October 30, 2019

1 My project: Initial Analysis of Home Credit Default Risk

My project is based on the data provided within the Home Credit Default Risk competition hosted and still running on Kaggle.

The objective of this competition is to use historical loan application data to predict whether or not an applicant will be able to repay a loan.

This Machine Learning application is a standard supervised classification task:

- **Supervised**: The labels are included in the training data and the goal is to train a model to learn to predict the labels from the features
- **Binary Classification**: The label is a binary variable, 0 (will repay loan on time), 1 (will have difficulty repaying loan)

2 Data

The data is provided by Home Credit, a service dedicated to provided lines of credit (loans) to the unbanked population.

Predicting whether or not a client will repay a loan or have difficulty is a critical business need. Home Credit is hosting this competition on Kaggle to see what sort of models the machine learning community can develop to help them in this task.

Home Credit provides various different sources of data. In my notebook, I will only use the main application train data, which will be divided in a training and test data set. This source contains data with information about each loan application at Home Credit. Every loan has its own row and is identified by the feature `SK_ID_CURR`. The data comes with the `TARGET` indicating 0: the loan was repaid or 1: the loan was not repaid.

We are provided with the definitions of all the columns (in Home-Credit_columns_description.csv.)

All other data sources are neglected for the baseline analysis in this notebook. The rationale is, that according to my initial data analysis and scope of the final project, the main application data ,first, contains data features that correlate with the target and second, this would be too work-intense and complex for the estimated effort and duration of this two-week-assignment.

2.1 Metric ROC AUC

Before starting, we need to understand the metric by which the competition result is judged. It is a common classification metric known as the Receiver Operating Characteristic Area Under the Curve (ROC AUC, also sometimes called AUROC).

The ROC AUC consists of two individual concepts. The Reciever Operating Characteristic (ROC) curve graphs the true positive rate versus the false positive rate.

A single line on the graph indicates the curve for a single model, and movement along a line indicates changing the threshold used for classifying a positive instance. The threshold starts at 0 in the upper right to and goes to 1 in the lower left. A curve that is to the left and above another curve indicates a better model. For example, the blue model is better than the red model, which is better than the black diagonal line which indicates a naive random guessing model.

The Area Under the Curve (AUC) explains itself by its name! It is simply the area under the ROC curve. (This is the integral of the curve.) This metric is between 0 and 1 with a better model scoring higher. A model that simply guesses at random will have an ROC AUC of 0.5.

When we measure a classifier according to the ROC AUC, we do not generate 0 or 1 predictions, but rather a probability between 0 and 1. This may be confusing because we usually like to think in terms of accuracy, but when we get into problems with inbalanced classes (we will see this is the case), accuracy is not the best metric. For example, if I wanted to build a model that could detect terrorists with 99.9999% accuracy, I would simply make a model that predicted every single person was not a terrorist. Clearly, this would not be effective (the recall would be zero) and we use more advanced metrics such as ROC AUC or the F1 score to more accurately reflect the performance of a classifier. A model with a high ROC AUC will also have a high accuracy, but the ROC AUC is a better representation of model performance..

2.2 Imports

We are using a typical data science stack: `numpy`, `pandas`, `sklearn`, `matplotlib`.

[5]:
```
# numpy and pandas for data manipulation
import numpy as np
import pandas as pd

# sklearn preprocessing for dealing with categorical variables
from sklearn.preprocessing import LabelEncoder

# sklearn libraries for decision tree classifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# File system manangement
import os

# Suppress warnings
import warnings
warnings.filterwarnings('ignore')

# matplotlib and seaborn for plotting
import matplotlib.pyplot as plt
import seaborn as sns
```

[6]:
```
# show path of the current directory
import os.path
os.getcwd()
```

[6]:
```
'/Users/joern.grimmer/Documents/010_Learnings_DataSciencePython/Week9_FinalProje
ct_Week1'
```

2.3 Read Data

First, we list all the available data files. There are a total of 9 files: 1 main file for training (with target) 1 main file for testing (without the target), 1 example submission file, and 6 other files containing additional information about each loan.

For this project we will use the main file for training and the main file for testing.

[7]:
```
# List files available
print(os.listdir("/Users/joern.grimmer/Documents/
 ↪010_Learnings_DataSciencePython/Week9_FinalProject_Week1/Data/
 ↪Home_Credit_Default_Risk/home-credit-default-risk/"))
```

```
['application_test.csv', 'HomeCredit_columns_description.csv',
'POS_CASH_balance.csv', 'credit_card_balance.csv', 'installments_payments.csv',
'application_train.csv', 'bureau.csv', 'previous_application.csv',
'bureau_balance.csv', 'sample_submission.csv']
```

[1]:
```
# Show description of the data
#!cat /Users/joern.grimmer/Documents/010_Learnings_DataSciencePython/
 ↪Week9_FinalProject_Week1/Data/Home_Credit_Default_Risk/
 ↪home-credit-default-risk/HomeCredit_columns_description.csv
```

[17]:
```
# Read description of the data
app_description = pd.read_csv('/Users/joern.grimmer/Documents/
 ↪010_Learnings_DataSciencePython/Week9_FinalProject_Week1/Data/
 ↪Home_Credit_Default_Risk/home-credit-default-risk/
 ↪HomeCredit_columns_description.csv', "utf-8")
print('Description data shape: ', app_description.shape)
```

```
Description data shape:  (219, 1)
```

2.4 Show Shape of Data Set

[18]:
```
# Data Set
# Show shape - colums and rows of data set
app_data = pd.read_csv('/Users/joern.grimmer/Documents/
 ↪010_Learnings_DataSciencePython/Week9_FinalProject_Week1/Data/
 ↪Home_Credit_Default_Risk/home-credit-default-risk/application_train.csv')
print('Data shape: ', app_data.shape)
app_data.head(10)
```

```
Data shape:  (307511, 122)
```

[18]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	\
0	100002	1	Cash loans	M	N	
1	100003	0	Cash loans	F	N	
2	100004	0	Revolving loans	M	Y	
3	100006	0	Cash loans	F	N	
4	100007	0	Cash loans	M	N	
5	100008	0	Cash loans	M	N	
6	100009	0	Cash loans	F	Y	
7	100010	0	Cash loans	M	Y	
8	100011	0	Cash loans	F	N	
9	100012	0	Revolving loans	M	N	

	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	\
0	Y	0	202500.0	406597.5	24700.5	
1	N	0	270000.0	1293502.5	35698.5	
2	Y	0	67500.0	135000.0	6750.0	
3	Y	0	135000.0	312682.5	29686.5	
4	Y	0	121500.0	513000.0	21865.5	
5	Y	0	99000.0	490495.5	27517.5	
6	Y	1	171000.0	1560726.0	41301.0	
7	Y	0	360000.0	1530000.0	42075.0	
8	Y	0	112500.0	1019610.0	33826.5	
9	Y	0	135000.0	405000.0	20250.0	

	...	FLAG_DOCUMENT_18	FLAG_DOCUMENT_19	FLAG_DOCUMENT_20	FLAG_DOCUMENT_21	\
0	...	0	0	0	0	
1	...	0	0	0	0	
2	...	0	0	0	0	
3	...	0	0	0	0	
4	...	0	0	0	0	
5	...	0	0	0	0	
6	...	0	0	0	0	
7	...	0	0	0	0	
8	...	0	0	0	0	
9	...	0	0	0	0	

	AMT_REQ_CREDIT_BUREAU_HOUR	AMT_REQ_CREDIT_BUREAU_DAY	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	NaN	NaN	
4	0.0	0.0	
5	0.0	0.0	
6	0.0	0.0	
7	0.0	0.0	

```
8                          0.0                          0.0
9                          NaN                          NaN

   AMT_REQ_CREDIT_BUREAU_WEEK  AMT_REQ_CREDIT_BUREAU_MON  \
0                         0.0                        0.0
1                         0.0                        0.0
2                         0.0                        0.0
3                         NaN                        NaN
4                         0.0                        0.0
5                         0.0                        0.0
6                         0.0                        1.0
7                         0.0                        0.0
8                         0.0                        0.0
9                         NaN                        NaN

   AMT_REQ_CREDIT_BUREAU_QRT  AMT_REQ_CREDIT_BUREAU_YEAR
0                        0.0                         1.0
1                        0.0                         0.0
2                        0.0                         0.0
3                        NaN                         NaN
4                        0.0                         0.0
5                        1.0                         1.0
6                        1.0                         2.0
7                        0.0                         0.0
8                        0.0                         1.0
9                        NaN                         NaN

[10 rows x 122 columns]
```

The dataset has 307511 observations (each one a separate loan) and 122 features (variables) including the TARGET (the label we want to predict).

3 Initial Data Exploration & Cleansing

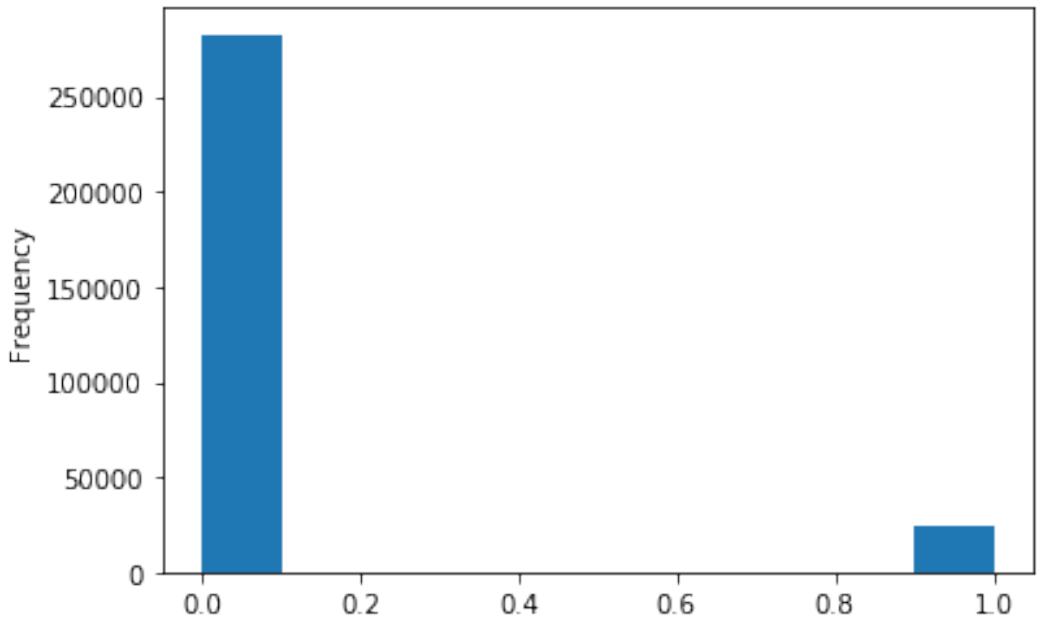
Exploratory Data Analysis is an iterative process where we calculate statistics and make figures to find trends, anomalies, patterns, or relationships within the data. The goal of the analysis is to learn what our data can tell us. It generally starts out with a high level overview, then narrows in to specific areas as we find intriguing areas of the data. The findings may be interesting in their own right, or they can be used to inform our modeling choices, such as by helping us decide which features to use.

[19]:
```
app_data['TARGET'].value_counts()
```

[19]:
```
0    282686
1     24825
Name: TARGET, dtype: int64
```

[20]:
```
app_data['TARGET'].astype(int).plot.hist()
```

[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3c17d630>



From this information, we see there is an *imbalanced class problem*. There are far more loans that were repaid on time than loans that were not repaid. In more sophisticated machine learning models we can weight the classes by their representation in the data to reflect this imbalance. However, this will not be part of the project and need to be considered as a limitation in the context of my results.

3.1 Examine Missing Values

Next we can look at the number and percentage of missing values in each column.

[21]:
```
# Function to calculate missing values by column# Funct
def missing_values_table(df):
        # Total missing values
        mis_val = df.isnull().sum()

        # Percentage of missing values
        mis_val_percent = 100 * df.isnull().sum() / len(df)

        # Make a table with the results
        mis_val_table = pd.concat([mis_val, mis_val_percent], axis=1)

        # Rename the columns
        mis_val_table_ren_columns = mis_val_table.rename(
        columns = {0 : 'Missing Values', 1 : '% of Total Values'})
```

```
        # Sort the table by percentage of missing descending
        mis_val_table_ren_columns = mis_val_table_ren_columns[
            mis_val_table_ren_columns.iloc[:,1] != 0].sort_values(
        '% of Total Values', ascending=False).round(1)

        # Print some summary information
        print ("Your selected dataframe has " + str(df.shape[1]) + " columns.\n"
            "There are " + str(mis_val_table_ren_columns.shape[0]) +
              " columns that have missing values.")

        # Return the dataframe with missing information
        return mis_val_table_ren_columns
```

[23]:
```
# Missing values statistics
missing_values = missing_values_table(app_data)
missing_values.head(10)
```

```
Your selected dataframe has 122 columns.
There are 67 columns that have missing values.
```

[23]:

	Missing Values	% of Total Values
COMMONAREA_MEDI	214865	69.9
COMMONAREA_AVG	214865	69.9
COMMONAREA_MODE	214865	69.9
NONLIVINGAPARTMENTS_MEDI	213514	69.4
NONLIVINGAPARTMENTS_MODE	213514	69.4
NONLIVINGAPARTMENTS_AVG	213514	69.4
FONDKAPREMONT_MODE	210295	68.4
LIVINGAPARTMENTS_MODE	210199	68.4
LIVINGAPARTMENTS_MEDI	210199	68.4
LIVINGAPARTMENTS_AVG	210199	68.4

When it comes time to build the required machine learning models, we will have to fill in these missing values (known as imputation).

In a more advanced analysis, we could use models such as XGBoost that can handle missing values with no need for imputation. Another option would be to drop columns with a high percentage of missing values, although it is impossible to know ahead of time if these columns will be helpful to our model.

Therefore, we will keep all of the columns for now.

3.2 Column Types

Let's look at the number of columns of each data type. `int64` and `float64` are numeric variables (which can be either discrete or continuous). `object` columns contain strings and are categorical features. .

[24]:
```
# Number of each type of column
app_data.dtypes.value_counts()
```

[24]:
```
float64    65
int64      41
object     16
dtype: int64
```

Let's now look at the number of unique entries in each of the `object` (categorical) columns.

[25]:
```
# Number of unique classes in each object column
app_data.select_dtypes('object').apply(pd.Series.nunique, axis = 0)
```

[25]:
```
NAME_CONTRACT_TYPE             2
CODE_GENDER                    3
FLAG_OWN_CAR                   2
FLAG_OWN_REALTY                2
NAME_TYPE_SUITE                7
NAME_INCOME_TYPE               8
NAME_EDUCATION_TYPE            5
NAME_FAMILY_STATUS             6
NAME_HOUSING_TYPE              6
OCCUPATION_TYPE               18
WEEKDAY_APPR_PROCESS_START     7
ORGANIZATION_TYPE             58
FONDKAPREMONT_MODE             4
HOUSETYPE_MODE                 3
WALLSMATERIAL_MODE             7
EMERGENCYSTATE_MODE            2
dtype: int64
```

Most of the categorical variables have a relatively small number of unique entries. We will need to find a way to deal with these categorical variables.

One variable, we are interested in, is the occupation type. Therefore we look at the categories in this feature.

[26]:
```
app_data['OCCUPATION_TYPE']
```

[26]:
```
0                 Laborers
1              Core staff
2                 Laborers
3                 Laborers
4              Core staff
5                 Laborers
6              Accountants
7                 Managers
8                       NaN
9                 Laborers
10             Core staff
11                      NaN
12                Laborers
13                 Drivers
14                Laborers
15                Laborers
```

```
16                      Drivers
17                     Laborers
18                     Laborers
19                   Core staff
20                     Laborers
21                   Sales staff
22                   Sales staff
23                           NaN
24                      Drivers
25                Cleaning staff
26                 Cooking staff
27                     Laborers
28                           NaN
29                     Laborers
                   ...
307481                 Managers
307482                 Managers
307483                       NaN
307484                 Managers
307485               Sales staff
307486                       NaN
307487                       NaN
307488                Core staff
307489               Sales staff
307490                  Laborers
307491                   Drivers
307492               Sales staff
307493            Security staff
307494                   Drivers
307495    High skill tech staff
307496             Cooking staff
307497               Sales staff
307498                  Laborers
307499            Medicine staff
307500                       NaN
307501       Low-skill Laborers
307502                 Laborers
307503                Core staff
307504               Sales staff
307505                       NaN
307506               Sales staff
307507                       NaN
307508                 Managers
307509                 Laborers
307510                 Laborers
Name: OCCUPATION_TYPE, Length: 307511, dtype: object
```

3.3 Encoding Categorical Variables

A machine learning model cannot deal with categorical variables (except for some models such as LightGBM). Therefore, we have to find a way to encode (represent) these variables as numbers before handing them off to the model. There are two main ways to carry out this process:

- Label encoding: assign each unique category in a categorical variable with an integer. No new columns are created.
- One-hot encoding: create a new column for each unique category in a categorical variable. Each observation recieves a 1 in the column for its corresponding category and a 0 in all other new columns.

The challenge with label encoding is that it gives the categories an arbitrary ordering. The value assigned to each of the categories is random and does not reflect any inherent aspect of the category. The actual assignment of the integers is arbitrary. Therefore, when we perform label encoding, the model might use the relative value of the feature to assign weights which is not what we want. If we only have two unique values for a categorical variable (such as Male/Female), then label encoding is fine, but for more than 2 unique categories, one-hot encoding is the safe option.

There is some debate about the relative merits of these approaches, and some models can deal with label encoded categorical variables with no issues. Here is a good Stack Overflow discussion. I think (and this is just a personal opinion) for categorical variables with many classes, one-hot encoding is the safest approach because it does not impose arbitrary values to categories. The only downside to one-hot encoding is that the number of features (dimensions of the data) can explode with categorical variables with many categories. To deal with this, we can perform one-hot encoding followed by PCA or other dimensionality reduction methods to reduce the number of dimensions (while still trying to preserve information).

In my notebook, I will use Label Encoding for any categorical variables with only 2 categories and One-Hot Encoding for any categorical variables with more than 2 categories.

3.3.1 Label Encoding and One-Hot Encoding

Let's implement the policy described above: for any categorical variable (`dtype == object`) with 2 unique categories, we will use label encoding, and for any categorical variable with more than 2 unique categories, we will use one-hot encoding.

For label encoding, we use the Scikit-Learn `LabelEncoder` and for one-hot encoding, the pandas `get_dummies(df)` function.

[27]:
```
# Create a label encoder object
le = LabelEncoder()
le_count = 0

# Iterate through the columns
for col in app_data:
    if app_data[col].dtype == 'object':
        # If 2 or fewer unique categories
        if len(list(app_data[col].unique())) <= 2:
            # Train on data
            le.fit(app_data[col])
            # Transform data
```

```
            app_data[col] = le.transform(app_data[col])

            # Keep track of how many columns were label encoded
            le_count += 1

print('%d columns were label encoded.' % le_count)
```

```
3 columns were label encoded.
```

[28]:
```
# one-hot encoding of categorical variables
app_data = pd.get_dummies(app_data)

print('Data Features shape: ', app_data.shape)
```

```
Data Features shape:  (307511, 243)
```

3.3.2 General descriptive statistics data set

Now, that we have dealt with the categorical variables and the outliers, let's continue with exploratory data analysis. First of all, we want to understand if general statistics are similar on training and test data. For this exploration we use the `describe` method.

[30]:
```
# Descriptive Statistics on data set
app_data.describe()
```

[30]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	FLAG_OWN_CAR	\
count	307511.000000	307511.000000	307511.000000	307511.000000	
mean	278180.518577	0.080729	0.095213	0.340108	
std	102790.175348	0.272419	0.293509	0.473746	
min	100002.000000	0.000000	0.000000	0.000000	
25%	189145.500000	0.000000	0.000000	0.000000	
50%	278202.000000	0.000000	0.000000	0.000000	
75%	367142.500000	0.000000	0.000000	1.000000	
max	456255.000000	1.000000	1.000000	1.000000	

	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	\
count	307511.000000	307511.000000	3.075110e+05	3.075110e+05	
mean	0.693673	0.417052	1.687979e+05	5.990260e+05	
std	0.460968	0.722121	2.371231e+05	4.024908e+05	
min	0.000000	0.000000	2.565000e+04	4.500000e+04	
25%	0.000000	0.000000	1.125000e+05	2.700000e+05	
50%	1.000000	0.000000	1.471500e+05	5.135310e+05	
75%	1.000000	1.000000	2.025000e+05	8.086500e+05	
max	1.000000	19.000000	1.170000e+08	4.050000e+06	

	AMT_ANNUITY	AMT_GOODS_PRICE	...	HOUSETYPE_MODE_terraced house	\
count	307499.000000	3.072330e+05	...	307511.000000	
mean	27108.573909	5.383962e+05	...	0.003941	

```
std     14493.737315     3.694465e+05  ...                      0.062656
min      1615.500000     4.050000e+04  ...                      0.000000
25%     16524.000000     2.385000e+05  ...                      0.000000
50%     24903.000000     4.500000e+05  ...                      0.000000
75%     34596.000000     6.795000e+05  ...                      0.000000
max    258025.500000     4.050000e+06  ...                      1.000000

       WALLSMATERIAL_MODE_Block  WALLSMATERIAL_MODE_Mixed  \
count             307511.000000             307511.000000
mean                   0.030090                  0.007466
std                    0.170835                  0.086085
min                    0.000000                  0.000000
25%                    0.000000                  0.000000
50%                    0.000000                  0.000000
75%                    0.000000                  0.000000
max                    1.000000                  1.000000

       WALLSMATERIAL_MODE_Monolithic  WALLSMATERIAL_MODE_Others  \
count                 307511.000000              307511.000000
mean                       0.005785                   0.005284
std                        0.075840                   0.072501
min                        0.000000                   0.000000
25%                        0.000000                   0.000000
50%                        0.000000                   0.000000
75%                        0.000000                   0.000000
max                        1.000000                   1.000000

       WALLSMATERIAL_MODE_Panel  WALLSMATERIAL_MODE_Stone, brick  \
count             307511.000000                    307511.000000
mean                   0.214757                         0.210773
std                    0.410654                         0.407858
min                    0.000000                         0.000000
25%                    0.000000                         0.000000
50%                    0.000000                         0.000000
75%                    0.000000                         0.000000
max                    1.000000                         1.000000

       WALLSMATERIAL_MODE_Wooden  EMERGENCYSTATE_MODE_No  \
count              307511.000000           307511.000000
mean                   0.017437                0.518446
std                    0.130892                0.499660
min                    0.000000                0.000000
25%                    0.000000                0.000000
50%                    0.000000                1.000000
75%                    0.000000                1.000000
max                    1.000000                1.000000
```

```
       EMERGENCYSTATE_MODE_Yes
count            307511.000000
mean                  0.007570
std                   0.086679
min                   0.000000
25%                   0.000000
50%                   0.000000
75%                   0.000000
max                   1.000000

[8 rows x 243 columns]
```

3.3.3 Anomalies

First, we want to look for anomalies within the data. These may be due to mis-typed numbers, errors in measuring equipment, or they could be valid but extreme measurements. The numbers in the DAYS_BIRTH column are negative because they are recorded relative to the current loan application.

[31]:
```
# check DAYS_BIRTH for anomaly and outlier
app_data['DAYS_BIRTH'].describe()
```

[31]:
```
count    307511.000000
mean     -16036.995067
std        4363.988632
min      -25229.000000
25%      -19682.000000
50%      -15750.000000
75%      -12413.000000
max       -7489.000000
Name: DAYS_BIRTH, dtype: float64
```

To see these stats in years, we can mutliple by -1 and divide by the number of days in a year:

[32]:
```
# Convert DAYS_BIRTH in years and to positive numbers
(app_data['DAYS_BIRTH'] / -365).describe()
```

[32]:
```
count    307511.000000
mean         43.936973
std          11.956133
min          20.517808
25%          34.008219
50%          43.150685
75%          53.923288
max          69.120548
Name: DAYS_BIRTH, dtype: float64
```

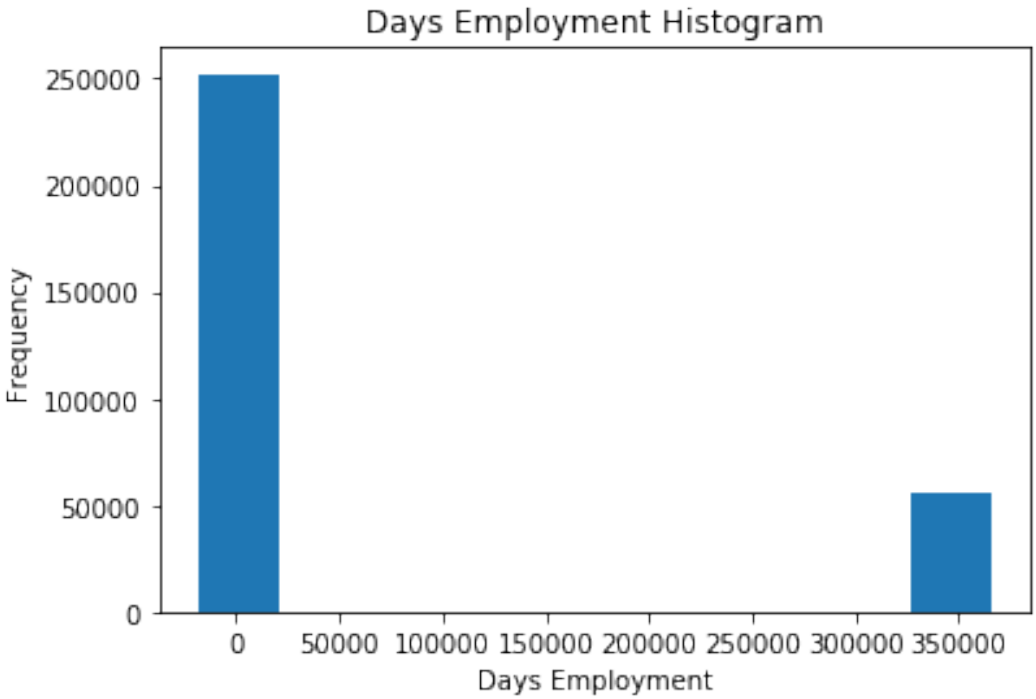
Those ages look reasonable. There are no outliers for the age on either the high or low end. We do the same check on the days of employment.

[33]:
```
# Check plausibility of days of employment
app_data['DAYS_EMPLOYED'].describe()
```

[33]:
```
count    307511.000000
mean      63815.045904
std      141275.766519
min      -17912.000000
25%       -2760.000000
50%       -1213.000000
75%        -289.000000
max      365243.000000
Name: DAYS_EMPLOYED, dtype: float64
```

The maximum value (besides being positive) is about 1000 years. Outlier needs to be dealt with.

[34]:
```
app_data['DAYS_EMPLOYED'].plot.hist(title = 'Days Employment Histogram');
plt.xlabel('Days Employment');
```



Just out of curiousity, let's subset the anomalous clients and see if they tend to have higher or low rates of default than the rest of the clients.

[35]:
```
anom = app_data[app_data['DAYS_EMPLOYED'] == 365243]
non_anom = app_data[app_data['DAYS_EMPLOYED'] != 365243]
print('The non-anomalies default on %0.2f%% of loans' % (100 *␣
 ↪non_anom['TARGET'].mean()))
```

```
print('The anomalies default on %0.2f%% of loans' % (100 * anom['TARGET'].
 ↪mean()))
print('There are %d anomalous days of employment' % len(anom))
```

```
The non-anomalies default on 8.66% of loans
The anomalies default on 5.40% of loans
There are 55374 anomalous days of employment
```

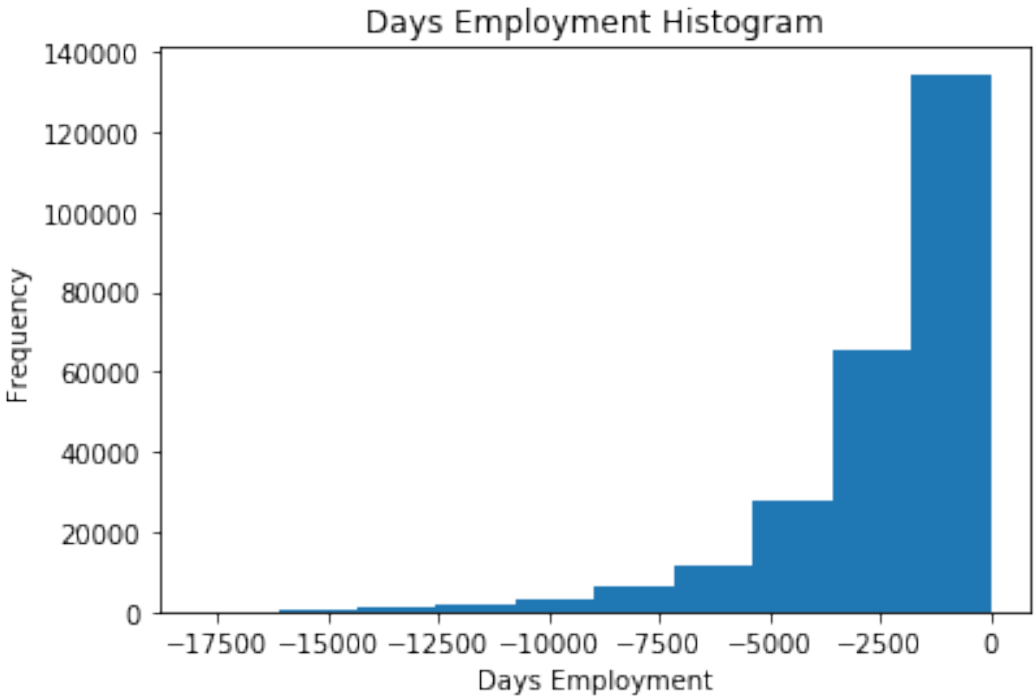
The result seems worth noting. It turns out that the anomalies have a lower rate of default.

Handling the anomalies depends on the exact situation, with no set rules. One of the safest approaches is just to set the anomalies to a missing value and then have them filled in (using Imputation) before machine learning. In this case, since all the anomalies have the exact same value, we want to fill them in with the same value in case all of these loans share something in common. The anomalous values seem to have some importance, so we want to tell the machine learning model if we did in fact fill in these values. As a solution, we will fill in the anomalous values with not a number (`np.nan`) and then create a new boolean column indicating whether or not the value was anomalous.

[36]:
```
# Create an anomalous flag column
app_data['DAYS_EMPLOYED_ANOM'] = app_data["DAYS_EMPLOYED"] == 365243

# Replace the anomalous values with nan
app_data['DAYS_EMPLOYED'].replace({365243: np.nan}, inplace = True)

app_data['DAYS_EMPLOYED'].plot.hist(title = 'Days Employment Histogram');
plt.xlabel('Days Employment');
```



[37]:
```
# Check plausibility of Days_employed after replacement of max value
app_data['DAYS_EMPLOYED'].describe()
```

[37]:
```
count    252137.000000
mean      -2384.169325
std        2338.360162
min      -17912.000000
25%       -3175.000000
50%       -1648.000000
75%        -767.000000
max           0.000000
Name: DAYS_EMPLOYED, dtype: float64
```

The distribution looks to be much more in line with what we would expect. The statistics are smaller than those of 'Days_Birth', which is in plausible, too. On top, we have created a new column to tell the model that these values were originally anomalous (because we will have to fill in the nans with some value, probably the median of the column). The other columns with `DAYS` in the dataframe look to be about what we expect with no obvious outliers.

3.4 Research questions and methods

3.4.1 1) What are the features with the highest impact on repayment?

3.4.2 2) Can you use a simple ML techniques ("Decision Tree") to predict loan repayment?

3.4.3 ad 1) What are the features with the highest impact on repayment?

3.4.4 Correlations

Now, we want to understand the data is by looking for correlations between the features and the target. We can calculate the Pearson correlation coefficient between every variable and the target using the `.corr` dataframe method.

The correlation coefficient is not the most advanced method to represent "relevance" of a feature, but it does give us an idea of possible relationships within the data. Some general interpretations of the absolute value of the correlation coefficent are:

- .00-.19 "very weak"
- .20-.39 "weak"
- .40-.59 "moderate"
- .60-.79 "strong"
- .80-1.0 "very strong"

[38]:
```
# Find correlations with the target and sort
correlations = app_data.corr()['TARGET'].sort_values()

# Display correlations
print('Most Positive Correlations:\n', correlations.tail(10))
print('\nMost Negative Correlations:\n', correlations.head(10))
```

```
Most Positive Correlations:
 REG_CITY_NOT_WORK_CITY         0.050994
DAYS_ID_PUBLISH                 0.051457
CODE_GENDER_M                   0.054713
DAYS_LAST_PHONE_CHANGE          0.055218
NAME_INCOME_TYPE_Working        0.057481
REGION_RATING_CLIENT            0.058899
REGION_RATING_CLIENT_W_CITY     0.060893
DAYS_EMPLOYED                   0.074958
DAYS_BIRTH                      0.078239
TARGET                          1.000000
Name: TARGET, dtype: float64

Most Negative Correlations:
 EXT_SOURCE_3                             -0.178919
EXT_SOURCE_2                             -0.160472
EXT_SOURCE_1                             -0.155317
NAME_EDUCATION_TYPE_Higher education     -0.056593
CODE_GENDER_F                            -0.054704
NAME_INCOME_TYPE_Pensioner               -0.046209
DAYS_EMPLOYED_ANOM                       -0.045987
ORGANIZATION_TYPE_XNA                    -0.045987
FLOORSMAX_AVG                            -0.044003
FLOORSMAX_MEDI                           -0.043768
Name: TARGET, dtype: float64
```

Let's take a look at some of more significant correlations: the `DAYS_BIRTH` is the most positive correlation. (except for `TARGET` because the correlation of a variable with itself is always 1) Looking at the documentation, `DAYS_BIRTH` is the age in days of the client at the time of the loan in negative days. The correlation is positive, but the value of this feature is actually negative, meaning that as the client gets older, they are less likely to default on their loan (ie the target == 0). That's a little confusing, so we will take the absolute value of the feature and then the correlation will be negative.

3.4.5 Effect of Age on Repayment

[40]:
```
# Find the correlation of the positive days since birth and target
app_data['DAYS_BIRTH'] = abs(app_data['DAYS_BIRTH'])
app_data['DAYS_BIRTH'].corr(app_data['TARGET'])
```

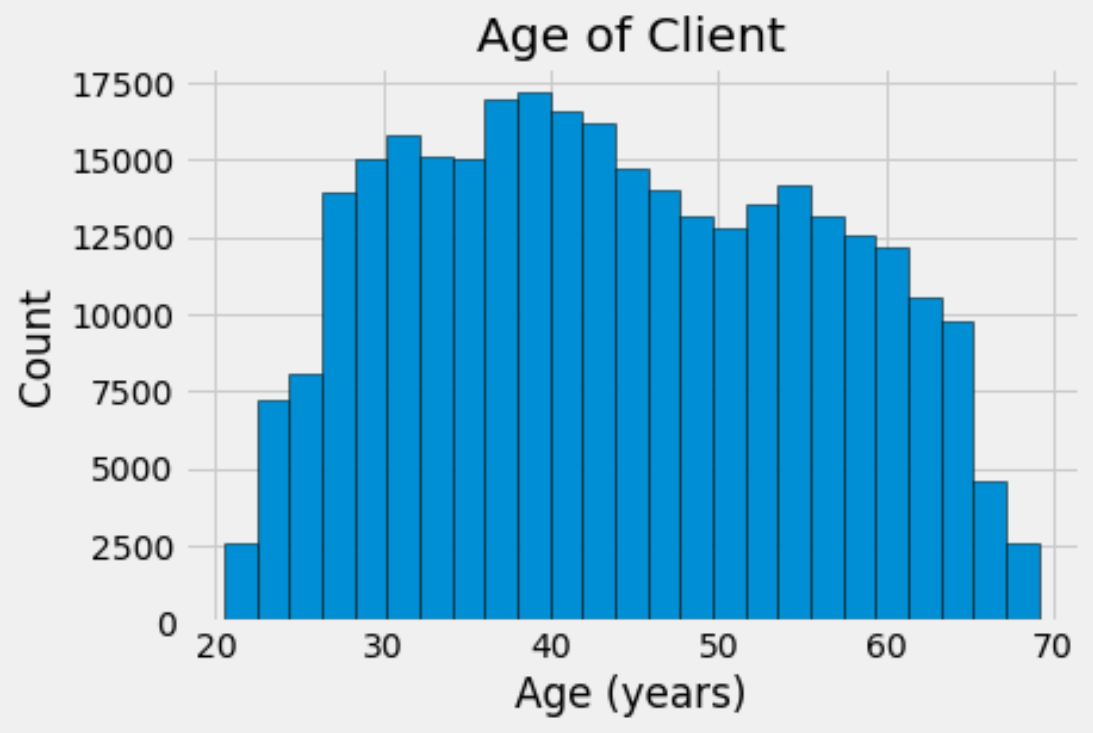
[40]:
```
-0.07823930830982712
```

As the client gets older, there is a negative linear relationship with the target meaning that as clients get older, they tend to repay their loans on time more often. Let's start looking at this variable. First, we can make a histogram of the age. We will put the x axis in years to make the plot a little more understandable.

[41]:
```
# Set the style of plots
plt.style.use('fivethirtyeight')
```

```
# Plot the distribution of ages in years
plt.hist(app_data['DAYS_BIRTH'] / 365, edgecolor = 'k', bins = 25)
plt.title('Age of Client'); plt.xlabel('Age (years)'); plt.ylabel('Count')
```

[41]:
```
Text(0, 0.5, 'Count')
```



By itself, the distribution of age does not tell us more than that there are no outliers as all the ages are reasonable. To visualize the effect of the age on the target, we will next make a kernel density estimation plot (KDE) colored by the value of the target. A kernel density estimate plot shows the distribution of a single variable and can be thought of as a smoothed histogram (it is created by computing a kernel, usually a Gaussian, at each data point and then averaging all the individual kernels to develop a single smooth curve). We will use the seaborn `kdeplot` for this graph.

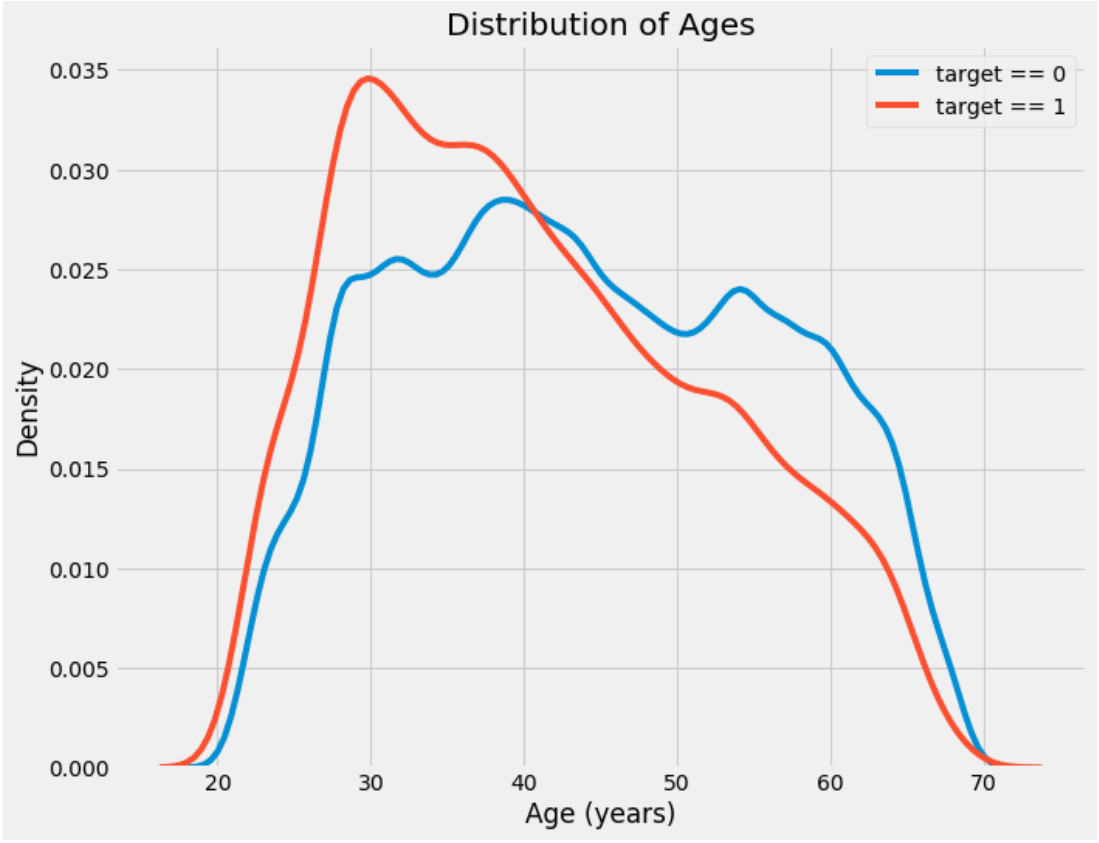
[42]:
```
plt.figure(figsize = (10, 8))

# KDE plot of loans that were repaid on time
sns.kdeplot(app_data.loc[app_data['TARGET'] == 0, 'DAYS_BIRTH'] / 365, label =␣
↪'target == 0')

# KDE plot of loans which were not repaid on time
sns.kdeplot(app_data.loc[app_data['TARGET'] == 1, 'DAYS_BIRTH'] / 365, label =␣
↪'target == 1')
```

```
# Labeling of plot
plt.xlabel('Age (years)'); plt.ylabel('Density'); plt.title('Distribution of␣
 ↪Ages')
```

[42]:
```
Text(0.5, 1.0, 'Distribution of Ages')
```



The target == 1 curve skews towards the younger end of the range. (Remember: target == 0, loan is unpaid. target == 1, loan is paid.) Although this is not a significant correlation (-0.07 correlation coefficient), this variable is likely going to be useful in a machine learning model because it does affect the target.

Let's look at this relationship in another way: average failure to repay loans by age bracket.

To make this graph, first we cut the age category into bins of 5 years each. Then, for each bin, we calculate the average value of the target, which tells us the ratio of loans that were not repaid in each age category.

[43]:
```
# Age information into a separate dataframe
age_data = app_data[['TARGET', 'DAYS_BIRTH']]
age_data['YEARS_BIRTH'] = age_data['DAYS_BIRTH'] / 365

# Bin the age data
age_data['YEARS_BINNED'] = pd.cut(age_data['YEARS_BIRTH'], bins = np.
 ↪linspace(20, 70, num = 11))
```

```
age_data.head()
```

[43]:

	TARGET	DAYS_BIRTH	YEARS_BIRTH	YEARS_BINNED
0	1	9461	25.920548	(25.0, 30.0]
1	0	16765	45.931507	(45.0, 50.0]
2	0	19046	52.180822	(50.0, 55.0]
3	0	19005	52.068493	(50.0, 55.0]
4	0	19932	54.608219	(50.0, 55.0]

[44]:
```
# Group by the bin and calculate averages
age_groups  = age_data.groupby('YEARS_BINNED').mean()
age_groups
```

[44]:

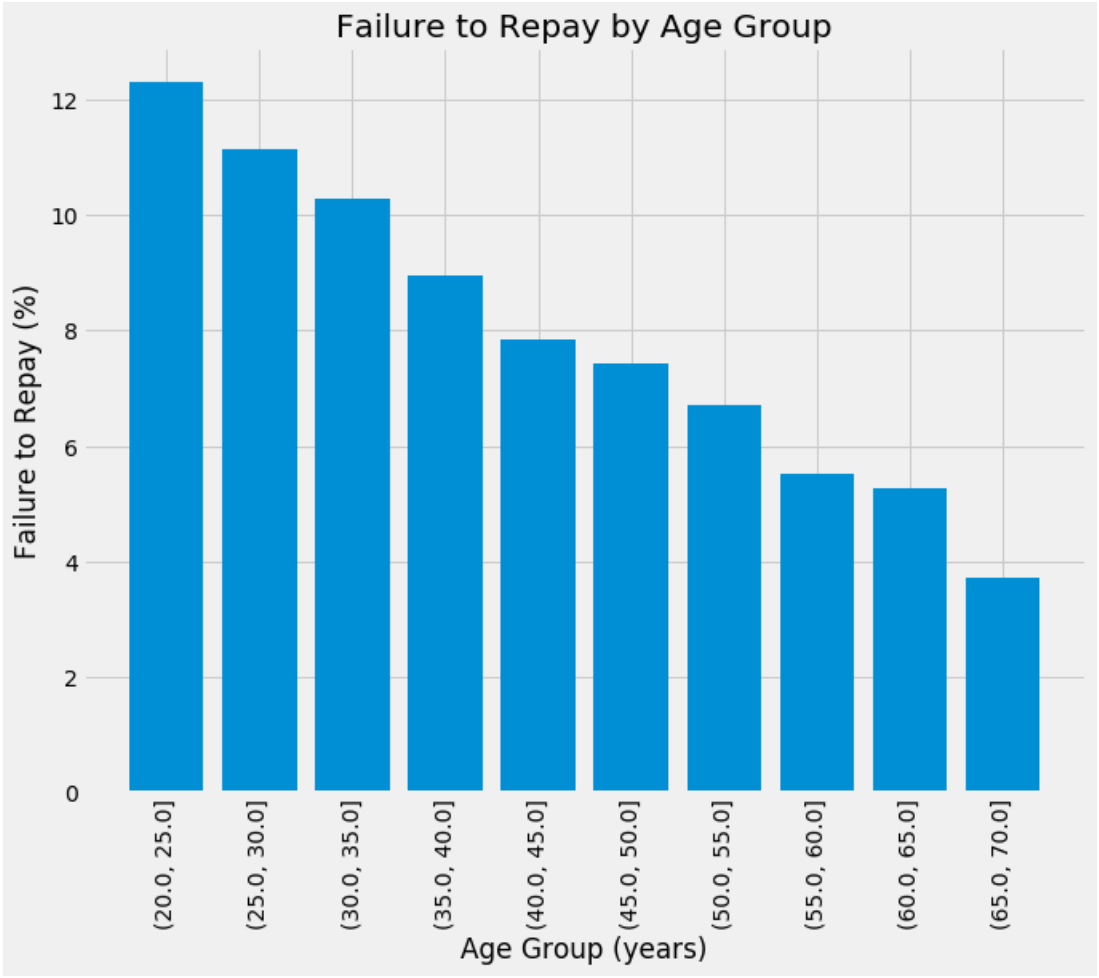
YEARS_BINNED	TARGET	DAYS_BIRTH	YEARS_BIRTH
(20.0, 25.0]	0.123036	8532.795625	23.377522
(25.0, 30.0]	0.111436	10155.219250	27.822518
(30.0, 35.0]	0.102814	11854.848377	32.479037
(35.0, 40.0]	0.089414	13707.908253	37.555913
(40.0, 45.0]	0.078491	15497.661233	42.459346
(45.0, 50.0]	0.074171	17323.900441	47.462741
(50.0, 55.0]	0.066968	19196.494791	52.593136
(55.0, 60.0]	0.055314	20984.262742	57.491131
(60.0, 65.0]	0.052737	22780.547460	62.412459
(65.0, 70.0]	0.037270	24292.614340	66.555108

[45]:
```
plt.figure(figsize = (10, 8))

# Graph the age bins and the average of the target as a bar plot
plt.bar(age_groups.index.astype(str), 100 * age_groups['TARGET'])

# Plot labeling
plt.xticks(rotation = 90); plt.xlabel('Age Group (years)'); plt.ylabel('Failure␣
 ↪to Repay (%)')
plt.title('Failure to Repay by Age Group')
```

[45]:
```
Text(0.5, 1.0, 'Failure to Repay by Age Group')
```



There is a clear trend: younger applicants are more likely to not repay the loan. The rate of failure to repay is above 10% for the youngest three age groups and beolow 5% for the oldest age group.

This is information that could be directly used by the bank: because younger clients are less likely to repay the loan, maybe they should be provided with more guidance or financial planning tips. This does not mean the bank should discriminate against younger clients, but it would be smart to take precautionary measures to help younger clients pay on time.

3.4.6 Exterior Sources

The 3 variables with the strongest negative correlations with the target are `EXT_SOURCE_1`, `EXT_SOURCE_2`, and `EXT_SOURCE_3`. According to the documentation, these features represent a "normalized score from external data source". I'm not sure what this exactly means, but it may be a cumulative sort of credit rating made using numerous sources of data.

Let's take a look at these variables.

First, we can show the correlations of the `EXT_SOURCE` features with the target and with each other including variable `DAYS_BIRTH`.

[46]:
```
# Extract the EXT_SOURCE variables and calculate correlations
ext_data = app_data[['TARGET', 'EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3',␣
↪'DAYS_BIRTH']]
ext_data_corrs = ext_data.corr()
ext_data_corrs
```

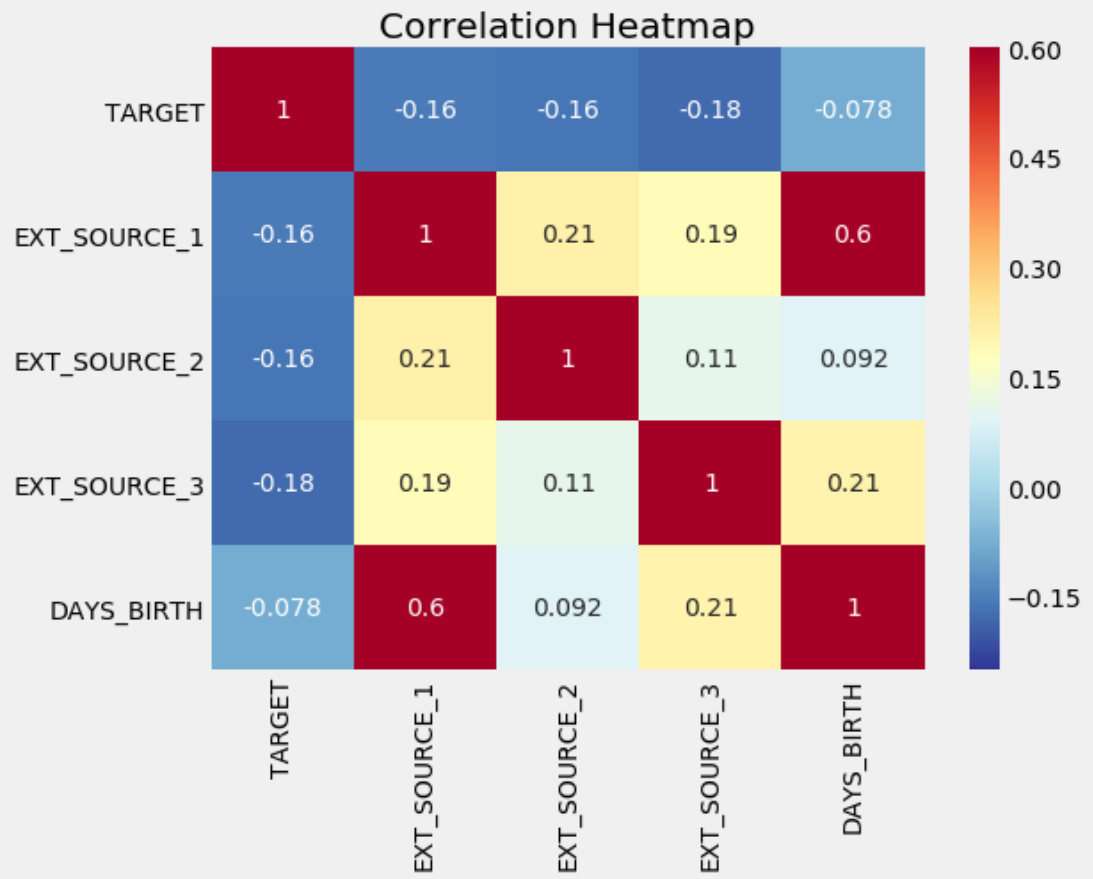
[46]:

	TARGET	EXT_SOURCE_1	EXT_SOURCE_2	EXT_SOURCE_3	DAYS_BIRTH
TARGET	1.000000	-0.155317	-0.160472	-0.178919	-0.078239
EXT_SOURCE_1	-0.155317	1.000000	0.213982	0.186846	0.600610
EXT_SOURCE_2	-0.160472	0.213982	1.000000	0.109167	0.091996
EXT_SOURCE_3	-0.178919	0.186846	0.109167	1.000000	0.205478
DAYS_BIRTH	-0.078239	0.600610	0.091996	0.205478	1.000000

[47]:
```
plt.figure(figsize = (8, 6))

# Heatmap of correlations
sns.heatmap(ext_data_corrs, cmap = plt.cm.RdYlBu_r, vmin = -0.25, annot = True,␣
↪vmax = 0.6)
plt.title('Correlation Heatmap')
```

[47]:
```
Text(0.5, 1, 'Correlation Heatmap')
```



All three EXT_SOURCE featureshave negative correlations with the target, indicating that as the value of the EXT_SOURCE increases, the client is more likely to repay the loan.

We can also see that DAYS_BIRTH is positively correlated with EXT_SOURCE_1 indicating that maybe one of the factors in this score is the client age.

Next we can look at the distribution of each of these features colored by the value of the target. This will let us visualize the effect of this variable on the target.

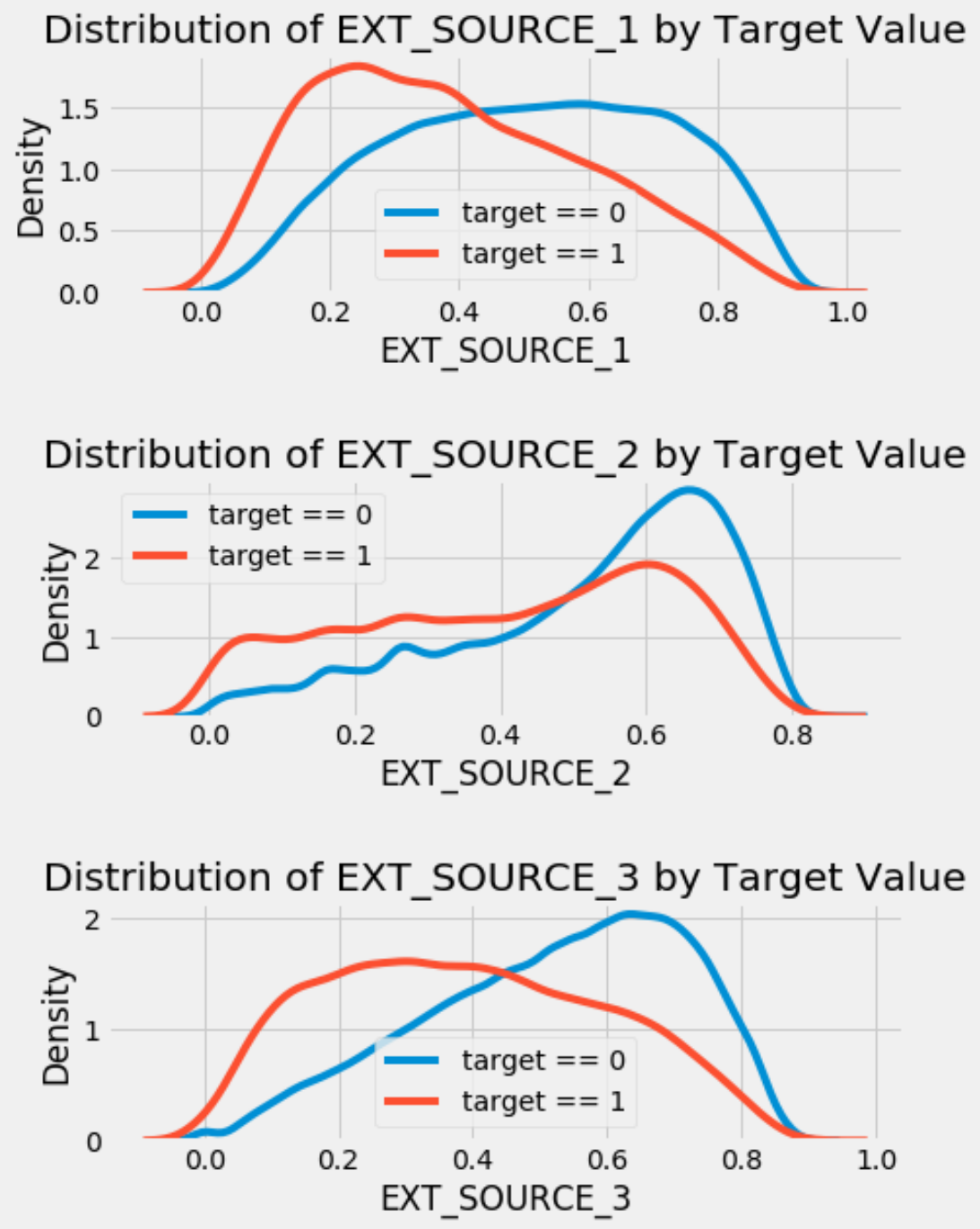
[48]:
```
plt.figure(figsize = (7, 9))

# iterate through the sources
for i, source in enumerate(['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3']):

    # create a new subplot for each source
    plt.subplot(3, 1, i + 1)
    # plot repaid loans
    sns.kdeplot(app_data.loc[app_data['TARGET'] == 0, source], label = 'target␣
 ↪== 0')
    # plot loans that were not repaid
    sns.kdeplot(app_data.loc[app_data['TARGET'] == 1, source], label = 'target␣
 ↪== 1')

    # Label the plots
    plt.title('Distribution of %s by Target Value' % source)
    plt.xlabel('%s' % source); plt.ylabel('Density');

plt.tight_layout(h_pad = 2.5)
```



`EXT_SOURCE_3` displays the greatest difference between the values of the target. We observe that this feature has a relationship to the likelihood of an applicant to repay a loan. The relationship is not very strong (in fact they are all considered very weak, but these variables will still be useful for a machine learning model to predict whether or not an applicant will repay a loan on time.

4 Baseline Machine Learning

For a naive baseline, we could guess the same value for all examples on the testing set. We are asked to predict the probability of not repaying the loan, so if we are entirely unsure, we would guess 0.5 for all observations on the test set. This will get us a Reciever Operating Characteristic Area Under the Curve (AUC ROC) of 0.5 in the competition (random guessing on a classification task will score a 0.5).

Since we already know what score we are going to get, we don't really need to make a naive baseline guess. Let's use a slightly more sophisticated model for our actual baseline: Decision Tree.

4.1 Decision Tree

To get a baseline, we will use all of the features after encoding the categorical variables. We will preprocess the data by filling in the missing values (imputation) and normalizing the range of the features (feature scaling). The following code performs both of these preprocessing steps.

[49]:
```
from sklearn.preprocessing import MinMaxScaler, Imputer

# Drop the target from the data
if 'TARGET' in app_data:
    train = app_data.drop(columns = ['TARGET'])
else:
    train = app_data.copy()

# Feature names
features = list(train.columns)

# Median imputation of missing values
imputer = Imputer(strategy = 'median')

# Scale each feature to 0-1
scaler = MinMaxScaler(feature_range = (0, 1))

# Fit on the data
imputer.fit(train)

# Transform data
train = imputer.transform(train)

# Repeat with the scaler
scaler.fit(train)
train = scaler.transform(train)

print('Data shape: ', train.shape)
```

```
Data shape:  (307511, 243)
```

4.2 Prepare Data for a classification task

[50]:
```
# sklearn libraries for decision tree classifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

4.2.1 Store 'Target' in Y

[51]:
```
y=app_data[['TARGET']].copy()
y.head()
```

[51]:
```
   TARGET
0       1
1       0
2       0
3       0
4       0
```

- Use train data to predict 'Target'

[53]:
```
X = train.copy()
```

[54]:
```
y.columns
```

[54]:
```
Index(['TARGET'], dtype='object')
```

4.3 Perform Test and Train Split

[55]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,␣
↪random_state=324)
```

[56]:
```
type(X_train)
type(X_test)
type(y_train)
type(y_test)
# X_train.head()
y_train.describe()
```

[56]:
```
              TARGET
count  206032.000000
mean        0.080337
std         0.271815
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000
```

[71]:
```
pd.DataFrame(X_train, columns=features).head()
```

[71]:

	SK_ID_CURR	NAME_CONTRACT_TYPE	FLAG_OWN_CAR	FLAG_OWN_REALTY	\
0	0.406571	0.0	0.0	1.0	
1	0.561960	0.0	0.0	1.0	
2	0.380606	1.0	0.0	1.0	
3	0.880837	0.0	0.0	1.0	
4	0.530446	0.0	0.0	1.0	

	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE	\
0	0.000000	0.000935	0.070763	0.055827	0.060606	
1	0.000000	0.000358	0.174290	0.148596	0.161616	
2	0.157895	0.000242	0.033708	0.028800	0.034792	
3	0.000000	0.001320	0.142349	0.091909	0.122334	
4	0.000000	0.000742	0.216854	0.098368	0.217733	

	REGION_POPULATION_RELATIVE	...	WALLSMATERIAL_MODE_Block	\
0	0.039215	...	0.0	
1	0.491595	...	0.0	
2	0.429796	...	0.0	
3	0.421848	...	0.0	
4	0.257000	...	0.0	

	WALLSMATERIAL_MODE_Mixed	WALLSMATERIAL_MODE_Monolithic	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	WALLSMATERIAL_MODE_Others	WALLSMATERIAL_MODE_Panel	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	WALLSMATERIAL_MODE_Stone, brick	WALLSMATERIAL_MODE_Wooden	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	1.0	0.0	
4	1.0	0.0	

	EMERGENCYSTATE_MODE_No	EMERGENCYSTATE_MODE_Yes	DAYS_EMPLOYED_ANOM
0	0.0	0.0	1.0
1	0.0	0.0	1.0

```
2                   0.0                   0.0                   0.0
3                   1.0                   0.0                   0.0
4                   1.0                   0.0                   0.0

[5 rows x 243 columns]
```

4.4 Train model - (Nodes 10)

[144]:
```
Credit_classifier = DecisionTreeClassifier(max_leaf_nodes=3, random_state=0)
Credit_classifier.fit(X_train, y_train)
```

[144]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=3,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=0, splitter='best')
```

[145]:
```
type(Credit_classifier)
```

[145]:
```
sklearn.tree.tree.DecisionTreeClassifier
```

4.5 Predict on Test Set - (Nodes 10)

[146]:
```
predictions = Credit_classifier.predict(X_test)
```

[147]:
```
predictions[:10]
```

[147]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[148]:
```
y_test['TARGET'][:10]
```

[148]:
```
1043      0
168655    0
279210    0
293208    0
301210    0
290228    0
186392    0
35461     0
123879    0
230136    0
Name: TARGET, dtype: int64
```

4.6 Measure Accuracy of the Classifier - (Nodes 10)

[149]:
```
accuracy_score(y_true = y_test, y_pred = predictions)
```

[149]:
```
0.9184757437499385
```

[150]:
```
from sklearn import metrics
```

[139]:
```
fpr, tpr, thresholds = metrics.roc_curve(predictions, y_test['TARGET'],␣
↪pos_label=2)
```

[151]:
```
fpr
```

[151]:
```
array([0.        , 0.08152426, 1.        ])
```

[152]:
```
tpr
```

[152]:
```
array([nan, nan, nan])
```

[153]:
```
thresholds
```

[153]:
```
array([2, 1, 0])
```

4.7 Calculate ROC Score

[154]:
```
import numpy as np
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, predictions)
```

[154]:
```
0.5
```