openPDC Datamining Tools Guide

Authors: Josh Patterson, Andy Hill, and Galen Riley

Summary

- Guide on general use of openPDC datamining tools
- Guide on general mechanics and techniques used (SAX, Weka)
- List of dependencies involved
- Step by step instructions on how to train and use a time series classifier

Command Line Tools

- Used generally from bash scripts
- Create Training Instances
- Create Test Instances
- Run Weka locally with training/test instances to test classfier
- Move classifier training data to HDFS

Map Reduce Tools

- MR job with flexible configuration to detect certain time series patterns based on 1NN classifier (Keogh)
- Interops with Weka classes for certain functionality (Instances, CoverTree, BallTree)

Keogh and SAX Representations

Dr. Eamonn Keogh's SAX (Symbolic Aggregate approximation) is an approximation we used for our time series data to reduce the dimensionality of our instances. SAX is a transformation of the data into a Piecewise Aggregate Approximation representation that is then symbolized into a discrete string. In this way, dimensionality is reduced and a lower bound is introduced [5]. For our time series data, the SAX representation is generated using a normal alphabet (characters A-Z). The length of the representation and size of the alphabet is determined based on the data being represented. We used functionality found in the jmotif library to build our SAX representation [6].

SAX: Dimensionality and Cardinality

A SAX representation is generated from a set of data using a specified alphabet, dimensionality, and cardinality. The alphabet is simply the set of characters used to represent an instance of the time series data. Dimensionality and cardinality are values used to determine the face of that representation. Dimensionality is size of the representation or the number of dimensions expressed by the approximation. Cardinality is the size or length of the alphabet being used. Dimensionality and cardinality will vary wildly among data sets and there are no universal values that will work well for all data sets. Additionally, the processing expense of varying values will also depend greatly on the distance function being used [1]. For example, Euclidean distance is linear with respect to dimensionality and constant regarding cardinality.

Distance Functions

The similarity between two instances of data is expressed using a distance function. If there is a sufficiently small distance between instances, we can say that they are similar, and that each instance belongs to the same classification. These tools use two methods of calculating this distance.

Euclidean Distance

Euclidean distance is a simple, but often very effective, metric to compare instances. The Euclidean distance between two points is the length of a line segment between them. For our instances, the distance is calculated as a sum of the squares of the difference of each sample in the instance, in order, and taking the square root of that sum.

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

Reference: http://en.wikipedia.org/wiki/Euclidean_distance

Modified Euclidean Distance

In “classic” Euclidean distance, squaring the difference between each sample has the effect of removing the sign, and preventing very different instances from coincidentally having a low distance. Unfortunately, this process is computationally slow on large datasets. As another solution for the same problem, we drop the squares and square roots in favor of taking the absolute value of each difference. This is effective in testing, and much faster. [Add numbers]

Standard Deviation

Distance functions based on standard deviation are uncommon, and in fact we could not find an example in our research. However, we have found this approach to be very successful. Simply, the distance between two instances is the difference between the standard deviations of samples in each instance.

Though it may not be useful in all domains, we can effectively classify notable events in synchrophasor data using this approach. We observe that interesting events in our data correspond to cases where the signal has high fluctuation in a relatively short timespan.

Others

We have also considered the following distance functions.

- Levenshtein Distances
- DTW
- LCSS

Distance Functions and SAX

Because SAX doesn't represent samples of data with a number, it is impossible to do the necessary arithmetic to calculate the distance between two samples. For SAX instances, each symbol is converted to a number before the distance function logic is performed.

Why Use a 1NN Classifier For Time Series Classification?

In the paper “Fast time series classification using numerosity reduction” [1], it is stated that a 1NN classifier is, while relatively simple, quite effective at time series classification. The first paragraph in section 2.2 states:

> *“**A central claim of this work is that 1NN-DTW is an exceptionally competitive classifier**, in spite of a massive research effort on time series classification problems. We arrived at this conclusion after an extensive literature search, which we highlight below.”*

From an implementation standpoint we felt this was a good place to start and provided our team a “baseline” approach to time series classification to work from. It also fit well with our desired goal of an implementation that works well with the Map Reduce [2] programming model on Hadoop.

How To Train a 1NN Classifier

Training of a 1NN classifier for use with our data mining tools is relatively simple, yet involves some hand labor.

In general:

1. Find a data sample that contains an acceptable number of classes of each instance type
2. Prepare and review the data to ensure quality and consistency
3. Create a CSV file containing the samples with the last element containing a class identifier
4. Test the classifier by using the CSV file from the command line tools included
5. For use with Map Reduce and Hadoop,
    a. Move the CSV instances to HDFS
    b. Run the Map Reduce job with a proper configuration and reference the desired training instance set

In the case of working directly with timeseries data in the openPDC historian archival format:

1. Use the included command line data mining tools to generate N training samples in CSV format
2. Examine the training data in a tool such as Excel;
    a. Hand classify the instances into domain specific set of classes
    b. This creates a CSV set of training and test instances
3. Test the instances locally by using the dm-tools script to see how well the 1NN classifier performs on the test data.
4. Move the training instance set CSV file to HDFS
5. Run the Hadoop Map Reduce job with the correct parameters. Example:

```
bin/hadoop
jar Classifier_1NN.jar TVA.Hadoop.MapReduce.Datamining.SAX.SlidingClassifier_1NN_EUC
-r 6
-pointTypeID 1603
-windowSize 10000
-windowStepSize 5000
-trainingSetCSV /user/input/my_training_data.csv
-samplesPerRawInstance 300
-saxDim 20
-saxCard 10
'/user/openPDC_HistorianArchiveToScan.d'
/user/output/
```

How does the Map Reduce 1NN classifier work?

In Map Reduce there are two main phases involved in every job: the map phase and the reduce phase. The overall process looks like:

1. Execute Map Reduce job from command line
2. Map phase reads all archive blocks and outputs on the proper points which match the –pointTypeID parameter, separating out data into buckets for parallelized scanning
3. Points for the section of time and point ID are temporally ordered (grouped / secondary sorted on composite key)
4. A sliding window defines the region of time/data that we example at each step
5. We slide through all the data in the "bucket" until we exhaust the data in the bucket
6. At each step, we decompose the window into a SAX representation instance of the data
7. We find the Nearest Neighbor of this instance, and classify the the instance as the same class as its nearest neighbor

We typically use the map phase to read the proper data from the archive splits on disks and "project" the time series data into segment or "buckets" in order for it to be ordered, decomposed into SAX, and classified. The classification phase involves a sliding window which defines the region of time we are currently classifying (-windowSize). The slide step (-windowStepSize) is the number of milliseconds we "slide" the window forward in time to examine the next region. The parameters used here are highly dependent on what window size was used to create the training instances in that they should match. It should also be noted that the mechanics of this approach can vary to a large degree based on which distance function is used. In our work we have mainly used a Euclidean distance function although many other options exist in this domain.

Use of WEKA Classes

In order to test certain functionality locally (as opposed to in Map Reduce), we used the Weka [3] codebase in order to provide certain functionality:

- The Instance class in order to represent our SAX based time series instances
- The BallTree and CoverTree classes in order to serve as our kNN / 1NN classifiers

Testing Your 1NN Classifier Locally

Cross-Validation

-testClassifierLocally -train <data file> -dim <sax dimensions> -card <sax cardinality> -folds <cross-validation folds>

This command will perform n-fold stratified cross validation on a SAX representation of the data created from the specified dimensionality and cardinality.

Specified Training and Test sets

-testClassifierLocally -train <training data file> -test <test data file> -dim <sax dimensions> -card <sax cardinality>

This command will attempt to classify instances generated from the test data file based on instances generated from the training data file. Both data sets will be converted to a SAX representation using the same SAX parameters defined by the user. Both data files should be labeled in order to compute accuracy.

Classify Single Instance

-testClassifierLocally -train <training data file> -inst <comma separated features> -dim <sax dimensions> -card <sax cardinality>

This command will attempt to classify a single unlabeled instance based on instances generated from the training data file. Both the training instances and the test instance will be converted to a SAX representation using the same SAX parameters defined by the user.

Initial Performance

Canonical Datasets

Testing against the UCI Machine Learning Dataset "Waveform Database Generator (Version 1) Data Set" [4], our classifier achieved performance which was typical for the dataset. Each instance in the dataset contained 20 samples along with a class identifier where there were 3 different classes of waveforms. The dataset states that a CART decision tree algorithm achieves a rating of 72% accuracy where the typical Nearest Neighbor Algorithm achieves an accuracy of 78%. Our results were around 78% as well for our 1NN classifier using a Euclidean distance function. We also used 300 training and 5000 test instances as well as 10 fold cross validation.

Live PMU Frequency Data

In early tests with the 1NN classifier used with a sliding window in Hadoop's Map Reduce was able to recognize an "unbounded oscillation" in PMU frequency time series data. At the time of composition of this document, enough data had not been collected for a true measurement with 10 fold cross validation.

Future Development

Things we'd like to add in the future include:

- Support for the Weka ARFF format
- More distance functions, such as DTW and Levenshtein
- Locality Sensitive Hashing
- More integration with our other project, Sparky, the time series indexing and search engine

References

[1] Xiaopeng Xi , Eamonn Keogh , Christian Shelton , Li Wei , Chotirat Ann Ratanamahatana, Fast time series classification using numerosity reduction, Proceedings of the 23rd international conference on Machine learning, p.1033-1040, June 25-29, 2006, Pittsburgh, Pennsylvania

[2] Jeffrey Dean , Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, p.10-10, December 06-08, 2004, San Francisco, CA

[3] Weka Homepage: http://www.cs.waikato.ac.nz/ml/weka/

[4] UCI Datasets: http://archive.ics.uci.edu/ml/datasets/Waveform+Database+Generator+(Version+1)

[5] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms.

[6] JMotif: http://code.google.com/p/jmotif/