Problem 1

The first exercise is about understanding NBI and deriving a new model by estimating a parameter of the original formula of NBI.

The problem of NBI is the following: Consider a recommendation system based on NBI with $U_1, ..., U_n$ users and $O_1, ..., O_m$ objects. Let a_{xj} the utility matrix (with reviews of objects from user x to object j)

$$w_{ij} = \frac{1}{\Gamma(i,j)} \sum_{l=1}^{n} \frac{a_{il} * a_{jl}}{D(u_l)}$$

The matrix $(w_{ij}) \in \mathbb{R}^{m \times m}$ estimate how much an object i will be liked by a user that likes object j. The Γ function is actually dependent on the model(for NBI It is $\Gamma(i,j) = D(o_j)$, that is the degree of object j in the bipartite graph representation).

The recommendation is done by multiplying the matrices of W and A:

$$R = WA$$

The rating predicted for object x for the user i is given by unrolling the matrix multiplication:

$$\hat{r}_{xi} = \sum_{l=1}^{m} \frac{1}{\Gamma(x,l)} \sum_{h=1}^{n} \frac{a_{xh} * a_{lh}}{D(u_h)} \times a_{li}$$

We transform the formula in the following way:

$$v_{xl} = \sum_{h=1}^{n} \frac{a_{xh} * a_{lh}}{D(u_h)} \qquad \gamma_{xl} = \frac{1}{\Gamma(x,l)} \implies$$

$$\hat{r}_{xi} = \sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}$$

Given the set of rating pairs $R = \{(user1, object1), (user1, object2)...\}$, the resultant RMSE that will act as a cost function is defined as:

$$RMSE = \frac{1}{|R|} \sqrt{\sum_{(x,i) \in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}$$

In the first problem, there are 2 tasks that needs to be done:

1. Find the **Gradient descent** equations(for estimating the parameters γ) that minimizes RMSE

2. Implementing the algorithm or the **Stochastic** variant and evaluating the performance on data at http://web.kuicr.kyoto-u.ac.jp/supp/yoshi/drugtarget

It must be taken into account that the cost function(RMSE in this case) must be **convex** to keep the sequence of values of the cost function monotonically decreasing(also work for a class of quasi-convex functions), otherwise the method could reach a local minima or saddle point. The RMSE is convex ([1]) so no problem

Getting the formula

For the first task, the equation that are needed to do the Gradient Descent or variants are :

- Getting the derivative of the cost function(RMSE) to get the gradient(all the partial derivatives for every parameter are computed)
- update the current parameters by subtracting the gradient multiplied by a number that could change during the algorithm(known as **learning rate schedule**) and control if the error is within a certain limit/threshold. If It is not, the computation return to readjusting the parameters, if It is, the algorithm have found a local minima(for convex cost function).

The partial derivation of the RMSE for every parameter $\lambda_{x'l'}$ is the following:

$$\frac{\partial}{\partial \gamma_{x'l'}} RMSE = \frac{\partial}{\partial \gamma_{x'l'}} \frac{1}{|R|} \sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2} =$$

$$\frac{1}{2|R|\sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}} \frac{\partial}{\partial \gamma_{x'l'}} [\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2] =$$

$$\frac{1}{2|R|\sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}} \sum_{(x,i)\in R} \frac{\partial}{\partial \gamma_{x'l'}} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2 =$$

$$\frac{1}{2|R|\sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}} \sum_{(x,i)\in R} 2((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi}) \frac{\partial}{\partial \gamma_{x'l'}} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi}) =$$

$$\frac{1}{2|R|\sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}} \sum_{(x,i)\in R} 2((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})(v_{x'l'} * a_{l'i})$$

From this derivation, the partial derivative of RMSE by $\gamma_{x'l'}$ is the following:

$$\frac{\partial}{\partial \gamma_{x'l'}} RMSE = \frac{v_{x'l'} * \sum_{(x,i)\in R} (a_{l'i})((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})}{|R| \sqrt{\sum_{(x,i)\in R} ((\sum_{l=1}^{m} \gamma_{xl} * v_{xl} * a_{li}) - r_{xi})^2}}$$

To update the current parameter, the equation is the following:

$$\gamma_{x'l'}^{n+1} = \gamma_{x'l'}^{n} - \psi_n \frac{\partial}{\partial \gamma_{x'l'}} RMSE(\gamma_{x'l'}^{n})$$

where ψ_n(learning rate) could be costant or adaptive, we follow a costant approach(that causes oscillations on the RMSE) but an hibrid approach with the following step sizes could be done(Barzilai-Borwein method):

$$\psi_n = \frac{|(\gamma_n - \gamma_{n-1})^T [\nabla RMSE(\gamma_n) - \nabla RMSE(\gamma_{n-1})]|}{||[\nabla RMSE(\gamma_n) - \nabla RMSE(\gamma_{n-1})]||^2}$$

Implementing the algorithm

Algorithm 1: Main spark code

```
try{
        val spark = SparkSession.builder().
            master("local[*]").
            appName("spark_session").
            getOrCreate()
        val data = spark.sparkContext.textFile("data/e_admat_dgc.txt").
        mapPartitionsWithIndex({
            (idx, iter) => if (idx == 0) iter.drop(1) else iter
        }).map(parseText(_)).cache()

        val degreesUsers = data.flatMap(arr => { //?
            arr.zipWithIndex.map({
                case (value,key) if value > 0=> (key,1)
                case (value,key) if value <= 0=> (key,0)
            })
        }).
        reduceByKey(_ + _).sortByKey().values.collect()

        val degreesObjects = data.map(arr => { //?
            arr.count(value => value>0)
        }).collect()

        val collectedData = data.collect
```

```
        val vMatrix = data.zipWithIndex().map({case (arrX,key) => {
                val tmpArr = collectedData.map(arrY =>{
                    arrX.zip(arrY).zip(degreesUsers).
                        map(x => (x._1._1 + x._1._2)/x._2).
                        filter(!_.isNaN()).
                        sum
                })
                (key,tmpArr)
            }
        }).sortByKey().values

        var parameters = data
        var rmse = 1.0
        val scheduleCoefficient = 0.01

        for(i <- 1 to 100){
            val gradientTuple = modifiedGradientRDD(data,
                parameters,
                vMatrix)
            rmse = gradientTuple._2
            parameters =subtractRDD(parameters,
                multiplyRDD(gradientTuple._1,scheduleCoefficient))
            println("finished " + i + " iteration with an RMSE of " + rmse)
        }

        println("RMSE of the estimated ratings after GD is: " + rmse)



    } catch {
        case e: Exception => {
            System.err.print("Exception in main")
            e.printStackTrace()

        }
        case _: Throwable =>
            println("Got some other kind of Throwable exception")
    } finally {
        spark.stop()
    }
```

Along the main class some methods have been written to refactor the code:

Algorithm 2: accessories methods

```
    def parseDouble(s: String): Option[Double] = Try { s.toDouble }.
     toOption

def parseText(s:String):Array[Double] = {
    s.split("\t").drop(1).map(rating => {
        val tmpDoub = parseDouble(rating)
```

```
            if (tmpDoub.isDefined){
                tmpDoub.get
            } else 0.0
        })
    }

    def myzip(matrix1: RDD[Array[Double]],
        matrix2: RDD[Array[Double]]): RDD[(Array[Double], Array[Double])] = {
        val matrix1Keyed: RDD[(Long, Array[Double])] = matrix1.
            zipWithIndex().
            map { case (n, i) => i -> n }
        val matrix2Keyed: RDD[(Long, Array[Double])] = matrix2.
            zipWithIndex().
            map { case (n, i) => i -> n }
        matrix1Keyed.join(matrix2Keyed).sortByKey().map(_._2)
    }

    def invertRDD(matrix:RDD[Array[Double]]):RDD[Array[Double]] = {
        matrix.flatMap(arr => arr.
            zipWithIndex.
            map({ case (value:Double,key:Int) =>(key,value)}) ).
            groupByKey().
            map({ case (key,iterator) => iterator.toArray})
    }

    def subtractRDD(matrix:RDD[Array[Double]],
        matrix2:RDD[Array[Double]]):RDD[Array[Double]] = {
        myzip(matrix,matrix2).map(moreArrays => {
            moreArrays._1.zip(moreArrays._2).
                map(ratingsValues =>ratingsValues._2-ratingsValues._1)
        })
    }

    def multiplyRDD(matrix:RDD[Array[Double]],
        multiplier:Double):RDD[Array[Double]] = {
        matrix.map(arr => {
            arr.map(x => x * multiplier)
        })
    }
```

Algorithm 3: Gradient and RMSE computation

```
    def modifiedGradientRDD(matrix:RDD[Array[Double]],
        parameters:RDD[Array[Double]],
        vmat:RDD[Array[Double]]):(RDD[Array[Double]],Double)={
        val Rcardinal = matrix.count() * matrix.first().size
        val zippedParVmat = myzip(parameters,vmat).
            map(moreArr => moreArr._1 zip moreArr._2).collect()
        val estimatedRatings = matrix.
            zipWithIndex().
            map({case (rowReviews,key) => {
```

```
                val tmpArr = zippedParVmat.map(arrParVmat =>{
                    arrParVmat.
                        zip(rowReviews).
                        map(x => (x._1._1 * x._1._2 * x._2) ).sum
                })
                (key,tmpArr)
            }
        }).sortByKey().values

    val errorMat = subtractRDD(estimatedRatings,matrix).collect()

    val denominator = Rcardinal * math.
        sqrt(errorMat.flatMap(arr=>
            arr.map(doubValue => doubValue*doubValue)).reduce(_ + _))
    val rmse = (1.0/Rcardinal) * math.
        sqrt(errorMat.flatMap(arr=>
            arr.map(doubValue => doubValue*doubValue)).reduce(_ + _))


    val tmp = matrix.map(arr => {
        errorMat.map(errorArr=>{
            errorArr.
                zip(arr).
                map(result => result._1 * result._2).sum
        })
    })

    val gradientMat = myzip(tmp,vmat).map(moreArrays => {
        moreArrays._1.
            zip(moreArrays._2).map(x => x._1 * x._2 / denominator)
    })

    (gradientMat, rmse)
}
```

Problem 2

For the second task, a variant of min-hashing will be analyzed. This variant will consider only a subset of the rows to do the hashing. With this variant, rises the possibility of getting an **"unknown"** result for a column that has no ones in any row considered. The task at hand is to analyze the probability of getting this **unknown** result.

Probability of getting unknown results with the modified min-hashing

To prove that the probability of getting an unknown result is at the most $(\frac{n-k}{n})^m$, conditional probabilities will be used (combinatorial statistics could also be used). We choose k rows that

represent the subset of rows used by the modified version of minhashing, the other $n-k$ rows are not considered during minhashing. The probability of getting an unknown result during execution is the probability of getting an unfortunate sequence of selection and permutation of the 1 values for a row, that is setting m values in the column to 1 and the remaining $n-m$ values to 0(by selecting random no setting is done). The probability of inserting(taking) a row that has 1 in the $n-k$ columns after i steps of insertion(marking the rows taken) is :

$$\mathcal{P}(ins[n-k]|i) = \frac{n-k-i}{n-i}$$

The probability of inserting at every step the row with 1 in the wrong part of the rows(getting the rows that are empty) is the following (derived from the chain rule):

$$\mathcal{P}(\text{unknown}) = \prod_{i=0}^{m-1} \frac{n-k-i}{n-i} = \frac{n-k}{n} * \frac{n-k-1}{n-1} * ... * \frac{n-k-m+1}{n-m+1}$$

Because $0 < k \leq n$ and $0 < m \leq n$, we get the following inequality:

$$\frac{n-k}{n} * \frac{n-k-1}{n-1} * ... * \frac{n-k-m+1}{n-m+1} < \frac{n-k}{n} * \frac{n-k}{n} * ... * \frac{n-k}{n} = (\frac{n-k}{n})^m$$

That is what we wanted to prove.

Upper bound to the probability of getting unknown records

As the previous section stated, the probability of getting an unknown result from the modified minhashing is $(\frac{n-k}{n})^m$. The objective of this task is to bound this probability by e^{-10}, for perfect accuracy the probability will be set equal to this value. By definition of e and generalization to exponentiation, $\lim_{n\to\infty}(1+\frac{y}{n})^n = \lim_{n\to\infty}(\frac{n+y}{n})^n = e^y$, When n is fixed $n \in \mathbb{N}$,$(\frac{n+y}{n})^n < e^y$, as $0 < m \leq n$ and $0 < k \leq n$ by the definition of the problem(m rows with 1 are only a subset of all the n rows), we get $(\frac{n-k}{n})^m \leq (\frac{n+y}{n})^n < e^y$, by setting $y = -10$, we get a sequence of **inequalities** that we need to resolve to get k.

$$(\frac{n-k}{n})^m \leq (\frac{n-10}{n})^n \implies$$

$$(\frac{n-k}{n}) \leq (\frac{n-10}{n})^{\frac{n}{m}} \implies$$

$$k \geq n[1-(\frac{n-10}{n})^{\frac{n}{m}}]$$

So if we get a k that is greater than or equal to that value we get the solution to the problem.

Problem 3

For the third problem, we need to implement an Algorithm in Map-Reduce to obtain the set distance from two given datasets of points. The definition of Set distance from every element of the other set is the following: Given two point sets $P = p_0, p_1, ..., p_n$ and $S = s_0, s_1, ..., s_k$ in a metric space (m,d) with $k \leq \sqrt{n}$ the distance between every point of S to P is defined as

$$d(s, P) = \min_{p \in P} d(s, p)$$

Because the points in P are more than points in S, the set that is passed as input to the map is set P, while set S will be passed to cache.

The file of points (both S and P) contains points with the following structure:

$$(\text{pointId}, coord_1\ coord_2\ ...\ coord_m)$$

All points should have the same number of dimensions(if points have different number of dimensions, the program will exit with an error).

To implement an algorithm to compute all the distances we pass the dataset with less data in distributed cache ,the code is the following:

Algorithm 4: Loading set S to distributed cache

```
try{
    Path set2Path = new Path(args[1]);
    job.addCacheFile(set2Path.toUri());
}catch(Exception e){
    System.out.println("File to cache not found");
    System.exit(1);
}
```

After loading the points of set S to distributed cache, the cache is read from the mapper and the points are loaded in memory to a List of Points (represented by ArrayLists) that will be used to obtain distances, the snippet of code of the **setup** method is the following:

Algorithm 5: setup to read points in S from cache to memory

```
public void setup(Context context)throws IOException,
  InterruptedException{
    pointIds = new ArrayList<String>();
    pointCoordinates = new ArrayList<ArrayList<Double>>();
    URI []cacheFiles = context.getCacheFiles();

    if(cacheFiles!=null && cacheFiles.length>0)
    {
        try
        {
            //lists of identifiers and coordinates in cache
```

```
                String linePoint ="";
                FileSystem fsPoints = FileSystem.get(context.
                    getConfiguration());
                Path getFilePathPoints = new Path(cacheFiles[0].toString());

                BufferedReader reader = new BufferedReader(
                                        new InputStreamReader(
                                            fsPoints.open(getFilePathPoints)));

                while((linePoint = reader.readLine())!=null)
                {
                    String []words = linePoint.split("[,]");

                    if(!words[0].isEmpty()){
                        String pId = words[0].substring(1);
                        pointIds.add(pId);
                        String[] strCoords = words[1].
                            substring(0,words[1].length()-1).split(" ");

                        pointCoordinates.add(new ArrayList<Double>(Arrays.
                            stream(strCoords).
                            map(Double::parseDouble).
                            collect(Collectors.toList())));
                    }

                }


            }catch(Exception e) {
                System.out.println("Unable to read the file in mapper (setup)");
                System.err.println("Unable to read the file in mapper (setup)");
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
```

After loading the points in memory, the mapper is called. The mapper is called on every point of P and will compute the euclidean distance from the single point in P to every point in S, at the end of the computation, the key will be the name of the point in S and the value will be the distance computed. The code is below:

Algorithm 6: Map

```
    public void map(Text key, Text value, Context context) throws IOException,
        InterruptedException {

        String pointCordsStr = value.toString().replaceAll("[()]","");

        try{
            if(!pointCordsStr.isEmpty()){
```

```
                String[] strCoords = pointCordsStr.split(" ");
                Text resultkey = new Text();
                DoubleWritable resultvalue = new DoubleWritable();

                ArrayList<Double> tmpPointCoord = new ArrayList<Double>(Arrays.
                    stream(strCoords).
                    map(Double::parseDouble).
                    collect(Collectors.toList()));
                for(int i=0;i<pointCoordinates.size();i++){
                    double tmpDist = pointDistance(
                        tmpPointCoord,pointCoordinates.get(i));
                    resultkey.set(pointIds.get(i));
                    resultvalue.set(tmpDist);


                    context.write(resultkey, resultvalue);
                }
            }
        } catch(Exception e){
            System.err.println("Exception during construction of parts in map");
            System.out.println("Exception during construction of parts in map");
            System.err.println("Value of key in mapper is ("+key.toString());
            e.printStackTrace();
            System.exit(1);
        }
    }
```

The reduce will search for the minimum distance from the values passed as an iterator and will return the point in S and the minimum distance, code below:

Algorithm 7: Reduce

```
    public void reduce(Text key,
      Iterable<DoubleWritable> values,
      Context context)
      throws IOException,InterruptedException {
        try{
            DoubleWritable result = new DoubleWritable();
            double minimumDist=Double.MAX_VALUE;
            for (DoubleWritable val : values) {
                if(minimumDist>val.get())
                    minimumDist = val.get();
            }
            result.set(minimumDist);
            context.write(key, result);

        }catch(Exception e){
            System.err.println("Exception during construction of parts in reduce");
            System.out.println("Exception during construction of parts in reduce");
            System.err.println("Value of key in reducer is " + key.toString());
            e.printStackTrace();
```

```
                System.exit(1);
            }

        }
```

After this computation the distance defined by the problem is implemented.

References

[1] breckbaldwin. Convexity of (root) mean square error, or why committees won the netflix prize. "https://lingpipe-blog.com/2009/09/29/convexity-of-root-mean-square-error-or-why-committees-won-the-netflix-prize/".