Text Mining

Abstract

(Lessions will be in English.) Lessons are divided between CS and DS + TTC students; the aim is to study and build search engines and recommendation systems. The first part is a presentations of Text Mining (an AI branch) with techniques of *Information Retrieval*, *Information Filtering*, *Text Classification* and *Summarization*, all using *open source* software. DS and TTC students will have labs using R and Python, CS students will use Java (Lucerne). The exam consists in a written test plus a project (done in groups of a maximum of 3 people), that can be followed by an oral test at will. On e-learning suggested books will be published.

Contents

Part I
Introduction

Text Mining can be supported by Machine Learning but its techniques are more general and is built to emulate human behavior: AI is born in early '900 and Machine Learning is only a small branch of it. An Agent is considered *intelligent* if can understand the semantics of its Ambient: in this case texts written by humans in a *natural language*; to understand a text, the Agent must have a previous knowledge of the matter.

In 2004, Ian Witten (Weka project leader) published a paper titled "Text Mining" in which he reports that the first workshop was started in summer 1999: texts were processed in an automatic way to extract useful information.

Text Mining generally denotes any system that analyzes large quantity of natural language texts and detects lexical or linguistic usage patterns in an attempt to extract probably useful information.

Examples of Text Mining are *Sentiment* and *Opinion Analysis*; another example is *Text Summarization* (the ability to write a *snippet*, a short description of a text). *Recommender Systems* are largely used by web-shopping platforms: they have revolutionized shopping and adverts suggesting useful information to users (positive filtering) or avoiding contents (negative filtering), analyzing text. Users are profiled by *Avatar*, their digital representations.

Text Analytics is used in healthcare to reduce the number of *fake news* on the web. *Knowledge Graphs* are used to go behind traditional Text Mining techniques and have a better understanding of the text and improving the model.

The particularity of Text Mining is that the information in the text is not hidden but well written, and humans can grasp it with no difficulty (but for text length); Text Mining is largely used to analyze in a semi-automatic way lot of unstructured documents for decision making.

There are a lot of challenges in Text Mining:

- Text annotation: documents are not in an accessible form for the computers;
- Dealing with large *corpora* and *streams*;
- Organizing semi- and un-structured data;
- Dealing with ambiguities on many levels (lexical, syntactic, semantic and pragmatic).

Information Retrieval aims to find things on the Web, and has its roots in 70s. Techniques have to balance precision and efficiency in order to extract documents of interest from a huge pile of data.

Text Mining needs big data: good performance are reached only with a huge quantity of data due to algorithms complexity, most of these algorithms are topic and language independent. For specific task contest-based, specific algorithms may have better performance or are less data-expensive.

Text analysis does not use databases but *corpora* (or *collections*): a group of unstructured documents; but sometimes information is well structured in the text (names or dates).

A *predictive analysis* of a text can recognize or detect a concept within a span of text, generally with supervised algorithms; examples are:

- opinion mining (positive or negative);
- sentiment analysis (usually from a predefined emotions ex. fear, hope, desperation...);
- bias detection (also here the view points are predefined ex. pro/against);
- information extraction (is a person, is a place...);
- relation learning (is CEO of, is parent of...);
- text-driven forecasting (predicting events from twitter);
- temporal summarization (monitoring news or opinions about an event).

On the other hand, an *exploratory analysis* automatically discover patterns and trends of interest, generally done with an unsupervised approach such as Text Clustering and Topic Modeling.

Part II
Natural Language Processing

To analyze a text, it have to be written so that a computer can easily read it: it have to be processed by a process that starts with *tokenization* (splitting text into units) and ends with the lexical analysis. Documents can be written in various ways, like free-format or in a semi-structured format (with descriptive and semantic metadata).

The first step consists in identifying input files and understand how to read them (creating a readable representation). The most used representation of text is the so called *bag of words*: introduced in 60s, each document is simply represented by a bag of words. Tokens are the basic features of the dataset, represented by the presence or absence (0 if absent, 1 if present, even if more times) in the document. The document is than processed to make it machine-readable. Another similar approach is to sign absolute frequency of words in the document. Both approaches do not consider words order, so "John is quicker than Mary" and "Mary is quicker than John" are represented in the same way.

Tokenization In this phase of the process, the document is divided in small units, each one representing a concept. Each unit (or *token*) is a sequence of characters with a semantic meaning (e.g. *"tree"*, *"San Francesco"* or *"01/01/1970"*). A token does not always correspond to a single word. The task is difficult due to language-specific syntax and conventions (numbers, dates), but ad-hoc techniques achive in general better performances compared to more general ones. For example, in German word-splitting can boost performance in information retrieval tasks. Arabic language is written right to left but have numbers written from left to right. Other oriental languages do not include spaces between words, so the tokenization process can return different solutions depending on the algorithm. Tokenization process depends also on corpus and context (a social-media corpus contains a lot of abbreviations for example).
The process is made with a *parser* based on specific rules (defined with regular expressions) or statistical methods (machine learning or conditional probability).

Normalization It maps the text to query terms to the same form: sequence of characters with the same meaning have to be written in the same way (e.g. "USA $\rightarrow$ U.S.A."). The same concept is represented only in a single form: it is finalized to reduce tokens referred to the same concept. Uppercase and lowercase letters may be problematic due to semantic differences and language specific grammar. The accents must be considered for the normalization process too since a lot of users don't type them while writing queries for the search engines. Thesaurus (or approximations like *WordNet*) that represent synonymy are largely used during the process. Spelling mistakes are managed using heuristics like phonetic pronunciation, an example of such normalization is Soundex.

Stemming and Lemmatization In the stemming process words are "cut" at the root which may be common with other words. In the lemmatization process instead words are transformed to their lemma. The second approach is more elegant but harder to implement (it is extremely language-depending and it requires a thesaurus), in some cases benefits are not

high enough to justify its use. Stemming on the other hand is faster and easier to implement, but it is based on the hypotheses that words with the same root have similar meanings, that is not always true (e.g. "author" and "authorize"); the task removes language specific suffixes, so algorithms are language depending. One of the most used stemming algorithm for English is Porter's algorithm, that uses simple morphological rules. Stemming and lemmatization do not always boost performance in English due to its syntax, but for other languages (Finnish) it works very well.

Stop words management Stop words are very frequent words with no semantic meaning; they are generally removed from the text due to little importance in the analysis, but in some cases the text assumes a completely different meaning. Stop words are generally defined in a list (a *stop-list*, that can be taken from internet and modified to adapt it for the specific task) but may also include common and meaningless words. Search engines do include stop-words in the algorithm due to the risk of misunderstanding (but are managed in a special way).

Fundamental parts of the process are just tokenization and normalization: stemming, lemmatization and stop words management are context-depending. The algorithm is evaluated with two values: *precision* (the number of related documents depending on the query) and *recall* (the number of documents that match the query on the total of documents about the argument). The second one is fundamental in patents finding, the first one is important considering using-experience in web search engine (related documents must be presented as first results).

A different approach to the bag-of-words is the N-gram construction: N consecutive words are concatenated to catch linguistic expressions. This approach is more resources-expensive but it can boost performance a lot. Than a vocabulary is used to extract meaningful sequence of words (instances) or to exclude stop words from the removing process (the word can be a stop word if used alone, but in that context it assumes a particular meaning). Data is not stored on a matrix (due to the amount of memory required) but is stored using dynamic data structures (such as dictionaries and lists). N-grams tries to consider word order, storing more information than Bag of Words but increases the vocabulary size.

Syntax is managed by two techniques: *Part of Speech Tagging* (POS) and *Entities Recognition*. The aim is to store identify and store positional information analyzing not N-grams but other data structures like pairs `<word, position>`.

Part of Speech Tagging The aim of this procedure is to assign to each word its syntax role in the phrase. POS Tagging helps making the Entities Recognition process easier, and can be managed in different ways. In addition, it can used to define rules in Parsing, word generation/prediction (e.g. "a possessive adjective is followed by a name"), and machine translation.

To have a perfect precision, a *lexicon* is required but it is not enough: it cannot manage ambiguity, that have to be solved following rules or use machine learning or statistical models such as hidden Markov models. An example of such ambiguity is:
Time[V/N] Flies[V/N] like[V/Prep] an[Det] arrow[N].

Statistic methodologies are based on conditional probability (from a given *corpus*): probability of a category depending on the previous one is maximized to assign the output for the word. Rule based is in general more accurate but more time consuming because it often requires manual intervention. The process is built starting with short sentences to identify word categories and then longer sentences to identify rules. POS tagging does not scale well with the data, it becomes slower for large collections, a better choice is to use the N-grams. They form a Zipf's distribution, but their downfall is that they uses a lot of space.

Entities Recognition The idea is to extract entities from text: it is a Information Extraction task. In this way the text representation is enriched with more information about its meaning. To complete the task, two approaches are identified: rule-based and statistics-based. According to the first approach, rules are used to identify if a (sequence of) word(s) can be an entity name, the rules can use lexicons or build manually by trial and error or use Machine Learning; the second approach instead tries to maximize the probability with a statistical model such as an hidden Markov model. HMM can resolve ambiguity in a word using context, which is developed using a generative model of the sequence of words (a small number of previous words).

Usually, a message is sent by the sender and received by the recipient, who *infers morphological* information of the words, uses lexical rules (*syntax*) to reconstruct the original message and compare it with its knowledge to understand its *semantic* (what is explicit, and does not change with the context) and *pragmatic* (what is implicit or context related). In the same way, a program have to rebuild the message and compare it whit a base of knowledge (usually modeled as a graph) to understand its content.

Anaphoras and cataphoras are constructions complex that need to be managed correctly: to manage them, a *parser* is used to build a tree of the sentence (according on a formal representation of the grammar).

Relation Extraction It identifies the relationships among named entities, this can be used to convert chunks of text into a more formal representation using first order logic structures.

1 Zipf's Law

Originally used in Economics and than applied to Computational Linguistics, Zipf's Law (1949) describes the frequency of an event in a set. In particular, sorting words by frequencies in decreasing order, the product of use frequency and the rank order is approximately constant:

$$f(w) \propto \frac{1}{r(w)} = \frac{K}{r(w)}$$

where K is the corpus constant and w is an event. Different collections have different constant K. So words can be divided in *head words* and *tail words* depending on the rank $r(w)$: head words are more frequent but in general are semantically meaningless.

According to this Luhn's Analysis, words do not describe document content with the same accuracy: word importance depends not only on frequency but also on position. So frequencies have to be weighted according to their position: two cut-offs are experimentally

signed to discriminate very common or very rare words. Most significant words are those between the two cut-offs, those that are neither common nor rare. It automatically generates a document specific stop list, the most frequent words.

2 Text Representation

Weights are assigned according to two factors: word importance in the document and document importance in the collection. These weights can be measured using two basic heuristics:

Term Frequency $tf_{t,d}$ It represent the frequency of the term t in the document d, often normalized by document length:

$$w_{t,d} = \frac{tf_{t,d}}{|d|}$$

To prevent bias towards longer documents, it doesn't consider the document length but the frequency of the most occurring word in the document:

$$w_{t,d} = \frac{tf_{t,d}}{\max\limits_{t_i \in d}\{tf_{t_i,d}\}}$$

Inverse Document Frequency idf_t It represent the inverse of the informativeness of the document for a term t:

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

where df_t is the number of documents that contains t, an inverse measure of informativeness of t, and N is the total number of documents.

The Weights, called tf-idf weights, are the product of the two indices:

$$w_{t,d} = tf.idf(t,d) = \frac{tf_{t,d}}{\max\limits_{t_i \in d}\{tf_{t_i,d}\}} \cdot \log\left(\frac{N}{df_t}\right)$$

The weight w increases with the number of occurrences within a document and with the rarity of the term in the collection; this procedure is the best-known to calculate wights from 1983 (G. Salton et al.).

3 Statistical Language Models

A topic model is a unsupervised technique, very similar to a clustering. The topic model generates a probability distribution of the words in topics. A language model is a formal representation of a specific language, it can be even used to profile a user based on his own writing-style, which can be used as a recommender system.
A probability to a sequence of words can be used for:

- Machine Translation (ex. $P(\textit{High}$ winds tonight$) > P(\textit{Large}$ winds tonight))
- Spelling Correction

- Speech Recognition
- Summarization

With a train sample (of the magnitude of Big Data), a probability distribution P can be extimed: beginning from a naive method with N training sentence $w_1...w_n$ is computed the probability for each sentence $P(w_i)$ as its relative frequency. This approach altough does not works well due to number of possible sentences.. Another task is the prediction of an upcoming word: $P(w_t|w_1, ..., w_{t-1})$. Any model which computes any of the precedent is called a language model.
The joint probability is computed relying on the chain rule of probability:

$$P(w_1, ..., w_n) = P(w_1)P(w_2|w_1) \cdot ... \cdot P(w_n|w_1, ..., w_{n-1})$$

A problem is that so many product could lead the numbers to decay pretty fast, so a solution is to consider the probability of one or two words preceding the word of which we are calculating the probability so we have the following bigram approximation:

$$P(w_i|w_1, ..., w_{i-1}) \approx \prod_i P(w_i|w_{i-1})$$

or we could use the unigram model (the simplest):

$$P(w_1, ..., w_n) \approx \prod_i P(w_i)$$

The most used in IR(Information Retrieval) is the unigram model.
The estimation of the probability of the words is done in log space, in order to avoid underflow, plus adding is faster than multypling:

$$log(p_1 \times ... \times p_n) = \sum_i log(p_i)$$

4 Topic Model

It is an unsupervised algorithm that clusterize documents according to their content. A topic is formal representation of the language used to write documents of a particular model, with a probability distribution for each token. In this way, it is easy to check which is the probability that a new document belongs to a topic. The same idea can be used using people own style of writing instead of topic, and so on. This algorithm supports recommending systems that suggest contents to the user depending on how they write; it can also be used to generate new contents in automatic way or in machine translation.

5 Word Embeddings

For *word embedding* is indicated a set of NLP techniques to map a large namespace in a smaller one: in particular to transform concept space in a lower dimensional space in which it is possible to catch semantics similarity between words. Models to generate this mapping can be based on count of words or predictive models. This model can be used to check

similarity between either documents or words, and to check topic similarity and words usage per topic.

To represent words as vectors, a $V \times V$ *1-hot* matrix is built in which each word has all values equal to 0 except for the corresponding column (that is equal to 1). This is a very sparse matrix, with no shared information between similar words. To reduce number of dimensions, not all words of the set V are used as columns: a small subset that includes just representative words is used as column. In this case, axis (columns) are putted manually and have a semantic meaning, but a previous knowledge of the domain is required. It is possible to catch relationships between words or to measure similarity between words using a distance index.

In a real scenario, it is not possible to catch all significant words in the domain (due to complexity of corpora), so automating algorithms are used. Results changes depending on the training corpus and its domain; often pre-trained (on general corpora such as dbpedia) models are retrained with domain-specific corpus to have a better representation of words. The model is trained trying to represent a word knowing its context (all other words in a defined *window* that set the size). So weights are taken counting how often a word co-occurs with another one. The number $f_{i.j}$ is the number of times that the word w_i appears in the context c_j: it is used to compute $P(w_i)$:

$$P(w_i) = \frac{f_{i.j}}{c_j}$$

The raw frequency is not a good representation: more used weights are frequency, tf-idf or PMI (*Pointwise Mutual Information*).

$$PMI(x, y) = \log_2 \left(\frac{P(x, y)}{P(x) \cdot P(y)} \right) \quad \in (-\infty; +\infty)$$

where x is the target word and y the context. To have better estimations, only positive estimation are taken:

$$PPMI(x, y) = \max(PMI(x, y),\ 0)$$

Rare words and *apax* produce higher scores, that risks to bias the analysis: probabilities of w and c are modified or a k-smoothing is used to reduce those scores.

$$P_\alpha(c) = \frac{count(c)^\alpha}{\sum_c count(c)^\alpha}$$

where (usually) $P_\alpha(c) > P(c)$ for rare c and $P_\alpha(c) < P(c)$ for common c. The second solution is the add-2 smoothing: a frequency of 2 is simply add to all elements of the matrix: effects are bigger with less frequent words, but high frequency words are little affected. From those high-dimensional matrices, dense lower-dimensional (usually around 300) vectors are learn as embeddings.

There are two mainly approaches to build an embedding: *count based* or *predictive models*. The first one uses statistic indices (based on frequencies like tf-idf) to map words in the new space; the second one tries to predict a word knowing its contest building an *auto-encoder* with a neural network.

5.1 Count based

This approach counts how many times two words (a word w and its neighbor c) co-occur in a large corpus; then the vector is mapped in a dense lower-dimensional space. Algorithms to reduce the number of dimensions are LDA (*Latent Dirichlet Allocation*), SVD (*Single Value Decomposition*) or GloVe (*Global Vectors for word representation*).

SVD is not really used due to its complexity (eigen values and vectors have to be computed for a very large matrix). From a mathematical point of view, the matrix X is decomposed in three matrices U (the matrix containing eigen vectors of XX'), S (a diagonal matrix containing eigen values of $X'X$) and V (the inverse of U) so that $X = USV'$. Eigen values σ_i (the element $S_{i.i}$) represent the variability of X captured by that eigen vector: removing them reduces the dimensions of the matrix. So, reducing dimensions of matrices to k (the selected eigen value that captures the desired percentage of variability) reduces dimensions of the matrix X minimizing information loss. In this way, words are mapped in a new space in which closer words are similar, but in which the original co-occurrence of words is lost (so it becomes a *black-box*). This algorithm is not really used due to its poor performance (decomposing matrices can be difficult, being $O(n^2)$ complex). Usually this algorithm is used only without stop-words and considering distance between words (using a *ramp window* to calculate proximity between words) or Pearson Correlation. Glove is used instead of SVD to solve its problems: it is trained only on non-zero elements in the matrix of the whole corpus. Its notation is similar to the previous one:

$$
\begin{aligned}
X &\quad \text{word-word co-occurrence matrix} \\
X_{i.j} &\quad \text{the frequency of } w_j \text{ occurring in the context } c_i \\
X_i = \sum_j X_{i.j} &\quad \text{frequency of a singular word} \\
P(j|i) = \frac{X_{i.j}}{X_i} &\quad \text{the probability of a word knowing its context}
\end{aligned}
$$

The ratio $\frac{P_{i.k}}{P_{j.k}}$ is larger then 1 if w_i is closer to w_k than w_j, lower otherwise and equal to 1 if the two words are equally distant from w_k. This ratio is used to modify the co-occurrence matrix: the quantity $\frac{P_{i.k}}{P_{j.k}}$ becomes a function $F(w_i, w_j, w_k)$ in which all variables w_i and w_j are word vectors and w_k the context. The problem is to find a valid F, but it can be simplified by

$$F(w_i, w_j, w_k) = \frac{P_{i.k}}{P_{j.k}} = w_i' w_k + b_i + b_k = \log(X_{i.k})$$

In this way however weights b are equally computed for both rare and frequent co-occurrences: the learning of vectors becomes a Least Squares Regression problem (with a weighting function):

$$
\begin{aligned}
J(\theta) &= \sum_{i,k=1}^{V} f(X_{i.k})(w_i' w_k + b_i + b_k - \log(X_{i.k}))^2 \\
&= \frac{1}{2} \sum_{i,k=1}^{V} f(X_{i.k})(w_i' w_k - \log(X_{i.k}))^2
\end{aligned}
$$

the weighting function f most common is:

$$f(x) = \begin{cases} \left(\frac{x}{x_{max}}\right)^{\alpha} & x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

but it can be any function that:

- $f(0) = 0$ and $\lim_{x \to 0} f(x) \log_2 x$ is finite;
- should be non-decreasing;
- should be small for big values of x.

This approach is fast to train, scalable and yields good performance.

5.2 Predictive models

Those models extract a subset of the corpus as train set, used to build an auto-encoder to map words in a new space with a Neural Network. The most popular algorithm belonging to this family is Word2Vec, introduced in 2013 to solve problems of SVD. It operate on a sample of the raw text, used to train an auto-encoder. Train data grows with the size of the window used to train the algorithm. The algorithm is used to predict the context knowing the original word (minimizing $J(\theta) = 1 - P(c|w)$ that maximize the probability of the prediction). Literature offers a better approach: the algorithm is used first to predict the context c (up to a certain level) giving a word w and then to predict the original word w knowing its context. Objective function (simplified) is the following:

$$\begin{aligned} J'(\theta) &= -\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P(w_{t+j}|w_t) \\ &= -\frac{1}{T} \sum_{t=1} T \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j}|w_t) \end{aligned}$$

and then the prediction is made with a *naïve softmax*:

$$P(c|w) = \frac{\exp(u_c' v_w)}{\sum_{v=1}^{V} \exp(u_v' v_w)}$$

It captures more complex patterns than other approaches and can be used to other tasks; but on the other hand it does not scale well with corpus size and its very inefficient because it is trained only on a sample of the corpus. This algorithm is very context depending (the training is made on a corpus), and does not consider the changing of meaning over time.

Part III
Information Retrieval

Information Retrieval is an old problem in computer science: its main purpose is to find a documents on the Web. The main problem is the fact that documents are distributed between servers and not stored on the same computer; but also the quantity of data (and how to reach as many pages as possible) that are constantly updating can be very difficult to manage. Documents are managed in different ways, according to various criteria. Information Retrieval is done by *search engines* (that can search on the web - *web search engine* - or not).

Information Retrieval is a decision problem: how to identify and define the importance of information that satisfies the user. It is important to understand user's needs, to interpret the context and to yield documents that answer the query, according to its relevance. First search engines considered only topic relevance: it was not important to interpret a user query but just to yield relevance documents. Now the idea is to give a better user experience: the search process considers other variables such as geographical location and previous query done by the user; it can also be done without a user query (*Information Filtering Systems*) just filtering a stream of documents in real time.

In a search engine the query is usually not written in a formal language (but in a natural language): conditions have to be extrapolated from the text to reduce number of results. Information can be extrapolated from supports of different kinds: there are a lot of technical (due to different formats) and semantic (information is synthesized and represented preserving its meaning) problems.

A search engine is composed by two aspects (off-line and on-line): in the first part (offline) documents have to be indexed (and updated regularly by sub-parts of the archive) and represented in a formal way so that query can be done; than in the online part, the query is parsed to represent it in a logical language and than the match is made, according to classic set theory (binary matching) or using a similarity index on a vector space (cosine similarity, proximity and so on, that permit a fuzzy match).

1 Indexing

Documents have to be indexed to retreive them easily: this process extracts a small portion of text that will be used to search a document. This process is extremely resource expensive, so are implemented in a particular way to optimize the process. Each document is characterized by a different set of terms weighted in a different way: documents are indexed to reduce the dimensionality of the matrix so that the number of terms will be lower than the number of documents. The resulting matrix is inverted so that documents are on columns and terms on rows: retrieving a file is just as easy as checking in which document the term is present. The data structure used in practise is a dictionary in which the key is the term, the value is a list of tuples `<document; weight>` so that looking for a document will not scan useless documents.

There are cases in which index can not be stored on RAM due to amount of documents: in this case the use of a column-storage database system is required. In fact this strategy garantees both efficent management of a sparse matrix and scalability on a cluster of

machines.

2 Matching

The query the user ask to the search engine is represented in a formal way to compare it with the (indexed) database. To speed up the process, the database is inverted (inverted-index) so that documents are indexed basing on terms, ordered by relevance (if available) or by progressive number (if the scoring is abstent or done *on-the-fly*).

2.1 Boolean matching

Boolean matching is based on set theory:

$$R(d_j) := \{t_i | w_{ij} = 1\} \qquad w_{ij} \in \{0, 1\}$$

the document is a set of terms, so relevance is modeled as a binary property (a document is either relevant or not, ranking is not permitted). A query is formally represented by boolean operations like `AND`, `OR` or `NOT`. Using the inverted representation the matching process returns a sub-set of documents that satisfy user's query. Evaluations order of operations must be clearly specified because a different order can produce different results. In general operations are a priority order pre-imposed in the representation process.

The evaluation is made using a recursive algorithm on a binary tree (the formal representation of the query) starting from leaves, with some optimizations that evaluate longer list for lasts (to save memory).

This matching model is fast and simple, but it is not able to produce a ranking of results: in some cases it is not sufficient for accomplish the task.

2.2 Vector Space matching

To solve problems of the Boolean model, the Vector Space model is presented: documents are presented as vectors (defined by a word-embedding algorithm) in a space with a number of dimensions equal to vocabulary size $|V|$. Documents are represented using some indices (like presence-absence of the term, (log or normalized) frequency, tf, tf-idf value...) as vectors (points) in this space. In general, are used:

$$w_{t,d} = \frac{tf_{t,d}}{\max\limits_{t_i \in d} tf_{t_i,d}}$$

$$w_{t,d} = \frac{tf_{t,d}}{|d|}$$

$$w_{t,d} = \begin{cases} 1 + \log(tf_{t,d}) & tf_{t,d} > 0 \\ 0 & otherwise \end{cases}$$

Also the query is presented as a vector, so that the matching problem is just as easy as compute similarity between vectors or using a KDTree to cache documents.

Part IV
Web Search

As its name says, the Web is structured as a graph, dynamic and oriented. Edges reachable in both direction ($A \to B$ e $B \to A$) are told *Strong connected spaces*: the whole structure is often abstracted as a set of Strong Connected Spaces instead of pages. In this way the graph assumes a *papillon* structure: there is a *source component* (from which it is easy to reach other pages but difficult to access), a *huge component* (that represent the majority of documents) and a *shaft component* (easily reachable but from which it is difficult to exit); there are also some noisy documents isolated or random linked. To retrieve a document, a web search engine must had indexed it: documents not retrievable by search engines belong to *deep web*. Those documents represent the majority of the web and include dynamic web pages (which content changes according to a input), private pages (hidden by CAPTCHAs or login) or just because there are no incoming links.

Web search can be done by *browsing* (using links between pages), direct link (if the address is known) or using a *web search engine*. First generation of *web search engines* considered only documents word frequency, but then also links are considered to rank documents by authority. *Semantic web* considers also the meaning of the documents to improove user experience.

1 Updating documents: *gathering*

Gathering is the process of index updating by a search engine: due to mutability of documents in the web, index have to be constantly updated to include new documents or to update existing ones. This process can be automatically done asking the search engine to update the document manually, but is ofen done in an automatic way using *crawlers*: bots that surf the net following links and storing documents. A *crawler* starts its search process with a *set seed* of reliable pages (such as public organization) taken randomly; then it checks the presence of unseen links in the text, stores them and opens one (per thread) so that a server is not filled of requests at the same time (latency is given by *politeness rules* in the `robot.txt` file). Then the document is processed and usfull information (such as url, title and content) are stored in a column-based database. The process goes on until there are no more links to visit or the disk is full.

During the process, also *meta-data* are stored, according to *politiness rules* (written on a sitemap file called `robots.txt`): those meta-data include importance, update log and frequency for each page, latency of requests and so on. Documents are then cleaned to parse text: it is considered

A document is considered *fresh* if its last version is indexed; otherwise it is *stale*: *freshness* is the percentage of fresh documents stored in the index. Freshness altough is maximized if no document is stored at all: to prevent this inconvenient, another metric is *Age*, the time passed from the updating of a document to the updating of the index.

2 Ranking

Pages are retrieved by a score calculated on relevance to the query, popularity and position in the Web structure. The main idea is that a good document (a document with high *authority*) is linked to many other good documents (*hub score*): analyzing the Web graph it is possible to estimate a document *authority* to improove the search.

2.1 *PageRank*

It is Google algorithm: the main idea is to simulate a *random walk* across the web: a user, every fixed time, goes to a random page on the web (with a very low probability of λ) or follows a link in the current document. The score given by *PageRank* is the probability that in a random moment the user is reading that page.

$$PR(p) = Kd + K(1-d)\sum \frac{PR(p_i)}{C(p_i)}$$

where K is a normalization constant, d depends on the system and $C(p)$ is the number of documents of the page p. The fact that PR is in both side of the formula means that it is computed recoursively until convergence.

2.2 HITS

This algorithm estimates both *authority* and *hub* scores for each page: the process is iterative and converges after a small number of passages (depending on graph size); during estimation only graph structure is considered. The algorithm initialize each document with both scores equal to 1 and for each iteration updates them according to formulas:

$$A(p) = \sum_{q \to p} H(q)$$
$$H(p) = \sum_{q \to p} A(q)$$

Part V
Recommender Systems

A *recommender system* is a *push* technology: it provides the user with contents without a query; it suggests other contents according to user's behaviour and previous queries. A good recommender system should not suggest common objects but unknown items that users do not know but might like. The system should consider user's feedback and not only contents that he or she had actually searched: a *relevance score* is extracted during the retrieval process. In general the systems are based on *Information Filtering*: contents are monitored producing data (virtually in real-time) that have to be filtered to inform the user.

A recommender system can follow different paradigms, and there are also hybrid approaches. Assumptions of a recommender system are that users rate and catalog items and that a person will not change tastes during time. This second assumption is too strong for a real environment: techinques were developed to update a user profile making the process dynamic (computing users embeddings in a *never-ending* learning pipeline).

1 Collaborative filtering

In this architecture the user is clusterized and contents are suggested according to the belonging cluster (a not-seen content is presented if is appreciated in the cluster). This approach is based to data provided by users: to get more data, some indicators are stored (such as web-surfing behaviour) to improve the process, although the behaviour can be misunderstood.

At beginning there are no user information: so he or she is forced to rate a set of items, demographic data are used to do some assumptions or a default set is presented.

In this paradigms, an important part is to compute the *user-item matrix*: the output is a numerical prediction that indicates if user will like a certain item or not. So a *top-N* list can be extracted. That matrix is a sparse matrix with users stored as rows and items as columns.

Given a *target user* a, an item i not yet seen by a is suggested after that procedure:

- is extracted a subset of users $U_{a,i}$ similar to a that have reted item i;
- a model (machine learning or an heuristics) is provided to compute a rating $\hat{r}_{a,i}$ for item i according to $U_{a,i}$;
- if $\hat{r}_{a,i}$ is greater a certain value, i is suggested to a.

The subset $U_{a,i}$ is extracted computing the *Pearson correlation* ϱ between ratings of a and each other user $U_{a,i} = \{b | \varrho_{a,b} \geq r \ \forall b \in \text{Users}\}$. A common predicting function for $\hat{r}_{a,i}$ is:

$$\hat{r}(a,i) = \overline{r}_a + \frac{\sum_{b \in U_{a,i}} sim(a,b) \cdot (r_{b,i} - \overline{r}_b)}{\sum_{b \in U_{a,i}} sim(a,b)}$$

This process have scalability issues: number of users is much higher than number of items $U >> I$, so computing similarity between users can be very expensive. In addiction, operating with very sparse matrices, is very difficult to have a match between users.

To respond to those problems, other approaches are used: for example, a *item-item* matrix is computed, so that the prediction $\hat{r}_{a,i}$ is computed with the formula:

$$sim(\vec{a}, \vec{b}) = \frac{\sum_{u \in U}(r_{u,a} - \overline{r}_u)(r_{u,b} - \overline{r}_u)}{\sqrt{\sum_{u \in U}(r_{u,a} - \overline{r}_u)^2} \cdot \sqrt{\sum_{u \in U}(r_{u,b} - \overline{r}_u)^2}}$$

$$\hat{r}_{a,i} = \frac{\sum_{p \in R_u} sim(\vec{p}, \vec{i}) \cdot r_{u,p}}{\sum_{p \in R_u} sim(\vec{p}, \vec{i})}$$

where R_u is the set of items rated by the user u.

A machine learning model (traied on a subset of users) can be used to predict future ratings; this approach requires regular updating of the model.

2 Content-based filtering

The algorithm considers only features of all the items and the user profile (with contextual parameters): a similarity measure is than used to match the profile with items. This approach is not based on a community, so it works even with a few users. The content can be presented in a structured or unstructured way, and is used to compute the *cosine similarity* $sim(\vec{i}, \vec{u})$ between user behaviour and item.

Another simpler approach is to use a *Nearest Neighbors* approach: given a set I_a of items rated by the user a, the k not-yet-seen nearest items $i \in (I - I_a)$ are presented to the user (according to cosine similarity). Users' rating can be also used as additional information to improve rating performance.

3 Knowledge-based filtering

This approach is similar to the previous one but products are stored in a *knowledge base* that should improve the item embedding and filtering.

4 Hybrid approaches

4.1 Monolithic

The algorithm that gives estimation of rating receives input from different sources of different nature (e.g. ratings, log of clicks and textual descriptions). So user profile is represented both as ratings and keywords.

4.2 Parallel

In this approach there are more than one recommender systems: each one returns a list of suggested items sorted by score; lists are than merged together in a new sorted list.

4.3 Pipeline

A recommender system yields a list of items that are filtered and sorted by a pipeline of other algorithm to obtain more accurate suggestions. This approach works well if algorithms used in the pipeline are different.

5 Evaluation

For a search engine, should be evaluated functionality (implementation is mathematically correct), *efficiency* (how much time between a query and the results) and *effectiveness* (how accurate and completed are results).

5.1 Effectiveness

A returned document is considered well included if is relevant, brings information about similar topics or can be used to reach other relevant documents. *Precision* and *Recall* are used to calculate effectiveness of a search engine, with a subset of documents labeled by users (non-labeled documents are considered always non-relevant).

$$P = \frac{\#\{relevant \wedge returned\}}{\#\{returned\}} \in [0, +1]$$
$$R = \frac{\#\{relevant \wedge returned\}}{\#\{relevant\}} \in [0, +1]$$

With a lot of documents, a high value of recall is not permitted (there are too much relevant document in the collection for the user), so the two measures are combined in *f-measure*

$$f_1 = 2 \cdot \frac{P + R}{P \cdot R} \in [0, +1]$$

in addiction, there are non labeled documents that do not permit computing measures (*recall* is particularly difficult due to number of documents).

To overcome those limits, different approaches have been introduced that consider relevance as fuzzy concept or consider documents in order by ranking and not as sets. $Precision_k$ and $Recall_k$ are measures that consider the measure only for the first k elements according to their relevance rank.

$$P_k = \sum_{i=1}^{k} \frac{r(i)}{k}$$
$$P_{average} = \sum_{i=1}^{\#\{relevant\}} \frac{P_i}{i}$$

Part VI
Classification and Clustering

1 Clustering

Documents are clustered using algorithms such as Doc2Vec to represent them on an embedding; the process should be based on a *semantic web* approach to embed them according to their semantics; but to improve algorithm speed a frequency approach is used.

After the embedding, algorithms are the same as numeric data: *soft* (if a document can belong to more than one cluster, even with a belonging value) and *hard* (each document belongs to one and only one cluster) clustering, *hierarchical* or *flat*.

With *k-means* (the most used hard-flat algorithm), centroids are stored to cache clusters:

$$\vec{\mu}(\omega_i) = \frac{1}{|\omega_i|} \sum_{\vec{x} \in \omega_i} \vec{x}$$

where ω is the subset of elements belonging to the same i-th cluster. A new document belongs to the closer cluster (computing distance between document and centroids).

Hierarchical clustering is less used due to computation needed: those algorithms starts with a number of clusters equal to number of documents to agglomerate them according to distance (method *botton-up*) until all documents belong to the same cluster or vice versa (*dop-down*). Splits and agglomerations can be decided in four ways: *single links* (distance between two clusters ω_i and ω_j is equal to the lower distance between elements $dist(d_i, d_j)$: $sim(\omega_i, \omega_j) = \min_{d_i \in \omega_i, d_j \in \omega_j} dist(d_i, d_j)$), *complete links* (same as previous one but with greater distance), *average links* (same, but with average distance) or *centroid links* (distance between two clusters ω_i and ω_j is equal to the distance between centroids c_i and c_j).

1.1 Common issues

In Information Retrieval, documents are often represented as vectors (Doc2Vec) to improve query speed (only documents belonging to the same cluster are considered): if clusters are *trivial* (too big or too small) the process impacts on process speed or accuracy. Cluster size should be decided according to user's behaviour: only a small part of results is checked, so big clusters are useless. In addition, number of clusters is decided by capillarity of results: using more clusters will yield more detailed results.

1.2 Evaluation

A good clustering algorithm should aggregates similar documents in the same cluster: two clusters should contain dissimilar documents. A clustering index is *silhouette* S:

$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \in [-1; +1]$$

where a is the mean *intra*-class distance and b is the distance between i and the closer other cluster.

Another evaluation index is *purity* P:

$$P(\omega_i) = \frac{1}{|\omega_i|} \max\{\}$$

Rand Index is the corresponding to *accuracy* in the unsupervised family: a *ground truth* is required. This approach considers in the same way a *False Positive* and a *False Negative*: to prevent this inconvenient, other indexes are used such as *precision*, *recall* or F_1.

Part VII
Summarization

Text summarization is the process to retrieval informations from a set of documents considering only sentences with a particoular meaning. Sumamrization is essential for humans to extract information from a huge amount of documents, but also for automatic algorithms that benefits in scalability parsing a lower number of documents. In *question answering* algorithms, this process extracts the answare from a collection.

Algorithms can be *generic* if there are no assumptions about document content, *domain specific* if it is built *ad hoc* for the domain, or *query-based* if is built interactivelly depending on a user query (as a search engine). An algorithm can simply retrieve sentence from the collection that are meaningfull; other more complex approaches generate a new text from the collection.

Simplest approach to text summarization is to consider just first sentence or paragraph for each document to create an intermediate representation of the text; then this representation is ranked according to a scoring algorithm. Sentences with higher scores are selected as a summary.

A more complex approach uses text representation to guess document topic, according to which sentences are scored. Scoring algorithms uses word frequencies to consider words that are both rare and significative.

1 Weightings

Word probability and *tf.idf* are used to create a representation of the document to the ranking algorithm. With *SumBasic system*, ranking score is equal to the sum of probabilities $P(w)$ of all words composing the sentence. There is a penality for repetition given by elevating the probability of the word by its frequency.

$$score(\text{sentence}) = \sum_{w \in \text{sentence}} P(w)^{f(w)}$$

Tf-idf is also used to cluster documents and then computing centroids, from which are removed words under a certain number: the score is the similarity between document and its centroid.

There are other approaches based on graphs: two sentences are linked toghether if they are similar: documents then are ranked according to number of links. Word semantic is ignored, but this approach can be used in multi-lingual contexts, if word spaces are aligned. *LexRank* and *TextRank* are the two most famouse algorithms using this approach.

Global summary selection is a NP-hard problem: sentences are not parsed one by one, but just one time for the whole collection.

2 Evaluation

To evaluate a summarization algorithm, human evaluation is often used, but with an ideal summary available, ROUGE measures are used:

- ROUGE-n: $\frac{p}{q}$, where p is the number of n-grams in common with ideal summary and the extracted one, divided by the number of n-grams of the reference summary;
- ROUGE-L: (*Longest common subsequence*) represent the length of the longest subsequence of text in common between the two summaries, divided by the length of the summary; it considers also semantic aspects ignored by ROUGE-n;
- ROUGE-SU: it is a hybrid approach that acts like ROUGE-L but consider also the possibility of words inside the sequence.