import libraries

In [ ]:
```
from openai import OpenAI
import os

os.environ['OPENAI_API_KEY'] = os.getenv('OPENAI_API_KEY')
client = OpenAI()
```

In [ ]:
```
import random
import numpy as np
import pandas as pd
import altair as alt
from tqdm import tqdm
from fuzzywuzzy import process, fuzz
from sklearn.decomposition import PCA
from sklearn.metrics.pairwise import cosine_similarity


tqdm.pandas()
```

In [ ]:
```
def print_mapping_stats(pairs_dict):
    recognized_count = 0
    unrecognized_count = 0
    recognized_pairs = []
    unrecognized_pairs = []

    for account, category in pairs_dict.items():
        if category == 'Unrecognized account':
            unrecognized_pairs.append((account, category))
            unrecognized_count += 1
        else:
            recognized_pairs.append((account, category))
            recognized_count += 1

    print(f"Total unique accounts: {len(pairs_dict)}")
    print(f"Recognized accounts: {recognized_count}")
    print(f"Unrecognized accounts: {unrecognized_count}")

    random.shuffle(recognized_pairs)
    random.shuffle(unrecognized_pairs)

    print("\nSample of recognized pairs:")
    for account, category in recognized_pairs[:5]:
        print(f'{account} -> {category}')

    print("\nSample of unrecognized pairs:")
    for account, category in unrecognized_pairs[:5]:
        print(f'{account} -> {category}')
```

read the data and merge transactions into one table

In [ ]:
```
master = pd.read_excel('../data/master-categories.xlsx')
trans1 = pd.read_csv('../data/transactions1.csv')
trans2 = pd.read_csv('../data/transactions2.csv')
```

```
transactions = pd.concat([trans1, trans2])

transactions.sample(5)
```

Out[ ]:

	Date	PL Account	Amount	Description	Counterparty
829	2023-08-08	WISE EUR	780.93	Transaction 830	Counterparty 830
301	2023-11-12	R&D expenses:R&D team salary tax expenses	-2412.43	Transaction 302	Counterparty 302
2524	2/10/2023	Shipping and delivery expense (deleted)	1657.75	NaN	NaN
2204	5/26/2024	Prepaid expenses administrative	-4707.00	NaN	NaN
1715	12/8/2023	Insurance - Liability (deleted)	300.58	NaN	NaN

In [ ]:

```
transactions.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10000 entries, 0 to 4999
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Date          10000 non-null  object 
 1   PL Account    10000 non-null  object 
 2   Amount        10000 non-null  float64
 3   Description   5000 non-null   object 
 4   Counterparty  5000 non-null   object 
dtypes: float64(1), object(4)
memory usage: 468.8+ KB
```

As we can see from above, the `Date` column is not unified in a singe date format. Apart that, one of the transactions table was completely missing the records in `Description` and `Counterparty` columns. So, we'll transform `Date` column so that the records are in the same format and drop the `Description` and `Counterparty` cloumnts completely (it won't be of any use for us).

Finally, we'll rename column names to a bit more managable format.

In [ ]:

```
transactions.drop(['Description', 'Counterparty'], axis=1, inplace=True)
transactions['Date'] = pd.to_datetime(transactions['Date'], format='mixed

master.rename(columns={'Master categories': 'master_category'}, inplace=T
transactions.rename(columns={'Date': 'date', 'Amount': 'amount', 'PL Acco
```

Now the dataset looks like this:

In [ ]:

```
transactions.head()
```

Out[ ]:

	date	pl_account	amount
0	2023-01-04	Professional services	741.87
1	2024-03-04	Marketing team salary	2673.80
2	2024-01-16	Direct labour - COS (deleted)	-1578.98
3	2024-05-20	Uncategorised Asset	2455.15
4	2024-06-12	Repairs and Maintenance (deleted)	2531.35

data preprocessing

We can observe that some `pl_account` categories have names like "Direct labour - COS *(deleted)*". Our approach involves using language models to capture semantic structures and compare PL Accounts with Master Categories and determine the similarity between these. So, in this context items in PL account names like "delete" can confuse a Language model and potentially negatively influence its performance. We should therefore work with cleaned version without "(deleted)" items in PL Accounts.

In [ ]:
```
transactions['clean_pl_account'] = transactions['pl_account'].apply(lambd
```

As a result, we obtain a table with one more column `clean_pl_account`:

In [ ]:
```
transactions.head()
```

Out[ ]:

	date	pl_account	amount	clean_pl_account
0	2023-01-04	Professional services	741.87	Professional services
1	2024-03-04	Marketing team salary	2673.80	Marketing team salary
2	2024-01-16	Direct labour - COS (deleted)	-1578.98	Direct labour - COS
3	2024-05-20	Uncategorised Asset	2455.15	Uncategorised Asset
4	2024-06-12	Repairs and Maintenance (deleted)	2531.35	Repairs and Maintenance

In [ ]:
```
def fuzzy_match(pl_account, master_list, token_threshold=95, partial_thre
    match, score = process.extractOne(pl_account.lower(), master_list, sc
    if score >= token_threshold:
        return match
    match, score = process.extractOne(pl_account.lower(), master_list, sc
    if score >= partial_threshold:
        intermediate_match, intermediate_score = process.extractOne(pl_ac
        if intermediate_score >= token_threshold:
            return intermediate_match
        return 'Unrecognized account'

    return 'Unrecognized account'
```

mapping: 1st stage

If we observe the data, we'll quickly find that there are quite a few 100% matching Master Categories and PL Accounts. Also, there are some straightforward correspondences like 'Financial Modeling' and '1 Financial Modeling'.

These can be mapped with one another very easily and we obviously don't need any rocket science to merge these. So, as a first stage, we'll try to map accounts with categories by calculating Levenshtein similarity between these two. This simple method will allow to merge items very efficiently.

In [ ]:
```
master_list = master['master_category'].tolist()
unique_clean_accounts = transactions['clean_pl_account'].unique()
fuzzy_account_mapping = {account: fuzzy_match(account, master_list) for a
```

```
Mapping accounts:  19%|██        | 45/243 [00:00<00:00, 444.53it/s]
Mapping accounts: 100%|██████████| 243/243 [00:00<00:00, 446.87it/s]
```

In [ ]:
```
print_mapping_stats(fuzzy_account_mapping)
```

```
Total unique accounts: 243
Recognized accounts: 124
Unrecognized accounts: 119

Sample of recognized pairs:
Design -> Design
R&D expenses:R&D team salary tax expenses -> R&D team salary
Social Tax -> Social Tax
Purchase & Sales of intangible assets -> Sales of intangible assets
Professional services:Financial consultancy -> Financial consultancy

Sample of unrecognized pairs:
Stationery and printing -> Unrecognized account
Amortisation -> Unrecognized account
Professional services:Professional services -> Unrecognized account
Prepaid Income -> Unrecognized account
Accrued non-current liabilities -> Unrecognized account
```

As we can see, we were able to map more than a half of PL accounts with this simple and efficient method!

Still, the are plenty of PL accounts left unmapped. These ones will be addressed with a more advanced technique involving a language model. But let's first select only accounts that were not recognised up to this point for further analysis.

In [ ]:
```
transactions['master_category'] = transactions['clean_pl_account'].map(fu
rec_trans = transactions[transactions['master_category'] != 'Unrecognized
unrec_trans = transactions[transactions['master_category'] == 'Unrecogniz

unique_unrec_clean_accs = unrec_trans['clean_pl_account'].unique()
```

obtaining embeddings

An embedding can be thought of here as a vector representation of a text. Such vectors are simply long arrays of numbers that capture various semantic features of texts. When working with language models, we often need embeddings, becasue computers can only make sense of text in a numeric, not symbolic form.

There are exist many different algorithms to convert a text into a vector of numbers. One of the currently most advanced embedding techniques were developed by Open AI. Thus, we'll use their embedding model to represent or categories as arrays of numbers.

As we've said, we'll only compute embeddings for items that **were not** recognised at the previous stage.

In [ ]:
```
def get_embedding(open_ai_client, text):
    '''Get the embedding of a text using OpenAI API'''
    response = open_ai_client.embeddings.create(input=text, model="text-e
    return np.array(response)
```

In [ ]:
```
master_embeddings = np.array([get_embedding(client, cat) for cat in tqdm(
unrec_embeddings = np.array([get_embedding(client, account) for account i
```

```
Generating master embeddings: 100%|██████████| 88/88 [00:27<00:00,  3.19i
t/s]
Generating embeddings for unrecognized accounts: 100%|██████████| 119/119
[00:38<00:00,  3.13it/s]
```

In [ ]:
```
pca = PCA(n_components=0.7)
reduced_master_emb = pca.fit_transform(master_embeddings)
reduced_unrec_emb = pca.transform(unrec_embeddings)
```

determining similarity

After we obtained embeddings, the only left step is to calculate and match the closest ones. As we said above, embeddings are basically vectors. We can therefore use various measures that calculate the distance between embeddings.

The principle is quite simple: the shorter the distance, the more likely it is that the two embedded texts share some semantic properties. These ones will be mapped to one another. However, if for one embedded PL account there is Master category embedding that is close enough, then we'll label this PL account as unrecognized.

In [ ]:
```
def find_best_match_embedding(embedding, master_list, master_embeddings,
    similarities = cosine_similarity([embedding], master_embeddings)[0]
    max_similarity = similarities.max()
    if max_similarity >= threshold:
        best_match_index = similarities.argmax()
        return master_list[best_match_index]
    return 'Unrecognized account'
```

In [ ]:
```
unrec_acc_map = {account: find_best_match_embedding(embedding, master_lis
                                for account, embedding in zip(unique_unre
```

```
unrec_trans.loc[:, 'master_category'] = unrec_trans['clean_pl_account'].m
```

Now, let's see how well our model performed.

In [ ]:
```
print_mapping_stats(unrec_acc_map)
```

```
Total unique accounts: 119
Recognized accounts: 59
Unrecognized accounts: 60

Sample of recognized pairs:
Current portion of long-term debt -> Proceeds from debt
Marketing and sales -> Digital Marketing
Dividend disbursed -> Dividends paid
Project's direct cost:Software expenses -> FP&A team software expenses
Income tax payable -> Federal Taxes

Sample of unrecognized pairs:
Amortisation / Depreciation:Depreciation -> Unrecognized account
Insurance - General -> Unrecognized account
CHASE SAV *2868 -> Unrecognized account
Amortisation / Depreciation -> Unrecognized account
Stripe (required for Synder) -> Unrecognized account
```

We can note that now our algorithm was able to map much more complex relationships between PL accounts and Master categories. For example, it recognised that Professional services may belong to Consulting category and that Interest expense may be related to Interest Loss.

In [ ]:
```
final_rec_trans = pd.concat([rec_trans, unrec_trans])
unmapped_trans = final_rec_trans[final_rec_trans['master_category'] == 'U
```

In [ ]:
```
final_rec_trans.drop(['clean_pl_account'], axis=1, inplace=True)
final_rec_trans.sample(15)
```

Out[ ]:

	date	pl_account	amount	master_category
4690	2023-09-19	Office expenses:Other office expenses	903.87	Other office expenses
514	2024-05-15	Staff expenses:Corporate events	4587.98	Corporate events
4693	2023-10-24	Payroll Clearing	-4998.17	Unrecognized account
4347	2023-06-19	Loans to Others	1188.69	Receiving of loans
1034	2023-11-12	Common stock	1955.43	Unrecognized account
3270	2023-10-15	Short-Term Investments	3489.67	Unrecognized account
1067	2024-04-08	Supplies (deleted)	897.16	Unrecognized account
4184	2023-06-03	Unrealised loss on securities, net of tax (del...	1794.54	Foreign Exchange Loss
1806	2024-04-17	Grants and other financial income	-2768.91	Grants and other non-operating income
3711	2023-06-17	Less	2812.03	Less: Discount
1463	2023-04-12	Long-Term Investments	-2627.07	Unrecognized account
1190	2024-05-17	Travelling expenses	-1407.21	Offline events + travelling expenses
2338	2023-11-30	State Taxes	-2524.84	State Taxes
1861	2023-02-13	Other Expenses:Other general and administrativ...	-3894.19	Other general and administrative expenses
2714	2023-09-24	Professional services:Contractors	745.70	Other subcontractors

In [ ]:

```
total_accounts = final_rec_trans['pl_account'].nunique()
recognized_df = final_rec_trans[final_rec_trans['master_category'] != 'Un
rec_accs = recognized_df['pl_account'].nunique()
unrec_accs = total_accounts - rec_accs

unmapped_num, mapped_num = len(unmapped_trans), len(final_rec_trans) - le
print(f"Number of unique recognised PL accounts: {rec_accs}\nNumber of un
```

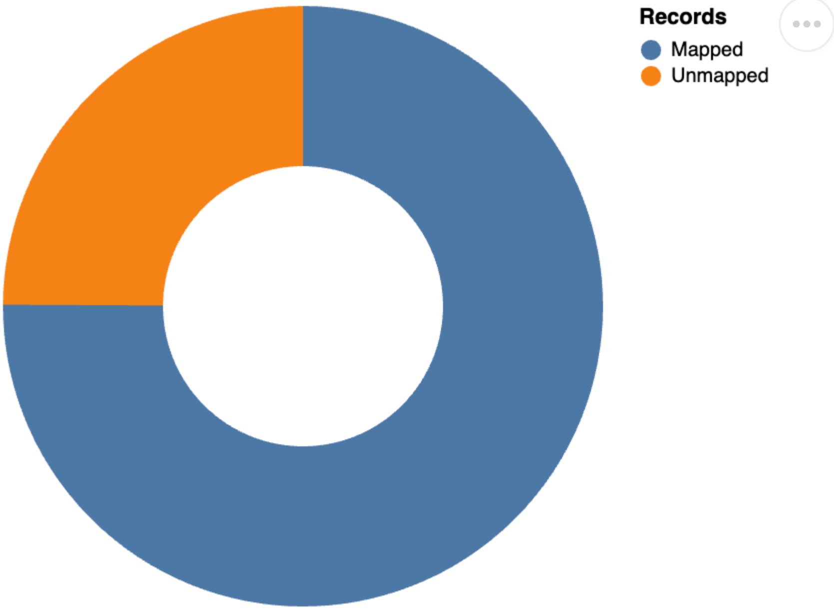
```
Number of unique recognised PL accounts: 185
Number of unique unrecognized PL accounts: 60
```

In [ ]:

```
df = pd.DataFrame({'Records': ['Mapped', 'Unmapped'], 'count': [mapped_nu
alt.Chart(df).mark_arc(innerRadius=70).encode(
    color='Records',
    theta='count:Q'
).properties(
    width=300,
```

```
    height=300
)
```

Out[ ]:



Thus, our procedure managed to to recognise 7508 out of 10k records (185 out of 245 unique types of PL accounts).