Raphael Tam

NoSQL Class Project

Objective

The objective of this project is to build a time series database with Cassandra. Data is then retrieved from the database and classified into activities with machine learning algorithms. One application for this kind of system is to support elderly people living independently and safely when support groups can monitor their activities remotely. Another application is to record one's physical activities for personal health and fitness purposes.

Why Cassandra

This project simulates a real-life production system in which real-time data is streamed into a data base. In a real life production system, data will be streaming continuously from many users. The machine learning code will continuously sample and classify this data into activities. Older data will be deleted with Cassandra's TTL feature. The system will rely on the fast write, horizontal scalability, reliability and wide row properties of Cassandra. These properties are not available in the SQL environment.

Project Scope

This project includes

- Setting up a one-node Cassandra
- Design the schema that facilitates classification of time series data into activities
- Clean and reformat the original dataset to support the schema
- Write a large block of data into the database using CQL
- Connect to Cassandra through the java driver
- With java code, query data to validate data is stored in the database as designed
- Measure the speed of reading a large block of data
- Build a training data set of 5000 examples by random sampling
- Use a machine learning algorithm in the WEKA library to build a model that classifies data into physical activities
- Remove the labels and apply the model to all the records
- Calculate the accuracy of the classifier

Data Set

The dataset, obtained from the UC Irvine website, is made up of raw, time series data[1]. Readings are recorded from 19 sensors: Heart rate, 3 groups of sensors worn at the chest, arm and ankle. Each group is made up of a temperature sensor, an orientation sensor, 1 magnetometer and three motion related sensors: 2 accelerometers and 1 gyroscope. Each accelerometer, gyroscope and magnetometer sensor produces readings in x, y and z coordinates. Each orientation sensor produces 4 readings at a time. Every sensor reading is associated with a time stamp.

In summary, at each point in time, a record is made up of 54 numbers:

1	timestamp
2	activity label
3	heart rate
4-20	sensor readings from the hand
21-37	sensor readings from the chest
38-54	sensor readings from the ankle

Readings from the hand, chest and ankle are made up of

1	temperature
2-4	3D acceleration with a wider range and higher granularity than the one below
5-7	3D acceleration with a narrower range and finer granularity
8-10	3D gyroscope data
11-13	3D magnetometer data
14-17	orientation

Due to wireless data dropping and problems with the hardware, missing data (NA) is in the dataset. All sensors except heart rate is sampled every 10 ms. Heart rate is sampled at a different frequency. Therefore data is recorded asynchronously and the records in between heart rate samples are left blank.

Readings from 9 subjects are provided in the original data set. The size of the dataset is over 1 GB. For this project, I only use data from subject 103. About 2528 seconds of data is recorded. Data for subject 103 is 100+ MB in size.

The original purpose of recording this data set was to provide a benchmark to the wearable computing community[2,3]. The data set demonstrates that given a large number of sensors worn on different parts of a person, the accuracy of classifying data into activity labels could be over 98%.

Labeled raw data from all sensors is merged into 1 data file. Each row represents data at one point in time. Each row has 54 columns.

Labels

For subject103, sensor readings for the following activities are recorded.

1	lying
2	sitting
3	standing
4	walking
12	ascending stairs
13	descending stairs
16	vacuum cleaning
17	ironing
0	recorded during transient periods – these records should not be used for classification

Data Structure

The schema is designed for fast read in order to speed activity classifications. Applications that monitor physical activities will read a block of data, say 10 records, every second to classify the activity. Therefore, fast query of a block of contiguous data records is important. Instead of getting every sensor reading individually, sensor readings are retrieved as a list. Java code is used to convert this list into a structure expected by machine learning algorithms.

The schema simulates the one used in the production system. Data from an individual with a subject ID will be stored in a row. The partition key identifies the individual. A compound key made up of the individual's ID and time stamp is used to create wide Cassandra rows. Sensor data with the same time

stamp will be stored as a column in the column family.

By consolidating all the sensor readings into a list, a record stored in Cassandra is made up of only 4 columns:
1. Subject ID
2. Time Stamp
3. Activity Label
4. List of sensor readings

The following CQL commands are used to create the keyspace and the column family:

```
create keyspace timeseries with replication={'class':'SimpleStrategy','replication_factor':1};

create table sensordata(subject_id double, time double, label int, raw list<float>, primary
key(subject_id, time));
```

Data Cleaning and Packing
The original data set needs to be cleaned for several reasons:
1. Fill in missing Heart Rate data and other NA's due to due to hardware problems during the recording session
2. Remove transient records from the dataset
3. Pack all the sensor data into a format that will be accepted as a list in Cassandra
4. Add subject ID to the original data set
5. Use a realistic time stamp. The time stamp used in the original dataset is an arbitrary index.
6. Turn the data into a CSV file so that it can be copied into Cassandra

Transient data is removed to simplify machine learning. In a real production system, transient data will be present which will lead to unavoidable classification errors.

I used the R language to clean and pack the data into a structure as described above to facilitate fast reads from Cassandra. The R code is provided in Data Cleaning R code.pdf . It produces csv files that can be copied into Cassandra.

Loading data into Cassandra
Data is copied into Cassandra in 2 installments. The first installment is used to test java code and contains only 20000 records. The second installment is made up of 154338 records. The total number of records is 174338. About 70 MB of data is stored in the sensordata column family.

Data is copied into Cassandra with this command:

```
copy sensordata from 'sub103r3.csv';
copy sensordata from 'sub103r4.csv';
```

Connect to Cassandra with java driver
Java driver uses the Cluster and Session classes for connection to Cassandra. The driver builds a cluster and creates contact points (IP addresses) where the cluster is listening for instructions. The cluster name is Test Cluster. Datacenter and rack are set up by default in the java driver. The code then

instantiates a Session object named session that maintains multiple connections to the cluster node.

```
public class client {
   private Cluster cluster;
   protected static Session session;

   public void connect(String node) {
      cluster = Cluster.builder()
            .addContactPoint(node).build();
      Metadata metadata = cluster.getMetadata();
      System.out.printf("Connected to cluster: %s\n",
            metadata.getClusterName());
      for ( Host host : metadata.getAllHosts() ) {
         System.out.printf("Datatacenter: %s; Host: %s; Rack: %s\n",
               host.getDatacenter(), host.getAddress(), host.getRack());
         System.out.println();
      }
      session=cluster.connect();
   }
// returns the session instance field
   public Session getSession(){
     return this.session;
   }

   public void close() {
      cluster.shutdown();
   }

   public static void main(String[] args) throws Exception
   {
//Connect to Cassandra server
      client client = new client();
      client.connect("127.0.0.1");
```

The console output of this code:

```
Connected to cluster: Test Cluster
Datatacenter: datacenter1; Host: /127.0.0.1; Rack: rack1
```

Set up Keyspace

timeseries was first created with CQL earlier. Java code specifies which keyspace to use with

```
session.execute("use timeseries;");
```

Preliminary Queries

Since the purpose of the system is to store time series data and classify it into physical activities, it is important to check the timestamp of the data. Instead of using an arbitrary index, a POSIXct time stamp is created for every record during the data cleaning and packing step as described above.

The code that handles time stamps is in the Timestamp class. It uses Prepared and Bound statements to execute CQL commands in java code. Results are provided through the ResultSet class. Methods within ResultSet provides the mechanism to read specific columns in the column family.

```
package edu.ucsc.nosqlproject;

import com.datastax.driver.core.BoundStatement;
import com.datastax.driver.core.PreparedStatement;
import com.datastax.driver.core.ResultSet;
```

Raphael Tam

```
import com.datastax.driver.core.Row;

public class Timestamp extends client{
		private int offset;
		private double v;
//		private String d;

		public Timestamp(int offset) //offset from start time
		{
		this.offset=offset;
		}

		public double value(){
		String qCQL="SELECT time FROM timeseries.sensordata WHERE subject_id=103"+"LIMIT ?;";
   PreparedStatement prepared=getSession().prepare(qCQL);
   offset=offset+1;
		BoundStatement bound =prepared.bind(offset);
   ResultSet results=getSession().execute(bound);
		for (Row row:results){
		v=row.getDouble("time");}
		return v;
		}
}
```

The time stamp of the first record is the start-time of the time series. The following code reads that datum from the data base and converts it into the date format. It then reads the10000th record and calculate the time difference between that and the first record. The difference should be 100 seconds since data is recored every 10 ms.

```
double subject=103;
//Get the time stamp for the first set of sensor readings
	Timestamp start_time=new Timestamp(0);
	double stime_innum=start_time.value();
	System.out.println("Start Time = "+ stime_innum+"\n");
//Print the time stamp for the first set of sensor readings in Date Format
	long l=(new Double(stime_innum*1000)).longValue();
	Date d = new Date(l);
	System.out.println("Start Time in Date Format is "+d+"\n");

//Get the time stamp after 10000 records and check if the time difference is 10000
	Timestamp end_time=new Timestamp(10000);
	double etime_innum=end_time.value();
	double timediff=etime_innum-stime_innum;
	System.out.println("Data recorded every 10 ms.  Time Difference between the 1st and the 10000th
record is "+ timediff+ "seconds");
```

Console output:

```
Start Time = 1.38670845728347E9

Start Time in Date Format is Tue Dec 10 12:47:37 PST 2013

Data recorded every 10 ms.  Time Difference between the 1st and the 10000th record is 100.0seconds
```

Set up local variables

```
	int num_records=174338;
	int num_sreadings=52;
	int num_readings4class=28;
	int n_trainrecords=5000;
```

```
 //Print set up information
      System.out.println("\nNumber of records stored in the database for subject 103 is "+num_records);
      System.out.format("%n%d sensor attributes are stored in the database",num_sreadings);
      System.out.format("%n%nNumber of records processed in this classification= %d",num_records);

      System.out.format("%n%n%d sensor attributes are used for classification",num_readings4class);
      System.out.format("%n%nNumber of records used for training= %d",n_trainrecords);
```

Console output:

```
Number of records stored in the database for subject 103 is 174338

52 sensor attributes are stored in the database

Number of records processed in this classification= 174338

28 sensor attributes are used for classification

Number of records used for training= 5000
```

Measure speed of block read

Since the data is set up for fast queries, reading a large block of data should be efficient. The following code reads the system time before and after reading a large block of data. The number of records read multiplied by the number of bytes per record (436) and divided by the time difference is the speed of block read in this Cassandra implementation. This speed varies by up to 50% from one run to the next. I think it is caused by resource contention on my MacBook Air.

```
//Measure read speed by starting a timer, read 100000 columns and stop timer
    //Speed = 100000/timer reading
      long start_timer=System.currentTimeMillis();
      Data_block large_block=new
Data_block(subject,stime_innum,num_records,num_sreadings,num_readings4class);
      large_block.db_block();
      long stop_timer=System.currentTimeMillis();
      float read_speed=100000/(stop_timer-start_timer);
      System.out.format("%n%nRead Speed = %5.0f records/second, or%5.0f MB/sec
%n",read_speed*1000,read_speed*.436);
      System.out.format("Reading a data block of %d records, clean these records and set up matrix for
classification%n",num_records);
      System.out.format(".....................................%n");
```

Console output:

```
Read Speed = 51000 records/second, or   22 MB/sec
```

Read, Clean and Reformat Data

The entire set of records is read and converted into matrix in order to facilitate machine learning. Not all the data is used. Data from the temperature, magnetometer and orientation sensors are not applicable for physical activity classification. Furthermore, any data that has a 0 label is also excluded.

Data from heart rate, the accelerometers and gyroscope is used. There are 6 accelerometers and 3 gyroscope from arm, chest and ankle, each producing readings in the x, y and z direction. Adding the heart rate data to the group makes up 28 sensor readings in the clean data set that is ready for classification.

The code that reads the big block of data, exclude extraneous data and pack it into a matrix is in the

Data_block class. The arguments of the class include:

```
Data_block big_block=new Data_block(subject, stime_innum,num_records,num_sreadings,num_readings4class);
```

subject_id: the identification of the user
stime_innum: the start time of the data block
num_records: the number of records to be read
num_sreadings: the number of sensor readings per record
num_readings4class: the number of sensor readings used for classification

The primary method that reads sensor data from the database, excludes extraneous data and sets up the matrix used for classification is sensor_readings. It uses Prepared and Bound Statements to read a block of data, calls select_sreadings to exclude irrelevant sensor readings and calls clean_set to filter records with a 0 label.

```
public float[][] sensor_readings()
        {
                float [][]sensor_array=new float[manycolumns][column_width];
                double end_time=start_time+manycolumns*0.01;
                String qCQL="SELECT * FROM timeseries.sensordata WHERE subject_id=?" + "AND time>=?"+"
AND time<?;";
            PreparedStatement prepared=getSession().prepare(qCQL);

                BoundStatement bound =prepared.bind(subject_id,start_time,end_time);
            ResultSet results=getSession().execute(bound);
//pack the data into an array
            int m=0;
            for (Row row:results){

            readings=row.getList("raw",Float.class);
            dataset[m][num_srdg4class]=(float) row.getInt("label");

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

                sensor_array[m][i]=readings.get(i);}

            m++;
            }
            which_sreadings=select_sreadings();
            for (int i=0; i<manycolumns; ++i){
                        for (int j=0; j<num_srdg4class; ++j){
                                dataset[i][j]=sensor_array[i][which_sreadings[j]];}}

                return clean_set();
        }
```

The final number of clean records to be classified is stored in Xy. The number of clean records is in big_block.n_cleanrecods.

```
 float[][] Xy=big_block.sensor_readings();
      System.out.println("\nUse only records that have non-zero data for classfication");
      System.out.format("Number of clean records for classification is %d%n
%n",big_block.n_cleanrecords);
```

Console Output:

```
Reading a data block of 174338 records, clean these records and set up matrix for classification
.................................

Use only records that have non-zero data for classfication
Number of clean records for classification is 154578
```

Compare Data with the Original Data Set

Fragments of data are printed and compared with the original data set to validate that data is stored as designed in Cassandra. It also checks whether data is packed as designed for classification later.

```
//Print a fragment of the first 10 records to check if data is ok
      System.out.println("Fragments of the first 10 records:");
      System.out.println("Heart Rate            Hand Accelerarometer          Label");
      for (int i=0; i<10; i++)
      {
      System.out.format("%10.4f %10.4f %10.4f %10.4f %10.4f %n",Xy[i][0],Xy[i][1],Xy[i][2],Xy[i]
[3],Xy[i][28]);
      }

 //Print the first 2 records and last record of the dataset completely to check dataset validity
      System.out.println("\nData for the first 2 and the last records:");
      System.out.println("Record 1"+"     "+"Record 2"+"     "+"Last Record");
      for (int i=0; i<29; i++){
          System.out.format("%10.4f %10.4f %10.4f %n",Xy[1][i],Xy[2][i],Xy[big_block.n_cleanrecords-1]
[i]);
      }
```

Console Output:

```
Fragments of the first 10 records:
Heart Rate            Hand Accelerarometer          Label
   91.0000    -1.4310     5.4059     7.7749     1.0000
   91.0000    -1.5747     5.1438     8.0430     1.0000
   91.0000    -1.8816     4.5012     8.0406     1.0000
   91.0000    -1.8438     4.5009     8.0412     1.0000
   91.0000    -1.7878     5.1869     8.4246     1.0000
   91.0000    -2.1883     6.0271     8.6851     1.0000
   91.0000    -2.3170     6.7535     9.1811     1.0000
   91.0000    -2.2869     6.5997     8.9896     1.0000
   91.0000    -2.1426     5.5645     7.9176     1.0000
   91.0000    -1.5966     1.0946     4.9774     1.0000

Data for the first 2 and the last records:
Record 1     Record 2     Last Record
   91.0000    91.0000   122.0000
   -1.5747    -1.8816    -1.1671
    5.1438     4.5012     9.6527
    8.0430     8.0406    -0.7412
   -1.4240    -1.6214    -1.1898
    5.4260     5.0651     9.9164
    7.9263     8.1085    -1.0838
   -0.1539    -0.0938    -2.6789
    0.0614     0.0950     1.4565
    0.0980     0.0656     0.5011
   -0.5355    -0.4949    -0.0881
```

```
    9.7045     9.7802     9.3997
    0.5044     0.5824     0.2038
   -0.6846    -0.6842    -0.2976
    9.6786     9.7692     9.2857
    0.8739     0.8439     0.6135
    0.1014     0.0143     0.0895
   -0.1726    -0.1545     0.1788
    0.0118     0.0038    -0.0431
    9.7936     9.8981    10.0578
    0.7054     0.8971     0.7031
   -1.0778    -1.3866    -0.9979
    9.6880     9.7177    10.0647
    0.9536     0.9686     0.7423
   -0.8179    -0.9089    -0.5316
   -0.2409    -0.3727    -0.0055
   -0.0675    -0.0603     0.0180
    0.0913     0.0950     0.7009
    1.0000     1.0000     4.0000
```

Build Training Set

The training set is built by selecting 5000 records through random sampling. Data records and their labels are assembled to train RandomForest, a supervised machine learning algorithm. The algorithm builds a model from a training data set. The model is then used to classify unlabeled sensor data into physical activities.

build_train is the method within the Data_block class to build the training set.

```
public float[][] build_train(int trainset_size)
       {

              float[][]trainset=new float[trainset_size][num_srdg4class+1];

              for (int i=0; i<trainset_size; ++i){
                     Random generator=new Random();
                     double r=generator.nextDouble();
                     int     rand_index =(int)(r*n_cleanrecords);
                     copyrow(clean_dataset, rand_index, trainset,i);}
       return trainset;
       }
```

Fragments of the first 20 records in the training set are displayed for visual inspection on the console

```
For the first 20 TRAINING records
    HR          Hand Acceleraometer             label
   76.0000    -0.0610     9.3732     2.8941     2.0000
   79.0000    -1.9232     8.8803    -7.1388     3.0000
   77.0000     0.7614     8.3994     5.0265     2.0000
   83.0000    -2.8734     8.7750     1.4300    17.0000
   86.0000     6.7339     1.1958     6.6413     1.0000
   89.0000    -5.3951     5.7307     6.6752    17.0000
   72.0000    -0.2558     5.5493     7.8690     2.0000
   87.0000    -4.6573     7.5468     2.5235    17.0000
   95.0000     0.7610     7.4793    -0.2432    16.0000
   83.0000    -9.0601     2.5767     2.9362     3.0000
   76.0000     6.9263     1.6876     6.6425     1.0000
   74.0000     6.9992     1.4970     6.6058     1.0000
  120.0000    -5.9707     1.9909    -0.4409     4.0000
  118.0000    -8.5799     3.3373     3.5566    12.0000
   73.0000     6.8826     1.9894     6.4099     1.0000
  151.0000    -9.2178     2.3306     4.9745    13.0000
```

```
   84.0000    -8.9118     2.7262     2.8224     3.0000
   74.0000     0.6439     7.5001     6.1824     2.0000
   71.0000     6.8110     1.6122     6.6026     1.0000
   75.0000    -1.5560     3.4443     8.9342     2.0000
```

Further Checking

Since activity label 12 and 13 are under-represented in the training set, the database is queried to count the number of examples that have the label 12 (descending stairs). Since the number of records is small compared to other activities, it is reasonable to expect a smaller number of samples in the training set.

```
public int check_activity(int activity)
	{
		int count_activity=0;
		for (int i=0; i<n_cleanrecords; ++i)
		{if (((int)clean_dataset[i][28])==activity){
		 ++count_activity;}}
		return count_activity;
	}
```

Console Output:

```
No. of descending stairs records= 3421
```

Set up data and build model

The Weka java library from the University of Waikato is used to classify the data. The Machine Learning class provides the methods for setting up data in a format expected by Weka classes and methods.

```
public MachineLearning (float[][]inputdata,int nrow)
```

inputdata: data to be classified
nrow: the number of examples

Weka expects data to be set up in a specific way before machine learning algorithms can learn. Details on how the data is set up are provided in the code comments. MachineLearning's set_trainInstances method sets up data for training the model.

```
/Construct training data set
      MachineLearning training= new MachineLearning (training_set,n_trainrecords);
      training.set_trainInstances(n_trainrecords);
      System.out.println("\nSetting up training set in Weka format");
```

The method rf runs the RandomForest algorithm on the training set to build a model

```
//Learn a model with Random Forest Algorithm, return model learned
      System.out.println("\nLearning from the training set with Random Forest");
      Classifier learned = training.rf();
```

RandomForest code from the weka library is invoked by calling the buildClassifier method and a learned model is returned.

```
//Run random forest algorithm
	public Classifier rf() throws Exception
	{

		model=(Classifier) new RandomForest();
```

```
                model.buildClassifier(trainset);
                return model;

        }
```

Testing and Calculating Accuracy

The set_testInstances method sets up the entire cleaned dataset for testing with the trained model.

```
//Construct testing data set.  Labels are not in the test set
      int n_testrecords=big_block.n_cleanrecords;
      MachineLearning testing = new MachineLearning (Xy,n_testrecords);
      Instances test_set=testing.set_testInstances(n_testrecords);
```

The set_testInstances method within the MachineLearning class removes all activity labels from the test set. Activity labels are predicted by the model in the cal_accuracy method. cal_accuracy calls classifyInstance in the Weka library to predict the physical activity for every record.

```
public int cal_accuracy(Classifier m, Instances Itest) throws Exception
        {
                outcome=new double[nrow];
                double epsilon=.000000001;
                int correct_class=0;
                for (int i=0;i<nrow; ++i)
                {
                Instance tobeclass=Itest.get(i);
                outcome[i]=    m.classifyInstance(tobeclass);
                if ((outcome[i]-inputdata[i][28])<epsilon){++correct_class;}
                }
                return correct_class;
        }
```

Classification accuracy is calculated by comparing the labels from the original data set with the labels predicted by the model. The accuracy obtained in this implementation is comparable to that published by the owner of the original dataset[2,3].

```
//Get accuracy and error rate
      System.out.println("\nNo. of correct predictions= "+correct_pred);
      System.out.println("\nNo. of erroneous predictions= "+(big_block.n_cleanrecords-correct_pred));
      float accuracy_rate=(float) correct_pred/big_block.n_cleanrecords;
      System.out.format("%nError Rate= %6.2f percent",(1.0-accuracy_rate)*100);
```

Console Output:

```
Classifying all clean records

No. of correct predictions= 151741

No. of erroneous predictions= 2837

Error Rate=   1.84 percent
```

In Closing

This project provides the learning experience in building a Cassandra database, in using the Cassandra java driver to build applications on top the database and in using Weka's java API to apply machine learning algorithms.

java code is provided in the jar file raphaeltam_nosqlproject.jar. Screen shots of console output is provided in eclipse_console.pdf

Please contact me at 408-203-0934 if you have any questions.

Attachments

1. raphaeltam_nosqlproject.jar
2. eclipse_console.pdf
3. Data Cleaning R Code.pdf

References

1. http://archive.ics.uci.edu/ml/datasets/PAMAP2+Physical+Activity+Monitoring
2. A. Reiss and D. Stricker. Introducing a New Benchmarked Dataset for Activity Monitoring. *The 16th IEEE International Symposium on Wearable Computers (ISWC)*, 2012.
3. A. Reiss and D. Stricker. Creating and Benchmarking a New Dataset for Physical Activity Monitoring. *The 5th Workshop on Affect and Behaviour Related Assistance (ABRA)*, 2012.