# Project presentation

## First idea

***Existing applications***

There are lots of books recommendations applications which also offer the ability to create its own personal library.

BUT, if we want :

* a french site,
* with lots of international referenced books (paper and electronic),
* which allows to give my opinion,
* which allows to export my personal created library
* and which gives me personal recommendations…

…there only one site left: Babelio.

However, its recommendation system seems to be not appreciated by users, because for example, it seems attaching too much importance on the books read (prior to the rating given !). But others examples of potential incorrect algorithm entries are given…

***What is already existing in term of database ?***

Some sites like TiteLive own a huge books data base…but they sell their database !

The BNF DataBase is free, but only deals with french books…

So no good idea except web scrapping on web site such as Google Books…

On books dedicated web site (Amazon, Good Reads…), no real solutions because often they no more provide their API.

But database can be found in machine learning dedicated web site, as for example Book Crossing, Kaggle (results of Good Reads web site scrapping for example…) or some GitHub.

The difficulty is to found data base with users ratings provided.

## Context

Features are information that has been retrieved about the books or the user. But before any features treatment, we have to ***clean our features*** :

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

***Book's features treatment :***

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***.

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

* So we can use ***different techniques of embedding*** for all information.

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 information***.

***Additional users features :***

We mainly have book's features, but we could also use users features in order to personalize recommendations. In our case no user features were available, but in that case a possible solution could be to allow users to modify ratings and to save its changes thanks to a kind of local version of our model (tensorflow allow such kind of treatment).

***Recommendations :***

Then we will have to use recommendation algorithms. Lots of algorithms are available, but we mainly used ordinal regression.

# Data for the project

## Context

Two main database have been used: *Book Crossing* and *Good Books*. Those database also contain user's ratings, which will be used for our recommendation algorithms training.

Book Crossing implies about 300 000 books, and Good Books 10 000.

The first 3rd of Book Crossing data base has been completed. It represents a number of 114 528 lines.

The following columns were already existing :

* ISBN
* Book-Title
* Book-Author
* Year-Of-Publication
* Publisher
* Image-URL-S / Image-URL-M / Image-URL-L

It has been decided to complete those information with the following features :

* isbn\_13 / OtherID
* Category
* book\_description
* book\_language
* number\_of\_pages
* author\_genres
* books\_in\_series
* average\_rating
* awards

All those information have been saved in a csv file.

## Scrapping

### Two web sites

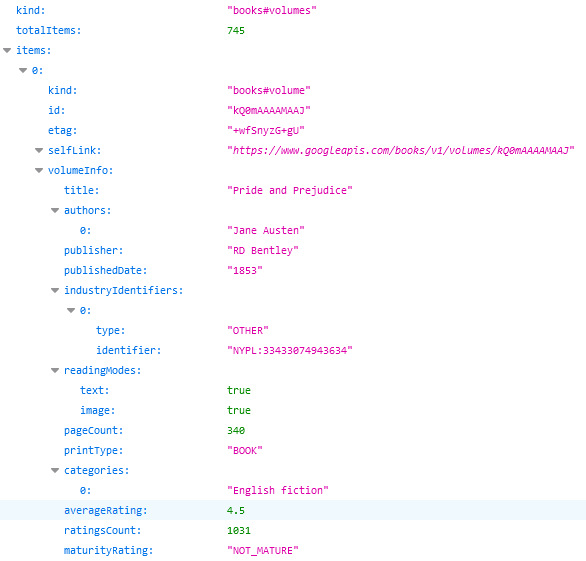
***Google books API :***

First the Google Books web site has been used as it provides an easy way to access to book's information thaks to public google API.

Indeed, in order to access to a book descriptive sheet, two methods have been used :

* https://www.googleapis.com/books/v1/volumes?q=isbn:***055321215X***
* [https://www.googleapis.com/books/v1/volumes?q=title:***Beloved%20(Plume%20Contemporary%20Fiction)***](https://www.googleapis.com/books/v1/volumes?q=title:Beloved%20(Plume%20Contemporary%20Fiction))

Then from those URL, a json file is returned and can be easily parsed :



But on Google books, there are some issues :

* Some information are not reliable, as for example the language information.
* Some information are not available, as for example the author gender.
* When the search is performed with title, there are always lots of books description sheets results, but sometimes none of them are even related to the title !

The last issue has been solved by an additional comparison of author in the first obtained books description sheet.

***Good Reads :***

By searching database on kaggle, we have noticed that the author gender and other information we wanted appeared in database scrapped from Good Reads web site.

So we have performed scrapping on Good Reads web site, but this time no API was available. So the scraping has been performed by directly parsing information on a web page.

To kinds of Python libraries have been used :

* *urlopen from urllib.request* (mainly to save web page during our tests session)
* *BeautifulSoup from bs4* (final solution chosen)

By analysing the web page structure, we retrieved key words which introduced information we need. For example, for the book's gender, the key words to search were "actionLinkLite bookPageGenreLink" :

if (soup.find("a", class\_="actionLinkLite bookPageGenreLink")):

catSearched = soup.find\_all("a", class\_="actionLinkLite bookPageGenreLink")[0].text

Information on Good Reads seem to be more reliable, but also some issues appear :

* Some information like the author gender are localized on another web page whose link was in the current books description sheets.

So it took lots of time to get information thanks to Good Reads, because of this regular jump from one web page to another.

* When current book was part of a serie, there were URL links towards other books of the same series. But those links were specific format for Good Reads, and not directly usable.

***Both web sites used :***

The final solution uses the both web sites, but in a sequential way :

* First Google Books was used with a search on ISBN
  + If the current book's ISBN was found

A complementary search was performed on Good Reads web site

* + Else

A search with the current book's ISBN was performed on Good Reads web site

* + - If the current book's ISBN was not found

A search, still on Good Reads web site, was performed with title and author

### AWS

The scrapping was very long, so we needed a solution in order to execute parallel scrapping. But on Google books, there is a limitation on the number of requests performed : not too many requests had to be made in a short time. Then it would have disabled our PCs for too long time.

So the other way to execute parallel scrapping was to rent other computers. So AWS has been used : an *Ubuntu EC2 instance* has been opened.

On this instance *Anaconda* has been installed in order to be able to use *jupyter notebooks*.

*GitHub services* were already installed, which allow us to use jupyter notebooks and files directly from our project GitHub.

Then, with the help of *Byobu*, our scrapping has been executed continuously on a remote computer.

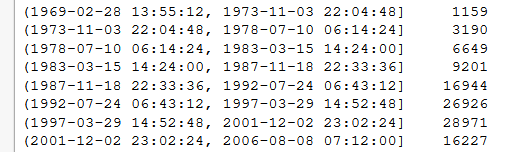
Thanks to a *sftp connexion*, scrapped information have been retrieved from the remote computer to our PC.

## First statistics

On the scrapped version of Book Crossing data base, some data analysis have been performed.

***Years of publication :***

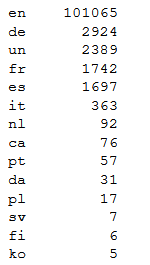
Lots of books published between 1983 and 2006 :



[…]

***Language :***

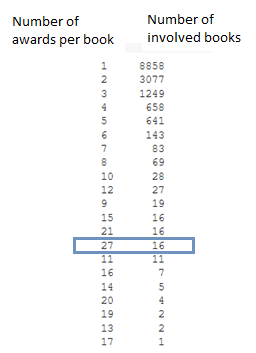
Mainly english books :



[…]

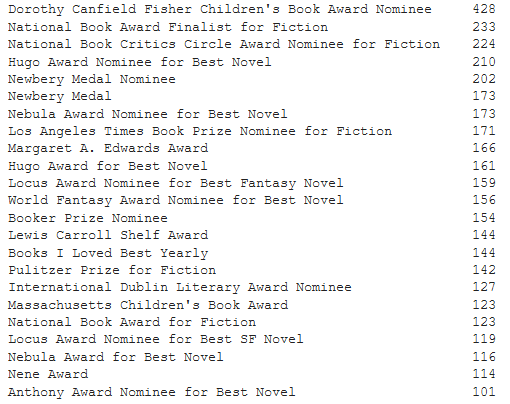
***Awards :***

The maximum number of awards for a same book is 27 !



* For example, "Harry Potter and the Sorcerer's Stone".

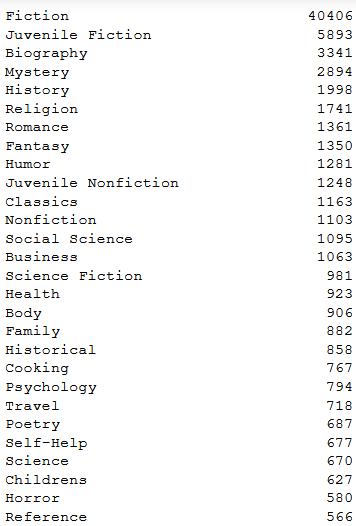
There are 2033 different kind of awards :



[…]

***Categories :***

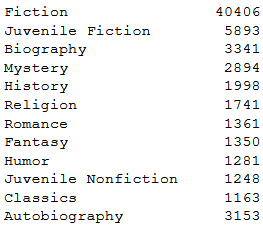
There are 5010 book's categories ! But some categories are only represented by one book, and for example lots of categories with less than 80 books inside.



[…]

* "Fiction" represents 35.28 % of all books.

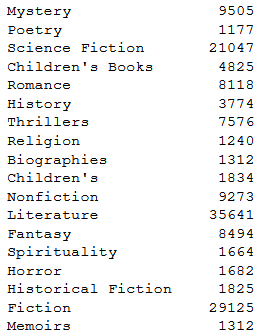
Categories representing at least 1% of all the books are :



***Author gender :***

There are lots of "Literature" as author gender. Some author gender are only represented by one book.

Author genders representing at least 1% of all the books are :



# Word embedding (theory)

## 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.

Lemmatisation is the algorithmic process of determining the lemma of a word based on its intended meaning. Unlike stemming, lemmatisation depends on correctly identifying the intended part of speech and meaning of a word in a sentence.

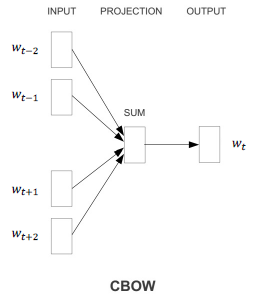
## Word embedding

### Word2Vec

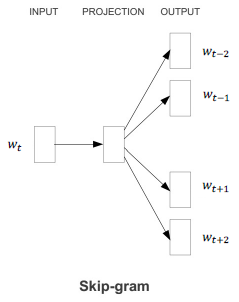
***Two available algorithms :***

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 to appear as the target word



* ***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.*

***Word meaning learned :***

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 :

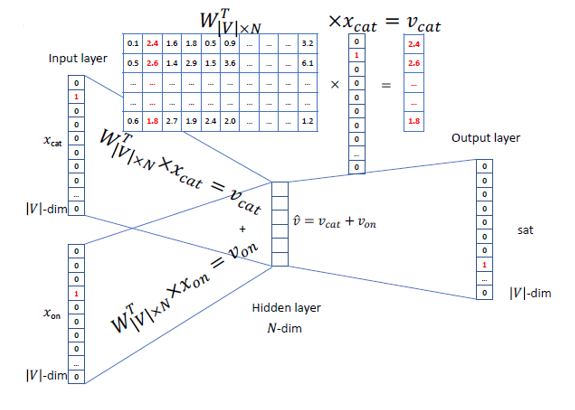
vector(Paris) — vector(France) + vector(Morocco) ~ vector(Rabat)

* syntactic relationships such as the Singular-Plural:

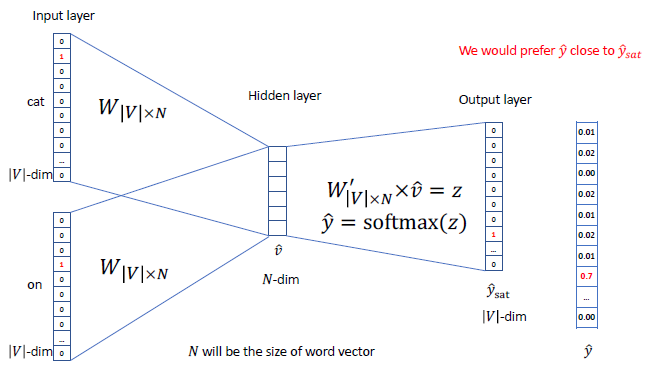
vector(Kings) - vector(King) + vector(Person) ~ vector(People)

***Training of this model :***

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



Then the predicted word is calculated :

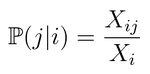


### GloVe

***Assumption performed :***

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 :

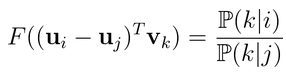


We can discover relationships between words by looking at ratios of these probabilities. For instance, by computing the following ration for lots of words j :

⇒ "ice" is to "solid"….

⇒ …what "steam" is to "gas"

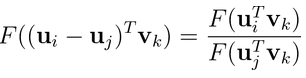
***Model built from this assumption :***

 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 :

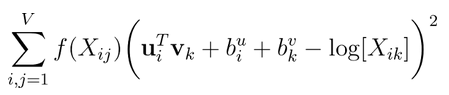
* F should be a homomorphism between the groups (R,+) and (R+, ×):

 ⇔ 

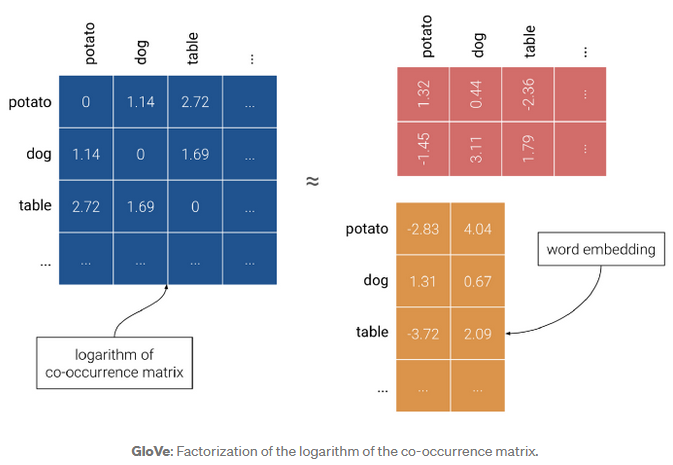
* Since the last logarithm is independent from k, it can therefore be absorbed in a bias bi. Adding a second bias bk completes the symmetry of the model:



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.*



### Fast Text

***No more out of vocabulary words :***

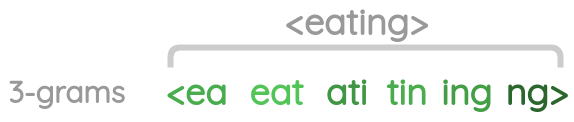
An issue with GloVe and Word2vec is that they only learn embeddings for words of the vocabulary. As a result, Out-of-vocabulary (OOV) words don’t have a vector representation and are usually either represented by a vector of zeros, by the average of all word vectors or simply ignored from the sentence.

FastText (embedding technique) learns embeddings of character n-grams, i.e. sequences of n successive characters :

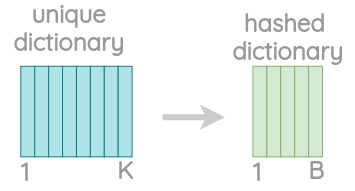
* We take a word and add angular brackets to denote the beginning and end of a word :



* Thus, we get a list of character n-grams for a word :



* Since there can be huge number of unique n-grams, we apply hashing to bound the memory requirements. Each character n-gram is hashed to an integer between 1 to B. Though this could result in collisions, it helps control the vocabulary size :





***Then the learning looks like a Skip Gram processing :***

Inputs :

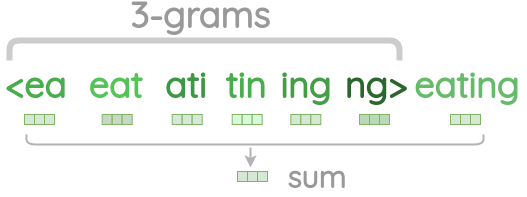
To understand the pre-training, let’s take a simple toy example. We have a sentence with :

* a center word “eating”
* and need to predict the context words “am” and “food”



Center word treatment :

First, the embedding for the center word is calculated by taking a sum of vectors for the character n-grams and the whole word itself :



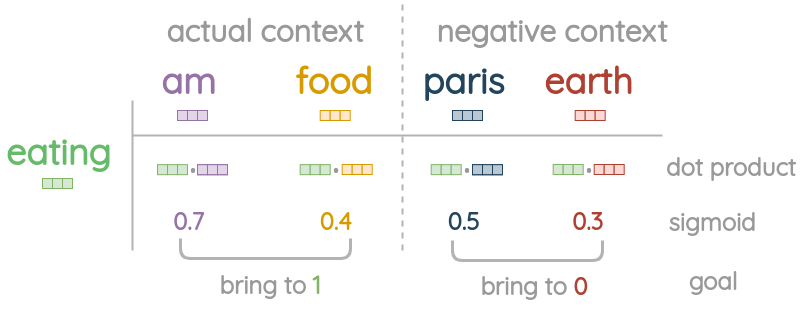
Context words treatment :

For the actual context words, we directly take their word vector from the embedding table without adding the character n-grams.



Internal training :

* We collect negative samples randomly.
* We take dot product between the center word and the actual context words and apply sigmoid function to get a match score between 0 and 1.
* Based on the loss, we update the embedding vectors with SGD optimizer to bring actual context words closer to the center word but increase distance to the negative samples.



## Sentence embedding

### Doc2Vec

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.

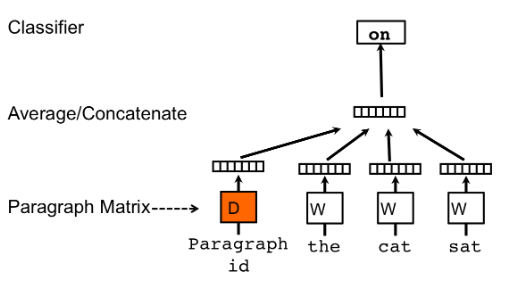
It is an unsupervised algorithm and adds on to the Word2Vec model by introducing another ‘paragraph vector’. Also, there are 2 ways to add the paragraph vector to the model.

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

It is an extension of the Continuous Bag-of-Word type of Word2Vec : the model learns to predict a center word based on the context.

Similarly, in PV-DM, the central idea is : paragraph vectors are obtained by training a neural network on the task of predicting a probability distribution of words in a paragraph given a randomly-sampled word from the paragraph.

On the following matrix D has embeddings for paragraphs :



* ***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

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
  + The BERT base uncased model contains 12 layers.
* Siamese Network

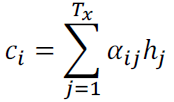
***BERT encoder :***

It's a bi-directional RNN. In it's base version, there are 6 encoders.

***BERT decoder :***

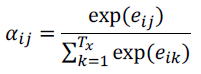
In it's base version, there are 6 decoders.

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

 where  with learned coefficients 

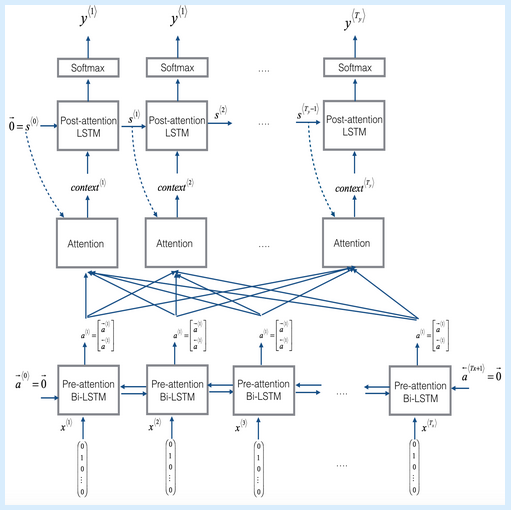
hj are the hidden states of encoder

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

 with 

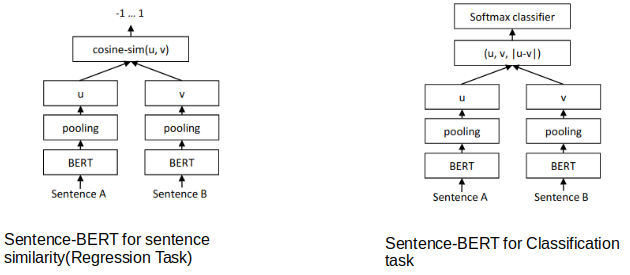
In such a way, the probabilities handled in decoder part take into account of the whole original sentence.

***Bert in a global view :***



***Siamese network :***

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 sub words 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

***Different elements :***

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 for example)
* The word / token number (22 tokens in a sentence for example)
* The hidden unit / feature number (768 features)
* That’s 219,648 unique values just to represent only one sentence !

***How use those elements :***

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 hidden layer*** of each token producing a single 768 length vector.

# Project data base cleaning

All the following cleaning process has been applied on Book Crossing and Good Books database. Then a concatenation of both cleanings has been performed. On the following, our recommendations algorithms will use this concatenated database, which has been again saved in a csv file.

## Cleaning of ISBN

Some ISBN are written with ***"\n"*** before and / or after ISBN number, it has then been removed.

Error of handing :

During the various file manipulations, one database file version has been saved through excel, which transform all the isbn\_13 in string ending with '+12' or '+11'. The isbn\_13 value has then been lost. For those books, the ***isbn\_13 has been removed to 0.0***.

## Language

The ***ISO 639 coding*** has been used to embed language :



[…]

Some languages were coded with words, or even several words :

* "Catalan; Valencian" => ca
* "English, Middle (1100-1500)" en
* "Multiple languages" en
* "zh-CN" zh
* "zh-TW" zh

## Awards

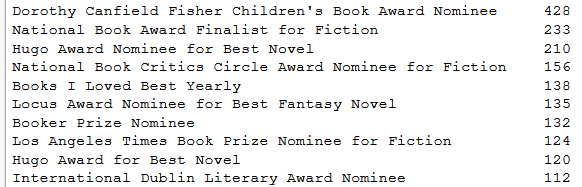
For each book:

* split according to ***","*** in order to obtain the list of awards
* then split according to ***"("*** in order to keep only the name (and not the date)

A list of all cleaned awards has been created, according to its importance (number of times where it has been mentioned in our data base).

We ***kept only one award name per book***, the most famous awards in each cells (compared to the previously created list).

***Kind of awards (after cleaning)***



[…]

* It remains 613 kind of awards.

## Category and author gender

### First cleaning

Books categories and author gender have been considered together for the cleaning. Indeed there represent quite the same information about the book.

Some gender are coded with two words, separated by ***&***. So a 2nd column has been created, which means that the author gender is now represented by the two columns : "author\_genres" and "author\_genres\_other".

Some categories are coded with several words, separated with either ***"&"***, ***"(…)"*** or ***","***. Then, as for author gender, a 2nd column has been created "Category\_other".

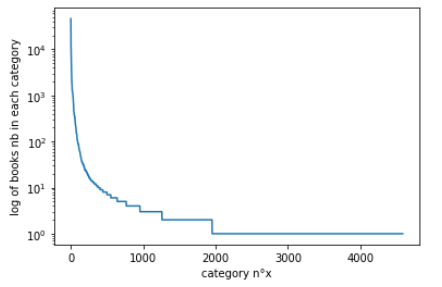
Then a stemming operation has been performed on categories and author gender columns. It has been performed with ***PorterStemmer***.

### Number of category to keep

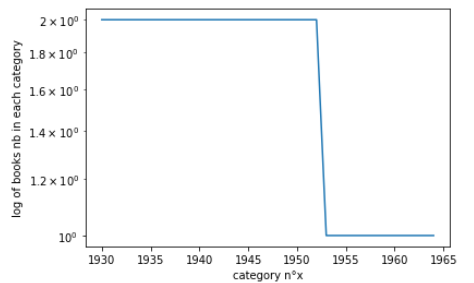
There are too many categories and then we decided to reduced this number in order to help the recommendation algorithm.

But for example, to keep 95% of the most important categories, we should keep the 3989 first categories (on 5010).

We plotted histogram of the number of each categories, but on a logarithmic scale :

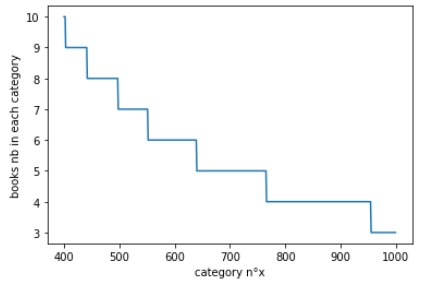


Zoomed view around the 2000th category :



We could chose to keep only the first 1953ieth categories… but it corresponds to categories with only two books inside.

The first ***403*** categories have at least 10 books inside : it's our limit :



### Categories replaced by their closest neighbour

***Word to vec :***

It has been decided to keep only the 403 mostly represented categories, we will name it CAT\_SEL on the following. But the other categories must then be replaced by one of those CAT\_SEL.

For that purpose a word embedding has been used in order to embeds all the categories, and then to be able to find it's closest neighbour among CAT\_SEL when needed.

First it has been tried to use already available words embeddings data base, 'glove-wiki-gigaword-50' and 'word2vec-google-news-300'. But two issues have been encountered :

* Our words categories are not inside.
* Colab do not accept such huge data set loading.

So a ***word to vec embedding*** has been trained from our own database, with the following columns :

* "book\_description"
* "Category"
* "Category\_other"
* "author\_genres"
* "author\_genres\_other"

Before training our word to vec model, a cleaning of "book\_description" column has been performed by removing (removed only for the word to vec training, not in our books database) :

* ***stopwords***
* ***words of length < 2***
* ***'...'***

We used the last version of *Word2Vec* inside *gensim.models*. The Skip Gram version has been used, with numerical vector of length 50 for embedding.

***Find the Category the most close to CAT\_SEL :***

We used the previous trained model function *model.wv.most\_similar\_to\_given*.

## Duplicates

***Duplicates have been removed***. Duplicates have been searched according to first ISBN and then ***title + author***.

# Project data embeddings

Three kinds of embeddings have been tested. In each cases, only the english books have been considered.

***BERT embedding on book's description :***

First the *'bert-base-uncased' pretrained tokenizer* has been used to transform our books description in tokens :

* For each book's description
  + for each sentence of the current book's description
    - encode\_plus method has been used
    - each attention masks and tokens indexes are concatenated
  + Then on the sentences the *pretrained BertModel 'bert-base-uncased'* has been executed

Outputs are of 4 dimensions :

`hidden\_states` have shape [13 layers, len(input\_ids), 64 (size of sentence), 768]

* + Dimension has been reduced in several steps :
    - one specific layer output has been used ([len(input\_ids), 64 (size of sentence), 768]
    - average of tokens vectors of a same sentence ([len(input\_ids), 768]
    - average of sentences vectors of a same book's description ([768])
  + This final vector has been saved in a books dataframe as a list of 768 strings

Values :

The minimum of all values is -24.282314, and the maximum is 2.1074116 :

* Percentage of embeddings value under -10: 0.13 %
* Percentage of embeddings value between -10 and -5: 0.0 %
* Percentage of embeddings value under -5 and -1: 0.28 %
* Percentage of embeddings value under -1 and 0: 52.1 %
* Percentage of embeddings value above 0: 47.49 %

***Fast Text on book's description :***

First data cleaning has been performed, mainly with the use of regular expressions :

* Remove all the special characters
* Remove all single characters
* Substituting multiple spaces with single space
* Converting to lowercase
* Lemmatization with *PorterStemmer*
* Removing of punctuation with *word\_punctuation\_tokenizer*

Then *FastText from gensim library* has been used to train a model from our book's description.

Once trained, each previous tokens created for each book's description have been embedded thanks to *wv* method.

All tokens embeddings resulting in a dimension 60 vectors. So each token vectors of a same book's description have been averaged in a global final vector for the current book's description.

As previously, this final vector has been saved in a books dataframe as a list of 60 strings.

Values :

The minimum of all values is -1.8926484078168868, and the maximum is 2.0181634426116943.

***Fast Text on categories context :***

This time, several book's features have been used :

* "book\_title"
* "book\_author"
* "Category"
* "Category\_other"
* "author\_genres"
* "author\_genres\_other"

For each book, a kind of sentence has been created by concatenation of the previous columns. Then FastText has been trained on those "sentences" and with the same process as previously, each book has been represented by a vector of dimension 60.

Values :

The minimum of all values is -0.6839812078202764, and the maximum is 0.6203103736042976.

To be completed by Asif

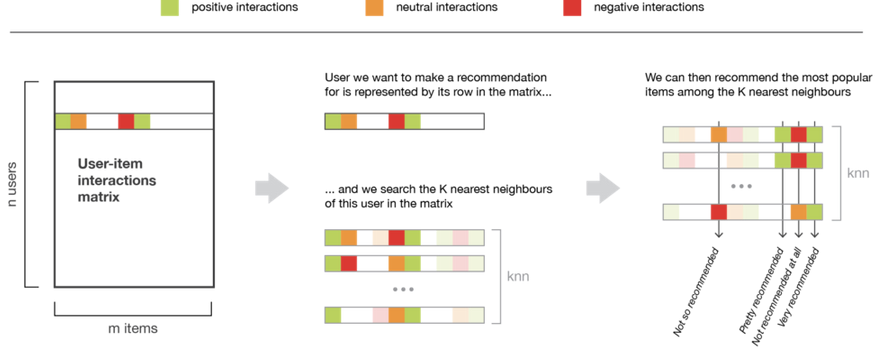
* The features date could be transformed in OneHot encoded classes
* The feature number of pages could just be concatenated to the embedding vector

# Recommendations algorithms (theory)

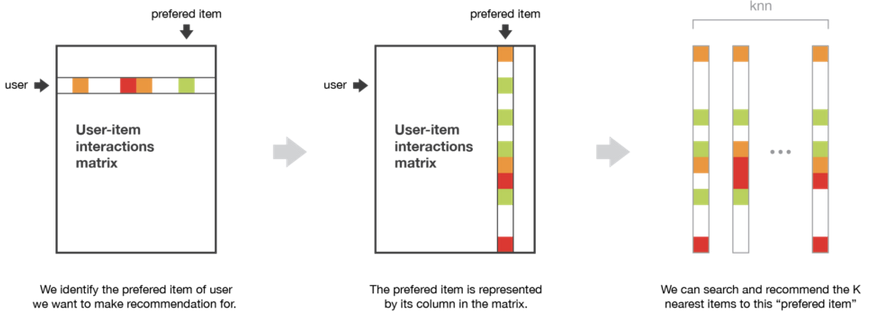
## Context

There are mainly two different methods for recommendations :

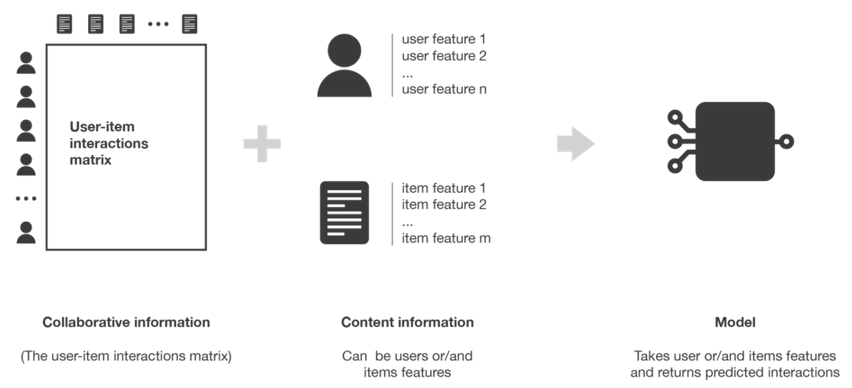
* **Collaborative filtering methods :** based solely on the past interactions recorded between users and items in order to produce new recommendations. These interactions are stored in the so-called “user-item interactions matrix” (a matrix with only ratings values) :
  + Collaborative User-User :



* + 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 information 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)

## Neighbours

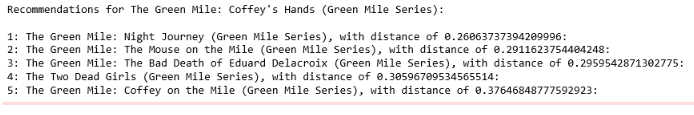
* The following methodology can be applied for example in the context of collaborative Item-Item / User-User.

***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 output will be : books with few ratings distance are recommended.



***Collaborative User-User***

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

## 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

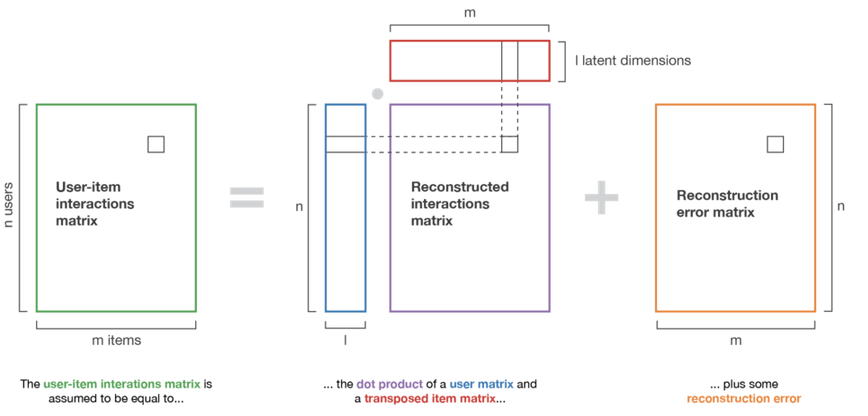
* The following methodology can be applied for example in the context of :
  + Content based methods as books and users features can be used
  + Collaborative based methods as only ratings matrix can be used

Matrix handled :

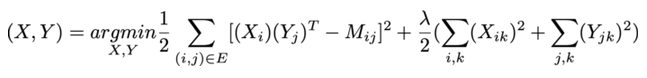
* Matrix M : only ratings (rows are users and columns are items)
* Vector Xi : coefficients to be learned for user i
* Matrix Yj : coefficients to be learned for books j in context of collaborative methods

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.

## 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.

* Each of the K segment corresponds to one of the K labels and a predictor value of θy−1< z < θy (i.e. in the yth segment) corresponds to a rating of y.

Ordinal regression :

We are given a training set (xt, yt)t=1...T of T rated items, where for each item, xt ∈ Rd is a feature vector describing the item and yt is the rating level for the item.

We want to predict preferences of future items. We do so by learning a prediction mapping z(x) : Rd → R such that for an item with feature vector x, z(x) corresponds as well as possible to the appeal of the item :

z(x) = wTx + w0

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.

We describe two different constructions for loss functions based on such thresholds. The constructions differ in how predictors outside the segment corresponding to the “correct” label are penalized.

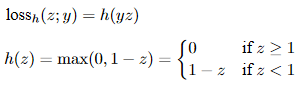
Penalty function :

Both constructions are based on combining penalties for threshold violation, where each threshold violation is penalized using some ***margin penalty function f(·)*** as for example :

* Zero-one error :



* Soft-margin of Support Vector Machine (SVM)



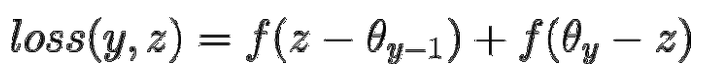
* Logistic regression

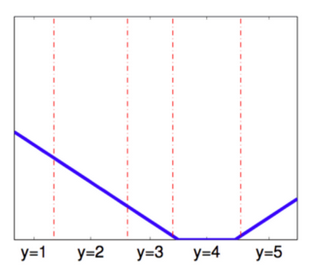


Loss function (based on any previous penalty function) :

On the following, the “correct” segment is (θy−1, θy), which means that θy−1 < z < θy, and the following losses penalizes violations of these thresholds :

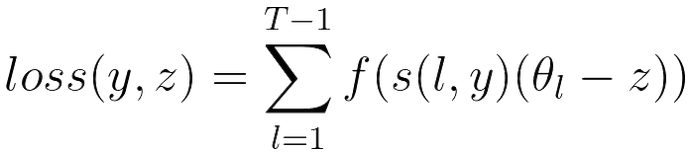
* the **immediate-threshold** : we can set a loss function that continuously increases in both directions outside the range of the correct prediction (when a threshold is crossed).





* 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. Indeed, it is better to predict ‘4’ than ‘1’ if the true label is ‘5’.

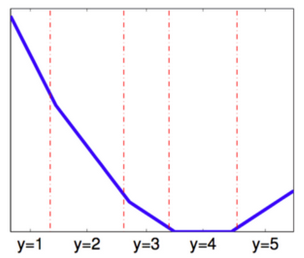
A more complete construction that bounds multiple penalties according to the number of threshold violations is the all-threshold method.



where l is the reference range for loss computation

and s(l, y) ≥ 0 if l > y

s(l, y) ≤ 0 otherwise



Note that the slope of the loss increases each time a threshold is crossed. Thus, solutions are encouraged that minimize the number of thresholds that are crossed.

Case where penalty function is the logistic regression :

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 modelled as a linear combination of the independent variables. But, the above approach of modelling ignores the ordering of the categorical dependent variable.

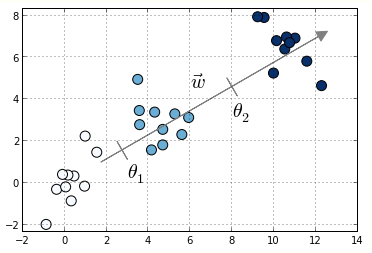
Whereas, in ordinal logistic regression the dependent variable is ordinal i.e. there is an explicit ordering in the categories.

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



Compared to multiclass logistic regression, we have added the constrain that the hyperplanes that separate the different classes are parallel for all classes, that is, the vector *w* is common across classes.

To decide to which class will *Xi*be predicted we make use of the vector of thresholds *θ*. We will then assign the class *j* if the prediction *wTX* (recall that it's a linear model) lies in the interval [*θj*−1, *θj*[ :



MORD :

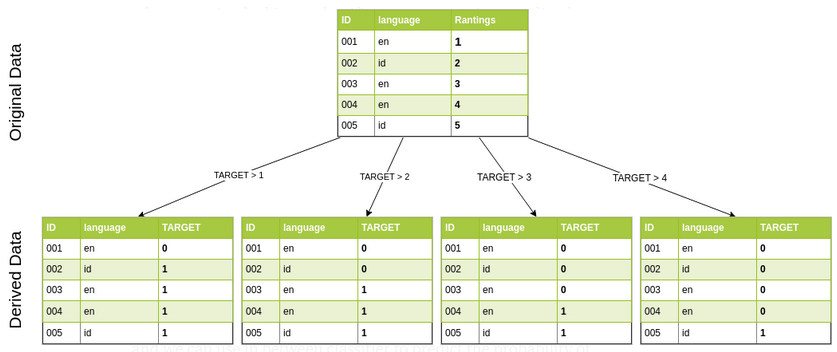
The Python library MORD (Multi-class classifier for ORDinal regression) allow to implement this algorithm.

This library provides another version which is the penalized Ridge version :



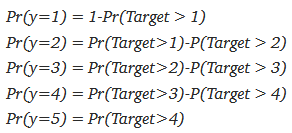
### Trick with classifiers

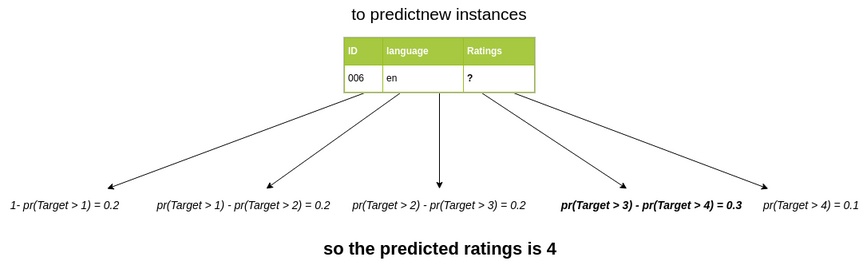
We can transform a k classifier into k-1 binary classifiers :



* Each binary classifier evaluate an ordered value.

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





# Project recommendation

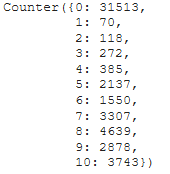
## Only one embedding at a time

On this chapter we use the embeddings previously explained.

### 10 unbalanced classes

Ordinal regression has been used with the three previously explained embeddings. The labels were the ratings given by users.

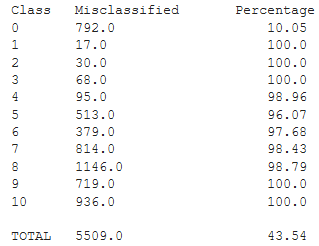
However, the ratings are very unbalanced (below are the labels used for regression training) :



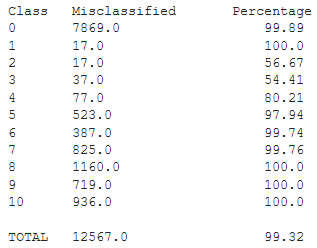
And then the results are not very good :

Book's description with Bert embedding :

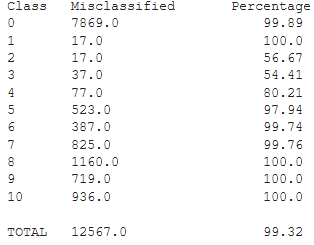
* Percentage of misclassified (LogisticAT): 43,54 %



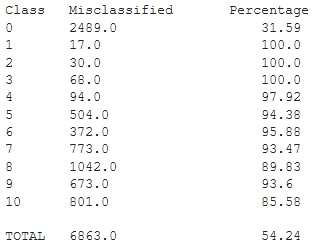
* Percentage of misclassified (OrdinalRidge): 99,32 %



* Percentage of misclassified (LinearRegression): 99,32 %

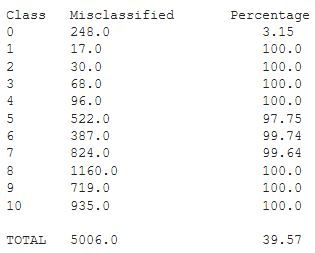


* Percentage of misclassified (RandomForestClassifier): 54,24 %

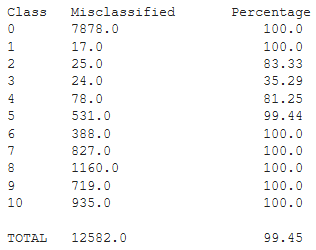


Book's description with fastText embedding :

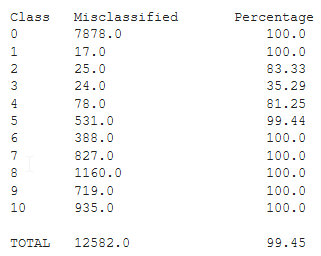
* Percentage of misclassified (LogisticAT): 39,57 %



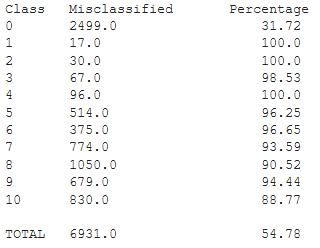
* Percentage of misclassified (OrdinalRidge): 99,45 %



* Percentage of misclassified (LinearRegression): 99,45 %

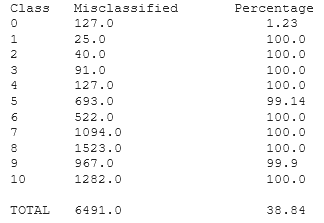


* Percentage of misclassified (RandomForestClassifier): 54,78 %

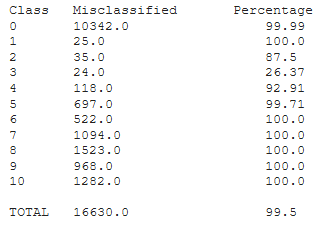


Categories context with fastText embedding :

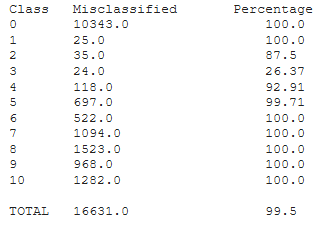
* Percentage of misclassified (LogisticAT): 38,84 %



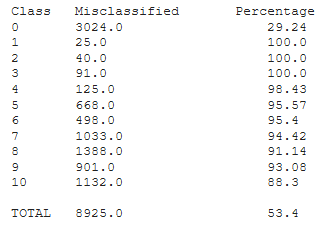
* Percentage of misclassified (OrdinalRidge): 99,50 %



* Percentage of misclassified (LinearRegression): 99,50 %



* Percentage of misclassified (RandomForestClassifier): 53,40 %



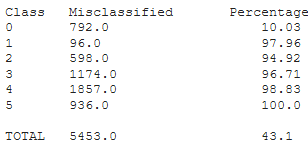
### 5 unbalanced classes

Now ratings are divided by 2 in order to reduce the number of classes to 5 (below are the labels used for regression training) :

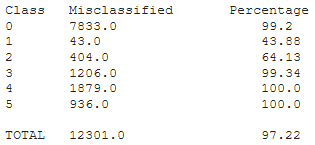


Book's description with Bert embedding :

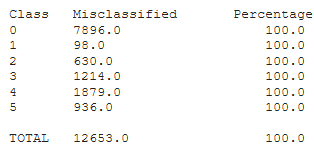
* Percentage of misclassified (LogisticAT): 43,10 %



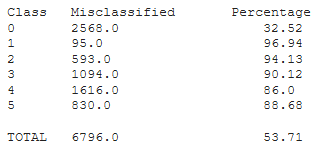
* Percentage of misclassified (OrdinalRidge): 97,22 %



* Percentage of misclassified (LinearRegression): 100 %



* Percentage of misclassified (RandomForestClassifier): 53,71 %



### Balanced binary classes

Then in order to understand the bad influence of the unblanced classes, the following kind of downsampling has been performed in order to keep only two balanced classes :

* Class '1' of well-reviewed books : old 7, 8, 9 and 10 ratings
* Class '0' of poorly rated books :
  + old 1, 2, 3 and 4 ratings
  + a part of old 0 ratings, the number needed to balance classes

With those binary classes, results are :

Book's description with Bert embedding :

* Percentage of misclassified (LogisticAT): 44,17 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 911(55,52 %) | ***1 531 (44,48 %)*** |
| Class '1' | ***1 650 (43,87 %)*** | 2 111 (56,13 %) |

* Percentage of misclassified (OrdinalRidge): 44,05 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 913 (55,64 %) | **1 525 (44,36 %)** |
| Class '1' | ***1 648 (43,77 %)*** | 2 117 (56,23 %) |

* Percentage of misclassified (RandomForestClassifier): 46,15 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 833 (53,46 %) | ***1 596 (46,54 %)*** |
| Class '1' | ***1 728 (45,79 %)*** | 2 046 (54,21 %) |

* Percentage of misclassified (LogisticRegression): 44,01 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 919 (55,67 %) | ***1 528 (44,33 %)*** |
| Class '1' | ***1 642 (43,72 %)*** | 2 114 (56,28 %) |

Book's description with fastText embedding :

* Percentage of misclassified (LogisticAT): 44,42 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 810 (55,56 %) | ***1 448 (44,44 %)*** |
| Class '1' | ***1 751 (44,40 %)*** | 2 193 (55,60 %) |

* Percentage of misclassified (OrdinalRidge): 44,32 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 811 (56,67 %) | ***1 442 (44,33 %)*** |
| Class '1' | ***1 750 (44,32 %)*** | 2 199 (55,68 %) |

* Percentage of misclassified (RandomForestClassifier): 46,72 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 806 (52,87 %) | ***1 610 (47,13 %)*** |
| Class '1' | ***1 755 (46,35 %)*** | 2 031 (53,65 %) |

* Percentage of misclassified (LogisticRegression): 44,25 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 814 (55,75 %) | ***1 440 (44,25 %)*** |
| Class '1' | ***1 747 (44,25 %)*** | 2 201 (55,75 %) |

Categories context with fastText embedding :

* Percentage of misclassified (LogisticAT): 45,15 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 2 375 (54,12 %) | ***2 013 (45,88 %)*** |
| Class '1' | ***2 316 (44,08 %)*** | 2 854 (55,20 %) |

* Percentage of misclassified (OrdinalRidge): 45,28 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 2 370 (54,20 %) | ***2 003 (45,80 %)*** |
| Class '1' | ***2 321 (44,76 %)*** | 2 864 (55,24 %) |

* Percentage of misclassified (RandomForestClassifier): 45,92 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 2 435 (53,31 %) | ***2 133 (46,69 %)*** |
| Class '1' | ***2 256 (45,21 %)*** | 2 734 (54,79 %) |

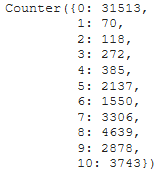
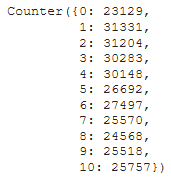
* Percentage of misclassified (LogisticRegression): 46,70 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 2 065 (52,92 %) | ***1 837 (47,08 %)*** |
| Class '1' | ***2 626 (46,43 %)*** | 3 030 (53,57 %) |

### Oversampling / undersampling to balance classes

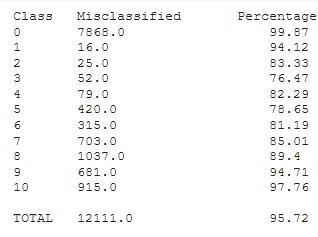
It has been applied with the following embedding as inputs : book's description with Bert embedding.

* *SMOTETomek*(ratio='auto') from imblearn library (only applied on train data set) :

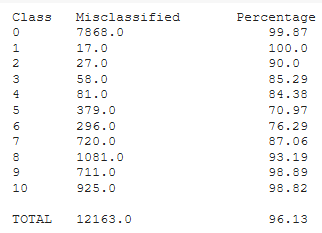
 

⇒

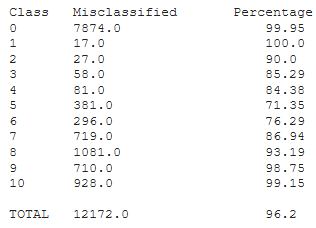
* + Percentage of misclassified (LogisticAT): 95,72 %



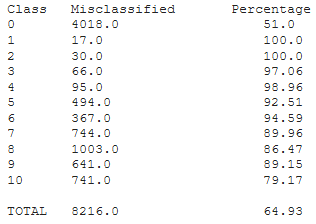
* + Percentage of misclassified (OrdinalRidge): 96,13 %



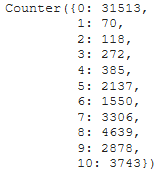
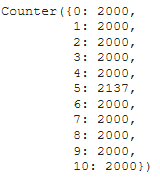
* + Percentage of misclassified (LinearRegression): 96,20 %



* + Percentage of misclassified (RandomForestClassifier): 64,93 %

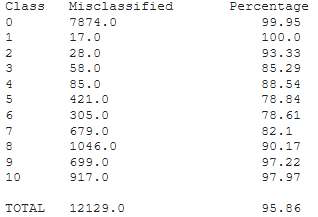


* A first oversampling with SMOTE on small classes, and then an undersampling with RandomUnderSampler on big classes, in order to obtain classes of about 2 000 elements each :

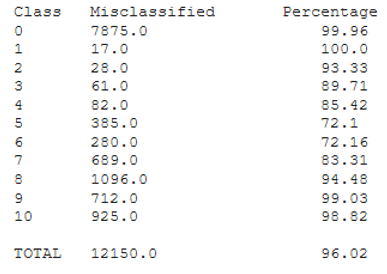
 

⇒

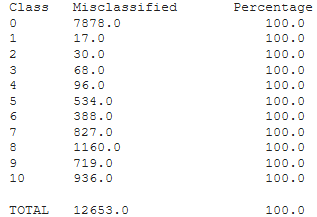
* + Percentage of misclassified (LogisticAT): 95,86 %



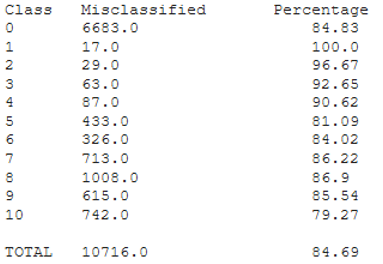
* + Percentage of misclassified (OrdinalRidge): 96,02 %



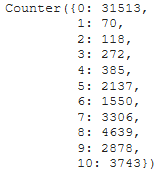
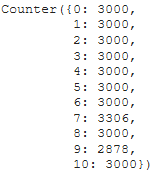
* + Percentage of misclassified (LinearRegression): 100 %



* + Percentage of misclassified (RandomForestClassifier): 84,69 %



* A first oversampling with SMOTE on small classes, and then an undersampling with RandomUnderSampler on big classes, in order to obtain classes of about 3 000 elements each :

⇒

Results are not better than previously (and then not reported here).

Now ratings are divided by 2 in order to reduce the number of classes to 5.

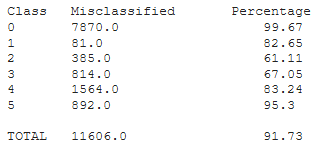
* *SMOTETomek*(ratio='auto') from imblearn library (only applied on train data set) :



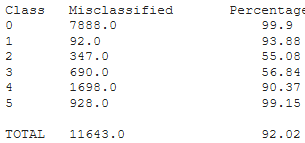
⇓



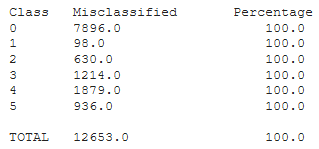
* + Percentage of misclassified (LogisticAT): 91,73 %



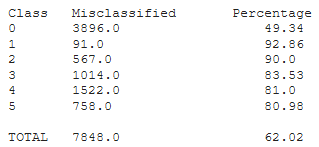
* + Percentage of misclassified (OrdinalRidge): 92,02 %



* + Percentage of misclassified (LinearRegression): 100 %



* + Percentage of misclassified (RandomForestClassifier): 62,02 %



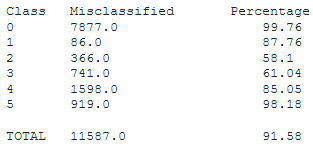
* A first oversampling with SMOTE on small classes, and then an undersampling with RandomUnderSampler on big classes, in order to obtain classes of about 4 000 elements each :



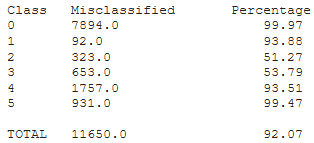
⇓



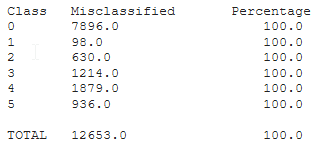
* + Percentage of misclassified (LogisticAT): 91,58 %



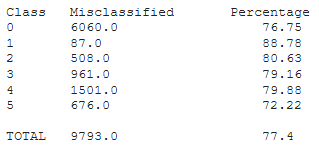
* + Percentage of misclassified (OrdinalRidge): 92,07 %



* + Percentage of misclassified (LinearRegression): 100 %



* + Percentage of misclassified (RandomForestClassifier): 77,40 %



### PCA

PCA has been tested on the following configuration :

* Book's description embedded with Bert
* 10 balanced classes thanks to over / under sampling

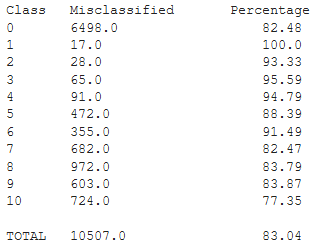
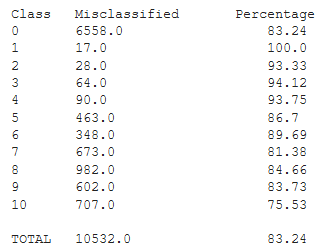
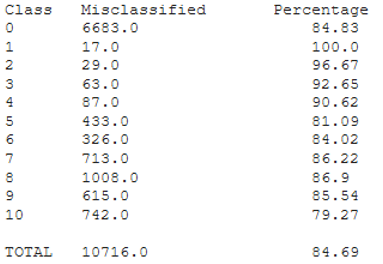
The number of components kept has been chosen according to the variance explained by the new components :

* With 10 components : 44% of variance explained
* With 20 components : 56% of variance explained
* With 30 components : 63% of variance explained
* With 50 components : 71% of variance explained
* With 100 components : 82% of variance explained
* With 150 components : 87% of variance explained

Two tests have then been performed with 100 and 150 components kept :

* Results are similar, but not better for the 3 regressions
* Results are slightly better for random Forest :

Original components 100 new components 150 new components



## Several embeddings at the same time

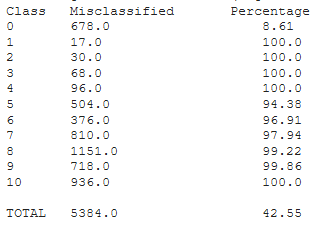
On this chapter we still use the 3 embeddings previously explained. But we tried to combine 2 of them before passing them into the input of the algorithm.

The concatenation and adition have been tested as combination of embeddings.

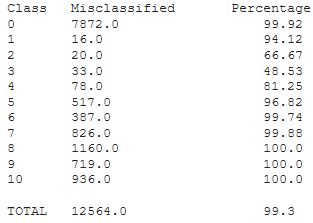
***Unbalanced classes :***

As a first attempt, it has been tried to concatenate the book's description with Bert embedding and the Categories context with fastText embedding. The results are the following :

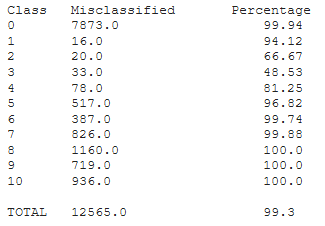
* Percentage of misclassified (LogisticAT): 42,55 %



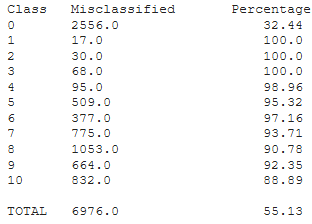
* Percentage of misclassified (OrdinalRidge): 99,30 %



* Percentage of misclassified (LinearRegression): 99,30 %



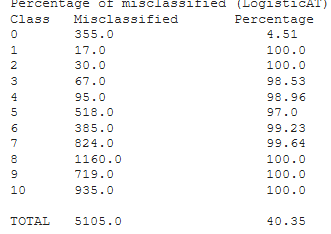
* Percentage of misclassified (RandomForestClassifier): 54,13 %



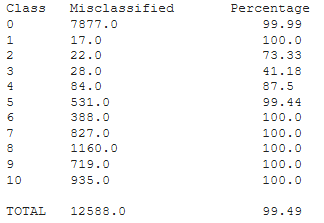
*Remark : Normalisation of all the embedding values do not improve such previous results.*

Then the concatenation of book's description with fastText embedding and the Categories context with fastText embedding has been tested. In that case, the two embeddings are of same length. The results are the following :

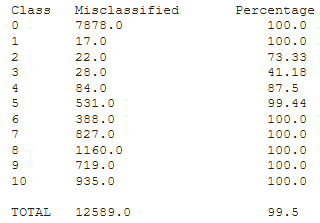
* Percentage of misclassified (LogisticAT): 40,35 %



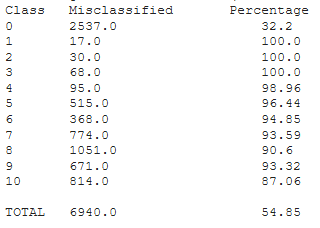
* Percentage of misclassified (OrdinalRidge): 99,49 %



* Percentage of misclassified (LinearRegression): 99,50 %



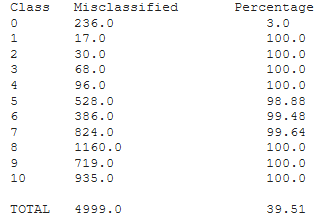
* Percentage of misclassified (RandomForestClassifier): 55.11 %



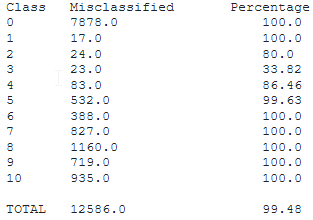
*Remark : It is slightly better for Random Forest and LogisticAT in case where all the embedding values are first normalized.*

Then the term by term ***addition*** of book's description with fastText embedding values and the Categories context with fastText embedding values has been tested. In that case, the two embeddings are of same length. The results are the following :

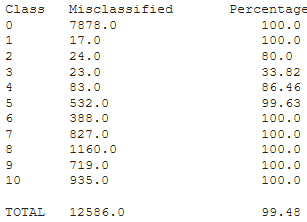
* Percentage of misclassified (LogisticAT): 39,51 %



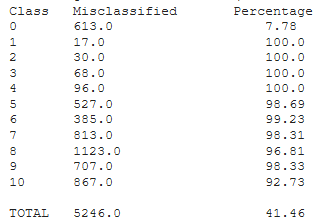
* Percentage of misclassified (OrdinalRidge): 99,48 %



* Percentage of misclassified (LinearRegression): 99,48%



* Percentage of misclassified (RandomForestClassifier): 41,46 %



***Balanced binary classes :***

Concatenation of book's description with Bert embedding and Categories context with fastText embedding. The results are the following :

* Percentage of misclassified (LogisticAT): 43,87 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 947 (55,74 %) | **1 546 (44,26 %)** |
| Class '1' | **1 614 (43,50 %)** | 2 096 (56,50 %) |

* Percentage of misclassified (OrdinalRidge): 44,00 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 948 (55,58 %) | **1 557 (44,42 )%** |
| Class '1' | **1 613 (43,62%)** | 2 085 (56,38 %) |

* Percentage of misclassified (RandomForestClassifier): 46,09%

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 852 (53,48 %) | **1 611 (46,52 %)** |
| Class '1' | **1 709 (45,70 %)** | 2 031 (54,30 %) |

* Percentage of misclassified (LogisticRegression): 44,40 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 921 (55,22 %) | **1 558 (44,78 %)** |
| Class '1' | **1 640 (44,04 %)** | 2 084 (55,96 %) |

Concatenation of book's description with fastText embedding and the Categories context with fastText embedding has been tested. In that case, the two embeddings are of same length. The results are the following :

* Percentage of misclassified (LogisticAT): 43,54 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 891 (56,33 %) | **1 466 (43,67 %)** |
| Class '1' | **1 670 (43,43 %)** | 2 175 (56,57 %) |

* Percentage of misclassified (OrdinalRidge): 43,45 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 892 (56,44 %) | **1 460 (43,56 %)** |
| Class '1' | **1 669 (43,35 %)** | 2 181 (56,65 %) |

* Percentage of misclassified (RandomForestClassifier): 46,39 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 799 (53,26 %) | **1 579 (46,74 %)** |
| Class '1' | **1 762 (46,08 %)** | 2 062 (53,92 %) |

* Percentage of misclassified (LogisticRegression): 43,93 %

|  |  |  |
| --- | --- | --- |
|  | Prediction '0' | Prediction '1' |
| Class '0' | 1 842 (56,04 %) | **1 445 (43,96 %)** |
| Class '1' | **1 719 (43,91 %)** | 2 196 (56,09 %) |

## Comparative table

In order to synthesize all the results, the following notations have been used :

* XE embedding E applied to data X
* Desc refers to the book's description feature
* CatContext refers to the categories features
* '' means same results as the box above
* - means no correct prediction

All the results still use the 3 embeddings previously explained.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | DescBERT | DescFastText | CatContextFastText | DescBERT ||  CategoriesFastText | DescFastText ||  CategoriesFastText | DescFastText +  CategoriesFastText |
| ***Unbalanced 10 classes*** | | | | | | |
| LogisticAT | "0" ≈ 10% | "0" ≈ 3% | "0" ≈ 1% | "0" ≈ 8% | "0" ≈ 5% | "0" ≈ 3% |
| OrdinalRidge | "2", "3" ≈ 55%  "4" ≈ 80% | "3" ≈ 35%  "2", "4" ≈ 80% | "3" ≈ 25%  "2", "4" ≈ 90% | "3" ≈ 50%  "2" ≈ 65%  "4" ≈ 80% | "3" ≈ 42%  "2" ≈ 74%  "4" ≈ 87% | "3" ≈ 34%  "2" ≈ 80%  "4" ≈ 86% |
| LinearReg | '' | '' | '' | '' | '' | '' |
| RandomForest | "0" ≈ 30%  "8", "10" ≈ 85% | "0" ≈ 30%  "8", "10" ≈ 90% | "0" ≈ 30%  "8", "10" ≈ 90% | "0" ≈ 30% | "0" ≈ 32%  "8", "10" ≈ 90% | "0" ≈ 8% |
| ***Unbalanced 5 classes*** | | | | | | |
| LogisticAT | "0" ≈ 10% |  |  |  |  |  |
| OrdinalRidge | "1" ≈ 45%  "2" ≈ 65% |  |  |  |  |  |
| LinearReg | - |  |  |  |  |  |
| RandomForest | "0" ≈ 30%  "3", "4", "5" ≈ 90% |  |  |  |  |  |
| ***Balanced 10 classes, thanks to oversampling / undersampling*** | | | | | | |
|  | ***(SMOTETomek)*** |  | ***(SMOTE + RandomUnderSampler)*** |  |  |  |
| LogisticAT | "3" ≈ 75%  "4", "5", "6" ≈ 80%  "2", "7" ≈ 85% |  | "5" ≈ 60%  "4", "6" ≈ 75% |  |  |  |
| OrdinalRidge | "6" ≈ 75%  "5" ≈ 70%  "3", "4", "7" ≈ 85% |  | "5" ≈ 45%  "4", "6" ≈ 80% |  |  |  |
| LinearReg | '' |  | - |  |  |  |
| RandomForest | "0" ≈ 50 %  "10" ≈ 80 %  "7", "8", "9" ≈ 90 % |  | "10" ≈ 75 %  "5" ≈ 80 %  "0","7" to "9"≈ 85% |  |  |  |
| ***Balanced 5 classes, thanks to oversampling / undersampling (SMOTE + RandomUnderSampler)*** | | | | | | |
| LogisticAT | "2", "3" ≈ 60 %  "1", "4" ≈ 85 % |  |  |  |  |  |
| OrdinalRidge | "2", "3" ≈ 55 % |  |  |  |  |  |
| LinearReg | - |  |  |  |  |  |
| RandomForest | "0" ≈ 75 %  "2" to "5" ≈ 80 % |  |  |  |  |  |
| ***Balanced binary classes*** | | | | | | |
| LogisticAT | "0", "1" ≈ 44% | "0", "1" ≈ 44% | "0" ≈ 46%  "1" ≈ 44% | "0", "1" ≈ 44% | "0", "1" ≈ 43% |  |
| OrdinalRidge | '' | '' | "0" ≈ 46%  "1" ≈ 45% | '' | '' |  |
| RandomForest | "0", "1" ≈ 46% | "0", "1" ≈ 47% | "0" ≈ 47%  "1" ≈ 45% | "0", "1" ≈ 46% | "0", "1" ≈ 46% |  |
| LogisticReg | "0", "1" ≈ 44% | "0", "1" ≈ 44% | "0", "1" ≈ 47% | "0", "1" ≈ 44% | "0", "1" ≈ 44% |  |

LogisticAT in unbalanced case is only able to correctly output the mostly represented class.

Random Forest performs slightly better in the sense where it also begins to predict the 2 other classes the mostly represented.

On the contrary, OrdinalRidge and LinearReg tend to better predict the least represented classes, but very poorly.

***Improvements :***

* Those results are improved (better predictions) when the number of classes is reduced to 5.
* Those results are also improved when the classes are balanced thanks to over / under sampling : improved in the sense where more classes are predicts, even if very poorly.
* In case of Random Forest, those results are slightly improved with a previous PCA.

***No improvements :***

* The benefit of using more than one embedded features as input of the recommendation algorithm is not so clear. But maybe it is due to the fact that results are highly biased because of the unbalanced classes.
* In some cases normalisation of all the embedding values slightly improve such previous results.
* In case of balanced binary classes, the predictions are better distributed, and about of 45% of misclassified. It is very close to coin toss.

## Incremental embeddings

In this section, it has been tried to embed features one by one, and not in the same time, as previously performed through categories context embeddings.

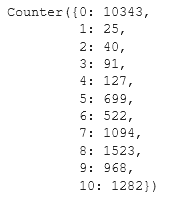
The process is the same as previously explained in the dedicated embedding session, except the fact that this time embeddings have been saved as a matrix of float insde a csv file.

So by keeping the same notation as previously, it has been compared for example :

* TitleE || AuthorE || CategoriesE
* versus (Title || Author || Categories)E = Categories context embedding previously performed

Those tests have been performed with a first oversampling with SMOTE on small classes, and then an undersampling with RandomUnderSampler on big classes. The resulting balanced classes (only for training) are of 3 000 elements each.

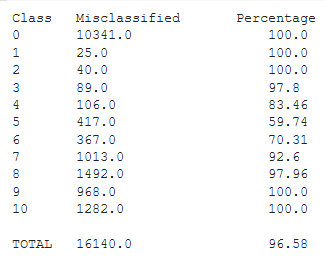
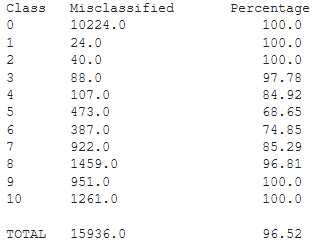
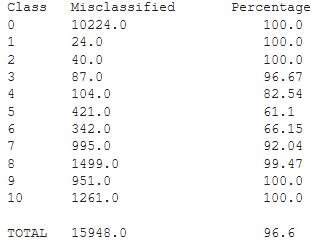
The test data set repartition is the following :



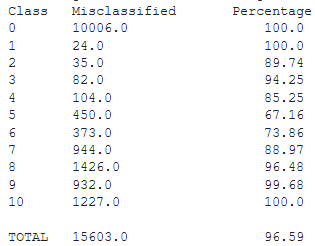
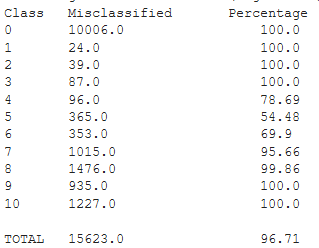
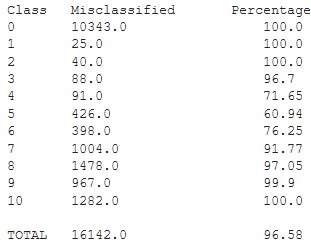
Those tests have been performed with fastText embedding :

* LogisticAT

TitleE  TitleE || AuthorE TitleE + AuthorE

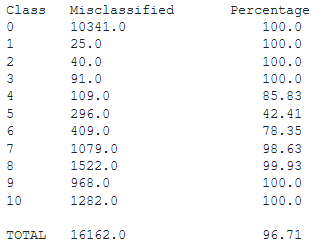
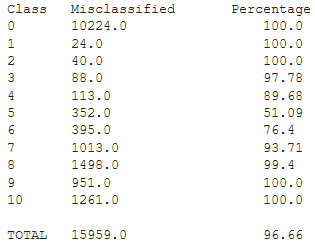
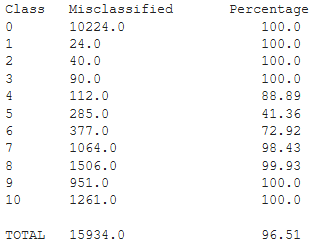
  

TitleE || AuthorE || CategoriesE  TitleE + AuthorE + CategoriesE (Title ||Author|| Categories)E

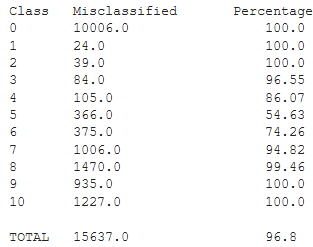
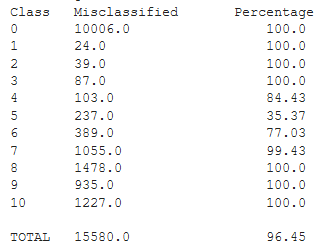
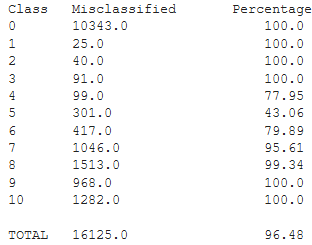
  

* OrdinalRidge

TitleE  TitleE || AuthorE TitleE + AuthorE

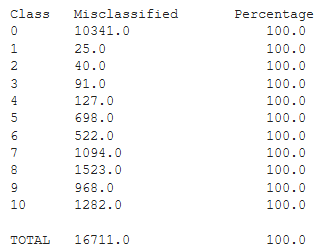
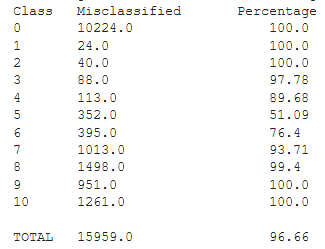
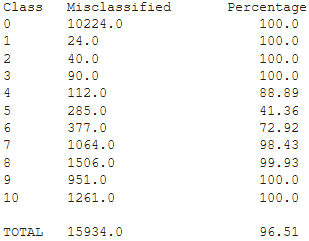
  

TitleE || AuthorE || CategoriesE TitleE + AuthorE + CategoriesE (Title + Author + Categories)E

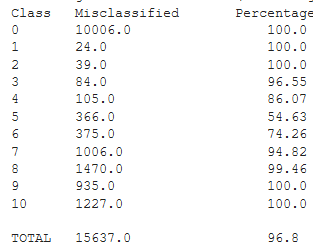
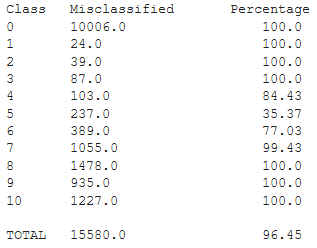
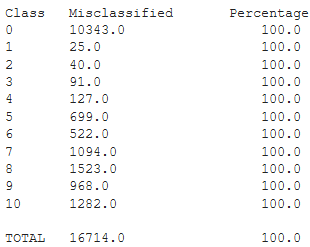
  

* LinearRegression :

TitleE  TitleE || AuthorE TitleE + AuthorE

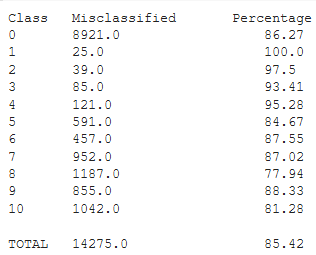
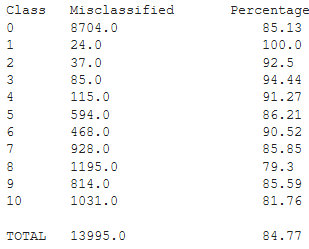
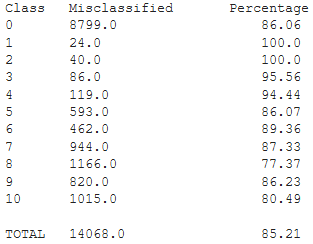
  

TitleE || AuthorE || CategoriesE  TitleE + AuthorE + CategoriesE (Title + Author + Categories)E

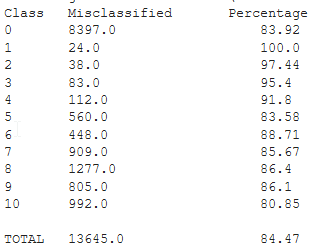
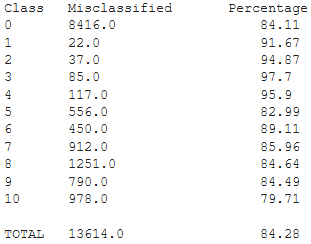
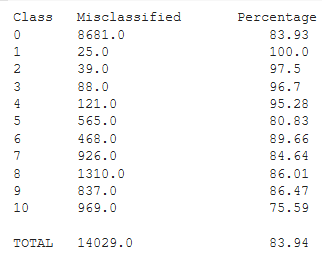
  

* RandomForestClassifier

TitleE  TitleE || AuthorE TitleE + AuthorE

TitleE || AuthorE || CategoriesE TitleE + AuthorE + CategoriesE (Title + Author + Categories)E

***Conclusion :***

For ordinal regression, it seems to have a benefit of embedding features separately, and then to add the embeddings in order to put as inputs in recommendation algorithm.

For linear regression and Random Forest, it is not so clear.

# Web application

The application has not been implemented, but the minimum features to reach had been though as explained on the following.

## Functionalities

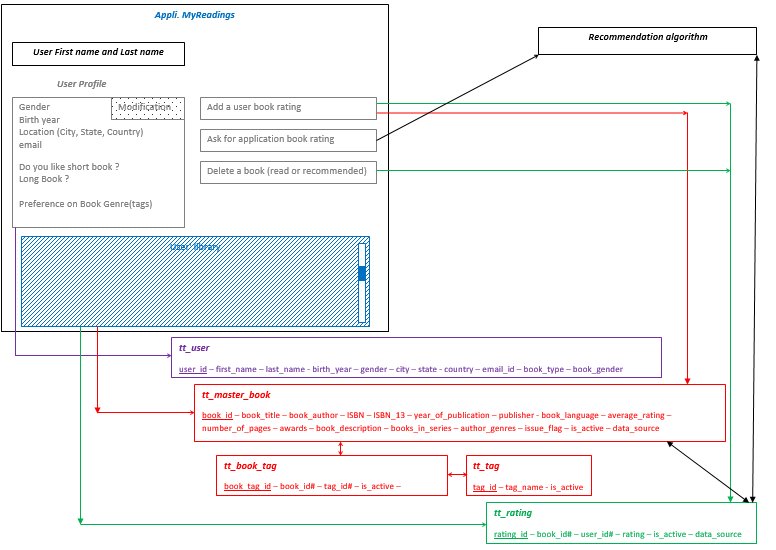
The web structure of our project is explained on the next session, but on the following fonctional view of our application some dynamoDB appear. Those dynamoDB are filled with our previous database csv file.

First a graphical view is reported : almost all the features appearing in the following dynamoDB (Asif, OK ?) databases are those already explained in scrapper session. For reminding the columns of our csv files are :

* isbn
* isbn\_13
* OtherID
* book\_authoryear
* year\_of\_publication
* Publisher
* Category
* Category\_other
* book\_description
* book\_language
* Image
* number\_of\_pages
* author\_genres
* author\_genres\_other
* books\_in\_series
* average\_ratings
* awards

Some new dynamoDB database columns have been added and are :

* In ***tt\_master\_book*** internal database :
  + issue\_flag : to be completed by Asif
  + is\_active : to be completed by Asif
  + data\_source : to be completed by Asif
* In ***tt\_book\_tag*** internal database :
  + tag\_id# : to be completed by Asif
  + is\_active : to be completed by Asif
* In ***tt\_tag*** internal database :
  + tag\_name : "Category" and "Category\_other" of our cleaned database (Ok Asif ?)
  + is\_active : to be completed by Asif
* In ***tt\_rating*** internal database :
  + is\_active : to be completed by Asif
  + data\_source : to be completed by Asif



***User Profile***

***User data***

* As long as this field is not filled in, no other fields are available
* If UserID found:
  + Keep in mind this UserID for the following application execution.
  + Display the *User Profile* fields.
  + A modification button is always present in order to change *User Profile* fields
* If UserID not found:
  + Create a new UserID.
  + Propose the empty *User Profile* fields to fill.

***User' library***

* Recommendations list in priority (because only few books). Then already Read books.
  + Output all book's information.
  + Output corresponding reader rating.

***Books Profile***

***Adding a book in user's library***

* It triggers the recommendation algorithm training

|  |  |  |
| --- | --- | --- |
|  | **Books**  (ISBN) | **UserLibrary**  (UserID - ISBN) |
| If this book already exists | We have to verify whether all columns are filled:   * If not: ***fill columns*** with scrapper * If yes: nothing more to do | We have to ***update existing data*** in other columns.   * Maybe it's a previous recommendation (Rating was 12, and then becomes a real rating). * Maybe it's a previous deleted book (Rating was 11). * Maybe it's a 2nd, 3rd,… reading of an already read book (Rating is then updated). * Maybe it's only the user's rating modification. |
| If this book does NOT exist | New entry to save  + fill all columns with scrapper | New entry to save + fill all columns |

***Deleting a book in user's library***

* It triggers the recommendation algorithm training.
* So the application could keep in mind all the deleted books: Rating is set to 11.

***Ask for appli. rating***

* It triggers the prevision capability of our recommendation algorithm.
* If this book is not in *Books* :
  + New entry saved in *Books* only (not in *UserLibrary*), and fill all columns as in previous *Add* functionality.
* Then perform a rating estimation :
  + If this rating
    - is high: update the user's application recommendations
    - is not high enough, just output the application rating estimation.

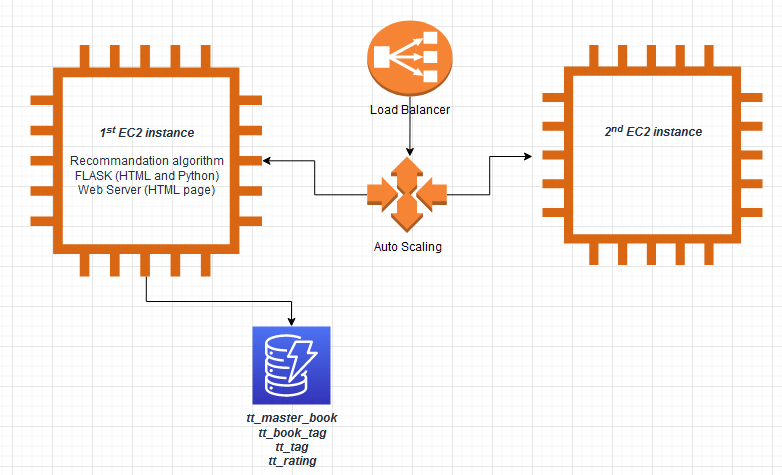
***Recommendation algorithm***

* Training triggered each time a book is added or deleted.
  + Updating of the 5 recommended books, by keeping the most highly rated books by application recommendations.
  + Verify that it was not a previously deleted one (Rating was 11 in tt\_rating). In that case, don't save again this recommended book.
  + Not recommend an already read book !
* Take into account trilogy
* Our recommendations must also bring an element of surprise, and not only deal with entries data.

## Web structure

To be completed / modified by Asif

A first version of our web application has been though as follows. But more simpler solutions exist with flask, in term of web traffic handling (more simpler than AWS load balancer and auto scaler).



# Annexes

## ISBN meaning

An ISBN is an ***International Standard Book Number***.

### History

The initial ISBN identification format was devised in 1967, based upon the 9-digit Standard Book Numbering (SBN) created in 1966.

The ISBN identification format was conceived in 1967 in the United Kingdom by ***David Whitaker*** (regarded as the "Father of the ISBN") and in 1968 in the United States by Emery Koltay.

The 10-digit ISBN format was developed by the [International Organization for Standardization](https://en.wikipedia.org/wiki/International_Organization_for_Standardization) (ISO) and ***was published in 1970*** as international standard ISO 2108.

### Nation dependant

The method of ***assigning an ISBN is nation-specific*** and varies between countries, often depending on how large the publishing industry is within a country.

ISBN issuance is country-specific, in that ISBNs are issued by the ISBN registration agency that is responsible for that country or territory regardless of the publication language.

The ranges of ISBNs assigned to any particular country are based on the publishing profile of the country concerned, and so the ranges will vary depending on the number of books and the number, type, and size of publishers that are active.

### What is identified

Any book made publicly available, whether for sale or on a gratis basis, can be identified by ISBN.

An ISBN is assigned to each separate edition and variation (except reprintings) of a publication. For example, an [e-book](https://en.wikipedia.org/wiki/E-book), a [paperback](https://en.wikipedia.org/wiki/Paperback) and a [hardcover](https://en.wikipedia.org/wiki/Hardcover) edition of the same book will each have a different ISBN

*Remark : Other identifiers exist :*

* *the International Standard Serial Number (ISSN), identifies periodical publications such as magazines and newspapers.*
* *The International Standard Music Number (ISMN) covers musical scores.*

### Who ask for ISBN creation

It is always the publisher of the book who should apply for the ISBN.

### Various formats

* ISBNs were 10 digits in length up to the end of December 2006.
* Since 1 January 2007 they now always consist of 13 digits.

An ***ISBN-10*** is converted to ***ISBN-13*** by prepending "978" to the ISBN-10 and recalculating the final checksum digit using the ISBN-13 algorithm.

Currently the barcodes on a book's back cover are ***EAN-13*** ([European Article Number](https://en.wikipedia.org/wiki/European_Article_Number)).



*Remark : They may have a separate barcode encoding five digits called an EAN-5 for the currency and the recommended retail price.*



### Mathematic formula

ISBNs are calculated using a specific mathematical formula and include a check digit to validate the number.



Each ISBN consists of 5 elements with each section being separated by spaces or hyphens. Three of the five elements may be of varying length:

* ***Prefix element*** – currently this can only be either 978 or 979.
  + It is always 3 digits in length.

Bookland is a fictitious country that exists solely in EAN for the purposes of non-geographically cataloguing books in the otherwise geographically keyed EAN coding system.

* ***Registration group element*** – this identifies the particular country, geographical region, or language area participating in the ISBN system.
  + This element may be between 1 and 5 digits in length

|  |  |  |
| --- | --- | --- |
| Prefix | Registration group | Meaning |
| 978 | 0 or 1 | English-speaking countries |
|  | 2 | French-speaking countries |
|  | 3 | German-speaking countries |
|  | 4 | Japan |
|  | 5 | Russian-speaking countries |
|  | 600–625 |  |
|  | 65 |  |
|  | 7 | People's Republic of China. |
|  | 80–94  *ex 80*  *ex 85* | *Czech Republic; Slovakia*  *Brazil* |
|  | 950–989  *ex 960* | *Greece* |
|  | 9917–9989  *ex 9971* | *Singapore* |
|  | 99901–99983  ex 99921 | *Qatar* |
| 979 | 8 | United States of America |
|  | 10 | France |
|  | 11 | Republic of Korea |
|  | 12 | Italy |

* ***Registrant element*** - this identifies the particular publisher or imprint.
  + This may be up to 7 digits in length
* ***Publication element*** – this identifies the particular edition and format of a specific title.
  + This may be up to 6 digits in length
* ***Check digit*** – this is always the final single digit that mathematically validates the rest of the number. It is calculated using a Modulus 10 system with alternate weights of 1 and 3.
  + For a 10 digits ISBN :



* + For a 13 digits ISBN:



### Errors in usage

Publishers and libraries have varied policies about the use of the ISBN check digit. Publishers sometimes fail to check the correspondence of a book title and its ISBN before publishing it; that failure causes book identification problems for libraries, booksellers, and readers.

For example, ISBN 0-590-76484-5 is shared by two books :

* Ninja gaiden®: a novel based on the best-selling game by Tecmo (1990)
* and Wacky laws (1997), both published by Scholastic.

The ***Library of Congress catalogue*** contains books published with invalid ISBNs, which it usually tags with the phrase "Cancelled ISBN".

However, book-ordering systems such as Amazon.com will not search for a book if an invalid ISBN is entered to its search engine.