CSCI 4/5253 -Datacenter Scale Computing
Final Report
The Job Solver

Team Members:
Ganesh Chandra Satish
Rahul Chowdhury
Si Shen

Date: 12/13/2018

Introduction

What is the problem that we are trying to solve and the motivation for the problem?

Job search is one of the *tedious things* everyone has done at one point or another whether they are new to job searching or experienced. We as a team have worked on alleviating the problem for job seekers by providing solutions to two of the issues faced by them. The first problem we want to address is focused towards new job seekers entering the field and trying to *align their interests with the existing jobs in the industry*. Many students starting off their career have specific interests of what they would like to work on, but are not sure of what is the job title for their "dream job" in the industry. It takes a while and considerable amount of research to get the answer. It can also happen that they get into the company thinking it is their job only to find out later that its not. We want to solve this problem faced by many students We have created a system where the user can type his/her interests and describe the work they like doing and we would provide them with the job title in their industry that is best suited to their description.

The second problem we wanted to solve was *reducing the amount of personal time* we invest in reading of a given *job description*. One of the issues when we are finding jobs is that we have to read through tons of job descriptions which include company's goals, mission statements, etc.. It would be a lot easier for the user if only the necessary details are read in a job description. We have provided a solution for this by developing another tool which *highlights the most important words* in a given job description. So that the user can skim through the job description and read all the important points.

Why this is a problem which needs to be solved on a datacenter scale?

To find these important words or in the prediction of the job titles, a lot of *computation* has to be done on *large job details data* to generate an accurate machine learning model. There are about

300,000 jobs that we get through a crawler and the obvious choice of storing this huge dataset is in the cloud! Amazon EC2 to the rescue and MongoDB as database for faster indexing of data.

Also, as Apache Spark is of the best Big Data frameworks to handle huge amounts of data and run inbuilt libraries for machine learning right on the cloud to get faster and getting results with an initial accuracy as high as 79.2% .

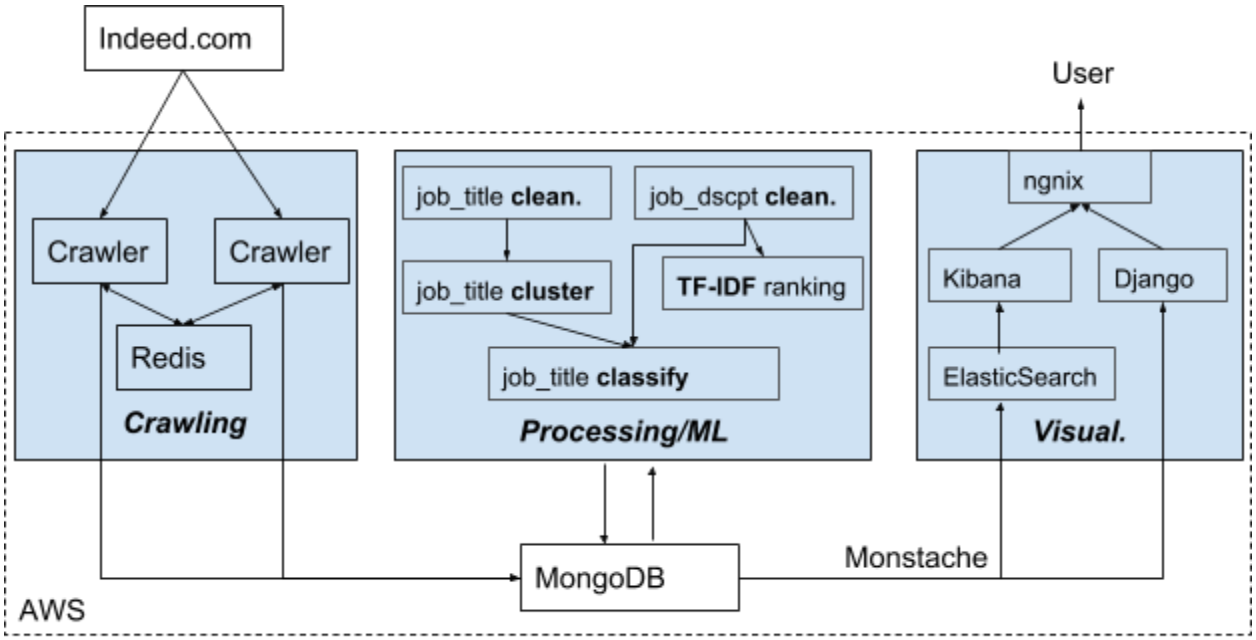
In fact, even our visualization pipeline was set up in the Amazon EC2 instance. All the amazing visualizations and dashboards in Kibana that you saw in the demo video are actually living in the EC2 instance!

Related Work:

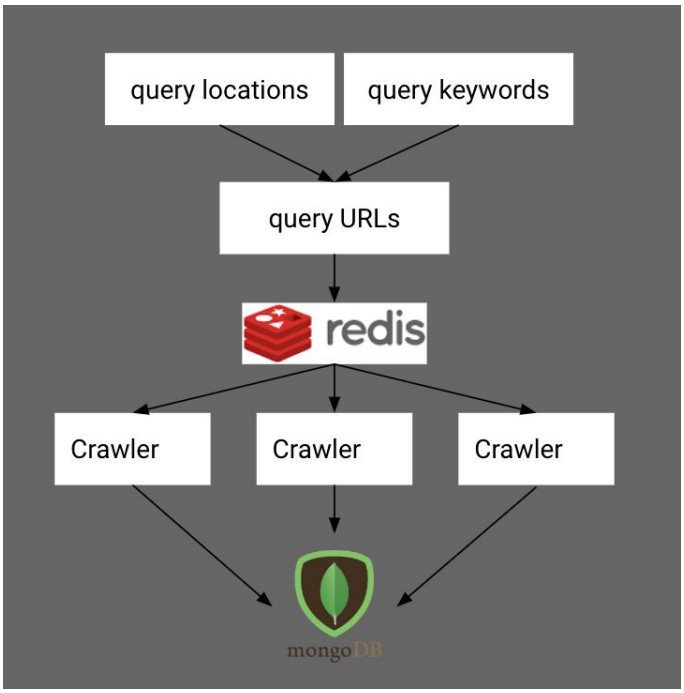
1. While finding related work to our problem statement, we found a blog on Kdnuggets that predicted the job titles from descriptions. (Here is the link: https://www.kdnuggets.com/2017/05/deep-learning-extract-knowledge-job-descriptions.html)
2. The approach here focussed on implementation of a neural net to map the descriptions to a job title.
3. The second approach proposed in the paper "Skill2vec: Machine Learning Approach for Determining the Relevant Skills from Job Description" describes skill embeddings where similar skills are clustered together.(Link: https://arxiv.org/pdf/1707.09751.pdf)
4. We have focussed on producing better features by data cleaning and k-means clustering to produce better labels and providing an approach using big data which makes our system scalable to train against much larger datasets without using GPU's.
5. Our second problem statement of providing highlighted words in a given job description is a ***novel idea*** that has not been applied before on any kind of job dataset.

System Design:

Diagram of how the components of our system work together.



How we made our dataset and the cleaning(or sanitization) done on the crawled data?



The primary step to solve our problem was to actually create a authentic dataset. Hence, instead of picking up the already existing datasets on the web(by the way, they are awesome too), we

decided to make our own dataset by developing a crawler which gathers data according to the configuration set by us from Indeed.com. We built a data crawler which crawled through Indeed.com and retrieved around 300k jobs. We also deployed *3 crawlers on 3 EC2 instances t*o retrieve the data in a faster way and leverage the power of the cloud!

Here are some of the tools we used for our data crawling process:

- We used **beautifulsoup** library to implement the crawler as that's easy to understand and implement. And Indeed.com is relatively easy to crawl as unlike other websites, it doesn't block automated programs hitting it continuously over the period of time.
- We used **Redis**, an in-memory database, to synchronize crawlers, because it's very fast and easy to use. Actually Redis is widely used in distributed crawlers.
- We used **MongoDB**, a document-based database, to save the crawled data, because it's schema-free, which allows us to create new fields on the fly. Actually we created a lot of new fields in data processing and machine learning period.

The *basic steps for data crawling* are:

1. create a list of querying keywords, e.g. "software engineer"
2. create a list of cities, e.g. "San Francisco, CA".
3. create a list of worktype and experience level, e.g. "internship", "fulltime, senior_level"
4. combine these lists into a long list of URLs, which are the start point
5. create a list of querying keywords, e.g. "software engineer"
6. create a list of cities, e.g. "San Francisco, CA".
7. create a list of worktype and experience level, e.g. "internship", "fulltime, senior_level"
8. combine these lists into a long list of URLs, which are the start point of each crawling task
9. feed these URLs into Redis as a task queue
10. each crawlers get a URL from Redis queue and start crawling. 1-2 seconds interval is set between each request to reduce traffic burden caused to Indeed.com. Each piece of crawled job information is saved to MongoDB.
11. Redis also maintained a set of crawled job ids, so duplicated download is avoided.

After crawling, here's the *initial schema* of our dataset:

We got 334000 jobs from crawling and saved in MongoDB. The basic fields are:

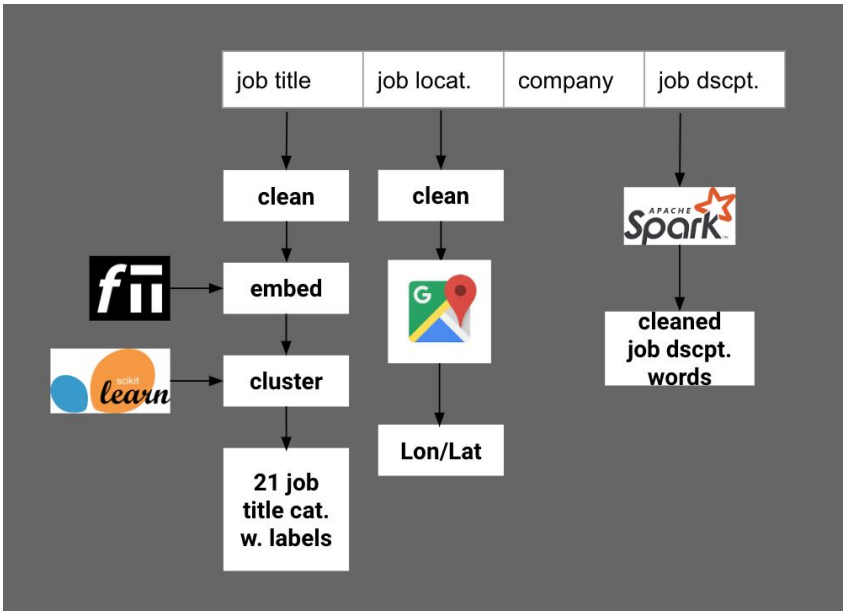
- "_id": job id, internal job id of Indeed.com
- "query_keywords": keyword used to query this job.
- "query_location": the location used to query this job.
- "query_jobtyp_explvl": job type and experience level used to query this job, e.g. "internship", "fulltime+entry_level", "fulltime+mid_level", "fulltime+senior_level"
- "job_title": job_title
- "company": company

- "job_location": location of the job
- "job_description": the long description of this job

And here are some of the problems with the dataset:

- job_description was a long string with lots of meaningless stop words. So it needs to be cleaned.
- job_title is in nonstandard form, as a result, 334K jobs have 100K different job titles. So it needs to be cleaned/clustered into categories.
- job_location is also nonstandard, and needs to be cleaned.

The data preprocessing that we did on our dataset:



The tools we used and in what context we used them:

- **Sci-kit Learn** is a powerful machine learning library for Python. We used that to perform clustering for job_title.
- **Google Geocoding API** can convert address to longitude/latitude, which is useful for coordinates map in Kibana visualization.

Tasks we did:

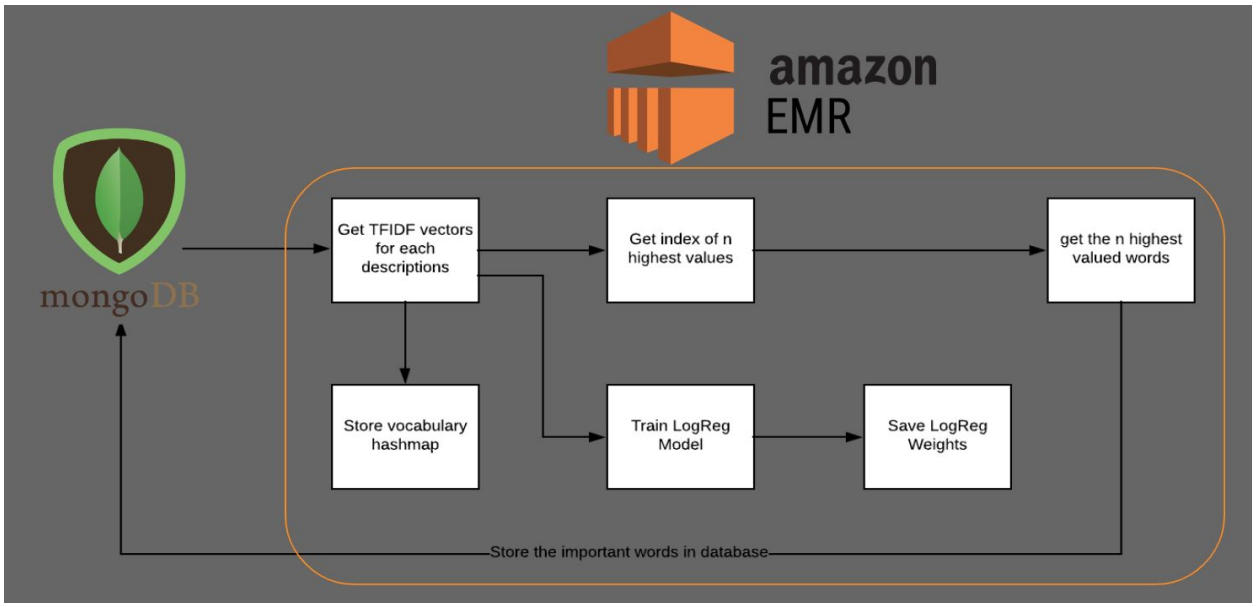
- *job_title cleaning/clustering/labeling*
  In order to perform job_description to job_title prediction, we need to make standard job titles.
- The first step is to clean job titles, e.g. to remove non-alphabetic characters, to remove words not related with job types, e.g. “senior”, “intern”, “contrast”, and to filter out jobs with too long job titles. But we still have about 12000 job titles after that.
- The next step is to cluster job titles. We used FastText (a word embedding tool) to

vectorize each word in job titles by words meanings, and then calculate the vector of each job title by averaging word vectors. Then we performed the clustering by k-means algorithm.

- By manually checking clustered job titles, we find the results are generally meaningful. But there are still too many clusters (we set 200) . There are multiple clusters with similar job titles. And there are also some clusters not separated well. So the final step is to manually check each cluster (luckily we have only 200 clusters), and merge, filter, label the clusters. In the end, we get 21 job categories (standardized job titles) with about 200K jobs.

- *job_location longitude/latitude conversion*
- cleaned job_locations (e.g. removed zip code)
- got job location longitude/latitude by Google Geocoding API
- uploaded longitude/latitude to mongoDB

The Machine Learning component of our architecture:



The following tools were used for our Machine Learning component:

- *PySpark* is a unified analytics engine for large-scale data processing by making use of memory in distributed nodes. It has *Spark-SQL* to provide easy accessing to MongoDB and *Spark-ML* to train machine learning models with large scale data. So it is perfect for our project in job title classification (job descriptions are of huge size)

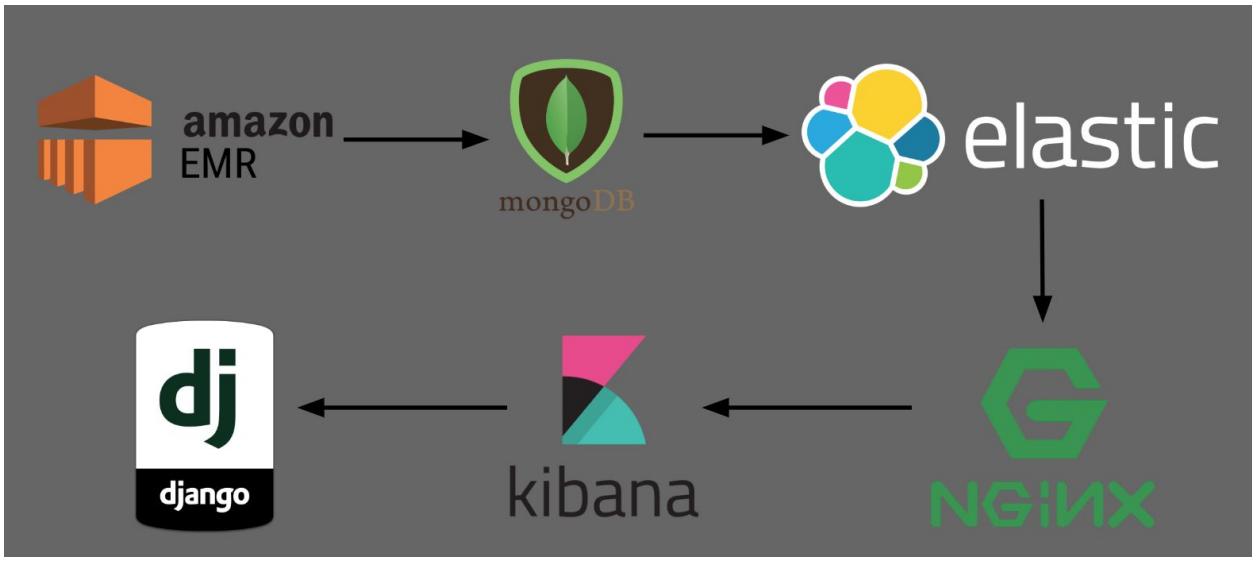
Steps involved in our Machine Learning pipeline:

1. Job descriptions and cleaned job titles were loaded as a dataframe.
2. TF-IDF vectors were generated for each descriptions using pyspark's Countvectoriser

and IDF functions.

3. Top 100 values were extracted from each vector.
4. The values were mapped to words to extract 100 most important words for each job description.
5. These important words were stored in mongoDB as a separate column.
6. The above vectors were also used to train the Logistic Regression model.
7. The weights and the intercepts from the Logistic regression model were stored.
8. These weights and intercepts were later used to predict the job title based on the input description.
9. Another approach we did was to extract the top 1000 important words for each category(Job title). These 1000 words were extracted using the 1000 highest valued weights of the logistic regression classifier for each job title and finding which word in the vocabulary this weight mapped to. The idea is to just highlight those words from the job description that fall into this list of 1000 words.

The Data Visualization architecture of our project explained:



The *tools* we used:

- *ELK* stack (ElasticSearch, Kibana). It's perfect for our project because it provides searching, filtering and various types of visualization.
- *Monstache* (a tool to sync data from MongoDB to ElasticSearch)
- *Django*: Django is a famous Python web framework. We used that to do tasks that can't be done in Kibana.
- **Nginx:** We used Nginx as a reverse proxy server for Kibana and Django servers for security reasons. We don't want Kibana directly exposed to Internet, as malicious users

may attack our server or modify elasticsearch index.

The *steps* that we performed for out data visualization pipeline:

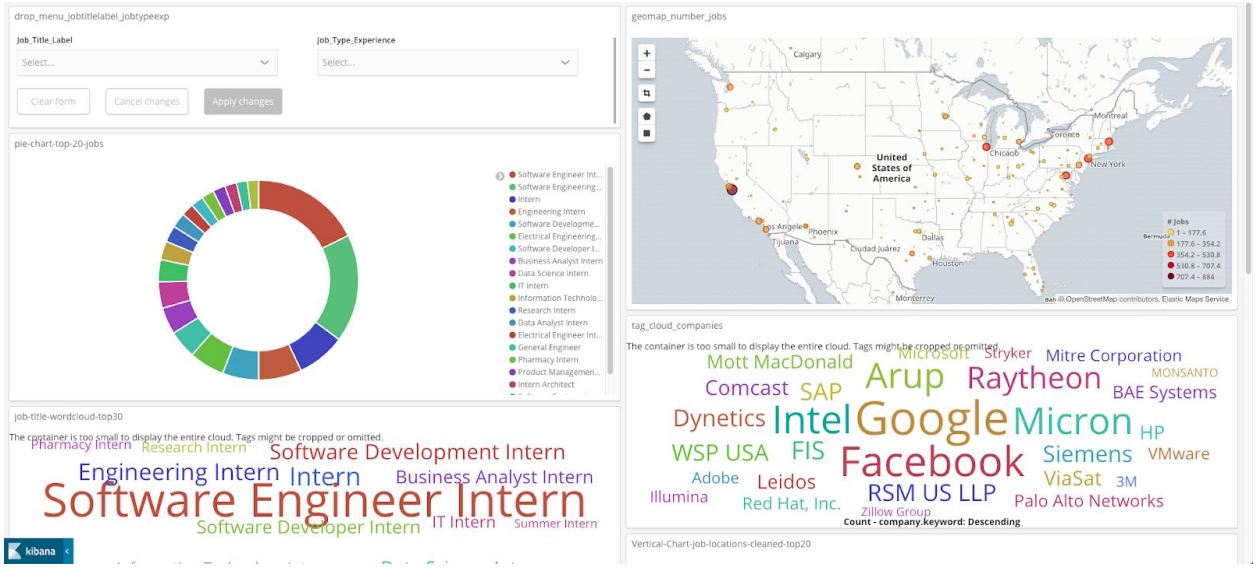
1. Used Monstache to sync MongoDB to ElasticSearch
2. Connected ElasticSearch with Kibana
3. Created various types of visualization in Kibana, including pie charts, coordinates map, word cloud, data table, metric, heat map and bar charts.
4. Created a separate web application with Django as the interfaces to show job-description and to predict job titles. In the job-description page, we used important words sets we got from Machine Learning part as highlighted words. We have different highlighting options by using different set of important words. In the job-title-prediction page,we used the trained model from Machine Learning part to predict the job title.
5. Configured Nginx to set password for accessing Kibana/Django.

There were a ***few false starts*** that were encountered while we were beginning to research the solutions:

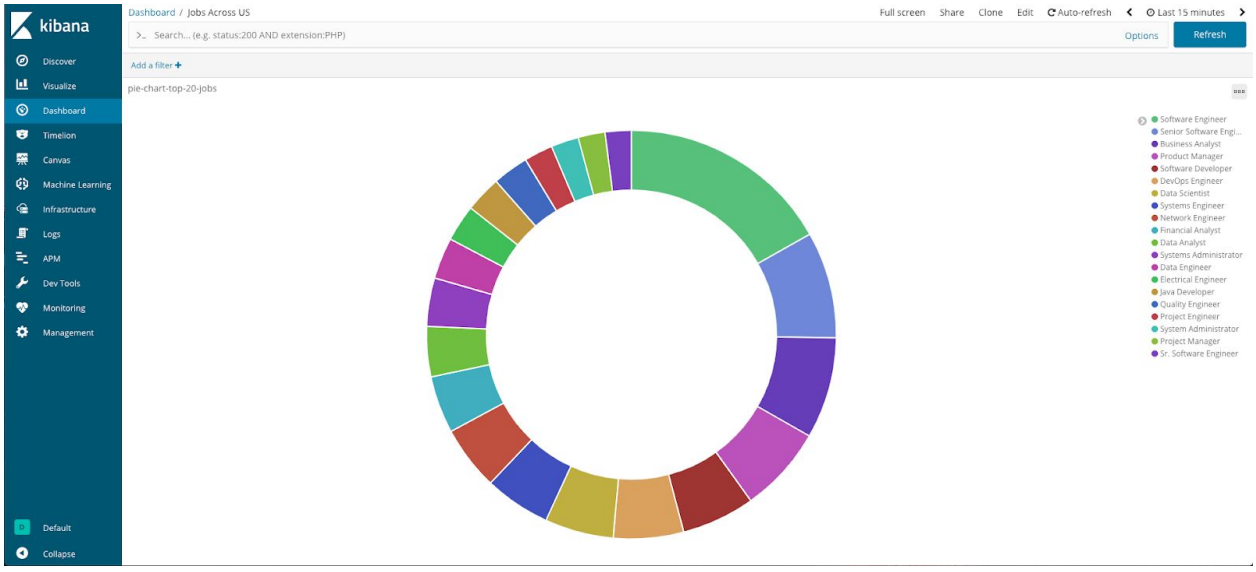
1. We started implementation of a Neural Net model, which took considerable amount of time to try as the model took long to train. If we would have started with simpler models such as naive bayes and went from there, we would have tested a lot more models before arriving at the best solution.
2. The initial architecture for job title prediction had a lot of moving parts which made the system very bulky. Once we figured out that we would be needing EMR clusters only for training the model and by extracting and storing the weights as an array, we could write much simpler code and design a much simpler architecture to implement the same.
3. When considering how to sync data from MongoDB to ElasticSearch, we first tried mongo-connector as it's the most famous one and is provided by MongoDB. However, after spending a whole day trying, debugging and online searching for solutions, we still couldn't run it. In the end we realized it's the version problem -- mongo-connector doesn't support ElasticSearch 6.x. Luckily, we find the alternative one soon, monstache, though not well-known, but it works perfectly and is very easy to use.

Evaluation/Findings:

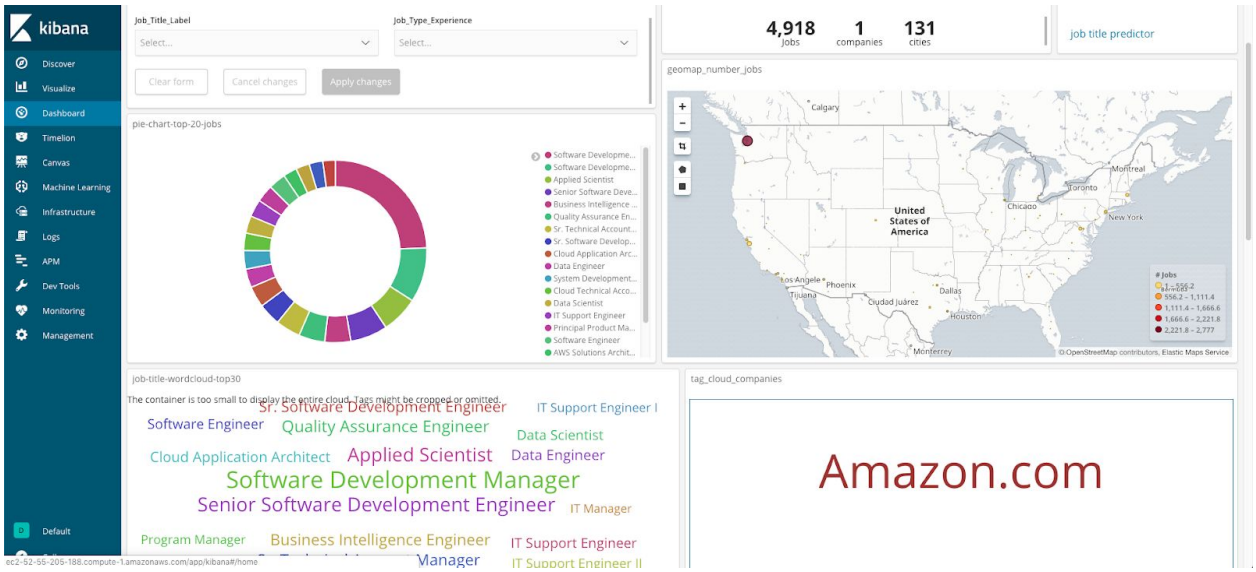
1. **The Kibana dashboard that shows the different types of visualizations we have for the entire 300k jobs in our dataset.**



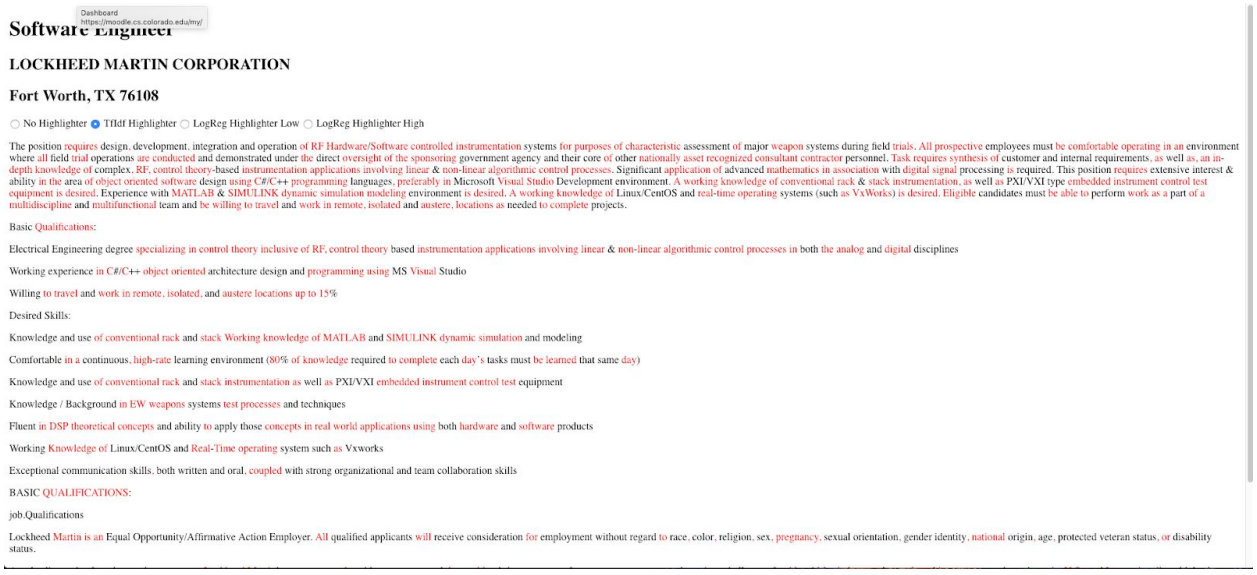
2. **The Kibana pi-chart showing distribution of jobs throughout U.S.A.**



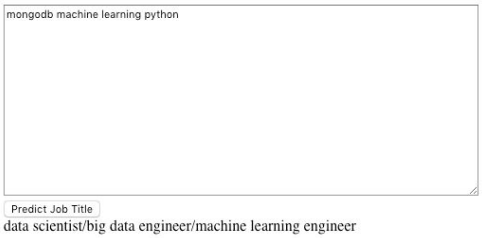
3. **The Kibana dashboard showing visualizations for a specific company, let's say "Amazon".**



4. **Our system showing the most important words in a job description using different ML methods such as Tf-IDF and Logistic Regression model.**



5. **Our system predicting the required job title when searched with a job description.**



NOTE: Here's also a link to our demo video:
https://drive.google.com/file/d/1CjqaA8UfvxAe7qIzHsZnCyH2qUkv9yQB/view?usp=sharing

Review of each Team Member's work:

Member	*Work*

Si Shen	- Completed the data crawling component
	- Managed MongoDB database
	- Worked on the data processing/machine learning component
	  - performed job_title cleaning, vectorization, clustering and manual labelling
	- Worked on the visualization component
	  - sync MongoDB to ElasticSearch by monstache
	  - job location longitude/latitude conversion
	  - some visualizations in Kibana
	  - Django web application
	  - Nginx deployment

Ganesh Chandra Satish	- Machine Learning on PySpark:
	  - TF-idf vector generation
	  - Generated important words for each job description
	  - Trained Logistic Regression model with our dataset
	  - Extracted weights of the ML model and the vocabulary.
	  - Implemented an independent service that used the above weights, intercepts and the vocabulary to classify an input description to a job title.
	  - Implemented Logistic regression:
	  - With only CountVectoriser.
	  - With tf-idf

	- With cross-validation Compared the accuracies and found that logistic regression with cross-validation provided the best accuracy
Rahul Chowdhury	- Was responsible for data cleaning after the data crawling was completed. - Did statistical analysis(built word counts, got statistical summary for each category in the data) and machine learning review for our project. - Researching and comparing the best Machine Learning model to fit our dataset. - Indexing MongoDB data into ElasticSearch. - Creating the visualizations and the dashboards in Kibana.

The timeline of our project for the semester so far:

Timeline	Data Crawling	Preprocessing	Machine Learning	Visualization
Week 10/29 - 11/02	Crawling code completed.		Research on best model to implement to classify job descriptions to job titles	
Week 11/02 - 11/09	redis added in		Tried implementation of neural network to classify job descriptions to job titles.	
Week 11/12 - 11/16	completed	Stop words removed and word clouds generated accordingly	As neural nets was a heavy model to train, started working on implementation of logistic regression	

Week 11/19 - 11/23		Cleaned job description column added to the database	Logistic regression model implemented. With an added problem statement of storing important words in MongoDB for highlighter.	
Week 11/26 - 11/30		Completed	Testing various options with logistic regression to increase accuracy.	ELK setup in EC2 instance.
Week 12/03 - 12/07		job title labeling done	Extracted weights, intercepts and implemented the simple service the classified a given job description to a job title using the weights from the above logistic regression model.	Visualizations and dashboard made.
Week 12/10 - 12/13		(google geocoding done) completed	completed	completed

Conclusion:

Over the course of the project it turns out that the base paper we were trying to implement, we did that with a different approach(used Logistic Regression instead of Deep neural nets) but also stumbled upon a *greater and unique problem* of mapping important words in a job description. We addressed that problem first and got surprisingly good results.Using Logistic regression with count vectoriser and idf the accuracy came upto 72.7% Using Logistic regression with only count vectoriser the accuracy came upto 72.89% Using Logistic regression with cross validation increased the accuracy to 79.20%

Future Work:

One problem we noticed is that currently, our dataset has a high volume of job descriptions for the title of "Software Engineer". Thus our model is a little biased towards that label. In future, we would like to first crawl and collect a dataset, that has an even distribution per job title and covers a larger area of jobs than to focus more on computer science. We would also like to increase accuracy of the prediction model by using other neural net models such as Long Short-Term Memory (LSTM).

For our second problem statement of highlighting important words in a given job description, We would like to make this more general and develop a chrome plugin, where you can click on this plugin to highlight the important words in anything that you are reading. For example a news article, a blog, a stackoverflow answer and many others. We would also like to improve the accuracy of the system that gives the importance value for each word. Currently we have tried 2 approaches, but there are many such we can.

References:

1. http://www.tfidf.com/
2. https://simple.wikipedia.org/wiki/Logistic_Regression
3. https://www.statisticssolutions.com/mlr/
4. https://spark.apache.org/docs/2.1.0/ml-classification-regression.html#multinomial-logistic-regression
5. https://spark.apache.org/docs/latest/ml-features.html#tf-idf
6. https://spark.apache.org/docs/latest/ml-features.html#countvectorizer