loan-default-prediction

January 2, 2024

0.1 SynDiffix Loan Prediction Tutorial

This notebook demonstrates how to use SynDiffix for a loan default prediction model. It is based on the example by Zhou Xu at:

https://github.com/zhouxu-ds/loan-default-prediction/blob/main/notebook/modeling.ipynb

which builds a model to predict the likelihood of a loan default using the Czech banking dataset.

Note that this is not a very good example, for two reasons. First, the dataset is small, and in particular the number of defaulted loans is very small. Second, none of the columns are particularly good features, leading to a low quality model in any event. As a result, different choices in for instance the training and test sets can have a significant effect on the model.

0.1.1 Setup

The `syndiffix` package requires Python 3.10 or later. Let's install it and other packages we'll need for the notebook.

[ ]:
```
%pip install -q syndiffix requests pandas matplotlib numpy seaborn scikit-learn
```

```
Note: you may need to restart the kernel to use updated packages.
```

0.1.2 Loading the datasets

These table have all been prepared and loaded onto the open-diffix.org website.

[ ]:
```
import requests
import bz2
import pickle
def download_and_load(url):
    response = requests.get(url)
    data = bz2.decompress(response.content)
    df = pickle.loads(data)
    return df

# Usage
df_loan = download_and_load('http://open-diffix.org/datasets/loan.pbz2')
df_account = download_and_load('http://open-diffix.org/datasets/account.pbz2')
df_district = download_and_load('http://open-diffix.org/datasets/district.pbz2')
df_order = download_and_load('http://open-diffix.org/datasets/order.pbz2')
```

```
df_trans = download_and_load('http://open-diffix.org/datasets/trans.pbz2')
df_disp = download_and_load('http://open-diffix.org/datasets/disp.pbz2')
df_card = download_and_load('http://open-diffix.org/datasets/card.pbz2')
df_client = download_and_load('http://open-diffix.org/datasets/client.pbz2')
df_client.rename(columns={'district_id': 'cli_district_id'}, inplace=True)
df_card.rename(columns={'type': 'card_type'}, inplace=True)
df_account.rename(columns={'date': 'acct_date'}, inplace=True)
```

0.1.3 Data prep: extract features

Zhou extracted a specific set of features that he used for his model. We copy that here.

[ ]:
```
import pandas as pd

df_loan_acct = pd.merge(df_loan, df_account, on='account_id', how='left')
df = pd.merge(df_loan_acct, df_district, on='district_id', how='left')
df['days_between'] = (df['loan_date'] - df['acct_date']).dt.days
print(df.columns)
```

```
Index(['loan_id', 'account_id', 'loan_date', 'amount', 'duration', 'payments',
       'status', 'defaulted', 'district_id', 'frequency', 'acct_date', 'city',
       'region', 'population', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10',
       'avg_salary', 'A12', 'A13', 'entrepreneur_rate', 'A15', 'A16',
       'average_unemployment_rate', 'average_crime_rate', 'days_between'],
      dtype='object')
```

[ ]:
```
df_order_grouped = df_order.groupby('account_id')['amount'].mean().reset_index()
df_order_grouped.rename(columns={'amount': 'avg_order_amount'}, inplace=True)
df = pd.merge(df, df_order_grouped, on='account_id', how='left')
print(df.columns)
```

```
Index(['loan_id', 'account_id', 'loan_date', 'amount', 'duration', 'payments',
       'status', 'defaulted', 'district_id', 'frequency', 'acct_date', 'city',
       'region', 'population', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10',
       'avg_salary', 'A12', 'A13', 'entrepreneur_rate', 'A15', 'A16',
       'average_unemployment_rate', 'average_crime_rate', 'days_between',
       'avg_order_amount'],
      dtype='object')
```

[ ]:
```
df_avg_bal = df_trans.groupby('account_id')['balance'].mean().reset_index()
df_avg_bal.rename(columns={'balance': 'avg_trans_balance'}, inplace=True)
df_avg_amt = df_trans.groupby('account_id')['amount'].mean().reset_index()
df_avg_amt.rename(columns={'amount': 'avg_trans_amount'}, inplace=True)
df_cnt = df_trans.groupby('account_id').count().iloc[:, 1]
df_cnt.name = 'n_trans'
df = pd.merge(df, df_avg_bal, on='account_id', how='left')
df = pd.merge(df, df_avg_amt, on='account_id', how='left')
df = pd.merge(df, df_cnt, on='account_id', how='left')
```

```
print(df.columns)
```

```
Index(['loan_id', 'account_id', 'loan_date', 'amount', 'duration', 'payments',
       'status', 'defaulted', 'district_id', 'frequency', 'acct_date', 'city',
       'region', 'population', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10',
       'avg_salary', 'A12', 'A13', 'entrepreneur_rate', 'A15', 'A16',
       'average_unemployment_rate', 'average_crime_rate', 'days_between',
       'avg_order_amount', 'avg_trans_balance', 'avg_trans_amount', 'n_trans'],
      dtype='object')
```

[ ]:
```
df_disp_owners = df_disp[df_disp['type'] == 'OWNER']
df = pd.merge(df, df_disp_owners, on='account_id', how='left')
df = pd.merge(df, df_card, on='disp_id', how='left')
df['card_type'].fillna('No', inplace=True)
print(df.columns)
```

```
Index(['loan_id', 'account_id', 'loan_date', 'amount', 'duration', 'payments',
       'status', 'defaulted', 'district_id', 'frequency', 'acct_date', 'city',
       'region', 'population', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10',
       'avg_salary', 'A12', 'A13', 'entrepreneur_rate', 'A15', 'A16',
       'average_unemployment_rate', 'average_crime_rate', 'days_between',
       'avg_order_amount', 'avg_trans_balance', 'avg_trans_amount', 'n_trans',
       'disp_id', 'client_id', 'type', 'card_id', 'card_type', 'issued'],
      dtype='object')
```

[ ]:
```
df = pd.merge(df, df_client, on='client_id', how='left')
df['same_district'] = df['district_id'] == df['cli_district_id']
df['owner_age'] = (df['loan_date'] - df['birth_number']).dt.days // 365
print(df.columns)
print(len(df))
```

```
Index(['loan_id', 'account_id', 'loan_date', 'amount', 'duration', 'payments',
       'status', 'defaulted', 'district_id', 'frequency', 'acct_date', 'city',
       'region', 'population', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10',
       'avg_salary', 'A12', 'A13', 'entrepreneur_rate', 'A15', 'A16',
       'average_unemployment_rate', 'average_crime_rate', 'days_between',
       'avg_order_amount', 'avg_trans_balance', 'avg_trans_amount', 'n_trans',
       'disp_id', 'client_id', 'type', 'card_id', 'card_type', 'issued',
       'birth_number', 'cli_district_id', 'sex', 'same_district', 'owner_age'],
      dtype='object')
682
```

Here is the final feature list selected by Zhou.

[ ]:
```
zhou_columns = ['amount', 'duration', 'payments', 'days_between', 'population',
            'avg_salary', 'average_unemployment_rate', 'entrepreneur_rate',
            'average_crime_rate', 'avg_order_amount', 'avg_trans_amount',
            'avg_trans_balance', 'n_trans', 'owner_age',
            'frequency', 'card_type', 'same_district', 'sex', 'defaulted']
```

```
df_ml = df[zhou_columns]
```

TODO: Do the correlations by synthesizing each pair and taking the correlations. Then compare with the original

0.1.4 Prepare data for synthesis

Before we synthesize the data, we need to split the original into training and test dataframes. This is because we will test the synthetic data model against the original test data, and so that data cannot be included in what gets synthesized. Note that the ML model for the original data will have its own train/test split.

In addition, we create a dataframe containing identifier for the protected entity. Here the protected entity is the banking account. (Note that in this particular dataset, there is only one loan per account, so this is strictly speaking unecessary. But better safe than sorry.)

[ ]:
```
pid_column = 'account_id'
# Add the pid_column back in because SynDiffix needs it
df_ml_pid = df[zhou_columns + [pid_column]]

df_dx_train = df_ml_pid.sample(n=int(len(df_ml)*0.7), random_state=1)
df_dx_test = df_ml_pid.drop(df_dx_train.index)

# Separate the pid_column because of the SynDiffix API
df_pid = df_dx_train[['account_id']]
# And remove it again because it has no analytic value
df_dx_train = df_dx_train.drop(columns=[pid_column])
df_dx_test = df_dx_test.drop(columns=[pid_column])
```

0.1.5 Synthesize the data

Since we know the target column, we should specify it when we synthesize the data. This will lead to better predictions. There are two options. One is to ask SynDiffix to synthesize every column, and the other is to ask SynDiffix to synthesize only those columns that it determines are good features. We do the latter here.

[ ]:
```
from syndiffix import Synthesizer
from syndiffix.clustering.strategy import MlClustering
from syndiffix.common import *

target_column = 'defaulted'

df_syn_feat = Synthesizer(df_dx_train, pids=df_pid,
    clustering=MlClustering(target_column=target_column, drop_non_features=True)
    ).sample()
feat_cols = list(df_syn_feat.columns)
feat_cols.remove(target_column)
```

[ ]:
```
print("Synthetic data columns without non-features:")
print(feat_cols)
print(f"{len(df_syn_feat)} (features only) rows, and {len(df_dx_train)}␣
 ↪original data rows")
```

```
Synthetic data columns without non-features:
['avg_trans_amount', 'avg_trans_balance', 'n_trans']
477 (features only) rows, and 477 original data rows
```

Listed above are the important features according to SynDiffix.

(Note that, alternatively, we could have selected the features ourselves on the original data rather than have SynDiffix do it, and then asked SynDiffix to only synthesize the feature columns.)

0.1.6 Transformations

[ ]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler, StandardScaler
# Original data transformations
num_cols_orig = df_ml.columns[:-5]
cat_cols_orig = df_ml.columns[-5:]
col_trans_orig = ColumnTransformer([
    ('num', MinMaxScaler(), num_cols_orig),
    ('cat', OneHotEncoder(drop='if_binary'), cat_cols_orig)
])
df_transformed_orig = col_trans_orig.fit_transform(df_ml)
X_orig = df_transformed_orig[:, :-1]
y_orig = df_transformed_orig[:, -1]

# For now, we make no transformations on the synthetic data
```

0.1.7 Modeling

For the original data, we are using the RandomForestClassifier parameters that Zhou derived in his notebook. We are skipping the steps that he took to decide on thoses parameters.

For the synthetic data, we are doing a simple LogisticRegression. Note that we have made no particular effort to fine tune the model.

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, accuracy_score

# Original
# Train test split
X_train_orig, X_test_orig, y_train_orig, y_test_orig = train_test_split(X_orig,␣
 ↪y_orig, test_size=0.3, stratify=y_orig, random_state=10)
# Skipping some details here, Zhou settled on the following parameters for his␣
 ↪model
```

```
clf_orig = RandomForestClassifier(n_estimators=10,
                                  max_depth=None,
                                  min_samples_split=5,
                                  min_samples_leaf=1,
                                  random_state=11)
clf_orig.fit(X_train_orig, y_train_orig)
y_orig_pred = clf_orig.predict(X_test_orig)
```

[ ]:
```
# Synthetic
# Note that df_dx_test is the original data. We test against the original data,
# not the synthetic data
X_train_syn = df_syn_feat[feat_cols]
y_train_syn = df_syn_feat[[target_column]]
X_test_syn = df_dx_test[feat_cols]
y_test_syn = df_dx_test[[target_column]]
y_train_syn_ravel = y_train_syn.values.ravel()

from sklearn.linear_model import LogisticRegression
clr_syn = LogisticRegression()
clr_syn.fit(X_train_syn, y_train_syn_ravel)
y_syn_pred = clr_syn.predict(X_test_syn)
```

[ ]:
```
print('Original Acc:', accuracy_score(y_test_orig, y_orig_pred))
print('Original F1:', f1_score(y_test_orig, y_orig_pred))
print('Synthetic LR Acc:', accuracy_score(y_test_syn, y_syn_pred))
print('Synthetic LR F1:', f1_score(y_test_syn, y_syn_pred))
```

```
Original Acc: 0.8878048780487805
Original F1: 0.14814814814814817
Synthetic LR Acc: 0.9121951219512195
Synthetic LR F1: 0.25
```