Text Mining and Sentiment Analysis

Luca Paoletti

June 2022

Contents

1 Sentiment Analysis and Opinion Mining

Sentiment analysis and opinion mining is the field of study that analyzes people's **opinions, sentiments, evaluations, attitudes, and emotions** from **written language**.
It is one of the most active research areas in **natural language processing (NLP)** and is also widely studied in **data mining, Web mining** and **text mining**.

Levels of Analysis
Document level: to classify whether a whole opinion document expresses a positive or negative sentiment. This level of analysis assumes that each document expresses opinions on a single entity.
Sentence level: to determine whether each sentence expressed a positive, negative or neutral opinion. This level of analysis is closely related to *subjectivity* classification in that it tries to distinguish between *objective* and *subjective* sentences.
Entity level: to directly detect opinions. An opinion consists of a sentiment (positive or negative) and a target (of opinion).

Sentiment Lexicon. One of the most straightforward ways of addressing sentiment analysis is to define a **lexicon of sentiment words** or **opinion words** (pr even small key phrases) that may be used as indicators for sentiment.
Issues with sentiment lexicon:

- Word orientation depends on the **application context** and **word sense disambiguation**
- A sentence containing sentiment words may not express any sentiment
- **Sarcasm**. 'What a **great** car! It stopped working in two days'
- Many (objective9 sentences without sentiment words can also imply opinions

2 Problems and Definitions in Sentiment Analysis

Definition (Opinion): an opinion is a quadruple, (g, s, h, t), where g is the opinion (or sentiment) **target**, s is the sentiment about the target (either polarity or score or even an emotion), h is the **opinion holder**, and t is the **time** when the opinion was expressed.

Definition (entity): an entity e is a product, service, topic, issue, person organization, or event. It is described with a pair, $e : (t, W)$, where t is a **hierarchy of parts**, sub-parts, and so on, and W is a set of **attributes** of e. Each part or sub-part also has its own set of attributes.
Example of entity: A particular model of camera is an entity. It has a set of attributes, e.g., picture quality, size, and weight, and a set of parts, e.g., lens, viewfinder, and battery. Battery also has its own set of attributes, e.g., battery life and battery weight.
A topic can be an entity too, e.g., tax increase, with its parts 'tax increases for the poor', 'tax increase for the middle class', and 'tax increase for the rich'.

Definition (opinion 2.0): an opinion is a quintuple, (e, a, s_{ea}, h, t), where e is an **entity**, a is an aspect of e, s_{ea} is the **sentiment** on aspect a of entity e, h is the **opinion holder**, and t is the **time** when the opinion is expressed by h. the sentiment s_{ea} is positive, negative or neutral or expressed with different strength / intensity levels, e.g., 1-5 starts as used by most review sits on the Web.

Objective of sentiment analysis: Given an opinion document d, discover all opinion quintuples (e, a, s_{ea}, h, t) in d.
Named Entity recognition (NER) $\Rightarrow$ Aspect Extraction $\Rightarrow$ Opinion Holder Extraction

⇒ Time Extraction ⇒ Aspect Sentiment Classification

Types of Opinion

Opinion	**Regular**	**Comparative**
Explicit	coke tastes great	coke tastes better than Pepsi
Implicit	I bought the mattress a week ago, and a valley has formed	the battery life of Nokia phones is longer than Samsung phones

Subjectivity
Definition (sentence subjective): an objective sentence presents some factual information about the world, while a subjective sentence expresses some personal feelings, views, or beliefs.

- **A subjective sentence may not express any sentiment**. For example, ' I think that he went home' is a subjective sentence, but does not express any sentiment, as well as 'I want a camera that can take good photos'.
- **Objective sentences can imply opinions or sentiments due to desirable and undesirable facts**. For example, the following two sentences which state some facts clearly imply negative sentiments (which are implicit opinions) about their respective products because the facts are undesirable: 'the earphone broke in two days' or 'I bought the mattress a week ago and a valley has formed'

Emotions
Definition (emotion): emotions are our subjective feelings and thought.

- **Rational evaluation**: such evaluations are form rational reasoning, tangible beliefs, and utilitarian attitudes. For example, the following sentences express rational evaluations: 'the voice of this phone is clear', 'this car is worth the price', 'I am happy with this car'
- **Emotional evaluation**: such evaluations are from non-tangible and emotional responses to entities which go deep into people's state of mind. For example, the following sentences express emotional evaluations: 'I love iPhone', 'I am so angry with their service people' and 'this is the best car ever built'

3 Aspect Based Sentiment Analysis

Problem definition
In AbSA, sentiment is a **subjective consciousness of human beings towards an aspect (objective existence)**. In this light: 'A sentiment is basically an opinion that a person expresses towards an aspect, entity, person, event, object, or certain target'.

TAsks in AbSA
Given an entity, e.g., a product, the tasks of AbSA are:

- identify **entity features**
- identify **opinions regarding entity features**
- Determine the **polarity of opinions**
- Rank opinions based on their **strength**

Frequency-based approaches
It has been observed that in reviews, a **limited set of words is used much more often than the rest of the vocabulary**. These frequent words (usually only single

nouns and compound nouns are considered) are likely ti be aspects.
Association rule mining: Let $I = \{i_1, \dots, i_n\}$ be a set of items (words), and D be a set of transactions (sentences). Each transaction consists of a subset of items in I. An association rule is an implication of the form $X \to Y$, where $X \subset I, Y \subset I$, and $X \cap Y = \varnothing$. The rule $X \to Y$ holds in D with **confidence** c if $c\%$ of transactions in D that support X also support Y. The rule has **support** s in D if $s\%$ of transactions in D contain $X \cup Y$.
Pre-processing. Identify **noun phrases** which are then given as input to clustering.
Product descriptions contain phrases which begin with a determiner word like 'your favorite music' and other single word noun phrases like 'comfort', which often explain an attribute of the product rather than define it. We employ two **pruning methods** to eliminate the above noun phrases. [i] discard all the noun phrases which begin with a determiner word. [ii] assume that **single word noun phrases mentioned above occur more frequently in general English than in product descriptions** . Let p and q be unigram probability distributions of input document set and a general English corpus respectively. Now we compute pointwise KL divergence score $\delta_w \forall w \in D$, where w are the unigram, which gives the **relative importance of the unigram in the inpt document set compared to the generic corpus**.

$$\delta_w(p||q) = p(w) \log \frac{p(w)}{q(w)}$$

The noun phrases obtained from the previous step are **clustered** so that **noun phrases describing tha same attribute are grouped together in the same cluster**.
We calculate N **gram overlap to measure the similarity between two noun phrases**. We consider unigram and bigram overlap for this. Bigrams are ordered pairs of words co-occuring within five words of each other. Let S_i and S_j be the sets of unigrams, bi-grams belonging to two noun phrases P_i and P_j respectively. Now we define the similarity between the two noun phrases P_i and P_j using **Dice's Coefficient similarity**.

$$\sigma(P_i, P_j) = \frac{2|S_i \cap S_j|}{|S_i| + |S_j|}$$

Assuming that each cluster has noun phrases that contain instances of same attribute, an **attribute us extracted from each cluster** (I.e., select the best n-gram for each cluster).
We define an Attribute Scoring Function AS to score each of these n-grams. We declare the n-gram with highest score is the attribute.

$$AS(x) = \frac{PKL(x)}{AHD(x)}$$

PKL: Let P be the probability distribution of a cluster and Q be the probability distribution of the rest of the clusters together

$$PKL(x) = P(x) \log \frac{P(x)}{Q(x)}$$

AHD: The average head noun distance of the n-gram x in its instances. Head Noun Distance is the distance of the n-gram x from the right most word (head noun) in the noun phrases.

$$AHD(x) = \frac{1}{N(x)} \sum_i D(x_i)$$

Semi-supervised approaches
Bootstrapping based Refinement Framework: We take opinion seed words set, dependency patterns and review data as the system input. We scan all the sentences in the dataset, and we adopt a syntactic parsing method to capture the dependency structure on each sentence, At the beginning, we generate two candidate sets of opinion words and targets by employing the pre-defined rules. then we iteratively extract opinion words and targets using predefined extraction rules and existing result set. There is a rule set containing several rules to identify the conditions for extraction. Most of the rules describe the latent relations between opinion words and targets, i.e., word co-occurence or dependency patterns.
We apply the structure of SGM to measure the relations between opinion words and targets, and quantize the relations by computing the weight on each edge on the graph. After the extraction, we employ several refinement methods to prune false results. In the refinement process, we check the conditions of the rules to prune false opinion words and targets in the candidate sets $\{OC\}$ and $\{TC\}$. After pruning and refining, the remained extracted opinion words and targets are added to the refined result set $\{O\}$ and $\{T\}$, and pruned words generate a false result set $\{O_{false}\}$ and $\{T_{false}\}$. Then we apply these refined result set $\{O\}$ and $\{T\}$ to update SGM by adjusting the model parameters. We also take the rule refinement by the false result set $\{O_{false}\}$ and $\{T_{false}\}$ to update or remove rules of extraction. The refined opinion words and targets $\{O\}$ and $\{T\}$, updated SGM, as well as the refined rules of extraction are all applied for further opinion words and targets extraction. Repeat the propagation of extraction until no new opinion words and targets are identified.
Sentiment Graph Model: is a weighted, directed graph. Opinion words, opinion targets and dependency patterns are represented as vertices in the graph model.
First we need to generate two candidate of opinion words and targets. The we connect pairs of co-occurrence candidates in these sets. As dependency patterns are useful to identify relations between opinions and targets, we add them as vertices to the SGM. Each dependency represents a syntactic relation between opinion words and targets. Though it is difficult to construct a comprehensive set of dependency relations between targets and opinions to cover all real-world cases, we discover potential dependency patterns and measure its confidence to discover new opinion words and targets. New edges that connect the patterns and opinion words or targets would be also added to the graph.

Ontology based (wordnet: another option to search for entity aspects is to exploit in a knowledge base, such as WordNet.

4 Word Embedding

Representing **words as vectors** in the multidimensional **space defined by the other words** is a very effective way of embedding words in a vector space representing the word meaning. Moreover, it makes it possible to:

- compute distances and similarity between words
- represent documents as regions of the feature (i.e., words) space
- providing a rich input training advanced supervised models

Naive embeddings
A natural but naive way of embedding a set W_n of words is to create an embedding matrix $E \in \mathbb{R}^{n \times n}$ where each entry e_{ij} represents a relation between words w_i and w_j in a corpus.
Wrod relations may be

- **co-occurrence**: is the number of times w_i appear within t words from w_j in documents

- **pmi**: the pointwise mutual information associated with the pair
- **context**: w_i and w_j appear in the same context according to a context model such as *skip-gram* or continuous *bag-of-words*

However, this kind of embedding has two main limitations: i) *very large number of dimensions* and ii) *sparsity.*

Dense embeddings
In order to deal with the issues of dimensionality and sparsity, we aim to obtaining **dense word vectors**. This can be done by **matrix factorization** or **machine learning**. The goal of reducing the dimensionality is not only related to the cost of processing high dimensional and sparse data, but also to minimize the impact of zero and outliers.

Linguistic and Philosophical Issues
What is the semantics, the *meaning* of a word?
A common sense hypothesis is to say that the meaning of a word is the real object that the word represents. In this framework, words that are not known to be mapped on real objects (e.g., new words for a reader or just random strings of characters like *xvul*) have no meaning at all.
However, this hypothesis is quite useless in a digital context, where we just have words, not real object.

An alternative approach if the **Distributional Hypothesis** about language and words meaning that states that *words that occur in the same contexts tend to have similar meanings.* In other words, you shall know a word by the company it keeps.

According to this hypothesis, any random word may have a meaning that we can infer from the other words in the context is appears. **Distributional semantics** is the research interested in quantifying semantic similarities between linguistic items according to their distributional properties in large text corpora.
One of the main advantages for us is that this way the *meaning of words* is quantifiable and measurable in terms of distances from the other words in a corpus.

Weight contextual relations
Given a word w_j in the context of w_i, we need to **quantify** the relation $(w_i, w_j 9$ and define the score $[i_{ij}]$ in the embedding matrix.
We can do this by just counting the occurrences of (w_i, w_j) (i.e., how many times w_i appears in the context of w_j) or taking the normalized count:

$$[e_{ij}] = \frac{count(w_i, w_j)}{\sum_{w'_i, w'_j \in D} count(w'_i, w'_j)}$$

The drawback of the solution is to overestimate frequent words. So pairs like 'the apple' will have higher score than 'red apple', although the last one is more informative.

Pointwise Mutual Information
A different option is to evaluate the relation between the words joint probability and their marginal probability through the **Pointwise Mutual Information (PMI)**

$$[e_{ij}] = PMI(w_i, w_j) = \log \frac{P(w_i, w_j)}{P(w_i)P(w_j)} = \log[(\sum_{w'_i, w'_j \in D} count(w'_i, w'_j)) \frac{count(w_i, w_j)}{count(w_i) count(w_j)}]$$

or, to avoid negative values, **positive PMI (PPMI)**, $PPMI(w_i, w_j) = \max(PMI(w_i, w_j), 0)$.
A drawback of PMI is that it tends to assign high value to rare events. It is therefore advisable to apply a count threshold before using the PMI metric, or to otherwise discount rare events.

Distributed word representation
Count-based methods (such as PMI) represent a word as a vector $w \in \mathbb{R}^n$ where each dimension corresponds to a word in the corpus dictionary, so that w_i represents the score of the relation between w and a word w_i (for example, $w_i = PMI(w, w_i)$). Such vectors are typically sparse and is usually large.
On the opposite, the **distributed representation** of words meaning associates each word with a **dense** vector $w \in \mathbb{R}^d$ with $d << n$. The vector dimensions **do not** represent words not concepts, and we are not allowed to interpret them as concepts.
the semantics of words is **completely represented by the mutual position of words** in the vector space. In particular, we want to preserve the **proximity assumption** for which if two word vectors are close one to the other, then the two words have similar meaning.

Neural network models
We exploit a neural network having as input an n-gram of words $w_{1:n}$ and having as output a probability distribution over the next word.
Preliminary set up: the training set is a sequence $w_1, \dots w_T$ of words, where w_t is a word in the corpus vocabulary V.
The **objective function** is $f(w_t, \dots, w_{t-n+1}) = \hat{P}(w_t | w_1 \dots, w_{t-1}$ (for a n-gram), under the constraint that, for any $w_1, \dots, w_{t-1}$

$$\sum_{i=1}^{V} f(w_i, w_{t-1}, \dots, w_{t-n+1}) = 1, \text{ with } f > 0$$

The function f is decomposed in two parts:
A mapping from any word $w_i \in V$ to a vector $w_i \in \mathbb{R}^m$. The mapping is then a matrix $W \in \mathbb{R}^{V \times m}$ of free parameters and represents the **distributed feature vectors** of each word in the vocabulary;
a probability function over words, that is a function g that maps and input sequence of word vectors $w_{t-n+1}, \dots, w_{t-1}$ to a conditional probability distribution for the next word w_t. The output is then a vector $g \in \mathbb{R}^V$ such that $g_i = \hat{P}(w_t = i | w_1, \dots, w_{t-1})$.
Thus, according to the decomposition, we have

$$f(w_i, w_{t-1}, \dots, w_{t-n+1}) = g(w_i, w_{t-1}, \dots, w_{t-n+1})$$

The function g is then implemented by a neural network that has its own parameters ω. the complete set of parameters is then $\theta = (X, \omega)$. Learning is performed by stochastic gradient ascent (with ϵ as learning rate) as:

$$\theta \leftarrow \theta + \epsilon \frac{\delta \log \hat{P}(w_t | w_{t-1}, \dots, w_{t-n+1})}{\delta \theta}$$

Neural network models for word embedding As we have seen, the neural language model *learns* the embedding as part of its parameter estimation process. Other models have been proposed in particular by relaxing the probabilistic output requirement. Instead of computing a probability distribution over target words given a context, the Collobert and Weston model only attempts to assign a score to each word, such that the correct word scores above incorrect ones.

Word2vec
Word2vec is a method for efficiently represent words in vector space.
It used either the **Continuous Bag of Words (CBOW)** or the **Skip-gram** models for representing the word context and two different optimization objectives that are **Negative-Sampling** and **Hierarchical Softmax**. We will see Negative-Sampling.

Consider a set D of **correct** word pairs (w_i, w_j) (e.g., valid bigrams) and a set $\bar{D}$ of

bad word pairs. The goal of the algorithm is to estimate $P(D = 1|w_i, w_j)$. The objective is to maximize the log-likelihood of the pairs in $D \cup \bar{D}$.

$$L(\Theta; D; \bar{D}) = \sum_{w_i, w_j \in D} \log P(D = 1|w_i, w_j) + \sum_{w_i, w_j \in \bar{D}} \log P(D = 0|w_i, w_j)$$

where, given the score $s(w_i, w_j)$

$$P(D = 1|w_i, w_j) = \frac{1}{1 + e^{-s(w_i, w_j)}}$$

The positive samples D are taken from the corpus, while the negative ones are sampled according to the word frequency in the corpus.
CBOW: the scoring function is defined as $\sum_{j=1}^{k} w_i \cdot w_j$ for a word context of k words
Skip-gram: in the skip-gram variant, assumes that the elements of the context are all independent, such that

$$P(D = 1|w_i, w_{1_k}) = \sum_{j=1}^{k} \log \frac{1}{1 + e^{-w_i \cdot w_j}}$$

In practice, Word2vec is implemented by a neural network composed by a $W^{V \times V}$input layer, a $H^{V \times m}$ hidden layer, and a w^V output layer, where V is the size of the vocabulary, ($W^{V \times V}$) is the one-hot encoding representation of words, and m is the dimension of the embedding vectors.
The hidden layer stores the weights that are used to feed the output layer and that are learned by the network.
The output layer is a **softmax regression classifier** that given the one-hot vector of word w_i and the weights in the hidden layer estimates the probability of each word w_j to be a word in the context of w_i.
The main idea of word2vec is to discard the input and the output layers and keep the hidden layer as the word embedding **dense** matrix of dimension $V \times m$.
The main interesting property of Word2vec is that it assigns **similar vectors** to words that have **similar context**. why? Suppose to have (w_i, w_j) and (w_z, w_j) such that the word w_j appears in the contexts of both w_i and w_z. Since the target to estimate for w_i and w_z is the same, the only way the network has to assign the same prediction to w_i and w_z is to learn the same weights in H_i and H_z.

GloVe
GloVe stands for *Global vectors* and this summarizes the main idea of the algorithm. We have seen that word2vec relies only on **local information** about a word because the word semantics is only affected by the surrounding words.
Glove starts instead from a global **co-occurrence matrix** having the intuition that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning.
by learning the vectors that are good for predicting co-occurrences instead of single words (like in Word2vec) we inject more information in the final word embedding vectors.
The main intuition of GloVe is represented by its objective function that is

$$w_i \cdot w_j + b_i + b_j = \log p(w_i, w_j)$$

where b_i and b_j represent bias parameters that are learned together with the word vectors.
In other terms, the training objective of GloVe is to learn word vectors such that their dot product equals the logarithm of the word's probability of co-occurrence. Owing to the fact that the logarithm of ratio equals the difference of logarithms, this objective associates (the logarithms of) ratios of co-occurrence probabilities with vector differences in the word vector space. Because these ratios can encode some form of meaning, this

information gets encoded as vector differences as well.

using word embedding
Word vectors can be used in tow ways: **training a specific word embedding model** (Word2vec, GloVe or others) for a corpus (assuming to have enough contents) or **exploit a pre-trained model** (trained usually over milions or even bilions of data) to embed the corpus words.
Word embedding is extremely useful for a variety of applications:

- Text search and retrieval
- Measuring the semantic distance among words
- Feeding a neural network language model
- text classification, wither supervised or unsupervised

Properties of word embedding
Similarity among group of words: thanks to linearity, computing the average cosine similarity from a group of words to all other can be calculated as

$$sim(w,(w_1,\dots,w_k)) = \frac{1}{k}\sum_{i=1}^{k} sim_{cos}(w,w_i)$$

Analogy

$$analogy(w_i : w_j \to w_k :?) = argmax_{w\in V\notin\{w_i,w_j,w_k\}} cos(w, w_k - w_i + w_j)$$

5 Introduction to Language Models

Introduction
A **language model** is essentially a probability distribution over a sequence of words

$$P(w_1, w_2, \dots, w_n)$$

which can be used for a surprisingly high number of tasks, including document search, document classification, text summarization, text generation, machine translation, and many others.
Note: instead of estimating the probability distribution of words, we can work at a finer granularity on the distribution of substrings of fixed length in words (e.g., characters, 2-chars blocks).
Example 1. A LM may be used to guess the next word in a sequence

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

Example 2. Or to guess the author (or any other categorical attribute) of as text

$$P(author|w_1, w_2, \dots, w_n)$$

Example 3. Select the correct translation for a sentence

$$P(English|option1) \text{ vs } P(English|option2)$$

Types of Language Models
Statistical Language Models: estimate the probability distribution of words by enforcing statistical techniques such as n-grams maximum likelihood estimation (MLE) or Hidden Markov models (HMM).
Neural language model: each word is associated with an embedding vector of fixed size and a Neural Network is used to estimate the next word given a sequence of proceeding words.

Introduction to neural networks
Neural Network are composed by two or more **layers of nodes**, where each layer output provides an input for the subsequent layer. Such an input is combined with **weights** associated with the node connections before feeding the subsequent layer. in general terms, a NN can be seen as a tool for computing a combination of linear and non linear transformations of the form

$$\hat{y} = \phi(Wx + b)$$

where $\hat{y}$ is vector representing the **network prediction**, x is the **input vector** (the data), W is a **matrix of weights** (that the network aims at learning), and $\phi(\cdot)$ is a (potentially) non linear transformation applied to data before the final prediction is returned, often called **activation function**.

Learning
The NN main goal is to find (**learn**) the value of its parameters Θ (the weights W and the bias b) that make it possible obtain a prediction $\hat{y}$ as much close as possible to the real output y, which is known from the examples available in the **training set** 8**supervised learning**.
To perform learning, several **iterations** are tried starting with random values for the weights. For each iteration we update weights and we base the subsequent iteration on the feedback provided by the **error in the prediction**, with the goal of **minimizing the error**. A crucial components of NN is thus to have a function that measures the error, called **loss function**. Given t as one of the iterations, η being the learning rate, $L(\cdot)$ the loss function, and X represents input data. The general form of the learning sep is

$$W^{t+1} \leftarrow W^t + \eta L(X)X$$

Overview
When NN are used for **textual data**, usually we have:
Input is **word vectors**, either sparse or dense. Dense vectors for words can be given as input (i.e., used a pre-trained models, such as **word embeddings, LDA**) or can be calculated by the network itself as part of the training process, starting from a sparse word representation (such as **one-hot encoding, co-occurrence count, n-grams matrix**).
The **output** depends on the task (**multi-class** or **multi-label classification, autoencoders**).

Autoencoders
Autoencoders are special network architecture where an input (represented by an input layer of size n) is mapped onto itself (an output layer of size n), such as when a network is used to map words on other words.
An example that we have seen is Word2vec. An interesting side-effect of this architecture is that we can take the hidden layer as a reduced dense representation of the input (as in word embeddings).

Activation function
there are many options for the choice of the activation function $\phi(\cdot)$ having that $\hat{y} = \phi(Wx)$

- $\phi(x) = x$ (identity)
- $\phi(x) = x^+$ (sign)
- $\phi(x) = \frac{1}{1-\epsilon^{-x}}$ (sigmoid)
- $\phi(x) = \max\{x, 0\}$ (**R**ectified **L**inear **U**nit)
- $\phi(x) = \frac{\epsilon^{2x}-1}{\epsilon^{2x}+1}$ (tahn)

- $\phi(x) = \max\{\min\{x, 1\}, -1\}$ (hard tahn)

Output nodes
When we need to predict one (or more) outputs among multiple options, such as in case of multi-class (or multi-label) classification or with autoencoders, a very common choice is to model the output layer as a **softmax** layer, in order to enforce a probabilistic interpretation of the results.
Having an output with n dimensions:

$$\phi(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp x_j} \forall i \in \{1, \dots, n\}$$

Loss functions
the use of *softmax* produces a probabilistic output, which requires also a some constraints on the choice of the loss functions.
Binary targte (logistic regression)

$$L = \log(1 + \exp(-y \cdot \hat{y}))$$

categorical targets (cross-entropy loss): given $\hat{y}_1 \dots, \hat{y}_k$ as the probabilities predicted for each of the k targets, assume that $r \in \{1, \dots, k\}$ is the correct target according to the training set for the instance under evaluation. then the cross-entropy loss is

$$L = -\log(\hat{y}_r)$$

Multilayer networks
When a network has multiple hidden layers, you can see it as a composition of function of the form $h_n(\dots h_1(f(x)))$, where each function h_i corresponds to the i-th hidden layer. However, this makes training more difficult, because the error gradient has to be back-propagated through the layers.
The algorithm performing training is articulated in two phases, a **forward phase** and a **backward phase**.
Forward: the input produces forward cascade of computation across the layers. The final output is compared with the training set expected output and the **derivative of the loss function is computed**.
Backward: denote w_{h_{r-1},h_r} as the weights of the transition between layer h_{r-1} and h_r and consider to have $h_1, \dots, h_k$ hidden layers before the output layer o. The Loss function derivative is decomposed along the path from h_1 to h_k as follows

$$\frac{\delta L}{\delta w_{h_{r-1},h_r}} = \frac{\delta L}{\delta o}[\frac{\delta o}{\delta h_k} \prod_{i=r}^{k-1} \frac{\delta h_{i+1}}{\delta h_i}] \frac{\delta h_r}{\delta w_{h_{r-1},h_r}} \forall r \in 1 \dots k$$

where the component $\frac{\delta o}{\delta h_k} \prod_{i=r}^{k-1} \frac{\delta h_{i+1}}{\delta h_i}$ has to be aggregated (summed) for each pattern of nodes connecting h_r to o.

Recurrent NN (RNN)
When dealing with **sequence-to-sequence** learning, we aim at predicting the value s_i in a sequence (e.g., the i-th word in a text) given the previous $s_{i-n+1}, \dots, s_{i-1}$ sequence elements (e.g., the previous words).
the issue here is that with a feed forward network, each prediction $\hat{s}_i$ may be based on data about the previous elements in the sequence (such as for n-gram models), but it is **independent** from the previous predictions of the network itself.
The idea of **RNN** is instead to use the output $\hat{s}_i$ of the network on the instance s_i as an input (together with data) for the subsequent prediction(s) $\hat{s}_{i+j}$.
The basic idea is that the state of the hidden layer h_t at time t if a function of the form

$$h_t = f(h_{t-1}, x_t)$$

Given d as the data dimensions (e.g., the vocabulary for language models) and p as the dimension of the hidden layers, we will have to work with three matrices of weights:$W_{xh} \in \mathbb{R}^{p \times d}$, $W_{hh} \in \mathbb{R}^{p \times p}$ and $W_{hy} \in \mathbb{R}^{d \times p}$ for input to hidden layer, hidden layer to hidden layer. and hidden layer to ouput. thus we will have:

$$h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1}),\ y_t = W_{hy}h_t$$

RNN Training
The softmax probabilities of the correct words at various time-stamps are aggregated to create the loss function.
The back-propagation algorithm is updated to **backpropagation through time (BPTT)**

1. running the input sequentially in the forward direction through time and computing the error/loss at each time-stamp (same as BP)
2. computing the changes in edge weights in the backwards direction on the network without any regard for the fact that weights in different time layers are shared (same as BP)
3. adding all the changes in the (shared) weights corresponding to different instantiations of an edge in time (BPTT specific)

Intuition of Long Short Term Memory networks (LSTM)
In RNN a common problem is that successive multiplication by the weight matrix is highly unstable (*vanishing / exploding gradients* problem).
This problem is an issue especially for long sequences, requiring the network to have a **long memory**.
To address the problem, in LSTM we introduce a new hidden vector of p dimensions, referred as the *cell* state. The cell state is a kind of long-term memory that retains at least a part of the information in earlier hidden states by using a combination of partial *forgetting* and *increment* operations on previous cell states.

6 NLP Basics

NLP
By 'Natural Language Processing' we refers to the computer processing of natural language, for any purpose, regardless of the level of depth of the analysis. With natural language we indicate the language we use in everyday life, such as English or Italian, and is synonymous with human language, mainly to be able to distinguish it from formal language, such as programming languages and mathematical notions.
Natural language is the most natural and common form of human communication, and it's growing exponentially in recent years with the spread of social networks and the internet in general. Compared to formal language, natural language is much more complex, and it often contains undertones and ambiguities, which makes it very difficult to process.
The ultimate aim of NLP is to read, understand, and decode human languages in a manner that is valuable.

How NLP can be helpful
NLP is the core of tools we use every day, such as:

- translation software
- chatbots
- spam filters
- ...

For example, our GMail automatically categorized as Promotions, Social, primary, or Spam the emails we receive.
Or personal assistant applications such as OK Google, Siri, Cortana, and Alexa.

Why is NLP so difficult?
NLP is a difficult problem in computer science due to the nature of the human language. The rules of a language can be of low-level, such as using the character 's' to signify the plurality of items, or high-level, for example when someone uses a sarcastic remark. in order to understand completely the meaning of a message we need to understand both *the meaning of words* and *the way in which concepts are linked together*. While humans can easily master a language, the ambiguity and imprecise characteristics of the natural languages are what make NLP difficult for machines to implement.

What are the techniques used in NLP?
The key technique used to complete the tasks of NLP are syntactic and semantic analysis. We use them to convert the unstructured language data into a form that computers can understand.

- Syntactic analysis (apply grammatical rules): is used to determine the way a natural language aligns with the rules of grammar. Some specific algorithms are used to apply grammar rules to words and extract their meaning.
- Semantic analysis (understand the meaning): it is more difficult and refers to the linguistic and logic that are conveyed through a text. it involves the definition of computer algorithms to find out the interpretation of words and the structure of the sentences.

Tools:

- **Natural language Toolkit (NLTK)**: is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to many corpora and lexical resources. Also, it contains a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, best of all, NLTK is a free open source, community-driven project.

- **Industrial-Strength Natural Language Processing (spaCy)**: is designed to be industrial grade but open-source library with a lot of in-built capabilities and it is becoming increasingly popular for processing and analyzing data in NLP. spaCy comes with pretrained NLP models that can perform common LP tasks, such as tokenization, parts of speech (POS) tagging, named entity recognition (NER), lemmatization, transforming to word vectors et..., and can deal large scale unstructured text data to process and derive insights. The factors that work in the favor of spaCy are the set of features it offers, the ease of use, and the fact that the library is always kept up to date... it is your numpy for NLP - it is reasonably low-level, but very intuitive and performant.

NLTK vs spaCy
Differences between them:

- for a particular problem NLTK contains a lot of algorithms while spaCy provides the best algorithm and keeps it up to date.
- NLTK processes strings while spaCy uses an object-oriented approach. So NLTK takes strings as input and return strings or lists of strings as output, instead spaCy return a document object whose words and phrases are objects themselves.
- spaCy has support for word vectors whereas NLTK does not.
- as spaCy uses the latest and best algorithms, its performances is usually good as compared to NLTK. In words tokenization and POS.tagging (Part-Of-Speech tagging) spaCy performs better, but in sentence tokenization, NLTK outperforms spaCy. This is due to the different approaches used, as spaCy builds a parse tree for each sentence and provides much more information about the text using a more robust method.

Tokenization
Tokenization is the task of chopping a defined document unit up into pieces, called tokens, perhaps at the same time throwing away certain characters such as punctuation.
It is a common task in NLP and it is a fundamental step in both traditional NLP methods and Advanced Deep Learning-based architectures. A tokenizer breaks unstructured data and natural language text into chunks of information (sentences, words, characters, or subwords) that can be considered as discrete elements which can be used directly by a computer to trigger useful actions and responses.
We can perform a RegEx (regular-Expression9 or corpus tokenization.

Corpus tokenization with NLTK
NLTK is a string processing library. It takes strings as input and return strings or lists of strings as output.

```
q = '''Muffins cost $9.88. Can you buy me?'''
from nltk.tokenize import word_tokenize
word_tokenize(q)
## ['Muffins', 'cost', '$', '9.88', '.', 'Can', 'you', 'buy', 'me', '?']
```

Corpus tokenization with spaCy
spaCy uses object-oriented approach and when we parse a text, it return document object whose words and sentences are objects themselves.

```
import spacy
import en_core_web_sm
a = '''I don't have enough money, buy it by yourself!'''
nlp = en_core_web_sm.load()
doc = nlp(a)
[word.text for word in doc]
## ['I', 'do', "n't", 'have', 'enough', 'money', ',', 'buy',
'it', 'by', 'yourself!]
```

Regular-Expression tokenization
Regular-Expression can be used if you want complete control over how to tokenize the text, but since they increase the complexity (sometimes even significantly) of the code it is advisable to use it only if the normal tokenizers are insufficient.
First you need to decide how you want to tokenize a piece of text as this will determine how you construct your regular expression.

RegEx Metacharacters
Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	$[a-m]$
\	Signals a special squence (can also be used to escape special charcters)	\d
.	Any character (except newline charcter)	he..0
ˆ	Starts with	ˆhello
$	Ends with	planet$
*	Zero or more occurrences	'he.*o
+	One or more occurrences	he.+o
?	Zero or one occurrences	he.?o
{}	Exactly the specified number of occurences	he{2}o
|	Either or	falls|stays
()	Capture and group	

RegEx Special Sequences
A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	if the specified characters are at the beginning of the string	\AThe
\b	where the specified characters are at the beginning or at the end of a word	\bain ain\
\B	where the specified characters are present, but NOT at the beginning (or at the end) of a word	\Bain ain\B
\d	where the string contains digits (numbers from 0-9)	\d
\D	where the string DOES NOT contains digits	\D
\s	where the string contains a white space character	\s
\S	where the string DOES NOT contains a white space character	\S
\w	where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	\w
\W	where the string DOES NOT contain any word characters	\W
\Z	if the specified characters are at the end of the string	Spain\Z

RegEx Sets
A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	where one of the specified characters (a, r, or n) are present
[a-n]	for any lower case character, alphabetically between a and n
[ˆarn]	for any character EXCEPT a, r, and n
[0123]	where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	for any digit between 0 and 9
[0-5] [0-9]	for any two-digit from 00 to 59
[a-zA-Z]	for ay character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, $, {} has no meaning, so [+] means: return a match for any + character in the string

Normalization
Converting a token into its base form (base form = word - inflectional form).

Why Normalize text?
To enhance our analysis we need to reduce randomness, gaining many benefits of normalizing the text input of our NLP systems:

1. Reduce the variation leaving us with fewer input variables to deal with, improving overall performance and avoid false negatives especially in the case of expert systems and information retrieval tasks;
2. reduce the dimensionality and lowers the amount of computation needed for creating embeddings;
3. Clean inputs and help to get a predefined 'standard' before text being used.

Two popular methods used for normalization are stemming and lemmatization.

Stemming and Lemmatization
Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. this indiscriminate cutting ca be successful in some occasions, but not always. the most common algorithm for stemming English is *porter's algorithm*, but other stemmer exist like *Snowball stemmer* and *Lancaster stemmer*.
Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. In the latter, deep linguistics knowledge is required to create the dictionaries that allow the algorithm to look for the proper form of the word. Once this is done, the noise will be reduced and the results provided on the information retrieval process will be more accurate.

But before we should...

- removing punctuation (but careful with Emoticon and Emoji) and stopwords.
  - you can remove Emoticon and Emoji or replace with a spoken text
- converting text to lowercase, but this is a sensitive issue because many proper names derive from common names (Bush, Bill). The best solution is to create a list of 'untouchable' words.
- number words $\rightarrow$ numeric
- handling unicode characters - accented letters and some punctuation

Vector Space Model
The representation of a set of documents as vectors in a common vector space.

What is a vector?
A **vector** is an ordered finite list of numbers. Its entries are called **elements** of the vector. The **dimension** of the vector is the number of elements it contains

$$[0, 1, 4, 6, 3, 9, 0, 5, 2]$$

Denoting an n-dimensional vector using the symbol a, the i-th element of the vector a is denoted with a_i, where the subscript i is an integer index that runs from 1 to n. A vector is said to be **sparse** if many of its elements are zero, i.e., if $a_i = 0$ for many i.

Vector Space

A vector space is a collection of vectors, which may be added together and multiplied by numbers, called scalars.
Two vectors of the same size can be added together by adding the corresponding element:

$$a + b = (a_1 + b_1, \dots, a_n + b_n)$$

A vector can be multiplied by a scalar, k, by multiplying every element of the vector by the scalar.

$$k \cdot a = (ka_1, \dots, ka_n)$$

Some Vector Space operations...

- **inner product** of two n-vectors:

$$a \cdot b = (a_1 \cdot b + \dots + a_n \cdot b_n)$$

- **euclidean norm** of a n-vector:

$$\|a\| = \sqrt{a_1^2 + \dots + a_n^2}$$

- **euclidean distance** between two n-vectors:

$$dist(a, b) = \|a - b\|$$

- **angle** between two n-vectors

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

Text Data Vectorization

Processing natural language text and extract useful information requires the text to be converted into a set of numerical features.
Word Vectorization and the more advanced Word Embeddings are methodologies in NLP to map text to a corresponding vector of real numbers which can be used to support automated text mining algorithms.

Vector Space Model: definitions

Terms are generic features that can be extracted from text documents.
Typically terms are single words, keywords, n-grams, or longer phrases.
Documents are represented as vectors of terms. Each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is non-zero.
Several different ways of computing these values, also known as (term) weights, have been developed.

$$d = (w_1, \dots, w_n)$$

The **Corpus** represents a collection of documents (the dataset). It is represented as a vector of documents, i.e. a matrix of terms. Each element $C_{d,t} = w_{d,t}$ represents the weight of the t-th term in the d-th document.
The **Vocabulary** is the set of all unique terms in the corpus.

Vector Space Model's limits

- Lack of support for semantic information (e.g. word senses)
- It is not sensitive to syntactic information (e.g. syntatgmatic structures, word ordering, proximity information)

- Independence between terms is assumed
- The control provided by a Boolean model is missing (e.g., the fact that a term MUST appear in the document). Given the query 'a b', documents in which 'a' is a frequent while 'b' is absent are indistinguishable from those in which both 'a' and 'b' occur but rarely.

Scoring
Essential for search engine to rank-order the documents matching a query.

When we use the scoring?
One of the most common uses of text mining is in search engines that score against the query in question for each matching document.
There are several ways you can score, such as:

- Binary scoring
- Count scoring
- Frequency scoring
- TF-IDF scoring

Simpler scoring

1. **binary scoring**: simply mark 1 when a particular word is present in a document, and 0 when the word is not present
2. **Count scoring**: count the number of times the particular word has occurred in the document
3. **Frequency scoring**: calculate the number of times the words appear in the document out of all the words in the document

TF-IDF scoring
TF-IDF is calculated by multiplying the number of times the word appears in the document and the inverse of the frequency of the word in the set of documents

$$TD - IDF_{td} = TD_{td} \times IDF_t$$

- *Term Frequency*: the number of times a word appears in the document.
  TF = number of repetition of a word in a document \ Number of words in the document
- *Inverse Document Frequency*: the inverse of the frequency of the word in the set of documents.
  IDF = log(Number of documents \ Number of documents containing the word)

We use also the IDF because the raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query even if certain terms have little or no discriminating power in determining relevance. The IDF attenuate the effect of terms that occur too often in the collection to be meaningful for relevance determination: the IDF of rare term is high, whereas the IDF of a frequent term is liekly to be low.

Feature Space Reduction
in VSM, a text will typically be a very sparsely populated vector living in a very high-dimensional space.
It is often desirable to reduce the dimension of the feature space while retaining as much information as possible.
Given an $d \cdot t$ matrix C with t large, it is often desirable to project the rows onto a

smaller-dimensional space, given a matrix of shape $d \cdot k$ with $k \ll t$.
We would like this projection to keep the variance of the samples as large as possible, because this corresponds to losing as little information as possible.

Principal Component Analysis (PCA)
A standard method for feature space reduction is PCA, which projects a set of points onto a smaller dimensional affine subspace of 'best fit'.
PCA is linear dimensionality reduction using SVD of the data to project it to a lower dimensional space.
The essence of the data is captured in a few principal components, which themselves convey the most variation in the dataset. PCA reduces the number of dimensions without selecting or discarding them. Instead, it construct principal components that focus on variation and account for the varied influences of dimensions. Such influences can be traced back from the PCA plot to find out what produces the differences among clusters.

Probability Ranking
Essential for a search engine to rank-order the documents matching a query.

Probability Ranking Principle
Using a probabilistic model, the obvious order in which to present documents to the user is to rank documents by their estimated probability of relevance with respect to the information need.
If a reference retrieval system's response to each request is a ranking of the documents in the collection in order of decreasing probability of relevance to the user who submitted the request, where the probabilities are estimated as accurately as possible on the basis of whatever data have been made available to the system for this purpose, the overall effectiveness of the system to its user will be the best that is obtainable on the basis of those data.
In the simplest case of the PRP, there are no retrieval costs or other utility concerns that would differentially weight actions or errors. You lose a point for either returning a nonrelevant document or failing to return a relevant document (such a binary situation where you are evaluated on your accuracy is called 1/0 loss).

The PRP with retrieval costs
Return a document if the expected cost of finding (or rejecting) it usefully is lower than the expected cost of finding (or rejecting) it wrongly.

- C_1: cost of not retrieving a relevant document
- C_0: cost of retrieval of a nonrelevan document
- d: a specific document
- d': all documents not yet retrieved

$$C_0 \cdot P(R=0|d) - C_1 \cdot P(R=1|d) \leq C_0 \cdot P(R=0|d') - C_1 \cdot P(R=1|d')$$

then d is the next document to be retrieved.

Binary Independence Model (BIM)
BIM is the model that has traditionally been used with the PRP. Documents and queries are both represented as binary term incidence vectors:

- a document d is represented by the vector $\vec{x} = (x_1, \ldots, x_M)$ where $x_t = 1$ if term t is present in a document d and $x_t = 0$ if t is not present in d.
- we represent q by the incidence vector $\vec{q}$

The model recognizes no association between words, so the terms are modeled as occurring in documents independently.

Given query q:

- for each document d need to compute $p(R|q,d)$
- replace with computing $p(R|q,x)$ where x is a binary term incidence vector representing d.
- interested only in ranking

Will use odds ratio and Bayes' Rule:

$$P(R=1|\vec{x},\vec{q}) = \frac{P(\vec{x}|R=1,\vec{q})P(R=1|\vec{q})}{P(\vec{x}|\vec{q})}$$

$$P(R=0|\vec{x},\vec{q}) = \frac{P(\vec{x}|R=0,\vec{q})P(R=0|\vec{q})}{P(\vec{x}|\vec{q})}$$

$$O(R|\vec{x},\vec{q}) = \frac{P(R=1|\vec{x},\vec{q})}{P(R=0|\vec{x},\vec{q})}$$

7 NLP other Features

N-grams
co-occurring words within a given document

Unigram, bigram, trigram, ..., N-gram
N-grams are continuous sequences of words/symbols/tokens in a document.
We use different types of n-grams because them are suitable for different types of applications. You should try different n-grams on your data in order to confidently conclude which one works the best among all for your text analysis. I.e., in some text classification projects, such as spam detection, varying the number of n-grams can produce different results.
When considering heterogeneous legitimate messages, short n-grams (3-grams) produced the best models while longer n-grams (4-grams) were optimal when considering homogeneous legitimate messages.

Skip-grams
Can be used to predict the context word for a given target word.

Morphological analysis
Extract the category of the word by analyzing its internal structure.

Morpheme
Morphology is the study of the structure and formation of words. Its most important unit is the morpheme, which is defined as the 'minimal unit of meaning'.
Consider a words like: 'unhappiness', this has three **morpheme** (parts):

1. 'un' - prefix
2. 'happy' - stem
3. 'ness' - suffix

Bound morphemes, like 'un' and 'ness', cannot be words in their own right and need to be attached to a *free morpheme* ('happy'), which can appear on its own.
What is clear nowadays, however, is that the morpheme concept is only of limited value. It can certainly display the minimal units of grammatical analysis in a vast amount of language data.
The irregularities of English are not, after all, very many. And when it comes to the

analysis of the agglutinative language, the morpheme concept is invaluable, as these languages are, as it were, tailor-made for it.
But when we consider the difficulties of morphemic identification s whole... it is clear that the concept is not as all-embracing as it has sometimes be made out to be.

Syntactic analysis
Extract the category of the word by analyzing the context in which it occurs.

Syntactic knowledge
how sequences of words form correct sentences; knowledge of the rules of grammar.
With syntactical analysis we mean the analysis of the structure / grammar of the sentences, determining the subject and predicate and the place of nouns, verbs, pronouns, etc. In this way, the computer would be able to just read through the input sentence word by word and produce a structural description.
The main problem is that a word may function as different parts of speech in different contexts (you need read the entire sentence); we have to determine which part of speech is relevant in the particular context.
'Green is the color of the Hope' VS ' The Hope is green'.

Syntax tree - Part-Of-Speech
The process of classifying words into their parts of speech and labeling them accordingly.
A syntax tree is a tree representation of different syntactic categories of a sentence and can help us understand the syntactic structure of a sentence.

Abbreviation	**Meaning**	**Abbreviation**	**Meaning**
CC	coordinating conjuction	CD	cardinal digit
DT	determiner	EX	existential there
FW	foreign word	IN	preposition/subordinating conjuction
JJ	adjective	JJR	adjective, compparative (larger)
JJS	adjective, superlative (largest)	LS	list market
MD	modal (could, will)	NN	noun, singular (cat, tree)
NNS	noun, plural (desks)	NNP	propoet noun, singular (sarah)
NNPS	proper noun, plural (indians)	PDT	predeterminer (all, both, half)
POS	possessive ending (parent\'s)	PRP	personal pronoun (hers, him)
PRP$	possessive pronoun (her, his)	RB	adverb (occasionally)
RBR	adverb, camparative (greater)	RBS	adver, superlative (biggest)
RP	particle (about)	TO	infinite marker (to)
UH	interjection (goodbye)	VB	verb (ask)
VBG	verb gerund (judging)	VBD	verb past tense (pleaded)
VBN	verb past participle (reunified)	VBP	verb, present tense not 3rd person singular (wrap)
VBZ	verb, present tense with 3rd person singular (bases)	WDT	wh-determiner (that, what)
WP	wh-pronoun (who)	WRB	wh-adverb (how)

Part-Of-Speech
POS Tagging is useful in sentence parsing, information retrieval, sentiment analysis, etc.
A word can take on different parts-of-speech depending on the context of the sentence, so marking it with parts-of-speech helps you understand how it works grammatically.
POS Tagging helps the machine underdtand how a word is used in a sentence.

```
# NLTK
from nltk import pos_tag ##pre-trained tagger
from nltk import DefaultTagger
from nltk import RegexpTagger
from nltk import UnigramTagger
from nltk import BigramTagger
```

```
# spaCy
import spacy
doc="Some text"
for token in doc:
 print(f"""{token.text:{8}} {token.pos_:{8}} {token.tag_:{8}}
 {token.dep_:{8}} {spacy.explain(token.pos_):{20}}
 {spacy.explain(token.tag_)}""")
```

Using spaCy we can get:

- the Universal POS tags with *pos_*: these tags mark the core part-of-speech categories
- the Fine-grained part-of-speech with *tag_*
- the Syntactic dependency relation with *dep_*

Chunking
Chunking works on top of POS tagging, it uses pos-tags as input and provides chunks as output.
Similarly to POS tags, there are a standard set of Chunk tags like Noun Phrases (NP), Verb Phrase (VP), etc.
In order to create NP chunk, we define the chunk grammar using POS tags.
Chunking is used for entity detection (the subsets of tokens) by which machine get the value for any intention.

```
import nltk
from nltk import pos_tag
from nltk import RegexpParser
text = "They think of everything in terms of money"
tag = nltk.pos_tag(nltk.word_tokenize(text))
output = RegexpParser("""mychunk:{<VB.*>*<NN.*>?}""").parse(tag)
```

Named Entity Recognition (NER)
Named Entities are definite chunks that refer to specific types of real-world objects, such as organizations, persons, dates, and so on. The goal of NER is to identify all textual mentions of the Named Entities. This can be broken down into two sub-tasks:

- identifying the boundaries of the Named Entity
- identifying the type of the Named Entity

The task is well-suited to the type of classifier-based approach that we saw for POS tagging and noun phrase chunking. In particular, we can

- extract chunks corresponding to noun phrases
- build a tagger that labels each chunk using the appropriate type based on the training data (unigram/n-gram tagger, ...).

8 Text Classification

By using NLP, text classification can automatically analyze text and then assign a set of tags or categories based on its context.
Typical TC tasks include sentiment analysis, news categorization and topic classification.
Text classification is a problem where we have fixed set of classes / categories and any given text is assigned to one of those categories (**supervised**).
In contrast, *Text clustering* is the task of grouping a set of unlabeled texts in such a way that texts in the same group (called cluster) are more similar to each other than to those in other clusters (**unsupervised**).

Supervise
We want to assign documents to predefined categories (e.g.: languages, topics, etc...).
We can have binary, multi-class or multi-label classification:

- binary: assigned only 1 label of 2 classes
- multi-class: assign only 1 label of 3 or more classes
- multi-label: assigned more labels of 3 or more classes

Classification techniques:

- manual classification - identify a set of rules that give documents label
- machine learning techniques (*Discriminative* and *Generative*)
  - Discriminative: directly assume some functional form for $P(Y|X)$ and then estimate the parameters of $P(Y|X)$ with the help of the training data (e.g.: Logistic Regression)
  - Generative: estimate the prior probability $P(Y)$ and likelihood probability $P(X|Y)$ with the help of the training data and uses the Bayes Theorem to calculate the posterior probability $P(Y|X)$ (e.g.: Naive Bayes Classifier)

$$P(Y|X) = \frac{P(Y) \cdot P(X|Y)}{P(X)}$$

Unsupervised
In text clustering we don't know in advance what sort of class we are looking for.
We need to identify how many clusters there are in the data and then decide which document go on in which cluster.
An important principle of information retrieval is that 'documents in the same cluster are usually relevant to the same query'.

Text classification pipeline

1. Text data sets contain sequences of documents
2. We create a structured set for our training: unstructured text sequences must be converted into a structured feature space
3. the dimensionality reduction step is an optional part of the pipeline: reduce the time and memory complexity for their applications
4. Partitioning Data: allows you to develop accurate models that are relevant to a general set of data, not just the data the model has been trained on.
5. Choose the best classification algorithm
6. Evaluation of the result: understanding how model performs is essential to the use and development of text classification methods.

Feature Extraction & Reduction

1. Text Text Cleaning:
   - Tokenization
   - deal with slang and abbreviation
   - Capitalization, remove stop words and punctuation
   - Stemming / Lemmatization

2. Weighted Words: TF-IDF
3. Dimensionality Reduction: PCA to find new variables that are uncorrelated and maximizing the variance to 'preserve as much variability as possible'

Partitioning Data
Training Set: it's the subsection of a dataset from which the ML algorithm uncovers, or 'learn', relationship between the features and the target variable.
Validation Set: it's the subset of the data to which we apply the ML algorithm to see how accurately it identifies relationships between the known outcomes and the input features, and it can be used for improving or tuning algorithms.
Test Set: provides a final estimate of the ML model's performance after it has been trained and validated. It should never be used to make decisions about which algorithms to use or for improving or tuning algorithms.

Underfitting and Overfitting
Splitting a dataset is important for detecting if the model suffers of underfitting or overfitting.
Underfitting occurs when a data model is unable to capture the relationship between the input and output variables accurately, generating a high error rate on both the training set and unseen data. It occurs when a model is too simple, which can be a result of a model needing more training time, more input features, or less regularization. If a model cannot generalize well to new data, then it cannot be leveraged for classification or prediction tasks.
Overfitting happens when the model has been overtrained or when it contains too much complexity, resukting in high error rates on test data. Overfitting a model is more common than underfitting one, and underfitting typically occurs in an effort to aboid overfitting through a process called early stopping'.

Rocchio (Nearest Centroid) Classification
This classification algorithm uses TF-IDF weights for each word and using a training set of documents, it builds a prototype vector for each class which is an average vector over the training documents' vectors that belong to a certain class.
It then assigns each test document to the class with the maximum similarity between the test document an each of the prototype vectors.
The average vector computes the centroid of a class c (center of mass of its members):

$$\vec{\mu}(c) = \frac{1}{|D_c|} \cdot \sum_{d \in D_c} \vec{v_d}$$

where D_c is the set of documents in D that belongs to class c and $\vec{v_d}$ is the weighted vector representation of document d.
The predicted label of document d is the one with the smallest Euclidean distance between the document and the centroid:

$$c* = argmin||\vec{\mu}(c) - \vec{v_d}||$$

Logistic Regression [Discriminative]
LR is a linear classifier which predicts probabilities rather than classes. The LR model specifies the probability of binary output $y_i = (0, 1)$ given the input x_i, so does not perform statistical classification (it is not a classifier), though it can be used to make a classifier.
For instance, using a cutoff value and classifying inputs with probability greater than the cutoff as one class, below the cutoff as the other; this is a common way to make a binary classifier.

The regression coefficients are usually estimated using maximum likelihood estimation. First, we project the original data onto the candidate line and this give to each sample

a candidate log(odds) value. Then we transform the candidate log(odds) to candidate probabilities using a sigmoid function: $p = \frac{e^{\log(odds)}}{1+e^{\log(odds)}}$ where $\log(\frac{p}{1-p}) = \log(odds)$.
This continue until we find a function that fits well and it's done computing the line which maximize the log-likelihood (probability): $\log(p_1 = 0) + \log(p_2 = 0) + \log(p_3 = 1) + \log(p_4 = 1)$.

Multinomial Logistic Regression
In multinomail version, the LR use the 1 VS rest approach: for n classes, $n-1$ independent binary logistic regression models are computed, in which one class is chosen as a 'pivot' and then the other $n-1$ classes are separately regressed against the pivot.
This would proceed as follows, if class c_n (the last one) is chosen as the pivot, then for each $0 < i < n : \log(\frac{p(c_i)}{p(c_n)}) = \log(odds_i)$.
If we exponentiate both sides, and solve for the probabilities, we get the relation between them: $p(c_i) = p(c_n) \cdot e^{\log(odds_i)}$.
Limits: logistic regression's prediction requires that each data point is independent, and if we include the wrong independent variables, the model will have little to no predictive value.

Logistic Regression
PROS:

- will provide probability predictions and not only classification labels
- even with few data it is still able to produce pretty useful predictions
- it is not a resource hungry model
- won't overfit easily
- it scale very nicely and let you harvest your millions of rows

CONS:

- LR is strictly a classification method and it has lots of competition
- it is not immune to missing data unlike some other machine learning model (e.g., Decision tree)

Naive Byes Classifier [Generative]
The Naive Bayes algorithm classifiers instances with the label that is most likely given the corresponding set of features. For a document d and a class c, we have

$$P(c|d) = \frac{P(c) \cdot P(d|c)}{P(d)}$$

The most likely class (c_{MAP} maximum a posteriori class) is computed like:

$$c_{MAP} = argmax P(c|d) = argmax \frac{p(c) \cdot P(d|c)}{p(d)} = argmax P(c) \cdot P(d|c)$$

$$argmax P(c) \cdot P(x_1, x_2, \ldots, x_n|c)$$

where $P(c)$ can be estimated counting the fraction of instances in class c (count the relative frequencies in a corpus), while $P(d_i|c)$ can be fitted by maximum likelihood which could only be estimated of a very large number of training examples are available. The different naive Bayes classifiers differ mainly by the assumption they make regarding the distribution of $P(d_i|c)$.

Bernoulli Naive Bayes Classifier
In the Bernoulli event model, features are binary variables (term is present / absent). This event model is especially for classifying short texts.

$$P(x_i|c) = p_{i,c} if x_i = 1$$

and

$$P(x_i|c) = 1 - p_{ic,} if x_i = 0$$

Where $p_{i,c}$ is the probability of class c generating term i. The maximum likelihood estimator of $p_{i,c}$ is the fraction of documents containing term i in class c.

Multinomial naive Bayes Classifier
The Multinomial Naive Bayes Classifier can be written as:

$$P(c|d) = \frac{P(c) \cdot \prod_{n \in d} P(d|c)^{n_{wd}}}{P(d)}$$

where n_{wd} is denoted to the number of times word w occurs in document, and $P(w|c)$ is the probability of observing word w given class c. $P(w|c)$ is calculated as:

$$P(w|c) = \frac{1 + \sum_{d \in D_c} n_{wd}}{k + \sum_{w'} \sum_{d \in D_c} n_{w'd}}$$

To avoid $P(w|c) = 0$, that lead us in a situation with posterior probability $P(c|d)$ will be 0, we smooth the probability estimates by adding a dummy count (Laplace smoothing). The smoothing parameter accounts for features not present in the learning samples and prevents zero probabilities in further computations (k is the vocabulary size).

Set	**Document**	**Class**
train	love love kiss	p
train	beauty	p
train	war choice hate	n
train	good choice	p
test	choice love war	?

Priors $P(c) : P(p) = 3/4; P(n) = 1/4$
Conditional probabilities: $P(love|p) = (2{+}1)/(6{+}7) = 3/13; P(kiss|p) = (1{+}1)/(6/7) = 2/13; P(beauty|p) = (1{+}1)/(6{+}7) = 2/13; P(choice|p) = (1{+}1)/(6{+}7) = 2/13; P(war|p) = (0 + 1)/(6 + 7) = 13; P(war|n) = (1 + 1)/(3 + 7) = 2/10 = 1/5; P(choice|n) = (1 + 1)/(3 + 7) = 2/10 = 1/5; P(hate|n) = (1 + 1)/(3 + 7) = 2/10 = 1/5; P(love|n) = (0 + 1)/(3 + 7) = 1/10$
Choosing a Class: $P(p|test) = 3/4 \times (3/13)^2 \times (1/13) = 0.0005 P(n|test) = 1/4 \times 1/5 \times 1/10 \times 1/5 = 0.001$

Complement Naive Bayes Classifier
Multinomial Nave Bayes performs poorly on data sets with unbalanced classes. Complement Naive Bayes is an adaptation of the standard Multinomial Naive Bayes algorithm that is particularly suited for imbalanced data sets.
Complement Naive Bayes regularly outperforms Multinomial Naive Bayes (often by a considerable margin) on text classification tasks.

Naive Bayes Classifier
PROS:

- Probably one of the fastest, easiest to implement and most straight-forward machine learning algorithm
- NB scales linearly which makes it a great candidate for large setups

- It is noise resilient, and since it is not sensitive to noisy features that are irrelevant these won't be well represented in the Model
- NB uses very little resources (RAM and CPU) compared to other algorithms

CONS:

- problems if the dataset come with features that are not so independent of each other
- NB processes all features as independent and this means some features might be processed with a much higher bias
- NB might be too naive if you have tons of complex features

K-Nearest Neighbour [Discriminative]
KNN is a really simple way of categorize data: the model classify the new data by looking at the k 8a number) annotated data.
Given a test document d, the KNN algorithm find the k nearest neighbour of d among all the documents in the training set, and scores the category candidates based on the class of the k neighbours.
After sorting the score values, the algorithm assigns the candidate to the class with the highest score, the basic nearest classification uses uniform weights: that is, the value assigned to a query point is computed from a simple majority vote of the nearest neighbours. Under some circumstances, it is better to weight the neighbours such that nearer neighbours contribute more to the fit.
About picking a value for k:

- there is no way to determine the best value for k
- , low values for k (1 or 2) can be noisy and subject to the effects of outliers
- a large value for k smooths things out, but a too large k can exclude categories with few value in favor of larger categories.

PROS:

- it is even simpler than Naive Bayes (almost without math)
- non-parametric means KNN doesn't make assumptions regarding the dataset (good if you do not know too much about the dataset)
- it is perfect to make a simple prediction
- if you want to explore features with complex relations or if your data has outliers that you would like to keep in the considerations, KNN can do a great job in this sense
- good performance in non-linear situations

CONS:

- KNN is a hungry algorithm (RAM and CPU)
- if you want to work on datasets with many features this can be problematic with KNN
- k parameter for neighbour amounts and parameter for how distance is calculated can make a huge difference in the outcomes (few parameters but significant)
- all the attributes will be treated as equally important for the results

Support Vector Machine (SVM) [Discriminative]
The main principle of SVM is to determine separators in the search space which can best separate the different classes.
Thus SVM tries to make a decision boundary (hyperplane) in such a way that the separation between the two classes is as wide as possible.
We note that it is not necessary to use a linear function for the SVM classifier.
SVM, with a function that maps the data to a higher dimension where data is separable called 'kernel trick', construct a nonlinear decision surface in the original feature space by mapping the data instances non-linearly to a new space where the classes can be separated linearly with a hyperplane.
Linear SVM is used most often because of their simplicity and ease of interpretability.
As most of real world data are not fully linearly separable, we will allow some margin violation to occur. Margin violation means choosing an hyperplane, which can allow some data point to stay in either incorrect side of hyperplane and between margin and correct side of hyperplane. This type of classification is called soft margin classification.
The hard-margin does not deal with outliers, while in soft-margin we can have a few points incorrectly classified with a margin less than 1 (for every such point we have to pay a penalty 'C parameter').
PROS:

- when data has high dimension SVM can be the way to go and produce really accurate results
- when it comes to prediction they are quite fast

CONS:

- can be hard to define the correct parameters
- it can be quite costly computationally to train them
- SVM do not provide very sophisticated and interpretable reports that can be interpreted in an easy fashion

Decision Tree [Discriminative]
A decision tree is a hierarchical decomposition of the (training) data space, in which a condition on the feature value is used in order to divide the data space hierarchically.
Algorithms for constructing decision trees usually work top-down, by choosing a variable at each step that best splits the set of items.
Different algorithms use different metrics for measuring 'best', such as Gini impurity ($1 - p(Y)^2 - p(N)^2$) or information gain ($1 - p(Y) \cdot \log(p(Y)).p(N) \cdot \log(p(N))$). These measure the homogeneity of the target variable within the subsets and the probability that a randomly chosen sample in a node would be incorrectly labeled if it was labeled by the distribution of samples in the node.
The division of the data space is performed recursively in the decision tree, until the leaf nodes contain a certain minimum number of records or some conditions on class purity.
For a given test instance, we apply the sequence of conditions at the nodes, in order to traverse a path of the tree in top-down fashion and determine the relevant leaf node.
The majority (weighted) class label in the leaf node is used for the purpose of classification. Two approaches can be used to avoid overfitting:

- pre-pruning: we stop the tree construction a bit early. We prefer not to split a ode if its goodness measure is below a threshold value. But it is difficult to choose an appropriate stopping point.
- post-pruning: we go deeper and deeper in the tree to build a complete tree. If the tree shows the overfitting problem then pruning is done as a post-pruning step. We use the cross-validation data to check the effect of our pruning.

PROS:

- easy interpretation
- does not require normalization, scaling or pre-handling missing data
- training process is relatively faster
- there are lots parameters that can affect tje results of a decision tree algorithm which gives you more control over the results, efficiency and performance

CONS:

- the risk of overfitting is considerably high
- when there are lots of features and complex large datasets, DT can be limited when it comes to prediction powers

Random Forest [Discriminative]
Random (decision tree) forests are an ensemble learning method that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes of the individuals trees.
While the predictions of a single tree are highly sensitive to noise in its training set, the average of many tree is not, as long as the trees are not correlated. Random forests correct for decision trees' habit of overfitting to their training set.
A flexible model is said to have high variance because the learned parameters (such as the structure of the decision tree) will vary considerably with the training data. On the other hand, an inflexible model is said to have high bias because it makes assumptions about the training data (it is biased towards pre-conceived ideas of the data).
The DT is prone to overfitting when we don not limit the maximum depth because it has unlimited flexibility, meaning that it can keep growing until it has exactly one leaf node for every single observation, perfectly classifying all of them.
Rather than just simply averaging the prediction of trees, this model:

- Random sampling with replacement (bootstrapping, like in bagging) of training data points when building trees, gaining a forest with lower variance but not at the cost of increasing the bias
- Random subsets of features considered when splitting nodes (usually set to $\sqrt{N}$). Where N is the number of features.

PROS:

- being consisted of multiple decision trees amplifies random forest's predictive capabilities and make it useful for application where accuracy really matters
- RF offers lots of parameters to tweak and improve your machine learning model
- RF handles large datasets pretty well
- low risk of overfitting

CONS:

- RF are found to be biased while dealing with categorical variables
- slow Training
- not suitable for linear methods with a lot of sparse features

Bagging [Discriminative]
A bootstrap (random sampling with replacement) generates N uniform sample from the training set $B_1, B_2, \ldots, B_N$.
Then we have N classifiers (C) which C_i is built from each bootstrap sample B_i. Finally, our classifier C is generated from $C_1, C_2, \ldots, C_N$ and the output is the class most often predicted by its subclassifiers. Bagging classifier helps reduce the variance of individual estimators by sampling technique and combining the predictions.
PROS:

- many weak learners aggregated typically outperform a single learner over the entire set, and has less overfit.
- removes variance in high-variance low-bias weak learner
- can be performed in parallel, as each separate bootstrap can be processed on its own before combination

CONS:

- for weak learner with high bias, bagging will also carry high bias into its aggregate
- loss of interpretability of a model
- can be computationally expensive depending on the data set

AdaBoost (Adaptive Boosting) [Discriminative]
In boosting we use a forest of stumps (one level decision tree). Stumps do not carry the same weight in the final classification, with some being more important than others. The errors that the first stump makes influence how the second stump is made and so on: the errors that the previous stump makes on the D_t sample affects the way the D_{t+1} sample is constructed (badly classified data is more likely to be picked up in D_{t+1}). Finally, we get several groups of stumps with different weights that predict different things for the same input. The output with a greater result is the one that is chosen by the classifier.
Limits: both bagging and boosting introduce a loss of interpretability (the feature importance could not be discovered by these models) and they can be computationally expensive.
PROS:

- very simple to implement
- does feature selection resulting in relatively simple classifier
- fairly good generalization (suited for any classification problem)
- not prone to overfitting

CONS:

- suboptimal solution
- sensitive to noisy data and outliers

Classification tips

GridSearchCV
By partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.
A solution to this problem is a procedure called Cross Validation, which involves considering multiple choices for the pair (train, validation) sets and averaging evaluation metrics.
CV can be used to perform model selection using GridSearch (GridSearchCV) for the

optimal hyperparamters of the model.
Using this technique we can avoid validating our model on a set, and therefore we will only need the training and the test set.

What is the Cross Validation?
In the basic k-fold CV approach the training set is split into k smaller sets. The following procedure is followed for each of the k 'folds':

- a model is trained using of the folds as training data
- the resulting model is validated on the remaining part of the data

The performance measure reported by k-fold CV is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set).

Pipeline (workflow)
Sklearn provides a pipeline module to automate the execution of a sequence of typical tasks: data normalization, imputation of missing values, outlier elicitation, dimensionality reduction, classification, etc...
Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement fit and transform methods. The final estimator only needs to implement fit.

```
from sklearn.neighbors import NearestCentroid
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
pipe = Pipeline([('tfidf', TfidfVectorizer())
  , ('clf_Rocchio', NearestCentroid)])
# pipe.fit(X_train, y_train)
# pipe.score(X_test, y_test)
```

Model save

```
import pickle
## save the model to disk
# filename = 'gridPipeModel.sav'
# pickle.dump(cv_grid, open(filename, 'wb'))
#
## load the model from disk
# loaded_model = pickle.load(open(filename, 'rb'))
# y_predict2 = loaded_model.predict(X_pipe_test)
```

9 Clustering and Evaluation

Clustering
Clustering is the most common form of unsupervised learning. No supervision means that there is no human expert who has assigned documents to classes. In clustering, it is the distribution and makeup of the data that will determine cluster membership.
Clustering analysis is broadly used in applications such as exploratory data analysis, market research, customer segmentation, topics discovery. In every task of grouping a set of objects in such a way that objects in the same group (called cluster) are *similar* to each other than to those in other groups.

Soft / Hard clustering
An important distinction can be made between hard and soft clustering algorithms.
Hard clustering computes a hard assignment - each document is a member of exactly one cluster.

The assignment of soft clustering algorithms is soft - a document's assignment is a distribution over all clusters.
In a soft assignment, a document has fractional membership in several clusters.

Flat / Hierarchical clustering
A second important distinction can be made between flat and hierarchical clustering algorithms.
Flat (efficient and conceptually simple) clustering creates a flat set of clusters without any explicit structure that would relate clusters to each other.
Hierarchical clustering creates a hierarchy of clusters, a structure that is more informative than the unstructured set of clusters returned by flat clustering.

Cluster hypothesis
Documents in the same cluster behave similarly with respect to relevance of information needs.
The cluster hypothesis states the fundamental assumption we make when using clustering in information retrieval. The hypothesis states that if there is a document from a cluster that is relevant to a search request, then it is likely that other documents from the same cluster are also relevant. This is because clustering puts together documents that share many terms.
Some applications of clustering in information retrieval are:

- **Search result clustering**: SRC clusters the search results, so that similar documents appear together. It is often easier to scan a few coherent groups than many individual documents. This is particularly useful if a search term has different word sense.
- **Language modeling clustering**: it use a standard inverted index to identify an initial set of documents that match the query, but we then add other documents from the same clusters even if they have low similarity to the query (automobile or vehicle instead of car)

The hard flat clustering problem
Given a set of documents $D = \{d_1, \ldots, d_N\}$, a desired number of cluster K and an objective function that evaluates the quality of a clustering, we want to compute an assignment that minimizes (or, in other cases, maximizes) the objective function.
The objective function is often defined in terms of similarity or distance between documents.
We will see that the objective in K-means clustering is to minimize the average distance between documents and their centroids or, equivalently, to maximize the similarity between documents and their centroids.
The choice of the similarity function is crucial: different distance measures give rise to different clusterings. Thus, the distance measure is an important means by which we can influence the outcome of clustering.

The number of clusters
A difficult issue in clustering is determining the cardinality of a clustering (K). It is possible to use some heuristic method for choosing K and an attempt to incorporate the selection of K into the objective function. The brute force solution would be to enumerate all possible clusterings and pick the best. However, there are exponentially many partitions, so this approach is not feasible.
The *initial criterion* for the quality of a clustering is to get highest intra-cluster similarity (documents within a cluster are similar) and lowest inter-cluster similarity (documents from different clusters are dissimilar).
In python 3 methods usually used to evaluate the appropriate number of clusters are:

- the Elbow method: sum of squared distances of samples to their closest cluster center (inertia attribute)

- the Silhouette coefficient: $\frac{b-a}{\max(a,b)}$ where a is the mean within-cluster and b is the mean nearest-cluster distance
- the Calinski Harabasz criterion: ratio between the within-cluster dispersion and the between-cluster dispersion

K-means: objective function
The K-means objective is minimize the averages squared Euclidean distance of documents from their cluster centers where a cluster center is defined as the mean or centroid $\vec{\mu}$ of the documents in a cluster ω.
A measure of how well the centroids represent the members of their clusters is the residual sum of squares or RSS, the squared distance of each vector from its centroid summed over all vectors:

$$RSS_k = \sum_{\vec{x} \in \omega_k} |\vec{x} - \vec{\mu}(\omega_k)|^2$$

RSS is the objective function in K-means and our goal is to minimize it.

K-means: parameters
n_clusters:(default 8) might take some expertise to get right and this expertise might mean machine learning expertise as well as domain expertise
n_init: (default 10) it is used to define the number of initialization attempts for centroids of clusters. Initialization of centroids is an important concept for K-Means algorithm. If the initialization is not correct or as intended this value can be increased to make more attempts to initialize the model with optimum centroids. Default value 10 usually produces good results without compromising too much computational efficiency.
max_iter: (default 300) it is the maximum iterations k-means algorithm will make before giving the end results. You can check efficiency of iterations and K-Means is general by checking the inertia of K-Means using inertia attribute (sum of squared distances of samples to their closest cluster center).

K-means: algorithm
Input K, set of points $x_1, \ldots, x_n$, place centroids $c_1, \ldots, c_m$ at random location

- Repeat until convergence:
  - for each point x_i find the nearest centroid c_j and assign the point to the cluster
- For each cluster j compute the new centroid c_j= mean of all points assigned to cluster j in the previous step
- Stop when none of the cluster assignment change

The computation complexity is due to $O(\#iterations, \#clusters, \#instances, \#dimensions)$.

Hierarchical Clustering
Hierarchical clustering determines cluster assignments by building a hierarchy. It does not requires us to prespecify the number of clusters and most hierarchical algorithms that have been used in IR are deterministic.
The advantages of hierarchical clustering come at the cost of lower efficiency. The most common hierarchical algorithms have a complexity that is at least quadratic in the number of documents compared to the linear complexity of K-Means.
There are few differences between the applications of flat and hierarchical clustering in information retrieval, i.e., HC is appropriate for any of the applications shown. In general, we select flat clustering when efficiency is important (instead of the effectiveness) and hierarchical clustering when one of the potential problems of flat clustering (not enough structure, predetermined number of clusters, non-determinism) is a concern.

Hierarchical agglomerative clustering
The agglomerative clustering (HAC) use a bottom-up approach: treat each document as a singleton cluster at the outset and then successively merge (or agglomerate) pairs of clusters until all clusters have been merged into a single cluster that contains all documents.
An HAC clustering is typically visualized as a dendrogram.
It is the inverse of the K.means, even if the aim is to ensure that nearby points end up in the same clister.

1. Start with a collection C of n singleton clusters, each cluster containing one data point $c_1 = \{x_i\}$
2. Repeat until only one cluster is left:
    - find a pair of clusters that is closest ($\min D(c_i, c_j)$)
    - merge the clusters c_i, c_j into a new cluster c_{i+j}
    - remove the clusters c_i, c_j from the collection C and add c_{i+j}

HAC Dendrogram
Each merge is represented by a horizontal line. The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where documents are viewed as singleton clusters. We call this similarity the combination similarity of the merged cluster. By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.
A fundamental assumption in HAC is that the merge operation is monotonic. Monotonic means that if $s_1, s_2, \ldots, s_{K-1}$ are the combination similarities of the successive merges of an HAC, then $s_1 \geq s_2 \geq \cdots \geq s_{K-1}$ holds.
A non-monotonic hierarchical clustering contains at least one inversion $s_i \leq s_{i+1}$ and contradicts the fundamental assumption that we chose the best merge available at each step.

Linkage Criteria
The linkage criteria refers to how the distance between clusters is calculated, and in HAC the distance between two clusters can be:

- *Single Linkage*: the shortest distance between two point in each cluster
- *Complete Linkage*: the longest distance between two point in each cluster
- *Average Linkage*: the average distance between each point in one cluster to every point in other cluster
- *Ward Linkage*: the sum of squared differences within all clusters

Model Evaluation

Accuracy , precision and recall
Accuracy and precision are two terms that are often used incorrectly in the context of measurement, so it is important to know difference well. Accuracy indicates how close a measurement is to the accepted value. More generally, the accuracy of a value is a measure of how closely the results agree with the true or accepted value.
Precision is the ability of the classifier not to label as positive a sample that is negative (what proportion of predicted positives is truly positive? $\frac{TP}{TP+FP}$), adn recall is the ability of the classifier to find all the positive samples (what proportion of truly positives is correctly classified? $\frac{TP}{TP+FN}$).

Accuracy classification score
The accuracy score function computes the accuracy, either the fraction (default) or the count (normalize = False) of correct predictions.

If $\hat{y}_i$ is the predicted value of the i-th sample and y_i is the corresponding true value, then the fraction of correct predictions over $n_{samples}$ is defined as:

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

The top k accuracy score s a generalization of accuracy score (special case of $k = 1$), and the difference is that a prediction is considered correct as long as the true label is associated with one of the k highest predicted scores.
The balanced accuracy score avoids inflated performances estimates on imbalanced datasets. It is the macro-average of recall scores per class or, equivalently, raw accuracy where each sample is weighted according to the inverse prevalence of its true class. Thus for balanced datasets, the score is equal yo accuracy.

Cohen's kappa statistic
Compare labeling by different human annotators not a classifier versus a ground truth, and it is used to measure inter-rater reliability (and also intra-rater reliability) for qualitative (categorical) items. It is generally thought to be a more robust measure than simple percent agreement calculation, as it takes into account the possibility of the agreement occurring by chance.
The kappa score is a number between -1 and $+1$. Scores above 0.8 are generally considered good agreement; zero or lower means no agreement (practically random labels).

Hamming loss
The hamming loss computes the average Hamming distance between two sets of samples. If $\hat{y}_j$ is the predicted value of the j-th sample and y_i is the corresponding true value, and n_{labels} is the number of classes or labels, then the Hamming loss $L_{Hamming}$ between two samples is defined as

$$L_{Hamming}(y, \hat{y}) = \frac{1}{n_{labels}} \sum_{i=0}^{n_{labels}-1} 1(\hat{y}_i \neq y_i)$$

Confusion Matrix
Also know as an error matrix, represents the instances in an actual class while each column represents the instances in a predicted class.
By definition, entry i, j in a confusion matrix is the number of observations actually in group i, but predicted to be i group j.
The parameter normalize allows to report ratios instead of counts. The confusion matrix can be normalized in 3 different ways: 'pred', 'true', and 'all' which will divide the counts by the sum of each columns, rows, or the entire matrix, respectively.

F-measure
The F-measure (F_β and F_1 measures) can be interpreted as a weighted harmonic mean of the precision and recall. A measure reaches its best value at 1 and its worst score at 0.
With $\beta = 1$, F_β and F_1 are equivalent, and the recall and the precision are equally importan.

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

Precision-Recall curve
It is a useful measure of success of prediction when then classes are very imbalanced. In information retrieval, precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned.
The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low

false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall). A system with high recall but low precision return many results, but most of its predicted labels are incorrect when compared to the training labels. A system with high precision but low recall is just the opposite, returning very few results, but most of its predicted labels are correct when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly.

ROC Curve
ROC curves typically feature true positive rate on the Y axis, and false positive rate on the X axis. This means that the top left corner of the plot is the 'ideal' point - a false positive rate of zero, and a true positive rate of one. This is not very realistic, but it does mean that a larger area under the curve (AUC) is usually better.
The 'steepness' of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.
ROC curves are typically used in binary classification to study the output of a classifier. In order to extend ROC curve and ROC are to multiclass classification, it is necessary to binarize the output. One ROC curve can be drawn per label, but one can also draw a ROC curve by considering each element of the class indicator matrix a binary prediction (micro-averaging).
Another evaluation measure for multiclass classification is macro-averaging, which gives equal weight to the classification for each label.

10 Language Models

Language models uses

- Speech recognition: P('have a nice day)¿P('have a light may')
- Spelling correction: P('I am busy')¿P('I am xusy')
- Text generation: P('She is charismatic')¿P('She are charismatic')
- or even in
  - OCR
  - Summarization
  - Document classification

Language models for IR
Language model is in our everyday life, e.g. in search engines, mail agents or translation service.
The best way to get the document you are looking for is to type the words that would likely appear in it, but in the traditional probabilistic approach to IR we model the probability $P(R = 1|q, d)$ of relevance of a document d to a query q.
The language modeling approach build a probabilistic language model M_d from each document d, and ranks documents based on the probability of the model generating the query: $P(q|M_d)$.

Probabilistic language models
A language model is a function that puts a probabilistic measure over strings drawn from some vocabulary. That is, for a language model M over an alphabet $\Sigma : \sum_{s\in\Sigma^*} P(S) = 1$.
After generating each word, we decide whether to stop or to loop around and then produce another word, and so the model also requires a probability of stopping in the finishing state.
Such a model places a probability distribution over any sequence of words. By construction, it also provides a model for generating text according to its distribution.

Usually, the probability of a particular string / document is a very small number.

Types of language models
Formally, let $y_1, y_2, \ldots, y_n$ be tokens in a sentence, and $P(y_1, y_2, \ldots, y_n)$ the probability to see all these tokens (in this order).
We can use the chain rule to decompose the probability of a sequence of events into the probability of each successive event conditioned on earlier events:

$$P(t_1, t_2, t_3, t_4) = P(t_1)P(t_2|t_1)P(t_3|t_1t_2)P(t_4|t_1t_2t_3)$$

There are many more complex kinds of languages models, such as bigram language models, which condition on the previous term, $P_{bi}(t_1, t_2, t_3, t_4) = P(t_1)P(t_2|t_1)P(t_3|t_2)P(t_4|t_3)$ or trigram and so on up to N-grams.

Markov (independence) assumption
The straightfprward way to compute $P(y_t|t_1, \ldots, y_{t-1})$ is:

$$\frac{N(y_1, \ldots, y_{t-1}, y_t)}{N(y_1, \ldots, y_{t-1})}$$

where $N(y_1, \ldots, y_k)$ is the number of times a sequence of tokens $y_1, \ldots, y_k$ occur in text.
A problem with this is that many of the fragments $y_1, \ldots, y_k$ will zero out the probability of the sentence since them don't occur in corpus.
To overcome this problem, we make an independence assumption: the probability of a word only depends on a fixed number of previous words (only look at limited history).

N-grams models
We assume that:

$$P(y_t|y_1, \ldots, y_{t-1}) = P_n(y_t|y_{t-(n+1)}, \ldots, y_{t-1})$$

For example:

- $n = 1$ (unigram model): $P(y_t|y_1, \ldots, y_{t-1}) = P_1(y_t)$
- $n = 2$ (bigram model): $P(y_t|y_1, \ldots, y_{t-1}) = P_2(y_t|y_{t-1})$
- $n = 3$ (trigram model): $P(y_t|y_1, \ldots, y_{t-1}) = P_3(y_t|y_{t-2}, y_{t-1})$

The higher the value of N, the more the maximum likelihood estimation (MLE) to be calculated will have values close to 0 (often on the order of 10^{-6} or less). This can lead to the underflow, and trick to avoid it is to use base 10 algorithms (instead of 10^{-6} we get -6).

Smoothing
Smoothing is used to reassign some probability mass to unseen data.
Let's imagine we have to deal a case like this:

$$P(pants|\text{I'm the first mammal to wear}) = P_5(pants|\text{first mammal to wear}) = \frac{N(\text{first mammal to wear pants})}{N(\text{first mammal to wear})}$$

We need to avoid that either denominator or numerator is zero, because both these cases are not really good for the model.
With smoothing we redistribute probability mass, stealing from seen events and give to the unseen ones.

Smoothing techniques

- Backoff: use the maximum that you can ($P_5(|)$ or $P_4(|)$ or $P_3(|)$ or $P_2(|)$, even $P_1(|)$)
- Linear interpolation: mix all probabilities using scalar positive weights $\sum \lambda = 1$ such that $P_5(|) \approx \lambda_5 P_5(|) + \lambda_4 P_4(|) + \lambda_3 P_3(|) + \lambda_2 P_2(|) + \lambda_1 P_1(|)$
- Laplace smooting: $P_5(pants|\text{first mammal to wear}) = \frac{\delta + N(\text{first mammal to wear pants})}{\delta \cdot |V| + N(\text{first mammal to wear})}$
- More advanced tecnhiques:
  - Good - Turing
  - Kneser-Ney
  - Class-based-N-grams

Neural Language Models

1. **process context** (model-specific): get a vector representation for the previous context and using this representation the model predicts a probability distribution for the next token. This part could change depending on NN model architecture (CNN, RNN, etc...), but the result is to encode context
2. **generate a probability distribution for the next token** (model-agnostic): using context encoded, the probability distribution is generated

Intro to Artificial Neural Networks
An artificial neural network (ANN) is the piece of a computing system designed to simulate the way the human brain and process information. It is the foundation of artificial intelligence (AI) and solves problems that would prove impossible or difficult by human or statistical standards.
ANNs have self-learning capabilities that enable them to produce better results as more data becomes available. information that flows through the network affects the structure of the ANN because a neural network changes, or learns, in a sense - based on that input and output.
ANNs are considered nonlinear statistical data modeling tools where the complex relationships between inputs and outputs are modeled or patterns are found.
An ANN has three or more layers that are interconnected. The first layer consists of input neurons. Those send data on to the hidden layers, which in turn will send the final output data to the last output layer.

All the inner layers are hidden and are formed by units which adaptively change the information received from layer to layer though a series of transformations. Each layer acts both as an input and output layer that allows the ANN to understand more complex objects. Collectively, these inner layers are called the neural layer.
The units in the neural layer try to learn about the information gathered by weighing it according to the ANN's internal system. These guidelines allow units to generate a transformed result, which is then provided as an output to the next layer.
An additional set of learning rules makes use of **backpropagation**, a process through which the ANN can adjust its output results by taking errors into account. Through backpropagation, each time the output is labeled as an error during the supervised training phase, the information is sent backward. *Each weight is updated proportionally to how much they were responsible for the error.*
Hence, the error is used to recalibrate the weight of the ANN's unit connections to take into account the difference between the desired outcome and the actual one. In due time, *the ANN will learn how minimize the chance for errors and unwanted results.* Training an ANN involves choosing from allowed models for which there are several associated algorithms.

An ANN has several advantages but one of the most recognized of these is the fact that *it can actually learn from observing data sets.* In this way, ANN is used as a random function approximation tool. These types of tools help estimate the most cost-effective and ideal methods for arriving at solutions while defining computing functions or distributions.
ANN takes data samples rather than entire data sets to arrive at solutions, which saves both time and money. ANNs are considered fairly simple mathematical models to enhance existing data analysis technologies.
The two main type of NN are:

- *Convolutional Neural Network* **(CNN)**: a NN that has one ore more convolutional layers and are used mainly for image processing, classification, segmentation and also for other auto correlated data. A convolution is essentially sliding a filter over the input.
- *Recurrent Neural Network* **(RNN)**: a Nn that takes decisions based on current and previous inputs. Can be used to generating text, machine translation, speech recognition.

Perceptron
Percpetron is a single layer NN, and a multi-layer perceptron is called NNs. Perceptron is a linear classifier (binary), and it is used in supervised learning. It helps to classify the given input data. *The perceptron consists of 4 parts:* 1) Input values; 2) Weights and Bias; 3) Net Sum; 4) Activation Function.
The perceptron works on these simple steps:

- All the inputs x are multiplied with their weights w.
- Add all the multiplied values (Weighted Sum).
- Apply weighted sum to the Activation Function

Why do we need Wights and Bias? Weights shows the strength of the particular node. A bias value allows you to shift the activation function curve up or down.
Why do we need Activation Function? The activation functions are used to map the input between the required values like $(0, 1)$ or $(-1, 1)$.
Where we use Perceptron? Perceptron is usually used to classify the data into two parts. therefore, it is also knows as a Linear Binary Classifier.
Perceptron based learning is an algorithm that learns the optimal weight coefficients to multiply with the input signals, in such a way as to decree whether the neuron should activate or not.
In Machine Learning, an algorithm of this type could assign the membership of a sample to a certain class. This problem can be classified in a binary way, where two classes, one positive 1, and one negative -1, can be decreed through an activation function $\phi(z)$ with $z = w_1x_1 + \cdots + w_nx_n$.
Such a function can take a linear combination of input values x and a corresponding weight vector w, where z represents the input of the network.
If a sample is greater than the activation threshold, then the sample will fall into class 1, otherwise if it were less than the threshold it would be classified in -1.
The input of the network $z = wTx$ passing though the activation function of the percpetron, is classified as a binary output in -1 or 1. This algorithm allows to discriminate between two linearly separable classes.
The convergence of perceptron occurs only in cases where the two classes are linearly separable and if the learning rate is low.
It is possible to define a number of steps on the learning dataset (epoch) and a threshold of wrong classifications. Otherwise, the perceptron would always continue to correct the weights. During the learning phase, the output is used to calculate the error in the forecast and to update the weights accordingly.

Feedforward Neural Network
We have two basic types of ANN:

- *Feedforward Neural Network* if the graph is acyclic
- *Recurrent Neural Network* if the graph contains cycles (bckward connections)

In Feedforward NN, also often called multilayer perceptrons (MLPs) because is obtained combining many layer of perceptrons, information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.
There are no feedback connections in which outputs of the model are fed back into itself (like in Recurrent NN).
One of the primary reason that NN are organized into layers is that this structure makes it very simple and efficient to evaluate NN using matrix vector operations.
Let's take a look at the example where we have the first hidden layer's weights W_1 of size $[4 \times 3]$, and the biases for all units would be in the vector b_1, of size $[4 \times 1]$. Here, every single neuron has its weights in a row of W_1, so the matrix vector multiplication $W_1 \cdot x$ evaluates the activations of all neurons in the layer. Similarly, W_2 would be a $[4 \times 4]$ matrix that stores the connections of the second hidden layer, and W_3 a $[1 \times 4]$ matrix for the last (output) layer. the full pass of this 3-layer NN is then simply three matrix multiplications, interwoven with the application of the activation function.
In the code W_1, W_2, W_3, b_1, b_2 are the learnable parameters of the network.
Notice that instead of having a single input column vector, the variable x could hold an entire batch of training data (where each input example would be a column of x) and then all examples would be efficiently evaluated in parallel. Notice that final NN layer usually doesn't have an activation function (e.g., it represents a (real-valued) class score in a classification setting).

Activation Functions
An activation function is just a function that you use to determine if the output of a node of the NN is yes or no. It maps the resulting values in between 0 to 1 or -1 to 1, etc.
The activation functions can be basically divided into 2 types: Linear Activation Functions and Non-linear Activation Functions.
The Linear functions' output will not be confined between any range and it doesn't help with the complexity or various parameters of usual data that is fed to the NN.
The Non-linear Activation Functions are the most used activation functions because they makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.
The Non-linear Activation Functions are mainly divided based on their range or curves:

- *Sigmoid*: the sigmoid non-linearity has the mathematical form$\sigma(x) = 1/(1 + e^{-x})$. It takes a real-valued number and 'squashes' it into range between 0 (large negative numbers) and 1 (large positive numbers). The sigmoid non-linearity has recently fallen out of favor and it is rarely every used because it saturate and kill gradients and the outputs are not zero-centered.
- *Softmax*: is an activation function that takes vectors of real numbers as inputs, and normalizes them into a probability distribution proportional to the exponentials of the input numbers. After applying Softmax, each element will be in the range of 0 to 1, and the elements will add up to 1. this way, they can be interpreted as a probability distribution. (Softamx is used for multi-classification in the Logistic Regression model, whereas Sigmoid is used for binary classification in the Logistic regression model).
- *Tanh*: it squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid, its activations saturate, but unlike the sigmoid neuron its output is zero-centered.

therefore, in practice the tanh non-linearity is always preferred to the sigmoid non-linearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$

- *ReLU*: the Rectified Linear unit computes the function $f(x) = \max(0, x)$. There are several pros and cons to using the ReLUs: (+) it was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid / tanh functions; (+) compared to tanh / sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero; (-) unfortunately, ReLU units can be fragile during training and can 'die'.

- *Leaky ReLU*: are one attempt to fix the 'dying ReLU' probelm. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes $f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$ where α is a small constant.

- *Maxout*: the maxout neuron computes the function $\max(w_1^T x + b_1, w_2^T x + b_2)$. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

Cost Functions
A cost function represent the prediction of output of some arbitrary model for the point x_i with parameters θ. Training the hypothetical model we stated above would be the process of finding the θ that minimizes this sum. Specifically, a cost function is of the form $C(W, B, S_r, E_r)$ where W is our neural network's weights, B is our neural network's biases, S_r is the input of a single training sample, and E_r is the desired output of that training sample. A cost function must satisfy two properties:

- the cost function C must be able to be written as an average

- the cost function C must not be dependent on any activation values of a NN besides the output values

Gradient Descent
Optimization algorithms (in the case of minimization) have on of the following goals:

- find the global minimum of the objective function. This is feasible if the objective function is convex, i.e., any local minimum is a global minimum.

- find the lowest possible value of the objective function within its neighbourhood. That is usually the case if the objective function is not convex as the case in most deep learning problems.

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model.
But in general, how can the gradient descent recognize a local minimum from a global one? We should use specific optimization algorithms like the Stochastic Gradient Descent algorithm, the Batch Gradient Descent algorithm and the Mini-Batch Gradient Descent algorithm. The main differences are:

- the number of data required for the gradient calculation at each iteration

- the time required by the algorithm for updating the parameters: it is learning rate

Backpropagation
It is a way of computing gradients of expressions through recursive application of chain rule and gives us detailed insights into how changing the weights and biases affects the overall behaviour of the network. The goal is to compute the partial derivatives $\delta C/\delta w$ and $\delta C/\delta b$ of the cost function C with respect to any weight w or bias b in the network. An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs:

- sum operation distributes gradients equally ti all its inputs
- max operation routes the gradient to the higher input
- multiply gate takes the input activations, swaps them and multiplies by its gradient

Overfitting
A model trained on more complete data will naturally generalize better. When that is no longer possible, the next best solution is to use techniques like regularization. These place constraints on teh quantity and type of information your model can store.
If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well. The most common ways to reduce overfitting are:

- **Early Stopping**: which means to stop the model as soon as the validation error reaches a minimum. Every model that performs better than all previous models is saved.
- **L1**: shrinking less important coefficients to zero, removing these features entirely from the model (Lasso).
- **L2**: reducing coefficients to close as zero, reducing the importance of less important features (Ridge).
- **DropOut**: every neuron as the probability p (defined by the user) of being temporarily ignored in calculations. Then, in each iteration, we randomly select the neurons that we drop according to the assigned probability. As a result, each time we work with a smaller neural network. This allows the neurons that are training to increase their learning capabilities.

Convolutional Neural Network (CNN)
A CNN is a Deep Learning algorithm, used mainly in images classification, which can take in an input, assign importance (learnable weights and biases) to various aspects / objects and be able to differentiate one from the other.
Convolutional layers consists of multiple features like detecting edges, corners, and multiple textures, making it a special tool for CNN to perform modeling. That layer slides across the image matrix and can detect its all features. This means each convolutional layer in the network can detect more complex features. As the feature expands, we need to expand the dimension of the convolutional layer.

CNN kernels (convolutional filters)
The role of the kernel on multiplying the values of the input image matrix with the corresponding values in the convolution filter. All of the multiplied values are then added together resulting in a single scalar, which is placed in the first position of a result matrix. The kernel is the moved x pixels to the right, where x is denoted stride length and is a parameter of the ConvNet structure. The process of multiplication is then repeated, so that the next value in the result matrix is computed and filled.
This process is then repeated, by first covering an entire row, and then shifting down the columns by the same stride length, until all the entries in the input image have been covered. The output of this process is a matrix with all its entries filled, called the convoluted feature or input feature map. An input image can be convoluted with multiple

convolution kernels at once, creating one output for each kernel.

CNN pooling
The pooling (or downsampling layer) consists of applying operation over regions / patches in the input feature map extracting some representative value for each of the analysed regions / patches.
Two of the most common pooling operations are max - and average - pooling. Max-pooling selects the maximum of the values in the input feature map region of each step and average-pooling the average value of the values in the region. The output in each step is therefore a single scalar, resulting in significant size reduction in output size.
'The reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).'

CNN for text
Instead of images, we have a 1 dimensional array representing the text. Here the architecture of the ConvNets is changed to 1D convolutional-and-pooling operations.
The convolution filter is applied moving a sliding-window from left to right of size K over the sentence, and applying the same kernel to each window in the sequence, i.e., a dot-product between the concatenation of the embedding vectors in a given window and a weight vector u, which is then often followed by a non-linear activation function g.
The pooling operation is used to combine the vectors resulting from different convolution windows into a single l-dimensional vector. This is done again by taking the max or the average value observed in resulting vector of the sentence / document.
This vector is then further down in the network - hence, the idea that ConvNet itself is just a feature extractor - most probably to a full connected layer to perform prediction.

Recurrent Neural Networks (RNN)
'Conceptually they differ from a standard NN as the *standard input in a RNN is a word instead of the entire sample* (they read a sequence of tokens one by one) as in the case of a standard NN. This gives the *flexibility for the network to work with varying lengths of sentences*, something which cannot be achieved in a standard NN due to its fixed structure. It also provides an additional advantages of *sharing features learned across different positions of text* which can not be obtained in a standard NN'.
A RNN treats each word of a sentence as a separate input occurring at time t and uses the activation value at $t-1$ also, as an input in addition to the input at time t.
RNNs effectively have an internal memory that allows the previous inputs to affect the subsequent predictions. It's much easier to predict the next word in a sentence with more accuracy, if you know what the previous words were.
At each step, a recurrent network receives a new input vector (e.g., token embedding) and the previous network state (which, hopefully, encodes all previous information). using this input, the RNN cell computes the new state which it gives as output. This new state now contains information about both current input and the information from previous steps.
RNN reads a text token by token, at each step using a new token embedding and the previous state.
The simplest recurrent network, Vanilla RNN, transformsh_{t-1} and x_t linearly, then applies a non-linearity (most often, the *tanh* function): $h_t = \tanh(h_{t-1}W_h + x_tW_t)$.
Vanilla RNNs suffer from the vanishing and exploding gradients problem. To alleviate this problem, more complex recurrent cells (e.g., Gated Recurrent Unit or Long Term Short Term Memory) perform several operations on the input and use gates.

RNN, a loop network model
Each iteration of the loop takes an input of a vector representing the next word in the sequence with the output activations from the last iteration. these inputs are added or

concatenated together. The output from the last iteration is representation of the next word in the sentence being put through the last layer activation function which converts it to a one hot encoded vector representing a word in the vocabulary. This allows the network to predict a word at the end of a sequence of any arbitrary length.
Once at the end of the sequence of words, the predicted output of the next word could be stored to be used as additional information in the next iteration. Each iteration then has access to the previous predictions.

LSTM and GRU
LSTMs and GRUs take the current input and previous hidden state, then compute the next hidden state. The sigmoid function squashes the values of these vectors between 0 and 1, and by multiplying them elementwise with another vector you define how much of that other vector you want to 'let through'. the differencing points are:

- the GRU has two gates, LSTM has three gates
- GRU does not posses any internal memory, they don't have an output gate that is present in LSTM
- in LSTM the input gate and target gate are coupled by an update gate and in GRU reset gate is applied directly to the previous hidden state. In LSTM the responsibility of reset gate is taken by the two gates i.e., input and target.

LSTM
LSTMs have the repeating module with different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.
The first step in our LSTM is to decide, using the 'forget gate layer' looking at h_{t-1} and x_t, which information to keep in the cell.
The next step pass through the 'input update gate layer' is to decide what new information we're going to store and update the cell state.
Than it is time to update the old cell state, C_{t-1}, into the new cell state C_t.
Finally, we need to decide what we're going to output using a filter (sigmoid and tanh).

GRU
The GRU is a newer generation of RNN and it only has a reset and update gate.
The *update gate* acts similar to the forget and input gate of a LSTM. It decides what information to throw away and what new information to add.
The *reset gate* is another gate is used to decide how much past information to forget.
GRUs has fewer tensor operations, so they are a little speedier to train then LSTMs.

LM evaluation
The most used evaluation of LM is the intrinsic evaluation, computed using the Cross-Entropy or the Perplecity.

- **Cross-Entropy**: it is a measure of the difference between two probability distributions for a given random variable or set of events. So, given an output text $y_{1:M} = (y_1, \ldots, y_M)$ the probability an LM assigns to this text characterizes how well a model 'agrees' with the text. Instead of cross-entropy we can compute the log-likelihood, its negation, as $L(y_{1:M}) = \sum_{t=1}^{M} \log_2 p(y_t|y < t)$.
- **Perplexity**: it is a transformation of cross-entropy $Perplexity(y_{1:M}) = 2^{\frac{-1}{M}L(y_{1:M}}$. The range for perplexity is between 1 and $|V|$ and a better model has higher log-likelihood and lower perplexity.

11 Word Embedding

Word embeddings can capture many different properties of a word and become the de-facto standard to replace feature engineering in NLP tasks.
It was the *first popular embeddgins* method for NLP tasks: essentially the model is a NN with a single hidden layer, and the embeddings are actually the weights of the hidden layer in the NN.
The hypothesis is that **words which frequently appear in similar contexts have similar meaning**.
The main idea is that we need to *put information about word contexts (meanings) into word representation.*
Applications: automatic summarization, machine translation, named entity resolution, sentiment analysis, information retrieval, speech recognition, question answering, music/ video recommendation based on users likes, etc.

Meaning
The **meaning** of something is what express or represents.

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

signifier (*symbol*)⇔signified (*idea* or *thing*). Different symbols can be used to express the same (or similar) idea / thing.

WordNet
An old school solution is to use WordNet. WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are *grouped into sets of cognitive synonyms (synsets)*, each expressing a distinct concept. *Synsets are interlinked by means of conceptual-semantic and lexical relations.*
Problems:

- some synonymous are correct only in some context (clever / apt)
- list 'offensive' synonymous in some synonym sets without any coverage of the connotations or appropriateness of word (clever / cunning)
- missing new meanings of words
- requires human labor to create and adapt
- it is difficult to use to accurately calculate the similarity of words

In traditional NLP, we ragrd words as discrete symbols represented by one-hot vectors ($A = [100\dots], B = [010\dots]$) and the size of the vector depends on the different words within the corpus (vocabulary).
Example: if we are looking for 'clever people' probably we would like to match documents containing 'smart people', but clever = $[0010000\dots]$ while smart = $[000\dots10\dots]$: these two vectors are orthogonal, and there is no natural notion of similarity fro one-hot vectors.
We can try to rely on WordNet's list of synonyms to get similarity, but there is a strong chance of failure (smart seems not to be a synonym for clever in WordNet).

Context
One of the most successful ideas of modern statistical NLP is that **a word's meaning is given by the words that frequently appear close-by (*distributional semantics*)**. When a word w appears in a text, its context is the set of words that appear

nearby (within a fixed-size window). Clearly, *many contexts of w are needed to build up a representation of w.*

$$\text{I saw a } \underbrace{\overbrace{\text{cute grey } \textbf{cat} \text{ playing in}}^{\text{2-sized window for } \textbf{cat}}}_{\text{(contexts for } \textbf{cat}\text{)}} \text{ in the garden}$$

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts, measuring similarity as the vector dot (scalar) product. In a basic approach, we compute the Positive Pointwise Mutual Information (PPMI) as:

$$PPMI(c, w) = \max(0, PMI(w, c))$$

where

$$PMI(w, c) = \log \frac{P(w, c)}{P(w)P(c)} = \log \frac{P(w|c)}{P(w)}$$

The idea of PMI is that *we want to qualify the likelihood of co-occurrence of two observations, taking into account the fact that it might be caused by the frequency of the single observations.*

Word2Vec idea
We have a large corpus (body) of text: a long list of words.
every word in a fixed vocabulary is represented by a **vector**.
Go through each position t in the text, which has a center word c and context (outside) words o.
use the **similarity of the word vectors** for c and o to **calculate the probability** of o given c (or vice versa).
Keep adjusting the word vectors to maximize this probability.

Objective Function: Negative Log-Likelihood
For each position $t = 1, \dots, T$ (T = text length) in a text corpus, Word2Vec predicts context words within a m-sized window given the central word w_t:

$$L(\theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} P(w_{t+j}|w_t; \theta)$$

where θ are all variables to be optimized. The objective (or loss/ cost) function $J(\theta)$ is the average negative log-likelihood:

$$J(\theta) = -\frac{1}{T} l(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m, j \ne 0} P(w_{t+j}|w_t; \theta)$$

Minimizing objective function $\Longleftrightarrow$ Maximizing predictive accuracy.
Note how well the loss agrees with our Likelihood: go over text with sliding window and compute probabilities.

Compute probabilities
For each word w we will have two vectors:

- v_w when it is a central word
- u_w when it is a context word

Then for the central word c and the context word o probability of the context word is:

$$P(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w \in V} exp(u_w^T v_c)}$$

The numerator is a dot product and measure the similarity between o and c: larger dot product = larger probability.
The denominator normalize over the entire vocabulary to get probability distribution.
θ is made up of all the v_t and all the u_t of our text.

Training
To train model, we gradually adjust parameters to minimize a loss. θ represents all the model parameters, in one long vector (all the v_t and all the u_t for all words in the vocabulary). These vectors are learned by optimizing the training objective via gradient descent (with some learning rate α): $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$.
We have a cost function $J(\theta)$ we want to minimize - Gradient Descent is an algorithm to minimize $J(\theta)$ - for current value of θ, calculate gradient of $J(\theta)$, then take small step in direction of negative gradient, and repeat.
For a single parameter we have: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\delta}{\delta \theta_j^{old}} J(\theta)$.

Word2Vec Parameter Learning
By making an update to minimize $J_{t,h}(\theta)$, we force the parameters to *increase similarity* (dot product) of the central word v_w and context word u_w and, at the same time, to *decrease similarity* between v_w and all other words in the vocabulary.
It sound strange, but *we want to decrease similarity between v_w and all the other words*, even if some of them are also valid context words, because *since we make updates for each context word* (and for all central words in your text), on average over all updates our vectors will learn the distribution of the possible contexts.

Model variants

- **Skip-grams** (SG): predict context words (position independent) given center word. Skip-Gram with negative sampling is the most popular approach.
- **Continuous Bag of Words** (CBOW): predict center word from (bag of) context words

Negative sampling
This is used also for the classical Word2Vec, because for each pair of a central word and its context word, we had to update all vectors for context words and this is highly inefficient (for each step, the time needed to make an update is proportional to the vocabulary size).
So *we don't consider all context vectors* in the vocabulary at each step, we decrease similarity between v_w and context vectors no for all words, but *only with a subset of K 'negative'* examples: since we have a large corpus, on average over all updates we will update each vector sufficient number of times, and the vectors will still be able to learn the relationships between words quite well.

Parameters
The size of the sliding window has a strong effect on the resulting vector similarities.
For example, larger windows tend to produce more topical similarities (i.e., *dog, bark* and *leash* will be grouped together, as well as *walked, run* and *walking*), while smaller windows tend to produce more functional and syntactic similarities (i.e., *Poodle, Pitbull, Rottweiler*, or *walking, running, approaching*).
As always, the choice of hyperparameters usually depends on the task at hand.

Count based vs direct prediction
Count based
pros:

- fast training

cons:

- primarily used to capture word similarity
- disproportionate importance given to large counts

Direct prediction
pros:

- can capture complex patterns beyond word similarity

cons:

- scales with corpus size

GloVe
The GloVe model is a combination of count-based methods and prediction methods (e.g., Word2Vec). GloVe ('Global vectors') uses global information from corpus to learn vectors. It uses co-occurrence counts to construct the loss function (the count-based method uses co-occurrence counts to measure the *association* between word w and context $c : N(w, c)$):

$$J(\theta) = \sum_{w,c \in V} f(X_{w,c}) \cdot (u_c v_w + b_c + \bar{b}_w - \log X_{w,c})^2$$

- V is the size of the vocabulary
- X denotes the word co-occurrence matrix (so $X_{w,c}$ is the number of times word c occurs ion the context of word w
- the weighting f is given by $f(x) = (x/x_{max})^{\alpha}$ if $x < x_{max}$ and 1 otherwise
- $x_{max} = 100$ and $\alpha = 0.75$ (determined empirically)
- u_c, v_w are the two layers of word vectors
- $b_c, \bar{b}_w$ are bias terms

GloVe *controls the influence* of are and frequent words ($f(X_{w,c})$: loss for each pair (w, c) is weighted in a way that:

- rare events are penalized ($0 \leq x < 1$)
- very frequent event are not over-weighted ($x = 1$)

note that the product is only over pairs w, c for which $X_{i,j}$ is non-zero. This means that GloVe (in contrast to Word2Vec with negative sampling) trains only 'positive samples' and also that we don't have to worry about the logarithm of zero.
This essentially just weighted matrix factorisation with bias terms. In the implementation, the $X_{i,j}$ are not raw co-occurrence counts, but rather the accumulated inverse distance between the two words.
GloVe results to be fast in training, scalable to huge corpora and get good performance even with small corpus and small vectors.

FastText
The concept of subword-level embeddings is based on the skip-gram model, but where each word is represented as a bag of character n-grams.
A vector representation is associated to each character n-gram, and words are represented as the sum of these representations.
This allows the model to compute word representations for words that did not appear in the training data.
Each word w is represented as a bag of character n-gram, plus a special boundary symbols ¿ and ¡ at the beginning and end of words, plus the word w itself in the set of its

n-grams.
By using a distinct vector representation for each word, the skipgram model ignores the internal structure of words. FastText propose a different scoring function s, in order to take into account this information.
Each word w is represented as a bag of character n-gram. FastText add special boundary symbols ¡ and ¿ at the beginning and end of words, allowing to distinguish prefixes and suffixes from other character sequences. FastText also include the word w itself in the set of its n-grams, to learn a representation for each word (in addition to character n-grams). Taking the word where and $n = 3$ as an example it will be represented by the character n-grams: ¡wh, whe, her, ere, re¿ and the special sequence ¡where¿.
note that the sequence ¡her¿, corresponding to the word her is different from the *tri*-gram her from the word where. In practice, FastText extract all the n-grams for n greater or equal to 3 and smaller or equal to 6. This is a very simple approach, and different sets of n-grams could be considered, for example taking all prefixes and suffixes.

FastText scoring function
In Word2Vec, from which FastText model is derived, given a vocabulary of size W where a word is identified by its index $w \in (1, \ldots, W)$, the goal is to learn a vectorial representation for each word w and word representations are trained to predict well words that appear in its context. it is used a softmax function to define the probability of a context word;

$$P(w_c|w_t) = \frac{e^{w_t, w_c}}{\sum_{j=1}^{W} s(w_t, j)}$$

Then the score can be computed as the scalar product between word and context vector

$$s(w_t, w_c) = u_{w_t}^T v_{wc}$$