Feature Selection

Python notebook using data from multiple data sources · 15,803 views · multiple data sources

Submission
Ran successfully
Submitted by y_test is all we need :) 8 months ago

Private Score 0.78414
Public Score 0.78205

Introduction: Feature Selection

In this notebook we will apply feature engineering to the manual engineered features built in two previous kernels. We will reduce the number of features using several methods and then we will test the performance of the features using a fairly basic gradient boosting machine model.

The main takeaways from this notebook are:

- Going from 1465 total features to 536 and an AUC ROC of 0.783 on the public leaderboard
- A further optional step to go to 342 features and an AUC ROC of 0.782

The full set of features was built in Part One and Part Two of Manual Feature Engineering

We will use three methods for feature selection:

1. Remove collinear features
2. Remove features with greater than a threshold percentage of missing values
3. Keep only the most relevant features using feature importances from a model

We will also take a look at an example of applying PCA although we will not use this method for feature reduction.

Standard imports for data science work. The LightGBM library is used for the gradient boosting machine.

In [1]:
```
# pandas and numpy for data manipulation
import pandas as pd
import numpy as np

# featuretools for automated feature engineering
import featuretools as ft

# matplotlit and seaborn for visualizations
import matplotlib.pyplot as plt
plt.rcParams['font.size'] = 22
import seaborn as sns

# Suppress warnings from pandas
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
```

- `train_bureau` is the training features built manually using the `bureau` and `bureau_balance` data
- `train_previous` is the training features built manually using the `previous`, `cash`, `credit`, and `installments` data

We first will see how many features we built over the manual engineering process. Here we use a couple of set operations to find the columns that are only in the `bureau`, only in the `previous`, and in both dataframes, indicating that there are `original` features from the `application` dataframe. Here we are working with a small subset of the data in order to not overwhelm the kernel. This code has also been run on the full dataset (we will take a look at some of the results).

In [2]:
```
# Read in data
train_bureau = pd.read_csv('../input/home-credit-manual-engineered-features/train_bureau_raw.csv', nrows = 1000)
test_bureau = pd.read_csv('../input/home-credit-manual-engineered-features/test_bureau_raw.csv', nrows = 1000)

train_previous = pd.read_csv('../input/home-credit-manual-engineered-features/train_previous_raw.csv', nrows = 1000)
test_previous = pd.read_csv('../input/home-credit-manual-engineered-features/test_previous_raw.csv', nrows = 1000)

# All columns in dataframes
bureau_columns = list(train_bureau.columns)
previous_columns = list(train_previous.columns)
```

In [3]:
```
# Bureau only features
bureau_features = list(set(bureau_columns) - set(previous_columns))

# Previous only features
previous_features = list(set(previous_columns) - set(bureau_columns))

# Original features will be in both datasets
original_features = list(set(previous_columns) & set(bureau_columns))

print('There are %d original features ' % len(original_features))
```

```
print('There are %d bureau and bureau balance features.' % len(bureau_features))
print('There are %d previous Home Credit loan features.' % len(previous_features))
```

```
There are 123 original features.
There are 179 bureau and bureau balance features.
There are 1043 previous Home Credit loan features.
```

That gives us the number of features in each dataframe. Now we want to combine the data without creating any duplicate rows.

In [4]:
```
train_labels = train_bureau['TARGET']
previous_features.append('SK_ID_CURR')

train_ids = train_bureau['SK_ID_CURR']
test_ids = test_bureau['SK_ID_CURR']

# Merge the dataframes avoiding duplicating columns by subsetting train_previous
train = train_bureau.merge(train_previous[previous_features], on = 'SK_ID_CURR')
test = test_bureau.merge(test_previous[previous_features], on = 'SK_ID_CURR')

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

```
Training shape:  (1000, 1345)
Testing shape:  (1000, 1344)
```

Next we want to one-hot encode the dataframes. This doesn't give the full features since we are only working with a sample of the data and this will not create as many columns as one-hot encoding the entire dataset would. Doing this to the full dataset results in 1465 features.

An important note in the code cell is where we **align the dataframes by the columns.** This ensures we have the same columns in the training and testing datasets.

In [5]:
```
# One hot encoding
train = pd.get_dummies(train)
test = pd.get_dummies(test)

# Match the columns in the dataframes
train, test = train.align(test, join = 'inner', axis = 1)
print('Training shape: ', train.shape)
print('Testing shape: ', test.shape)
```

```
Training shape:  (1000, 1447)
Testing shape:  (1000, 1447)
```

When we do this to the full dataset, we get **1465** features.

Admit and Correct Mistakes!

When doing manual feature engineering, I accidentally created some columns derived from the client id, `SK_ID_CURR`. As this is a unique identifier for each client, it should not have any predictive power, and we would not want to build a model trained on this "feature". Let's remove any columns built on the `SK_ID_CURR`.

In [6]:
```
cols_with_id = [x for x in train.columns if 'SK_ID_CURR' in x]
cols_with_bureau_id = [x for x in train.columns if 'SK_ID_BUREAU' in x]
cols_with_previous_id = [x for x in train.columns if 'SK_ID_PREV' in x]
print('There are %d columns that contain SK_ID_CURR' % len(cols_with_id))
print('There are %d columns that contain SK_ID_BUREAU' % len(cols_with_bureau_id))
print('There are %d columns that contain SK_ID_PREV' % len(cols_with_previous_id))

train = train.drop(columns = cols_with_id)
test = test.drop(columns = cols_with_id)
print('Training shape: ', train.shape)
print('Testing shape: ', test.shape)
```

```
There are 49 columns that contain SK_ID_CURR
There are 0 columns that contain SK_ID_BUREAU
There are 0 columns that contain SK_ID_PREV
Training shape:  (1000, 1398)
Testing shape:  (1000, 1398)
```

After applying this to the full dataset, we end up with **1416** features. More features might seem like a good thing, and they can be if they help our model learn. However, irrelevant features, highly correlated features, and missing values can prevent the model from learning and decrease generalization performance on the testing data. Therefore, we perform feature selection to keep only the most useful variables.

We will start feature selection by focusing on collinear variables.

Remove Collinear Variables

Collinear variables are those which are highly correlated with one another. These can decrease the model's availablility to learn, decrease model interpretability, and decrease generalization performance on the test set. Clearly, these are three things we want to increase, so removing collinear variables is a useful step. We will establish an admittedly arbitrary threshold for removing collinear variables, and then remove one out of any pair of variables that is above that threshold.

The code below identifies the highly correlated variables based on the absolute magnitude of the Pearson correlation coefficient being greater than 0.9. Again, this is not entirely accurate since we are dealing with such a limited section of the data. This code is for illustration purposes, but if we read in the entire dataset, it would work (if the kernels allowed it)!

This code is adapted from work by Chris Albon.

Identify Correlated Variables

Identify Correlated Variables

In [7]:
```
# Threshold for removing correlated variables
threshold = 0.9

# Absolute value correlation matrix
corr_matrix = train.corr().abs()
corr_matrix.head()
```

Out[7]:

	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE	REGION_POPUL
CNT_CHILDREN	1.000000	0.055960	0.036836	0.055732	0.035851	0.060210
AMT_INCOME_TOTAL	0.055960	1.000000	0.429317	0.491143	0.439981	0.184339
AMT_CREDIT	0.036836	0.429317	1.000000	0.797209	0.986046	0.074287
AMT_ANNUITY	0.055732	0.491143	0.797209	1.000000	0.799121	0.106685
AMT_GOODS_PRICE	0.035851	0.439981	0.986046	0.799121	1.000000	0.073531

In [8]:
```
# Upper triangle of correlations
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
upper.head()
```

Out[8]:

	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE	REGION_POPUL
CNT_CHILDREN	NaN	0.05596	0.036836	0.055732	0.035851	0.060210
AMT_INCOME_TOTAL	NaN	NaN	0.429317	0.491143	0.439981	0.184339
AMT_CREDIT	NaN	NaN	NaN	0.797209	0.986046	0.074287
AMT_ANNUITY	NaN	NaN	NaN	NaN	0.799121	0.106685
AMT_GOODS_PRICE	NaN	NaN	NaN	NaN	NaN	0.073531

In [9]:
```
# Select columns with correlations above threshold
to_drop = [column for column in upper.columns if any(upper[column] > threshold)]

print('There are %d columns to remove.' % (len(to_drop)))
```

```
There are 584 columns to remove.
```

Drop Correlated Variables

In [10]:
```
train = train.drop(columns = to_drop)
test = test.drop(columns = to_drop)

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

```
Training shape:  (1000, 814)
Testing shape:  (1000, 814)
```

Applying this on the entire dataset **results in 538 collinear features** removed.

This has reduced the number of features singificantly, but it is likely still too many. At this point, we'll read in the full dataset after removing correlated variables for further feature selection.

The full datasets (after removing correlated variables) are available in `m_train_combined.csv` and `m_test_combined.csv`.

Read in Full Dataset

Now we are ready to move on to the full set of features. These were built by applying the above steps to the entire `train_bureau` and `train_previous` files (you can do the same if you want and have the computational resources)!

In [11]:
```
train = pd.read_csv('../input/home-credit-manual-engineered-features/m_train_combined.csv')
test = pd.read_csv('../input/home-credit-manual-engineered-features/m_test_combined.csv')
```

In [12]:
```
print('Training set full shape: ', train.shape)
print('Testing set full shape: ' , test.shape)
```

```
Training set full shape:  (307511, 865)
Testing set full shape:  (48744, 864)
```

Remove Missing Values

A relatively simple choice of feature selection is removing missing values. Well, it seems simple, at least until we have to decide what percentage of missing values is the minimum threshold for removing a column. Like many choices in machine learning, there is no right answer, and not even a general rule of thumb for making this choice. In this implementation, if any columns have greater than 75% missing values, they will be removed.

Most models (including those in Sk-Learn) cannot handle missing values, so we will have to fill these in before machine learning. The Gradient Boosting Machine (at least in LightGBM) can handle missing values. Imputing missing values always makes me a little uncomfortable because we are adding information that actually isn't in the dataset. Since we are going to be evaluating several models (in a later notebook), we will have to use some form of imputation. For now, we will focus on removing columns above the threshold.

In [13]:
```
# Train missing values (in percent)
train_missing = (train.isnull().sum() / len(train)).sort_values(ascending = False)
train_missing.head()
```

Out[13]:
```
client_credit_AMT_PAYMENT_CURRENT_min_min            0.801438
client_credit_AMT_PAYMENT_CURRENT_mean_max           0.801438
client_credit_AMT_DRAWINGS_OTHER_CURRENT_min_mean    0.801178
client_credit_CNT_DRAWINGS_OTHER_CURRENT_min_max     0.801178
client_credit_AMT_DRAWINGS_ATM_CURRENT_mean_max      0.801178
dtype: float64
```

In [14]:
```
# Test missing values (in percent)
test_missing = (test.isnull().sum() / len(test)).sort_values(ascending = False)
test_missing.head()
```

Out[14]:
```
client_credit_CNT_DRAWINGS_OTHER_CURRENT_max_max     0.773223
client_credit_CNT_DRAWINGS_POS_CURRENT_mean_max      0.773223
client_credit_CNT_DRAWINGS_ATM_CURRENT_min_max       0.773223
client_credit_AMT_DRAWINGS_OTHER_CURRENT_mean_min    0.773223
client_credit_CNT_DRAWINGS_POS_CURRENT_min_mean      0.773223
dtype: float64
```

In [15]:
```
# Identify missing values above threshold
train_missing = train_missing.index[train_missing > 0.75]
test_missing = test_missing.index[test_missing > 0.75]

all_missing = list(set(set(train_missing) | set(test_missing)))
print('There are %d columns with more than 75%% missing values' % len(all_missing))
```

```
There are 19 columns with more than 75% missing values
```

Let's drop the columns, one-hot encode the dataframes, and then align the columns of the dataframes.

In [16]:
```
# Need to save the labels because aligning will remove this column
train_labels = train["TARGET"]
train_ids = train['SK_ID_CURR']
test_ids = test['SK_ID_CURR']

train = pd.get_dummies(train.drop(columns = all_missing))
test = pd.get_dummies(test.drop(columns = all_missing))

train, test = train.align(test, join = 'inner', axis = 1)

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

```
Training set full shape:  (307511, 845)
Testing set full shape:  (48744, 845)
```

In [17]:
```
train = train.drop(columns = ['SK_ID_CURR'])
test = test.drop(columns = ['SK_ID_CURR'])
```

Feature Selection through Feature Importances

The next method we can employ for feature selection is to use the feature importances of a model. Tree-based models (and consequently ensembles of trees) can determine an "importance" for each feature by measuring the reduction in impurity for including the feature in the model. I'm not really sure what that means (any explanations would be welcome) and the absolute value of the importance can be difficult to interpret. However, the relative value of the importances can be used as an approximation of the "relevance" of different features in a model. Moreover, we can use the feature importances to remove features that the model does not consider important.

One method for doing this automatically is the Recursive Feature Elimination method in Scikit-Learn. This accepts an estimator (one that either returns feature weights such as a linear regression, or feature importances such as a random forest) and a desired number of features. In then fits the model repeatedly on the data and iteratively removes the lowest importance features until the desired number of features is left. This means we have another arbitrary hyperparameter to use in out pipeline: the number of features to keep!

Instead of doing this automatically, we can perform our own feature removal by first removing all zero importance features from the model. If this leaves too many features, then we can consider removing the features with the lowest importance. We will use a Gradient Boosted Model from the LightGBM library to assess feature importances. If you're used to the Scikit-Learn library, the LightGBM library has an API that makes deploying the model very similar to using a Scikit-Learn model.

Since the LightGBM model does not need missing values to be imputed, we can directly `fit` on the training data. We will use Early Stopping to determine the optimal number of iterations and run the model twice, averaging the feature importances to try and avoid overfitting to a certain set of features.

In [18]:
```
# Initialize an empty array to hold feature importances
feature_importances = np.zeros(train.shape[1])

# Create the model with several hyperparameters
model = lgb.LGBMClassifier(objective='binary', boosting_type = 'goss', n_estimators = 10000, cl
ass_weight = 'balanced')
```

In [19]:
```
# Fit the model twice to avoid overfitting
for i in range(2):

    # Split into training and validation set
    train_features, valid_features, train_y, valid_y = train_test_split(train, train_labels, te
st_size = 0.25, random_state = i)
```

```
    # Train using early stopping
    model.fit(train_features, train_y, early_stopping_rounds=100, eval_set = [(valid_features,
valid_y)],
              eval_metric = 'auc', verbose = 200)

    # Record the feature importances
    feature_importances += model.feature_importances_
```

```
Training until validation scores don't improve for 100 rounds.
[200]   valid_0's auc: 0.782469
Early stopping, best iteration is:
[196]   valid_0's auc: 0.782633
Training until validation scores don't improve for 100 rounds.
[200]   valid_0's auc: 0.78326
Early stopping, best iteration is:
[152]   valid_0's auc: 0.784236
```

In [20]:

```
# Make sure to average feature importances!
feature_importances = feature_importances / 2
feature_importances = pd.DataFrame({'feature': list(train.columns), 'importance': feature_impor
tances}).sort_values('importance', ascending = False)

feature_importances.head()
```

Out[20]:

	feature	importance
24	EXT_SOURCE_1	167.5
25	EXT_SOURCE_2	141.5
26	EXT_SOURCE_3	136.0
368	client_installments_AMT_PAYMENT_min_sum	120.5
5	DAYS_BIRTH	98.5

In [21]:

```
# Find the features with zero importance
zero_features = list(feature_importances[feature_importances['importance'] == 0.0]['feature'])
print('There are %d features with 0.0 importance' % len(zero_features))
feature_importances.tail()
```

```
There are 271 features with 0.0 importance
```

Out[21]:

	feature	importance
348	previous_loans_RATE_INTEREST_PRIMARY_sum	0.0
352	client_cash_NAME_CONTRACT_STATUS_XNA_count_nor...	0.0
635	previous_loans_NAME_CASH_LOAN_PURPOSE_Buying a...	0.0
353	previous_loans_PRODUCT_COMBINATION_POS mobile ...	0.0
843	EMERGENCYSTATE_MODE_Yes	0.0

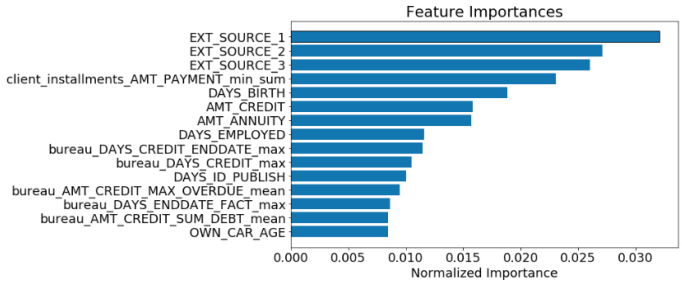
We see that one of our features made it into the top 5 most important! That's a good sign for all of our hard work making the features. It also looks like many of the features we made have literally 0 importance. For the gradient boosting machine, features with 0 importance are not used at all to make any splits. Therefore, we can remove these features from the model with no effect on performance (except for faster training).

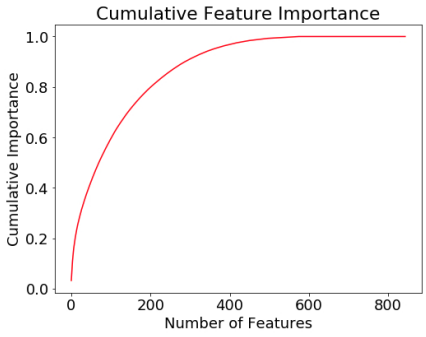
In [22]:

```
def plot_feature_importances(df, threshold = 0.9):
    """
    Plots 15 most important features and the cumulative importance of features.
    Prints the number of features needed to reach threshold cumulative importance.

    Parameters
    --------
    df : dataframe
        Dataframe of feature importances. Columns must be feature and importance
    threshold : float, default = 0.9
        Threshold for prining information about cumulative importances

    Return
    --------
    df : dataframe
        Dataframe ordered by feature importances with a normalized column (sums to 1)
        and a cumulative importance column

    """

    plt.rcParams['font.size'] = 18

    # 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()
    df['cumulative_importance'] = np.cumsum(df['importance_normalized'])

    # 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()

    # Cumulative importance plot
    plt.figure(figsize = (8, 6))
    plt.plot(list(range(len(df))), df['cumulative_importance'], 'r-')
    plt.xlabel('Number of Features'); plt.ylabel('Cumulative Importance');
    plt.title('Cumulative Feature Importance');
    plt.show();

    importance_index = np.min(np.where(df['cumulative_importance'] > threshold))
    print('%d features required for %0.2f of cumulative importance' % (importance_index + 1, th
reshold))

    return df
```

In [23]:
```
norm_feature_importances = plot_feature_importances(feature_importances)
```






```
288 features required for 0.90 of cumulative importance
```

Let's remove the features that have zero importance.

In [24]:
```
train = train.drop(columns = zero_features)
test = test.drop(columns = zero_features)

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

```
Training shape:  (307511, 573)
Testing shape:  (48744, 573)
```

At this point, we can re-run the model to see if it identifies any more features with zero importance. In a way, we are implementing our own form of recursive feature elimination. Since we are repeating work, we should probably put the zero feature importance identification code in a function.

In [25]:
```
def identify_zero_importance_features(train, train_labels, iterations = 2):
    """
    Identify zero importance features in a training dataset based on the
    feature importances from a gradient boosting model.

    Parameters
    --------
    train : dataframe
        Training features

    train_labels : np.array
        Labels for training data

    iterations : integer, default = 2
        Number of cross validation splits to use for determining feature importances
    """

    # Initialize an empty array to hold feature importances
    feature_importances = np.zeros(train.shape[1])

    # Create the model with several hyperparameters
    model = lgb.LGBMClassifier(objective='binary', boosting_type = 'goss', n_estimators = 10000
, class_weight = 'balanced')
```

```
    # Fit the model multiple times to avoid overfitting
    for i in range(iterations):

        # Split into training and validation set
        train_features, valid_features, train_y, valid_y = train_test_split(train, train_labels
, test_size = 0.25, random_state = i)

        # Train using early stopping
        model.fit(train_features, train_y, early_stopping_rounds=100, eval_set = [(valid_featur
es, valid_y)],
                  eval_metric = 'auc', verbose = 200)

        # Record the feature importances
        feature_importances += model.feature_importances_ / iterations

    feature_importances = pd.DataFrame({'feature': list(train.columns), 'importance': feature_i
mportances}).sort_values('importance', ascending = False)

    # Find the features with zero importance
    zero_features = list(feature_importances[feature_importances['importance'] == 0.0]['featur
e'])
    print('\nThere are %d features with 0.0 importance' % len(zero_features))

    return zero_features, feature_importances
```

In [26]:

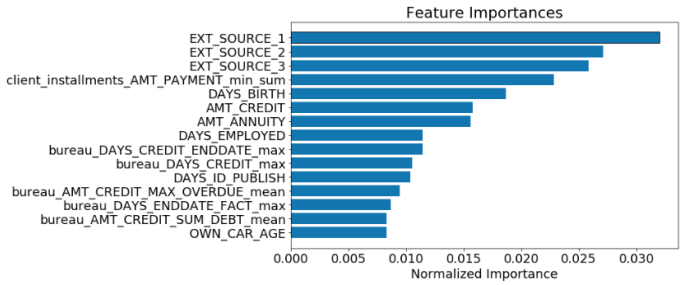
```
second_round_zero_features, feature_importances = identify_zero_importance_features(train, trai
n_labels)
```

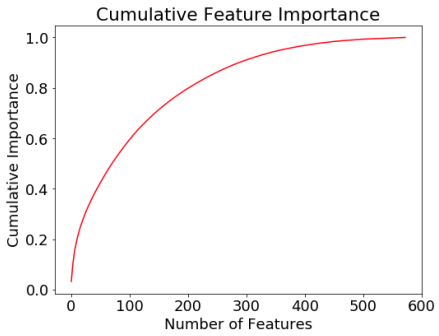
```
Training until validation scores don't improve for 100 rounds.
[200]   valid_0's auc: 0.782664
Early stopping, best iteration is:
[201]   valid_0's auc: 0.782809
Training until validation scores don't improve for 100 rounds.
[200]   valid_0's auc: 0.783292
Early stopping, best iteration is:
[152]   valid_0's auc: 0.784236

There are 0 features with 0.0 importance
```

There are now no 0 importance features left (I guess we should have expected this). If we want to remove more features, we will have to start with features that have a non-zero importance. One way we could do this is by retaining enough features to account for a threshold percentage of importance, such as 95%. At this point, let's keep enough features to account for 95% of the importance. Again, this is an arbitrary decision!

In [27]:

```
norm_feature_importances = plot_feature_importances(feature_importances, threshold = 0.95)
```






```
360 features required for 0.95 of cumulative importance
```

We can keep only the features needed for 95% importance. This step seems to me to have the greatest chance of harming the model's learning ability, so rather than changing the original dataset, we will make smaller copies. Then, we can test both versions of the data to see if the extra feature removal step is worthwhile.

In [28]:

```
# Threshold for cumulative importance
threshold = 0.95

# Extract the features to keep
features_to_keep = list(norm_feature_importances[norm_feature_importances['cumulative_importanc
e'] < threshold]['feature'])

# Create new datasets with smaller features
train_small = train[features_to_keep]
```

```
test_small = test[features_to_keep]
```

In [29]:

```
train_small['TARGET'] = train_labels
train_small['SK_ID_CURR'] = train_ids
test_small['SK_ID_CURR'] = test_ids

train_small.to_csv('m_train_small.csv', index = False)
test_small.to_csv('m_test_small.csv', index = False)
```

Test New Featuresets

The last step of feature removal we did seems like it may have the potential to hurt the model the most. Therefore we want to test the effect of this removal. To do that, we can use a standard model and change the features.

We will use a fairly standard LightGBM model, similar to the one we used for feature selection. The main difference is this model uses five-fold cross validation for training and we use it to make predictions. There's a lot of code here, but that's because I included documentation and a few extras (such as feature importances) that aren't strictly necessary. For now, understanding the entire model isn't critical, just know that we are using the same model with two different datasets to see which one performs the best.

In [30]:

```
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 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 = False, 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', boosting_type='gos
s',
                                   class_weight = 'balanced', learning_rate = 0.05,
                                   reg_alpha = 0.1, reg_lambda = 0.1, 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_i
teration)[:, 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_importa
nce_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
```

Test "Full" Dataset

This is the expanded dataset. To recap the process to make this dataset we:

- Removed collinear features as measured by the correlation coefficient greater than 0.9
- Removed any columns with greater than 80% missing values in the train or test set
- Removed all features with non-zero feature importances

In [31]:

```
train['TARGET'] = train_labels
train['SK_ID_CURR'] = train_ids
```

```
test['SK_ID_CURR'] = test_ids

submission, feature_importances, metrics = model(train, test)
```

```
Training Data Shape:  (307511, 573)
Testing Data Shape:  (48744, 573)
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.781367   train's auc: 0.828769
[400]   valid's auc: 0.782692   train's auc: 0.866597
Early stopping, best iteration is:
[340]   valid's auc: 0.783203   train's auc: 0.856398
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.784772   train's auc: 0.828511
[400]   valid's auc: 0.786131   train's auc: 0.866562
Early stopping, best iteration is:
[453]   valid's auc: 0.786335   train's auc: 0.875061
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.77768    train's auc: 0.829992
[400]   valid's auc: 0.780077   train's auc: 0.86754
Early stopping, best iteration is:
[420]   valid's auc: 0.78027    train's auc: 0.870681
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.783364   train's auc: 0.828584
[400]   valid's auc: 0.784053   train's auc: 0.866526
Early stopping, best iteration is:
[358]   valid's auc: 0.784283   train's auc: 0.859476
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.784376   train's auc: 0.828393
[400]   valid's auc: 0.785964   train's auc: 0.866387
Early stopping, best iteration is:
[339]   valid's auc: 0.786298   train's auc: 0.85595
```

In [32]:
```
metrics
```

Out[32]:

	fold	train	valid
0	0	0.856398	0.783203
1	1	0.875061	0.786335
2	2	0.870681	0.780270
3	3	0.859476	0.784283
4	4	0.855950	0.786298
5	overall	0.863513	0.783973

In [33]:
```
submission.to_csv('selected_features_submission.csv', index = False)
```

The full features after feature selection score **0.783** when submitted to the public leaderboard.

Test "Small" Dataset

The small dataset requires one additional step over the ful l dataset:

- Keep only features needed to reach 95% cumulative importance in the gradient boosting machine

In [34]:
```
submission_small, feature_importances_small, metrics_small = model(train_small, test_small)
```

```
Training Data Shape:  (307511, 359)
Testing Data Shape:  (48744, 359)
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.781338   train's auc: 0.828914
[400]   valid's auc: 0.783637   train's auc: 0.866698
Early stopping, best iteration is:
[477]   valid's auc: 0.784184   train's auc: 0.878738
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.784262   train's auc: 0.828316
[400]   valid's auc: 0.786618   train's auc: 0.86647
[600]   valid's auc: 0.786229   train's auc: 0.894735
Early stopping, best iteration is:
[552]   valid's auc: 0.786815   train's auc: 0.888692
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.777036   train's auc: 0.830146
[400]   valid's auc: 0.780154   train's auc: 0.86792
Early stopping, best iteration is:
[439]   valid's auc: 0.78031    train's auc: 0.874029
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.782512   train's auc: 0.828736
[400]   valid's auc: 0.783394   train's auc: 0.866127
Early stopping, best iteration is:
[312]   valid's auc: 0.783844   train's auc: 0.851267
Training until validation scores don't improve for 100 rounds.
[200]   valid's auc: 0.783516   train's auc: 0.82822
[400]   valid's auc: 0.785809   train's auc: 0.865933
Early stopping, best iteration is:
[458]   valid's auc: 0.786073   train's auc: 0.874865
```

In [35]:
```
metrics_small
```

Out[35]:

	fold	train	valid
0	0	0.878738	0.784184
1	1	0.888692	0.786815
2	2	0.874029	0.780310
3	3	0.851267	0.783844

4	4	0.874865	0.786073
5	overall	0.873518	0.784121

In [36]:
```
submission_small.to_csv('selected_features_small_submission.csv', index = False)
```

The smaller featureset scores **0.782** when submitted to the public leaderboard.

Other Options for Dimensionality Reduction

We only covered a small portion of the techniques used for feature selection/dimensionality reduction. There are many other methods such as:

- PCA: Principle Components Analysis (PCA)
- ICA: Independent Components Analysis (ICA)
- Manifold learning: also called non-linear dimensionality reduction

PCA is a great method for reducing the number of features provided that you do not care about model interpretability. It projects the original set of features onto a lower dimension, in the process, eliminating any physical representation behind the features. Here's a pretty thorough introduction to the math for anyone interested. PCA also assumes that the data is Gaussian distributed, which may not be the case, especially when dealing with real-world human generated data.

ICA representations also obscure any physical meaning behind the variables and presevere the most "independent" dimensions of the data (which is different than the dimensions with the most variance).

Manifold learning is more often used for low-dimensional visualizations (such as with T-SNE or LLE) rather than for dimensionality reduction for a classifier. These methods are heavily dependent on several hyperparameters and are not deterministic which means that there is no way to apply it to new data (in other words you cannot `fit` it to the training data and then separately `transform` the testing data). The learned representation of a dataset will change every time you apply manifold learning so it is not generally a stable method for feature selection.

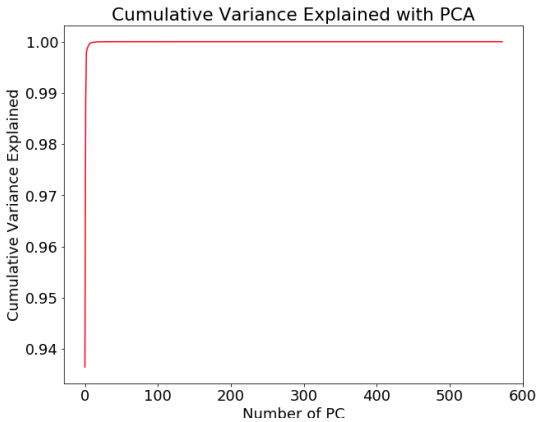
PCA Example

We can go through a quick example to show how PCA is implemented. Without going through too many details, PCA finds a new set of axis (the principal components) that maximize the amount of variance captured in the data. The original data is then projected down onto these principal components. The idea is that we can use fewer principal components than the original number of features while still capturing most of the variance. PCA is implemented in Scikit-Learn in the same way as preprocessing methods. We can either select the number of new components, or the fraction of variance we want explained in the data. If we pass in no argument, the number of principal components will be the same as the number of original features. We can then use the `variance_explained_ratio_` to determine the number of components needed for different threshold of variance retained.

In [37]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import Imputer
from sklearn.pipeline import Pipeline

# Make sure to drop the ids and target
train = train.drop(columns = ['SK_ID_CURR', 'TARGET'])
test = test.drop(columns = ['SK_ID_CURR'])

# Make a pipeline with imputation and pca
pipeline = Pipeline(steps = [('imputer', Imputer(strategy = 'median')),
             ('pca', PCA())])

# Fit and transform on the training data
train_pca = pipeline.fit_transform(train)

# transform the testing data
test_pca = pipeline.transform(test)
```

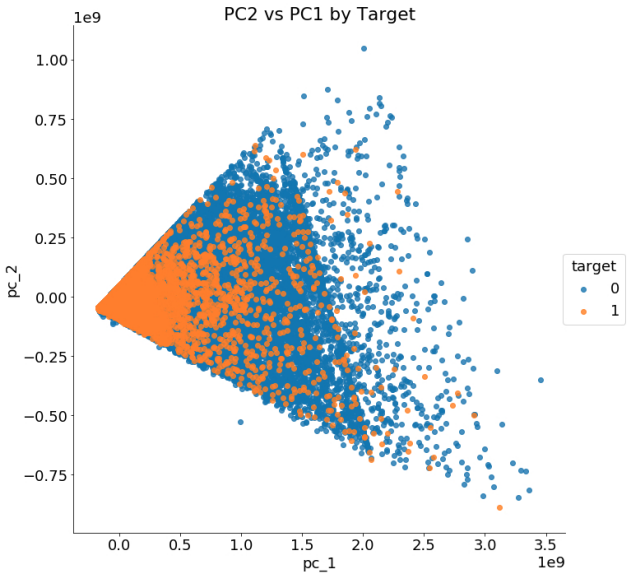
```
/opt/conda/lib/python3.6/site-packages/sklearn/utils/deprecation.py:58: DeprecationWarning: Cla
ss Imputer is deprecated; Imputer was deprecated in version 0.20 and will be removed in 0.22. I
mport impute.SimpleImputer from sklearn instead.
  warnings.warn(msg, category=DeprecationWarning)
```

In [38]:
```
# Extract the pca object
pca = pipeline.named_steps['pca']

# Plot the cumulative variance explained

plt.figure(figsize = (10, 8))
plt.plot(list(range(train.shape[1])), np.cumsum(pca.explained_variance_ratio_), 'r-')
plt.xlabel('Number of PC'); plt.ylabel('Cumulative Variance Explained');
plt.title('Cumulative Variance Explained with PCA');
```



We only need a few prinicipal components to account for the majority of variance in the data. We can use the first two principal components to visualize the entire dataset. We will color the datapoints by the value of the target to see if using two principal components clearly separates the classes.

In [39]:
```
# Dataframe of pca results
pca_df = pd.DataFrame({'pc_1': train_pca[:, 0], 'pc_2': train_pca[:, 1], 'target': train_labels
})

# Plot pc2 vs pc1 colored by target
sns.lmplot('pc_1', 'pc_2', data = pca_df, hue = 'target', fit_reg=False, size = 10)
plt.title('PC2 vs PC1 by Target');
```



In [40]:
```
print('2 principal components account for {:.4f}% of the variance.'.format(100 * np.sum(pca.exp
lained_variance_ratio_[:2])))
```

```
2 principal components account for 98.7908% of the variance.
```

Even though we have accounted for most of the variance, that does not mean the pca decomposition makes the problem of identifying loans repaid vs not repaid any easier. PCA does not consider the value of the label when projecting the features to a lower dimension. Feel free to try a classifier on top of this data, but when I have done so, I noticed that it was not very accurate.

Conclusions

In this notebook we employed a number of feature selection methods. These methods are necessary to reduce the number of features to increase model interpretability, decrease model runtime, and increase generalization performance on the test set. The methods of feature selection we used are:

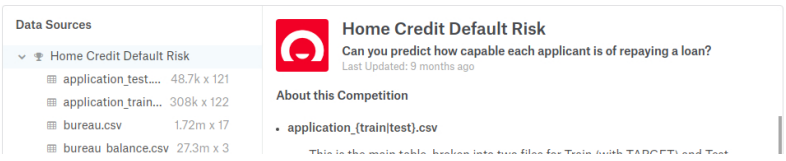
1. Remove highly collinear variables as measured by a correlation coefficient greater than 0.9
2. Remove any columns with more than 75% missing values.
3. Remove any features with a zero importance as determined by a gradient boosting machine.
4. (Optional) keep only enough features to account for 95% of the importance in the gradient boosting machine.

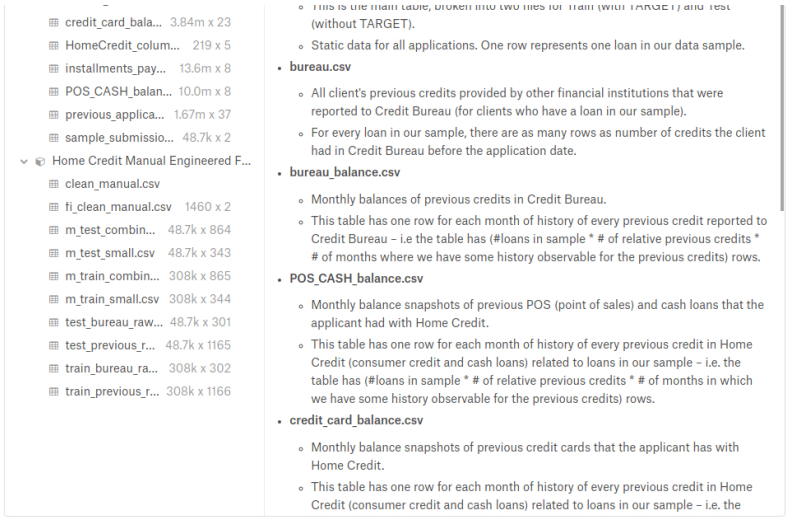
Using the first three methods, we reduced the number of features from **1465** to **536** with a 5-fold cv AUC ROC score of 0.7838 and a public leaderboard score of 0.783.

After applying the fourth method, we end up with 342 features with a 5-fold cv AUC SCORE of 0.7482 and a public leaderboard score of 0.782.

Going forward, we might actually want to add *more* features except this time, instead of naively applying aggregations, think about what features are actually important from a domain point of view. There are a number of kernels that have created useful features that we can add to our set here to improve performance. The process of feature engineering - feature selection is iterative, and it may require several more passes before we get it completely right!

Data





Output Files

New Kernel Download All

Output Files About this file

- m_test_small.csv
- m_train_small.csv
- selected_features_small_submissio...
- selected_features_submission.csv

This file was created from a Kernel, it does not have a description.

m_test_small.csv

	EXT_SOURCE_1	EXT_SOURCE_2	EXT_SOURCE_3	client_installments_AMT_PAYMENT_min_sum	DAYS_BIRTH	AMT_CREDIT	AMT_ANNUITY	DAYS_EMPLOYED	bureau_DAYS_CREDIT_ENDDATE_max	bureau_DAYS_CREDIT_max
2	0.752614490603174	0.7896543511176771	0.1595195404777181	27746.775	-19241	568800.0	20560.5	-2329	1778.0	-49.0
3	0.564990201796924	0.2916555320093651	0.4329616670974407	43318.8	-18064	222768.0	17370.0	-4469	1324.0	-62.0
4		0.6997868302051784	0.6109913288868294	66875.26499999997	-20038	663264.0	69777.0	-4458	-567.0	-1210.0
5	0.525733977682448	0.5096770801723647	0.6127042441012546	172044.3150000001	-13976	1575000.0	49018.5	-1866	30885.0	-269.0
6	0.202144992067999	0.4256872940912297		133169.4	-13040	625500.0	32067.0	-2191		
7		0.6289043246811542	0.3927738606033134	1842460.8749999972	-18604	959688.0	34600.5	-12009	8957.0	-234.0
8	0.760851017838052	0.5710842061008	0.6512602186973006	68259.33	-16685	499221.0	22117.5	-2580	5813.0	-337.0
9	0.565289988819839	0.6130330592407085	0.3123653692278984	90619.65	-9516	180000.0	14220.0	-1387	27225.0	-107.0
10	0.718507476599168	0.8087877917779122	0.5226973172821112	451534.86	-12744	364896.0	28957.5	-1013	1363.0	-463.0
11	0.210562207633703	0.4448475614048825	0.1940678276718812	272273.8499999991	-10395	45000.0	5337.0	-2625	30886.0	-57.0
12		0.6600149063064604	0.2985949897873972	124279.19999999998	-23670	675000.0	25447.5	365243	-300.0	-1030.0
13	0.756722238389638	0.3937146969745023	0.4365064990977374	897850.4400000006	-15524	261621.0	16848.0	-3555	157.0	-208.0
14	0.263237598013629	0.4700355103285236	0.4153471448843	35394.75	-12278	296280.0	23539.5	-929	1707.0	-119.0
15	0.304939766076750	0.2702553559832115	0.3757110095574066	257082.075	-19687	360000.0	18535.5	-3578	1296.0	-84.0
16		0.4857699312968558	0.5979274227921155	195691.6800000005	-12091	157500.0	7875.0	-1830		-1070.0
17	0.34416525	0.68440672	0.26364681	80453.5199	-13563	296280.0	21690.0	-1007	349.0	-17.0

Run Info

Succeeded	True	Run Time	2180.3 seconds
Exit Code	0	Queue Time	0 seconds
Docker Image Name	kaggle/python(Dockerfile)	Output Size	0
Timeout Exceeded	False	Used All Space	False
Failure Message			

Log

Download Log

```
Time  Line #  Log Message
5.8s        1  [NbConvertApp] Converting notebook script.ipynb to html
5.9s        2  [NbConvertApp] Executing notebook with kernel: python3
104.5s      3  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
107.0s      4  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
205.2s      5  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
208.2s      6  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
292.8s      7  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
294.7s      8  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
374.9s      9  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
376.8s     10  [LightGBM] [Warning] Find whitespaces in feature_names, replace with underlines
2179.4s    11  [NbConvertApp] Support files will be in __results___files/
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
2179.4s    12  [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Making directory __results___files
               [NbConvertApp] Writing 410269 bytes to __results__.html
2179.4s    13
2179.4s    15  Complete. Exited with code 0.
```

Comments (26)

Sort by: All Comments | Hotness

Please sign in to leave a comment.

Angela • Posted on Latest Version • 7 months ago • Options

5

Hi Will, great kernel and I really like your explanation.

Regarding your question in Feature Selection through Feature Importances

"Tree-based models (and consequently ensembles of trees) can determine an "importance" for each feature by measuring the reduction in impurity for including the feature in the model. I'm not really sure what that means (any explanations would be welcome)." Here is my understanding, any discussion would be very welcome.

Information Entropy

Entropy tells you the average information you can get after knowing the result of an event. If the probability of an event is p, the corresponding information is -log(p). The smaller the p is, the more information the event has. Suppose you have a dice with six sides {1, 1, 1, 2, 2, 3}. If you roll it once and get {1}, the information you get is -log(1/2). If the outcome is {2}, the information would be -log(1/3). Also {3} would give you information -log(1/6). But you only have 1/2 chance to get {1}, 1/3 chance to get {2}, and 1/6 chance to get {3}. Thus on average, the dice's information is about -0.5*log(0.5) -0.33*log(0.33)-0.17*log(0.17). We call this information entropy for this dice. Now let's consider an extreme case. What if you have a dice with six sides {1, 1, 1, 1, 1, 1}. You won't get any information by rolling it. The entropy for this extreme dice is -log(1) = 0.

For Tree Method

Intuitively, higher entropy reflects how random (or impure) the data is. Tree method will try to segment data to two branches at each node. If our data at a node is pure and unable to be separated, the entropy would be *-1log(1)-0*log(0)=0. Of course when the data can segment to half-half, the entropy would be -0.5*log(0.5)-0.5*log(0.5) = 1. In this case we can still divide our data because it's not impure yet. The goal of the feature selection is to find the features or attributes which lead to split in children nodes whose combined entropy sums up to lower entropy than the entropy value of data segment before the split. We want to use features that can make subtree as pure as possible while building a decision tree method.

Aigege • Posted on Latest Version • 7 months ago • Options 1

Hi @Will, Thank you for a lot of work. This is very useful for me, new to this competition. But when I used the several dimensionality reductions you mentioned above, my CV dropped from 0.7987 to about 0.795, and my LB score dropped from 0.793 to 0.792. does it look like the result is getting worse? Should I keep all these features and train better?Because most people say cv is more credible than lb scores .

Will Koehrsen Kernel Author • Posted on Latest Version • 6 months ago • Options 2

Feature selection can definitely hurt your model performance which is why it's important to try several different strategies. For this problem, I'd trust the cross validation scores because the public leaderboard is only using 10% of the test data and optimizing for such a small number of observations will lead to overfitting. I'd recommend trying several different versions of feature selection using different thresholds. Machine learning is still largely empirical which means you need to try out different combinations to see what works the best!

Max Lukyanenko • Posted on Version 5 • 8 months ago • Options 1

Nice kernel for beginners! It seems to me that removing features by correlation needs some supervision. E.g. AMT*CREDIT and AMT*GOODS_PRICE has very high correlation but significant difference between this features in particular case may be a sign of some issues of the loan.

None • Posted on Version 5 • 8 months ago • Options 0

I also find this problem. Are there any methods to solve it?

Will Koehrsen Kernel Author • Posted on Version 5 • 8 months ago • Options 0

That's a great point. Both of these features also have high importance according to a gradient boosting machine so it may not be a smart idea to remove them. Manual guidance is necessary in this case!

CoreyLevinson • Posted on Version 5 • 8 months ago • Options 1

This is an awesome kernel!! One thing is that in the second to last paragraph, the fourth method achieves a 5-fold cv AUC score of 0.784121 not .7482. I will definitely be trying out some of these methods to speed up training or see if it improves my score, because often when I run my kernels in Kaggle cloud they are terminated for running longer than 6 hours! Cheers.

Will Koehrsen Kernel Author • Posted on Version 5 • 8 months ago • Options 0

I think the scores change from run to run and I haven't done the best job of keeping everything up-to-date. I'll try to address this when I get a chance.

Yes, feature selection is crucial to speeding up the run time of models. I found that collinearity was the most effective method for feature selection in this problem followed by removing zero/low importance features.

Reinhard • Posted on Version 5 • 8 months ago • Options 2

Hi Will, its a great kernel I enjoyed reading it.

Regarding the removal of features with missing values I was wondering if the information itself that values are missing could be useful for prediction. So before removing them, it could be interesting to transform them to something like feature_missing: [true, false]. Did you try that?

I recently wrote a kernel on the same topic applied on the Titanic data set with much fewer features. You may have a look if you like.

I am also struggling to understand what importance actually means in the context of tree-based predictors. Therefore, I applied logistic regression and used the features' p-value as metric. I also tried L1-regularisation to eliminate features, this worked even better than RFE.

Interestingly, PCA was the method which performed worse in my analysis. Would be interesting if you could evaluate the performance of PCA in your kernel as well.

Will Koehrsen Kernel Author • Posted on Version 5 • 8 months ago • Options 0

That's great advice about adding the flag column before removing values! However, that wouldn't end up removing any features, because for every column you remove, you would add another one. It still might be worth trying though, or you could do something where you remove many missing columns, and then for each observation, add up the total number of columns removed.

The importance in a tree-based predictor is the reduction in impurity from including that feature which isn't exactly clear. This answer on Stack Exchange sort of explains it. I never see people discuss what the importances mean, only that they can be used to "rank" features. Also, I'm not sure if the importance in a random forest applies to other methods, such as support vector machines or neural networks. That is, would the most important features to a random forest also be "most important" to other machine learning methods?

I tried some modeling with PCA (keeping 99% of the variance) and the model performed extremely poorly. PCA only cares about the variance, which does not correspond to the features with the best predictive value! I don't see PCA used as much anymore now that tree-based ensemble machine learning models are very popular. InfoGainAttributeEval, a feature selection method used in Weka seems like it may be worth trying and is explained in this Stack Overflow answer.

Feature selection, like most other aspects of machine learning, is still almost entirely empirical! What works for one problem will not necessarily work for others, and what is best for one model might not be a good choice for other models.

Suchith • Posted on Version 5 • 8 months ago • Options 0

Its a big surprise that 2 components occupy variance of 98.7%. But it depends of the imputation of the missing values also. So

how much did it effect when we change the imputation of missing value?

By the way excellent kernel

Will Koehrsen Kernel Author • Posted on Version 5 • 8 months ago • Options 0

That's a good point. Sklearn's version of PCA does require first imputing the missing values. It would be interesting to see if different imputation strategies have an effect on the explained variance ratio.

Sathish Nambale • Posted on Version 5 • 8 months ago • Options 0

Inspiring work. very helpful for beginners like me. Stay blessed

cloty • Posted on Version 5 • 8 months ago • Options 0

surprise!

Janio Martinez Bachmann • Posted on Version 5 • 8 months ago • Options 0

Have you tried using t-SNE? t-SNE tends to preserve the original structure of the data. Maybe it will show a clearly separation between the targets. Anyways, great kernel Will!

Will Koehrsen Kernel Author • Posted on Version 5 • 8 months ago • Options 1

t-SNE is very cool but also computationally expensive. It's only used for visualization and not for actual feature reduction for machine learning because it is non-parameteric and the results change every time (compared to PCA which always returns the same result).

It might be cool for a visualization, and perhaps I will try it out on a subset of the data. In case you haven't seen this, here is a great article on using t-SNE for visualization and some of the drawbacks.

Panda • Posted on Version 5 • 8 months ago • Options 0

Thank you for your nice kernel! But i'm wondering that can we improve our cv in models like lgbm by feature selection or it just improve the speed?

Hyun woo kim • Posted on Version 5 • 8 months ago • Options 0

Thanks good kernel. very helpful to me. But i wondering what was the public score for the number of features 1465?

LeonF • Posted on Version 5 • 8 months ago • Options 0

Thanks for this epic work!!! I am stunned that you contribute that much to this community

ZengXiangyu • Posted on Latest Version • 8 months ago • Options 0

thanks a lot , a poor student with an old laptop facing a lot difficulties in loading dataset QAQ

ramu • Posted on Latest Version • 8 months ago • Options 0

great work

ramu • Posted on Latest Version • 8 months ago • Options 0

really learned a lot..

Michashak • Posted on Latest Version • 8 months ago • Options 0

Hi!
Just before In [11] you wrote: "Now we are ready to move on to the full set of features. These were built by applying the above steps to the entire train*bureau and train*previous files..." Did you think about all above steps? I wonder, because your full set of features are not align after read, so I have doubs if I understood you well or when I got lost. Thanks for your good job.

shreshth saxena • Posted on Latest Version • 8 months ago • Options 0

Inspiring work mate. This kernel and the other two about manual feature engineering have helped me and surely lot of other beginners a lot. Cheers!

LongYin • Posted on Latest Version • 7 months ago • Options 0

Is there some mistake in this sentence?

After applying the fourth method, we end up with 342 features with a 5-fold cv AUC SCORE of

0.7482

and a public leaderboard score of 0.782.

8 months ago

This Comment was deleted.