Code Challenge two: Web crawler for job search

Dhruba Pujary

October 2019

1 Crawler

Basic web crawling algorithm: Given a set of seed *Uniform Resource Locators* (URLs), a crawler downloads all the web pages addressed by the URLs, extracts the hyperlinks contained in the pages, and iteratively downloads the web pages addressed by those hyperlinks.

Challenges:

- web is very large and continuously evolving
- content selection based on relevance, job posting in our case.
- avoid misleading/spam web pages

The program that queries any URLs should consider the following:

- Web page content may be get updated frequently. Thus we need to revisit the page frequently.
- queries sent to a web server should not be very frequent which may lead to denial-of-service attack.

This part of problem related to web crawling is very vast. Therefore, we will not focus on this portion instead we would like to optimize the crawler in a focused topic using NLP and machine learning. [1]

To start, the crawler should select the known website for Jobs such as LinkedIn, Monstar etc and collect URLs for company profile career page and store it in a database. The database can have the following field: Unique ID, URL, web server ID or IP, last visited or a flag that is refreshed periodically stating the page should be revisited as its stored contents are old, priority, relevance/topic etc.

Our crawler should be able to traverse the web and select job related pages. This is a classification problem. A dictionary based approach will be very simple where we store a list of keywords that are common to job posting. For example, *offer*, *job*, *salary* etc. Keywords may be stemmed for better performance. If

[1] Here is a survey of 2010 giving a good overview of different aspects of web crawling and the challenges.: `http://infolab.stanford.edu/~olston/publications/crawling_survey.pdf`

one or more keywords occur in a page, then it is likely to be a job posting. We can define a threshold of how many keywords should occur at least to classify it as relevant. Also, we can make this approach language independent by keeping similar keywords for every language we want to address. We can identify the language of the web page using any of the methods described by [1] [2]. We can initialize the dictionary with most common known words and later based on frequency of most common words in job posting pages we can prioritize our keywords. We can use different dictionary for different language.

Pros: Dictionary based method will be very fast as we only need to search the keywords occurrence to qualify the page to be a probable candidate of job posting. Cons: We need to identify the keywords to search. Assuming we have very fast searching mechanism, the problem is that pages other than job posting might also have these keywords. We can mitigate this problem by using a document vector model. Given a training set containing documents of job posting and non-job posting we can train a multi-layer perceptron (MLP) to classify any given document into respective binary label, i.e 1 if job posting or else 0. For practical use case, we first can use the dictionary base approach to select the probable certain job posting and later filter using document vector model.

Note: We process and identify the important page data(main content) in the HTML page using tags etc. Tools: `https://www.scrapehero.com/python-web-scraping-frameworks/`. We also assume that extracted data from the web are well formatted, i.e clean and usable. For example, because of poor extraction, the sentential structure may get disoriented. We describe an approach in the next section to clean such data.

2 Analysis and Extraction

In the above section, we have developed a way to crawl the web and consider only the relevant pages. In this section we will describe ways to analyze the page and extract the relevant content.

Our first assumption was web page data extracted are well formatted. However, this is not always the case. One common error is occurrence of end line \n character at non-relevant positions. This can be solved using a language model $p(w_t|w_{1:t-1})$ where we evaluate probability of occurrence of the newline character (or any ill represented character) given a sequence of tokenized strings. By defining a threshold, we can filter such ill occurrence of ill characters at any position. We can also pre-process the text with spelling correction models to enrich the data[3].

For a job posting, we need to extract the following fields: *minimal requirement*, *skillset*, *location*, *job title*, *salary*, *employment type* and *perks and benefits*.

[2]Other references: 1) `https://github.com/topics/language-identification`, 2) `https://medium.com/coinmonks/language-prediction-using-deep-neural-networks-42eb131444a5`

[3]theory: `https://web.stanford.edu/~jurafsky/slp3/B.pdf`, Implementations: `https://github.com/mdcramer/Deep-Speeling`

Among the following fields *location* and *salary* can be extracted using a pre-trained *Named Entity Recognition*(NER) tools. The open-domain NER system assign LOC and MONEY to these words . But fields like *requirement*, *job title*, *employment type* and *skillset* would require supervised dataset. The labels associated with these fields could be minimal, title, e-type and skill respectively. For example, Bachelor's or Masters, CS are of minimal type entity; Developer is of title type; full time, contract, etc are e-type; and problem solver, ASP.NET etc are skill type.

Model architecture for NER using Bi-LSTM and CRF with pre-trained contextualised embeddings such as ELMo/BERT have proven to be very good in many domains[4]. However, for the network to be most effective the pre-trained embeddings should be related to the domain. This can be achieved by fine tuning the pre-trained model specific to the domain and would require textual data of decent size (100k sentences or more). The fine turing does not require any supervision because these language models are trained in unsupervised manner.

There maybe cases where the labels associated to entity type requirements and skills may be interchanged depending on the job profile. To validate this, we can keep count of skills and requirements which occur in both type. Our assumption is that the trained NER model will be able to distinguish skills and requirement correctly. However, if it fails to do so, then we can train another neural network (MLP) which will take a predicted skill or a requirement and job title as input and do a multi-label classification of skills and requirements. We can train such a network separately for instead of using a pipeline architecture which will allow both network training simultaneously. The second network will have dependency on the output of the first network but will be less troublesome than stalling first network training.

Finally, we need to address the perks and benefits. This portion can be addressed by NER but depending upon company to attract talent there could be wide variety of perks and benefits. This will result in poor performance of NER alone. The motivation behind putting perks and benefits is associated with the happiness of the employee. Hence, these should associate a positive sentiment. Our solution could be sentiment analysis of text (or assigning a binary label to a given sentence). We need to build our own dataset where we mark sentences with perks and benefits as 1 and others 0. For sentiment analysis we could use a Convolutional Neural Network with a binary classifier. The input to the network will be pre-trained word embeddings [5]. The feature extractor will recognize to assign sentence with words related work related like experience, knowledge etc a set of feature different from sentences with words related to benefits like holiday, fun party etc. A detailed survey for sentiment analysis [2].

As of now, we have a system that can extract data from a web page related to job. Next step would be to analyze the extracted data to further enhance our extraction process and overall usability of the system. This includes improving

[4]Other networks are: https://paperswithcode.com/task/named-entity-recognition-ner

[5]References: `https://d2l.ai/chapter_natural-language-processing/sentiment-analysis-cnn.html`

and increase our training data as well.

First and foremost requirement is to have a dataset. To build a system without a supervised data and later enhance the system based on predicted data would be a difficult task. We can minimise the involvement of human inspection in later stage as we filter our prediction. Assuming we don't have any supervision dataset, we can use open-domain NER systems to extract the entities which will be categoried to different types different from our requirement. We can keep account of most frequent type are relevant to job related keywords. Then we can prioritize the types based on total frequency of the keywords relevant to job. Now, any keyword predicted by the open-domain NER system of the top k type of entity can be assumed as a job relevant entity. Further, using a final annotation by human intervention, we can update the keywords with relevant type for our requirement[6].

To assist the annotation process further we can use a graph structure. We define nodes for each of our entities, namely job title, skills, requirements, salary (range) and locations, and build edges only between each job title and other entities. Each graph node will have attribute *ID* with which each node is referred, a *name* field and *type* field for the type of entity. We can build a vocabulary for names for each type of entity and store the graph node ID for easy access.

In this stage, we use our own NER system which can predict the confidence estimate of the tagged entity. We first use some annotated data to train our NER system. Using the weakly trained NER, we predict the probable entities in a document and also use the confidence estimate for correction.

To prioritize the importance of each entity found by NER system relevant to the job, we can use two solution. First is we build the graph using the title and found entities. We can use the frequency of co-occurrence of type of entity to job title and weight the edge in the graph. This is assuming our NER system predicts correct entities. Or we use the predicted entities word embedding space to extract top k nearby words and keep a count of those for each document. The most frequent nearby words will capture the most common sense of description the job. We then prioritize the predicted entities with a similarity score to the most frequent words. We use the priority and the co-occurrence statistics to weight the edge between title and entities. We also have to keep in mind of entity type while using the priority score for calculating the weight. After following one of the solution, we can prune the graph nodes whose degree is less than a threshold. This will remove association of erroneous nodes.

Now using the prediction of open-domain NER prediction, trained NER prediction and graph structure we can extract and enhance our annotation system with gradual decease of human involvement. Especially, the graph structure will allow to identify most relevant entities as keywords before even process a web page and can aid in decision making for qualification of found entities. Few properties of the graph are following: The degree of the nodes of the graph is associated to most common entity, for example, knowing Microsoft office or using G-suite is very common skill. The least connected nodes could be a erroneous

[6]Here is a tool for annotation: `https://brat.nlplab.org/`

node or a very specific detail. Weights between the edges between job title and similar type of entity denotes the importance of the entity for the job title. Any two adjacent nodes will have at least one job title node by definition. Any node of similar type will be $2n$ hops away, however the similarity score is not linear because of weight between the edges.

For our final system, we want it to be robust to language specification. This can be achieved by using a dictionary for translating the nodes in the graph. We can add another attribute of translated names for each language. Then we can build a dataset for training the NER specific to that language using the translated graph and crawled web pages. We can verify our NER method by either visual inspection of a few results by a language expert. Or we can use our english dataset and translate to other language using SOTA machine translation (MT) model[7] and verify our prediction using the dictionary and graph. However, this process is dependent on quality of our MT model. Similarly, for our sentiment analysis model we build a dataset using translated text and assign 1 or 0 depending on the labels of English. For prediction, we can replace our choice of neural models based on the language of the web page data and extract the information.

Further enhancement on our base model is to use graph embeddings trained simultaneously with document vector in Siamese network architecture. This will make the generation of document vectors in similar embedding space as the graph embedding. Thus will allow the removal of dictionary based decision making for crawling only relevant topics altogether. For graph embedding we could use Graph Convolutional Networks or Relational Graph Convolutional Networks which is a composition function for graph and can be trained in supervised manner.

Note: All footnotes refer to web pages and PDF are relevant for further research and understanding.

References

[1] Tommi Jauhiainen, Marco Lui, Marcos Zampieri, Timothy Baldwin, and Krister Lindén. Automatic language identification in texts: A survey. *CoRR*, abs/1804.08186, 2018.

[2] Lei Zhang, Shuai Wang, and Bing Liu. Deep learning for sentiment analysis : A survey. *CoRR*, abs/1801.07883, 2018.

[7]https://paperswithcode.com/sota/machine-translation-on-wmt2014-english-german