Home Credit Default Risk (data preprocessing,baseline Model)

Import Libraries

In [1]:

```
# numpy and pandas for data manipulation

import numpy as np
import pandas as pd

# matplotlib and seaborn for plotting

import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import plotly.offline as py
py.init_notebook_mode(connected=True)
from plotly.offline import init_notebook_mode, iplot
init_notebook_mode(connected=True)
import plotly.graph_objs as go
import plotly.offline as offline


# sklearn preprocessing for dealing with categorical variables

from sklearn.preprocessing import LabelEncoder

# File system manangement
import os

# Suppress warnings

import warnings
warnings.filterwarnings('ignore')

# modeling
import lightgbm as lgb

# utilities
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score
from sklearn.preprocessing import LabelEncoder

# memory management
import gc

# featuretools for automated feature engineering

#import featuretools as ft

#!pip install cufflinks
import cufflinks as cf
cf.go_offline()
cf.set_config_file(offline=False, world_readable=True)

print("Libraries imported")
```

```
Libraries imported
```

Read in Data

First, we can 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.

Load the data

In [2]:

```
app_train = pd.read_csv("application_train.csv")
app_test = pd.read_csv("application_test.csv")

print("data loaded......")
```

```
data loaded......
```

Check the data

In [3]:

```
print("application_train -  rows:",app_train.shape[0]," columns:", app_train.shape[1])
print("application_test -  rows:",app_test.shape[0]," columns:", app_test.shape[1])
```

```
application_train -  rows: 307511  columns: 122
application_test -  rows: 48744  columns: 121
```

Function to Convert Data Types

This will help reduce memory usage by using more efficient types for the variables. For example category is often a better type than object (unless the number of unique categories is close to the number of rows in the dataframe).

In [4]:

```
import sys

def return_size(df):
    """Return size of dataframe in gigabytes"""
    return round(sys.getsizeof(df) / 1e9, 2)

def convert_types(df, print_info = False):

    original_memory = df.memory_usage().sum()

    # Iterate through each column
    for c in df:

        # Convert ids and booleans to integers
        if ('SK_ID' in c):
            df[c] = df[c].fillna(0).astype(np.int32)

        # Convert objects to category
        elif (df[c].dtype == 'object') and (df[c].nunique() < df.shape[0]):
            df[c] = df[c].astype('category')

        # Booleans mapped to integers
        elif list(df[c].unique()) == [1, 0]:
            df[c] = df[c].astype(bool)

        # Float64 to float32
        elif df[c].dtype == float:
            df[c] = df[c].astype(np.float32)

        # Int64 to int32
        elif df[c].dtype == int:
            df[c] = df[c].astype(np.int32)

    new_memory = df.memory_usage().sum()

    if print_info:
        print(f'Original Memory Usage: {round(original_memory / 1e9, 2)} gb.')
        print(f'New Memory Usage: {round(new_memory / 1e9, 2)} gb.')
```

```
        print(f'New Memory Usage: {round(new_memory / 1e9, 2)} gb.')

    return df
```

In [5]:

```
app_train=convert_types(app_train, print_info=True)
app_train.head()
```

```
Original Memory Usage: 0.3 gb.
New Memory Usage: 0.17 gb.
```

Out[5]:

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

5 rows × 122 columns

Application_train

In [6]:

```
app_train.head()
```

Out[6]:

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

5 rows × 122 columns

In [7]:

```
#app_train.columns.values
```

Application_test

In [8]:

```
app_test.head()
```

Out[8]:

	SK_ID_CURR	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_TO
0	100001	Cash loans	F	N	Y	0	1350
1	100005	Cash loans	M	N	Y	0	990
2	100013	Cash loans	M	Y	Y	0	2025
3	100028	Cash loans	F	N	Y	2	3150
4	100038	Cash loans	M	Y	N	1	1800

	SK_ID_CURR	NAME_CONTRACT_TYPE	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_TO

5 rows × 121 columns

In [9]:

```
#app_test.columns.values
```

Exploratory Data Analysis

Distribution of the Target Column

TARGET value 0 means loan is repayed, value 1 means loan is not repayed.

In [10]:

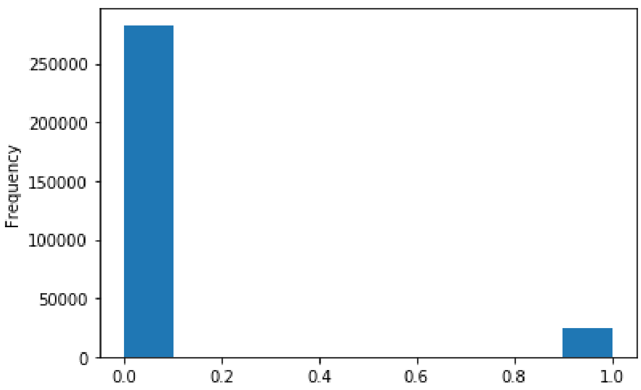
```
app_train['TARGET'].value_counts()
```

Out[10]:

```
False    282686
True      24825
Name: TARGET, dtype: int64
```

In [11]:

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



In [12]:

```
cf.set_config_file(theme='polar')
temp = app_train["TARGET"].value_counts()
df = pd.DataFrame({'labels': temp.index,
                   'values': temp.values
                  })
df
```

Out[12]:

	labels	values
0	False	282686
1	True	24825

In [13]:

```
df.iplot(kind='pie',labels='labels',values='values', title='Loan Repayed or not',hole = 0.5)
```

Observations:

1. The data is imbalanced (91.9%(Loan repayed-0) and 8.07%(Loan not repayed-1)) and we need to handle this problem.

Examine Missing Values

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

In [14]:

```
#sns.heatmap(app_train.isnull(), cbar=False,)
```

In [15]:

```
# 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
```

In [16]:

```
missing_values_table(app_train)
```

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

Out[16]:

	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
...	...	...
EXT_SOURCE_2	660	0.2
AMT_GOODS_PRICE	278	0.1
AMT_ANNUITY	12	0.0
CNT_FAM_MEMBERS	2	0.0
DAYS_LAST_PHONE_CHANGE	1	0.0

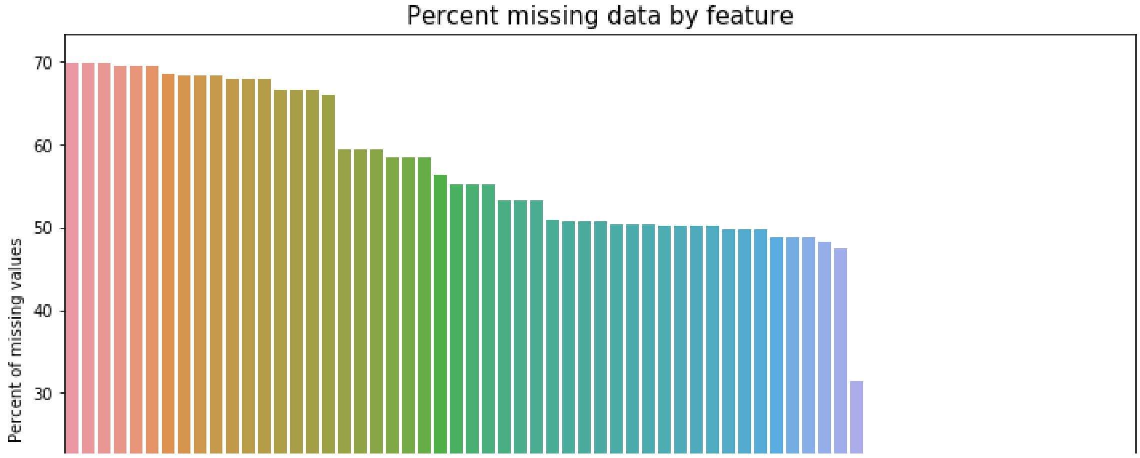
67 rows × 2 columns

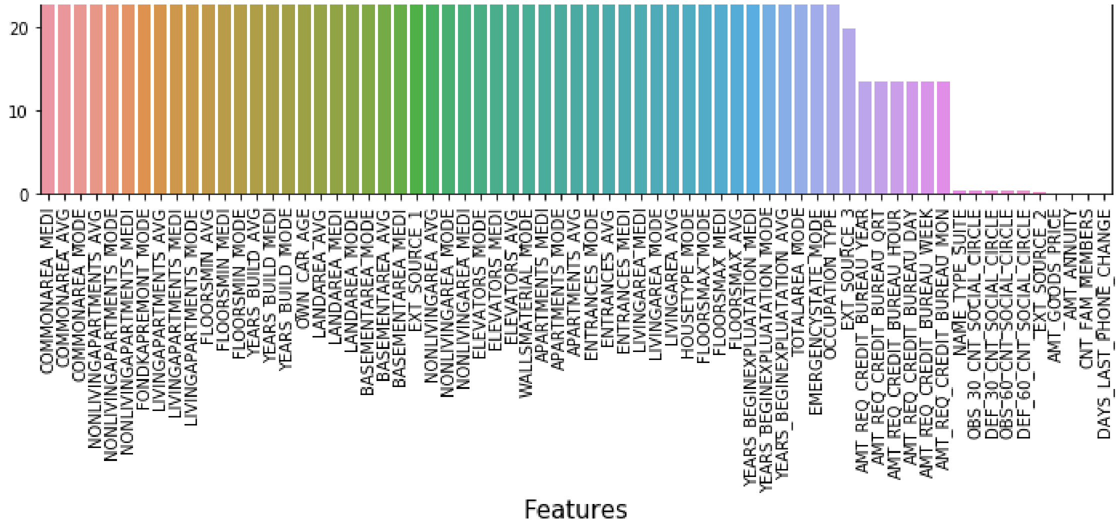
In [17]:

```
# Function to missing data plot

def missingdata_plot(df):
    all_data_na = (df.isnull().sum() / len(df)) * 100
    all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascending=False
)
    f, ax = plt.subplots(figsize=(12, 7))
    plt.xticks(rotation='90')
    sns.barplot(x=all_data_na.index, y=all_data_na)
    plt.xlabel('Features', fontsize=15)
    plt.ylabel('Percent of missing values', fontsize=10)
    plt.title('Percent missing data by feature', fontsize=15)
```

In [18]:

```
missingdata_plot(app_train)
```





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. .

In [19]:

```
# Number of each type of column
app_train.dtypes.value_counts()
```

Out[19]:

```
float32      65
int64        34
bool          6
category      2
category      1
category      1
int32         1
category      1
category      1
category      1
category      1
category      1
category      1
category      1
category      1
category      1
category      1
category      1
dtype: int64
```

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

In [20]:

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

Out[20]:

```
Series([], dtype: float64)
```

Label Encoding and One-Hot Encoding

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 [21]:

```
# Create a label encoder object
le = LabelEncoder()
le_count = 0
```

In [22]:

```
# 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)
```

```
0 columns were label encoded.
```

In [23]:

```
# 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, 246)
Testing Features shape:  (48744, 242)
```

Aligning Training and Testing Data

There need to be the same features (columns) in both the training and testing data. One-hot encoding has created more columns in the training data because there were some categorical variables with categories not represented in the testing data. To remove the columns in the training data that are not in the testing data, we need to align the dataframes. First we extract the target column from the training data (because this is not in the testing data but we need to keep this information). When we do the align, we must make sure to set axis = 1 to align the dataframes based on the columns and not on the rows!

In [24]:

```
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, 243)
Testing Features shape:  (48744, 242)
```

The training and testing datasets now have the same features which is required for machine learning. The number of features has grown significantly due to one-hot encoding. At some point we probably will want to try dimensionality reduction (removing features that are not relevant) to reduce the size of the datasets.

Back to Exploratory Data Analysis

Anomalies

Distribution of Clients Age

In [25]:

```
app_train['DAYS_BIRTH'].head()
```

Out[25]:

```
0    -9461
1   -16765
2   -19046
3   -19005
4   -19932
Name: DAYS_BIRTH, dtype: int64
```

The numbers in the DAYS_BIRTH column are negative because they are recorded relative to the current loan application. To see these stats in years, we can mutliple by -1 and divide by the number of days in a year:

In [26]:

```
(app_train['DAYS_BIRTH'] / -365).head()
```

Out[26]:

```
0    25.920548
1    45.931507
2    52.180822
3    52.068493
4    54.608219
Name: DAYS_BIRTH, dtype: float64
```

In [27]:

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

Out[27]:

```
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
```

In [28]:

```
cf.set_config_file(theme='pearl')
(app_train['DAYS_BIRTH']/(-365)).iplot(kind='histogram',
             xTitle = 'Age', bins=50,
             yTitle='Count of type of applicants in %',
             title='Distribution of Clients Age')
```

Those ages look reasonable. There are no outliers for the age on either the high or low end.

Distribution of years before the application the person started current employment.

In [29]:

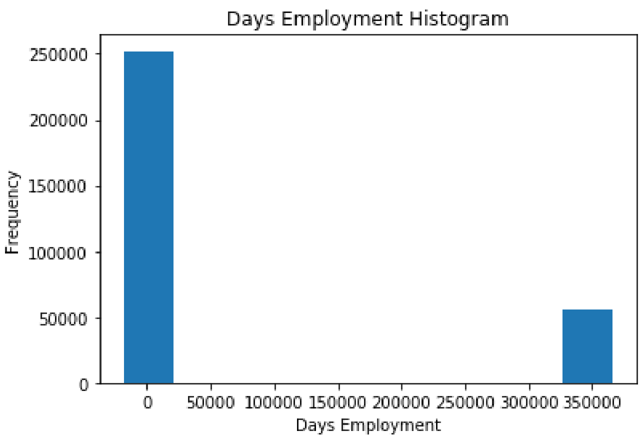
```
app_train['DAYS_EMPLOYED'].describe()
```

Out[29]:

```
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
```

In [30]:

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



In [31]:

```
cf.set_config_file(theme='pearl')

(app_train['DAYS_EMPLOYED']).iplot(kind='histogram',
             xTitle = 'Days',bins=50,
             yTitle='Count of applicants in %',
             title='Days before the application the person started current employment')
```

Observations:

1. The data looks strange(we have -1000.66 years(-365243 days) of employment which is impossible) looks like there is data entry error.

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 [32]:

```
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
```

Handling the anomalies

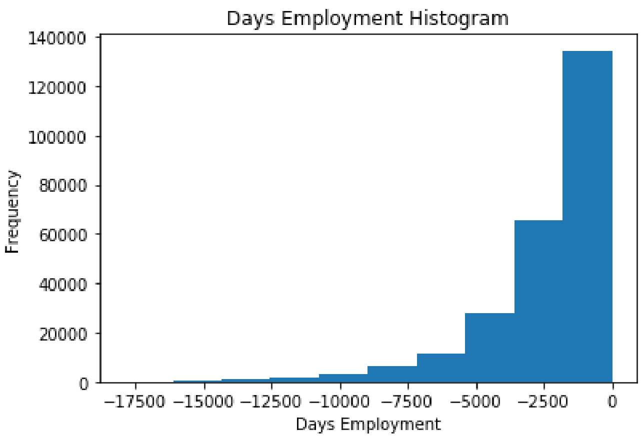
- anomalies have a lower rate of default.
- Handling the anomalies depends on the exact situation, with no set rules. One of the safest approaches is just to set the anomalies to a missing value and then have them filled in (using Imputation) before machine learning.
- In this case, since all the anomalies 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.

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.

In [33]:

```
# 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');
```



In [34]:

```
cf.set_config_file(theme='pearl')
(app_train['DAYS_EMPLOYED']/(-365)).iplot(kind='histogram', xTitle = 'Years of Employment',bins=50,
             yTitle='Count of applicants in %',
             title='Years before the application the person started current employment')
```

In [35]:

```
app_train[app_train['DAYS_EMPLOYED']>(-365*2)]
```

```
['TARGET'].value_counts()/sum(app_train['DAYS_EMPLOYED']>(-365*2))
```

Out[35]:

```
False    0.887924
True     0.112076
Name: TARGET, dtype: float64
```

Observations:

1. The applicants with less than 2 years of employment are less likely to repay the loan.

Note-anything we do to the training data we also have to do to the testing data. Let's make sure to create the new column and fill in the existing column with np.nan in the testing data.

In [36]:

```
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(app_test)))
```

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

Correlations

Now that we have dealt with the categorical variables and the outliers, let's continue with the EDA. One way to try and 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.

In [37]:

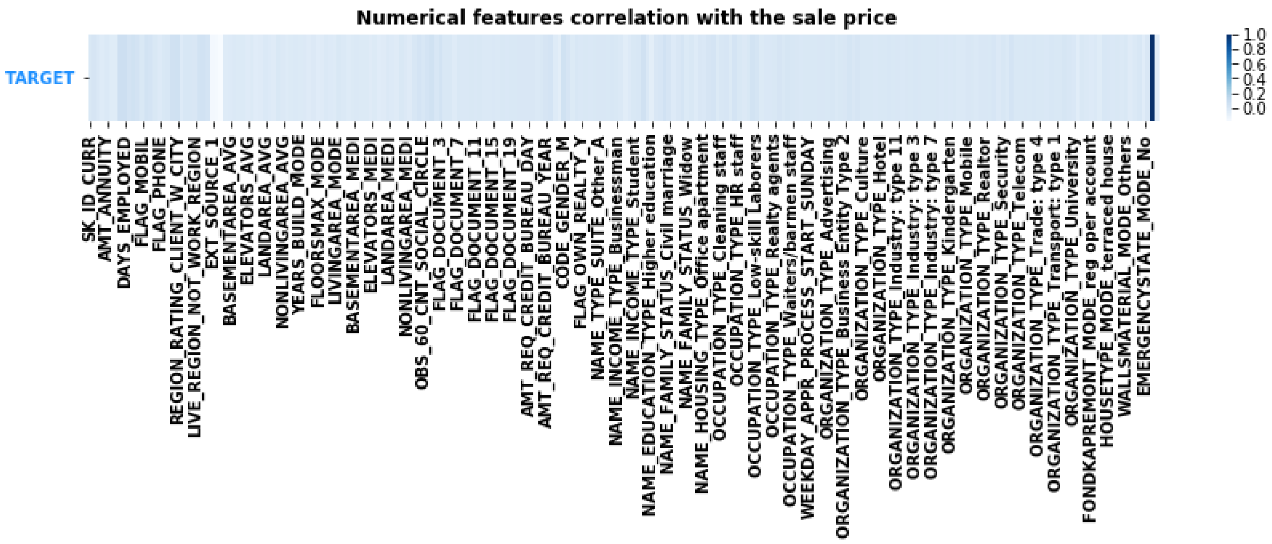
```
correlations = app_train.corr()['TARGET'].sort_values()

# 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
```

In [38]:

```
num=app_train.select_dtypes(exclude='object')
numcorr=num.corr()
f,ax=plt.subplots(figsize=(15,1))
sns.heatmap(numcorr.sort_values(by=['TARGET'], ascending=False).head(1), cmap='Blues')
plt.title(" Numerical features correlation with the sale price", weight='bold', fontsize=12)
plt.xticks(weight='bold')
plt.yticks(weight='bold', color='dodgerblue', rotation=0)


plt.show()
```



In [39]:

```
corr=app_train.corr()
```

- the DAYS_BIRTH is the most positive correlation.Looking at the documentation, DAYS_BIRTH is the age in days of the client at the time of the loan in negative days (for whatever reason!).
- 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.

Effect of Age on Repayment

In [40]:

```
# 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[40]:

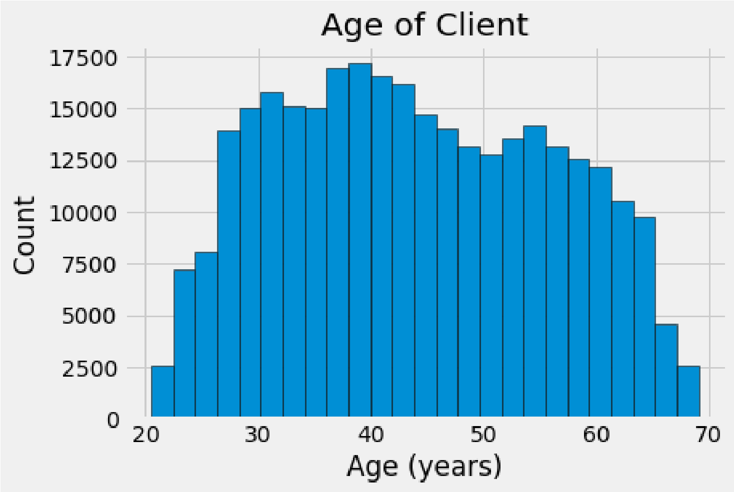
```
-0.07823930830982709
```

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.

In [41]:

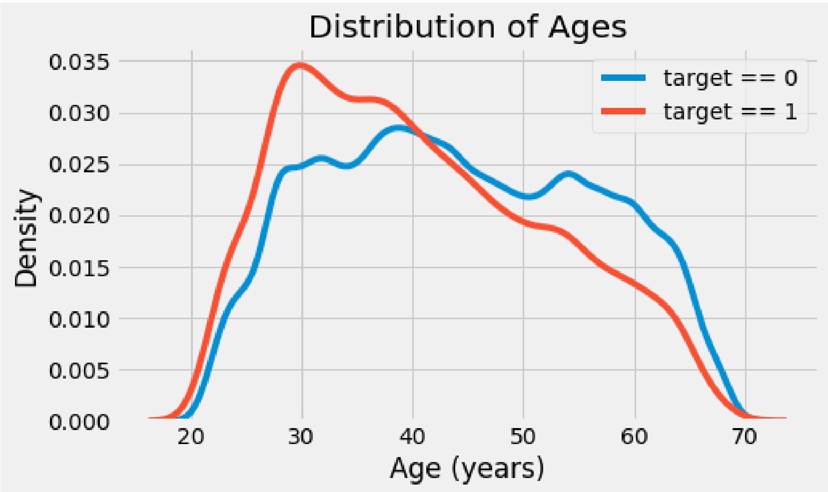
```
# Set the style of plots
plt.style.use('fivethirtyeight')

# 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');
```



the distribution of age does not tell us much other 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.

In [42]:

```
plt.figure(figsize = (7, 4))

# 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 target == 1 curve skews towards the younger end of the range. 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.

In [43]:

```
# 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[43]:

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

In [44]:

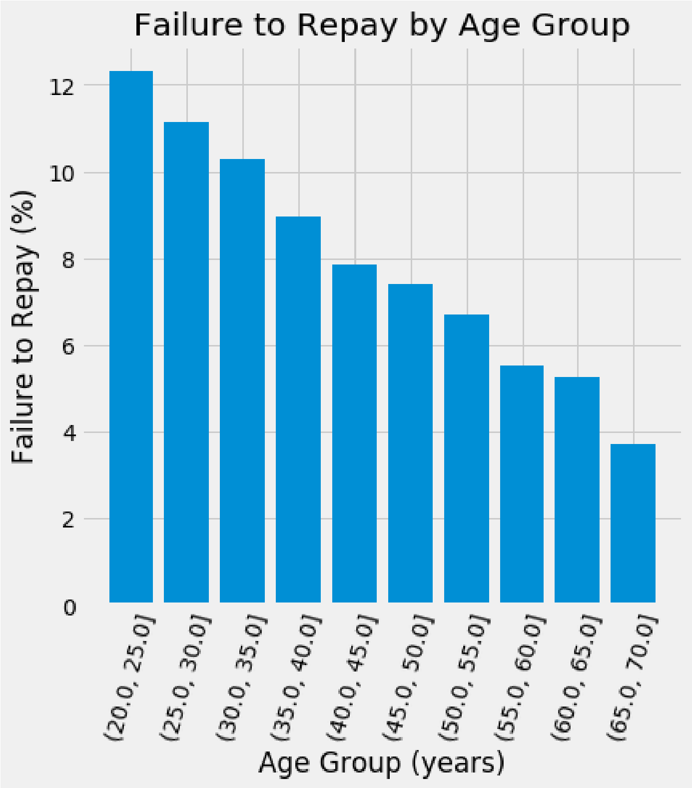
```
# Group by the bin and calculate averages
age_groups  = age_data.groupby('YEARS_BINNED').mean()
age_groups
```

Out[44]:

	TARGET	DAYS_BIRTH	YEARS_BIRTH
YEARS_BINNED			
(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 [45]:

```
plt.figure(figsize = (6, 6))

# 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 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.

Let's take a look at 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.
- First, we can show the correlations of the EXT_SOURCE features with the target and with each other.

In [46]:

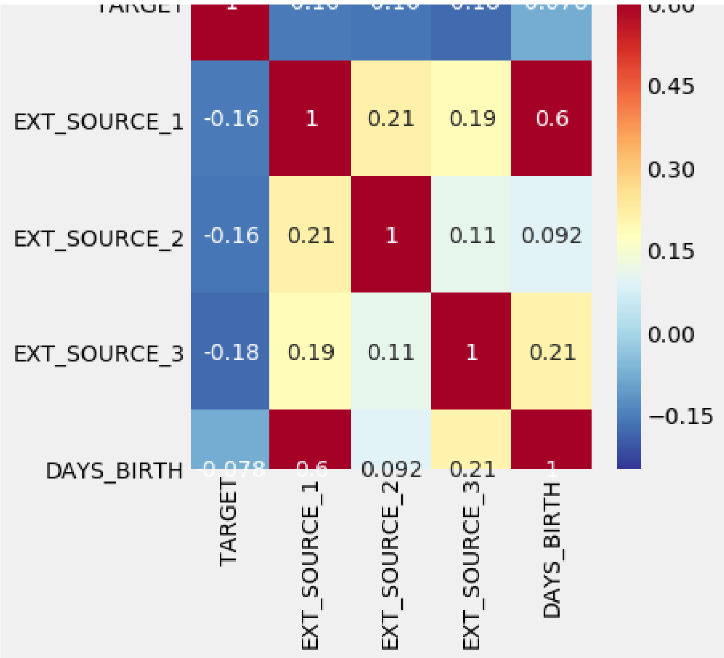
```
# Extract the EXT_SOURCE variables and show correlations
ext_data = app_train[['TARGET', 'EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3', 'DAYS_BIRTH']]
ext_data_corrs = ext_data.corr()
ext_data_corrs
```

Out[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

In [47]:

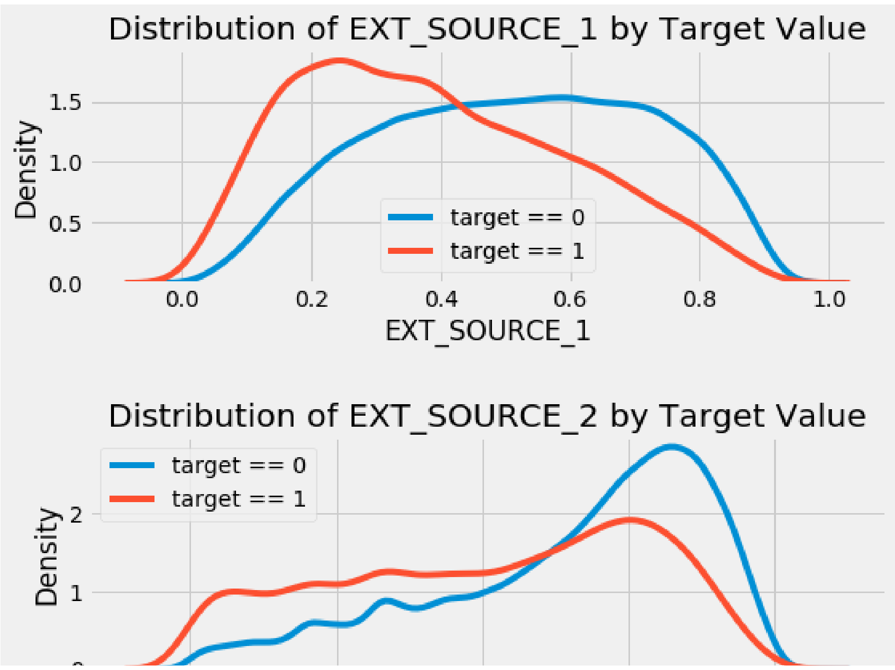
```
plt.figure(figsize = (5, 5))

# 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');
```

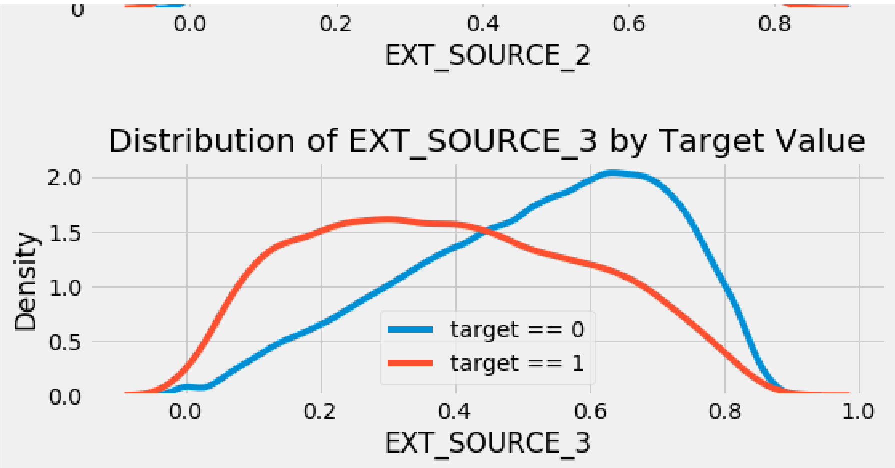


All three EXT_SOURCE featureshave negative correlations with the target, indicating that as the value of the EXT_SOURCE increases, the client is more likely to repay the loan. We can also see that DAYS_BIRTH is positively correlated with EXT_SOURCE_1 indicating that maybe one of the factors in this score is the client age.

In [48]:

```
plt.figure(figsize = (8, 10))

# 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 displays the greatest difference between the values of the target. We can clearly see that this feature has some 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.

Feature Engineering

in this notebook we will try only two simple feature construction methods:

- Polynomial features
- Domain knowledge features

Polynomial Features

we create polynomial features using the EXT_SOURCE variables and the DAYS_BIRTH variable.

In [49]:

```
# 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']]
```

In [50]:

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

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


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

In [51]:

```
from sklearn.preprocessing import PolynomialFeatures

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

# Train the polynomial features
poly_transformer.fit(poly_features)
```

Out[51]:

```
PolynomialFeatures(degree=3, include_bias=True, interaction_only=False,
                   order='C')
```

In [52]:

```
# 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 [53]:

```
poly_transformer.get_feature_names(input_features = ['EXT_SOURCE_1', 'EXT_SOURCE_2', 'EXT_SOURCE_3'
, 'DAYS_BIRTH'])[:15]
```

Out[53]:

```
['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 [54]:

```
# Create a dataframe of the features
poly_features = pd.DataFrame(poly_features,
                             columns = poly_transformer.get_feature_names(['EXT_SOURCE_1', 'EXT_SOU
CE_2',
                                                                           'EXT_SOURCE_3', 'DAYS_BI
']))

# 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
```

Several of the new variables have a greater (in terms of absolute magnitude) correlation with the target than the original features. When we build machine learning models, we can try with and without these features to determine if they actually help the model learn.

We will add these features to a copy of the training and testing data and then evaluate models with and without the features. Many times in machine learning, the only way to know if an approach will work is to try it out!

In [55]:

```
# Put test features into dataframe
poly_features_test = pd.DataFrame(poly_features_test,
                                  columns = poly_transformer.get_feature_names(['EXT_SOURCE_1', 'EX
_SOURCE_2',
                                                                               'EXT_SOURCE_3', 'DA
BIRTH']))
```

In [56]:

```
# 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, 278)
Testing data with polynomial features shape:   (48744, 278)
```

Domain Knowledge Features

Here I'm going to use five features that were inspired by this script by Aguiar:

- CREDIT_INCOME_PERCENT: the percentage of the credit amount relative to a client's income
- ANNUITY_INCOME_PERCENT: the percentage of the loan annuity relative to a client's income
- CREDIT_TERM: the length of the payment in months (since the annuity is the monthly amount due
- DAYS_EMPLOYED_PERCENT: the percentage of the days employed relative to the client's age

In [57]:

```
app_train_domain = app_train.copy()
app_test_domain = app_test.copy()
```

In [58]:

```
app_train_domain['CREDIT_INCOME_PERCENT'] = app_train_domain['AMT_CREDIT'] / app_train_domain['AMT_
INCOME_TOTAL']
app_train_domain['ANNUITY_INCOME_PERCENT'] = app_train_domain['AMT_ANNUITY'] / app_train_domain['AM
T_INCOME_TOTAL']
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['D
AYS_BIRTH']
```

In [59]:

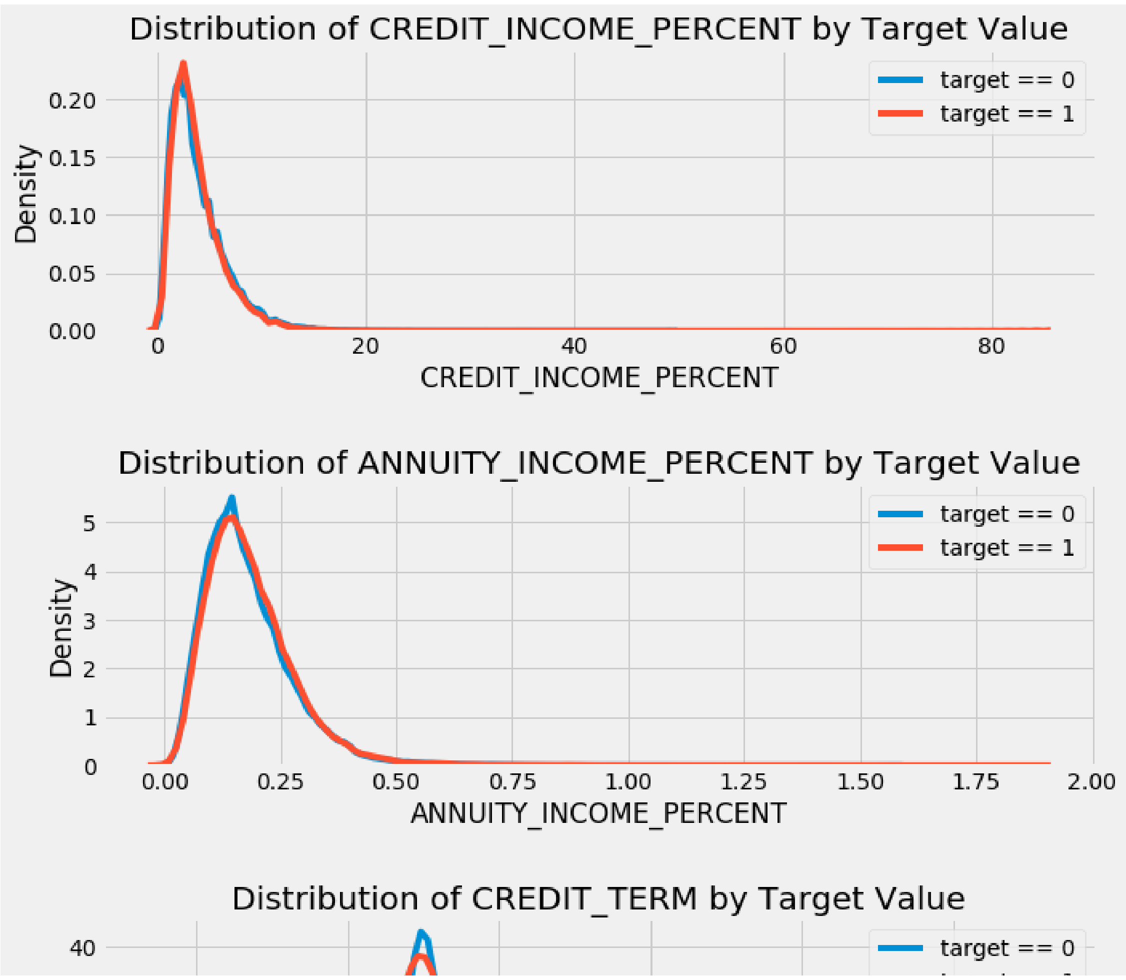
```
app_test_domain['CREDIT_INCOME_PERCENT'] = app_test_domain['AMT_CREDIT'] / 
app_test_domain['AMT_INCOME_TOTAL']
app_test_domain['ANNUITY_INCOME_PERCENT'] = app_test_domain['AMT_ANNUITY'] / app_test_domain['AMT_I
NCOME_TOTAL']
app_test_domain['CREDIT_TERM'] = app_test_domain['AMT_ANNUITY'] / app_test_domain['AMT_CREDIT']
app_test_domain['DAYS_EMPLOYED_PERCENT'] = app_test_domain['DAYS_EMPLOYED'] / 
app_test_domain['DAYS_BIRTH']
```

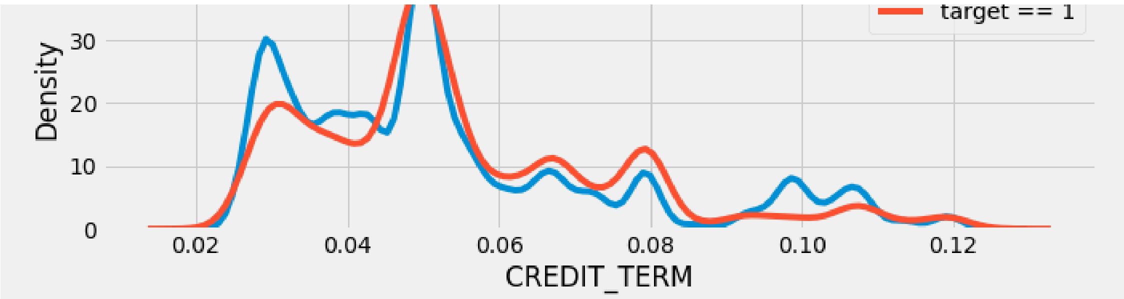
Visualize New Variables

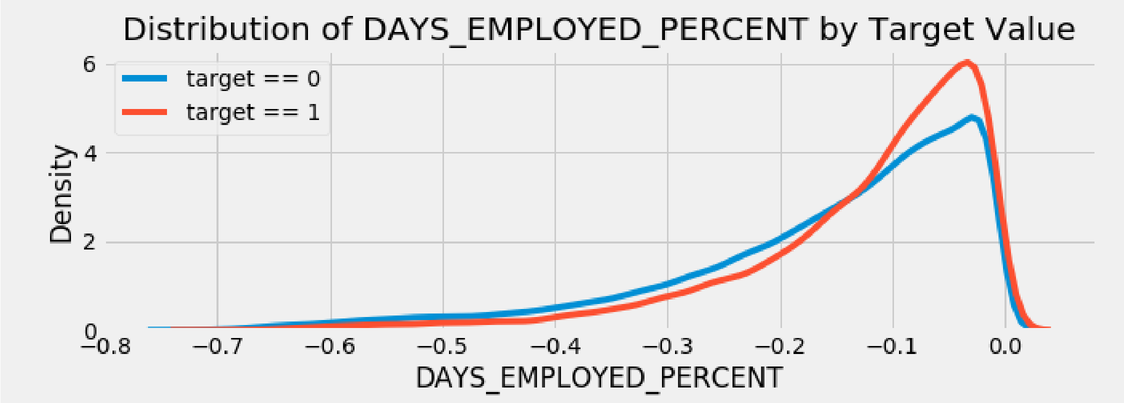
We should explore these domain knowledge variables visually in a graph. For all of these, we will make the same KDE plot colored by the value of the TARGET.

In [60]:

```
plt.figure(figsize = (10, 15))
# iterate through the new features
for i, feature in enumerate(['CREDIT_INCOME_PERCENT', 'ANNUITY_INCOME_PERCENT', 'CREDIT_TERM', 
'DAYS_EMPLOYED_PERCENT']):

    # 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)
```







Modeling --Baseline

Model-1 (Logistic Regression Model with all feature of training set)

To get a baseline,following preprocessing steps performs

- use all of the features of training set
- encoding the categorical variables.
- preprocess the data by filling in the missing values (imputation).
- normalizing the range of the features (feature scaling).

The following code performs both of these preprocessing steps.

In [61]:

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

```
Training Features shape:  (307511, 244)
Testing Features shape:  (48744, 243)
```

In [62]:

```
from sklearn.preprocessing import MinMaxScaler, Imputer
```

In [63]:

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

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

In [64]:

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

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

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

In [65]:

```
# Scale each feature to 0-1

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range = (0, 1))

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

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

In [66]:

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

```
Training data shape:  (307511, 243)
Testing data shape:  (48744, 243)
```

In [67]:

```
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)
print("the model has been trained......")
```

```
the model has been trained......
```

We want to predict the probabilities of not paying a loan, so we use the model predict.proba method. This returns an m x 2 array where m is the number of observations. The first column is the probability of the target being 0 and the second column is the probability of the target being 1 (so for a single row, the two columns must sum to 1). We want the probability the loan is not repaid, so we will select the second column.

In [68]:

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

In [69]:

```python
# Submission dataframe
submit = app_test[['SK_ID_CURR']]
submit['TARGET'] = log_reg_pred

submit.head()
```

Out[69]:

	SK_ID_CURR	TARGET
0	100001	0.087852
1	100005	0.163767
2	100013	0.110051
3	100028	0.077199
4	100038	0.151428

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 [70]:

```python
# Save the submission to a csv file
submit.to_csv('log_reg_baseline.csv', index = False)
```

The logistic regression baseline score around 0.67041 when submitted.

Applied Data Science Special…

Issuer: Coursera

PROVIDED BY Acclaim

Model-2

Improved Model: Random Forest

In [71]:

```python
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 [72]:

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

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  2.5min finished
```

Out[72]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=-1, oob_score=False, random_state=50, verbose=1,
                       warm_start=False)
```

In [73]:

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

In [74]:

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

```
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.6s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    1.4s finished
```

In [75]:

```
# 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)
```

This model should score around 0.67877 when submitted.

Make Predictions using Engineered Features

Model-3

Random Forest Model with Polynomial Features

In [76]:

```
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_j
```

```
obs = -1)
```

In [77]:

```
# 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:  1.3min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  3.0min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.3s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    0.7s finished
```

In [78]:

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

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

This model scored 0.60744 when submitted to the competition.

Model-4

Random forest with Domain Features

In [79]:

```
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_values_domain})

# 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:  1.0min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  2.3min finished
[Parallel(n_jobs=4)]: Using backend ThreadingBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:    0.4s
[Parallel(n_jobs=4)]: Done 100 out of 100 | elapsed:    1.2s finished
```

In [80]:

```
# 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 0.6799 when submitted which probably shows that the engineered features do not help in this model (however they do help in the Gradient Boosting Model at the end of the notebook).

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.

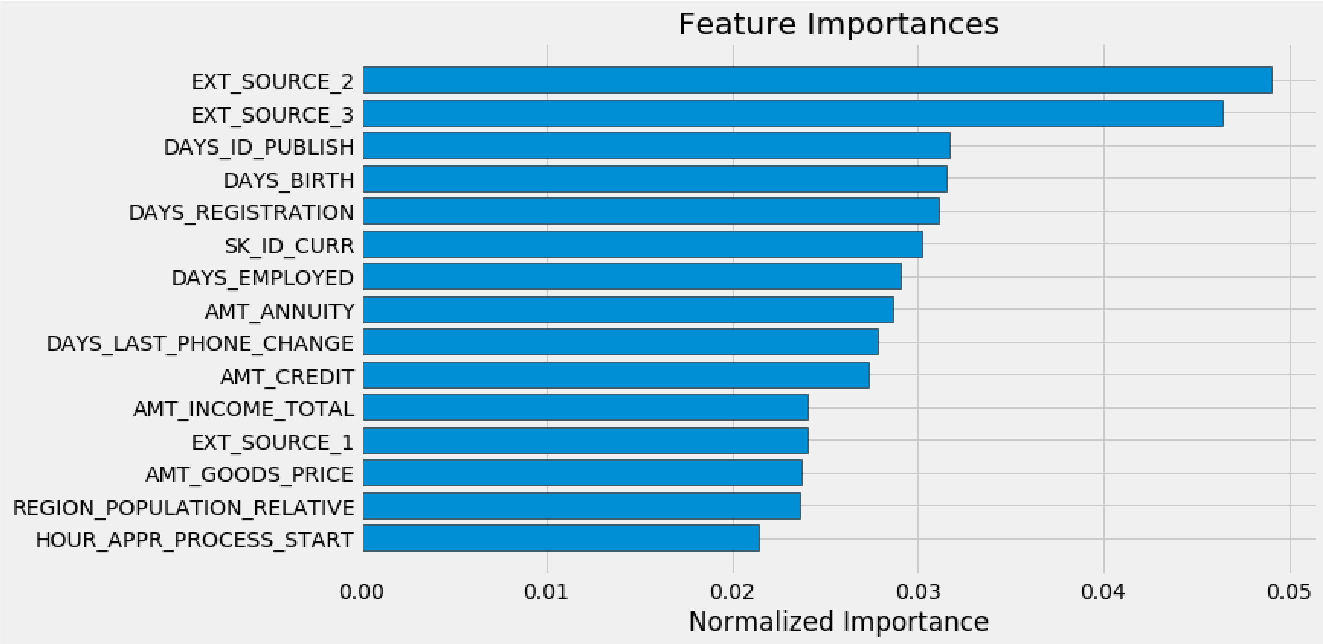
In [81]:

```
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 [ ]:

In [82]:

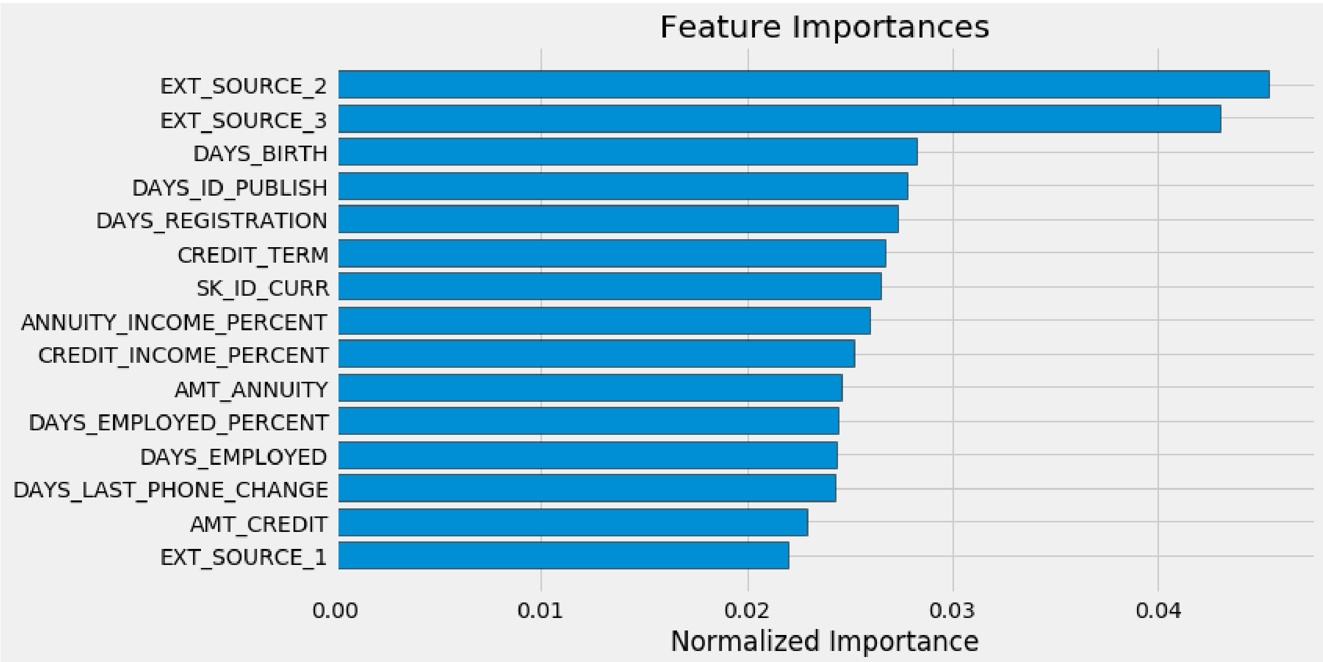
```
# Show the feature importances for the default features
feature_importances_sorted = plot_feature_importances(feature_importances)
```



- the most important features are those dealing with EXT_SOURCE and DAYS_BIRTH

In [83]:

```
feature_importances_domain_sorted = plot_feature_importances(feature_importances_domain)
```



In [84]:

```
#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.
```

baseline: Light Gradient Boosting Machine

In [85]:

```
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' f
or integer label encoding
            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((-1,)))

                # 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.n_splits

        # 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
```

Model-5

Light Gradient Boosting Machine

In [86]:

```
submission, fi, metrics = model(app_train, app_test)

print('done...')
```

```
Training Data Shape:  (307511, 242)
Testing Data Shape:  (48744, 242)
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.798935 train's binary_logloss: 0.547605 valid's auc: 0.754818 valid's
binary_logloss: 0.563207
Early stopping, best iteration is:
[177] train's auc: 0.795018 train's binary_logloss: 0.551417 valid's auc: 0.755057 valid's
binary_logloss: 0.565457
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.798518 train's binary_logloss: 0.548144 valid's auc: 0.758534 valid's
binary_logloss: 0.563479
Early stopping, best iteration is:
[217] train's auc: 0.801374 train's binary_logloss: 0.545314 valid's auc: 0.758609 valid's
binary_logloss: 0.561733
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.797543 train's binary_logloss: 0.549406 valid's auc: 0.762864 valid's
binary_logloss: 0.564299
Early stopping, best iteration is:
[231] train's auc: 0.802737 train's binary_logloss: 0.544391 valid's auc: 0.763044 valid's
binary_logloss: 0.561131
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.799107 train's binary_logloss: 0.547723 valid's auc: 0.757496 valid's
binary_logloss: 0.562014
Early stopping, best iteration is:
[183] train's auc: 0.796125 train's binary_logloss: 0.550639 valid's auc: 0.75759 valid's
binary_logloss: 0.563796
Training until validation scores don't improve for 100 rounds
```

```
[200] train's auc: 0.798268 train's binary_logloss: 0.548198 valid's auc: 0.758099 valid's
binary_logloss: 0.564499
Early stopping, best iteration is:
[227] train's auc: 0.802746 train's binary_logloss: 0.543868 valid's auc: 0.758251 valid's
binary_logloss: 0.561904
done...
```

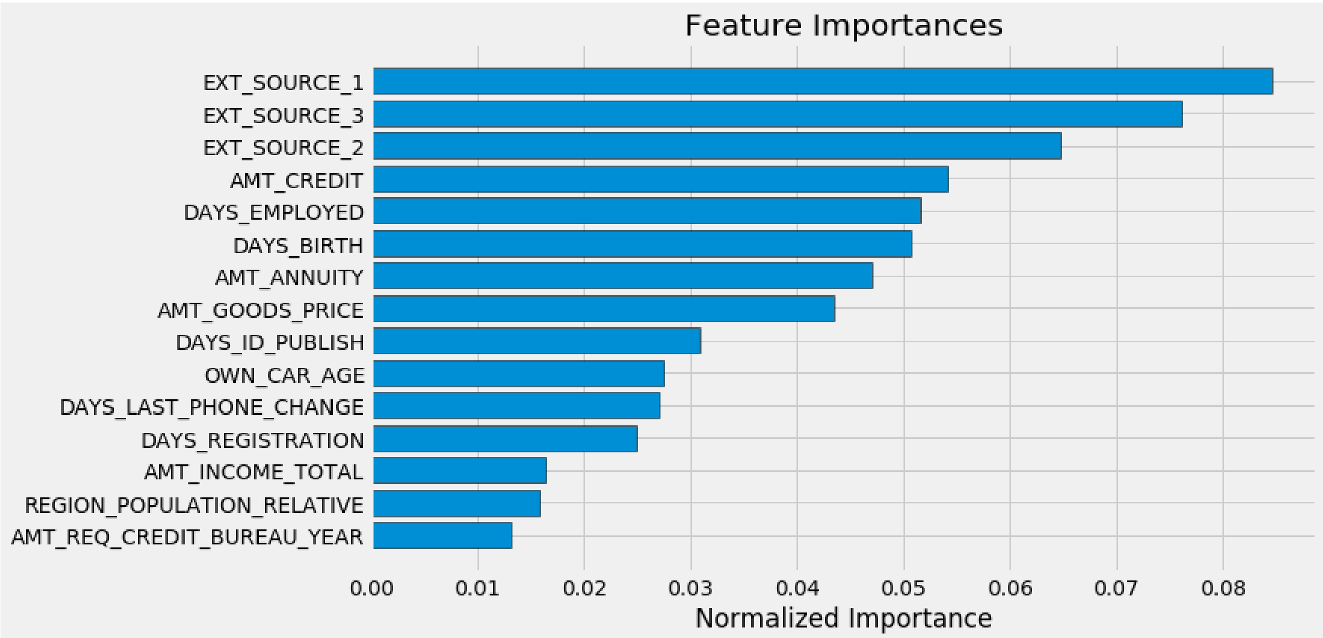
In [87]:

```
print('Baseline metrics')
print(metrics)
```

```
Baseline metrics
      fold     train     valid
0        0  0.795018  0.755057
1        1  0.801374  0.758609
2        2  0.802737  0.763044
3        3  0.796125  0.757590
4        4  0.802746  0.758251
5  overall  0.799600  0.758517
```

In [88]:

```
fi_sorted = plot_feature_importances(fi)
```



In [89]:

```
submission.to_csv('baseline_lgb.csv', index = False)
```

This submission should score about 0.73397 on the leaderboard.

Model-6

Light Gradient Boosting MachineTest the domain knolwedge features

In [90]:

```
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('done...')
```

```
Training Data Shape:  (307511, 246)
Testing Data Shape:  (48744, 246)
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.804523 train's binary_logloss: 0.54166 valid's auc: 0.762143 valid's
binary_logloss: 0.557463
Early stopping, best iteration is:
[295] train's auc: 0.819338 train's binary_logloss: 0.526606 valid's auc: 0.762798 valid's
binary_logloss: 0.548189
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.804311 train's binary_logloss: 0.542026 valid's auc: 0.765623 valid's
binary_logloss: 0.55807
Early stopping, best iteration is:
[230] train's auc: 0.809158 train's binary_logloss: 0.537055 valid's auc: 0.765988 valid's
binary_logloss: 0.554956
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.803588 train's binary_logloss: 0.54289 valid's auc: 0.770364 valid's
binary_logloss: 0.55776
Early stopping, best iteration is:
[210] train's auc: 0.805174 train's binary_logloss: 0.54134 valid's auc: 0.770412 valid's
binary_logloss: 0.556746
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.804487 train's binary_logloss: 0.542071 valid's auc: 0.765653 valid's
binary_logloss: 0.556352
Early stopping, best iteration is:
[262] train's auc: 0.815066 train's binary_logloss: 0.53137 valid's auc: 0.766316 valid's
binary_logloss: 0.549787
Training until validation scores don't improve for 100 rounds
[200] train's auc: 0.804527 train's binary_logloss: 0.541724 valid's auc: 0.764456 valid's
binary_logloss: 0.558821
Early stopping, best iteration is:
[235] train's auc: 0.810422 train's binary_logloss: 0.535826 valid's auc: 0.764517 valid's
binary_logloss: 0.555191
done...
```

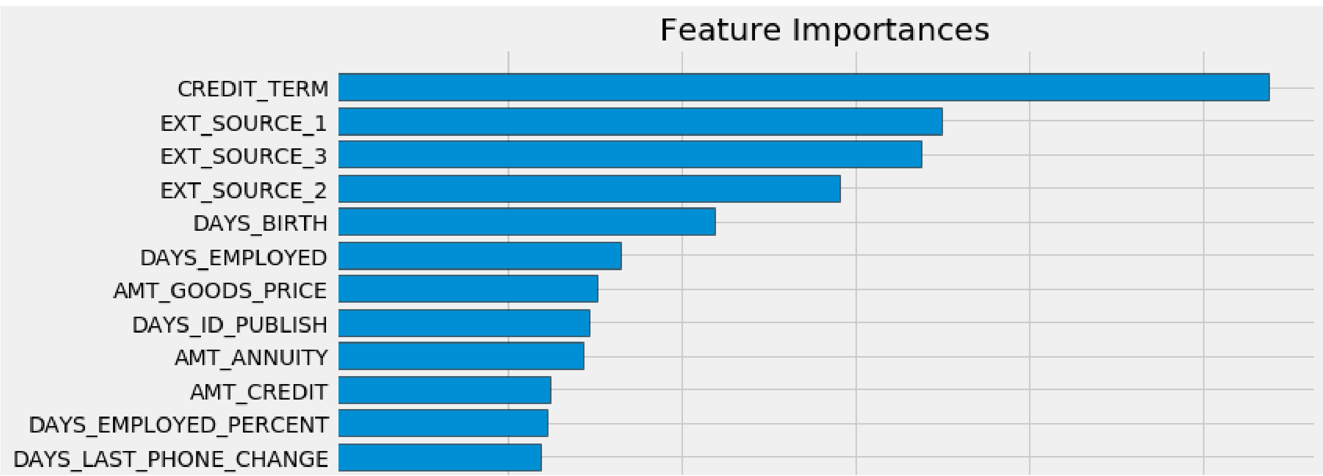
In [91]:

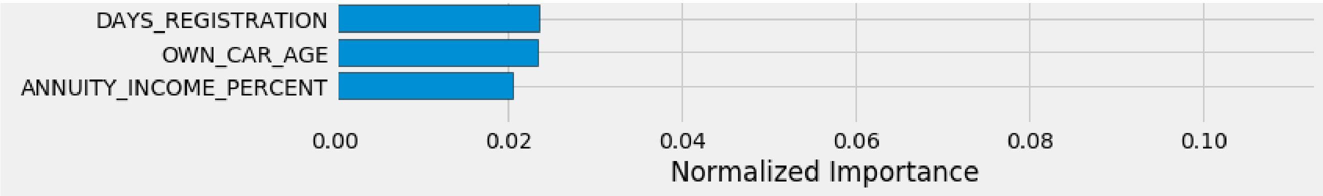
```
print('Baseline with domain knowledge features metrics')
print(metrics_domain)
```

```
Baseline with domain knowledge features metrics
      fold     train     valid
0        0  0.819338  0.762798
1        1  0.809158  0.765988
2        2  0.805174  0.770412
3        3  0.815066  0.766316
4        4  0.810422  0.764517
5  overall  0.811832  0.765993
```

In [92]:

```
fi_sorted = plot_feature_importances(fi_domain)
```





In [93]:

```
submission_domain.to_csv('baseline_lgb_domain_features.csv', index = False)
```

This model scores about 0.75459 when submitted to the public leaderboard