Week05 - Summary

Analysing Unstructured Data

- Social media data is unstructured
- consists mainly of text, images, videos
- Interesting questions
    - Classification of texts or images
    - Information extraction
    - Building ML models based on unstructured data

Unstructured Data

- Unstructured data refers to information that usually does not have a pre-defined model, such as text, images, videos,…
- Unstructured (text) data is typically text-heavy, but may contain dates, numbers and facts as well as meta-data
- This results in ambiguities that make it more difficult to understand than data is structured databases

Structured Data

- Data in fields
- Easily stored in databases
- E.g.:
    - Sensor data
    - Financial data
    - Click streams

    - Measurements

Text Classification

Modelling spam detection as classification task

- Input:
    - Emails
    - SMS messages
    - Facebook pages
- Predict:
    - 1 (Spam)
    - 0 (not-spam)

Core idea: text to feature vectors

- Represent document as a multiset of words
- Keep frequency information
- Disregard grammar and word order

- Feature Vector

  Which words occur how often in a given text?

Tokenisation

```
"Friends, Romans, Romans,
countrymen"
```

⬇️

```
["Friends",
"Romans",
"Romans",
"countrymen"]
```

- Split a string (document) into pieces called **tokens**
- Possibly remove some characters, e.g., punctuation
- Remove "stop words" such as "a", "the", "and" which are considered irrelevant

Normalisation

```
["Friends",
"Romans",
"Romans",
"countrymen"]
```

⬇️

```
["friend",
"roman",
"roman",
"countrymen"]
```

- Map similar words to the same token
- Stemming/lemmatisation
  - Avoid grammatical and derivational sparseness
  - E.g., "was" ⇒ "be"
- Lower casing, encoding

- E.g., “Naive” ⇒ “naive”

Indicator Features

```
["friend",
 "roman",
 "roman",
"countrymen"]

⬇️

{"friend": 1,
 "roman": 1,
"countrymen": 1}
```

- Binary indicator feature for each word in a document
- Ignore frequencies

Term Frequency Weighting

- Term Frequency
  - Give more weight to terms that are common in document
  - **TF = |occurrences of term in doc|**

- Damping
  - Sometimes want to reduce impact of high counts
  - **TF = log(|occurrences of term in doc|)**

TF-IDF Weighting

```
["friend",
"roman",
"countrymen"]
```

⬇️

```
{"friend": 0.1,
"roman": 0.8,
"countrymen": 0.2}
```

- Inverse document frequency
    - Give less weight to terms that are common across documents
    - deals with the problems of the Zipf distribution
    - **IDF = log(|docs|/|docs containing term|)**
- TF-IDF
    - **TFIDF = TF * IDF**

Vector Space Model

- Documents are represented as vectors in term space
    - Terms are usually stems
    - Document vector values can be weighted by e.g., frequency
- Queries represented the same as documents

	nova	galaxy	heat	h'wood	film	role	diet	fur
A	10	5	3					

"Nova" occurs 10 times in text A
"Galaxy" occurs 5 times in text A
"Heat" occurs 3 times in text A
(Blank means 0 occurrences.)

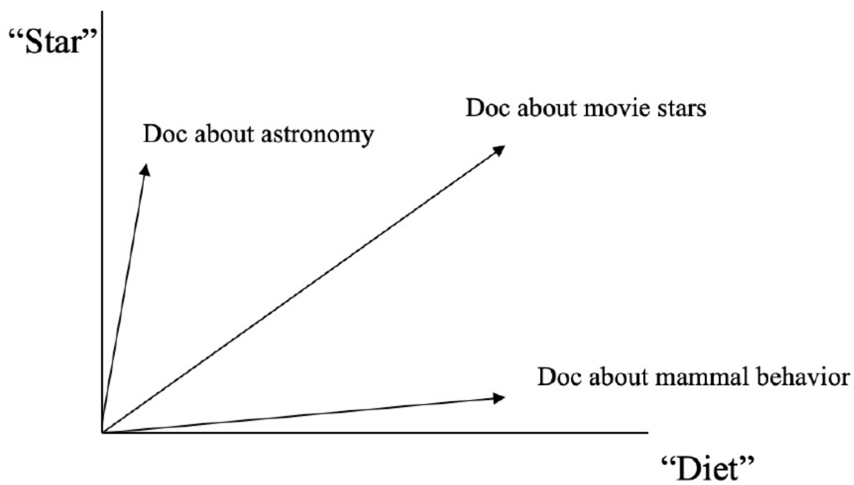
These numbers all represents Term Frequency (TF)

Document Vectors

- All document vectors together: Document-Term-Matrix (Feature-Matrix)

Document ids	nova	galaxy	heat	h' wood	film	role	diet	fur
A	10	5	3					
B	5	10						
C				10	8	7		
D				9	10	5		
E							10	10
F							9	10
G	5		7			9		
H		6	10	2	8			
I				7	5		1	3

We can plot the vectors



Assumption: Documents that are close in direction and length are similar to one another

Feature Extraction in Python

- Scikit-learn library provides corresponding functionality via its **CountVectorizer**

- Example:

```
from sklearn.feature_extraction.text import CountVectorizer
from pprint import pprint
corpus = ['This is the first document.',
          'This is the second second document.',
          'And the third one.',
          'Is this the first document?', ...]
vectorizer = CountVectorizer()
matrix = vectorizer.fit_transform(corpus)
pprint(matrix)
```

- **CountVectorizer** can be configured in quite some detail
  - By default, CountVectorizer does tokenisation for single words of minimum length 2
    - Change to also consider bigrams (terms consisting of 2 words):

    ```
    vectorizer = CountVectorizer(ngram_range(1, 2))
    ```

  - Convert input text to lower case; also ignore certain accents in text:

  ```
  vectorizer = CountVectorizer(lowercase = True, strip_accent = 'ascii')
  ```

  - Use indicator feature (0 or 1) rather than frequencies

  ```
  vectorizer = CountVectorizer(binary = True)
  ```

  - Specify a list of stop words that get ignored

  ```
  vectorizer = CountVectorizer(stop_words = ['the', 'a'])
  ```

  - Only keep features within a certain document frequency range

```
vectorizer = CountVectorizer(min_df = 0.1, max_dif = 0.5)
```

- Example: `CountVectorizer(lowercase = True, strip_accents = ‘ascii’, binary = True)`

Text Classification - Part 2

Using scikit-learn Pipeline to manage Cross Validation

```
from sklearn.model_selection import GridSearchCV
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.metrics import classification_report
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
```

Scikit-learn Pipelines provide a mechanism for fitting and predicting a sequence of components. This is good practice to avoid **data leakage**

```
# Pipeline for multinomial naive bayes
mb = Pipeline([('vect', CountVectorizer(lowercase = False)),
               ('tfidf', TfidfTransformer()),
               ('clf', MultinomialNB())
              ])
```

1. Convert string to a bag-of-words token vector.
2. Transform vector counts using TFIDF weighting.
3. Train/predict using multinomial naive Bayes

Data Leakage

What it is:

- Allowing your algorithm to use information that will not be available in production
- E.g., using a market index to predict individual stock performance

What to do:

- Understood your problem and data
- Don't trust nonsensical model components (e.g., index)
- If a result seems too good to be true, it probably is!

Using scikit learn for GridSearch-CrossValidation

```
from sklearn.model_selection import GridSearchCV
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.metric import classification_report
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
```

```
# Define grid search parameters
param_grid = [{'vector_binary': [True],
               'vected_ngram_range': [(1, 1), (1, 2)],
               'tfidf_use_idf': [True, False]
              },
              {'vect_binary': [False],
               'vect_ngram_range': [(1, 1), (1, 2)],
               'tfidf_use_idf': [True, False]
              }
            ]

# Find best parameters for MNB and SVM
gs_mnb = GridSearchCV(mnb, param_grid, cv = 3)
gs_mnb.fit(X_train, y_train)
print('\nMNB best params:\n', gs_mnb.best_params_)

gs_svm = GridSearchCV(svm, param_grid, cv = 3)
gs_svm.fit(X_train, y_train)
print('\nSVM best params:\n', gs_svm.best_params_)
```

Text-driven Forecasting

- Predict box office gross for films

- T: description, script, review, etc
- M: how much the film earns at the box office

- Predict volatility of a stock
  - T: annual report, etc
  - M: volatility over the following year
- Predict blog reader behaviour
  - T: political blog posts, etc
  - M: number of reader comments

Information Extraction

Word	POS tag
This	DT (determiner)
is	VBZ (verb)
a	DT (determiner)
tagged	JJ (adjective)
sentence	NN (noun)
.	.

- Structured prediction: problems where output is a structured object, rather than discrete or real values
- E.g., sequence tagging for part-of-speech (POS) tagging or named entity recognition

Natural Language Processing

- Understanding
  - Tokenisation
  - POS tagging

- Parsing

- Generation
- Summarisation

Parsing etc in Python

- spaCy is an open-source software library for advanced Natural Language Processing, written in the programming languages Python and Cython