Telecom Churn Case Study

Presented By –

Ritvik Yash

Content

- Problem Statement
- Data Preparation
- EDA
- Training Data
- Building Model
- Model Evaluation
- Recommendations

Problem Statement

- To reduce customer churn, telecom companies need to predict which customers are at high risk of churn. In this project, we will analyse customer-level data of a leading telecom firm, build predictive models to identify customers at high risk of churn and identify the main indicators of churn.
- Retaining high profitable customers is the main business goal here.

Data Preparation

- Importing the Libraries, loading the data in data frame. Checking on the shape and datatypes
- Checking on the Null & Select values and replacing it with Not known.
- Processing null values accordingly & dropping irrelevant columns.

Handling missing values

Handling missing values in columns

In [8]:
```
# Cheking percent of missing values in columns
df_missing_columns = (round(((df.isnull().sum()/len(df.index))*100),2).to_frame('null')).sort_values('null', ascending=False)
df_missing_columns
```

last_day_rch_amt_9	0.00
last_day_rch_amt_8	0.00
last_day_rch_amt_7	0.00
last_day_rch_amt_6	0.00
max_rech_amt_9	0.00
max_rech_amt_8	0.00
max_rech_amt_7	0.00
max_rech_amt_6	0.00
total_rech_amt_9	0.00
total_rech_amt_8	0.00
sep_vbc_3g	0.00

226 rows × 1 columns

In [9]:
```
# List the columns having more than 30% missing values
col_list_missing_30 = list(df_missing_columns.index[df_missing_columns['null'] > 30])
```

In [10]:
```
# Delete the columns having more than 30% missing values
df = df.drop(col_list_missing_30, axis=1)
```

In [11]:
```
df.shape
```

Out[11]: (99999, 186)

Deleting the date columns as the date columns are not required in our analysis

In [12]:
```
# List the date columns
date_cols = [k for k in df.columns.to_list() if 'date' in k]
print(date_cols)
```

```
['last_date_of_month_6', 'last_date_of_month_7', 'last_date_of_month_8', 'last_date_of_month_9', 'date_of_last_rech_6', 'dat
e_of_last_rech_7', 'date_of_last_rech_8', 'date_of_last_rech_9']
```

In [13]:
```
# Dropping date columns
df = df.drop(date_cols, axis=1)
```

Dropping circle_id column as this column has only one unique value. Hence there will be no impact of this column on the data analysis.

In [14]:
```
# Drop circle_id column
df = df.drop('circle_id', axis=1)
```

Handling missing values in rows

In [20]:
```
# Count the rows having more than 50% missing values
df_missing_rows_50 = df[(df.isnull().sum(axis=1)) > (len(df.columns)//2)]
df_missing_rows_50.shape
```

Out[20]: (114, 178)

In [21]:
```
# Deleting the rows having more than 50% missing values
df = df.drop(df_missing_rows_50.index)
df.shape
```

Out[21]: (29897, 178)

In [22]:
```
# Checking the missing values in columns again
df_missing_columns = (round(((df.isnull().sum()/len(df.index))*100),2).to_frame('null')).sort_values('null', ascending=False)
df_missing_columns
```

Out[22]:

	null
loc_ic_mou_9	5.32
og_others_9	5.32
loc_og_t2t_mou_9	5.32
loc_ic_t2t_mou_9	5.32
loc_og_t2m_mou_9	5.32
loc_og_t2f_mou_9	5.32
loc_og_t2c_mou_9	5.32
std_ic_t2m_mou_9	5.32
loc_og_mou_9	5.32
std_og_t2t_mou_9	5.32
roam_og_mou_9	5.32
std_ic_t2o_mou_9	5.32

Looks like MOU for all the types of calls for the month of September (9) have missing values together for any particular record.

Lets check the records for the MOU for Sep(9), in which these coulmns have missing values together.

Tag churners

Now tag the churned customers (churn=1, else 0) based on the fourth month as follows: Those who have not made any calls (either incoming or outgoing) AND have not used mobile internet even once in the churn phase.

In [42]:
```
df['churn'] = np.where((df['total_ic_mou_9']==0) & (df['total_og_mou_9']==0) & (df['vol_2g_mb_9']==0) & (df['vol_3g_mb_9']==0
```

In [43]:
```
df.head()
```

Out[43]:

	mobile_number	loc_og_t2o_mou	std_og_t2o_mou	loc_ic_t2o_mou	arpu_6	arpu_7	arpu_8	arpu_9	onnet_mou_6	onnet_mou_7	onnet_mou_8	onr
8	7001524846	0.0	0.0	0.0	378.721	492.223	137.362	166.787	413.69	351.03	35.08	
13	7002191713	0.0	0.0	0.0	492.846	205.671	593.260	322.732	501.76	108.39	534.24	
16	7000875565	0.0	0.0	0.0	430.975	299.869	187.894	206.490	50.51	74.01	70.61	
17	7000187447	0.0	0.0	0.0	690.008	18.980	25.499	257.583	1185.91	9.28	7.79	
21	7002124215	0.0	0.0	0.0	514.453	597.753	637.760	578.596	102.41	132.11	85.14	

Deleting all the attributes corresponding to the churn phase

In [44]:
```
# List the columns for churn month(9)
col_9 = [col for col in df.columns.to_list() if '_9' in col]
print(col_9)
```

```
['arpu_9', 'onnet_mou_9', 'offnet_mou_9', 'roam_ic_mou_9', 'roam_og_mou_9', 'loc_og_t2t_mou_9', 'loc_og_t2m_mou_9', 'loc_og_
t2f_mou_9', 'loc_og_t2c_mou_9', 'loc_og_mou_9', 'std_og_t2t_mou_9', 'std_og_t2m_mou_9', 'std_og_t2f_mou_9', 'std_og_t2c_mou_
9', 'std_og_mou_9', 'isd_og_mou_9', 'spl_og_mou_9', 'og_others_9', 'total_og_mou_9', 'loc_ic_t2t_mou_9', 'loc_ic_t2m_mou_9',
'loc_ic_t2f_mou_9', 'loc_ic_mou_9', 'std_ic_t2t_mou_9', 'std_ic_t2m_mou_9', 'std_ic_t2f_mou_9', 'std_ic_t2o_mou_9', 'std_ic_
mou_9', 'total_ic_mou_9', 'spl_ic_mou_9', 'isd_ic_mou_9', 'ic_others_9', 'total_rech_num_9', 'total_rech_amt_9', 'max_rech_a
mt_9', 'last_day_rch_amt_9', 'vol_2g_mb_9', 'vol_3g_mb_9', 'monthly_2g_9', 'sachet_2g_9', 'monthly_3g_9', 'sachet_3g_9']
```

In [45]:
```
# Deleting the churn month columns
df = df.drop(col_9, axis=1)
```

In [46]:
```
# Dropping sep_vbc_3g column
df = df.drop('sep_vbc_3g', axis=1)
```

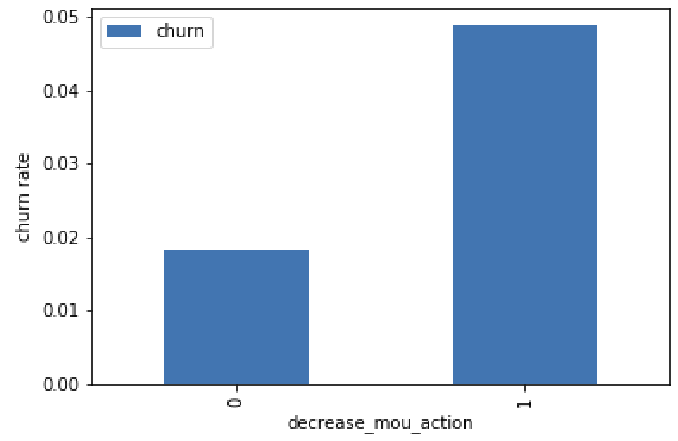
Checking churn percentage

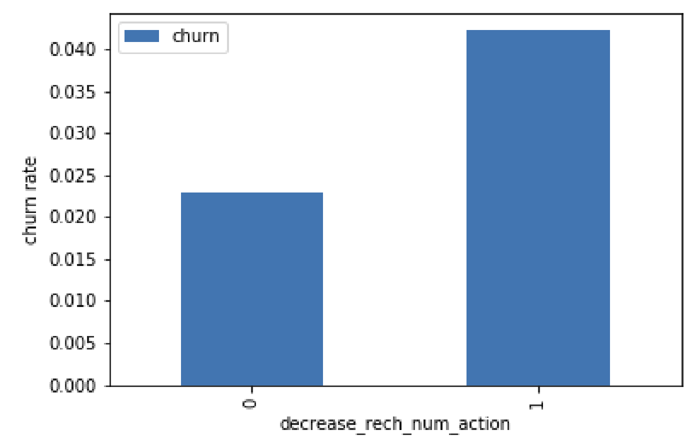
In [47]:
```
round(100*(df['churn'].mean()),2)
```

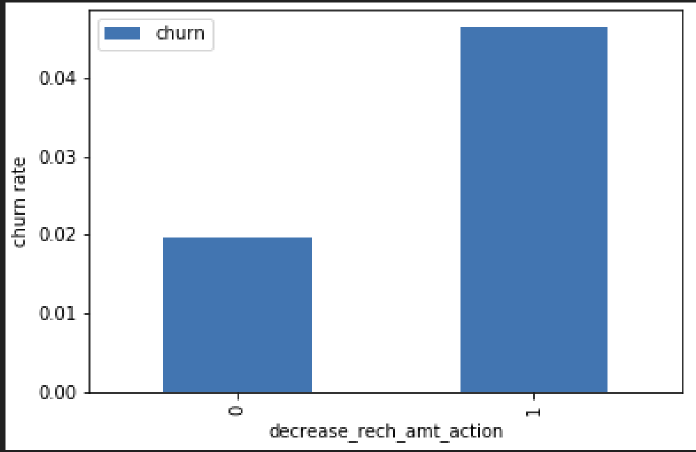
Out[47]: 3.39

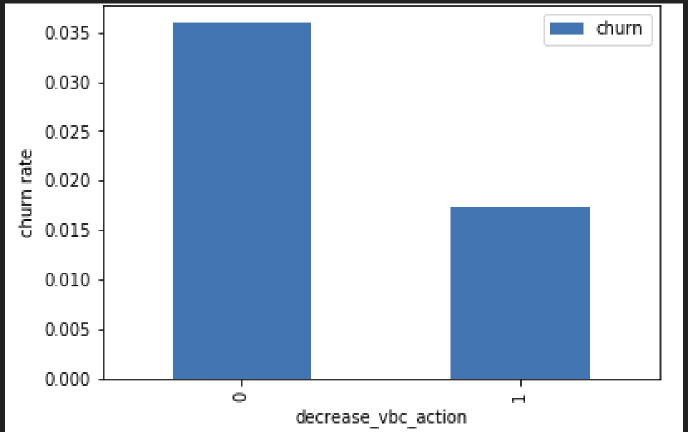
There is very little percentage of churn rate. We will take care of the class imbalance later.

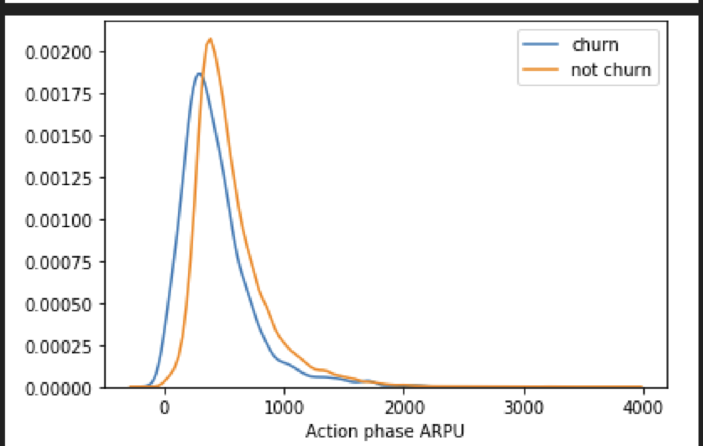
Exploratory Data Analysis(EDA)

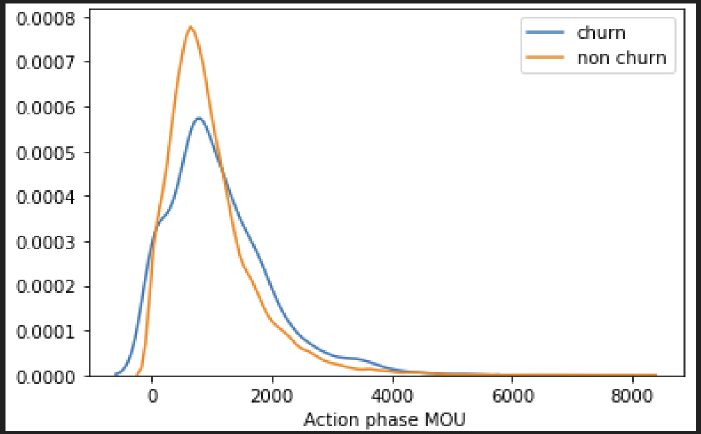


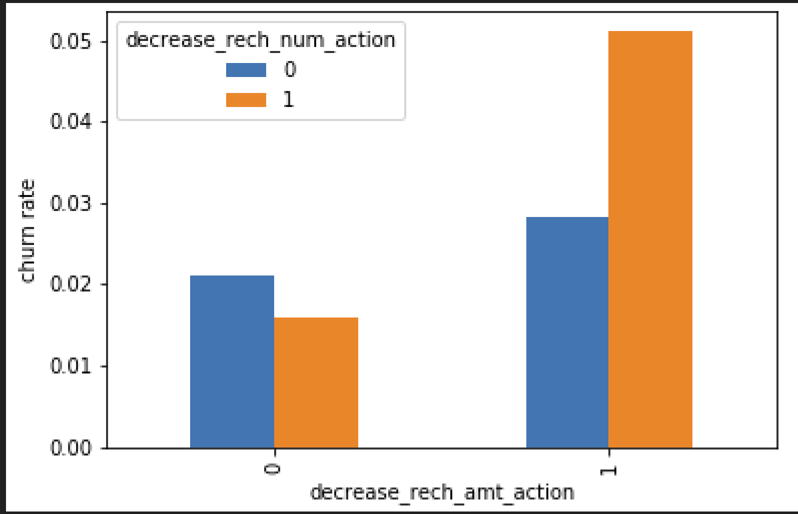




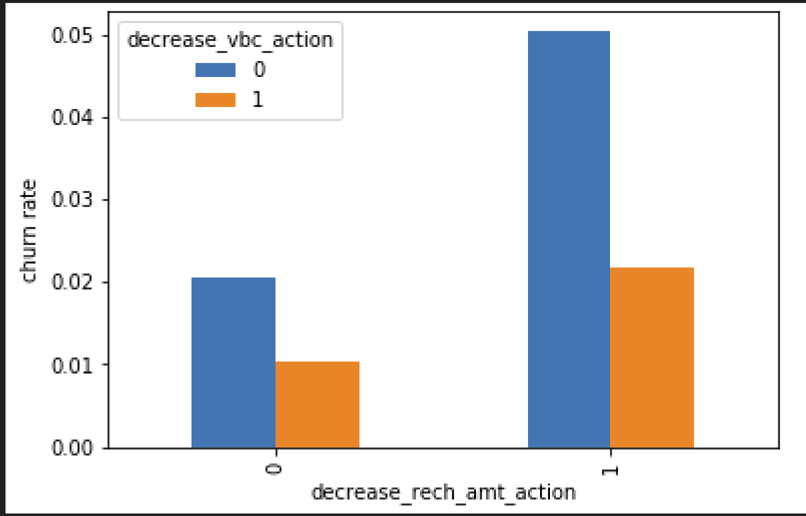




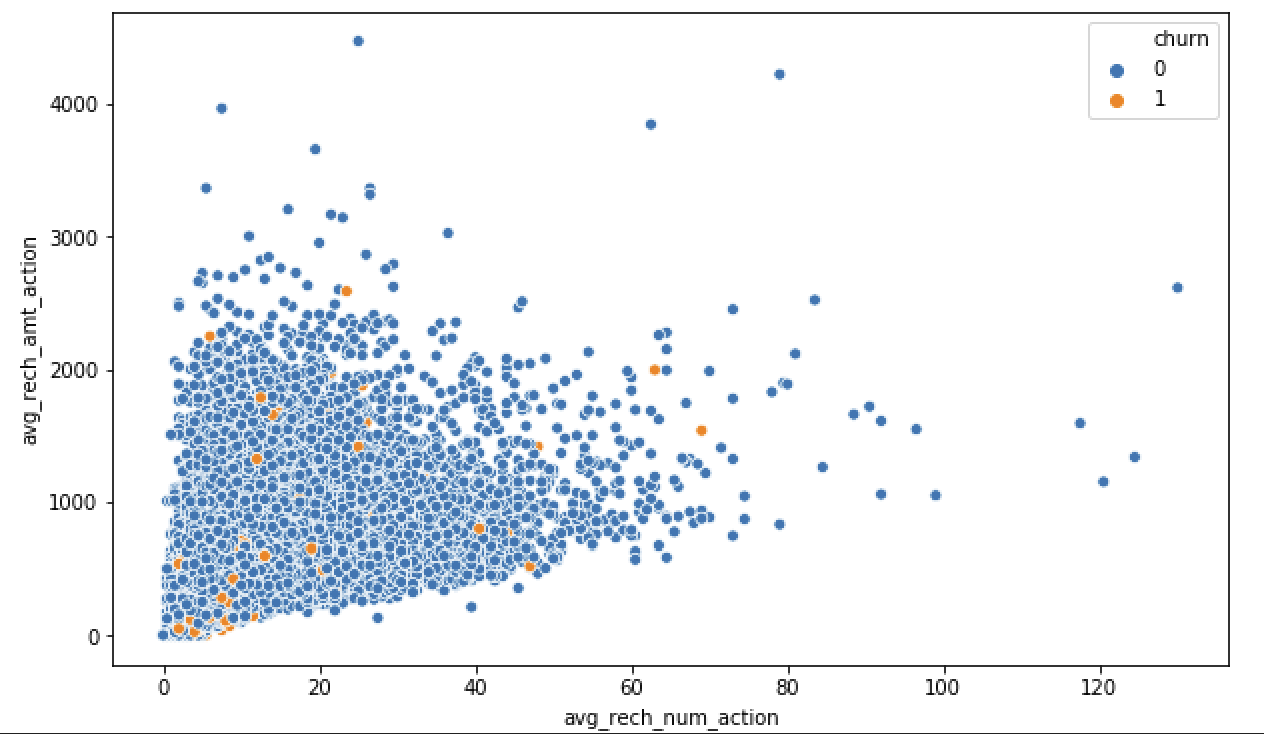




decrease_rech_num_action
0
1
churn rate
0.00
0.01
0.02
0.03
0.04
0.05
0
1
decrease_rech_amt_action



decrease_vbc_action
0
1
churn rate
0.00
0.01
0.02
0.03
0.04
0.05
0
1
decrease_rech_amt_action



churn
0
1
avg_rech_amt_action
0
1000
2000
3000
4000
0
20
40
60
80
100
120
avg_rech_num_action

Training Data

- Splitting the dataset for training & testing, fitting the training data

Train-Test Split

In [87]:
```
# Import library
from sklearn.model_selection import train_test_split
```

In [88]:
```
# Putting feature variables into X
X = data.drop(['mobile_number','churn'], axis=1)
```

In [89]:
```
# Putting target variable to y
y = data['churn']
```

In [90]:
```
# Splitting data into train and test set 80:20
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size=0.2, random_state=100)
```

Dealing with data imbalance

We are creating synthetic samples by doing upsampling using SMOTE(Synthetic Minority Oversampling Technique).

In [91]:
```
# Imporing SMOTE
from imblearn.over_sampling import SMOTE
```

In [92]:
```
# Instantiate SMOTE
sm = SMOTE(random_state=27)
```

In [93]:
```
# Fittign SMOTE to the train set
X_train, y_train = sm.fit_sample(X_train, y_train)
```

Feature Scaling

In [94]:
```
# Standardization method
from sklearn.preprocessing import StandardScaler
```

In [95]:
```
# Instantiate the Scaler
scaler = StandardScaler()
```

In [97]:
```
# List of the numeric columns
cols_scale = X_train.columns.to_list()
# Removing the derived binary columns
cols_scale.remove('decrease_mou_action')
cols_scale.remove('decrease_rech_num_action')
cols_scale.remove('decrease_rech_amt_action')
cols_scale.remove('decrease_arpu_action')
cols_scale.remove('decrease_vbc_action')
```

In [98]:
```
# Fit the data into scaler and transform
X_train[cols_scale] = scaler.fit_transform(X_train[cols_scale])
```

In [99]:
```
X_train.head()
```

Out[99]:

	loc_og_t2o_mou	std_og_t2o_mou	loc_ic_t2o_mou	arpu_6	arpu_7	arpu_8	onnet_mou_6	onnet_mou_7	onnet_mou_8	offnet_mou_6	offnet_mo
0	0.0	0.0	0.0	0.140777	-0.522792	-0.276289	0.106540	-0.662084	-0.465777	-0.211202	-0.636
1	0.0	0.0	0.0	-1.427243	4.428047	3.254270	-0.658491	-0.236590	-0.004450	-0.776075	2.523
2	0.0	0.0	0.0	-0.222751	0.543206	0.809117	-0.601239	-0.599206	-0.331043	-0.363395	-0.495
3	0.0	0.0	0.0	-0.911173	0.842273	0.731302	-0.702232	-0.650471	-0.458464	-0.789784	-0.654
4	0.0	0.0	0.0	0.271356	0.247684	1.256421	-0.356392	-0.180394	0.114727	0.899204	0.904

Scaling the test set

We don't fit scaler on the test set. We only transform the test set.

In [100]:
```
# Transform the test set
X_test[cols_scale] = scaler.transform(X_test[cols_scale])
X_test.head()
```

Out[100]:

	loc_og_t2o_mou	std_og_t2o_mou	loc_ic_t2o_mou	arpu_6	arpu_7	arpu_8	onnet_mou_6	onnet_mou_7	onnet_mou_8	offnet_mou_6	offnet_
5704	0.0	0.0	0.0	0.244310	-0.268832	1.005890	-0.725286	-0.690223	-0.476634	0.483540	0
64892	0.0	0.0	0.0	0.048359	-0.779609	-0.157969	-0.734066	-0.698072	-0.502219	-0.358555	-0
39613	0.0	0.0	0.0	0.545470	0.184388	1.403349	-0.537110	-0.521615	-0.206890	0.694901	0
93118	0.0	0.0	0.0	0.641508	0.816632	-0.211023	-0.058843	0.029897	-0.155872	-0.148197	-0
81235	0.0	0.0	0.0	3.878627	0.911619	2.745295	4.117829	1.452446	2.809582	-0.002634	-0

Building Model

Model with PCA

In [101]:
```
#Import PCA
from sklearn.decomposition import PCA
```

In [102]:
```
# Instantiate PCA
pca = PCA(random_state=42)
```

In [103]:
```
# Fit train set on PCA
pca.fit(X_train)
```

Out[103]:
```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=42,
    svd_solver='auto', tol=0.0, whiten=False)
```

In [104]:
```
# Principal components
pca.components_
```

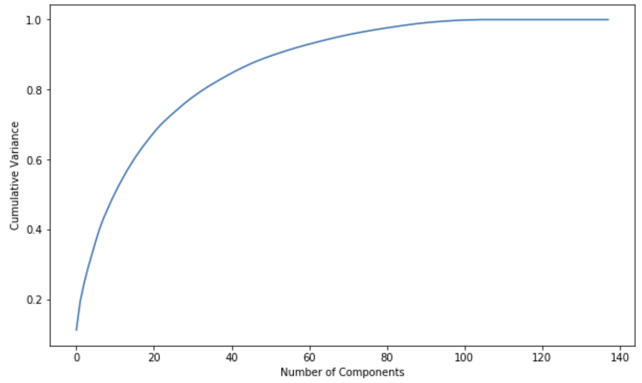
Out[104]:
```
array([[-7.50315936e-20,  4.16333634e-17,  1.11022302e-16, ...,
        -2.59799614e-02, -2.57740516e-02,  1.40032998e-02],
       [-1.61507486e-19, -5.55111512e-17,  0.00000000e+00, ...,
        -1.16737642e-02, -9.94022864e-03, -1.42598315e-02],
       [ 1.91332162e-19, -2.77555756e-17,  0.00000000e+00, ...,
        -4.18532955e-02, -4.28357226e-02,  2.46812846e-02],
       ...,
       [-0.00000000e+00, -3.78694731e-02, -3.56427844e-02, ...,
         1.23056947e-16, -4.06575815e-17, -0.00000000e+00],
       [ 0.00000000e+00,  2.32804774e-01,  3.95374959e-02, ...,
         6.41847686e-17,  3.12250226e-17,  8.32667268e-17],
       [ 9.99999199e-01, -3.85782335e-04,  1.19512948e-03, ...,
         1.35525272e-20,  3.11708125e-19, -1.99086624e-17]])
```

In [105]:
```
# Cumuliative varinace of the PCs
variance_cumu = np.cumsum(pca.explained_variance_ratio_)
print(variance_cumu)
```

```
[0.11213256 0.19426234 0.24575583 0.28953571 0.32841891 0.36623473
 0.40173361 0.43144425 0.45702167 0.48194328 0.50480575 0.52673812
 0.54724457 0.5670202  0.58530008 0.60304258 0.6190213  0.63473458
 0.64927873 0.66341423 0.67712828 0.69025011 0.7020618  0.71278516
 0.72309435 0.73290234 0.74255604 0.75209676 0.76151565 0.77010093
 0.77861315 0.7866115  0.79429496 0.80173555 0.80878909 0.81538157
 0.82193734 0.8283476  0.83472622 0.84089758 0.84687761 0.85280024
 0.85840083 0.86374029 0.86901646 0.87418749 0.87891437 0.88341796
 0.887723   0.89186057 0.89588256 0.89966074 0.90339384 0.90704071
 0.91060084 0.91411689 0.91752343 0.92076319 0.92395413 0.92705111
 0.93001239 0.93296077 0.93580029 0.93862291 0.94138851 0.9441162
 0.94678675 0.94937767 0.95188405 0.95433786 0.95665036 0.95893735
 0.96116409 0.96323063 0.96526039 0.967203   0.96912626 0.97100138
 0.97284931 0.9746657  0.97639261 0.97806622 0.97972617 0.98133794
 0.98290963 0.98446566 0.98601222 0.98753485 0.98877905 0.98998795
 0.99114751 0.99224606 0.99321228 0.99407803 0.9949224  0.99573799
 0.99652652 0.99717502 0.99776401 0.99831985 0.99880793 0.99912289
 0.99942656 0.99969174 0.99985313 0.99994737 0.99998103 0.99999839
 0.99999963 0.99999989 1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.        ]
```

In [106]:
```
# Plotting scree plot
fig = plt.figure(figsize = (10,6))
plt.plot(variance_cumu)
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Variance')
```

Out[106]:
```
Text(0, 0.5, 'Cumulative Variance')
```



We can see that `60 components` explain amost more than 90% variance of the data. So, we will perform PCA with 60 components.

Performing PCA with 60 components

In [107]:
```
# Importing incremental PCA
from sklearn.decomposition import IncrementalPCA
```

In [108]:
```
# Instantiate PCA with 60 components
pca_final = IncrementalPCA(n_components=60)
```

In [109]:
```
# Fit and transform the X_train
X_train_pca = pca_final.fit_transform(X_train)
```

Applying transformation on the test set

We are only doing Transform in the test set not the Fit-Transform. Because the Fitting is already done on the train set. So, we just have to do the transformation with the already fitted data on the train set.

In [110]:
```
X_test_pca = pca_final.transform(X_test)
```

Emphasize Sensitivity/Recall than Accuracy

We are more focused on higher Sensitivity/Recall score than the accuracy.

Beacuse we need to care more about churn cases than the not churn cases. The main goal is to reatin the customers, who have the possiblity to churn. There should not be a problem, if we consider few not churn customers as churn customers and provide them some incentives for retaining them. Hence, the sensitivity score is more important here.

Logistic regression with PCA

In [111]:
```
# Importing scikit logistic regression module
from sklearn.linear_model import LogisticRegression
```

In [112]:
```
# Impoting metrics
from sklearn import metrics
from sklearn.metrics import confusion_matrix
```

Tuning hyperparameter C

C is the the inverse of regularization strength in Logistic Regression. Higher values of C correspond to less regularization.

In [113]:
```
# Importing libraries for cross validation
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

In [114]:
```
# Creating KFold object with 5 splits
folds = KFold(n_splits=5, shuffle=True, random_state=4)

# Specify params
params = {"C": [0.01, 0.1, 1, 10, 100, 1000]}

# Specifing score as recall as we are more focused on acheiving the higher sensitivity than the accuracy
model_cv = GridSearchCV(estimator = LogisticRegression(),
                        param_grid = params,
                        scoring= 'recall',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# Fit the model
model_cv.fit(X_train_pca, y_train)
```

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

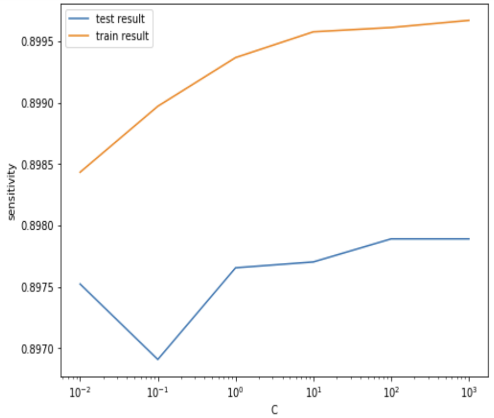
```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  30 out of  30 | elapsed:   21.6s finished
```

Out[114]:
```
GridSearchCV(cv=KFold(n_splits=5, random_state=4, shuffle=True),
             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='lbfgs',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='recall', verbose=1)
```

In [115]:
```
# results of grid search CV
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[115]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	params	split0_test_score	split1_test_score	split2_test_score	split3_test_score
0	0.478627	0.060932	0.007600	1.200167e-03	0.01	{'C': 0.01}	0.900071	0.897759	0.895814	0.906425
1	0.731842	0.021868	0.006801	3.999949e-04	0.1	{'C': 0.1}	0.898177	0.896359	0.894651	0.905959
2	0.743043	0.008100	0.007000	6.325605e-04	1	{'C': 1}	0.898650	0.898693	0.895581	0.905028



[119]:
```
# Best score with best C
best_score = model_cv.best_score_
best_C = model_cv.best_params_['C']

print(" The highest test sensitivity is {0} at C = {1}".format(best_score, best_C))
```

```
 The highest test sensitivity is 0.8978916608693863 at C = 100
```

Logistic regression with optimal C

[120]:
```
# Instantiate the model with best C
logistic_pca = LogisticRegression(C=best_C)
```

[121]:
```
# Fit the model on the train set
log_pca_model = logistic_pca.fit(X_train_pca, y_train)
```

Prediction on the train set

[122]:
```
# Predictions on the train set
y_train_pred = log_pca_model.predict(X_train_pca)
```

[123]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_train, y_train_pred)
print(confusion)
```

```
[[17908  3517]
 [ 2154 19271]]
```

[124]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

[125]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_train, y_train_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.8676546091015169
Sensitivity:- 0.899463243873979
Specificity:- 0.8358459743290548
```

Prediction on the test set

In [126]:
```
# Prediction on the test set
y_test_pred = log_pca_model.predict(X_test_pca)
```

In [127]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test, y_test_pred)
print(confusion)
```

```
[[4452  896]
 [  36  157]]
```

In [128]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [129]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test, y_test_pred))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.8317993142032124
Sensitivity:- 0.8134715025906736
Specificity:- 0.8324607329842932
```

Model summary

- Train set
  - Accuracy = 0.86
  - Sensitivity = 0.89
  - Specificity = 0.83
- Test set
  - Accuracy = 0.83
  - Sensitivity = 0.81
  - Specificity = 0.83

Overall, the model is performing well in the test set, what it had learnt from the train set.

Model Evaluation

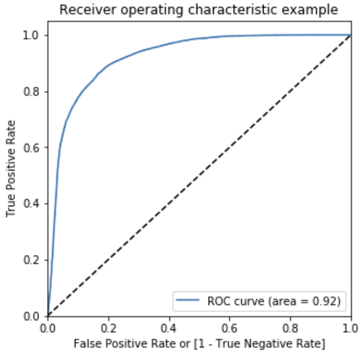
Plotting the ROC Curve (Trade off between sensitivity & specificity)

In [194]:
```
# ROC Curve function

def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve( actual, probs,
                                              drop_intermediate = False )
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(5, 5))
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return None
```

In [195]:
```
draw_roc(y_train_pred_final['churn'], y_train_pred_final['churn_prob'])
```



We can see the area of the ROC curve is closer to 1, whic is the Gini of the model.

Metrics

In [214]:
```
# Confusion matrix
confusion = metrics.confusion_matrix(y_test_pred_final['churn'], y_test_pred_final['test_predicted'])
print(confusion)
```

```
[[4190 1158]
 [  34  159]]
```

In [215]:
```
TP = confusion[1,1] # true positive
TN = confusion[0,0] # true negatives
FP = confusion[0,1] # false positives
FN = confusion[1,0] # false negatives
```

In [216]:
```
# Accuracy
print("Accuracy:-",metrics.accuracy_score(y_test_pred_final['churn'], y_test_pred_final['test_predicted']))

# Sensitivity
print("Sensitivity:-",TP / float(TP+FN))

# Specificity
print("Specificity:-", TN / float(TN+FP))
```

```
Accuracy:- 0.7848763761053962
Sensitivity:- 0.8238341968911918
Specificity:- 0.7834704562453254
```

Recommendations

1.Target the customers, whose minutes of usage of the incoming local calls and outgoing ISD calls are less in the action phase (mostly in the month of August).

2.Target the customers, whose outgoing others charge in July and incoming others on August are less.

3.Also, the customers having value based cost in the action phase increased are more likely to churn than the other customers. Hence, these customers may be a good target to provide offer.

4.Cutomers, whose monthly 3G recharge in August is more, are likely to be churned.

5.Customers having decreasing STD incoming minutes of usage for operators T to fixed lines of T for the month of August are more likely to churn.

6.Cutomers decreasing monthly 2g usage for August are most probable to churn.

7.Customers having decreasing incoming minutes of usage for operators T to fixed lines of T for August are more likely to churn.

8.roam_og_mou_8 variables have positive coefficients (0.7135). That means for the customers, whose roaming outgoing minutes of usage is increasing are more likely to churn.

Thank You!