1 Code Challenge - Praktikum Data Analytics

[1]:
```
import pandas as pd
import numpy as np
import scipy as sp
import sklearn as skl
import nltk
import re
import string
```

[2]:
```
data = pd.read_csv("account_turnovers.csv",␣
↪parse_dates=["entry_date"], na_values='')
detail_category = np.array(data["detail_category"])
data = data.drop(columns=[
    "detail_category", "account_id", "turnover_id", "user_id",␣
↪"holder_name", "entry_date"
])
# "account_id", "turnover_id", "user_id" sind eindeutig und können␣
↪somit zu overfitting führen
# "holder_name" *muss* nicht eindeutig sein, ist aber logischerweise␣
↪so
# "entry_date" hat nur 122 von 1575 Einträge gefüllt, wtf
```

[3]:
```
umlauts = {'ü': 'ue', 'ä': 'ae', 'ö': 'oe', 'ß': 'ss'}
stopwords_de = nltk.corpus.stopwords.words('german')

def cleanup(in_str):
    out_str = str(in_str).lower()
    # reduziere Redundanz dass durch Großschreibung kommt

    out_str = re.sub('bic ........', '', out_str)
    out_str = re.sub('bic ..........', '', out_str)
    # finde und entferne BIC-codes: immer 8- oder 11-stellig

    out_str = re.sub(
        '[{}]'.format(re.escape(string.punctuation)),
        ' ', out_str
    ) # entferne unnötige Zeichen

    out_str = re.sub('\d', '', out_str)
    # fast alle Zahlen im Verwendungszweck sind für Identifizierung,
    # Potential für overfitting
```

```
    for key, val in umlauts.items():
        out_str = re.sub(key, val, out_str)
    # reduziere Redundanz dass durch Ümläute kommt

    out_str = re.sub(r"((?<=^)|(?<= )).((?=$)|(?= ))", '', out_str)
    # nach Entfernen von Zahlen gibt es viele einstellige Wörter.
    # Dies wird später zu overfitting führen nach dem encoding

    out_str = out_str.split()
    out_str = ' '.join([
        token for token in out_str if not token in stopwords_de
    ]) # normalisiere die Anzahl Leerzeichen und entferne Stopwörter
↪im letzten Schritt

    return out_str
```

[4]:
```
for column in ("acct_name", "acct_type", "entry_text",
↪"payee_payer_name", "paymt_purpose"):
    data[column] = data[column].apply(cleanup)
# erste Runde von preprocessing auf die Spalten mit Strings
```

[5]:
```
most_com = nltk.FreqDist( ' '.join(data["paymt_purpose"]).split() ).
↪most_common(25)
most_com
```

[5]:
```
[('svwz', 974),
 ('de', 537),
 ('eref', 461),
 ('mref', 447),
 ('miete', 422),
 ('dezzz', 408),
 ('cred', 396),
 ('end', 340),
 ('ref', 264),
 ('iban', 249),
 ('nr', 238),
 ('abwa', 238),
 ('bic', 217),
 ('tan', 188),
 ('datum', 179),
 ('to', 172),
 ('uhr', 163),
 ('kref', 153),
 ('danke', 141),
 ('purp', 122),
 ('id', 122),
```

```
 ('sepa', 110),
 ('rg', 100),
 ('karte', 98),
 ('eur', 94)]
```

Wie man sieht, sind die häfigsten 25 Wörter von der Spalte "paymt_purpose" meistens nicht nutzvoll. Nach 25 kommt "basislastschrift", was nutzvoll sein könnte. Deswegen werden wir diese 25 von der Spalte entfernen (bis auf ein Paar Ausnahmen).

[6]:
```
most_com.pop(4); most_com.pop(22)
# wir nehmen die nutzvollen Wörter aus unserer Blacklist

for word, freq in most_com:
    stopwords_de.append(word)
data["paymt_purpose"] = data["paymt_purpose"].apply(cleanup)
# zweite Runde proprocessing, diesmal mit Blacklist

data.to_csv("editeddata.csv")
# immer schön die proprocessed daten ausspucken
```

[7]:
```
amount = np.array(data["amount"]).reshape(-1, 1)
gv_code = np.nan_to_num(data["gv_code"]).reshape(-1, 1)
# "gv_code" hat ein Paar unbenutzbare NaNs

data_matrix = sp.sparse.csr_matrix(
    np.hstack((
        (amount - amount.mean()) / amount.std(),
        (gv_code - gv_code.mean()) / gv_code.std()
    ))
) # normalisiere die float-Spalten

for column in ("acct_name", "acct_type", "entry_text",␣
↪"payee_payer_name", "paymt_purpose"):
    data_matrix = sp.sparse.hstack([
        data_matrix,
        skl.feature_extraction.text.CountVectorizer().
↪fit_transform(data[column])
    ]) # encode die Wörter in den kategorischen String-Spalten
```

[8]:
```
data_matrix = data_matrix.tocsr()
test_data_matrix = data_matrix[1500:, :]
data_matrix = data_matrix[:1500, :]
test_detail_category = detail_category[1500:]
detail_category = detail_category[:1500]
# splite die Daten in Train- und Testset
```

[9]:
```
clf = skl.svm.SVC().fit(data_matrix, detail_category)
# ganz normales SVM ohne Kernel weil ich keine Zeit mehr habe
```

[10]:
```
test_detail_category_preds = clf.predict(test_data_matrix)
(test_detail_category == test_detail_category_preds).sum() /␣
↪test_detail_category.shape[0]
```

[10]:
```
0.8666666666666667
```

86% ist gar ned mal so schlecht