DMT2023_HW3

May 20, 2023

0.1 Group composition:

————YOUR TEXT STARTS HERE————

Mignella, Laura, 1920520

Vestini, Maria Vittoria, 1795724

0.2 Homework 3

The homework consists of two parts:

1. Dimensionality Reduction

and

2. Supervised Learning

   Ensure that the notebook can be faithfully reproduced by anyone (hint: pseudo random number generation).

   If you need to set a random seed, set it to `160`.

1 Part 1

In this part of the homework, you have to deal with Dimensionality Reduction.

[ ]:
```
#REMOVE_OUTPUT#
!pip install --upgrade --no-cache-dir gdown
from bs4 import BeautifulSoup
#YOUR CODE STARTS HERE#
from gensim import corpora
from gensim.models import LsiModel
from gensim.models.coherencemodel import CoherenceModel
import nltk
nltk.download('stopwords')
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
import re
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time

#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

1.1 Part 1.1

The data you need to process comes from the book *Le Morte D'Arthur* by Thomas Malory.

You have to carry out Topic Modeling on book chapters.

The goal is to achieve a topic division within the following limits:

- The total computation may not exceed 10 minutes (starting from Part 1.1.5; Parts 1.1.1 to 1.1.4 are not considered for time calculation)
- The division into topics must be the "best one"

1.1.1 1.1.1

Download the data from the Drive link (code already provided).

[ ]:
```
#REMOVE_OUTPUT#
!gdown 1zHgvidy9FvhZvE68SOmXWkoF-hHMpiUL
!gdown 1VjpTkFcbfaLIi4TXVafokW9e_bvGnfut
```

1.1.2 1.1.2

Parse the HTML. **Part** of code already provided: follow the comments to complete the code.

[3]:
```
with open('The Project Gutenberg eBook of Le Morte D’Arthur, Volume I (of II),␣
 ↪by Thomas Malory.html') as fp:
    vol1 = BeautifulSoup(fp, 'html.parser')
with open('The Project Gutenberg eBook of Le Morte D’Arthur, Volume II (of II),␣
 ↪by Thomas Malory.html') as fp:
    vol2 = BeautifulSoup(fp, 'html.parser')

def clean_text(txt):
    words_to_put_space_before = [".",",",";",":","’","'"]
    words_to_lowercase =␣
 ↪["First","How","Some","Yet","Of","A","The","What","Fifth"]

    app = txt.replace("\n"," ")
    for word in words_to_put_space_before:
        app = app.replace(word," "+word)
    for word in words_to_lowercase:
        app = app.replace(word+" ",word.lower()+" ")
    return app.strip()

def parse_html(soup):
    titles = []
    texts = []
    for chapter in soup.find_all("h3"):
        chapter_title = chapter.text
        if "CHAPTER" in chapter_title:
            chapter_title = clean_text("".join(chapter_title.split(".")[1:]))
            titles.append(chapter_title)

            chapter_text = [p.text for p in chapter.findNextSiblings("p")]
            chapter_text = clean_text(" ".join(chapter_text))
            texts.append(chapter_text)
    return titles, texts
```

[4]:
```
#YOUR CODE STARTS HERE#
#Extract all the chapters' titles and texts from the two volumes
```

```
# Extract the titles and texts from both the volumes using the parse_html␣
 ↪function
titles, texts = parse_html(vol1)
titles_1, texts_1 = parse_html(vol2)

# Append together the lists
titles = titles + titles_1
texts = texts + texts_1

#Transform the list into a pandas DataFrame.

# Create the dataframe with columns titles and texts
data_frame_book = pd.DataFrame(data = {'titles':titles, 'texts': texts})



#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

[5]:
```
#YOUR CODE STARTS HERE#


# Show only the last 8 rows of the dataframe
data_frame_book.tail(8)



#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

[5]:
```
                                                titles  \
495  how Sir Bedivere found him on the morrow dead …
496  of the opinion of some men of the death of Kin…
497  how when Sir Lancelot heard of the death of Ki…
498  how Sir Launcelot departed to seek the Queen G…
499  how Sir Launcelot came to the hermitage where …
500  how Sir Launcelot went with his seven fellows …
501  how Sir Launcelot began to sicken , and after …
502  how Sir Ector found Sir Launcelot his brother …

                                                 texts
495  Then was Sir Bedivere glad , and thither he we…
496  yet some men say in many parts of England that…
497  And when he heard in his country that Sir Mord…
498  Then came Sir Bors de Ganis , and said : My lo…
499  But sithen I find you thus disposed , I ensure…
500  Then Sir Launcelot rose up or day , and told t…
```

501 Then Sir Launcelot never after ate but little …
502 And when Sir Ector heard such noise and light …

1.1.3 1.1.3

Extract character's names from the **titles** only. **Part** of code already provided: follow the comments to complete the code.

[6]:
```
all_characters = set()
def extract_character_names_from_string(string_to_parse):
    special_tokens = ["of","the","le","a","de"]

    remember = ""
    last_is_special_token = False

    tokens = string_to_parse.split(" ")
    characters_found = set()
    for i,word in enumerate(tokens):
        if word[0].isupper() or (remember!="" and word in special_tokens):
            #word = word.replace("'s","").replace("’s","")
            last_is_special_token = False
            if remember!="":
                if word in special_tokens:
                    last_is_special_token = True
                remember = remember+" "+word
            else: remember = word
        else:
            if remember!="":
                if last_is_special_token:
                    for tok in special_tokens:
                        remember = remember.replace(" "+tok,"")
                characters_found.add(remember)
            remember = ""
            last_is_special_token = False
    return characters_found

#all_characters = set([x for x in all_characters if x[-2:]!="'s"])
```

[7]:
```
#YOUR CODE STARTS HERE#
#Extract all characters' names



# Iterate over the titles column
for title in data_frame_book.titles:
    # Extract all characters' names using the function␣
 ↪"extract_character_names_from_string"
    # And save everything in the all_characters set using the "union" method of␣
 ↪the sets
    all_characters = all_characters.
 ↪union(extract_character_names_from_string(title))
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 15#
```

[8]:
```
#YOUR CODE STARTS HERE#

# Iterate over the characters names
for name in all_characters:
    # Print the name if the string "Sir" is in it
    if 'Sir' in name:
        print(name)

#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

```
Sir Beaumains
Sir Berluse
Sir Agravaine
Sir Persant of Inde
Sir Bors
Sir Bleoberis
Sir Lanceor
Sir Suppinabiles
Sir Mador
Sir Sagramore le Desirous
Sir Frol
Sir Breuse Saunce Pité
Sir Malgrin
Sir Alisander
Sir Pedivere
Sir Meliagrance
Sir Nabon
Sir Accolon of Gaul
Sir Anguish
Sir Pervivale
Sir Tor
Sir Uwaine
Sir Lamorak de Galis
Sir Mordred
Sir Turquine
Sir Urre
Sir Sadok
Sir Dinadan
Sir Breunor
Sir Safere
```

Sir Archade
Sir Accolon
Sir Marhaus
Sir Carados
Sir Bedivere
Sir Persant
Sir Kay
Sir Pelleas
Sir Bliant
Sir Lancelot
Sir Gareth
Sir Gaheris
Sir Brian
Sir Aglovale
Sir Galahad
Sir Epinogris
Sir Segwarides
Sir Colgrevance
Sir Launcelot
Sir Lavaine
Sir Belliance
Sir Percivale
Sir Lamorak
Sir Blamore
Sir Tristram de Liones
Sir Galahalt
Sir Tristram
Sir Elias
Sir Amant
Sir Ector
Sir Lionel
Sir Palomides
Sir Dagonet
Sir Gawaine
Sir Galihodin
Sir Uriens
Sir Meliagaunce

1.1.4 1.1.4

Preprocess the data

Consider only the titles

Each document must be a list of terms

Discard documents that have less than 10 (non-unique) words before the preprocessing (split by whitespace, ignore punctuation)

After preprocessing, each document must be represented by at least 5 tokens

- Several preprocessing options are possible

[9]:
```
#YOUR CODE STARTS HERE#

tokenizer = RegexpTokenizer(r'\w+')
# the function tokenize split the titles into tokens based only on words
# ignoring punctuation. The tokenized data are inserted into the column␣
↪'clean_titles'
data_frame_book['clean_titles'] = data_frame_book.titles.apply(lambda x:␣
↪tokenizer.tokenize(x))

# drop titles that have less than 10 words before preprocessing
index = [x for x, t in enumerate(data_frame_book['clean_titles']) if len(t)<10]
data_book = data_frame_book.drop(index)

# we'll use our version of the function seen during the lab5
def preprocess_data(doc_set):
  # list of english stopwords from nltk
  en_stop = set(stopwords.words('english'))
  # let's remove the stopwords 'of' and 'for' from en_stop list in order to␣
↪have
  # each document represented by at least 5 tokens
  en_stop = en_stop.difference({'of', 'for'})
  # create p_stemmer of class PorterStemmer
  p_stemmer = PorterStemmer()

  processed_tokenized_texts = []
  for text in doc_set: # loop through document list
    cleaned_text = text.lower()
    tokenized_text = cleaned_text.split(" ") # divide text in tokens
    stopped_tokens = [token for token in tokenized_text if not token in␣
↪en_stop] # remove stop words from tokens
    stemmed_tokens = [p_stemmer.stem(token) for token in stopped_tokens] # stem␣
↪tokens
    processed_tokenized_texts.append(stemmed_tokens) # add tokens to list
  return processed_tokenized_texts
```

```
# joining the tokens in the column 'clean_titles'
doc_set = [' '.join(x) for x in data_book.clean_titles]
# preprocessing our data
prep_data = preprocess_data(doc_set)
# enter the preprocessed data in the column 'clean_titles'
data_book['clean_titles'] = prep_data
print('All the documents have at least', min([len(x) for x in data_book.
  ↪clean_titles]), 'tokens.')

#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

All the documents have at least 5 tokens.

[10]:
```
#YOUR CODE STARTS HERE#
# preprocessing also the word 'Bedivere'
clean_term = preprocess_data(['Bedivere'])[0][0]
# looking for the preprocessed word in the column 'clean_titles'
for i, row in data_book.iterrows():
  if clean_term in row['clean_titles']:
    # print all the titles in which the term 'Bedivere' appears
    print(row['titles'])
#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

how Sir Bedivere found him on the morrow dead in an hermitage , and how he abode there with the hermit

1.1.5 1.1.5

Build a dictionary of the terms in the documents.

[11]:
```
#YOUR CODE STARTS HERE#

start = time.time()




# dictionary containing all the terms in the documents
dictionary = corpora.Dictionary(data_book.clean_titles)











#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

[12]:
```
#YOUR CODE STARTS HERE#
all_titles = []
# put all the title in a list
for title in data_book.clean_titles:
  all_titles = all_titles + title
term_freq = dictionary.doc2bow(all_titles) # compute the frequency of the words
for term in sorted(term_freq, key=lambda x: x[1], reverse=True)[0:5]:
  print(dictionary[term[0]], term[1]) # print the 5 most common terms
#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

```
sir 589
of 387
king 175
launcelot 148
tristram 130
```

1.1.6 1.1.6

Perform a document-term encoding of the dataset.

- Several encodings are possible

[13]:
```
#YOUR CODE STARTS HERE#




# document-term encoding of the dataset using doc2bow
doc_term_matrix = [dictionary.doc2bow(doc) for doc in data_book.clean_titles]













#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

[14]:
```
#YOUR CODE STARTS HERE#
sparcity = 0
# count the zeros for each row in doc_term_matrix
for row in doc_term_matrix:
  sparcity += len(dictionary) - len(row)
# compute the sparcity
print('The full matrix sparcity would be:',
      sparcity/(len(dictionary)*len(doc_term_matrix)))
#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

```
The full matrix sparcity would be: 0.9886901504459614
```

1.1.7 1.1.7

Perform Latent Semantic Analysis for at least 5 different numbers of topics.

[15]:
```
#YOUR CODE STARTS HERE#

# choosing possible numbers of topics
possible_numbers_of_topics = range(2,51)
lsa_models = []

# train the model for all the possible numbers of topics
for number_of_topics in possible_numbers_of_topics:
  lsa_models.append(LsiModel(doc_term_matrix, num_topics=number_of_topics,
                             id2word = dictionary, random_seed=160))









#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

1.1.8 1.1.8

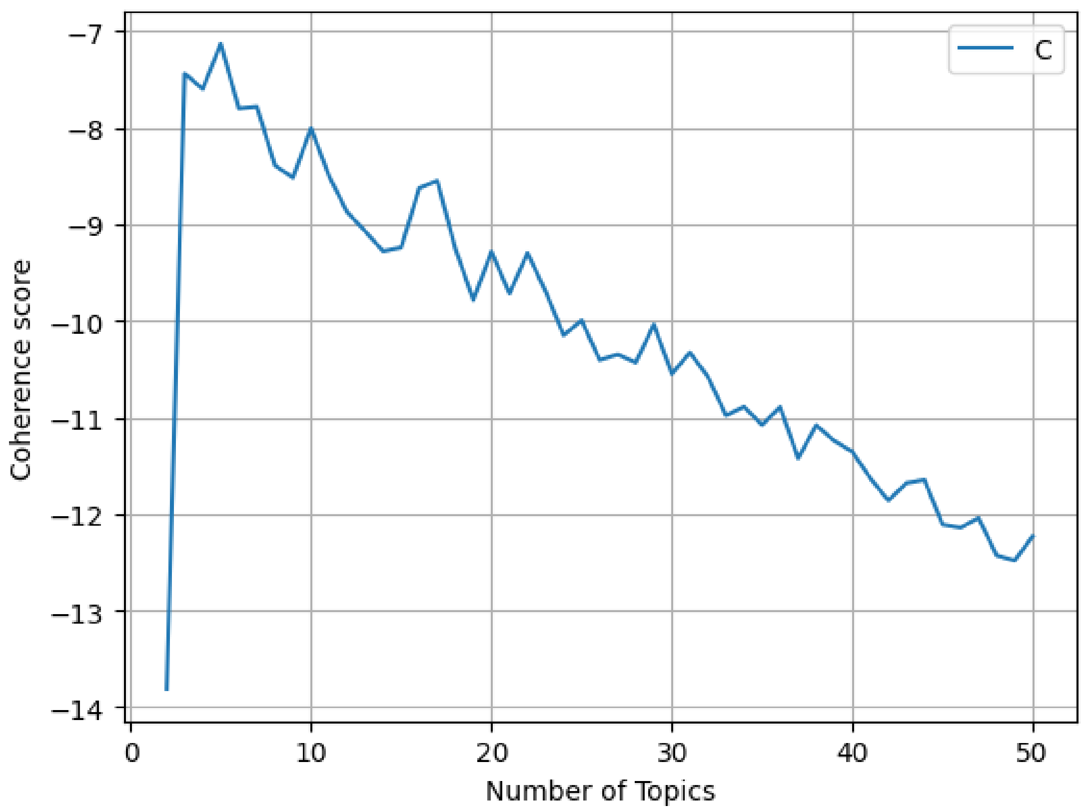
For each of the calculations above, calculate a measure of the “goodness” of the division into topics.

[16]:
```
#YOUR CODE STARTS HERE#

coherence_values = []


# calculate the coherence score for each model using 'u_mass'
for i, number_of_topics in enumerate(possible_numbers_of_topics):
  coherence_model = CoherenceModel(model=lsa_models[i], texts=data_book.
 ↪clean_titles,
                                   dictionary=dictionary, coherence='u_mass')
  coherence_values.append(coherence_model.get_coherence())
```

```
#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

[17]:

```
#YOUR CODE STARTS HERE#

# plot the coherence score with respect to the possible_numbers_of_topics used
# to train the LsiModel
plt.plot(possible_numbers_of_topics, coherence_values)
plt.xlabel("Number of Topics")
plt.ylabel("Coherence score")
plt.legend(("Coherence values"), loc='best')
plt.grid()
plt.show()


print(f'\n\nFinal time: {round(time.time()-start,3)} seconds.')






#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```



```
Final time: 5.031 seconds.
```

————-YOUR TEXT STARTS HERE————-

The best number of topics to model this dataset is 5. As we can see from the plot, when the number of topics is 5 the model had the highest coherence score.

1.1.9 1.1.9

Print the 10 most important words for the 5 most important topics.

[18]:
```python
#YOUR CODE STARTS HERE#
# get the coherence score of the single topics
coherence_model = CoherenceModel(model=lsa_models[3], texts=data_book.
 ↪clean_titles,
                                   dictionary=dictionary, coherence='u_mass')
single_scores = coherence_model.get_coherence_per_topic()
# select the five with the highest score
best_topics = sorted(zip(range(len(single_scores)),single_scores), key = lambda␣
 ↪x: x[1], reverse = True)[0:5]
# show the content of the topics as seen in lab5
number_of_topics = 5
for topic_i,words_and_importance in lsa_models[-1].
 ↪print_topics(num_topics=number_of_topics, num_words=10):
   if topic_i in [i for i,x in best_topics]:
    print("TOPIC:",topic_i)
    for app in words_and_importance.split(" + "):
      value,token = app.split("*")
      value = float(value)
      token = str(token.replace('"',""))
      print("\t",value,token)
    print()
#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

```
TOPIC: 0
         0.81 sir
         0.413 of
         0.184 launcelot
         0.179 tristram
         0.156 king
         0.099 knight
         0.089 for
         0.086 arthur
         0.08 palomid
         0.077 came

TOPIC: 1
         0.769 of
         -0.46 sir
         0.284 king
         0.188 arthur
         -0.133 tristram
         0.11 knight
         -0.078 palomid
```

```
         0.067 came
         0.059 made
         0.051 for

TOPIC: 2
         0.753 king
         0.455 arthur
         -0.404 of
         0.128 mark
         0.061 tristram
         0.054 for
         0.053 came
         -0.045 launcelot
         0.037 court
         -0.035 queen

TOPIC: 3
         -0.594 tristram
         0.531 launcelot
         0.392 knight
         -0.162 isoud
         -0.154 of
         -0.14 palomid
         0.139 queen
         -0.122 beal
         0.102 came
         -0.093 la

TOPIC: 4
         0.746 knight
         0.311 tristram
         -0.249 launcelot
         0.247 for
         0.192 fought
         0.161 ladi
         -0.107 of
         -0.103 sir
         0.103 slew
         -0.099 king
```

———-YOUR TEXT STARTS HERE———-

Given the computed coherence score we decide to select as the most important topics the ones with the best scores.

———-YOUR TEXT STARTS HERE———-

Looking at the topics shown we can recognize some 'well' define topics for the first 2: - Topic 0: knights - Topic 1: king Arthur

As for other three we are having some difficulties extracting themes different from the previous ones, we belive this is due to the fact that the document come from the same book.

1.2 Part 1.2

1.2.1 1.2.1

Suppose you have a dataset with N samples and M features.

You only have B units of memory available on your storage medium.

Assume further that each feature occupies a constant number b of memory units and that this cannot be changed (e.g. you cannot change the precision of floats).

Assuming that the entire dataset cannot fit on your storage medium, how would you accommodate all N samples while retaining as much information about your data as possible?

Use at most 3 sentences.

———-YOUR TEXT STARTS HERE———-

The best way to solve this issue is to perform some kind of dimensionality reduction method like PCA.

In this specific case, since we have N samples and each feature occupies b memory, if we select $K < M$ principal components then we will need $N \cdot (b \cdot K)$ memory to store the result.

So the maximum number of principal components that we can take, to fit the whole dataset in the storage medium, is $K = \lfloor \frac{B}{b \cdot N} \rfloor$.

2 Part 2

In this part, your goal is to obtain the best classification on a dataset according to a metric specified in each section.

[ ]:
```
#REMOVE_OUTPUT#
#YOUR CODE STARTS HERE#
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split #to split train/test
from sklearn.feature_extraction.text import TfidfVectorizer #to compute tf-idf
from sklearn.pipeline import Pipeline #pipeline
from sklearn.model_selection import GridSearchCV #to perform GridSearch
from sklearn import metrics #evaluate model
nltk.download('punkt')
from nltk import word_tokenize
from nltk.stem.snowball import EnglishStemmer
import time
#YOUR CODE ENDS HERE#
#THIS IS LINE 15#
```

2.1 Part 2.1

In this part, you will perform a tf-idf encoding of the data, and then train a classifier, optimising its hyper-parameters.

In the various steps, we will slowly prepare a pipeline to perform a hyper-parameter optimisation; try to prepare the required objects with this target in mind.

The goal is to maximise the accuracy on the test set.

2.1.1 2.1.1

Prepare the dataset for Supervised Learning.

It should be a Pandas DataFrame with two fields: `Text`, `Label`.

The `Text` column must contain the text of a chapter

The `Label` column must contain a value of 0 or 1

- The `Label` is 0 if the chapter is in Book 1
- The `Label` is 1 if the chapter is in Book 2

[20]:
```
#YOUR CODE STARTS HERE#

# Extract the titles and texts from both the volumes using the parse_html␣
↪function
titles, texts = parse_html(vol1)
titles_1, texts_1 = parse_html(vol2)
```

```python
# we create the label vector
Label = np.concatenate((np.zeros(len(titles)),np.ones(len(titles_1))))

# put everything together in a dataframe
data = pd.DataFrame()
data['Text'] = texts+texts_1
data['Label'] = Label
















#YOUR CODE ENDS HERE#
#THIS IS LINE 30#
```

[21]:

```python
#YOUR CODE STARTS HERE#



# showing the first and last 2 rows
print(data.iloc[[0,1,-2,-1]])









#YOUR CODE ENDS HERE#
#THIS IS LINE 15#
```

```
                                                  Text  Label
0    It befell in the days of Uther Pendragon , whe…    0.0
1    Then Ulfius was glad , and rode on more than a…    0.0
501  Then Sir Launcelot never after ate but little …    1.0
502  And when Sir Ector heard such noise and light …    1.0
```

2.1.2 2.1.2

Divide the dataset into training (68%), validation (17%) and test set (15%).

[22]:
```
#YOUR CODE STARTS HERE#

# we first divide in train + validation and test
train_x, test_x, train_y, test_y = train_test_split(data.Text, data.Label,␣
↪test_size=0.15, shuffle=True, random_state = 160)

# and then split the train and validation
train_x, val_x, train_y, val_y = train_test_split(train_x, train_y, test_size=0.
↪2, shuffle=True, random_state = 160)














#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

[23]:
```
#YOUR CODE STARTS HERE#

# show the percentages
print('The percentages are:\n -Train:', sum(train_y == 0)/len(train_y),
      '\n -Test:', sum(test_y == 0)/len(test_y),
      '\n -Validation:', sum(val_y == 0)/len(val_y))


#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

```
The percentages are:
 -Train: 0.4897360703812317
 -Test: 0.39473684210526316
 -Validation: 0.47674418604651164
```

2.1.3 2.1.3

Create an object that performs a tf-idf transformation on the data. The transformation must **NOT** lowercase character names.

Create a dictionary containing configurations for the tf-idf vectorizer. Each hyper-parameter should have exactly **3 values**.

[24]:
```python
#YOUR CODE STARTS HERE#

# define the models
vectorizer = TfidfVectorizer()
stemmer = EnglishStemmer()

# extract the english stopwords
english_stopwords = set(stopwords.words('english'))

# as seen during lab6 we define two function for the possible tokenizers
def stemming_tokenizer(text):
        stemmed_text = [stemmer.stem(word) for word in word_tokenize(text, language='english')]
        return stemmed_text

def stemming_stop_tokenizer(text):
        stemmed_text = [stemmer.stem(word) for word in word_tokenize(text, language='english') if word not in english_stopwords]
        return stemmed_text

# we create dictionary containing the possible configurations for the vectorizer
vec_parameters = {'vec__tokenizer': [None, stemming_tokenizer, stemming_stop_tokenizer],
                  'vec__ngram_range': [(1, 1), (1, 2), (1, 3)],
                  'vec__lowercase': [False]}





#YOUR CODE ENDS HERE#
#THIS IS LINE 30#
```

2.1.4 2.1.4

Choose a maximum of 2 classification algorithms (from those seen during the course) and prepare objects containing them.

For each of the selected classification algorithms, prepare a hyper-parameter configuration.

Each configuration must vary **at least 4 different hyper-parameters**.

> *If a parameter is itself composed of several parameters (if it is a dictionary, for example), each of these must vary at least 4 different hyper-parameters.*

[25]:
```
#YOUR CODE STARTS HERE#

# choosing our methods
methods = {'svc': SVC(),
           'decision_tree': DecisionTreeClassifier()}

# dictionary of the configuration dictionaries
met_parameters = {'svc':{'clf__kernel':['poly','rbf', 'sigmoid'],
                        'clf__C':[1, 2],
                        'clf__gamma':['scale', 'auto'],
                        'clf__max_iter': [-1, 10],
                        'clf__random_state': [160]},
                  'decision_tree':{'clf__criterion':['gini','entropy',␣
 ↪'log_loss'],
                                  'clf__splitter':['best', 'random'],
                                  'clf__min_samples_split':[2, 5],
                                  'clf__min_samples_leaf':[1, 2],
                                  'clf__random_state': [160]}
                  }










#YOUR CODE ENDS HERE#
#THIS IS LINE 30#
```

2.1.5 2.1.5

For each of the classification algorithms selected in step 2.1.4, perform a 5-fold Cross-Validation on the validation set, combining the configurations of the vectorizer defined in step 2.1.3 and those of the classifier being used defined in step 2.1.4.

Perform the best hyper-parameter optimisation you can afford in **LESS than 15 minutes**.

If you are using two classifications algorithms, the maximum total optimisation time is **INSTEAD** 30 minutes.

[ ]:
```
#YOUR CODE STARTS HERE#

start_time = time.time()
Grid_res = {}

# for each of the classification algorithms
for name, method in methods.items():
        # we define a pipeline
        pipeline = Pipeline([
                ('vec', vectorizer),
                ('clf', method),
                ])

        # perform the 5-folds cv on the validation set and store the result in␣
 ↪Grid_res
        Grid_res[name] = GridSearchCV(pipeline, param_grid = {**vec_parameters,␣
 ↪**met_parameters[name]},
                                      scoring = metrics.make_scorer(metrics.
 ↪matthews_corrcoef), cv = 5, n_jobs = -1)

        Grid_res[name].fit(val_x, val_y)


time_final = (time.time() - start_time)/60














#YOUR CODE ENDS HERE#
#THIS IS LINE 40#
```

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits

/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:528:
UserWarning:
```

```
The parameter 'token_pattern' will not be used since 'tokenizer' is not None'


Fitting 5 folds for each of 216 candidates, totalling 1080 fits

/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:528:
UserWarning:

The parameter 'token_pattern' will not be used since 'tokenizer' is not None'
```

[ ]:
```python
#YOUR CODE STARTS HERE#


# print out the total time taken
print(f"Time needed in minutes: {time_final}" )



#YOUR CODE ENDS HERE#
#THIS IS LINE 10#
```

```
Time needed in minutes: 27.284403475125632
```

2.1.6 2.1.6

For each of the optimisations run in step 2.1.5:

Select the 5 best configurations and print them.

```
[ ]: #YOUR CODE STARTS HERE#

# for each method
for name, opt in Grid_res.items():

  print('\033[1m' +f'\nThe best scores for the {name} methods were:')
  # extract the scores
  scores = [(i, x) for i, x in enumerate(opt.cv_results_['mean_test_score'])]
  # print the five best scores
  for i, score in sorted(scores, reverse=True, key = lambda x: x[1])[:5]:
    print('\033[0m' +'\nConfiguration:', i, '\n - params: %s\n - mean: %0.3f\n␣
↪- std: %0.3f' %
        (opt.cv_results_['params'][i],
        opt.cv_results_['mean_test_score'][i],
        opt.cv_results_['std_test_score'][i]))




#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```

```
The best scores for the svc methods were:


Configuration: 5
 - params: {'clf__C': 1, 'clf__gamma': 'scale', 'clf__kernel': 'poly',
'clf__max_iter': -1, 'clf__random_state': 160, 'vec__lowercase': False,
'vec__ngram_range': (1, 2), 'vec__tokenizer': <function stemming_stop_tokenizer
at 0x7f2f688b71c0>}
 - mean: 0.765
 - std: 0.117

Configuration: 113
 - params: {'clf__C': 2, 'clf__gamma': 'scale', 'clf__kernel': 'poly',
'clf__max_iter': -1, 'clf__random_state': 160, 'vec__lowercase': False,
'vec__ngram_range': (1, 2), 'vec__tokenizer': <function stemming_stop_tokenizer
at 0x7f2f688b71c0>}
 - mean: 0.745
 - std: 0.101

Configuration: 131
```

```
 - params: {'clf__C': 2, 'clf__gamma': 'scale', 'clf__kernel': 'rbf',
'clf__max_iter': -1, 'clf__random_state': 160, 'vec__lowercase': False,
'vec__ngram_range': (1, 2), 'vec__tokenizer': <function stemming_stop_tokenizer
at 0x7f2f688b71c0>}
 - mean: 0.729
 - std: 0.176

Configuration: 134
 - params: {'clf__C': 2, 'clf__gamma': 'scale', 'clf__kernel': 'rbf',
'clf__max_iter': -1, 'clf__random_state': 160, 'vec__lowercase': False,
'vec__ngram_range': (1, 3), 'vec__tokenizer': <function stemming_stop_tokenizer
at 0x7f2f688b71c0>}
 - mean: 0.699
 - std: 0.075

Configuration: 138
 - params: {'clf__C': 2, 'clf__gamma': 'scale', 'clf__kernel': 'rbf',
'clf__max_iter': 10, 'clf__random_state': 160, 'vec__lowercase': False,
'vec__ngram_range': (1, 2), 'vec__tokenizer': None}
 - mean: 0.694
 - std: 0.094
```

The best scores for the decision_tree methods were:

```
Configuration: 121
 - params: {'clf__criterion': 'entropy', 'clf__min_samples_leaf': 2,
'clf__min_samples_split': 2, 'clf__random_state': 160, 'clf__splitter':
'random', 'vec__lowercase': False, 'vec__ngram_range': (1, 2), 'vec__tokenizer':
<function stemming_tokenizer at 0x7f2f688b7370>}
 - mean: 0.567
 - std: 0.157

Configuration: 193
 - params: {'clf__criterion': 'log_loss', 'clf__min_samples_leaf': 2,
'clf__min_samples_split': 2, 'clf__random_state': 160, 'clf__splitter':
'random', 'vec__lowercase': False, 'vec__ngram_range': (1, 2), 'vec__tokenizer':
<function stemming_tokenizer at 0x7f2f688b7370>}
 - mean: 0.567
 - std: 0.157

Configuration: 49
 - params: {'clf__criterion': 'gini', 'clf__min_samples_leaf': 2,
'clf__min_samples_split': 2, 'clf__random_state': 160, 'clf__splitter':
'random', 'vec__lowercase': False, 'vec__ngram_range': (1, 2), 'vec__tokenizer':
<function stemming_tokenizer at 0x7f2f688b7370>}
 - mean: 0.560
```

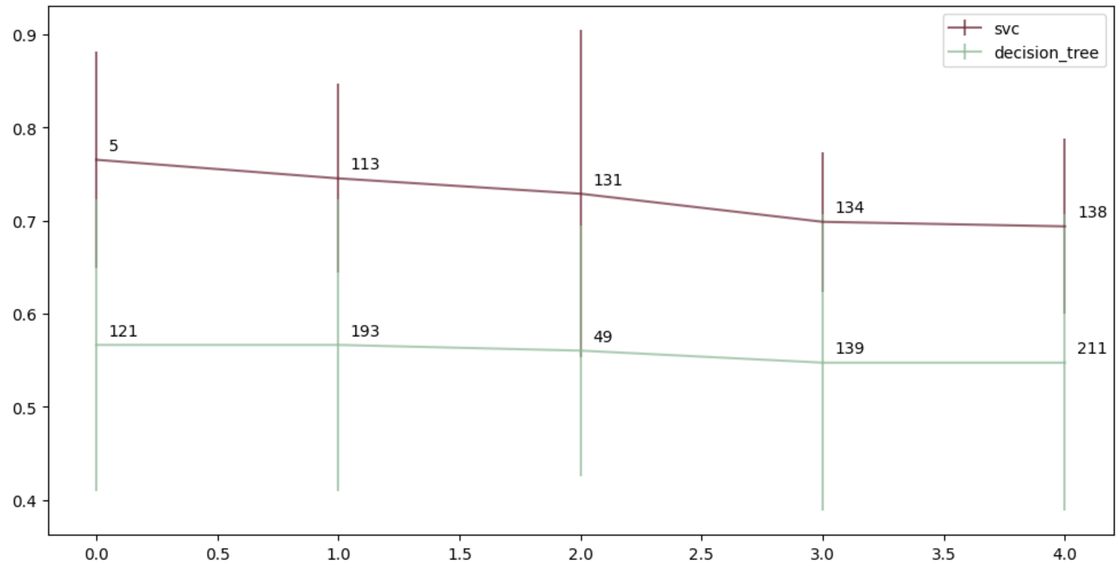
```
 - std: 0.134

Configuration: 139
 - params: {'clf__criterion': 'entropy', 'clf__min_samples_leaf': 2,
'clf__min_samples_split': 5, 'clf__random_state': 160, 'clf__splitter':
'random', 'vec__lowercase': False, 'vec__ngram_range': (1, 2), 'vec__tokenizer':
<function stemming_tokenizer at 0x7f2f688b7370>}
 - mean: 0.547
 - std: 0.159

Configuration: 211
 - params: {'clf__criterion': 'log_loss', 'clf__min_samples_leaf': 2,
'clf__min_samples_split': 5, 'clf__random_state': 160, 'clf__splitter':
'random', 'vec__lowercase': False, 'vec__ngram_range': (1, 2), 'vec__tokenizer':
<function stemming_tokenizer at 0x7f2f688b7370>}
 - mean: 0.547
 - std: 0.159
```

2.1.7 2.1.6

For each of the optimisations run in step 2.1.5:

> Produce a plot with mean and standard deviation of the accuracy calculated on the test set (**of each fold**) for the 5 configuration selected in step 2.1.6.

[ ]:
```python
from matplotlib import colors
#YOUR CODE STARTS HERE#

plt.figure(figsize = (12,6))
colors = ['#8FB996','#6B2737']
for col, opt in enumerate(Grid_res.items()):
    scores = [(i, x) for i, x in enumerate(opt[1].
 ↪cv_results_['mean_test_score'])]
    confs = [i for i,x in sorted(scores, reverse=True, key = lambda x: x[1])[:
 ↪5]]
    mean = opt[1].cv_results_['mean_test_score'][confs]
    sd = opt[1].cv_results_['std_test_score'][confs]
    plt.errorbar(x = list(range(len(mean))),  y=list(mean), yerr=sd, color =␣
 ↪colors[col-1], alpha = 0.7)
    for i, m in enumerate(confs):
        plt.text(i+0.05, mean[i]+0.01, str(m))
plt.legend(['svc','decision_tree'])
plt.show()



#YOUR CODE ENDS HERE#
#THIS IS LINE 20#
```



———-YOUR TEXT STARTS HERE———-

1. Svc: Configuration 5
2. Decision three: Configuration 49

We decided to choose these two configurations for the same reasons: they both have a small interval and their lower bound is the higher among all the configurations.

2.1.8 2.1.8

For each of the optimisations, obtain a classifier using the parameters you selected in step 2.1.6.

[ ]:
```
#YOUR CODE STARTS HERE#

classifiers = {}
vectorizers = {}
# dictionary with the best configurations indexes
best_conf = {'svc':5, 'decision_tree':49}

for name, method in methods.items():
        # we define a pipeline
  vectorizers[name] = TfidfVectorizer()
  classifiers[name] = Pipeline([('vec', vectorizers[name]), ('clf', method)])

  # extract the configuration parameters
  params = Grid_res[name].cv_results_['params'][best_conf[name]]

  # set the parameters
  classifiers[name].set_params(**params)

  # fit on the training set
  classifiers[name].fit(train_x, train_y)









#YOUR CODE ENDS HERE#
#THIS IS LINE 30#
```

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

[ ]:
```
#YOUR CODE STARTS HERE#
results = {}

# For each of the classifiers
```

```python
for name, classifier in classifiers.items():
  # find the predictions
  preds = classifier.predict(test_x)
  # Compute the confusion matrix
  results[name] = metrics.confusion_matrix(test_y, preds)
  # Print the results
  print(f'The confusion matrix for the {name} classifier is:')
  print(pd.DataFrame(results[name]), '\n') # font size


#YOUR CODE ENDS HERE#
#THIS IS LINE 15#
```

```
The confusion matrix for the svc classifier is:
    0   1
0  25   5
1   0  46

The confusion matrix for the decision_tree classifier is:
    0   1
0  24   6
1  14  32
```

2.2 Part 2.2

2.2.1 2.2.1

You have a training set containing N documents. There are M_1 unique terms within the dataset.

The test dataset will have M_2 unique terms within it. However, we know that only a small amount of these will be in common with the training dataset.

What precautions could we use to preprocess the data?

What could we change at test time and which of the classification algorithms seen in class would best suit the change?

Use at most 4 sentences.

———-YOUR TEXT STARTS HERE———-

As for the first part, other then the "classic" preprocessing (Tokenizations, remove StopWords...), what we think could be a good strategy is to increase the size of the training set's vecabulary. A way to do this is to, given the terms already present in the training set, add their synonyms, or words correlated to them, hoping to increase the number of terms in common.

Instead, for the last part, we would chose the Nayve Bayes algorithm where we can easily apply the laplace smoothing. In this, if we have to classify a document made only of unseen terms, we will predict it as a member of the most common class.