

Tags prediction using NLP

NLP, Topic modeling, supervised learning

Amine Fatmi | NLP | 17/01/2021

Contents

[Scope of work 2](#_Toc61818643)

[Data collection 3](#_Toc61818644)

[EDA and data cleaning 4](#_Toc61818645)

[Data visualization 4](#_Toc61818646)

[Data cleaning 7](#_Toc61818647)

[Document-Term Matrix 8](#_Toc61818648)

[Topic modeling 8](#_Toc61818649)

[Latent Dirichlet Allocation 9](#_Toc61818650)

[Non-Negative Matrix Factorization 12](#_Toc61818651)

[Supervised learning algorithms 13](#_Toc61818652)

[Problem statement 13](#_Toc61818653)

[Preprocessing 13](#_Toc61818654)

[Results 13](#_Toc61818655)

[Model improvement 14](#_Toc61818656)

[Model deployment 15](#_Toc61818657)

# Scope of work

Stackoverflow is a famous question and answer website for professional and enthusiast programmers. When asking a question, you need to enter some tags so that it is found more easily. This is generally not a problem for experienced users. However, it can be trickier for new users of the platform.

For that purpose, it can be helpful to build a Machine Learning model that suggests tags based on the question asked. In particular, we will focus on a specific part of ML called Natural Language Processing (NLP). We will be working with unstructured text data. Hence, we will see what are the techniques and tools from NLP we could use to transform the data so that a machine can process it.

In this report, I will take you through all the steps from data collection to model deployment on a cloud platform.

Let us dive into it!

# Data collection

Stackoverflow provides a tool to export data from its website called “stackexchange explorer”. We will be using simple SQL queries to get the data we need.

To limit the risk of having questions that are not tagged properly, I focused on popular questions only (view count > 500 and score > 75).

There is a limitation of 50 000 elements for each single query so I made five different queries to extract the data:

“””

SELECT id, AcceptedAnswerId, Score, ViewCount, Body, Title, Tags, AnswerCount,

FavoriteCount

FROM posts

WHERE viewcount > 500 and score > 100

“””

“””

SELECT id, AcceptedAnswerId, Score, ViewCount, Body, Title, Tags, AnswerCount,

FavoriteCount

FROM posts

WHERE viewcount > 500 and score <= 100 and score > 75

“””

“””

SELECT id, AcceptedAnswerId, Score, ViewCount, Body, Title, Tags, AnswerCount,

FavoriteCount

FROM posts

WHERE viewcount > 500 and score <= 75 and score > 50

“””

“””

SELECT id, AcceptedAnswerId, Score, ViewCount, Body, Title, Tags, AnswerCount,

FavoriteCount

FROM posts

WHERE viewcount > 500 and score <= 1000 and score > 100

“””

“””

SELECT id, AcceptedAnswerId, Score, ViewCount, Body, Title, Tags, AnswerCount,

FavoriteCount

FROM posts

WHERE viewcount > 500 and score > 1000

“””

I ended up with a dataset of 100428 questions. I have gathered more columns than required but from now on we will be using only ‘Body’, ‘’Title’ and ‘Tags’.

# EDA and data cleaning

## Data visualization

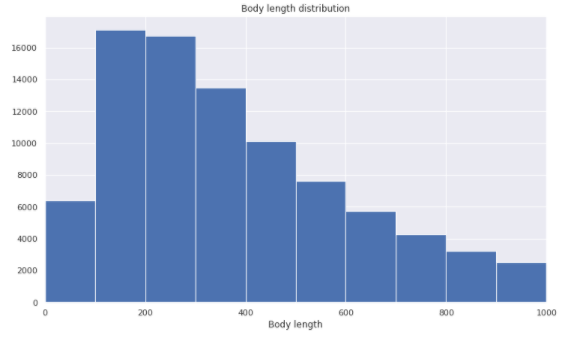
When asking a question, you need to enter a title, a body and tags.



**Figure 1 Example of a question on Stackoverflow**

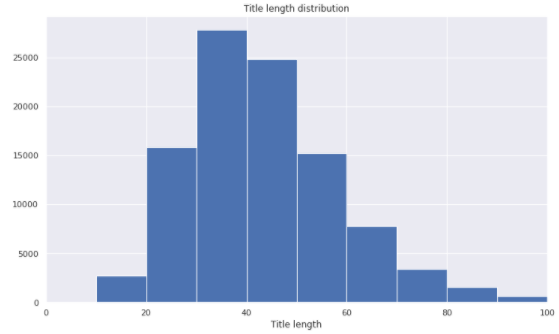
I have highlighted on **Figure 1** the three sections we are interested in.

Let us now have a look at some visualizations to understand our dataset.



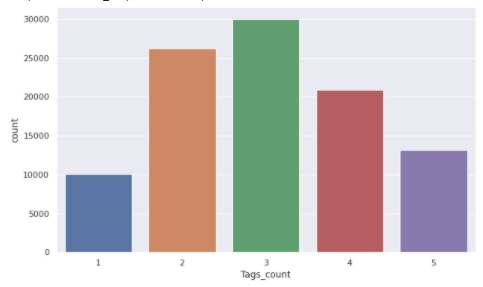
**Figure 2 Body length distribution**

The body length (number of characters) of most questions is around 200 characters.



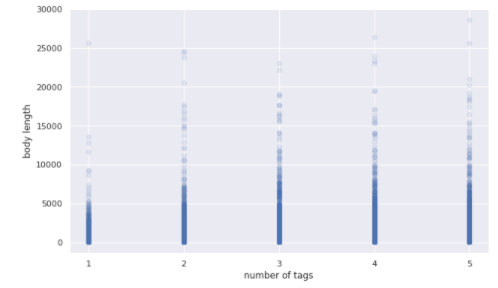
**Figure 3 Title length distribution**

As we would expect, the title length is smaller than the body length and it is on average around 40 characters.



**Figure 4 Number of tags**

The questions in our dataset have between one and five tags. Questions with three tags being the most represented.



**Figure 5 Body length vs Tags count**

There is no clear relation between the length of the body and the number of tags attributed to the question.



**Figure 6 WordCloud of popular tags**

Most of the tags are programming languages. The biggest tags are the most popular: C, javascript, python just to name a few.

## Data cleaning

Below are all the steps I have used to clean the data:

* Remove HTML tags
* Make text all lower case
* Remove punctuation
* Remove non alphabetic
* Tokenize text
* Remove stop words
* Lemmatization

I have mainly used Beautifulsoup (parse html) and NLTK library for the cleaning process.

Tokenization is the process of splitting data into smaller pieces. The most popular token size is a word.

Stop words are words that are not essential to understand the topic within a document. The NLTK stopwords function has 179 words for ‘english’ language. On top of that, I have extended the list of stop words to include some that are specific to our dataset.

Stemming is the process of reducing words to their stem or root. I have preferred Lemmatization to the Stemming because the root is always an existing word.

Now we have to find a way to extract features from our cleaned text data.

# Document-Term Matrix

Let us first define what a corpus, document and term is in the context of NLP:

* Corpus = collection of text (all our questions)
* Document = text (one single question)
* Term = word (or n-grams) within a document

If we take the following question:

“How can I prevent SQL injection in PHP?” 🡪 document

And apply the cleaning process we might end up with something like:

[prevent, sql, injection, php] 🡪 terms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | prevent | sql | injection | php |
| How can I prevent SQL injection in PHP?” | 1 | 1 | 1 | 1 |

The table above is a document term matrix. In that case, it is obtained by counting the occurrence of each term within a document.

The Vocabulary is the list of all unique terms across the corpus. Let us assume we have 10 000 documents and the corresponding vocabulary size is 20 000. Then our document-term matrix shape is 10000\*20000.

There are different ways to fill in the values within this matrix.

The simplest method is to count the words in each document and report the values like in the table above (TF).

Another popular feature extraction method is TF-IDF (term frequency – inverse document frequency). The advantage of this method is that it will also put more weights on terms that are specific to some documents only.

They are the two feature extraction techniques I have used in this project. DTM (or feature matrix) is the input format required to train our machine learning models.

# Topic modeling

A topic model is a type of statistical model for discovering topics from a collection of documents. A document is generally about multiple topics in different proportions. Every topic is a distribution of words.

I will be using two popular topic-modeling algorithms: Latent Dirichlet Allocation (LDA) and Non-Negative Matrix Factorization (NMF).

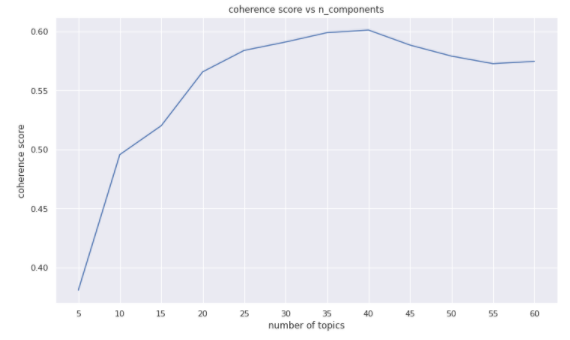
These two models are probabilistic algorithms. We input a DTM and we get a Document-Topic distribution. Every topic being also a distribution of words. These are unsupervised algorithms, which means we do not know the number of topic a priori.

We will be testing different number of topics and choose n\_topic that gives us the best results. Best in that case is not as straightforward as for supervised algorithms where we optimize a metric. However, we have some metrics we can use to help us decide which parameters are best. Nevertheless, it is important that we validate the results by checking the topics found make sense.

## Latent Dirichlet Allocation

I have used the LDA model from scikit-learn library. Another popular library for topic modeling is genism. The coherence metric from genism was used to evaluate my LDA models.

One of the key parameter for LDA is the number of topics. I have run several models from five to 60 topics and computed the coherence score (c\_v measure).



**Figure 7 Coherence score vs number of topics**

The coherence score is significantly increasing until n\_topic = 20, then growing up slowly to a maximum of 0.6 and finally decreasing. By looking at this chart, a number of topic around 20 looks reasonable as the coherence score is high (close to max) and the number of topics is not too big. I did not choose n\_topic = 40 because it is hard to interpret and there are sub-topics. It is better to focus on main topics only.

I have narrowed down my search around 20 topics (15, 20 and 25) and by visually looking at the top words for each topic; n\_topic = 15 was the best choice for the LDA model.



**Figure 8 LDA – top 20 words by topic**

This is not perfect but we can see there are well defined topics such as java, python, database, git…



**Figure 9 LDA – Document-Topic distribution**

The document-topic matrix on Figure 9 highlights the dominant topics (coefficient > 0.1).

I have used the following strategy to extract tags from the LDA results:

* Check the dominant topic(s) for each document
* Identify the top 50 words for each dominant topic
* Intersect the top words from step 2 with the remaining words in the cleaned document
* Set results of intersection = tags
* If #tags found > 5, keep only the first five tags.

I have then used a similarity measure to evaluate the performance of my LDA model. The most relevant I found was Jaccard.

I have computed the Jaccard score for each document by comparing the tags predicted with LDA and the real tags from stackoverflow:

* Jaccard = 0 🡪 no match
* 0 < Jaccard < 1 🡪 partial match
* Jaccard = 1 🡪 perfect match

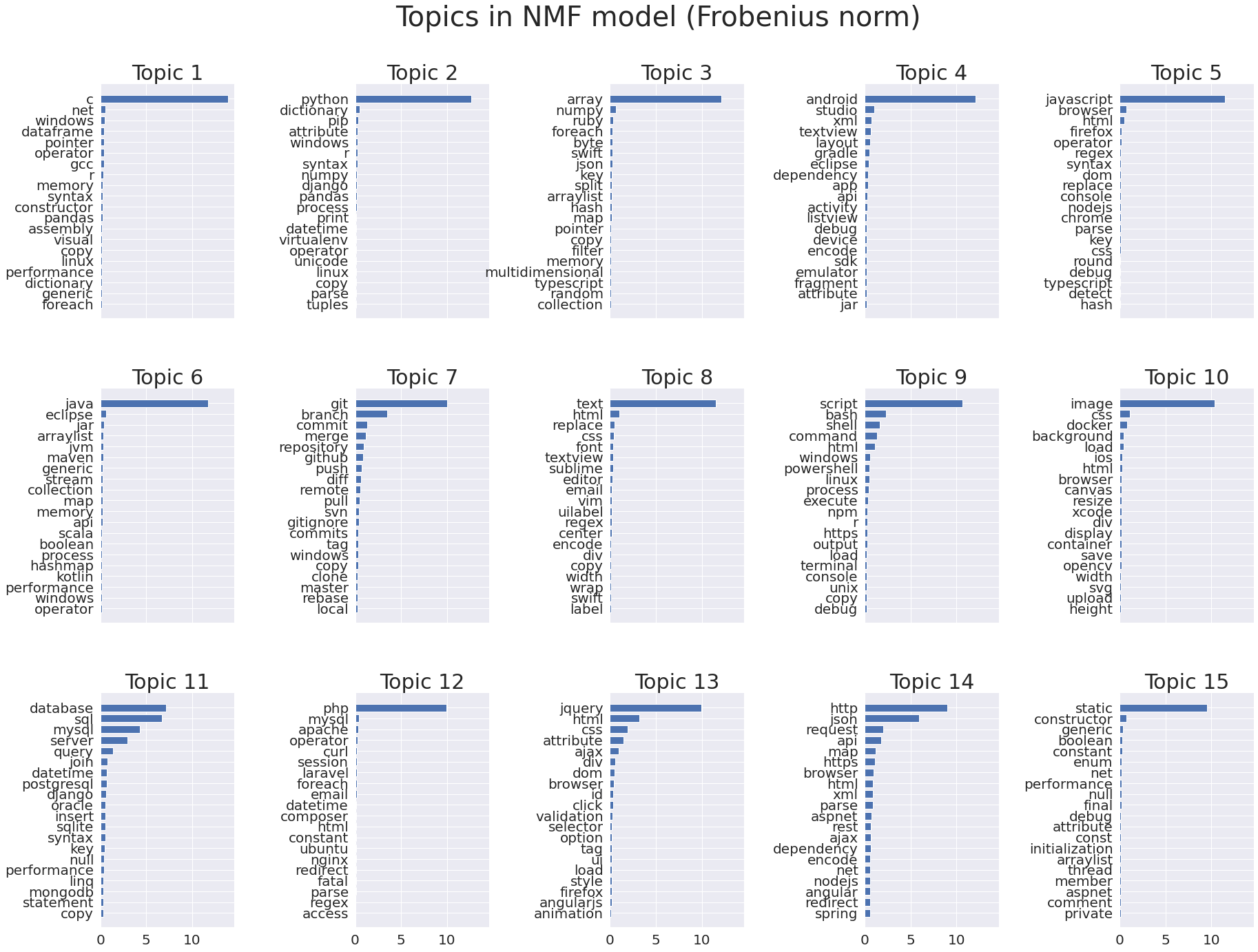
The mean Jaccard score across the all corpus is 0.21.

Key metrics for LDA model:

* Mean Jaccard score = **0.21**
* **63%** of questions with at least 1 tag correctly predicted
* **2%** of questions perfectly predicted

## Non-Negative Matrix Factorization

I will present the results of the NMF model with n\_topic = 15.



**Figure 10 NMF – top 20 words by topic**

The topics look much better than the LDA model. The top word for each topic defines quite well the topic itself. Looking at the bar chart, the top word is also preponderant with respect to the other words of the same topic.

Let us have a look at the results now.

The same tag extraction process was used with the NMF model.

Key metrics for NMF model:

* Mean Jaccard score = 0.22
* 66% of questions with at least 1 tag correctly predicted
* 2.5% of questions perfectly predicted

These metrics confirm that the NMF model is better than the LDA model.

In the next section, I will be comparing different supervised classification models for predicting tags.

# Supervised learning algorithms

## Problem statement

We need first to define clearly the problem in order to use relevant preprocessing techniques and models.

We are trying to predict more than two tags, so it is a multi-class problem. Besides, one question could have more than one tag, so it is also a multi-label problem.

Thankfully, we have models within the scikit-learn library that can handle such classification problems. One popular model is One-Versus-Rest that we can use along with a classifier.

## Preprocessing

1. Multi-label binarizer

I have used MultiLabelBinarizer class from scikit-learn to transform the target variables (tags). I have restrained the model to predict only the top 402 tags (more than 100 occurrences). There are in total 12677 tags in the dataset but the top 402 tags represent 80% of all tags.

My target variable will then be a 2D-matrix with n\_rows = #documents and n\_columns = 402 where indices [i, j] = 1 if tag j is present in document i.

1. TFIDF vectorizer for feature extraction
2. Train / Test split with 80/20 ratio

I have tested four different classifiers along with OVR. See results in next section.

## Results

I will be using the same three metrics as previously to compare the classification models + run time:

* Mean Jaccard score
* % questions with at least one tag correctly predicted
* % questions perfectly predicted
* Training run time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Mean Jaccard | % questions at least 1 tag | % questions perfect match | Run time  (s) |
| OVR + Linear SVC | 0.45 | 69 % | 25.8 % | 46 |
| OVR + Multinomial NB | 0.21 | 32 % | 12.5 % | 9 |
| OVR + Logistic Regression | 0.39 | 61 % | 22 % | 126 |
| OVR + Decision Tree | 0.43 | 74 % | 25.5 % | 1123 |

Table 1 Supervised model comparison

Based on the results above, it looks like Linear SVC is a good model choice for this dataset. I have tuned the regularization parameter C of Linear SVC bringing the Jaccard score to 0.46. I do not think this score can be highly improved unless we do more feature engineering.

I have not tried other more complex models such as Random Forest because it is computationally costly (based on computational cost of the Decision Tree Classifier). The model would have been probably heavy too. This would have led to memory-related issues during model deployment.

Linear SVC is a fair choice and it is the model selected for the cloud deployment.

## Model improvement

I have tried to improve the Linear SVC model by doing further text preprocessing. I have removed the punctuation and special characters from all words within the corpus. However, there are several programming languages such as:

* C#, C++, asp.net,.net,…

that include punctuation or special characters as part of their name.

Besides, I tried to include bi-grams such as ‘android studio’ (without success) and I have increased the number of questions.

By applying these changes, Linear SVC Jaccard score improved to **0.55**, which is more than 10% increase.

I have then decided to deploy this model on the Heroku platform.

You can find the notebooks and code on my GitHub:

https://github.com/aminefatmi

# Model deployment

Framework: Flask

Web server: Gunicorn

Version control system: Git

Cloud: Heroku platform

I will not explain in detail how I created the app using Flask or the whole process to deploy the model (all the files are on GitHub). However, I will highlight some key steps to bear in mind in order to deploy the model successfully:

1. Specify in the requirements.txt the same package versions used when training the model.
2. Create an nltk.txt file where you include the NLTK package you are using in your app.
3. Verify your text file has Unix end of line character (LF).
4. Make sure your app is not too heavy; otherwise, it might crash during the deployment. R14 or R15 – Memory quota exceeded.
5. If you need to push heavy files to GitHub (more than 100Mo), you need to use Git-LFS. Additionally, you will need to add Heroku build pack for Git-LFS.