Combain AB

Combain Positioning System

DBSCAN_MOD in Scala

A clustering algorithm to reduce storage space.

Author:
Erik Norlander
erik@combain.com
eoj.norlander@gmail.com

Contents

1 Introduction

Combain is a company with many products but their CPS-serivce, which this report is focused around, tries to predict where IoT-devices are based on data from the wifi-connections this IoT-device has made. How exactly is a topic for another time, but it's safe to say that this requires vast amounts of data.

As the amount of data in the world becomes more and more abundant, data-driven businesses start to struggle with where to put it all. AWS RDS was the choice Combain went with, but this proved costly when the datasets grew. Therefore, they needed a way to make their existing data storage needs more cost effective while retaining computational accuracy for their CPS-service.

Entering DBSCAN, a renowned clustering algorithm based on density of points. This is different from algorithms such as K-Means and EM because it doesn't assume a specific shape of the clusters created. How exactly it works can be found on Wikipedia [1] (highly recommended before you continue) which is where I found the pseudo-code I based my Scala-version on. Albeit, with some modifications...

2 Quick Start

First of, let's get the application working. Clone and pull the git repository `https://gitlab.com/combainmobile/dbscan-spark` by opening the terminal on your unix-system and writing:

```
$ git clone https://gitlab.com/combainmobile/dbscan-spark
$ git pull
```

In the file `build.sbt` you will find the Scala, SBT and Spark versions needed to run the application. They obviously have to be installed. On Mac this would be done with a package manager like Homebrew.

```
$ brew install sbt
$ brew install scala
$ brew install apache-spark
```

In the `README.md` file you will find a quick start guide to get started on either AWS or your local machine. Simply follow the steps there and you'll be good to go!

2.1 Input

In order for the algorithm to run input data has to be formatted in a CSV file with the columns:

wifiId	gpslatitude	gpslongitude	foundtime
...	...	...	...

2.2 Output

The output comes in the form of a JSON-file which is described below. Where

- wifiId is internally used at Combain to classify wifis
- samples are the the amount of samples gathered on that wifi
- lat and lon are the mean of the cluster in coordinates
- Sxx, Syy and Sxy can be used to create a covariance matrix
- n is the amount of points in this cluster
- Str, timestap when the first point in the cluster was noticed
- End, timestap when the last point in the cluster was noticed


```
{
  "wifiId": "43",
  "samples": "3162",
  "clusters": [
    {
      "lat": "55.60249084415584",
      "lon": "13.00548020129867",
      "Sxx": "0.010248990005195076",
      "Syy": "0.03500580747519494",
      "Sxy": "2.854427766229388E-4",
      "n": "3080",
      "Str": "2010-03-07 14:55:31",
      "End": "2012-03-27 18:32:51"
    }
  ]
}
```


For more advanced users there are ways to get the information in CSV instead and multiple other ways to manipulate the output which will get discussed later.

3 What can be expected?

From a storage saving point of view the picture didn't look great when I first started analyzing the data available in Sweden. One would expect to save more space clustering a wifi with 1000 positions than one with 10. The resulting JSON-file even have more than 10 rows. As it turned out, about half of all wifis in Combains database of wifis Sweden had 10 or less samples as can be seen in figure 1. What also becomes clear from this figure is that the amount of wifis in the span 100 samples and higher are very small in comparison with the bulk of wifis available (less than 10% in fact).

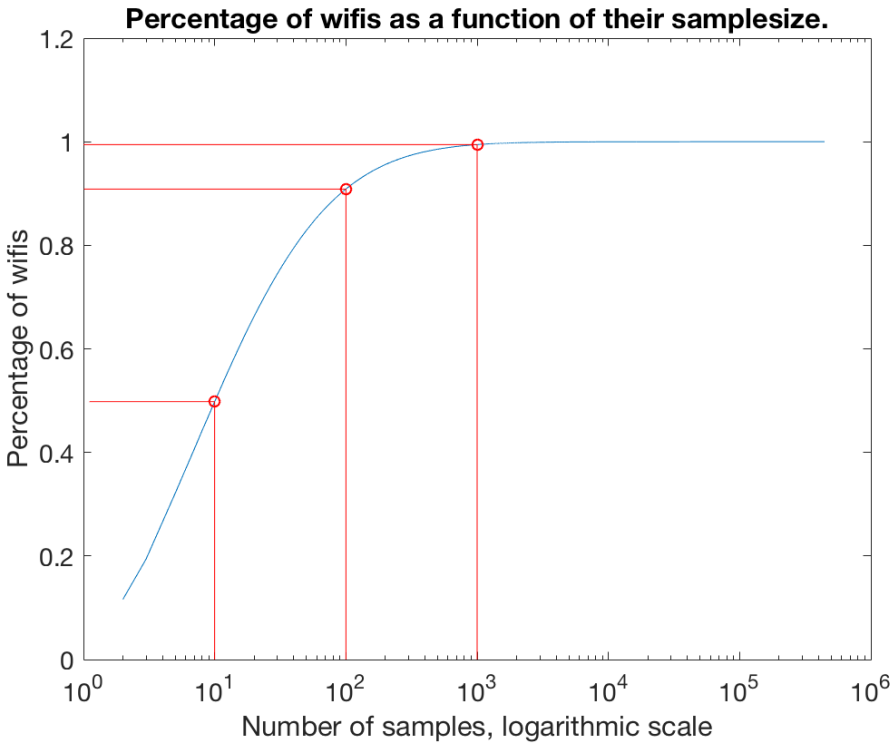

Figure 1: A graph describing the percentage of wifis with a certain amount of samples in Sweden.

So just how bad is this? Well, a rough estimate of how much memory is saved in each order of magnitude is:

- 1:10 in the span $10 <$ samples < 100.
- 1:50 in the span $100 <$ samples < 1000.
- 1:300 in the span $1000 <$ samples $< 10\ 000$.
- 1:1000 for samples $> 10\ 00$

From this, you wouldn't expect to save much space. However, when I instead looked at the *memory* each wifi-group occupies as a function of samples the prospects became much more promising. As can be seen in figure 2 the amount of memory that huge swath of wifis with 10 samples or less actually only occupies about 4% of the memory in the database. This means that 96% of the database still has something to gain from clustering.

So how much exactly can you expect to save? As an estimate I ran the computation for a large number of wifis in the span $10 <$ samples $< 10\ 000$ and found that the new, clustered data would have a size of about 2.2GB. In comparison with the original 75.1GB this is a decrease of 97% percent. As seen above, the data will compress even more for wifis with more than 10 000 samples.

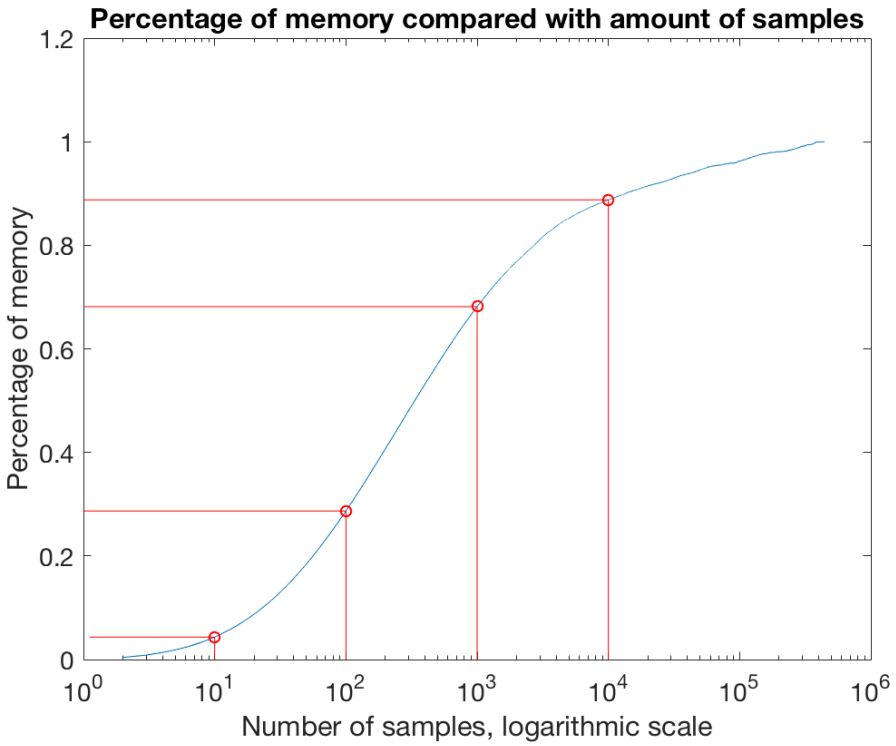

Figure 2: A graph describing the percentage of memory each wifi-group occupies as a function of sample size in Sweden.

Now that we know there's something to gain from clustering, let's take a look at how I approached the problem.

4 The workspace

In this section I want to highlight some of the important classes and methods used to compute the clusters. There are plenty of files in the workspace that is not necessarily used in the main function of the application which I am not going to spend any time on. Nor will I spend any time on the `build.sbt`, `project/assembly.sbt` and `spark_cluster.pem`-files. There are plenty of sources available online for reading about them if you wish.

4.1 Outside of other folders

In the directory `Distributer/src/main/scala` you'll find 3 files outside of any other folder. These are the files that are dictating what happens when you run the application.

- `Distributer.scala`: This is the main method that takes 4 arguments: `input_filepath`, `output_filepath`, `epsilon` and `minPts`. While the first 2 are self-explanatory, the last 2 will be subject to some discussion later. This is also where the `SparkContext` is created and the input file is converted to an RDD.

- `ClusterByWifi.scala`: Splits upp the data according to `wifiId` and performs the DBSCAN-algorithm on each of these data sets.
- `Writer`: A simple class to first create strings of the result and then output them to some filepath. This is where you have methods to output in different formats.

4.2 DBSCAN

This is the only folder that actually regards clustering, the rest are simply utilities, derived from the work done by Oskar Jermakowicz [2] which is why this is the only folder that's going to be discussed here. In this folder we have:

- `Compute.scala`: Does computations needed for DBSCAN.
- `Stats.scala`: Computes statistics of each cluster.
- `DBSCAN.scala`: Performs the DBSCAN-aglorithm.
- `DBSCAN_MOD.scala`: Performs DBSCAN_MOD-algorithm.
- `ReadWriter.scala`: Currently only used to format the JSON-file structure. Most of the code is legacy from when the application didn't run with RDD's.

4.2.1 `Compute.scala`

If you're experiencing issues with memory handling, such as the infamous [3]

```
java.lang.OutOfMemoryError: Java heap space error
```

this is where you should start optimizing. The issue arise because it saves the entire Euclidean Distance Matrix in the RAM while doing the calculations for DBSCAN. This is fine for small data sets, but anyone that knows their power series realize that the size of this matrix grows as N^2 if N is the number of samples.

```
/** Computes the euclidean distance matrix for all points in an array. */
def euclidean(coord : Array[(Double, Double)]) : Array[Array[Double]] = {
 val n = coord.length
  var euclidean = ofDim[Double](n,n)
  for (i <- 0 to n-1) {
    for (j <- i+1 to n-1) {
      euclidean(i)(j) = distance(coord(i)._1,
                                 coord(i)._2, coord(j)._1, coord(j)._2)
      euclidean(j)(i) = euclidean(i)(j)
    }
  }
  val t1 = System.nanoTime()
  return euclidean
}
```

This could be improved if one finds a way to simply save the upper triangle of the matrix, then the space would only grow half as fast ($N^2/2$). This proved quite difficult to do considering how the rest of the code was written, but it's possible. However the same issue will arise again for even larger data sets.

From a memory-point-of-view, best would be to calculate each row in the Euclidean Distance Matrix each time the DBSCAN-algorithms starts investigating that point. This would make the algorithm (very) slow, but not particulary memory dependent. Maybe there's a way to parallelize this?

There are also some methods to convert the geographical coordinate system to meters or vice verca. These methods could be improved as they rely on sampling two points in the data set to convert the distance.

4.2.2 `DBSCAN_MOD.scala`

The interested reader might already have noticed in the source code that the application actually doesn't use the included DBSCAN-algorithm (`DBSCAN.scala`) but rather something called `DBSCAN_MOD.scala`. During initial investigation and research some strange things were discovered with the original algorithm, which is why this modification was made.

For instance, it would frequently classify individual points as clusters. This is obviously unwanted if the goal of the application is to reduce storage space. It would also often classify points within obvious clusters as noise. Moreover, on occasion it classified clusters within other clusters. With these issues in mind me and Anders Mannesson, the resident algorithmic expert, changed the algorithm in some fundamental ways.

I will now take you through the basic building blocks of the algorithm, which are the same as in `DBSCAN.scala`, and describe the difference between this modified version and the original.

`notNoise`

```
/** Investigates which points are noise and returns them.
@param epsilonMeters is radius between points to be clustered in meters.
@param minPts  is minimum required points within each the radius epsilon
               to be considered part of the cluster.
@param data    is input data of coordinates.
@return rtn    which is a an array of information about each cluster. */
def notNoise(minimumPoints : Int, epsilonMeters : String,
        data : Array[(Double, Double, String)])
        : Array[(Double, Double, Double, Double, Double,
                          Double, String, String)] = { ... }
```

This is the part of of the algorithm that keeps track of which points is classified as which cluster. Performed by the vector `IDX` which is a logical vector that has an index of each point as well as an integer describing which cluster that point belongs to.

Here the mayor difference from the `DBSCAN` algorithm is that there is no `isnoise`-vector. In the previous version, once one point was regarded as noise, there were no going back. This is the reason why points that would obviously be part of a cluster was categorized as noise. In effect, this meant that the original algorithm was non-deterministic, as it depended on what end of the data you started reading.

The central logic of this method will be discussed after I introduce the 2 other important methods. For now we just need to know that the following is defined:

```
val prepData = data.map(x => (x._1, x._2))
val comp = new Compute(prepData)
val epsilon = comp.toLatLong(epsilonMeters)
var C = 1
val D = comp.euclidean(prepData)
val n = D.length
var IDX = Array.fill(n){0}
```

regionQuery

```
/** Finds the amount of neighbours within a radius of epsilon.*/
def regionQuery(i : Int) : Array[Int] = {
  var indexD = D(i).zipWithIndex
  var filterD = indexD.filter{ case (k,_) => k < epsilon }
  return filterD.map(_._2)
}
```

This method is almost identical to the one in `DBSCAN.scala` and is the reason why improving the memory handling I talked about before becomes problematic. `D` is the Euclidean Distance Matrix and the result from `regionQuery` is entirely based on the information it holds. Therefore, if one wanted to change the structure of `D`, this method would also have to change.

expandCluster

This is a method with quite a tricky logic but with a straight forward purpose; to increase the amount of points regarded as "neighbours" and therefore categorize the cluster. The largest difference from the original algorithm is from row 8, where `A` becomes the set of new possible neighbours which are not yet classified or part of the cluster we're building.

Continuing on, `logical` becomes the union of `neighbours2` and `A` and the `expanded` adds on `neighbours2` if the difference between the amount of neighbours being tried and the amount of neighbours in this set that are classified as part of the cluster or

unclassified is larger than `minPts`.

This might seem counter intuitive, but line 10 is what makes this algorithm take points that previously were classified as noise and therefore could never be tried again have a new chance at being part of the cluster being built. In effect, making this algorithm deterministic, where the original was not (this is related to the DBSCAN*-algorithm).

```
/** Increase the amount of neighbours included. */
1. def expandCluster(i : Int, neighbours : Array[Int], C : Int) {
2.   IDX(i) = C
3.   var expanded = neighbours
4.   var k = 0
5.   while (k < expanded.length) {
6.     var j = expanded(k)
7.     var neighbours2 = regionQuery(j)
8.     var A = neighbours2.map(IDX).filter{ case (a) => a == 0 || a == C }
9.     var logical = neighbours2.map(IDX) diff A
10.    if (neighbours2.length - logical.length >= minPts) {
11.      var temp = expanded ++ neighbours2
12.      expanded = temp.distinct
13.    }
14.    if (IDX(j) == 0) {
15.      IDX(j) = C
16.    }
17.    k += 1
18.  }
19.  val t1 = System.nanoTime()
20.}
```

Taking another look at `notNoise`

Here the algorithm runs the `regionQuery` method for all uncategorized points and expands the cluster if the size difference of the available neighbours and uncategorized points are larger than `minPts`. In effect, this sets the minimum cluster size to what ever you have set as `minPts`. This eliminates the possibility of getting clusters that are "too small" in our estimation. Noting, there's some ambiguity to the definition of "too small".

```
for (i <- 0 to n-1) {
  if (IDX(i) == 0) {
    var neighbours = regionQuery(i)
    if ((neighbours.length - neighbours.map(IDX)
                         .filter{case (a) => a!=0}.length) >= minPts) {
      expandCluster(i,neighbours,C)
      C += 1
    }
```

```
  }
}
```

5 Appropriate ε and minPts

Now that the reader is familiar with the modified DBSCAN-algorithm, it's time to talk about the input parameters ε and minPts; a topic that I've noticed many find confusing so I want to delve into it in its own section.

As you might have seen on Wikipedia [1] they present figure 3 where I want to highlight that their version of the algorithm categorizes border points as part of the cluster as well which differs from DBSCAN_MOD. However, the illustration and caption is still useful to get a feeling for the input parameters.

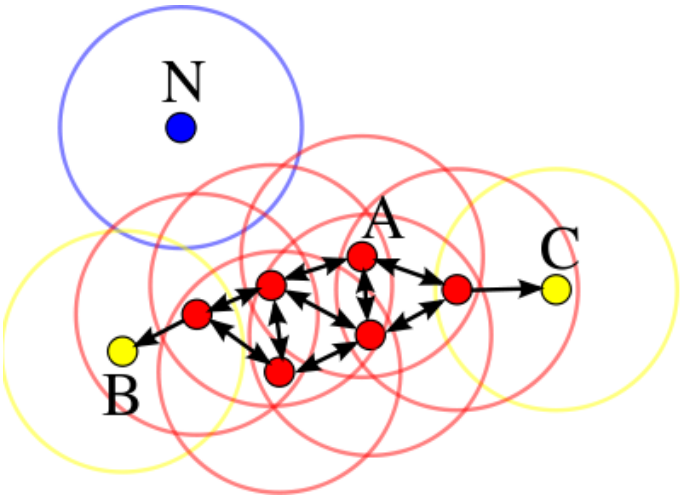

Figure 3: "In this diagram, minPts = 4. Point A and the other red points are core points, because the area surrounding these points in an ε radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and *thus belong to the cluster as well.* Point N is a noise point that is neither a core point nor directly-reachable." [1].

Shubert [4] suggests minPts to be twice as large as the dimensionality of the data set. This would put minPts to 4 in our 2D-case. However, since we're using DBSCAN_MOD and not DBSCAN it's reasonable to think of minPts as the "minimum size of clusters" which gives the user some power to set a threshold on the clustering. Here figure 1 and 2 give us some context, and for the purposes of this report I will use minPts = 10.

Now, ε is a different story. You might have gathered from the caption of figure 3 it's not simply a radius of a cluster but rather the maximum distance within which a point has to have minPts neighbours in order to be part of the cluster. If that sounds like a mouthful, it's because it is. Furthermore, it's very difficult to set this

parameter as data can have wildly different properties even though we're studying WIFI-positions.

However, some sort of requirement has to be determined as the clusters has to be compared on equal terms in order to decide if the WIFI should be classified as invalid or not. Therefore, I've been using $\varepsilon = 100$ throughout my tests. This seem to produce comparable results.

6 Running on AWS EMR

When you launch a cluster to AWS EMR you'll need to specify the input and output paths as folders in a S3-bucket you own. Preferably following the structure layout in figure 4. When that's done you upload the jar file and launch an EMR Cluser as specified in the `README.md`-file.

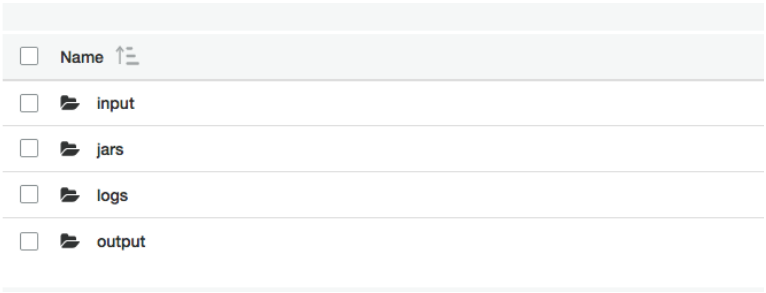

Figure 4: The S3-folders required to run the application.

In order to make this really deployable a way to put all input data into the input folder on S3 is required as well as a way to fetch the output. This data has to be fetched with an SQL query so that it follows the structure in the Quick Start guide.

How and when exactly a new clustering shall be run, creating a new EMR cluster and running a server is not obvious.

6.1 Valid or invalid?

We will classify a wifi as invalid if:

- No clusters are created.
- More than two clusters that are overlapping in time.
- Semi-major axis of a clusters covariance ellipse is larger than 500 meters.
- Noise is more than 50% of the dataset.

If more than two clusters are produced, but they are not overlapping in time, it suggests that the wifi has been moved. Therefore, all points associated with the older cluster should be classified as invalid.

6.2 When to cluster?

The purpose of this application is partly to reduce the amount of storage space on AWS but also to decrease the amount of calculations. If a new point is inside a valid cluster, we therefore don't do a new computation until a certain threshold. A statistically skilled reader will realize that the mean and variance will change if points are added to a cluster. Meaning, we will have to cluster the data again at some point.

However, in order to give a good answer to this question a lot more research will have to be done. Therefore, I leave this up to the user to decide at a later stage.

7 Conclusion & Discussion

The DBSCAN_MOD algorithm not only does the job in lowering storage space, but also opens up new possibilities to analyze data. In the future it's feasible to find new ways of categorizing a moving WIFI, roads and other cases that are lowering the quality of the data today. Overall, it seems to be a step in the right direction in creating higher quality data for Combains positioning services.

When you'd like to perform the clustering for datasets with more than 40 000 samples, I would recommend to allocate more heap space to the JVM that runs on AWS. I did not find a way to do this, but locally it can be done by adding

```
export SBT_OPTS="-Xms512M -Xmx16g -Xss2M -XX:MaxMetaspaceSize=16g"
```

to your `.bash_profile`. Maybe there's a way to do the same on AWS. Here you can set the maximum available space used by JVM. I'm limited to 16GB which is where you'll reach the limit of 40 000 points but an EC2-instance should be able to allocate far more memory.

References

[1] *Read about DBSCAN on wikipedia.*

[2] *Scalable processing of globally crowd-sourced geolocation data, by Oskar Jermakowic*

[3] *A discussion on java.lang.OutOfMemoryError: Java heap space*

[4] *DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN by E. Shubert et al.*