Minimizing Round-Off Error

Karl Mutch

UCB Introduction to Web Algorithms, 2010.

GOAL:

This weeks' project was done to try and expand my math knowledge in relation to recommendation techniques in chapter 3.

INTENDED RESULTS:

The goal is to successfully improve the recommendation engine error, using RMSE, based on the 4th TODO item from Chapter 3.

The main thrust of the two pass method for calculating variance is to make use of summation and then apply the division and multiplication outside of the iteration steps. This seeks to minimize loss of precision due to repeated multiplication, a typical characteristic of naïve implementations

By way of explanation the variance can be converted to a standard deviation by using a square root.

WEEK 2 : TASKS

This week I will use the two examples from the book, 3.19 and 3.22 and improve each to hopefully improve the existing example.

Firstly I will identify and use an arbitrary precision library to compare the results of a slow but near optimal implementation of the Pearsons' r coefficient with the existing implementation that uses Java types and the Java.Math library.

I then will implement various changes to the variance, std deviation, calculation inside the function to see how close I can be to the RMSE calculated using the arbitrary precision library.

WEEK 2 : DIARY

In order to evaluate our results we first run the 2 Movie Lens data sets through the example code from the book. To do this we unpack the ml.zip file for the 100K data set into the C:\Devel\iWeb2\data\ch03\MovieLens\ directory where it can be found.

We then run the sample code to calculate the RMSE for this data set given this data.

To do this we start BeanShell and run the example code. The following shows our results of the RMSE on this data set.

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import *;
bsh % MovieLensDataset ds = MovieLensData.createDataset(10000);
*** Loading MovieLens dataset...
make sure that you are using at least: -Xmx1024m

*** Loaded MovieLens dataset.
users: 943
movies: 1682
ratings: 90000
test ratings: 10000
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 63(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 90000
Test ratings count: 10000
RMSE:1.0339138284716327
bsh %
```

Now copy the million ratings database from the following location, http://grouplens.org/system/files/million-ml-data.tar__0.gz. We now unpack the the larger million ratings dataset into the same directory and run the example a second time to calculate the RMSE.

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import *;
bsh % MovieLensDataset ds = MovieLensData.createDataset(100000);
*** Loading MovieLens dataset...
make sure that you are using at least: -Xmx1024m

*** Loaded MovieLens dataset.
users: 6040
movies: 3952
ratings: 900209
test ratings: 100000
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 1287(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 900209
Test ratings count: 100000
RMSE:1.0118471615930231
bsh %
```

Now we go ahead and install the Arbitrary precision library from http://www.apfloat.org/apfloat_java/download.html. To do this we copy the apfloat.jar file from the installation into the c:/devel/iWeb2/lib directory. Having done this we include a reference to this new jar within the build/build.xml file for ant. The old content reading from line 26:

```
<!-- list of all jars that are currently used in iweb2 app -->
<patternset id="thirdparty.jars">
  <include name="commons-codec-1.3.jar"/>
  <include name="commons-httpclient-3.1.jar"/>
```

And the new including the apfloat.jar library:

```
<!-- list of all jars that are currently used in iweb2 app -->
<patternset id="thirdparty.jars">
  <include name="apfloat.jar"/>
  <include name="commons-codec-1.3.jar"/>
  <include name="commons-httpclient-3.1.jar"/>
```

You will also have to modify the bsc.bat or shell script files in a similar manner. For example for windows the lines in bsh.bat reading:

```
set IWEB2_HOME=C:\Devel\iWeb2

set LIBJARS=
set LIBJARS=%LIBJARS%;%IWEB2_HOME%\deploy\lib\commons-codec-1.3.jar
```

Will be modified to read:

```
set IWEB2_HOME=C:\Devel\iWeb2

set LIBJARS=
set LIBJARS=%LIBJARS%;%IWEB2_HOME%\deploy\lib\apfloat.jar
set LIBJARS=%LIBJARS%;%IWEB2_HOME%\deploy\lib\commons-codec-1.3.jar
```

We now modify the C:\Devel\iWeb2\src\iweb2\ch3\collaborative\similarity\PearsonCorrelation.java file to use the Arbitary precision package. To do this, modify the getStdDev method to read as follows:

```
private double getStdDev(double m, double[] v)
        {
                Apfloat mean = new Apfloat(m);
                Apfloat sigma = new Apfloat(0);
               for (double xi : v ) {
                       sigma = sigma.add((new Apfloat(xi).subtract(mean)).multiply(new
Apfloat(xi).subtract(mean)));
               }
                return(ApfloatMath.sqrt(sigma.divide(new Apfloat(v.length))).doubleValue());
}
```

Now unpack the ml.zip file a second time to run the RMSE calculations for the small dataset. Run the commands in beanShell as we did previously and as shown in the following screen shot. You will that the run time has increased by a factor of ten however the RMSE have dropped significantly.

```
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 651(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
```

```
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 90000
Test ratings count: 10000
RMSE:1.0151030579260731
```

We now unpack the large dataset a second time and overwrite the existing data. Running the RMSE calculation again results in a much longer run because of the new code, however the result is as shown below:

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import *;
bsh % MovieLensDataset ds = MovieLensData.createDataset(100000);
*** Loading MovieLens dataset...
make sure that you are using at least: -Xmx1024m

*** Loaded MovieLens dataset.
users: 6040
movies: 3952
ratings: 900209
test ratings: 100000
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 12137(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 900209
Test ratings count: 100000
RMSE:1.0112224407441863
```

Our run results now appear as follows

	1 Million Ratings		100K Ratings	
	Arbitrary Precision	Java standard	Arbitrary Precision	Java Standard
RMSE	1.0112224407441863	1.0118471615930231	1.0151030579260731	1.0339138284716327
Runtime	12137	1287	651	63

The most obvious result is in the case of the smaller dataset where there was a large improvement in the error rate. Of course run times do become a real issue and so the purpose of this project is to re-implement the variance function to run in a reasonable amount of time while minimizing error rates.

To do this we obtain pseudo code implementation of the Chan, Glub, and LeVeque paper cited in the text book.

We in this case will use the following:

```
def variance(data):
    n = 0
    sum1 = 0
    for x in data:
        n = n + 1
        sum1 = sum1 + x
    mean = sum1/n

    sum2 = 0
    sumc = 0
```

```
    for x in data:
        sum2 = sum2 + (x - mean)**2
        sumc = sumc + (x - mean)
    variance = (sum2 - sumc**2/n)/(n - 1)
    return variance
```

We still need to account for potential divide by zero issues for smaller sets of ratings. Therefore we check the input parameters before doing the calculation and run the older variation of this method if appropriate.

Implemented as Java the getStdDev method, the file C:\Devel\iWeb2\src\iweb2\ch3\collaborative\similarity\PearsonCorrelation.java, now becomes:

```
    private double getStdDev(double mean, double[] v)
    {
        if (3 > v.length) {
            double sigma=0.0d;

            for (double xi : v ) {
                sigma += (xi - mean)*(xi - mean);
            }

            sigma = sigma / v.length;

            return Math.sqrt(sigma);
        }

        double sum2 = 0.0d;
        double sumc = 0.0d;
        for (double xi : v) {
            sum2 += (xi - mean) * (xi - mean);
            sumc += (xi - mean);
        }

        return(Math.sqrt((sum2 - (sumc * sumc)/v.length) / (v.length-1)));
    }
```

Using the small data set as document above the results are now:

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % MovieLensDataset ds = MovieLensData.createDataset(10000);
*** Loading MovieLens dataset...
make sure that you are using at least: -Xmx1024m

*** Loaded MovieLens dataset.
users: 943
movies: 1682
ratings: 90000
test ratings: 10000
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 55(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 90000
Test ratings count: 10000
RMSE: 1.0376963389522005
```

And repeating the test with the larger data set the results are:

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import *;
bsh % MovieLensDataset ds = MovieLensData.createDataset(100000);
```

```
*** Loading MovieLens dataset...
make sure that you are using at least: -Xmx1024m

*** Loaded MovieLens dataset.
users: 6040
movies: 3952
ratings: 900209
test ratings: 100000
bsh % MovieLensDelphi delphi = new MovieLensDelphi(ds);
Entering MovieLensDelphi(Dataset) constructor ...
Calculating item based similarities...
Item based similarities calculated in 1254(sec).
Similarities ready.
Leaving MovieLensDelpi(Dataset) constructor ...
bsh % RMSEEstimator rmseEstimator = new RMSEEstimator();
bsh % rmseEstimator.calculateRMSE(delphi);
Calculating RMSE ...
Training ratings count: 900209
Test ratings count: 100000
RMSE: 1.0064281320666533
```

Our updated table appears as follows:

	1 Million Ratings			100K Ratings		
	Arbitrary Precision	2 Pass	Java standard	Arbitrary Precision	2 Pass	Java Standard
RMSE	1.011222440744 1863	1.006428132066 6533	1.011847161593 0231	1.015103057926 0731	1.037696338952 2005	1.033913828471 6327
Runtime	12137	1254	1287	651	55	63

So we have made the discovery that at the very least our 2 pass implementation is not stable.

Now we have to figure out why. From looking at the code it can be seen that several values are not quite what we expect, namely that the Average function being used to feed data into the getStdDev method is using integers to initialize the double. This introduces a small but significant precision problem which is then propagated into the method we are modifying.

To fix this the getAverage method becomes:

```
        private double getAverage(double[] v)
        {
                double avg=0.0d;

                for (double xi : v ) {
                        avg += xi;
                }

                avg = avg/v.length;

                return avg;
        }
```

So now we rerun the test. The results now look like the following.

	1 Million Ratings			100K Ratings		
	Arbitrary Precision	2 Pass	Java standard	Arbitrary Precision	2 Pass	Java Standard
RMSE	Not yet recomputed	1.0083180956313909	1.0118471615930231	1.0151030579260731	1.0278733888068046	1.0339138284716327
Runtime	61397 Projected	1253	1287	651	56	63

Now it is apparent that precision issues inside the getAverage method have polluted the correlation and so the Arbitrary Precision calculations need redoing.

A key lesson here is that while it improved the error rate for the small data set the large data set when in the opposite direction. Errors are accumulating at different rates in different portions in our implementation based upon the size of the input dataset, ouch. If I had not test both datasets I would never have fully known the problem.

To correct this issue for the Arbitrary Precision library I had to convert the entire correlation class, PearsonCorrelation, to using the ApFloat library. After correcting these problems the results now stand at:

	100K Ratings		
	Arbitrary Precision	2 Pass	Java Standard
RMSE	1.013777818301727	1.0278733888068046	1.0339138284716327
Runtime	2739	56	63

The new ApFloat version is included as an appendix, Appendix 1, for future reference due to several non-obvious techniques being used to avoid precision issues.

```
    /** Calculate the average using a total of infinte precision and a minimum precision division
of 128 significant digits */
    private Apfloat getAverage(Apfloat[] v)
    {
        Apfloat avg = new Apfloat(0);

        for (Apfloat xi : v ) {
                avg = avg.add(xi);
        }

        avg = avg.divide(new Apfloat(v.length, 128));

        return avg;
    }

    /** @brief Calculate a standard deviation for the vector given a precomputed mean
     *
     * This implementation uses a naive implementation when only 2 or less items are provided and
     * switches to a two pass implementation when appropriate.
     *
     */
    private Apfloat getStdDev(Apfloat mean, Apfloat[] v)
    {
```

```
        if (v.length < 2) {
            Apfloat sigma = new Apfloat(0);
            for (Apfloat xi : v ) {
                sigma = sigma.add((xi.subtract(mean)).multiply(xi.subtract(mean)));
            }
            return(ApfloatMath.sqrt(sigma.divide(new Apfloat(v.length, 128))));
        }
        Apfloat sum2 = new Apfloat(0);
        Apfloat sumc = new Apfloat(0);
        for (Apfloat xi : v) {
                sum2 = sum2.add((xi.subtract(mean)).multiply(xi.subtract(mean)));
                sumc = sumc.add(xi.subtract(mean));
        }

        return(ApfloatMath.sqrt((sum2.subtract(sumc.multiply(sumc).divide(new Apfloat(v.length,
128))).divide(new Apfloat(v.length-1, 128)))));

    }
```

Another technique used worth mentioning is the use of significant digits when comparisons are performed. For example when comparing to zero (0) I used code similar to the following that performs comparisons to 16 digit precision.

```
        //No variation -- all points have the same values for either X or Y or both
        if ( 16 < sX.equalDigits(ZERO) || 16 < sY.equalDigits(ZERO) ) {

            Apfloat indX = ZERO;
            Apfloat indY = ZERO;

            for (int i=1; i < n; i++) {

                    indX = indX.add(x[0].subtract(x[i]));
                    indY = indY.add(y[0].subtract(y[i]));
            }

            if (indX == ZERO && indY == ZERO) {
                    // All points refer to the same value
                    // This is a degenerate case of correlation
                    return 1.0;
            } else {
                    //Either the values of the X vary or the values of Y
                    if (16 < sX.equalDigits(ZERO)) {
                            sX = sY;
                    } else {
                            sY = sX;
                    }
            }
        }
```

I then attempted to recompute the 1 Million Ratings Arbitrary Precision numbers but was not successful because of the compute times involved, a projected 17 hours.

	1 Million Ratings		
	Arbitrary Precision	2 Pass	Java Standard
RMSE	Not yet recomputed	1.0083180956313909	1.0118471615930231
Runtime	61397 Projected	1253	1287

My unscientific gut tells me that the 2 pass RMSE is too low even with with the corrections detailed above. However even with fixing some of the precision issues it would still appear that we need to select a more stable algorithm.

LESSONS LEARNED

The main lesson I learnt is the necessity of having someone involved in these projects that has serious computational mathematics experience. High school level math, such as calculating averages and variances, while simple in theory is complex when dealing with modern datasets principally because of their magnitude.

Problems lurking in code being used to process datasets can be very difficult to find as in our example. Although I had a hunch something was wrong in the original code with how averages were being calculated, incorrect initialization for a float, it was not until I had fixed the problem that I knew that indeed the problem was real.

As a result bugs in the computation prove very difficult to both detect and find. Even initializations should be done with care.

If the results of different phases of computation are reused in other phases they can have subtle interactions if for example when the output of one computation results in truncation and in another in rounding errors. This happened to me when I had portions of my coding using Arbitrary Precision and other code was using java double data types. This resulted in undershooting the regression with values less than 1.0. If the code had instead not undershot the regression line then I might have been much closer to the line that I deserved to be and would never have known.

When implementations of various algorithms are chosen they should evaluated as a whole for their computation integrity.

The two pass looks initially to be more numerically reliable than the naïve algorithm for large sets of data. However it could be worse if portions of the data are very close to the mean but are not exactly equal to it, and some of the data is far away from the mean.

And of course having parallelism or a threaded implementation is a huge bonus. Maybe if I had threaded the implementation at the very least I would have been able to complete the final Arbitrary Precision results of the million ratings dataset.

In short the lessons of this weeks' project are legion.

FUTURE WORK

I did find a promising article concerning a method called Welford's method. In this method a single pass is used to calculate the variance. The variance is also available as an estimate during the calculation which could be used in circumstances where only a broad guess is required, i.e. you could terminate after the estimate appears to converge. In our case this is not much of a bonus but I imagine it would be useful in some circumstances.

Here is an excerpt from an online article, http://www.johndcook.com/standard_deviation.html detailing the pseudo code.

Initialize $M_1 = x_1$ and $S_1 = 0$.

For subsequent x's, use the recurrence formulas

$M_k = M_{k-1} + (x_k - M_{k-1})/k$
$S_k = S_{k-1} + (x_k - M_{k-1})*(x_k - M_k)$.

For $2 \le k \le n$, the k^{th} estimate of the variance is $s^2 = S_k/(k - 1)$.

APPENDIX 1 : PearsonCorrelation.java

This code contains the Arbitrary Precision version of the correlation being used for this experiment.

```
package iweb2.ch3.collaborative.similarity;

import iweb2.ch3.collaborative.model.Dataset;
import iweb2.ch3.collaborative.model.Item;
import iweb2.ch3.collaborative.model.User;

import org.apfloat.Apfloat;
import org.apfloat.ApfloatMath;

/**
 * This code was modified from the code base supplied by the Algorithms of the Intelligent Web
 * book used within the Introduction to Web Algorithms class from UC Berkeley extension program
 * from the Summer of 2010.
 *
 * This code represents the original code enhanced to make use of a Arbitrary Precision library
 * found at http://www.apfloat.org/apfloat_java/download.html
 *
 * @author babis
 *
 */
public class PearsonCorrelation
{
    private static final Apfloat ZERO = new Apfloat(0);   // 0 Has automatic infinate precision because it is
an integer

    int n;                          // Stores the number of items that the statistic is being calculated for

    Apfloat[] x;                    // Stores a vector of ratings
    Apfloat[] y;                    //

    /** @brief Construct and populate an instance of this statistic for two items across the dataset of all
users */
    public PearsonCorrelation(Dataset ds, Item iA, Item iB) {

        Apfloat aAvgR = new Apfloat(iA.getAverageRating(), Apfloat.INFINITE);
        Apfloat bAvgR = new Apfloat(iB.getAverageRating(), Apfloat.INFINITE);
```

```
        Integer [] uid = Item.getSharedUserIds(iA, iB);
        n = uid.length;

        x = new Apfloat[n];
        y = new Apfloat[n];

        User u;
        double urA=0.0d;
        double urB=0.0d;

        for (int i=0; i<n; i++) {

                u = ds.getUser(uid[i]);
                urA = u.getItemRating(iA.getId()).getRating();
                urB = u.getItemRating(iB.getId()).getRating();

                x[i] = new Apfloat(urA, Apfloat.INFINITE).subtract(aAvgR);
                y[i] = new Apfloat(urB, Apfloat.INFINITE).subtract(bAvgR);
        }
    }

    /** @brief Construct and populate this statistic with two vectors of ratings */
    public PearsonCorrelation(double[] x, double[] y) throws java.lang.IllegalArgumentException
    {
        if (x.length != y.length) {
                throw new IllegalArgumentException("Arrays x and y should have the same length!");
        }

        n = x.length;

            for (int i=1; i < n; i++) {
                this.x[i] = new Apfloat(x[i], Apfloat.INFINITE);
                this.y[i] = new Apfloat(y[i], Apfloat.INFINITE);
        }
    }

    /** @brief Calculate the statistic and return the correlation with 1 being perfect. */
    public double calculate()
    {

        if( n == 0) {
            return 0.0d;
        }

        Apfloat avgX = getAverage(x);
        Apfloat avgY = getAverage(y);

        Apfloat sX = getStdDev(avgX,x);
        Apfloat sY = getStdDev(avgY,y);

        Apfloat xy = new Apfloat(0);

        for (int i=0; i < n; i++) {

                xy = xy.add(x[i].subtract(avgX).multiply((y[i].subtract(avgY))));
        }

        //No variation -- all points have the same values for either X or Y or both
        if ( 16 < sX.equalDigits(ZERO) || 16 < sY.equalDigits(ZERO) ) {

            Apfloat indX = ZERO;
            Apfloat indY = ZERO;

            for (int i=1; i < n; i++) {

                    indX = indX.add(x[0].subtract(x[i]));
                    indY = indY.add(y[0].subtract(y[i]));
            }

            if (indX == ZERO && indY == ZERO) {
                    // All points refer to the same value
                    // This is a degenerate case of correlation
                    return 1.0;
            } else {
                    //Either the values of the X vary or the values of Y
                    if (16 < sX.equalDigits(ZERO)) {
                            sX = sY;
                    } else {
                            sY = sX;
                    }
            }
```

```
        }

        Apfloat rho = xy.divide(new Apfloat(n).multiply(sX.multiply(sY)));

        return(rho.doubleValue());
    }

    /** Calculate the average using a total of infinte precision and a minimum precision division of 128
significant digits */
    private Apfloat getAverage(Apfloat[] v)
    {
        Apfloat avg = new Apfloat(0);

        for (Apfloat xi : v ) {
                avg = avg.add(xi);
        }

        avg = avg.divide(new Apfloat(v.length, 128));

        return avg;
    }

    /** @brief Calculate a standard deviation for the vector given a precomputed mean
     *
     * This implementation uses a naive implementation when only 2 or less items are provided and
     * switches to a two pass implementation when appropriate.
     *
     */
    private Apfloat getStdDev(Apfloat mean, Apfloat[] v)
    {

        if (v.length < 2) {
            Apfloat sigma = new Apfloat(0);
            for (Apfloat xi : v ) {
                sigma = sigma.add((xi.subtract(mean)).multiply(xi.subtract(mean)));
            }
            return(ApfloatMath.sqrt(sigma.divide(new Apfloat(v.length, 128))));
        }
        Apfloat sum2 = new Apfloat(0);
        Apfloat sumc = new Apfloat(0);
        for (Apfloat xi : v) {
                sum2 = sum2.add((xi.subtract(mean)).multiply(xi.subtract(mean)));
                sumc = sumc.add(xi.subtract(mean));
        }

        return(ApfloatMath.sqrt((sum2.subtract(sumc.multiply(sumc).divide(new Apfloat(v.length,
128))).divide(new Apfloat(v.length-1, 128)))));

    }

}
```