Clustering faults iterating KMeans

Andrea Ranieri

https://github.com/andr3aranieri

andrea.ranieri@protonmail.com

Introduction

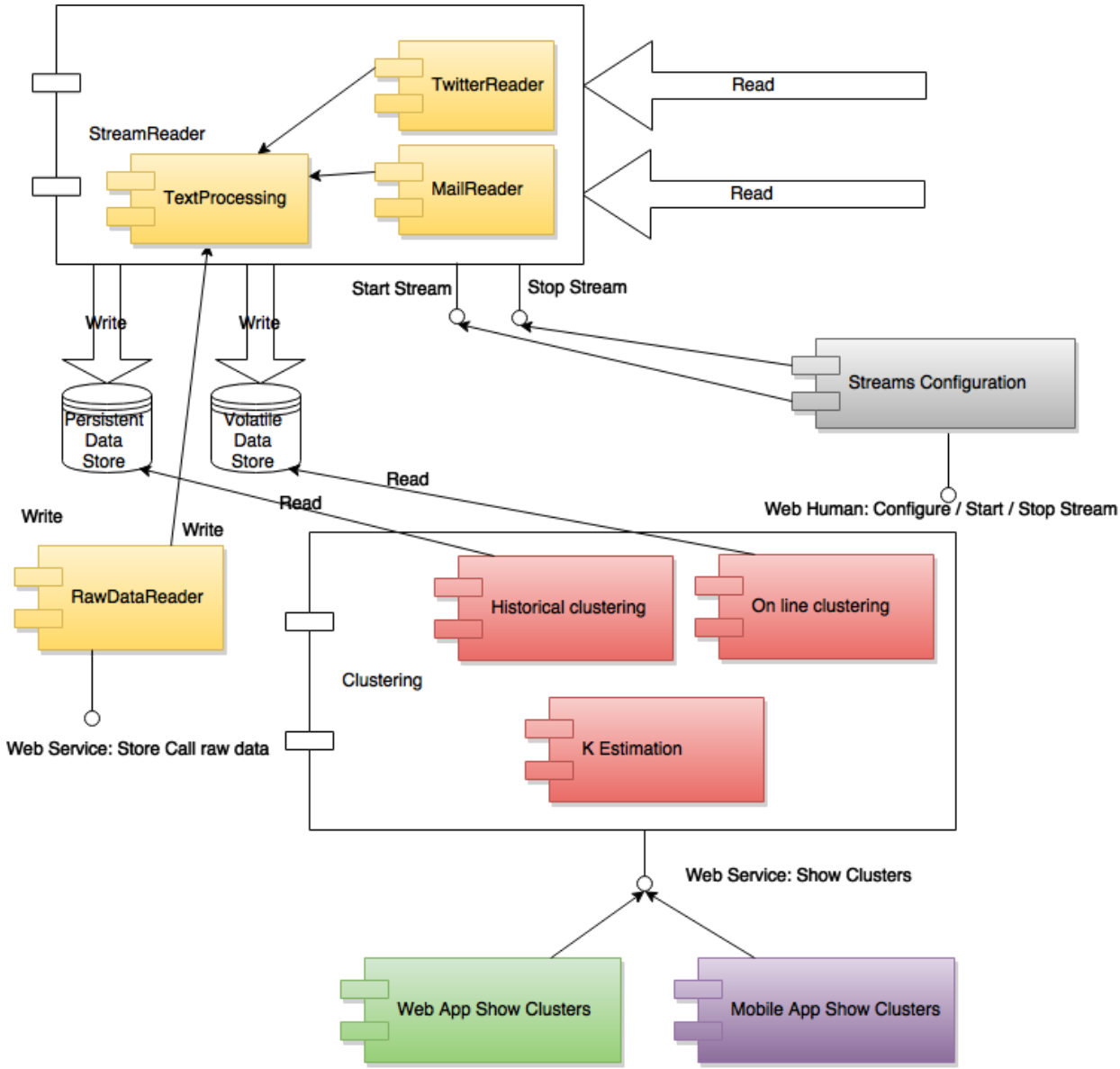
A fault is essentially a text description of what's not working as it should.

The goal of this project is to try to estimate the number of cluster in a dinamically populated set of faults. Essentially i'll try to estimate the K in K-Means.

The initial estimate will be done using the Silhouette coefficient (https://en.wikipedia.org/wiki/Silhouette_(clustering)). Starting from this, the FaultsClustering Hadoop job will be iterated to evaluate if the actual K-Clusteing is natural.

Project Description

In the following picture, the architecture of the system.



The basic idea is to implement a StreamReader server which lets users to create new source streams from which read raw data.
Raw data are problem descriptions: they will be preprocessed and mapped to a N-Dimensional tfidf Euclidean space (N = number of terms).

In this project i will implement 2 StreamReader modules:

- TwitterReader: i'll modify the python twitter reader implemented in the homework; It will permit to download a twitter stream filtered by account (account mentions and direct messages), by location, or by keywords. *Telco* could try to mine the whole

twitter to detect information abount him (search "*Telco* name" + "sms" + "problem" in a certain location);
- MailReader: i'll implement a python pop3 mail client which periodically download mail from a configured account. This could be used to intercept mails sent to control room or to any problem-notification mail box.

I will also implement a simple RawDataReader web service, which lets external call centers software systems, or *telco* customer mobile app, send problem description programmatically.

These 3 modules will send received data to the TextProcessing module (remove punctuation, remove stopwords, lemming,...) which will save formatted docs in datastore: each doc will have the following structure:

doc_i\t tfidf_i1 tfidf_i2 tfidf_i3 tfidf_i3 …. tfidf_iN

tfidf_ij, 1 <= i <= M, 1 <= j <= N (N: num terms in the document collection; M: num docs in document collection)

I'll represent each doc with a vector in the N-Dimensional tfidf euclidean space.

Documents distance will be chosen after some test; it could be configurable (clustering modules could choose a distance measure in function of the clustering they have to make).

For online clustering, space dimension will grow each time we receive docs with new terms.

For offline clustering the space dimension could be not efficiently storable in memory, depending on the chosen time period of analysis.

To control memory usage dimension, i could use real tfidf space until a certain dimension N_max is reached, and an estimated N_max dimension space after that. The estimated tfidf space uses an hash function to hash terms in a fixed number of buckets (N_max buckets): 2 terms that hash to the same bucket, will have the same estimated tfidf.

The streamreader module will be a python multithreading batch program: each new stream is executed in a different thread.

There will be 2 datastores:

- Persistent data store that will store docs for a certain period (for example a year);
- Volatile data store that will store daily docs;

Stored docs will be clusterized by 2 clustering modules:

- Historical clustering: clusterizes docs in a specified time period;
- On line clustering: gives the online daily docs clustering;

The 2 clustering modules will be implemented in hadoop.

The number of clusters will be estimated by clustering modules.

The "Show Clusters" web service (Soap or RESTfull) will be implemented in java. This web service will be used by mobile app and web app show clusters, and it could be used by external existing monitoring / notification systems.

StreamReader Module

To start the module, from “StreamerModuleFolder”:

```
python StartStreamerModule.py
```

It’s a python multithreaded module which contains 2 different modules:

- TwitterReader;
- MailReader;

It has a Stream Socket interface listening on port number 8899 that can be tested using *telnet*

```
telnet 127.0.0.1 8899
```

The command has this structure:

```
op:param1=value1|param2=value2|...|paramN=valueN##STOP##
```

TwitterReader

It reads a twitter stream of any of these kinds:

- Direct Messages and mentions of a specified account;
- Public tweets filtered by keywords and location;

For public tweets are provided the following filters:

- **locations**: the bounding box which indicates the area from which we want to receive tweets. es. ‘12.341707,41.769596,12.730289,42.050546'
- **track**: keywords. They can be provided
  - separated by space to obtain tweets with each keyword. es “problema sms vodafone”;
  - separated by comma to obtain tweets with any of the keyword. es “problema LTE tim, rallentamento LTE tim”;

A command example to create a public twitter stream is:

```
twtr_pub:idUser=1|loc=12.341707,41.769596,12.730289,42.050546|kw=LTE TIM##STOP##
```

To create a private twitter stream, the user create a twitter app connected to his account and pass its configuration parameters to the streamer module. The command will have the following form:

```
twtr_pri:idUser=1|cku=kONfnsrP16egMgBr39f9o3He3|csu=Xe1zdwUzT1MOfcqPaZuzZ9jJEj4hFx
6M1nwtwc62ESs43J4Rfc|atu=255160588-ZDtazGlaU95uIA0oJCNVNxiY1H3t9YlpEMFIaMvp|atus=N
EUk61nw7VeIG0GPPd5gASe2qDXGbeSEPi0TeSkTSK3Bi##STOP##
```

where

- cku: comsumer key;
- csu: consumer secret;
- atu: access token;
- atus: access token secret;

MailReader

This module implements an imap client that periodically downloads mails from the configured account.

It can be configured an imap folder from which to download mails. The default value is 'INBOX'.

A command example to create a mail stream is:

```
mail_ima:idUser=1|server=imap.gmail.com|userName=and.ranieri.datamining2015@gmail.
com|password=datamining2015|folder=ticket##STOP##
```

To avoid duplicates, sfter downloading the UNSEEN emails in the configured folder, it sets their flag to SEEN. A future improvement of the algorithm will be to avoid this not to alter human user mail inbox interation.

Closing a stream

The command to close a stream:

```
clos_str:23##STOP##
```

closes the stream with id 23.

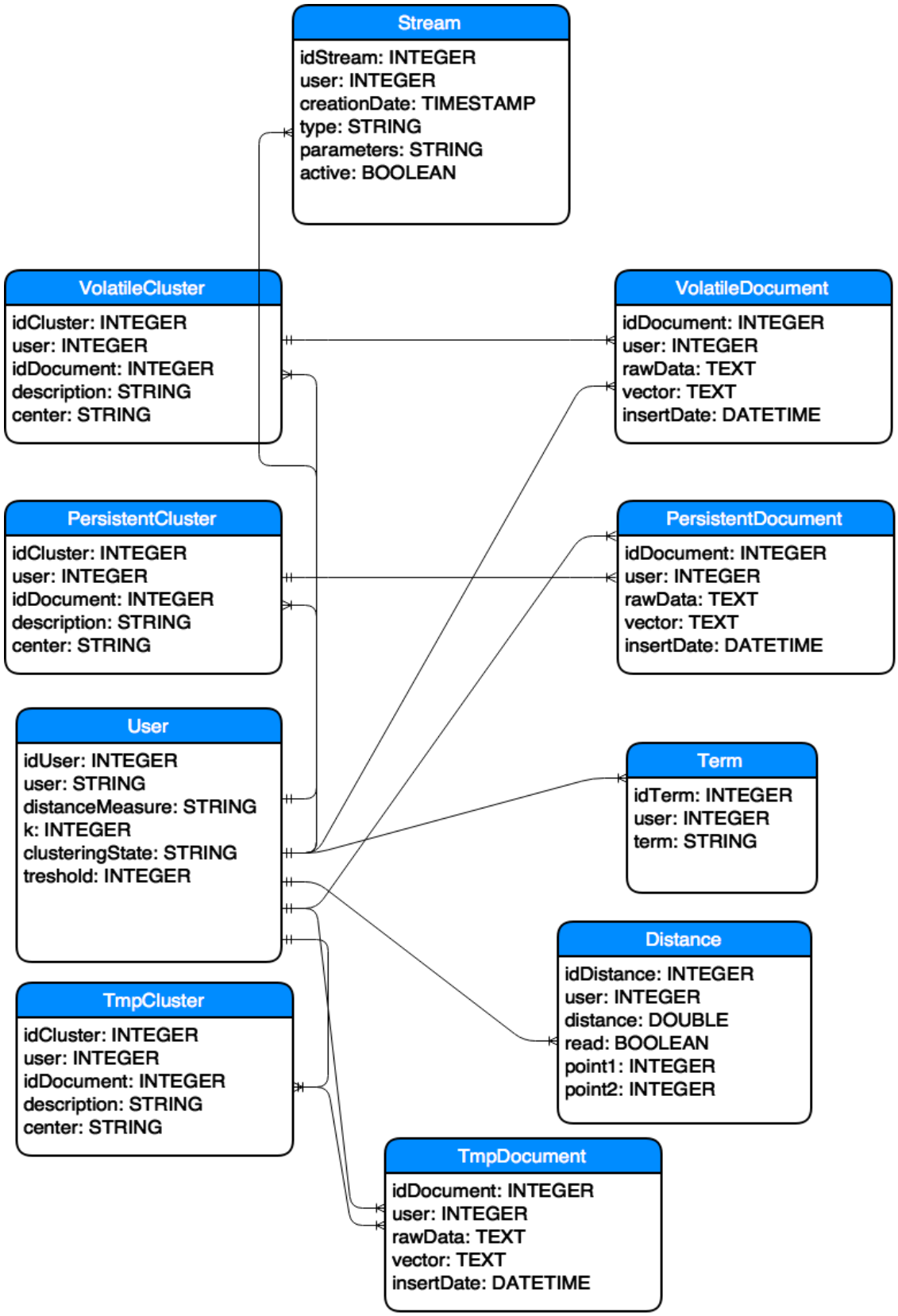
Module Structure

```
./Module
./Module/__init__.py
./Module/BusinessLogic
./Module/BusinessLogic/__init__.py
./Module/BusinessLogic/MailReader.py
./Module/BusinessLogic/TextPreprocessManager.py
./Module/BusinessLogic/ThreadDispatcher.py
./Module/BusinessLogic/TwitterReader.py
./Module/ModuleInterface
./Module/ModuleInterface/__init__.py
./Module/ModuleInterface/StreamerServer.py
./Module/ResourceManagementLayer
./Module/ResourceManagementLayer/__init__.py
./Module/ResourceManagementLayer/DB
./Module/ResourceManagementLayer/DB/__init__.py
./Module/ResourceManagementLayer/DB/DBConfig.py
./Module/ResourceManagementLayer/DB/DBManager.py
./Module/ResourceManagementLayer/DB/PersistentDocumentDB.py
./Module/ResourceManagementLayer/DB/StreamDB.py
./Module/ResourceManagementLayer/DB/VolatileDocumentDB.py
./Module/ResourceManagementLayer/Entity
./Module/ResourceManagementLayer/Entity/__init__.py
./Module/ResourceManagementLayer/Entity/PersistentDocument.py
./Module/ResourceManagementLayer/Entity/Stream.py
./Module/ResourceManagementLayer/Entity/VolatileDocument.py
./Module/ResourceManagementLayer/Manager
./Module/ResourceManagementLayer/Manager/__init__.py
./Module/ResourceManagementLayer/Manager/DocumentManager.py
./Module/ResourceManagementLayer/Manager/StreamManager.py
./StartStreamerModule.py
```

The StreamerModule is a 3-tier python application:

1. ***ModuleInterface***: provides a TCP socket interface (configurable port, default = 8899);
2. ***BusinessLogic***: contains the logic needed to parse a client request, to start and end a stream, to preprocess received documents;
3. ***ResourceManagementLayer***: contains the logic to access datastore:
    a. DB submodule: direct access to the particular datastore manager (MySQL DBMS in this case);
    b. Entity submodule: logical rappresentation of datastore entities;
    c. Manger submodule: interface to datastore for business logic module;

DataBase


Stream
idStream: INTEGER
user: INTEGER
creationDate: TIMESTAMP
type: STRING
parameters: STRING
active: BOOLEAN
VolatileCluster
idCluster: INTEGER
user: INTEGER
idDocument: INTEGER
description: STRING
center: STRING
VolatileDocument
idDocument: INTEGER
user: INTEGER
rawData: TEXT
vector: TEXT
insertDate: DATETIME
PersistentCluster
idCluster: INTEGER
user: INTEGER
idDocument: INTEGER
description: STRING
center: STRING
PersistentDocument
idDocument: INTEGER
user: INTEGER
rawData: TEXT
vector: TEXT
insertDate: DATETIME
User
idUser: INTEGER
user: STRING
distanceMeasure: STRING
k: INTEGER
clusteringState: STRING
treshold: INTEGER
Term
idTerm: INTEGER
user: INTEGER
term: STRING
Distance
idDistance: INTEGER
user: INTEGER
distance: DOUBLE
read: BOOLEAN
point1: INTEGER
point2: INTEGER
TmpCluster
idCluster: INTEGER
user: INTEGER
idDocument: INTEGER
description: STRING
center: STRING
TmpDocument
idDocument: INTEGER
user: INTEGER
rawData: TEXT
vector: TEXT
insertDate: DATETIME

User table is used to adapt the system to different users needs: each user can create his own streams and the system will cluster his documents. It contains user clustering configuration:

- distance measure chosen for this user;
- k value, the estimated number of clusters for a certain user. This value is updated by the system as explained next in the document;
- clustering state, used by java hadoop jobs that compute clustering for the user: using this value, we can return to user consistent clustering informations when i requests them (through web service invocation);

The ***Term*** table contains the term vocabulary for documents of a specific user. Fixing a user value, the term table number of rows determines the VolatileDocument and PersistentDocument vectors dimension. As explained next in the document, depending on this dimension, the system could decide to hash terms to contain memory usage.

TmpCluster table is used in hadoop ***K estimation*** job: a thread periodically computes clustering simulations on user's daily data, using user's actual K value, and stores results in TmpCluster table.

This clustering simulations are evalued by a hadoop ***Silhouette*** job, which uses ***Distance*** table to compute Silhouette coefficient of each point in this clustering: if overall Silhouette coefficient is good enough (boolean method that decides it), K value is OK, and the real user clustering will be done with K.

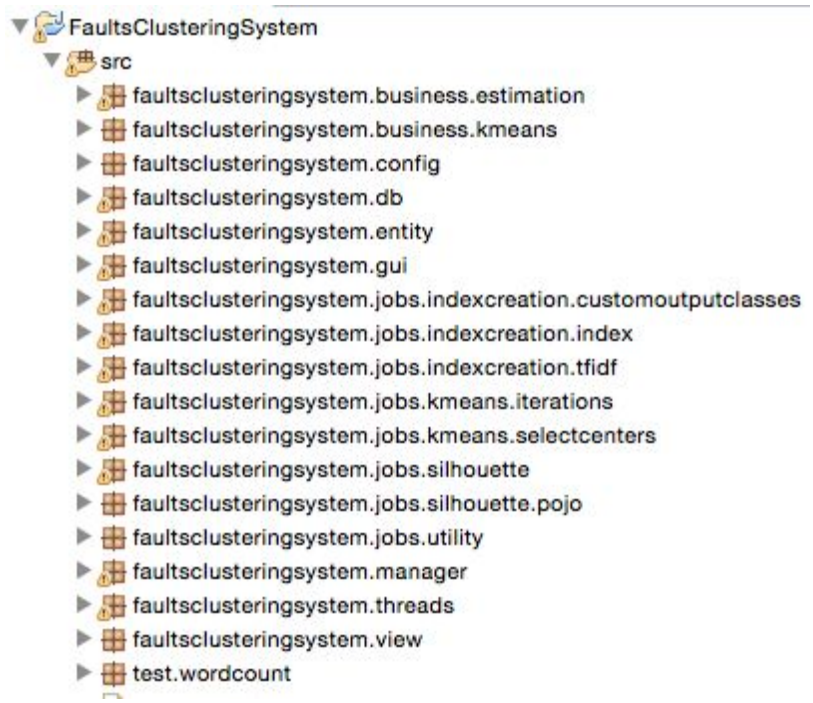
Initial K estimation is computed as follows

```
public int getInitialKEstimation(List<VolatileDocument> documents) {
    int M = documents.size(); //num documents
    int N = 0; //num terms
    int T = 0; //num non-zero entries

    MyVector vector = null;
    for(VolatileDocument vd: documents) {
        vector = new MyVector(vd.getVector(), vd.getIdVolatileDocument() + "");
        if(N == 0) {
            N = vector.getInnerVector().length;
        }

        for(double d: vector.getInnerVector()) {
            if (d > 0) {
                T += 1;
            }
        }
    }

    return (M * N) / T;
}
```

To be able to execute hadoop ***Silhouette*** job efficiently, Distance table is updated periodically by a thread ***DistanceMeasureThread*** launched by java FaultsClusteringSystem application.

FaultsClusteringSystem

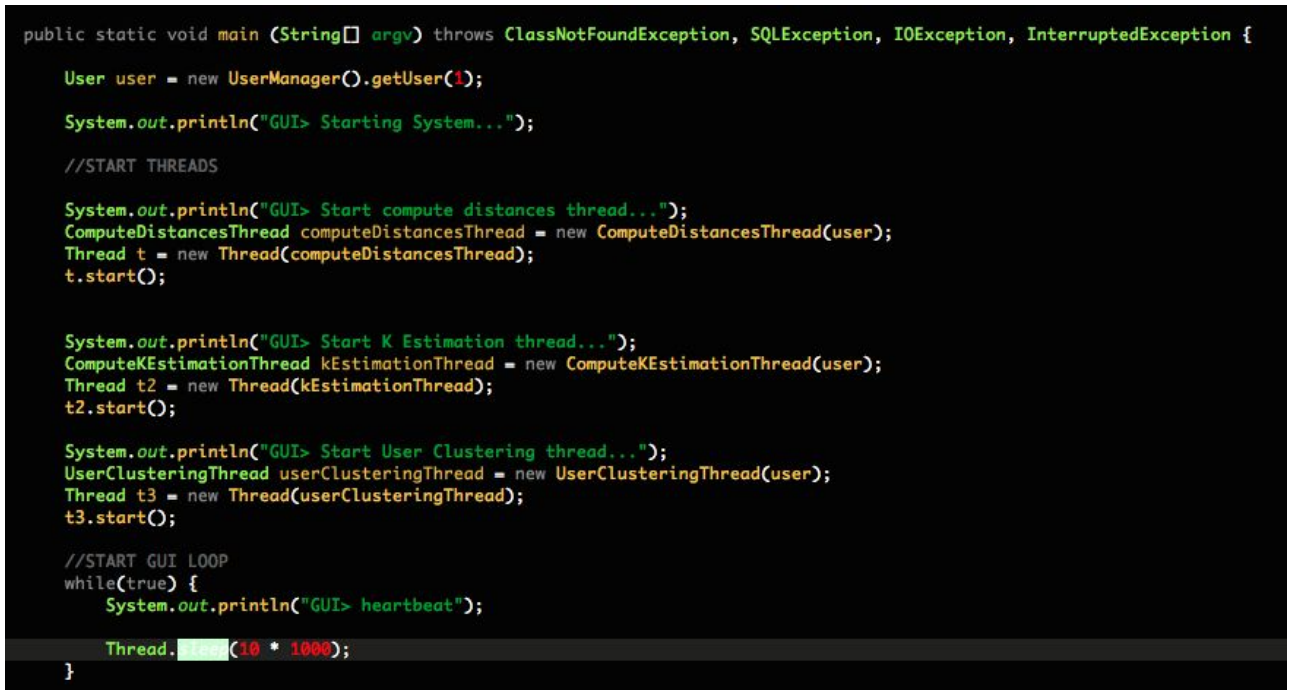
It's the clustering application. It's presented as an Eclipse project compiled as a jar executed in Hadoop.



To execute the application:

```
hadoop jar FaultsClusteringSystem.jar faultsclusteringsystem.gui.StartSystem
```

The application is a 3 thread process. In the following image the main method:



```
public static void main (String[] argv) throws ClassNotFoundException, SQLException, IOException, InterruptedException {

    User user = new UserManager().getUser(1);

    System.out.println("GUI> Starting System...");

    //START THREADS

    System.out.println("GUI> Start compute distances thread...");
    ComputeDistancesThread computeDistancesThread = new ComputeDistancesThread(user);
    Thread t = new Thread(computeDistancesThread);
    t.start();


    System.out.println("GUI> Start K Estimation thread...");
    ComputeKEstimationThread kEstimationThread = new ComputeKEstimationThread(user);
    Thread t2 = new Thread(kEstimationThread);
    t2.start();

    System.out.println("GUI> Start User Clustering thread...");
    UserClusteringThread userClusteringThread = new UserClusteringThread(user);
    Thread t3 = new Thread(userClusteringThread);
    t3.start();

    //START GUI LOOP
    while(true) {
        System.out.println("GUI> heartbeat");

        Thread.sleep(10 * 1000);
    }
```

DistanceMeasureThread

Periodically computes distances between each documents couple iterating twice documents space (O(N^2)). The algorithm is very expensive, so the execution frequency can be decided in function to the number N of documents, modifiying the thread sleep time between 2 executions.

During the 2 annidated iterations, i use a hashmap to compute `dist(doc1, doc2)` only once, considering that `dist(doc1, doc2) = dist(doc2, doc1)`. The key used in the hashmap is

```
doc1.id + ":" + doc2.id
```

The computation output is a bulk insert query in the form:

```
INSERT INTO Distance(idUser, distance, hasBeenRead, point1,
point2) VALUES (1, 0.00123, 0, doc1ID, doc2ID),(1, 0.01313, 0,
doc1ID, doc3ID),....
```

to be able to efficiently write distances to DB.

Insert operation is done atomically together with delete of distances for this user:

```java
public boolean insertDistances(int idUser, String bulkInsertQuery) throws ClassNotFoundException, SQLException {
    Connection conn = null;
    int ret = 0;
    try {
        conn = this.dbManager.getConnection();
        conn.setAutoCommit(false);

        //delete old distances;
        PreparedStatement stmt = (PreparedStatement) conn.prepareStatement("DELETE FROM Distance WHERE idUser=?");
        stmt.setInt(1, idUser);
        ret = stmt.executeUpdate();

        //bulk insert new distances;
        Statement stmt2 = (Statement) conn.createStatement();
        stmt2.execute(bulkInsertQuery);

        conn.commit();
    }
    catch(Exception ex) {
        conn.rollback();
        ex.printStackTrace();
    }
    finally {
        conn.close();
    }

    return ret > 0;
```

KEstimation Algorithm

It's implemented by the following method. It makes a clustering simulation with the actual K value, and estimates if it's a natural clustering.

```
public void estimationExe(User user) throws SQLException, ClassNotFoundException, IOExcepti
    System.out.println("************************************************************");
    System.out.println("************************************************************");
    System.out.println("****************K ESTIMATION EXECUTION**********************");
    System.out.println("************************************************************");

    System.out.println("K Estimation> copy volatile documents in tmp documents");
    this.documentManager.copyVolatileDocumentInTmpDocuments(user.getIdUser());

    System.out.println("K Estimation> create today tmp documents index");
    this.indexCreationEstimation.createIndexToday(user);

    System.out.println("K Estimation> compute kmeans on tmp documents");
    this.kMeansEstimation.kMeansToday(user);

    System.out.println("K Estimation> compute tmp clusters silhouette");
    this.silhouette.computeSilhouette(user);

    System.out.println("************************************************************");
    System.out.println("********************END K ESTIMATION************************");
    System.out.println("************************************************************");
    System.out.println("************************************************************");

}
```

It works on a copy of volatile datastore, realized by db tables `TmpDocument` and `TmpCluster`

It's divided in 3 steps:

1. index creation of today documents;
2. kmeans on today documents using index created;
3. evaluation of silhouette index of points in computed clusters, to determine if actual K value is good enough.

The output of this algorithm is a list of silhouette values for clustered points:

```
s(i) = (b(i) - a(i)) / max(a(i), b(i)) (silhouette value for doc i)
```

where

```
a(i) = avg distance of i from other points in its cluster

b(i) = min distance between i and points in other clusters
```

If most of the points have a silhouette near to value 1, the clustering is natural, so we use the actual K value for real clustering (the clustering returned to users). Otherwise, we make another clustering simulation with K+1.

User Clustering Algorithm

```
public void kmeansExe(User user) throws ClassNotFoundException, SQLException, IOException, InterruptedExcept

    System.out.println("************************************************************");
    System.out.println("************************************************************");
    System.out.println("*****************CLUSTERING EXECUTION*******************");
    System.out.println("************************************************************");

    System.out.println("Clustering Execution> create today documents index");

    this.indexCreation.createIndexToday(user);

    System.out.println("Clustering Execution> compute k means clustering");

    this.kMeans.kMeansToday(user);

    System.out.println("************************************************************");
    System.out.println("**********************END CLUSTERING************************");
    System.out.println("************************************************************");
    System.out.println("************************************************************");

}
```

Index Creation

This method creates inverted index matrix taking in input documents as list of terms. It uses 2 hadoop jobs to:

1. compute tfidf coefficient for terms;
2. create inverted index matrix;

```
    System.out.println(JOBNAME + "Read documents from DB...");
    // Read raw documents from DB;
    File f = this.documentManager.getTodayDocuments(user.getIdUser());

    // each user has an input dir and an output dir;
    String inputDir = "/FinalProject/IndexCreation/input_" + user.getIdUser();
    String tfidfOutputDir = "/FinalProject/IndexCreation/output_" + user.getIdUser();
    String indexOutputDir = "/FinalProject/IndexCreation/outputindex_" + user.getIdUser();

    this.hadoopManager.tryToCreateDirectory(inputDir);

    System.out.println(JOBNAME + "Write documents to HDFS...");
    // write raw documents to HDFS IndexCreation job input directory;
    this.hadoopManager.writeFileToHDFS(f.getAbsolutePath(), inputDir);

    System.out.println(JOBNAME + "Delete old tfidf output directory...");
    this.hadoopManager.deleteHDFSDirectory(tfidfOutputDir);

    System.out.println(JOBNAME + "Launch tfidf matrix creation job...");
    this.hadoopManager.launchTfidfMappingJob(inputDir, tfidfOutputDir);

    System.out.println(JOBNAME + "Delete old index output directory...");
    this.hadoopManager.deleteHDFSDirectory(indexOutputDir);
    System.out.println(JOBNAME + "Launch index creation creation job...");
    this.hadoopManager.launchIndexCreationJob(tfidfOutputDir, indexOutputDir, "VolatileDocument");

    System.out.println(JOBNAME + "Make bulk update to DB...");
    this.hadoopManager.readFilesFromHDFSToDB(indexOutputDir);
}
```

The output of the index creation hadoop job has to be written to DB: job reducer writes a list of SQL updates command

```
@Override
public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    try {
        String vector = "";
        for (Text value : values) {
            vector += value + "-";
        }

        vector = vector.substring(0, vector.length() - 1);

        String updateQuery = "UPDATE " + this.documentTable + " SET vector='" + vector + "' WHERE " + this.primaryKey + "=" + key.toString();

        System.out.println(updateQuery + "\n");

        // write update query in output key;
        context.write(key,  new Text(updateQuery));
        ;
    } catch (Exception ex) {
        StringWriter sw = new StringWriter();
        ex.printStackTrace(new PrintWriter(sw));
        String exceptionAsString = sw.toString();
        context.write(new Text("ERRORE"),  new Text(exceptionAsString));
        System.err.println(exceptionAsString);
    }
}
```

that are atomically written to DB with a bulk update.

KMeans

It's a classical map reduce KMeans implementation. It's implemented in 2 hadoop jobs:

1. select k centers;
2. iteration job to compute clusters till convergence to a treshold:
    a. mapper assigns documents to k selected center;
    b. reducer computes new k centers and decides to stop iterations if they differ from old ones less than the specified treshold;