

Openclassrooms - Parcours Data Scientist

Projet N° 6

Catégorisez automatiquement des questions

Classify stackoverflow.com questions automatically

DOCUMENT	
Reference	File Name
	Openclassrooms Data Scientist projet 6 rapport Christian Muths.docx

Versions	
Date	Reason for change

Table of contents

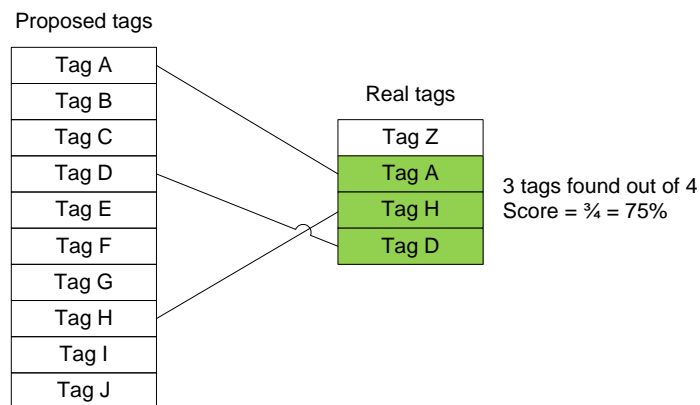
0. Introduction	3
1. Data preparation.....	3
1.1 Data extraction	3
1.1.1 Train set.....	3
1.1.2 Test set.....	3
1.2 List of stems	4
1.3 Stems vectorization	4
2. Principle of tag recommendation system based on non-supervised algorithm	5
2.1 Matrix of topics per question	5
2.1.1 Topic extraction	5
2.1.2 Question-topic probability	6
2.2 Tag decomposition	6
2.3 Tag weight per topic.....	7
2.4 New questions classification	8
2.5 Tag suggestions for new questions	8
2.6 Optimizations	8
2.6.1 Normalize the matrix of weights.....	8
2.6.2 Add bigrams	8
2.6.3 Grid search to find the optimum number of topics.....	8
3. Principle of tag recommendation system based on supervised algorithm.....	9
3.1 List of stems	9
3.2 Multi-label SVM	9
3.3 Classifying new questions	10
4. Conclusion.....	10
5. API development	10

0. Introduction

The target of this project is to propose to a user creating a question on the stackoverflow.com site, a set of relevant tags.

The algorithms propose 5 tags of which the user is invited to pick and choose some of them to add to the question. Performance has been evaluated both with 5 and with 10 tags proposed. In the case tags are proposed to the user to choose from 10 tags make sense. If the system should automatically assign tags without user intervention, 5 tags looks more relevant.

Whatever the algorithm, the performance measurement is based on a comparison with the real tags the user has given. Out of the real tags, we measure how many we guessed. For example, if the user has set 4 tags, and if in the proposed list 3 of them were proposed, the score for this question is 75%



The overall performance indicator is the relative number of questions for which the proposed tags contain at least 50% of the tags the user had chosen.

1. Data preparation

1.1 Data extraction

1.1.1 Train set

A set of legacy questions is extracted from the stackoverflow web site to train the model. The query loads questions where last modification date was last year and score is higher than 100 to use most relevant questions.

Stackexchange query is <https://data.stackexchange.com/stackoverflow/query/edit/847084>

```
select Id, Score, ViewCount, CreationDate, LastActivityDate, title, tags, body
from Posts where (score > 100) and (LastActivityDate > '2017-04-01') and
(LastActivityDate < '2018-04-01') and (PostTypeId = 1)
```

This returns a set of 15089 questions for the training set.

1.1.2 Test set

The test set is extracted with the same rule but last activity date is greater than April 1st 2018.

<https://data.stackexchange.com/stackoverflow/query/edit/846505>

```
select Id, Score, ViewCount, CreationDate, LastActivityDate, title, tags, body
from Posts where (score > 100) and (LastActivityDate > '2018-04-01') and
(PostTypeId = 1)
```

This returns a set of 3865 questions for the test set.

1.2 List of stems

A stem is the root of a word, regardless of the suffixes, conjugation, gender, etc... This reduces the number of values and helps grouping similar notions. However, it may lose some useful information.

A column is added into the questions dataset with the list of stems for each question. Stems are extracted both from the title and the body.

The HTML content of the description gets its HTML tags cleaned out, then a stemmer is used to extract the useful words into stems.

The steps to get the list of stems for each question are:

- Concatenate title and body
- Use BeautifulSoup to remove HTML tags
- Convert into lower case
- Tokenize the resulting string into an array of words
- Remove stopwords using the nltk.stowords list for English language.
- Convert remaining words into stems using nltk SnowballStemmer
- Append the list of stems into a new column of the questions dataframe.

The Python code for this part is:

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
from bs4 import BeautifulSoup

# Take only alphanumeric words, no punctuation signs
tokenizer = nltk.RegexpTokenizer('\w+')

# Prepare set of stopwords
stopWords = set(stopwords.words('english'))

# Define stemmer
snowball_stemmer = SnowballStemmer("english")

wordsFiltered = []
wordsArray = []

for html_text in df_questions['body'] + " " + df_questions['title']:
    soup = BeautifulSoup(html_text, "lxml").get_text()
    words = tokenizer.tokenize(soup.lower())
    his_words = ''
    for w in words:
        if w not in stopWords:
            stem = snowball_stemmer.stem(w)
            wordsFiltered.append(stem)
            his_words = his_words + ' ' + stem
    wordsArray.append(his_words)
```

Out of 15.000 questions, we got a total of approx. 1 Million stem occurrences, with 40.000 unique stems.

1.3 Stems vectorization

The set of stems gets vectorized as unigrams and optionally bigrams. Each question is a line of the matrix and each stem or bigram is a column. The values are the number of occurrences of each stem in each question. Any stem that is used in more than 95% of the questions or less than 5 times across all questions is removed, because it's either too common or too specific to be used for the topic determination.

CountVectorizer is used for unigrams and bigrams extraction. This procedure has many options which would allow much less preprocessing. The raw text could be passed to the vectorizer, and options set to get a similar result. However, it's good to do each step separately to get a better understanding of what happens.

```
from sklearn.feature_extraction.text import CountVectorizer
stem_vectorizer = CountVectorizer(ngram_range=(1, 2), max_df=0.95, min_df=5)
stem_matrix = stem_vectorizer.fit_transform(df_questions['words'])
```

The result is the stem_matrix which has 5.755 stems (or 25500 unigrams + bigrams) in columns, and 15.000 lines. Only approx. 25% of the initial stems were kept. The figures given here are about unigrams. Same vectorization could be done with multigrams.

5700 stems as columns

15.000 questions as lines

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & e \\ d & 0 & 0 & 0 \\ 0 & b & 0 & 0 \end{pmatrix}$$

stem_matrix

```
Maximum number of stems per question: 2386
Minimum number of stems per question: 2
Average number of stems per question: 56
Average number of unique stems per question: 32
```

2. Principle of tag recommendation system based on non-supervised algorithm

This algorithm is based on a non-supervised topic extraction, associated with a statistical approach to get a list of the most likely tags to propose.

Based on a list of questions having titles, descriptions and human-selected tags, several data structures are being built:

2.1 Matrix of topics per question

2.1.1 Topic extraction

The method used to extract topics is the Latent Dirichlet Allocation (LDA). An optimization is done on the number of topics as a parameter.

The measurement of score and perplexity shows that:

- The score doesn't vary much with the number of topics
- the perplexity strictly increases with the number of topics when tested between 2 and 90. It almost doubles with each additional topic. It's close to $1000 \cdot 2^{(ntopics-1)}$.

These measurements aren't useful to choose the optimal number of topics. The final evaluation will be based on the ratio of correct tags proposed.

As an example, with 15 topics, here are the top words for each of them. Some of them are intuitive (Topic 0 = git, Topic 1 = Python, Topic 3 = numbers, dates and times, Topic 6 = SQL databases, Topic 11 = CSS), while some others are less meaningful in a human perspective.

```
Topic #0: file git command use directori commit run chang project branch local
work path want window folder line repository tri script
```

Topic #1: python test line number print import self py modul def virtualenv get key return code str file function tri call

Topic #2: java class public method object new string static void return thread privat call except system null properti code use get

Topic #3: date 00 10 11 05 12 format 15 time 01 02 03 datetim 24 androidruntime 19 13 20 26 18

Topic #4: use differ code understand googl mean question map read vs oper implement data time one warn could applic compil know

Topic #5: div use control set class html work item id model tag element name scope like get properti content ng want

Topic #6: tabl column databas key sql select null queri data mysql row id valu insert creat name server updat set db

Topic #7: function var name js javascript json valu node php type foo npm option script data input get return jqueryi text

Topic #8: android instal version org error id packag build com lib eclips librari app activ java xml usr project view depend

Topic #9: error user app get server http request use applic com system url web tri connect log run messag client password

Topic #10: int array return valu function std type size integ byte swift length fals true doubl element char convert const number

Topic #11: text color css imag background style width left center border 100 right height box margin size pad align red label

Topic #12: button page view click event jqueryi want use div li chang imag class show element like disabl load work tab

Topic #13: net framework td asp height mvc entiti tr use width font differ size linq space pixel content scale pdf vs

Topic #14: string use way like list would one want someth know need exampl variabl get best two case look code question

2.1.2 Question-topic probability

The result of the LDA transformation `lda.fit_transform(stem_matrix)` is the matrix of probability for a question to belong to a topic.

15.000 questions as columns

15 topics as lines $\begin{pmatrix} 0.15 & 0.75 & 0.05 & 0.20 \\ 0.55 & 0.02 & 0.15 & 0.45 \\ 0.10 & 0.03 & 0.55 & 0.25 \\ 0.15 & 0.05 & 0.15 & 0.05 \\ 0.05 & 0.15 & 0.1 & 0.05 \end{pmatrix}$

`lda_corpus.T`

2.2 Tag decomposition

First step is to vectorize the list of tags assigned by the author. The result is a sparse matrix with

5400 tags as columns

15.000 questions as lines

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

tag_matrix

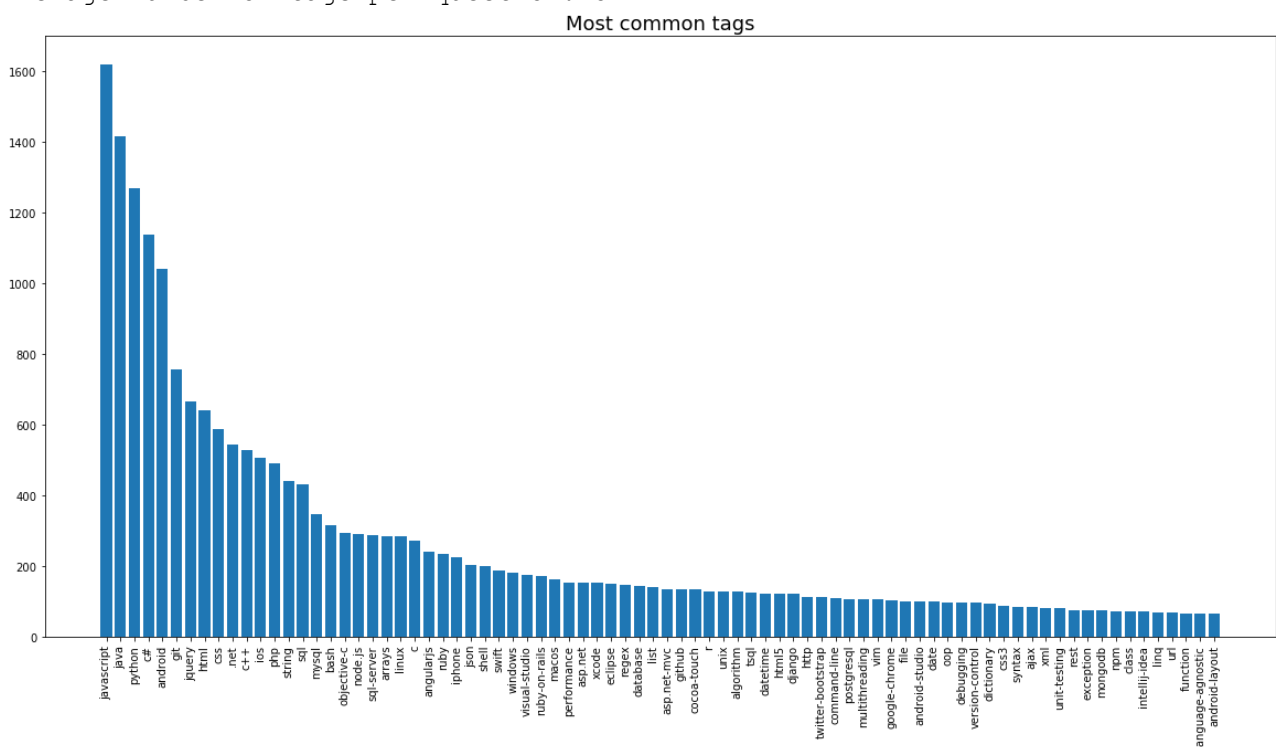
Some statistic about tags distribution shows that some tags are much more used than the average. This has an impact on the performance of the algorithm, because these tags have a very high probability to be proposed, whatever the question content.

It also gives the number of how many tags should be proposed to the user to pick and choose from.

Maximum number of tags per question: 5

Minimum number of tags per question: 1

Average number of tags per question: 3



2.3 Tag weight per topic

A matrix that gives the weight of each tag for each topic is created by multiplying the `lda_corpus` matrix by the `tag_matrix`. Weight can be greater than 1 if a tag is used several times in a topic.

$$\begin{pmatrix} 0.15 & 0.75 & 0.05 & 0.20 \\ 0.55 & 0.02 & 0.15 & 0.45 \\ 0.10 & 0.03 & 0.55 & 0.25 \\ 0.15 & 0.05 & 0.15 & 0.05 \\ 0.05 & 0.15 & 0.1 & 0.05 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.35 & 0.05 & 0.75 \\ 1.00 & 0.15 & 0.02 \\ 0.35 & 0.55 & 0.03 \\ 0.20 & 0.15 & 0.05 \\ 0.10 & 0.1 & 0.15 \end{pmatrix}$$

lda_corpus.T tag_matrix topic_tag_weight

2.4 New questions classification

New questions are classified into topics using the classifier which was trained on the train set. The result is the vector of the question's probabilities to belong to each topic.

The steps followed to reach this are very similar to the process done on the known questions:

- Concatenate title and body
- Use BeautifulSoup to remove HTML tags
- Convert into lower case
- Tokenize the resulting string into an array of words
- Remove stopwords using the nltk.stowords list for English language.
- Convert remaining words into stems using nltk SnowballStemmer
- Vectorize the list of stems using the vectorizer which was fitted with the training set. This ensures to stick to the same set of stems.
- Transform the stem matrix using the LDA transformer which was fitted with the training set. This ensures to use the same exact topics and gets the probabilities for each new question to belong to each topic.

The result is the probability matrix for a question to belong to a topic.

2.5 Tag suggestions for new questions

The probability matrix for each new question to belong to a topic is multiplied by the matrix of tag weight per topic. This gives a distribution of tags with their weight for each question. The 5 to 10 questions having the highest weight are proposed.

2.6 Optimizations

2.6.1 Normalize the matrix of weights

Some tags are over-represented, which makes them appear almost all the time, even if they are not relevant. To reduce this effect, the matrix of weights is normalized on the topics axis.

This change increases the performance by 3 to 6 points

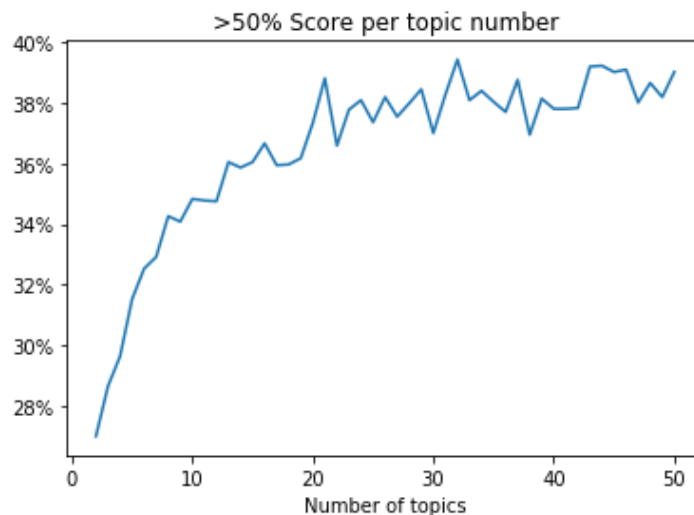
2.6.2 Add bigrams

A try has been made to include bigrams in addition to unigrams. The total number of unigrams + bigrams is 25500 compared to 5700 unigrams. With this algorithm, there is no improvement doing so.

2.6.3 Grid search to find the optimum number of topics

For each case mentioned above, a search of the best number of topics is done.

	5 tags	10 tags
Unigrams without normalization	27% (20 topics)	36% (21 topics)
Unigrams with normalization	29% (43 topics)	39% (32 topics)
Bigrams without normalization	24% (43 topics)	33% (43 topics)
Bigrams with normalization	28% (49 topics)	39% (50 topics)



Example of performance increase depending on number of topics (unigrams only, normalized weights).

3. Principle of tag recommendation system based on supervised algorithm

3.1 List of stems

This algorithm uses the same matrix of stems as the non-supervised algorithm, both for unigrams only as well as bigrams added. → X matrix

Tags are vectorized using a similar process than the stem vectorization, but without removing any data.

```
# Vectorize all the tags used in the training set
tag_vectorizer = CountVectorizer(lowercase = True, max_df=1.0, min_df=0,
token_pattern = '[^<>]+')
tag_matrix = tag_vectorizer.fit_transform(df_questions['tags'])
```

3.2 Multi-label SVM

A multi-label "one vs. rest" SVM is used to get the probability of each tag to belong to a question. This algorithm is first fit on the training set. This results in training as many SMV classifiers as labels to predict.

- Matrix of frequencies of unigrams/bigrams for each question → X matrix
- Labels to predict are the tags given by the users. → Y matrix

A multi-label SVM is trained on the X/Y values.

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
MultiLabelClassif = OneVsRestClassifier(SVC(kernel='linear', probability=True))
MultiLabelClassif.fit(X, Y)
```

The classifier itself is `SVC(kernel='linear', probability=True)`

`OneVsRestClassifier` will run the classifier once for each class (tag). For each tag, the SVC is run against all other tags.

The SVC with option `probability=True` calculates a probability based on the distance between the observation and the separation hyperplan. A logistic sigmoid function is applied to the distance (more generally a 1st degree polynomial of this distance) to get this probability.

3.3 Classifying new questions

The trained classifier is used to classify new questions with the `predict_proba` method. The result is the probability for each tag to belong to each of this new question.

The 5 to 10 tags having the highest probability are proposed to the user to qualify his question.

Results are much better than the non-supervised methodology.

Using bigrams in addition to unigrams increases the score.

	5 tags	10 tags
Unigrams	58%	68%
Unigrams + Bigrams	63.7%	71.5%

4. Conclusion

The supervised algorithm gives much better results. It yields more than 70% of performance with the defined measurement method, whereas the non-supervised algorithm remains below 40%

The fitting time of the supervised algorithm is very long, but it can run off-line and will not harm the user.

The time to classify one new question with the fitted classifier is fast enough to be used on a web site.

Globally, even with the better results of the multi-label algorithm, it's not realistic to build a complete automatic classification program. The ratio of wrong tags would still be too high. With the current solution, it's better to propose the list of tags to the user, and that the user chooses the tags most relevant to him.

One drawback is that such algorithm suggests only existing tags, and more frequent tags are proposed more often. This doesn't allow new tags, related to new topics, to be proposed. A completely non-supervised approach could mitigate this drawback, but it's more difficult to evaluate as it requires human evaluation.

5. API development

The training of the multi-label SVM takes a long time. With 25000+ bigrams, it takes about 8 hours on a standard laptop to fit. This process is done offline.

The trained classifier is dumped into a pickle file to be used in the API. This pickle file is not small. For 25000+ bigrams it's more than 400MB.

A new question requires a title and a description body. Both contents are sent to the API using POST request to avoid encoding special characters in the URL. The processing of these data is

- Concatenate title and body
- Use BeautifulSoup to remove HTML tags
- Convert into lower case
- Tokenize the resulting string into an array of words
- Remove stopwords using the `nlk.stowords` list for English language.
- Convert remaining words into stems using `nlk.SnowballStemmer`
- Use the classifier's `predict_proba` method to get a vector of probability for each tag to be associated to this question.
- Select the 5 to 10 tags having the highest probability and display them to the user. Tags having a probability higher than 50% are displayed in bold.

END OF DOCUMENT