Natural language processing project

Advanced Numerical Methods and Data Analysis
University of St. Gallen

Course professor:
Dr. Peter Gruber

St. Gallen, 29.05.2018

Damien Baldy
damien.baldy@student.unisg.ch
17-620-626

Claire Fromholz
claire.fromholz@student.unisg.ch
17-624-602

Thomas Mendelin
thomas.mendelin@student.unisg.ch
13-051-156

A. The problem to solve and the way to do it

Many job ads website provide a brute description of the required job, without showing any sign at first sight of technological investment in helping the job announcers to qualify their jobs. A job hunter reading offers would often skip the corporate part, read the titles and go directly to the skills required, before investing more time in reading the exact description of the job. Indeed, if one doesn't qualify for a job, it means that either the job doesn't match his domain of predilection and he or she should look at other job ads, either the job offer has been poorly specified and he or she should read between the lines to get skills that will be required from him.

The idea of the algorithm is to build an algorithm that will be able to specify the key skills for a job without them being explicitly written in the job description. For instance, a data science job would likely require skills in Python, although the job might read "good programming skills". The real-case algorithm would therefore provide a set of tags to the job announcer that he can put on his job ad so that when a potential recruit reads job ads, he can focus on skills required before clicking on the actual job ad.

B. Implementation

1. Retrieving the data

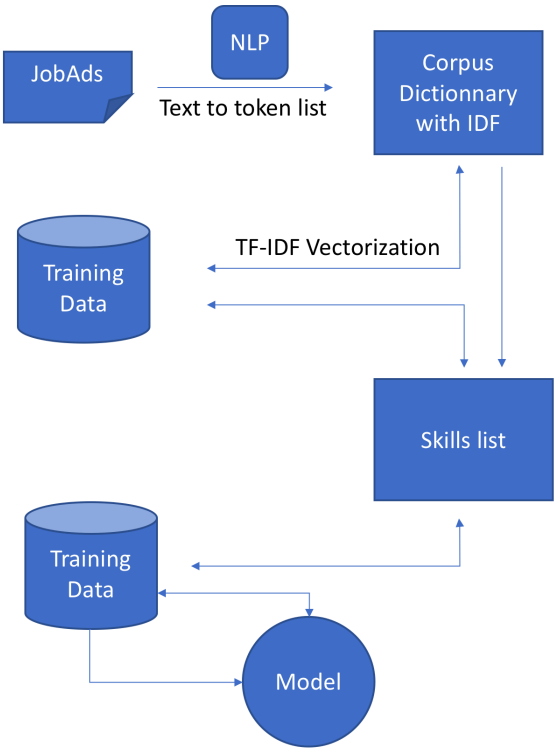
One of the main challenges for the project was getting a sufficient amount of training data. There are sometimes pre-classified training sets available in various repositories on the internet which directly can be used for training purposes. However, we decided to scrape our own data to make sure that the training data consist of a wide range of jobs and has real case descriptions. We scraped at around 30'000 job description from one of the largest job websites on the world, *indeed.com*.[1] The scraper has been set-up in R. It takes job search terms and location as input and exports a Json file containing an id, job title, job description and the search term. Overall, 40 job search terms were used as input. Furthermore, we focused on the United States as a location to get consistent data in terms of language. The search terms were chosen such that they cover a wide range of jobs meaning the training data has sufficient variation in it. For each search term a maximum amount of 1'000 descriptions was scraped.

In a first step, the scraper is looking for all individual job URLs presented in the search results of indeed.com. After gathering these URLs, it uses them to send a get request to each of the individual job ads and scrapes the description and job title. Once all descriptions from one search query are scraped the scraper moves on. The retrieved data is parsed to a dataframe object inside R and ultimately exported as Json.

There are various challenges when setting up such a scraper. First, the amount of get requests which are sent to server is quite high, meaning there is a potential threat of getting the IP temporarily banned from which the scraper is running. We implemented some safety measures for this case. This includes error handling for all get requests inside the scraper. In case a request already fails when retrieving the search query, the scraper breaks the loop, stops scraping and stores the already scraped data. If a request only fails for a specific job offering, then the scraper proceeds to the next. Other challenges include the correct formatting of the job descriptions and finding general and reliable xpaths for isolating the desired content in the search results.

[1] Indeed.com offers an APi but this is apparently for business purposes and there is an application process for it. Scraping may be harder than using an API, on the upside the output is customized to our project needs.

2. Text preprocessing



Our text preprocessing consists of splitting the text into words using a regular expression to form the tokens. We then remove the stop words and stem the tokens with a Porter stemmer.

We create a corpus of words by merging the tokens from a representative sample of job ads (about 4000 ads) and we use a TF-IDF computation to qualify these words. Then we filter those words based on 3 criteria: the word should be present in at least 1% of the documents, at most in 90% of the documents and should occur overall at least 100 times. This constitutes the useful vocabulary that will be used to vectorize our job ads.

After this vectorization, we generate a TF-IDF vector for each job ad that we've scrapped based on the predefined vocabulary. We define the main skills in each domain using the most relevant words (based on TF-IDF). Then we get the training data from testing for each skill if it is present in the job ad or not and inputting 0 or 1 depending on the case. The figure above sum up all the tasks realized for the first preprocessing and elaboration of the training data.

3. Learning

We split the training data into a training and test set and transform them, so we get as features the 1423x1 TF-IDF vector and as class to predict one of the skill from the skill list. For each skill, three types of algorithms are tested using the sklearn library[2]: Logistic regression, Naïve Bayes and Support Vector Machine. For each of this model, a list of parameters is tested to adjust each model to the specific skill to classify.

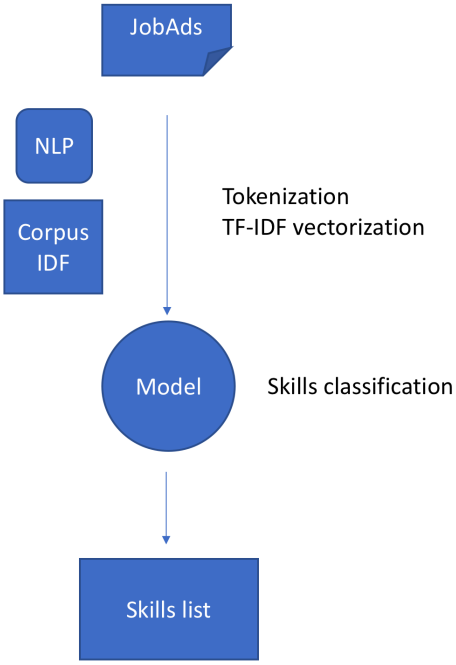
For the regularized logistic regression, we optimize the parameter C which is the learning rate from the following loss function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1).$$

To implement the support vector machine, the SGDClassifier (Stochastic Gradient Descent) with loss function = "hinge" is used and one of the most relevant parameter to adjust is alpha which is the learning rate. Finally, for the multinomial Naïve Bayes the N parameters (N features) are computed based on frequency appearance and smoothed according to the parameter alpha that need to be adjusted.

To define which parameters are the best, we run all the models on a ten-folder cross-validation and select the parameter which provide the best accuracy. After each model has been adjusted to the skill, we select the best model according to its accuracy rate on the test set (data which are still unknown by the model). We get then a list of models adapted for each skill.

[2] sklearn : http://scikit-learn.org/



is available in json format.

C. Results and going further

We worked mostly using a tiny portion of our dataset to test it and make sure each portion of the code was working on a small scale. We got already decent results with a small-scale dataset. Unfortunately, the whole dataset is too big and too memory consuming for our algorithm to run on it properly. One idea to solve that would be to run the modelling on a server, because we cannot dedicate our computers to a few days of full time calculations. Our algorithm isn't optimized enough to give efficient results and have decent calculation times.

Several choices were made that could have been different. The set of job ads we scrapped was on the US job market and was extremely diverse in terms of domains. Maybe it would have been more relevant to scrape all the job ads in one domain and provide very specific hints for job ads. We also considered that skills or keywords were individual, while it is obvious that sometimes skills are described using 2 words or more. Next step would be to take into account expressions and context using CBOW(continuous bag of words) and skip-gram methods. The Word2Vec two-layer neural net could ease the implementation of these methods and ease the word embeddings. Taking more time to study the data would maybe have enabled us to improve the stop words used, but we got satisfactory results already using the stop words from the library *nltk*.

4. Back testing the learning

The learning provided often the keywords that were already present inside the text, but also, we able to provide with new keywords, which was the point of the algorithm: isolate the keywords and add new keywords. For instance, medical job inside a hospital proposed the keywords out of the keywords that were not present inside the job ad: "servic", "minimum", "care", "develop".

This is interesting for two reasons: the job didn't specify that a minimum service would be required, yet it was called "Emergency Medicine Physician", which is extremely likely to contain a minimum service. Also "care" is obvious and could categorize the job well. However, "develop" is too generic and shows that we have to increase our stop words to improve the quality of our keywords and our propositions. A set of results from the learning displaying words that are NOT present inside the job