SomePlace*Else*

A Food + Restaurant Recommendation System for New York City

Samarth Shah
UAlbanyID - *001227544*
University at Albany, SUNY
New York, USA
Email: sshah4@albany.edu

***Abstract*—With a total of *25,000* restaurants, New York City is one of those cities with the maximum number of restaurants in the world. With such a plethora of choices, one is often confused about the very basic questions like, *"What do I eat?"*. Our application mostly attempts to answer that question. Moreover, we also attempt to suggest a set of restaurants for the user, based on the food items recommended to him.**

I. Introduction

A. Motivation

With a total of *25,000* restaurants, NYC is one of the cities with the highest number of restaurants in the world. With such a plethora of choices, one is often confused about the very basic question, *"What do I eat?"*

Avid Internet users always look for *quick* solutions to problems or questions like these. Most often, it can be as trivial as what should I eat today! But, the solutions that they might find has nothing to do with the user's specific taste - which I've briefly acknowledged in Section III. Rather, there are solutions that provide a list of the *best* restaurants in NYC, or, list of *best* food items in NYC. We go one step further and recommend a list of food items and restaurants based on pure statistics.

We use a concept called *Association Rule Mining*, and discuss it in great detail in Section II-B.

Our application mostly attempts to answer questions like *"What / Where do I eat?"*. Moreover, we also attempt to suggest a set of restaurants for the user, based on the food items recommended to him.

B. Problem Statement

By making extensive use of Twitter RestAPI, we find similar users to the input user and based on their taste in different kinds of cuisines or restaurants, we develop a list of food items to be recommended to the input user. Once we do that, we recommend a list of restaurants from where a user can seek such food items. To make it more user friendly, we also developed a front-end where dynamic statistical graphs are generated on-the-fly. We make an assumption that an ideal user's tweet include the name of the food item (name of cuisine or name straight from a menu item) and a restaurant name (in NYC).

C. Significance of the problem

What sets our solution apart from most others available solutions is, as aforementioned, our results, i.e. food recommendations and restaurant name recommendations are more likely to be of interest of user than other such solutions. This is based on the statistical theory of Collaborative Filtering [2] and Associative Rule Mining [3] (explained elaborately in Section II-B), which endorses the claim that the item recommended to the user is very much based on tastes' of users similar to the input user, and hence, the user will probably like that item.

II. Proposed approaches

A. Terms & Definitions

A few terms one might come across in this paper are as follows:

Recommender Systems [1], a subclass of *information filtering systems*, try to predict the preference of a user, given their background, or knowledge of similar users. It has become common in recent years, and are applied to a variety of applications (for instance, Netflix) in order to suggest items that a user might like.

Collaborative Filtering [2] is a technique for Recommendation Systems, which typically builds a model based on the input user's past behavior or users similar to input user (based on similar taste) and decisions made by them. This model is then used to recommend items (in our case, food) to the input user.

Association Rule Mining / Learning [3] is a very popular well-researched method for discovering interesting relations between items (i.e. food items, in our case). The most noted use of Association Rule Mining is Market Basket Analysis. But, it can also be used to build recommendation systems. An association rule is defined as an implication of the form

$$X \implies Y, where X, Y \subseteq I, X \cap Y = \emptyset$$

To select interesting rules, there are various metrics on constraints. Association rules are usually required to satisfy user-specified minimum support and minimum confidence. They are defined below:

Support of an itemset [3] of an itemset X $(support(X))$ is defined as the proportion of the tweets in the dataset that contain the itemset. Formally,

$$support(X) = \frac{\#\_of\_tweets\_containing\_X}{\#\_of\_tweets}$$

Confidence of a rule [3] is defined as

$$confidence(X \implies Y) = \frac{support(X \cup Y)}{support(X)}$$

Apriori Algorithm [4] is an algorithm used in Association Rule Mining. The basic agenda of Apiori is to generate association rules. In this project, we are going to use Apriori Algorithm to generate Association Rules - which is the core functionality.

B. *Detailed Approach with Implementation Details*

Collaborative filtering techniques as defined in Section II-A are based on collecting and analyzing a large amount of users' data, such as crawling their entire timeline. We use such Collaborative Filtering techniques to suggest food items and restaurants based on similar users and build a recommendation system for every new user.

Our initial aim was to find a good and optimal set of Queries to shoot at Twitter RestAPI, which involved a lot of hits and misses. Initially, we focused on very generic terms (mostly cuisine genres) related to food. For instance, we used keywords like *"meat OR chicken OR beef OR pork OR ham OR burger..."* and so on to form our query. We have used ≈ 400 such generic keywords.

The Twitter API allows the queries to be only *471* characters long, while our query was approximately *7000* characters long, which forced us to break down our query to a list of *21* such queries. The parameters we set before firing the Twitter query were:

```
myApi.GetSearch(list_of_queries , NYC_geo, count=200,
                max_id=MAX_ID, result_type='mixed')
# NYC geo within radius 20mi
# Setting new MAX_ID on every loop.
# list_of_queries are 21 such queries.
```

After receiving very generic results, we optimized our query to include the names of the food items. Since we found many similar queries like: *"I love eating McChicken Sandwich"*. We decided to include such menu items straight under the query keywords, which brings us to our next step.

We crawled Recipe API from Mashape in Java to collect 100 most common menu item names of 12 different cuisines, which increased our keyword corpora greatly. After such optimizations to our queries, we were able to collect approximately 8400 tweets.

In order to find top influential users (i.e. *topK* users), we maintained a dictionary *user_count* of type $username \rightarrow count$. Given a tweet, the following script extracts the username and stores it in the dictionary, increases the count or sets the count to 1.

```
def countUser(tweet):
    user_name = tweet['user']['screen_name']
    #stores & increases count of tweets/username
    user_count[user_name] = user_count[user_name] + 1
        if user_count.has_key(user_name) else 1
```

I managed to write a small script to dynamically calculate topK users from named dictionary.

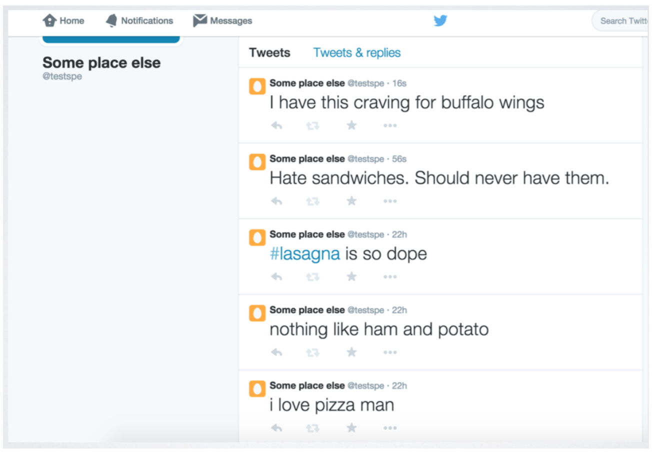
```
def topUsers():
    sorted_users = sorted(user_count.items(),
        key=lambda (k, v):v, reverse=True)
    # Sort based on Values
    # returns a list of topK names
    return [sorted_users[i][0] for i in range(topK)]
```

Suppose $topK = 20$, then it means we find top 20 users. The next step is to crawl their timeline. Mostly, we choose to crawl *600* tweets per username and dump the tweets using *json* to *username.txt* files.

Once we have crawled the timelines of *topK* users, we find the itemSet of every user. ItemSet of each user consists subset of list of items (keywords: as we mentioned in the beginning of this section) that we maintained initially while firing the queries. The following script achieves that:

```
def createItemSet(user_name):
    itemList = [] # itemList maintains all the
    # menu items and food terms
    words_by_user = [] # stores ALL the words in the
    # itemList that the user wrote in their tweets.
    with open('Users/'+user_name+'.txt', 'r') as r:
    for line in r.readlines():
        for word in line.strip().split():
            for i in range(len(itemList)):
                if itemList[i] in word:
                    words_by_user.append(itemList[i])
    item_set.append(list(set(words_by_user)))
    # Converting it into a set removes duplicates
    # and appends it to our itemSet.
```

Once the **item_set** consists of all the itemSets by our *topK* users, we get the input user's username and go through similar procedures as above. An account as shown below was used for the testing purposes.



Some place else @testspe · 16s
I have this craving for buffalo wings

Some place else @testspe · 56s
Hate sandwiches. Should never have them.

Some place else @testspe · 22h
#lasagna is so dope

Some place else @testspe · 22h
nothing like ham and potato

Some place else @testspe · 22h
i love pizza man

We crawl the input user's timeline, dump all the tweets in a file; after which we generate the itemSet for the input User and append it to item_set list.

The most important part or the "input" to Associative Rule Mining is a basket of itemSets. We call this basket as **data.basket**. We dump the list item_set to the data.basket using this script:

```
with open('Basket/data.basket', 'w') as w:
    # Writes it into data.basket
    for idx, item in enumerate(item_set):
    result = ''
        for idx, iii in enumerate(item):
        # iii = items in ItemSet
        result += (iii + ',')
            if (idx < len(item)-1) else iii
       w.write(result + '\n')
    # Dumped the itemSet in to data.basket
```

The data.basket looks like this after the dumping of itemSet is done:

```
1. salad , corn , sushi
2. ham, beef , burger , roll , corn , bacon
3. pita , roll , sushi ,ham, tacos
4. ham, cheesecake , corn , burger , cake
5. cake , steak , egg ,ham
```

As we can see, the itemSet at (1) will be for user #1, itemSet at (2) will be for user #2, and so on. *Note that we must remember that the* ***last*** *itemSet in the basket is for our input user* since it was the one added last.

Now that we have our basket ready, we use **Orange** package [6]. Orange package provides two algorithms for induction of association rules - out of which, we will be using Apriori Algorithm for sparse data analysis. Note that our basket is sparse, and not all itemSets have fixed number of items. Some may have *3* items (salad, corn, sushi) while some may have *4* items (cake, steak, egg, ham) in their itemSet.

In order to get more accurate results, the support and confidence constraints need to be tuned according to heuristics. We mostly used the support value of 0.3 and 0.4, which means that the itemset should be present in the basket 30% and 40% of times respectively.

I managed to devise the following script to get the recommendation list of food items for the input user:

```
def getRecommendation():
    #  Load data from the text file: data.basket
    data = Orange.data.Table("Basket/data.basket")
    data_instance = data[len(data)-1]
    # The last item in the basket is
    # the one of userinput!
    rules = Orange.associate
            .AssociationRulesSparseInducer(data,
                                        support=0.3)
    # Since the data is sparse
    # Assuming that support = 0.3
    for rule in rules:
        if rule.applies_left(data_instance)
        and not rule.applies_right(data_instance):
        # If Left itemset matches, but not Right
            rec_list = rule.right
                        .get_metas(str).keys()
            for item in rec_list:
            if ranked_recommendations.has_key(item):
                ranked_recommendations[item] += 1
            else:
                ranked_recommendations[item] = 1
```

The snippet of rules generated by Orange API, which uses Apriori Algorithm in it's backend, were as follows:

```
egg -> pizza chicken
egg chicken -> pizza
chicken -> pizza egg
pizza -> egg chicken ham
pizza egg -> chicken ham
pizza egg chicken -> ham
```

The value on support made vast changes to the accuracy and the number of rules generated. So my first instinct was to record all the number of rules generated per support value. Hence, I made the algorithm flexible enough to take the value of Support as input dynamically. Some of those values are:

```
[0.3, 687]
[0.3, 718]
[0.4, 96]
[0.4, 48]
```

As you can see, when the support is 30%, there are more rules generated, definitely at the cost of accuracy; whereas when the support is 40%, there are less rules but more accuracy. We went ahead with the decision of generating more rules for now, hence we maintained the support value to be 0.3.

The dictionary **ranked_recommendations** now contains all the names (keywords) of the food items that gets recommended to the user. But, we need to also recommend restaurants based on the food items.

Also note that dictionary ranked_recommendations is a $item \rightarrow count$ dictionary where the more times an item appears in the generated rules, then it should mean that it should be ranked higher than the others.

The ranked_recommendation looks something like this:

```
'sausage': 324,
'sandwich': 320,
'ham': 912,
'salad': 544
```

It simply means that out of these recommended items, ham showed most times in the generated rules. Ergo, it should be recommended at the top. This is done dynamically on the front-end.

Now using such a list, we again need to query Twitter RestAPI - but *dynamically*. So I created a script that dynamically creates a query from the recommended items and shoots it at the Twitter API to crawl lots of tweets. The script is as shown below:

```
def getRestaurants():
    queryItemList = []
    for k in ranked_recommendations.keys():
        queryItemList.append(k)
        query = '_OR_'.join(queryItemList)
        # concoct a query dynamically
        MAX_ID = None
        for idx in range(3):
        # Get about 600 tweets using that query
            tweets = [json.loads(str(raw_tweet)) for
                raw_tweet in myApi
                .GetSearch(query, NYC_geo, count=200
                max_id=MAX_ID, result_type='mixed')]
            if tweets:
                MAX_ID = tweets[-1]['id']
                for tweet in tweets:
                    countUser(tweet)
                storeOnFile(tweets)
            populateRestaurantList()
            rankRestaurants()
```

The code snippet above is for the most part, self-explanatory, resulting into an output file of all the tweets retrieved by the query generated dynamically. The step following this would be to populate the restaurants list, which has been done before and that list is stored in a static text file. The procedure to retrieve $25,000$ restaurants was arduous and we had to face the curse of multi-dimensional raw data. First attempt to retrieve names of restaurants in NYC from Yelp API was unsuccessful since it somehow restricted us to fetch more than 1000 names. Fortunately, we came across an Open DataSet [11] containing a list of $25,000$ restaurant names appended to their corresponding Addresses. After a lot of preprocessing and duplicate entry removal, we were able to narrow it down to $19,000$ unique restaurants in NYC. For now, we assume that restaurants with the same name and different addresses are one.

The next step was to compare the collected twitter corpus to this list of $19,000$ restaurants. Which basically means that *every* term in all the tweets in the corpus was required to be compared to $19,000$ entries of restaurant list. This seems like a very big figure, but I was able to figure out an $O(1)$ solution: Using *dictionary* - which turned out to be blindingly fast. A dictionary is a hashtable data structure with $O(1)$ lookups. Using all the restaurant names as keys in a hash table, all we had to do was to lookup if the term in the tweet matches the key (i.e. restaurant name); and if it did, we needed to increment the count by 1. The dictionary with $19,000$ keys was of the form $restaurant\_name \rightarrow count$. The dictionary with a non-zero count means that it was mentioned in the twitter corpus that we collected. Also, it goes a similar procedure for ranking like that of ranking recommended food items.

The script to get the recommended restaurant names is as follows:

```
def rankRestaurants():
    with open('Files/tweets_crawled_by_query.txt',
                                            'r') as r:
        for line in r.readlines():
        # Goes through every tweet
            for idx in range(len(restaurantList)):
                r_name = restaurantList[idx]
                if r_name in line:
                # If restaurant_name is present
                # in the line
                    if ranked_restaurants
                            .has_key(r_name):
                        ranked_restaurants[r_name]
                                              += 1
                    else:
                        ranked_restaurants[r_name]
                                               = 1
```

At the end of this procedure, the dictionary ranked_restaurants looks something like this:

```
'Cherche␣Midi': 2,
'Jackson␣Hole': 2,
'Astor␣Bake␣Shop': 2,
```

As we can notice, now we have **ranked_recommendations** (food list) and **ranked_restaurants** (restaurant list). We simply pass it to the front-end, where the graphs are generated dynamically.

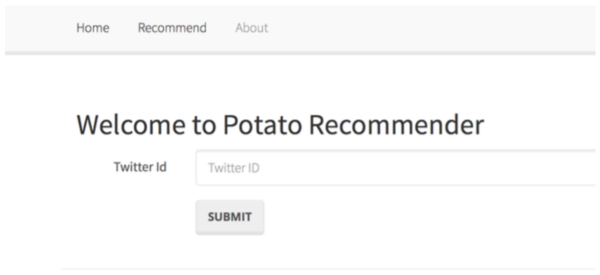
Before going into detail about visualization techniques that we used, I decided to clock the runtime of this application i.e. the time taken to generate a recommended list of food items and restaurants, since this is a front-end application and generally users don't like to wait much. The time taken to run this application took approximately 1 minute, which is too much for a web page to load. So after a lot of analyzing, I realized that for any user, the basket can remain unchanged and the results are still somewhat similar; which means that there is no necessity to generate data.basket everytime. We call the procedure to generate the data.basket *everytime* for a *new user* as **Eager Technique**.

Also, we came up with a solution to store the data.basket statically on a file, and not have it generate everytime for every new user. We call this technique as **Lazy Technique**, since it doesn't do all the hard work of populating the data basket everytime.

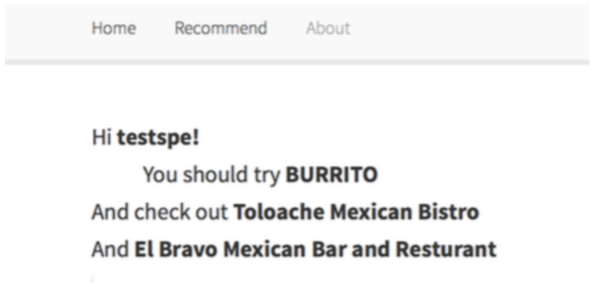
The time to display results improved significantly, i.e. it took about 15 *seconds* to to run this application - that is 66% improvement of time and such latency seems tolerable by any user. Moreover, the results using the Eager Technique and Lazy Technique were similar approximately 75% of the time; which we consider to be the classic *speed vs accuracy* trade-off.

In the front-end of our application, there are two main pages, one takes in the input user's twitter username and the other shows the recommendation list and results (alongwith the graphs).

A screenshot of the page is as follows:



A screenshot of the *display results* page is as follows:



Hi **testspe!**

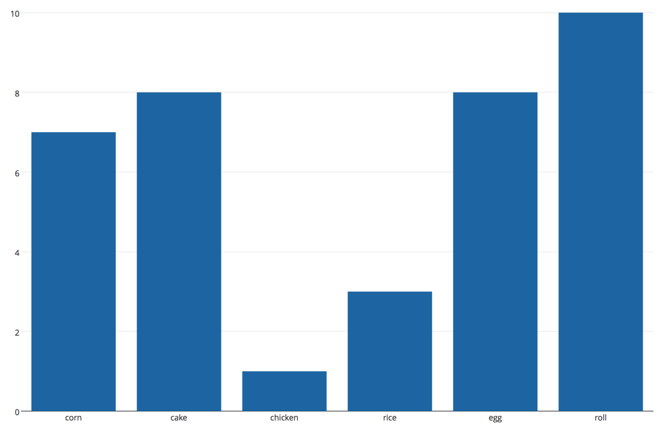
You should try **BURRITO**

And check out **Toloache Mexican Bistro**

And **El Bravo Mexican Bar and Resturant**

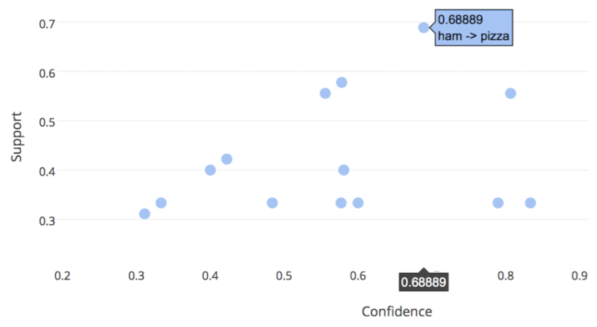
The display results show the top ranked food item at the top, and then followed by a list of the 2^{nd} and 3^{rd} ranked items.

A screenshot of a *histogram* is as shown, where each bin consists of the name of the food item recommended, and the bin size is the number of times it was mentioned in the rules generated:



As shown in the histogram above, the item *roll* appeared 10 times in the rules, followed by *cake* and *egg* which appeared 8 times.

A screenshot of a $Support \rightarrow Confidence$ graph generated dynamically is as shown below:



This graph shows the how the value of support changes with change in confidence.

III. Related Work

Last.fm: Last.fm builds a detailed profile of each user's musical taste by recording details of the tracks the user listens to, either from Internet radio stations, or the user's computer or many portable music devices. It uses a similar concept of Collaborative Filtering and is considered to be one of the best applications using recommendation system based on similar user's profiles and taste in music; and hence it provides for a very interesting case study in our application.

Nara.me By uncovering connections between places in the world and learning about your tastes, Nara instantly reveals restaurants and hotels you'll love. The only shortcoming is that instead of using Mining techniques like Association Rule Mining and learning your features, it takes a survey before it can recommend to understand your background.

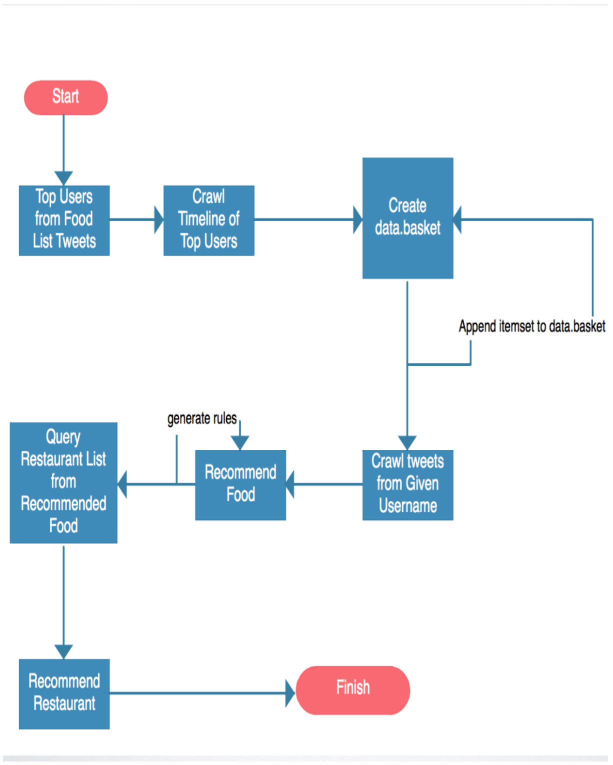
Foodspotting.com Foodspotting does not uses data mining techniques in terms of recommendation systems, but it only suggests best restaurants in the area using restaurant reviews and feedbacks. Plus, it does not recommend cuisines like our application aims to do.

Urbanspoon.com Urbanspoon is a mobile application which works when you shake the phone and spin through your choices in a virtual slot machine to get a random restaurant recommendation. Its random choices are irrespective of the input user's taste as well as location unlike ours.

IV. System Design and Implementation

A. Architecture

A brief overview of the application infrastructure as opposed to a detailed description in II-B is as shown below:


Start
Top Users from Food List Tweets
Crawl Timeline of Top Users
Create data.basket
Append itemset to data.basket
Crawl tweets from Given Username
Recommend Food
generate rules
Query Restaurant List from Recommended Food
Recommend Restaurant
Finish

The basic flow of the application is very clearly mentioned here in the diagram, moreover, it is explained alongwith the code in Section II-B

B. Infrastructure

Our language of choice for this project is Python, for plenty reasons. Since Python is a very robust language and contains very strong APIs for Data Mining tasks like Crawling Tweets, Association Rule Mining and Recommendation Systems, we chose to use Python. Besides a very robust language, Python is very easy to understand.

For crawling tweets through Twitter RestAPI, we used the *python-twitter* package, which provides many methods for firing queries, providing geolocation data, crawling a user's timeline using the username. For visualization purposes as mentioned in detail in IV-C, we used *plotly* package. The data exchange format is performed in json format, and we have used the in-built package called *json*.

We also used Java for crawling the data from NYC Health Department Restaurants Open Dataset [11] and Mashape Recipe API [12].

All the testing and experiments were performed on Mac OSX: Yosemite with 8GB Memory.

Real-time collaboration and Version-Controlling of the code was done using GitHub. The GitHub URL for our projects is https://github.com/SomePlaceElse/SomePlaceElse.git

C. GUI

For the Graphical User Interface, we have used the Flask framework [8]. It is very quick and easy to setup. Moreover, it is in Python, so it is seamless to integrate the Python back-end to the front-end. Moreover, we have also used *Jinja2* templates to do less amount of HTML coding since it provides a ready-to-use template. For styling purposes, we have extensively used (and are planning to use) Bootstrap API by Twitter [9]. Bootstrap is a CSS and JavaScript framework that provides responsive design structure that enables websites to render similarly on desktop and mobile devices

For generating graphs on-the-fly and directly embedding to the website, we have used Plotly API [10]. Plotly plots web-friendly graphs based on python code. Can be considered to be an extension of matplotlib. It is an API based for quick embedding through iframe or $< div >$ tags.

V. Conclusion

The curse of multi-dimensional real-world data sets is such that it is very raw and needs too much preprocessing. We faced issues, but we were able to provide meaningful food and restaurant recommendations. We also learned a lot about improving accuracy through heuristics.

VI. Future Work

This application can be a daily driver for a lot of people with its great advantages, albeit, it has its drawbacks too. And hence, we consider to improve this application and attend to these shortcomings.

This application is not able to find optimal results for input users *not* having any tweets related to food. Since, we are including the itemset of the input user in the basket, if there are 0 keywords (zero items in the itemSet) for the input user, the application will return an empty list of recommended food items and restaurants. It could use more features rather than just the food keywords, for instance, the age of the user, the mentioned location of the user, etc. and then find similar users based on those features. That way, even if the user hasn't ever tweeted about food, he could still get recommendations.

We could use state-of-the-art and more robust technique called PSL (Probability Soft Logic) in order to retrieve the restaurant names from the recommended list of food items. One basis for it could be: if a user X who eats spaghetti and goes to Chipotle, then a user Y who follows X and likes spaghetti, should get Chipotle as a recommended restaurants. With PSL, the applications and improvements could be infinitely many.

While collecting the tweets initially, we did not focus on the sentiments of the tweets. For instance, "I *love* Pizza" and "I *hate* pizza" is not known to our application and is still considered positive. We could incorporate sentiment analysis and preprocess the tweet corpus to remove false positives.

With the advances in mobile world and its possibilities, we could develop an Android and an iOS application for the same. It could also be tightly integrated with the input user's twitter account, and provide recommendation dynamically everyday.

Moreover, the UI could be significantly improved. Meanwhile, we are still considering to add more visualization techniques like statistical graphs and charts, wordclouds, etc. to provide more meaningful results and manifest the internal data mining techniques.

Acknowledgment

I would like to express my gratitude to Professor Feng Chen for guiding us through this project and constantly helping us through finding new ways to a solution.

References

[1] Francesco Ricci and Lior Rokach and Bracha Shapira, Introduction to Recommender Systems Handbook, Recommender Systems Handbook, Springer, 2011, pp. 1-35

[2] John S. Breese, David Heckerman, and Carl Kadie, Empirical Analysis of Predictive Algorithms for Collaborative Filtering, 1998

[3] Piatetsky-Shapiro, Gregory (1991), Discovery, analysis, and presentation of strong rules, in Piatetsky-Shapiro, Gregory; and Frawley, William J.; eds., Knowledge Discovery in Databases, AAAI/MIT Press, Cambridge, MA.

[4] Rakesh Agrawal and Ramakrishnan Srikant Fast algorithms for mining association rules in large databases. Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, pages 487-499, Santiago, Chile, September 1994.

[5] Lin, Weiyang. Association rule mining for collaborative recommender systems. Diss. Worcester Polytechnic Institute, 2000.

[6] Orange Package Python: http://docs.orange.biolab.si/reference/rst/Orange.associate.html?highlight=association#

[7] Twitter RestAPI: https://dev.twitter.com/rest/public

[8] Flask API Python http://flask.pocoo.org/

[9] Twitter Bootstrap API http://getbootstrap.com/

[10] Plotly API for statistical figures https://plot.ly/

[11] NYC Health Dept. Restaurants Open Datasets https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/xx67-kt59

[12] Mashape Recipe API https://www.mashape.com/webknox/recipes/