CS 757- Mining Massive Datasets using MapReduce

George Mason University

**A Project Report On**

# Yelp Dataset – Inverted Index and Clustering on SPARK

Vinit D Muchhala
Jatin N Mistry

## 1. Problem Description

In this project we are performing two different tasks. One is creating an Inverted Index and the other is Clustering the business dataset to check if some relation exists between reviews and the business features.

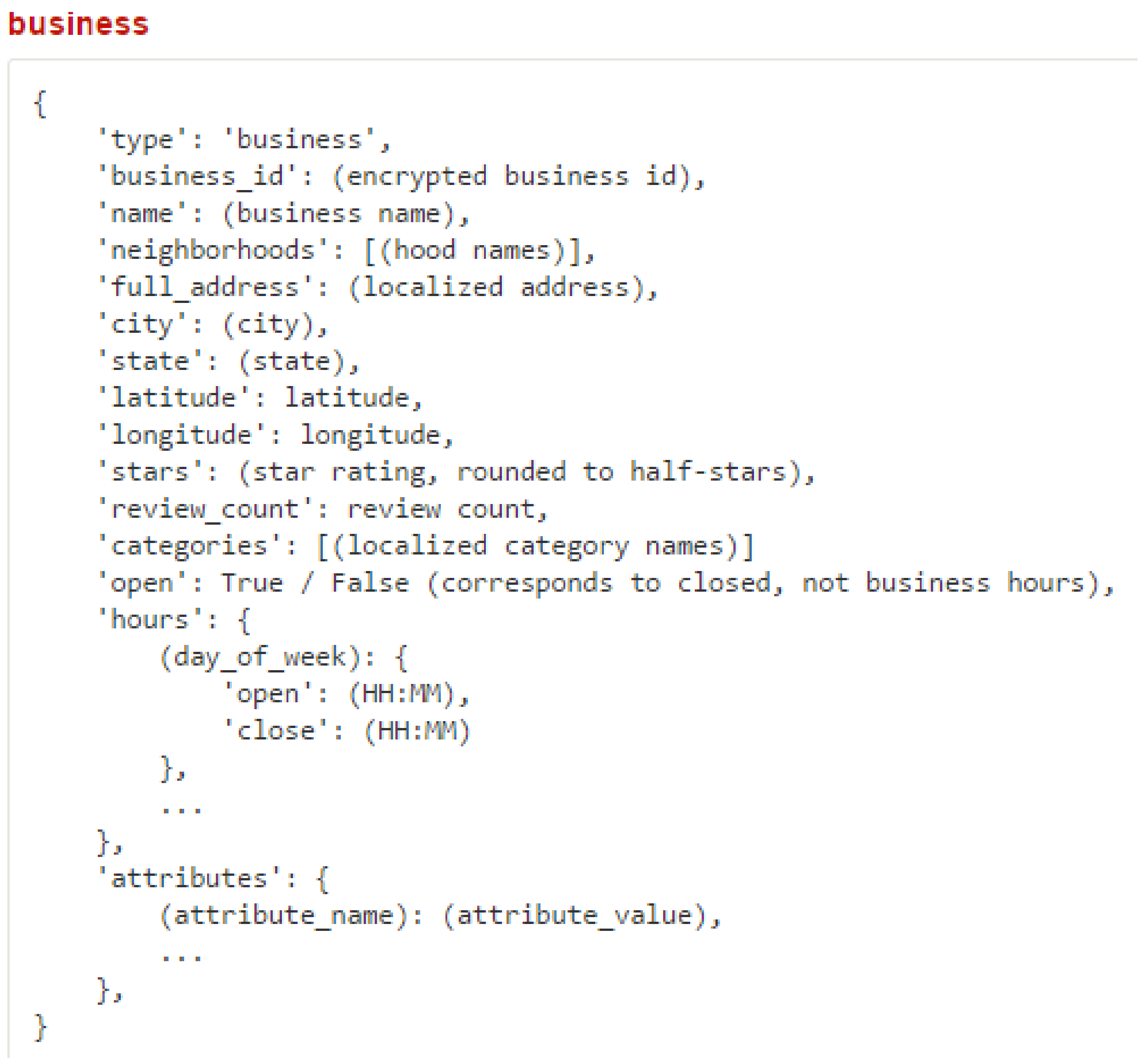
The key challenge for creating an Inverted Index is that it takes a lot of time to make one without using a distributed system and the size of the Inverted Index. In this project we attempt to build an inverted index on the Yelp dataset [1] as fast as possible and also which is not that huge in size. We used the two different techniques outlined in "Data-Intensive Text Processing with MapReduce" [2] by Jimmy Lin.

Clustering on the business dataset to see if some relation exists between the reviews and any one of the business features.

We have used Yelp dataset for performing the above tasks. The dataset is released to the public as a part of Yelp Dataset Challenge. The dataset can be obtained from the following location: http://www.yelp.com/dataset_challenge.
The yelp dataset contains various business and review information from 10 different cities in the U.S., Canada, England and Germany.

## 2. Dataset Exploration

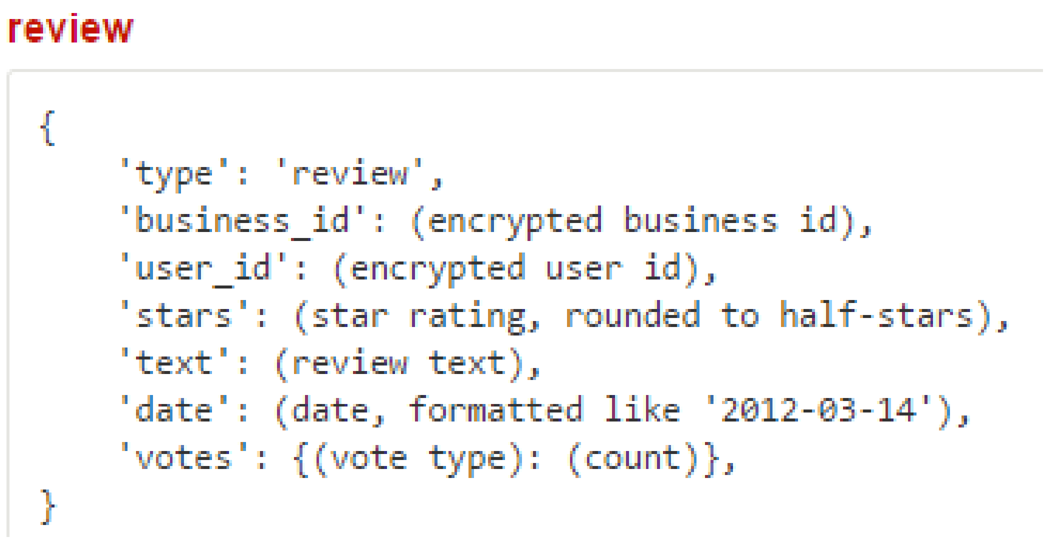
The dataset in its purest form is in JSON format, which makes it very difficult to process, especially for data mining tasks.
Below is a snapshot of the data files and their contents along with their descriptions
Business.json – contains information regarding the business, for ex, categories ("restaurant", "eye doctor"), rating, etc.



**business**

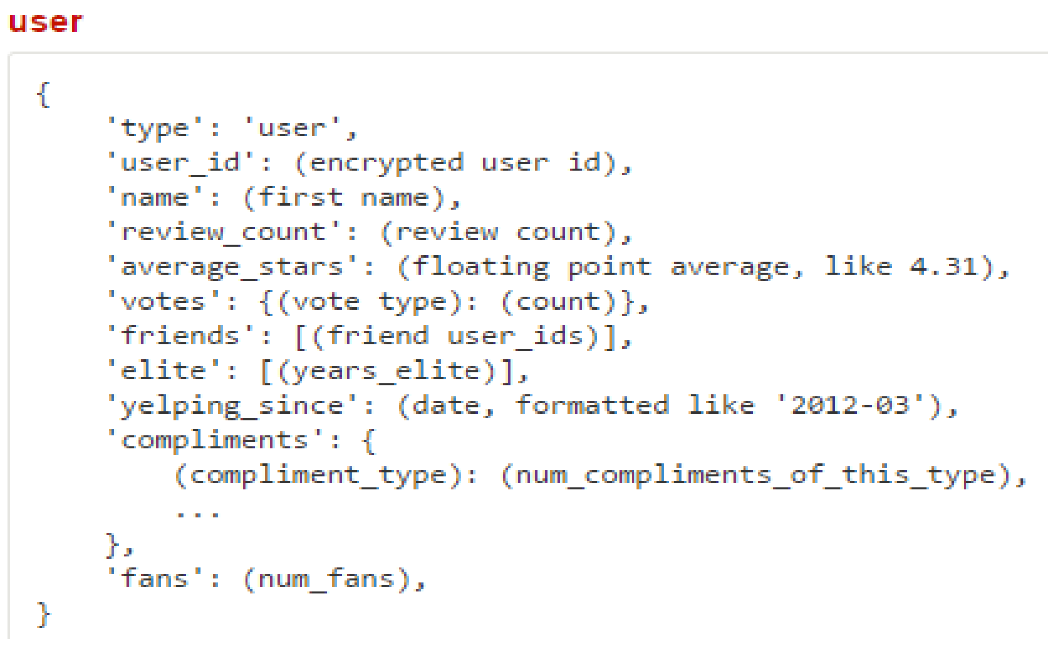
```
{
    'type': 'business',
    'business_id': (encrypted business id),
    'name': (business name),
    'neighborhoods': [(hood names)],
    'full_address': (localized address),
    'city': (city),
    'state': (state),
    'latitude': latitude,
    'longitude': longitude,
    'stars': (star rating, rounded to half-stars),
    'review_count': review count,
    'categories': [(localized category names)]
    'open': True / False (corresponds to closed, not business hours),
    'hours': {
        (day_of_week): {
            'open': (HH:MM),
            'close': (HH:MM)
        },
        ...
    },
    'attributes': {
        (attribute_name): (attribute_value),
        ...
    },
}
```

Review.json – contains the review text, the one which will be used to create an inverted index


review

```
{
    'type': 'review',
    'business_id': (encrypted business id),
    'user_id': (encrypted user id),
    'stars': (star rating, rounded to half-stars),
    'text': (review text),
    'date': (date, formatted like '2012-03-14'),
    'votes': {(vote type): (count)},
}
```

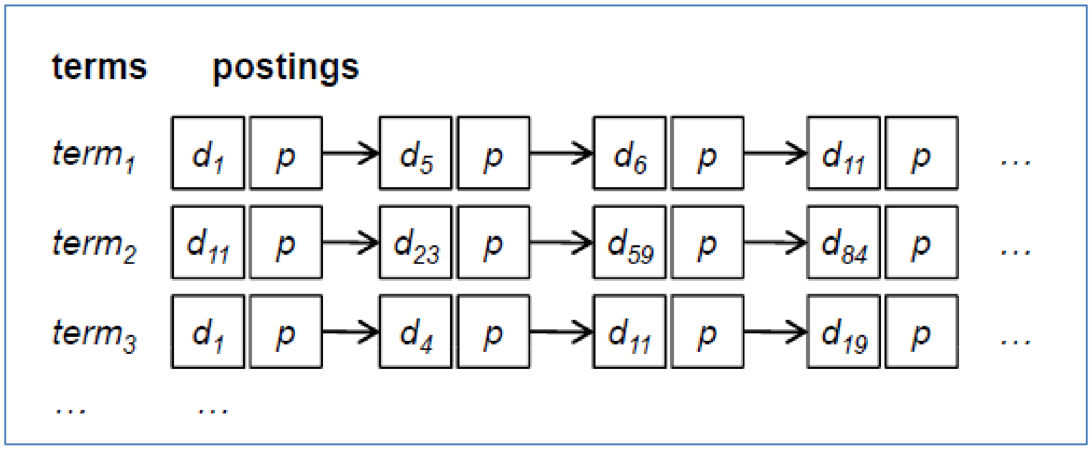
User.json – contains information regarding the user, number of reviews given according to review id


user

```
{
    'type': 'user',
    'user_id': (encrypted user id),
    'name': (first name),
    'review_count': (review count),
    'average_stars': (floating point average, like 4.31),
    'votes': {(vote type): (count)},
    'friends': [(friend user_ids)],
    'elite': [(years_elite)],
    'yelping_since': (date, formatted like '2012-03'),
    'compliments': {
        (compliment_type): (num_compliments_of_this_type),
        ...
    },
    'fans': (num_fans),
}
```

## 3. Inverted Index

In its basic form, an inverted index consists of postings lists, one associated with each term that appears in the collection. A postings list is comprised of individual postings, each of which consists of a document id and a payload information about occurrences of the term in the document. The most common payload, however, is term frequency (tf), or the number of times the term occurs in the document. More complex payloads include positions of every occurrence of the term in the document, properties of the term, or even the results of additional linguistic processing.



The above figure shows a simple illustration of inverted index. Each term is associated with a list of postings. Each posting is comprised of document id ($d_i$) and a payload p. An inverted index provides a quick access to document ids that contain a term.

Generally, postings are sorted by document id, although other sort orders are possible as well. The document ids have no inherent semantic meaning, although assignment of numeric ids to documents need not be arbitrary. For example, pages from the same domain may be consecutively numbered.

We first pre-processed the reviews to tokenize them. For preprocessing, we first converted the review text to lower case, applied stemming and filtered out stop words.

Pseudo code for Preprocessing:

```
input : review data

nCnt = 0
let preprocess() =
      foreach line in stdin:
            // each line is a review
            data = split the line and extract the review text
            convert the review text to lower case.
            tokenize the review text to remove special characters.
            remove stop words from the text
            perform stemming on the review text
            nCnt = nCnt + 1
            emit(nCnt + "::" + preprocessed_review_text)
```

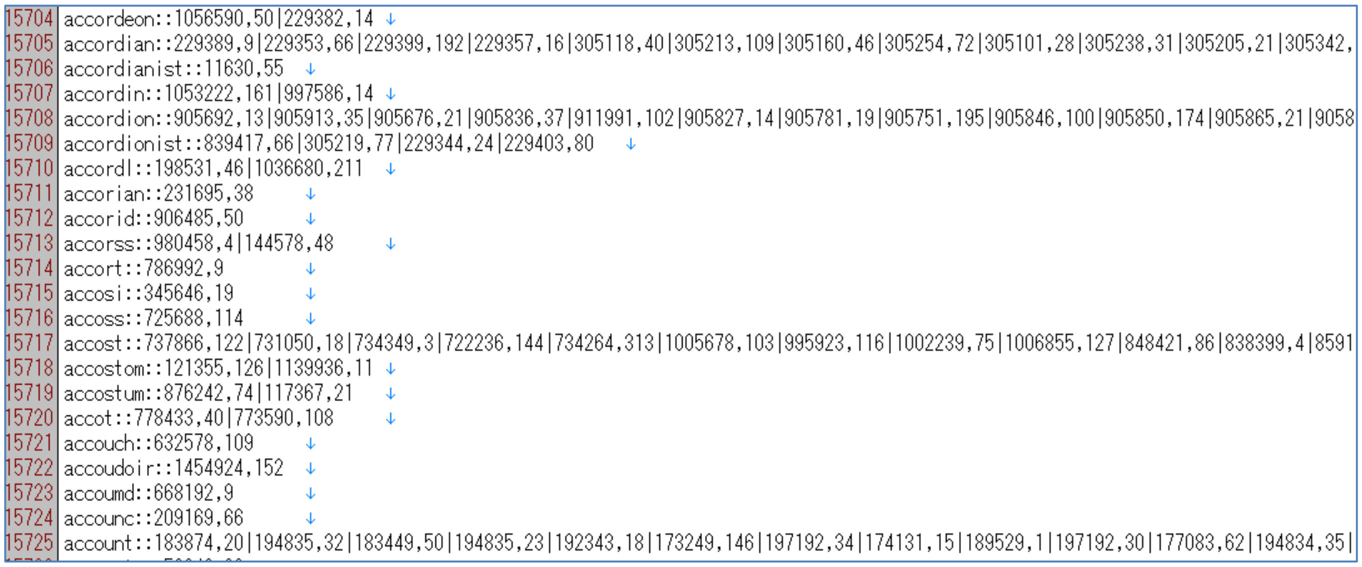
We then created two types of inverted indexes; one with posting information containing document ids and term positions and the other with posting information containing document ids and term frequency.

a) Inverted Index with positional info:
   This inverted index contains word and its positional info in the document.
   The size of the file created is 1.03 GB.

Pseudo Code:

```
input : preprocessed file

Mapper
--------
let map(k,v) =
      foreach line in stdin:
            data = split the line on "::"
            lineNum = data[0]
            reviewTxt = data[1]

            nPos = 0
            foreach word in reviewTxt:
                  emit(word, (nLineNum + "," + nPos))
                  nPos = nPos + 1

Reducer
---------
current_key = null
dictInvIdx = new Dictionary()

let addToDictionary(word, val):
    if !(word in dictionary dictInvIdx):
            create a list and add val to it.
            add this list to dictionary dictInvIdx for that word
      else:
            append the val for the word to the existing list value.

let emitInvIdx(word):
    invertedIndexes = dictInvIdx[word]
    invIdxStr = create a single string by joining the data on '|'
    print current_key + "::" + invIdxStr

let reduce(k,vals) =
      foreach line in stdin:
            Split line into a list to get(key and val)
            if current_key != null and current_key != key:
                  emitInvIdx(current_key)
```

```
                delete the old key from dictionary as its already emitted.
                current_key = key
                addToDictionary(current_key, val)
        else
                current_key = key
                addToDictionary(key, val)

    emitInvIdx(current_key)
```



```
accordeon::1056590,50|229382,14
accordian::229389,9|229353,66|229399,192|229357,16|305118,40|305213,109|305160,46|305254,72|305101,28|305238,31|305205,21|305342,
accordianist::11630,55
accordin::1053222,161|997586,14
accordion::905692,13|905913,35|905676,21|905836,37|911991,102|905827,14|905781,19|905751,195|905846,100|905850,174|905865,21|9058
accordionist::839417,66|305219,77|229344,24|229403,80
accordl::198531,46|1036680,211
accorian::231695,38
accorid::906485,50
accorss::980458,4|144578,48
accort::786992,9
accosi::345646,19
accoss::725688,114
accost::737866,122|731050,18|734349,3|722236,144|734264,313|1005678,103|995923,116|1002239,75|1006855,127|848421,86|838399,4|8591
accostom::121355,126|1139936,11
accostum::876242,74|117367,21
accot::778433,40|773590,108
accouch::632578,109
accoudoir::1454924,152
accoumd::668192,9
accounc::209169,66
account::183874,20|194835,32|183449,50|194835,23|192343,18|173249,146|197192,34|174131,15|189529,1|197192,30|177083,62|194834,35|
```

*Figure 1: Positional Index*

The above figure shows a snapshot of positional inverted index file.

b) Inverted index with term frequency data.
   We implement the value-to-key conversion design pattern.
   This inverted index contains the word and its frequency info in that document.
   The size of this file is 767 MB.

Pseudo Code:

Step1: We first compute the frequency of the word in the given document.

```
input : preprocessed file

Mapper1
--------
let map(k,v) =
        foreach line in stdin:
                data = split the line on "::"
                lineNum = data[0]
                reviewTxt = data[1]
```

```
            nPos = 0
            foreach word in reviewTxt:
                  emit((word,nLineNum), 1)

Reducer1
---------
current_key = null
sum = 0

let reduce(k,vals) =
      foreach line in stdin:
            Split line into a list to get(key and val)
            if current_key != null and current_key != key:
                  emit(current_key, sum)

                  sum = 0
                  current_key = key
                  sum = sum + val
            else
                  current_key = key
                  sum = sum + val

      emit(current_key, sum)
```

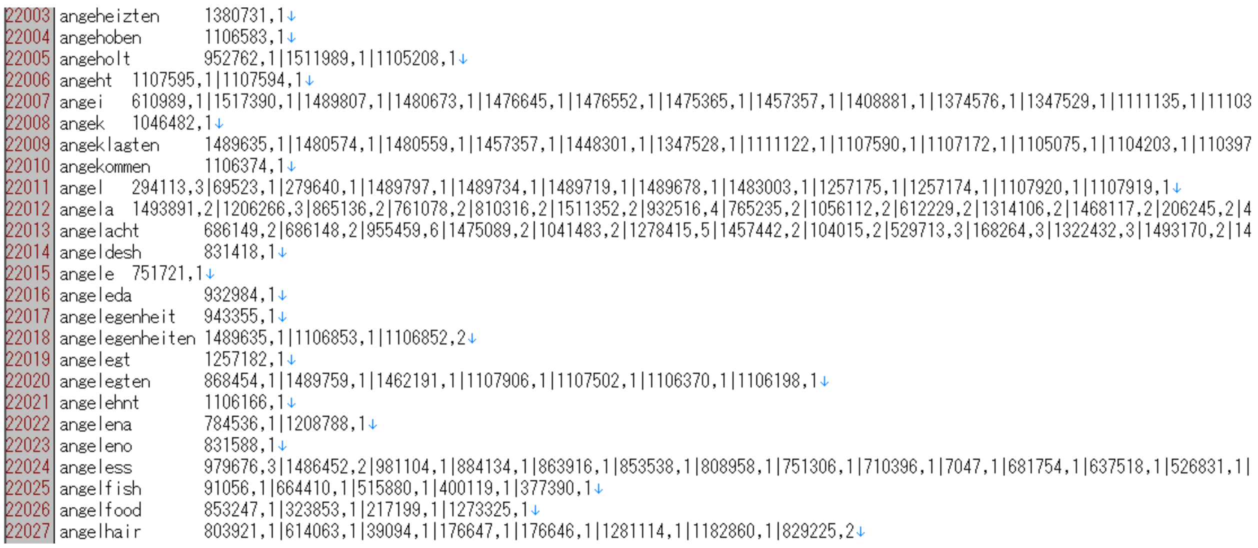
Step2: The input to this MapReduce task is partitioned on the key and then fed to the reducer. The mapper has two fields as keys but the reducer has only one key as the input.

```
input = output from the previous mapreduce task

Mapper2
--------
let map(k,v) =
      foreach line in stdin:
            data = split the line on "\t"
            key = data[0]
            val = data[1]
            emit(key, val)

Reducer2
---------
current_key = null
lstVal = new List()

let reduce(k,vals) =
      foreach line in stdin:
            Split line into a list to get(key1 and val1)
            split the key1 on "," to get wordkey and linekey
            if current_key != null and current_key != wordkey:
                  outVal = create a single string by joining the items in
lstVal by "|"
                  emit(current_key, outVal)

                  lstVal = []
                  current_key = wordkey
                  val2 = linekey + "," + val1
                  add val2 to lstVal

            else
                  current_key = wordkey
                  val2 = linekey + "," + val1
                  add val2 to lstVal

      outVal = create a single string by joining the items in lstVal by "|"
      emit(current_key, outVal)
```

Step3: This Map Reduce task is an auxiliary task and just sorts the output if it’s not sorted. This is done for the efficient searching within the Inverted index file.

```
Mapper3
--------
let map(k,v) =
      foreach line in stdin:
            data = split the line on "\t"
            key = data[0]
            val = data[1]
            emit(key, val)

Reducer3
---------
current_key = null
lstVal = new List()

let reduce(k,vals) =
      foreach line in stdin:
            Split line into a list to get(key and val)

            if current_key != null and current_key != key:
                  outVal = create a single string by joining the items in
lstVal by "|"
                  emit(current_key, outVal)

                  lstVal = []
                  current_key = key
                  add val1 to lstVal

            else
```

```
                current_key = key
                add val2 to lstVal

        outVal = create a single string by joining the items in lstVal by "|"
        emit(current_key, outVal)
```



```
angeheizten     1380731,1
angehoben       1106583,1
angeholt        952762,1|1511989,1|1105208,1
angeht  1107595,1|1107594,1
angei   610989,1|1517390,1|1489807,1|1480673,1|1476645,1|1476552,1|1475365,1|1457357,1|1408881,1|1374576,1|1347529,1|1111135,1|11103
angek   1046482,1
angeklagten     1489635,1|1480574,1|1480559,1|1457357,1|1448301,1|1347528,1|1111122,1|1107590,1|1107172,1|1105075,1|1104203,1|110397
angekommen      1106374,1
angel   294113,3|69523,1|279640,1|1489797,1|1489734,1|1489719,1|1489678,1|1483003,1|1257175,1|1257174,1|1107920,1|1107919,1
angela  1493891,2|1206266,3|865136,2|761078,2|810316,2|1511352,2|932516,4|765235,2|1056112,2|612229,2|1314106,2|1468117,2|206245,2|4
angelacht       686149,2|686148,2|955459,6|1475089,2|1041483,2|1278415,5|1457442,2|104015,2|529713,3|168264,3|1322432,3|1493170,2|14
angeldesh       831418,1
angele  751721,1
angeleda        932984,1
angelegenheit   943355,1
angelegenheiten 1489635,1|1106853,1|1106852,2
angelegt        1257182,1
angelegten      868454,1|1489759,1|1462191,1|1107906,1|1107502,1|1106370,1|1106198,1
angelehnt       1106166,1
angelena        784536,1|1208788,1
angeleno        831588,1
angeless        979676,3|1486452,2|981104,1|884134,1|863916,1|853538,1|808958,1|751306,1|710396,1|7047,1|681754,1|637518,1|526831,1|
angelfish       91056,1|664410,1|515880,1|400119,1|377390,1
angelfood       853247,1|323853,1|217199,1|1273325,1
angelhair       803921,1|614063,1|39094,1|176647,1|176646,1|1281114,1|1182860,1|829225,2
```

*Figure 2: Term Frequency Index*

These were the two techniques which we used to implement inverted indices and we compared their performance.
We loaded both the indices into the memory, one by one, we did that using python shell, once it was loaded in memory in the form of an associated array – dictionary in python, we performed 100 searches of random words (same words for both the indices) on both the indices and came averaged the per query result for both the indices
For positional index the per query result took 0.19 seconds
For term frequency index the per query result took 0.09 seconds

## 4. Clustering using Apache SPARK

This was, essentially, a sub task, that we hoped to accomplish in order to gain more insight to the Yelp dataset and to gain a better understanding of machine learning on Apache SPARK [3].
Firstly, I would like to compare the pseudo-codes for performing a simple K-Means clustering on Hadoop MapReduce compared to Apache SPARK.

<u>Pseudo-code for MapReduce</u>

```
Initialize centroids for clusters randomly

Mapper
method map(id, features)
```

```
  centroids < - read from file
  for every centroid do
    distance <- features - centroid
  emit(nearest centroid, features)

Reducer
method reduce(nearest centroid, features)
  for features in nearest centroid:
    centroid < - mean(features)
  save centroid to file
  emit(id, features)

Repeat till converged
```

Pseudo-code for Apache SPARK [4]

```
Initialize K clusters
centers = data.takeSample( false, K, seed)

Assign each data point to closest cluster
closest = data.map(p => (closestPoint(p,centers),p))

Assign each cluster center to be the mean of its cluster’s data points.
pointsGroup =  closest.groupByKey()
newCenters =pointsGroup.mapValues(ps => average(ps))
```

This just goes to show, how simple it is to code on SPARK as compared to MapReduce, although the psyche is almost the same, given that both have to be coded in terms of mappers and reducers, but the code-length is exponentially smaller nonetheless, which counts a lot when looking to making a data scientist’s life simpler.

## 4.1 Preprocessing

As the data was provided in JSON format, it had to be preprocessed to bring it to the desired format. The K-Means implementation only takes numeric inputs for feature vectors, so to that purpose, we had to convert the JSON objects, some of them nested deep, to a more appropriate feature vector, where in the strings were converted to integer values using a reversible hash [5], for non-trivial strings (trivial string include – true, false)

For ex,
A JSON excerpt from the business.json file

```
“categories” : {
        “parking” : {
                “street”: “true”,
                    “valet”: “false”
          }
}
```

Had to be mapped into a .csv format like this:

```
Parking*street, Parking*valet
1                   , 0
```

Which makes it more machine readable and SPARK readable.
All the objects, collectively, whether present in a particular instance or not, were mapped to a feature

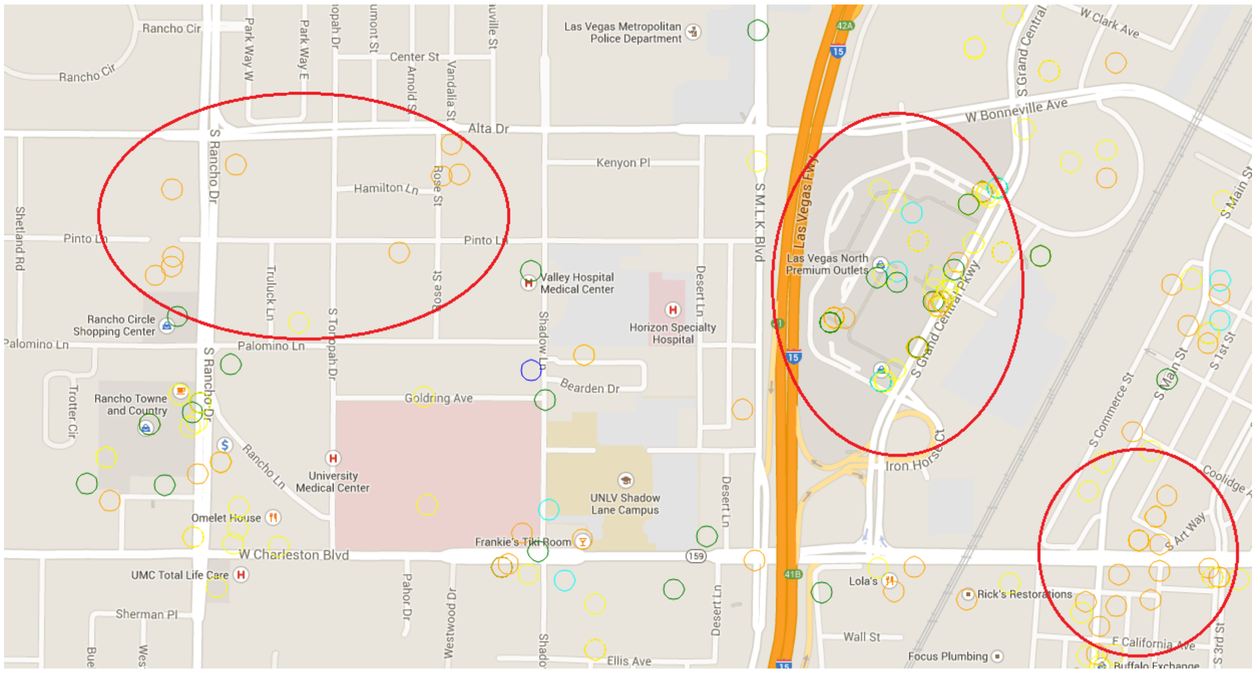
4.2 Clustering

Once the json objects were mapped to the feature space, 867 features to be precise, the data was now ready to be processed by the clustering algorithm.
We clustered on a per state basis, this was done to remove certain aspects and limitations that a particular state has on its business.
For ex, a business that closes down at midnight, due to the state laws might not get bad reviews, but a business that closes down before the other business, which are not mandated by the state laws, might receive bad reviews and lower ratings.
A more precise example would be that of a bar, as in some states the bars shut down due to state laws not allowing the sale of alcohol after certain hours, while some states do not have such laws.
Once the data was divided per state basis, 26 to be precise, they were run through the clustering algorithm separately.
First try was to cluster all the data with all 867 features at once, that provided to be wasteful, as conceived, it did not give any separable cluster structure in terms of rating, which is what we basically want.
Second try was to cluster only based on latitude, longitude and popularity (popularity = currentRatingCount/totalRatingCount) value, which unfortunately failed to give us any separable classes as well.
We gave a final try, which included only taking into account the features that seemed intuitive to contribute towards a business’ rating, for ex, parking, valet, payment methods, etc. Even that proved to be futile, as there still failed to be a separable cluster structure in terms of ratings.

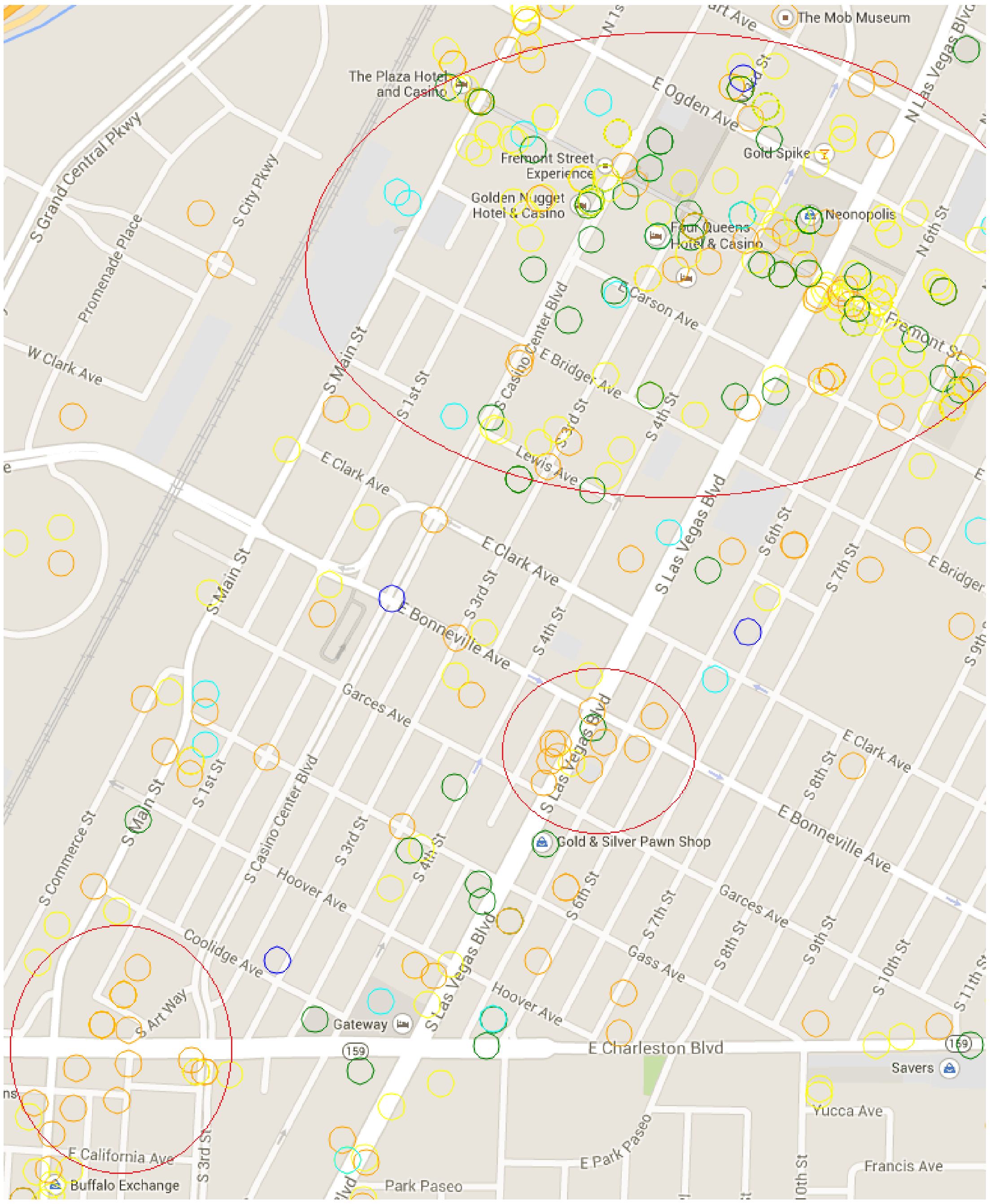
We believe, this might be happening due to us ignoring the most intuitive of feature separation factor. Although, we have kept the states separate, we failed to separately cluster each category, which makes sense, as different categories respond to different attributes when it comes to contributing towards a good rating.

For ex, A medical facility may not be judged by the type of it plays, although a recent study shows that the type of music played in the waiting room of a hospital contributes a lot towards patient’s families feeling more calm and composed.

Although we failed to find any direct relation between a business’ rating and its other features, we did uncover something interesting.



In this image, the businesses are mapped onto a google map using pygmaps – python library. The orange circles are businesses with 4-5 rating, yellow with 3-4, green wit 2-3, cyan with 1-2 and blue with 0-1 rating.
As you can see from this image, businesses that are more sparsely located almost always have a better rating than the ones located in a very dense formation.
This was not an isolated incident, one more image below shows the same pattern.



Although this does not really prove anything without doubt, but some further research into this matter will surely hold some interesting results.

## 5. Conclusion:

When comparing the two indices, we saw that building an inverted index using term frequencies rather than positional index proved better for search as well as storing the index.
As for clustering, we saw how to implement a machine learning algorithm using mllib in Apache SPARK, which didn't give us any good results but we believe further research using a more intuitive approach, as suggested above, should prove to be fruitful

## 6. References:

1 - Yelp Dataset Challenge - http://www.yelp.com/dataset_challenge
2 - Jimmy Lin and Chris Dyer - “Data-Intensive Text Processing with MapReduce
3 - Mllib Clustering - https://spark.apache.org/docs/latest/mllib-clustering.html
4 - Shivaraman Venkata – “Machine Learning on Spark”
5 – Persistence hashing of string in pyton -
http://stackoverflow.com/questions/2511058/persistent-hashing-of-strings-in-python