https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/?utm\_source=blog&utm\_medium=top\_5\_sentence\_embedding

<https://www.analyticsvidhya.com/blog/2017/01/ultimate-guide-to-understand-implement-natural-language-processing-codes-in-python/?utm_source=blog&utm_medium=top_5_sentence_embedding>

<https://www.analyticsvidhya.com/blog/2020/03/pretrained-word-embeddings-nlp/?utm_source=blog&utm_medium=top_5_sentence_embedding>

<https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/?utm_source=blog&utm_medium=top_5_sentence_embedding>

# Context

As machine learning algorithms inputs are numbers and not words, we have to transform our words features into vectors of numbers.

* This transformation is called ***words embedding***.

But before word embedidng we have to ***clean our features*** :

* Remove stop words
* Use techniques such Stemming or lemmatization.

For books recommendations, there are different kinds of information scrapped from internet. For example authors and title don't encode the same kind of data.

* So we can use ***differents techniques of encoding*** for all information.

For example, we could use a simple n-gram representation for words inside a grouping of the columns author, author gender and books category :

* From thos artificial "sentences", we will create all the possible n-grams
* We trained a simple network: embedding + 2\*linear layers
* It outputs a probability for each words of this vocabulary

After having embedded all words, we will have to perform an average of all numerical vectors, for example all vectors representing all words inside a title. The aim here is to ***reduce the dimension of the embedding***, in order to put it efficiently as input into a recommendation algorithm.

* We should give ***different weight on different informations***.

# Words cleaning

When we are running a search, we want to find relevant results not only for the exact expression we typed on the search bar, but also for the other possible forms of the words we used.

For example, it’s very likely we will want to see results containing the form “skirt” if we have typed “skirts” in the search bar.

This can be achieved through two possible methods: stemming and lemmatization. The aim of both processes is the same: reducing the inflectional forms of each word into a common base or root. However, these two methods are not exactly the same.

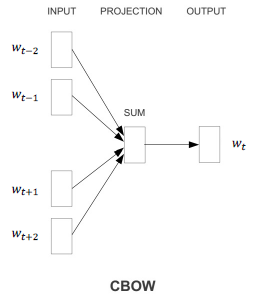
* ***Stemming*** algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found in an inflected word.
* ***Lemmatization***, on the other hand, takes into consideration the morphological analysis of the words. To do so, it is necessary to have detailed dictionaries which the algorithm can look through to link the form back to its lemma.

# Word embedding

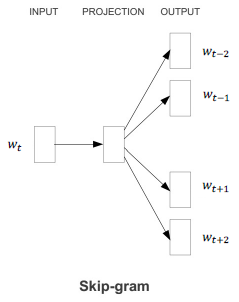
## Word2Vec

This algorithm uses a shallow neural network to learn word vectors so that each word of a given corpus is good at predicting its own contexts (Skip-Gram) or vice versa (CBOW) :

* ***CBOW (Word2Vec)*** :
  + From sentence, we created all the possible contexts with 2 words before and after the target word
  + We trained a simple network: embedding + 2\*linear layers
  + It outputs a probability for each words of vocabulary



* ***Skip-gram***: it's the invers, from a word, it will output a possible context



Remark : Gensim library propose a more generic implementation, without specifying any target words.

Empirical results using these methods seem to show that this approach is successful at learning the meaning of words. In fact, the resulting embedding space seems to have directions of semantic and syntactic meaning that can be exposed through simple operations on word vectors.

For example, we can retrieve words that have :

* semantic relations such as Country-Capital :

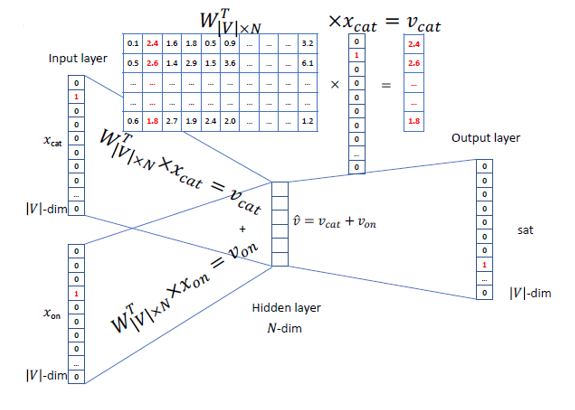
vector(Paris) — vector(France) + vector(Morocco) ~ vector(Rabat)

* syntactic relationships such as the Singular-Plural:

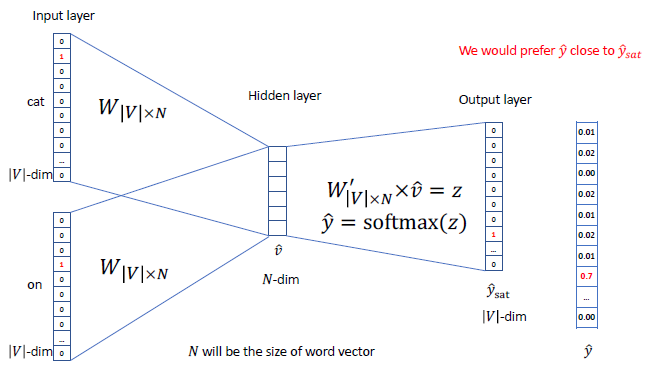
vector(Kings) - vector(King) + vector(Person) ~ vector(People)

Training of this model :

The matrix W is learned by the network: each columns represents a word.



Then the predicted word is calculated :

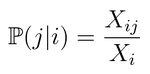


## GloVe

Assumption :

This algorithm is based on the observation that word relationships can be recovered from the co-occurrence statistics of any (large enough) corpus.

Let Xij be the number of times words i and j “co-occur” in a given corpus, and Xi be the number of times the word i occurred in general. Provided the corpus is large enough, we can consider that the probability that a word i occurs next to the word j is :

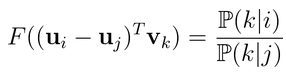


We can discover relationships between words by looking at ratios of these probabilities. For instance, by computing the following ration for lots of words j :

⇒ "ice" is to "solid"….

⇒ …what "steam" is to "gas"

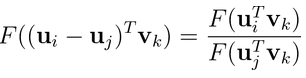
Model built from this assumption :

 Here **u** denotes the word vector

and **v** denotes the context word vectors.

In this model we must have a symmetry between u and v, it then implies the following properties :

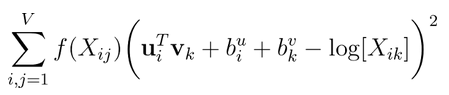
* F should be a homomorphism between the groups (R,+) and (R+, ×):

 ⇔ 

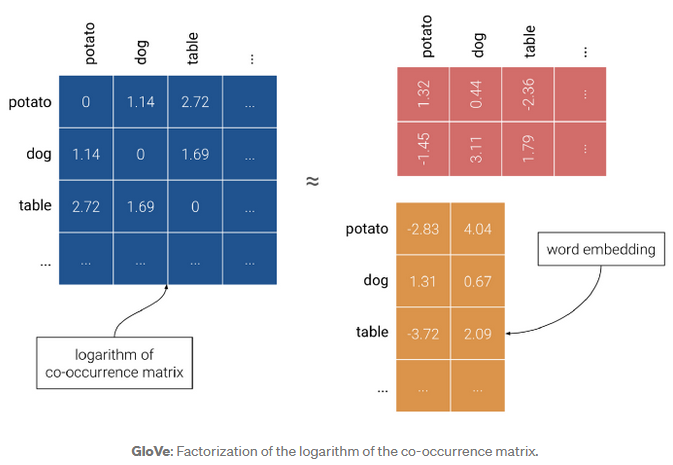
* Since the last logarithm is independent from k, it can therefore be absorbed in a bias bi. Adding a second bias bk completes the symmetry of the model:



GloVe seeks to verify this equality for every possible pair of words i & j by minimizing the weighted least-squares objective:



Remark : If we ignore the biases in the previous training objective, we see that the algorithm tries to make the dot product of the word and context vectors as close as possible to the logarithm of the words’ co-occurrence. We can interpret this as GloVe implicitly factorizing the logarithm of the co-occurrence matrix X.

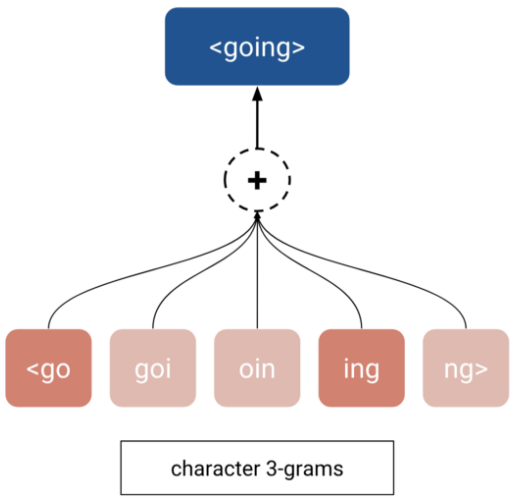


## fastText

Remark : The name “FastText” actually stands for two different -but related- algorithms. On one hand there is the embedding method we are interested in. But on the other hand there is also the “FastText” classification algorithm that uses optimization tricks for efficiently classifying text. This classifier may or may not use FastText embeddings depending on the user’s choice.

An issue with GloVe and Word2vec is that they only learn embeddings for words of the vocabulary. As a result, Out-of-vocabulary (OOV) words don’t have a vector representation and are usually either represented by a vector of zeros, by the average of all word vectors or simply ignored from the sentence.

FastText (embedding technique) learns embeddings of character n-grams, i.e. sequences of n successive characters. After the training, the embedding of each word is computed as the sum of its constituent n-grams.



In the example above, we suppose that the word going is out-of-vocabulary. Using FastText with n=3, the embedding for going can be obtained as the sum of its character 3-grams. Among all of these n-grams, those of <go and ing might inform us that the word going is a continuous action related to the verb go without the word going actually occurring in the training data.

SVD :

Neural networks are not necessarily required to accomplished this task. In fact, Word2vec is barely a neural network since it has no hidden layers and no non-linearities but still achieves good results. Then, GloVe implicitly factorizes a co-occurrence matrix and achieves even better results. So how simple can a model be and still learn good word embeddings ?

It turns out, that we can directly factorize a co-occurrence matrix and get good word embeddings. In practice, we can follow these simple steps:

* Compute the probability of occurrence of each word p(x)
* Compute the probability of co-occurrence of each couple of words p(x,y)
* Divide each co-occurrence probability by each word’s probability p(x,y)/p(x)p(y)
* Apply the logarithm to the ratio: log[p(x,y)/p(x)p(y)].

The only thing that is left to do is to decompose that matrix using SVD to get fresh word embeddings.

# Sentence embedding

## Context

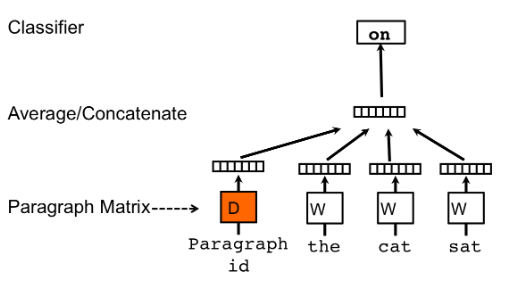
Sentence embedding techniques represent entire sentences and their semantic information as vectors. This helps the machine in understanding the context, intention, and other nuances in the entire text.

## Doc2Vec

It is an unsupervised algorithm and adds on to the Word2Vec model by introducing another ‘paragraph vector’. Also, there are 2 ways to add the paragraph vector to the model.

* ***PVDM***(Distributed Memory version of Paragraph Vector) :

We assign a paragraph vector sentence while sharing word vectors among all sentences. It is an extension of the Continuous Bag-of-Word type of Word2Vec where we predict the next word given a set of words. It is just that in PVDM, we predict the next sentence given a set of sentences.



* ***PVDOBW***( Distributed Bag of Words version of Paragraph Vector) :

Just lime PVDM, PVDOBW is another extension, this time of the Skip-gram type.

## SentenceBERT

### Principle

You can either use these models to extract high quality language features from your text data, or you can fine-tune these models on a specific task (classification, entity recognition, question answering, etc.) with your own data to produce state of the art predictions.

At the heart of this BERT-based model, there are 4 key concepts:

* Attention
* Transformers
  + transformers provides a number of classes for applying BERT to different tasks (token classification, text classification, …). Here, we’re using the basic BertModel which has no specific output task–it’s a good choice for using BERT just to extract embeddings.
* BERT
* Siamese Network

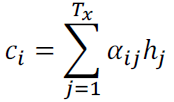
The BERT base uncased model contains 12 layers.

***Encoder :***

It's a bi-directional RNN.

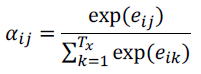
***Decoder :***

It's always a RNN, but this time its input is not only a fixed vector outputed by previous encoder. It takes into account of all the previously internal state of the encoder :

 where  with learned coefficients 

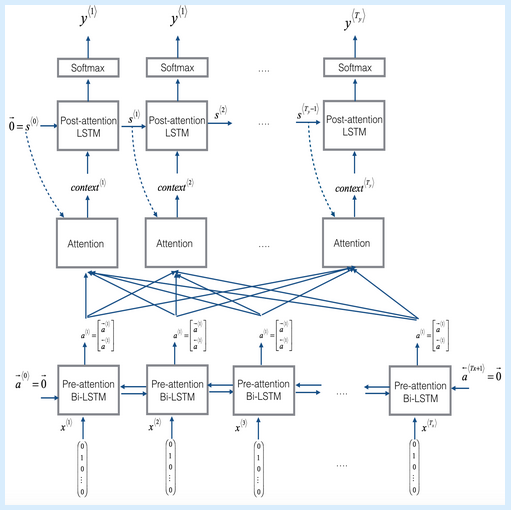
hj are the hidden states of encoder

The learned coefficients measure to which extend the encoder hidden states and the current entry i of decoder are correlated :

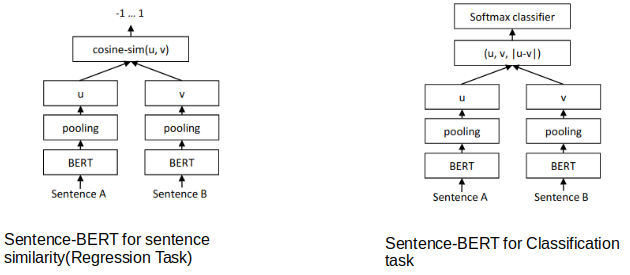
 with 

In such a way, the probabilities handled in decoder part take into account of the whole original sentence.

***It's a bidirectional RNN :***



Sentence-BERT uses a Siamese network like architecture to provide 2 sentences as an input. These 2 sentences are then passed to BERT models and a pooling layer to generate their embeddings :



Remark : pre-trained model are available, released by Google that ran for many, many hours on Wikipedia and Book Corpus, a dataset containing +10,000 books of different genres.

### Inputs

Because BERT is a pretrained model that expects input data in a specific format, we will need:

* A special token, ***[SEP]***, to mark the end of a sentence, or the separation between two sentences
* A special token, ***[CLS]***, at the beginning of our text. This token is used for classification tasks, but BERT expects it no matter what your application is.
* Tokens that conform with the fixed vocabulary used in BERT
* The ***Token ID***s for the tokens, from BERT’s tokenizer
* ***Mask IDs*** to indicate which elements in the sequence are tokens and which are padding elements
* ***Segment ID****s* used to distinguish different sentences
* ***Positional Embeddings*** used to show token position within the sequence

Tokens :

For example, the following sentence will be tokenized as :

* Sentence :
  + "Here is the sentence I want embeddings for."
* Tokenized version :
  + ['[CLS]', 'here', 'is', 'the', 'sentence', 'i', 'want', 'em', '##bed', '##ding', '##s', 'for', '.', '[SEP]']

To tokenize a word under this model, the tokenizer first checks if the whole word is in the vocabulary. If not, it tries to break the word into the largest possible subwords contained in the vocabulary, and as a last resort will decompose the word into individual characters. Note that because of this, we can always represent a word as, at the very least, the collection of its individual characters.

Segments :

BERT is trained on and expects sentence pairs, using 1s and 0s to distinguish between the two sentences. That is, for each token in “tokenized\_text,” we must specify which sentence it belongs to: sentence 0 (a series of 0s) or sentence 1 (a series of 1s). For our purposes, single-sentence inputs only require a series of 1s, so we will create a vector of 1s for each token in our input sentence.

### Outputs

The full set of hidden states for this model, stored in the object hidden\_states, is a little dizzying. This object has four dimensions, in the following order:

* The layer number (13 layers: it’s 13 because the first element is the input embeddings, the rest is the outputs of each of BERT’s 12 layers.)
* The batch number (1 sentence)
* The word / token number (22 tokens in our sentence)
* The hidden unit / feature number (768 features)
* That’s 219,648 unique values just to represent our one sentence !

Now, what do we do with these hidden states? We would like to get individual vectors for each of our tokens, or perhaps a single vector representation of the whole sentence, but for each token of our input we have 13 separate vectors each of length 768.

In order to get the individual vectors we will need to combine some of the layer vectors…but which layer or combination of layers provides the best representation?

Unfortunately, there’s no single easy answer… Let’s try a couple reasonable approaches, though:

* Word vectors
  + First, let’s ***concatenate the last four layers***, giving us a single word vector per token. Each vector will have length 4 x 768 = 3,072.
  + As an alternative method, let’s try creating the word vectors by ***summing together the last four layers***.
* Sentence vectors
  + A simple approach is to ***average the second to last hiden layer*** of each token producing a single 768 length vector.

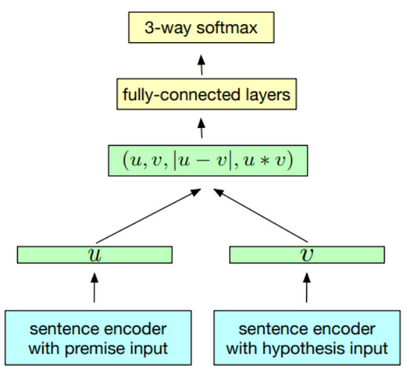
## InferSent

InferSent is a supervised sentence embedding technique.

The main feature of this model is that it is trained on Natural language Inference(NLI) data, more specifically, the SNLI (Stanford Natural Language Inference) dataset. It consists of 570k human-generated English sentence pairs, manually labeled with one of the three categories – entailment, contradiction, or neutral.

Just like SentenceBERT, we take a pair of sentences and encode them to generate the actual sentence embeddings. Then, extract the relations between these embeddings using:

* concatenation
* element-wise product
* absolute element-wise difference



Another important feature is that InferSent uses GloVe vectors for pre-trained word embeddings. A more recent version of InferSent, known as InferSent2 uses fastText.

## Universal Sentence Encoder

One of the most well-performing sentence embedding techniques right now is the Universal Sentence Encoder.

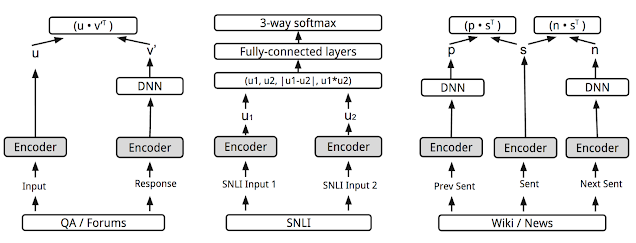
The key feature here is that we can use it for Multi-task learning. This means that the sentence embeddings we generate can be used for multiple tasks like sentiment analysis, text classification, sentence similarity, etc, and the results of these asks are then fed back to the model to get even better sentence vectors that before.

The most interesting part is that this encoder is based on two encoder models and we can use either of the two:

* Transformer
* Deep Averaging Network(DAN)

Both of these models are capable of taking a word or a sentence as input and generating embeddings for the same. The following is the basic flow:

* Tokenize the sentences after converting them to lowercase
* Depending on the type of encoder, the sentence gets converted to a 512-dimensional vector
  + If we use the transformer, it is similar to the encoder module of the transformer architecture and uses the self-attention mechanism.
  + The DAN option computes the unigram and bigram embeddings first and then averages them to get a single embedding. This is then passed to a deep neural network to get a final sentence embedding of 512 dimensions.
* These sentence embeddings are then used for various unsupervised and supervised tasks like Skipthoughts, NLI, etc. The trained model is then again reused to generate a new 512 dimension sentence embedding.

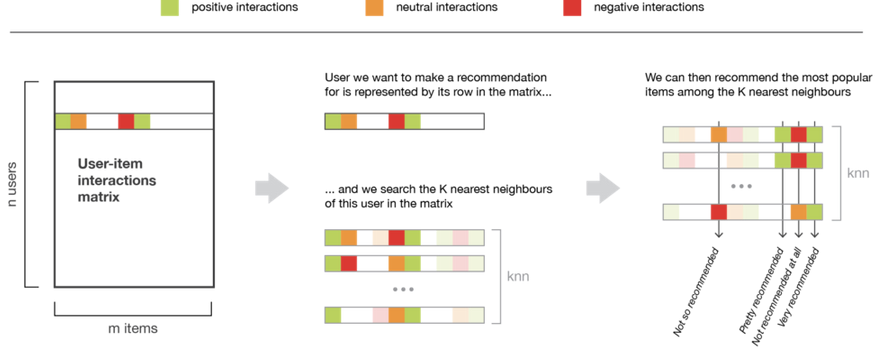


# Recommendation methods

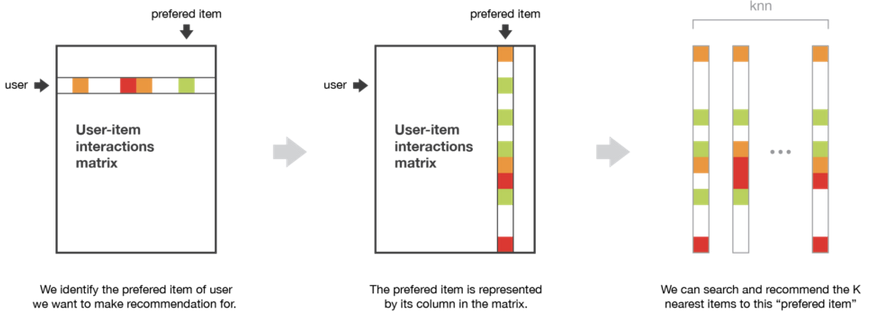
## Context

There are mainly two different methods for recommendations :

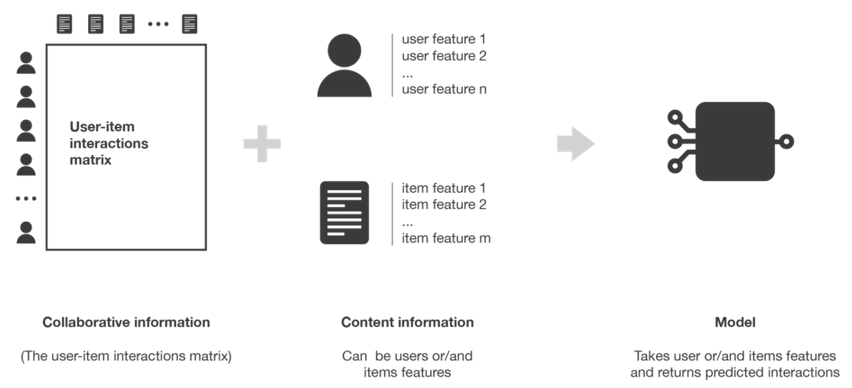
* **Collaborative filtering methods :** based solely on the past interactions recorded between users and items in order to produce new recommendations. These interactions are stored in the so-called “user-item interactions matrix” (a matrix with only ratings values) :
  + Collaborative User-User :



* + Collaborative Item-Item :



* **Content based methods :** unlike collaborative methods that only rely on the user-item interactions, content based approaches use additional information about users and/or items.



Content based methods can also be ***neither user nor item centred***: both informations about user and item can be used for our models, for example by stacking the two features vectors and making them go through a neural network architecture.

* Highest bias (highest personalization)

In case where we don't have user's features, a way of even so personalize the recommendation is to allow the used to modify the rating of recommended book.

## Neighbors

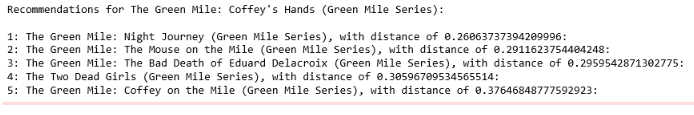
* The following methodology can be applied for example in the context of collaborative Item-Item / User-User.

***Collaborative Item-Item***

The input matrix has :

* for row the books title
* for columns the users identifiers
* and for values, the total ratings of each book

The ouput will be : books with few ratings distance are recommended.



***Collaborative User-User***

We coukd also transpose the input matrix so that the algorithm find the nearest neighbors of a user, and then we could propose to this user, books with high rating read by its neigbors.

## LSTM

* The following methodology can be applied for example in the context of content based methods as books and users features can be used.

**LSTM network**

* Books / users features are inputs, and outputs are ratings
* We trained a simple network: embedding + LSTM layer + linear layer
* It outputs a rating for the proposed Book / user features

## Features matrix decomposition

### Principles

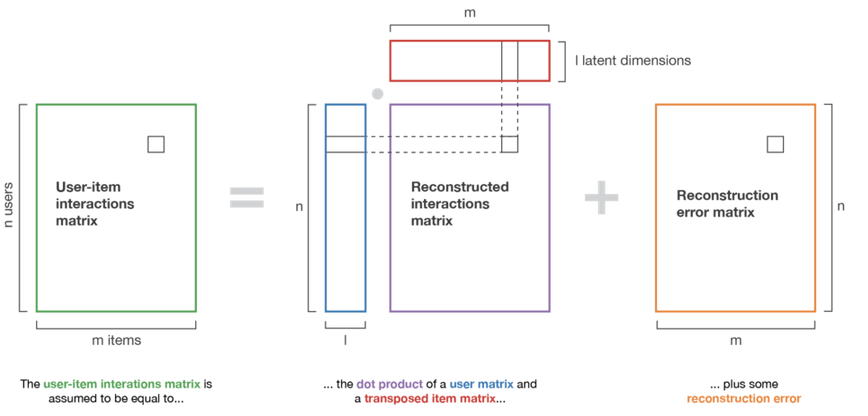
* The following methodology can be applied for example in the context of :
  + Content based methods as books and users features can be used
  + Collaborative based methods as only ratings matrix can be used

Matrix handled :

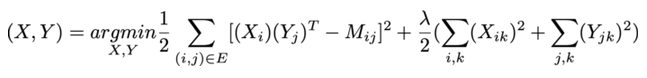
* Matrix M : only ratings (rows are users and columns are items)
* Vector Xi : coefficients to be learned for user i
* Matrix Yj : coefficients to be learned for books j in context of collaborative methods

OR coefficients proposed thanks to our scrapped data

Visualisation :



Underlying equation :



* A prediction for product k is then performed by multiplying the Xi by (Yk)T.

### FastFM

<https://github.com/ibayer/fastFM>

All performance critical code has been written in C and wrapped with Cython.

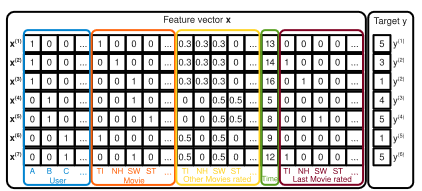
***fastFM provides :***

* Optimization :
  + stochastic gradient descent (SGD)
  + and coordinate descent (CD) optimization routines
* Bayesian inference
  + Markov Chain Monte Carlo (MCMC) for Bayesian inference.

The solvers can be used for regression, classification and ranking problems.

#### Inputs

Proposed input in article :



Assume we have the transaction data of a movie review system.

The system records

* which user u ∈ U rates
  + U = {Alice (A), Bob (B), Charlie (C), . . .}
* a movie (item) I ∈ I
  + I={Titanic (TI), Notting Hill (NH), Star Wars (SW), Star Trek (ST), . . .}
* at a certain time t ∈ R
* with a rating r ∈ {1,2,3,4,5}

The feature vectors also contain indicator variables (yellow) for all the other movies the user has ever rated. For each user, the variables are normalized suchthat they sum up to 1. E.g. Alice has rated Titanic, Notting Hill and Star Wars.

And finally the vector contains information of the last movie (brown) the user has rated before (s)he rated the activeone – e.g. for x(2), Alice rated Titanic before she rated Notting Hill.

But in article, there are several other inputs proposed, for example the following one which takes into account for example books features :



#### Differents models

For the 3 following models, regression and classification are proposed. The input dataset is classical :



A ranking form is also proposed. In that case input data set is the following :

 with x(A) ranked higher than x(B)

*Remark: input for the implementation is the sparse matrix X and several pairs of samples (the first element of the pair is ranked higher).*

Optimisation SGD

Optimisation ALS (Alternating Least-Squares) / CD (Coordinate descent)

This allows us to derive a least-squares learning algorithm that it-eratively solves a least-squares problem per model parameter and updates each model parameter with the optimal (local) solution.

Optimisation MCMC (Markov Chain Monte Carlo)

Both ALS and SGD learn the best parameters which are used for a point estimate of . MCMC is a Bayesian inference technique that generates the distribution of by sampling.

The difference is that MCMC samples from the posterior distribution, while ALS uses the expected value.

#### Parameters

The algorithm equation is the following :

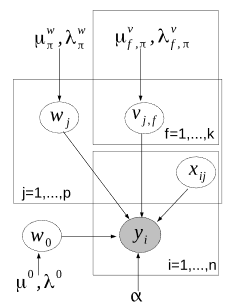
 , where  (n vectors of size p)

The first part of the FM model contains the unary interactions of each input variable xj with the target—exactly as in a linear regression model.

The second part with the two nested sums contains all pairwise interactions of input variables, that is, xjxj′.The important difference to standard polynomial regression is that the effect of the interaction is not modeled by an independent parameter wj,j but with a factorized parametrization :



So the parameters are :



## Ordinal regression

### Principles

* The following methodology can be applied for example in the context of :
  + Content based methods as books and users features can be used
  + Collaborative based methods as only ratings matrix can be used

These situations detach from the scope of traditional classification and regression methods as far as the subjectiveness of the predicted target (e.g. rating movies or tasteful dishes) obliterates the true scale of a target variable and leaves the relative order of values as the only purpose.

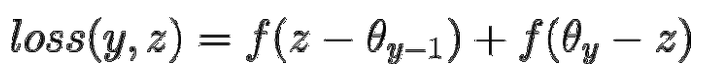
Therefore, if we consider a target within a scale of K ordered numbers, we can divide it into a set of K disjoint segments by thresholds K-1 thresholds (θ₁ < θ₂ < … < θK−1) to represent progressive response levels.

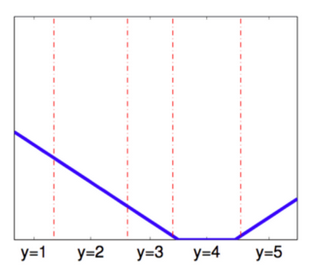
Ordinal regression :

Based on the generalization described above, we can apply statistical constructions to estimate the optimal threshold in order to reduce prediction errors (minimize a loss function) in ordinal regression models.

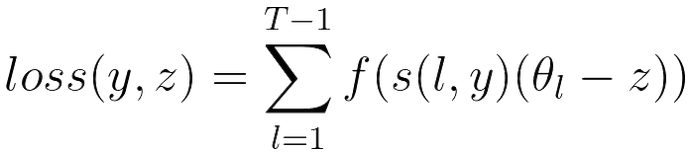
Two possible implementations that allow fitting models based on the sum of loss penalties in all segments of a target are (on the following, z is the predicted response) :

* the **immediate-threshold** : we can set a loss function that continuously increases in both directions outside the range of the correct prediction (when a threshold is crossed).





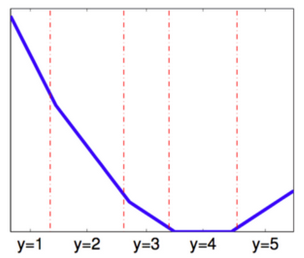
* and the **all-threshold** methods : As observed in the graphic above, a limitation of the immediate-threshold loss is that it ignores how many thresholds are crossed relative to the origin of the correct label. A more complete construction that bounds multiple penalties according to the number of threshold violations is the all-threshold method.



where l is the reference range for loss computation

and s(l, y) ≥ 0 if l > y

s(l, y) ≤ 0 otherwise



Therefore, more than reducing the distance of the predicted value from the correct label, this method drives the model to minimize the number of thresholds crossed from the correct response.

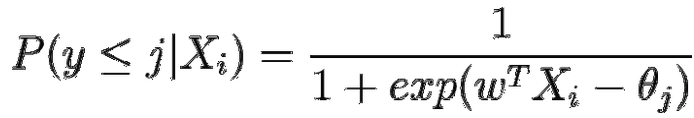
Ordinal classification :

Ordinal logistic regression is an extension of simple logistic regression model. In simple logistic regression, the dependent variable is categorical and follows a Bernoulli distribution.

In simple logistic regression, log of odds that an event occurs is modeled as a linear combination of the independent variables. But, the above approach of modeling ignores the ordering of the categorical dependent variable.

Whereas, in ordinal logistic regression the dependent variable is ordinal i.e. there is an explicit ordering in the categories.

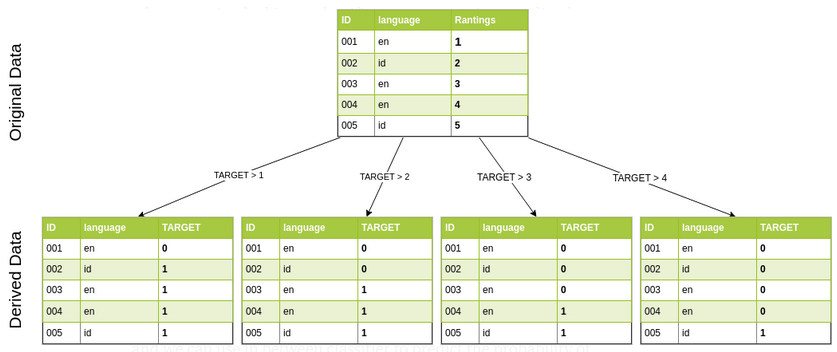
logit( P(Y < yi | X, θ) ) = log ( ) = ai + β1.x1 + … + βn.xn

* 

The Python library MORD (Multi-class classifier for ORDinal regression) allow to implement this algorithm.

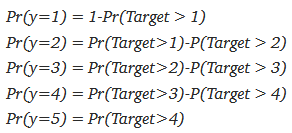
### Trick with classifiers

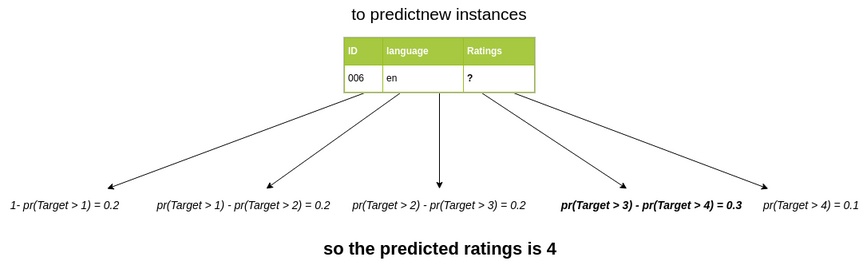
We can transform a k classifier into k-1 binary classifiers :



* Each binary classifier evaluate an ordered value.

Then once those k-1 binary classifiers are trained, we can deduce the following probabilties (for k = 5 for example) :





## Ranking

* The following methodology can be applied for example in the context of content based methods as books and users features can be used.

It differs from the more common cases classification and regression in that, instead of predicting the outcome of one data point, it takes a set of data points, a query, and ranks the data points.

When ranking with ***XGBoost*** (eXtreme Gradient Boosting) there are three objective-functions which are different methods of finding the rank of a set of items, and each has its own strengths and weaknesses.

* ***Pointwise***: One instance of the set is considered at a time, use any kind of classifier or regressor to predict how relevant it is in the current query. Use each points predicted relevance to order the set.
* ***Pairwise***: A pair of instances is chosen and the order of those two is predicted. Repeat this for each pair of the query to find the final order of the entire query.
* ***Listwise***: Many or all instances are considered at once. Try to find the optimal order.

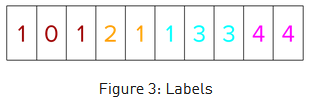
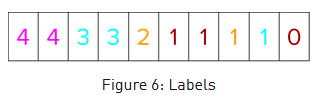
When ranking, the aim is not to accurately predict the final order. Instead you are essentially trying to find what data points are relevant in the current query, and which are not. The target for Learning to Rank is a relevance score, which tells you how relevant the data point is in the current group.

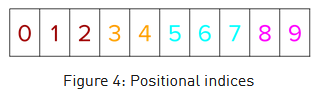
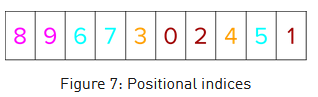
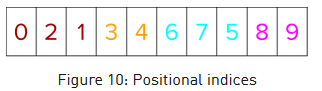
***Algorithm inputs :***

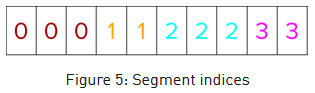
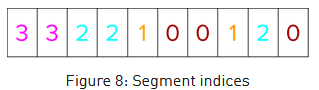
When performing Learning to Rank we must pass another key word argument to the model, our group. This arguments takes an array of the sizes of the groups in the training data.

If you have two groups in your training data, one with 10 instances and one with 7, the groups array should simply be [10, 7].

Relevance indexes : ⇒ Labels are then sorted : ⇒ Segments indices are sorted :

* Segments indices delineate every group