# Data Science Exercise

This excercise is about retention versus churn (departure) of customers from HubPay. All retail service providers experience churn, and there is competitive advantage to be gained from predicting the customers that will depart and taking steps to retain them

The Objectives of the excercise are to:

```
1.Perform basic descriptive analytics to understand whether any of the f
eatures in the data are associated with churn

2.Prepare and cleanse the data to make it suitable for modelling

3.Build classification models to predict churn, using at least two alter
native machine learning techniques, and perform appropriate validation u
pon these models

4.Evaluate the performance of the models and any shortcomings that are e
vident, and opine as to whether the models are adequate for decision-mak
ing

5.Build logistic regression model(s) explaining churn in terms of the ex
planatory variables and provide interpretations of coefficients and coef
ficient standard errors in these model(s)
```

# Importing Relevant Libaries

In this section, we import all relevant libraries needed to meet our objective

In [103]:
```
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
import seaborn as sns
import statsmodels
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_repo


%matplotlib inline
```

In [13]:
```
# I don't have xgboost installed so I installed here
# pip install xgboost
from xgboost import XGBClassifier
```

# Exploratory Data Analysis

Reading Dataset

In [14]:
```
df = pd.read_csv("Customer_Churn_Data_v2.csv")
df.sample(5)
```

Out[14]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_len |
|---|---|---|---|---|---|
| **4248** | 4249 | 25729.941010 | 42238.321450 | 9.0 | |
| **2305** | 2306 | 4648.660816 | 6522.113365 | 2.0 | |
| **6787** | 6788 | 12722.774890 | 24059.185130 | 6.0 | |
| **1937** | 1938 | 9811.613190 | 6477.897880 | 3.0 | |
| **163** | 164 | 34658.848340 | 54179.568990 | 13.0 | |

In [31]:
```
df.shape
```

Out[31]:
```
(7432, 14)
```

Checking Data Type

In [15]:
```
df.dtypes
```

Out[15]:
```
cust_id                                 int64
income                                float64
debt_with_other_lenders               float64
credit_score                          float64
has_previous_defaults_other_lenders     int64
num_remittances_prev_12_mth             int64
remittance_amt_prev_12_mth            float64
main_remittance_corridor               object
opened_campaign_1                       int64
opened_campaign_2                       int64
opened_campaign_3                       int64
opened_campaign_4                       int64
tenure_years                          float64
churned                                 int64
dtype: object
```

Income, debt_with_other_lenders, credit_score are meant to be float or int data type

In [16]:
```
#Lets check and see what the issue is
df1 = df[pd.to_numeric(df.income,errors='coerce').isnull()]
df1.head()
```

Out[16]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lenders |
|---|---|---|---|---|---|
| **59** | 60 | NaN | 43289.045190 | 9.0 | 0 |
| **94** | 95 | NaN | 24642.825310 | 7.0 | 1 |
| **212** | 213 | NaN | 1632.956718 | 2.0 | 0 |
| **232** | 233 | NaN | 16213.468180 | 4.0 | 0 |
| **235** | 236 | NaN | 55011.606990 | 8.0 | 1 |

In [17]:
```
df2 = df[pd.to_numeric(df.debt_with_other_lenders,errors='coerce').isnull()]
df2.head()
```

Out[17]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lend |
|---|---|---|---|---|---|
| **15** | 16 | 12664.869050 | NaN | 5.0 | |
| **22** | 23 | 13734.955730 | NaN | 7.0 | |
| **40** | 41 | 6677.353290 | NaN | 4.0 | |
| **109** | 110 | 6174.622364 | NaN | 3.0 | |
| **130** | 131 | 8900.689168 | NaN | 4.0 | |

In [18]:
```
df3 = df[pd.to_numeric(df.credit_score,errors='coerce').isnull()]
df3.head()
```

Out[18]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_le |
|---|---|---|---|---|---|
| **26** | 27 | 17050.88032 | 13796.82406 | NaN | |
| **44** | 45 | 29062.36911 | 44209.97239 | NaN | |
| **50** | 51 | 17098.72002 | 19315.82411 | NaN | |
| **89** | 90 | 15567.36742 | 26949.65022 | NaN | |
| **101** | 102 | 16093.97296 | 48352.61055 | NaN | |

the column seem to contain some white spaces, lets see how many

In [21]:
```
df1.shape, df2.shape, df3.shape
```

Out[21]:
```
((233, 14), (295, 14), (295, 14))
```

Lets convert whitespaces to null values

In [23]:
```
new_df = df.replace(r'^\s*$', np.NaN, regex=True)
```

In [24]:
```
new_df.head()
```

Out[24]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lender |
|---|---|---|---|---|---|
| **0** | 1 | 63863.135880 | 87983.134390 | 20.0 | |
| **1** | 2 | 51537.479640 | 63655.109150 | 17.0 | |
| **2** | 3 | 3298.248451 | 4776.336091 | 2.0 | |
| **3** | 4 | 14402.605700 | 13925.390670 | 5.0 | |
| **4** | 5 | 8635.683507 | 10143.513660 | 3.0 | |

Lets see what dataFrame looks like

In [25]:
```
df[pd.to_numeric(df.income,errors='coerce').isnull()].head()
```

Out[25]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lenders |
|---|---|---|---|---|---|
| **59** | 60 | NaN | 43289.045190 | 9.0 | 0 |
| **94** | 95 | NaN | 24642.825310 | 7.0 | 1 |
| **212** | 213 | NaN | 1632.956718 | 2.0 | 0 |
| **232** | 233 | NaN | 16213.468180 | 4.0 | 0 |
| **235** | 236 | NaN | 55011.606990 | 8.0 | 1 |

In [26]:
```
df[pd.to_numeric(df.debt_with_other_lenders,errors='coerce').isnull()].head()
```

Out[26]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lend |
|---|---|---|---|---|---|
| **15** | 16 | 12664.869050 | NaN | 5.0 | |
| **22** | 23 | 13734.955730 | NaN | 7.0 | |
| **40** | 41 | 6677.353290 | NaN | 4.0 | |
| **109** | 110 | 6174.622364 | NaN | 3.0 | |
| **130** | 131 | 8900.689168 | NaN | 4.0 | |

In [28]:
```
df[pd.to_numeric(df.credit_score,errors='coerce').isnull()].head()
```

Out[28]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lende |
|---|---|---|---|---|---|
| **26** | 27 | 17050.88032 | 13796.82406 | NaN | |
| **44** | 45 | 29062.36911 | 44209.97239 | NaN | |
| **50** | 51 | 17098.72002 | 19315.82411 | NaN | |
| **89** | 90 | 15567.36742 | 26949.65022 | NaN | |
| **101** | 102 | 16093.97296 | 48352.61055 | NaN | |

To Further Explore my Data I want to drop rows with NaN values

In [29]:
```
df_no_na = new_df.dropna()
df_no_na.head()
```

Out[29]:

| defaults_other_lenders | num_remittances_prev_12_mth | remittance_amt_prev_12_mth | main_remittance_c |
|---|---|---|---|
| 0 | 22 | 23377.338230 | |
| 0 | 20 | 8353.525522 | |
| 0 | 26 | 1213.782465 | |
| 0 | 18 | 6202.880445 | |
| 0 | 21 | 6175.393029 | |

In [30]:
```
df_no_na.shape
```

Out[30]:
```
(6638, 14)
```

After droping missing value, lets see howmuch data left for analysis

In [124]:
```
print('Percentage of data left after dropping missing values {}%'.format(((df_no_
```

```
Percentage of data left after dropping missing values 89.31646932185146%
```

Lets look at what the values of our data columns look like

In [39]:
```
def print_unique_col_values(df):
       for column in df_no_na:
            print(f'{column}: {df_no_na[column].unique()}')
print_unique_col_values(df_no_na)
```

```
cust_id: [   1    2    3 ... 7430 7431 7432]
income: [63863.13588  51537.47964   3298.248451 ... 46424.99755  28140.26622
 14095.82627 ]
debt_with_other_lenders: [87983.13439  63655.10915   4776.336091 ... 24527.6742
8  58965.30648
 13166.6542  ]
credit_score: [20. 17.  2.  5.  3.  7.  1.  8.  9.  4.  6. 12. 19. 11. 13. 14.
16. 10.
 15. 18.]
has_previous_defaults_other_lenders: [0 1]
num_remittances_prev_12_mth: [ 22  20  26  18  21  13  24  25  31  28  27  30
29  23 316  16  12  34
   8  19  33 324  17  32  35 327  15 347 339  14  11 314 328 329 323   9
  36 330 349  37  40 336 341 319 344 348 345 325 311 303 309  10 338 321
 337  38 300 313 302 340 308 322 342 304 333 334 317 343 306 335 346 310
 320 307 301 312 318 305 332 350  39 315 331 326   7]
remittance_amt_prev_12_mth: [23377.33823   8353.525522  1213.782465 ... 22261.9
5628   6162.548544
  4289.214953]
main_remittance_corridor: ['AE_IN' 'AE_PK' 'AE_PH']
opened_campaign_1: [0 1]
opened_campaign_2: [0 1]
opened_campaign_3: [0 1]
opened_campaign_4: [0 1]
tenure_years: [2.06525811 2.76167614 0.2970638  ... 0.48482528 0.49382951 0.425
786   ]
churned: [0 1]
```

And again our data types

In [40]:
```
df_no_na.dtypes
```

Out[40]:
```
cust_id                                  int64
income                                 float64
debt_with_other_lenders                float64
credit_score                           float64
has_previous_defaults_other_lenders      int64
num_remittances_prev_12_mth              int64
remittance_amt_prev_12_mth             float64
main_remittance_corridor                object
opened_campaign_1                        int64
opened_campaign_2                        int64
opened_campaign_3                        int64
opened_campaign_4                        int64
tenure_years                           float64
churned                                  int64
dtype: object
```

One of our feature/colums apears to be a categorical data, lets treat that

Performing One hot encoding for categorical column (main_remittance_corridor)

In [41]:
```
hot_encoded = pd.get_dummies(data=df_no_na, columns=['main_remittance_corridor'])
hot_encoded.columns
```

Out[41]:
```
Index(['cust_id', 'income', 'debt_with_other_lenders', 'credit_score',
       'has_previous_defaults_other_lenders', 'num_remittances_prev_12_mth',
       'remittance_amt_prev_12_mth', 'opened_campaign_1', 'opened_campaign_2',
       'opened_campaign_3', 'opened_campaign_4', 'tenure_years', 'churned',
       'main_remittance_corridor_AE_IN', 'main_remittance_corridor_AE_PH',
       'main_remittance_corridor_AE_PK'],
      dtype='object')
```

In [42]:
```
hot_encoded.head()
```

Out[42]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lender |
|---|---|---|---|---|---|
| **0** | 1 | 63863.135880 | 87983.134390 | 20.0 | ( |
| **1** | 2 | 51537.479640 | 63655.109150 | 17.0 | ( |
| **2** | 3 | 3298.248451 | 4776.336091 | 2.0 | ( |
| **3** | 4 | 14402.605700 | 13925.390670 | 5.0 | ( |
| **4** | 5 | 8635.683507 | 10143.513660 | 3.0 | ( |

More Exploration

In [43]:
```
# Lets Describe our dataset
hot_encoded.describe()
```

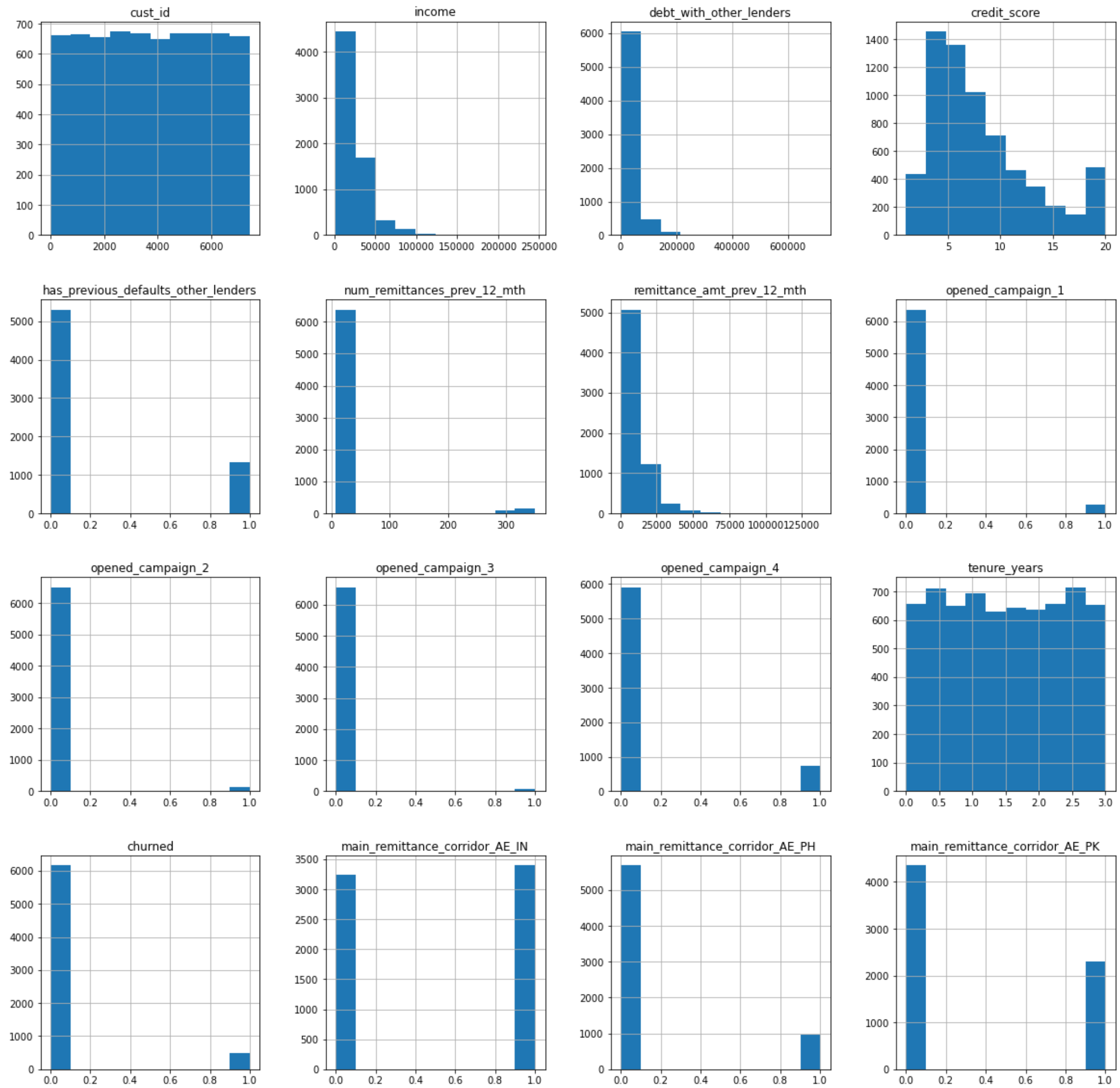
Out[43]:

| | cust_id | income | debt_with_other_lenders | credit_score | has_previous_defaults_ot |
|---|---|---|---|---|---|
| **count** | 6638.000000 | 6638.000000 | 6638.000000 | 6638.000000 | 6 |
| **mean** | 3717.925279 | 24077.287863 | 31987.129237 | 8.015065 | |
| **std** | 2144.916168 | 18905.538858 | 36988.660096 | 5.020562 | |
| **min** | 1.000000 | 1434.354208 | 653.062575 | 1.000000 | |
| **25%** | 1865.250000 | 11657.025275 | 11083.185205 | 4.000000 | |
| **50%** | 3710.500000 | 19010.887035 | 20660.959015 | 7.000000 | |
| **75%** | 5575.750000 | 30246.748850 | 39374.946608 | 10.000000 | |
| **max** | 7432.000000 | 244970.926100 | 715752.663000 | 20.000000 | |

# Data Visualization

Lets visualise all our columns

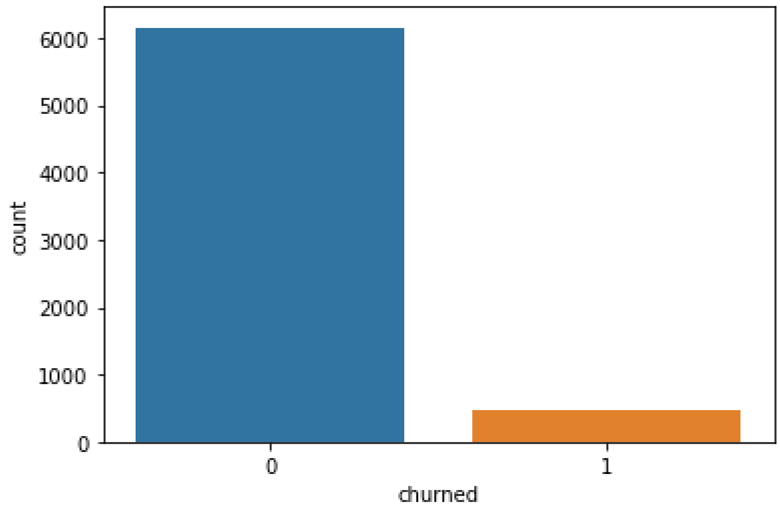
In [44]:
```
hot_encoded.hist(figsize=(20,20))
plt.show()
```




Lets see relationship between the churned customer as to against the un_churned ones

In [46]:
```
sns.countplot(df_no_na["churned"])
plt.show()
```

```
C:\Users\FrankEnedu\anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futu
reWarning: Pass the following variable as a keyword arg: x. From version 0.12,
the only valid positional argument will be `data`, and passing other arguments
without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```



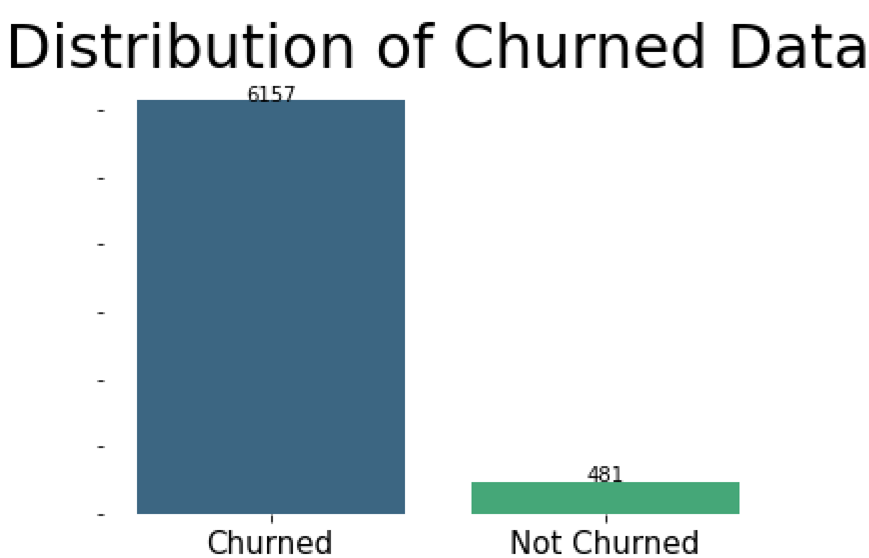
In [125]:
```
# using seaborns countplot to show distribution
fig, ax = plt.subplots()
g = sns.countplot(df_no_na.churned, palette='viridis')
g.set_xticklabels(['Churned', 'Not Churned'])
g.set_yticklabels([])

# function to show values on bars
def show_values_on_bars(axs):
    def _show_on_single_plot(ax):
        for p in ax.patches:
            _x = p.get_x() + p.get_width() / 2
            _y = p.get_y() + p.get_height()
            value = '{:.0f}'.format(p.get_height())
            ax.text(_x, _y, value, ha="center")

    if isinstance(axs, np.ndarray):
        for idx, ax in np.ndenumerate(axs):
            _show_on_single_plot(ax)
    else:
        _show_on_single_plot(axs)
show_values_on_bars(ax)

sns.despine(left=True, bottom=True)
plt.xlabel('')
plt.ylabel('')
plt.title('Distribution of Churned Data', fontsize=30)
plt.tick_params(axis='x', which='major', labelsize=15)
plt.show()
```

```
C:\Users\FrankEnedu\anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futu
reWarning: Pass the following variable as a keyword arg: x. From version 0.12,
the only valid positional argument will be `data`, and passing other arguments
without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```




Lets also see the percentage distribution

In [48]:
```
print("Percentage of Customer Churned is {:.1f}% and non-promoted employees is: {
    df_no_na[df_no_na['churned'] == 1].shape[0] / df_no_na.shape[0]*100,
    df_no_na[df_no_na['churned'] == 0].shape[0] / df_no_na.shape[0]*100))
```

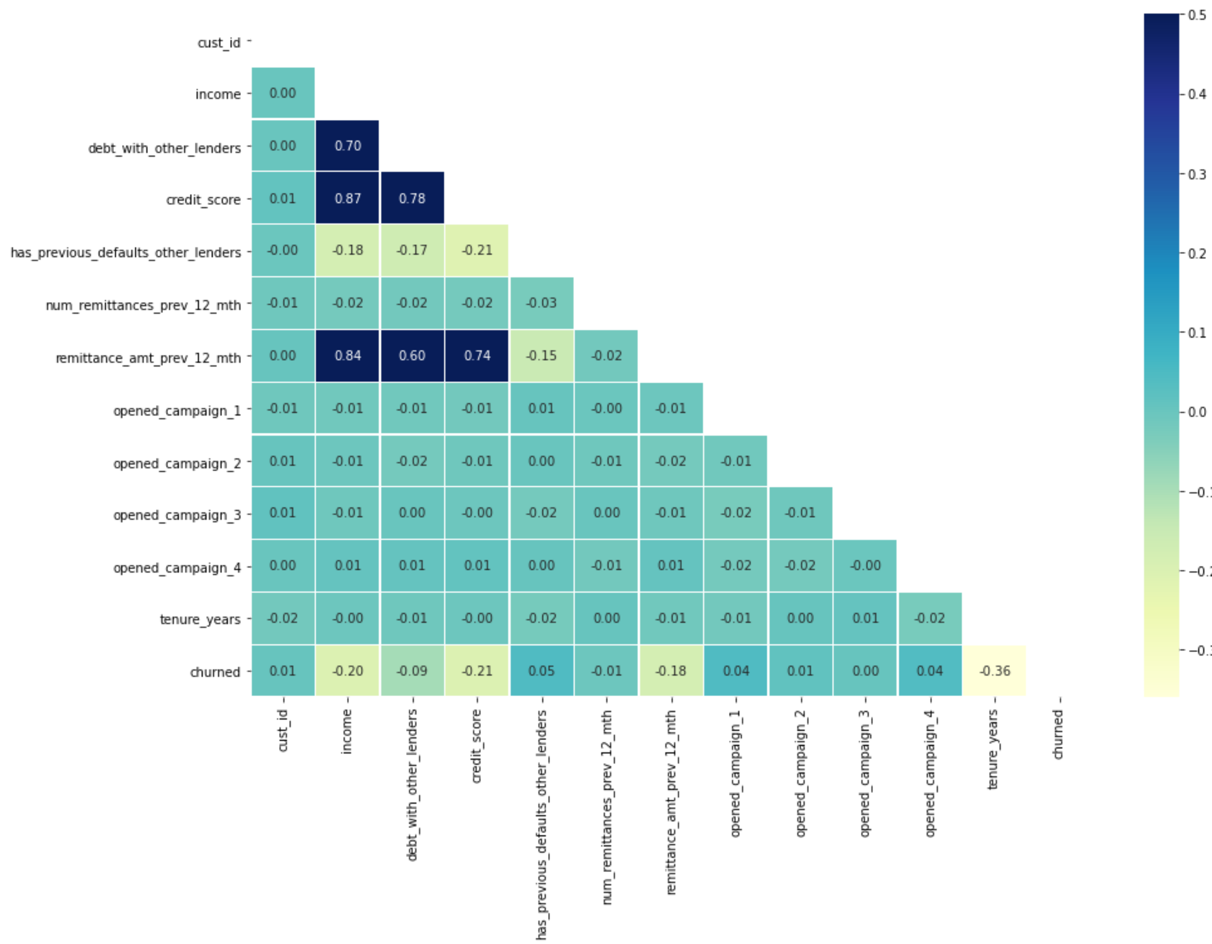
```
Percentage of Customer Churned is 7.2% and non-promoted employees is: 92.8%
```

Lets also look at the correlations visual of the columns

In [111]:
```
# Calculate correlations
corr = df_no_na.corr()
mask = np.zeros_like(corr)
mask[np.triu_indices_from(mask)] = True
```

In [112]:
```
# Heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(corr,
            vmax=.5,
            mask=mask,
             annot=True, fmt='.2f',
            linewidths=.2, cmap="YlGnBu")
```

Out[112]: <AxesSubplot:>




From the result of the correlation heatmap, we see that churned is either negatively correlated or has very poor correlate=ion with all othe features.

So when Income is increasing, its is less likely for customers to churn, when credit score or debt is high, their is less likelyhood for customer to churn

In [51]:
```
correlations = df_no_na.corr()['churned'].sort_values()
print('Most Positive Correlations: \n', correlations.tail(5))
print('\nMost Negative Correlations: \n', correlations.head(5))
```

```
Most Positive Correlations:
 opened_campaign_2                     0.013552
opened_campaign_4                     0.041635
opened_campaign_1                     0.044241
has_previous_defaults_other_lenders    0.045323
churned                               1.000000
Name: churned, dtype: float64

Most Negative Correlations:
 tenure_years                 -0.359714
credit_score                 -0.205139
income                       -0.203039
remittance_amt_prev_12_mth   -0.181006
debt_with_other_lenders      -0.093960
Name: churned, dtype: float64
```

# Data Modelling

Declearing Dependent and Independent Variable

In [52]:
```
X = hot_encoded.drop(["churned", 'cust_id'], axis = 1)
```

In [55]:
```
X.head()
```

Out[55]:

| | income | debt_with_other_lenders | credit_score | has_previous_defaults_other_lenders | num_r |
|---|---|---|---|---|---|
| **0** | 63863.135880 | 87983.134390 | 20.0 | 0 | |
| **1** | 51537.479640 | 63655.109150 | 17.0 | 0 | |
| **2** | 3298.248451 | 4776.336091 | 2.0 | 0 | |
| **3** | 14402.605700 | 13925.390670 | 5.0 | 0 | |
| **4** | 8635.683507 | 10143.513660 | 3.0 | 0 | |

In [57]:
```
y = hot_encoded["churned"]
```

In [58]:
```
y.head()
```

Out[58]:
```
0    0
1    0
2    1
3    0
4    0
Name: churned, dtype: int64
```

Spliting Test and Train dataset

In [59]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2)
```

In [60]:
```
X_test.shape
```

Out[60]:
```
(1328, 14)
```

In [61]:
```
y_train.shape, y_test.shape
```

Out[61]:
```
((5310,), (1328,))
```

# Building Data Models (Random Forest)

Using Random Forest

In [63]:
```
rf = RandomForestClassifier()
```

In [64]:
```
model = rf.fit(X_train,y_train)
```

In [65]:
```
pred = model.predict(X_test)
```

In [73]:
```
rf_results = pd.DataFrame({'Actual value': y_test, "Predicted value": pred})
```

In [74]:
```
rf_results.head()
```

Out[74]:

| | Actual value | Predicted value |
|---|---|---|
| **2125** | 0 | 0 |
| **2536** | 0 | 0 |
| **1672** | 0 | 0 |
| **3628** | 0 | 0 |
| **4874** | 0 | 0 |

The table above shows how good our model was able to predit the actual value, les now evaluate our model

In [70]:
```
accuracy_score(y_test, pred)
```

Out[70]:
```
0.9721385542168675
```

Our model has an acuracy of 97%, This simply means that if we make prediction 100 time, our model is likely to make the right prediction 97 time. this is pretty good. Now lets look at way to evaluate our model

In [107]:
```
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.99      1232
           1       0.94      0.66      0.77        96

    accuracy                           0.97      1328
   macro avg       0.96      0.83      0.88      1328
weighted avg       0.97      0.97      0.97      1328
```

We can see that the model precision an 1 and two are also pretty good on 96% and 97% respectively

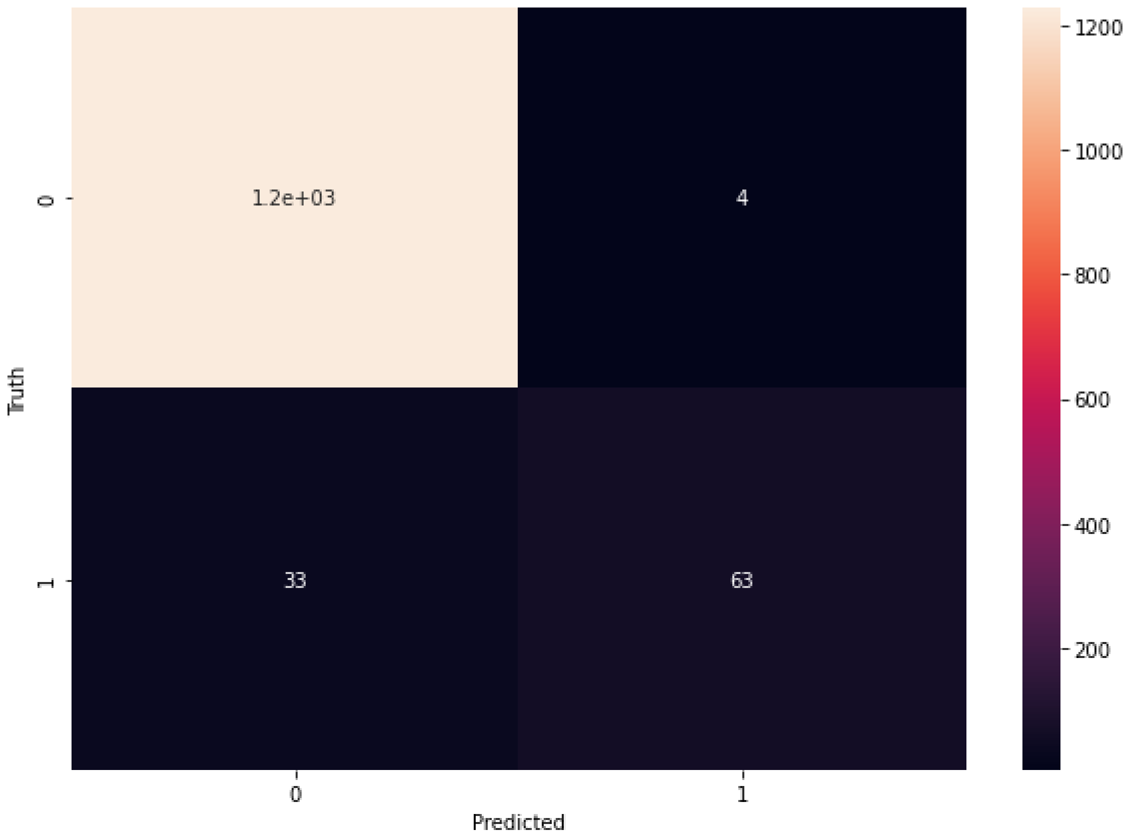
Lets now see the distribution on how our model faired with prediction using the confussion matrix heatmap

In [116]:
```
cm_rf = confusion_matrix(y_test,pred)
cm_rf
```

Out[116]:
```
array([[1228,    4],
       [  33,   63]], dtype=int64)
```

In [117]:
```
plt.figure(figsize = (10,7))
sn.heatmap(cm_rf, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

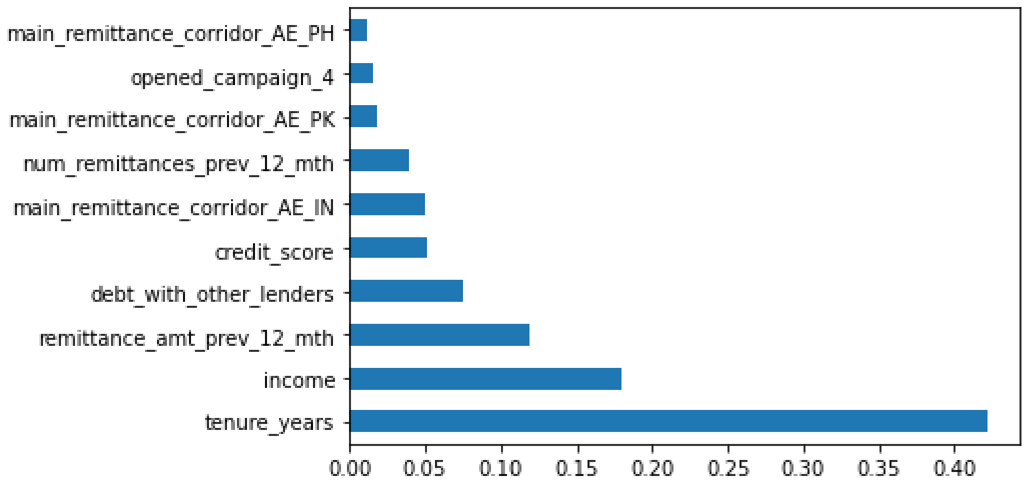
Out[117]: Text(69.0, 0.5, 'Truth')




# Building Data Models ( XGBoost)

In [76]:
```
model2 = ExtraTreesClassifier()
model2.fit(X,y)
print(model2.feature_importances_)
feat_importance = pd.Series(model.feature_importances_,index=X.columns)
feat_importance.nlargest(10).plot(kind = 'barh')
plt.show()
```

```
[0.17336712 0.08107227 0.10179593 0.01042481 0.05818066 0.118145
 0.01042213 0.00588728 0.0039046  0.02009212 0.37284926 0.02616268
 0.00749512 0.01020103]
```




the above chat has ranked our various feature base on the correlation with our dependent variable

In [77]:
```
classifier = XGBClassifier(base_score=0.8,n_estimators=350,max_depth=4,learning_r
classifier.fit(X_train, y_train)
```

```
C:\Users\FrankEnedu\anaconda3\lib\site-packages\xgboost\sklearn.py:1224: UserWa
rning: The use of label encoder in XGBClassifier is deprecated and will be remo
ved in a future release. To remove this warning, do the following: 1) Pass opti
on use_label_encoder=False when constructing XGBClassifier object; and 2) Encod
e your labels (y) as integers starting with 0, i.e. 0, 1, 2, ..., [num_class -
1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
```

```
[16:01:18] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.
0/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default evaluation metric
used with the objective 'binary:logistic' was changed from 'error' to 'loglos
s'. Explicitly set eval_metric if you'd like to restore the old behavior.
```

Out[77]:
```
XGBClassifier(base_score=0.8, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.1, max_delta_step=0,
              max_depth=4, min_child_weight=4, missing=nan,
              monotone_constraints='()', n_estimators=350, n_jobs=8,
              num_parallel_tree=1, predictor='auto', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='exact', validate_parameters=1, verbosity=None)
```

In [88]:
```
xg_pred = classifier.predict(X_test)
```

In [89]:
```
xg_results = pd.DataFrame({'Actual value': y_test, "Predicted value": xg_pred})
```

In [119]:
```
accuracy_score(y_test, xg_pred)
```

Out[119]:
```
0.9766566265060241
```

Our model has an acuracy of approximately 98%, This simply means that if we make prediction 100 time, our model is likely to make the right prediction 98 time. this is pretty good. Now lets look at way to evaluate our model

In [81]:
```
xg_results.head()
```

Out[81]:

| | Actual value | Predicted value |
|---|---|---|
| **2125** | 0 | 0 |
| **2536** | 0 | 0 |
| **1672** | 0 | 0 |
| **3628** | 0 | 0 |
| **4874** | 0 | 0 |

The table above shows how good our model was able to predit the actual value, les now evaluate our model

In [106]:
```
print(classification_report(y_test,xg_pred))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      1232
           1       0.92      0.74      0.82        96

    accuracy                           0.98      1328
   macro avg       0.95      0.87      0.90      1328
weighted avg       0.98      0.98      0.98      1328
```

We can see that the model precision an 0 and 1 are also pretty good on 98% and 92% respectively

Lets now see the distribution on how our model faired with prediction using the confussion matrix heatmap

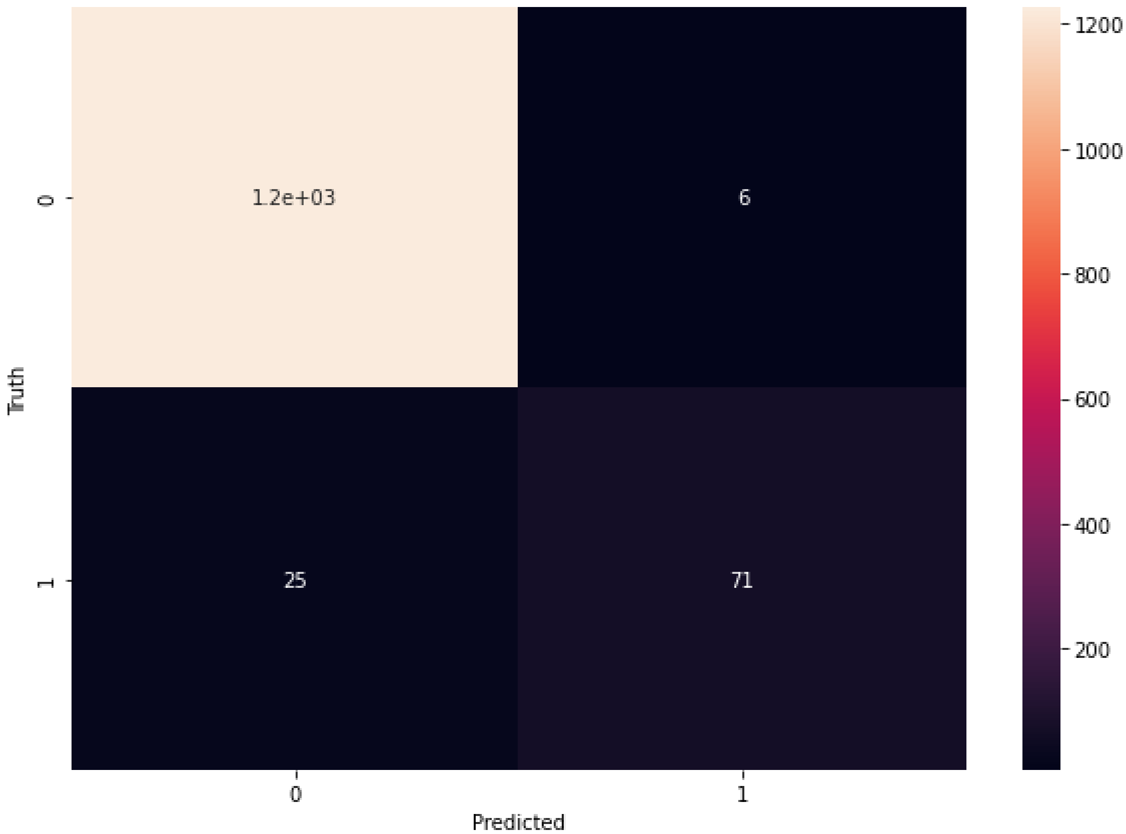
In [92]:
```
cm_xg = confusion_matrix(y_test, xg_pred)
```

In [93]:
```
cm_xg
```

Out[93]:
```
array([[1226,    6],
       [  25,   71]], dtype=int64)
```

In [94]:
```
import seaborn as sn
plt.figure(figsize = (10,7))
sn.heatmap(cm_xg, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Out[94]:
```
Text(69.0, 0.5, 'Truth')
```




# Building Data Models (Logistics Regresion)

In [96]:
```
model3 = LogisticRegression()
```

In [97]:
```
model3.fit(X_train, y_train)
```

```
C:\Users\FrankEnedu\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.
py:763: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-
learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regressi
on (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regressi
on)
  n_iter_i = _check_optimize_result(
```

Out[97]:
```
LogisticRegression()
```

In [98]:
```
model3.score(X_test, y_test)
```

Out[98]:
```
0.9382530120481928
```

```
Our model has an acuracy of approximately 94%, This simply means that if we
make prediction 100 time, our model is likely to make the right prediction 94
time.
```

In [99]:
```
log_pred = model3.predict(X_test)
```

In [109]:
```
log_results = pd.DataFrame({'Actual value': y_test, "Predicted value": xg_pred})
```

In [110]:
```
log_results.head()
```

Out[110]:

| | Actual value | Predicted value |
|---|---|---|
| **2125** | 0 | 0 |
| **2536** | 0 | 0 |
| **1672** | 0 | 0 |
| **3628** | 0 | 0 |
| **4874** | 0 | 0 |

The table above shows how good our model was able to predit the actual value, let's now evaluate our model

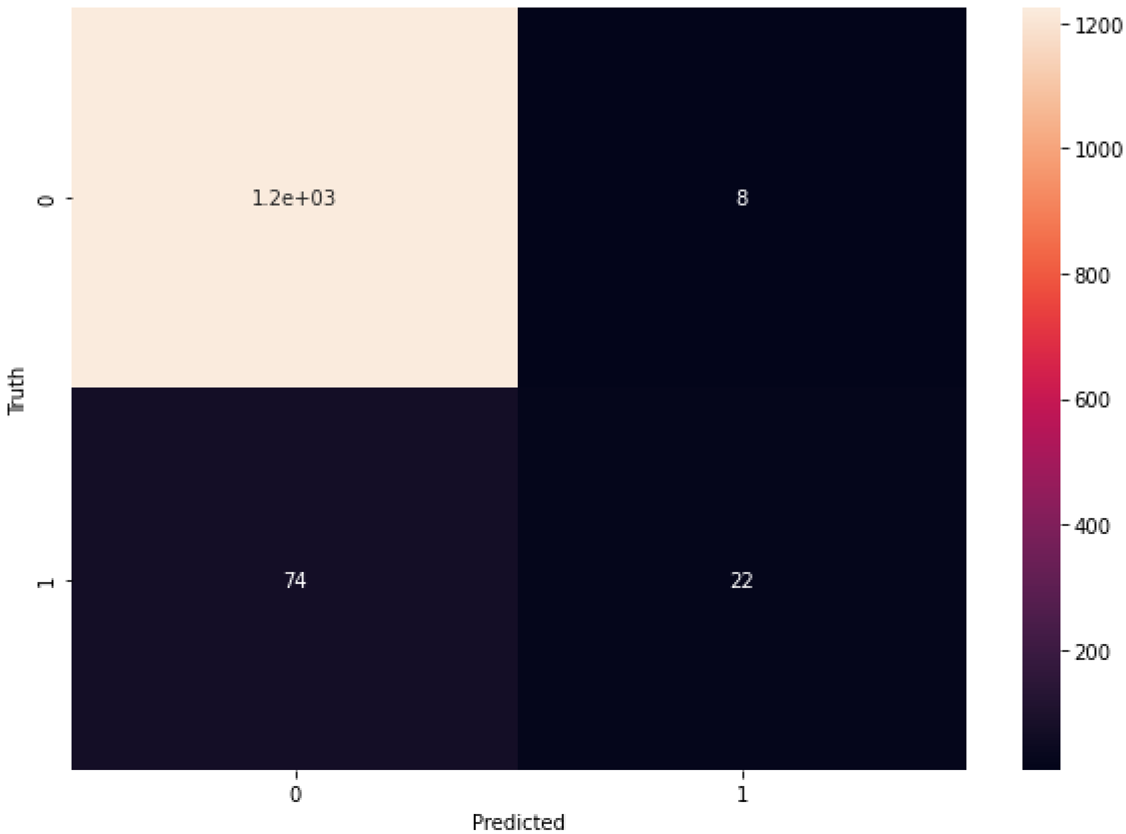
In [101]:
```
cm_log = confusion_matrix(y_test, log_pred)
cm_log
```

Out[101]:
```
array([[1224,    8],
       [  74,   22]], dtype=int64)
```

In [ ]:

In [102]:
```
import seaborn as sn
plt.figure(figsize = (10,7))
sn.heatmap(cm_log, annot=True)
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Out[102]:
```
Text(69.0, 0.5, 'Truth')
```




In [105]:
```
print(classification_report(y_test,log_pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.99      0.97      1232
           1       0.73      0.23      0.35        96

    accuracy                           0.94      1328
   macro avg       0.84      0.61      0.66      1328
weighted avg       0.93      0.94      0.92      1328
```

In [ ]: