AI for Trading

Term 2 Notes by Pranjal Chaubey
https://www.linkedin.com/in/pranjallchaubey/
https://github.com/pranjalchaubey

UDACITY

Text Processing Steps

18th January 2018

- Remove HTML Tags
- Convert to lowercase
- Remove punctuations & extra spaces
- Split the text into words/tokens
- Remove too common words ('a', 'the', 'are', 'of' etc)
  - STOP WORDS
- Identify different parts of speech & named entities
- Convert words into canonical forms using stemming and lemmatization

```
text = re.sub(r"[^a-zA-Z0-9]", " ", text)
```

All characters that are NOT EQUAL TO 'a' to 'z', 'A' to 'Z' or '0' to '9', substitute them with a " " space.

This effectively removes all the punctuation

TEXT NORMALIZATION = Lowercase + Remove Punctuation

TOKEN
- Term for a symbol
- Holds meaning and cannot be split further
- 'WORDS' in text processing

TOKENIZATION — Splitting sentence in a sequence of words

NLTK — Natural Language Toolkit

NLTK

```
from nltk.tokenize import word_tokenize
words = word_tokenize(text)
print(words)
```

import sent_tokenize — To tokenize sentences

import stop_words — Stop words in NLTK (from nltk.corpus)

```
words = [w for w in words if w not in stopwords.words("english")]
```

└ List comprehension to remove stop words from a text

import pos_tag

pos_tag (tokenized_words) — Labelling parts of speech through NLTK

```
for tree in parser.parse(sentence):
    tree.draw()
```

} Draws a tree of the parsed sentence

import pos_tag, ne_chunk

Usually performed on news articles for obvious reasons { └ Named entity recognition can only be done after text has been tokenized and Parts of Speech have been marked

Stemming — Reducing a word to its stem, or root form.

Lemmatization — Also reduces the words to their root form but uses a Dictionary for it.

branching
branched
branches
} = branch

NLTK uses WordNet Database by default.

~~import~~ from nltk.stem.wordnet import WordNetLemmatizer

"Jenna went back to University."

↓

Normalize → "jenna went back to university"

↓

Tokenize → <"jenna", "went", "back", "to", "university">

↓

Remove stop words → <"jenna", "went", "university">

↓

Usually Lemmatization is performed before stemming { Stem / Lemmatize → <"jenna", "go", "univers">

X ——————————— X

30th January 2019

Feature Extraction

- Bag of Words
- TF-IDF
- One-Hot Encoding
- Word Embeddings
- Word2Vec
- Glotte
- Embeddings for Deep Learning
- t-SNE

Bag of Words

- Treats each document as a Bag of Words
  - Unit of text being analysed
- Each document in the data will produce a set of words of different sizes
  - Inefficient!
    - Turn the document into a Vector of numbers

Little House on the Praire →
Marry had a little lamb →
The Silence of the Lambs →
Twinkle Twinkle Little Star →

little hous prairi mari lamb silenc twinkl star

vocabulary (V)

corpus (D)
(Set of Documents)

Gives the content for the vectors to be calculated

DOCUMENT TERM MATRIX

	litl	hous	prairi	mari	lamb	silenc	twinkl	star
Little House on the Prairie	1	1	1	0	0	0	0	0
Mary had a little Lamb	1	0	0	1	1	0	0	0
The Silence of the Lambs	0	0	0	0	1	1	0	0
Twinkle Twinkle Little Star	1	0	0	0	0	0	2	1

Multi-Dimensional Vector

This document term matrix can be used to find **Similarity** between 2 documents

1. ~~[illegible]~~ use dot product b/w two vectors — not recommended
2. use Cosine Similarity b/w two vectors

$$\cos(\theta) = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$

- identical vectors = 1
- orthogonal vectors = 0
- Opposite vectors = -1

27th May 2019

Problem with Bag-of-Words approach
↳ Treats every word as EQUALLY IMPORTANT

	little	hous	prairi	mari	lamb	silenc	twinkl	star
Little House on the prairie	1/3	1/1	1/1	0/1	0/2	0/1	0/1	0/1
Mary had a little lamb	1/3	0/1	0/1	1/1	1/2	0/1	0/1	0/1
The silence of the lambs	0/3	0/1	0/1	0/1	1/2	1/1	0/1	0/1
Twinkle twinkle little star	1/3	0/1	0/1	0/1	0/2	0/1	2/1	1/1
Document Frequency	3	1	1	1	2	1	1	1

Term frequencies / Document Frequency

$\propto$ frequency of occurence of a term in a document

$\propto \frac{1}{\text{number of documents it appears in}}$

TF-IDF

Term Frequency - Inverse Document Frequency

$$tf\text{-}idf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

term frequency

$$count(t, d) \div |d|$$

Raw count of the term 't' in a document 'd'

Total no. of terms in 'd'

inverse document frequency

$$\log\left(|D| \div |\{d \in D : t \in d\}|\right)$$

Total number of documents in a collection

The number of documents where 't' is present

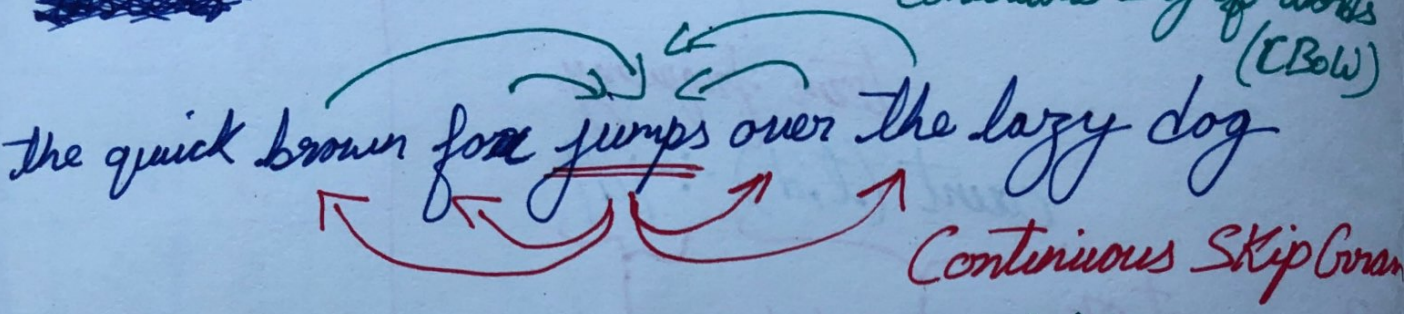
One-Hot Encoding

- BoW & TF-IDF work at document level
  - One-Hot encoding scheme works at the 'word' level
    - Similar to BoW, but applied on a ~~sentence~~ sentence typically

Breaks down in case of a Large Vocabulary

Word2Vec

Superior over one-hot encoding since it converts the vectors into fixed size, Word Embeddings



CBoW → predicts a word, given neighboring words

Skip-gram → predicts neighboring words, given a center word.

GloVe

Calculates co-occurence probabilites of words in a given content.

	solid	water
ice	P(solid/ice)	P(water/ice)
steam	P(solid/steam)	P(water/steam)

} Example

$$\frac{P(solid/ice)}{P(solid/steam)} >> 1$$

$$\frac{P(water/ice)}{P(water/steam)} \approx 1$$

t-SNE

t-Distributed Stochastic Neighbor Embedding

- Dimensionality Reduction Technique (like PCA)
  - Tries to maintain the Relative Distance between objects (unlike PCA)
    - ∴ used for visualizing Word-Embeddings

× ——— ×

28th May 2019

FINANCIAL STATEMENTS

Securities and Exchange Commissions
- 10-K Annual
- 10-Q Quarterly

10-K
- Part 1: Business Overview
  - Part 1, Item 1a
    - Business Risk Factors
- Part 2: Markets/Finance
  - Part 2, Item 7 & 7a
    - Market Risks
- Part 3: Governance
- Part 4: Full Financial

Electronic
Data
Gathering
Analysis
Retrieval

Introduction to Regexes

```
print("Hello World")
```
— 'Standard' python string

```
print(r"Hello World")
```
— 'Raw' String

Regexes use a lot of special characters including '\'. Raw string avoids the problem of python incorrectly interpreting our regex sequence.

Finding Words using Regexes

```
import re
sample_text = "This is a sample."
regex = re.compile(r'a')
matches = regex.finditer(sample_text)
for match in matches
    print(match)
```

Case-Sensitive

Finds all occurrences of the letter 'a'

match.span()[0] ⟺ match.start()
match.span()[1] ⟺ match.end()

Metacharacters

. ^ $ * + ? { } [ ] \ | ( )

'Find' a metacharacter by adding '\' backslash before them in the raw string.

Searching for Simple Patterns

\d — Matches any decimal digit [0-9]

\D — Matches any non-digit character [^0-9]
- including whitespace, newline et.al.

\s — Matches any whitespace character [\t\n\r\f\v]
- including newlines, carriage returns and form feeds

\S — Matches any non-whitespace character [^\t\n\r\f\v]
- Matches letters and numbers and punctuation ONLY

\w — Matches letters and numbers and underscore [a-zA-Z0-9_]
- NO punctuation

\W — Matches any non-alphanumeric character [^a-zA-Z0-9_]
- Including whitespaces
- Newlines AND Punctuation AND '@'

29th May 2019

Word Boundaries

\b — Determines word boundaries
- re.compile(r'\bclass\b')
  - Only finds those instances of 'class' that are standalone

\B — Opposite of \b, matches when the current position is not a word boundary.

Simple Metacharacters

. — Matches any character except newline \n

^ — Matches the sequence of characters located at the begining of a string

$ — Matches a sequence of characters at the end of a string

└ re.compile(r'watch$')

Character Sets

555-123-4567
655-777-7346

r'\d{3}.\d{3}.\d{4}'

Match exactly 3 copies of the previous regular expression

{x} — Match exactly 'x' copies of the regex before itself

└ Kinds of a looping system in Regex

[-] — 'Character Set', matches ANY of the characters, but never more than ONE.

└ Kinds of a Logical OR in Regex Universe

└ example matches either '-' or whitespace, but never both.

[6-9] – Acts like a range ('-')
⇒ [6789]

[a-d] – Range from [abcd]
↳ a to d

↳ But only ONE Character is matches at a time

[^6-9] – NOT 6, 7, 8 or 9
↳ Logical NOT inside the character set [ ]
(sort of)

re.compile(r'\d{3}.\d{3}.\d{3}[6-9]')

↳ Character type that we are looking for

↳ Loop Specification

↳ Higher level logic

{m, n} — AT Least 'm' repetitions, AT MAX 'n' repetitions.
↳ More sophisticated looping

a/{1,3}b

matches a/b

matches a//b

matches a///b

No Match a////b

No MATCH ab

Finding Complicated Patterns

? — Indicates the preceding regex being OPTIONAL.
- matches either 1 or 0 repetitions of the preceding regex

`r'Mt\.?'` ⟹ dot after Mt is optional

* — Matches **zero or more** characters of the preceding regular expression.
- Used when the length of characters is varying or not known

`r'\w*'` ⟹ Matches all alphanumeric characters till it encounters some other character type, or the string ends.

ab*
- a
- ab
- abb
- abb....b

(matches ab+) } Matching strings with ab*

\+ — Matches **ONE or More** characters of the preceding regex.

( ) — Defines a group

(ab)*
- ab
- abab
- ababab....

} Can use qualifiers like *, ?, or {m} before a group

| — Used as an OR inside the group.

(Mt|Mnt) ⟹ matches Mt or Mnt

Substitutions

```
sample_text = "Jack & Jill"
regex = re.compile(r'&')
new_text = regex.sub(r'and', sample_text)
```

↳ Jack and Jill

`.group(n)` — References the 'n'th group in the MatchObject.

↳ Indexing starts from 1, NOT 0.

```
regex.sub(r'\1 \3', sample_text)
```

Group Referencing

Replace sample text matches with groups 1 and 3

Flags

`re.IGNORECASE` — Forces the regex string to become case insensitive

```
regex = re.compile(r'walter', re.IGNORECASE)
```

31st May 2019

Introduction to BeautifulSoup

```
from bs4 import BeautifulSoup
with open(filename) as f:
    page_content = BeautifulSoup(f, 'lxml')
```

- `page_content` → BeautifulSoup Object
- `'lxml'` → BeautifulSoup Parser

```
print(page_content.prettify())
```

- Prints the parsed HTML content in a readable indented format

```
page_head = page_content.head
print(page_head.prettify())
```

Access the tags like regular attributes of the BS object

```
page_head = page_content.head
page_title = page_head.title
```

$\Leftrightarrow$

```
page_content.head.title
```

Tag objects contain the HTML Tags along with the text.

We usually want only the text

```
page_head.title.get_text()
```

Solves the problem

Getting HTML Tags' Attributes

```
<h1 id='intro'>
```

└ Need to extract this attribute

```
page_h1 = page_content.body.h1
```

Access the h1 tag

```
h1_id_attr = page_h1['id']
```

Extracts the value of ID attribute

Searching the Parse Tree

```
h2_list = page_content.find_all('h2')
```

This could be a HTML Tag, Tag Name, Attribute, or a Regular Expression

Returns a list

Different forms of '.find_all'

```
.find_all(['h2', 'p'])
.find_all('h2', id='know')
.find_all(id='intro')
```

```
.find_all(class_='h2_style')
.find_all(re.compile(r'i'))
```

} CSS class tag

} Regular Expression use case

```
page_content.head.contents
page_content.head.children
```

Returns children tags in the form of a list

Same as '.contents', but returns an iterable

```
.find_all('title', recursive=False)
```

Will only search for the Tag's Direct Children

3rd June 2019

Basic NLP Analysis

Readability Index

Flesch-Kincaid Grade Index

$$= 0.39\left(\frac{\#\text{words}}{\#\text{sentences}}\right) + 11.8\left(\frac{\#\text{syllables}}{\#\text{words}}\right) - 15.59$$

Gunning-Fog Grade Index

$$= 0.4\left[\frac{\#\text{words}}{\#\text{sentences}} + 100\left(\frac{\#\text{hard words}}{\#\text{words}}\right)\right]$$

words with 3 or more syllables

Young Adult Novels — Grade Level 8-10

Academic Paper in Theoretical Physics — Grade Level 16+

Financial Documents — 20+

4th June 2019

text = "Hi, How are you?
What is your name?
Where do you work?"

```
sent_tokenizer.tokenize(text)
```

↳ ['Hi, How are you?', 'What is your name?',
'Where do you work?']

```
[word_tokenize(s) for s in sent_tokenizer.tokenize(text)]
```

↳ [['Hi', 'How', 'are', 'you']
['What', 'is', 'your', 'name']
['Where', 'do', 'you', 'work']]

Bag-of-Words doesn't give any importance to the sequence of words.

↳ Same BoW for 2 sentences having a completely different meaning.

∴ We use TF-IDF to improve performance

for Bag of Words,

$$tf(w,d) = f_{w,d}$$

Term Frequency — word, document

Word Frequency

$f(w,d)$	$tf(w,d)$
0	0
1	1
10	10
100	100

To normalize Term Frequency $tf(w,d)$, we divide by Average Word Frequency.
We also use logarithms so that the numbers don't become too big for large documents.

$$tf(w,d) = \begin{cases} \dfrac{1+\log f_{w,d}}{1+\log a_d} & f_{w,d} > 0 \\ 0 & f_{w,d} = 0 \end{cases}$$

a_d = Average Word Frequency

$f(w,d)$	$tf(w,d)$
0	0
1	0.4
10	1.3
100	2.3

Only involves words in a single document.

⇓

Need to reduce the impact of common words that appear in multiple documents

— Include inverse document frequency, $idf(w)$

idf(w) for a word = inverse of the fraction of documents containing that word

$$idf(w) = \frac{N_d}{df_w}$$

N_d — Total no. of documents

df_w — Document frequency of a list of words

idf for word 'w'

df(w)	idf(w)
1	100
2	50
10	10
100	1

Higher idf for unique words

Lower idf for less common words

$$idf(w) = 1 + \log \frac{N_d}{df_w}$$

- Log avoids large numbers
- Adding '1' avoids a '0'

$tf(w,d) \cdot idf(w)$ — Converts documents into a collection of numbers, or document vectors.

To measure the similarity between two documents, take the Cosine Similarity b/w two tf·idf vectors or use Jaccard Similarity.

$$\text{Cosine Similarity} = \cos\theta = \frac{u \cdot v}{\|u\|\|v\|}$$

$$\text{Jaccard Similarity} = \frac{|u \cap v|}{|u \cup v|}$$