China Climate Policy Risk Hedging Portfolio

基于中国股市的气候政策风险的对冲策略量化分析

Oct 1st-21st , 2024

基于一般性宏观气候风险的因子模拟对冲策略

1.基于宏观层面的国家政策信息构建反应宏观非流动性的因子指数指标. 考虑到解释性和调参，采用几种词嵌入生成方法。宏观政策文本首先完成基本的预处理，剔除不必要的无意义的词减少文本的噪声。进一步的，利用气候政策参考文献（GPT 或者书本都可以，以气候风险政策为主题的）生成和气候风险政策相关的关键词，作为气候政策风险的测度对标词汇或者种子词汇：

- TFIDF
- Word2Vec
- LLM

数据来源：南方周末、经济日报。

构造过程比较灵活，在基于参考文献生成相应的种子词汇之后，可以采用 cosine similarity 生成相应文本的气候风险政策评分，也可以采用利用预训练的模型扩张种子词汇，进而 Bag of Words 的方法。

注意，这里采用不同方法的调参区间比较大，一切以生成的宏观测度的合理性检验和最后调参结果的样本外表现结果为准。其中，合理性检验包括两个部分，时间趋势的图示检验，文本范例，以及可以用相应的行业完成对微观层面的差异性检验。

2.提供基于第一步的内部合理性和外部合理性检验的基本工具，内部合理性的展示，如上提及可以包括行业特征划分，时间趋势划分以及文本内容划分。外部性的检验，严格的可以采用宏观指数下不同行业的经济反应，来作为支撑证据。

$$股票收益_{i,t} = Treat_i \times 宏观气候风险政策_t + 其他控制变量 + 固定效应_i + 固定效应_t$$

3.基于计量方法构造具有可解释性的风险对冲策略。针对非流动性的宏观因子首先采取因子模仿法构造基本的。首先，将合理化的宏观指标投射在股票市场上面：

$$CC_t = Const + \omega Z'_{t-1} r_t + \epsilon_t$$

其中，Z_{t-1}代表乐企业层面的相关特征变量，用以捕捉企业特征的动态变化，构建时候可以考虑给公司相关的特征标准化并且重新排序等多重构造方法，第二项代表了相应的具有流动性的气候风险政策。

4.最后，我们进一步衡量和把控对冲策略的表现。第一，完成内部相关性检验，完成 Step three 里对模拟因子的显著性分析。第二，类比生成的因子拟合对冲策勒与已知的行业或者第三方评级的对冲策略之间关系。第三，考虑 OOS 或者 Cross Validation 下的多种样本外的估计发，采用滚动回归的方式构建新指标。(滚动回归采取方法，可以用 12 个月做训练，而未来三个月做检验诸如此类等等）第四可以用构造好的持仓信息，检验企业层面是不是具有合理的风险敞口。

以上是针对非流动宏观因子的，固定风险对冲构造方法，该方法的优势是以准确和少误差为主，并且充分考虑了可行性和传统的低买高卖基于 Fama 的市场。

本文主要内容：加入jieba和停用词的TFIDF&Word2Vec余弦相似度和时间趋势图。（附录：NLP文本分析的系统学习代码及示例）。

Code

October 22, 2024

[1]:
```
import pandas as pd
import os
import datetime

#
folder_path = '/rds/general/ephemeral/user/yz1117/ephemeral/policy_data/  /
 ↪new_txts'

#
df = pd.DataFrame(columns=['date', 'text'])
#
for subfolder_name in sorted(os.listdir(folder_path)):
    subfolder_path = os.path.join(folder_path, subfolder_name)
    #print('>>>', subfolder_path)
    #
    if os.path.isdir(subfolder_path):
        #
        for file_name in sorted(os.listdir(subfolder_path)):
            file_path = os.path.join(subfolder_path, file_name)
            with open(file_path, 'r', encoding='UTF-8') as file:
                content = file.read()

            #     10       date
            date_str = file_name[:10]
            try:
                date_format = datetime.datetime.strptime(date_str, '%Y-%m-%d').
 ↪date()
            except ValueError:
                continue  #

            #
            new_row = pd.DataFrame({'date': [date_format], 'text': [content]})

            #   pd.concat
            df = pd.concat([df, new_row], ignore_index=True)
#
df = df.sample(frac=0.5)
```

```
#
print(df)
```

```
           date                                                text
6920  2019-07-06    1919       14             …
9400  2023-11-08           \n                 …
7134  2019-11-05    11 4                       …
5255  2016-10-14    10 13      2016              …
4865  2015-12-22  "                 "         …
…          …                                                  …
8296  2021-11-04  "  "       80                …
2207  2011-08-16                   G6         …
4284  2015-02-07                             …
746   2009-04-08     \n\n2009-04-08 21:14:22\n\n"       …
4337  2015-03-07          "   " \n\n2011-03-07 14:53:55\…

[4720 rows x 2 columns]
```

[ ]:

[2]:
```
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
```

[3]:
```
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
import pandas as pd
import numpy as np
import gdown
import lda
import re
import datetime
import os
import sys
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import contractions
import string
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
import unicodedata
import spacy
import nltk
from nltk import SnowballStemmer
import random

#
nlp_standard = spacy.load('en_core_web_sm')
```

[4]:
```
#
sys.path.append('/rds/general/user/yz1117/home/Hangyun_Work/text-preprocessing/
 ↪src')
import preprocessing_class as pc
import preprocess_data
```

[5]:
```
import matplotlib.colors as mcolors
from wordcloud import WordCloud, STOPWORDS
```

[6]:
```
cli_words = [" ", " ", "  ", " ", " ", " ", "   ", "   ", "  ", "  ", " ", " ",␣
 ↪" ", "   ", "  ", "  ", " ", "  ", "   ", "  ", " ", " ", " ", " ", " ", "  ",␣
 ↪"  ", " "]
```

[7]:
```
#######################################################
################TFIDF##########################
#######################################################
# function for counting the number of words from a list that appear on a give␣
 ↪text
def count_instances(text, target_list):
    target_string = '|'.join(target_list)
    matches = re.findall(fr"\b({target_string})\b", text)
    return len(matches)
# apply counting function
df['counts'] = df.text.apply(lambda x: count_instances(text=x,␣
 ↪target_list=cli_words))
print(df)
```

```
           date                                                text  counts
6920  2019-07-06    1919       14             …        0
9400  2023-11-08           \n                 …        4
7134  2019-11-05    11 4                       …       14
5255  2016-10-14    10 13      2016              …        8
4865  2015-12-22  "                 "         …       18
…          …                                                 …      …
8296  2021-11-04  "  "       80                 …       12
2207  2011-08-16                   G6          …        9
4284  2015-02-07                              …        0
746   2009-04-08     \n\n2009-04-08 21:14:22\n\n"       …        4
4337  2015-03-07          "   " \n\n2011-03-07 14:53:55\…        0

[4720 rows x 3 columns]
```

[8]:
```
#######################################################
#####Preprocessing text with scikit-learn###########
#######################################################
# create a CountVectorizer object straight from the raw text
import jieba
```

```
with open('/rds/general/user/yz1117/home/Hangyun_Work/hitstopwords.txt.txt',␣
 ↪'r', encoding='utf8') as f:
    stop_words = [i.strip() for i in f]

count_vectorizer = CountVectorizer(token_pattern=r"(?u)\b[\w-]+\b",       #␣
 ↪regular expression to split documents into tokens
                                   lowercase=True,                        #␣
 ↪convert all characters to lower-case
                                   strip_accents=None,                 # remove␣
 ↪non-ascii characters (WARNING: this is dangerous in Romance languages)
                                   stop_words=stop_words,                 #␣
 ↪remove stopwords from a built-in list. We can also provide our own list
                                   ngram_range=(1, 1),                    #␣
 ↪generate only unigrams
                                   tokenizer=jieba.lcut,
                                   #analyzer="word",                       #␣
 ↪build matrix at the word-level
                                  # max_df=0.8,                           #␣
 ↪ignore tokens that have a higher document frequency (can be int or percent)
                                  # min_df=20,                            #␣
 ↪ignore tokens that have a lower document frequency (can be int or percent)
                                   max_features=None,                     # we␣
 ↪could impose a maximum number of vocabulary terms
                                   )
count_vectorizer
```

[8]:
```
CountVectorizer(stop_words=['---', '  ', ' ÷ ', '" ', ' ', ' ', ':', '→',
                            '℃', '&', '*', ' ', '~~~~', '’', '.', ' ', '. ',
                            './', '--', ' ', ' ', ' ', '  ', ' ', '  ',
                            '  ', ' ', '  ', ' ', '//', …],
                token_pattern='(?u)\\b[\\w-]+\\b',
                tokenizer=<bound method Tokenizer.lcut of <Tokenizer
dictionary=None>>)
```

[ ]:

[9]:
```
from sklearn.metrics.pairwise import cosine_similarity
```

[10]:
```
# directly transform our data into a document-term matrix. We just need to pass
# our documents as an argument to the fit_transform function
dt_matrix_simple = count_vectorizer.fit_transform(df["text"])
print(f"Document-term matrix created with shape: {dt_matrix_simple.shape}")   ␣
 ↪# (documents, vocabulary size)
# transform to regular numpy array to easily manipulate.
# this might not be convenient (or even possible) if your matrix is too large
```

```
# dt_matrix_simple = dt_matrix_simple.toarray()
```

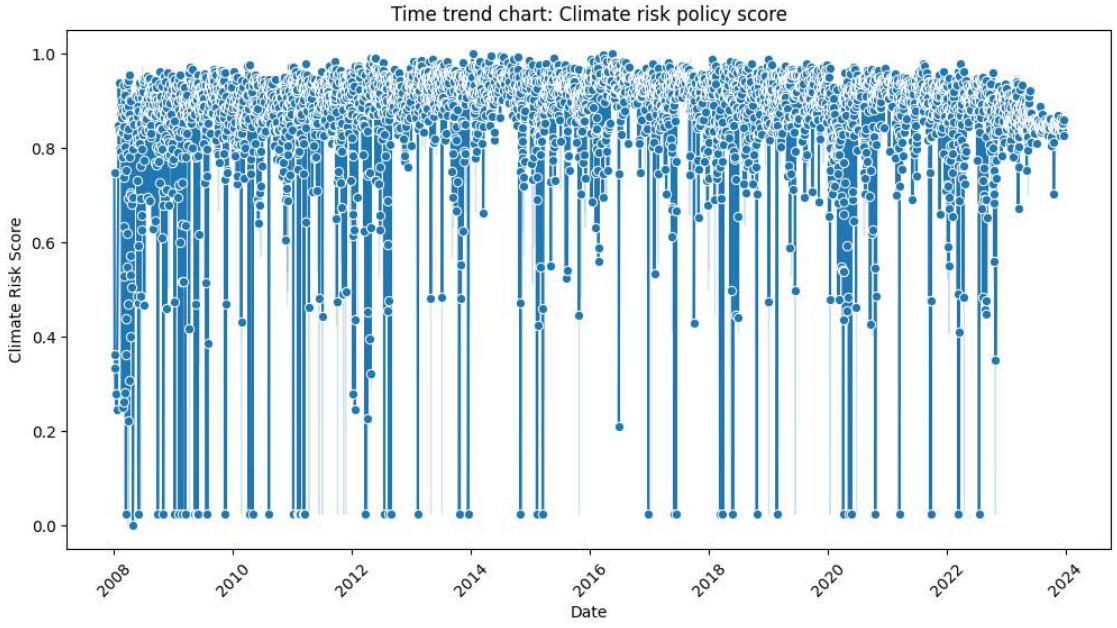
```
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/sklearn/feature_extraction/text.py:525: UserWarning: The parameter
'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(
Building prefix dict from the default dictionary …
Dumping model to file cache /var/tmp/pbs.152919.pbs/jieba.cache
Loading model cost 0.538 seconds.
Prefix dict has been built successfully.
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/sklearn/feature_extraction/text.py:408: UserWarning: Your stop_words
may be inconsistent with your preprocessing. Tokenizing the stop words generated
tokens ['##', 'lex', '~', '±', '÷', ' ', ' ', ' ', ' ', ' ', ' ', '…', ' ', ' ',
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
' ', ' '] not in stop_words.
  warnings.warn(

Document-term matrix created with shape: (4720, 963499)
```

[11]:
```
#
cos_similarities = cosine_similarity(dt_matrix_simple, dt_matrix_simple)
#
average_similarities = np.mean(cos_similarities, axis=1)
#          0 1
normalized_scores = (average_similarities - np.min(average_similarities)) / (np.
  ↪max(average_similarities) - np.min(average_similarities))
#
df['climate_risk_score'] = normalized_scores

print(df)
```

```
           date                                                text  counts  \
6920  2019-07-06    1919       14            …        0
9400  2023-11-08           \n                …        4
7134  2019-11-05    11 4                     …       14
5255  2016-10-14    10 13      2016             …        8
4865  2015-12-22  "               "         …       18
…          …                                                …      …
8296  2021-11-04  "  "       80              …       12
2207  2011-08-16                  G6        …        9
4284  2015-02-07                           …        0
746   2009-04-08     \n\n2009-04-08 21:14:22\n\n"       …        4
4337  2015-03-07          "  " \n\n2011-03-07 14:53:55\…        0

      climate_risk_score
6920            0.931499
```

```
9400              0.841911
7134              0.903362
5255              0.954025
4865              0.930872
…                    …
8296              0.920219
2207              0.974617
4284              0.917432
746               0.970430
4337              0.881646

[4720 rows x 4 columns]
```

[12]:
```
# 1.
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='date', y='climate_risk_score', marker='o')
plt.title('Time trend chart: Climate risk policy score')
plt.xlabel('Date')
plt.ylabel('Climate Risk Score')
plt.xticks(rotation=45)
plt.show()
# 2.
#
sample_texts = df.sample(n=5, random_state=42)[['date', 'text',␣
 ↪'climate_risk_score']]
print("Sample texts and their climate risk policy scores ")
for idx, row in sample_texts.iterrows():
    print(f"date: {row['date']},score: {row['climate_risk_score']}\n")
```



```
Sample texts and their climate risk policy scores
date: 2017-06-27,score: 0.9476288793110467

date: 2022-03-27,score: 0.9783435736089121

date: 2023-04-25,score: 0.8987938496253238

date: 2008-02-07,score: 0.7915006525012187

date: 2009-12-23,score: 0.934460156437096
```

[13]:
```
############################Word2Vec#####################
#########################################################
def apply_preprocessing(data, item_type, stopwords_type, replacing_dict,␣
↪pattern, punctuation):
    """ Function to apply the steps from the preprocessing class in the correct
        order to generate a term frequency matrix
    """

    # initialize the class with the text data and some parameters
    prep = pc.RawDocs(data)

    # replace some specific phrases of interest
    prep.phrase_replace(replace_dict=replacing_dict,
                        sort_dict=True,
                        case_sensitive_replacing=False)

    # lower-case text and expand contractions
    prep.basic_cleaning(lower_case=True,
                        contraction_split=True)

    # split the documents into tokens
    prep.tokenize_text(tokenization_pattern=pattern)

    # clean tokens (remove non-ascii characters, remove short tokens, remove␣
↪punctuation and numbers)
    #
    # prep.token_clean(length=2,                    punctuation=punctuation,  ␣
↪                 numbers=True)

    # remove stopwords
    if item_type == "tokens":
        prep.stopword_remove(items='tokens', stopwords=stopwords_type)
    elif item_type == "lemmas":
```

```
        prep.lemmatize()
        prep.stopword_remove("lemmas", stopwords=stopwords_type)
    elif item_type == "stems":
        prep.stem()
        prep.stopword_remove("stems", stopwords=stopwords_type)

    return prep

# define tokenization pattern and punctuation symbols
pattern = r"(?u)\b\w\w+\b"
punctuation = string.punctuation.replace("-", "")

# use preprocessing class
prep = apply_preprocessing(df["text"],      # our documents
                           "stems",              # tokens, stems or lemmas
                           "long",               # long or short
                           {},                   # dictionary with expressions␣
 ↪we want to preserve
                           pattern,              # tokenization pattern
                           punctuation           # string with punctuation␣
 ↪symbols to remove
                           )
```

```
Reading data from iterator
```

[14]:
```
# define dictionary for preprocessing class with terms we want to preserve
replacing_dict = {
    ' ': ' ',
    ' ': ' ',
    '  ': '  ',
    ' ': ' ',
    ' ': ' ',
    ' ': ' ',
    '   ': '   ',
    '   ': '   ',
    '  ': '  ',
    '  ': '  ',
    ' ': ' ',
    ' ': ' ',
    ' ': ' ',
    '   ': '   ',
    '   ': '   ',
    '  ': '  ',
    ' ': ' ',
    '   ': '   ',
    '   ': '   ',
    ' ': ' ',
```

```
    ' ': ' ',
    ' ': ' ',
    ' ': ' ',
    ' ': ' ',
    ' ': ' ',
    '  ': '  ',
    '  ': '  ',
    ' ': ' '
}
```

[15]:
```
# train Gensim's Word2Vec model (takes less than a minute)
gensim_model = Word2Vec(sentences=prep.tokens,      # corpus
                        vector_size=100,            # embedding dimension
                        window=5,                   # words before and after to␣
 ↪take into consideration
                        sg=1,                       # use skip-gram
                        negative=10,                # number of negative␣
 ↪examples for each positive one
                        alpha=0.025,                # initial learning rate
                        min_alpha=0.0001,           # minimum learning rate
                        epochs=5,                   # number of passes through␣
 ↪the data
                        min_count=1,                # words that appear less␣
 ↪than this are removed
                        workers=1,                  # we use 1 to ensure␣
 ↪replicability
                        seed=92                     # for replicability
                       )
```

[16]:
```
# extract the word embeddings from the model
word_vectors = gensim_model.wv
word_vectors.vectors.shape  # vocab_size x embeddings dimension
```

[16]:
```
(17065236, 100)
```

[17]:
```
# use a PCA decomposition to visualize the embeddings in 2D
def pca_scatterplot(model, words):
    pca = PCA(n_components=2, random_state=92)
    word_vectors = np.array([model[w] for w in words])
    low_dim_emb = pca.fit_transform(word_vectors)
    plt.figure(figsize=(21,10))
    plt.scatter(low_dim_emb[:,0], low_dim_emb[:,1], edgecolors='blue', c='blue')
    plt.xlabel("Component 1")
    plt.ylabel("Component 2")

    # get the text of the plotted words
    texts = []
```

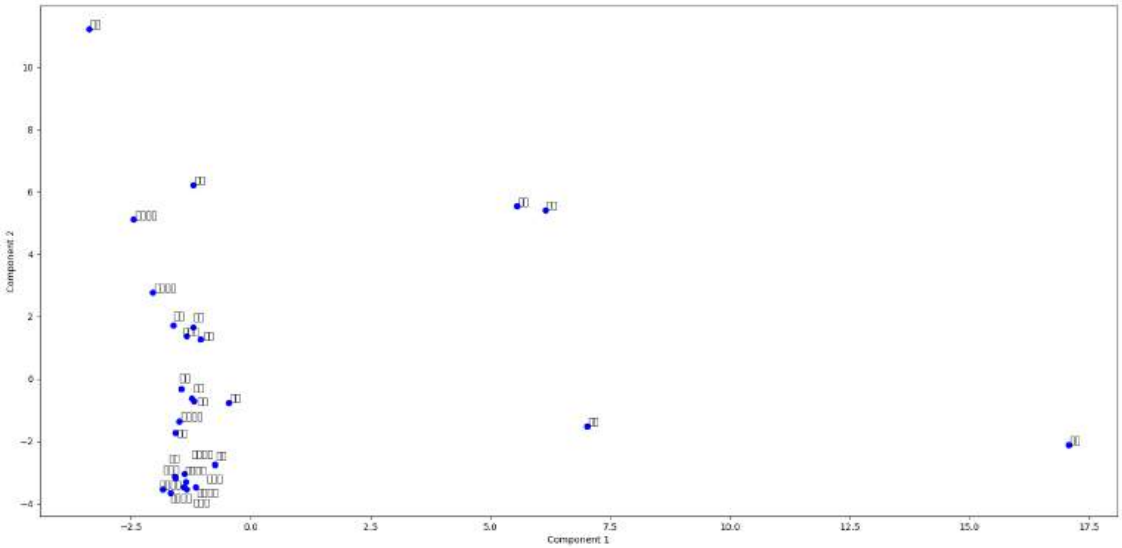
```
    for word, (x,y) in zip(words, low_dim_emb):
        texts.append(plt.text(x+0.01, y+0.01, word, rotation=0))

    # adjust the position of the labels so that they dont overlap
    adjust_text(texts)
    # show plot
    plt.show()
```

[18]:
```
from adjustText import adjust_text
```

[19]:
```
# define the tokens to use in the plot
tokens_of_interest = [" ", " ", "  ", " ", " ", " ", "   ", "   ", "  ", "  ",␣
 ↪" ", " ", " ",
                      "   ", "   ", "  ", " ", "   ", "   ", " ", " ", " ", " ",␣
 ↪" ", " ",
                      "   ", "   ", " "]

# expand the list of tokens with all the tokens from the replacement dictionary
tokens_of_interest = set(tokens_of_interest + list(replacing_dict.values()) )

# plot
pca_scatterplot(word_vectors, list(tokens_of_interest))
```

```
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 29615 (\N{CJK UNIFIED
IDEOGRAPH-73AF}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 22659 (\N{CJK UNIFIED
IDEOGRAPH-5883}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 20445 (\N{CJK UNIFIED
IDEOGRAPH-4FDD}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 25252 (\N{CJK UNIFIED
IDEOGRAPH-62A4}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 29983 (\N{CJK UNIFIED
IDEOGRAPH-751F}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/adjustText/__init__.py:367: UserWarning: Glyph 24577 (\N{CJK UNIFIED
IDEOGRAPH-6001}) missing from current font.
  ax.figure.draw_without_rendering()
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
```

```
packages/IPython/core/pylabtools.py:152: UserWarning: Glyph 27745 (\N{CJK
UNIFIED IDEOGRAPH-6C61}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/rds/general/user/yz1117/home/anaconda3/envs/RCS_TEST/lib/python3.11/site-
packages/IPython/core/pylabtools.py:152: UserWarning: Glyph 38477 (\N{CJK
UNIFIED IDEOGRAPH-964D}) missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```



[20]:
```
# create a positive dictionary by finding the nearest neighbors to a␣
 ↪combination of relevant words
N = 20
pos_center_terms = [' ', '  ']
pos_nn = [w for w, _ in word_vectors.most_similar(positive=pos_center_terms,␣
 ↪topn=N)]
pos_word2vec = pos_center_terms + pos_nn
print(pos_word2vec)
```

```
[' ', '  ', ' ', ' ', ' ', '  ', ' ', '   ', '   ', '    ', ' ',
'  ', ' ', ' ', ' ', ' ', ' ', ' ', '    ', '  ', '  ', '  ']
```

[21]:
```
#  CountVectorizer
vectorizer = CountVectorizer(vocabulary=list(pos_word2vec))
X = vectorizer.fit_transform(df["text"])
#
cos_similarities = cosine_similarity(X, X)
#
average_similarities = np.mean(cos_similarities, axis=1)
#           0 1
```

```
normalized_scores = (average_similarities - np.min(average_similarities)) / (np.
 ↪max(average_similarities) - np.min(average_similarities))
#
df['climate_risk_score2'] = normalized_scores

#
word_freq = pd.DataFrame(X.toarray(), columns=vectorizer.
 ↪get_feature_names_out())
#
df["climate_risk_score3"] = word_freq.sum(axis=1)

#
print(df)
```

```
           date                                                text  counts  \
6920  2019-07-06    1919       14             …        0
9400  2023-11-08           \n                …        4
7134  2019-11-05    11 4                     …       14
5255  2016-10-14    10 13      2016            …        8
4865  2015-12-22  "                "         …       18
…          …                                                 …       …
8296  2021-11-04  "  "       80              …       12
2207  2011-08-16                   G6        …        9
4284  2015-02-07                            …        0
746   2009-04-08     \n\n2009-04-08 21:14:22\n\n"       …        4
4337  2015-03-07          "   " \n\n2011-03-07 14:53:55\…        0

      climate_risk_score  climate_risk_score2  climate_risk_score3
6920            0.931499             0.000000                  NaN
9400            0.841911             0.798439                  NaN
7134            0.903362             0.360553                  NaN
5255            0.954025             0.000000                  NaN
4865            0.930872             0.549365                  NaN
…                  …                    …                    …
8296            0.920219             0.716943                  NaN
2207            0.974617             0.478582                  0.0
4284            0.917432             0.000000                  5.0
746             0.970430             0.290580                  4.0
4337            0.881646             0.000000                 27.0

[4720 rows x 6 columns]
```

[ ]:
```
##############################################################################
##############################################LLM#############################
%%capture

# install required libraries
```

```
!pip3 install transformers                    # HuggingFace library for␣
 ↪interacting with BERT (and multiple other models)
!pip3 install datasets                        # HuggingFace library to process␣
 ↪dataframes
!pip3 install sentence-transformers           # library to use Sentence␣
 ↪Similarity BERT
!pip3 install bertviz                         # visualize BERT's attention weigths
!pip3 install annoy                           # Spotify's library for finding␣
 ↪nearest neighbours
!pip3 install ipywidgets
```

[ ]:
```
# import libraries
import gdown
import pandas as pd
import numpy as np
import gdown
import random
from tqdm.auto import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
import torch

from transformers import AutoModel, BertModel, AutoTokenizer,␣
 ↪BertForSequenceClassification, pipeline, TrainingArguments, Trainer, utils
from transformers.pipelines.base import KeyDataset
from datasets import load_dataset, load_dataset_builder, load_from_disk,␣
 ↪Dataset, DatasetDict

from gensim.models import Word2Vec
import gensim.downloader as api

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import f1_score, precision_score, recall_score,␣
 ↪accuracy_score

# from google.colab import output
# output.enable_custom_widget_manager()

# test GPU
print(f"GPU: {torch.cuda.is_available()}")
```

[ ]:
```
# load a tokenizer using the name of the model we want to use
sec_tokenizer = AutoTokenizer.from_pretrained("nlpaueb/sec-bert-base")

# inspect the configuration of the tokenizer
```

```
sec_tokenizer
```

[ ]:
```
# load the model trained on 10-K report using its name
sec_model = AutoModel.from_pretrained("nlpaueb/sec-bert-base")

# put model in evaluation model (we will not do any training)
sec_model = sec_model.eval()
```

[ ]:
```
# get only the NAICS2 code for each sentence to use as the labels for our␣
 ↪regression
labels = df[["date"]]
```

[ ]:
```
# create list with all the indexes of available sentences
sent_idxs = list(range(0, len(labels)))
len(sent_idxs)
```

[ ]:
```
# perform a train/test split
train_idxs, test_idxs = train_test_split(sent_idxs, test_size=0.2,␣
 ↪random_state=92)
print(f" Train sentences: {len(train_idxs)}\n", f"Test sentences:␣
 ↪{len(test_idxs)}")
```

[ ]:
```
# format the train data adequately
df_finetune = df.loc[train_idxs].copy()

df_finetune = df_finetune[["text", "climate_risk_score"]]
df_finetune.columns = ["text", "label"]

# transform labels into integers
df_finetune["label"] = df_finetune["label"].astype(int)
df_finetune

# map labels from original sector code to ints from 0 to num_sectors
num_sectors = len(df_finetune.groupby('label').size())
label2id_label = {k:v for k,v in zip(df_finetune.groupby('label').size().index.
 ↪values, range(0, num_sectors))}
df_finetune["label"] = df_finetune["label"].apply(lambda x: label2id_label[x])
df_finetune
```

[ ]:
```
# format the test data adequately
df_test = df.loc[test_idxs].copy()

df_test = df_test[["text", "climate_risk_score"]]
df_test.columns = ["text", "label"]

# transform labels into integers
df_test["label"] = df_test["label"].astype(int)
```

```
df_test

# map labels from original sector code to ints from 0 to num_sectors
df_test["label"] = df_test["label"].apply(lambda x: label2id_label[x])
df_test
```

[ ]:
```
# transform data into Dataset class
finetune_dataset = Dataset.from_pandas(df_finetune)
test_dataset = Dataset.from_pandas(df_test)
```

[ ]:
```
# tokenize the dataset
def tokenize_function(examples):
    return sec_tokenizer(examples["text"], max_length=60, padding="max_length",␣
 ↪truncation=True)

tokenized_ft = finetune_dataset.map(tokenize_function, batched=True)    #␣
 ↪batched=True is key for training
tokenized_test = test_dataset.map(tokenize_function, batched=True)

tokenized_ft
```

[ ]:
```
from transformers import AutoModelForSequenceClassification
```

[ ]:
```
from transformers import AutoModelForSequenceClassification, Trainer,␣
 ↪TrainingArguments
import accelerate
```

[ ]:
```
# load the model for finetunning.
# NOTE that we use a different class from the transformers library:
# AutoModel vs. AutoModelForSequenceClassification
num_labels = len(df_finetune.groupby('label').size())
model_ft = AutoModelForSequenceClassification.from_pretrained("nlpaueb/
 ↪sec-bert-base",
                                                              ␣
 ↪num_labels=num_labels,
                                                              ␣
 ↪output_hidden_states=False)
```

[ ]:
```
# define the main arguments for training
training_args = TrainingArguments("./",                                  # path to␣
 ↪save model
                                  learning_rate=3e-5,                    # we use␣
 ↪a very small learning rate
                                  num_train_epochs=5,                    # number␣
 ↪of iterations through the corpus
                                  per_device_train_batch_size=8,
                                  per_device_eval_batch_size=1,
```

```
                                    evaluation_strategy="no",
                                    save_strategy="no")
```

[ ]:
```
# define the set of metrics to be computed through the training process
def compute_metrics(eval_pred):
    metric1 = load_metric("precision")
    metric2 = load_metric("recall")
    metric3 = load_metric("f1")
    metric4 = load_metric("accuracy")

    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    precision = metric1.compute(predictions=predictions, references=labels,␣
 ↪average="micro")["precision"]
    recall = metric2.compute(predictions=predictions, references=labels,␣
 ↪average="micro")["recall"]
    f1 = metric3.compute(predictions=predictions, references=labels,␣
 ↪average="micro")["f1"]
    accuracy = metric4.compute(predictions=predictions,␣
 ↪references=labels)["accuracy"]

    return {"precision": precision, "recall": recall,
            "f1": f1, "accuracy": accuracy}

# by default the Trainer will use MSEloss from (torch.nn) for regression and
# CrossEntropy loss for classification
trainer = Trainer(
    model=model_ft,
    args=training_args,
    train_dataset=tokenized_ft,
    eval_dataset=tokenized_ft,  # in-sample evaluation
    compute_metrics=compute_metrics
)
```

[ ]:
```
# train model (should take around 5 minutes with GPU)
trainer.train()

# save final version of the model
#trainer.save_model("./models/")
```

[ ]:
```
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model_ft = AutoModelForSequenceClassification.
 ↪from_pretrained("bert-base-uncased", num_labels=num_sectors)
```

[ ]:
```
# use the trained model to predict climate risk scores for all texts in df
def predict_climate_risk(texts):
```

```
    encodings = tokenizer(texts, padding=True, truncation=True,␣
 ↪return_tensors="pt")
    predictions = trainer.predict(encodings).predictions
    scores = np.argmax(predictions, axis=-1)
    return scores

# apply the model to df['text']
df['climate_risk_score'] = predict_climate_risk(df["text"].tolist())

# print the updated DataFrame with the new scores
print(df[["text", "climate_risk_score"]])
```

[ ]:

[ ]:

[ ]:

[ ]:

NLP文本分析

Text Mining Code

01

Dictionary Methods

This notebook illustrates how to use list of terms (i.e. dictionaries) to **measure certain aspects of text data**. Particularly, we will show how to build an index that *captures the sentiment of policymakers towards the economy*. Methodologically, we will use regular expressions to *count the number of appereances of words* in the text.

```
1. import pandas as pd
2. import numpy as np
3. import gdown
4. import re
5. from sklearn.preprocessing import StandardScaler
6. import matplotlib.pyplot as plt
```

```
1. # define paths and seed
2. seed = 92
3. data_path = "./"
```

data_url_dict 是一个字典，字典的键是数据集的名称（例如 "mpc_minutes" 和 "ons_quarterly_gdp"），值是一个元组，包含两个元素：第一个元素是 Google Drive 文件的下载链接。第二个元素是文件的类型（例如 "txt" 表示文本文件，"csv" 表示逗号分隔值文件）。

```
1. # define dictionary with paths to data in Google Drive
2. data_url_dict = {"mpc_minutes":
("https://drive.google.com/uc?id=1k086m12gBHKuVMPv4GUD1aB_kZOZpznR", "txt"),
3.                  "ons_quarterly_gdp":
("https://drive.google.com/uc?id=1_8JwCNUsq9wnPiTDmcJ2xrWklGGDFbQD", "csv")
4.                  }
```

```
1. # download each file in the dictionary from Google Drive
2. for file_name, attributes in data_url_dict.items():
遍历 data_url_dict 字典的每一个键值对。file_name 是字典的键（即数据集的名称），attributes 是字典的值（即包含下载链接和文件类型的元组）。
3.     url = attributes[0]
4.     extension = attributes[1]
提取链接和文件类型：url = attributes[0] 提取元组中的第一个元素，即下载链接；extension = attributes[1] 提取元组中的第二个元素，即文件类型。
5.     gdown.download(url, f"{file_name}.{extension}", quiet=False)
下载文件：gdown.download(url, f"{file_name}.{extension}", quiet=False) 使用 gdown 模块下载文件。url 是下载链接，f"{file_name}.{extension}" 是保存文件的名称和扩展名，quiet=False 表示在下载过程中显示进度条。
```

0. Load data

This tutorial uses text data from **the monetary policy discussions at the Bank of England**. Concretely, we use the minutes from the monthly Monetary Policy Committee meetings. You can browse through all of these minutes here or explore one particular minute from September 2017 here.

As a first step, we inspect the data. We observe that we have a total number of 7,277 strings of data (hereafter **documents**) with an associated date (YYYYMM) that ranges between 06/1997 and 10/2014. The average length of a document is roughly 161 words.

```
1. data = pd.read_csv(data_path + 'mpc_minutes.txt', delimiter='\t', header=0,
names=['date', 'minutes'])
2. print(data.shape)
3. data.head()
4. # turn date into date format and create quarter column
5. data['date'] = pd.to_datetime(data.date, format='%Y%m')
6. data['year'] = data.date.dt.year.astype(int)
7. data['quarter'] = data.date.dt.quarter.astype(int)
8. data['date'] = data['date'].dt.strftime('%Y%m')
9. data.head()
```

	date	minutes	year	quarter
0	199706	1 Sections I to V of this minute summarise t...	1997	2
1	199706	The 12-month growth rate of notes and coins ...	1997	2
2	199706	Broad money, too, decelerated in April: its ...	1997	2
3	199706	Lending growth continued at around 9% in ...	1997	2
4	199706	Lending to individuals remained robust in...	1997	2

```
1. # get a sense of the number of words per document
2. data['length'] = data.minutes.apply(lambda x: len(x.split()))
3. print(f'average document length: {np.round(np.mean(data.length),0)} words')
4. # inspect the coverage of our data
5. print('n. months:', len(data.date.unique()))
6. print('n. years:', len(data.year.unique()))
```

```
average document length: 161.0 words
n. months: 209
n. years: 18
```

1. Applying a sentiment dictionary

Consider a situation where we want to investigate **the overall sentiment of each monetary policy meeting**. This migth be a very interesting quantity since it reflects the perception that policy makers have of the current state of the economy.

A simple and sensible way to estimate this quantity would be to **build a document-sentiment index** based on the difference between the number of words having a "positive" sentiment and those having a "negative" sentiment in a document. For each document i we could calculate:

$$\text{sentiment}_i = \frac{\#\text{ Positive terms}_i - \#\text{ Negative terms}_i}{\#\text{ Positive terms}_i + \#\text{ Negative terms}_i}$$

We will do precisely this at the year-quarter level and will then evaluate the resulting index by comparing it with the UK's GDP growth.

Firstly, we need to define the terms having positive or negative sentiment. For this

example, we use a set of terms from Apel and Blix-Grimaldi (2012) who analyzed the minutes from the Swedish Central Bank. In general, it is a good practice to **build on dictionaries** that have been already used in the literature.

```
pos_words = ["high", "higher", "highest",
             "strong", "stronger", "strongest",
             "increase", "increases", "increased", "increasing",
             "fast", "faster", "fastest"]
neg_words = ["low", "lower", "lowest",
             "weak", "weaker", "weakest",
             "decrease", "decreases", "decreased", "decreasing",
             "slow", "slower", "slowest"]
```

We can immediately notice that the dictionaries are lowercased, while the minutes data is not. When searching the minutes for sentiment terms, we do not want casing to interefere. Therefore, it is a good idea to **lower-case the minutes data first**. To do so, we can call the method `lower()` on each string.

```
# lowercase all text
data['minutes_lower'] = data.minutes.apply(lambda x: x.lower())
```

By using regular expressions, we can now **count the number of times** a positive or negative words was used in each of the minutes.

```
# function for counting the number of words from a list that appear on a give text
def count_instances(text, target_list):
text（要搜索的文本）和 target_list（包含目标单词的列表）
    target_string = '|'.join(target_list)
将目标单词列表中的单词用竖线 | 连接起来，形成一个目标字符串。例如，如果 target_list 是
['apple', 'banana']，那么 target_string 就是 'apple|banana'
    matches = re.findall(fr"\b({target_string})\b", text)
使用正则表达式查找文本中与目标字符串匹配的单词。\b 表示单词边界，确保只匹配完整的单词。
return len(matches)
# apply counting function
data['pos_counts'] = data.minutes_lower.apply(lambda x: count_instances(text=x,
target_list=pos_words))
data['neg_counts'] = data.minutes_lower.apply(lambda x: count_instances(text=x,
target_list=neg_words))
```

We can now **compute sentiment at the year-quarter level**, and **compare it to the UK's GDP growth**.

```
# aggregate all minutes to the year-quarter level (by summing all the counts)
data_agg = data.loc[:, ['year', 'quarter', 'pos_counts',
'neg_counts']].groupby(['year', 'quarter']).sum()
```

选择数据框中的 year、quarter、pos_counts 和 neg_counts 列，并按 year 和 quarter 分组，然后对每组的数据进行求和。

```
3. # compute sentiment at year-quarter level
4. data_agg['sentiment'] = (data_agg.pos_counts -
data_agg.neg_counts)/(data_agg.pos_counts + data_agg.neg_counts)
```

公式为（正面计数 - 负面计数）/（正面计数 + 负面计数），并将结果存储在 data_agg 数据框的 sentiment 列中。

```
5. data_agg.head()
```

year	quarter	pos_counts	neg_counts	sentiment
1997	2	27	10	0.459459
	3	222	72	0.510204
	4	229	86	0.453968
1998	1	254	114	0.380435
	2	255	132	0.317829

```
1. # load GPD growth data
2. ons = pd.read_csv(data_path + 'ons_quarterly_gdp.csv', names=['label', 'gdp_growth',
'quarter_long'], header=0)


3. ons['year'] = ons.label.apply(lambda x: x[:4]).astype(int)
```

从 label 列中提取前四个字符作为年份，并将其转换为整数。

```
4. ons['quarter'] = ons.label.apply(lambda x: x[6]).astype(int)
```

从 label 列中提取第七个字符作为季度，并将其转换为整数。

```
5. ons = ons[['year', 'quarter', 'gdp_growth']]


6. ons = ons.drop_duplicates().reset_index(drop=True).copy()
```

删除重复行，重置索引，并创建数据框的副本。

```
7. ons.head()
```

	year	quarter	gdp_growth
0	1997	2	1.2
1	1997	3	0.6
2	1997	4	1.3
3	1998	1	0.6
4	1998	2	0.6

```
1. # merge to sentiment data
2. df = data_agg.merge(ons, how='left', on=['year',
'quarter']).reset_index(drop=True).copy()
```

使用 merge 方法将 data_agg 和 ons 数据框按 year 和 quarter 列进行左连接（how='left'）。这意味着 data_agg 中的所有行都会保留，如果 ons 中有匹配的行，则会将其添加到结果中。

reset_index(drop=True) 重置合并后数据框的索引，并丢弃旧索引。copy() 创建数据框的副本，以避免对原始数据框的修改。

```
3. df.head()
```

	year	quarter	pos_counts	neg_counts	sentiment	gdp_growth
0	1997	2	27	10	0.459459	1.2
1	1997	3	222	72	0.510204	0.6
2	1997	4	229	86	0.453968	1.3
3	1998	1	254	114	0.380435	0.6
4	1998	2	255	132	0.317829	0.6

```
4. # # save for later use
5. # df = df[['year', 'quarter', 'sentiment', 'gdp_growth']].copy()
6. # df.to_csv(data_path + 'gdp_sentiment.csv')
7. # explore the correlation between sentiment and GDP
8. print(df[['sentiment', 'gdp_growth']].corr())
```

	sentiment	gdp_growth
sentiment	1.000000	0.271934
gdp_growth	0.271934	1.000000

In spite of its arguable lack of subtlety, here dictionary methods have produced a sentiment indicator that indeed correlates with real activity. Below we **plot standardized series for growth and sentiment**.

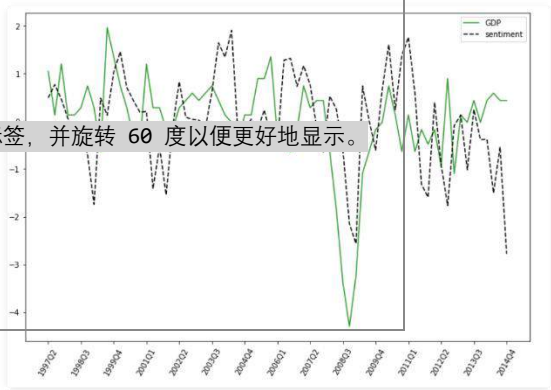
```
1. # plot both time series (after standardizing them)
2. scaler = StandardScaler()
```

创建一个 StandardScaler 对象，用于标准化数据。

```
4. fig, ax = plt.subplots(figsize=(12,8))
```

创建一个大小为 12x8 英寸的图表和轴对象。

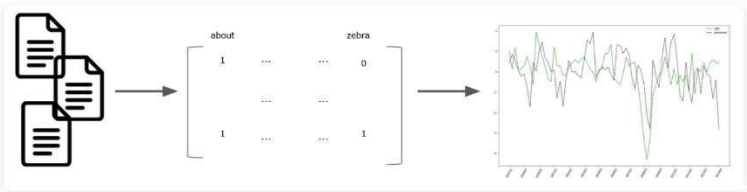
```
ax.plot(scaler.fit_transform(df.gdp_growth.values.reshape(-1, 1)).flatten(),
        label='GDP', c='green', alpha=0.8)
```
将 GDP 增长数据标准化后绘制在图表上，颜色为绿色，透明度为 0.8。
```
ax.plot(scaler.fit_transform(df.sentiment.values.reshape(-1, 1)).flatten(),
        label='sentiment', linestyle='dashed', c='black')
```
将情感得分数据标准化后绘制在图表上，颜色为黑色，线型为虚线。
```

ticks = np.arange(0, df.shape[0], 5)
```
生成从 0 到数据框行数的刻度，每隔 5 个数据点一个刻度。
```
labs = np.array([str(x)+'Q'+str(y) for x,y in zip(df['year'],df['quarter'])])
```
创建一个包含年份和季度标签的数组。
```
labs = labs[ticks]
```
选择对应刻度的标签。
```
ax.set_xticks(ticks)
```
设置 x 轴的刻度。
```
ax.set_xticklabels(labs, rotation=60)
```
设置 x 轴的刻度标签，并旋转 60 度以便更好地显示。
```
ax.legend()
```
添加图例，以区分 GDP 增长和情感得分曲线。
```

plt.show()

```



02

Text preprocessing and dictionary methods

This notebook illustrates multiple strategies in order to **transform raw text into a document-term matrix**. As a starting point, we will **use the built-in functions from the `scikit-learn` library to process our text and generate the document-term matrix**. We will then refine this strategy by applying a more complex **text preprocessing pipeline**.
The document-term matrix can be used as the input to many bag-of-words text analysis algorithms such as dictionary methods, Latent Semantic Analysis (LSA) or Linear Dirichlet Allocation (LDA). Through the notebook, we will focus on a concrete application of **dictionary methods**. We will show how the document-term matrix can be used to measure the sentiment of documents and how this sentiment is correlated with economic variables of interest.



0. Setup

```
%%capture

```

```
# install required libraries
!pip3 install flashtext                # easy phrase replacing methods
!pip3 install contractions             # expand English contractions
!pip3 install --upgrade spacy==2.2.4    # functions for lemmatizing
!pip3 install gdown                    # download files from Google Drive
!pip3 install nltk                     # NLP library

# install Spacy's language model
# for more languages and models check: https://spacy.io/models
!python3 -m spacy download en_core_web_sm

# clone the GitHub repository with the preprocessing scripts
!git clone https://github.com/unstructured-data-research/text-preprocessing

# import libraries
import pandas as pd
import numpy as np
import gdown
import random
import string
import re
import random
import sys
import matplotlib.pyplot as plt
import seaborn as sns
import contractions
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
加载 Spacy 的英语语言模型
import unicodedata
import spacy
nlp_standard = spacy.load('en_core_web_sm')

import nltk
from nltk import SnowballStemmer
stemmer = SnowballStemmer(language='english')

# import our own modules
sys.path.append('./text-preprocessing/src')
import preprocessing_class as pc
import preprocess_data

# define paths and seed
seed = 92
```

```
data_path = "./"

# define dictionary with paths to data in Google Drive
data_url_dict = {"mpc_minutes":
("https://drive.google.com/uc?id=1k086m12gBHKuVMPv4GUD1aB_kZOZpznR", "txt"),
                 "ons_quarterly_gdp":
("https://drive.google.com/uc?id=1_8JwCNUsq9wnPiTDmcJ2xrWklGGDFbQD", "csv"),
                 "gdp_sentiment":
("https://drive.google.com/uc?id=1xxGecTJzbXLMgGE-2-YVKlMD2P0JYmgH", "csv")
                 }

# download each file in the dictionary from Google Drive
for file_name, attributes in data_url_dict.items():
    url = attributes[0]
    extension = attributes[1]
gdown.download(url, f"{file_name}.{extension}", quiet=False)
```

1. Load data

This tutorial uses text data from **the monetary policy discussions at the Bank of England**. Concretely, we use the minutes from the monthly Monetary Policy Committee meetings. You can browse through all of these minutes here or explore one particular minute from September 2017 here.

As a first step, we **inspect the data**. We observe that we have a total number of 7,277 strings of data (hereafter **documents**) with an associated date (YYYYMM) that ranges between 06/1997 and 10/2014. The average length of a document is roughly 161 words.

```
# load the data an inspect it
data = pd.read_csv('mpc_minutes.txt', delimiter='\t', header=0, names=['date',
'minutes']) print(data.shape)
data.head()
# start by reading a document. What are they?
data.loc[929, "minutes"]
```

```
(7277, 2)
```

	date	minutes
0	199706	1 Sections I to V of this minute summarise t...
1	199706	The 12-month growth rate of notes and coins ...
2	199706	Broad money, too, decelerated in April: its ...
3	199706	Lending growth continued at around 9% in ...
4	199706	Lending to individuals remained robust in...

```
'      The Committee noted that the preliminary estimate of 0.2% GDP growth in 1998 Q4 was consistent with the degree of slowdown expecte
d in its November central projection. Some members had feared lower growth, and the possibility of revisions remained. Surveys suggested t
hat consumer and business sentiment had stopped deteriorating and had perhaps improved slightly, possibly helped by the 150 basis point re
duction in official interest rates since October. The preliminary data on Q4 expenditure suggested, however, that household spending might
have grown by less in relation to income and wealth than the Committee had assumed in November.'
```

```
# calculate the average number of words in a document
data["minutes"].apply(lambda x: len(x.split())).mean()
# turn date into date format and create quarter column
data['date'] = pd.to_datetime(data.date, format='%Y%m')
data['year'] = data.date.dt.year.astype(int)
data['quarter'] = data.date.dt.quarter.astype(int)
data['date'] = data['date'].dt.strftime('%Y%m')
data.head()
```

```
160.6952040676103
```

	date	minutes	year	quarter
0	199706	1 Sections I to V of this minute summarise t...	1997	2
1	199706	The 12-month growth rate of notes and coins ...	1997	2
2	199706	Broad money, too, decelerated in April: its ...	1997	2
3	199706	Lending growth continued at around 9% in ...	1997	2
4	199706	Lending to individuals remained robust in...	1997	2

```
# inspect the coverage of our data
print('number of months:', len(data.date.unique()))
print('number of years:', len(data.year.unique()))
```

```
number of months: 209
number of years: 18
```

2. Preprocessing text with scikit-learn

We will **use the CountVectorizer class** from **scikit-learn** in order to both preprocess our text and transform it into a document-term matrix. Check the documentation of the class to learn more about its parameters.

```
# create a CountVectorizer object straight from the raw text
count_vectorizer = CountVectorizer(token_pattern=r"(?u)\b\w\w+\b",      # regular expression to split documents into tokens 正则表达式，用于将文档分割成标记
       lowercase=True,                    # convert all characters to lower-case
       strip_accents="ascii",              # remove non-ascii characters (WARNING: this is dangerous in Romance languages)
       stop_words="english",               # remove stopwords from a built-in list. We can also provide our own list 移除内置的英语停用词列表
       ngram_range=(1, 1),                 # generate only unigrams
       analyzer="word",             # build matrix at the word-level  在词级别构建矩阵
       max_df=0.8,  # ignore tokens that have a higher document frequency (can be int or percent)
       min_df=20, # ignore tokens that have a lower document frequency (can be int or percent)
       max_features=None,      # we could impose a maximum number of vocabulary terms
                                  )
count_vectorizer
```

```
# directly transform our data into a document-term matrix. We just need to pass
# our documents as an argument to the fit_transform function
dt_matrix_simple = count_vectorizer.fit_transform(data["minutes"])
print(f"Document-term matrix created with shape: {dt_matrix_simple.shape}")    # (documents, vocabulary size)
```

```
Document-term matrix created with shape: (7277, 2313)
```

```
# inspect the document-term matrix. Why is it sparse?
dt_matrix_simple
# transform to regular numpy array to easily manipulate.
# this might not be convenient (or even possible) if your matrix is too large
dt_matrix_simple = dt_matrix_simple.toarray()  转换为常规 numpy 数组以便操作
# we can explore the vocaburaly as pairs of (word, position in matrix)
vocabulary = count_vectorizer.vocabulary_
random.sample(list(vocabulary.items()), 10)
```

```
# another way of looking at the vocabulary (ordered by index in the document-term matrix)
count_vectorizer.get_feature_names_out()
```

```
['notable', 1414),
 ('300', 59),
 ('undertaken', 2180),
 ('recruit', 1729),
 ('access', 146),
 ('91', 105),
 ('revival', 1833),
 ('marian', 1301),
 ('mild', 1346),
 ('consumers', 529)]
```

3. Tracking the sentiment towards the economy

Consider a situation where we want to investigate the overall sentiment of each monetary policy meeting. This migth be a very interesting quantity since it reflects the perception that policy makers have of the current state of the economy.

A simple and sensible way to estimate this quantity would be to build a document-sentiment index based on the difference between the number of words having a "positive" sentiment and those having a "negative" sentiment in a document. **We will use our document-term matrix to do precisely this**. For each document i we will calculate:

$$\text{net sentiment}_i = \frac{\#\text{ Positive terms}_i - \#\text{ Negative terms}_i}{\#\text{ Positive terms}_i + \#\text{ Negative terms}_i}$$

We will then evaluate such an index by comparing the measured sentiment with the UK's GDP growth.

Building dictionaries

Firstly, we need to define the terms having positive or negative sentiment. For this example, we use a set of terms from Apel and Blix-Grimaldi (2012) who analyzed the minutes from the Swedish Central Bank. In general, it is a good practice to build on dictionaries that have been already used in the literature.

```
pos_words = ["high", "higher", "highest",
             "strong", "stronger", "strongest",
             "increase", "increases", "increased","increasing",
             "fast", "faster", "fastest"]
neg_words = ["low", "lower", "lowest",
             "weak", "weaker", "weakest",
             "decrease", "decreases", "decreased", "decreasing",
             "slow", "slower", "slowest"]
```

```
# find the position in the document-term matrix of each word from the dictionary
pos_ixs = [v for k,v in vocabulary.items() if k in pos_words]
neg_ixs = [v for k,v in vocabulary.items() if k in neg_words]
# for each document, sum all the positive words
pos_counts = np.take(dt_matrix_simple, pos_ixs, axis=1)
pos_counts = pos_counts.sum(axis=1)
# for each document, sum all the negative words
neg_counts = np.take(dt_matrix_simple, neg_ixs, axis=1)
neg_counts = neg_counts.sum(axis=1)
# add counts to dataframe
data['pos_counts_simple'] = pos_counts
data['neg_counts_simple'] = neg_counts
data
```

	date	minutes	year	quarter	pos_counts_simple	neg_counts_simple
0	199706	1 Sections I to V of this minute summarise t...	1997	2	0	0
1	199706	The 12-month growth rate of notes and coins ...	1997	2	0	0
2	199706	Bank lending decelerated in April: its ...	1997	2	2	0
3	199706	Lending growth continued at around 9% in ...	1997	2	1	0
4	199706	Lending to individuals remained robust in...	1997	2	1	0
...	...	...	...	...	...	...
7272	201410	For most members, there remained insuffi...	2014	4	4	4
7273	201410	Set against this, the level of Bank Rate...	2014	4	0	4
7274	201410	For two members, economic circumstances ...	2014	4	1	1
7275	201410	The Governor invited the Committee to vo...	2014	4	1	0
7276	201410	The following members of the Committee w...	2014	4	0	0

7277 rows × 6 columns

```
# aggregate all minutes to the year-quarter level (by summing all the counts)
data_agg = data.loc[:, ['year', 'quarter', 'pos_counts_simple',
'neg_counts_simple']].groupby(['year', 'quarter'], as_index=False).sum()
data_agg.head()
```

	year	quarter	pos_counts_simple	neg_counts_simple
0	1997	2	27	10
1	1997	3	222	72
2	1997	4	229	86
3	1998	1	254	113
4	1998	2	255	131

```
# compute net positive sentiment
data_agg['net_sentiment_simple'] = (data_agg.pos_counts_simple -
data_agg.neg_counts_simple)/(data_agg.pos_counts_simple + data_agg.neg_counts_simple)
data_agg = data_agg[['year', 'quarter', 'net_sentiment_simple']].copy()
data_agg.head()
```

	year	quarter	net_sentiment_simple
0	1997	2	0.459459
1	1997	3	0.510204
2	1997	4	0.453968
3	1998	1	0.384196
4	1998	2	0.321244

Next we add quarterly GDP data collected from the ONS website in order to compare our sentiment index with GDP growth.

```
# load GDP data
df_gdp = pd.read_csv('ons_quarterly_gdp.csv', names=['label', 'gdp_growth',
'quarter_long'], header=0)
df_gdp['year'] = df_gdp.label.apply(lambda x: x[:4]).astype(int)
df_gdp['quarter'] = df_gdp.label.apply(lambda x: x[6]).astype(int)
df_gdp = df_gdp[['year', 'quarter', 'gdp_growth']]
df_gdp = df_gdp.drop_duplicates().reset_index(drop=True).copy()
df_gdp.head()
```

```
# merge both dataframes
df = data_agg.merge(df_gdp, how='left', on=['year', 'quarter']).copy()
# check correlation betweeen both series
print(df[['gdp_growth', 'net_sentiment_simple']].corr())
```

Downsides of simple preprocessing

Although `scikit-learn` makes it easy to preprocess text and transform it into a document-term matrix, we might want to go beyond the functionality provided by this library. For instance we may want to:

- Split contractions (e.g. don't --> do not)
- Remove all numbers
- Transform words into lemmas
- Transform words into stems
- Read documents while we are transforming them

In the next section, we will show how to implement these custom preprocessing steps.

4. Custom preprocessing pipeline

Basic cleaning

```
# lower case via the lower() built-in method
docs = [s.lower() for s in data.minutes]   #文本转为小写
# remove leading and trailing whitespaces
docs = [d.strip() for d in docs]    #去除空白
# split contractions with the "contractions" library
# the library mainly supports English but it provides the tools to extend its
# functionality to other languages: https://github.com/kootenpv/contractions
contractions.fix("We can't increase the interest rate. They won't accept it. It's too risky.") # (example)
```

```
# apply contractions function
docs = list(map(lambda x: contractions.fix(x), docs))  #将缩写词展开
# tokenize with the spaCy tokenizer
def spacy_tokenizer_standard(sent):     将文本分割成单词和标记
    sent = nlp_standard.tokenizer(sent)
    tokens = [t.text for t in sent]
    return tokens
# apply function
tokens = list(map(spacy_tokenizer_standard, docs))
```

```
# we could use apply our own regular expression
custom_pattern = r'''
                (?x)               # set flag to allow verbose regexps (to separate logical sections of pattern and add comments) 允许详细的正则表达式
                \w+(?:-\w+)*       # word characters with internal hyphens 包含内部连字符的单词字符
                | [][.,;"'?():-_`]  # preserve punctuation as separate tokens 将标点符号作为单独的标记保留
                '''
tokens_custom = list(map(lambda x:nltk.regexp_tokenize(x, pattern=custom_pattern), docs))
# inspect a document to see the differences in the tokenization
i = 10
print(data.loc[i, "minutes"][0:118], "\n")
print(tokens[i][0:16])
print(tokens_custom[i][0:16])
```

```
There was as yet no clear downtrend in net exports. Exports to non-EU countries had risen sharply: the USA had grown

['there', 'was', 'as', 'yet', 'no', 'clear', 'downtrend', 'in', 'net', 'exports', '.', 'exports', 'to', 'non', '-', 'eu']
['there', 'was', 'as', 'yet', 'no', 'clear', 'downtrend', 'in', 'net', 'exports', '.', 'exports', 'to', 'non-eu', 'countries', 'had']
```

```
# remove non-ascii characters via unicodedata package 移除非 ASCII 字符
def remove_non_ascii(tokens):
    tokens = [unicodedata.normalize('NFKD', t).encode('ascii', 'ignore').decode('utf-8', 'ignore') for t in tokens]
    return tokens
# apply function
tokens = list(map(remove_non_ascii, tokens))
```

```
# BRIEF NOTE: Removing non-ascii characters from other languages is very dangerous.
# You will loose a lot of information
```

```
3. unicodedata.normalize('NFKD', "à è ü ç").encode('ascii', 'ignore').decode('utf-8',
'ignore')
```

```
 1. # remove extra white spaces and short tokens via simple list comprehension
 2. def remove_short(tokens, min_length=2): 移除额外空白和短标记
 3.     return [t.strip() for t in tokens if t.strip() != '' and len(t.strip()) >
min_length]
 4. # apply function
 5. tokens = list(map(remove_short, tokens))
 6. # remove punctuation using a regular expression. We can use this same
 7. # structure to remove any other character
 8. def remove_punctuation(tokens, punctuation):           移除标点符号
 9.     regex = re.compile('[%s]' % re.escape(punctuation))
10.     tokens = [regex.sub('', t) for t in tokens]
11.     return tokens
12. # load a list of punctuation signs
13. string.punctuation
14. # apply punctuation removal function
15. tokens = list(map(lambda x: remove_punctuation(x, string.punctuation), tokens))
```

```
 1. # remove stopwords via simple list comprehension 移除停用词
 2. def remove_stopwords(tokens, stopwords):
 3.     return [t for t in tokens if t not in stopwords]
 4. # load a custom list of stopwords
 5. custom_stopwords = list(preprocess_data.stp_long)
 6. custom_stopwords[0:10]
 7. # apply stopword removal function
 8. tokens = list(map(lambda x:
 9.                  remove_stopwords(x, stopwords=custom_stopwords),
10.                  tokens))
```

```
1. # remove numbers  移除数字
 2. def remove_numbers(tokens, min_length=2):
 3.     translation_table = str.maketrans('', '', string.digits)
 4.     tokens = [t.translate(translation_table) for t in tokens if not t.isdigit()]
 5.     return  [t for t in tokens if t != '' and len(t) > min_length]
 6. # apply number removal function
 7. tokens = list(map(lambda x:
 8.                  remove_numbers(x, min_length=2),
 9.                  tokens))
```

Stemming and lemmatization

As a next step, one might attempt to **group together words that are grammatically different but thematically identical**. For example, it could be reasonable to keep words such as 'obligations', 'obligation' and 'oblige' under a single token. Ultimately these three words denote the same concept, and so we might want them to share the same symbol.

The two most popular techniques to achieve this goal are **stemming and lemmatization**.

Stemming replaces each word with its root form. The resulting token is less readable by humans but encompases multiple words (e.g. 'tradition' and 'traditional' have the same stem: 'tradit').

Lemmatization replaces words with their dictionary form i.e. lemma. The words 'changing', 'changes' and 'changed' all have the same lemma: 'change'.

Below, we use both techniques (stemming [via nltk Snowball stemmer](https://www.geeksforgeeks.org/snowball-stemmer-nlp/) and lemmatization via [spaCy lemmatizer](https://spacy.io/api/lemmatizer)) and look at the different results on a specific document.

```
# stem via standard nltk Snowball Stemmer
stemmer = SnowballStemmer(language='english')
def stem(tokens):
    """stem all tokens that do not contain hyphens"""
    return [stemmer.stem(t) if "-" not in t else t for t in tokens]

# apply function
stems = list(map(stem, tokens))
```

```
# inspect a document
i = 10
print(data.loc[i, "minutes"][0:162], "\n")
print(stems[i][0:14])
```

```
There was as yet no clear downtrend in net exports. Exports to non-EU countries had risen sharply: the USA had grown very fast in Q1, bu
t it was expected by US

['yet', 'clear', 'downtrend', 'net', 'export', 'export', 'non', 'countri', 'risen', 'sharpli', 'usa', 'grown', 'fast', 'expect']
```

```
# lemmatize with standard spaCy lemmatizer (takes a couple of minutes)
nlp_standard = spacy.load('en_core_web_sm')
def lemmatize_sent(sent):
    sent = ' '.join(sent)
    doc = nlp_standard(sent)
    lemmas = [token.lemma_ if token.lemma_ != '-PRON-' else token.text for token in
doc]
    return lemmas

# apply function
lemmas = list(map(lemmatize_sent, tokens))
```

```
# compare the same document
i = 10
print(data.loc[i, "minutes"][0:162], "\n")
print("Stems: ", stems[i][0:14], "\n")
print("Lemmas: ", lemmas[i][0:14])
```

```
There was as yet no clear downtrend in net exports. Exports to non-EU countries had risen sharply: the USA had grown very fast in Q1, bu
t it was expected by US

Stems:  ['yet', 'clear', 'downtrend', 'net', 'export', 'export', 'non', 'countri', 'risen', 'sharpli', 'usa', 'grown', 'fast', 'expect']

Lemmas:  ['yet', 'clear', 'downtrend', 'net', 'export', 'export', 'non', 'country', 'rise', 'sharply', 'usa', 'grow', 'fast', 'expect']
```

Restricting the vocabulary by looking at the term frequency

Finally, one might want to **drop very frequent terms**, as they could be regarded as corpus-specific terms that do not add up much value. Similarly, one might want to **drop very rare terms**, considering that not much can be learnt from such infrequent items.

To explore these ideas we will rank tokens according to pure document frequency (df) or term frequency inverse document frequency (tf-idf), where high weight is given to terms that appear frequently in the entire dataset, but in relatively few documents. In particular, the tf-idf score for term v is computed according to the formula below:

$$tfidf_v = \left(1 + log(tf_v)\right)\left(log\ \frac{N}{df_v + 1}\right)$$

```
# simple auxiliary function to override the preprocessing done by sklearn 词频（Term Frequency，TF）进行排名
def do_nothing(doc):
    return doc
# generate document frequency ranking for all terms in the vocabulary
vectorizer = TfidfVectorizer(use_idf=False, norm=None, tokenizer=do_nothing, preprocessor=do_nothing)
df_matrix = vectorizer.fit_transform(tokens).toarray()    拟合并转换文本数据
df_matrix_bool = np.where(df_matrix > 0, 1, 0)   创建布尔矩阵
scores_df = df_matrix_bool.sum(axis=0)  计算每个词出现次数

sorted_vocab = sorted(vectorizer.vocabulary_.items(),   key=lambda x: x[1])
sorted_vocab_keys = list(np.array(sorted_vocab)[:,0])
sorted_scores_df = sorted(set(scores_df), reverse=True)

rank_dict = {k:val for k,val in zip(sorted_scores_df, list(range(len(sorted_scores_df))))}
rank_tup = sorted(zip(scores_df, sorted_vocab_keys),  key=lambda x: x[0], reverse=True)
df_ranking = [x + (rank_dict[x[0]],) for x in rank_tup]
df_ranking[:10]     排名字典、元组、数据框
```

```
[(3431, 'growth', 0),
 (3062, 'inflation', 1),
 (2641, 'rate', 2),
 (2489, 'committee', 3),
 (2388, 'prices', 4),
 (2294, 'market', 5),
 (2163, 'month', 6),
 (1941, 'demand', 7),
 (1905, 'remained', 8),
 (1862, 'output', 9)]
```

As we can see, the token "growth" ranks first, with a document frequency of 3,431 out of 7,277 (i.e. "growth" is present in around 47% of the documents), and "inflation" and "rate" come right after. Let's now look at the top ranked tokens in terms of tf-df:

```
# function to compute term-frequency inverse document frequency 词频-逆文档频率（TF-IDF）进行排名
def tf_idf_compute(term, num_docs, scores_tf, scores_df):
    return (1+np.log(scores_tf[term])*np.log(num_docs/scores_df[term]))

```

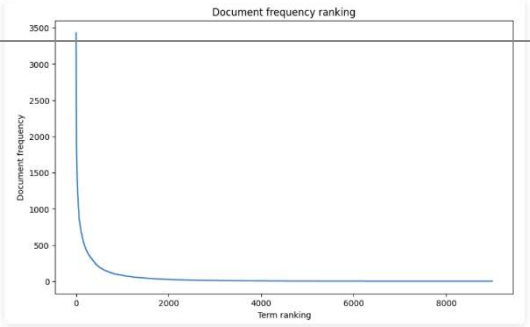
```
# generate term frequency inverse document frequency ranking for all terms in the vocabulary
sorted_vocab = sorted(vectorizer.vocabulary_.items(),   key=lambda x: x[1])
sorted_vocab = list(np.array(sorted_vocab)[:,0])
scores_tf = df_matrix.sum(axis=0)
scores_tfidf = [tf_idf_compute(t, len(tokens), scores_tf=scores_tf, scores_df=scores_df) for t in range(len(sorted_vocab))]
sorted_scores_tfidf = sorted(set(scores_tfidf), reverse=True)

rank_dict = {k:val for k,val in zip(sorted_scores_tfidf, list(range(len(sorted_scores_tfidf))))}
rank_tup = sorted(zip(scores_tfidf, sorted_vocab),  key=lambda x: x[0], reverse=True)
tfidf_ranking = [x + (rank_dict[x[0]],) for x in rank_tup]
tfidf_ranking[:10]
```

```
[(25.89188880741839, 'securitisations', 0),
 (25.517881250415055, 'ips', 1),
 (24.963009728445, 'nfc', 2),
 (24.65859148064065, 'bcc', 3),
 (24.460899792318422, 'pnfcsa', 4),
 (24.163147658906983, 'annex', 5),
 (23.920156435028385, 'respectively', 6),
 (23.786282775779966, 'egarding', 7),
 (23.76832183070415, 'attacks', 8),
 (23.688367959279212, 'surplus', 9)]
```

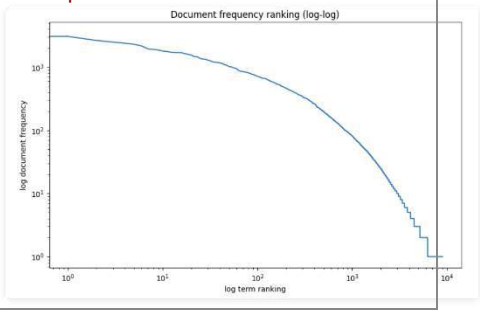
The two methods do not agree. Indeed, tf-idf weights more those terms that are frequent, but in relatively few documents.

Let's now inspect the overall behavior of the document frequency ranking and the tf-idf ranking. As we can see, document frequency decays very fast (following a power-law), while tf-idf decreases at a slower pace.

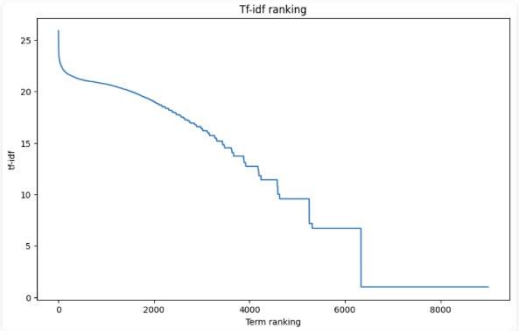
```
# plot document frequency
plt.figure(figsize=(10,6))
plt.plot([x[0] for x in df_ranking])
plt.title('Document frequency ranking')
plt.ylabel("Document frequency")
plt.xlabel("Term ranking")
plt.show()
```




```
# we can use a log-log scale to observe more clearly the power-law distribution (Zipf's law)
plt.figure(figsize=(10,6))
plt.loglog([x[0] for x in df_ranking])
plt.title('Document frequency ranking (log-log)')
plt.ylabel("log document frequency")
plt.xlabel("log term ranking")
plt.show()
```




```
# plot tf-ids ranking
plt.figure(figsize=(10,6))
plt.plot([x[0] for x in tfidf_ranking])
plt.title('Tf-idf ranking')
plt.ylabel("tf-idf")
plt.xlabel("Term ranking")
```



```
plt.show()
```

Using the tfidf ranking, for instance, we can see that removing stems that have a score lower than the score of stems in positions around 5,800 might be reasonable; they are relatively uninformative and lots of them have a similar score.

```
# get the score of the stem in position 58000
tfidf_ranking[5800][0]
# auxiliary function to easily remove undesired tokens
def remove(tokens, to_remove):
    if to_remove:
        return [t for t in tokens if t not in to_remove]
    else:
        return tokens

# function to remove tokens in documents according to their df or tfidf score
def rank_remove(ranking, docs, min_cutoff_score=-np.inf, max_cutoff_score=np.inf):

    # define list of tokens to remove
    to_remove_low = set([t[1] for t in ranking if t[0] <= min_cutoff_score])
    to_remove_high = set([t[1] for t in ranking if t[0] > max_cutoff_score])

    # remove tokens
    docs_clean = [remove(d, to_remove_low) for d in docs]
    docs_clean = [remove(d, to_remove_high) for d in docs]

    return docs_clean

# update documents
stems_clean = rank_remove(tfidf_ranking, stems,
min_cutoff_score=tfidf_ranking[5800][0])

# we can also remove stems that have a very high score
stems_clean = rank_remove(tfidf_ranking, stems_clean, max_cutoff_score=20)
```

Vectorization

Now that we have finished preprocessing our text we will use again scikit-learn to transform our list of tokens, lemmas or stems into a document-term matrix. Check the documentation for the [CountVectorizer](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html) and the [TfidfVectorizer](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.text.TfidfVectorizer) to explore all the parameters and functionalities of these classes.

```
1. # create a CountVectorizer object straight from the raw
   textcount_vectorizer = CountVectorizer(preprocessor=do_nothing, # apply no additional
preprocessing
 2.             tokenizer=do_nothing,                # apply no additional tokenization
 3.             lowercase=False,                     # convert all characters to lower-
case
 4.             strip_accents=None,                  # remove non-ascii characters
 5.             stop_words=None,                     # remove stopwords from a built-in
list. We can also provide our own list
 6.             ngram_range=(1, 1),                  # generate only unigrams
 7.             analyzer='word',                     # analysis at the word-level
 8.             max_df=1.0,    # ignore tokens that have a higher document frequency (can
be int or percent)
 9.             min_df=0,      # ignore tokens that have a lowe document frequency (can
be int or percent)
10.             max_features=None,    # we could impose a maximum number of vocabulary
terms
11.             )
12. count_vectorizer
```

```
1. # create the document term matrix using stems
2. dt_matrix_custom = count_vectorizer.fit_transform(stems_clean).toarray()
3. vocabulary_custom = count_vectorizer.vocabulary_
4. print(len(vocabulary_custom))
```

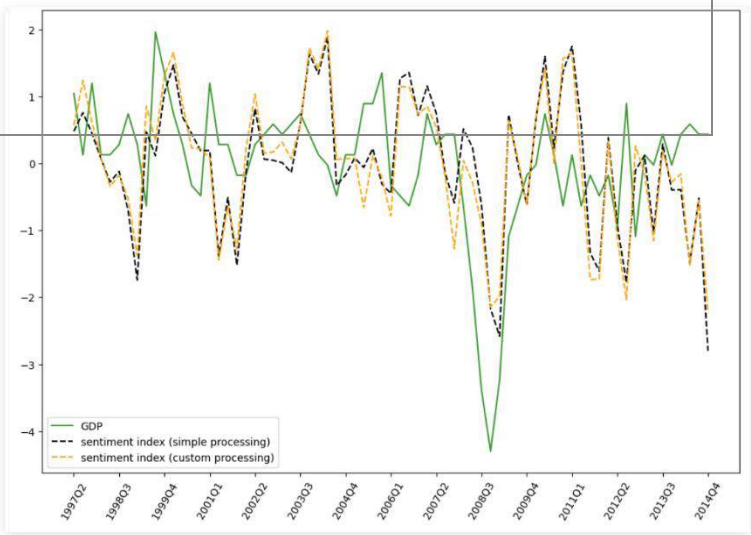
```
 1. # we can also create a weighted vectorizer object
 2. # however we will not use it in the analysis
 3. weighted_vectorizer = TfidfVectorizer(preprocessor=do_nothing,             # apply
no additional preprocessing
 4.tokenizer=do_nothing, # apply no additional tokenization
 5.lowercase=False,      # convert all characters to lower-case
 6.strip_accents=None,   # remove non-ascii characters
 7.stop_words=None,      # remove stopwords from a built-in list. We can also provide our
own list
 8.ngram_range=(1, 1),   # generate only unigrams
 9.analyzer='word',      # analysis at the word-level
10.max_df=0.8,    # ignore tokens that have a higher document frequency (can be int or
percent)
11.min_df=20,     # ignore tokens that have a lowe document frequency (can be int or
percent)
12.max_features=None,    # we could impose a maximum number of vocabulary terms
13.
14.use_idf=True,  # wether to apply inverse document frequency weights
15.smooth_idf=True,      # add +1 to idf weighting
16.sublinear_tf=True     # add +1 to log(tf)
17.             )
```

```
weighted_vectorizer
```

Dictionary methods

```
# transform dictionary terms into stems (in order for them to match our new vocabulary)
pos_stems = [stemmer.stem(t) for t in pos_words]
pos_stems = list(set(pos_stems))
neg_stems = [stemmer.stem(t) for t in neg_words]
neg_stems = list(set(neg_stems))

# In[62]:

# find positions of the stems in the document-term matrix
pos_ixs = [v for k,v in vocabulary_custom.items() if k in pos_stems]
neg_ixs = [v for k,v in vocabulary_custom.items() if k in neg_stems]

# In[63]:

# count positive stems per document
pos_counts_custom = np.take(dt_matrix_custom, pos_ixs, axis=1)
pos_counts_custom = pos_counts_custom.sum(axis=1)

# count negative stems per document
neg_counts_custom = np.take(dt_matrix_custom, neg_ixs, axis=1)
neg_counts_custom = neg_counts_custom.sum(axis=1)

# add counts to dataframe
data['pos_counts_custom'] = pos_counts_custom
data['neg_counts_custom'] = neg_counts_custom
data

# aggregate to year-quarter level
data_agg = data.groupby(['year', 'quarter']).sum(numeric_only=True).reset_index()
data_agg.head()

# compute net sentiment
data_agg['net_sentiment_simple'] = (data_agg.pos_counts_simple - data_agg.neg_counts_simple)/(data_agg.pos_counts_simple + data_agg.neg_counts_simple)
data_agg['net_sentiment_custom'] = (data_agg.pos_counts_custom - data_agg.neg_counts_custom)/(data_agg.pos_counts_custom + data_agg.neg_counts_custom)

data_agg = data_agg[['year', 'quarter', 'net_sentiment_simple', 'net_sentiment_custom']].copy()
data_agg.head()
```

```
38.
39. # merge to GDP data
40. df = data_agg.merge(df_gdp, how='left', on=['year', 'quarter']).copy()
41. df.head()
42.
43. # Looking at correlations, we see that the new sentiment index is slightly more
correlated with GDP than our initial measure.
44.
45. print(df[['gdp_growth', 'net_sentiment_simple', "net_sentiment_custom"]].corr())
46.
47. # plot time series
48. scaler = StandardScaler()
49. ticks = np.arange(0, df.shape[0], 5)
50. labs = np.array([str(x)+'Q'+str(y) for x,y in zip(df['year'],df['quarter'])])
51. labs = labs[ticks]
52.
53. fig, ax = plt.subplots(figsize=(12,8))
54. ax.plot(scaler.fit_transform(df.gdp_growth.values.reshape(-1, 1)).flatten(),
55.         label='GDP', c='green', alpha=0.8)
56.
57. ax.plot(scaler.fit_transform(df["net_sentiment_simple"].values.reshape(-1,
1)).flatten(),
58.         label='sentiment index (simple processing)', linestyle='dashed', c='black')
59.
60. ax.plot(scaler.fit_transform(df["net_sentiment_custom"].values.reshape(-1,
1)).flatten(),
61.         label='sentiment index (custom processing)', linestyle='dashed', c='orange')
62.
63. ax.set_xticks(ticks)
64. ax.set_xticklabels(labs, rotation=60)
65. ax.legend()
66. plt.show()
67.
```



03

Dimensionality reduction with LDA

Open in Colab

This notebook introduces how to use the `lda` Python library for estimating Latent Dirichlet Allocation using the collapsed Gibbs sampling algorithm of Griffiths and Steyvers (2004).

To illustrate LDA, the tutorial uses text data obtained from minutes from the Monetary Policy Committee meetings at the Bank of England. You can browse through all of these minutes here.

In [1]:
```
%%capture

# install required libraries
!pip3 install flashtext                    # easy phrase replacing methods
!pip3 install contractions                 # expand English contractions
!pip3 install spacy                        # functions for lemmatizing
!pip3 install gdown                        # download files from Google Drive
!pip3 install nltk                         # NLP library
!pip3 install lda                          # LDA library
!pip3 install wordcloud                    # library for wordclouds

# install Spacy's language model
# for more languages and models check: https://spacy.io/models
!python3 -m spacy download en_core_web_sm
```

In [2]:
```
# clone the GitHub repository with the preprocessing scripts
!git clone https://github.com/unstructured-data-research/text-preprocessing
```

```
Cloning into 'text-preprocessing'...
remote: Enumerating objects: 31, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 31 (delta 11), reused 17 (delta 4), pack-reused 0
Unpacking objects: 100% (31/31), 1.95 MiB | 5.28 MiB/s, done.
```

In [3]:
```
# import required libraries
import gdown
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sys
import string
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from wordcloud import WordCloud, STOPWORDS
import matplotlib.colors as mcolors
import lda

import spacy
nlp_standard = spacy.load('en_core_web_sm')

# import our own modules
```

```
sys.path.append('./text-preprocessing/src')
import preprocessing_class as pc
import preprocess_data
```

In [4]:
```
# define dictionary with paths to data and Python scripts in Google Drive
urls_dict = {"mpc_minutes": ("https://drive.google.com/uc?id=1k086m12gBHKuVMPv4G
             }
```

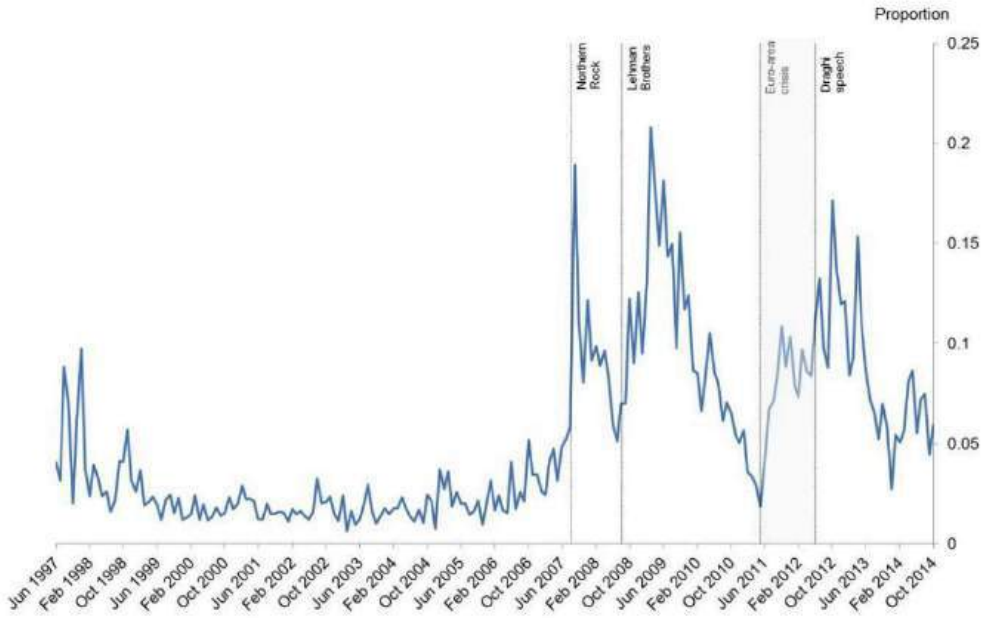
In [5]:
```
# download all files
for file_name, attributes in urls_dict.items():
    url = attributes[0]
    extension = attributes[1]
    gdown.download(url, f"./{file_name}.{extension}", quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1k086m12gBHKuVMPv4GUD1aB_kZOZpznR
To: /content/mpc_minutes.txt
100%|██████████| 7.36M/7.36M [00:00<00:00, 41.2MB/s]
```

MPC Minutes

We will illustrate LDA using the MPC minutes data. We will re-create one particular figure in Andy Haldane's speech *Central bank psychology* in which he estimates the share of time the MPC minutes spend discusing banking. The figure is below:

Chart 13: Estimate of the proportion of MPC minutes covering the topic of 'banking'



In [6]:
```
# load the MPC data
data = pd.read_csv("mpc_minutes.txt", sep="\t")
data
```

Out[6]:

	year	minutes
0	199706	1 Sections I to V of this minute summarise t...
1	199706	The 12-month growth rate of notes and coins ...
2	199706	Broad money, too, decelerated in April: its ...
3	199706	Lending growth continued at around 9% in ...
4	199706	Lending to individuals remained robust in...
...	...	...
7272	201410	For most members, there remained insuffi...
7273	201410	Set against this, the level of Bank Rate...
7274	201410	For two members, economic circumstances ...
7275	201410	The Governor invited the Committee to vo...
7276	201410	The following members of the Committee w...

7277 rows × 2 columns

In [7]:
```
# explore one observation
data.loc[0, "minutes"]
```

Out[7]:
```
'  1 Sections I to V of this minute summarise the analysis presented to the MPC
by Bank staff and the BankÃ\x95s regional agents, and incorporate also informat
ion that became available to the Committee after the presentation but prior to
the 6 June meeting.     Monetary conditions'
```

Cleaning Raw Text Data

Before implementing a topic model, it is important to pre-process the data. To do this, we will use a custom-built preprocessing Python library. This library implements a wide-range of functions for cleaning text. If you want to inspect the source code the public repository is here.

For a more detailed discussion on preprocessing check the **2_preprocessing.ipynb notebook** within this same repository.

我们将首先创建一个函数，以正确的顺序应用所有预处理步骤。我们应提供要预处理的文档和一些关键参数。该函数将返回一个对象预处理，我们稍后将用它来创建文档术语矩阵。

We will start by creating a function that applies all the preprocessing steps in the correct order. We should provide the documents that we want to preprocess along side some key parameters. The function will return an object `prep` that we will use to later create a document-term matrix.

In [8]:
```
def apply_preprocessing(data, item_type, stopwords_type, replacing_dict, pattern
    """ Function to apply the steps from the preprocessing class in the correct
        order to generate a term frequency matrix
    """

    # initialize the class with the text data and some parameters
    prep = pc.RawDocs(data)
```

```
    # replace some specific phrases of interest
    prep.phrase_replace(replace_dict=replacing_dict,
                        sort_dict=True,
                        case_sensitive_replacing=False)

    # lower-case text and expand contractions
    prep.basic_cleaning(lower_case=True,
                        contraction_split=True)

    # split the documents into tokens
    prep.tokenize_text(tokenization_pattern=pattern)

    # clean tokens (remove non-ascii characters, remove short tokens, remove pun
    prep.token_clean(length=2,
                     punctuation=punctuation,
                     numbers=True)

    # remove stopwords
    if item_type == "tokens":
        prep.stopword_remove(items='tokens', stopwords=stopwords_type)
    elif item_type == "lemmas":
        prep.lemmatize()
        prep.stopword_remove("lemmas", stopwords=stopwords_type)
    elif item_type == "stems":
        prep.stem()
        prep.stopword_remove("stems", stopwords=stopwords_type)

    return prep
```

In [9]:
```
# define tokenization pattern and punctuation symbols
pattern = r"(?u)\b\w\w+\b"
punctuation = string.punctuation.replace("-", "")
```

In [10]:
```
# use preprocessing class
prep = apply_preprocessing(data["minutes"],      # our documents
                           "stems",              # tokens, stems or lemmas
                           "long",               # long or short
                           {},                   # dictionary with expressions w
                           pattern,              # tokenization pattern
                           punctuation           # string with punctuation symbo
                           )
```

```
Reading data from iterator
```

In [11]:
```
# inspect a particular tokenized document and compare it to its original form
i = 10
print(data["minutes"][i])
print("\n ------------------------------ \n")
print(prep.stems[i])
```

```
 There was as yet no clear downtrend in net exports. Exports to non-EU countries
had risen sharply: the USA had grown very fast in Q1, but it was expected by US c
ommentators to slow down spontaneously. Net exports to EU countries had been resi
lient. GDP growth in France and Germany (though not Italy) had picked up to just
below trend, but domestic demand growth in those countries had been subdued.

 ------------------------------

['yet', 'clear', 'downtrend', 'net', 'export', 'export', 'non', 'countri', 'rise
n', 'sharpli', 'usa', 'grown', 'veri', 'fast', 'expect', 'comment', 'slow', 'spon
tan', 'net', 'export', 'countri', 'resili', 'gdp', 'growth', 'franc', 'germani',
'though', 'itali', 'pick', 'trend', 'domest', 'demand', 'growth', 'countri', 'sub
du']
```

The final step in pre-processing is to drop remaining words that are not useful for identifying content. We have already dropped standard stopwords, but there may also be data-dependent common words. For example, in data from Supreme Court proceedings, "justice" might be treated as a stopword. Also, words that appear just once or twice in the collection are not informative of content either. Ideally, one would like a measure of informativeness that both punishes common words in the data, and rare words. One such option is to give each stem a tf-idf (term frequency - inverse document frequency) score. This is standard in the language processing literature, so we omit details here.

In [12]:
```
# generate document frequency ranking for stems
prep.get_term_ranking("stems", score_type="df")
```

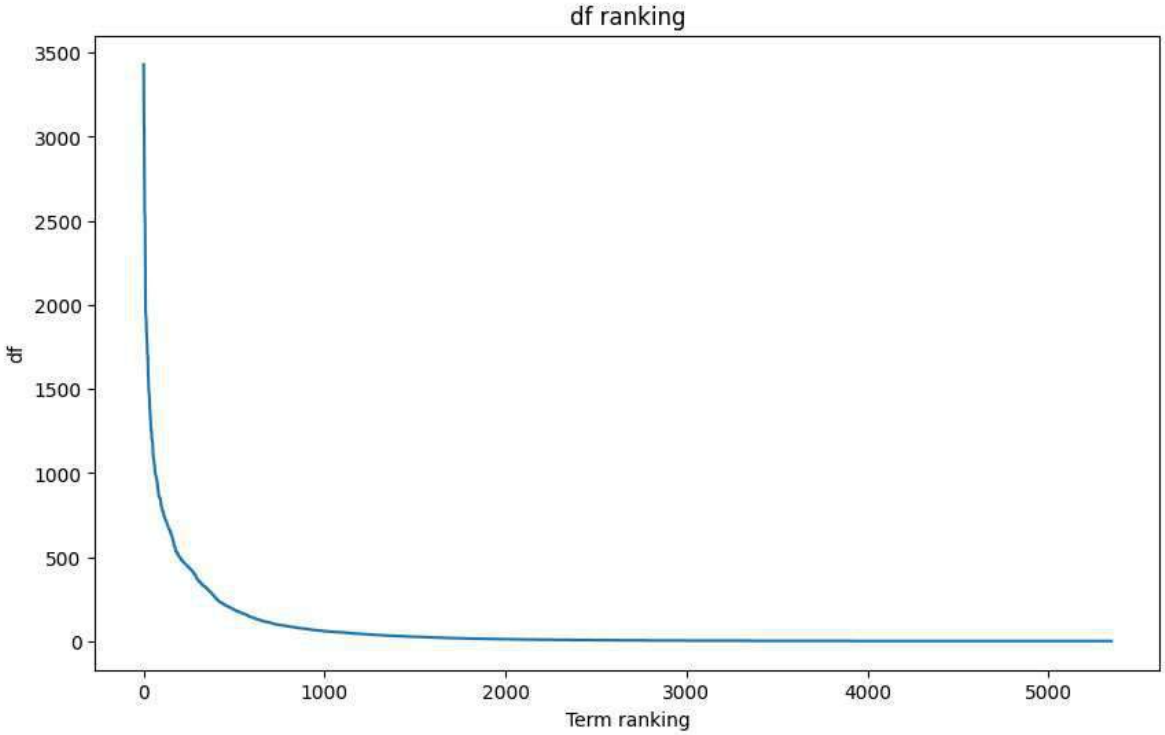
df方法

```
/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:528: U
serWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is n
ot None'
  warnings.warn(
```

In [13]:
```
# explore top stems
prep.df_ranking["stems"][0:10]
```

Out[13]:
```
[(3431.0, 'growth', 0),
 (3350.0, 'rate', 1),
 (3065.0, 'inflat', 2),
 (3061.0, 'month', 3),
 (2852.0, 'price', 4),
 (2767.0, 'market', 5),
 (2558.0, 'increas', 6),
 (2530.0, 'committe', 7),
 (2527.0, 'expect', 8),
 (2259.0, 'year', 9)]
```

In [14]:
```
# plot df ranking
plt.figure(figsize=(10,6))
plt.plot([x[0] for x in prep.df_ranking["stems"]])
plt.title('df ranking')
plt.ylabel("df")
plt.xlabel("Term ranking")
plt.show()
```



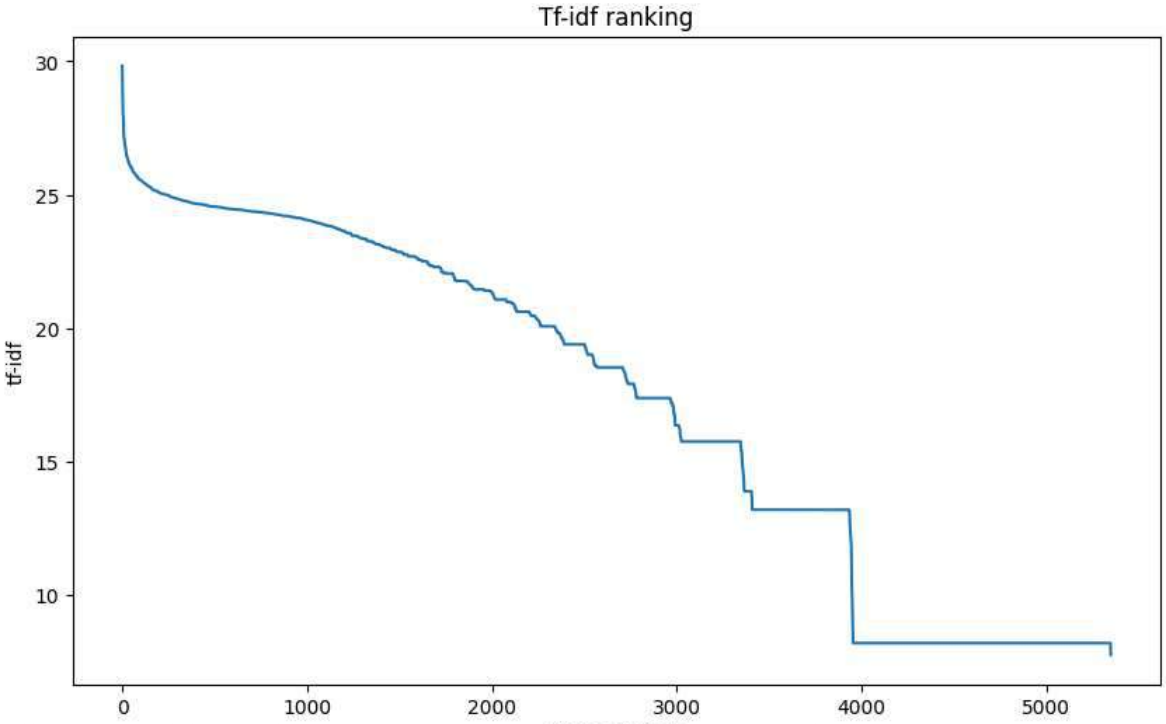
In [15]:
```
# generate term frequency inverse document frequency ranking of stems
prep.get_term_ranking("stems", score_type="tfidf")
```

tfidf方法

In [16]:
```
# explore top stems
prep.tfidf_ranking["stems"][0:10]
```

Out[16]:
```
[(29.825243257254172, 'nfc', 0),
 (29.417855025455637, 'ip', 1),
 (28.780350233635524, 'securitis', 2),
 (28.20096850526625, 'surveya', 3),
 (28.09284413266011, 'pnfcsa', 4),
 (28.013583133605582, 'attack', 5),
 (27.89033249950207, 'bcc', 6),
 (27.731835312191592, 'egard', 7),
 (27.221006498334184, 'annex', 8),
 (27.177586953744225, 'gear', 9)]
```

In [17]:
```
# plot tfidf ranking
plt.figure(figsize=(10,6))
plt.plot([x[0] for x in prep.tfidf_ranking["stems"]])
plt.title('Tf-idf ranking')
plt.ylabel("tf-idf")
plt.xlabel("Term ranking")
plt.show()
```



rank_remove 的第一个参数定义了是删除 “词块 ” 还是 “词干”（因为我们是根据词干进行排名的，所以应该指定词干），第二个参数指定了使用的排名方法（“df ”或 “tfidf”）。

At this stage, the user can decide how many stems to drop based on either the df or tf-idf scores. The first argument to rank_remove defines whether to drop from "tokens" or "stems" (since we formed the rankings based on stems above, we should specify stems) and the second specifies the ranking method to use ("df" or "tfidf"). Finally, you can either provide a *min_cutoff_value* which will remove all items with a score below the value or a *max_cutoff_value* which will remove all items with a score above the value.

最后，您可以提供一个 min_cutoff_value（最小截断值）或 max_cutoff_value（最大截断值），前者将删除所有得分低于该值的项目，后者将删除所有得分高于该值的项目。

One can determine the cutoffs from exploring rankings that we have already created. Here we plot the ranking in Python, which indicates a reasonable cutoff minimum cutoff around the term in position 3,500.

In [18]:
```
# remove stems with a tfidf score equal or lower than the score of stem 3,500
cutoff_score = prep.tfidf_ranking["stems"][3500][0]
cutoff_score
```

Out[18]:
```
13.19615492862266
```

In [19]:
```
# apply removal
prep.rank_remove("stems", "tfidf", min_cutoff_score=cutoff_score)
```

In [20]:
```
# create document-term matrix using stems
prep.dt_matrix_create(items='stems', score_type='df')
# extract document-term matrix
dt_matrix_stems = prep.df_matrix["stems"].astype(int)
print(dt_matrix_stems.shape)
```

```
(7277, 3408)
```

获取词汇和适当的字典，以便从索引映射到单词

In [21]:
```
# get the vocabulary and the appropriate dictionaries to map from indices to wor
word2idx = prep.vocabulary["stems"]
idx2word = {i:word for word,i in word2idx.items()}
```

In [22]:
```
all_stems = [s for d in prep.stems for s in d]
print("number of unique stems = %d" % len(set(all_stems)))
print("number of total stems = %d" % len(all_stems))
```

```
number of unique stems = 3408
number of total stems = 496792
```

After pre-processing, we have 3,408 unique stems We now proceed to estimate a topic model on them.

Estimating a Topic Model

The first step in estimation is to initialize a model using the LDA class. We will pass the list of stems we just created as the set of documents.

There are three main parameters in LDA, the number of topics, and the two hyperparameters of the Dirichlet priors. We will follow the advice of Griffiths and Steyvers (2004) and set the hyperparameter of the Dirichlet prior on topics to $200/V$, where V is the number of unique vocabulary elements, and the hyperparameter of the Dirichlet prior on document-topic distributions to $50/K$, where K is the number of topics. 参数设置遵循文献

In [23]:
```
# create LDA object with our parameters
num_topics = 30
vocab_size = dt_matrix_stems.shape[1]
lda_model = lda.LDA(n_topics=num_topics,          # Number of topics
                    alpha=50/num_topics,          # Dirichlet parameter for distr
                    eta=200/vocab_size,           # Dirichlet parameter for distr
                    n_iter=5000,                  # Number of sampling iterations
                    random_state=92               # The generator used for the in
                    )

lda_model
```

采样迭代次数
初始主题使用的生成器

Out[23]:
```
<lda.lda.LDA at 0x7fe1a47b3e80>
```

In [24]:
```
# check some paramters
print(lda_model.alpha)
print(lda_model.eta)
print(lda_model.n_topics)
```

```
1.6666666666666667
0.05868544600938967
30
```

In [25]:
```
# estimate model (takes around 7 minutes)
lda_model.fit(dt_matrix_stems)
```

Out[25]:
```
<lda.lda.LDA at 0x7fe1a47b3e80>
```

In [26]:
```
# topics: probability distribution over the vocabulary (num_topics x vocab_size)
lda_model.topic_word_.shape
```

Out[26]:
```
(30, 3408)
```

In [27]:
```
# distribution of topics over documents (num_documents x num_topics)
lda_model.doc_topic_.shape
```

Out[27]: (7277, 30)

The estimated topics are represented by a Num Topics × Vocab Size matrix (30 × 3408 in our case) whose rows sum to one, while the estimated distribution of topics within documents are represented by a Num Documents × Num Topics matrix (7277 × 30 in our case) rows sum to one as well.

To get an idea of the topics that have been estimated, and whether they make sense, we will explore the stems with the highest probability for each topic. It's a good idea to check the topics are "reasonable" before proceeding with any analysis.为了了解已估算出的主题以及它们是否合理，我们将探索每个主题中概率最高的词干。在进行任何分析之前，最好先检查一下这些主题是否“合理”。

In [28]:
```
# explore the words with highest probability per topic
topics_dist = lda_model.topic_word_ 获取主题-词分布矩阵
n_top_words = 15 设置要显示的词汇数量
for i, topic_dist in enumerate(topics_dist):
    top_idxs = np.argsort(topic_dist)[:-n_top_words:-1]获取每个主题中概率最高的词汇索引
    top_words = [idx2word[idx] for idx in top_idxs] 从 idx2word 字典中获取对应的词汇
    print('Topic {}: {}'.format(i, ' '.join(top_words)))
```

Topic 0: world slowdown activ global countri econom emerg prospect outlook affect particular japan slow far
Topic 1: risk target medium outlook downsid balanc member upsid near uncertainti abov prospect view news
Topic 2: decis immedi develop discuss befor financi cost turn credit money note intern world consider
Topic 3: oil higher reflect pressur commod import suppli futur produc input upward good past energi
Topic 4: invest level possibl weak busi lower relat past howev low sector earli perhap surpris
Topic 5: recoveri activ indic although half modest onli pace pickup global sustain prospect weak pick
Topic 6: sterl effect exchang depreci relat appreci dollar movement sinc chang sterlinga past meet around
Topic 7: employ earn labour unemploy pay measur sector settlement wage averag work privat lfs bonus
Topic 8: cpi line ahead near measur dure pre time impact reflect energi around abov part
Topic 9: trade domest export net contribut import posit weak account larg strong offset extern good
Topic 10: interest point basi short chang impli meet reduct forward offici probabl futur curv longer
Topic 11: possibl chang ani effect difficult clear becaus much explan exampl run affect factor yet
Topic 12: unit euro area state kingdom confid indic countri non germani weak fiscal pmi european
Topic 13: quarter gdp revis estim third expenditur fourth upward nation account latest releas previous invest
Topic 14: bank asset purchas monetari meet stock billion programm reserv stimulus size agre condit financ
Topic 15: littl seem evid news weaker sign although consist stronger grow broad appear look robust
Topic 16: project central forecast committeea period abov assum target view path assumpt publish around outlook
Topic 17: equiti yield bond financi spread govern corpor dure major debt reflect share volatil low
Topic 18: hous retail sale indic consum strong survey confid consumpt slow mortgag distribut balanc institut
Topic 19: governor member respons vote present proposit deputi repres paul follow monetari favour financi bean
Topic 20: consumpt spend household real incom nomin consum adjust tax past fiscal budget support save
Topic 21: product pressur capac margin cost wage extent labour spare suppli degre depend impact slack
Topic 22: august fallen juli septemb sinc octob june index averag risen previous slight level measur
Topic 23: novemb januari decemb fallen februari octob sinc risen slight previous index declin point averag
Topic 24: survey servic manufactur sector banka agent busi region balanc cip order industri cbi quarter
Topic 25: risen fallen annual point percentag sector manufactur index compar unchang exclud euro lend invest
Topic 26: strong sinc though reflect dure billion mpc twelv staff annual firm stock larg fell
Topic 27: bank credit lend condit money financi compani fund borrow non capit household broad hold
Topic 28: need member reduct view given tighten current cut repo interest argument place case domest
Topic 29: may april march fallen risen june slight februari index previous sinc averag level measur

制作词云图

In [29]:
```
# define a set of colors for each topic
cols = [color for name, color in mcolors.XKCD_COLORS.items()]  # more colors: 'm
n_top_words = 15
# create a wordcloud object
cloud = WordCloud(background_color='white',
                  width=2500,
                  height=1800,
                  max_words=n_top_words,
                  colormap='tab10',
                  color_func=lambda *args, **kwargs: cols[i],
                  prefer_horizontal=1.0)
```

创建了一个包含3行5列子图的图形，所有子图共享x轴和y轴，并将整个图形的大小设置为16x12英寸。

In [30]:
```
# create figure for the first 15 topics
fig, axes = plt.subplots(3, 5, figsize=(16,12), sharex=True, sharey=True)
for i, ax in enumerate(axes.flatten()):

    # select top words from topic
    topic_dist = topics_dist[i]
    top_idxs = np.argsort(topic_dist)[:-n_top_words:-1]
    top_words = [idx2word[idx] for idx in top_idxs]
    top_probs = [topic_dist[idx] for idx in top_idxs]
    plot_dict = {top_words[i]:top_probs[i] for i in range(len(top_words))}

    # generate subplot
    fig.add_subplot(ax)
    cloud.generate_from_frequencies(plot_dict, max_font_size=300)
    plt.gca().imshow(cloud)
    plt.gca().set_title('Topic ' + str(i), fontdict=dict(size=16))
    plt.gca().axis('off')
```

axes.flatten()将2D数组的子图转换为1D数组

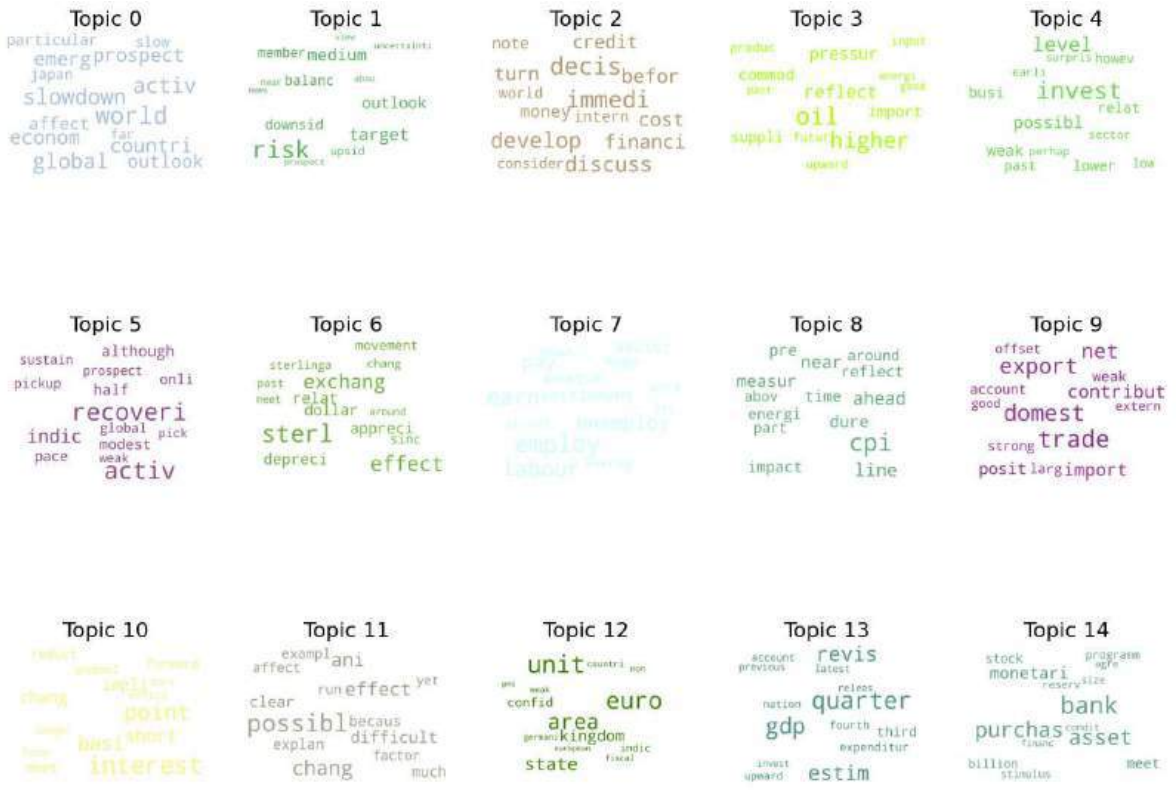
获取当前主题的词分布

按词概率降序排序并选择前n_top_words个索引

将这些索引映射到对应的词

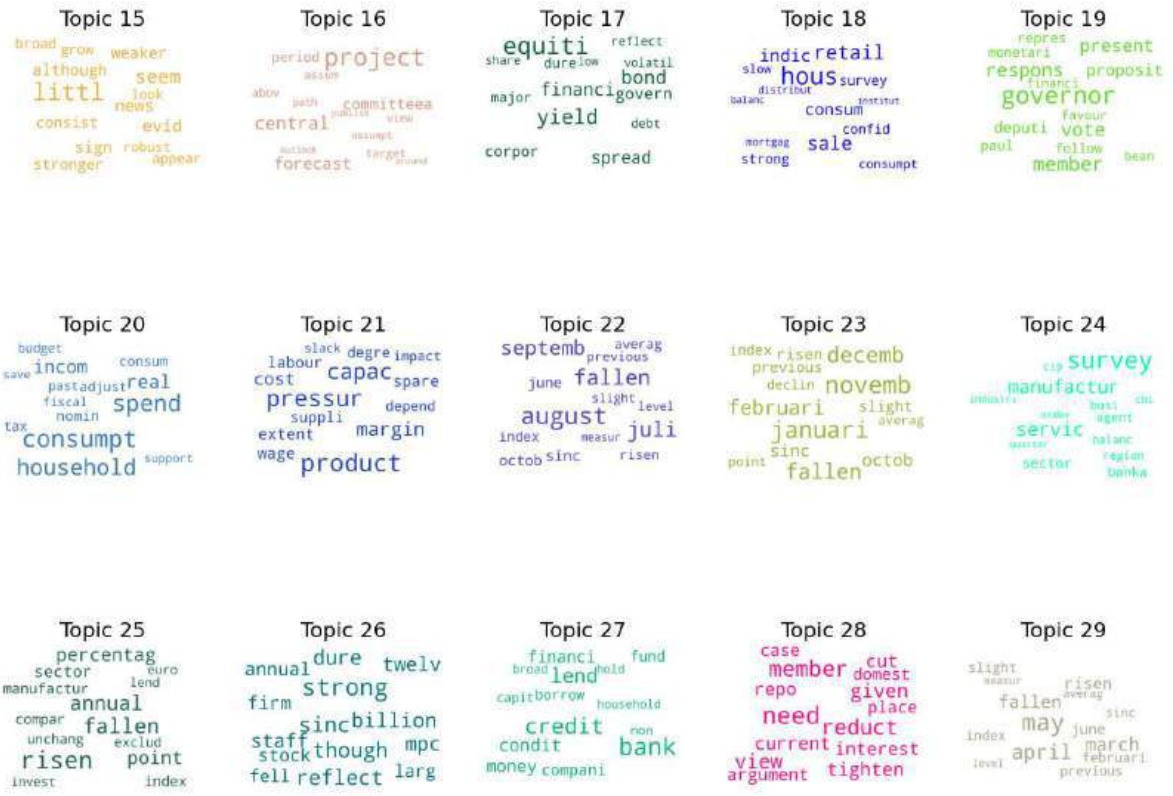
获取这些词的概率

创建一个词和其概率的字典，用于绘图



In [31]:
```
# create figure for the last 15 topics
fig, axes = plt.subplots(3, 5, figsize=(16,12), sharex=True, sharey=True)
for i, ax in enumerate(axes.flatten(), 15):

    # select top words from topic
```

```
    topic_dist = topics_dist[i]
    top_idxs = np.argsort(topic_dist)[:-n_top_words:-1]
    top_words = [idx2word[idx] for idx in top_idxs]
    top_probs = [topic_dist[idx] for idx in top_idxs]
    plot_dict = {top_words[i]:top_probs[i] for i in range(len(top_words))}

    # generate subplot
    fig.add_subplot(ax)
    cloud.generate_from_frequencies(plot_dict, max_font_size=300)
    plt.gca().imshow(cloud)
    plt.gca().set_title('Topic ' + str(i), fontdict=dict(size=16))
    plt.gca().axis('off')
```



In this particular topic model, topic 27 appears to be about banking and financial markets.

In [32]:
```
n_top_words = 20
target_topic = 27
topic_dist = topics_dist[target_topic]
top_idxs = np.argsort(topic_dist)[:-n_top_words:-1]
top_words = [idx2word[idx] for idx in top_idxs]
print('Topic {}: {}'.format(target_topic, ' '.join(top_words)))
```

```
Topic 27: bank credit lend condit money financi compani fund borrow non capit hou
sehold broad hold corpor cost sector mortgag avail
```

Querying Using Estimated Topics

估算未包含在估算中的文档的主题分布

After estimating a topic model, one is often interested in estimating the distribution of topics for documents not included in estimation. In this case, one option is to *query* those documents by holding fixed the topics estimated from LDA, and only estimating the distribution of topics for the out-of-sample

通过固定 LDA 估计出的主题来查询这些文档，而只估计样本外文档的主题分布

documents. The `lda` library also provides a way to do this through the `transform()` function.

在每分钟内的段落层次上估计主题

We will apply querying to the **complete minutes of meetings** that are produced at the end of each MPC (recall that we estimated topics on the level of the paragraph within each minute). In terms of estimating topics, the paragraph level is preferable to the meeting level since individual paragraphs are more likely to be based around a single theme. But, in terms of econometric work, the entire minute of a meeting is a more natural unit of analysis. At the same time, there is no general way of "adding up" probability distribution at the paragraph level in order to arrive at a meeting-level distribution. Hence the need for querying, which allows us to estimate the meeting-level distributions.

段落层面比会议层面更好，因为单个段落更有可能围绕一个主题

通过查询来估算会议层面的分布

In [33]:
```
# replace the original text with its cleaned and stemmed version
data['minutes_stems'] = [' '.join(s) for s in prep.stems]

# aggregate up to the meeting level using the stemmed version of the text
agg_minutes = data.groupby(['year'], as_index = False)['minutes_stems'].apply(la
agg_minutes
```

(lambda x:''.join(x))

使用词干化后的文本，将数据按年份聚合到会议级别。每一行代表一个完整的货币政策委员会（MPC）会议的文本

Out[33]:

	year	minutes_stems
0	199706	section minut summaris analysi present mpc ban...
1	199707	section minut summaris analysi present mpc ban...
2	199708	section minut summaris analysi present mpc ban...
3	199709	meet took place background earlier present ban...
4	199710	inut mpc meet octob meet preced present bank s...
...	...	...
204	201406	befor turn immedi decis discuss financi develo...
205	201407	befor turn immedi decis discuss financi develo...
206	201408	befor turn immedi decis backdrop latest projec...
207	201409	befor turn immedi decis discuss financi develo...
208	201410	befor turn immedi decis discuss financi develo...

209 rows × 2 columns

我们的数据中有一行是货币政策委员会的完整词干文本

In [39]:
```
# at this point one row in our data is the complete stemmed text of a Monetary P
agg_minutes.loc[0, "minutes_stems"][0:204] + "  [CONTINUED ... ]"
```

Out[39]:
```
'section minut summaris analysi present mpc bank staff banka region agent incor
por inform becam avail present prior june meet monetari condit note coin fallen
sinc januari fallen april provision estim may   [CONTINUED ... ]'
```

In [40]:
```
# simple auxiliary function to override the preprocessing done by sklearn
def do_nothing(doc):
    return doc

# generate document-term matrix of agreggated speeches
```

```
count_vectorizer = CountVectorizer(preprocessor=do_nothing,                # apply
                                   token_pattern=pattern,                  # use s
                                   lowercase=False,                        # conve
                                   strip_accents=None,                     # remov
                                   stop_words=None,                        # remov
                                   vocabulary=word2idx                     # impos
                                   )          不移除各种词
count_vectorizer
```

Out[40]:

CountVectorizer

```
CountVectorizer(lowercase=False,
                preprocessor=<function do_nothing at 0x7fe09a35f6d0>,
                vocabulary={'ab': 0, 'abandon': 1, 'abat': 2, 'abcp': 3,
                            'abil': 4, 'abl': 5, 'abnorm': 6, 'abolit': 7,
                            'abour': 8, 'abov': 9, 'abroad': 10, 'abrupt':
11,
                            'absenc': 12, 'absent': 13, 'absolut': 14,
                            'absorb': 15, 'absorpt': 16, 'abstract': 17,
                            'academ': 18, 'acceler': 19, 'accentu': 20,
```

In [41]:

```
# create the document-term matrix using stems
dt_matrix_stems_agg = count_vectorizer.fit_transform(agg_minutes["minutes_stems"
print(dt_matrix_stems_agg.shape)
# extract the vocabulary
vocab_agg = count_vectorizer.vocabulary_
```

```
(209, 3408)
```

In [42]:

```
# transform documents into their topic proportions
topic_proportions = lda_model.transform(dt_matrix_stems_agg,
                                        max_iter=20
                                        )
topic_proportions.shape
```

Out[42]: (209, 30)

Finally, we follow similar steps as for LDA to output the estimated distribution of topics for entire meetings

In [43]:

```
# add the topic proportions to each document
for i in range(topic_proportions.shape[1]): agg_minutes['T' + str(i)] = topic_pr
```

In [44]:

```
# explore new data
agg_minutes.head(5)
```

Out[44]:

	year	minutes_stems	T0	T1	T2	T3	T4	
0	199706	section minut summaris analysi present mpc ban...	0.003914	0.026854	0.028800	0.025295	0.015878	0.013
1	199707	section minut summaris analysi present mpc ban...	0.011186	0.005956	0.024556	0.028939	0.026700	0.005
2	199708	section minut summaris analysi present mpc ban...	0.011784	0.026056	0.023620	0.030221	0.022929	0.008
3	199709	meet took place background earlier present ban...	0.005934	0.018535	0.016133	0.027931	0.043716	0.010
4	199710	inut mpc meet octob meet preced present bank s...	0.004237	0.009055	0.020829	0.028945	0.017295	0.005

5 rows × 32 columns

In [45]:
```
# save data
# agg_minutes.to_csv("final_output_agg.csv", index=False)
```

In [46]:
```
# clean date format
agg_minutes["year_clean"] = pd.to_datetime(agg_minutes["year"], format="%Y%m")
agg_minutes
```

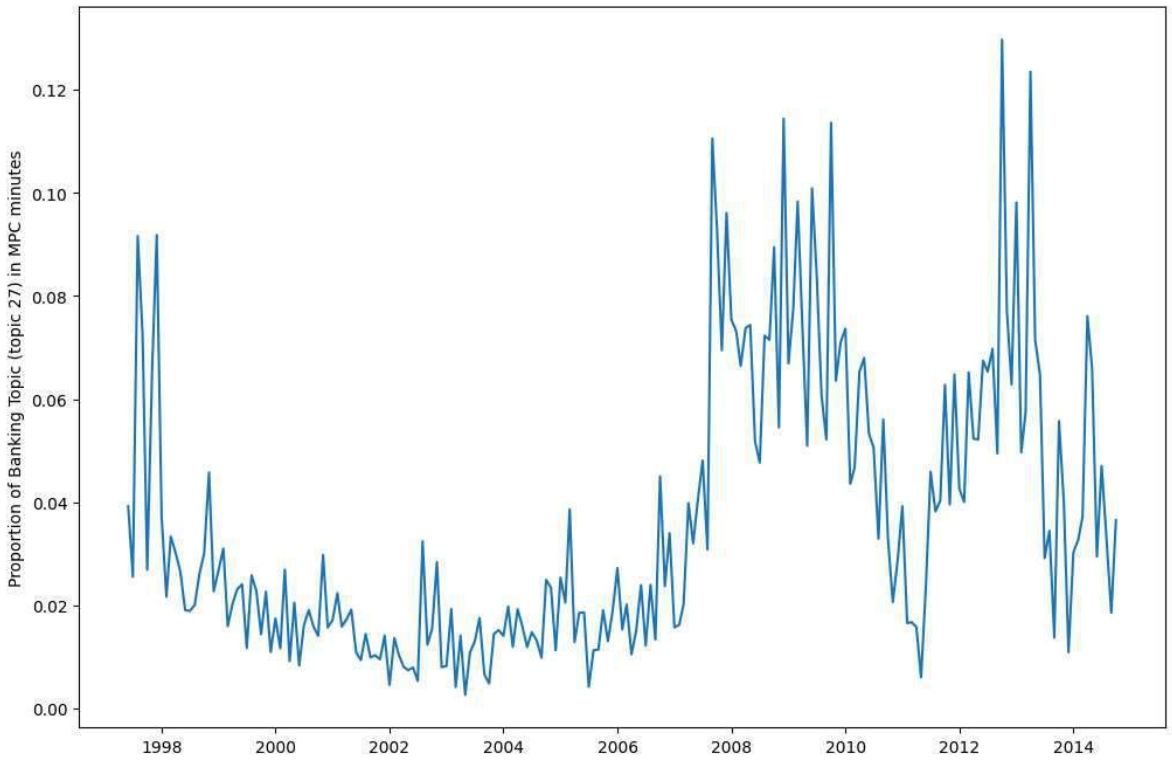
Out[46]:

	year	minutes_stems	T0	T1	T2	T3	T4	
0	199706	section minut summaris analysi present mpc ban...	0.003914	0.026854	0.028800	0.025295	0.015878	0.0
1	199707	section minut summaris analysi present mpc ban...	0.011186	0.005956	0.024556	0.028939	0.026700	0.0
2	199708	section minut summaris analysi present mpc ban...	0.011784	0.026056	0.023620	0.030221	0.022929	0.0
3	199709	meet took place background earlier present ban...	0.005934	0.018535	0.016133	0.027931	0.043716	0.0
4	199710	inut mpc meet octob meet preced present bank s...	0.004237	0.009055	0.020829	0.028945	0.017295	0.0
...	...	...	...	...	...	...	...	
204	201406	befor turn immedi decis discuss financi develo...	0.012647	0.017991	0.030269	0.016010	0.049260	0.0
205	201407	befor turn immedi decis discuss financi develo...	0.022599	0.025695	0.027885	0.018204	0.021642	0.0
206	201408	befor turn immedi decis backdrop latest projec...	0.040388	0.026149	0.027606	0.011464	0.025392	0.0
207	201409	befor turn immedi decis discuss financi develo...	0.035985	0.037457	0.021152	0.018893	0.043713	0.0
208	201410	befor turn immedi decis discuss financi develo...	0.040710	0.036063	0.021142	0.024289	0.029985	0.0

209 rows × 33 columns

In [47]:
```
target_topic = 27
plt.figure(figsize=(12,8))
plt.plot(agg_minutes.year_clean.values, agg_minutes[f"T{target_topic}"].values)
plt.ylabel(f"Proportion of Banking Topic (topic {target_topic}) in MPC minutes")
plt.plot()
```

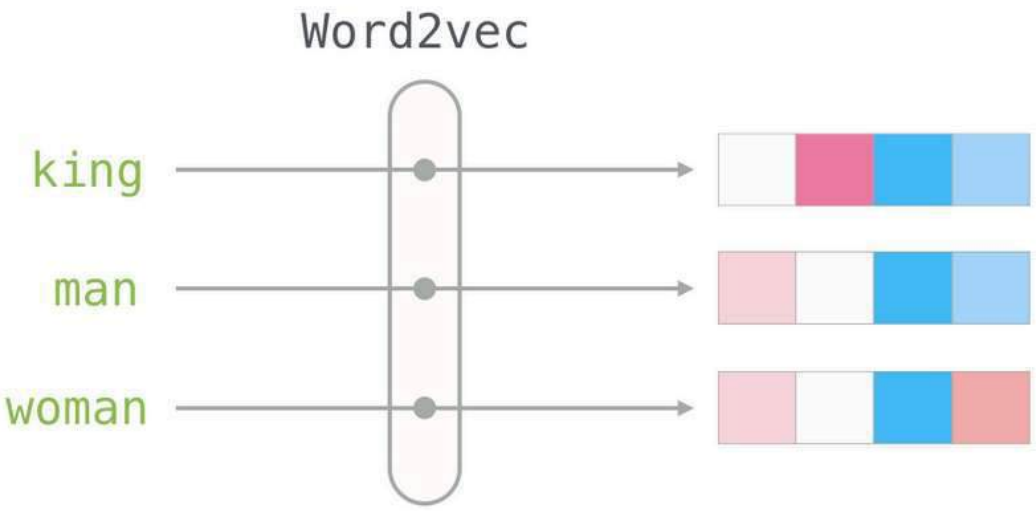
Out[47]: []



04 Introduction to Word2Vec

Open in Colab

This tutorial illustrates several applications of word embeddings by estimating a Word2Vec model using an off-the-shelf Python library (`Gensim`).



Source: Alammar (2019)

Some additional resources on Word2Vec:

- **Book chapter**. Jurafsky & Martin (2021). *Vector Semantics and

Embeddings*. Available online.

- **Original word2vec paper**. Mikolov et al. (2013). *Efficient Estimation of Word Representations in

Vector Space*. Available online.

- **Paper.** Rong (2016). *word2vec Parameter Learning Explained*. Available online.
- **Blog post.** Alammar (2019). *The Illustrated Word2vec*.Available online.
- **Applied paper.** Garg et al. (2018) . *Word embeddings quantify 100 years of gender and ethnic stereotypes*. Available online.
- Gensim documentation

Setup

In [51]:
```
%%capture

# install required libraries
!pip3 install flashtext                    # easy phrase replacing methods
```

```
!pip3 install contractions               # expand English contractions
!pip3 install spacy                      # functions for lemmatizing
!pip3 install gensim==4.0.0              # word2vec estimation
!pip3 install adjustText                 # generate plots with lots of text labe
!pip3 install gdown                      # download files from Google Drive
!pip3 install nltk                       # NLP library

# install Spacy's language model
# for more languages and models check: https://spacy.io/models
!python3 -m spacy download en_core_web_sm
```

In [11]:
```
# clone the GitHub repository with the preprocessing scripts
!git clone https://github.com/unstructured-data-research/text-preprocessing
```

```
fatal: destination path 'pymodules-test' already exists and is not an empty direc
tory.
```

In [ ]:
```
# import libraries
import gdown
import sys
import pandas as pd
import numpy as np
import string
from gensim.models import Word2Vec
import matplotlib.pyplot as plt
from adjustText import adjust_text
from sklearn.decomposition import PCA
from collections import Counter
from sklearn.preprocessing import StandardScaler
import spacy
nlp_standard = spacy.load('en_core_web_sm')

# import our own modules
sys.path.append('./text-preprocessing/src')
import preprocessing_class as pc
import dictionary_methods as dictionary_methods
```

In [12]:
```
# define dictionary with paths to data and Python scripts in Google Drive
urls_dict = {"mpc_minutes":              ("https://drive.google.com/uc?id=1k086m1
             "ons_quarterly_gdp":        ("https://drive.google.com/uc?id=1_8JwCN
             "ir_data_final":            ("https://drive.google.com/uc?id=1-COWuk
            }
```

In [13]:
```
# download all files
for file_name, attributes in urls_dict.items():
    url = attributes[0]
    extension = attributes[1]
    gdown.download(url, f"./{file_name}.{extension}", quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1k086m12gBHKuVMPv4GUD1aB_kZOZpznR
To: /content/mpc_minutes.txt
100%|██████████| 7.36M/7.36M [00:00<00:00, 174MB/s]
Downloading...
From: https://drive.google.com/uc?id=1_8JwCNUsq9wnPiTDmcJ2xrWklGGDFbQD
To: /content/ons_quarterly_gdp.csv
100%|██████████| 3.33k/3.33k [00:00<00:00, 6.97MB/s]
Downloading...
From: https://drive.google.com/uc?id=1-COWuk2ZitrywYnLoJ7HVx2GJumTmgTs
To: /content/ir_data_final.txt
100%|██████████| 9.53M/9.53M [00:00<00:00, 63.0MB/s]
```

Off-the-shelf Word2Vec using Gensim

Gensim is a very powerful library that contains efficient (written in `C`) implementations of several NLP models. Word2Vec is included among these. We use this library to demonstrate multiple use cases for word embeddings.

Load data and preprocess text

Data Source

We will start by loading some real text data over which we will estimate our word embeddings. Concretely, we will work with the Inflation Reports produced by the Bank of England. The data starts on 1998 and ends in 2015. Reports are produced four times a year in the months of February, May, August and November. Here you can see how these reports look like.

In [15]:
```
# load the Inflation Reports (IR) data
data = pd.read_csv("ir_data_final.txt", sep="\t")
data = data[['ir_date', 'paragraph']]
data.columns = ['yearmonth', 'paragraph']
print(data.shape)
data.head(10)
```

```
(15023, 2)
```

Out[15]:

	yearmonth	paragraph
0	199802	It is almost six years since output reached it...
1	199802	Monetary policy is currently being pulled in o...
2	199802	On the other hand, the delayed demand effect o...
3	199802	The scale of the slowdown depends, in part, on...
4	199802	Net trade is weakening, but domestic demand gr...
5	199802	The combination of sharply weakening net trade...
6	199802	The MPC's probability distribution for the fou...
7	199802	The MPC's projection of the twelve-month RPIX ...
8	199802	Overall, the balance of risks to inflation in ...
9	199802	Against the background of this projection, the...

In [16]:
```
# explore one of the paragraphs
data.loc[0, "paragraph"]
```

Out[16]:
```
'It is almost six years since output reached its trough in the last recession.
Since then, output has risen at an average rate of 3% a year and inflation has
fallen from almost 5% to below 3% a year. The combination of above-trend growth
and falling inflation is unsustainable, and has probably already come to an en
d. At this juncture, with output growth likely to fall sharply, monetary policy
is more finely balanced than at any point since the inflation target was introd
uced in 1992. The central issue is whether the existing policy stance will slow
the economy sufficiently quickly to prevent further upward pressure on earnings
growth and retail price inflation.'
```

In [17]:
```
# check how often these reports are produced
grouped = data.groupby("yearmonth", as_index=False).size()
print(grouped.head(5))
print("\n\n")
print(grouped.tail(5))
```

```
   yearmonth  size
0     199802   177
1     199805   161
2     199808   195
3     199811   176
4     199902   191



    yearmonth  size
65     201405   235
66     201408   229
67     201411   220
68     201502   214
69     201505   214
```

Before estimating word embeddings, it is important to pre-process the data. To do this, we will use a custom-built preprocessing Python library. This library implements a wide-range of functions for cleaning text. If you want to inspect the source code the public repository is here.

For a more detailed discussion on preprocessing check the **2_preprocessing.ipynb notebook** within this same repository.

We will start by creating a function that applies all the preprocessing steps in the correct order. We should provide the documents that we want to preprocess along side some key parameters. The function will return an object `prep` that we will use.

In [27]:
```
def apply_preprocessing(data, item_type, stopwords_type, replacing_dict, pattern
    """ Function to apply the steps from the preprocessing class in the correct
        order to generate a term frequency matrix
    """

    # initialize the class with the text data and some parameters
    prep = pc.RawDocs(data)

    # replace some specific phrases of interest
    prep.phrase_replace(replace_dict=replacing_dict,
                        sort_dict=True,
                        case_sensitive_replacing=False)

    # lower-case text and expand contractions
    prep.basic_cleaning(lower_case=True,
                        contraction_split=True)

    # split the documents into tokens
    prep.tokenize_text(tokenization_pattern=pattern)

    # clean tokens (remove non-ascii characters, remove short tokens, remove pun
    prep.token_clean(length=2,
                     punctuation=punctuation,
                     numbers=True)

    # remove stopwords
    if item_type == "tokens":
        prep.stopword_remove(items='tokens', stopwords=stopwords_type)
    elif item_type == "lemmas":
        prep.lemmatize()
        prep.stopword_remove("lemmas", stopwords=stopwords_type)
    elif item_type == "stems":
        prep.stem()
        prep.stopword_remove("stems", stopwords=stopwords_type)

    return prep
```

In [28]:
```
# define dictionary for preprocessing class with terms we want to preserve
replacing_dict = {'monetary policy':'monetary-policy',
                  'interest rate':'interest-rate',
                  'interest rates':'interest-rate',
                  'yield curve':'yield-curve',
                  'repo rate':'repo-rate',
                  'bond yields':'bond-yields',
                  'real estate':'real-estate',
                  'economic growth':'economic-growth'}
```

In [29]:
```
# define tokenization pattern and punctuation symbols
pattern = r'''
          (?x)                # set flag to allow verbose regexps (to separate l
          \w+(?:-\w+)*        # word characters with internal hyphens
          | [][.,;"'?():-_`]  # preserve punctuation as separate tokens
          '''
punctuation = string.punctuation.replace("-", "")
```

In [30]:
```
# use preprocessing class
prep = apply_preprocessing(data.paragraph,
                           "tokens",
                           "long",
                           replacing_dict,
                           pattern,
                           punctuation)
```

In [31]:
```
# inspect a particular tokenized document and compare it to its original form
i = 10
print(data.paragraph[i])
print("\n ------------------------------ \n")
print(prep.tokens[i])
```

```
Broad money continues to grow at double-digit rates (see Chart 1.1). But there ar
e signs that the pace of growth has moderated since the first half of 1997. The s
lowdown in money growth was preceded by a slowdown in lending, particularly to th
e corporate sector. Official interest rates have been unchanged since the rise in
the Bank's repo rate on 6 November to 7.25%. Long-term nominal interest rates hav
e fallen by around 50 basis points in the United Kingdom and by around 40 basis p
oints overseas during the same period. The starting-point for the nominal effecti
ve exchange rate in the inflation projection, based on its average value in the 1
5 working days to 4 February, was 104.9—around 2.8% higher than the starting-poin
t used in the November Report.

 ------------------------------

['broad', 'money', 'continues', 'grow', 'double-digit', 'rates', 'chart', 'sign
s', 'pace', 'growth', 'moderated', 'half', 'slowdown', 'money', 'growth', 'preced
ed', 'slowdown', 'lending', 'particularly', 'corporate', 'sector', 'official', 'i
nterest-rate', 'unchanged', 'rise', 'bank', 'repo-rate', 'november', 'long-term',
'nominal', 'interest-rate', 'fallen', 'around', 'basis', 'points', 'united', 'kin
gdom', 'around', 'basis', 'points', 'overseas', 'period', 'starting-point', 'nomi
nal', 'effective', 'exchange', 'rate', 'inflation', 'projection', 'based', 'avera
ge', 'value', 'working', 'days', 'february', 'around', 'higher', 'starting-poin
t', 'used', 'november', 'report']
```

Model estimation

Now that we have our text preprocessed we can use the Gensim library to efficiently estimate word embeddings using word2vec.

In [32]:
```
# train Gensim's Word2Vec model (takes less than a minute)
gensim_model = Word2Vec(sentences=prep.tokens,      # corpus
                        vector_size=100,            # embedding dimension
                        window=5,                   # words before and after to
                        sg=1,                       # use skip-gram
                        negative=10,                # number of negative example
                        alpha=0.025,                # initial learning rate
```

```
                        min_alpha=0.0001,           # minimum learning rate
                        epochs=5,                   # number of passes through t
                        min_count=1,                # words that appear less tha
                        workers=1,                  # we use 1 to ensure replica
                        seed=92                     # for replicability
                       )
```

In [33]:
```
# extract the word embeddings from the model
word_vectors = gensim_model.wv
word_vectors.vectors.shape  # vocab_size x embeddings dimension
```

Out[33]: (8886, 100)

这段代码的目的是训练一个Gensim的Word2Vec模型，并提取词嵌入。首先，它使用预处理后的标记作为语料库，设置词嵌入的维度为100，并使用skip-gram模型进行训练。模型考虑目标词前后5个词，每个正样本使用10个负样本进行训练，初始学习率为0.025，最小学习率为0.0001，训练数据遍历5次，忽略出现次数少于1次的词。为了确保结果的可重复性，使用1个线程进行训练，并设置随机种子为92。训练完成后，从模型中提取词嵌入，得到一个形状为(8886, 100)的矩阵，表示词汇表中有8886个词，每个词的嵌入维度为100。这些词嵌入可以用于各种自然语言处理任务，如文本分类、聚类和相似度计算等。

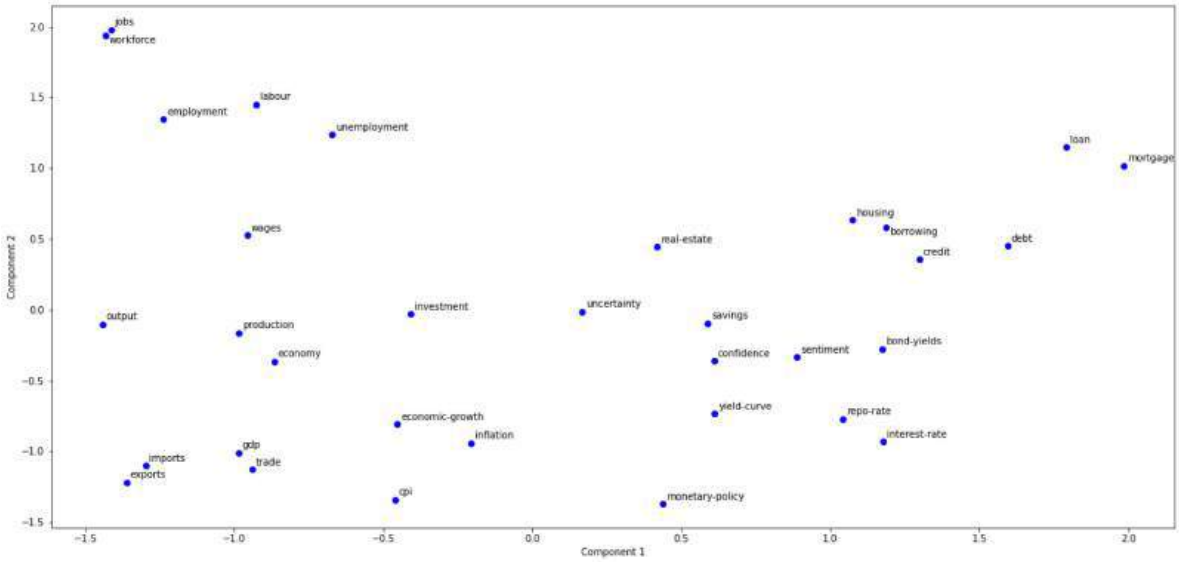
There a lot of different ways in which we can use these estimated word embeddings. We will start by showing a simple way to visualize them in 2-dimensions.

Visualization

In [34]:
```
# use a PCA decomposition to visualize the embeddings in 2D
def pca_scatterplot(model, words):
    pca = PCA(n_components=2, random_state=92)
    word_vectors = np.array([model[w] for w in words])
    low_dim_emb = pca.fit_transform(word_vectors)
    plt.figure(figsize=(21,10))
    plt.scatter(low_dim_emb[:,0], low_dim_emb[:,1], edgecolors='blue', c='blue')
    plt.xlabel("Component 1")
    plt.ylabel("Component 2")

    # get the text of the plotted words
    texts = []
    for word, (x,y) in zip(words, low_dim_emb):
        texts.append(plt.text(x+0.01, y+0.01, word, rotation=0))

    # adjust the position of the labels so that they dont overlap
    adjust_text(texts)
    # show plot
    plt.show()
```

这段代码使用主成分分析（PCA）将词嵌入降维到二维，并绘制散点图以可视化这些词在二维空间中的分布。具体来说，它定义了一个函数pca_scatterplot，通过PCA将词向量降维到二维，并绘制包含词标签的散点图。函数参数包括词嵌入模型和要可视化的词列表。然后，代码定义了一组感兴趣的词，并将其与替换字典中的词合并。最后，调用pca_scatterplot函数，使用这些词的词嵌入生成散点图。这有助于理解词嵌入的结构和词之间的关系。

In [35]:
```
# define the tokens to use in the plot
tokens_of_interest = ['economy', 'gdp', 'production', 'output',
                      'investment', 'confidence', 'sentiment',
                      'uncertainty', 'inflation', 'cpi',
                      'loan', 'mortgage', 'credit', 'debt', 'savings',
                      'borrowing', 'housing', 'labour', 'workforce',
                      'unemployment', 'employment', 'jobs', 'wages',
                      'trade', 'exports', 'imports']

# expand the list of tokens with all the tokens from the replacement dictionary
tokens_of_interest = set(tokens_of_interest + list(replacing_dict.values()) )

# plot
pca_scatterplot(word_vectors, list(tokens_of_interest))
```



We can clearly observe how words form some thematically cohesive groups; trade (e.g. exports, imports, trade, output), job-market (e.g. workforce, jobs, employment), housing (e.g. real-state, housing, borrowing, mortgage).

Nearest neighbors analysis

We can further explore how words cluster in the embedded space by analyzing the nearest neighbours of some selected words.

In [36]:
```
# find the K nearest neighbours of relevant words
K = 5
words = ["uncertainty", "risk", "stable",
         "contraction", "expansion",
         "monetary-policy", "interest-rate", "inflation"]

for word in words:
    print(f"Nearest neighbors of: {word}")
    print(word_vectors.most_similar(word, topn=K))
    print("\n")
```

```
Nearest neighbors of: uncertainty
[('surrounding', 0.6552051901817322), ('surrounds', 0.6302632689476013), ('uncert
ainties', 0.6213497519493103), ('tension', 0.6160809397697449), ('uncertain', 0.6
077167391777039)]


Nearest neighbors of: risk
[('risks', 0.7024600505828857), ('possibility', 0.6914620995521545), ('posing',
0.6614995002746582), ('possibilities', 0.6543980240821838), ('pose', 0.6543081998
825073)]


Nearest neighbors of: stable
[('remarkably', 0.728036642074585), ('speaking', 0.7205280065536499), ('flat', 0.
7081785202026367), ('fairly', 0.7034920454025269), ('stabilised', 0.6907343864440
918)]


Nearest neighbors of: contraction
[('bounceback', 0.7915180921554565), ('contracted', 0.761755645275116), ('destock
ing', 0.7608803510665894), ('run-down', 0.7458294034004211), ('surge', 0.74246180
05752563)]


Nearest neighbors of: expansion
[('vigorous', 0.7654820680618286), ('upswing', 0.7611563205718994), ('brisk', 0.7
60983407497406), ('briskly', 0.7505147457122803), ('upturn', 0.7435798048973083)]


Nearest neighbors of: monetary-policy
[('policy', 0.7636595368385315), ('stance', 0.7158397436141968), ('accommodativ
e', 0.6671982407569885), ('stimulatory', 0.6670347452163696), ('monetary', 0.6457
614302635193)]


Nearest neighbors of: interest-rate
[('market-implied', 0.6929869651794434), ('yield-curve', 0.6887417435646057), ('r
epo-rate', 0.6666507124900818), ('bound', 0.6634966731071472), ('convenient', 0.6
606504321098328)]


Nearest neighbors of: inflation
[('rpix', 0.7222162485122681), ('cpi', 0.716386079788208), ('below-target', 0.700
162410736084), ('target', 0.6970875263214111), ('well-anchored', 0.69004499912261
96)]
```

Analogy tasks 类比任务

A very interesting, and surprising, use of word embeddings is to find word analogies. The famous example used by Mikolov et al. (2013) searches for a word X in the embedded space that is similar to "woman" in the same sense that "king" is similar to "man". This task can be expressed in terms of a simple vector arithmetic problem as follows:

$$\vec{King} - \vec{Man} = \vec{X} - \vec{Woman}$$
$$\vec{King} - \vec{Man} + \vec{Woman} = \vec{X}$$

Mikolov et al. (2013) find that when performing this operation on their trained embeddings, they are able to recover the word "queen".

$$\vec{King} - \vec{Man} + \vec{Woman} \approx \vec{Queen}$$

Using `Gensim` this operation can be very easily perfomed by simply using the `.most_similar()` function as follows:

```
word_vectors.most_similar(positive=['king', 'woman'], negative=['man'])
```

We will play with this idea and try to extend it to our own domain. Some of the analogies that we will try to solve are:

$$\vec{Contraction} - \vec{Expansion} + \vec{Downward} = \vec{X}$$
$$\vec{Inflation} - \vec{CPI} + \vec{GDP} = \vec{X}$$

In [37]:
```
# create the analogy tasks for our data
positive_words = [['contraction', 'downward'],
                  ['inflation', 'gdp'],
                  ['company', 'wages'],
                  ['expansion', 'tighten']]

negative_words = [['expansion'],
                  ['cpi'],
                  ['profits'],
                  ['contraction']]

for pw, nw in zip(positive_words, negative_words):
    #print(f"Analogy task for positive words: {pw} and negative words {nw}")
    print(f"'{nw[0]}' is to '{pw[0]}' as '{pw[1]}' is to:")
    print(word_vectors.most_similar(positive=pw, negative=nw))
    print("\n")
```

这段代码的目的是创建类比任务，并使用词嵌入模型来解决这些类比问题。具体来说，它使用Word2Vec模型的most_similar方法，通过给定的正词和负词来找到最相似的词。类比任务的形式是“A之于B，如同C之于D”，其中D是要找到的词。

```
Analogy task for positive words: ['contraction', 'downward'] and negative words
['expansion']
[('upward', 0.68913733959198), ('faster-than-expected', 0.6073570847511292), ('un
affected', 0.5849462151527405), ('upwards', 0.5783886909484863), ('pulling', 0.57
82693028450012), ('weaker-than-expected', 0.5646066069602966), ('responsible', 0.
5617024302482605), ('occurred', 0.5597019791603088), ('hump', 0.558463573455810
5), ('lower-than-expected', 0.558194100856781)]


Analogy task for positive words: ['expansion', 'tighten'] and negative words ['co
ntraction']
[('improve', 0.638952910900116), ('loosen', 0.6204016804695129), ('expand', 0.603
9195656776428), ('thawing', 0.5945644378662109), ('ease', 0.5771167278289795),
('expansionary', 0.5693738460540771), ('deteriorate', 0.5681977868080139), ('part
icipate', 0.5636993050575256), ('loosening', 0.5568081736564636), ('act', 0.55184
73982810974)]


Analogy task for positive words: ['inflation', 'gdp'] and negative words ['cpi']
[('fourth-quarter', 0.5542969107627869), ('moderately', 0.550415575504303), ('eco
nomic-growth', 0.5491822361946106), ('sluggishly', 0.5481258034706116), ('below-t
rend', 0.5460458993911743), ('unrevised', 0.540518045425415), ('displayed', 0.539
558470249176), ('turns', 0.5389357209205627), ('constant-rate', 0.537750005722045
9), ('intact', 0.5371445417404175)]


Analogy task for positive words: ['company', 'wages'] and negative words ['profit
s']
[('demands', 0.5929698348045349), ('wage', 0.5879050493240356), ('bargaining', 0.
5740780830383301), ('erosion', 0.5699042081832886), ('recoup', 0.564629614353179
9), ('employers', 0.5591343641281128), ('resistance', 0.5517959594726562), ('nego
tiations', 0.5486984848976135), ('exerts', 0.547100305557251), ('inertia', 0.5458
434820175171)]
```

Building dictionaries

One last use of word embeddings is to expand existing dictionaries by finding the nearest neighbours to a set of "center" terms. To illustrate this, we will come show how to generate dictionaries of positive and negative terms to analyze text data from the Bank of England Monetary Police Comittee minutes.

In [38]:
```python
# create a positive dictionary by finding the nearest neighbors to a combination
N = 20
pos_center_terms = ['expansion', 'stable']
pos_nn = [w for w, _ in word_vectors.most_similar(positive=pos_center_terms, top
pos_word2vec = pos_center_terms + pos_nn
print(pos_word2vec)
```

```
['expansion', 'stable', 'steady', 'remarkably', 'robust', 'brisk', 'apace', 'resi
lient', 'vigorous', 'stagnation', 'sluggishness', 'healthily', 'decelerating', 'd
ouble-digit', 'soften', 'slackened', 'buoyed', 'loose', 'unevenly', 'non-japan',
'accelerating', 'tandem']
```

In [39]:
```python
# create a negative dictionary by finding the nearest neighbors to a combination
N = 20
neg_center_terms = ['contraction', 'uncertainty']
```

```
neg_nn = [w for w, _ in word_vectors.most_similar(positive=neg_center_terms, top
neg_word2vec = neg_center_terms + neg_nn
print(neg_word2vec)
```

```
['contraction', 'uncertainty', 'sluggishness', 'destocking', 'bounceback', 'shall
ower', 'gentler', 'cushioned', 'faster-than-expected', 'worsening', 'softness',
'stronger-than-expected', 'bright', 'bounce-back', 'pause', 'solidity', 'ambiguou
s', 'slacken', 'nascent', 'spreading', 'wake', 'hot']
```

In [40]:
```
# load data for dictionary method example (might take a couple of minutes if we
data_dict, prep_dict = dictionary_methods.dict_example("mpc_minutes.txt", "token
```

In [41]:
```
# generate the count of positive and negative tokens in the corpus with our new
pos_counts_word2vec, neg_counts_word2vec = dictionary_methods.pos_neg_counts(pre
                                                                              ite
                                                                              pos
                                                                              neg
                                                                              nlp
```

In [42]:
```
# add counts to the data
data_dict['pos_counts_word2vec'] = pos_counts_word2vec
data_dict['neg_counts_word2vec'] = neg_counts_word2vec
data_dict
```

load data for dictionary method example (might take a couple of minutes if we use lemmas or stems)
data_dict, prep_dict = dictionary_methods.dict_example("mpc_minutes.txt", "tokens", replacing_dict) # dataframe, preprocessing object
generate the count of positive and negative tokens in the corpus with our new dictionaries
pos_counts_word2vec, neg_counts_word2vec = dictionary_methods.pos_neg_counts(prep_dict,
items="tokens", pos_terms=pos_word2vec,
neg_terms=neg_word2vec, nlp_standard=nlp_standard)

Out[42]:

	date	minutes	year	quarter	pos_counts_word2vec	neg_counts_wor
0	199706	1 Sections I to V of this minute summarise t...	1997	2	0.0	
1	199706	The 12-month growth rate of notes and coins ...	1997	2	0.0	
2	199706	Broad money, too, decelerated in April: its ...	1997	2	0.0	
3	199706	Lending growth continued at around 9% in ...	1997	2	0.0	
4	199706	Lending to individuals remained robust in...	1997	2	1.0	
...	...	...	...	...	...	
7272	201410	For most members, there remained insuffi...	2014	4	1.0	
7273	201410	Set against this, the level of Bank Rate...	2014	4	1.0	
7274	201410	For two members, economic circumstances ...	2014	4	1.0	
7275	201410	The Governor invited the Committee to vo...	2014	4	0.0	
7276	201410	The following members of the Committee w...	2014	4	0.0	

7277 rows × 6 columns

In [43]:
```
# Apel and Blix-Grimaldi (2012) dictionaries
pos_words_AB = ["high", "higher", "highest",
                "strong", "stronger", "strongest",
                "increase", "increases", "increased","increasing",
                "fast", "faster", "fastest"]

neg_words_AB = ["low", "lower", "lowest",
                "weak", "weaker", "weakest",
                "decrease", "decreases", "decreased", "decreasing",
                "slow", "slower", "slowest"]

# generate the count of positive and negative lemmas in the corpus with Apel and
pos_counts_AB, neg_counts_AB = dictionary_methods.pos_neg_counts(prep_dict,
                                                 items="tokens",
                                                 pos_terms=pos_words_AB,
                                                 neg_terms=neg_words_AB,
                                                 nlp_standard=nlp_standard)

# add counts to the data
data_dict['pos_counts_AB'] = pos_counts_AB
data_dict['neg_counts_AB'] = neg_counts_AB
data_dict.head()
```

Out[43]:

	date	minutes	year	quarter	pos_counts_word2vec	neg_counts_word2vec
0	199706	1 Sections I to V of this minute summarise t...	1997	2	0.0	0.0
1	199706	The 12-month growth rate of notes and coins ...	1997	2	0.0	0.0
2	199706	Broad money, too, decelerated in April: its ...	1997	2	0.0	0.0
3	199706	Lending growth continued at around 9% in ...	1997	2	0.0	0.0
4	199706	Lending to individuals remained robust in...	1997	2	1.0	0.0

In [44]:
```
# aggregate to year-month level
data_agg = data_dict.groupby(['date']).agg({'pos_counts_word2vec': 'sum', 'neg_c
                                            'pos_counts_AB': 'sum', 'neg_counts_
```

```
                                                'year': 'mean', 'quarter':'mean'})
data_agg.head()
```

Out[44]:

date	pos_counts_word2vec	neg_counts_word2vec	pos_counts_AB	neg_counts_A
199706	4.0	2.0	27.0	10
199707	2.0	0.0	80.0	25
199708	4.0	7.0	73.0	19
199709	6.0	6.0	67.0	26
199710	5.0	7.0	89.0	34

In [45]:

```
# aggregate to year-quarter level removing incomplete quarters
data_agg['months_x_quarter'] = 1
data_agg = data_agg.groupby(['year', 'quarter']).sum()[['pos_counts_word2vec', '
                                                        'pos_counts_AB', 'neg_co
                                                        'months_x_quarter']]

data_agg = data_agg[data_agg['months_x_quarter']==3]
del data_agg['months_x_quarter']

data_agg.head()
```

Out[45]:

year	quarter	pos_counts_word2vec	neg_counts_word2vec	pos_counts_AB	neg_
1997.0	3.0	12.0	13.0	220.0	
	4.0	15.0	22.0	228.0	
1998.0	1.0	23.0	27.0	251.0	
	2.0	30.0	22.0	251.0	
	3.0	21.0	17.0	264.0	

In [46]:

```
# compute sentiment at year-quarter level
data_agg['sentiment_word2vec'] = (data_agg.pos_counts_word2vec - data_agg.neg_co
data_agg['sentiment_AB'] = (data_agg.pos_counts_AB - data_agg.neg_counts_AB)/(da
data_agg.head()
```

Out[46]:

year	quarter	pos_counts_word2vec	neg_counts_word2vec	pos_counts_AB	neg_
1997.0	3.0	12.0	13.0	220.0	
	4.0	15.0	22.0	228.0	
1998.0	1.0	23.0	27.0	251.0	
	2.0	30.0	22.0	251.0	
	3.0	21.0	17.0	264.0	

Next we add quarterly GDP data collected from the ONS website.

In [47]:
```
# prepare GDP data
ons = pd.read_csv('ons_quarterly_gdp.csv', names=['label', 'gdp_growth', 'quarte
ons['year'] = ons.label.apply(lambda x: x[:4]).astype(int)
ons['quarter'] = ons.label.apply(lambda x: x[6]).astype(int)
ons = ons[['year', 'quarter', 'gdp_growth']]
ons = ons.drop_duplicates().reset_index(drop=True).copy()
ons.head()
```

Out[47]:

	year	quarter	gdp_growth
0	1997	2	1.2
1	1997	3	0.6
2	1997	4	1.3
3	1998	1	0.6
4	1998	2	0.6

In [48]:
```
# merge to sentiment data
df = data_agg.merge(ons, how='left', on=['year', 'quarter']).copy()

# create year-quarter variable
df["year_quarter"] = df.apply(lambda x: f"{int(x['quarter'])}Q{int(x['year'])}",
df["year_quarter"] = df["year_quarter"].apply(lambda x: pd.Period(value=x, freq=
df.head()
```

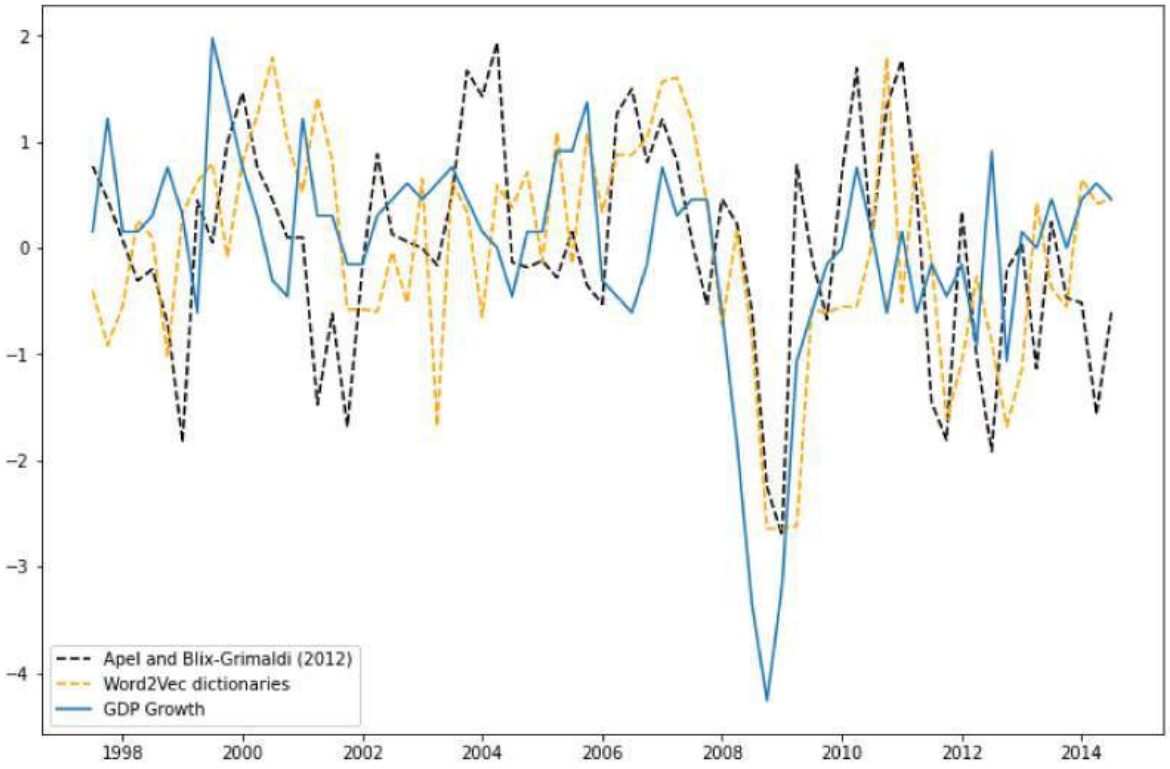
Out[48]:

	year	quarter	pos_counts_word2vec	neg_counts_word2vec	pos_counts_AB	n
0	1997.0	3.0	12.0	13.0	220.0	
1	1997.0	4.0	15.0	22.0	228.0	
2	1998.0	1.0	23.0	27.0	251.0	
3	1998.0	2.0	30.0	22.0	251.0	
4	1998.0	3.0	21.0	17.0	264.0	

In [49]:
```
print(df[['gdp_growth', 'sentiment_AB', 'sentiment_word2vec']].corr())
```

	gdp_growth	sentiment_AB	sentiment_word2vec
gdp_growth	1.000000	0.296561	0.431459
sentiment_AB	0.296561	1.000000	0.332324
sentiment_word2vec	0.431459	0.332324	1.000000

In [50]:
```
# standardize and plot all series
scaler = StandardScaler()

fig, ax = plt.subplots(figsize=(12,8))
ax.plot(df["year_quarter"], scaler.fit_transform(df.sentiment_AB.values.reshape(
                                                 label="Apel and Blix-Grimaldi (

ax.plot(df["year_quarter"], scaler.fit_transform(df.sentiment_word2vec.values.re
                                                 label="Word2Vec dictionaries",

ax.plot(df["year_quarter"], scaler.fit_transform(df.gdp_growth.values.reshape(-1

plt.legend()
plt.show()
```

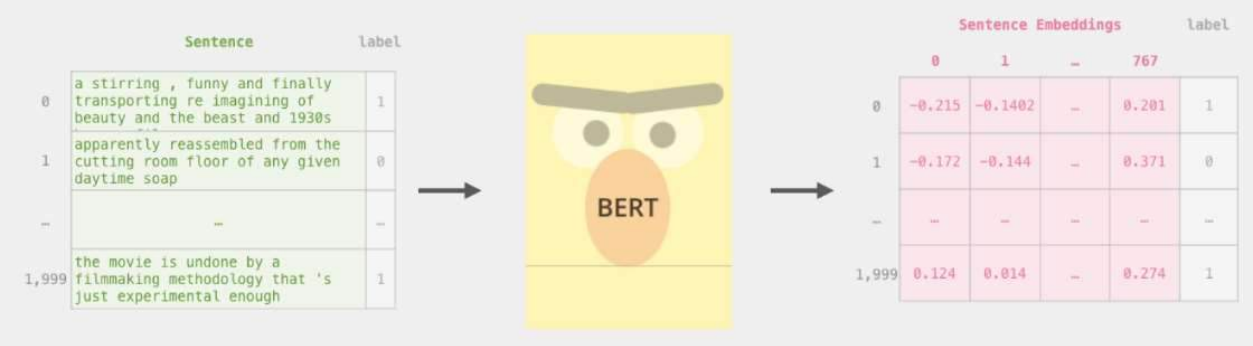


05 Language models for feature generation

本笔记本将展示如何使用现代语言模型从文本数据中创建数字特征并执行分类任务。尽管本笔记本将演示一种特定的语言模型（即 BERT），但相同的管道和代码可用于其他多种语言模型。

Open in Colab (https://colab.research.google.com/github/sekhansen/text_algorithms_econ/blob/main/notebooks/5_llm_features.ipynb)

This notebook will show how to use a modern language model to create numeric features from text data and perform a classification task. Although the notebook will demonstrate one particular language model (i.e. BERT) the same pipeline and code can be used with multiple other language models.



Some useful external resources:

- **Original BERT paper.** Devlin et al. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Available online here (https://arxiv.org/pdf/1810.04805.pdf)
- **Book chapter.** Jurafsky & Martin (2021). *Transfer Learning with Pre-trained Language Models and Contextual Embeddings*. Available online here. (https://web.stanford.edu/~jurafsky/slp3/11.pdf)
- **Book chapter.** Jurafsky & Martin (2021). *Deep Learning Architectures for Sequence Processing*. Available online. (https://web.stanford.edu/~jurafsky/slp3/9.pdf)
- **Blog post.** Alammar (2018). Available online here (https://jalammar.github.io/illustrated-bert/)
- **Blog post.** Alammar (2019). Available online here (https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/)
- **Applied tutorial**. AssemblyAI (2022). Available online here. (https://www.youtube.com/watch?v=QEaBAZQCtwE)

0. Setup

In [ ]:
```
# Instructions for Colab:
# 1. First CHANGE RUNTIME TYPE to GPU
# 2. Run install commands
# 3. You might need to RESTART RUNTIME
# 4. Run the rest of the cells below
```

In [2]:
```
%%capture

# install required libraries
!pip3 install transformers                # HuggingFace library for interacting with BERT (and multiple other models)
!pip3 install datasets                    # HuggingFace library to process dataframes
!pip3 install sentence-transformers       # library to use Sentence Similarity BERT
!pip3 install bertviz                     # visualize BERT's attention weigths
!pip3 install annoy                       # Spotify's library for finding nearest neighbours
!pip3 install ipywidgets
```

In [ ]:
```
# (COLAB) you might need to restart RUNTIME after installing packages!
```

In [ ]:
```
# import libraries
import gdown
import pandas as pd
import numpy as np
import gdown
import random
from tqdm.auto import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
import torch

from transformers import AutoModel, BertModel, AutoTokenizer, BertForSequenceClassification, pipeline, TrainingArguments, Train
from transformers.pipelines.base import KeyDataset
from datasets import load_dataset, load_metric, Dataset, DatasetDict

from gensim.models import Word2Vec
import gensim.downloader as api

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import f1_score, precision_score, recall_score, accuracy_score

from google.colab import output
output.enable_custom_widget_manager()

# test GPU
print(f"GPU: {torch.cuda.is_available()}")
```

```
GPU: True
```

In [ ]:
```
# define dictionary with paths to data in Google Drive
urls_dict = {"10k_sent_2019":        ("https://drive.google.com/uc?id=17PQbZ6EotMxyhpt2Laqqh9z-EKUbwDhX", "parquet"),
             "covariates_2019":      ("https://drive.google.com/uc?id=1ELRq69FOiFvNpSvXOijGeGKZB5DiNXt4", "csv"),
            }
```

In [ ]:
```
# download all files
for file_name, attributes in urls_dict.items():
    url = attributes[0]
    extension = attributes[1]
    gdown.download(url, f"./{file_name}.{extension}", quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=17PQbZ6EotMxyhpt2Laqqh9z-EKUbwDhX (https://drive.google.com/uc?id=17PQbZ6EotMxyhpt2Laqqh9
z-EKUbwDhX)
To: /content/10k_sent_2019.parquet
100%|██████████| 162M/162M [00:04<00:00, 39.6MB/s]
Downloading...
From: https://drive.google.com/uc?id=1ELRq69FOiFvNpSvXOijGeGKZB5DiNXt4 (https://drive.google.com/uc?id=1ELRq69FOiFvNpSvXOijGeG
KZB5DiNXt4)
To: /content/covariates_2019.csv
100%|██████████| 428k/428k [00:00<00:00, 126MB/s]
```

1. Load and prepare the data

This tutorial uses text data from the **10-K reports** filed by publicly-traded firms in the U.S. in 2019. 10-K reports are a very rich source of data since firms include information regarding their organizational structure, financial performance and risk factors. We will use a version of the data where the risk factors section of each report has been splitted into sentences and each sentence has been assigned an ID that combines the firm identifier (i.e. **CIK**) and a sentence number. The raw data we use has a total of 1,744,131 sentences for 4,033 firms.

More on the 10-K reports here (https://www.investor.gov/introduction-investing/getting-started/researching-investments/how-read-10-k).

In [ ]:
```
# read data
df = pd.read_parquet("10k_sent_2019.parquet")
df
```

Out[5]:

	sentences	cik	year	sent_no	sent_id
0	You should carefully review the following disc...	1065088	2019	0	1065088_0
1	Risk Factors That May Affect our Business, Res...	1065088	2019	1	1065088_1
2	Our operating and financial results are subjec...	1065088	2019	2	1065088_2
3	Our operating and financial results have varie...	1065088	2019	3	1065088_3
4	It is difficult for us to forecast the level o...	1065088	2019	4	1065088_4
...	...	...	...	...	...
1744126		1069533	2019	189	1069533_189
1744127	The Company is generally isolated from commodi...	1069533	2019	190	1069533_190
1744128		1069533	2019	191	1069533_191
1744129	With respect to interest rate risk, the Compan...	1069533	2019	192	1069533_192
1744130	However, increases in interest rates could adv...	1069533	2019	193	1069533_193

1744131 rows × 5 columns

In [ ]:
```
# firm-level additional data
covariates = pd.read_csv("covariates_2019.csv")
covariates.head()
```

Out[6]:

	gvkey	datadate	fyear	indfmt	consol	popsrc	datafmt	tic	conm	curcd	act	at	emp	cik	costat	na
0	1004	20200531	2019	INDL	C	D	STD	AIR	AAR CORP	USD	1438.700	2079.000	5.400	1750	A	423
1	1050	20191231	2019	INDL	C	D	STD	CECE	CECO ENVIRONMENTAL CORP	USD	179.498	408.637	0.830	3197	A	333
2	1078	20191231	2019	INDL	C	D	STD	ABT	ABBOTT LABORATORIES	USD	15667.000	67887.000	107.000	1800	A	334
3	1104	20191231	2019	INDL	C	D	STD	ACU	ACME UNITED CORP	USD	73.146	110.749	0.441	2098	A	332
4	1117	20191231	2019	INDL	C	D	STD	BKTI	BK TECHNOLOGIES CORP	USD	23.886	37.940	0.111	2186	A	334

从数据集中选择一些特定公司的数据，并对这些公司的数据进行聚合。

In [ ]:
```
# we will choose some specific firms of interest using their TIC identifiers
tics = ["AAPL", "GOOGL", "TWTR", "ORCL", "TSLA", "GM", "F", "BAC",
        "COF", "JPM", "AXP", "HBC2", "TGT", "M", "WMT", "COST",
        "BNED", "DIS", "FOXA", "ADSK", "CAT", "BA"]

# subset covariates
covariates_focus = covariates.loc[covariates["tic"].isin(tics)]
covariates_focus = covariates_focus.groupby("tic", as_index=False).max()
covariates_focus
```

Out[7]:

	tic	gvkey	datadate	fyear	indfmt	consol	popsrc	datafmt	conm	curcd	act	at	emp	cik	cos
0	AAPL	1690	20190930	2019	INDL	C	D	STD	APPLE INC	USD	162819.000	338516.000	137.000	320193	
1	ADSK	1878	20200131	2019	INDL	C	D	STD	AUTODESK INC	USD	2659.300	6179.300	10.100	769397	
2	AXP	1447	20191231	2019	INDL	C	D	STD	AMERICAN EXPRESS CO	USD	NaN	198321.000	64.500	4962	
3	BA	2285	20191231	2019	INDL	C	D	STD	BOEING CO	USD	102229.000	133625.000	161.100	12927	
4	BAC	7647	20191231	2019	INDL	C	D	STD	BANK OF AMERICA CORP	USD	NaN	2434079.000	208.131	70858	
5	BNED	23208	20200430	2019	INDL	C	D	STD	BARNES & NOBLE EDUCATION INC	USD	584.919	1156.432	5.500	1634117	
6	CAT	2817	20191231	2019	INDL	C	D	STD	CATERPILLAR INC	USD	39193.000	78453.000	102.300	18230	
7	COF	30990	20191231	2019	INDL	C	D	STD	CAPITAL ONE FINANCIAL CORP	USD	NaN	390365.000	51.900	927628	
8	COST	29028	20190831	2019	INDL	C	D	STD	COSTCO WHOLESALE CORP	USD	23485.000	45400.000	254.000	909832	
9	DIS	3980	20190930	2019	INDL	C	D	STD	DISNEY (WALT) CO	USD	28124.000	193984.000	223.000	1744489	
10	F	4839	20191231	2019	INDL	C	D	STD	FORD MOTOR CO	USD	114047.000	258537.000	190.000	37996	
11	FOXA	34636	20190630	2019	INDL	C	D	STD	FOX CORP	USD	6478.000	19509.000	7.700	1754301	
12	GM	5073	20191231	2019	INDL	C	D	STD	GENERAL MOTORS CO	USD	74992.000	228037.000	164.000	1467858	
13	GOOGL	160329	20191231	2019	INDL	C	D	STD	ALPHABET INC	USD	152578.000	275909.000	118.899	1652044	
14	HBC2	9061	20191231	2019	INDL	C	D	STD	HSBC USA INC	USD	NaN	175375.000	4.828	83246	
15	JPM	2968	20191231	2019	INDL	C	D	STD	JPMORGAN CHASE & CO	USD	NaN	2687379.000	256.981	19617	
16	M	4611	20200131	2019	INDL	C	D	STD	MACY'S INC	USD	6810.000	21172.000	123.000	794367	
17	ORCL	12142	20200531	2019	INDL	C	D	STD	ORACLE CORP	USD	52140.000	115438.000	135.000	1341439	
18	TGT	3813	20200131	2019	INDL	C	D	STD	TARGET CORP	USD	12902.000	42779.000	368.000	27419	
19	TSLA	184996	20191231	2019	INDL	C	D	STD	TESLA INC	USD	12103.000	34309.000	48.016	1318605	
20	TWTR	18872	20191231	2019	INDL	C	D	STD	TWITTER INC	USD	7620.075	12703.389	4.900	1418091	
21	WMT	11259	20200131	2019	INDL	C	D	STD	WALMART INC	USD	61806.000	236495.000	2200.000	104169	

In [ ]:
```
# look at their 2 digit NAICS sector code
covariates_focus.groupby("naics2_name").size()
```

Out[8]:
```
naics2_name
Finance and Insurance    5
Information              6
Manufacturing            6
Retail Trade             5
dtype: int64
```

In [ ]:
```
# generate a dictionary mapping from CIK to name
cik2name = {row["cik"]: row["conm"] for i, row in covariates_focus.iterrows()}

# select the 10K reports from the choosen firms only
df = df.loc[df["cik"].isin(covariates_focus["cik"])]
df.reset_index(drop=True, inplace=True)
df
```

Out[9]:

	sentences	cik	year	sent_no	sent_id
0	The following discussion sets forth the materi...	19617	2019	0	19617_0
1	Readers should not consider any descriptions o...	19617	2019	1	19617_1
2	Any of the risk factors discussed below could ...	19617	2019	2	19617_2
3	Regulatory	19617	2019	3	19617_3
4	JPMorgan Chase's businesses are highly regulat...	19617	2019	4	19617_4
...	...	...	...	...	...
7279	With the completion of the TFCF acquisition, o...	1744489	2019	194	1744489_194
7280	The increased indebtedness could have the effe...	1744489	2019	195	1744489_195
7281	The increased levels of indebtedness could als...	1744489	2019	196	1744489_196
7282	repurchases and dividends, and other activitie...	1744489	2019	197	1744489_197
7283	Our financial flexibility may be further const...	1744489	2019	198	1744489_198

7284 rows × 5 columns

In [ ]:
```
# lets explore a random sentence from our corpus
i = np.random.randint(0, len(df))
print(f"Sentence from: {cik2name[df.loc[i, 'cik']]}\n")
print(df.loc[i, "sentences"])
```

```
Sentence from: APPLE INC

The Company is subject to various legal proceedings and claims that have arisen in the ordinary course of business and have no
t yet been fully resolved, and new claims may arise in the future.
```

In [ ]:
```
# merge each sentence with the NAICS2 code and name from its corresponding firm
df = pd.merge(df, covariates_focus[["cik", "naics2", "naics2_name"]], how="left", on="cik")
df
```

Out[11]:

	sentences	cik	year	sent_no	sent_id	naics2	naics2_name
0	The following discussion sets forth the materi...	19617	2019	0	19617_0	52	Finance and Insurance
1	Readers should not consider any descriptions o...	19617	2019	1	19617_1	52	Finance and Insurance
2	Any of the risk factors discussed below could ...	19617	2019	2	19617_2	52	Finance and Insurance
3	Regulatory	19617	2019	3	19617_3	52	Finance and Insurance
4	JPMorgan Chase's businesses are highly regulat...	19617	2019	4	19617_4	52	Finance and Insurance
...	...	...	...	...	...	...	...
7279	With the completion of the TFCF acquisition, o...	1744489	2019	194	1744489_194	51	Information
7280	The increased indebtedness could have the effe...	1744489	2019	195	1744489_195	51	Information
7281	The increased levels of indebtedness could als...	1744489	2019	196	1744489_196	51	Information
7282	repurchases and dividends, and other activitie...	1744489	2019	197	1744489_197	51	Information
7283	Our financial flexibility may be further const...	1744489	2019	198	1744489_198	51	Information

7284 rows × 7 columns

In [ ]:
```
# drop empty sentences or sentences with very few words
min_words = 3
df["sentence_len"] = df["sentences"].apply(lambda x: len(x.split()))
df["keep_sent"] = df["sentence_len"].apply(lambda x: x > min_words)
df = df.loc[df["keep_sent"]]
df.reset_index(drop=True, inplace=True)
df
```

Out[12]:

	sentences	cik	year	sent_no	sent_id	naics2	naics2_name	sentence_len	keep_sent
0	The following discussion sets forth the materi...	19617	2019	0	19617_0	52	Finance and Insurance	18	True
1	Readers should not consider any descriptions o...	19617	2019	1	19617_1	52	Finance and Insurance	23	True
2	Any of the risk factors discussed below could ...	19617	2019	2	19617_2	52	Finance and Insurance	53	True
3	JPMorgan Chase's businesses are highly regulat...	19617	2019	4	19617_4	52	Finance and Insurance	25	True
4	JPMorgan Chase is a financial services firm wi...	19617	2019	5	19617_5	52	Finance and Insurance	10	True
...	...	...	...	...	...	...	...	...	...
6793	With the completion of the TFCF acquisition, o...	1744489	2019	194	1744489_194	51	Information	19	True
6794	The increased indebtedness could have the effe...	1744489	2019	195	1744489_195	51	Information	22	True
6795	The increased levels of indebtedness could als...	1744489	2019	196	1744489_196	51	Information	14	True
6796	repurchases and dividends, and other activitie...	1744489	2019	197	1744489_197	51	Information	21	True
6797	Our financial flexibility may be further const...	1744489	2019	198	1744489_198	51	Information	23	True

6798 rows × 9 columns

In [ ]:
```
# save the dataset
# df.to_parquet("10k_sent_2019_firms.parquet", index=False)
```

In [ ]:
```
# # lastly we will load the names of the NAICS2 codes
# df_naics = pd.read_csv("naics2_codes.csv")
# df_naics
```

2. Accessing BERT through HuggingFace

HuggingFace

We will use use the `transformers` library developed by HuggingFace to access and interact with BERT. This library provides very convenient classes (e.g. `Tokenizer`, `Model`, `Pipeline`) that will help us to easily pass our text through BERT (or any other transformer model we wish).

不区分大小写

As a starting point, we will use a basic version of the original BERT model in English that is not case sensitive. We access this model with the name `bert-base-uncased`. You can read more about the model here (https://huggingface.co/bert-base-uncased).

Through the Model Hub (https://huggingface.co/models) you can browse all the available models currently hosted by HuggingFace. Here you will find other types of language models and many more languages (including multilingual models (https://huggingface.co/bert-base-multilingual-cased) and models in Spanish (https://huggingface.co/dccuchile/bert-base-spanish-wwm-cased)).

Text tokenization

我们将首先使用 AutoTokenizer 类从 bert-base-uncased 中加载标记符。BERT 的标记化器是在英语维基百科和图书语料库上训练的，共包含 30,522 个独特标记。

We will start by using the `AutoTokenizer` class to load the tokenizer from `bert-base-uncased`. BERT´s Tokenizer was trained on English Wikipedia and the Book Corpus and contains a total amount of 30,522 unique tokens.

In [ ]:
```
# load a tokenizer using the name of the model we want to use
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

```
Downloading (…)okenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading (…)lve/main/config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading (…)/main/tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

In [ ]:
```
# inspect the configuration of the tokenizer
tokenizer
```

Out[15]:
```
BertTokenizerFast(name_or_path='bert-base-uncased', vocab_size=30522, model_max_length=512, is_fast=True, padding_side='righ
t', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token':
'[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True)
```

In [ ]:
```
# explore the vocabulary of the tokenizer by looking at some words
vocab = tokenizer.get_vocab()
print(f"Total number of tokens in vocabulary: {len(vocab)} \n---------")
for _ in range(10):
    word, idx = random.choice(list(vocab.items()))
    print(word, idx)
```

```
Total number of tokens in vocabulary: 30522
---------
reign 5853
lange 21395
poking 21603
##bush 22427
marguerite 15334
720 22857
〉 1638
sumatra 18262
bungalow 27563
musee 18070
```

将序列列表传递给标记化器对象后，将对每个序列执行以下步骤：将序列分解为 BERT 词汇表中的单个 token；将标记转换为其 ID；添加特殊标记；应用截断和填充（可选）

Passing a list of sequences to the tokenizer object will apply the following steps to each sequence:

1. Breakdown the sequence into individual tokens that are part of BERT's vocabulary
2. Transform tokens into their ids
3. Add special tokens
4. Apply truncation and padding (optional)

In [ ]:
```
# pass all sequences through the tokenizer
encoded_sentences = tokenizer(list(df["sentences"].values),      # list of sequences we want to tokenize
                              truncation=True,                    # truncate sequences longer than specified length
                              max_length=60,                      # maximum number of tokens per sequence
                              padding="max_length",               # pad all sequences to the same size
                              return_tensors='pt'                 # data type of results
                              )
# inspect the results
encoded_sentences.keys()
```

Out[17]:
```
dict_keys(['input_ids', 'token_type_ids', 'attention_mask'])
```

In [ ]:
```
# examine BERT's tokenization in detail for a random sentence
i = np.random.randint(0, len(df))
print("Original sentence:")
print(df.loc[i, "sentences"])
print("\n-----------------------------------------\n")
print("Tokens:")
temp_tokens = encoded_sentences["input_ids"][i]
print(tokenizer.convert_ids_to_tokens(temp_tokens))
print("\n-----------------------------------------\n")
print("Tokens IDs:")
print(temp_tokens)
```

```
Original sentence:
• due to limitations within the debt instruments, restrict our ability to grant liens on property, enter into certain mergers,
dispose of all or substantially all of the assets of Autodesk and its subsidiaries, taken as a whole, materially change our bu
siness and incur subsidiary indebtedness, subject to customary exceptions.

-----------------------------------------


Tokens:
['[CLS]', '•', 'due', 'to', 'limitations', 'within', 'the', 'debt', 'instruments', ',', 'restrict', 'our', 'ability', 'to', 'g
rant', 'lie', '##ns', 'on', 'property', ',', 'enter', 'into', 'certain', 'mergers', ',', 'dispose', 'of', 'all', 'or', 'substa
ntially', 'all', 'of', 'the', 'assets', 'of', 'auto', '##des', '##k', 'and', 'its', 'subsidiaries', ',', 'taken', 'as', 'a',
'whole', ',', 'material', '##ly', 'change', 'our', 'business', 'and', 'inc', '##ur', 'subsidiary', 'ind', '##eb', '##ted', '[S
EP]']

-----------------------------------------


Tokens IDs:
tensor([  101,  1528,  2349,  2000, 12546,  2306,  1996,  7016,  5693,  1010,
        21573,  2256,  3754,  2000,  3946,  4682,  3619,  2006,  3200,  1010,
         4607,  2046,  3056, 28585,  1010, 27764,  1997,  2035,  2030, 12381,
         2035,  1997,  1996,  7045,  1997,  8285,  6155,  2243,  1998,  2049,
        20178,  1010,  2579,  2004,  1037,  2878,  1010,  3430,  2135,  2689,
         2256,  2449,  1998,  4297,  3126,  7506, 27427, 15878,  3064,   102])
```

There are several important features of the tokenization process that are worth highlighting:

1. **Special Tokens**: BERT's tokenizer introduces three types of special tokens to each sentence it tokenizes.
    - *Class token* `[CLS]` : Gets introduced at the start of each sequence and, broadly speaking, it is intented to capture the relevant information of a sequence for a particular prediction task
    - *End of sequence token* `[SEP]` : Demarcates the end of a sequence. This token becomes very relevant in sceneraios where a single sequence contains two distinc pieces of information (e.g. question/answer)
    - *Padding token* `[PAD]` : Facillitates the use of arrays and tensors by making all sequences of equal length
2. **Punctuation marks** get their own tokens
3. **Subwords**: Words that are not included in BERT's vocabulary get divided into subwords that are part of the vocabulary.

Loading and using a model

We will now use the `AutoModel` class to load our model and transform our tokenized sequences into their embedded representations.

In [ ]:
```
# HuggingFace's generic class for working with language models out-of-the-box
AutoModel
```

Out[36]:
```
transformers.models.auto.modeling_auto.AutoModel
```

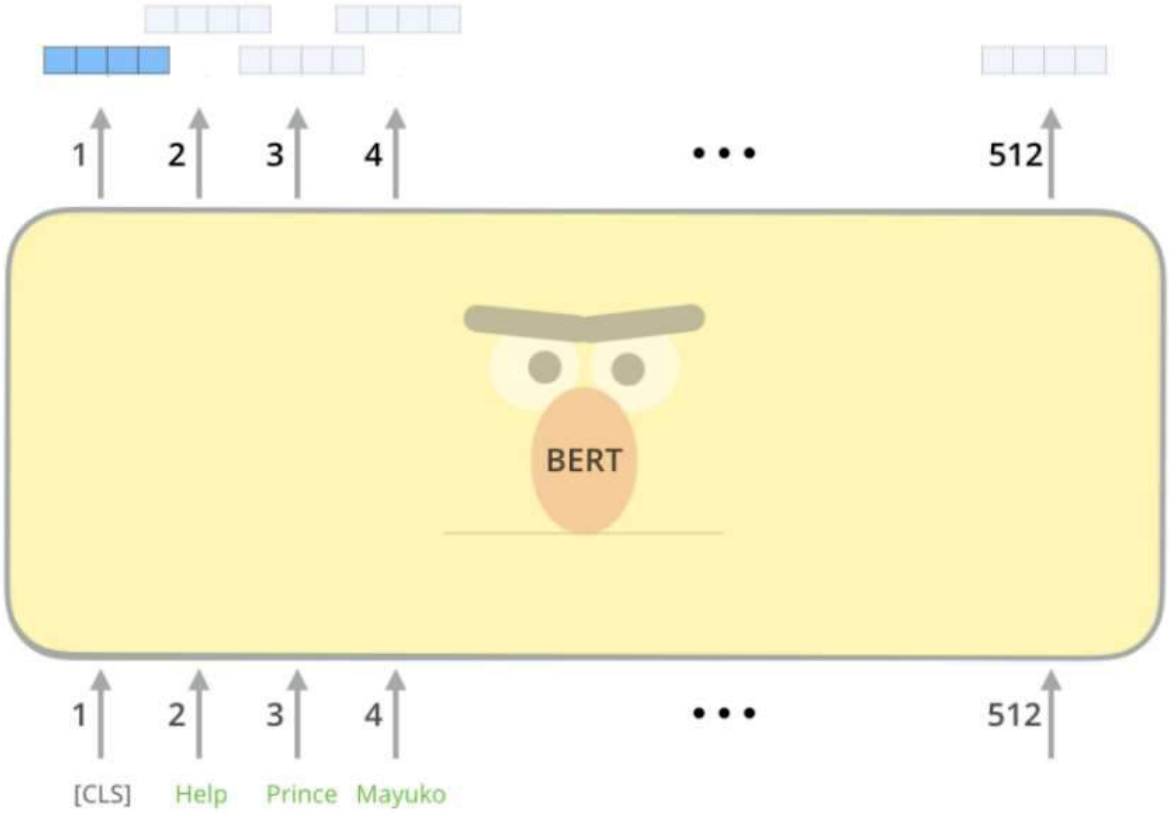
In [ ]:
```
# load a model using its name and explore its configuration
model = AutoModel.from_pretrained("bert-base-uncased",          # our choice of model
                                  output_hidden_states=True,    # output all hidden states so that we can fully explore the mode
                                  output_attentions=True        # output attention weigths so that we can fully explore the mode
                                  )

# put model in evaluation model (we will not do any training)
model = model.eval()
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.seq_relationship.bi
as', 'cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.weight', 'cls.predict
ions.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.transfor
m.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another ar
chitecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identica
l (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

In [ ]:
```
# if we wish to further inspect the model's configuration in detail we can use the config attribute
#print(model.config)
```

Passing a sequence through the model

Generating an embedded representation of a sequence with BERT requires passing its tokens through multiple layers of trained weights.



In [ ]:
```
# lets first get a single sentence as an example
sent_position = 7
sent = df.loc[sent_position, "sentences"]
print(sent + "\n")

# tokenize
sent_encoded = tokenizer(sent,  max_length=60, padding="max_length", truncation=True, return_tensors='pt')
sent_encoded["input_ids"]
```

```
JPMorgan Chase has experienced an extended period of significant change in laws and regulations affecting the financial servic
es industry, both within and outside the U.S.
```

Out[39]:
```
tensor([[  101, 16545,  5302, 16998,  5252,  2038,  5281,  2019,  3668,  2558,
          1997,  3278,  2689,  1999,  4277,  1998,  7040, 12473,  1996,  3361,
          2578,  3068,  1010,  2119,  2306,  1998,  2648,  1996,  1057,  1012,
          1055,  1012,   102,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0]])
```

In [ ]:
```
# apply forward pass through the model (do not accumulate gradients; we are not training)
with torch.no_grad():
    result = model(**sent_encoded)
```

In [ ]:
```
# what is "result" ?
```

In [ ]:
```
# explore output from model
print(f"Number of hidden layers: {len(result.hidden_states)}")
print(f"Shape of output of each hidden layer: {result.hidden_states[5].shape}") # batch_size, number of tokens, embedding dimensi
print(f"Shape of pooler output: {result.pooler_output.shape}") # batch_size, embedding dimension
```

```
Number of hidden layers: 13
Shape of output of each hidden layer: torch.Size([1, 60, 768])
Shape of pooler output: torch.Size([1, 768])
```

Creating a sequence representation

There are several ways in which the output of BERT can be used to generate an embedded representation of a sequence of text. We will show some of them below. However, following the way in which BERT was pre-trained, we will focus on the embedded representation of the `[CLS]` token in the last hidden layer. This representation, is the one used to fullfil the next sentence prediction task on which BERT is trained.

In [ ]:
```
# use the embedding of the [CLS] token as the representation of the sequence
cls_emb = result.hidden_states[-1][0][0]
print(f"Shape of [CLS] embedding: {cls_emb.shape}")
```

```
Shape of [CLS] embedding: torch.Size([768])
```

In [ ]:
```
# we can also average the embeddings of all tokens in a given hidden state (e.g. layer 11)
avg_emb = torch.mean(result.hidden_states[-2][0], dim=0)
print(f"Shape of average embedding: {avg_emb.shape}")
```

```
Shape of average embedding: torch.Size([768])
```

3. Generating features for a regression model

We will use the embedded sequence representation that we have constructed as a covariate of a prediction model. Concretely, we will estimate a multivariate logistic regression with regularization where, for each sentence i in our corpus, we predict the 2-digit NAICS sector of the firm.

BERT features

HuggingFace provides a very convenient interface for tokenizing and passing sequences through the model with few lines of code. To do this, we will use the `pipeline()` class. This class allows us to choose a particular task (e.g. features-extraction, text-classification) and, with the appropriate model and tokenizer, it will generate the correct output for the task (e.g. embedded features, classification probabilitites). All the available pipelines can be explored here (https://huggingface.co/docs/transformers/main_classes/pipelines).

In [ ]:
```
# generate features from text using a pipeline object
feature_extraction = pipeline(task="feature-extraction",       # define the task
                              model=model,                     # model should be appropriate for the task
                              tokenizer=tokenizer,             # selected tokenizer
                              batch_size=16,                   # batching only supported for GPUs
                              device=0,                        # -1 for CPU and 0 for GPU
                              framework="pt"                   # data type (pt = pytorch)
                              )
# verify the task we selected
feature_extraction.task
```

Out[45]:
```
'feature-extraction'
```

In [ ]:
```
# transform dataframe into Dataset class (easier to work with pipeline)
dataset = Dataset.from_pandas(df)
dataset
```

Out[46]:
```
Dataset({
    features: ['sentences', 'cik', 'year', 'sent_no', 'sent_id', 'naics2', 'naics2_name', 'sentence_len', 'keep_sent'],
    num_rows: 6798
})
```

In [ ]:
```
# iterate through our sequences to extract the [CLS] embedding (takes a couple of minutes)
all_cls = []
for out in tqdm(feature_extraction(KeyDataset(dataset, "sentences"),
                                   truncation=True,
                                   max_length=60,
                                   padding=False)
                ):

    # extract the [CLS] embedding from all sentences
    all_cls.append(out[0][0])
```

```
  0%|          | 0/425 [00:00<?, ?it/s]
```

In [ ]:
```
# build a dataframe with the features obtained using BERT (num_docs x 768)
df_features_bert = pd.DataFrame(all_cls)
df_features_bert
```

Out[48]:

	0	1	2	3	4	5	6	7	8	9	...	758	759	
0	-0.824840	-0.004663	-0.458258	-0.664349	-0.133645	-0.039017	-0.155485	0.280789	0.480075	-0.193740	...	-0.206574	-0.292948	-0.32(
1	-0.369104	0.082162	-0.140538	-0.194881	-0.238334	-0.413053	0.269552	0.327644	0.107019	-0.212886	...	0.087334	-0.093015	0.07
2	-0.433662	0.031701	-0.500533	-0.002441	0.079370	-0.057810	-0.189215	0.499592	0.035363	0.071956	...	0.286745	-0.215691	0.00
3	-0.509347	0.245026	-0.325838	-0.351404	-0.158214	0.089534	-0.100731	0.706634	-0.016048	0.215428	...	0.005461	-0.109407	0.39
4	-0.479953	-0.079441	-0.270499	-0.252689	-0.241998	0.164565	-0.226165	0.382813	-0.022743	0.010087	...	0.024555	-0.068406	0.28
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
6793	-0.227583	-0.172389	0.536661	-0.039504	0.132648	-0.312585	0.151738	-0.140319	0.089651	-0.462485	...	0.005395	0.064725	0.24
6794	-0.073973	0.056038	-0.006543	-0.051898	-0.151528	0.068489	0.358183	0.371048	0.424025	-0.356336	...	-0.266280	-0.085258	0.05
6795	-0.397435	0.080185	0.016016	0.373276	-0.316202	0.139612	-0.013123	0.159674	0.068773	-0.583276	...	0.072732	0.022394	0.07
6796	-0.641719	-0.102444	-0.131075	0.406779	-0.020079	-0.033183	-0.295510	0.254862	-0.240674	-0.299448	...	0.162494	-0.244314	0.09
6797	-0.256703	-0.164259	-0.046505	0.164935	0.208567	-0.196918	0.063201	0.261453	0.225409	-0.072651	...	-0.193597	0.008161	0.15

6798 rows × 768 columns

In [ ]:
```
# get only the NAICS2 code for each sentence to use as the labels for our regression
labels = df[["naics2"]]
```

In [ ]:
```
print(df_features_bert.shape, labels.shape)
```

```
(6798, 768) (6798, 1)
```

Word embeddings features

We will also generate an embedded representation of sequences using word embeddings from a pre-trained model. This will give us a reference point with respect to which we can assess the quality of the features generated with BERT.

We will use word embeddings estimated with the GloVe algorithm on Wikipedia and a large news corpus. In order to generate a single representation for a whole sentence, we will average the individual word embeddings of all words from the sentence.

In [ ]:
```
# download the model and return as an object ready to use (takes a couple of minutes)
# other available models can be found here: https://kavita-ganesan.com/easily-access-pre-trained-word-embeddings-with-gensim/
#w2v_model = api.load("word2vec-google-news-300")    # model is too large to load in Colab
w2v_model = api.load("glove-wiki-gigaword-300")
```

```
[==================================================] 100.0% 376.1/376.1MB downloaded
```

In [ ]:
```
# explore some of the functionality of the model
w2v_model.most_similar("cat")
```

Out[53]:
```
[('dog', 0.6816747188568115),
 ('cats', 0.6815836429595947),
 ('pet', 0.5870364904403687),
 ('dogs', 0.5407667756080627),
 ('feline', 0.48979708552360535),
 ('monkey', 0.4879434406757355),
 ('horse', 0.4732131063938141),
 ('pets', 0.4634858667850494),
 ('rabbit', 0.4608757495880127),
 ('leopard', 0.4585462808609009)]
```

In [ ]:
```
def w2v_embed(text, w2v_model):
    """ function to generate a single embedded representation of a text
        by averaging individual word embeddings
    """

    # lowercase all text
    lower_text = text.lower()
    # split sentence into words by spliting on white spaces
    words = lower_text.split(" ")

    # get the word2vec embedding of all words in the vocabulary
    word_embeddings = []
    for w in words:
        if w in w2v_model.index_to_key:
            word_emb = w2v_model[w]
            word_embeddings.append(word_emb)

    if word_embeddings:
        # generate a sequence embedding by averaging all embeddings
        word_embeddings = np.array(word_embeddings)
        seq_embedding = np.mean(word_embeddings, axis=0)
        return seq_embedding
    else:
        return np.nan
```

In [ ]:
```
# transform all sequences into their w2v embedded representation (takes a couple of minutes)
all_w2v = []
valid_sents = []
for sent in tqdm(df["sentences"].values):
    seq_emb = w2v_embed(sent, w2v_model)
    if type(seq_emb) == np.ndarray:
        all_w2v.append(seq_emb)
        valid_sents.append(True)
    else:
        valid_sents.append(False)

print(len(all_w2v))
```

```
  0%|          | 0/6798 [00:00<?, ?it/s]
```

```
6798
```

In [ ]:
```
# build a dataframe with the features obtained using Word2Vec
df_features_w2v = pd.DataFrame(all_w2v)
df_features_w2v
```

Out[60]:

	0	1	2	3	4	5	6	7	8	9	...	290	291	
0	-0.032227	0.026865	-0.028346	-0.132104	-0.049463	0.149346	0.043420	0.156053	0.046826	-1.747511	...	-0.087153	-0.113003	0.016
1	-0.095706	0.193188	-0.016498	-0.212017	-0.073117	0.107209	-0.008054	0.090408	0.097971	-1.793021	...	-0.134389	-0.263703	0.118
2	-0.045260	0.183034	-0.002297	-0.129991	-0.194229	0.086583	-0.003636	0.219974	0.031304	-1.657202	...	-0.113471	-0.223888	0.017
3	0.077517	0.060096	-0.083158	-0.160014	-0.056113	0.095514	0.032513	0.268923	0.053303	-1.599497	...	-0.004302	-0.252440	-0.031
4	-0.073049	0.000522	0.065742	-0.130519	-0.045992	-0.017848	-0.023752	0.297938	-0.037348	-1.521862	...	-0.076941	-0.237747	-0.194
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
6793	-0.203348	0.040914	0.139001	-0.167000	-0.168100	0.101075	-0.056294	0.076812	-0.034586	-1.110907	...	-0.083327	-0.112596	-0.076
6794	-0.116184	0.126143	-0.065551	-0.168580	-0.025757	0.100584	-0.052877	0.151812	0.039062	-1.816366	...	-0.114093	-0.246312	0.011
6795	-0.177607	0.135831	0.021253	-0.127163	0.070258	0.050349	0.253045	0.228192	-0.016486	-1.813566	...	-0.220819	-0.221146	0.007
6796	-0.162263	0.093744	-0.030305	-0.230433	-0.089058	0.042652	-0.039274	0.131659	0.080708	-1.544417	...	-0.038346	-0.156336	-0.058
6797	-0.109844	-0.031461	0.069368	-0.184846	0.047674	0.025851	-0.025306	0.173858	-0.005223	-1.651489	...	-0.078782	-0.186980	0.104

6798 rows × 300 columns

In [ ]:
```
# get only the labels for those sentences with w2v embeddings
labels_w2v = labels[valid_sents]
labels_w2v.reset_index(drop=True, inplace=True)
len(labels_w2v), len(df_features_w2v)
```

Out[61]: (6798, 6798)

In [ ]:
```
## we could also train a Word2Vec model on our own corpus
# gensim_model = Word2Vec(sentences=df.sentences,       # corpus
#                         vector_size=100,              # embedding dimension
#                         window=4,                     # words before and after to take into consideration
#                         sg=1,                         # use skip-gram
#                         negative=5,                   # number of negative examples for each positive one
#                         alpha=0.025,                  # initial learning rate
#                         min_alpha=0.0001,             # minimum learning rate
#                         epochs=5,                     # number of passes through the data
#                         min_count=1,                  # words that appear less than this are removed
#                         workers=1,                    # we use 1 to ensure replicability
#                         seed=92                       # for replicability
#                        )
```

In [ ]:
```
# restrict BERT features to those sentences for which
# we have a word embedding representation
df_features_bert = df_features_bert[valid_sents]
df_features_bert.reset_index(drop=True, inplace=True)
len(labels), len(df_features_bert)
```

Out[66]: (6798, 6798)

Estimate regressions

In [ ]:
```
# create list with all the indexes of available sentences
sent_idxs = list(range(0, len(labels_w2v)))
len(sent_idxs)
```

Out[64]: 6798

In [ ]:
```
# perform a train/test split
train_idxs, test_idxs = train_test_split(sent_idxs, test_size=0.2, random_state=92)
print(f" Train sentences: {len(train_idxs)}\n", f"Test sentences: {len(test_idxs)}")
```

```
Train sentences: 5438
Test sentences: 1360
```

In [ ]:
```
# select idxs for training and testing

# BERT
train_bert_features = df_features_bert.loc[train_idxs]
test_bert_features = df_features_bert.loc[test_idxs]

# Word embeddings
train_w2v_features = df_features_w2v.loc[train_idxs]
test_w2v_features = df_features_w2v.loc[test_idxs]

# Labels
train_labels = labels_w2v.loc[train_idxs]
test_labels = labels_w2v.loc[test_idxs]
```

In [ ]:
```
# BERT: fit a multinomial logistic regression to predict the sector of each sentence
lr_bert = LogisticRegression(penalty="l2",
                             multi_class = "multinomial",
                             solver="lbfgs",
                             max_iter=100000)

lr_bert.fit(train_bert_features, train_labels.values.ravel())

# Word embeddings: fit a multinomial logistic regression to predict the sector of each sentence
lr_w2v = LogisticRegression(penalty="l2",
                            multi_class = "multinomial",
                            solver="lbfgs",
                            max_iter=100000)

lr_w2v.fit(train_w2v_features, train_labels.values.ravel())
```

Out[68]:
```
LogisticRegression(max_iter=100000, multi_class='multinomial')
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:
```
# get in sample predictions for both models
pred_train_bert = lr_bert.predict(train_bert_features)
pred_train_w2v = lr_w2v.predict(train_w2v_features)

print(f"Random guess accuracy: {1/len(df.groupby('naics2').size())}\n")

print("============================\nBERT results:")
train_acc_bert = accuracy_score(train_labels, pred_train_bert)
print(f"In sample accuracy: {train_acc_bert}")
train_f1_bert = f1_score(train_labels, pred_train_bert, average="micro")
print(f"In sample F1 score: {train_f1_bert}")
train_precision_bert = precision_score(train_labels, pred_train_bert, average="micro")
print(f"In sample precision score: {train_precision_bert}")
train_recall_bert = recall_score(train_labels, pred_train_bert, average="micro")
print(f"In sample recall score: {train_recall_bert}")

print("\n============================\nWord embeddings results:")
train_acc_w2v = accuracy_score(train_labels, pred_train_w2v)
print(f"In sample accuracy: {train_acc_w2v}")
train_f1_w2v = f1_score(train_labels, pred_train_w2v, average="micro")
print(f"In sample F1 score: {train_f1_w2v}")
train_precision_w2v = precision_score(train_labels, pred_train_w2v, average="micro")
print(f"In sample precision score: {train_precision_w2v}")
train_recall_w2v = recall_score(train_labels, pred_train_w2v, average="micro")
print(f"In sample recall score: {train_recall_w2v}")
```

```
Random guess accuracy: 0.25

============================
BERT results:
In sample accuracy: 0.7782272894446488
In sample F1 score: 0.7782272894446489
In sample precision score: 0.7782272894446488
In sample recall score: 0.7782272894446488

============================
Word embeddings results:
In sample accuracy: 0.6456417800662008
In sample F1 score: 0.6456417800662008
In sample precision score: 0.6456417800662008
In sample recall score: 0.6456417800662008
```

In [ ]:
```
# get out-of-sample predictions for both models
pred_test_bert = lr_bert.predict(test_bert_features)
pred_test_w2v = lr_w2v.predict(test_w2v_features)

print(f"Random guess accuracy: {1/len(df.groupby('naics2').size())}\n")

print("==========================\nBERT results:")
test_acc_bert = accuracy_score(test_labels, pred_test_bert)
print(f"Test accuracy: {test_acc_bert}")
test_f1_bert = f1_score(test_labels, pred_test_bert, average="micro")
print(f"Test F1 score: {test_f1_bert}")
test_precision_bert = precision_score(test_labels, pred_test_bert, average="micro")
print(f"Test precision score: {test_precision_bert}")
test_recall_bert = recall_score(test_labels, pred_test_bert, average="micro")
print(f"Test recall score: {test_recall_bert}")

print("\n==========================\nWord embeddings results:")
test_acc_w2v = accuracy_score(test_labels, pred_test_w2v)
print(f"Test accuracy: {test_acc_w2v}")
test_f1_w2v = f1_score(test_labels, pred_test_w2v, average="micro")
print(f"Test F1 score: {test_f1_w2v}")
test_precision_w2v = precision_score(test_labels, pred_test_w2v, average="micro")
print(f"Test precision score: {test_precision_w2v}")
test_recall_w2v = recall_score(test_labels, pred_test_w2v, average="micro")
print(f"Test recall score: {test_recall_w2v}")
```

```
Random guess accuracy: 0.25

==========================
BERT results:
Test accuracy: 0.6272058823529412
Test F1 score: 0.6272058823529412
Test precision score: 0.6272058823529412
Test recall score: 0.6272058823529412

==========================
Word embeddings results:
Test accuracy: 0.6161764705882353
Test F1 score: 0.6161764705882353
Test precision score: 0.6161764705882353
Test recall score: 0.6161764705882353
```

In [ ]:
```
# compare the 2 methods on the test sample
df_compare = pd.DataFrame({"accuracy": [test_acc_w2v, test_acc_bert],
                           "model": ["word_embeddings", "BERT"]})
df_compare
```

Out[71]:

	accuracy	model
0	0.616176	word_embeddings
1	0.627206	BERT

Out-of-corpus sentence

We can now also use our estimated regression to predict the sector of any given sequence of text we migth imagine. We will demonstrate this by using BERT.

In [ ]:
```
# define a target sentence
outside_target = "We are worried about misinformation and fake news."

# tokenize sentence
outside_tokens = tokenizer(outside_target, return_tensors='pt')
outside_tokens = outside_tokens.to("cuda")  # required when using GPU

# apply forward pass through the model (do not accumulate gradients; we are not training)
with torch.no_grad():
    result = model(**outside_tokens, output_attentions=True)

# extract [CLS] token embedding from last layer
outside_cls_emb = result.hidden_states[-1][0][0]
outside_cls_emb = outside_cls_emb.cpu() # required when using GPU
print(f"Shape of [CLS] embedding: {outside_cls_emb.shape}")
```

```
Shape of [CLS] embedding: torch.Size([768])
```

In [ ]:
```
df.head(2)
```

Out[75]:

	sentences	cik	year	sent_no	sent_id	naics2	naics2_name	sentence_len	keep_sent
0	The following discussion sets forth the materi...	19617	2019	0	19617_0	52	Finance and Insurance	18	True
1	Readers should not consider any descriptions o...	19617	2019	1	19617_1	52	Finance and Insurance	23	True

In [ ]:
```
# generate class prediction for out-of-corpus sentence
outside_prediction = lr_bert.predict(outside_cls_emb.numpy().reshape(1, -1))
print(f"Predicted NAICS2 code: {outside_prediction[0]}")

# transform sector code into name
prediction_name = df.loc[df["naics2"] == outside_prediction[0]]["naics2_name"].values[0]
print(f"Predicted NAICS2 sector: {prediction_name}\n--------\n")

# generate probability predictions for out-of-corpus sentence
outside_probs = lr_bert.predict_proba(outside_cls_emb.numpy().reshape(1, -1))
outside_probs

for prob, code in zip(outside_probs[0], lr_bert.classes_):
    code_name = df.loc[df["naics2"] == code]["naics2_name"].values[0]
    print(f"Predicted probability for {code_name}: {np.round(prob,3)}")
```

```
Predicted NAICS2 code: 51
Predicted NAICS2 sector: Information
--------

Predicted probability for Manufacturing: 0.003
Predicted probability for Retail Trade: 0.279
Predicted probability for Information: 0.702
Predicted probability for Finance and Insurance: 0.016
```

In [ ]:
```
# define a target sentence
outside_target = "Our production of cars is affected by the price of steel."

# tokenize sentence
outside_tokens = tokenizer(outside_target, return_tensors='pt')
outside_tokens = outside_tokens.to("cuda")  # required when using GPU

# apply forward pass through the model (do not accumulate gradients; we are not training)
with torch.no_grad():
    result = model(**outside_tokens, output_attentions=True)

# extract [CLS] token embedding from last layer
outside_cls_emb = result.hidden_states[-1][0][0]
outside_cls_emb = outside_cls_emb.cpu() # required when using GPU

# generate class prediction for out-of-corpus sentence
outside_prediction = lr_bert.predict(outside_cls_emb.numpy().reshape(1, -1))
print(f"Predicted NAICS2 code: {outside_prediction[0]}")

# transform sector code into name
prediction_name = df.loc[df["naics2"] == outside_prediction[0]]["naics2_name"].values[0]
print(f"Predicted NAICS2 sector: {prediction_name}\n--------\n")

# generate probability predictions for out-of-corpus sentence
outside_probs = lr_bert.predict_proba(outside_cls_emb.numpy().reshape(1, -1))
outside_probs

for prob, code in zip(outside_probs[0], lr_bert.classes_):
    code_name = df.loc[df["naics2"] == code]["naics2_name"].values[0]
    print(f"Predicted probability for {code_name}: {np.round(prob,3)}")
```

```
Predicted NAICS2 code: 33
Predicted NAICS2 sector: Manufacturing
--------

Predicted probability for Manufacturing: 0.955
Predicted probability for Retail Trade: 0.039
Predicted probability for Information: 0.005
Predicted probability for Finance and Insurance: 0.0
```

In [ ]:
```
# define a target sentence
outside_target = "The decisions from the federal reserve board can affect us greatly."

# tokenize sentence
outside_tokens = tokenizer(outside_target, return_tensors='pt')
outside_tokens = outside_tokens.to("cuda")  # required when using GPU

# apply forward pass through the model (do not accumulate gradients; we are not training)
with torch.no_grad():
    result = model(**outside_tokens, output_attentions=True)

# extract [CLS] token embedding from last layer
outside_cls_emb = result.hidden_states[-1][0][0]
outside_cls_emb = outside_cls_emb.cpu() # required when using GPU

# generate class prediction for out-of-corpus sentence
outside_prediction = lr_bert.predict(outside_cls_emb.numpy().reshape(1, -1))
print(f"Predicted NAICS2 code: {outside_prediction[0]}")

# transform sector code into name
prediction_name = df.loc[df["naics2"] == outside_prediction[0]]["naics2_name"].values[0]
print(f"Predicted NAICS2 sector: {prediction_name}\n--------\n")

# generate probability predictions for out-of-corpus sentence
outside_probs = lr_bert.predict_proba(outside_cls_emb.numpy().reshape(1, -1))
outside_probs

for prob, code in zip(outside_probs[0], lr_bert.classes_):
    code_name = df.loc[df["naics2"] == code]["naics2_name"].values[0]
    print(f"Predicted probability for {code_name}: {np.round(prob,3)}")
```

```
Predicted NAICS2 code: 52
Predicted NAICS2 sector: Finance and Insurance
--------

Predicted probability for Manufacturing: 0.004
Predicted probability for Retail Trade: 0.001
Predicted probability for Information: 0.001
Predicted probability for Finance and Insurance: 0.994
```

4. Domain-specific models

So far, we have used the base version of BERT to generate features. However, one concern with this approach is that the language in the training data used for base BERT (i.e. Wikipedia and Books) migth be very different from the language in 10-K reports. In order to alleviate this concern, we will use a different version of the model trained on 260,773 10-K filings from 1993-2019.

These family of models are called SEC-BERT (https://huggingface.co/nlpaueb/sec-bert-base) and were developed by the Natural Language Processing Group at the Athens University of Economics and Business.

> Remember that you can use HuggingFace's ModelHub (https://huggingface.co/models) to explore more models trained on different corpora.

In [ ]:
```
# load a tokenizer using the name of the model we want to use
sec_tokenizer = AutoTokenizer.from_pretrained("nlpaueb/sec-bert-base")
# inspect the configuration of the tokenizer
sec_tokenizer
```

```
Downloading (…)okenizer_config.json:   0%|          | 0.00/263 [00:00<?, ?B/s]

Downloading (…)lve/main/config.json:   0%|          | 0.00/568 [00:00<?, ?B/s]

Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/221k [00:00<?, ?B/s]

Downloading (…)cial_tokens_map.json:   0%|          | 0.00/112 [00:00<?, ?B/s]
```

Out[82]:
```
BertTokenizerFast(name_or_path='nlpaueb/sec-bert-base', vocab_size=30000, model_max_length=512, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True)
```

In [ ]:
```
# load the model trained on 10-K report using its name
sec_model = AutoModel.from_pretrained("nlpaueb/sec-bert-base")

# put model in evaluation model (we will not do any training)
sec_model = sec_model.eval()
```

```
Downloading pytorch_model.bin:   0%|          | 0.00/439M [00:00<?, ?B/s]

Some weights of the model checkpoint at nlpaueb/sec-bert-base were not used when initializing BertModel: ['cls.seq_relationship.bias', 'cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.weight', 'cls.predictions.bias', 'cls.predictions.decoder.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

In [ ]:
```
# we will use the pipeline object again (but with the new model and tokenizer)
sec_feature_extraction = pipeline(task="feature-extraction",          # define the task
                                  model=sec_model,                    # model should be appropriate for the task
                                  tokenizer=sec_tokenizer,            # selected tokenizer
                                  batch_size=16,                      # batching only supported for GPUs
                                  device=0,                           # -1 for CPU and 0 for GPU
                                  framework="pt"                      # data type
                                 )
sec_feature_extraction.task
```

Out[84]:
```
'feature-extraction'
```

In [ ]:
```
# iterate through our sequences to extract the [CLS] embedding
all_cls_sec = []
for out in tqdm(sec_feature_extraction(KeyDataset(dataset, "sentences"),
                                       truncation=True,
                                       max_length=60,
                                       padding=False)):

    # extract the [CLS] embedding from all sentences
    all_cls_sec.append(out[0][0])
```

```
  0%|          | 0/425 [00:00<?, ?it/s]
```

In [ ]:
```
# build a dataframe with the features obtained using SEC BERT
df_features_sec = pd.DataFrame(all_cls_sec)
df_features_sec = df_features_sec[valid_sents]
df_features_sec.reset_index(drop=True, inplace=True)
df_features_sec
```

Out[86]:

	0	1	2	3	4	5	6	7	8	9	...	758	759	
0	0.376939	-0.140579	-0.428750	0.295816	0.271135	-0.164485	0.030321	-0.265742	0.385736	0.260340	...	0.003370	0.311562	0.08
1	0.533621	0.050102	-0.311264	0.495807	-0.068415	-0.264094	-0.088141	-0.010286	0.499246	0.296850	...	-0.532170	0.676394	0.21
2	0.408385	0.025169	-0.312310	0.154127	0.127272	-0.208127	-0.052731	-0.392785	0.280328	0.131218	...	-0.449754	0.467374	0.13
3	0.254688	0.179559	-0.061918	-0.161153	0.315859	0.179094	0.210361	-0.021678	-0.120718	0.121269	...	-0.608426	-0.188357	0.29
4	-0.389278	-0.208216	0.183656	0.204948	0.430644	-0.412392	0.351476	-0.038184	0.205474	0.198621	...	-0.167565	0.059471	0.25
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
6793	0.112713	0.065506	0.354329	0.322967	0.433434	0.116434	-0.329728	-0.267638	-0.343811	0.358039	...	-0.284816	0.245086	-0.07
6794	0.084064	-0.047360	-0.087702	-0.107321	-0.081663	-0.226985	-0.772778	-0.170690	-0.044984	-0.238042	...	-0.243459	0.212811	0.00
6795	0.460132	0.103867	-0.123413	-0.395757	-0.081955	-0.298237	-0.544906	-0.214941	-0.178917	0.107354	...	0.033614	0.328427	0.00
6796	0.635484	-0.012616	-0.035677	-0.500130	0.373538	-0.077321	-0.458544	-0.192691	-0.322040	-0.010693	...	-0.186516	-0.092872	0.08
6797	0.493296	-0.091529	0.079728	-0.305662	0.220932	0.004647	-0.414569	0.122064	-0.307049	0.261391	...	-0.294529	0.041775	0.11

6798 rows × 768 columns

In [ ]:
```
# split features using the previous train/test split
train_sec_features = df_features_sec.loc[train_idxs]
test_sec_features = df_features_sec.loc[test_idxs]
```

In [ ]:
```
# fit a logistic regression to predict the sector of each sentence
# (with the same parameters as before)
lr_sec = LogisticRegression(penalty="l2",
                            multi_class = "multinomial",
                            solver="lbfgs",
                            max_iter=100000)

lr_sec.fit(train_sec_features, train_labels.values.ravel())
```

Out[88]:
```
LogisticRegression(max_iter=100000, multi_class='multinomial')
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:
```
# get in sample predictions
pred_train_sec = lr_sec.predict(train_sec_features)

print(f"Random guess accuracy: {1/len(df.groupby('naics2').size())}\n")

print("==========================\nIn sample SEC BERT results:")
train_acc_sec = accuracy_score(train_labels, pred_train_sec)
print(f"In sample accuracy: {train_acc_sec}")
train_f1_sec = f1_score(train_labels, pred_train_sec, average="micro")
print(f"In sample F1 score: {train_f1_sec}")
train_precision_sec = precision_score(train_labels, pred_train_sec, average="micro")
print(f"In sample precision score: {train_precision_sec}")
train_recall_sec = recall_score(train_labels, pred_train_sec, average="micro")
print(f"In sample recall score: {train_recall_sec}")

# get out-of-sample predictions
pred_test_sec = lr_sec.predict(test_sec_features)

print("==========================\nOut of sample SEC BERT results:")
test_acc_sec = accuracy_score(test_labels, pred_test_sec)
print(f"Test accuracy: {test_acc_sec}")
test_f1_sec = f1_score(test_labels, pred_test_sec, average="micro")
print(f"Test F1 score: {test_f1_sec}")
test_precision_sec = precision_score(test_labels, pred_test_sec, average="micro")
print(f"Test precision score: {test_precision_sec}")
test_recall_sec = recall_score(test_labels, pred_test_sec, average="micro")
print(f"Test recall score: {test_recall_sec}")
```

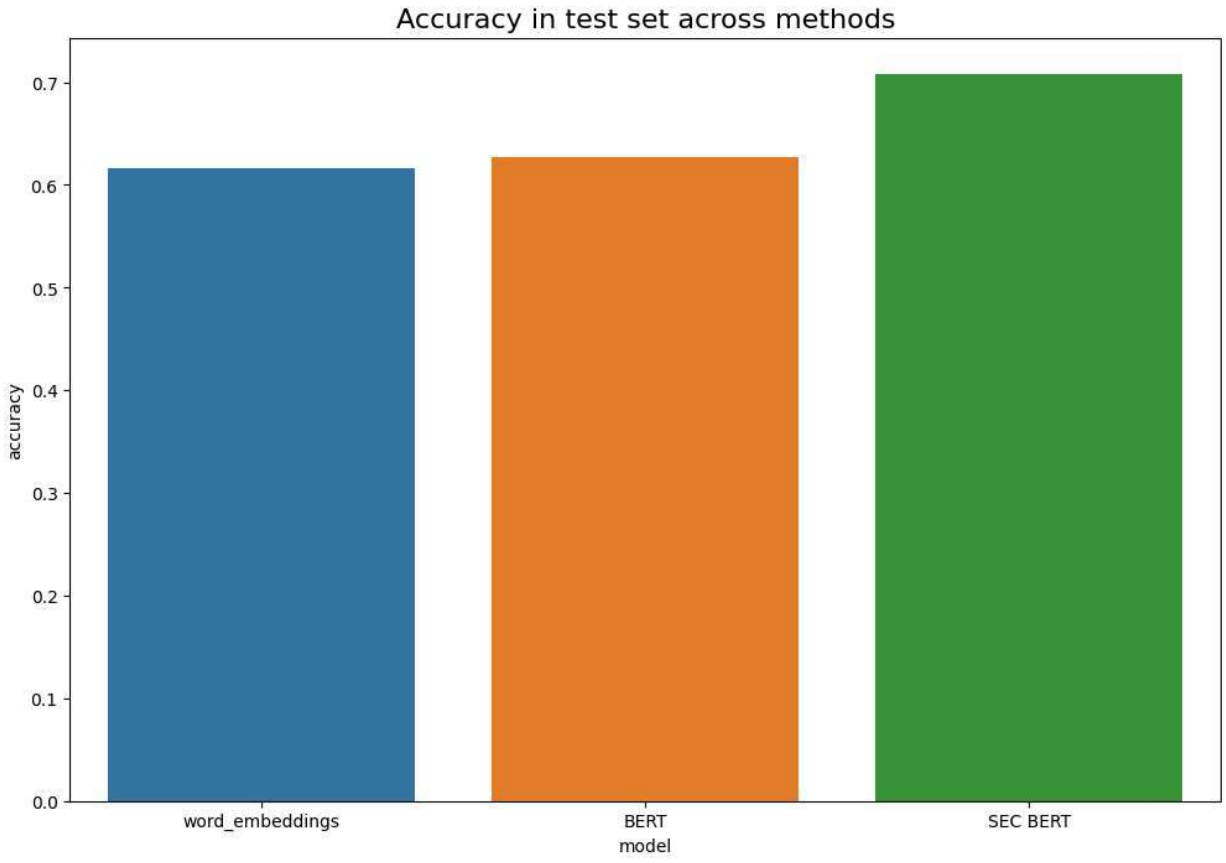
```
Random guess accuracy: 0.25

==========================
In sample SEC BERT results:
In sample accuracy: 0.8433247517469658
In sample F1 score: 0.8433247517469658
In sample precision score: 0.8433247517469658
In sample recall score: 0.8433247517469658
==========================
Out of sample SEC BERT results:
Test accuracy: 0.7080882352941177
Test F1 score: 0.7080882352941177
Test precision score: 0.7080882352941177
Test recall score: 0.7080882352941177
```

In [ ]:
```
# compare all 3 methods
df_compare = pd.DataFrame({"accuracy": [test_acc_w2v, test_acc_bert, test_acc_sec],
                           "model": ["word_embeddings", "BERT", "SEC BERT"]})
df_compare
```

Out[90]:

	accuracy	model
0	0.616176	word_embeddings
1	0.627206	BERT
2	0.708088	SEC BERT

In [ ]:
```
plt.figure(figsize=(12,8))
sns.barplot(data=df_compare, x="model", y="accuracy")
plt.title("Accuracy in test set across methods", fontsize=16)
plt.show()
```



Finetuning a large language model

(https://colab.research.google.com/github/sekhansen/text_algorithms_econ/blob/main/notebooks/6_llm_finetuning.ipynb)

0. Setup

In [1]:
```
# Instructions for Colab:
# 1. First CHANGE RUNTIME TYPE to GPU
# 2. Run install commands
# 3. You might need to RESTART RUNTIME
# 4. Run the rest of the cells below
```

In [1]:
```
%%capture

# install required libraries
!pip3 install transformers                  # HuggingFace library for interacting with BERT (and multiple other models)
!pip3 install datasets                      # HuggingFace library to process dataframes
!pip3 install sentence-transformers         # library to use Sentence Similarity BERT
!pip3 install bertviz                       # visualize BERT's attention weigths
!pip3 install annoy                         # Spotify's library for finding nearest neighbours
!pip3 install ipywidgets
```

In [3]:
```
# (COLAB) you might need to restart RUNTIME after installing packages!
```

In [4]:
```
# import libraries
import gdown
import pandas as pd
import numpy as np
import gdown
import seaborn as sns
import matplotlib.pyplot as plt
import torch

from transformers import AutoModel, BertModel, AutoTokenizer, AutoModelForSequenceClassification, pipeline, TrainingA
from transformers.pipelines.base import KeyDataset
from datasets import load_dataset, load_metric, Dataset, DatasetDict

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.metrics import f1_score, precision_score, recall_score, accuracy_score

from google.colab import output
output.enable_custom_widget_manager()

# test GPU
print(f"GPU: {torch.cuda.is_available()}")
```

```
GPU: True
```

In [5]:
```
# define dictionary with paths to data in Google Drive
urls_dict = {"10k_sent_2019_firms":        ("https://drive.google.com/uc?id=1eQB8rwSklyVD3u8sZImFII74b7jBeUIL", "parque
             }
```

In [6]:
```
# download all files
for file_name, attributes in urls_dict.items():
    url = attributes[0]
    extension = attributes[1]
    gdown.download(url, f"./{file_name}.{extension}", quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1eQB8rwSklyVD3u8sZImFII74b7jBeUIL (https://drive.google.com/uc?id=1eQB8rwSklyVD
3u8sZImFII74b7jBeUIL)
To: /content/10k_sent_2019_firms.parquet
100%|██████████| 720k/720k [00:00<00:00, 82.0MB/s]
```

1. Load and prepare the data

This tutorial uses text data from the **10-K reports** filed by publicly-traded firms in the U.S. in 2019. 10-K reports are a very rich source of data since firms include information regarding their organizational structure, financial performance and risk factors. We will use a version of the data where the risk factors section of each report has been splitted into sentences and each sentence has been assigned an ID that combines the firm identifier (i.e. **CIK**) and a sentence number. The raw data we use has a total of 1,744,131 sentences for 4,033 firms.

In [7]:
```
# read data
df = pd.read_parquet("10k_sent_2019_firms.parquet")
df
```

Out[7]:

	sentences	cik	year	sent_no	sent_id	naics2	naics2_name	sentence_len	keep_sent
0	The following discussion sets forth the materi...	19617	2019	0	19617_0	52	Finance and Insurance	18	True
1	Readers should not consider any descriptions o...	19617	2019	1	19617_1	52	Finance and Insurance	23	True
2	Any of the risk factors discussed below could ...	19617	2019	2	19617_2	52	Finance and Insurance	53	True
3	JPMorgan Chase's businesses are highly regulat...	19617	2019	4	19617_4	52	Finance and Insurance	25	True
4	JPMorgan Chase is a financial services firm wi...	19617	2019	5	19617_5	52	Finance and Insurance	10	True
...	...	...	...	...	...	...	...	...	...
6793	With the completion of the TFCF acquisition, o...	1744489	2019	194	1744489_194	51	Information	19	True
6794	The increased indebtedness could have the effe...	1744489	2019	195	1744489_195	51	Information	22	True
6795	The increased levels of indebtedness could als...	1744489	2019	196	1744489_196	51	Information	14	True
6796	repurchases and dividends, and other activitie...	1744489	2019	197	1744489_197	51	Information	21	True
6797	Our financial flexibility may be further const...	1744489	2019	198	1744489_198	51	Information	23	True

6798 rows × 9 columns

2. Load model and tokenizer

In [8]:
```
# load a tokenizer using the name of the model we want to use
sec_tokenizer = AutoTokenizer.from_pretrained("nlpaueb/sec-bert-base")

# inspect the configuration of the tokenizer
sec_tokenizer
```

```
Downloading (…)okenizer_config.json:   0%|          | 0.00/263 [00:00<?, ?B/s]
Downloading (…)lve/main/config.json:   0%|          | 0.00/568 [00:00<?, ?B/s]
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/221k [00:00<?, ?B/s]
Downloading (…)cial_tokens_map.json:   0%|          | 0.00/112 [00:00<?, ?B/s]
```

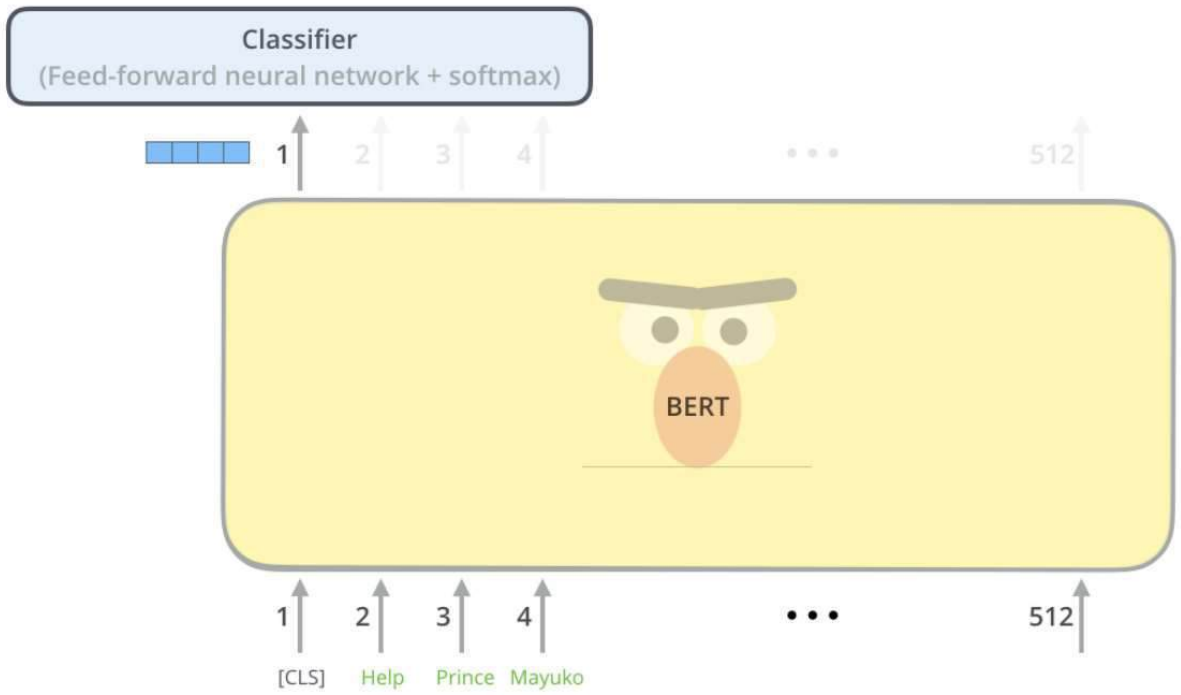
Out[8]:
```
BertTokenizerFast(name_or_path='nlpaueb/sec-bert-base', vocab_size=30000, model_max_length=512, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True)
```

In [9]:
```
# load the model trained on 10-K report using its name
sec_model = AutoModel.from_pretrained("nlpaueb/sec-bert-base")

# put model in evaluation model (we will not do any training)
sec_model = sec_model.eval()
```

```
Downloading pytorch_model.bin:   0%|          | 0.00/439M [00:00<?, ?B/s]

Some weights of the model checkpoint at nlpaueb/sec-bert-base were not used when initializing BertModel: ['cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.predictions.decoder.bias', 'cls.seq_relationship.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.bias', 'cls.seq_relationship.weight', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

3. Finetuning a model

We will show how we can finetune a BERT model for any classification task we want. Finetuning involves training (with a very small learning rate) all of the parameters of the model for a particular task. This is one of the most powerful ways of using modern language models.



Source: Alammar (2018)

In [10]:
```
# get only the NAICS2 code for each sentence to use as the labels for our regression
labels = df[["naics2"]]
```

In [11]:
```
# create list with all the indexes of available sentences
sent_idxs = list(range(0, len(labels)))
len(sent_idxs)
```

Out[11]: 6798

In [12]:
```
# perform a train/test split
train_idxs, test_idxs = train_test_split(sent_idxs, test_size=0.2, random_state=92)
print(f" Train sentences: {len(train_idxs)}\n", f"Test sentences: {len(test_idxs)}")
```

```
Train sentences: 5438
Test sentences: 1360
```

In [13]:
```
# format the train data adequately
df_finetune = df.loc[train_idxs].copy()

df_finetune = df_finetune[["sentences", "naics2"]]
df_finetune.columns = ["sentences", "label"]

# transform labels into integers
df_finetune["label"] = df_finetune["label"].astype(int)
df_finetune

# map labels from original sector code to ints from 0 to num_sectors
num_sectors = len(df_finetune.groupby('label').size())
label2id_label = {k:v for k,v in zip(df_finetune.groupby('label').size().index.values, range(0, num_sectors))}
df_finetune["label"] = df_finetune["label"].apply(lambda x: label2id_label[x])
df_finetune
```

Out[13]:

	sentences	label
3639	We experienced a work stoppage in 2008 when a ...	0
2680	Finally, holders of the Tesla Convertible Note...	0
1507	There can be significant differences between o...	3
911	We also rely on other companies to maintain re...	2
621	The techniques used for attacks by third parti...	0
...	...	...
5007	Our revenues and cash requirements are affecte...	1
710	As is common in our industry, our advertisers ...	2
6162	Global markets for the Company's products and ...	0
4138	Longer payment cycles in some countries, incre...	2
4218	The concentration of our stock ownership limit...	2

5438 rows × 2 columns

In [14]:
```
# format the test data adequately
df_test = df.loc[test_idxs].copy()

df_test = df_test[["sentences", "naics2"]]
df_test.columns = ["sentences", "label"]

# transform labels into integers
df_test["label"] = df_test["label"].astype(int)
df_test

# map labels from original sector code to ints from 0 to num_sectors
df_test["label"] = df_test["label"].apply(lambda x: label2id_label[x])
df_test
```

Out[14]:

	sentences	label
1236	• integration of the acquired company's accou...	2
474	Competition for qualified personnel within the...	3
3418	Any reduction in our and our subsidiaries' cre...	3
6564	Natural disasters or other catastrophes could ...	1
2646	For the battery and drive unit on our current ...	0
...	...	...
1950	Regulatory requirements in the U.S. and in non...	3
6648	If personal information of our customers or em...	2
6529	The evolution of retailing in online and mobil...	1
6452	Our success depends, in part, on our ability t...	1
1040	• recruiting and retaining talented and capab...	2

1360 rows × 2 columns

In [15]:
```
# transform data into Dataset class
finetune_dataset = Dataset.from_pandas(df_finetune)
test_dataset = Dataset.from_pandas(df_test)
```

In [16]:
```
# tokenize the dataset
def tokenize_function(examples):
    return sec_tokenizer(examples["sentences"], max_length=60, padding="max_length", truncation=True)

tokenized_ft = finetune_dataset.map(tokenize_function, batched=True)    # batched=True is key for training
tokenized_test = test_dataset.map(tokenize_function, batched=True)

tokenized_ft
```

```
Map:   0%|          | 0/5438 [00:00<?, ? examples/s]

Map:   0%|          | 0/1360 [00:00<?, ? examples/s]
```

Out[16]:
```
Dataset({
    features: ['sentences', 'label', '__index_level_0__', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 5438
})
```

In [25]:
```
# load the model for finetunning.
# NOTE that we use a different class from the transformers library:
# AutoModel vs. AutoModelForSequenceClassification
num_labels = len(df_finetune.groupby('label').size())
model_ft = AutoModelForSequenceClassification.from_pretrained("nlpaueb/sec-bert-base",
                                                              num_labels=num_labels,
                                                              output_hidden_states=False)
```

```
Some weights of the model checkpoint at nlpaueb/sec-bert-base were not used when initializing BertForSequenceClassif
ication: ['cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.predictions.decoder.bia
s', 'cls.seq_relationship.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.weig
ht', 'cls.predictions.bias', 'cls.seq_relationship.weight', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertForSequenceClassification from the checkpoint of a model trained on a
nother task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreT
raining model).
- This IS NOT expected if you are initializing BertForSequenceClassification from the checkpoint of a model that you
expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassificat
ion model).
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at nlpaueb/sec-bert-bas
e and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

You can find a complete description of all the available parameters of the `TrainingArguments()` class here (https://huggingface.co/docs/transformers/v4.21.2/en/main_classes/trainer#transformers.TrainingArguments).

In [26]:
```
# define the main arguments for training
training_args = TrainingArguments("./",                                  # path to save model
                                  learning_rate=3e-5,                    # we use a very small learning rate
                                  num_train_epochs=5,                    # number of iterations through the corpus
                                  per_device_train_batch_size=8,
                                  per_device_eval_batch_size=1,
                                  evaluation_strategy="no",
                                  save_strategy="no")
```

In [27]:
```
# define the set of metrics to be computed through the training process
def compute_metrics(eval_pred):
    metric1 = load_metric("precision")
    metric2 = load_metric("recall")
    metric3 = load_metric("f1")
    metric4 = load_metric("accuracy")

    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    precision = metric1.compute(predictions=predictions, references=labels, average="micro")["precision"]
    recall = metric2.compute(predictions=predictions, references=labels, average="micro")["recall"]
    f1 = metric3.compute(predictions=predictions, references=labels, average="micro")["f1"]
    accuracy = metric4.compute(predictions=predictions, references=labels)["accuracy"]

    return {"precision": precision, "recall": recall,
            "f1": f1, "accuracy": accuracy}

# by default the Trainer will use MSEloss from (torch.nn) for regression and
# CrossEntropy loss for classification
trainer = Trainer(
    model=model_ft,
    args=training_args,
    train_dataset=tokenized_ft,
    eval_dataset=tokenized_ft,  # in-sample evaluation
    compute_metrics=compute_metrics
)
```

In [28]:
```
# train model (should take around 5 minutes with GPU)
trainer.train()

# save final version of the model
#trainer.save_model("./models/")
```

```
/usr/local/lib/python3.9/dist-packages/transformers/optimization.py:391: FutureWarning: This implementation of AdamW
is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or
set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
```

[3400/3400 06:20, Epoch 5/5]

Step	Training Loss
500	0.831300
1000	0.524400
1500	0.376100
2000	0.246800
2500	0.133800
3000	0.077700

Out[28]:
```
TrainOutput(global_step=3400, training_loss=0.32789999905754535, metrics={'train_runtime': 380.6943, 'train_samples_
per_second': 71.422, 'train_steps_per_second': 8.931, 'total_flos': 838373210251200.0, 'train_loss': 0.3278999990575
4535, 'epoch': 5.0})
```

In [29]:
```
# evaluate final model on the test dataset
results = trainer.predict(tokenized_test)
final_metrics = results[2]
print(final_metrics)
```

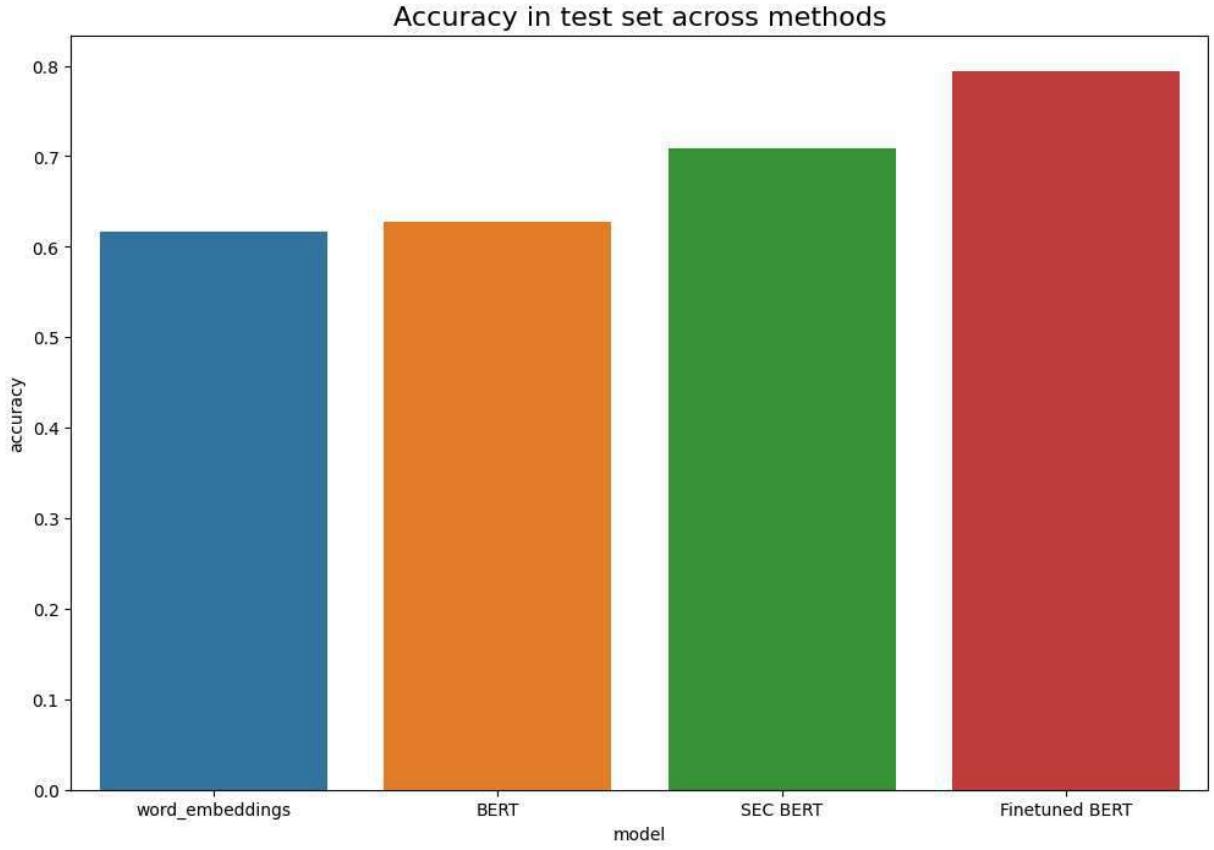
```
{'test_loss': 1.2546815872192383, 'test_precision': 0.7941176470588235, 'test_recall': 0.7941176470588235, 'test_f
1': 0.7941176470588235, 'test_accuracy': 0.7941176470588235, 'test_runtime': 18.7918, 'test_samples_per_second': 72.
372, 'test_steps_per_second': 72.372}
```

In [30]:
```
# compare finetuning to other methods
df_compare = pd.DataFrame({"accuracy": [0.616176, 0.627206, 0.708088, final_metrics["test_accuracy"]],
                           "model": ["word_embeddings", "BERT", "SEC BERT", "Finetuned BERT"]})
df_compare
```

Out[30]:

	accuracy	model
0	0.616176	word_embeddings
1	0.627206	BERT
2	0.708088	SEC BERT
3	0.794118	Finetuned BERT

In [31]:
```
plt.figure(figsize=(12,8))
sns.barplot(data=df_compare, x="model", y="accuracy")
plt.title("Accuracy in test set across methods", fontsize=16)
plt.show()
```



GPT Demonstration

This notebook introduces how to use OpenAI's API to interact with any GPT model. As a illustrative example, we will show how to use ChatGPT to classify a sequence of text from a job posting according to the extent to which it allows remote work.

In [8]:
```
%%capture
!pip3 install transformers openai
```

Setup

In [10]:
```
import openai
import pandas as pd
import numpy as np
from transformers import GPT2TokenizerFast
import time

# add your OpenAI key
openai.api_key = "[YOUR-OPENAI-KEY]"
```

In [12]:
```
## list all available models
#openai.Model.list()
```

Prompt generation

为了从模型中获得足够的结果，必须精心制作提示，以确定我们希望模型执行的任务。在本例中，我们将创建一个提示，提供与文本相关的广泛背景，定义我们感兴趣的四个类别，并指定生成文本的格式。

In order to obtain adequate results from the models, it's important to carefully craft prompts that define the task the we want the model to perform. In this case, we will create a prompt that gives a broad context related to the text, defines our four categories of interest, and specifies a format for the generated text.

In [19]:
```
def prompt_generator(text):
    wfh_prompt = f"""
    You are a data expert working for the Bureau of Labor Statistics specialized
    Your task is to read the text of fragments of job postings and classify them
    The four categories and their definitions are:

    1. No remote work: The text doesn't offer the possibility of working any day
    2. Hybrid work: The text offers the possibility of working one or more days
    3. Fully remote: The text offers the possibility of working all days of the
    4. Unspecified remote: The text mentions the possibility of working remotely

    You always need to provide a classification. If classification is unclear, s
    Please provide the classification and an explanation for your choice in the

    - Classification: [Category Number] . [Category Name]
    - Explanation: [Explanatory text]
```

```
    Text of job posting: "{text}"
    """

    return wfh_prompt
```

In [20]:
```
example_posting = "*This position is eligible for telework and other flexible wo
prompt = prompt_generator(example_posting)
print(prompt)
```

```
    You are a data expert working for the Bureau of Labor Statistics specialized
in analyzing job postings.
    Your task is to read the text of fragments of job postings and classify them
into one of four categories based on the degree of remote work they allow.
    The four categories and their definitions are:

    1. No remote work: The text doesn't offer the possibility of working any day
of the week remotely.
    2. Hybrid work: The text offers the possibility of working one or more days p
er week remotely but not the whole week.
    3. Fully remote: The text offers the possibility of working all days of the w
eek remotely.
    4. Unspecified remote: The text mentions the possibility of working remotely
but doesn't clearly specify the extent of this possibility.

    You always need to provide a classification. If classification is unclear, sa
y "Could not classify".
    Please provide the classification and an explanation for your choice in the f
ollowing format:

    - Classification: [Category Number] . [Category Name]
    - Explanation: [Explanatory text]

    Text of job posting: "*This position is eligible for telework and other flexi
ble work arrangements. Employee participation is at the discretion of the supervi
sor."
```

现在我们有了一个提示和一个可以测试的示例，我们将使用 OpenAI 的 API 生成一个回复。请注意，用于与 ChatGPT（即 ChatCompletion）交互的类与用于与其他 GPT 模型（即 Completion）交互的类不同。

Generate answers with GPT models

Now that we have a prompt and an example to test it on, we will use OpenAI's API to generate a response. Notice that the class that is used for interacting with ChatGPT (i.e. ChatCompletion) is not the same as the one to interact with other GPT models (i.e. Completion).

In [18]:
```
# process prompt with chat-gpt
response = openai.ChatCompletion.create(
                    model="gpt-3.5-turbo",          # ChatGPT
                    messages=[{"role": "system", "content": "You are a data expe
                              {"role": "user", "content": prompt}],
                    n=1,
                    temperature=0.1,                # Higher values means the m
                    max_tokens=50,
                    top_p=1,
                    frequency_penalty=0.0,
                    presence_penalty=0.0,
                    )
```

```
print(response['choices'][0]['message']['content'].strip())
```

```
Classification: 4. Unspecified remote
Explanation: The text mentions the possibility of telework and other flexible wor
k arrangements, but it doesn't clearly specify the extent of this possibility. It
only states that employee participation is at the discretion of the supervisor
```

In [15]:

```
# # process prompt with GPT-3
# response = openai.Completion.create(
#                     model="text-davinci-003",
#                     prompt=prompt,
#                     n=1,
#                     temperature=0,                  # Higher values means the
#                     max_tokens=20,
#                     top_p=1,
#                     frequency_penalty=0.0,
#                     presence_penalty=0.0,
#                     stop=["\n"]                     # Up to 4 sequences where
#                     )

# print(response["choices"][0]["text"])
```