DSC510: Introduction to Data Science and Analytics
Lab 10: Natural Language Processing

University of Cyprus
Department of Computer Science

Pavlos Antoniou

Office: B109, FST01

Natural Language Processing (NLP)

- According to industry estimates, only around 20-30% of the available data is present in structured form (tabular data)
- Data is being generated as we speak, as we chat, as we send messages, as we write articles / reviews ...
- Much of this data is highly unstructured (text, images, audio, video)
- Information present in it is not directly accessible unless it is processed (read and understood) manually or analyzed by an automated system – extracting text from images/audio/video is outside of the scope of this course
- In order to produce significant and actionable insights from text data, it is important to get acquainted with the techniques and principles of Natural Language Processing

Outline

- Short introduction to NLP
- Text Preprocessing techniques
- Transforming text to numerical features (feature engineering) to be used in Machine Learning algorithms (text classification/clustering applications)
- Important Tasks of NLP

Introduction to NLP

- NLP is a branch of data science that consists of systematic processes for analyzing, understanding, and deriving information from the text data in a smart and efficient manner
- Utilizing NLP components one can
  - organize the massive chunks of text data into structured data (like a table with rows and columns)
  - perform numerous automated tasks and solve a wide range of problems
    - automatic summarization
    - machine translation
    - named entity recognition: identify Person names, Locations, Organizations
    - relationship extraction: extracted relationships between two or more entities of a certain type (e.g. Person, Organization, Location) and fall into a number of semantic categories (e.g. married to, employed by, lives in).
    - sentiment analysis: determine whether data is positive, negative or neutral
    - speech recognition
    - topic segmentation (clusters of topics)

Important Keywords in NLP

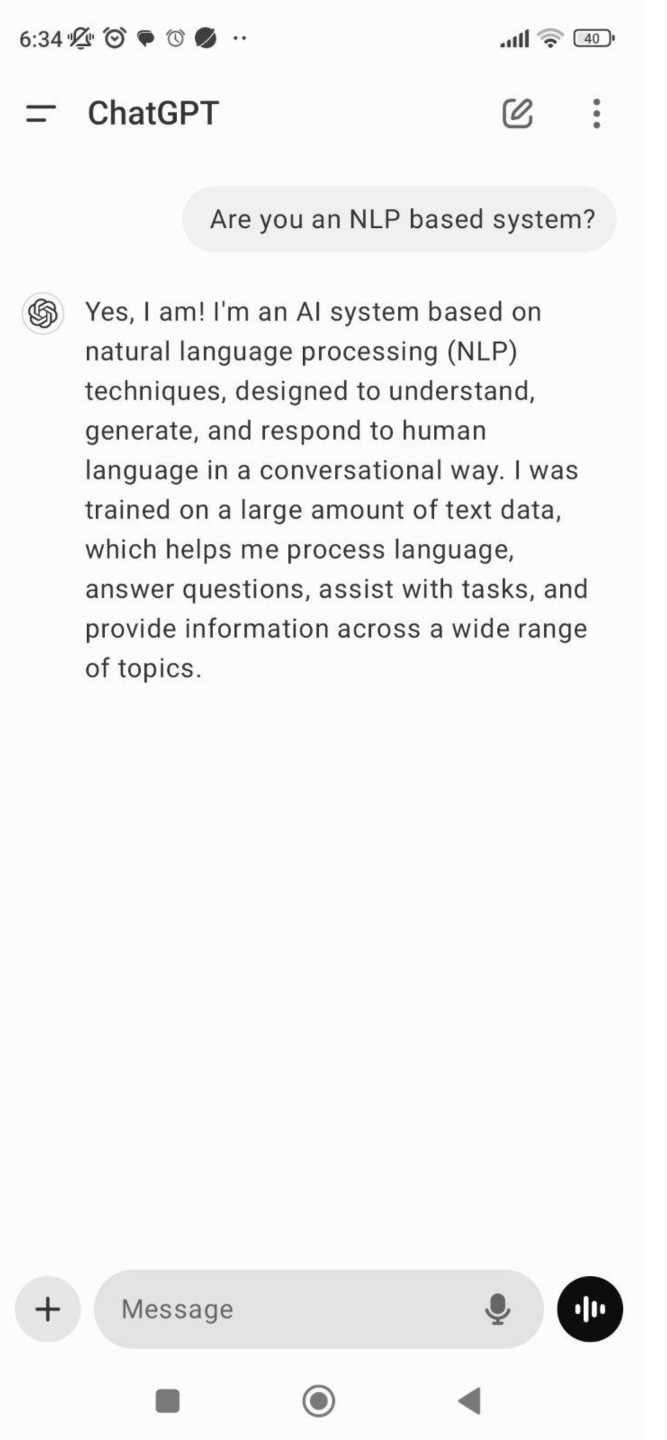
- **Text object**: a document or a sentence or a phrase
- **Tokenization**: process of breaking down text objects into tokens
  - E.g. tokenize a document or a sentence into a list of words
- **Tokens**: smallest units of text extracted from tokenization that carry meaning
  - tokens can be words, syllables, letters or base pairs according to the task
- **N-grams**: a sequence of tokens from already tokenized text objects
  - n-gram of size (n=)1 is referred to as a "unigram“
  - size (n=)2 is a "bigram" (or "digram"), size 3 is a "trigram", etc.
    - Example: if tokens are words, the set of bigrams in the sentence “this is an example” is [“this is”, “is an”, “an example”]
  - Usage: auto completion of sentences, auto spell check
- **NLTK**: Most important NLP Python library – Available in Anaconda

NLP in Real life

- Conversational AI services to understand and respond to human conversation (ChatGPT, Bard are NLP platforms)
- Information Retrieval (search engines to find relevant and similar results)
- Information Extraction (Gmail structures events from emails)
- Machine Translation (translate language from one language to another)
- Text Simplification (Rewordify simplifies the meaning of sentences)
- Sentiment Analysis (Hater News gives us the sentiment of the user)
- Text Summarization (Smmry or Reddit's autotldr gives a summary of sentences)
- Spam Filter (Mail providers filter spam emails separately)
- Auto-Predict (Search Engines predict user search results)
- Auto-Correct (Google Keyboard and Grammarly correct words spelled wrong)
- Speech Recognition (Google WebSpeech or Vocalware)
- Question Answering (IBM Watson's answers to a query)
- Natural Language Generation (Generation of text from image or video data)

NLP in Real life

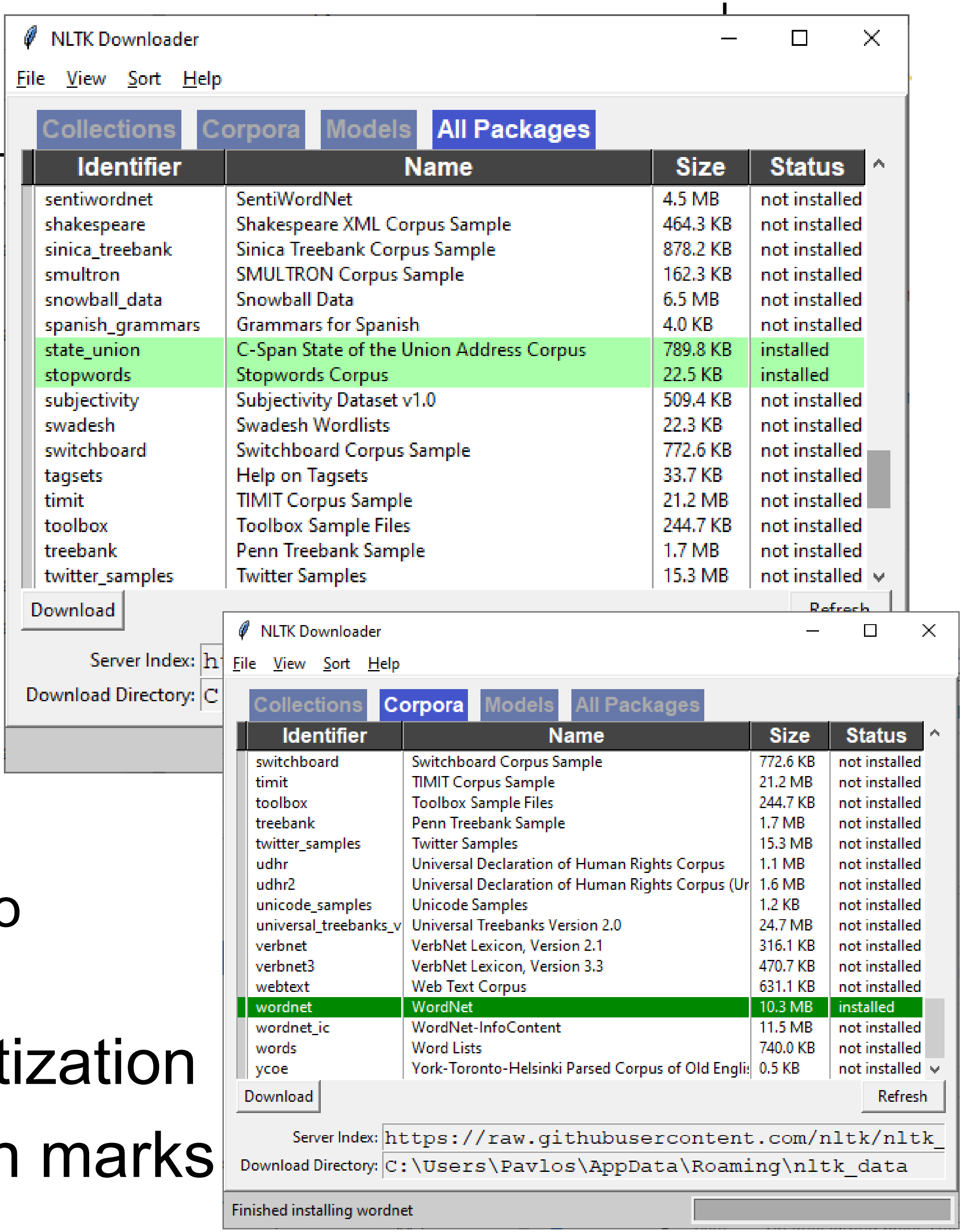
- Conversational AI services to understand and respond to hum
  (ChatGPT, Bard are NLP platforms)
- Information Retrieval (search engines to find relevant and sim
- Information Extraction (Gmail structures events from emails)
- Machine Translation (translate language from one language to
- Text Simplification (Rewordify simplifies the meaning of sente
- Sentiment Analysis (Hater News gives us the sentiment of the
- Text Summarization (Smmry or Reddit's autotldr gives a sumr
- Spam Filter (Mail providers filter spam emails separately)
- Auto-Predict (Search Engines predict user search results)
- Auto-Correct (Google Keyboard and Grammarly correct words
- Speech Recognition (Google WebSpeech or Vocalware)
- Question Answering (IBM Watson's answers to a query)
- Natural Language Generation (Generation of text from image



Import NLTK library

```
In [ ]: import nltk
        nltk.download()
```

- After running the above, we get an NLTK Downloader Application (as popup) which is helpful in NLP Tasks
- We can install useful packages by double clicking on them
- Download stopwords corpus
  - Set of common words that will be used to remove redundant repeated words
- Download wordnet corpus for lemmatization
- Download punkt for using punctuation marks



Text Preprocessing

- Text is the most unstructured form of all the available data
  - various types of noise are present in it
    - Low importance words (and, or, for), HTML tags, spelling mistakes
  - data is not readily analyzable without any pre-processing
- The entire process of cleaning and standardization of text, making it noise-free and ready for analysis is known as text preprocessing
- Text preprocessing is predominantly comprised of three steps:
  - Noise Removal
  - Lexicon Normalization
  - Object Standardization

  Goal: minimize the set of unique tokens
- Text preprocessing is a "dimensionality reduction" process
  - Features in NLP datasets can be tokens
  - Dataset dimension relates to the number of unique tokens

Access your data

```
In [ ]: import pandas as pd
        # Read tab separated values
        data = pd.read_csv('SMSSpamCollection.tsv', sep='\t',
        names=['label', 'body_text'], header=None)
        print(data.head())
```

	label	body_text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

`data['body_text']` is a column of dataframe which involves the text of each SMS

Noise Removal

- Any piece of text which is not relevant to the context of the data and the end-output can be (but not always) specified as the noise
  - punctuation marks (,!;?.) and industry specific words
  - URLs or links
  - social media entities (mentions, hashtags)
  - language stopwords (commonly used words of a language – a, is, am, the, of, in etc.)
    - Lowercase text before removing stop words
- A general approach for noise removal is to prepare a dictionary of noisy entities, and iterate the text object by tokens (or by words), eliminating those tokens which are present in the noise dictionary

Html tags removal

```
from bs4 import BeautifulSoup

# Remove html tags: gets a line of text as input and returns it without html tags.
def remove_htmltags(text):
    return BeautifulSoup(text).get_text()
```

Pandas Series (a column of Dataframe)

Apply function iterates on every element (row) of the Pandas Series (in this example, every element is an SMS, thus a line of text) and applies the given function in parenthesis

```
data['text_notags'] = data['body_text'].apply(remove_htmltags)
```

```
body_text
Go until jurong point, crazy.. Available only ...
Ok lar... Joking wif u oni...
Free entry in 2 a wkly comp to win FA Cup fina...
U dun say so early hor... U c already then say...
Nah I don't think he goes to usf, he lives aro...
```

Puctuation and stopwords removal

```
import string # punctuation characters in string.punctuation
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
punctuation = string.punctuation        # a list of all punctuation marks
stop_words = stopwords.words('english') # a list of all stop words in English lang

# function takes a line of text as input and discards all punctuation
# returns a new line without punctuation
def remove_punct(text):
    text = "".join([char for char in text if char not in punctuation])
    return text

# remove stopwords (words must be lowercase prior stop words removal)
def remove_stopwords(text):
    tokens = word_tokenize(text.lower()) # tokens is a list of words
    text = " ".join([word for word in tokens if word not in stop_words])
    return text
data['text_nopunct'] = data['text_notags'].apply(remove_punct)
data['text_nostop'] = data['text_nopunct'].apply(remove_stopwords)
data.head()
```

Puctuation and stopwords removal – result

	label	body_text	text_notags	text_nopunct	text_nostop
0	ham	Go until jurong point, crazy.. Available only ...	Go until jurong point, crazy.. Available only ...	Go until jurong point crazy Available only in ...	go jurong point crazy available bugis n great ...
1	ham	Ok lar... Joking wif u oni...	Ok lar... Joking wif u oni...	Ok lar Joking wif u oni	ok lar joking wif u oni
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	Free entry in 2 a wkly comp to win FA Cup fina...	Free entry in 2 a wkly comp to win FA Cup fina...	free entry 2 wkly comp win fa cup final tkts 2...
3	ham	U dun say so early hor... U c already then say...	U dun say so early hor... U c already then say...	U dun say so early hor U c already then say	u dun say early hor u c already say
4	ham	Nah I don't think he goes to usf, he lives aro...	Nah I don't think he goes to usf, he lives aro...	Nah I dont think he goes to usf he lives aroun...	nah i dont think goes usf lives around though

Lexicon (set of dataset tokens) Normalization

- A word can be found in multiple representations, contextually similar
  - "play", "player", "played", "plays" and "playing": different variations of "play"
- This step converts all the disparities of a word into their normalized form (also known as lemma)
  - Normalization is a pivotal step for feature engineering with text as it converts the high dimensional features (N words → N different features) to lower dimensional space (even to 1 feature) which is an ideal task for any ML model
- Common lexicon normalization practices:
  - **Stemming**: A rudimentary rule-based process of stripping the suffixes ("ing", "ly", "es", "s" etc) from a word, e.g. the words *go*, *goes*, *going* are stemmed to the same root which is the word *go*
  - **Lemmatization**: An organized & step by step procedure of obtaining the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations)

Stemming and lemmatization

```
ps = nltk.PorterStemmer()
wl = nltk.WordNetLemmatizer()

def stemming(text):
    tokens = word_tokenize(text)
    stemmed_text = " ".join([ps.stem(word) for word in tokens])
    return stemmed_text

def lemmatizing(text):
    tokens = word_tokenize(text)
    lemmatized_text = " ".join([wl.lemmatize(word) for word in tokens])
    return lemmatized_text

data['text_stemmed'] = data['text_nostop'].apply(stemming)
data['text_lemmatized'] = data['text_nostop'].apply(lemmatizing)
data.head()
```

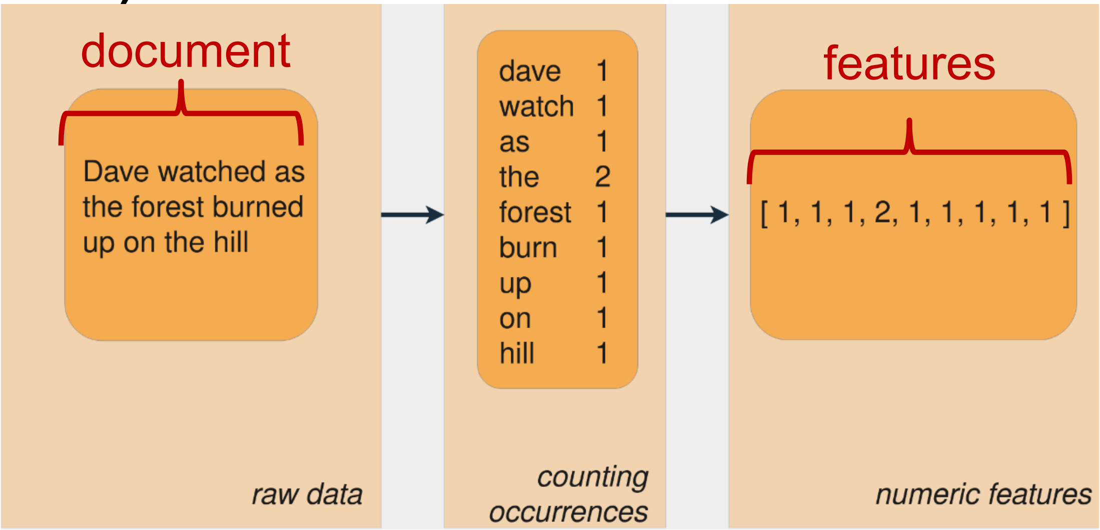
Object Standardization

- Text data often contains words or phrases which are not present in any standard lexical dictionaries
- These pieces are not recognized by search engines and models
  - acronyms, hashtags with attached words, colloquial slangs*
- Common solution is to fix this type of noise with the help of regular expressions and manually prepared data dictionaries
  - provide mappings of “noisy words” to standard words
  - the code on the next slide uses a dictionary lookup method to replace social media slangs from a text

* Slang words are informal, non-standard words or phrases that often emerge within specific social groups, cultures, or regions and may have different meanings than their literal interpretations. Slang is typically more casual, and its usage can vary significantly across generations, locations, and social groups.

Text to Features – Feature Engineering

- Preprocessed text objects (documents) need to be converted into a set of features for further analysis
  - For example, each preprocessed document (e.g an SMS or a tweet or a news article) can be converted into a vector (something computer algorithms can work with)



- Depending upon the usage, text features can be constructed using assorted techniques:
  - Syntactic Parsing
  - Entities / N-grams / word-based features

For distinguishing different parts of speech (i.e. nouns, verbs), adding syntactic relations to text, discovering entities in text, etc. More relevant in the language sciences. More info in Appendix.

  - Statistical features
  - Word embeddings

Convert every text object to a numerical vector → for analysis using ML techniques e.g. classification, clustering

Statistical Features – Bag of words model

- Converts document to a numerical vector, i.e. to a linear combination of words: v = a1*x1 + a2*x2 + …
  - a1, a2, … : weights
  - x1, x2, … : components (tokens e.g. words)
- The **weight** of a component of a document vector can be represented by **term frequency (tf)**
- Term frequency denoted by tf, is the number of occurrences of a token t in the document D
  - E.g. given document "hello world hello",
vocabulary of tokens = ["hello", "world"]
document vector = [2, 1]
  - This is the **Bag of words model**

Bag of words model example

- Review 1: This movie is very scary and long
- Review 2: This movie is not scary and is slow
- Review 3: This movie is spooky and good
- Vocabulary: [‘This’, ‘movie’, ‘is’, ‘very’, ‘scary’, ‘and’, ‘long’, ‘not’, ‘slow’, ‘spooky’, ‘good’]

	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good
Review 1	1	1	1	1	1	1	1	0	0	0	0
Review 2	1	1	2	0	0	1	1	0	1	0	0
Review 3	1	1	1	0	0	0	1	0	0	1	1

- Vector of Review 1: [1 1 1 1 1 1 1 0 0 0 0]
- Vector of Review 2: [1 1 2 0 0 1 1 0 1 0 0]
- Vector of Review 3: [1 1 1 0 0 0 1 0 0 1 1]

Bag of words model example

```
# import Bag of words model: CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

corpus = ['This movie is very scary and long', 'This movie is not scary and is
slow', 'This movie is spooky and good']

vectorizer = CountVectorizer()

# run the bag of words model which returns the matrix of numerical vectors
X = vectorizer.fit_transform(corpus)
# print the vocabulary
print(vectorizer.get_feature_names())
# print the matrix
print(X.toarray())
```

- The dataset vocabulary of feature set involves all words without any pre-processing techniques applied → we have 11 dimensions/features in this dataset
- **By default, count vectorizer performs only tokenization** without any other pre-processing technique.

```
['and', 'good', 'is', 'long', 'movie', 'not', 'scary', 'slow', 'spooky', 'this', 'very']
[[1 0 1 1 1 0 1 0 0 1 1]
 [1 0 2 0 1 1 1 1 0 1 0]
 [1 1 1 0 1 0 0 0 1 1 0]]
```

Bag of words model example + Preprocessing

```
# import Bag of words model: CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
corpus = ['This movie is very scary and long', 'This movie is not scary and is slow', 'This movie is spooky and good']
# provide a function to perform the pre-preprocessing steps (by default performs only tokenization) - clean_text is a custom function (see .ipynb in website) that removes punctuation, turns to lowercase, removes stop words, performs stemming
vectorizer = CountVectorizer(analyzer=clean_text)
# run the bag of words model which returns the matrix of numerical vectors
X = vectorizer.fit_transform(corpus)
# print the vocabulary
print(vectorizer.get_feature_names())
# print the matrix
print(X.toarray())
```

```
['good', 'long', 'movi', 'scari', 'slow', 'spooki']
[[0 1 1 1 0 0]
 [0 0 1 1 1 0]
 [1 0 1 0 0 1]]
```

Statistical Features – tfidf model

- Problem of Bag of words: all terms are considered equally important
  - certain tokens have little or no discriminating power: they will not help in distinguishing one document from another
    - e.g. collection of documents on the auto industry is likely to have the token auto in almost every document
- Inverse document frequency of a term t, denoted by $idf(t) = log\left(\frac{N}{df_t}\right)$ where:
  - N: total number of documents in the space,
  - df_t: total number of documents that contain the token t
- small idf:
  - a token occurs in many documents (not important token)
- high idf:
  - a token occurs in a small number of documents (important token)

Without log(), rare terms would receive very large values, while common terms would receive very small values. This could lead to highly skewed distributions, which might not improve model performance. Log() reduces skewness.

Statistical Features – tfidf model

- The product of tf and idf is the most popular weight (of each token) used in case of documents similarity exercises

- $\text{tf-idf}_{t,d} = \text{tf}_{t,d} * \text{idf}_t$

Low df => high idf

High tf

- Weight is the highest, when token t occurs many times within a small number of documents : high discriminating power

Low tf

- Weight is the lowest, when term t occurs fewer times in a document or occurs in many documents : low discriminating power

High df => low idf

Statistical Features – tfidf model example

- D1 = "Shipment of gold damaged in a fire"
- D2 = "Delivery of silver arrived in a silver truck"
- D3 = "Shipment of gold arrived in a truck"

Document tfidf vector for D1

Vectors created using the Bags of word model

Vectors created using the tfidf model

Tokens (volabulary)	tf_i					idf_i	Weights = tf_i * idf_i		
	D1	D2	D3	df_i	N/df_i		D1	D2	D3
a	1	1	1	3	1	0	0.0000	0.0000	0.0000
arrived	0	1	1	2	1.5	0.1761	0.0000	0.1761	0.1761
damaged	1	0	0	1	3	0.4771	0.4771	0.0000	0.0000
delivery	0	1	0	1	3	0.4771	0.0000	0.4771	0.0000
gold	1	0	1	2	1.5	0.1761	0.1761	0.0000	0.1761
fire	1	0	0	1	3	0.4771	0.4771	0.0000	0.0000
in	1	1	1	3	1	0	0.0000	0.0000	0.0000
of	1	1	1	3	1	0	0.0000	0.0000	0.0000
shipment	1	0	1	2	1.5	0.1761	0.1761	0.0000	0.1761
silver	0	2	0	1	3	0.4771	0.0000	0.9542	0.0000
truck	0	1	1	2	1.5	0.1761	0.0000	0.1761	0.1761

Statistical Features – tfidf model

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = ['This movie is very scary and long', 'This movie is not scary and is slow', 'This movie is spooky and good']
vectorizer = TfidfVectorizer()
# run the model and get the tf-idf matrix
tfidf_matrix = vectorizer.fit_transform(corpus)
# print vocabulary
print(vectorizer.get_feature_names())
# print tf-idf matrix
print(tfidf_matrix.toarray())
```

```
['and', 'good', 'is', 'long', 'movie', 'not', 'scary', 'slow', 'spooky', 'this', 'very']
[[0.29628336 0.         0.29628336 0.50165133 0.29628336 0.         0.38151877 0.         0.         0.29628336 0.50165133]
 [0.26359985 0.         0.5271997  0.         0.26359985 0.44631334 0.3394328  0.44631334 0.         0.26359985 0.        ]
 [0.32052772 0.54270061 0.32052772 0.         0.32052772 0.         0.         0.         0.54270061 0.32052772 0.        ]]
```

Pros and cons of tfidf model

- Pros:
  - Statistical model (based on terms frequencies)
  - Simple to understand and computationally cheap to calculate
- Cons:
  - Does not capture semantic meaning of tokens
  - Considers the importance of the words based on how it weighs them, but this cannot necessarily derive the contexts of the words and understand their importance. For example:
    - compound nouns like “Queen of England” will not be considered as a “single unit”
    - negation makes a big difference in sentence meaning “not pay the bill” vs “pay the bill”
  - Suffers from the curse of dimensionality
    - Length of TF-IDF vectors is equal to the size of the vocabulary

Word Embedding

- Modern way of representing words as numerical vectors
- Widely used in deep learning models such as Convolutional Neural Networks and Recurrent Neural Networks
- Word2Vec and GloVe are the two popular models to create word embedding of a text
  - models take a text corpus as input and produce word vectors as output
  - embedding vectors are relatively low dimensional (typically 50-300 dimensions) vectors
  - vectors capture semantic relationships and contextual information

Word Embedding – Word2Vec

```
from gensim.models import Word2Vec
corpus = [['data', 'science'], ['computer', 'science', 'data', 'analytics'],
['machine', 'learning'], ['deep', 'learning']]

# train the model on your corpus
model = Word2Vec(corpus, min_count = 1)

print(model.wv['learning'])

print(model.wv.similarity('data', 'science'))

print(model.wv.most_similar('science'))
```

```
[-5.3622725e-04  2.3643016e-04  5.1033497e-03  9.0092728e-03
 -9.3029495e-03 -7.1168090e-03  6.4588715e-03  8.9729885e-03
 -5.0154282e-03 -3.7633730e-03  7.3805046e-03 -1.5334726e-03
 -4.5366143e-03  6.5540504e-03 -4.8601604e-03 -1.8160177e-03
  2.8765798e-03  9.9187379e-04 -8.2852151e-03 -9.4488189e-03
  7.3117660e-03  5.0702621e-03  6.7576934e-03  7.6286553e-04
  6.3508893e-03 -3.4053659e-03 -9.4640255e-04  5.7685734e-03
 -7.5216386e-03 -3.9361049e-03 -7.5115822e-03 -9.3004224e-04
  9.5381187e-03 -7.3191668e-03 -2.3337698e-03 -1.9377422e-03
  8.0774352e-03 -5.9308959e-03  4.5161247e-05 -4.7537349e-03
 -9.6035507e-03  5.0072931e-03 -8.7595871e-03 -4.3918253e-03
 -3.5099984e-05 -2.9618264e-04 -7.6612402e-03  9.6147414e-03
  4.9820566e-03  9.2331432e-03 -8.1579182e-03  4.4957972e-03
 -4.1370774e-03  8.2453492e-04  8.4986184e-03 -4.4621779e-03
  4.5175003e-03 -6.7869616e-03 -3.5484887e-03  9.3985079e-03
 -1.5776539e-03  3.2137157e-04 -4.1406299e-03 -7.6826881e-03
 -1.5080094e-03  2.4697948e-03 -8.8802812e-04  5.5336617e-03
 -2.7429771e-03  2.2600652e-03  5.4557943e-03  8.3459523e-03
 -1.4537406e-03 -9.2081428e-03  4.3705511e-03  5.7178497e-04
  7.4419067e-03 -8.1328390e-04 -2.6384138e-03 -8.7530091e-03
 -8.5655687e-04  2.8265619e-03  5.4014279e-03  7.0526553e-03
 -5.7031228e-03  1.8588186e-03  6.0888622e-03 -4.7980524e-03
 -3.1072616e-03  6.7976285e-03  1.6314745e-03  1.8991709e-04
  3.4736372e-03  2.1777629e-04  9.6188262e-03  5.0606038e-03
 -8.9173913e-03 -7.0415614e-03  9.0145587e-04  6.3925339e-03]
```

```
-0.023671666
```

```
[('deep', 0.06797593832015991), ('analytics', 0.009391184896230698),
('machine', 0.004503015894442797), ('learning', -0.010839187540113926),
('data', -0.023671669885516167), ('computer', -0.11410722136497498)]
```

Important tasks of NLP

- Document vectors (tfidf) and word vectors (word2vec) can be used as feature vectors in ML models to measure
  - document matching / similarity
    - Usage: automatic spelling correction, data de-duplication, genome analysis
  - document classification / clustering ML techniques
    - Usage: email spam identification, topic classification of news, sentiment classification and organization of web pages by search engines

Example: tfidf transformation for doc classification

- Steps:
  - Apply preprocessing techniques (html tags/punctuation removal, lowercasing, stop-words removal, stemming) on the whole doc dataset
  - Split document dataset into training(,validation), testing

```
X_train, X_test, y_train, y_test = train_test_split(df["text_nostop"],df["label"],test_size=0.2)
```

  - Train the tfidf transformation on the training dataset in order to obtain the vocabulary (set of unique tokens/features)

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(X_train)
```

  - Transform training dataset to numerical vectors

```
X_train_tfidf_vector = vectorizer.transform(X_train)
```

  - Transform testing dataset to vectors using the vocabulary of training dataset

```
X_test_tfidf_vector = vectorizer.transform(X_test)
```

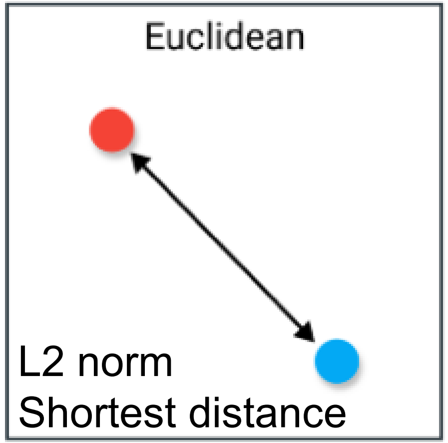
  - Train the classifier and make predictions →

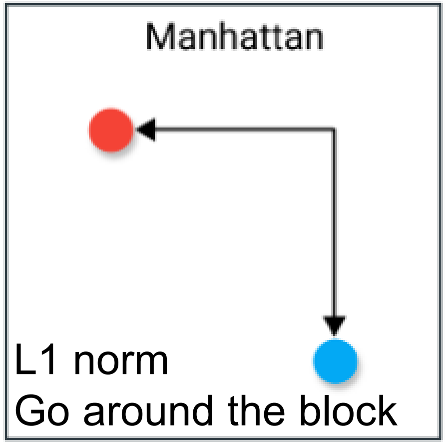
```
rf = RandomForestClassifier()
rf.fit(X_train_count_vector, y_train)
y_pred = rf.predict(X_test_count_vector)
```

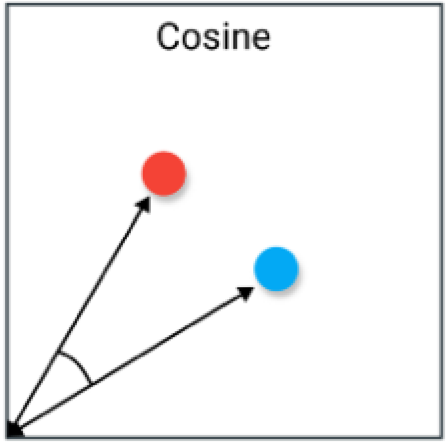
Distance vs similarity

- When using distance-based clustering/classification methods (such as kMeans, KNN, SVM) on document datasets we can use two metrics to identify which docs are related to each other

1. → **distance** (default metric in kMeans, KNN, SVM)
   – Example: Euclidean distance, Manhattan (city block) distance
     - Typically, use Euclidean metric; Manhattan may be appropriate if different dimensions are not comparable (not scaled) [source]
     - Manhattan may be preferable to Euclidean distance for the case of high dimensional data [source]
2. → **similarity**
   – Example: Cosine Similarity: concerned with the *orientation* of 2 docs in space than the exact distance from one another
   – We need to modify kMeans, KNN, SVM implementations

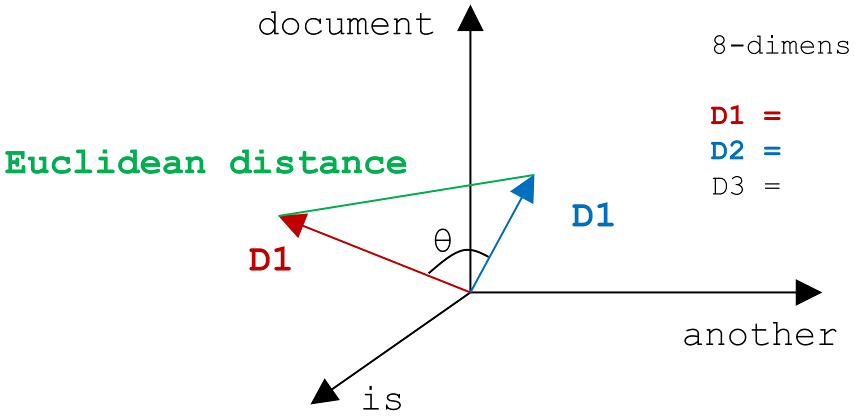






Cosine similarity

- Cosine similarity is a metric used to measure how similar the documents are, irrespective of the vector magnitude (length)
  - measures the cosine of the angle (θ) between two vectors projected in a multi-dimensional space



8-dimensions Dataset

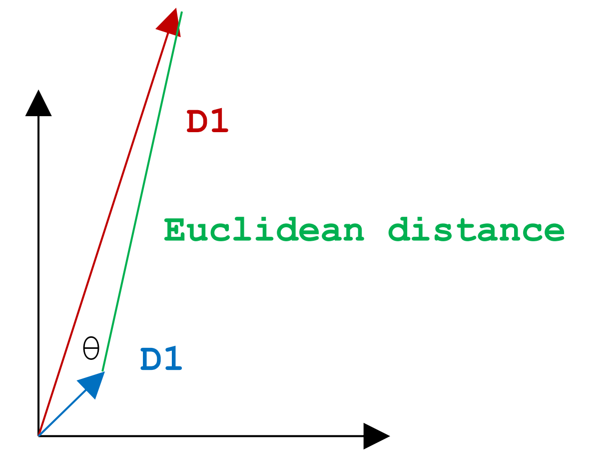
	another	document	is	random	sample	text	third	this
D1 =	0.000000	0.345205	0.584483	0.000000	0.444514	0.000000	0.000000	0.584483
D2 =	0.652491	0.385372	0.000000	0.652491	0.000000	0.000000	0.000000	0.000000
D3 =	0.000000	0.345205	0.000000	0.000000	0.444514	0.584483	0.584483	0.000000

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

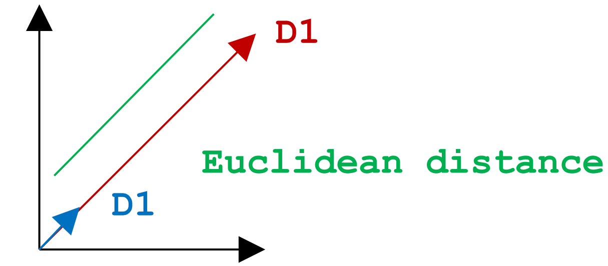
  - the smaller the angle, the higher the cosine similarity
  - cosine similarity ranges from -1 (low similarity) to +1 (high similarity)

Which measure to use?

- Previous measures aren't at all the same thing and they yield quite different results
- **Euclidean distance** is concerned with the **magnitudes** of vectors between two documents
- **Cosine similarity** is only concerned with the **orientation** of two documents



  - Much less affected by magnitude or how large the numbers of vectors are
  - If 2 docs have proportional weights among the various dimensions (features) cosine similarity will be high (small θ → high cosθ → high similarity) regardless of the actual weights. Example: one doc is (1,1) and other (600, 600)
    - Cosine similarity: θ=0°, cosθ = 1 ➔ high similarity, relevant docs
    - Euclidean distance: 847 ➔ high distance, irrelevant docs



Which measure to use?

- Cosine distance is sometimes very good for text-related data since texts, usually, create vectors of very different magnitudes
- However, if you know your sample texts are all roughly the same magnitude, you might prefer to account for the small differences in magnitude by using Euclidean distance
- There's no one answer for which distance measure to choose
  - it's highly dependent on your data
  - you can experiment with the different measures and see their effectiveness on accuracy / error (applicable in supervised problems)

Document similarity using tfidf matrix

- Cosine similarity matrix contains the pairwise cosine similarity score for every pair of documents

```
from sklearn.metrics.pairwise import cosine_similarity

# compute and print the cosine similarity matrix
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
print(cosine_sim)
```

```
             D1          D2          D3
D1   [[1.          0.13303222 0.31675948]
D2    [0.13303222 1.          0.13303222]
D3    [0.31675948 0.13303222 1.        ]]
```

Highest similarity: D1 with D3

```
D1 = 'This is sample document.'
D2 = 'another random document.'
D3 = 'third sample document text'
```

Last in-lab Assignment

- Load the "Natural Language Processing with Disaster Tweets" dataset in order to predict whether a given tweet is about a real disaster (target=1) or not (target=0)
  - Drop columns id, keyword, location
- Provide a plot to show whether the dataset is balanced or unbalanced (number of samples between classes is even or uneven) and comment on whether you need to apply balancing techniques
- Use the **clean_text** function (provided in this notebook file) to pre-process your dataset (column "text")
- Split your dataset into training (80%) and testing (20%)
- Use both the bag of words model and the tfidf model to transform each doc of the training and testing datasets to a numerical vector

Last in-lab Assignment

- Train XGBClassifier and RandomForestClassifier (using transformed training dataset), make predictions (using transformed testing dataset) and print the F1 scores for the 4 combinations of scenarios
- Submit .ipynb file to Moodle by Friday 28th of November @ 23.59
  - Login to Moodle: https://moodle.cs.ucy.ac.cy/course/view.php?id=312 using your UCY credentials
  - Follow “Lab10 Submission” link
  - Upload your .ipynb file

APPENDIX
Syntactic Parsing
Entities / N-grams / word-based features

Syntactic Parsing

- Analysis of words in the sentence for grammar and their arrangement in a manner that shows the relationships among the words
- Important attributes of text syntactics:
  - Dependency Grammar
  - Part of Speech (POS) tags

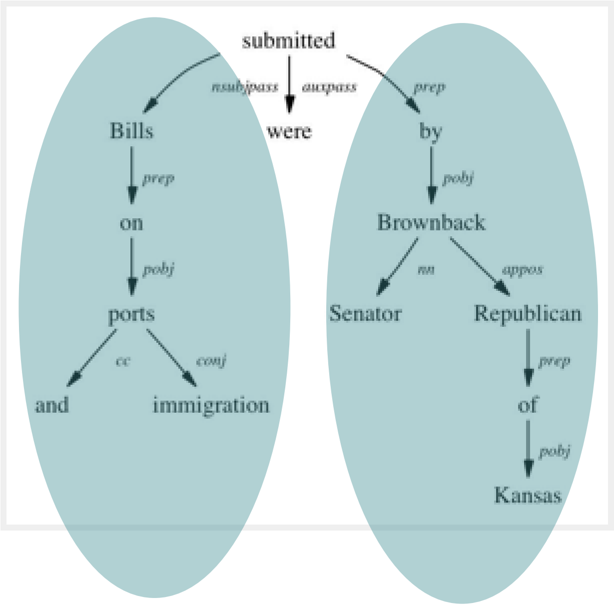
Dependency trees

- Sentences are composed of some words sewed together
- Every relation between two words can be represented in the form of a triplet (relation, governor, dependent)
- Example: *"Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas."*

- "submitted" is the root word of this sentence and is linked by two sub-trees (subject and object subtrees)
- each subtree itself is a dependency tree

Python libraries:

- [NLTK Dependency Grammar]
- [Stanford Core]



Object (action on?) Subject (who?)

- This type of tree, when parsed recursively in top-down manner gives grammar relation triplets as output
- **Relation triplets can be used as features for many NLP problems like entity wise sentiment analysis, actor & entity identification, and text classification.**

Part of Speech (POS) tagging

- Apart from the grammar relations, every word in a sentence is also associated with a POS tag:
  - Nouns
  - Verbs
  - Adjectives
  - Adverbs
  - …
- POS tags define the usage and function of a word in the sentence

POS tagging annotation on input text

```
nltk.download('averaged_perceptron_tagger')
```

```
from nltk import word_tokenize, pos_tag
text = "I am learning Natural Language Processing on the Introduction to Data 
Science and Analytics course"
tokens = word_tokenize(text)
print(pos_tag(tokens))
```

```
[('I', 'PRP'), ('am', 'VBP'), ('learning', 'VBG'), ('Natural', 'NNP'),
('Language', 'NNP'), ('Processing', 'NNP'), ('on', 'IN'), ('the', 'DT'),
('Introduction', 'NNP'), ('to', 'TO'), ('Data', 'NNP'), ('Science', 'NNP'),
('and', 'CC'), ('Analytics', 'NNP'), ('course', 'NN')]
```

POS usage

- **Word sense disambiguation**
  - Some language words have multiple meanings according to their usage. For example, in the two sentences below:
    I. “Please book my flight for Delhi”
    II. “I am going to read this book in the flight”
  - “book” is used with different context, however the part of speech tag for both cases are different
  - In sentence I. the word “book” is used as **verb**, while in II. it is used as **noun**.
  - Lesk Algorithm is also used for similar purposes

POS usage

- **Improving word-based features**
  - A learning model could learn different contexts of a word when used word as the features, however if the part of speech tag is linked with them, the context is preserved, thus making strong features. For example:
  - Sentence -“book my flight, I will read this book”
  - Tokens – (“book”, 2), (“my”, 1), (“flight”, 1), (“I”, 1), (“will”, 1), (“read”, 1), (“this”, 1)
  - Tokens with POS – (“book_VB”, 1), (“my_PRP$”, 1), (“flight_NN”, 1), (“I_PRP”, 1), (“will_MD”, 1), (“read_VB”, 1), (“this_DT”, 1), (“book_NN”, 1)

POS usage

- **Normalization and Lemmatization**
  - POS tags are the basis of lemmatization process for converting a word to its base form (lemma)

- **Efficient stopword removal**
  - POS tags are also useful in efficient removal of stopwords
  - For example, there are some tags which always define the low frequency / less important words of a language
  - For example: (IN – “within”, “upon”, “except”), (CD – “one”, “two”, “hundred”), (MD – “may”, “must” etc)

Entity Extraction (Entities as features)

- Entities: the most important chunks of a sentence – noun phrases, verb phrases or both.
- Entity Detection algorithms are generally ensemble models of rule-based parsing, dictionary lookups, pos tagging and dependency parsing
- Two key entity detection methods in NLP:
  - Named Entity Recognition
  - Topic Modelling

Named Entity Recognition (NER)

- The process of detecting the named entities such as person names, location names, company names etc. from the text is called as NER
- For example :
  - Sergey Brin, the manager of Google Inc. is walking in the streets of New York.
  - Named Entities – (“person” : “Sergey Brin”), (“org” : “Google Inc.”), (“location” : “New York”)

Named Entity Recognition (NER)

- A typical NER model consists of three blocks:
  - Noun phrase identification
    - This step deals with extracting all the noun phrases from a text using dependency parsing and part of speech tagging
  - Phrase classification
    - This is the classification step in which all the extracted noun phrases are classified into respective categories (locations, names etc). Google Maps API provides a good path to disambiguate locations. Then, the open databases from dbpedia, wikipedia can be used to identify person names or company names. Apart from this, one can curate the lookup tables and dictionaries by combining information from different sources
  - Entity disambiguation
    - Sometimes it is possible that entities are misclassified, hence creating a validation layer on top of the results is useful. Use of knowledge graphs can be exploited for this purposes. The popular knowledge graphs are – Google Knowledge Graph, IBM Watson and Wikipedia

Topic Modelling

- The process of automatically identifying the topics present in a text corpus, it derives the hidden patterns among the words in the corpus in an unsupervised manner
- A topic is defined as a collection of dominant keywords that are typical representatives
  - Just by looking at the keywords, you can identify what the topic is all about
  - A good topic model will cluster together the words “health”, “doctor”, “patient”, “hospital” – for a topic “Healthcare”
  - Another good topic model will cluster together the words “farm”, “crops”, “wheat” – for a topic “Farming”
- Latent Dirichlet Allocation (LDA) is the most popular topic modelling technique (implemented in Python Gensim package)

LDA Example

```
import gensim
from gensim import corpora
from sklearn.datasets import fetch_20newsgroups
newsgroups_train = fetch_20newsgroups(subset='train', categories=['sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',])

doc_words = [doc.split() for doc in newsgroups_train.data[0:100]]

doc_no_punct = list()
for lista in doc_words:
    doc_no_punct.append([remove_punct(word.lower()) for word in lista])
#print(doc_no_punct)

doc_no_stopwords = list()
for lista in doc_no_punct:
    doc_no_stopwords.append([word.lower() for word in lista if word not in stop_words])
#print(doc_no_stopwords)

doc_lemmatized = list()
for lista in doc_no_stopwords:
    doc_lemmatized.append([wl.lemmatize(word) for word in lista])

# Creating the term dictionary of our corpus, where every unique term is assigned an index.
dictionary = corpora.Dictionary(doc_lemmatized)
# Converting list of documents (corpus) into Document Term Matrix using dictionary prepared above.
doc_term_matrix = [dictionary.doc2bow(doc) for doc in doc_lemmatized]
# Creating the object for LDA model using gensim library
Lda = gensim.models.ldamodel.LdaModel
# Running and Training LDA model on the document term matrix
ldamodel = Lda(doc_term_matrix, num_topics=4, id2word = dictionary, passes=50)
# Results
print(ldamodel.print_topics())
```

If gensim library is not installed on your machine, open Anaconda Prompt and type:

```
conda install -c anaconda gensism
```

N-Grams as features

- A combination of N words together are called N-Grams
- N grams (N > 1) are generally more informative as compared to 1-Grams (Unigrams) as features
  - N=2: bigrams
  - N=3: trigrams
  - …
- Bigrams are considered as the most important features of all the others:

```
text = "To be or not to be"
tokens = nltk.word_tokenize(text)
bigrm = nltk.bigrams(tokens) # bigrm = nltk.ngrams(tokens,2)
print(list(bigrm))
```

```
[('To', 'be'), ('be', 'or'), ('or', 'not'), ('not', 'to'), ('to', 'be')]
```