Recommendation Systems: An Implementation Guide

Author Names
Armine Khachatryan
Vanessa Klotzman
Marvin Harootoonyan

Professor Mike Boctor
California State University, Northridge

Chapter 1

Abstract

Recommender systems, also known as recommendation systems, are one of the most successful applicant machine learning technologies because they are capable of predicting the future preference of a set of items for a user and their top items. As we move towards more of the modern world, everything in society today is influenced by some form of recommendation from friends, family, or reviews. For instance, it has become increasingly important to mine data to serve personalized rankings to users who use video streaming sites and e-commerce stores because it can lead to an increase in revenue or usage.[1] Google Play Store install rates increase by 2% after improvements were made to the Play Store recommendation system. This leads to the point that recommendation systems are everywhere and are responsible for a huge portion of the modern economy, as they have a direct influence on a company's sales. It works by not only understanding a customer's actions, but it also starts off with an assortment of data about every user who uses the application and then it can figure out the user's individual tastes and interests. Finally, the user event data will merge with the data of all users who have applied to at least one job. We can think of recommendations as a matching problem. Given a set of users and items, we match the users to their preferred items. Therefore, this idea leads to the idea of the main focus of this text. The main focus is the development of recommendation systems for jobs, as job recommendation systems have become the [2] primary recruitment channel in most companies. They help decrease the recruitment time and match job hunters to the role that is most significant to their interests and background, as job hunters will be applying to more significant roles. Within this paper, we will introduce the problem of why recommendation systems are necessary for job hunters, and why this is an essential tool for the market. After, we will give a brief description of the previous work, current work, and procedures we are using to solve this dilemma. Then, we will give a brief project description, which will lead to the description of the reference architecture of building a recommendation API for matching job hunters to their preferred job. Additionally, we will describe the implementation and develop implementation guides for building this API, as well as utilizing Google

Cloud Platform and Amazon Web services. Finally, we will summarize what we concluded from this exploration of developing an implementation with two different cloud technologies, problems encountered and solved, approaches, what we can do better, and what can be done for future work. As we move into a more tech savvy world, machine learning will be able to solve any problem a human can solve. One of the many problems it can solve is developing an optimal solution for building a personalized recommender system for candidates.

Contents

Chapter 2

Introduction and Background

2.1 Statement of Problem

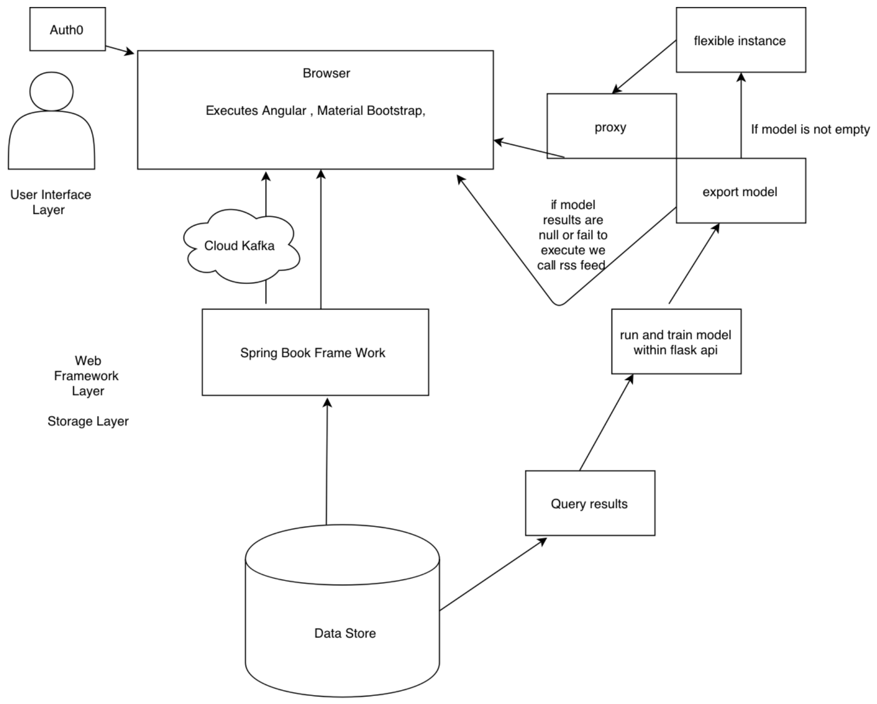
The job market is highly competitive, and personal preferences cause people to change their career at one point in their lifetime. Moving to a new job is not easy, as numerous factors need to be considered, such as salary, location, description, etc. Making a successful career decision is essential for the continuation of a successful professional career. It can be unfortunate while job hunting because job hunters will suffer the inappropriateness of being given information irrelevant to them, which causes companies to hire the wrong applicant and have a high turnover. This is due to a recruiter reaching out to the wrong candidate, asking irrelevant questions, going silent, failing to prep the candidate, and using too much drama. The solution is to develop a personalized recommender system for candidate and job matching, so the recommender system can help the job hunter apply to more suitable roles. When candidates apply to more suitable jobs based on their background, the recruiter will have to spend a smaller amount of time shuffling through resumes, as the sample of potential candidates will match closer to the needs of the job. This will lower the amount of time spent for recruiting for a job for the recruiter and the team because they will be receiving more relevant applicants. When a candidate is matched to a job based on their personal interests, they are more likely to stay longer, and the job will be a successful career decision for them, rather than starting them off with an awkward job transition pattern. Selecting a job can either have a positive or negative affect on your professional career. Recommendation systems can propose a positive affect on your professional career and eliminate all irrelevant job postings to help you be matched to the job of your dreams.

2.1.1 Goal

The goal of both seminars is for all of us to gain familiarity of how to deploy and continuously serve a model, which would make job recommendations, or rather, in general, serve as a reference for deployment of other recommendation system based tasks. Within the first seminar, we analyzed the Google Cloud Platform and the reference architecture it utilizes to deploy Machine Learning algorithms. Additionally, we gave demonstrations of the application of Google Cloud Platform from into the work-flow and automation of deploying a recommendation API to the cloud. We also tried other tasks as well, such as predicting wages based on job applicant meta-data to better evaluate the platforms we used for these tasks. Within the second seminar, we did more exploration with the Google Cloud Platform and the machine learning tools it offers. Additionally, we all explored Amazon Web Services, such as training and deploying machine learning models with Amazon EKS, training and deploying machine learning models with Kubernetes using Amazon EKS, and deploying our recommendation application through ECR and training models through the SageMaker instance. The investigation between both platforms allowed us to design and implement a server less recommendation API to continuously serve job recommendations on 360 job search. **360 Job Search** is a platform established by Armine, Marvin, and Vanessa that would give society the opportunity to have a platform to apply to jobs, receive recommendations, communicate with a recruiter in real-time, receive unique recommendations, and receive help on the fly with cutting edge machine learning technology tools. The goal throughout this text is to determine which cloud service will be more usefully for meeting the requirement of our job recommendation API. In order to achieve our goal, we must go through the project in depth, method and procedures of development, the architecture, the implementation in Google Cloud Platform and Amazon Web Services, and we must be able to do a successful analysis among both to see what is stronger.

2.2 Previous and Current Work, Methods and Procedures

These job recommendation APIs are derived from 360 job search. 360 Job Search is an application that allows society the opportunity to have a platform to apply to jobs. This platform would support four different types of users. It would support a guest, a registered user, and recruiter.



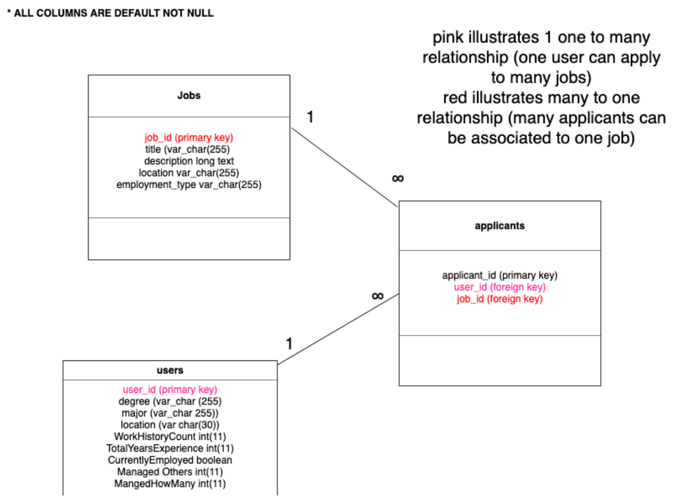
As a user applies to a job, a Flask API is connected to 360 job search which is running on Heroku. As they apply to more jobs, the recommendation algorithm will continuously be evaluated. Recommendations will change as data is continuously being processed in the model pipeline. If we are in the scenario if the recommendation algorithm fails either by returning an empty list or we get a failure from unable being to connect to the data store to go through the model by not serving results,we automatically call an RSS feed which scraps Stack Overflow jobs and we give jobs to a user that are in their city. But, to ensure a user is not given the same job over and over again, jobs are randomized each time. Currently recommendations are being served on a flask API that lives on **Heroku** a cloud service. The current work is to determine which cloud provider will be beneficial for us to continuously serve recommendations. As machine learning can help us make intelligent capabilities accessible without requirements advanced skills of data science and machine learning as we are at different levels. Additionally the benefit of machine learning in the cloud as the more data you have the more optimal your algorithms will be. Since the cloud as the pay per use, with a the machine learning workloads we do not have to limit ourselves with the amount of data, we can train more optimal algorithms. and leverage the power in training with additional hardware investment which can be costly. If we take 360 Job Search even farther the cloud will make it easy for us to experience and scale up ans projects go into production and demand increases, which is something Heroku does not offer. We do not need to use a cloud provider but using a cloud provider is helpful in limiting the barrier of dealing with large clusters. There are problems that cloud computing can solve that an in house machine learning algorithm is not capable in handling from processing power and reducing processing time. The research that is being done is that does either Amazon Web Services or Google Cloud Platform provide the general-purpose and machine learning services in order for us to meet our requirements and scale up as we go.

2.3 Background

Machine learning is the field of computer science that gives computers the ability to learn without explicitly being programmed. Machine learning is the field of computer science that gives computers the ability to learn without being explicitly programmed. It also has the ability to program computers to identify patterns in data, and to inform algorithms to make data driven decisions or predictions. It employs previously capabilities previously possessed by humans to solve unique tasks.The tasks that machine learning algorithms can apply are those that can be quickly accomplished by a human like identifying an image, or interpreting a meaning of a sentence.Why not utilize fitting models to data and to learn, to replicate the the tasks that human brains perform. That is what machine learning is starting to do. As we move towards a more tech savvy word, machine learning is being applied to new scenarios like self-driving cars, robots, etc. Since machine learning is limitless, why not apply machine learning to the market of job hunting, so job hunters [3] can have an up and coming companion to fight for their interests. As the hiring process has been breaking for years as not all recruiters are equally as good at their jobs. This will cause people to have a less favorable employment experience causing people to look for a job after three months of hire data. Which leads to companies having to start the recruitment process over,which can cost them. We can fix the broken system with artificial intelligence as a recommender system can help job hunters with the role and skill set. As AI makes relevant opportunities more efficient and will allow job hunter's not have to go to interviews that are unlikely to match their skills. Despite machine learning algorithms having a bias with a good enough data set a bias impact can be reduced. This is what we want to do. We want to limit the bias, limit the the damage to the market of finding a job, and help help people find the job they are meant to be at.

2.4 Brief Project Description

Due to the time constraint, the Kaggle Career Builder Job data set is being utilized. The reason for this is in order to meet requirements in the limited time frame, this was the solution we used to have a large enough data set to train a machine learning model. To generate recommendations for users for jobs, we took into consideration the users table, applicants table, and jobs table. As these data sets will help us gain valuable insight. We can do so based on the one to many relationship among the users and applicants table and the many to one relationship among the applicants and jobs table.

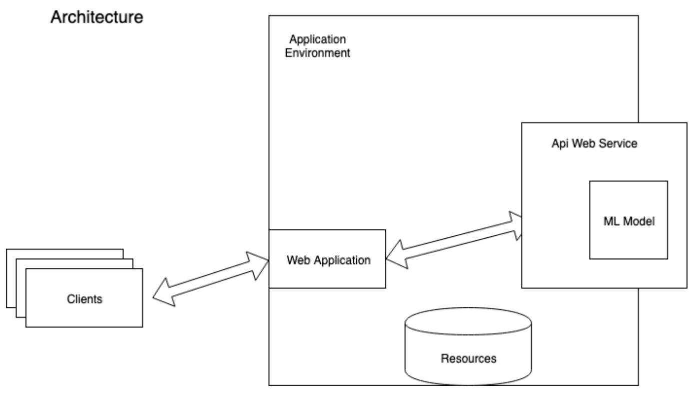

* ALL COLUMNS ARE DEFAULT NOT NULL
pink illustrates 1 one to many relationship (one user can apply to many jobs) red illustrates many to one relationship (many applicants can be associated to one job)
Jobs
job_id (primary key)
title (var_char(255)
description long text
location var_char(255)
employment_type var_char(255)
1
∞
applicants
applicant_id (primary key)
user_id (foreign key)
job_id (foreign key)
∞
1
users
user_id (primary key)
degree (var_char (255)
major (var_char 255))
location (var char(30))
WorkHistoryCount int(11)
TotalYearsExperience int(11)
CurrentlyEmployed boolean
Managed Others int(11)
MangedHowMany int(11)

The data model for the recommendation API and how elements relate to one another to get the components we need Within in this project, two approaches were done one machine learning API was created utilizing the Google cloud machine learning app engine and another approach was done via amazon web services. Within both approaches we are using similarity learning.**Similarity learning** we are able to learn from examples of how similarity or related two objects are. The recommendation APIs that are attempted to be created in this we are trying to learn from an job hunter's application history to recommend them similar jobs they should apply too. Recommendation systems are starting to become one of the widely used AI application learning tools. Implementation and architecture of two different approaches on two different platforms will give us a more in depth understanding of which service would be more beneficial in order for us to comply with the user's needs. Recommendations can either be personalized or non personalized. But, for the purposes of this project personalized recommendations are going to be constructed based on a job hunter's previous applicant history.

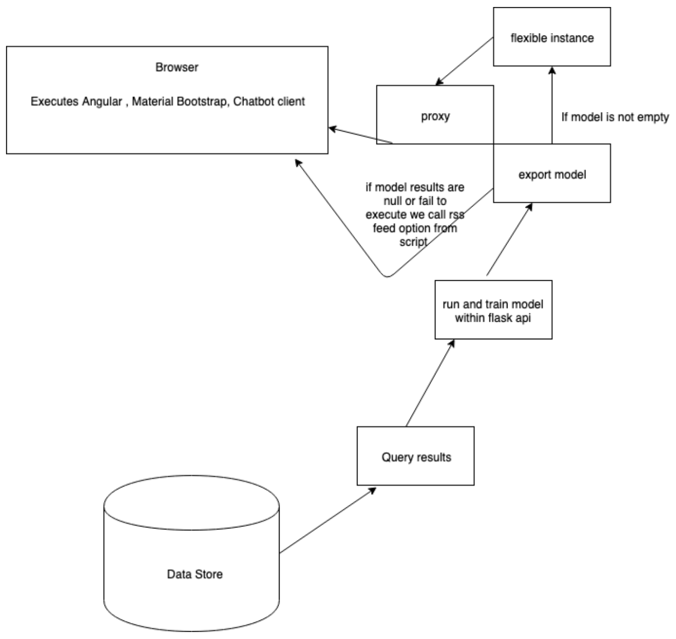
Chapter 3

Reference Architecture

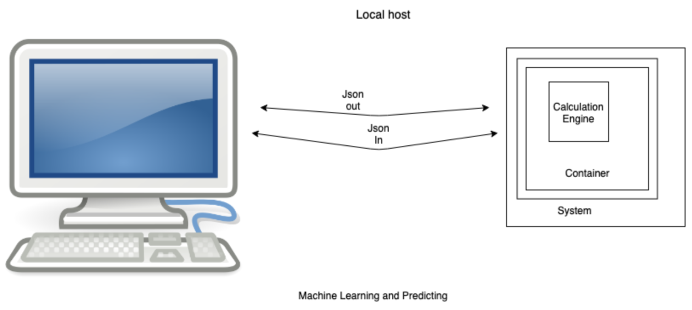
With building a job recommendation API a variety of different of architectural standpoints have to be taken into consideration to ensure our system is a success.



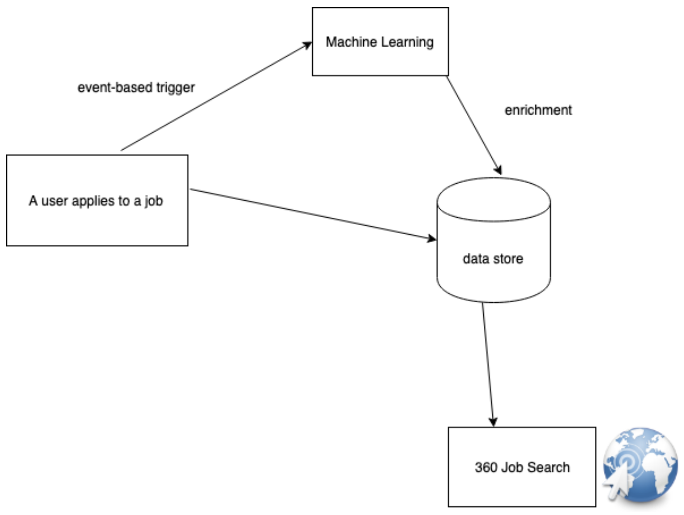
The main idea of the model is that since the model is an API the client can issue a GET request to receive predictions.


flexible instance
Browser
Executes Angular , Material Bootstrap, Chatbot client
proxy
If model is not empty
export model
if model results are null or fail to execute we call rss feed option from script
run and train model within flask api
Query results
Data Store

The big picture of the job recommendation api is that all data will live within the data store. From the data store, we will query the necessary results that we need to train our model. After model training has been complete we can export the model. If model results or null or the model fails to execute the model will automatically call an RSS feed which will scrap data from stack overflow jobs and give user's jobs from the top cities that people usually apply to. If the model's results are not empty, then the model's result can be called via an api. The user can do so with a GET request. Later within this documentation it will be explained how we can utilize this model as an api



`Data going into the system is in JSON form and data coming out (predictions)`



`is in JSON from too.`

This API will be useful in the scenario of when a user applies to a job. When a user applies to a job this information is added into the data store. As the user applies to more jobs, their applicant history will increase. Since their applicant history will continuously grow, this will allow us to continuously train a machine learning algorithm that can generate similar jobs for a user as they apply to more jobs.

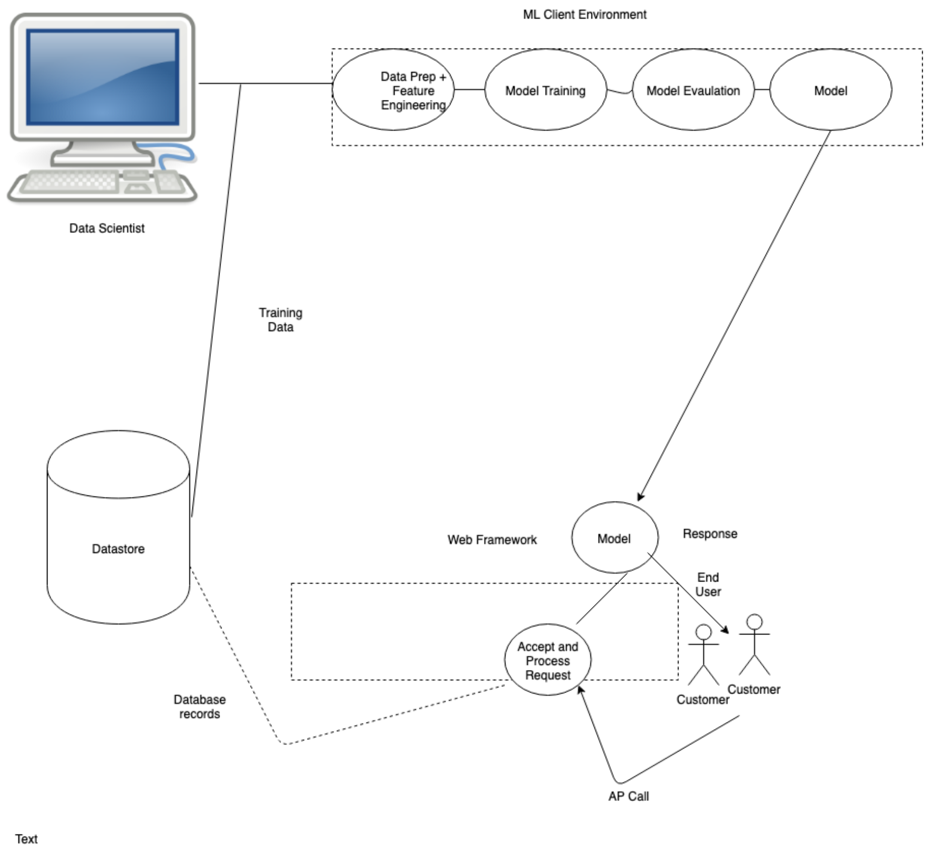


Figure 3.1: Deploying as a Web Service

Chapter 4

Implementation Guides

4.1 Creating a Job Recommendation Utilizing Google Cloud Platform

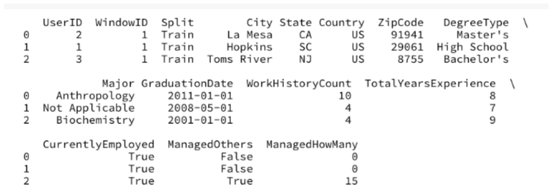
4.2 What are recommendation systems?

Recommendation systems filter data using different algorithms and recommend the most relevant items to users. Recommendation engines capture past behavior, and based on the past behavior, they recommend something to users they would mostly like.

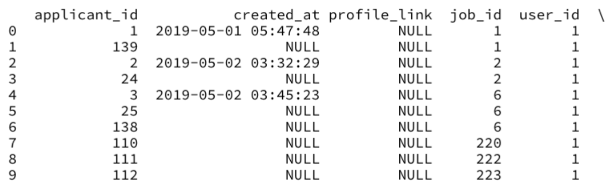
4.3 How do they work

4.4 Data Set

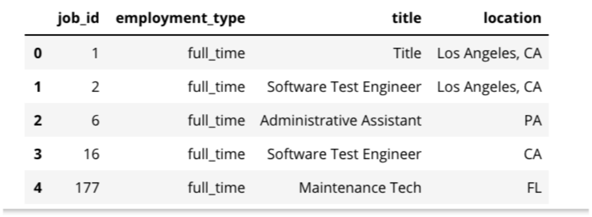
For this model, by utilizing the Google Platform, we used cosine similarity to predict similar jobs for users based on the previous jobs they have applied to. The data came from the Kaggle Competition. The data set we are using is the applicant table, user table, and job table.



```
   UserID  WindowID  Split        City State Country  ZipCode   DegreeType  \
0       2         1  Train     La Mesa    CA      US    91941     Master's
1       1         1  Train     Hopkins    SC      US    29061  High School
2       3         1  Train  Toms River    NJ      US     8755   Bachelor's

            Major GraduationDate  WorkHistoryCount  TotalYearsExperience  \
0    Anthropology     2011-01-01                10                     8
1  Not Applicable     2008-05-01                 4                     7
2    Biochemistry     2001-01-01                 4                     9

   CurrentlyEmployed  ManagedOthers  ManagedHowMany
0               True          False               0
1               True          False               0
2               True           True              15
```



```
   applicant_id           created_at profile_link  job_id  user_id  \
0             1  2019-05-01 05:47:48         NULL       1        1
1           139                 NULL         NULL       1        1
2             2  2019-05-02 03:32:29         NULL       2        1
3            24                 NULL         NULL       2        1
4             3  2019-05-02 03:45:23         NULL       6        1
5            25                 NULL         NULL       6        1
6           138                 NULL         NULL       6        1
7           110                 NULL         NULL     220        1
8           111                 NULL         NULL     222        1
9           112                 NULL         NULL     223        1
```



	job_id	employment_type	title	location
0	1	full_time	Title	Los Angeles, CA
1	2	full_time	Software Test Engineer	Los Angeles, CA
2	6	full_time	Administrative Assistant	PA
3	16	full_time	Software Test Engineer	CA
4	177	full_time	Maintenance Tech	FL

⇕ User, Applicant, and, Job Table

4.4.1 General Machine Learning Architecture

4.4.2 REST API for Predicting

Pre-processing Data

Preparing training and evaluation data

Training the model

Deployment

4.4.3 Deployment to the Google Cloud Platform

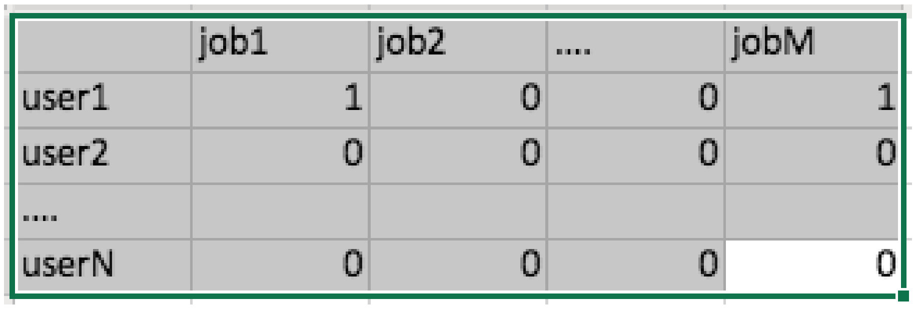
Calling the model for online predictions

4.5 Creating a Job Recommendation Utilizing Amazon Web Services

4.5.1 Initial Steps

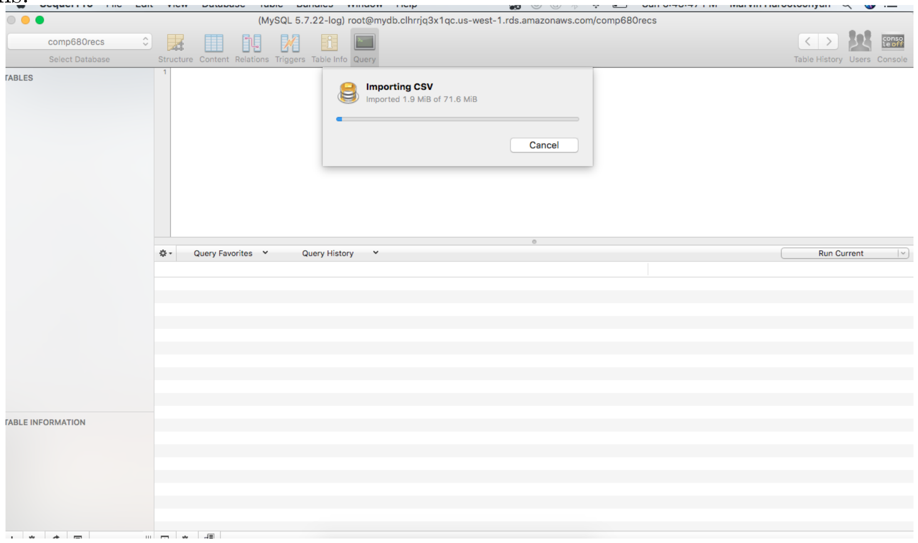
For the AWS implementation guide, we choose to use a item-item based collaborative filtering method on the same Kaggle data set. For the model we have to represent every job that a user could have applied to. For the initial training we created an RDS instance and loaded the applicant data and job data from Kaggle. Once we have our data properly loaded we can construct the item-item matrix. The item-item matrix representation:

4.5.2 Training a Job



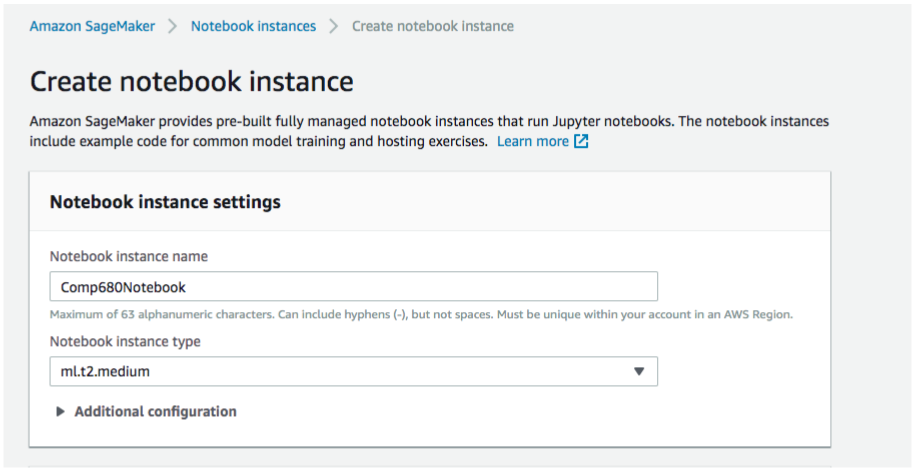
	job1	job2	....	jobM
user1	1	0	0	1
user2	0	0	0	0
....				
userN	0	0	0	0

We represent jobs that a user has applied to as either 1 if they applied or 0 if they have yet to apply. What makes this useful is we can add weights to the user vector by getting the total number of jobs they have applied to so instead of a binary 0 or 1 we can get a value between 0 and 1 for more robust recommendations.


(MySQL 5.7.22-log) root@mydb.clhrrjq3x1qc.us-west-1.rds.amazonaws.com/comp680recs

Importing CSV
Imported 1.9 MiB of 71.6 MiB

Our sequel pro desktop app which is connected to our RDS instance uploading TSV file to a MySQL table. Use the RDS endpoint of your database to connect your sequel pro desktop app.


Amazon SageMaker > Notebook instances > Create notebook instance

Create notebook instance

Amazon SageMaker provides pre-built fully managed notebook instances that run Jupyter notebooks. The notebook instances include example code for common model training and hosting exercises. Learn more

Notebook instance settings

Notebook instance name

Comp680Notebook

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique within your account in an AWS Region.

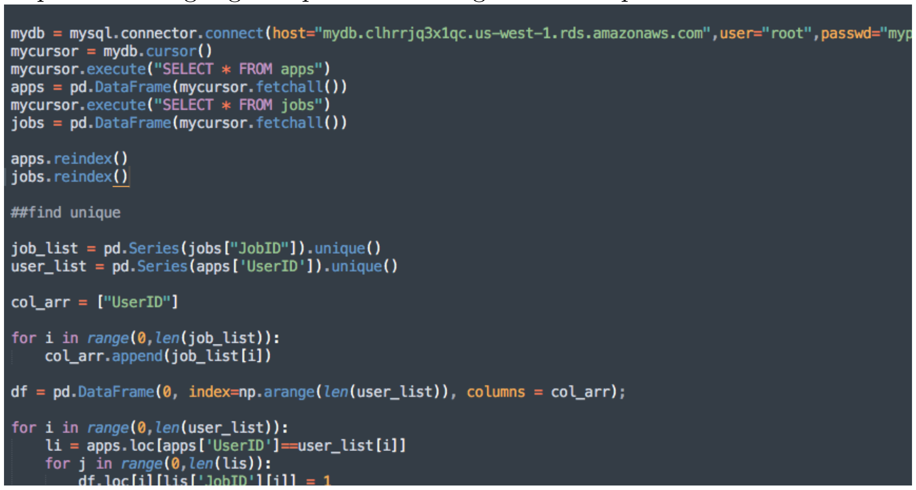
Notebook instance type

ml.t2.medium

▶ Additional configuration

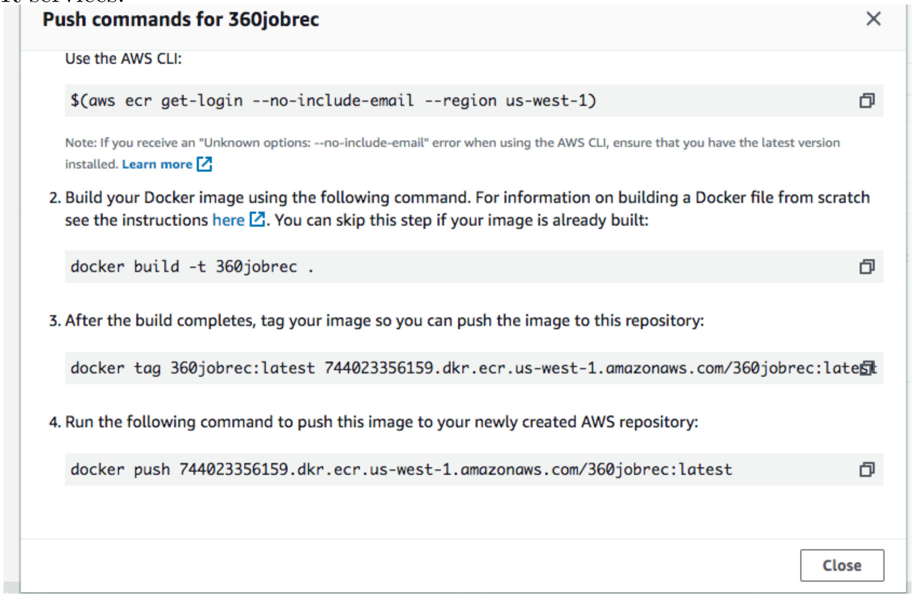
The SageMaker Jupyter Notebook instance serves more as a way to configure our Job Recommendation API and training jobs. This is where you would define your S3 buckets, data location, transform data, as well as define the docker instance which will serve our recommendation engine. Once we deploy our SageMaker endpoint it will retrain when we call the train method of our API and serve recommendations when we call our serve method.

We used a roll your own algorithm template [4] when setting up our models for training and serving. Its absolutely necessary to follow the template because we want to be able to automate this work-flow. Updating the model becomes as simple as making a get request to our SageMaker endpoint


```
mydb = mysql.connector.connect(host="mydb.clhrrjq3x1qc.us-west-1.rds.amazonaws.com",user="root",passwd="myp
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM apps")
apps = pd.DataFrame(mycursor.fetchall())
mycursor.execute("SELECT * FROM jobs")
jobs = pd.DataFrame(mycursor.fetchall())

apps.reindex()
jobs.reindex()

##find unique

job_list = pd.Series(jobs["JobID"]).unique()
user_list = pd.Series(apps['UserID']).unique()

col_arr = ["UserID"]

for i in range(0,len(job_list)):
    col_arr.append(job_list[i])

df = pd.DataFrame(0, index=np.arange(len(user_list)), columns = col_arr);

for i in range(0,len(user_list)):
    li = apps.loc[apps['UserID']==user_list[i]]
    for j in range(0,len(lis)):
        df.loc[i][lis['JobID'][j]] = 1
```

This is a portion of the training script it will be called when we call SageMaker train endpoint our model should update itself. You can see it connects to the RDS instance to pull the training data.

SageMaker uses docker instances through ECR to serve and train the Job Recommendation API so we needed to connect to the AWS CLI in order to build and upload our docker image. The docker build script we used is based of amazon roll your own algorithm example[4]. Amazon Web Services provides an example in the template how to create our docker image online in order to do this we also had to add custom permissions for an IAM user roles to access ECR services.



Push commands for 360jobrec

Use the AWS CLI:

```
$(aws ecr get-login --no-include-email --region us-west-1)
```

Note: If you receive an "Unknown options: --no-include-email" error when using the AWS CLI, ensure that you have the latest version installed. Learn more

2. Build your Docker image using the following command. For information on building a Docker file from scratch see the instructions here. You can skip this step if your image is already built:

```
docker build -t 360jobrec .
```

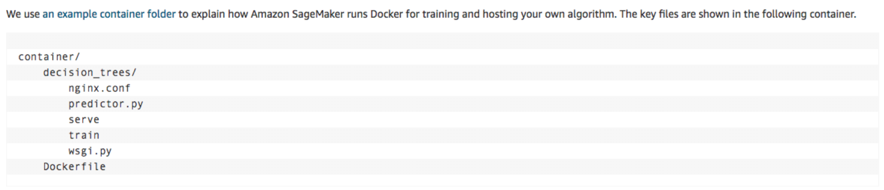
3. After the build completes, tag your image so you can push the image to this repository:

```
docker tag 360jobrec:latest 744023356159.dkr.ecr.us-west-1.amazonaws.com/360jobrec:latest
```

4. Run the following command to push this image to your newly created AWS repository:

```
docker push 744023356159.dkr.ecr.us-west-1.amazonaws.com/360jobrec:latest
```

The docker container needs to first be built on a local instance then uploaded back to ECR so SageMaker can make use of the application, and for us to make recommendations with our model. The file structure is as follows:



We use an example container folder to explain how Amazon SageMaker runs Docker for training and hosting your own algorithm. The key files are shown in the following container.

```
container/
    decision_trees/
        nginx.conf
        predictor.py
        serve
        train
        wsgi.py
    Dockerfile
```

```
# The flask app for serving recomendations
# UserID and JobID get added to our RDS database
# instance ‘apps‘ tables and now when we retrain will have
# new data for our item-item matrix
app = flask.Flask(__name__)
@app.route('/job-recs-after-apply/<UserID>/<JobID>', methods=['GET'
                                   ])
def job-recs-after-applying(UserID,JobID):
    mydb = mysql.connector.connect(host="mydb.clhrrjq3x1qc.us-west-
                                       1.rds.amazonaws.com",user="
                                       root",passwd="mypassword",
                                       port=3306, db='comp680recs')
    mycursor = mydb.cursor()
    mycursor.execute("SELECT * FROM apps WHERE UserID="+UserID+"
                                       AND WHERE JobID="+JobID)
    value   = mycursor.fetchAll()
    if value.empty:
        mycursor.execute("INSERT INTO `apps` (`UserID`,`JobID`)
                                          VALUES ("+UserID+","+
                                          JobID+")")

    pandasMatrix = ScoringService.get_model()
    recs = pandasMatrix.loc[JobID].nlargest(11).to_json();
    return flask.Response(response=recs,status=200, mimetype='
                                      application/json')
```

```
#The Train script that we need to periodically
#call in order to update our model
#This has to be on the docker container its
#Not executed through an http endpoint
#Although we can modify our app script above to do so
def train():
    print('Starting the training.')
    try:
        # Read in any hyperparameters that the user passed with the
                                           training job
        with open(param_path, 'r') as tc:
            trainingParams = json.load(tc)

        mydb = mysql.connector.connect(host="mydb.clhrrjq3x1qc.us-
                                          west-1.rds.amazonaws.com"
                                          ,user="root",passwd="
                                          mypassword",port=3306, db
                                          ='comp680recs')
        mycursor = mydb.cursor()
        mycursor.execute("SELECT * FROM apps LIMIT 10000")
        apps = pd.DataFrame(mycursor.fetchall())
        #mycursor.execute("SELECT * FROM jobs LIMIT 200000")
        #jobs = pd.DataFrame(mycursor.fetchall())

        job_list = pd.Series(apps[4]).unique()
        user_list = pd.Series(apps[0]).unique()




        cols =  ["UserID"]

        for i in range(0,len(job_list)):
            cols.append(job_list[i])


        df = pd.DataFrame(0, index=np.arange(len(user_list)),
                                          columns=cols);

        df["UserID"] = user_list

        lis = []

        for i in range(0,len(user_list)):
            lis = apps.loc[apps[0]==user_list[i]]
            lis = array(lis[4])
            for j in range(0,len(lis)):
                col = lis[0]
                df[col][i]=1


        print df
        data_items  = df.drop('UserID',axis=1)

        data_items.reset_index()

        data_items.fillna(0)

        magnitude = np.sqrt(np.square(data_items).sum(axis=1))

        # unitvector = (x / magnitude, y / magnitude, z / magnitude
                                          , ...)
        data_items = data_items.divide(magnitude, axis='index')

        def calculate_similarity(data_items):
            """Calculate the column-wise cosine similarity for a
                                                   sparse
            matrix. Return a new dataframe matrix with similarities
                                                   .
            """
            data_sparse = sparse.csr_matrix(data_items)
            similarities = cosine_similarity(data_sparse.transpose
                                                   ())
            sim = pd.DataFrame(data=similarities, index= data_items
                                                   .columns, columns=
                                                   data_items.columns)
            return sim

        data_matrix = calculate_similarity(data_items)
        #filehandler = open("data_matrix.obj","r+b")
        with open(os.path.join(model_path, 'decision-tree-model.pkl
                                          '), 'w') as out:
            pickle.dump(data_matrix, out)
        print('Training complete.')
    except Exception as e:
```

Chapter 5

Conclusion

5.1 Summary

AWS SageMaker ECR vs GCP ML Engine App Engine. Overall when working with AWS we found that it was more flexible than GCP. We noticed this by the way we would add user permissions by uploading scripts. AWS documentation is very dense while GCP is concise which makes GCP overall more intuitive for beginners. For SageMaker there exists an official developer's guide written which we used heavily for our implementation while GCP has a 4 page blog about ML Engine and App Engine. We didn't really make use of built in algorithms for training models we had to roll most of our own code for the collaborative filtering models we used since they didn't really make sense to fit in a classifier which caused a lot of confusion during seminar 1. This time around we just "pickled" pandas data frames of user-item matrices or item-item matrices and rewrote the REST API code templates. Overall pretty much the same experience on both platforms since were using Flask, Feed-Parser,Python, Sci-Kit, MySQL-Connect,and an auto-scaling virtual machine or container instance.

How do we recover in case we run out of jobs or recommendations? For both instances we check the magnitude of the users applications as a whole if its above a certain threshold divided by total number of jobs we introduce new jobs from the Stack Overflow RSS feed:

```
#We would insert these results to our jobs table
#with a url link to link users back to Stack Overflow
#And make columns for them in our matrices by adding
#a mock user with id 0 or -1
#along with a unique job
#id referencing back to the Stack Overflow post
    def recStack():
    randomCity=['losangeles','sanfransico','newyork','miami','
                                   london','washington']
    randomElement=random.choice(randomCity)
    feed=feedparser.parse("http://careers.stackoverflow.com/jobs/
                                          feed?location="+randomElement
                                          )
    entry=sorted(feed.entries[:50], key=lambda x: random.random())
    stackOverflowJobs = []
    for e in entry:
        job = {'title': e.title, 'desc': e.description, 'location':
                                                e.location}
        stackOverflowJobs.append(job)
        return stackOverflowJobs;
```

5.2 Problems Encountered

Most of the services encourage use of built-in algorithms. This makes it hard to roll your own ML Models because examples are scarce.

5.3 Suggestions for Future Extensions

A future expansion of this work is utilizing real time,machine learning with Kafka. As with the incorporation of Kafka, we can stream and classify recommendations simultaneously. It will also be useful for event tracking.

Appendices

[1] Baylor, Denis, et al. "Tfx." Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 17, 2017, doi:10.1145/3097983.3098021.

[2]Al-Otaibi, Shaha T. "A Survey of Job Recommender Systems." International Journal of the Physical Sciences, vol. 7, no. 29, 2012, doi:10.5897/ijps12.482.

[3] "The Dream Job Strategy." Get the Job You Want, Even When No One's Hiring, 2011, pp. 37–38., doi:10.1002/9781118257951.ch22.

[4]`https://github.com/awslabs/amazon-sagemaker-examples/tree/master/advance_functionality/scikit_bring_your_own`

[5] `https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf`