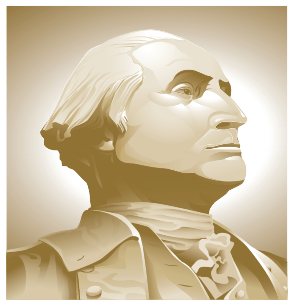
Regular Expression & Bag of Words

Natural Language Processing

Lecture 4



THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Regular Expression, Patterns, Anchors, Quantifiers, Groups and Methods

Regular Expressions

- Regular expressions are a powerful language for matching text patterns.
- The power of regular expressions is that they can specify patterns, not just fixed characters.
- Things get more interesting when you use quantifiers (specify repetition in the pattern).
- Regular expression patterns pack a lot of meaning into just a few characters.
- They are so dense, you can spend a lot of time debugging your patterns.

- Load a text and data.

```
import re
with open('re_text.txt','r') as f:
    text = f.read()
```

- Search for an specific charters. We use re.compile to specify the patter and finditer to iterate over a results.

```
pattern1 = re.compile(r'abc')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
print(text[0:3])
```

- **Period .** : Any Character Except New Line

```
print(20 * '-' + 'Back slash escapes special characters' + 20 * '-' )
pattern3 = re.compile(r'\.')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

More Patterns - Digits

- **\d** : Digit (0-9)

```
pattern1 = re.compile(r'\d')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- **\D** : Not a Digit (0-9), basically capital letter negate the lower case pattern.

```
pattern2 = re.compile(r'\D')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- **\d** : Example

```
pattern3 = re.compile(r'\d\d')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

More Patterns - Words

- \w : Word Character (a-z, A-Z, 0-9, _)

```
print(20 * '-' + 'Word' + 20 * '-' )
pattern1 = re.compile(r'\w')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- \W : Not a Word Character

```
pattern2 = re.compile(r'\W')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- \w : Example

```
pattern3 = re.compile(r'\w\.\w')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

More Patterns - Space

- \s : Whitespace (space, tab, newline)

```
print(20 * '-' + 'Space' + 20 * '-' )
pattern1 = re.compile(r'\s')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- \S : Not Whitespace (space, tab, newline)

```
pattern2 = re.compile(r'\S')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- \s : Example

```
pattern3 = re.compile(r'\s\.\s')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

Anchors - Word Boundary

- **\b** : Word Boundary - They don't match any characters. They are indicated by **white space or a non alpha numeric**. Start a line is a word boundary

```
print(20 * '-' + 'Word Boundry' + 20 * '-' )
pattern1 = re.compile(r'\bHa')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- **\B** : Not a Word Boundary

```
pattern2 = re.compile(r'\BHa')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- **\b** : Example

```
pattern3 = re.compile(r'\b\s\(N')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

Anchors - Caret and Dollar Sign

- ^ : Caret : Beginning of a String . It has to be at start. It cannot find a char in middle of text.

```
print(20 * '-' + 'Anchor' + 20 * '-' )
pattern1 = re.compile(r'^Start')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- $: Dollar Sign: End of a String

```
pattern2 = re.compile(r'end$')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- ^ : Example

```
pattern3 = re.compile(r'^a')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

Example - Simple Pattern

- Lets find all the phone number in the text

```
pattern1 = re.compile(r'\d\d\d')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- Most of the phone numbers has 3 digit followed by another 3 digits and the last 4 digits. Lets create that.

```
pattern2 = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- Read the re_fakenames.txt and find all the numbers.

```
pattern3 = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d')
matches = pattern3.finditer(text1)
for match in matches:
    print(match)
```

Example - Character Set 1

- Opening a bracket make a char set. Character set are having slight different rules.

```
pattern1 = re.compile(r'\d\d\d[-.]\d\d\d[-.]\d\d\d\d')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- We do not need a backslash in charset for period. Still only mach one char in string not more than one.

```
pattern2 = re.compile(r'[89]00[-.]\d\d\d[-.]\d\d\d\d')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- Dash in char set means values between from one to another char.

```
pattern3 = re.compile(r'[1-5]')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

Example - Character Set 2

- Lower case pattern

```
pattern1 = re.compile(r'[a-z]')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- Upper case and lower case pattern

```
pattern2 = re.compile(r'[a-zA-Z]')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- Caret in char set means negate the pattern

```
pattern3 = re.compile(r'[^a-zA-Z]')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

- lets find all the words ends with **at** but not a word **bat**

```
pattern1 = re.compile(r'[^b]at')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

Quantifiers

- `*` —— 0 or More
- `+` —— 1 or More
- `?` —— 0 or One
- `{3}` —— Exact Number
- `{3,4}` —— Range of Numbers (Minimum, Maximum)

Example - Quantifier Names

- Use quantifier for phone number searching.

```
pattern1 = re.compile(r'\d{3}[-.]\d{3}[-.]\d{4}')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- Find Mr names.

```
pattern2 = re.compile(r'Mr\.?\s[A-Z]\w*')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- Find Mr names and Ms names.

```
pattern3 = re.compile(r'M(r|s|rs)\.?\s[A-Z]\w*')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

```
pattern4 = re.compile(r'(Mr|Ms|Mrs)\.?\s[A-Z]\w*')
matches = pattern4.finditer(text)
for match in matches:
    print(match)
```

Example - Quantifier Email

- First email

```
pattern1 = re.compile(r'[a-zA-z]+@[a-zA-z]+\.com')
matches = pattern1.finditer(text)
for match in matches:
    print(match)
```

- Second email . Groups with parenthesis and vertical bar.

```
pattern2 = re.compile(r'[a-zA-z.]+@[a-zA-z]+\.(com|edu)')
matches = pattern2.finditer(text)
for match in matches:
    print(match)
```

- Third email.

```
pattern3 = re.compile(r'[a-zA-z0-9.-]+@[a-zA-z-]+\.(com|edu|net)')
matches = pattern3.finditer(text)
for match in matches:
    print(match)
```

- Alternative pattern. Lets interpret it.

```
pattern4 = re.compile(r'[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+')
matches = pattern4.finditer(text)
for match in matches:
    print(match)
```

Example - Quantifier URL and groups

- Find URL's.

```
import re
urls = '''https://www.google.com
          http://amir.com
          https://youtube.com
          https://www.epa.gov
        '''
pattern1 = re.compile(r'https?://(www\.)?\w+\.\w+')
matches = pattern1.finditer(urls)
for match in matches:
    print(match)
# ---------------------------
```

- Use group.

```
matches = pattern2.finditer(urls)
for match in matches:
    print(match)
# ---------------------------
```

- Use group method to find the domain.

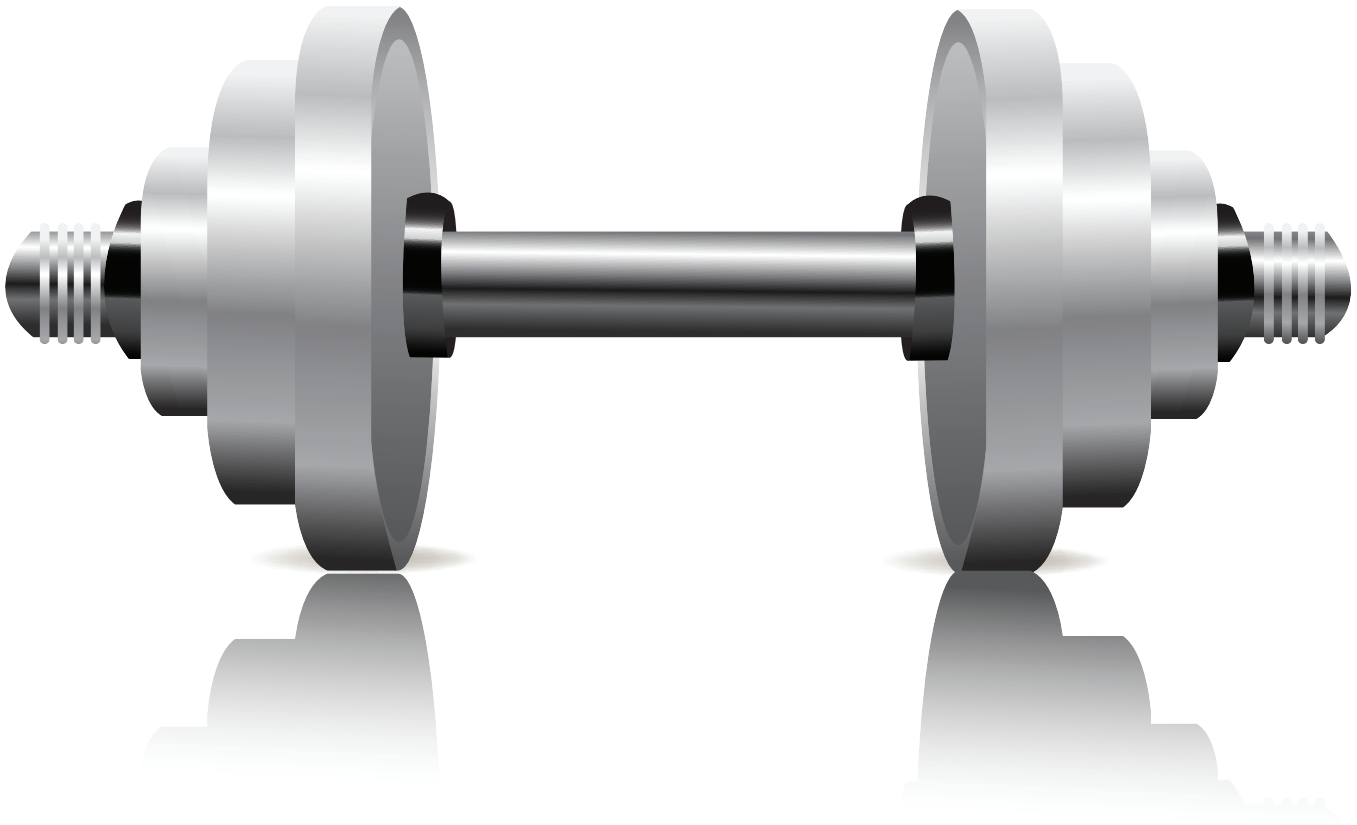
```
matches = pattern2.finditer(urls)
for match in matches:
    print(match.group(2))
```

What Else Can We Do With RE

- **re.findall(pat,str)**: finds all (nonoverlapping) matches. If nothing was found: returns an empty list
- **re.match(pat,str)** : matches only at the beginning of the string. If nothing was found: returns None
- **re.search(pat,str)**: matches anywhere in the string. If nothing was found: returns None

Exercise - Lecture 4

- Exercise 1 to 6
- Class-Ex-Lecture4.py



Build Your Vocabulary (Word Tokenization)

Tokenization

- Retrieving tokens from a document will require some string manipulation beyond just the **str.split().**
- You'll need to separate **punctuation** from words, like quotes at the beginning and end of a statement.
- And you'll need to split **contractions** like "we'll" into the words that were combined to form them.
- Once you've **identified** the tokens in a document that you'd like to **include in your vocabulary**.
- you'll return to the regular expression toolbox to try to combine words with similar meaning in a process called **stemming**.

Challenges stemming 1

- stemming is **grouping** the various inflections of a word into the same “bucket” or cluster.
- Imagine trying to remove verb endings like “ing” from “**ending**” so you’d have a stem called “end” to represent both words.
- And you’d like to stem the word “**running**” to “run,” so those two words are treated the same.
- And that’s tricky, because you have to remove not only the “ing” but also the **extra “n.”**
- But you want the word “**sing**” to stay whole.

Challenges stemming 2

- Or imagine trying to discriminate between a **pluralizing** “s” at the end of a word.
- Like “**words**” and a normal “s” at the end of words like “bus” and “**lens**.”
- Do isolated individual letters in a word or parts of a word provide any information at all about that **word’s meaning**?
- Can the letters be **misleading**?
- **Yes**

Building Vocabulary with a Tokenizers

- In NLP, tokenization is a particular kind of document **segmentation**.
- Segmentation breaks up text into **smaller chunks or segments**, with more focused information content.
- Segmentation can include breaking a document into paragraphs, paragraphs into sentences, sentences into phrases, or phrases into tokens (usually words) and punctuation.
- We focus on segmenting text into tokens, which is called **tokenization**.

Tokenizer

- Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline.
- A tokenizer breaks unstructured data, natural language text, into chunks of information that can be counted as discrete elements.
- This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning.

```
1 | sentence ='Thomas Jefferson began building Monticello at theage of 26.'
2 | print(sentence.split())
3 | print(str.split(sentence))
```

- As you can see, this built-in Python method already does a decent job tokenizing a simple sentence.
- Its only "mistake" was on the last word.

Tokenizers

- Normally you'd like tokens to be **separated from neighboring punctuation** and other meaningful tokens in a sentence.
- The token "26." is a perfectly fine representation of a **floating** point number 26.0.
- But that would make this token different than another **word "26"**
- A **good tokenizer** should strip off this extra character to create the word "26" as an equivalent class for the words "26," "26!", "26?", and "26."

Tokenizer - Token One Hot Encoding

- For now, let's forge ahead with your imperfect tokenizer.
- With a bit more Python, you can create a numerical vector representation for each word. These vectors are called one-hot vectors.

```
1  import numpy as np
2  import pandas as pd
3  sentence ='Thomas Jefferson began building Monticello at the age of 26.'
4  token_sequence = str.split(sentence)
5  vocab = sorted(set(token_sequence))
6  print(', '.join(vocab))
7  num_tokens = len(token_sequence)
8  vocab_size = len(vocab)
9  onehot_vectors = np.zeros((num_tokens,vocab_size), int)
10 for i, word in enumerate(token_sequence):
11     onehot_vectors[i, vocab.index(word)] = 1
12 ' '.join(vocab)
13 print(onehot_vectors)
14 df = pd.DataFrame(onehot_vectors, columns=vocab)
15 print(df)
```

Tokenizer - One Hot Encoding

- One nice feature of this vector representation of words and **tabular representation** of documents is that **no information** is lost.
- As long as you keep track of which words are indicated by which column, you can **reconstruct the original document** from this table of one-hot vectors.
- And this reconstruction process is 100% accurate, even though your tokenizer was only 90% accurate at generating the tokens you thought would be useful.
- They're a good choice for any model or NLP pipeline that needs to **retain all the meaning** inherent in the original text.
- You could also play a sequence of one-hot encoded vectors back

One Hot - Long Document

- This representation of a sentence in one-hot word vectors **retains all the detail, grammar**, and order of the original sentence.
- For a long document this might not be practical.
- Your document size (the length of the vector table) would grow to be huge.
- Let's say you have a meager 3,000 books with 3,500 sentences each and 15 words per sentence

```
num_rows = 3000 * 3500 * 15
print('{} Number of Rows'.format(num_rows))
num_bytes = num_rows * 1000000
print('{} Bytes'.format(num_bytes))
Size = num_bytes / 1e9
print('{} Terabytes'.format(Size/1000))
```

A Token Improvement

- In some situations, **other characters** besides spaces are used to separate words in a sentence.
- You need your tokenizer to split a sentence not just on whitespace, but also on punctuation such as commas, periods, quotes, semicolons, and even hyphens (dashes).
- In some cases you want these punctuation marks to be treated like words, as independent tokens.

```
1 import re
2 sentence ='Thomas Jefferson began building Monticello at theage of 26.'
3 tokens = re.split(r'[-\s.,;!?]+', sentence)
4 print(tokens)
```

NLTK Tokenizer

- As you can imagine, tokenizers can easily become complex. NLTK and Stanford CoreNLP have been around the longest and are the most widely used

```
1 | from nltk.tokenize import RegexpTokenizer
2 | tokenizer = RegexpTokenizer(r'\w+|$[0-9.]+|\S+')
3 | sentence ='Thomas Jefferson began building Monticello at theage of 26.'
4 | print(tokenizer.tokenize(sentence))
5 |
6 | from nltk.tokenize import TreebankWordTokenizer
7 | sentence = "Monticello wasn't designated as UNESCO World Heritage Site until 1987."
8 | tokenizer = TreebankWordTokenizer()
9 | print(tokenizer.tokenize(sentence))
```

Contractions

- You might wonder why you would split the contraction **wasn't** into was and n't.
- For some applications, like **grammar-based NLP models** that use syntax trees.
- It's important to separate the words was and not to allow the syntax tree parser to have a **consistent, predictable set of tokens** with known grammar rules as its input.
- There are a **variety of standard and nonstandard ways** to contract words.

```
1 from nltk.tokenize.casual import casual_tokenize
2 message = "RT @TJMonticello Best day everrrrrrr at Monticello." \
3           "Awesommmmmmeeeeeeee day :*)"
4 print(casual_tokenize(message))
5 print(casual_tokenize(message, reduce_len=True, strip_handles=True))
```

Stop Words

- Stop words are **common words** in any language that occur with a **high frequency** but carry **much less substantive information** about the meaning of a phrase.
- Some **common** stop words : a, an, the, this, and, or, of, on
- Historically, stop words have been **excluded** from NLP pipelines.
- **Designing** a filter for stop words depends on your **particular application**.
- Vocabulary size will drive the **computational complexity** and **memory requirements** of all subsequent steps in the NLP pipeline

Canonical Stop Words

- To get a complete list of "canonical" stop words, NLTK is probably the most generally applicable list.

```
import nltk
nltk.download('stopwords')
stop_words = nltk.corpus.stopwords.words('english')
print(stop_words)
print(len(stop_words))
```

- Here's a comparison of sklearn stop words and nltk stop words.
- NLTK and sklearn agree on fewer than a third of their stop words (119 out of 378).

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS as sklearn_stop_words
print(len(sklearn_stop_words))
print(len(set(stop_words).union(sklearn_stop_words)))
print(len(set(stop_words).intersection(sklearn_stop_words)))
```

Normalizing your Vocabulary

- Another vocabulary reduction technique is to normalize your vocabulary.
- So that tokens that mean similar things are **combined into a single**, normalized form.
- Case folding is when you consolidate multiple "spellings" of a word that differ only in their **capitalization**.
- Words can become case "denormalized" when they are capitalized because of their presence at the **beginning of a sentence**.
- They're written in ALL CAPS for **emphasis**.

Stemming

- Another common vocabulary normalization technique is to eliminate the small meaning differences of **pluralization** or **possessive endings of words**, or even **various verb forms**.
- For example, the words **housing** and **houses** share the same stem, house.
- Stemming **removes suffixes** from words in an attempt to **combine words with similar meanings** together under their common stem.
- A stem **isn't required** to be a properly **spelled word**.

```
1 | import re
2 | def stem(phrase):
3 |     return ' '.join([re.findall('^(.*ss|.*?)(s)?$', word)
4 |                      [0][0].strip("'") for word in phrase.lower().split()])
5 | print(stem("Doctor House's calls"))
6 | from nltk.stem.porter import PorterStemmer
7 | stemmer = PorterStemmer()
8 | print(' '.join([stemmer.stem(w).strip("'")
9 |                 for w in "dish washer's washed dishes".split()]))
```

Lemmatization

- If you have access to information about connections between the **meanings of various words**, you might be able to associate several words together even if their **spelling is quite different**.
- you can use the **part of speech** to identify a better "root" of a word than stemming could?
- Stemmers would strip the "er" ending from "**better**" and return the stem "**bett**" or "**bet**."
- More similar words like "betterment," "**best**," or even "**good**" and "**goods**."

```
1 import nltk
2 nltk.download('wordnet')
3 from nltk.stem import WordNetLemmatizer
4 lemmatizer = WordNetLemmatizer()
5 print(lemmatizer.lemmatize("better"))
6 print(lemmatizer.lemmatize("better", pos="a"))
7 print(lemmatizer.lemmatize("better", pos="n"))
```

Bag of Words & TFIDF

Math with Words 1

- Having collected and **counted words** (tokens), and bucketed them into **stems or lemmas**, it's time to do something **interesting with them**.
- Detecting words is useful for simple tasks, like **getting statistics about word usage** or doing keyword search.
- But you'd like to know **which words are more important** to a particular document and across the corpus as a whole.
- Then you can use that **"importance" value to find relevant document**s in a corpus based on keyword importance within each document.

Math with Words 2

- The next step in your adventure is to turn the words into **continuous numbers** rather than just integers representing word counts or binary.
- With representations of words in a continuous space, you can operate on their representation with more **exciting math.**
- Your goal is to find **numerical representation** of words that somehow **capture** the **importance** or information content of the words they represent.
- Then you can use that "importance" value to find relevant documents in a corpus based on keyword importance within each document.

Bag of Words

- We created our first vector space model of a text (**one-hot encoding**).
- And this binary BOW vector makes a great index for document retrieval when loaded into a data structure such as a Pandas Data Frame.
- Then looked at even more useful vector representation that **counts the number of occurrences,** or **frequency**, of each word in the given text.
- As a first approximation, you assume that the **more times a word occurs, the more meaning** it must contribute to that document.
- A document that refers to **"wings" and "rudder**" frequently may be more relevant to a problem involving **jet airplanes** or air travel than document with **"cats" and "gravity.**"

Counting Occurrences

- Let's look at an example where counting occurrences of words is useful

```
1  from nltk.tokenize import TreebankWordTokenizer
2  from collections import Counter
3  sentence = "The faster Harry got to the store, the faster Harry " \
4             "the faster, would get home."
5  tokenizer = TreebankWordTokenizer()
6  tokens = tokenizer.tokenize(sentence.lower())
7  print(tokens)
8
9  bag_of_words = Counter(tokens)
10 print(bag_of_words)
```

- collections.Counter object is an **unordered collection**, also called a bag or multiset.
- For short documents like this one, the unordered bag of words still **contains a lot of information**.
- And the information in a bag of words is **sufficient to do some powerful things** such as detect spam.

Term Frequency

- Specifically, the number of times a **word occurs in a given document** is called the term frequency, commonly abbreviated TF.

```
1  | from nltk.tokenize import TreebankWordTokenizer
2  | from collections import Counter
3  | sentence = "The faster Harry got to the store, the faster Harry " \
4  |            "the faster, would get home."
5  | tokenizer = TreebankWordTokenizer()
6  | tokens = tokenizer.tokenize(sentence.lower())
7  | bag_of_words = Counter(tokens)
8  | print(bag_of_words.most_common(4))
9  |
10 | times_harry_appears = bag_of_words['harry']
11 | num_unique_words = len(bag_of_words)
12 | tf = times_harry_appears / num_unique_words; print(tf)
```

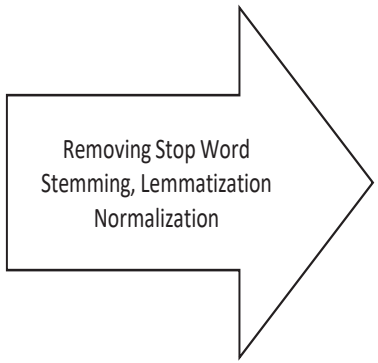
- collections.Counter object is an **unordered collection**, also called a bag or multiset.
- For short documents like this one, the unordered bag of words still **contains a lot of information**.
- And the information in a bag of words is **sufficient to do some powerful things** such as detect spam.

TF Example

Sentence 1: He is a good boy

Sentence 2: She is a good girl

Sentence 3: Boy and girl are good



Sentence 1: [,]

Sentence 2: [,]

Sentence 3: [,]

Words	Frequency

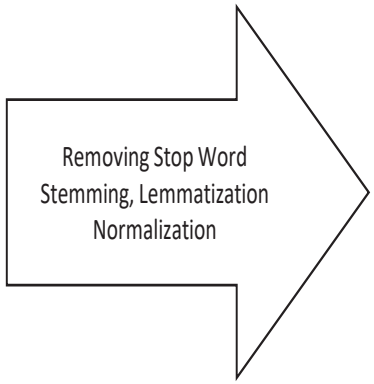
Sentence 1			
Sentence 2			
Sentence 3			

TF Example Solution

Sentence 1: He is a good boy

Sentence 2: She is a good girl

Sentence 3: Boy and girl are good



Sentence 1: [good , boy]

Sentence 2: [good , girl]

Sentence 3: [boy , girl, good]

Words	Frequency
good	3
boy	2
girl	2

	good	boy	girl
Sentence 1	1	1	0
Sentence 2	1	0	1
Sentence 3	1	1	1

Term Frequency Descriptive Example

- Let's say you find the word "dog" 3 times in document A and 100 times in document B
- Clearly "dog" is way more important to document B.
- But wait. Let's say you find out document A is a 30-word email to a veterinarian and document B is War & Peace (approx 580,000) words!).
- Your first analysis was straight-up backwards.
  - $TF(dog, documentA) = 3/30 = .1$
  - $TF(dog, documentB) = 100/580000 = .00017$
- Now you have something you can see that describes "something" about the two documents and their relationship to the word "dog" and each other.

Term Frequency Counting Example

- Let's look at a bigger piece of text. Take these first few paragraphs from the Wikipedia article on kites:

```
1  import nltk
2  from collections import Counter
3  from nltk.tokenize import TreebankWordTokenizer
4
5  with open('kite.txt','r') as f:
6      kite_text = f.read()
7
8  tokenizer = TreebankWordTokenizer()
9  tokens = tokenizer.tokenize(kite_text.lower())
10 token_counts = Counter(tokens)
11 print(token_counts)
12 print(20 *'-')
13 nltk.download('stopwords', quiet=True)
14 stopwords = nltk.corpus.stopwords.words('english')
15 tokens = [x for x in tokens if x not in stopwords]
16 kite_counts = Counter(tokens)
17 print(kite_counts)
```

- The terms kite(s), wing, and lift are all important.

Vectorizing TF

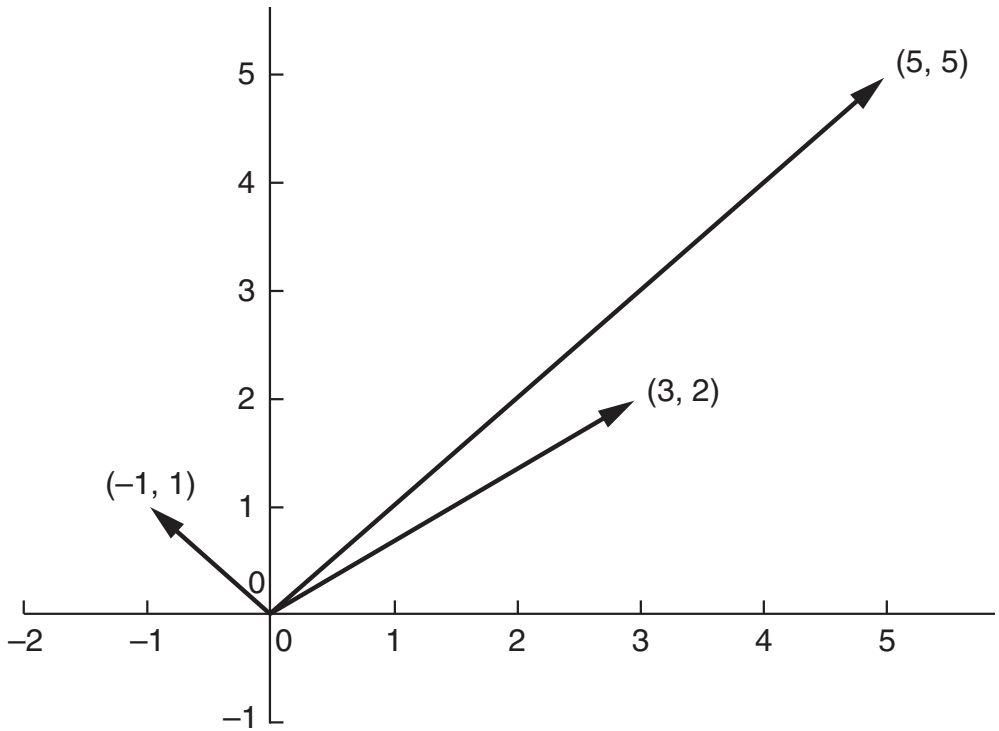
```
from nltk.tokenize import TreebankWordTokenizer
from collections import OrderedDict, Counter
import copy
import pandas as pd
tokenizer = TreebankWordTokenizer()
docs = ["The faster Harry got to the store, the faster and faster "
        "Harry would get home."]
docs.append("Harry is hairy and faster than Jill.")
docs.append("Jill is not as hairy as Harry.")
doc_tokens = []
for doc in docs:
    doc_tokens += [sorted(tokenizer.tokenize(doc.lower()))]
print(len(doc_tokens[0]))
all_doc_tokens = sum(doc_tokens, []) ; print(len(all_doc_tokens))
lexicon = sorted(set(all_doc_tokens)); print(len(lexicon))
zero_vector = OrderedDict((token, 0) for token in lexicon); print(zero_vector)
doc_vectors = []
for doc in docs:
    vec = copy.copy(zero_vector)
    tokens = tokenizer.tokenize(doc.lower())
    token_counts = Counter(tokens)
    for key, value in token_counts.items():
        vec[key] = value / len(lexicon)
    doc_vectors.append(vec)
df = pd.DataFrame(doc_vectors);print(df)
```

Vector Spaces

- Vectors are the primary building blocks of **linear algebra**, or vector algebra.
- They're an ordered list of numbers, or coordinates, in a **vector space**.
- They describe a **location or position in that space**.
- Or they can be used to identify a particular **direction and magnitude** or distance in that space.
- A space is the collection of **all possible vectors** that could appear in that space.
- So a vector with two values would lie in a 2D vector space, a vector with three values in 3D vector space, and so on.

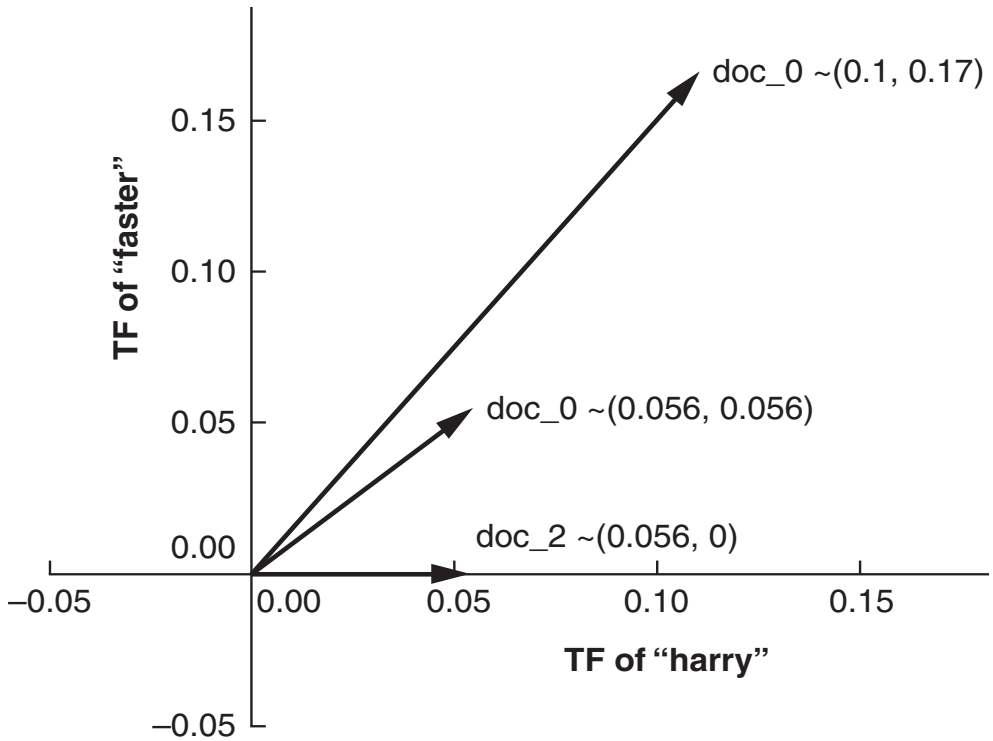
Vector Spaces Representation

- One way to draw the 2D vectors (5, 5), (3, 2), and (-1, 1). The head of a vector (represented by the pointy tip of an arrow) is used to identify a location in a vector space.



TF Vector Spaces Representation

- Two vectors are "similar" if they share similar direction. They might have similar magnitude (length), which would mean that the word count (term frequency) vectors are for documents of about the same length.



Cosine Similarity

- Cosine similarity is merely the cosine of the angle between two vectors (theta).
- $A \cdot B = |A||B| * cos\theta$

```
1 import numpy as np
2 from numpy.linalg import norm
3 def cosine_sim(a,b):
4     return  np.dot(a, b)/(norm(a)*norm(b))
5 print(cosine_sim([1,1], [1,1]))
6 print(cosine_sim([1,1], [-1,1]))
7 print(cosine_sim([1,1], [-1,-1]))
```

- A cosine similarity of 0 represents two vectors that share no components.
- Means two document has noting in common.

Topic Modeling

- Now back to your document vectors.
- Word counts are useful, but **pure word count**, even when normalized by the length of the document, **doesn't tell you much about the importance of that word in that document** relative to the rest of the documents in the corpus.
- Say you have a corpus of every kite book ever written.
- "Kite" would almost surely occur many times in every book (document) you counted, but that doesn't provide any new information
- It doesn't help distinguish between those documents.
- For this you need another tool.

Inverse Document Frequency (IDF)

- Let's take your term frequency counter from earlier and expand on it.
- You can count tokens and bin them up two ways: **per document and across the entire corpus**.
- Say you have a corpus of every kite book ever written.
- You're going to be **counting just by document**.
- It doesn't help distinguish between those documents.
- For this you need another tool.

Document Frequency - Kite text

- Let's return to the Kite example from **Wikipedia** and grab another section (the History section)
- First let's get the total **word count** for each document in your corpus, intro_doc and history_doc:
- Now with a couple tokenized kite documents in hand, let's look at the term frequency of "kite" in each document.
- Lets look at the **Term Frequencies** and see if we can answer the following question.
- If the value of TF for intro is **twice bigger** then can we say.
- The intro section twice as much about kites.

TF Code From Scratch

```
1  | kite_intro = kite_text.lower()
2  | intro_tokens = tokenizer.tokenize(kite_intro)
3  |
4  | kite_history = kite_history.lower()
5  | history_tokens = tokenizer.tokenize(kite_history)
6  |
7  | intro_total = len(intro_tokens) ; print(intro_total)
8  | history_total = len(history_tokens) ; print(history_total)
9  |
10 | intro_tf = {};history_tf = {}
11 |
12 | intro_counts = Counter(intro_tokens)
13 | history_counts = Counter(history_tokens)
14 | intro_tf['kite'] = intro_counts['kite'] / intro_total
15 | print('Term Frequency of "kite" in intro is: {:.4f}'.format(intro_tf['kite']))
16 |
17 | history_tf['kite'] = history_counts['kite'] / history_total
18 | print('Term Frequency of "kite" in history is: {:.4f}'.format(history_tf['kite']))
19 |
20 | intro_tf['and'] = intro_counts['and'] / intro_total
21 | history_tf['and'] = history_counts['and'] / history_total
22 | print('Term Frequency of "and" in history is: {:.4f}'.format(history_tf['and']))
```

Inverse Document Frequency - IDF

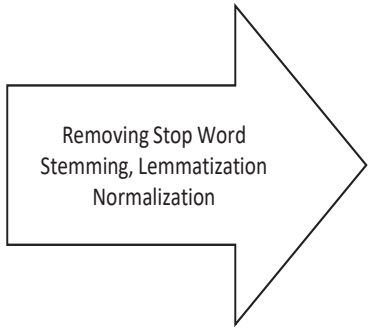
- A good way to think of a term's inverse document frequency is this
- How strange is it that this token is in this document?
- If a term appears in one document a lot of times, but occurs rarely in the rest of the corpus, one could assume it's important to that document specifically.
- Your first step toward topic analysis
- A term's IDF is merely **the ratio of the total number of documents to the number of documents the term appears in**.

IDF Code From Scratch

```
for doc in [intro_tokens, history_tokens]:
    if 'and' in doc:
        num_docs_containing_and += 1
    if 'kite' in doc:
        num_docs_containing_kite += 1
intro_total = len(intro_tokens) ;history_total = len(history_tokens)
intro_counts = Counter(intro_tokens);history_counts = Counter(history_tokens)
intro_tf = {};history_tf = {};intro_tfidf = {};history_tfidf = {}
intro_tf['and'] = intro_counts['and'] / intro_total
history_tf['and'] = history_counts['and'] / history_total
intro_tf['kite'] = intro_counts['kite'] / intro_total
history_tf['kite'] = history_counts['kite'] / history_total
num_docs = 2;intro_idf = {};history_idf = {};num_docs = 2
intro_idf['and'] = num_docs / num_docs_containing_and
history_idf['and'] = num_docs / num_docs_containing_and
intro_idf['kite'] = num_docs / num_docs_containing_kite
history_idf['kite'] = num_docs / num_docs_containing_kite
intro_tfidf['and'] = intro_tf['and'] * intro_idf['and']
intro_tfidf['kite'] = intro_tf['kite'] * intro_idf['kite']
history_tfidf['and'] = history_tf['and'] * history_idf['and']
history_tfidf['kite'] = history_tf['kite'] * history_idf['kite']
```

TFDIF Example Solution

Sentence 1: He is a good boy
Sentence 2: She is a good girl
Sentence 3: Boy and girl are good



Sentence 1: [good ,boy]
Sentence 2: [good , girl]
Sentence 3: [good, boy, girl]

Words	Frequency
good	3
boy	2
girl	2

$$TF = \frac{Number\ of\ repetion\ of\ word\ in\ sentence}{Total\ number\ of\ words\ in\ sentence}$$

$$IDF = \log\left(\frac{Number\ sentence}{Number\ of\ senetnces\ containg\ word}\right)$$

$$TFIDF = TF * IDF$$

TF			
	Sent 1	Sent 2	Sent 3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

Words	IDF
good	log(3/3) = 0
boy	log(3/2)
girl	log(3/2)

TFIDF			
	good	boy	girl
Sent 1	0 * ½ = 0	½* log(3/2)	0*log(3/2)
Sent 2	½ * 0	0*log(3/2)	½*log(3/2)
Sent 3	1/3*0	1/3*log(3/2)	1/3*log(3/2

Big Data & TF and IDF Calculation

- Let's say, though, you have a corpus of 1 million documents (maybe you're baby-Google), someone searches for the word "cat". The raw IDF of this is
- $1,000,000/1 = 1,000,000$
- Let's imagine you have 10 documents with the word "dog" in them. Your IDF for "dog" is
- $1,000,000/10 = 100,000$
- That's a big difference. You scale all your word frequencies (and document frequencies) with the log() function, the inverse of exp().
- idf = log(1,000,000/1) = 6 and idf = log(1,000,000/10) = 5

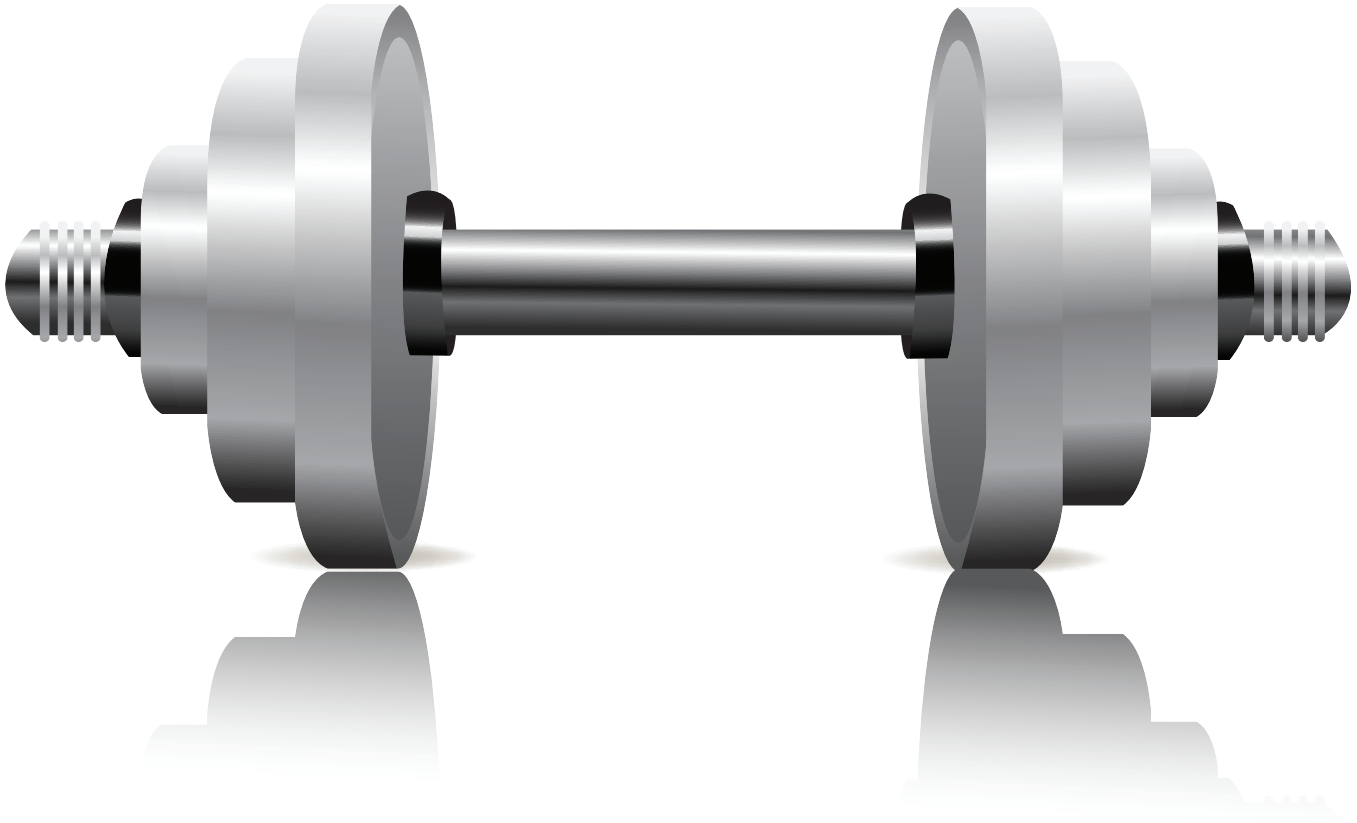
- And then finally, for a given term, t, in a given document, d, in a corpus, D, you get:
  - $tf(t,d) = \frac{count(t)}{count(d)}$
  - $idf(t,D) = \log \frac{number\ of\ documents}{number\ of\ documents\ containing\ t}$
  - $tfidf(t,d,D) = tf(t,d) * idf(t,D)$
- So the more times a word appears in the document, the TF (and hence the TF-IDF) will go up.
- At the same time, as the number of documents that contain that word goes up, the IDF (and hence the TF-IDF) for that word will go down.

- Here's how you can use sklearn to build a TF-IDF matrix

```
1  from sklearn.feature_extraction.text import TfidfVectorizer
2  corpus = [
3       'This is the first document.',
4       'This document is the second document.',
5       'And this is the third one.',
6       'Is this the first document?',
7   ]
8  vectorizer = TfidfVectorizer()
9  X = vectorizer.fit_transform(corpus)
10 print(vectorizer.get_feature_names())
11 print(X.shape)
```

Exercise - Lecture 4

- Exercise 7 to 10
- Class-Ex-Lecture4.py



PCA, LSA and LDA

Important Concepts

- **Vectors, Vector Length, Vector Addition**
- **Matrices, Matrix Notation,**
- **Scalar Multiplication, Inner Product, Orthogonality**
- **Normal Vector, Orthonormal Vectors**
- **Square Matrix, Transpose, Matrix Multiplication, Identity Matrix**
- **Diagonal Matrix, Determinant, Eigenvectors and Eigenvalues**

Eigenvectors and Eigenvalues

- An eigenvector is a nonzero vector that satisfies the equation
- $Av = \lambda v$
- where **A** is a square matrix,λ is a scalar, and v is the eigenvector.
- λ is called an eigenvalue. Eigenvalues and eigenvectors are also known as, respectively, characteristic roots and characteristic vectors, or latent roots and latent vectors.
- You can fin eigenvalues and eigenvectors by treating a matrix as a system of linear equations and solving for the values of the variables that make up the components of the eigenvector.
- Example: Find eigenvalues and eigenvectors of matrix

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

Singular Value Decomposition

- Singular value decomposition (SVD) can be looked at from three mutually compatible points of view.

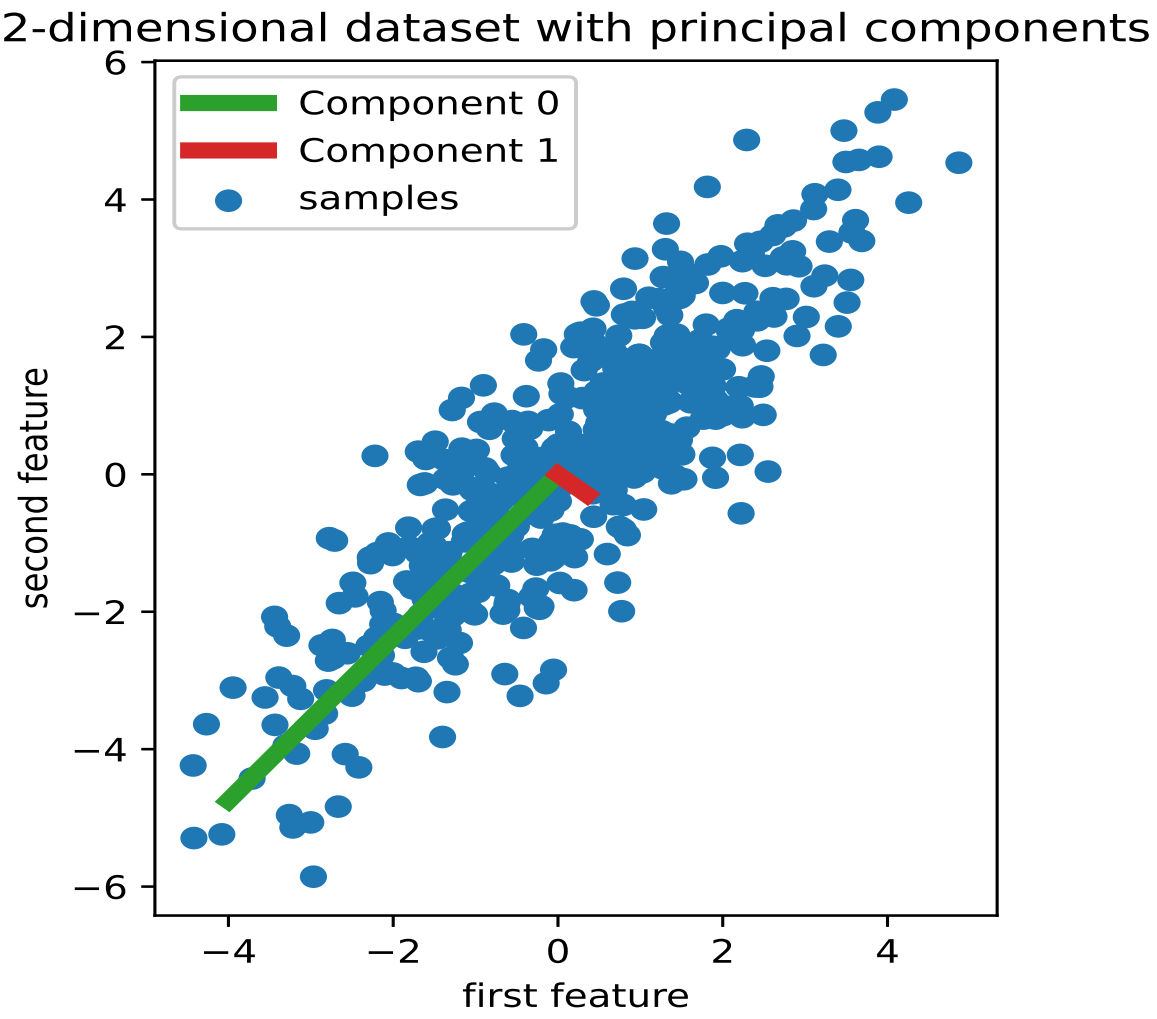
  1- We can see it as a method for transforming correlated variables into a set of uncorrelated ones that better expose the various relationships among the original data items.

  2- At the same time, SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation.

  3- That once we have identified where the most variation is, it's possible to find the best approximation of the original data points using fewer dimensions.

- SVD can be seen as a method for data reduction.

Principal Component (PCA)



Click here to open an Example.

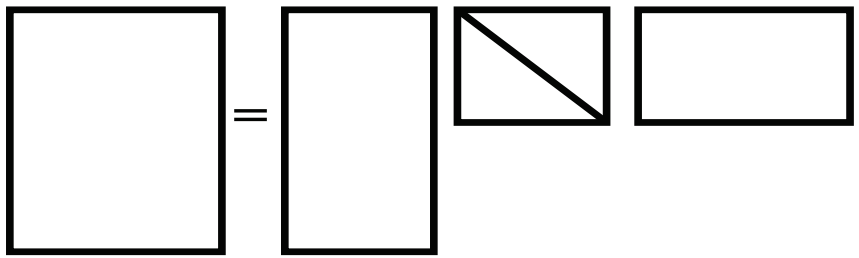
LSA - Example of Text Data

- doc1 = ”Data Science Machine Learning”
- doc2 = ”Money fun Family Kids home”
- doc3 = ”Programming Java Data Structures”
- doc4 = ”Love food health games energy fun”
- doc5 = ”Algorithms Data Computers”

LSA - Matrix of Words

	algorithms	computers	data	energy	family	food	fun	games	health	home	java	kids	learning	love	machine	money	programming	science	structures
0	0.00	0.00	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.54	0.00	0.54	0.00	0.00	0.54	0.00
1	0.00	0.00	0.00	0.00	0.46	0.00	0.37	0.00	0.00	0.46	0.00	0.46	0.00	0.00	0.00	0.46	0.00	0.00	0.00
2	0.00	0.00	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.54	0.00	0.00	0.00	0.00	0.00	0.54	0.00	0.54
3	0.00	0.00	0.00	0.42	0.00	0.42	0.34	0.42	0.42	0.00	0.00	0.00	0.00	0.42	0.00	0.00	0.00	0.00	0.00
4	0.64	0.64	0.43	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

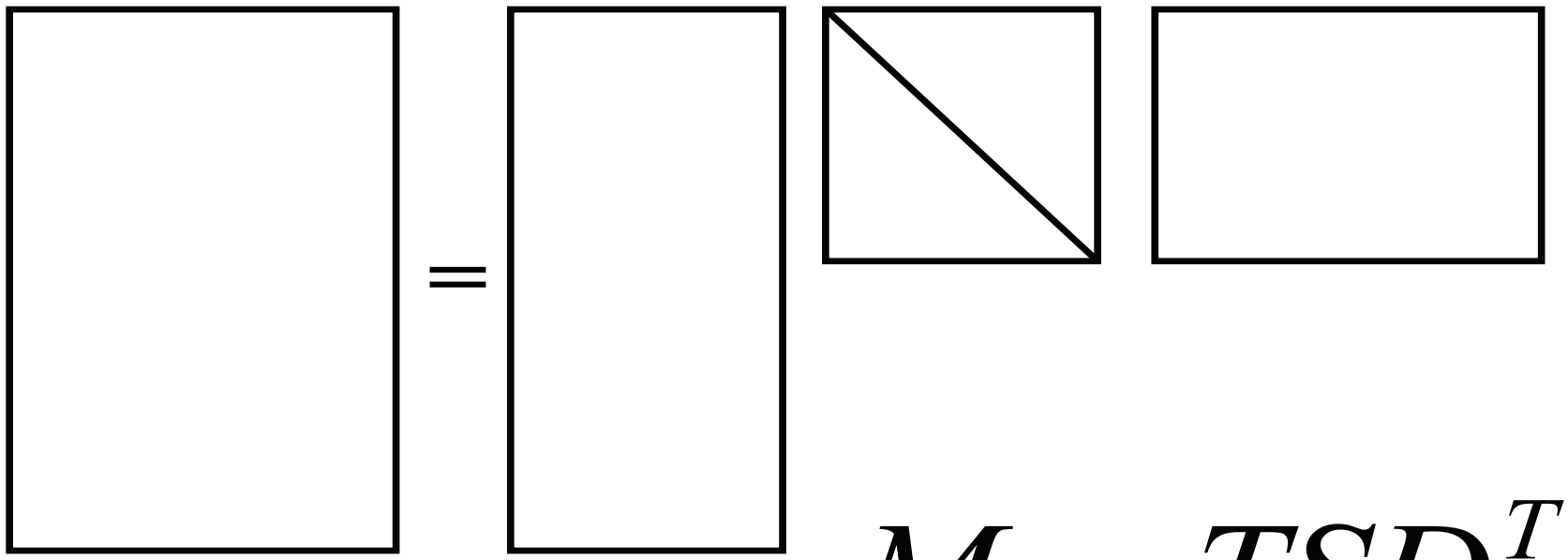
LSA - SVD Calculation



Singular value Decomposition

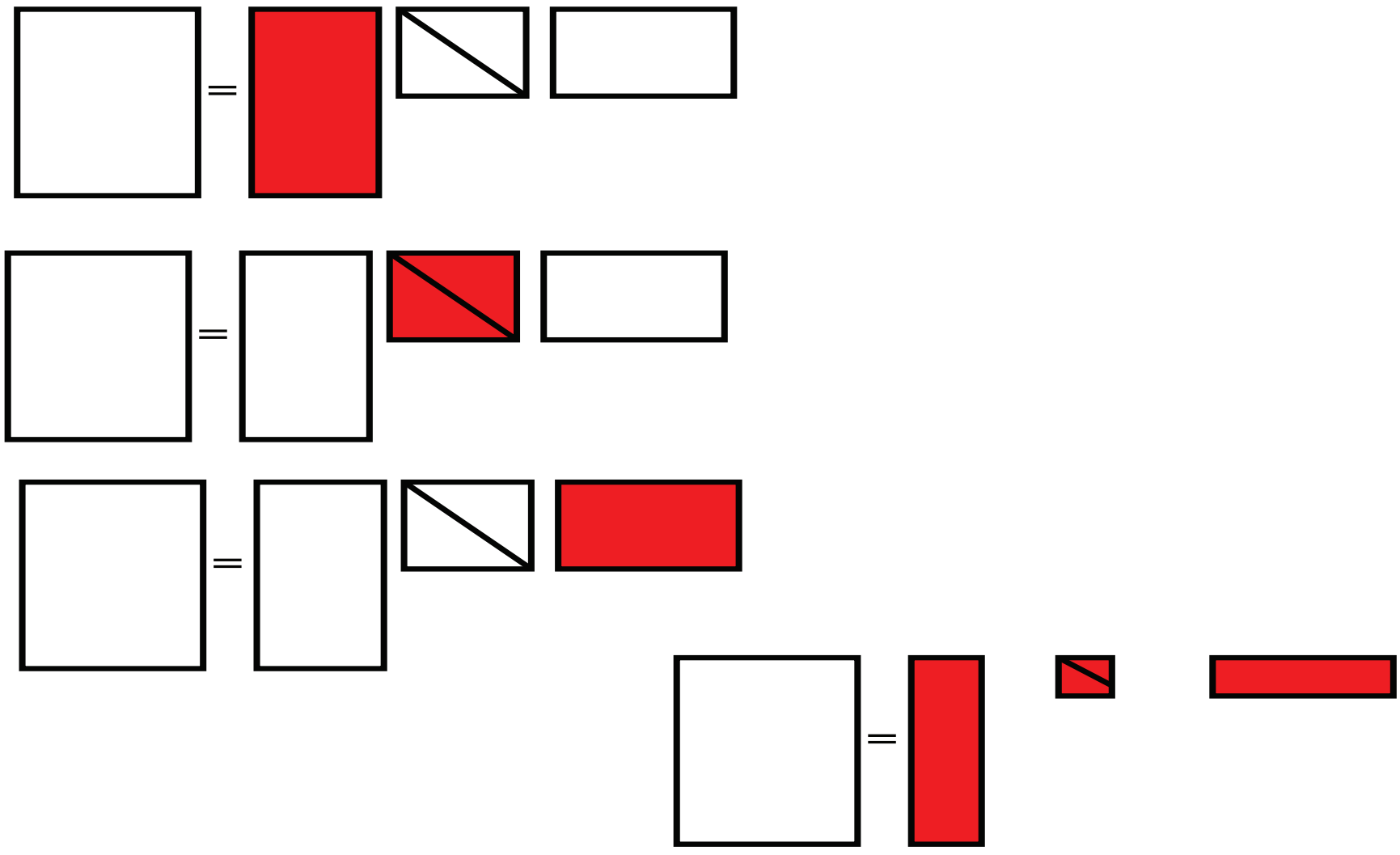
TFIDF

Words (states)



$$M = TSD^T$$

LSA - Visulization



LSA Code

```
doc1 = "Data Science Machine Learning"
doc2 = "Money fun Family Kids home"
doc3 = "Programming Java Data Structures"
doc4 = "Love food health games energy fun"
doc5 = "Algorithms Data Computers"

doc_complete = [doc1, doc2, doc3, doc4, doc5]

from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X =vectorizer.fit_transform(doc_complete)

from sklearn.decomposition import TruncatedSVD
lsa = TruncatedSVD(n_components=2,n_iter=100)
lsa.fit(X)
terms = vectorizer.get_feature_names()

for i,comp in enumerate(lsa.components_):
    termsInComp = zip(terms,comp)
    sortedterms = sorted(termsInComp, key=lambda x: x[1],reverse=True)[:10]
    print("Concept %d:" % i)
    for term in sortedterms:
        print(term[0])
    print(" ")
```

Exercise - Lecture 4

- Exercise 11 to 13
- Class-Ex-Lecture4.py