283464566929134
recall is: 0.36345776031434185
```

Part 3.4: Model Evaluation - ROC & AUC

Part 3.4.1: ROC of RF Model

In [120]:

```
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)
```

In [121]:

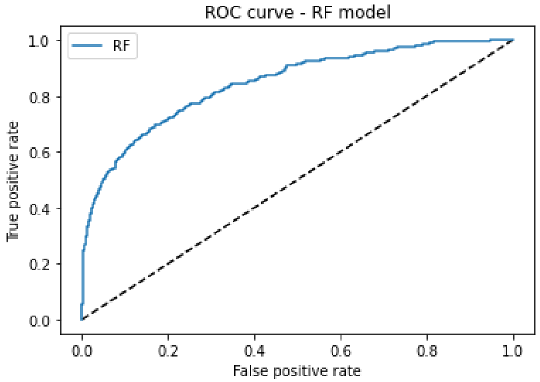
```
best_RF_model.predict_proba(X_test)
```

Out[121]:

```
array([[0.75138239, 0.24861761],
       [0.92837177, 0.07162823],
       [0.73237493, 0.26762507],
       ...,
       [0.84666041, 0.15333959],
       [0.92533831, 0.07466169],
       [0.90282888, 0.09717112]])
```

In [122]:

```
# 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()
```



In [123]:

```
from sklearn import metrics

# AUC score
# The probability that a randomly-chosen positive example is ranked more highly than a randomly-chos

metrics.auc(fpr_rf,tpr_rf)
```

Out[123]:

```
0.8459640089637159
```

Part 3.4.2: ROC of Logistic Regression Model

In [124]:

```
# 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)
```

In [125]:

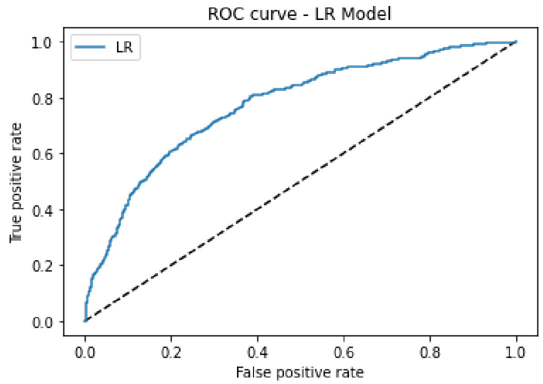
```
best_LR_model.predict_proba(X_test)
```

Out[125]:

```
array([[0.82435829, 0.17564171],
       [0.9317178 , 0.0682822 ],
       [0.85520934, 0.14479066],
       ...,
       [0.71449535, 0.28550465],
       [0.89278331, 0.10721669],
       [0.85561097, 0.14438903]])
```

In [126]:

```
# 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()
```



In [127]:

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

Out[127]:

```
0.7722008369687169
```

Part 3.4.3: ROC of KNN Model

In [128]:

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

In [129]:

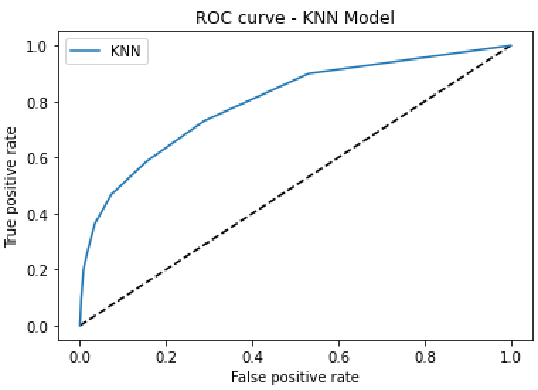
```
best_KNN_model.predict_proba(X_test)
```

Out[129]:

```
array([[1.        , 0.        ],
       [1.        , 0.        ],
       [0.88888889, 0.11111111],
       ...,
       [0.77777778, 0.22222222],
       [1.        , 0.        ],
       [1.        , 0.        ]])
```

In [130]:

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



In [131]:

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

Out[131]:

0.7986385690420251

Part 4: Model Extra Functionality

Part 4.1: Logistic Regression Model

In [132]:

```
X_with_corr = X.copy()

X_with_corr = OneHotEncoding(X_with_corr, enc_ohe, ['Geography'])
X_with_corr['Gender'] = enc_oe.transform(X_with_corr[['Gender']])
X_with_corr.head()
```

Out[132]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember
0	619	0.0	42	2	0.00	1	1	1
1	608	0.0	41	1	83807.86	1	0	1
2	502	0.0	42	8	159660.80	3	1	0
3	699	0.0	39	1	0.00	2	0	0
4	850	0.0	43	2	125510.82	1	1	1

In [133]:

```
# 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]
```

```
Logistic Regression (L1) Coefficients
Age : 0.7307
IsActiveMember : -0.5046
Geography_Germany : 0.3121
Gender : -0.2409
Balance : 0.1509
CreditScore : -0.0457
NumOfProducts : -0.0439
Tenure : -0.0271
EstimatedSalary : 0.0092
Geography_France : -0.0042
HasCrCard : -0.0022
Geography_Spain : 0.0
```

In [134]:

```
# Add L2 regularization to logistic regression
# Check the coef for feature selection

np.random.seed()
scaler = StandardScaler()
X_12 = scaler.fit_transform(X_with_corr)
LRmodel_12 = LogisticRegression(penalty="12", C = 0.1, solver='liblinear', random_state=42)
LRmodel_12.fit(X_12, y)
LRmodel_12.coef_[0]

indices = np.argsort(abs(LRmodel_12.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_12.coef_[0][indices[ir
```

```
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
EstimatedSalary : 0.0272
HasCrCard : -0.0199
```

Part 4.2: Random Forest Model - Feature Importance Discussion

In [135]:

```
X_RF = X.copy()

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

X_RF.head()
```

Out[135]:

	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember
0	619	0.0	42	2	0.00	1	1	1
1	608	0.0	41	1	83807.86	1	0	1
2	502	0.0	42	8	159660.80	3	1	0
3	699	0.0	39	1	0.00	2	0	0
4	850	0.0	43	2	125510.82	1	1	1

In [136]:

```
# Check feature importance of random forest for feature selection
forest = RandomForestClassifier()
forest.fit(X_RF, 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_RF.columns[indices[ind]],round(importances[indices[ind]], 4)))
```

```
Feature importance ranking by Random Forest Model:
Age : 0.2395
EstimatedSalary : 0.1469
CreditScore : 0.1449
Balance : 0.1403
NumOfProducts : 0.1271
Tenure : 0.0827
IsActiveMember : 0.0423
Geography_Germany : 0.0206
Gender : 0.0185
HasCrCard : 0.0181
```