DOPP_G18_Ex3_20210126_Training200_All_Decades

January 26, 2021

Table of Contents

1 Introduction to Assignment

This is the third Exercise of **188.995 Data-Oriented Programming Paradigms**

We are group 18, and consist of: * Guillermo Alamán Requena, Matr. Nr: 11937906 * Michael Ferdinand Moser, Matr. Nr: 01123077 * Paul Joe Maliakel, Matr. Nr: 12012422 * Gunnar Sjúrðarson Knudsen, Matr. Nr: 12028205

In this task we were asked to choose one vaguely worded question, and then narrow the scope, figuring out how to get the data, before finally solving the question at hand. We chose **Question 21**, which contains: * How does the use of various communication languages in countries change over time? * Which languages grow and which disappear, and what are their characteristics? * Are there other factors that correlate with the appearance or disappearance of languages?

We soon realized that the question as stated is far too broad, and we therefore had to limit it.

After having discussed amoung our groups, we came to the following plan:

1.1 First draft

1.1.1 Topic and Questions to answer

We've selected question 21, which is regarding how communication languages in countries change over time.

After having discussed the data available, and planned a workflow, we've decided to try to answer the questions: * How has the English language changed in the past 100 years based on word frequencies, sentence length, …? * Can we find parallel developments between different genres of text? * Can the publication year of a movie/article/whatever be predicted based on the text and its characteristics?

1.1.2 Justification For Limit Of Scope

The sample questions stated in the task description are too broad, to be answered in a single 160 hour project. * Lot's of issues, such as: * Lack of census data; * other changes such as phonetic, semantic and syntactic meanings; * High correlation with e.g: * country population * age of speakers * … * What counts as a language? * dialect? * Mutually Intelligible? * Political dimensions * Multilingual people * How do we check accuracy of the available data? * …

Historical data for language use is likely not available for most languages, as it's topics for great research to estimate merely historical populations - especially before 1850 or so. The evolution of languages are much less documented. Lack of census data overall, but other changes are even harder to gauge, such as phonetic, semantic, and syntactic meanings. Highly correlated with population of countries, but also with "hidden" correlations, such as age of speakers, … Even dead languages can be revived.

What constitutes a language? Dialect? Mutually Intelligible? Also do not forget the political dimension, e.g. Croatian/Serbian really are just dialects of the same language but they want to keep separate. On the other end of this scheme the variant of Chinese spoken in Beijing may be drastically different from the Chinese spoken in other regions of the country, but still falls under the same "Chinese" umbrella to communicate unity.

How much is spoken? Should we consider people who studied a language as their second, third… language? If so, how well should be the command over the language for the person to count? A1/B1/C2 level? %How do we check accuracy of the available data?

1.1.3 Workflow plan & Project management

- Outline the plan
  - Get, understand and clean data: articles/movie scripts/video transcripts over the years (see next section)
  - Train-test split: keeping proportion of publication years within the splits.
  - Preprocessing: text feature extraction, feature selection, scaling, etc. (Come back here if necessary)
  - Visualization: evolution of words over the years, word-clouds and other relevant characteristics.
  - Define evaluation metrics, train different models/parameters using CV and select best one for predictions.
  - Predict, conclude, report and publish notebook in Kaggle Kernel.

- How the work will be divided up between group members
  - Acquisition, cleaning and prepossessing of the data will be done commonly.
  - Each member of the group will train a model and report results using same evaluation metrics.
  - Jointly choose the best model and conclude.
  - Presentation, report and publishing will be also split.
- Timeline: To be defined after review meeting

1.1.4 Data

Our goal is to get a dataset similar to:

Corpus	**Year Published**	**Type**	...
Text1	1976	News	...
Text2	1976	Movie Script	...
...	...	...	...
TextN	2009	Scientific Article	...

Feature extraction from texts will be performed to obtain appropriate features for modeling. To build a dataset like this one, we will rely on the following kind sources:

- https://www.kaggle.com/asad1m9a9h6mood/news-articles - News articles from 2015 until date.
- https://www.kaggle.com/snapcrack/all-the-news - 143000 articles from 15 American Publications.
- NLTK
- ...

1.2 Second Draft

After having a preliminary meeting with Univ.Prof. Dr. Hanbury and Dipl.-Ing. Dr. Piroi, who gave great input, we decided to further limit out goal to only use Project Gutenberg as a datasource, and setting our hypothesis to see whether it was possible to generate a model that predicted the publication year/decade for a set of books.

1.3 Pivoting Point

After having done a decent portion of work, we reached to the conclusion that our dataset was not suitable to solve the question we had original set out, and we were forced to pivot.

We discussed whether we wanted to change the goal from classifying, but as we were all quite interrested in a classification algorithm, and wanted to do proper NLP, we instead searched for another dataset.

1.4 Language change in Icelandic Parliamentary Speeches

We found the dataset with all icelandic parliamentary speeches going back a century. This is further described in section 3. With this great dataset, our goal was to develop a model that could try to predict which decade a speech is from

2 Estimating publication year from Project Gutenberg

This was the attempt at our first hypothesis. We import a large corpus of books from Project Gutenberg, and cleanse the data, so it's ready for machine learning

2.1 Setup

We start by setting up all packages needed for the project

2.1.1 Import packages

[1]:
```
from __future__ import absolute_import
from builtins import str
import os
from six import u

from os import listdir
from os.path import isfile, join

import nltk
import re
from operator import itemgetter
import pandas as pd
from functools import reduce

import random

pd.set_option('display.max_rows', None)

import math

from sklearn.feature_extraction.text import TfidfTransformer

from pprint import pprint
from time import time
import logging

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

from sklearn.feature_selection import SelectKBest, chi2
```

2.1.2 Define Constants

Constant that are used in this part is also set

[2]:
```
file_path = "processedData"

TEXT_START_MARKERS = frozenset((u(_) for _ in (
    "*END*THE SMALL PRINT",
    "*** START OF THE PROJECT GUTENBERG",
    "*** START OF THIS PROJECT GUTENBERG",
    "This etext was prepared by",
    "E-text prepared by",
    "Produced by",
    "Distributed Proofreading Team",
    "Proofreading Team at http://www.pgdp.net",
    "http://gallica.bnf.fr)",
    "      http://archive.org/details/",
    "http://www.pgdp.net",
    "by The Internet Archive)",
    "by The Internet Archive/Canadian Libraries",
    "by The Internet Archive/American Libraries",
    "public domain material from the Internet Archive",
    "Internet Archive)",
    "Internet Archive/Canadian Libraries",
    "Internet Archive/American Libraries",
    "material from the Google Print project",
    "*END THE SMALL PRINT",
    "***START OF THE PROJECT GUTENBERG",
    "This etext was produced by",
    "*** START OF THE COPYRIGHTED",
    "The Project Gutenberg",
    "http://gutenberg.spiegel.de/ erreichbar.",
    "Project Runeberg publishes",
    "Beginning of this Project Gutenberg",
    "Project Gutenberg Online Distributed",
    "Gutenberg Online Distributed",
    "the Project Gutenberg Online Distributed",
    "Project Gutenberg TEI",
    "This eBook was prepared by",
    "http://gutenberg2000.de erreichbar.",
    "This Etext was prepared by",
    "This Project Gutenberg Etext was prepared by",
    "Gutenberg Distributed Proofreaders",
    "Project Gutenberg Distributed Proofreaders",
    "the Project Gutenberg Online Distributed Proofreading Team",
    "**The Project Gutenberg",
    "*SMALL PRINT!",
    "More information about this book is at the top of this file.",
    "tells you about restrictions in how the file may be used.",
    "l'authorization à les utilizer pour preparer ce texte.",
    "of the etext through OCR.",
    "*****These eBooks Were Prepared By Thousands of Volunteers!*****",
    "We need your donations more than ever!",
    " *** START OF THIS PROJECT GUTENBERG",
    "****     SMALL PRINT!",
    '["Small Print" V.',
    '      (http://www.ibiblio.org/gutenberg/',
    'and the Project Gutenberg Online Distributed Proofreading Team',
    'Mary Meehan, and the Project Gutenberg Online Distributed Proofreading',
    '                this Project Gutenberg edition.',
)))


TEXT_END_MARKERS = frozenset((u(_) for _ in (
    "*** END OF THE PROJECT GUTENBERG",
    "*** END OF THIS PROJECT GUTENBERG",
    "***END OF THE PROJECT GUTENBERG",
    "End of the Project Gutenberg",
    "End of The Project Gutenberg",
    "Ende dieses Project Gutenberg",
    "by Project Gutenberg",
    "End of Project Gutenberg",
    "End of this Project Gutenberg",
    "Ende dieses Projekt Gutenberg",
    "        ***END OF THE PROJECT GUTENBERG",
    "*** END OF THE COPYRIGHTED",
    "End of this is COPYRIGHTED",
    "Ende dieses Etextes ",
    "Ende dieses Project Gutenber",
    "Ende diese Project Gutenberg",
    "**This is a COPYRIGHTED Project Gutenberg Etext, Details Above**",
    "Fin de Project Gutenberg",
    "The Project Gutenberg Etext of ",
    "Ce document fut presente en lecture",
    "Ce document fut présenté en lecture",
    "More information about this book is at the top of this file.",
    "We need your donations more than ever!",
    "END OF PROJECT GUTENBERG",
    " End of the Project Gutenberg",
    " *** END OF THIS PROJECT GUTENBERG",
)))


LEGALESE_START_MARKERS = frozenset((u(_) for _ in (
    "<<THIS ELECTRONIC VERSION OF",
)))


LEGALESE_END_MARKERS = frozenset((u(_) for _ in (
    "SERVICE THAT CHARGES FOR DOWNLOAD",
)))

TITLE_MARKERS = frozenset((u(_) for _ in (
    "Title:",
)))

AUTHOR_MARKERS = frozenset((u(_) for _ in (
    "Author:",
)))
DATE_MARKERS = frozenset((u(_) for _ in (
    "Release Date:","Release Date:"
)))
LANGUAGE_MARKERS = frozenset((u(_) for _ in (
    "Language:",
)))
ENCODING_MARKERS = frozenset((u(_) for _ in (
    "Character set encoding:",
)))
```

2.2 Importing the data

This is a very rough first draft at importing and cleansing the data. Solution is heavily inspired by https://gist.github.com/mbforbes/cee3fd5bb3a797b059524fe8c8ccdc2b

2.2.1 Getting the content

Start by downloading the repository of (english) books. This is done in bash. Only tested on Ubuntu, but mac should work the same

```
wget -m -H -nd "http://www.gutenberg.org/robot/harvest?filetypes[]=txt&langs[]=en"
```

```
                http://www.gutenberg.org/robot/harvest?offset=40532&filetypes[]=txt&langs[]=en
```

Takes a few hours to run, and is stored in a folder called rawContent. This is then copied to another folder, and we can start to clean up the mess

First we delete some dublications of the same books:

```
ls | grep "\-8.zip" | xargs rm
ls | grep "\-0.zip" | xargs rm
```

We can then unzip the files, and remove the zip files

```
unzip "*zip"
rm *.zip
```

Next we take care of some nested foldering

```
mv */*.txt ./
```

And finally, we remove all rubbish that isn't a real book:

```
ls | grep -v "\.txt" | xargs rm -rf
```

2.2.2 Data Cleansing

As the data is not given in a computer-friendly format, a lot of string operations are needed

Read a single file

[3]:
```python
def read_file(file_name):
    file = open(file_name, encoding="ISO-8859-1")
    file_content = file.read()

    lines = file_content.splitlines()
    sep = str(os.linesep)

    # Initialize results for single book
    content_lines = []
    i = 0
    footer_found = False
    ignore_section = False

    title = ""
    author = ""
    date = ""
    language = ""
    encoding = ""
    year = 0

    # Reset flags for each book
    title_found = False
    author_found = False
    date_found = False
    language_found = False
    encoding_found = False

    for line in lines:
            reset = False

            #print(line)
            if i <= 600:
                # Shamelessly stolen
                if any(line.startswith(token) for token in TEXT_START_MARKERS):
                    reset = True
                # Extract Metadata
                if title_found == False:
                    if any(line.startswith(token) for token in TITLE_MARKERS):
                        title_found = True
                        title = line
                if author_found == False:
                    if any(line.startswith(token) for token in AUTHOR_MARKERS):
                        author_found = True
                        author = line
                if date_found == False:
                    if any(line.startswith(token) for token in DATE_MARKERS):
                        date_found = True
                        date = line
                        year = int(re.findall(r'\d{4}', date)[0])
                if language_found == False:
                    if any(line.startswith(token) for token in␣
↪LANGUAGE_MARKERS):
                        language_found = True
                        language = line
                if encoding_found == False:
                    if any(line.startswith(token) for token in␣
↪ENCODING_MARKERS):
                        encoding_found = True
                        encoding = line

                # More theft from above
                if reset:
                    content_lines = []
                    continue

            # I feel like a criminal by now. Guess what? Also stolen
            if i >= 100:
                if any(line.startswith(token) for token in TEXT_END_MARKERS):
                    footer_found = True

                if footer_found:
                    break

            if any(line.startswith(token) for token in LEGALESE_START_MARKERS):
                ignore_section = True
                continue
            elif any(line.startswith(token) for token in LEGALESE_END_MARKERS):
                ignore_section = False
                continue

            if not ignore_section:
                if line != "": # Screw the blank lines
```

```
                    content_lines.append(line.rstrip(sep))
                i += 1

            sep.join(content_lines)

    # Do more cleaning
    for token in TITLE_MARKERS:
        title = title.replace(token, '').lstrip().rstrip()
    for token in AUTHOR_MARKERS:
        author = author.replace(token, '').lstrip().rstrip()
    for token in LANGUAGE_MARKERS:
        language = language.replace(token, '').lstrip().rstrip()
    for token in DATE_MARKERS:
        date = date.replace(token, '').lstrip().rstrip()
    for token in ENCODING_MARKERS:
        encoding = encoding.replace(token, '').lstrip().rstrip()
    return title, author, date, year, language, encoding, content_lines
```

Return list of all words Currently quite an empty function. However, I assume that some cleaning of cases etc. will be done here

[4]:
```
def get_words(content_lines):
    all_text_lower = " ".join(content_lines).lower()
    words = re.findall(r'(\b[A-Za-z][a-z]{2,9}\b)', all_text_lower)

    # Do more cleansing. E.g. cases and stuff

    return words
```

2.3 Statistics

We start by doing some exploratory data analysis, to see how well our scraping works

2.3.1 First attempt

Trying a simple word frequency

[5]:
```
def get_word_frequencies(words):
    frequency = {}
    for word in words:
        count = frequency.get(word,0)
        frequency[word] = count + 1

    word_count = len(words)
    unique_word_count = 0
    word_list = []
    word_list_count = []
    for key, value in reversed(sorted(frequency.items(), key = itemgetter(1))):
        word_list.append(key)
        word_list_count.append(value)
        unique_word_count = unique_word_count + 1

    word_list_freq = [freq / word_count for freq in word_list_count]

    word_freq = pd.DataFrame(list(zip(word_list, word_list_count,␣
↪word_list_freq))
                             , columns = ['Word', 'count', 'freq'])

    word_freq['rank'] = word_freq['count'].rank(ascending = False,␣
↪method="dense")

    return(word_freq, unique_word_count)
```

2.3.2 Read all files, and do preprocessing

Well... Only ten files currently

```
[6]: # Get all filenames
files = [f for f in listdir(file_path) if isfile(join(file_path, f))]
files = list(filter(lambda file: file[0].isdigit(), files))
random.shuffle(files)



# Do only subset
files = files[0:10]

list_of_file = []
list_of_title = []
list_of_author = []
list_of_date = []
list_of_year = []
list_of_language = []
list_of_encoding = []
list_of_word_count = []
list_of_unique_word_count = []
list_of_word_frequencies = []
iter_ = 0

for file in files:
    # Read in basic information from file
    title, author, date, year, language, encoding, content_lines =␣
↪read_file(file_path + "/" + file)
    line_count = len(content_lines)
    # Not sure if we want this for later:
    #content_all = " ".join(content_lines)

    # Split into words (and do various cleaning)
    words = get_words(content_lines)
    word_count = len(words)

    # First analysis, but should do something proper
    word_frequencies_table, unique_word_count = get_word_frequencies(words)

    # Append to results
    list_of_file.append(file)
    list_of_title.append(title)
    list_of_author.append(author)
    list_of_date.append(date)
    list_of_year.append(year)
    list_of_language.append(language)
    list_of_encoding.append(encoding)
    list_of_word_count.append(word_count)
    list_of_unique_word_count.append(unique_word_count)
    list_of_word_frequencies.append(word_frequencies_table)


    # Show basic information
    #print(iter_)
    iter_ = iter_ + 1
    #print("################################")
    #print("################################")
    #print("Filename: " + str(file))
    #print("Title: " + str(title))
    #print("Author(s): " + str(author))
    #print("Date: " + str(date))
    #print("Year: " + str(year))
    #print("Language: " + str(language))
    #print("Encoding: " + str(encoding))
    #print("################################")
    #print("Words in book: " + str(word_count))
    #print("Unique words in book: " + str(unique_word_count))
    #print("################################")
    #print(word_frequencies_table)

# Feel free to change to dict? list? separate files?
## nested dataframes works, but looks super ungly when printing
### Fuck it - This is tooo useless killing it again
#all_res = pd.DataFrame(list(zip(list_of_file
#                                , list_of_title
#                                , list_of_author
#                                , list_of_date
#                                , list_of_language
#                                , list_of_encoding
#                                , list_of_word_count
#                                , list_of_unique_word_count
#                                , list_of_word_frequencies
#                                ))
#                             , columns = ['file'
#                                         , 'title'
#                                         , 'author'
#                                         , 'date'
#                                         , 'language'
#                                         , 'encoding'
#                                         , 'word_count'
#                                         , 'unique_word_count'
#                                         , 'word_frequencies'
#                                        ]
#                       )
```

2.3.3 Compare Word ranking between titles

This is our first attemt at seeing how the ranking of words change between titles. Idea is to see that the zipf-distribution changes as time passes buy

[7]:
```
list_count= []
list_freq = []
list_rank = []



col_names = list_of_title.copy()
col_names.insert(0,'Word')

for df in list_of_word_frequencies:
    list_count.append(df[['Word', 'count']])
    list_freq.append(df[['Word', 'freq']])
    list_rank.append(df[['Word', 'rank']])

df_count = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
 ↪how='outer'), list_count)
df_count.columns = col_names
df_count['Sum'] = df_count.drop('Word', axis=1).apply(lambda x: x.sum(), axis=1)
df_count = df_count.sort_values(ascending = False, by=['Sum'])

df_freq = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
 ↪how='outer'), list_freq)
df_freq.columns = col_names
```

```
df_freq['Avg'] = df_freq.drop('Word', axis=1).apply(lambda x: x.mean(), axis=1)
df_freq = df_freq.sort_values(ascending = False, by=['Avg'])

df_rank = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
 ↪how='outer'), list_rank)
df_rank.columns = col_names
df_rank['Avg'] = df_rank.drop('Word', axis=1).apply(lambda x: x.mean(), axis=1)
df_rank = df_rank.sort_values(by=['Avg'])
```

[8]:
```
df_rank.head(30)
```

[8]:
```
              Word       Tonio, Son of the Sierras  The Modern Railroad  \
0              the   1.0                        1.0                  1.0
1              and   2.0                        2.0                  2.0
17417    gutenberg   NaN                        NaN                  NaN
17418    donations   NaN                        NaN                  NaN
19892         bors   NaN                        NaN                  NaN
2             that   3.0                        6.0                  3.0
5              was   6.0                        3.0                  6.0
4              for   5.0                        8.0                  4.0
3             with   4.0                        7.0                 10.0
17419          org   NaN                        NaN                  NaN
6              his   7.0                        5.0                  7.0
17420       ebooks   NaN                        NaN                  NaN
17421       donate   NaN                        NaN                  NaN
17424        email   NaN                        NaN                  NaN
6510       willett   NaN                       11.0                  NaN
17427       exempt   NaN                        NaN                  NaN
17426        pglaf   NaN                        NaN                  NaN
8              but   9.0                       10.0                 20.0
17450    fairbanks   NaN                        NaN                  NaN
17453          ein   NaN                        NaN                  NaN
17428   newsletter   NaN                        NaN                  NaN
17429     editions   NaN                        NaN                  NaN
17431      concept   NaN                        NaN                  NaN
17432   originator   NaN                        NaN                  NaN
17435    addresses   NaN                        NaN                  NaN
17436     donation   NaN                        NaN                  NaN
17437        swamp   NaN                        NaN                  NaN
17439       donors   NaN                        NaN                  NaN
17452   deductible   NaN                        NaN                  NaN
17440         fees   NaN                        NaN                  NaN

       The Home Mission  The Everlasting Mercy  \
0                   1.0                    1.0
1                   2.0                    2.0
17417               NaN                    NaN
```

```
17418                NaN                      NaN
19892                NaN                      NaN
2                    5.0                      6.0
5                    4.0                      8.0
4                   12.0                      9.0
3                   10.0                      5.0
17419                NaN                      NaN
6                    8.0                      7.0
17420                NaN                      NaN
17421                NaN                      NaN
17424                NaN                      NaN
6510                 NaN                      NaN
17427                NaN                      NaN
17426                NaN                      NaN
8                   13.0                     12.0
17450                NaN                      NaN
17453                NaN                      NaN
17428                NaN                      NaN
17429                NaN                      NaN
17431                NaN                      NaN
17432                NaN                      NaN
17435                NaN                      NaN
17436                NaN                      NaN
17437                NaN                      NaN
17439                NaN                      NaN
17452                NaN                      NaN
17440                NaN                      NaN

       Chambers's Journal of Popular Literature, Science, and Art, No. 725,
November 17, 1877  \
0                                                             1.0
1                                                             2.0
17417                                                         NaN
17418                                                         NaN
19892                                                         NaN
2                                                             3.0
5                                                             4.0
4                                                             5.0
3                                                            10.0
17419                                                         NaN
6                                                             8.0
17420                                                         NaN
17421                                                         NaN
17424                                                         NaN
6510                                                          NaN
17427                                                         NaN
17426                                                         NaN
```

```
8                                              11.0
17450                                           NaN
17453                                           NaN
17428                                           NaN
17429                                           NaN
17431                                           NaN
17432                                           NaN
17435                                           NaN
17436                                           NaN
17437                                           NaN
17439                                           NaN
17452                                           NaN
17440                                           NaN

       Eternal Father, Strong to Save  The Masked Bridal  The Farmer's Boy  \
0                                 1.0                1.0               1.0
1                                 2.0                2.0               2.0
17417                             3.0                NaN               NaN
17418                             4.0                NaN               NaN
19892                             NaN                NaN               NaN
2                                12.0                6.0               5.0
5                                 NaN                7.0              13.0
4                                10.0               10.0               7.0
3                                 7.0                8.0               4.0
17419                             8.0                NaN               NaN
6                                 NaN               11.0               3.0
17420                             9.0                NaN               NaN
17421                            10.0                NaN               NaN
17424                            11.0                NaN               NaN
6510                              NaN                NaN               NaN
17427                            12.0                NaN               NaN
17426                            12.0                NaN               NaN
8                                12.0               14.0              12.0
17450                            13.0                NaN               NaN
17453                            13.0                NaN               NaN
17428                            13.0                NaN               NaN
17429                            13.0                NaN               NaN
17431                            13.0                NaN               NaN
17432                            13.0                NaN               NaN
17435                            13.0                NaN               NaN
17436                            13.0                NaN               NaN
17437                            13.0                NaN               NaN
17439                            13.0                NaN               NaN
17452                            13.0                NaN               NaN
17440                            13.0                NaN               NaN

       Talents, Incorporated        Avg
```

```
0                         1.0   1.000000
1                         2.0   2.000000
17417                     NaN   3.000000
17418                     NaN   4.000000
19892                     5.0   5.000000
2                         4.0   5.300000
5                         3.0   6.000000
4                         7.0   7.700000
3                        15.0   8.000000
17419                     NaN   8.000000
6                        19.0   8.333333
17420                     NaN   9.000000
17421                     NaN  10.000000
17424                     NaN  11.000000
6510                      NaN  11.000000
17427                     NaN  12.000000
17426                     NaN  12.000000
8                         8.0  12.100000
17450                     NaN  13.000000
17453                     NaN  13.000000
17428                     NaN  13.000000
17429                     NaN  13.000000
17431                     NaN  13.000000
17432                     NaN  13.000000
17435                     NaN  13.000000
17436                     NaN  13.000000
17437                     NaN  13.000000
17439                     NaN  13.000000
17452                     NaN  13.000000
17440                     NaN  13.000000
```

[9]:
```
#df_freq['Avg'] = df_freq.drop('Word', axis=1).apply(lambda x: x.mean(), axis=1)
df_freq = df_freq.sort_values(ascending = False, by=['Avg'])

df_freq.head(20)
```

[9]:
```
            Word            Tonio, Son of the Sierras  The Modern Railroad  \
0            the  0.083286                   0.080558             0.109511
1            and  0.052821                   0.052852             0.035380
17417  gutenberg       NaN                        NaN                  NaN
17418  donations       NaN                        NaN                  NaN
19892       bors       NaN                        NaN                  NaN
17419        org       NaN                        NaN                  NaN
2           that  0.010917                   0.014303             0.018102
5            was  0.008057                   0.019219             0.008843
17420     ebooks       NaN                        NaN                  NaN
6            his  0.007784                   0.016740             0.007617
```

```
3           with  0.010321                     0.010634             0.007177
4            for  0.009251                     0.009499             0.012415
11           her  0.005645                     0.005885             0.000913
7            you  0.007212                     0.005899             0.003580
17421     donate       NaN                          NaN                  NaN
33           had  0.002611                     0.017474             0.003669
13           not  0.004949                     0.005677             0.004101
17424      email       NaN                          NaN                  NaN
8            but  0.006416                     0.007269             0.004005
9           from  0.006317                     0.005968             0.007345

       The Home Mission  The Everlasting Mercy  \
0              0.059477               0.068943
1              0.039817               0.066598
17417               NaN                    NaN
17418               NaN                    NaN
19892               NaN                    NaN
17419               NaN                    NaN
2              0.015689               0.010037
5              0.016688               0.008536
17420               NaN                    NaN
6              0.012575               0.008723
3              0.010935               0.010412
4              0.010103               0.007316
11             0.022654               0.003564
7              0.013502               0.013226
17421               NaN                    NaN
33             0.010388               0.003658
13             0.011125               0.003002
17424               NaN                    NaN
8              0.009009               0.006003
9              0.006513               0.006097

       Chambers's Journal of Popular Literature, Science, and Art, No. 725,
November 17, 1877  \
0                                               0.070279
1                                               0.041293
17417                                                NaN
17418                                                NaN
19892                                                NaN
17419                                                NaN
2                                               0.016969
5                                               0.012454
17420                                                NaN
6                                               0.009176
3                                               0.008230
4                                               0.011725
```

```
11                                                0.011652
7                                                 0.010342
17421                                                  NaN
33                                                0.008958
13                                                0.007647
17424                                                  NaN
8                                                 0.008084
9                                                 0.005608

       Eternal Father, Strong to Save  The Masked Bridal  The Farmer's Boy  \
0                            0.056075           0.061594          0.082150
1                            0.042056           0.037046          0.048715
17417                        0.032710                NaN               NaN
17418                        0.028037                NaN               NaN
19892                             NaN                NaN               NaN
17419                        0.014019                NaN               NaN
2                            0.004673           0.019774          0.012615
5                                 NaN           0.016346          0.004923
17420                        0.011682                NaN               NaN
6                                 NaN           0.010248          0.020255
3                            0.016355           0.013447          0.013435
4                            0.009346           0.011631          0.007282
11                                NaN           0.028121          0.004769
7                                 NaN           0.020796          0.000462
17421                        0.009346                NaN               NaN
33                                NaN           0.012124          0.002923
13                           0.014019           0.008419          0.004974
17424                        0.007009                NaN               NaN
8                            0.004673           0.008335          0.004974
9                            0.009346           0.005641          0.009641

       Talents, Incorporated       Avg
0                   0.074133  0.074600
1                   0.029507  0.044609
17417                    NaN  0.032710
17418                    NaN  0.028037
19892               0.014087  0.014087
17419                    NaN  0.014019
2                   0.015294  0.013837
5                   0.019294  0.012707
17420                    NaN  0.011682
6                   0.006389  0.011057
3                   0.007547  0.010849
4                   0.011295  0.009986
11                  0.001811  0.009446
7                   0.009458  0.009386
17421                    NaN  0.009346
```

```
33                    0.008729  0.007837
13                    0.008175  0.007209
17424                      NaN  0.007009
8                     0.010414  0.006918
9                     0.006113  0.006859
```

[10]:
```
df_count.head(20)
```

[10]:
```
      Word          Tonio, Son of the Sierras  The Modern Railroad  \
0      the  3349.0                     5818.0              13672.0
1      and  2124.0                     3817.0               4417.0
2     that   439.0                     1033.0               2260.0
5      was   324.0                     1388.0               1104.0
4      for   372.0                      686.0               1550.0
6      his   313.0                     1209.0                951.0
3     with   415.0                      768.0                896.0
11     her   227.0                      425.0                114.0
7      you   290.0                      426.0                447.0
33     had   105.0                     1262.0                458.0
16     she   164.0                      258.0                 42.0
8      but   258.0                      525.0                500.0
9     from   254.0                      431.0                917.0
13     not   199.0                      410.0                512.0
19    have   158.0                      311.0                585.0
21    this   151.0                      260.0                658.0
22    they   144.0                      430.0                597.0
15   there   175.0                      398.0                607.0
10     all   240.0                      392.0                457.0
44     him    84.0                      575.0                281.0

    The Home Mission  The Everlasting Mercy  \
0             2502.0                  735.0
1             1675.0                  710.0
2              660.0                  107.0
5              702.0                   91.0
4              425.0                   78.0
6              529.0                   93.0
3              460.0                  111.0
11             953.0                   38.0
7              568.0                  141.0
33             437.0                   39.0
16             649.0                   40.0
8              379.0                   64.0
9              274.0                   65.0
13             468.0                   32.0
19             243.0                   23.0
21             328.0                   32.0
```

```
22             177.0                    73.0
15             172.0                    49.0
10             209.0                   124.0
44             257.0                    53.0

    Chambers's Journal of Popular Literature, Science, and Art, No. 725,
November 17, 1877  \
0                                                   965.0
1                                                   567.0
2                                                   233.0
5                                                   171.0
4                                                   161.0
6                                                   126.0
3                                                   113.0
11                                                  160.0
7                                                   142.0
33                                                  123.0
16                                                  102.0
8                                                   111.0
9                                                    77.0
13                                                  105.0
19                                                  105.0
21                                                   83.0
22                                                   86.0
15                                                   49.0
10                                                   67.0
44                                                   64.0

    Eternal Father, Strong to Save  The Masked Bridal  The Farmer's Boy  \
0                             24.0             5121.0            1602.0
1                             18.0             3080.0             950.0
2                              2.0             1644.0             246.0
5                              NaN             1359.0              96.0
4                              4.0              967.0             142.0
6                              NaN              852.0             395.0
3                              7.0             1118.0             262.0
11                             NaN             2338.0              93.0
7                              NaN             1729.0               9.0
33                             NaN             1008.0              57.0
16                             NaN             1903.0              32.0
8                              2.0              693.0              97.0
9                              4.0              469.0             188.0
13                             6.0              700.0              97.0
19                             2.0              772.0              98.0
21                             1.0              434.0             130.0
22                             NaN              193.0              71.0
15                             NaN              260.0              38.0
```

```
10                             2.0             312.0              76.0
44                             NaN             517.0              82.0

    Talents, Incorporated       Sum
0                  2947.0   36735.0
1                  1173.0   18531.0
2                   608.0    7232.0
5                   767.0    6002.0
4                   449.0    4834.0
6                   254.0    4722.0
3                   300.0    4450.0
11                   72.0    4420.0
7                   376.0    4128.0
33                  347.0    3836.0
16                   97.0    3287.0
8                   414.0    3043.0
9                   243.0    2922.0
13                  325.0    2854.0
19                  214.0    2511.0
21                  119.0    2196.0
22                  397.0    2168.0
15                  394.0    2142.0
10                  169.0    2048.0
44                  122.0    2035.0
```

2.4 Second testing

This definately needs some proper refactoring, but Was curious whether we get anything decent from reading a bunch of random books in

Requires an additional folder "decades" in the root directory

[11]:
```
# Get all filenames
files = [f for f in listdir(file_path) if isfile(join(file_path, f))]

# Do only subset
## Is done for 5000 files already, so set down to 20 to increase performance.␣
↪5000 books are currently stored in the file
files = files[0:20]


counter = 0
for file in files:
    counter = counter + 1
    # Read in basic information from file
    title, author, date, year, language, encoding, content_lines =␣
↪read_file(file_path + "/" + file)
    #line_count = len(content_lines)
    decade = math.floor(year / 10) * 10
    decade_file = "decades/" + str(decade) + ".txt"
    content_all = " ".join(content_lines)

    if os.path.exists(decade_file):
        append_write = 'a' # append if already exists
    else:
        append_write = 'w' # make a new file if not

    fileWriter = open(decade_file,append_write)
    fileWriter.write(content_all + '\n')
    fileWriter.close()
```

2.4.1 Read in from the decades files, and see the distributions

[12]:
```
# Get all filenames
files = [f for f in listdir("decades") if isfile(join("decades", f))]
print(files)
files.sort(reverse=True)


col_names = []
col_names.append("Word")

tables = []

for file_name in files:
    print(file_name)

    file = open("decades/" + file_name, encoding="ISO-8859-1")
    file_content = file.read()

    # Split into words (and do various cleaning)
    all_text_lower = file_content.lower()
    words = re.findall(r'(\b[A-Za-z][a-z]{2,9}\b)', all_text_lower)

    # First analysis, but should do something proper
    word_frequencies_table, unique_word_count = get_word_frequencies(words)
    tables.append(word_frequencies_table)
    col_names.append(file_name)
```

```
['00.txt', '0.txt', '2010.txt', '2000.txt', '2020.txt', '1990.txt']
2020.txt
2010.txt
2000.txt
1990.txt
00.txt
```

```
0.txt
```

2.4.2 Preliminary Conclusion

We see that even though the books are quite old, no decade prior to 1990s is found.

This is when we found out that the "year" that's registered in the dataset is the upload-date.

Haven gotten this far, we however decided to see if we could find a pattern in this

2.4.3 Compare ranking between upload-decades

```
[13]: list_count= []
      list_freq = []
      list_rank = []

      for df in tables:
          #list_count.append(df[['Word', 'count']])
          #list_freq.append(df[['Word', 'freq']])
          list_rank.append(df[['Word', 'rank']])

      #df_count = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
       ↪how='outer'), list_count)
      #df_count.columns = col_names

      #df_freq = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
       ↪how='outer'), list_freq)
      #df_freq.columns = col_names

      df_rank = reduce(lambda left, right: pd.merge(left, right, on="Word",␣
       ↪how='outer'), list_rank)
      df_rank.columns = col_names
```

```
[14]: df_rank.head(100)
```

```
[14]:          Word  2020.txt  2010.txt  2000.txt  1990.txt  00.txt  0.txt
      0         the       1.0       1.0       1.0       1.0     1.0    1.0
      1         and       2.0       2.0       2.0       2.0     2.0    2.0
      2        that       3.0       3.0       3.0       3.0     4.0    3.0
      3         was       4.0       4.0       4.0       4.0    23.0    5.0
      4         you       5.0      10.0       8.0       5.0   163.0   19.0
      5        with       6.0       6.0       6.0       7.0     3.0    6.0
      6         for       7.0       7.0       7.0      11.0    13.0    8.0
      7         his       8.0       5.0       5.0       6.0     5.0    4.0
      8         not       9.0       8.0      11.0      12.0    11.0    9.0
      9         had      10.0       9.0       9.0      10.0    40.0   14.0
      10        but      11.0      11.0      10.0      13.0    72.0   10.0
      11      which      12.0      12.0      15.0      22.0    82.0    7.0
```

12	they	13.0	17.0	14.0	16.0	58.0	21.0
13	from	14.0	14.0	18.0	21.0	25.0	15.0
14	were	15.0	20.0	21.0	20.0	81.0	22.0
15	have	16.0	16.0	16.0	18.0	22.0	11.0
16	this	17.0	13.0	17.0	15.0	8.0	13.0
17	are	18.0	18.0	24.0	27.0	37.0	16.0
18	she	19.0	21.0	13.0	9.0	65.0	47.0
19	all	20.0	19.0	20.0	17.0	15.0	12.0
20	their	21.0	24.0	25.0	26.0	21.0	30.0
21	him	22.0	22.0	19.0	14.0	28.0	20.0
22	her	23.0	15.0	12.0	8.0	20.0	33.0
23	its	24.0	40.0	55.0	98.0	68.0	67.0
24	one	25.0	23.0	22.0	25.0	17.0	28.0
25	there	26.0	25.0	23.0	23.0	37.0	32.0
26	them	27.0	28.0	28.0	32.0	55.0	38.0
27	what	28.0	33.0	32.0	24.0	56.0	24.0
28	has	29.0	36.0	46.0	46.0	176.0	40.0
29	been	30.0	29.0	31.0	33.0	30.0	23.0
30	will	31.0	31.0	33.0	37.0	85.0	27.0
31	would	32.0	30.0	29.0	31.0	194.0	25.0
32	said	33.0	32.0	26.0	19.0	61.0	141.0
33	when	34.0	27.0	27.0	28.0	105.0	34.0
34	more	35.0	34.0	36.0	42.0	190.0	26.0
35	who	36.0	26.0	30.0	30.0	27.0	18.0
36	into	37.0	37.0	37.0	39.0	97.0	63.0
37	out	38.0	35.0	34.0	29.0	62.0	77.0
38	then	39.0	38.0	35.0	35.0	47.0	50.0
39	other	40.0	44.0	56.0	58.0	88.0	66.0
40	men	41.0	72.0	78.0	81.0	108.0	53.0
41	only	42.0	48.0	53.0	60.0	192.0	71.0
42	can	43.0	59.0	48.0	45.0	98.0	54.0
43	upon	44.0	51.0	57.0	80.0	121.0	89.0
44	our	45.0	55.0	49.0	91.0	149.0	43.0
45	than	46.0	47.0	50.0	64.0	201.0	31.0
46	now	47.0	43.0	38.0	38.0	134.0	42.0
47	time	48.0	42.0	43.0	49.0	96.0	57.0
48	power	49.0	241.0	257.0	327.0	204.0	111.0
49	great	50.0	62.0	63.0	79.0	64.0	37.0
50	these	51.0	49.0	67.0	87.0	48.0	60.0
51	government	52.0	379.0	440.0	534.0	NaN	146.0
52	man	53.0	41.0	41.0	34.0	222.0	29.0
53	over	54.0	67.0	64.0	62.0	92.0	139.0
54	could	55.0	45.0	40.0	41.0	199.0	64.0
55	very	56.0	46.0	42.0	50.0	159.0	65.0
56	your	57.0	57.0	45.0	36.0	201.0	55.0
57	first	58.0	64.0	76.0	77.0	219.0	81.0
58	society	59.0	579.0	603.0	535.0	NaN	231.0

```
59         two      60.0     52.0     59.0     70.0  124.0  100.0
60        made      61.0     61.0     62.0     83.0  174.0   82.0
61        such      62.0     63.0     75.0     78.0  146.0   45.0
62       about      63.0     53.0     44.0     40.0   99.0  113.0
63        some      64.0     39.0     39.0     44.0   70.0   49.0
64         any      65.0     54.0     54.0     55.0  130.0   49.0
65         did      66.0     60.0     52.0     54.0  154.0   99.0
66        know      67.0     83.0     74.0     48.0  159.0   98.0
67   pendleton      68.0   3542.0   3792.0    773.0    NaN    NaN
68        same      69.0     97.0    125.0    118.0  203.0  112.0
69        well      70.0     68.0     58.0     52.0   54.0   58.0
70       under      71.0    103.0    121.0    153.0  129.0  103.0
71         may      72.0     50.0     70.0     86.0  162.0   36.0
72     general      73.0    196.0    225.0    374.0  254.0  195.0
73      before      74.0     66.0     65.0     59.0  101.0   84.0
74        most      75.0     80.0     88.0    120.0  243.0   46.0
75        even      76.0     88.0    101.0    125.0   77.0   78.0
76        much      77.0     77.0     77.0     76.0  228.0   73.0
77        like      78.0     65.0     51.0     47.0   10.0   74.0
78  stephanie      79.0      NaN   3771.0      NaN    NaN    NaN
79    lorraine      80.0   3413.0   3652.0    766.0    NaN  383.0
80       those      81.0     79.0     91.0    138.0   26.0   44.0
81        down      82.0     76.0     68.0     63.0   63.0  126.0
82        back      83.0     98.0     86.0     85.0  165.0  217.0
83        came      84.0     93.0     85.0     74.0  172.0  236.0
84         see      85.0     69.0     60.0     51.0  150.0   88.0
85         how      86.0     84.0     73.0     57.0  152.0   56.0
86         way      87.0     87.0     82.0     69.0  118.0  145.0
87       think      88.0    125.0    103.0     82.0  227.0  114.0
88      little      89.0     56.0     47.0     43.0  229.0   80.0
89     without      90.0    107.0    107.0    107.0  105.0   79.0
90        here      91.0     82.0     81.0     68.0   89.0   93.0
91     against      92.0    134.0    144.0    171.0  220.0   92.0
92      people      93.0    111.0    116.0    121.0  203.0  124.0
93       after      94.0     58.0     61.0     53.0  103.0   95.0
94        must      95.0     71.0     79.0     88.0  213.0   69.0
95         don      95.0    148.0    102.0     56.0    NaN  274.0
96       where      96.0     75.0     71.0     84.0  151.0  114.0
97       never      97.0     90.0     83.0     75.0  172.0   83.0
98         own      98.0     92.0     92.0     96.0  111.0   68.0
99       right      99.0    133.0    135.0    117.0  236.0  161.0
```

2.5 Trying to fit models to predict

2.5.1 Read in files

[15]:
```
file_contents = []
targets = []

files = [f for f in listdir(file_path) if isfile(join(file_path, f))]
files = list(filter(lambda file: file[0].isdigit(), files))
random.shuffle(files)

targets_=['70','80','90','00','10']
iter_ = 0

for f in files[:120]:
    file = open("processedData/" + f, encoding="ISO-8859-1")
    file_contents.append(file.read())
    iter_ = iter_+1
    targets.append(targets_[iter_%5])
```

2.5.2 Train models

[16]:
```
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('kbest', SelectKBest(chi2, k=100)),
    ('nb', MultinomialNB()),
])

parameters = {
    #'vect__max_df': [1.0),
    # 'vect__max_features': (None, 5000, 10000, 50000),
    #'vect__ngram_range': ((1, 1), (1, 2)),  # unigrams or bigrams
    # 'tfidf__use_idf': (True, False),
    # 'tfidf__norm': ('l1', 'l2'),
    #'clf__max_iter': (20),
    #'clf__alpha': (0.00001),
    #'clf__penalty': ('l2'),
    # 'clf__max_iter': (10, 50, 80),
}

grid_search = GridSearchCV(pipeline, parameters, verbose=1)

grid_search.fit(file_contents, targets)
best_parameters = grid_search.best_estimator_.get_params()

for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   5 out of   5 | elapsed:   36.4s finished
```

2.6 Realisation and conclusion

At this point, we came to the conclusion that "year" in the Gutenberg dataset shows when the data **was published** to the project, and not the release date of the book.

We searched for possible solutions to get the years for book publications, but were unable to find any free API that we could link to our current dataset.

We therefore went on a search for other datasets, and to remake our hypothesis entirely. Thus, this part ended in a blind spot. However science is not only about the results, but also about the discoveries along the way, and therefore it is added into this file.

3 Studying language change in Icelandic parliamentary speeches

Our task involves research into **language change over the past 100 years**. Additionally we have been tasked with working out factors that influence language change.

Another proposed research question could have been focused on figuring out which languages are going extinct. This particular task has been found out to be near impossible to answer given the available data. It is estimated to be very hard to come up with data that capture the amount of speakers for a large enough ranges of combinations of language and year. Furthermore, any data that are available are likely to apply a different definition of "speaker" (sometimes including second/third… language speakers, sometimes not) and is also likely to contain politically motivated noise.

3.1 Introduction

Therefore, we decided to search for English language corpora containing a wide array of text documents collected over the past century for predefined dialects of English and genre of text (movie, articles, books, …). This surprisingly turned out to be a complex endeavour as all high quality corpora were available only for a big price tag.

We also looked into the material provided by the Guttenberg Project Link. This turned out to be promising at first sight as it appears that there is a lot of recently published material. However release date of these documents does not match the year when the documents were actually written and soon enough we figured out that all material is from before 1923. This obviously did not allow us to look much into language change of the 20th and 21st century.

Gerlach, M., & Font-Clos, F. (2020). A standardized Project Gutenberg corpus for statistical analysis of natural language and quantitative linguistics. Entropy, 22(1), 126.

Theoretically one could obtain books from after 1923 and include them into the analysis. But one would quickly run into copyright/licensing issues here.

Obtaining the content of these books and preprocessing them for the purposes of data analysis turned out to be quite cumbersome as well. Look at Gunnar's notebooks (first draft here, second draft here) for the details.

Finally we turned to looking for non-English corpora and **found an annotated corpus including pre-factured lemmatization of Icelandic parlimentary speeches from 1911 until 2018:**

Steingrímsson, Steinþór, Sigrún Helgadóttir, Eiríkur Rögnvaldsson, Starkaður Barkarson and Jón Guðnason. 2018. Risamálheild: A Very Large Icelandic Text Corpus. Proceedings of LREC 2018, pp. 4361-4366. Myazaki, Japan.

3.2 The task

In the line with our goal of analyzing the change in language over the past 100 years, we decided to train different models and assess their ability to predict whether an speech held in the Icelandic Parlament belongs to a particular decade. In the end, this is a **document classification task** in which the input is a large set of parlament speeches and the target/class is the decade in which the speeches were held.

A good performance of our proposed classifiers may support the idea that Icelandic has envolved in the years. However, the fact that the models would perform well is not enough to assert that the language has changed. It could be that what has actually changed are the topics or even the way of documenting the speeches. Anyway, for us it was really exiciting to check whether we are able to **fit a model that predicts reasonably well the decade of an speech by only using the speech itself.**

3.3 Setup

In this section, we provide the **setup for a successful** implementation (or replication) of our experiment within this Jupyter Notebook.

3.3.1 Load required libraries

The following libraries are used during the next sections and therefore need to be imported.

[17]:
```
import pandas as pd
import numpy as np
import xml.etree.ElementTree as ET
import glob
from nltk.probability import FreqDist
import random
from functools import reduce
from nltk import ngrams
# Used for building models for classifying:
from pprint import pprint
from time import time
import logging
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import confusion_matrix
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
from gensim.models import Word2Vec
from gensim.models import Doc2Vec
from gensim.models.doc2vec import TaggedDocument
from sklearn.metrics import plot_confusion_matrix
```

[18]:
```
#needed afterwards too
namespace = "{http://www.tei-c.org/ns/1.0}"
```

3.3.2 Get the data

Data can be downloaded from here: http://www.malfong.is/index.php?dlid=81&lang=en. However, we provided already in our submission file the specifications on how to get the data of our assignment.

Then extract zip folder such that a folder labelled `CC_BY` shows up in the parent folder of this notebook. *Test*: `ls ../CC_BY/althingi` should work when run from `.../IcelandicParliamentSpeeches.ipynb`.

3.3.3 Preprocessing helpers

The data are available as XML. The text has already been preprocessed to be separated into paragraphs, sentences and words. Furthermore each word tag also includes a `lemma` attribute relating inflected/declensed forms of words to its lemma. This has been done by the authors of the original paper using Machine Learning approaches.

Given a relative path to a file, pull out a list with all the words. This can be achieved by looking for all tags of type `w`, additionally also retrieve the lemma for each word.

We will discard all sentences of length 3 or smaller to remove noise and to avoid that our models are able to detect year of speech just based on some short introductory/outro phrases. Furthermore the raw data appear to contain plenty of elements tagged as words that comprise of just a single letter followed by a dot. These will be removed here as well.

Pitfall: The namespace from above must be included when parsing out content from these XML files based on tag names.

In this kind of preprocessing we lose information about sentence boundaries as all punctuation items from the raw data are dropped.

```
[19]: def extract_words(path):
          xml_tree = ET.parse(open(path, 'r', encoding="utf8"))
          words = []
          lemmata = []

          for sentence in xml_tree.getroot().iter('{}s'.format(namespace)):
              words_in_sent = sentence.findall('{}w'.format(namespace))
              if len(words_in_sent) > 2:
                  for word in sentence.findall('{}w'.format(namespace)):
                      if not word.text.endswith('.'):
                          words.append(word.text)
                          lemmata.append(word.attrib['lemma'])

          return words, lemmata
```

Extract content of files separated into sentences, note that all stop items are wrapped in a `p` tag in the original documents and are not included here.

Also note that some further pre-processing could be done here to exclude items such as numbers, percentages, names, abbreviations, etc. In the original documents these are also assigned to be words:

```
[20]: def extract_sentences(path, lemma=False):
          xml_tree = ET.parse(open(path, 'r', encoding="utf8"))
          sentences = []

          for sentence in xml_tree.getroot().iter('{}s'.format(namespace)):
              sentence_cur = []
              words_in_sent = sentence.findall('{}w'.format(namespace))

              if len(words_in_sent) > 2:
                  for word in sentence.findall('{}w'.format(namespace)):
                      if not word.text.endswith('.'):
                          if lemma:
                              sentence_cur.append(word.attrib['lemma'])
                          else:
                              sentence_cur.append(word.text)

                  sentences.append(sentence_cur)

          return sentences
```

Retrieve a random selection of `k` file names from the entire corpus. The files must be of type `xml`. This method does not load the entire corpus into memory and allows you to work with smaller selections for test purposes. This method samples only from the `althingi` folder so far:

```
[21]: def get_random_sample(k):
          files = [filename for filename in glob.iglob('../CC_BY/althingi/**/*.xml',
                                                        recursive=True)]
          return random.sample(files, k)
```

```
[22]: files = [filename for filename in glob.iglob('../CC_BY/althingi/**/*.xml',␣
       ↪recursive=True)]
      #print(files)
```

Do the same as above but choose `k` files only from a given year (range: 1911-2017)

```
[23]: def get_files_for_year(year, k = None):
          files = [filename for filename in glob.iglob('../CC_BY/althingi/{}/'.
       ↪format(year) + '**/*.xml',
                                                        recursive=True)]
          if k == None:
              newK = len(files)
              res = files
          else:
              newK = k
              res =  random.sample(files, min(len(files), k))
          if len(files) != 0:
              percentage = 100*newK/len(files)
          else:
              percentage = 0
          print("For year " + str(year) + ": Fetching " + str(newK) +" samples out of␣
       ↪" + str(len(files)) + " (~" + str(percentage) + "%)")
          return res
```

3.4 Preliminary Data Analysis

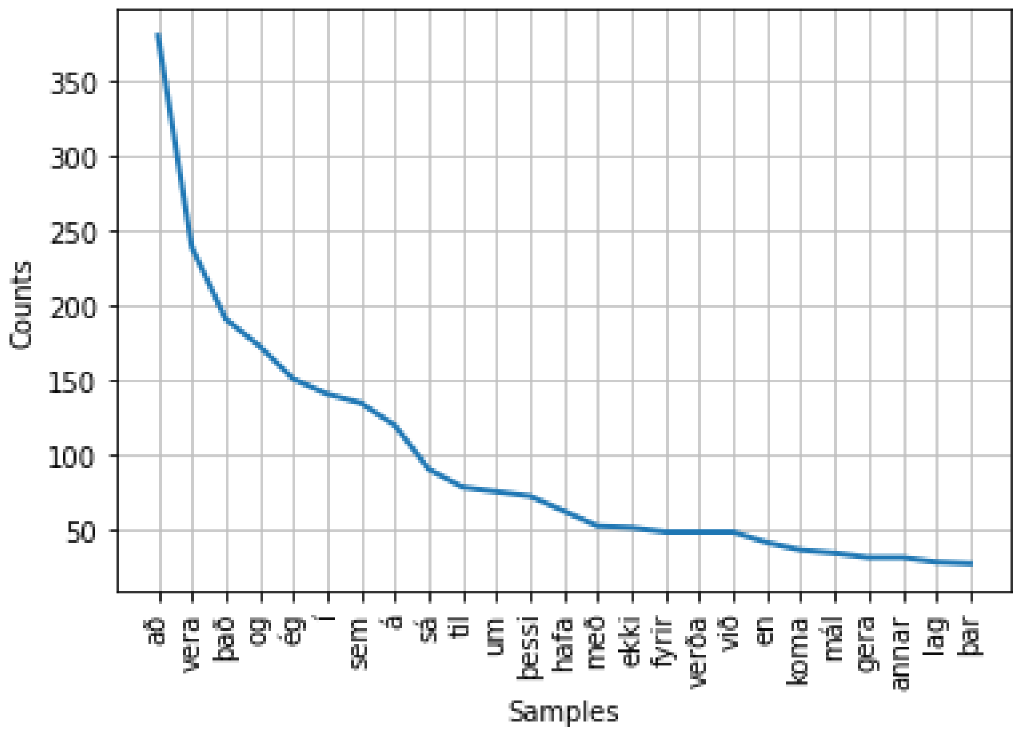
In this section, we perform a preliminary data analysis to get a better insight of our data.

3.4.1 Zipf's Law

First using frequency distributions of the Natural Language ToolKit (`NLTK`) to look into whether or not we can confirm Zipf's Law based on the data we have.

Note that the analysis is done based on 15 randomly selected files from the entire corpus at this point:

```
[24]: words = []

      for file in get_random_sample(15):
          words.extend(extract_words(file)[1])

      fq = FreqDist(word.lower() for word in words)
      fq.plot(25, cumulative=False)
```

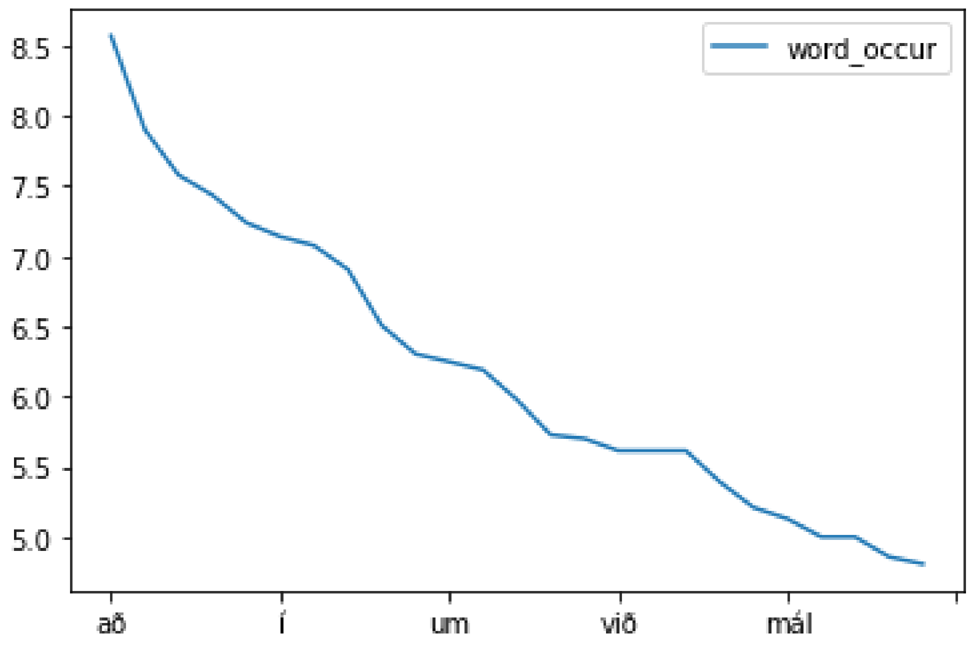


```
[24]: <AxesSubplot:xlabel='Samples', ylabel='Counts'>
```

Visualizing the same data but with using the logarithm of the occurrences, this should ideally obtain a straight line:

```
[25]: freq_df = pd.DataFrame.from_dict(fq, orient='index', columns=['word_occur'])
      freq_df.sort_values(by='word_occur', inplace=True, ascending=False)
      freq_df.word_occur = np.log2(freq_df['word_occur'])
      freq_df.head(25).plot(kind='line')
```

```
[25]: <AxesSubplot:>
```



3.4.2 Disappearing words / new words

Here is a description

[26]:
```
words_1914 = []
words_2014 = []

for file in get_files_for_year(1914, 25):
    words_1914.extend(extract_words(file)[1])

for file in get_files_for_year(2014, 25):
    words_2014.extend(extract_words(file)[1])
```

```
For year 1914: Fetching 25 samples out of 1306 (~1.9142419601837672%)
For year 2014: Fetching 25 samples out of 12404 (~0.2015478877781361%)
```

3.4.3 Development of average sentence length

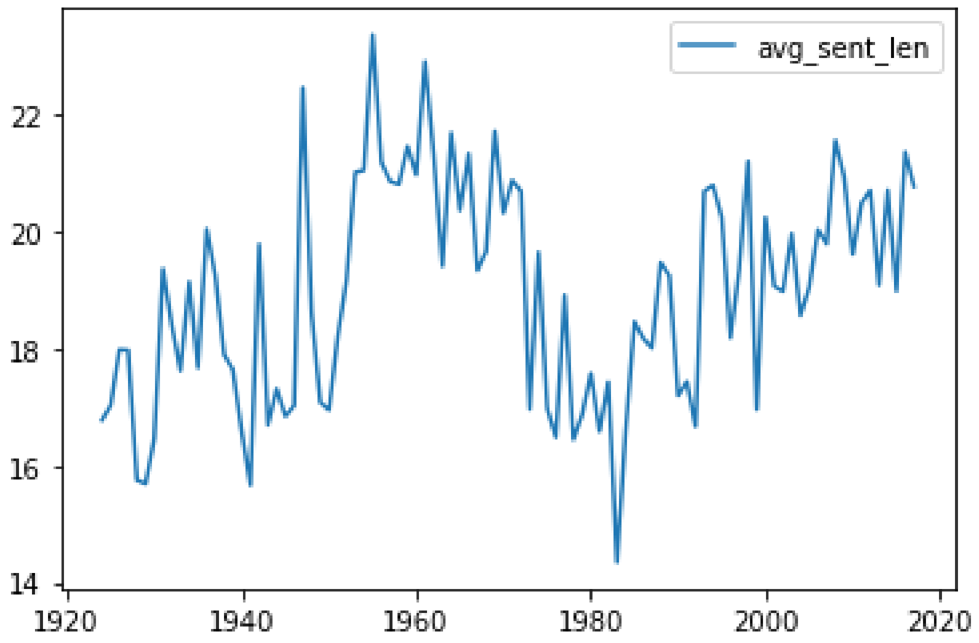
This is just one possible metric for the development/analysis of language complexity. There is so much more you could come up with here.

Obviously our choice to discard very short sentences in the preprocessing step has an impact on the values here:

[27]:
```
def avg_sentence_length_year(year, k):
    sentence_len = []
    for file in get_files_for_year(year, k):
        sentences = extract_sentences(file)
        sentence_len.extend([len(s) for s in sentences])

    return reduce(lambda a, b: a + b, sentence_len) / len(sentence_len)

sentence_len_years = []

for year in range(1924, 2018):
    sentence_len_years.append(avg_sentence_length_year(year, 20))

avg_df = pd.DataFrame(sentence_len_years, index=range(1924, 2018),␣
 ↪columns=['avg_sent_len'])
avg_df.plot(kind='line')
```

```
For year 1924: Fetching 20 samples out of 41 (~48.78048780487805%)
For year 1925: Fetching 20 samples out of 23 (~86.95652173913044%)
For year 1926: Fetching 20 samples out of 37 (~54.054054054054056%)
For year 1927: Fetching 20 samples out of 59 (~33.898305084745765%)
For year 1928: Fetching 20 samples out of 21 (~95.23809523809524%)
For year 1929: Fetching 20 samples out of 96 (~20.833333333333332%)
For year 1930: Fetching 20 samples out of 30 (~66.66666666666667%)
For year 1931: Fetching 20 samples out of 25 (~80.0%)
For year 1932: Fetching 20 samples out of 28 (~71.42857142857143%)
For year 1933: Fetching 20 samples out of 52 (~38.46153846153846%)
For year 1934: Fetching 20 samples out of 20 (~100.0%)
For year 1935: Fetching 20 samples out of 24 (~83.33333333333333%)
For year 1936: Fetching 20 samples out of 25 (~80.0%)
For year 1937: Fetching 20 samples out of 1381 (~1.448225923244026%)
For year 1938: Fetching 20 samples out of 1676 (~1.1933174224343674%)
For year 1939: Fetching 20 samples out of 1632 (~1.2254901960784315%)
For year 1940: Fetching 20 samples out of 1458 (~1.3717421124828533%)
For year 1941: Fetching 20 samples out of 2066 (~0.968054211035818%)
For year 1942: Fetching 20 samples out of 2357 (~0.8485362749257531%)
For year 1943: Fetching 20 samples out of 3960 (~0.5050505050505051%)
For year 1944: Fetching 20 samples out of 1072 (~1.8656716417910448%)
For year 1945: Fetching 20 samples out of 1859 (~1.0758472296933836%)
For year 1946: Fetching 20 samples out of 2789 (~0.7171029042667623%)
For year 1947: Fetching 20 samples out of 2838 (~0.704721634954193%)
For year 1948: Fetching 20 samples out of 2262 (~0.8841732979664014%)
For year 1949: Fetching 20 samples out of 2661 (~0.7515971439308531%)
For year 1950: Fetching 20 samples out of 3017 (~0.6629101756711966%)
For year 1951: Fetching 20 samples out of 2453 (~0.8153281695882593%)
For year 1952: Fetching 20 samples out of 1689 (~1.1841326228537596%)
For year 1953: Fetching 20 samples out of 1415 (~1.4134275618374559%)
For year 1954: Fetching 20 samples out of 1546 (~1.2936610608020698%)
For year 1955: Fetching 20 samples out of 1433 (~1.3956734124214933%)
For year 1956: Fetching 20 samples out of 1213 (~1.6488046166529267%)
For year 1957: Fetching 20 samples out of 1763 (~1.1344299489506522%)
For year 1958: Fetching 20 samples out of 1169 (~1.7108639863130881%)
For year 1959: Fetching 20 samples out of 1326 (~1.5082956259426847%)
For year 1960: Fetching 20 samples out of 1862 (~1.0741138560687433%)
For year 1961: Fetching 20 samples out of 1409 (~1.4194464158978%)
For year 1962: Fetching 20 samples out of 1475 (~1.3559322033898304%)
For year 1963: Fetching 20 samples out of 1376 (~1.4534883720930232%)
For year 1964: Fetching 20 samples out of 1677 (~1.1926058437686344%)
For year 1965: Fetching 20 samples out of 1520 (~1.3157894736842106%)
For year 1966: Fetching 20 samples out of 1404 (~1.4245014245014245%)
For year 1967: Fetching 20 samples out of 1274 (~1.5698587127158556%)
For year 1968: Fetching 20 samples out of 1726 (~1.1587485515643106%)
For year 1969: Fetching 20 samples out of 1740 (~1.1494252873563218%)
For year 1970: Fetching 20 samples out of 1921 (~1.041124414367517%)
For year 1971: Fetching 20 samples out of 2071 (~0.9657170449058425%)
For year 1972: Fetching 20 samples out of 2368 (~0.8445945945945946%)
For year 1973: Fetching 20 samples out of 2329 (~0.8587376556462001%)
For year 1974: Fetching 20 samples out of 2350 (~0.851063829787234%)
For year 1975: Fetching 20 samples out of 2430 (~0.823045267489712%)
For year 1976: Fetching 20 samples out of 2555 (~0.7827788649706457%)
For year 1977: Fetching 20 samples out of 2105 (~0.9501187648456056%)
For year 1978: Fetching 20 samples out of 2641 (~0.7572889057175313%)
For year 1979: Fetching 20 samples out of 2165 (~0.9237875288683602%)
For year 1980: Fetching 20 samples out of 3406 (~0.5871990604815033%)
For year 1981: Fetching 20 samples out of 3491 (~0.5729017473503294%)
For year 1982: Fetching 20 samples out of 2894 (~0.691085003455425%)
For year 1983: Fetching 20 samples out of 2651 (~0.7544322897019993%)
For year 1984: Fetching 20 samples out of 4171 (~0.4795013186286262%)
For year 1985: Fetching 20 samples out of 4709 (~0.42471862391165854%)
For year 1986: Fetching 20 samples out of 3432 (~0.5827505827505828%)
For year 1987: Fetching 20 samples out of 3212 (~0.6226650062266501%)
For year 1988: Fetching 20 samples out of 4198 (~0.47641734159123394%)
For year 1989: Fetching 20 samples out of 5085 (~0.39331366764995085%)
For year 1990: Fetching 20 samples out of 4662 (~0.429000429000429%)
For year 1991: Fetching 20 samples out of 4747 (~0.4213187276174426%)
For year 1992: Fetching 20 samples out of 8925 (~0.22408963585434175%)
For year 1993: Fetching 20 samples out of 7412 (~0.26983270372369134%)
For year 1994: Fetching 20 samples out of 8187 (~0.2442897276169537%)
For year 1995: Fetching 20 samples out of 5129 (~0.38993955936829794%)
For year 1996: Fetching 20 samples out of 7184 (~0.27839643652561247%)
For year 1997: Fetching 20 samples out of 6960 (~0.28735632183908044%)
For year 1998: Fetching 20 samples out of 7393 (~0.27052617340727714%)
For year 1999: Fetching 20 samples out of 6056 (~0.33025099075297226%)
For year 2000: Fetching 20 samples out of 7466 (~0.2678810608090008%)
For year 2001: Fetching 20 samples out of 8210 (~0.243605359317905%)
For year 2002: Fetching 20 samples out of 8061 (~0.24810817516437167%)
For year 2003: Fetching 20 samples out of 5872 (~0.3405994550408719%)
For year 2004: Fetching 20 samples out of 9466 (~0.21128248468201985%)
For year 2005: Fetching 20 samples out of 8269 (~0.24186721489902044%)
For year 2006: Fetching 20 samples out of 8810 (~0.22701475595913734%)
For year 2007: Fetching 20 samples out of 7863 (~0.2543558438255119%)
For year 2008: Fetching 20 samples out of 8764 (~0.22820629849383842%)
For year 2009: Fetching 20 samples out of 17262 (~0.11586142973004288%)
For year 2010: Fetching 20 samples out of 11089 (~0.18035891423933628%)
For year 2011: Fetching 20 samples out of 13957 (~0.14329727018700295%)
For year 2012: Fetching 20 samples out of 16356 (~0.12227928588897041%)
For year 2013: Fetching 20 samples out of 10240 (~0.1953125%)
For year 2014: Fetching 20 samples out of 12404 (~0.16123831022250887%)
For year 2015: Fetching 20 samples out of 18052 (~0.11079104808331487%)
For year 2016: Fetching 20 samples out of 8165 (~0.2449479485609308%)
For year 2017: Fetching 20 samples out of 7270 (~0.2751031636863824%)
```

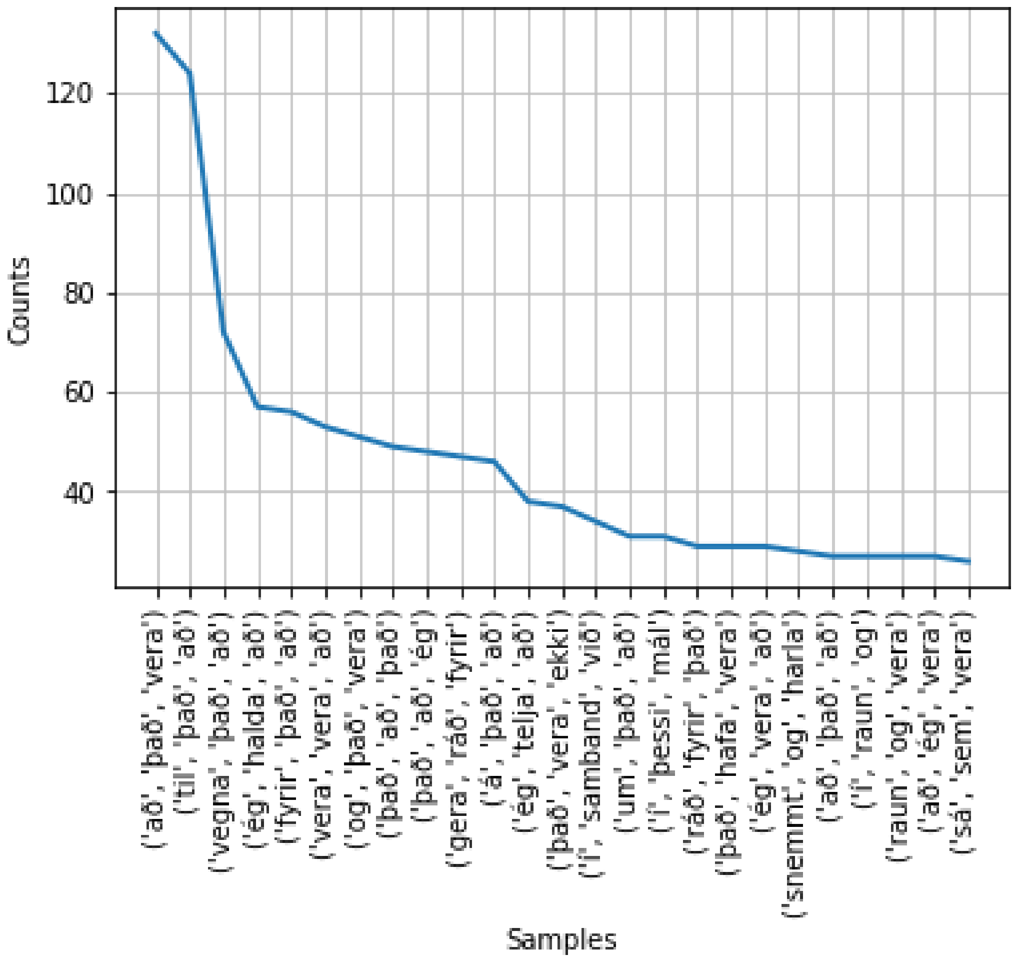
[27]:
```
<AxesSubplot:>
```



3.4.4 n-grams

Here is a description

[28]:
```
def most_common_ngrams(n, top_k, sample):
    file_contents = []

    for file in get_random_sample(sample):
        file_contents.extend(extract_words(file)[1])
    fq_ngr = FreqDist(ngrams(file_contents, n))
    fq_ngr.plot(top_k, cumulative=False)

most_common_ngrams(n=3, top_k=25, sample=100)
```



3.5 Building model for classifying speeches

This section is the main core of the modelling task of this assignment. It is organized as follows:

- Firstly, the data is splitted into **training and test sets** according to the criteria that in our opinion fits better to the nature of data and task.
- Secondly, we describe the **methods** that are going to be used for **feature extraction** from our documents.
- After that, we describe the **classifiers chosen** to be trained and why they were selected.
- Then, we **train 7 models** combining the feature extraction techniques described and the classifiers selected. This is done through a cross-validated grid search in which many hyperparameters are combined. The goal of this search is to find the **best hyperparameter**

combination of each of the 7 models.

- Finally, we **compare the results from the training within and between the models.**
- Evaluation on test data will be performed in the next section.

3.5.1 Constructing training and test data

Our whole dataset contains **380285 speeches** hold in the Icelandic parliament from 1911 to 2017. In order to perform our train-test split, we took into account the following considerations:

- Documents are **classified in directories by year and month instead of decade.**
- Decades (classes) are highly **unbalanced.** There are much more documents from laterdecades as from the earlier ones. As an example, 1912 has only 14 documents while 2011 has 13957. This may introduce bias in the training of the models if not dealt.

To solve the first problem, we use the help function get_files_for_year() created above, which takes n documents from an specified year. After that, for each of the documents, the year is substituted by the decade as shown in the next two sections. This can be done iteratively through a list of years. In this way, **we obtain a dataset with a bunch of corpora labelled by decade.**

To solve the problem of unbalance within classes, we **limit the number of documents to be extracted from each year to 200 for the training set.** This way, we ensure that there will not be too big differences within the number of documents sampled within the years (maximum of 200 vs minimum of 14) and neither within the decades. We choose 200 since we consider it to be a good balance for **undersampling the majority classes but not loosing as much information as we would keep it to minimum of 14**.

Note that, in order to make the runtimes of our notebook shorter (feasible) we **skip intermediate decades from our classification task.** This would simulate that there were not speeches held in some decades. We like to imagine it as weird regime which combines a decade of democracy followed by a decade of dictatorship. In summary:

- 1910s, 1930s, 1950s, 1970s, 1990s, and 2010s are considered.
- Whereas 1920s, 1940s, 1960s, 1980s, and 2000s are discarded.

We will perform a train/test split of the approximate proportion 80/20. We will see why it will be approximate in the next two sections.

Train data **8 years out of the 10 years** that form a decade are chosen for each of the 6 decades considered for the train set. **The other 2 are left for the test set**. The selection of the years was completly random. For each of the decades a maximum of 1600 documents are chosen. However, this will not be equal for all the decandes, since, as explained above, not all the years have at least 200 documents.

Note that for the last decade, we just have documents until 2017. The split will be 6 years (train) vs 1 (test) in this case. Same applies for first decade (in this case, 7 vs. 2).

[29]:
```
#set seed for reproducibility
random.seed(123)

file_contents = []
targets = []
```

```python
for year in [1911, 1912, 1914, 1915, 1916, 1918, 1919,
             1920, 1921, 1922, 1924, 1925, 1926, 1928, 1929,
             1931, 1932, 1933, 1934, 1935, 1936, 1937, 1938,
             1941, 1942, 1943, 1944, 1945, 1946, 1947, 1948,
             1951, 1952, 1953, 1955, 1956, 1957, 1958, 1959,
             1961, 1962, 1963, 1965, 1966, 1967, 1968, 1969,
             1970, 1971, 1972, 1973, 1974, 1975, 1978, 1979,
             1980, 1981, 1982, 1983, 1984, 1985, 1988, 1989,
             1990, 1991, 1992, 1993, 1995, 1996, 1997, 1999,
             2000, 2001, 2002, 2003, 2005, 2006, 2007, 2009,
             2010, 2011, 2012, 2013, 2014, 2016, 2017]:
    for file in get_files_for_year(year, 200):
        file_contents.append(extract_words(file)[1])
        targets.append(year - year%10)
```

```
For year 1911: Fetching 200 samples out of 125 (~160.0%)
For year 1912: Fetching 200 samples out of 14 (~1428.5714285714287%)
For year 1914: Fetching 200 samples out of 1306 (~15.313935681470138%)
For year 1915: Fetching 200 samples out of 1383 (~14.461315979754158%)
For year 1916: Fetching 200 samples out of 0 (~0%)
For year 1918: Fetching 200 samples out of 0 (~0%)
For year 1919: Fetching 200 samples out of 0 (~0%)
For year 1920: Fetching 200 samples out of 0 (~0%)
For year 1921: Fetching 200 samples out of 0 (~0%)
For year 1922: Fetching 200 samples out of 0 (~0%)
For year 1924: Fetching 200 samples out of 41 (~487.8048780487805%)
For year 1925: Fetching 200 samples out of 23 (~869.5652173913044%)
For year 1926: Fetching 200 samples out of 37 (~540.5405405405405%)
For year 1928: Fetching 200 samples out of 21 (~952.3809523809524%)
For year 1929: Fetching 200 samples out of 96 (~208.33333333333334%)
For year 1931: Fetching 200 samples out of 25 (~800.0%)
For year 1932: Fetching 200 samples out of 28 (~714.2857142857143%)
For year 1933: Fetching 200 samples out of 52 (~384.61538461538464%)
For year 1934: Fetching 200 samples out of 20 (~1000.0%)
For year 1935: Fetching 200 samples out of 24 (~833.3333333333334%)
For year 1936: Fetching 200 samples out of 25 (~800.0%)
For year 1937: Fetching 200 samples out of 1381 (~14.48225923244026%)
For year 1938: Fetching 200 samples out of 1676 (~11.933174224343675%)
For year 1941: Fetching 200 samples out of 2066 (~9.68054211035818%)
For year 1942: Fetching 200 samples out of 2357 (~8.485362749257531%)
For year 1943: Fetching 200 samples out of 3960 (~5.05050505050505%)
For year 1944: Fetching 200 samples out of 1072 (~18.65671641791045%)
For year 1945: Fetching 200 samples out of 1859 (~10.758472296933835%)
For year 1946: Fetching 200 samples out of 2789 (~7.171029042667623%)
For year 1947: Fetching 200 samples out of 2838 (~7.047216349541931%)
For year 1948: Fetching 200 samples out of 2262 (~8.841732979664014%)
For year 1951: Fetching 200 samples out of 2453 (~8.153281695882592%)
For year 1952: Fetching 200 samples out of 1689 (~11.841326228537596%)
For year 1953: Fetching 200 samples out of 1415 (~14.134275618374557%)
For year 1955: Fetching 200 samples out of 1433 (~13.956734124214934%)
For year 1956: Fetching 200 samples out of 1213 (~16.488046166529266%)
For year 1957: Fetching 200 samples out of 1763 (~11.344299489506524%)
For year 1958: Fetching 200 samples out of 1169 (~17.108639863130882%)
For year 1959: Fetching 200 samples out of 1326 (~15.082956259426847%)
For year 1961: Fetching 200 samples out of 1409 (~14.194464158977999%)
For year 1962: Fetching 200 samples out of 1475 (~13.559322033898304%)
For year 1963: Fetching 200 samples out of 1376 (~14.534883720930232%)
For year 1965: Fetching 200 samples out of 1520 (~13.157894736842104%)
For year 1966: Fetching 200 samples out of 1404 (~14.245014245014245%)
For year 1967: Fetching 200 samples out of 1274 (~15.698587127158556%)
For year 1968: Fetching 200 samples out of 1726 (~11.587485515643106%)
For year 1969: Fetching 200 samples out of 1740 (~11.494252873563218%)
For year 1970: Fetching 200 samples out of 1921 (~10.41124414367517%)
For year 1971: Fetching 200 samples out of 2071 (~9.657170449058427%)
For year 1972: Fetching 200 samples out of 2368 (~8.445945945945946%)
For year 1973: Fetching 200 samples out of 2329 (~8.587376556462%)
For year 1974: Fetching 200 samples out of 2350 (~8.51063829787234%)
For year 1975: Fetching 200 samples out of 2430 (~8.23045267489712%)
For year 1978: Fetching 200 samples out of 2641 (~7.5728890571753125%)
For year 1979: Fetching 200 samples out of 2165 (~9.237875288683602%)
For year 1980: Fetching 200 samples out of 3406 (~5.871990604815032%)
For year 1981: Fetching 200 samples out of 3491 (~5.729017473503294%)
For year 1982: Fetching 200 samples out of 2894 (~6.91085003455425%)
For year 1983: Fetching 200 samples out of 2651 (~7.544322897019993%)
For year 1984: Fetching 200 samples out of 4171 (~4.795013186286262%)
For year 1985: Fetching 200 samples out of 4709 (~4.247186239116585%)
For year 1988: Fetching 200 samples out of 4198 (~4.764173415912339%)
For year 1989: Fetching 200 samples out of 5085 (~3.933136676499508%)
For year 1990: Fetching 200 samples out of 4662 (~4.29000429000429%)
For year 1991: Fetching 200 samples out of 4747 (~4.213187276174426%)
For year 1992: Fetching 200 samples out of 8925 (~2.2408963585434174%)
For year 1993: Fetching 200 samples out of 7412 (~2.698327037236913%)
For year 1995: Fetching 200 samples out of 5129 (~3.899395593682979%)
For year 1996: Fetching 200 samples out of 7184 (~2.7839643652561246%)
For year 1997: Fetching 200 samples out of 6960 (~2.8735632183908044%)
For year 1999: Fetching 200 samples out of 6056 (~3.3025099075297226%)
For year 2000: Fetching 200 samples out of 7466 (~2.678810608090008%)
For year 2001: Fetching 200 samples out of 8210 (~2.43605359317905%)
For year 2002: Fetching 200 samples out of 8061 (~2.4810817516437167%)
For year 2003: Fetching 200 samples out of 5872 (~3.4059945504087192%)
For year 2005: Fetching 200 samples out of 8269 (~2.4186721489902046%)
For year 2006: Fetching 200 samples out of 8810 (~2.2701475595913734%)
For year 2007: Fetching 200 samples out of 7863 (~2.543558438255119%)
For year 2009: Fetching 200 samples out of 17262 (~1.1586142973004288%)
For year 2010: Fetching 200 samples out of 11089 (~1.8035891423933628%)
For year 2011: Fetching 200 samples out of 13957 (~1.4329727018700293%)
For year 2012: Fetching 200 samples out of 16356 (~1.2227928588897041%)
For year 2013: Fetching 200 samples out of 10240 (~1.953125%)
For year 2014: Fetching 200 samples out of 12404 (~1.6123831022250887%)
For year 2016: Fetching 200 samples out of 8165 (~2.449479485609308%)
For year 2017: Fetching 200 samples out of 7270 (~2.751031636863824%)
```

Let's randomly choose a fixed number of documents (here currently: 200) from various different decades. Then passing (document, decade) pairs to the model below. The decade is computed by subtracting `mod(<year>, 10)` from `<year>`.

Test data Choose the other **2 years that were not selected** within the decades in the train set. In this case, **we do not have to limit the number of documents for year.** It doesn't make sense to undersample the test set since it represents "unseen" data. And, unseen data should be as close to reality as possible. That means, that it is normal that there are much more documents from later decades than from earlier.

So, instead of 200, we will put there a very large number to be sure that all the documents from every year are selected.

```
[30]: #seed for reproducibility
      random.seed(123)

      file_contents_test = []
      targets_test = []

      for year in [1913, 1917,1930, 1939, 1950, 1954, 1976, 1977, 1994, 2013,2015,
                   1923, 1927,1940, 1949, 1960, 1964, 1986, 1987, 2004]:
          for file in get_files_for_year(year):
              file_contents_test.append(extract_words(file)[1])
              targets_test.append(year - year%10)
```

```
For year 1913: Fetching 2037 samples out of 2037 (~100.0%)
For year 1917: Fetching 0 samples out of 0 (~0%)
For year 1930: Fetching 30 samples out of 30 (~100.0%)
For year 1939: Fetching 1632 samples out of 1632 (~100.0%)
For year 1950: Fetching 3017 samples out of 3017 (~100.0%)
For year 1954: Fetching 1546 samples out of 1546 (~100.0%)
For year 1976: Fetching 2555 samples out of 2555 (~100.0%)
For year 1977: Fetching 2105 samples out of 2105 (~100.0%)
For year 1994: Fetching 8187 samples out of 8187 (~100.0%)
For year 2013: Fetching 10240 samples out of 10240 (~100.0%)
For year 2015: Fetching 18052 samples out of 18052 (~100.0%)
For year 1923: Fetching 26 samples out of 26 (~100.0%)
For year 1927: Fetching 59 samples out of 59 (~100.0%)
For year 1940: Fetching 1458 samples out of 1458 (~100.0%)
For year 1949: Fetching 2661 samples out of 2661 (~100.0%)
For year 1960: Fetching 1862 samples out of 1862 (~100.0%)
For year 1964: Fetching 1677 samples out of 1677 (~100.0%)
For year 1986: Fetching 3432 samples out of 3432 (~100.0%)
For year 1987: Fetching 3212 samples out of 3212 (~100.0%)
For year 2004: Fetching 9466 samples out of 9466 (~100.0%)
```

See classes distribution within train and test sets

[31]:
```
from collections import Counter

print(Counter(targets).keys())
print(Counter(targets).values())

print(Counter(targets_test).keys())
print(Counter(targets_test).values())
```

```
dict_keys([1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990, 2000, 2010])
dict_values([539, 218, 574, 1600, 1600, 1600, 1600, 1600, 1600, 1600, 1400])
dict_keys([1910, 1930, 1950, 1970, 1990, 2010, 1920, 1940, 1960, 1980, 2000])
dict_values([2037, 1662, 4563, 4660, 8187, 28292, 85, 4119, 3539, 6644, 9466])
```

We see that although there are some differences within the classes for the train split, it is acceptable to perform the classification task. Maximum within the classes for training is 1600.

Test set is expected to have much more class imbalance. However, our model should dealt with it thanks to the undersampling that was performed.

3.5.2 Text feature extraction

We have considered 3 different methods for text feature extraction: Tf-idf, word2vec and doc2vec. All of them will be implemented through the corresponding functions from *sklearn* library.

TF-IDF Helper function to transform the data so that it is in the right format for the tfidfVectorizer() function that will be used later on:

[32]:
```
class JoinElement(object):
    def fit(self, X, y):
        return self

    def transform(self, X):
        #joins the elements of a list (which represents a document) into a␣
 ↪single string
        #with a blank space separation between each word
        return [' '.join(X[i]) for i in range(len(X))]
```

More information about it: sklearn documentation.

Word2Vec Original paper

: Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems, 26, 3111-3119.

With this model every word is assigned a unique vector of configurable cardinality such that the dot product of two randomly chosen vectors should be proportional to the semantic similarity for the associated words. This happens during the training step using logistic regression and sliding windows. Personally I found that this video delivers a solid explanation of the concepts: https://www.youtube.com/watch?v=QyrUentbkvw

However, since we are working with entire documents as training items we have to somehow aggregate the vectors for every word in a given document. This can be done e.g. by taking the mean and/or summing up the vectors (see `MeanEmbeddingVectorizer`), optionally weighted by TF-IDF (see `MeanEmbeddingVectorizerTfidf`).

[33]:
```
class MeanEmbeddingVectorizer(BaseEstimator):
    def fit(self, X, y):
        self.word2vec = Word2Vec(X)
        return self

    def transform(self, X):
        return np.array([
            np.mean([self.word2vec.wv[w] for w in words if w in self.word2vec.
↪wv.vocab]
                    or [np.zeros(self.word2vec.vector_size)], axis=0)
            for words in X
        ])

    def fit_transform(self, X, y):
        self.fit(X, y)
        return self.transform(X)
```

[34]:
```
class MeanEmbeddingVectorizerTfidf(BaseEstimator):
    def fit(self, X, y):
        self.word2vec = Word2Vec(X)
        self.X_joined = [' '.join(X[i]) for i in range(len(X))]
        self.vectorizer = TfidfVectorizer()
        self.transformed = self.vectorizer.fit_transform(self.X_joined)
        self.transformed = pd.DataFrame.sparse.from_spmatrix(self.transformed)
        return self

    def tfidf(self, w, docid):
        if w in self.vectorizer.vocabulary_:
            return self.transformed[self.vectorizer.vocabulary_[w]][docid]
        else:
            return 0

    def transform(self, X):
        return np.array([
            np.mean([self.word2vec.wv[w] * self.tfidf(w, i) for w in words if w␣
↪in self.word2vec.wv.vocab]
                    or [np.zeros(self.word2vec.vector_size)], axis=0)
            for i, words in enumerate(X)
        ])

    def fit_transform(self, X, y):
        self = self.fit(X, y)
        return self.transform(X)
```

Doc2Vec Finally we are attempting to build a model using *Doc2Vec*. After training this model with our training corpus we receive a vector of configurable cardinality for each document.

Original paper

: Le, Quoc, and Tomas Mikolov. "Distributed representations of sentences and documents." International conference on machine learning. 2014.

[35]:
```
class Doc2Vectorizer(BaseEstimator):
    def __init__(self, window=2, vector_size=100):
        self.window = window
        self.vector_size = vector_size

    def fit(self, X, y):
        docs = [TaggedDocument(X[i], [y[i]]) for i in range(len(X))]
        self.doc_vec = Doc2Vec(docs, vector_size=self.vector_size, window=self.
↪window, min_count=1, workers=4)
        return self

    def transform(self, X):
        return [self.doc_vec.infer_vector(X[i]) for i in range(len(X))]
```

BERT (*Bidirectional Encoder Representations from Transformers*) is also interesting to look at, but we'll skip this here because we predict training a model from scratch would use up too many resources. Given more time however you could search for pretrained networks that roughly serve the purpose of classification of documents according to publication year.

Paper

: Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

3.5.3 Classifiers

3 different classifiers are going to be trained: Multinomial Naive Bayes, Support Vector Machines and Random Forest Classifier. All of them will be implemented using sklearn library.

Multinominal Naive Bayes (MNB) MNB is a common method for document classification due to its good balance between computational efficiency and predictive performance (Eibe, 2006). Therefore, we decided to choose it as one of our classifiers.

Details on the algorithm implementation can be found in the sklearn documentation..

The set of hyperparameters chosen to perform the grid search cross-validation during the training are based on the recommendationsfrom this article.

Support Vector Machines Support vector machines are widely used for classification purposes. What is more, it improves Multinominal Naive Bayes in terms of performance in most of the classification taks. Thus, it was also chosen as one of our classifiers to be trained.

Details on the algorithm implementation can be found in sklearn documentation.

The set of hyperparameters chosen to perform the grid search cross-validation during the training are based on the recommendations from this article.

Random Forest Classifier Random Forest Classifier is one of the best methods according to the literature for classification tasks. However, the runtime may be extremly large (specially when increasing the size of the forest within grid search CV setups).

Details on the algorithm implementation can be found in sklearn documentation..

The set of hyperparameters chosen to perform the grid search cross-validation during the training are based on the recommendations from this article..

3.5.4 Train models

Since there are 3 methods for feature extraction and 3 classifiers, we should train 9 kind of models with their different combinations of hyperparameters. However, multinomial naive bayes does not take negative values produced by Word2Vec and Doc2Vec. Therefore, we have 7.

For each model, **a grid search is performed with different combinations of hyperparameters** for the classifiers and the text extraction methods. Afterwards, the most relevant results of each of the models are stored in a pandas data frame.

The goal of this grid search is to find the best combination of hyperparameters for each of our 7 combinations.

Note that **ideally we should perform a random search prior to the grid search to limit the scope of the best hyperparameters** to be used and then perform a more accurate search. However, this would lead to a tedious notebook and extremly large runtimes.

Model 1: TFIDF vectorizer, select K best and Multinomial Naive Bayes

[36]:
```
#choose parameters for the different steps in the pipeline
parameters_model_1 = {

    #select KBest
    #k number of top features to select (default 10)
    "k_best__k": [10, 500],
    #score function to be used (default f_classif)
    "k_best__score_func": [chi2],

    #MultinomialNaiveBayes
    #alpha is a parameter for smoothing (default value is 1)
    "MNB__alpha": np.linspace(0.5, 1.5, 4),
    #whether to learn class prior probabilities or not (dafult value is True)
    "MNB__fit_prior": [True,False],

    #TFIDF Vectorizer
    #Whether the feature should be made of word or character n-grams (default␣
 ↪word)
    "tfidf__analyzer": ["word"],
    #Smooth idf weights by adding one to document frequencies,
    #as if an extra document was seen containing every term in the collection␣
 ↪exactly once.
    #Prevents zero divisions (default True)
    "tfidf__smooth_idf": [True, False]


}


#build a pipeline
model_1_pipeline = Pipeline([
        #joins list into a single string
        ('join', JoinElement()),
        #tfidf vectorizer
        ('tfidf', TfidfVectorizer()),
        #select 1000 best from word vectors
        ('k_best', SelectKBest()),
        #apply naive bayes
        ('MNB', MultinomialNB())
    ])

#design grid search
grid_search_model_1 = GridSearchCV(
    #pipeline to be followed
    model_1_pipeline,
    #parameters
    param_grid=parameters_model_1,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)
```

```
#fit the grid search for training data
grid_search_model_1.fit(file_contents, targets)

#save results of cross validation
cv_results_model_1 = pd.DataFrame(grid_search_model_1.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_1 if (col.startswith("param_") or␣
 ↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_1 = cv_results_model_1[filter_col]

#save name of the model for later comparison
cv_results_model_1.insert(loc=0, column="Model", value= "1")

#round mean_test_score
cv_results_model_1["mean_test_score"] = cv_results_model_1["mean_test_score"].
 ↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_1.sort_values(by="mean_test_score", ascending=False).
 ↪head(5))
```

```
   Model  mean_fit_time  mean_score_time param_MNB__alpha  \
6      1       8.387684         2.188630              0.5
7      1       8.919221         1.888311              0.5
15     1       8.542751         1.953357         0.833333
14     1       7.195633         1.739061         0.833333
23     1       7.141477         2.184534         1.166667

   param_MNB__fit_prior param_k_best__k           param_k_best__score_func  \
6                 False             500  <function chi2 at 0x7fac4119dc10>
7                 False             500  <function chi2 at 0x7fac4119dc10>
15                False             500  <function chi2 at 0x7fac4119dc10>
14                False             500  <function chi2 at 0x7fac4119dc10>
23                False             500  <function chi2 at 0x7fac4119dc10>

   param_tfidf__analyzer param_tfidf__smooth_idf  mean_test_score  \
6                   word                    True             0.49
7                   word                   False             0.49
15                  word                   False             0.48
14                  word                    True             0.48
23                  word                   False             0.46

    rank_test_score
```

```
6                  1
7                  2
15                 4
14                 3
23                 8
```

Model 2: TFIDF vectorizer, select K best and SVC

```
[37]: #choose parameters for the different steps in the pipeline
parameters_model_2 = {

    #select KBest
    #k number of top features to select (default 10)
    "k_best__k": [10, 500],
    #score function to be used (default f_classif)
    "k_best__score_func": [chi2],

    #SVC
    #Specifies the kernel type to be used in the algorithm
    "SVC__kernel" : ["linear", "poly", "sigmoid"],
    #Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid'
    "SVC__gamma": [1,0.1,0.001],
    #Regularization parameter. The strength of the regularization is inversely␣
 ↪proportional to C.
    #Must be strictly positive. The penalty is a squared l2 penalty.
    "SVC__C": [0.1,1, 10, 100],

    #TFIDF Vectorizer
    #Whether the feature should be made of word or character n-grams (default␣
 ↪word)
    "tfidf__analyzer": ["word"],
    #Smooth idf weights by adding one to document frequencies,
    #as if an extra document was seen containing every term in the collection␣
 ↪exactly once.
    #Prevents zero divisions (default True)
    "tfidf__smooth_idf": [True, False]

}



#build a pipeline
model_2_pipeline = Pipeline([
        #joins list into a single string
        ('join', JoinElement()),
        #tfidf vectorizer
        ('tfidf', TfidfVectorizer()),
        #select 1000 best from word vectors
        ('k_best', SelectKBest()),
        #apply naive bayes
        ('SVC', SVC())
    ])

#design grid search
grid_search_model_2 = GridSearchCV(
    #pipeline to be followed
    model_2_pipeline,
    #parameters
    param_grid=parameters_model_2,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)

#fit the grid search for training data
grid_search_model_2.fit(file_contents, targets)

#save results of cross validation
cv_results_model_2 = pd.DataFrame(grid_search_model_2.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_2 if (col.startswith("param_") or␣
 ↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_2 = cv_results_model_2[filter_col]

#save name of the model for later comparison
cv_results_model_2.insert(loc=0, column="Model", value= "2")

#round mean_test_score
cv_results_model_2["mean_test_score"] = cv_results_model_2["mean_test_score"].
 ↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_2.sort_values(by="mean_test_score", ascending=False).
 ↪head(5))
```

```
    Model  mean_fit_time  mean_score_time param_SVC__C param_SVC__gamma  \
111     2      45.537666        12.704505          100                1
135     2      49.423878        14.831820          100            0.001
123     2      46.233693        13.522951          100              0.1
134     2      37.751620        16.068168          100            0.001
```

```
110     2      36.764385        14.584715         100                 1

    param_SVC__kernel param_k_best__k           param_k_best__score_func  \
111            linear             500  <function chi2 at 0x7fac4119dc10>
135            linear             500  <function chi2 at 0x7fac4119dc10>
123            linear             500  <function chi2 at 0x7fac4119dc10>
134            linear             500  <function chi2 at 0x7fac4119dc10>
110            linear             500  <function chi2 at 0x7fac4119dc10>

    param_tfidf__analyzer param_tfidf__smooth_idf  mean_test_score  \
111                  word                   False             0.54
135                  word                   False             0.54
123                  word                   False             0.54
134                  word                    True             0.53
110                  word                    True             0.53

     rank_test_score
111                1
135                1
123                1
134                4
110                4
```

Model 3: TFIDF vectorizer, select K best and Random Forest Classifier

[38]:
```
#choose parameters for the different steps in the pipeline
parameters_model_3 = {

    #select KBest
    #k number of top features to select (default 10)
    "k_best__k": [10, 500],
    #score function to be used (default f_classif)
    "k_best__score_func": [chi2],

    #RF classifier
    #nThe number of trees in the forest (default is 100)
    "clf__n_estimators" : [10,100,200],
    #The minimum number of samples required to split an internal node (default␣
 ↪is 2 but it is a large dataset)
    "clf__min_samples_split": [10, 40, 80],
    #The number of features to consider when looking for the best split␣
 ↪(default "auto" but sparse dataset)
    'clf__max_features': ["auto", 10],
    #maximum depth of the tree (default None)
    'clf__max_depth': [10, None],
    #TFIDF Vectorizer
    #Whether the feature should be made of word or character n-grams (default␣
 ↪word)
    "tfidf__analyzer": ["word"],
    #Smooth idf weights by adding one to document frequencies,
    #as if an extra document was seen containing every term in the collection␣
 ↪exactly once.
    #Prevents zero divisions (default True)
    "tfidf__smooth_idf": [True, False]




}



#build a pipeline
model_3_pipeline = Pipeline([
        #joins list into a single string
        ('join', JoinElement()),
        #tfidf vectorizer
        ('tfidf', TfidfVectorizer()),
        #select 1000 best from word vectors
        ('k_best', SelectKBest()),
        #apply naive bayes
        ('clf', RandomForestClassifier())
    ])

#design grid search
grid_search_model_3 = GridSearchCV(
    #pipeline to be followed
    model_3_pipeline,
    #parameters
    param_grid=parameters_model_3,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)

#fit the grid search for training data
grid_search_model_3.fit(file_contents, targets)

#save results of cross validation
cv_results_model_3 = pd.DataFrame(grid_search_model_3.cv_results_)
```

```
#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_3 if (col.startswith("param_") or␣
↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_3 = cv_results_model_3[filter_col]

#save name of the model for later comparison
cv_results_model_3.insert(loc=0, column="Model", value= "3")

#round mean_test_score
cv_results_model_3["mean_test_score"] = cv_results_model_3["mean_test_score"].
↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_3.sort_values(by="mean_test_score", ascending=False).
↪head(5))
```

```
    Model  mean_fit_time  mean_score_time param_clf__max_depth  \
119     3      36.584321         3.331602                None
143     3      17.497095         2.516746                None
142     3      19.232731         2.954226                None
131     3      27.325266         3.639129                None
130     3      23.597651         3.784672                None

    param_clf__max_features param_clf__min_samples_split  \
119                      10                           10
143                      10                           80
142                      10                           80
131                      10                           40
130                      10                           40

    param_clf__n_estimators param_k_best__k  \
119                     200             500
143                     200             500
142                     200             500
131                     200             500
130                     200             500

              param_k_best__score_func param_tfidf__analyzer  \
119  <function chi2 at 0x7fac4119dc10>                  word
143  <function chi2 at 0x7fac4119dc10>                  word
142  <function chi2 at 0x7fac4119dc10>                  word
131  <function chi2 at 0x7fac4119dc10>                  word
130  <function chi2 at 0x7fac4119dc10>                  word

    param_tfidf__smooth_idf  mean_test_score  rank_test_score
```

```
119                     False            0.51                 1
143                     False            0.50                 2
142                      True            0.50                 8
131                     False            0.50                 3
130                      True            0.50                 5
```

Model 4: Word2Vec and SVC

[39]:
```
#choose parameters for the different steps in the pipeline
parameters_model_4 = {

    #SVC
    #Specifies the kernel type to be used in the algorithm
    "SVC__kernel" : ["linear", "poly", "sigmoid"],
    #Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid'
    "SVC__gamma": [1,0.1,0.001],
    #Regularization parameter. The strength of the regularization is inversely␣
 ↪proportional to C.
    #Must be strictly positive. The penalty is a squared l2 penalty.
    "SVC__C": [0.1,1, 10, 100]

    #defaults for Word2Vec

}



#build a pipeline
model_4_pipeline = Pipeline([
        #tfidf vectorizer
        ('word2vec', MeanEmbeddingVectorizer()),
        #apply naive bayes
        ('SVC', SVC())
    ])

#design grid search
grid_search_model_4 = GridSearchCV(
    #pipeline to be followed
    model_4_pipeline,
    #parameters
    param_grid=parameters_model_4,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)
```

```
#fit the grid search for training data
grid_search_model_4.fit(file_contents, targets)

#save results of cross validation
cv_results_model_4 = pd.DataFrame(grid_search_model_4.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_4 if (col.startswith("param_") or␣
↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_4 = cv_results_model_4[filter_col]

#save name of the model for later comparison
cv_results_model_4.insert(loc=0, column="Model", value= "4")

#round mean_test_score
cv_results_model_4["mean_test_score"] = cv_results_model_4["mean_test_score"].
↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_4.sort_values(by="mean_test_score", ascending=False).
↪head(5))
```

```
   Model  mean_fit_time  mean_score_time param_SVC__C param_SVC__gamma  \
31     4     343.308681        34.224248          100              0.1
22     4     288.424709        26.118967           10              0.1
18     4     278.652019        34.882533           10                1
1      4     389.094729        50.654311          0.1                1
27     4     497.729028        27.616480          100                1

   param_SVC__kernel  mean_test_score  rank_test_score
31              poly             0.42                2
22              poly             0.42                1
18            linear             0.41                4
1               poly             0.41                3
27            linear             0.41                8
```

Model 5: Word2Vec and Random Forest Classifier

[40]:
```
#choose parameters for the different steps in the pipeline
parameters_model_5 = {

    #RF classifier
    #nThe number of trees in the forest (default is 100)
    "clf__n_estimators" : [10,100,200],
    #The minimum number of samples required to split an internal node (default␣
↪is 2 but it is a large dataset)
```

```
    "clf__min_samples_split": [10, 40, 80],
    #The number of features to consider when looking for the best split␣
 ↪(default "auto" but sparse dataset)
    'clf__max_features': ["auto", 10],
    #maximum depth of the tree (default None)
    'clf__max_depth': [10, None]

    #defaults word2vec

}


#build a pipeline
model_5_pipeline = Pipeline([
        #tfidf vectorizer
        ('word2vec', MeanEmbeddingVectorizer()),
        #apply naive bayes
        ('clf', RandomForestClassifier())
    ])

#design grid search
grid_search_model_5 = GridSearchCV(
    #pipeline to be followed
    model_5_pipeline,
    #parameters
    param_grid=parameters_model_5,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)

#fit the grid search for training data
grid_search_model_5.fit(file_contents, targets)

#save results of cross validation
cv_results_model_5 = pd.DataFrame(grid_search_model_5.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_5 if (col.startswith("param_") or␣
 ↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_5 = cv_results_model_5[filter_col]
```

```
#save name of the model for later comparison
cv_results_model_5.insert(loc=0, column="Model", value= "5")

#round mean_test_score
cv_results_model_5["mean_test_score"] = cv_results_model_5["mean_test_score"].
↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_5.sort_values(by="mean_test_score", ascending=False).
↪head(5))
```

```
   Model  mean_fit_time  mean_score_time param_clf__max_depth  \
20     5     177.896227         9.965774                 None
32     5     154.525328         8.964136                 None
29     5     163.433541         8.489580                 None
35     5     113.754320         2.856136                 None
13     5     155.250164        10.702845                   10

   param_clf__max_features param_clf__min_samples_split  \
20                    auto                           10
32                      10                           40
29                      10                           10
35                      10                           80
13                      10                           40

   param_clf__n_estimators  mean_test_score  rank_test_score
20                     200             0.33                1
32                     200             0.32                2
29                     200             0.32                3
35                     200             0.31               15
13                     100             0.31               18
```

Model 6: Doc2Vec and Support Vector Machines

[41]:
```
#choose parameters for the different steps in the pipeline
parameters_model_6 = {


    #SVC
    #Specifies the kernel type to be used in the algorithm
    "SVC__kernel" : ["linear", "poly", "sigmoid"],
    #Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid'
    "SVC__gamma": [1,0.1,0.001],
    #Regularization parameter. The strength of the regularization is inversely␣
↪proportional to C.
    #Must be strictly positive. The penalty is a squared l2 penalty.
    "SVC__C": [0.1,1, 10, 100]

    #defaults doc2vec
}


#build a pipeline
model_6_pipeline = Pipeline([
        #tfidf vectorizer
        ('doc2vec', Doc2Vectorizer()),
        #apply naive bayes
        ('SVC', SVC())
    ])

#design grid search
grid_search_model_6 = GridSearchCV(
    #pipeline to be followed
    model_6_pipeline,
    #parameters
    param_grid=parameters_model_6,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)

#fit the grid search for training data
grid_search_model_6.fit(file_contents, targets)

#save results of cross validation
cv_results_model_6 = pd.DataFrame(grid_search_model_6.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_6 if (col.startswith("param_") or␣
 ↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_6 = cv_results_model_6[filter_col]

#save name of the model for later comparison
cv_results_model_6.insert(loc=0, column="Model", value= "6")

#round mean_test_score
cv_results_model_6["mean_test_score"] = cv_results_model_6["mean_test_score"].
 ↪round(2)

#show best 5 sorted by mean_test_score
```

```
display(cv_results_model_6.sort_values(by="mean_test_score", ascending=False).
 ↪head(5))
```

```
   Model  mean_fit_time  mean_score_time param_SVC__C param_SVC__gamma  \
18     6     273.025912        28.329361           10                1
12     6     245.849821        29.302286            1              0.1
33     6     431.278508        23.150362          100            0.001
3      6     228.663433        26.545853          0.1              0.1
27     6     441.331615        27.363341          100                1

   param_SVC__kernel  mean_test_score  rank_test_score
18            linear             0.43                4
12            linear             0.43                2
33            linear             0.43                8
3             linear             0.43                9
27            linear             0.43                6
```

Model 7: Doc2Vec and Random Forest Classifier

```
[42]: #choose parameters for the different steps in the pipeline
parameters_model_7 = {



    #RF classifier
    #nThe number of trees in the forest (default is 100)
    "clf__n_estimators" : [10,100,200],
    #The minimum number of samples required to split an internal node (default␣
 ↪is 2 but it is a large dataset)
    "clf__min_samples_split": [10, 40, 80],
    #The number of features to consider when looking for the best split␣
 ↪(default "auto" but sparse dataset)
    'clf__max_features': ["auto", 10],
    #maximum depth of the tree (default None)
    'clf__max_depth': [10, None]

}


#build a pipeline
model_7_pipeline = Pipeline([
        #tfidf vectorizer
        ('doc2vec', Doc2Vectorizer()),
        #apply naive bayes
        ('clf', RandomForestClassifier())
    ])

#design grid search
grid_search_model_7 = GridSearchCV(
    #pipeline to be followed
    model_7_pipeline,
    #parameters
    param_grid=parameters_model_7,
    #number of folds for CV
    cv=5,
    #scoring to be considered for the cv
    scoring = "accuracy",
    #parallelize if possible
    n_jobs=-1
)

#fit the grid search for training data
grid_search_model_7.fit(file_contents, targets)

#save results of cross validation
cv_results_model_7 = pd.DataFrame(grid_search_model_7.cv_results_)

#filter columns to be kept in the dataframe
filter_col = [col for col in cv_results_model_7 if (col.startswith("param_") or␣
↪col.startswith("mean_") or col.startswith("rank"))]

#save results with only filtered columns
cv_results_model_7 = cv_results_model_7[filter_col]

#save name of the model for later comparison
cv_results_model_7.insert(loc=0, column="Model", value= "7")

#round mean_test_score
cv_results_model_7["mean_test_score"] = cv_results_model_7["mean_test_score"].
↪round(2)

#show best 5 sorted by mean_test_score
display(cv_results_model_7.sort_values(by="mean_test_score", ascending=False).
↪head(5))
```

```
   Model  mean_fit_time  mean_score_time param_clf__max_depth  \
23     7     252.317726        21.616184                None
20     7     260.065633        21.306356                None
28     7     255.810332        26.719237                None
29     7     269.841011        22.845155                None
19     7     242.370124        21.902374                None

   param_clf__max_features param_clf__min_samples_split  \
23                    auto                           40
20                    auto                           10
28                      10                           10
```

```
29                     10                          10
19                   auto                          10

   param_clf__n_estimators  mean_test_score  rank_test_score
23                     200             0.37                1
20                     200             0.37                2
28                     100             0.36                5
29                     200             0.36                4
19                     100             0.36                7
```

3.5.5 Compare CV results from trained models

In this section, the results from CV are compared within the trained models.

Raw results Export results from grid serach. This allows us to experiment with visualiazations and results from CV without having to rerun the whole script.

[43]:
```
#cv_results.to_csv("cv_results.csv",index=False)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-43-d3b59e1ef88e> in <module>
----> 1 cv_results.to_csv("cv_results.csv",index=False)

NameError: name 'cv_results' is not defined
```

A dataframe showing the best models according to the **mean accuracy within the test folds** used for cross validation.

[45]:
```
#merge cv results into 1 that keeps the relevant information

#empty dataframe that will keep all the results
cv_results = pd.DataFrame()

#loop over cv results
for i in [cv_results_model_1, cv_results_model_2, cv_results_model_3,␣
 ↪cv_results_model_4,
          cv_results_model_5, cv_results_model_6, cv_results_model_7]:

    #select relevant columns
    selected = i[["Model","mean_fit_time","mean_score_time","mean_test_score"]]

    #append to cv results
    cv_results = cv_results.append(selected)

#show models with best scores
display(cv_results.sort_values(by="mean_test_score", ascending=False).head(20))
```

```
cv_results.to_csv("cv_results2.csv",index=False)
```

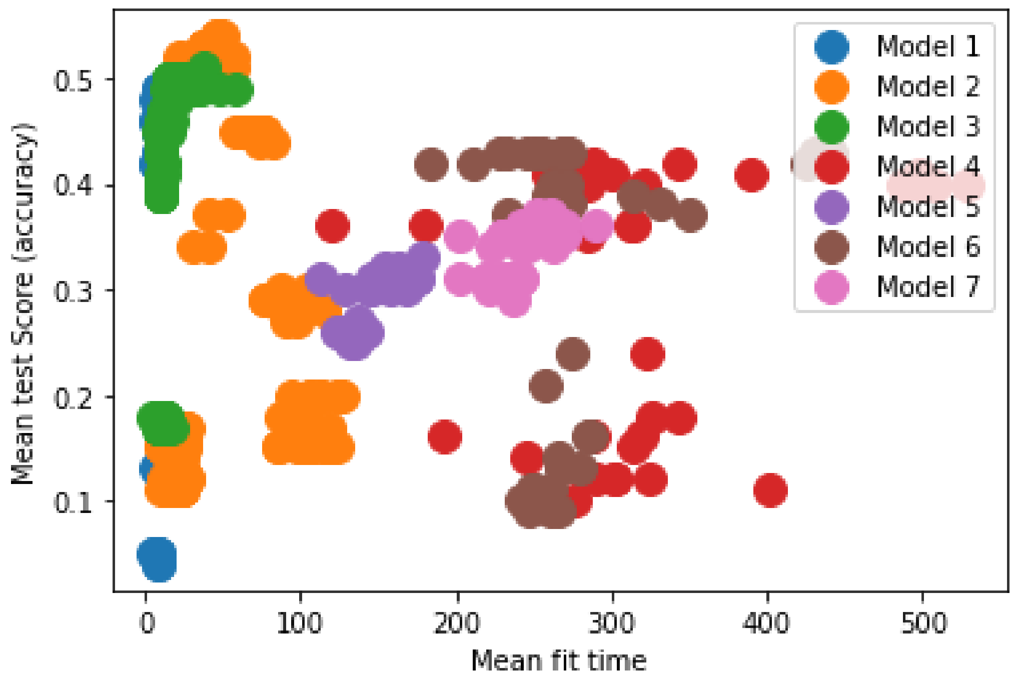
```
    Model  mean_fit_time  mean_score_time  mean_test_score
111     2      45.537666        12.704505             0.54
135     2      49.423878        14.831820             0.54
123     2      46.233693        13.522951             0.54
122     2      37.660268        16.118198             0.53
110     2      36.764385        14.584715             0.53
134     2      37.751620        16.068168             0.53
87      2      50.362453        13.930855             0.52
74      2      42.097400        17.334257             0.52
75      2      53.180003        15.568920             0.52
86      2      39.190326        16.937734             0.52
98      2      22.771912         9.259802             0.52
99      2      28.150809         8.853756             0.52
131     2      55.788226        16.034667             0.52
130     2      42.225446        18.179445             0.52
114     2      42.626273        18.166295             0.51
119     3      36.584321         3.331602             0.51
115     2      55.720585        15.588012             0.51
118     3      33.057600         3.987268             0.50
131     3      27.325266         3.639129             0.50
115     3      23.073168         3.311102             0.50
```

This table could be improved by also indicating the parameters used in each model but I thought it would be a bit overwhelming

Models x y seem to achieve a better accuracy since they appear more often within the first positions.

Tradeoff score vs mean fit time A plot to check if there is some kind of tradeoff between accuracy and runtime of the algorithms.

[46]:
```
#group by model
groups = cv_results.groupby("Model")

# Plot
fig, ax = plt.subplots()
ax.margins(0.05) # Optional, just adds 5% padding to the autoscaling
for name, group in groups:
    ax.plot(group.mean_fit_time, group.mean_test_score, marker='o',␣
 ↪linestyle='', ms=12, label="Model %s" %name)
ax.legend(loc = 1)
plt.xlabel("Mean fit time")
plt.ylabel("Mean test Score (accuracy)")
plt.show()
```



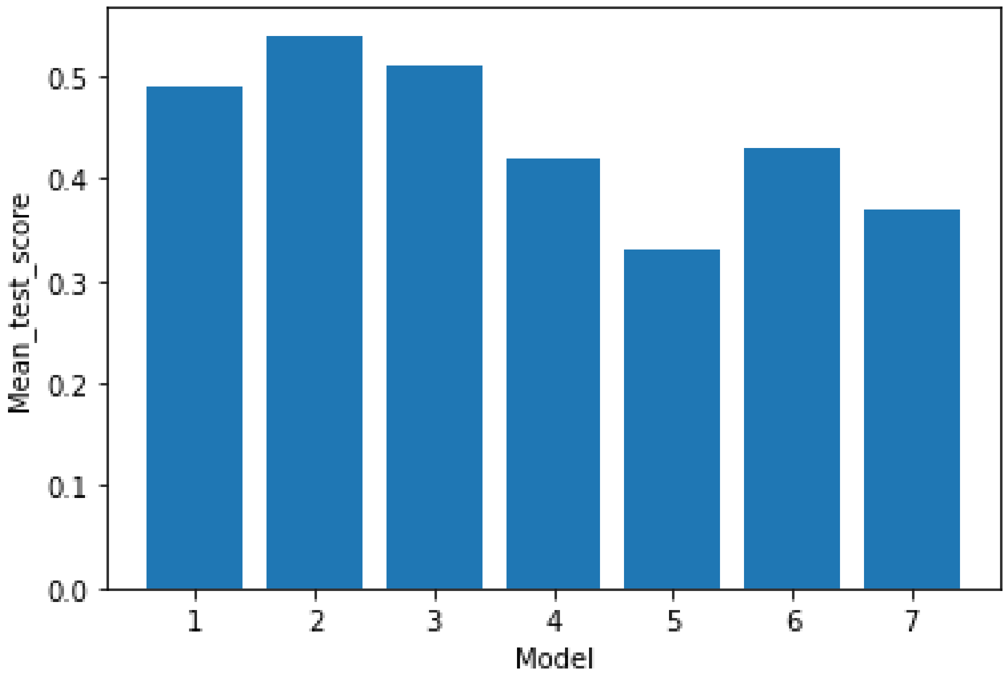
Clear relation between runtime of the models and accuracy within test folds?

Some pre-processing or classifier takes more time to be run?

Further analysis on what it increases runtime of the models

Best estimator from each model

[47]:
```
#group by model and take best mean_test_score for each type of model
best_models = cv_results.groupby("Model")[["mean_test_score"]].max()

#plot
plt.bar(best_models.index, best_models["mean_test_score"])
plt.xlabel("Model")
plt.ylabel("Mean_test_score")
plt.show()
```



As expected best models are x and y

Worse models are x and y

Are they stable to changes in hyperparameters?

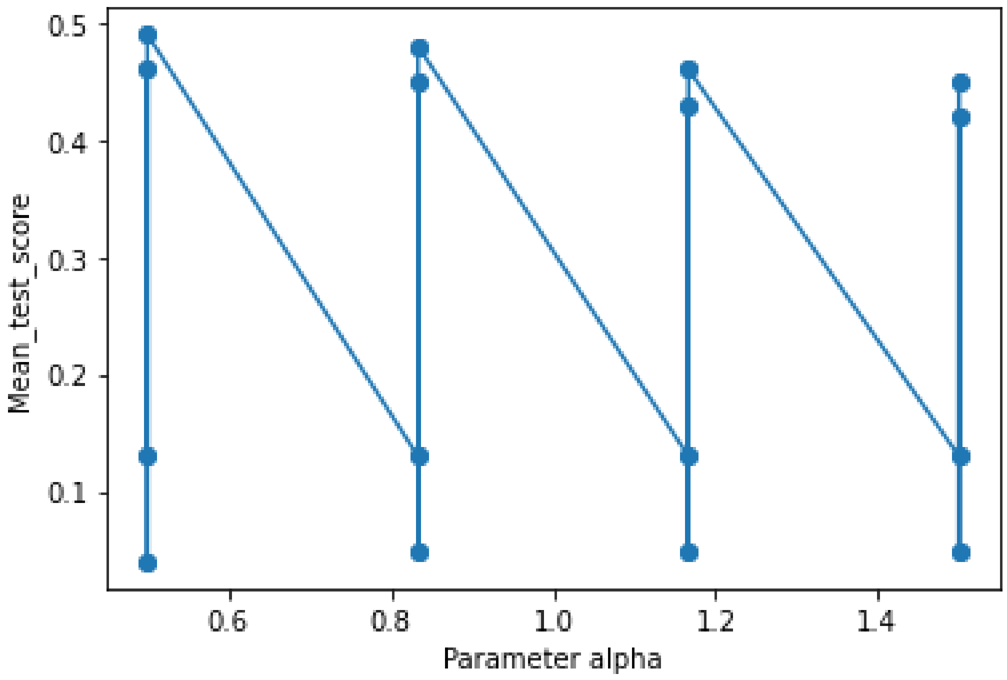
```
[48]: #mean of mean scores within the folds for each model
stability = cv_results.groupby("Model")[["mean_test_score"]].mean()
#standard deviation of the same
stability["standard_deviation"] = cv_results.
 ↪groupby("Model")[["mean_test_score"]].std()

#show
stability
```

```
[48]:        mean_test_score  standard_deviation
Model
1                   0.271875            0.189063
2                   0.234861            0.139133
3                   0.314028            0.143524
4                   0.290556            0.128973
5                   0.292778            0.024913
6                   0.302222            0.140471
7                   0.336944            0.023030
```

Model x is not very stable to the change of hyperparameters. What is influencing it so much?

Show an example

[49]:
```
plt.plot(cv_results_model_1["param_MNB__alpha"],␣
 ↪cv_results_model_1["mean_test_score"], marker="o")
plt.xlabel("Parameter alpha")
plt.ylabel("Mean_test_score")
plt.show()
```



Next steps This grid searches helped us to see which combinations of models and hyperparameters are expected to be the best, how stable they are and other insights. However, **we cannot draw strong conclusions on this since we are still dealing with train data.** This step is only helping us to understand the models better and choose the ones with which we want to test (or validate). To proceed further we decided to select **the best combination of hyperparameters for each of the 7 models, predict on test data**, evaluate and draw conclusions. That is done in the next section of this notebook. We know that this are not strictly the best 7 models (see raw results), but we wanted to include more diversity.

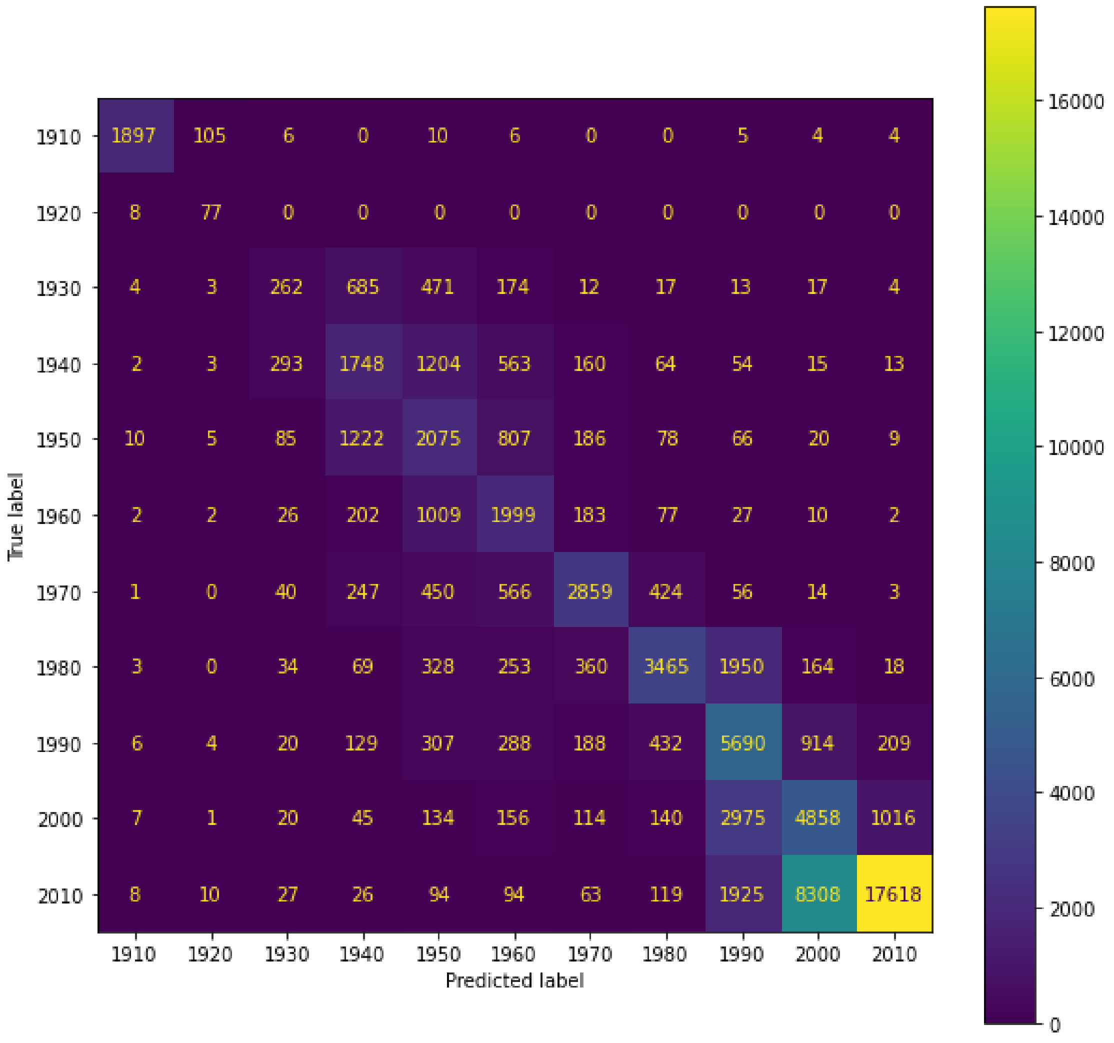
3.6 Evaluation and model selection

Predict on test data using the best combinations of hyperparameters used in the models obtained in the training phase and evaluate using different metrics.

This simulates predictions on unseen data. However, since it is done for many models and then we will choose the best model out of them, it behaves more like a validation set that would help us choose which model we would apply to actually unseen data.

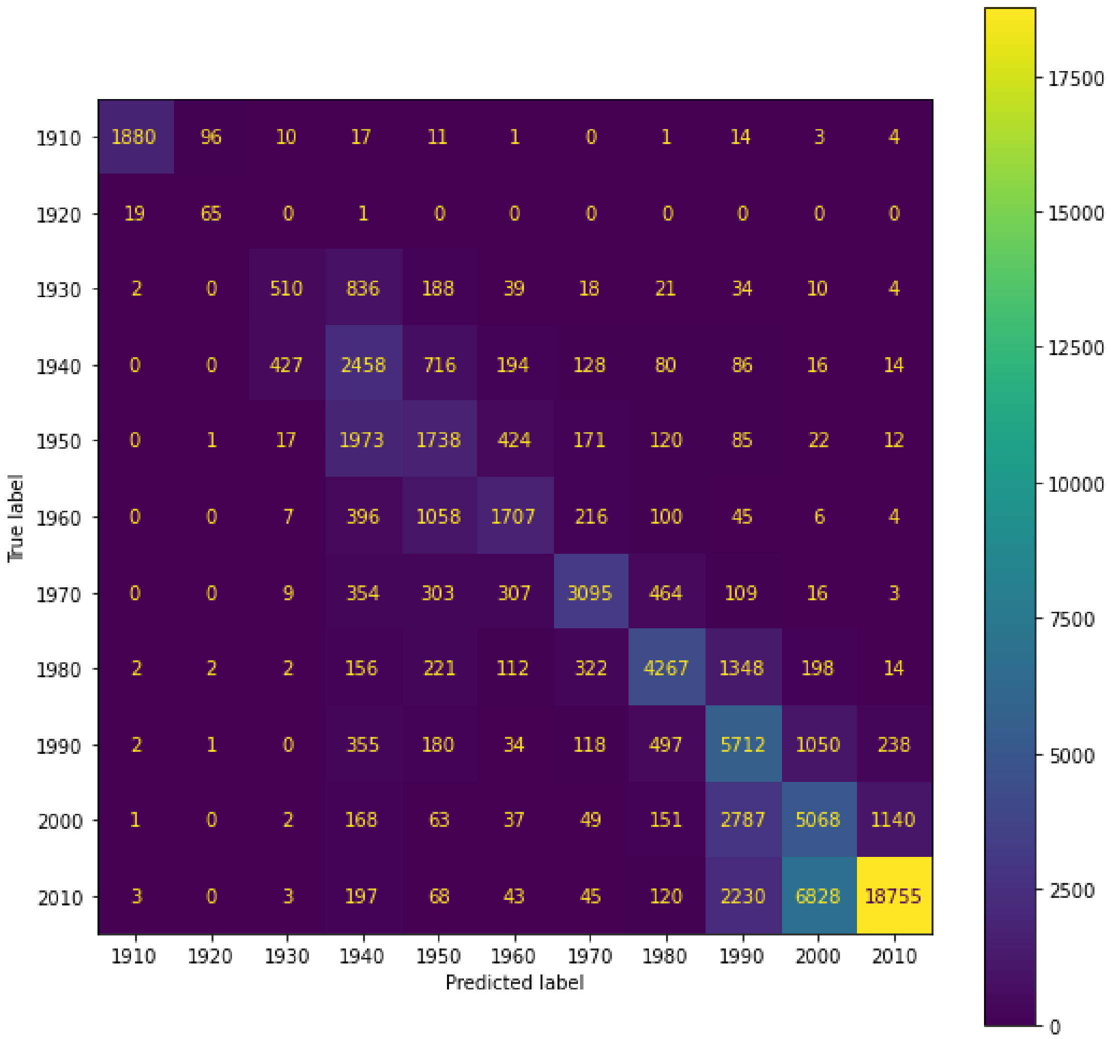
```
[50]: #add models to be evaluated
models = [
    grid_search_model_1,
    grid_search_model_2,
    grid_search_model_3,
    grid_search_model_4,
    grid_search_model_5,
    grid_search_model_6,
    grid_search_model_7
]

evaluation = pd.DataFrame(columns=["model"
                                   , "mean_fit_time", "accuracy"
                                   , "recall_macro", "recall_micro"
                                   , "precision_macro", "precision_micro"
                                   , "f1_macro", "f1_micro"
                                   , "model_definition"
                                  ])

i = -1 # Ensure that first item is index 0 in the loop
for model_ in models:
    # Yucky method of finding mean fit times:
    i = i +1
    mean_fit_time = cv_results.groupby("Model")["mean_fit_time"].mean()[i]

    # Predict
    preds = model_.best_estimator_.predict(file_contents_test)
    model = cv_results.iloc[model_.best_index_,0]

    # Calculate metrix
    to_append = [
            "Model " + str(i+1),
            mean_fit_time,
            accuracy_score(y_true=targets_test,y_pred=preds),
            #choose micro or macro according to criteria
            recall_score(y_true=targets_test,y_pred=preds, average="macro"),
            recall_score(y_true=targets_test,y_pred=preds, average="micro"),
            precision_score(y_true=targets_test,y_pred=preds, average="macro"),
            precision_score(y_true=targets_test,y_pred=preds, average="micro"),
            f1_score(y_true=targets_test,y_pred=preds, average="macro"),
            f1_score(y_true=targets_test,y_pred=preds, average="micro"),
            model_
            ]

    #Append Metrics
    evaluation_length = len(evaluation)
    evaluation.loc[evaluation_length] = to_append
    # Print results and Confusion Matrix for each model
   ␣
↪print("####################################################################")
   ␣
↪print("####################################################################")
    print("                        Model "+ str(i+1) + ":")
   ␣
↪print("--------------------------------------------------------------------")

    print(evaluation.loc[i, 'mean_fit_time':'f1_macro'])
    print("\n")
    print("Pipeline: ")
    print(model_.best_estimator_)

    print("\n")
    print("Confusion Matrix: ")
    fig, ax = plt.subplots(figsize=(10, 10))
    plot_confusion_matrix(estimator=model_.best_estimator_
                          , X=file_contents_test
                          , y_true=targets_test
                          , ax=ax
                         )
    plt.show()

# Print table of the models compared and sorted:
evaluation.sort_values(by="accuracy", ascending = False)
```

```
####################################################################
####################################################################
                        Model 1:
--------------------------------------------------------------------
mean_fit_time       8.04884
accuracy           0.580828
recall_macro       0.582249
recall_micro       0.580828
precision_macro    0.540098
precision_micro    0.580828
f1_macro           0.537704
Name: 0, dtype: object


Pipeline:
Pipeline(steps=[('join', <__main__.JoinElement object at 0x7fab1fd0be50>),
                ('tfidf', TfidfVectorizer()),
                ('k_best',
                 SelectKBest(k=500,
                             score_func=<function chi2 at 0x7fac4119dc10>)),
                ('MNB', MultinomialNB(alpha=0.5, fit_prior=False))])


Confusion Matrix:
```



```
####################################################################
####################################################################
                        Model 2:
--------------------------------------------------------------------
mean_fit_time      49.922866
accuracy            0.617782
recall_macro        0.605168
recall_micro        0.617782
precision_macro     0.588218
precision_micro     0.617782
```

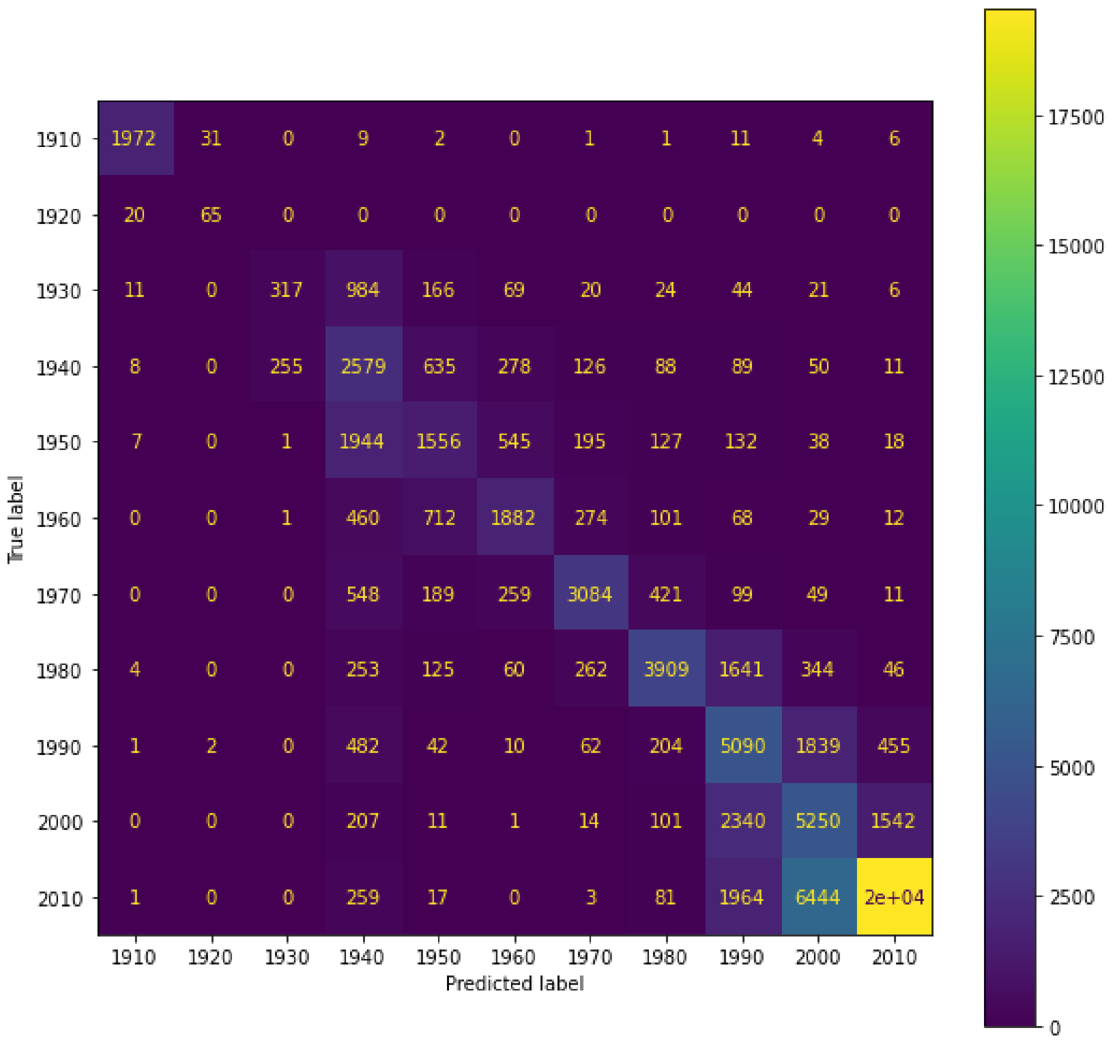
```
f1_macro            0.579638
Name: 1, dtype: object


Pipeline:
Pipeline(steps=[('join', <__main__.JoinElement object at 0x7faaed06afa0>),
                ('tfidf', TfidfVectorizer(smooth_idf=False)),
                ('k_best',
                 SelectKBest(k=500,
                             score_func=<function chi2 at 0x7fac4119dc10>)),
                ('SVC', SVC(C=100, gamma=1, kernel='linear'))])


Confusion Matrix:
```



	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010
1910	1880	96	10	17	11	1	0	1	14	3	4
1920	19	65	0	1	0	0	0	0	0	0	0
1930	2	0	510	836	188	39	18	21	34	10	4
1940	0	0	427	2458	716	194	128	80	86	16	14
1950	0	1	17	1973	1738	424	171	120	85	22	12
1960	0	0	7	396	1058	1707	216	100	45	6	4
1970	0	0	9	354	303	307	3095	464	109	16	3
1980	2	2	2	156	221	112	322	4267	1348	198	14
1990	2	1	0	355	180	34	118	497	5712	1050	238
2000	1	0	2	168	63	37	49	151	2787	5068	1140
2010	3	0	3	197	68	43	45	120	2230	6828	18755

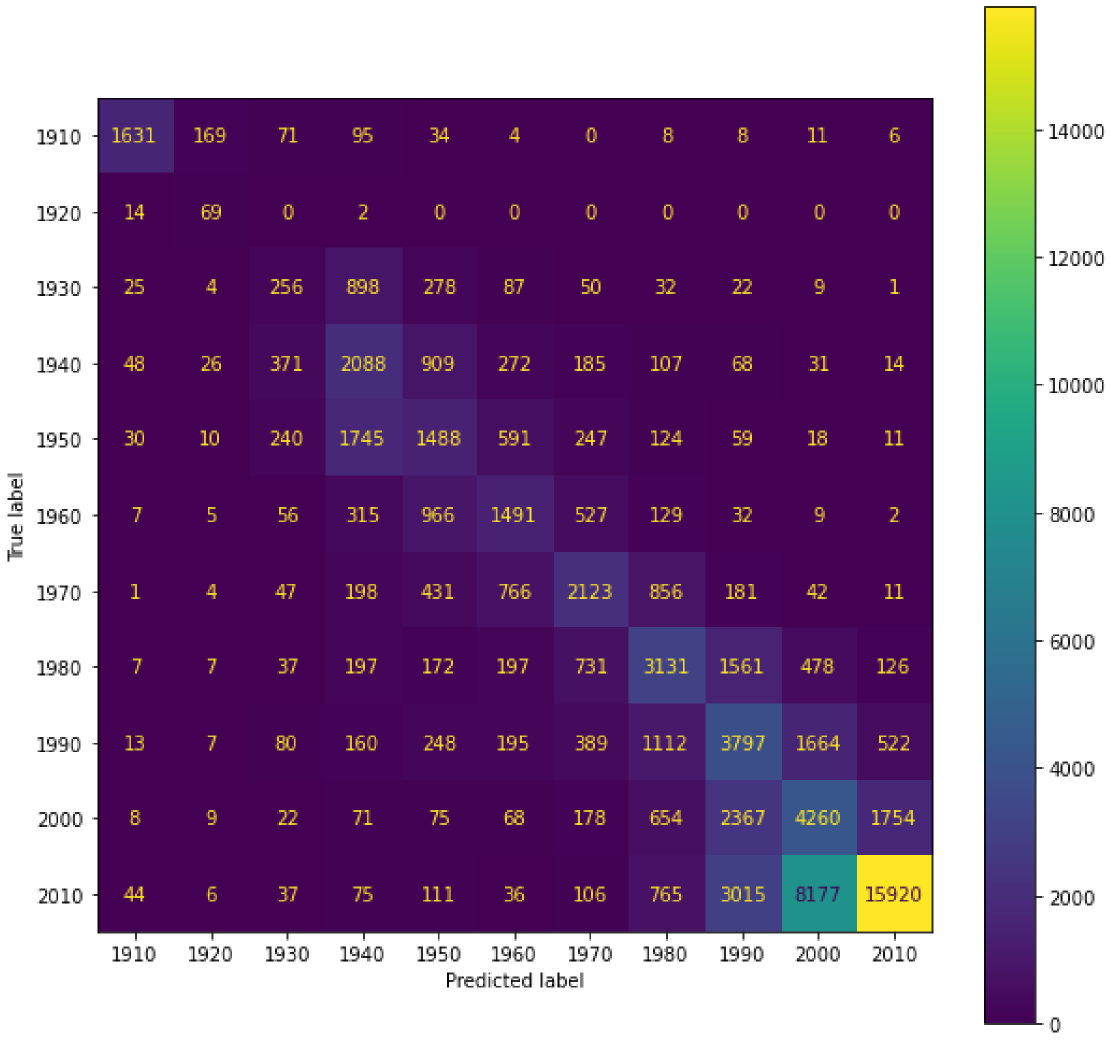
```
######################################################################
```

```
####################################################################
                        Model 3:
--------------------------------------------------------------------
mean_fit_time      13.328344
accuracy              0.6174
recall_macro        0.594453
recall_micro          0.6174
precision_macro     0.621433
precision_micro       0.6174
f1_macro            0.588941
Name: 2, dtype: object


Pipeline:
Pipeline(steps=[('join', <__main__.JoinElement object at 0x7fab1fd064f0>),
                ('tfidf', TfidfVectorizer(smooth_idf=False)),
                ('k_best',
                 SelectKBest(k=500,
                             score_func=<function chi2 at 0x7fac4119dc10>)),
                ('clf',
                 RandomForestClassifier(max_features=10, min_samples_split=10,
                                        n_estimators=200))])


Confusion Matrix:
```



```
####################################################################
####################################################################
                        Model 4:
--------------------------------------------------------------------
mean_fit_time      320.122003
accuracy             0.494908
recall_macro         0.493105
recall_micro         0.494908
precision_macro      0.437746
precision_micro      0.494908
f1_macro              0.44268
Name: 3, dtype: object


Pipeline:
```

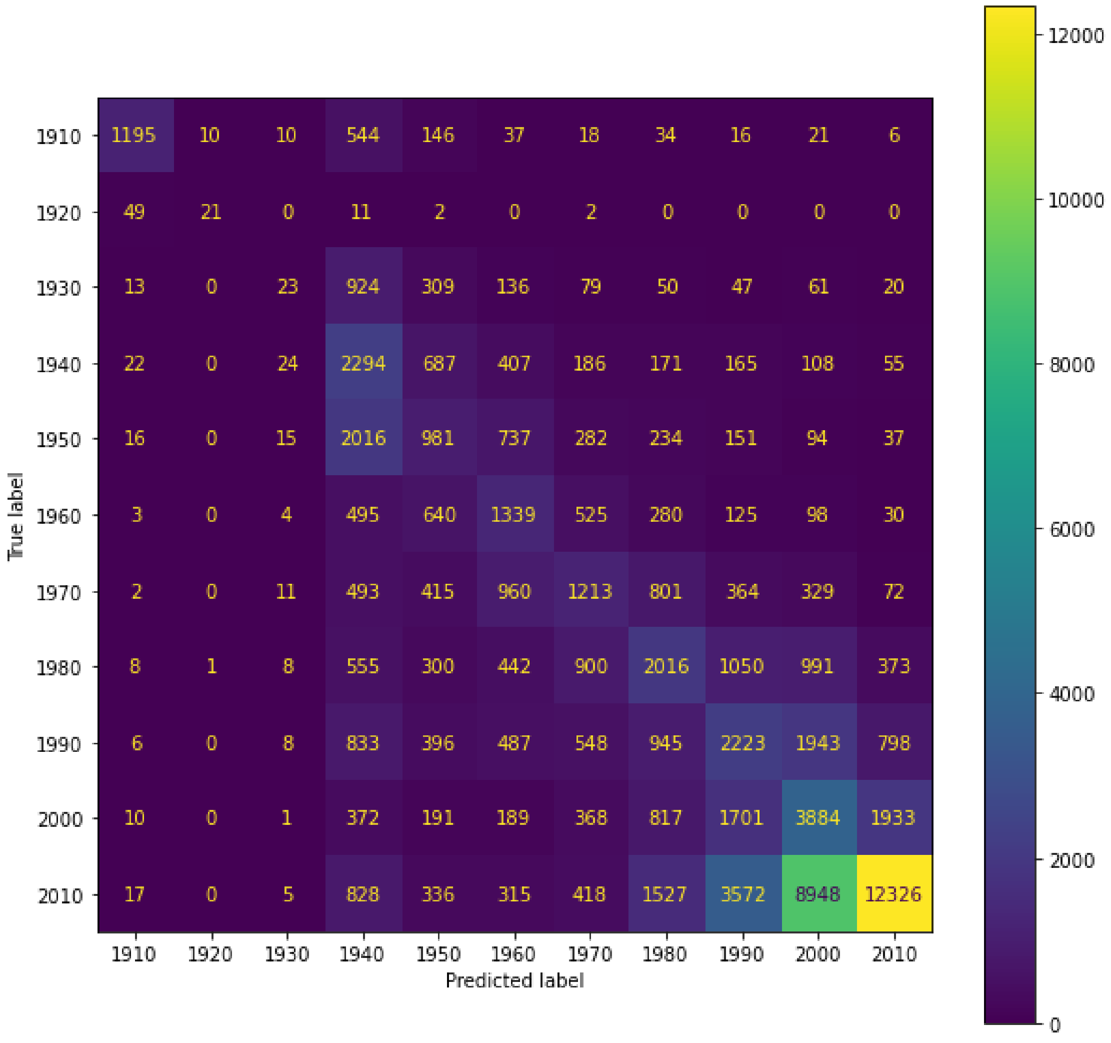
```
Pipeline(steps=[('word2vec', MeanEmbeddingVectorizer()),
                ('SVC', SVC(C=10, gamma=0.1, kernel='poly'))])


Confusion Matrix:
```



	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010
1910	1631	169	71	95	34	4	0	8	8	11	6
1920	14	69	0	2	0	0	0	0	0	0	0
1930	25	4	256	898	278	87	50	32	22	9	1
1940	48	26	371	2088	909	272	185	107	68	31	14
1950	30	10	240	1745	1488	591	247	124	59	18	11
1960	7	5	56	315	966	1491	527	129	32	9	2
1970	1	4	47	198	431	766	2123	856	181	42	11
1980	7	7	37	197	172	197	731	3131	1561	478	126
1990	13	7	80	160	248	195	389	1112	3797	1664	522
2000	8	9	22	71	75	68	178	654	2367	4260	1754
2010	44	6	37	75	111	36	106	765	3015	8177	15920

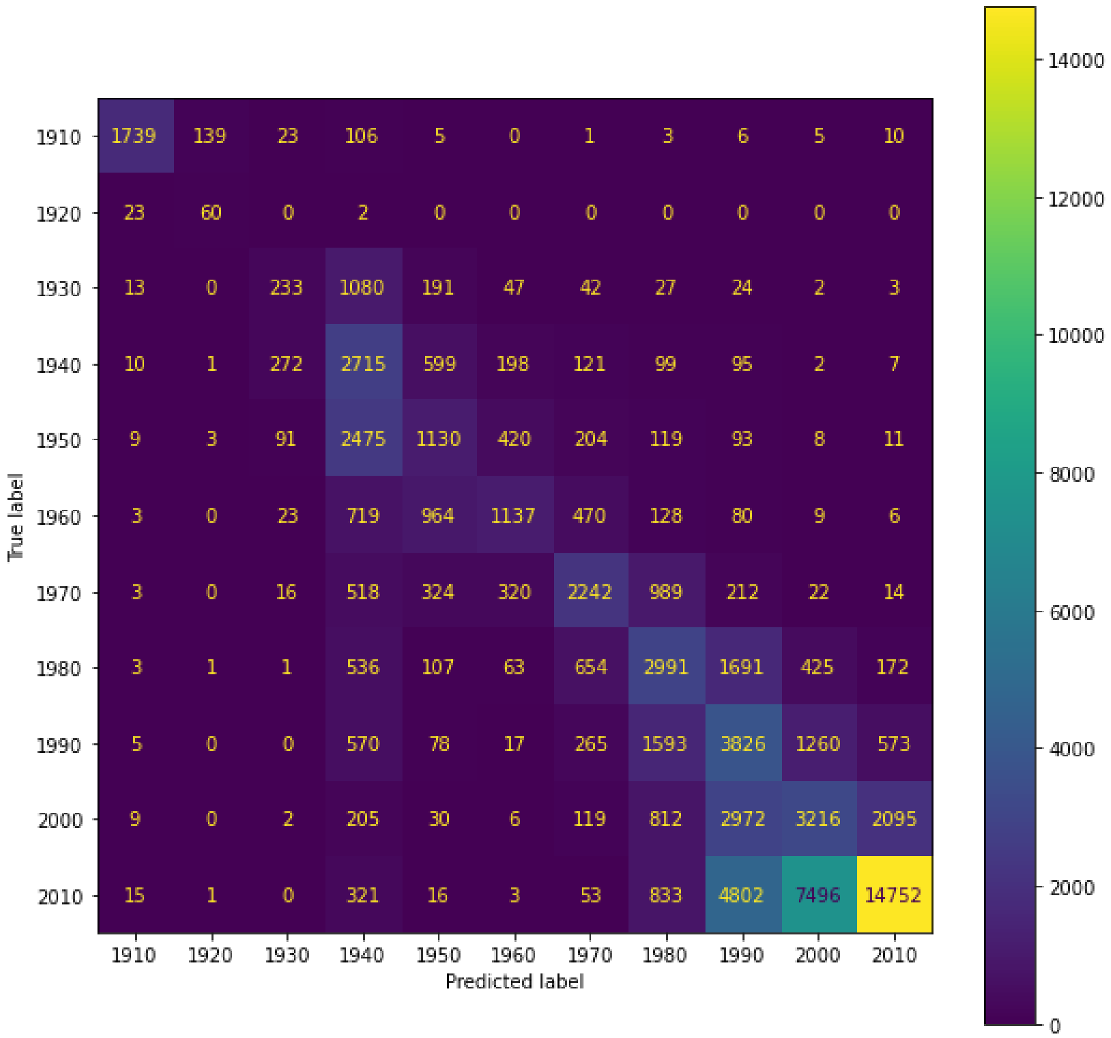
```
####################################################################
####################################################################
                        Model 5:
--------------------------------------------------------------------
mean_fit_time      148.909811
accuracy             0.375611
recall_macro          0.33446
recall_micro         0.375611
precision_macro      0.391935
precision_micro      0.375611
```

```
f1_macro             0.330763
Name: 4, dtype: object


Pipeline:
Pipeline(steps=[('word2vec', MeanEmbeddingVectorizer()),
                ('clf',
                 RandomForestClassifier(min_samples_split=10,
                                        n_estimators=200))])


Confusion Matrix:
```



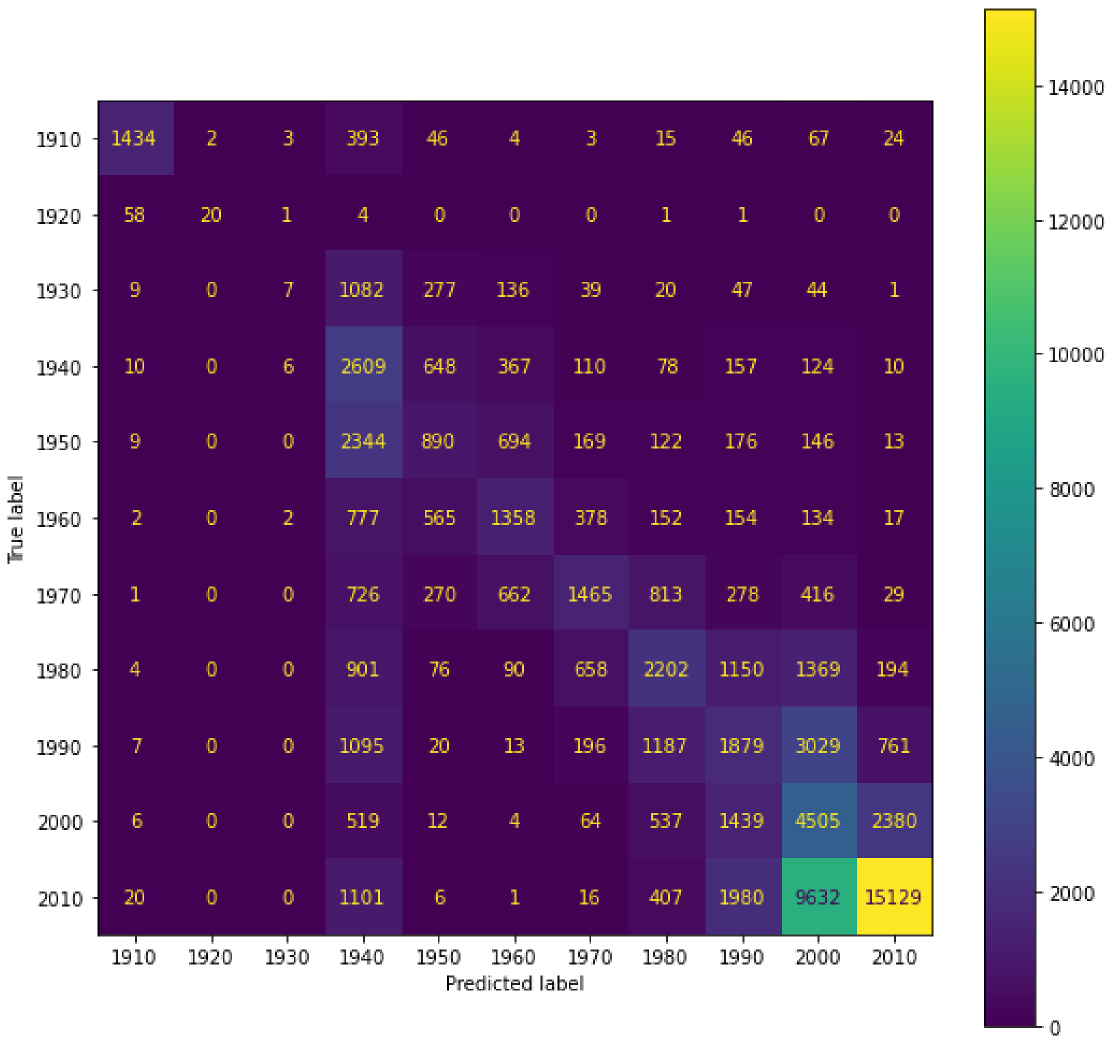
```
######################################################################
######################################################################
                        Model 6:
```

```
----------------------------------------------------------------------
mean_fit_time      276.770212
accuracy             0.465367
recall_macro         0.475932
recall_micro         0.465367
precision_macro      0.463425
precision_micro      0.465367
f1_macro             0.442343
Name: 5, dtype: object


Pipeline:
Pipeline(steps=[('doc2vec', Doc2Vectorizer()),
                ('SVC', SVC(C=10, gamma=0.001, kernel='linear'))])


Confusion Matrix:
```



True label \ Predicted label	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000	2010
1910	1739	139	23	106	5	0	1	3	6	5	10
1920	23	60	0	2	0	0	0	0	0	0	0
1930	13	0	233	1080	191	47	42	27	24	2	3
1940	10	1	272	2715	599	198	121	99	95	2	7
1950	9	3	91	2475	1130	420	204	119	93	8	11
1960	3	0	23	719	964	1137	470	128	80	9	6
1970	3	0	16	518	324	320	2242	989	212	22	14
1980	3	1	1	536	107	63	654	2991	1691	425	172
1990	5	0	0	570	78	17	265	1593	3826	1260	573
2000	9	0	2	205	30	6	119	812	2972	3216	2095
2010	15	1	0	321	16	3	53	833	4802	7496	14752

```
####################################################################
####################################################################
                        Model 7:
--------------------------------------------------------------------
mean_fit_time      241.609706
accuracy             0.430543
recall_macro         0.364123
recall_micro         0.430543
precision_macro      0.474474
precision_micro      0.430543
f1_macro             0.366073
Name: 6, dtype: object


Pipeline:
Pipeline(steps=[('doc2vec', Doc2Vectorizer()),
                ('clf',
                 RandomForestClassifier(min_samples_split=40,
                                        n_estimators=200))])


Confusion Matrix:
```



```
[50]:     model  mean_fit_time  accuracy  recall_macro  recall_micro  \
      1  Model 2      49.922866  0.617782      0.605168      0.617782
      2  Model 3      13.328344  0.617400      0.594453      0.617400
      0  Model 1       8.048840  0.580828      0.582249      0.580828
      3  Model 4     320.122003  0.494908      0.493105      0.494908
      5  Model 6     276.770212  0.465367      0.475932      0.465367
      6  Model 7     241.609706  0.430543      0.364123      0.430543
      4  Model 5     148.909811  0.375611      0.334460      0.375611

         precision_macro  precision_micro  f1_macro  f1_micro  \
      1         0.588218         0.617782  0.579638  0.617782
      2         0.621433         0.617400  0.588941  0.617400
      0         0.540098         0.580828  0.537704  0.580828
      3         0.437746         0.494908  0.442680  0.494908
      5         0.463425         0.465367  0.442343  0.465367
```

```
6        0.474474        0.430543  0.366073  0.430543
4        0.391935        0.375611  0.330763  0.375611

                                  model_definition
1  GridSearchCV(cv=5,\n           estimator=Pip…
2  GridSearchCV(cv=5,\n           estimator=Pip…
0  GridSearchCV(cv=5,\n           estimator=Pip…
3  GridSearchCV(cv=5,\n           estimator=Pip…
5  GridSearchCV(cv=5,\n           estimator=Pip…
6  GridSearchCV(cv=5,\n           estimator=Pip…
4  GridSearchCV(cv=5,\n           estimator=Pip…
```

The **best estimator we found after everything is:**

Main **conclusion**: for unseen data, we would choose to use the estimator from above.

4 Conclusion

Draw conclusions from above. but add more evaulations and stuff…. Should probably be done after proper train/test is defined, and full model is tried

Some observations: * Model 1,2,3 (TFIDF) seems to work best overall, no matter training size * Model 5,6,7 (Doc2Vec) takes by far the most time, but their accuracy greatly increased when increasing training set

As often is the case in the fields of science, not all research leads to useable results. We ended up having to remodel our plans several times during this project, including a complete pivot of the datasets.

This did however give us some insight into how larger projects are managed. This also lead us to an interesting path of looking at a relatively obscure language.

Although further works is possible, we reached the conclusion that there is a change in the Icelandic spoken language throughout time, and it is therefore possible to train models that estimates which decade a given speech is from. However, take into account what explained in section 3.2: it may be also due to other factors (for instance, topic used).

Overall we did work with Data-Oriented Programming best practices. We were able to develop a scientific workflow. From the given data, we managed to train a model for prediction on test data with **decent results.**

5 Further Works

As we drilled down this dataset, we kept getting new ideas that we would like to experiment with, and try to gain better insight. Specifically, our next steps would be:

5.1 Predict Different Sources

As it currently stands, we are trying to estimate a decade of speeches from "Althingi". However, the dataset has several other sources of Icelandic; both written and spoken (TV scripts, cinema and others).

We would like to see if it was possible to extend our model to be able to classify the source.

5.2 Treating years as Contious Variables

We are currently treating decades as a class. By discretizing results from a regression algorithm, we think it should be possible to keep some nominal knowledge of the ordering of the years, and thus improving our predictions. It would be also interesting to see if we can achieve also decent prediction by narrowing a bit the intervals for the years (instead of decades, lustrums). And of course, it would be interesting to rerun the model in a more powerful machine using all the decades instead of discarding the intermediate ones.

5.3 Gaining insight into Explanatory Variables

From our results, it is clear that it is somewhat possible to predict the decades. However, we are still treating the algorithms as "Black Boxes". We would like to dive deeper into the decision treas/boundaries, to see if we can locate what it is that makes the predictions possible. It might be new words introduced, semantic changes, or something entirely different.

5.4 Additional Feature Extraction and Classifiers

We would like to extend the list to include more classifiers, as well as trying to develop some additional feature extractions. E.g. "Glove Embedding" E.g. "Neural networks"