Loan Repayment Predcting

The objective of this task is to use historical loan application data to predict whether or not an applicant will be able to repay a loan. This is a 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
- **Classification**: The label is a binary variable, 0 (will repay loan on time), 1 (will have difficulty repaying loan)

Import Libraries

In [1]:
```
# 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

# 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
```

Read in Data

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

In [2]:
```
# List files available
print(os.listdir("../input/"))
```

```
['sample_submission.csv', 'bureau_balance.csv', 'POS_CASH_balance.csv', 'application_train.csv', 'HomeCredit_
columns_description.csv', 'application_test.csv', 'previous_application.csv', 'credit_card_balance.csv', 'ins
tallments_payments.csv', 'bureau.csv']
```

In [3]:
```
# Training data
app_train = pd.read_csv('../input/application_train.csv')
print('Training data shape: ', app_train.shape)
app_train.head()
```

```
Training data shape:  (307511, 122)
```

Out[3]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INC
0	100002	1	Cash loans	M	N	Y	0	
1	100003	0	Cash loans	F	N	N	0	
2	100004	0	Revolving loans	M	Y	Y	0	
3	100006	0	Cash loans	F	N	Y	0	
4	100007	0	Cash loans	M	N	Y	0	

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

In [4]:
```
# Testing data features
app_test = pd.read_csv('../input/application_test.csv')
print('Testing data shape: ', app_test.shape)
app_test.head()
```

```
Testing data shape:  (48744, 121)
```

Out[4]:

	SK_ID_CURR	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_TOTA
0	100001	Cash loans	F	N	Y	0	135000
1	100005	Cash loans	M	N	Y	0	99000
2	100013	Cash loans	M	Y	Y	0	202500
3	100028	Cash loans	F	N	Y	2	315000
4	100038	Cash loans	M	Y	N	1	180000

The test set is considerably smaller and lacks a `TARGET` column.

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is an open-ended process involving the calculation of statistics and the creation of figures to identify trends, anomalies, patterns, or relationships within the data. The primary aim of EDA is to extract insights from our data. It typically begins with a high-level overview and gradually delves into specific areas that exhibit intriguing characteristics within the dataset. These findings might be inherently interesting or serve as crucial inputs informing our modeling decisions, aiding in the selection of features to incorporate.

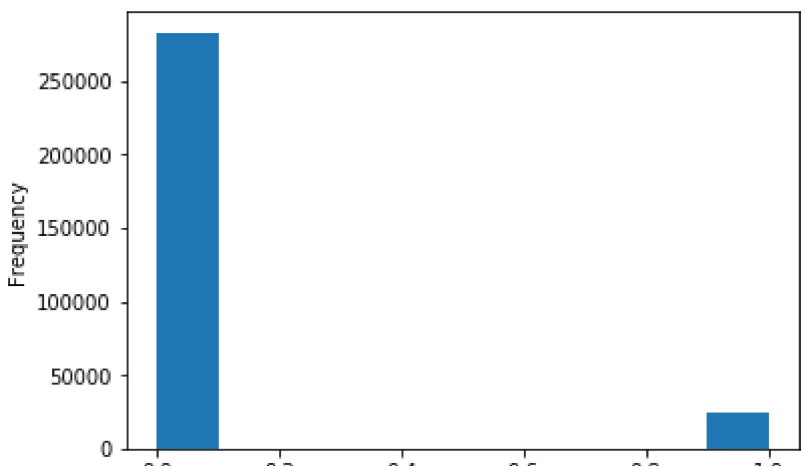
Examine the Distribution of the Target Column

The target represents what we aim to predict: a binary outcome where '0' signifies a loan repaid on time, while '1' indicates payment difficulties faced by the client. Initially, it's beneficial to analyze the distribution of loans across these categories.

In [5]:
```
app_train['TARGET'].value_counts()
```

Out[5]:
```
0    282686
1     24825
Name: TARGET, dtype: int64
```

In [6]:
```
app_train['TARGET'].astype(int).plot.hist();
```



From this information, we can observe that this is an imbalanced class problem. There are far more loans that were repaid on time than loans that were not repaid. Once we delve into more sophisticated machine learning models, we can weight the classes by their representation in the data to address this imbalance.

Examine Missing Values

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

In [7]:
```
# Function to calculate missing values by column
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
```

In [8]:
```
# Missing values statistics
missing_values = missing_values_table(app_train)
missing_values.head(20)
```

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

Out[8]:

	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
FLOORSMIN_MODE	208642	67.8
FLOORSMIN_MEDI	208642	67.8
FLOORSMIN_AVG	208642	67.8
YEARS_BUILD_MODE	204488	66.5
YEARS_BUILD_MEDI	204488	66.5
YEARS_BUILD_AVG	204488	66.5
OWN_CAR_AGE	202929	66.0
LANDAREA_AVG	182590	59.4
LANDAREA_MEDI	182590	59.4
LANDAREA_MODE	182590	59.4

When it's time to construct our machine learning models, we'll need to fill in these missing values, a process known as imputation. In subsequent phases, we'll utilize models like XGBoost, capable of handling missing values without requiring imputation. An alternative approach could involve discarding columns with a high percentage of missing values, although it's impossible to predict whether these columns will prove beneficial to our model in advance. As a result, we'll retain all columns for the time being.

Column Types

Let's examine the count of columns based on their data types. Columns of 'int64' and 'float64' represent numeric variables, which can be either discrete or continuous. On the other hand, 'object' columns consist of strings and represent categorical features.

In [9]:
```
# Number of each type of column
app_train.dtypes.value_counts()
```

Out[9]:
```
float64    65
int64      41
object     16
dtype: int64
```

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

In [10]:
```
# Number of unique classes in each object column
app_train.select_dtypes('object').apply(pd.Series.nunique, axis = 0)
```

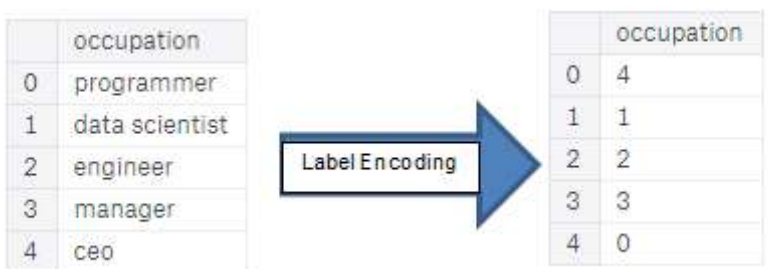
Out[10]:
```
NAME_CONTRACT_TYPE             2
CODE_GENDER                    3
FLAG_OWN_CAR                   2
FLAG_OWN_REALTY                2
NAME_TYPE_SUITE                7
NAME_INCOME_TYPE               8
NAME_EDUCATION_TYPE            5
NAME_FAMILY_STATUS             6
NAME_HOUSING_TYPE              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
```

The majority of categorical variables exhibit a relatively small number of unique entries. We'll need to determine an approach for handling these categorical variables.

Encoding Categorical Variables

Before proceeding, we must address the issue of categorical variables. Most machine learning models, apart from certain exceptions like LightGBM, cannot directly process categorical variables. Hence, we need to encode these variables into numerical representations before feeding them into the model. There are two primary methods to accomplish this task:

- Label encoding: assign each unique category in a categorical variable with an integer. No new columns are created. An example is shown below



- 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 issue with label encoding lies in the arbitrary ordering it assigns to categories. The numerical values attributed to each category lack inherent meaning and can vary if the process is repeated, leading to arbitrary integer assignments. Consequently, label encoding might mislead the model by inadvertently assigning weights based on these arbitrary values, compromising the model's interpretation. While label encoding suffices for binary variables like 'Male/Female,' it becomes unreliable for multiple categories, making one-hot encoding a more secure choice.

Though models exist that handle label-encoded categorical variables effectively, the consensus leans towards one-hot encoding for variables with numerous classes due to its avoidance of arbitrary value assignments. However, one-hot encoding can exponentially increase the feature space, especially with many categories. To manage this, post-one-hot encoding, dimensionality reduction techniques such as PCA can be applied to condense dimensions while preserving critical information.

In this notebook, we'll implement Label Encoding for categorical variables with only two categories and One-Hot Encoding for those with more than two categories. This strategy might evolve as the project progresses. Currently, we aim to assess its effectiveness. Additionally, while this notebook won't include dimensionality reduction, future iterations will explore this avenue.

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.

In [11]:
```
# Create a label encoder object
le = LabelEncoder()
le_count = 0

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

In [12]:
```
# one-hot encoding of categorical variables
app_train = pd.get_dummies(app_train)
app_test = pd.get_dummies(app_test)

print('Training Features shape: ', app_train.shape)
print('Testing Features shape: ', app_test.shape)
```

```
Training Features shape:  (307511, 243)
Testing Features shape:  (48744, 239)
```

Aligning Training and Testing Data

For both the training and testing data, it's crucial to maintain identical features (columns). Due to one-hot encoding, the training data now possesses additional columns stemming from categorical variables with categories absent in the testing data. Aligning the dataframes becomes necessary to eliminate columns in the training data not present in the testing set. Initially, we extract the target column from the training data as it's absent in the testing set but crucial for our analysis. During alignment, it's imperative to specify 'axis = 1' to synchronize the dataframes based on their columns rather than their rows.

In [13]:
```
train_labels = app_train['TARGET']

# Align the training and testing data, keep only columns present in both dataframes
app_train, app_test = app_train.align(app_test, join = 'inner', axis = 1)

# Add the target back in
app_train['TARGET'] = train_labels

print('Training Features shape: ', app_train.shape)
print('Testing Features shape: ', app_test.shape)
```

```
Training Features shape:  (307511, 240)
Testing Features shape:  (48744, 239)
```

Ensuring identical features in both the training and testing datasets is crucial for machine learning. However, one-hot encoding has substantially increased the number of features. Eventually, we may consider employing dimensionality reduction techniques to trim irrelevant features and diminish the dataset sizes.

Anomalies

One problem we need to watch out for during EDA is anomalies within the data. These anomalies might stem from mis-typed numbers, errors in measuring equipment, or they could represent valid but extreme measurements. One quantitative method to detect anomalies involves examining the statistics of a column using the describe method. The numbers in the DAYS_BIRTH column are negative since they are recorded relative to the current loan application. To view these statistics in years, we can multiply by -1 and divide by the number of days in a year:

In [14]:
```
(app_train['DAYS_BIRTH'] / -365).describe()
```

Out[14]:
```
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
```

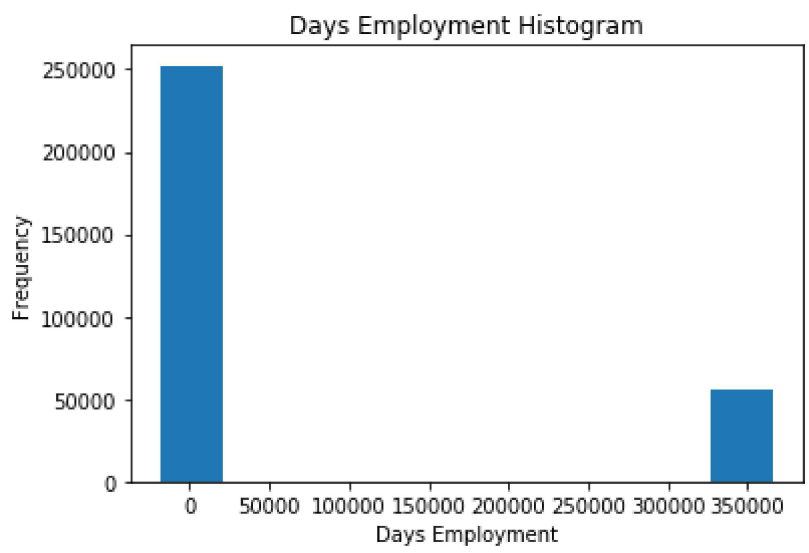
Those ages look reasonable. There are no outliers for the age on either the high or low end. How about the days of employment?

In [15]:
```
app_train['DAYS_EMPLOYED'].describe()
```

Out[15]:
```
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
```

That doesn't look right. The maximum value (besides being positive) is about 1000 years.

In [16]:
```
app_train['DAYS_EMPLOYED'].plot.hist(title = 'Days Employment Histogram');
plt.xlabel('Days Employment');
```



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.

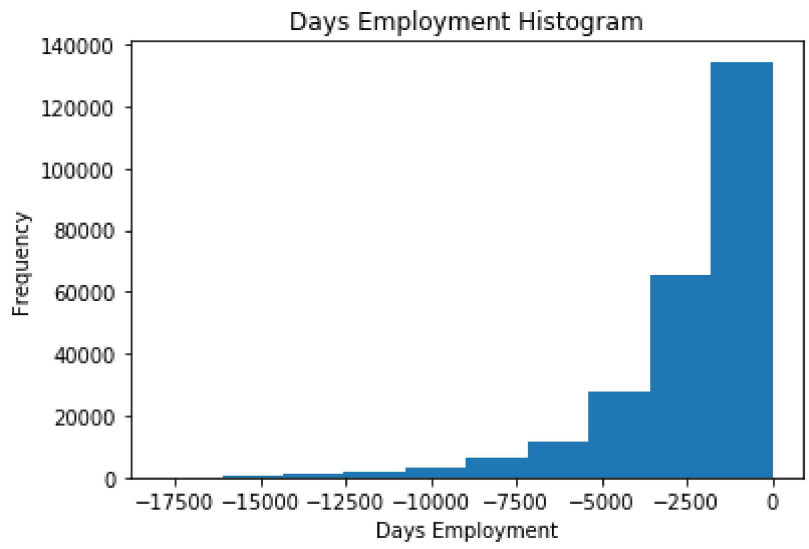
In [17]:
```
anom = app_train[app_train['DAYS_EMPLOYED'] == 365243]
non_anom = app_train[app_train['DAYS_EMPLOYED'] != 365243]
print('The non-anomalies default on %0.2f%% of loans' % (100 * non_anom['TARGET'].mean()))
print('The anomalies default on %0.2f%% of loans' % (100 * anom['TARGET'].mean()))
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
```

It appears that anomalies exhibit a lower default rate.

Addressing anomalies depends on the specific scenario and lacks fixed rules. One of the safest approaches involves marking anomalies as missing values and subsequently filling them in (using Imputation) before applying machine learning techniques. In this instance, since all anomalies share the same value, filling them with a consistent value could uncover any shared characteristics among these loans. As these anomalous values seem to carry some significance, we aim to signal to the machine learning model that these values have been filled. To achieve this, we'll replace the anomalous values with 'not a number' (np.nan) and then introduce a new boolean column indicating whether the value was initially anomalous.

In [18]:
```
# Create an anomalous flag column
app_train['DAYS_EMPLOYED_ANOM'] = app_train["DAYS_EMPLOYED"] == 365243

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

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



The distribution now appears more aligned with our expectations, and we've introduced a new column to indicate to the model that these values were initially anomalous. This information will be crucial as we'll likely need to fill the NaNs with a value, probably the median of the column.

Other columns in the dataframe containing 'DAYS' seem to align with our expectations, showing no apparent outliers.

It's crucial to note that any alterations made to the training data should also be applied to the testing data. Hence, let's ensure we create the new column and replace the existing column values with np.nan in the testing data as well.

In [19]:
```
app_test['DAYS_EMPLOYED_ANOM'] = app_test["DAYS_EMPLOYED"] == 365243
app_test["DAYS_EMPLOYED"].replace({365243: np.nan}, inplace = True)

print('There are %d anomalies in the test data out of %d entries' % (app_test["DAYS_EMPLOYED_ANOM"].sum(), len
```

```
There are 9274 anomalies in the test data out of 48744 entries
```

Correlations

Now that we've addressed the categorical variables and outliers, let's proceed with our exploratory data analysis (EDA). One approach to understanding the data involves examining correlations between features and the target variable. We can compute the Pearson correlation coefficient between each variable and the target using the .corr method on the dataframe.

The correlation coefficient isn't the sole indicator of a feature's relevance, but it does offer insights into potential relationships within the data. Broad interpretations based on the absolute value of the correlation coefficient include:

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

In [20]:
```
# Find correlations with the target and sort
correlations = app_train.corr()['TARGET'].sort_values()

# Display correlations
print('Most Positive Correlations:\n', correlations.tail(15))
print('\nMost Negative Correlations:\n', correlations.head(15))
```

```
Most Positive Correlations:
 OCCUPATION_TYPE_Laborers                              0.043019
FLAG_DOCUMENT_3                                       0.044346
REG_CITY_NOT_LIVE_CITY                                0.044395
FLAG_EMP_PHONE                                        0.045982
NAME_EDUCATION_TYPE_Secondary / secondary special     0.049824
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
FLOORSMAX_MODE                         -0.043226
EMERGENCYSTATE_MODE_No                 -0.042201
HOUSETYPE_MODE_block of flats          -0.040594
AMT_GOODS_PRICE                        -0.039645
REGION_POPULATION_RELATIVE             -0.037227
Name: TARGET, dtype: float64
```

Among the most noteworthy correlations is with DAYS_BIRTH, exhibiting the strongest positive correlation, disregarding the TARGET variable since its correlation with itself is always 1. According to the documentation, DAYS_BIRTH represents the client's age in negative days at the time of the loan (a rather peculiar format!). Although the correlation appears positive, the feature's values are actually negative, suggesting that as the client ages, they are less likely to default on their loan (i.e., TARGET == 0).

To alleviate confusion, we'll take the absolute value of the feature, transforming it into a negative correlation. This adjustment should clarify that as the client's age increases, their likelihood of loan default decreases.

Effect of Age on Repayment

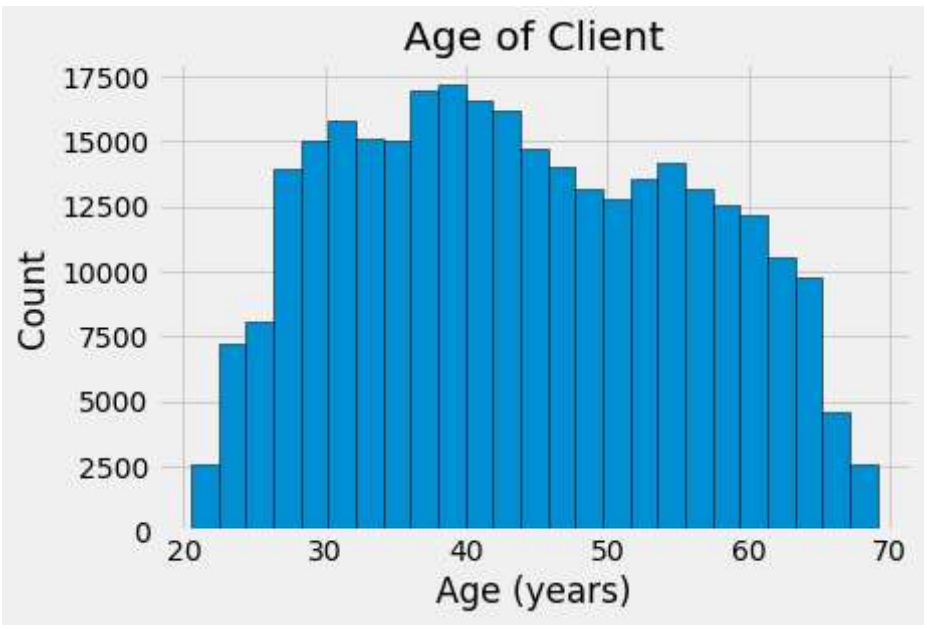
In [21]:
```
# Find the correlation of the positive days since birth and target
app_train['DAYS_BIRTH'] = abs(app_train['DAYS_BIRTH'])
app_train['DAYS_BIRTH'].corr(app_train['TARGET'])
```

Out[21]: -0.07823930830982694

As clients age, a negative linear relationship emerges with the target, indicating that older clients tend to exhibit a higher likelihood of repaying their loans on time.

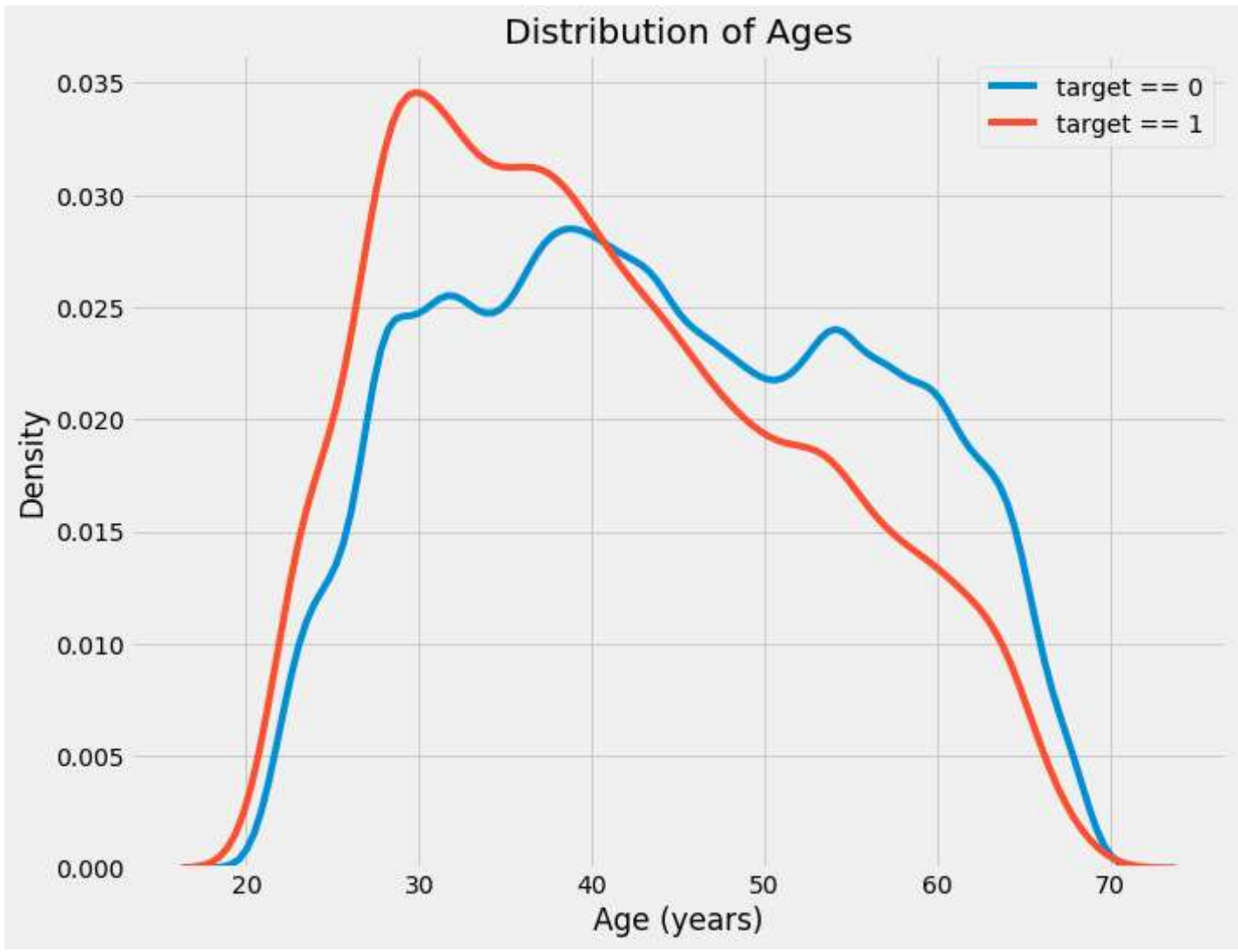
To better visualize this variable, let's create a histogram of age, adjusting the x-axis to display ages in years for a more intuitive plot.

In [22]:
```
# Set the style of plots
plt.style.use('fivethirtyeight')

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



Examining the age distribution alone doesn't yield substantial insights, although it confirms the absence of outliers, with all ages falling within reasonable bounds. To better visualize the impact of age on the target variable, our next step involves creating a kernel density estimation plot (KDE). This type of plot illustrates the distribution of a single variable and can be likened to a smoothed histogram. The KDE plot is generated by computing a kernel, typically Gaussian, at each data point and then averaging all individual kernels to produce a single smooth curve. We'll utilize seaborn's kdeplot to craft this graph, color-coding it by the target variable for clearer interpretation.

In [23]:
```
plt.figure(figsize = (10, 8))

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

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

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



The KDE plot reveals a tendency for the target == 1 curve to skew toward the younger end of the age range. Despite its relatively modest correlation coefficient of -0.07, this variable is likely to hold significance in a machine learning model due to its impact on the target variable. Now, let's explore this relationship differently: by examining the average failure to repay loans across age brackets.

To construct this graph, we'll segment the age variable into bins of 5 years each. Within each bin, we'll calculate the average value of the target, providing us with the proportion of loans not repaid in respective age categories.

In [24]:
```
# Age information into a separate dataframe
age_data = app_train[['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(10)
```

Out[24]:

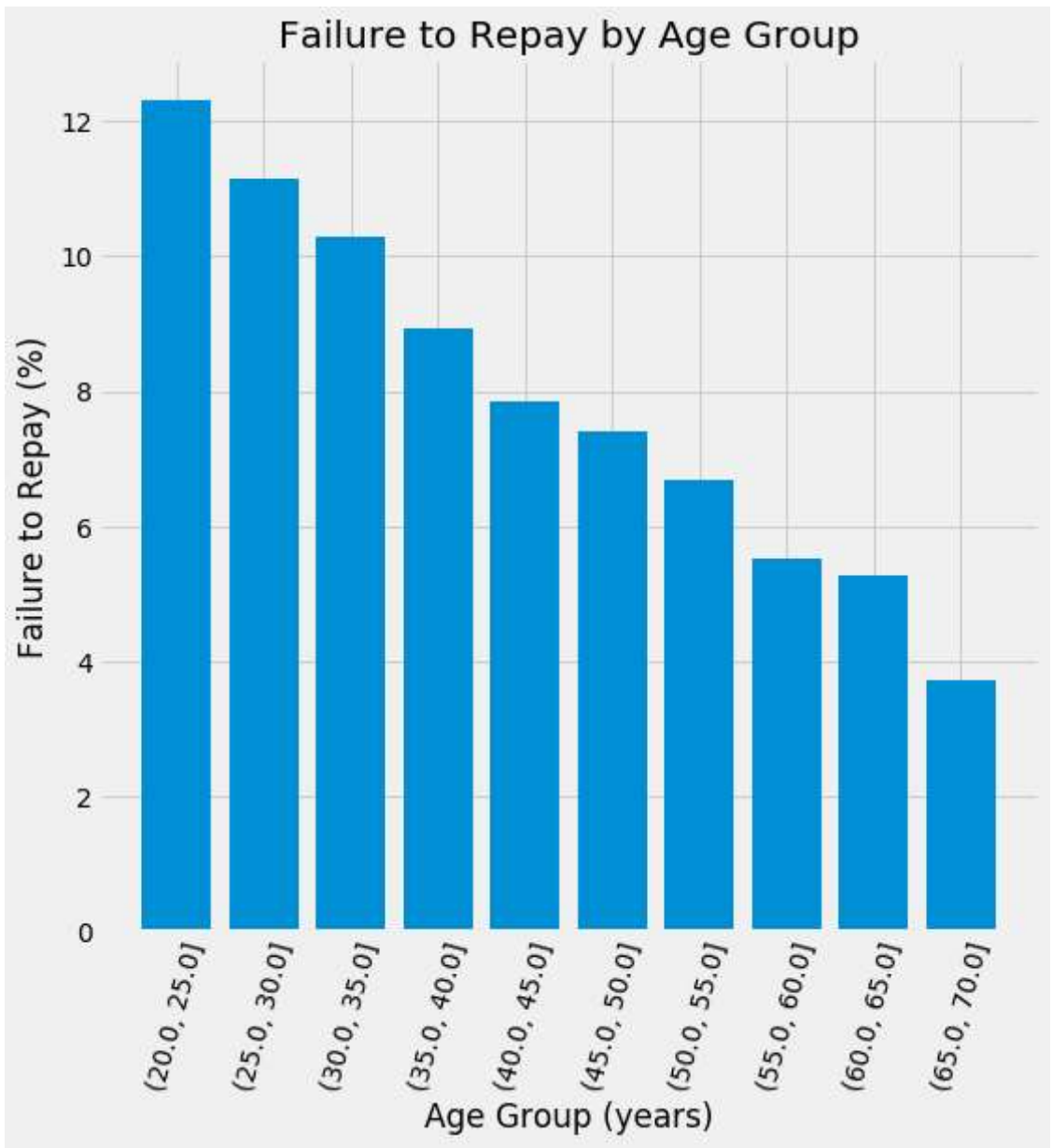
	TARGET	DAYS_BIRTH	YEARS_BIRTH	YEARS_BINNED
0	1	9461	25.920548	(25.0, 30.0]
1	0	16765	45.931507	(45.0, 50.0]
2	0	19046	52.180822	(50.0, 55.0]
3	0	19005	52.068493	(50.0, 55.0]
4	0	19932	54.608219	(50.0, 55.0]
5	0	16941	46.413699	(45.0, 50.0]
6	0	13778	37.747945	(35.0, 40.0]
7	0	18850	51.643836	(50.0, 55.0]
8	0	20099	55.065753	(55.0, 60.0]
9	0	14469	39.641096	(35.0, 40.0]

In [25]:
```
# Group by the bin and calculate averages
age_groups  = age_data.groupby('YEARS_BINNED').mean()
age_groups
```

Out[25]:

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

In [26]:
```
plt.figure(figsize = (8, 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 = 75); plt.xlabel('Age Group (years)'); plt.ylabel('Failure to Repay (%)')
plt.title('Failure to Repay by Age Group');
```



There's a noticeable trend: younger applicants exhibit a higher likelihood of loan repayment failure! The repayment failure rate surpasses 10% for the three youngest age groups, while it remains below 5% for the oldest age group.

This information could directly benefit the bank: as younger clients are less likely to repay loans, offering additional guidance or financial planning tips might prove beneficial. It's essential to note that this doesn't imply the bank should discriminate against younger clients. However, implementing precautionary measures to assist younger clients in timely repayments could be a wise approach.

Exterior Sources

The three variables exhibiting the strongest negative correlations with the target are EXT_SOURCE_1, EXT_SOURCE_2, and EXT_SOURCE_3. As per the documentation, these features represent a 'normalized score from an external data source.' Although the exact meaning is unclear, it might indicate a cumulative credit rating derived from various data sources.

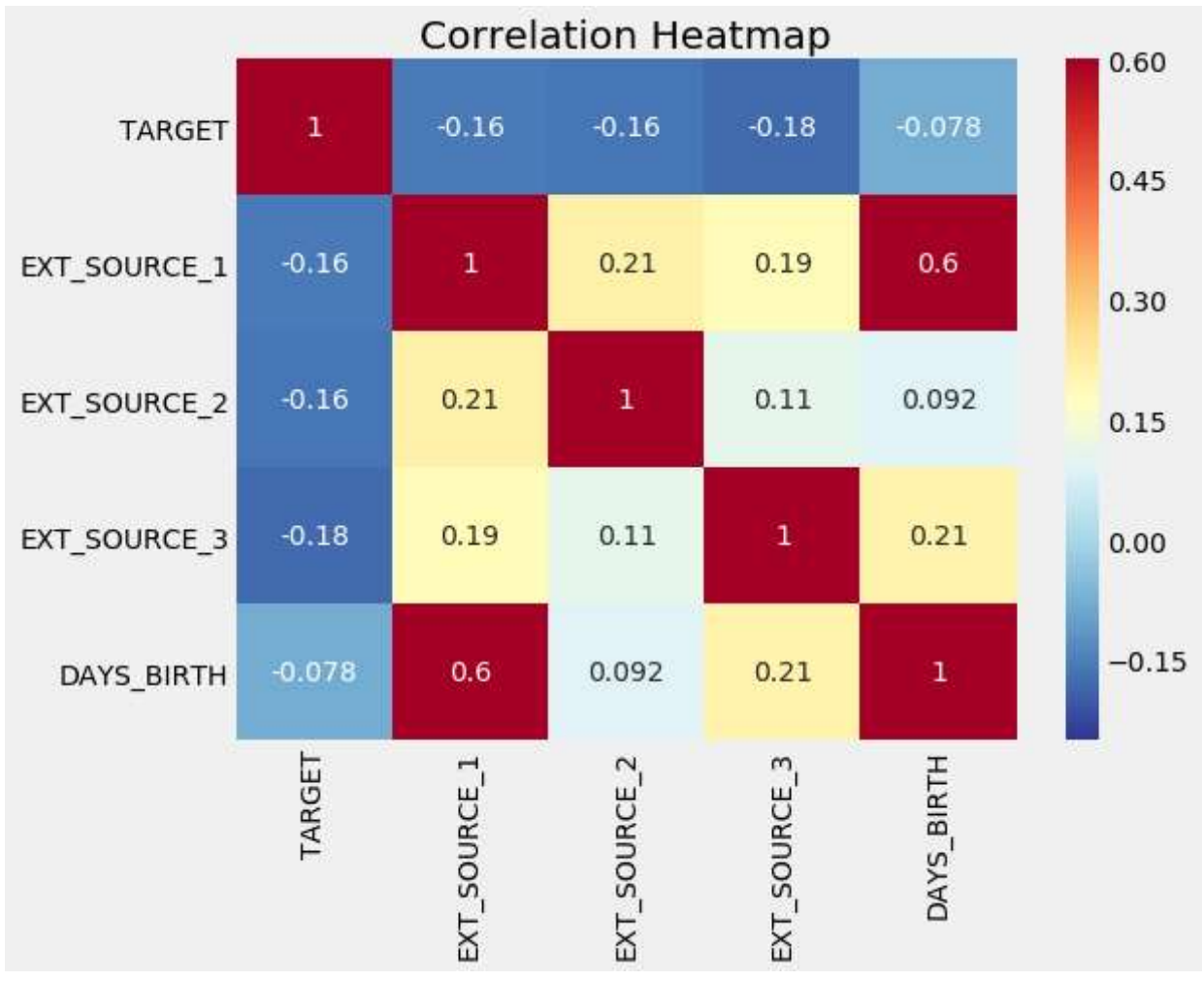
Now, let's delve into these variables. Firstly, we'll display the correlations of the EXT_SOURCE features with the target and among themselves.

In [27]:
```
# Extract the EXT_SOURCE variables and show correlations
ext_data = app_train[['TARGET', 'EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH']]
ext_data_corrs = ext_data.corr()
ext_data_corrs
```

Out[27]:

	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

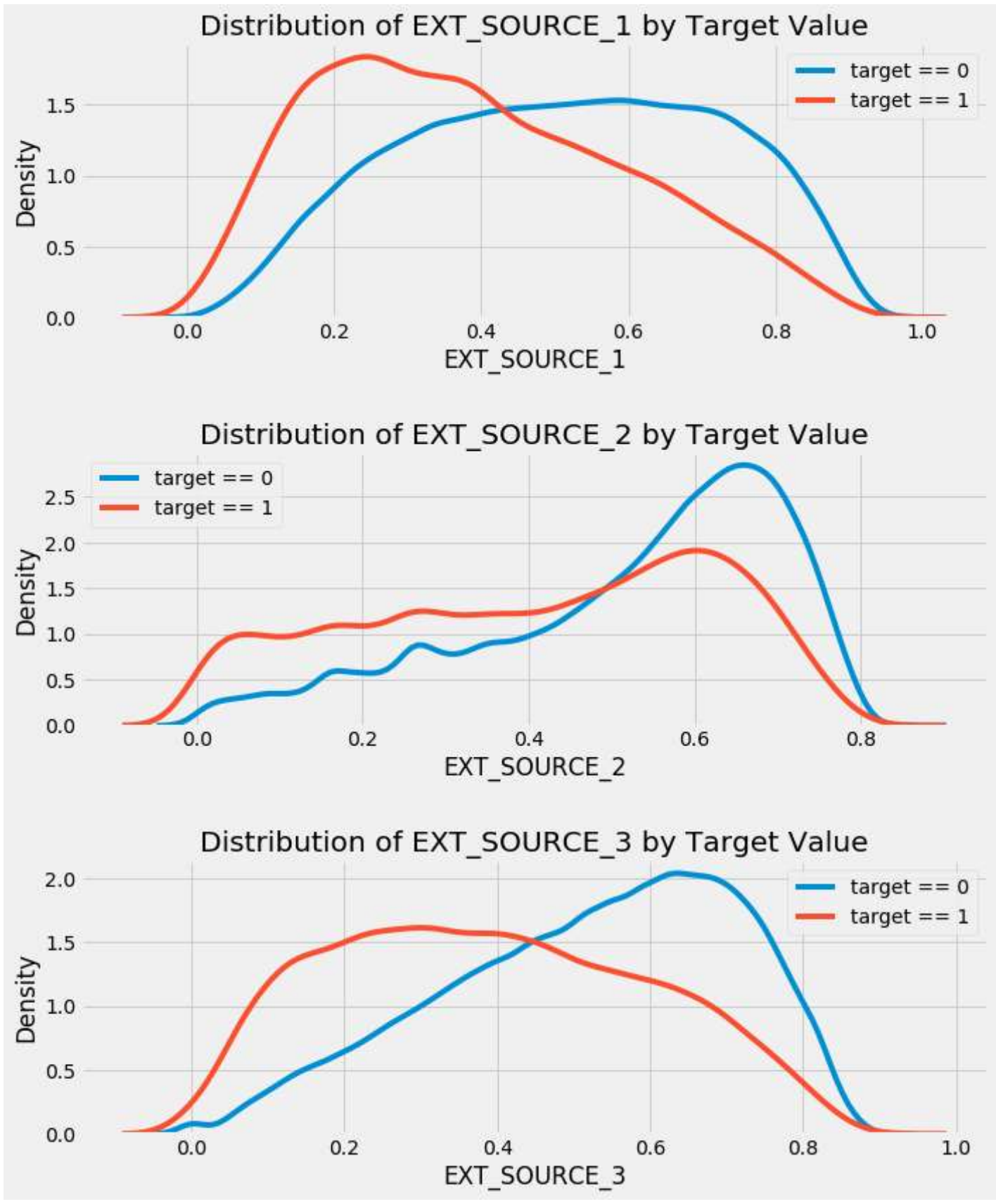
In [28]:
```
plt.figure(figsize = (8, 6))

# 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');
```



All three EXT_SOURCE features display negative correlations with the target, suggesting that higher EXT_SOURCE values correspond to a higher likelihood of loan repayment. Additionally, the positive correlation between DAYS_BIRTH and EXT_SOURCE_1 implies that perhaps one contributing factor to this score is the client's age.

To further understand the impact of these variables on the target, our next step involves visualizing the distribution of each feature, color-coded by the target value. This visualization will provide insights into how these variables affect the target.

In [29]:
```
plt.figure(figsize = (10, 12))

# 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_train.loc[app_train['TARGET'] == 0, source], label = 'target == 0')
    # plot loans that were not repaid
    sns.kdeplot(app_train.loc[app_train['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)
```



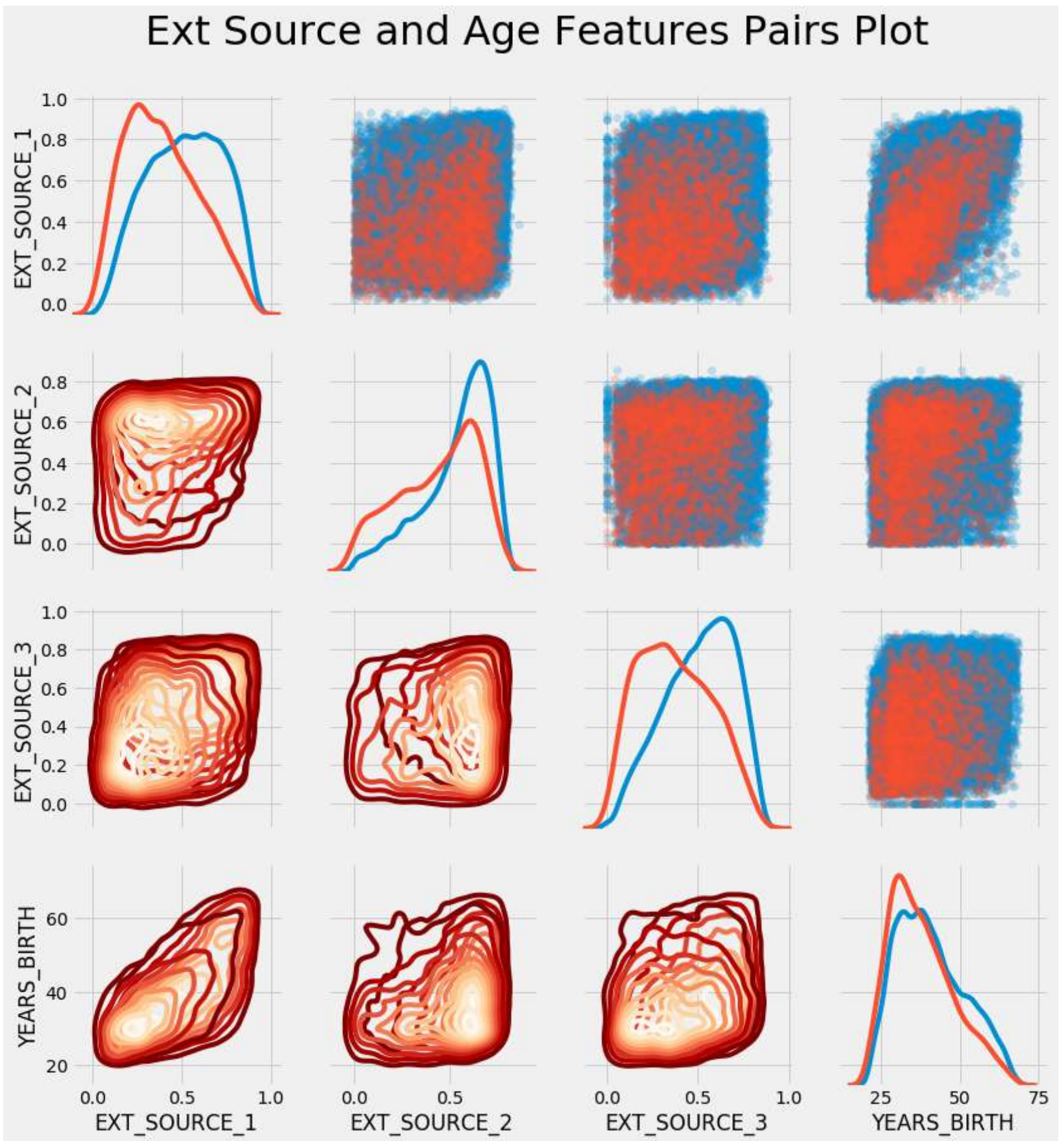
EXT_SOURCE_3 exhibits the most noticeable discrepancy between the target values. It's evident that this feature holds some association with the likelihood of an applicant repaying a loan. While the relationship isn't particularly strong (all considered weak), these variables remain valuable for a machine learning model in predicting an applicant's timely loan repayment.

Pairs Plot

For our final exploratory plot, we'll generate a pairs plot featuring the EXT_SOURCE variables alongside the DAYS_BIRTH variable. The Pairs Plot is an excellent exploration tool as it allows us to visualize relationships among multiple pairs of variables while also displaying the distributions of individual variables. To create this plot, we'll leverage the seaborn visualization library and utilize the PairGrid function. The resulting plot will showcase scatterplots in the upper triangle, histograms along the diagonal, and 2D kernel density plots along with correlation coefficients in the lower triangle.

Plotting in Python can indeed become quite intricate, especially for complex graphs. For tasks beyond the basics, leveraging existing implementations and adapting code can be a smart strategy to avoid unnecessary repetition.

In [30]:
```
# Copy the data for plotting
plot_data = ext_data.drop(columns = ['DAYS_BIRTH']).copy()

# Add in the age of the client in years
plot_data['YEARS_BIRTH'] = age_data['YEARS_BIRTH']

# Drop na values and limit to first 100000 rows
plot_data = plot_data.dropna().loc[:100000, :]

# Function to calculate correlation coefficient between two columns
def corr_func(x, y, **kwargs):
    r = np.corrcoef(x, y)[0][1]
    ax = plt.gca()
    ax.annotate("r = {:.2f}".format(r),
                xy=(.2, .8), xycoords=ax.transAxes,
                size = 20)

# Create the pairgrid object
grid = sns.PairGrid(data = plot_data, size = 3, diag_sharey=False,
                    hue = 'TARGET',
                    vars = [x for x in list(plot_data.columns) if x != 'TARGET'])

# Upper is a scatter plot
grid.map_upper(plt.scatter, alpha = 0.2)

# Diagonal is a histogram
grid.map_diag(sns.kdeplot)

# Bottom is density plot
grid.map_lower(sns.kdeplot, cmap = plt.cm.OrRd_r);

plt.suptitle('Ext Source and Age Features Pairs Plot', size = 32, y = 1.05);
```



In this plot, the red dots represent loans that were not repaid, while the blue dots represent paid loans. Examining the relationships within the data, a notable pattern emerges. There seems to be a moderate positive linear relationship between EXT_SOURCE_1 and DAYS_BIRTH (or equivalently YEARS_BIRTH), suggesting that this particular feature might indeed consider the age of the client in its calculation.

Feature Engineering

Feature engineering encompasses a broad process that includes both feature construction—introducing new features derived from existing data—and feature selection—choosing the most significant features or employing dimensionality reduction methods. There exist various techniques for both creating and selecting features.

While a more extensive feature engineering process will be employed when integrating other data sources, for this notebook, we'll focus on two straightforward feature construction methods:

- Polynomial features
- Domain knowledge features

Polynomial Features

One straightforward feature construction method is termed 'polynomial features.' This technique involves generating features that represent powers of existing features and interaction terms among them. For instance, we can produce variables like EXT_SOURCE_1^2 and EXT_SOURCE_2^2, as well as variables such as EXT_SOURCE_1 x EXT_SOURCE_2, EXT_SOURCE_1 x EXT_SOURCE_2^2, EXT_SOURCE_1^2 x EXT_SOURCE_2^2, and so forth. These composite features derived from multiple individual variables are known as interaction terms since they capture interactions between variables. Essentially, while two variables on their own might not strongly influence the target, combining them into a single interaction variable could reveal a relationship with the target. Interaction terms are frequently used in statistical models to capture the effects of multiple variables, although their utilization in machine learning isn't as common. Nevertheless, experimenting with a few might aid our model in predicting whether a client will repay a loan.

Jake VanderPlas covers polynomial features in detail in his exceptional book 'Python for Data Science' for those seeking further information.

In the subsequent code, we generate polynomial features using the EXT_SOURCE variables and the DAYS_BIRTH variable. Scikit-Learn offers a useful class called PolynomialFeatures, which constructs the polynomials and interaction terms up to a specified degree. We'll use a degree of 3 to observe the results. It's essential to avoid excessively high degrees when creating polynomial features to prevent exponential feature scaling and potential overfitting issues.

In [31]:
```
# Make a new dataframe for polynomial features
poly_features = app_train[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH', 'TARGET']]
poly_features_test = app_test[['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH']]

# imputer for handling missing values
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy = 'median')

poly_target = poly_features['TARGET']

poly_features = poly_features.drop(columns = ['TARGET'])

# Need to impute missing values
poly_features = imputer.fit_transform(poly_features)
poly_features_test = imputer.transform(poly_features_test)

from sklearn.preprocessing import PolynomialFeatures

# Create the polynomial object with specified degree
poly_transformer = PolynomialFeatures(degree = 3)
```

In [32]:
```
# Train the polynomial features
poly_transformer.fit(poly_features)

# Transform the features
poly_features = poly_transformer.transform(poly_features)
poly_features_test = poly_transformer.transform(poly_features_test)
print('Polynomial Features shape: ', poly_features.shape)
```

```
Polynomial Features shape:  (307511, 35)
```

This creates a considerable number of new features. To get the names we have to use the polynomial features `get_feature_names` method.

In [33]:
```
poly_transformer.get_feature_names(input_features = ['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIR
```

Out[33]:
```
['1',
 'EXT_SOURCE_1',
 'EXT_SOURCE_2',
 'EXT_SOURCE_3',
 'DAYS_BIRTH',
 'EXT_SOURCE_1^2',
 'EXT_SOURCE_1 EXT_SOURCE_2',
 'EXT_SOURCE_1 EXT_SOURCE_3',
 'EXT_SOURCE_1 DAYS_BIRTH',
 'EXT_SOURCE_2^2',
 'EXT_SOURCE_2 EXT_SOURCE_3',
 'EXT_SOURCE_2 DAYS_BIRTH',
 'EXT_SOURCE_3^2',
 'EXT_SOURCE_3 DAYS_BIRTH',
 'DAYS_BIRTH^2']
```

There are 35 features with individual features raised to powers up to degree 3 and interaction terms. Now, we can see whether any of these new features are correlated with the target.

In [34]:
```
# Create a dataframe of the features
poly_features = pd.DataFrame(poly_features,
                             columns = poly_transformer.get_feature_names(['EXT_SOURCE_1', 'EXT_SOURCE_2',
                                                                           'EXT_SOURCE_3', 'DAYS_BIRTH']))

# Add in the target
poly_features['TARGET'] = poly_target

# Find the correlations with the target
poly_corrs = poly_features.corr()['TARGET'].sort_values()

# Display most negative and most positive
print(poly_corrs.head(10))
print(poly_corrs.tail(5))
```

```
EXT_SOURCE_2 EXT_SOURCE_3                -0.193939
EXT_SOURCE_1 EXT_SOURCE_2 EXT_SOURCE_3   -0.189605
EXT_SOURCE_2 EXT_SOURCE_3 DAYS_BIRTH     -0.181283
EXT_SOURCE_2^2 EXT_SOURCE_3              -0.176428
EXT_SOURCE_2 EXT_SOURCE_3^2              -0.172282
EXT_SOURCE_1 EXT_SOURCE_2                -0.166625
EXT_SOURCE_1 EXT_SOURCE_3                -0.164065
EXT_SOURCE_2                             -0.160295
EXT_SOURCE_2 DAYS_BIRTH                  -0.156873
EXT_SOURCE_1 EXT_SOURCE_2^2              -0.156867
Name: TARGET, dtype: float64
DAYS_BIRTH     -0.078239
DAYS_BIRTH^2   -0.076672
DAYS_BIRTH^3   -0.074273
TARGET          1.000000
1                    NaN
Name: TARGET, dtype: float64
```

Some of the newly created variables exhibit stronger correlations (in absolute terms) with the target compared to the original features. When constructing machine learning models, we'll experiment by including and excluding these features to assess their actual impact on model learning.

To facilitate this comparison, we'll append these features to copies of both the training and testing data. Subsequently, we'll evaluate models with and without these additional features. In machine learning, often the most effective way to gauge the efficacy of an approach is through practical experimentation.

In [35]:
```
# Put test features into dataframe
poly_features_test = pd.DataFrame(poly_features_test,
                                  columns = poly_transformer.get_feature_names(['EXT_SOURCE_1', 'EXT_SOURCE_2'
                                                                                'EXT_SOURCE_3', 'DAYS_BIRTH'])

# Merge polynomial features into training dataframe
poly_features['SK_ID_CURR'] = app_train['SK_ID_CURR']
app_train_poly = app_train.merge(poly_features, on = 'SK_ID_CURR', how = 'left')

# Merge polnomial features into testing dataframe
poly_features_test['SK_ID_CURR'] = app_test['SK_ID_CURR']
app_test_poly = app_test.merge(poly_features_test, on = 'SK_ID_CURR', how = 'left')

# Align the dataframes
app_train_poly, app_test_poly = app_train_poly.align(app_test_poly, join = 'inner', axis = 1)

# Print out the new shapes
print('Training data with polynomial features shape: ', app_train_poly.shape)
print('Testing data with polynomial features shape:  ', app_test_poly.shape)
```

```
Training data with polynomial features shape:  (307511, 275)
Testing data with polynomial features shape:   (48744, 275)
```

Domain Knowledge Features

Though not stemming from expert credit knowledge, these could be seen as 'endeavors to apply limited financial insight.' Within this perspective, let's create a couple of features aimed at encapsulating what we believe might be significant indicators for predicting a client's loan default:

CREDIT_INCOME_PERCENT: Represents the percentage of the credit amount relative to a client's income.

ANNUITY_INCOME_PERCENT: Indicates the percentage of the loan annuity concerning a client's income.

CREDIT_TERM: Refers to the length of the payment in months, derived from the annuity as the monthly amount due.

DAYS_EMPLOYED_PERCENT: Represents the percentage of days employed relative to the client's age.

These features could offer valuable insights into a client's ability to handle loan repayments, even if they're not directly rooted in credit expertise.

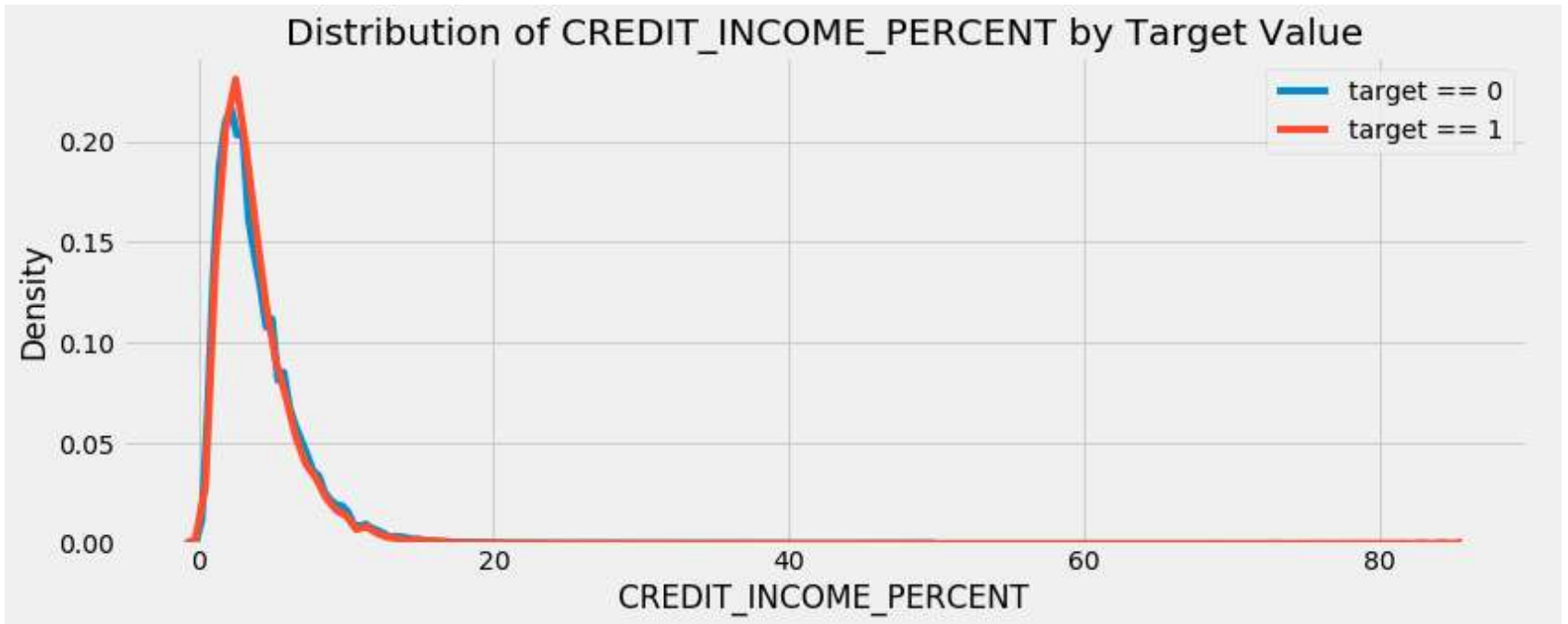
In [36]:
```
app_train_domain = app_train.copy()
app_test_domain = app_test.copy()

app_train_domain['CREDIT_INCOME_PERCENT'] = app_train_domain['AMT_CREDIT'] / app_train_domain['AMT_INCOME_TOTA
app_train_domain['ANNUITY_INCOME_PERCENT'] = app_train_domain['AMT_ANNUITY'] / app_train_domain['AMT_INCOME_TO
app_train_domain['CREDIT_TERM'] = app_train_domain['AMT_ANNUITY'] / app_train_domain['AMT_CREDIT']
app_train_domain['DAYS_EMPLOYED_PERCENT'] = app_train_domain['DAYS_EMPLOYED'] / app_train_domain['DAYS_BIRTH']
```

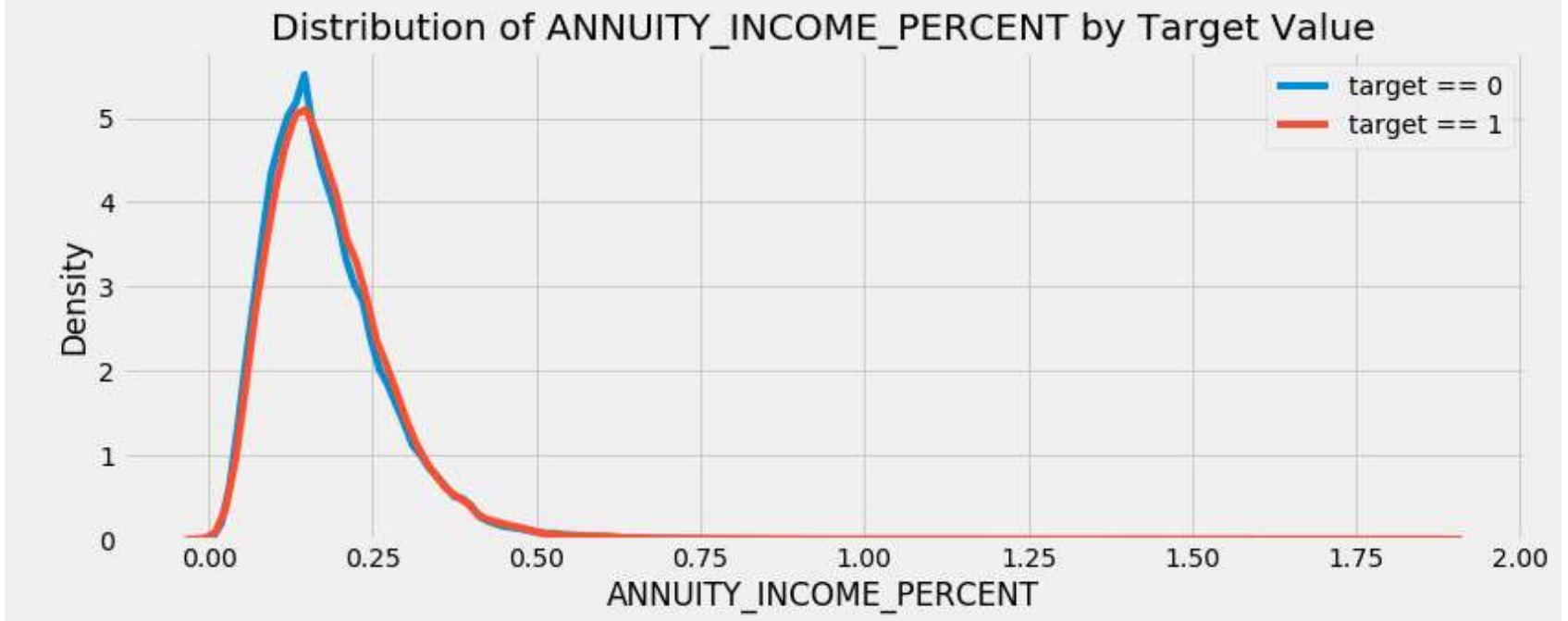
In [37]:
```
app_test_domain['CREDIT_INCOME_PERCENT'] = app_test_domain['AMT_CREDIT'] / app_test_domain['AMT_INCOME_TOTAL']
app_test_domain['ANNUITY_INCOME_PERCENT'] = app_test_domain['AMT_ANNUITY'] / app_test_domain['AMT_INCOME_TOTAL
app_test_domain['CREDIT_TERM'] = app_test_domain['AMT_ANNUITY'] / app_test_domain['AMT_CREDIT']
app_test_domain['DAYS_EMPLOYED_PERCENT'] = app_test_domain['DAYS_EMPLOYED'] / app_test_domain['DAYS_BIRTH']
```

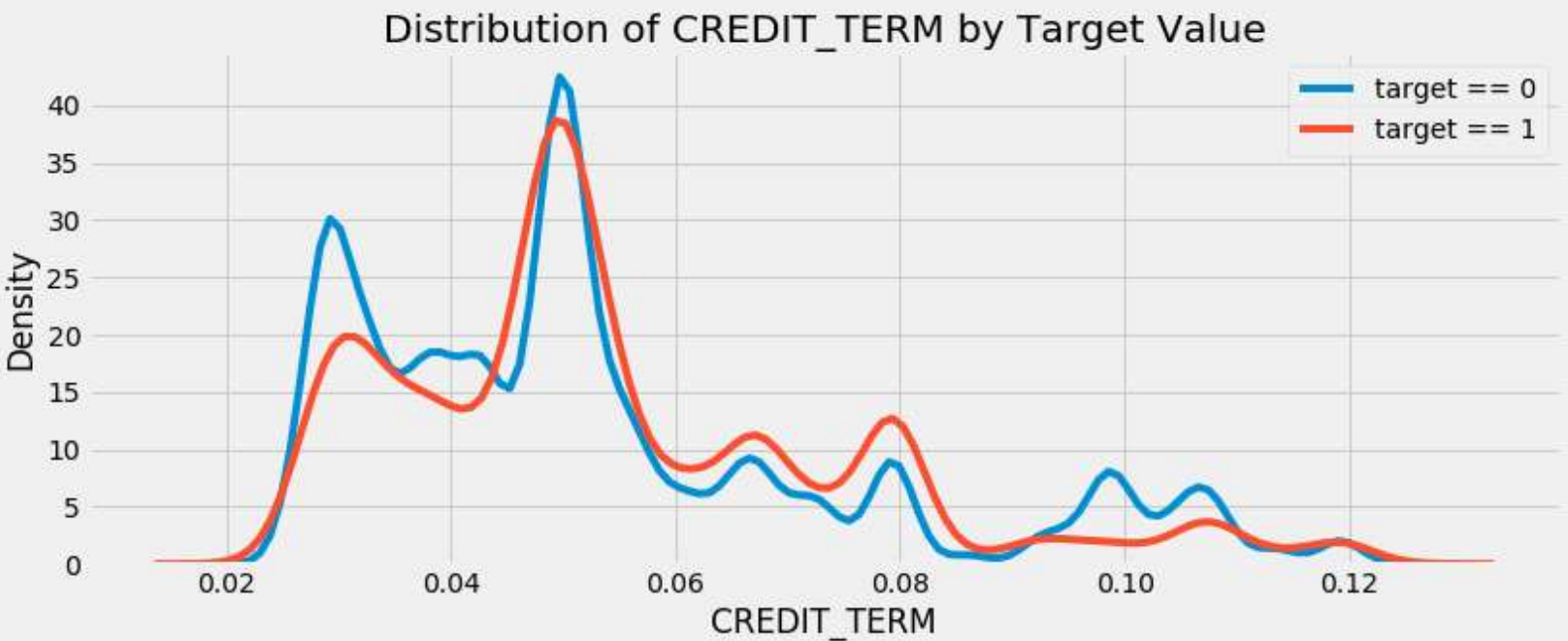
Visualize New Variables

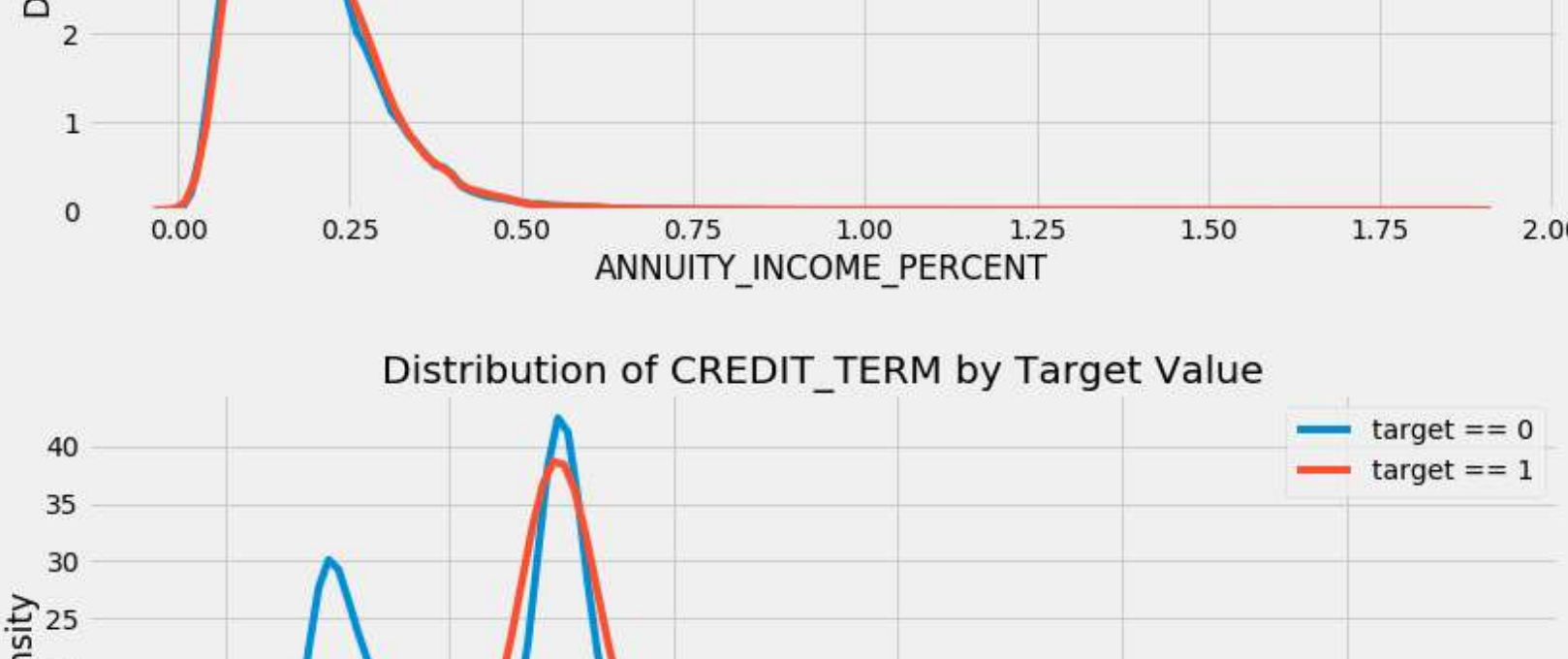
We should explore these **domain knowledge** variables visually in a graph. For all of these, we will make the same KDE plot colored by the value of the `TARGET` .

In [38]:
```
plt.figure(figsize = (12, 20))
# iterate through the new features
for i, feature in enumerate(['CREDIT_INCOME_PERCENT', 'ANNUITY_INCOME_PERCENT', 'CREDIT_TERM', 'DAYS_EMPLOYED_

    # create a new subplot for each source
    plt.subplot(4, 1, i + 1)
    # plot repaid loans
    sns.kdeplot(app_train_domain.loc[app_train_domain['TARGET'] == 0, feature], label = 'target == 0')
    # plot loans that were not repaid
    sns.kdeplot(app_train_domain.loc[app_train_domain['TARGET'] == 1, feature], label = 'target == 1')

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

plt.tight_layout(h_pad = 2.5)
```


Distribution of CREDIT_INCOME_PERCENT by Target Value
target == 0
target == 1
Density
0.00
0.05
0.10
0.15
0.20
0
20
40
60
80
CREDIT_INCOME_PERCENT


Distribution of ANNUITY_INCOME_PERCENT by Target Value
target == 0
target == 1
Density
0
1
2
3
4
5
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
ANNUITY_INCOME_PERCENT


Distribution of CREDIT_TERM by Target Value
target == 0
target == 1
Density
0
5
10
15
20
25
30
35
40
0.02
0.04
0.06
0.08
0.10
0.12
CREDIT_TERM


Distribution of DAYS_EMPLOYED_PERCENT by Target Value
target == 0
target == 1
Density
0
1
2
3
4
5
6
−0.8
−0.7
−0.6
−0.5
−0.4
−0.3
−0.2
−0.1
0.0
DAYS_EMPLOYED_PERCENT

Baseline

Logistic Regression Implementation

To establish a baseline, we'll utilize all the features post-encoding of categorical variables. Our preprocessing steps involve handling missing values through imputation and normalizing the feature range via feature scaling. The forthcoming code executes these preprocessing steps.

In [39]:
```
from sklearn.preprocessing import MinMaxScaler, Imputer

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

# Feature names
features = list(train.columns)

# Copy of the testing data
test = app_test.copy()

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

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

# Fit on the training data
imputer.fit(train)

# Transform both training and testing data
train = imputer.transform(train)
test = imputer.transform(app_test)

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

print('Training data shape: ', train.shape)
print('Testing data shape: ', test.shape)
```

```
Training data shape:  (307511, 240)
Testing data shape:  (48744, 240)
```

We'll start with Scikit-Learn's LogisticRegression for our initial model. The only adjustment we'll make from the default settings is to reduce the regularization parameter, C, which governs overfitting (lowering it should mitigate overfitting to some extent). While this adjustment should yield slightly improved results compared to the default LogisticRegression, it's intended to set a conservative benchmark for forthcoming models.

We'll follow the familiar Scikit-Learn modeling workflow: creating the model, training it using .fit, and subsequently generating predictions on the testing data using .predict_proba (remembering that we require probabilities rather than discrete 0s or 1s).

In [40]:
```
from sklearn.linear_model import LogisticRegression

# Make the model with the specified regularization parameter
log_reg = LogisticRegression(C = 0.0001)

# Train on the training data
log_reg.fit(train, train_labels)
```

Out[40]:
```
LogisticRegression(C=0.0001, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=None, penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

After training the model, it's time to employ it for predictions. Our aim is to predict the probabilities associated with loan repayment default. Utilizing the model's predict_proba method, we receive an m x 2 array (where m represents the number of observations). The first column denotes the probability of the target being 0, while the second column signifies the probability of the target being 1 (thus, the two columns sum up to 1 for each row). As we're interested in the probability of loan non-repayment, we'll select the second column.

The following code executes predictions and selects the appropriate column.

In [41]:
```
# Make predictions
# Make sure to select the second column only
log_reg_pred = log_reg.predict_proba(test)[:, 1]
```

To conform to the format specified in the sample_submission.csv file, where there are solely two columns—SK_ID_CURR and TARGET—we'll generate a dataframe adhering to this structure using the test set and their respective predictions. This new dataframe, named 'submit,' will adopt this format."

In [42]:
```
# Submission dataframe
submit = app_test[['SK_ID_CURR']]
submit['TARGET'] = log_reg_pred

submit.head()
```

Out[42]:

	SK_ID_CURR	TARGET
0	100001	0.087750
1	100005	0.163957
2	100013	0.110238
3	100028	0.076575
4	100038	0.154924

The predictions represent a probability between 0 and 1 that the loan will not be repaid. If we were using these predictions to classify applicants, we could set a probability threshold for determining that a loan is risky.

In [43]:
```
# Save the submission to a csv file
submit.to_csv('log_reg_baseline.csv', index = False)
```

The submission has now been saved to the virtual environment in which our notebook is running. To access the submission, at the end of the notebook, we will hit the blue Commit & Run button at the upper right of the kernel. This runs the entire notebook and then lets us download any files that are created during the run.

Once we run the notebook, the files created are available in the Versions tab under the Output sub-tab. From here, the submission files can be submitted to the competition or downloaded. Since there are several models in this notebook, there will be multiple output files.

The logistic regression baseline should score around 0.671 when submitted.

Improved Model: Random Forest

To try and beat the poor performance of our baseline, we can update the algorithm. Let's try using a Random Forest on the same training data to see how that affects performance. The Random Forest is a much more powerful model especially when we use hundreds of trees. We will use 100 trees in the random forest.

In [44]:
```
from sklearn.ensemble import RandomForestClassifier

# Make the random forest classifier
random_forest = RandomForestClassifier(n_estimators = 100, random_state = 50, verbose = 1, n_jobs = -1)
```

In [45]:
```
# Train on the training data
random_forest.fit(train, train_labels)

# Extract feature importances
feature_importance_values = random_forest.feature_importances_
feature_importances = pd.DataFrame({'feature': features, 'importance': feature_importance_values})

# Make predictions on the test data
predictions = random_forest.predict_proba(test)[:, 1]
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   42.5s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  1.6min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    1.0s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    2.3s finished
```

In [46]:
```
# Make a submission dataframe
submit = app_test[['SK_ID_CURR']]
submit['TARGET'] = predictions

# Save the submission dataframe
submit.to_csv('random_forest_baseline.csv', index = False)
```

These predictions will also be available when we run the entire notebook.

This model should score around 0.678 when submitted.

Make Predictions using Engineered Features

The only way to see if the Polynomial Features and Domain knowledge improved the model is to train a test a model on these features. We can then compare the submission performance to that for the model without these features to gauge the effect of our feature engineering.

In [47]:
```
poly_features_names = list(app_train_poly.columns)

# Impute the polynomial features
imputer = Imputer(strategy = 'median')

poly_features = imputer.fit_transform(app_train_poly)
poly_features_test = imputer.transform(app_test_poly)

# Scale the polynomial features
scaler = MinMaxScaler(feature_range = (0, 1))

poly_features = scaler.fit_transform(poly_features)
poly_features_test = scaler.transform(poly_features_test)

random_forest_poly = RandomForestClassifier(n_estimators = 100, random_state = 50, verbose = 1, n_jobs = -1)
```

In [48]:
```
# Train on the training data
random_forest_poly.fit(poly_features, train_labels)

# Make predictions on the test data
predictions = random_forest_poly.predict_proba(poly_features_test)[:, 1]
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:  1.0min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  2.4min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.7s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    1.7s finished
```

In [49]:
```
# Make a submission dataframe
submit = app_test[['SK_ID_CURR']]
submit['TARGET'] = predictions

# Save the submission dataframe
submit.to_csv('random_forest_baseline_engineered.csv', index = False)
```

This model scored 0.678 when submitted, exactly the same as that without the engineered features. Given these results, it does not appear that our feature construction helped in this case.

Testing Domain Features

Now we can test the domain features we made by hand.

In [50]:
```
app_train_domain = app_train_domain.drop(columns = 'TARGET')

domain_features_names = list(app_train_domain.columns)

# Impute the domainnomial features
imputer = Imputer(strategy = 'median')

domain_features = imputer.fit_transform(app_train_domain)
domain_features_test = imputer.transform(app_test_domain)

# Scale the domainnomial features
scaler = MinMaxScaler(feature_range = (0, 1))

domain_features = scaler.fit_transform(domain_features)
domain_features_test = scaler.transform(domain_features_test)

random_forest_domain = RandomForestClassifier(n_estimators = 100, random_state = 50, verbose = 1, n_jobs = -1)

# Train on the training data
random_forest_domain.fit(domain_features, train_labels)

# Extract feature importances
feature_importance_values_domain = random_forest_domain.feature_importances_
feature_importances_domain = pd.DataFrame({'feature': domain_features_names, 'importance': feature_importance_

# Make predictions on the test data
predictions = random_forest_domain.predict_proba(domain_features_test)[:, 1]
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   44.3s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  1.7min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    1.0s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    2.3s finished
```

In [51]:
```
# Make a submission dataframe
submit = app_test[['SK_ID_CURR']]
submit['TARGET'] = predictions

# Save the submission dataframe
submit.to_csv('random_forest_baseline_domain.csv', index = False)
```

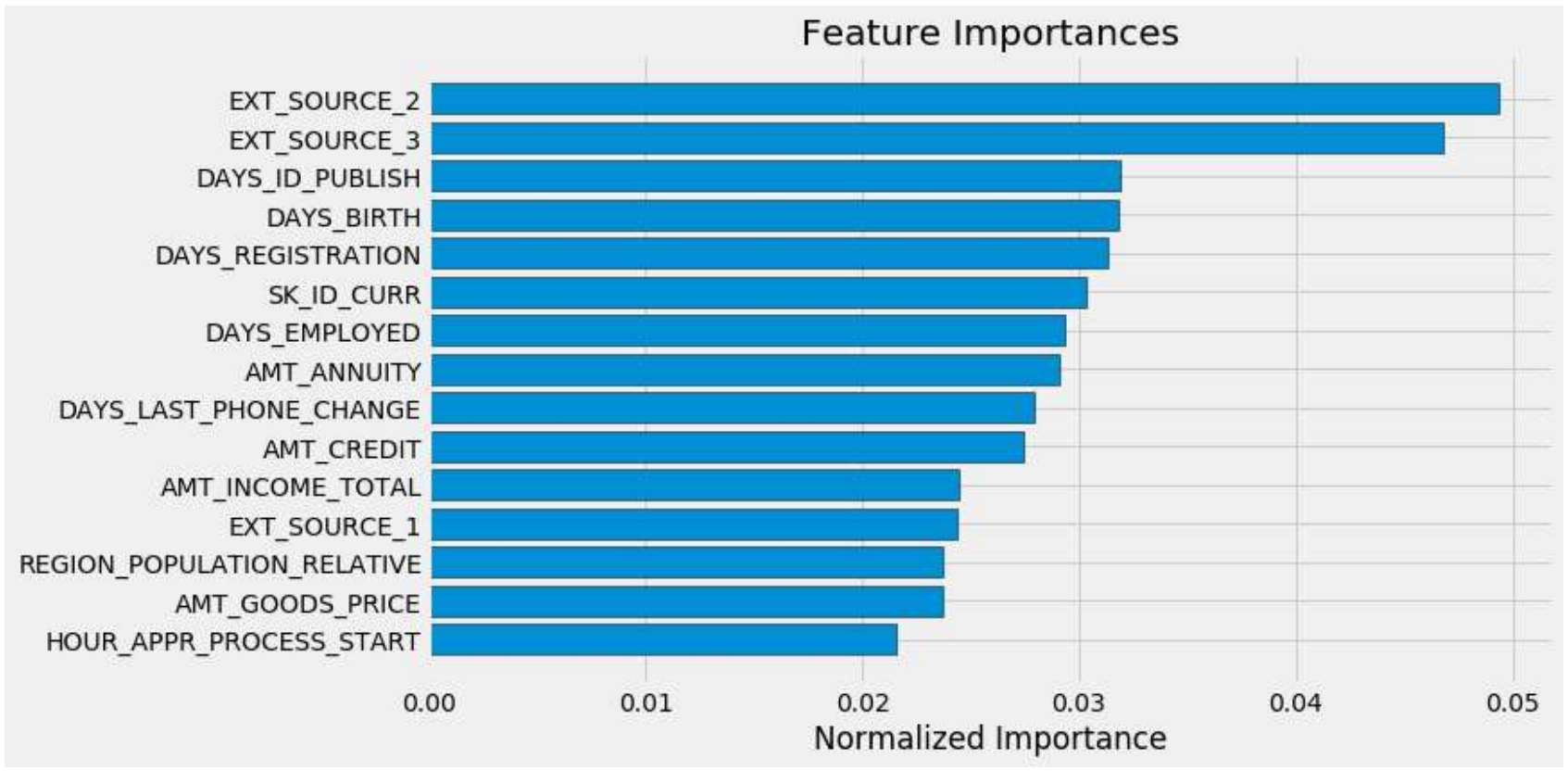
This scores 0.679 when submitted which probably shows that the engineered features do not help in this model (however they do help

Model Interpretation: Feature Importances

As a simple method to see which variables are the most relevant, we can look at the feature importances of the random forest. Given the correlations we saw in the exploratory data analysis, we should expect that the most important features are the `EXT_SOURCE` and the `DAYS_BIRTH`. We may use these feature importances as a method of dimensionality reduction in future work.

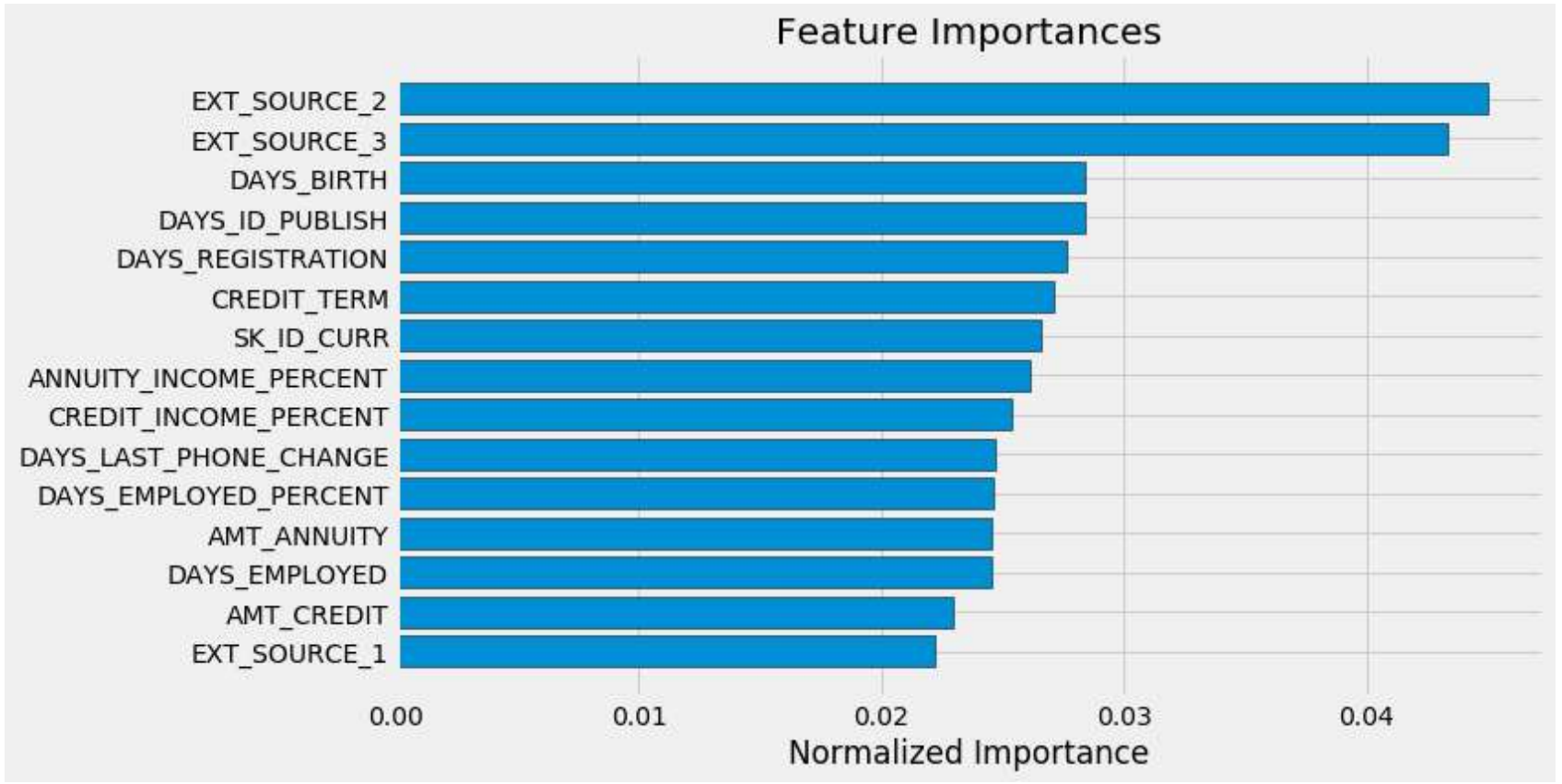
In [52]:
```
def plot_feature_importances(df):
    """
    Plot importances returned by a model. This can work with any measure of
    feature importance provided that higher importance is better.

    Args:
        df (dataframe): feature importances. Must have the features in a column
        called `features` and the importances in a column called `importance

    Returns:
        shows a plot of the 15 most importance features

        df (dataframe): feature importances sorted by importance (highest to lowest)
        with a column for normalized importance
        """

    # Sort features according to importance
    df = df.sort_values('importance', ascending = False).reset_index()

    # Normalize the feature importances to add up to one
    df['importance_normalized'] = df['importance'] / df['importance'].sum()

    # Make a horizontal bar chart of feature importances
    plt.figure(figsize = (10, 6))
    ax = plt.subplot()

    # Need to reverse the index to plot most important on top
    ax.barh(list(reversed(list(df.index[:15]))),
            df['importance_normalized'].head(15),
            align = 'center', edgecolor = 'k')

    # Set the yticks and labels
    ax.set_yticks(list(reversed(list(df.index[:15]))))
    ax.set_yticklabels(df['feature'].head(15))

    # Plot labeling
    plt.xlabel('Normalized Importance'); plt.title('Feature Importances')
    plt.show()

    return df
```

In [53]:
```
# Show the feature importances for the default features
feature_importances_sorted = plot_feature_importances(feature_importances)
```



As expected, the most important features are those dealing with `EXT_SOURCE` and `DAYS_BIRTH`. We see that there are only a handful of features with a significant importance to the model, which suggests we may be able to drop many of the features without a decrease in performance (and we may even see an increase in performance.) Feature importances are not the most sophisticated method to interpret a model or perform dimensionality reduction, but they let us start to understand what factors our model takes into account when it makes predictions.

In [54]:
```
feature_importances_domain_sorted = plot_feature_importances(feature_importances_domain)
```



We see that all four of our hand-engineered features made it into the top 15 most important. This should give us confidence that our domain knowledge was at least partially on track.

Light Gradient Boosting (LightGBM)

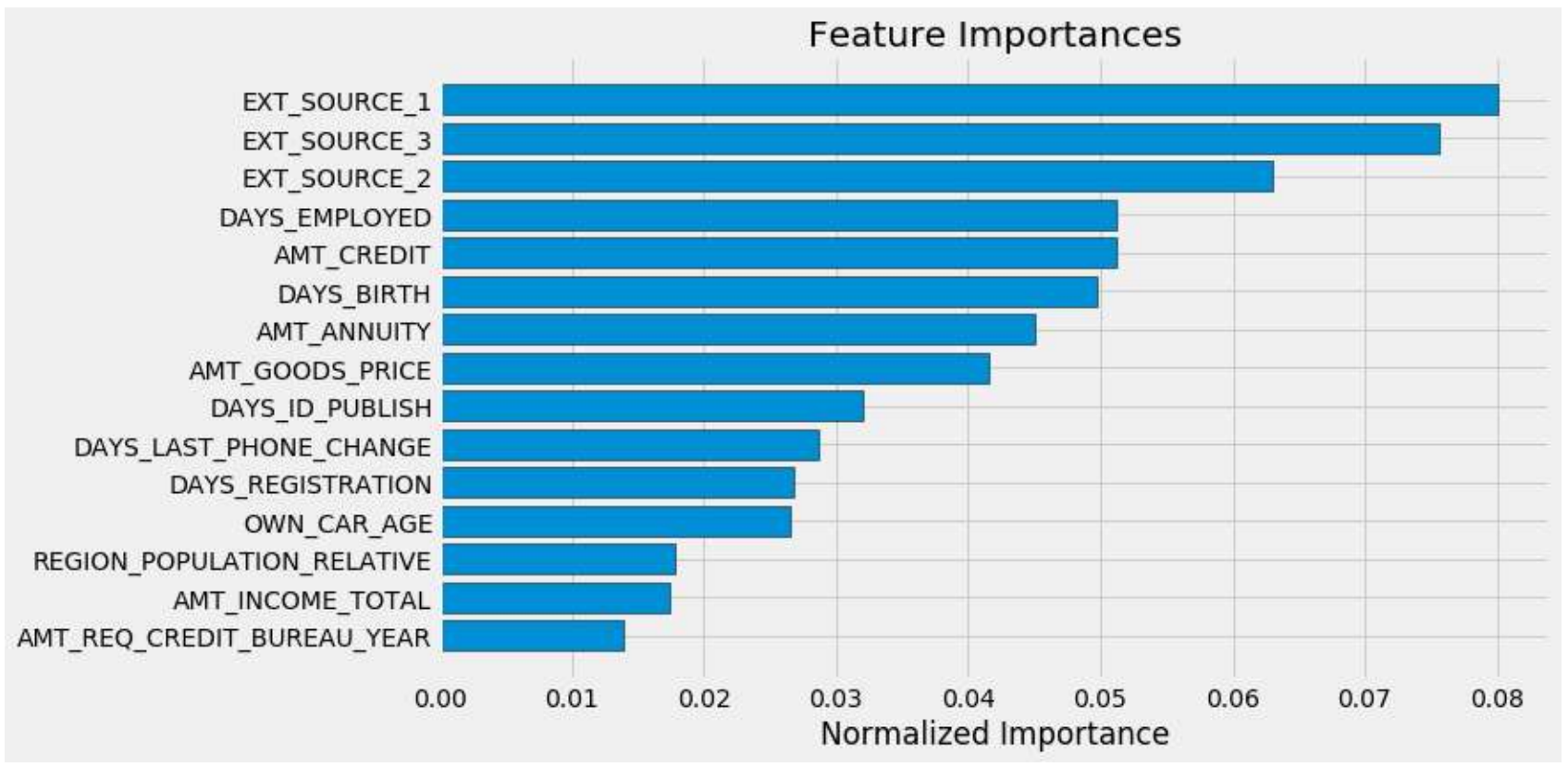
In [55]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score
import lightgbm as lgb
import gc

def model(features, test_features, encoding = 'ohe', n_folds = 5):

    """Train and test a light gradient boosting model using
    cross validation.

    Parameters
    --------
        features (pd.DataFrame):
            dataframe of training features to use
            for training a model. Must include the TARGET column.
        test_features (pd.DataFrame):
            dataframe of testing features to use
            for making predictions with the model.
        encoding (str, default = 'ohe'):
            method for encoding categorical variables. Either 'ohe' for one-hot encoding or 'le' for integer l
            n_folds (int, default = 5): number of folds to use for cross validation

    Return
    --------
        submission (pd.DataFrame):
            dataframe with `SK_ID_CURR` and `TARGET` probabilities
            predicted by the model.
        feature_importances (pd.DataFrame):
            dataframe with the feature importances from the model.
        valid_metrics (pd.DataFrame):
            dataframe with training and validation metrics (ROC AUC) for each fold and overall.

    """

    # Extract the ids
    train_ids = features['SK_ID_CURR']
    test_ids = test_features['SK_ID_CURR']

    # Extract the labels for training
    labels = features['TARGET']

    # Remove the ids and target
    features = features.drop(columns = ['SK_ID_CURR', 'TARGET'])
    test_features = test_features.drop(columns = ['SK_ID_CURR'])


    # One Hot Encoding
    if encoding == 'ohe':
        features = pd.get_dummies(features)
        test_features = pd.get_dummies(test_features)

        # Align the dataframes by the columns
        features, test_features = features.align(test_features, join = 'inner', axis = 1)

        # No categorical indices to record
        cat_indices = 'auto'

    # Integer label encoding
    elif encoding == 'le':

        # Create a label encoder
        label_encoder = LabelEncoder()

        # List for storing categorical indices
        cat_indices = []

        # Iterate through each column
        for i, col in enumerate(features):
            if features[col].dtype == 'object':
                # Map the categorical features to integers
                features[col] = label_encoder.fit_transform(np.array(features[col].astype(str)).reshape((-1,))
                test_features[col] = label_encoder.transform(np.array(test_features[col].astype(str)).reshape(

                # Record the categorical indices
                cat_indices.append(i)

    # Catch error if label encoding scheme is not valid
    else:
        raise ValueError("Encoding must be either 'ohe' or 'le'")

    print('Training Data Shape: ', features.shape)
    print('Testing Data Shape: ', test_features.shape)

    # Extract feature names
    feature_names = list(features.columns)

    # Convert to np arrays
    features = np.array(features)
    test_features = np.array(test_features)

    # Create the kfold object
```

```
    k_fold = KFold(n_splits = n_folds, shuffle = True, random_state = 50)

    # Empty array for feature importances
    feature_importance_values = np.zeros(len(feature_names))

    # Empty array for test predictions
    test_predictions = np.zeros(test_features.shape[0])

    # Empty array for out of fold validation predictions
    out_of_fold = np.zeros(features.shape[0])

    # Lists for recording validation and training scores
    valid_scores = []
    train_scores = []

    # Iterate through each fold
    for train_indices, valid_indices in k_fold.split(features):

        # Training data for the fold
        train_features, train_labels = features[train_indices], labels[train_indices]
        # Validation data for the fold
        valid_features, valid_labels = features[valid_indices], labels[valid_indices]

        # Create the model
        model = lgb.LGBMClassifier(n_estimators=10000, objective = 'binary',
                                   class_weight = 'balanced', learning_rate = 0.05,
                                   reg_alpha = 0.1, reg_lambda = 0.1,
                                   subsample = 0.8, n_jobs = -1, random_state = 50)

        # Train the model
        model.fit(train_features, train_labels, eval_metric = 'auc',
                  eval_set = [(valid_features, valid_labels), (train_features, train_labels)],
                  eval_names = ['valid', 'train'], categorical_feature = cat_indices,
                  early_stopping_rounds = 100, verbose = 200)

        # Record the best iteration
        best_iteration = model.best_iteration_

        # Record the feature importances
        feature_importance_values += model.feature_importances_ / k_fold.n_splits

        # Make predictions
        test_predictions += model.predict_proba(test_features, num_iteration = best_iteration)[:, 1] / k_fold.

        # Record the out of fold predictions
        out_of_fold[valid_indices] = model.predict_proba(valid_features, num_iteration = best_iteration)[:, 1]

        # Record the best score
        valid_score = model.best_score_['valid']['auc']
        train_score = model.best_score_['train']['auc']

        valid_scores.append(valid_score)
        train_scores.append(train_score)

        # Clean up memory
        gc.enable()
        del model, train_features, valid_features
        gc.collect()

    # Make the submission dataframe
    submission = pd.DataFrame({'SK_ID_CURR': test_ids, 'TARGET': test_predictions})

    # Make the feature importance dataframe
    feature_importances = pd.DataFrame({'feature': feature_names, 'importance': feature_importance_values})

    # Overall validation score
    valid_auc = roc_auc_score(labels, out_of_fold)

    # Add the overall scores to the metrics
    valid_scores.append(valid_auc)
    train_scores.append(np.mean(train_scores))

    # Needed for creating dataframe of validation scores
    fold_names = list(range(n_folds))
    fold_names.append('overall')

    # Dataframe of validation scores
    metrics = pd.DataFrame({'fold': fold_names,
                            'train': train_scores,
                            'valid': valid_scores})

    return submission, feature_importances, metrics
```

In [56]:
```
submission, fi, metrics = model(app_train, app_test)
print('Baseline metrics')
print(metrics)
```

```
Training Data Shape:  (307511, 239)
Testing Data Shape:  (48744, 239)
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.754949   train's auc: 0.79887
Early stopping, best iteration is:
[208]   valid's auc: 0.755109   train's auc: 0.80025
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.758539   train's auc: 0.798518
Early stopping, best iteration is:
[217]   valid's auc: 0.758619   train's auc: 0.801374
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.762652   train's auc: 0.79774
[400]   valid's auc: 0.762202   train's auc: 0.827288
Early stopping, best iteration is:
[320]   valid's auc: 0.763103   train's auc: 0.81638
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.757496   train's auc: 0.799107
Early stopping, best iteration is:
[183]   valid's auc: 0.75759    train's auc: 0.796125
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.758099   train's auc: 0.798268
Early stopping, best iteration is:
[227]   valid's auc: 0.758251   train's auc: 0.802746
Baseline metrics
      fold     train     valid
0        0  0.800250  0.755109
1        1  0.801374  0.758619
2        2  0.816380  0.763103
3        3  0.796125  0.757590
4        4  0.802746  0.758251
5  overall  0.803375  0.758537
```

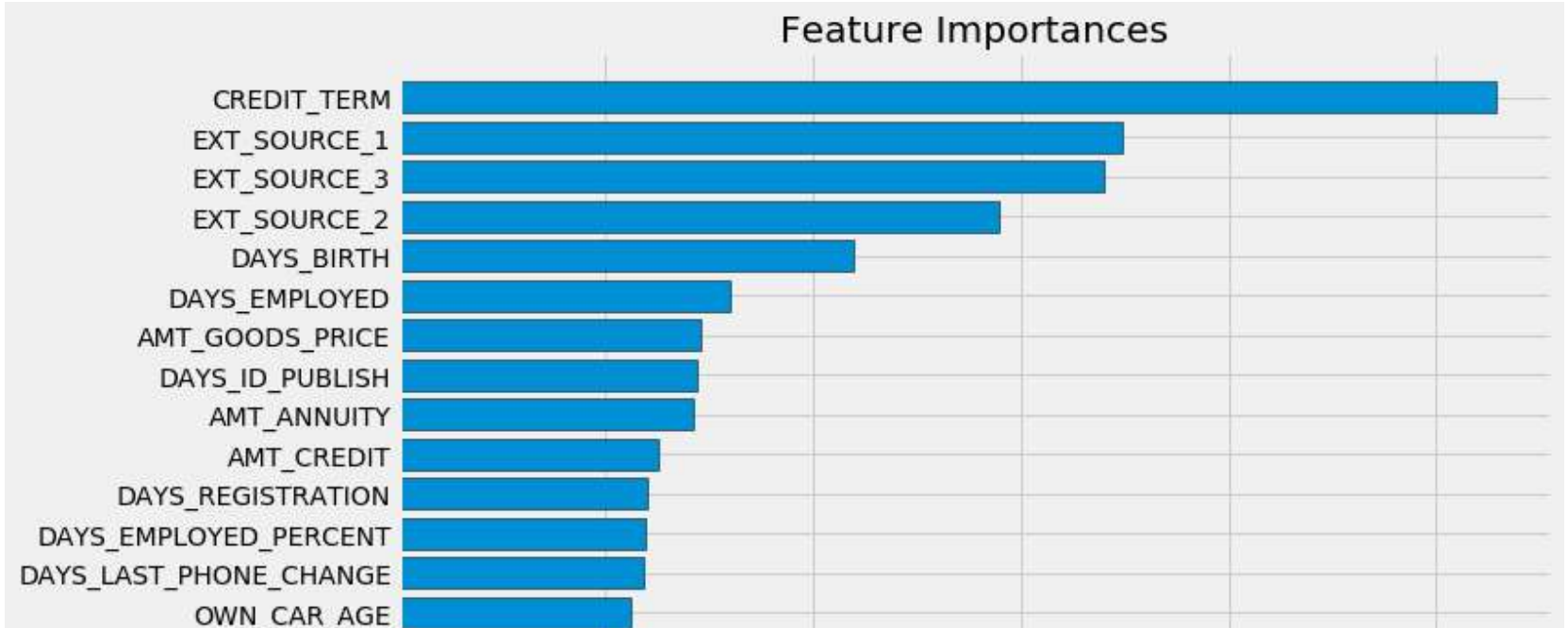
In [57]:
```
fi_sorted = plot_feature_importances(fi)
```



In [59]:
```
app_train_domain['TARGET'] = train_labels

# Test the domain knolwedge features
submission_domain, fi_domain, metrics_domain = model(app_train_domain, app_test_domain)
print('Baseline with domain knowledge features metrics')
print(metrics_domain)
```

```
Training Data Shape:  (307511, 243)
Testing Data Shape:  (48744, 243)
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.762577   train's auc: 0.804531
Early stopping, best iteration is:
[237]   valid's auc: 0.762858   train's auc: 0.810671
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.765594   train's auc: 0.804304
Early stopping, best iteration is:
[227]   valid's auc: 0.765861   train's auc: 0.808665
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.770139   train's auc: 0.803753
[400]   valid's auc: 0.770328   train's auc: 0.834338
Early stopping, best iteration is:
[302]   valid's auc: 0.770629   train's auc: 0.820401
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.765653   train's auc: 0.804487
Early stopping, best iteration is:
[262]   valid's auc: 0.766318   train's auc: 0.815066
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.764456   train's auc: 0.804527
Early stopping, best iteration is:
[235]   valid's auc: 0.764517   train's auc: 0.810422
Baseline with domain knowledge features metrics
      fold     train     valid
0        0  0.810671  0.762858
1        1  0.808665  0.765861
2        2  0.820401  0.770629
3        3  0.815066  0.766318
4        4  0.810422  0.764517
5  overall  0.813045  0.766050
```

In [60]:
```
fi_sorted = plot_feature_importances(fi_domain)
```



In [61]:
```
submission_domain.to_csv('baseline_lgb_domain_features.csv', index = False)
```