Stella: An Environment for Experimental Machine Learning

Antonio Kantek
CVN Student

1. Introduction

1. Machine Learning

Machine Learning covers a variety of topics. In this work, Machine Learning (or simply ML) stands for data analysis using computational statistics. Data analysis can be understood as the process of extracting a set of patterns (knowledge) from raw data (information). One can understand extracting information from raw data as running a SQL query against a relational database (task related to low level operational support purpose – e.g. ***select all customer where birthday is 10/31/75***). Extracting a set of patterns from a dataset means to create ML models that discover rules and associations hidden in datasets (e.g. ***If customer is male and age < 15 then buy videogame xyz with confidence = 80% and support = 75%***). Implementing ML models is a both iterative and interactive (semi-automatic) process.

There are three main types of ML models: i) ***classifiers*** (both nominal and numeric attribute predictors), ii) ***clustering*** and iii) ***association rules learner***. They all share the same common input: a dataset composed by instances. Each instance (or row) is composed by an array of numbers, strings or dates. Classifier (also know as Classification Learning) is a ML model that predicts what will happen in new (unseen before) data. As an example, consider a medical diagnosis, where a classifier will predict whether or not a patient has a given disease. The outcome (class to predict) can be a nominal one (like buy or not buy) or a numeric quantity. Bayesian Networks [1] and C4.5 [2] are both famous types of classifiers. Clustering is the second type of ML. It is similar to Classification Learning, but the attribute to predict will be defined by the model itself (rather than the user). By doing that, the model will be able to group similar classes (the class represents the relationship between predictor attributes and the goal attribute values). Some algorithms for clustering (like K-Means [3]) use the concept of geometric distance between instances in order to group the closest ones. The last type of ML model is Association Rule, which is related to structural data description instead of class prediction. Rules are commonly represented as **if then** rules. A Decision Tree is a data structure composed by the join of several **if then** rules. In a Decision Tree, each internal node contains a rule for a predictor attribute (e.g. Attribute: customer age < 15 yes/no) and each leaf node represents the instance classification (e.g. Decision: customer buy yes/no). Apriori [4] is a popular algorithm for association rule extraction.

2. Motivation

The process of discovering patterns in data is a semiautomatic (empirical) process. There is no universally best algorithm across all datasets (datasets are different according to their attribute types, some have more numerical attributes while others have more nominal attributes). Some algorithms have better performance in one type of dataset and a disastrous one in another type. They are biased according to the type of data to process. Stella is a dynamic language for ML model implementation. A dynamic language is the best tool for experimental computing. They provide you with a simple way to load and unload data structures (e.g. Easily dynamic class loading). Models are easily implemented and tested. You should be able to run a piece of code as easily as running a SQL script.

The two main common approaches for data analysis using ML are: specialized query languages and frameworks written in general purposes languages like C/C++ and Java. Commercial databases products like Microsoft SQL Server provide some sort of data mining query language [5]. This is a very limited solution, since the user can not build his own models. OR-Objects [6], Oracle Java Data Mining (OJDM) [7] and Weka (Waikato Environment for Knowledge Analysis) [8] are examples of the second approach (frameworks for ML). Weka is a superb, well documented generic framework for ML and it is written in Java. Java is not static as C++ but still is not dynamic enough. It is not possible to load and unload classes in Java without dealing with ClassLoaders issues.

Stella is a Domain Specific Language for ML model implementation (and testing). It is an Object Oriented Language (but not Object Oriented Obsessed like Smalltalk). Stella's API is composed by two parts: small generic API (e.g. Generic types like Integer, Double, String, Date and Object) and a extensive ML API (e.g. Classes like DataSet, Classifier, ClassifierEvaluation, Instance, and so on). Besides that, the language offers declarative constructions for some tedious tasks, and of course, a good array manipulation support.

2. Language Reference

1. Constants, Enumerations and Functions

Constants contain immutable values. Functions in Stella is defined in the same way as in (non-OOP) procedural languages like Pascal or C. Constants and functions are the easiest way to declare and implement mathematical functions. Some common functions will be natively implemented in Java. I/O is also done by functions. An enumeration, like in C, defines a sequence of elements.

Examples of constants enumerations and functions:

```
enum AttributeType { NOMINAL, NUMERIC, DATE };

constant double SMALL := 1.e-6
constant double NORMAL_DISTRIBUTION := sqrt(2 * PI);

function void out(Object obj); //Console output function

function Object fout(Object obj, String file); //File output function

function double min(double[] doubles) {
  check notNull(doubles, "doubles is null");
  check notEmpty(doubles, "doubles is empty");

  double min; //Default value for numbers is NaN
  foreach(doubles[i]) {
    if (min == NaN || min > doubles[i]) {
        min := doubles[i];
    }
  }
  ^ min;
}
```

2. Classes and Objects

The main API is composed by a few classes and some special constructions in order to deal with numbers and arrays. All classes inherent from class Object (you do not need to specify that). Object is a virtual class (the only one) and no one can directly create an instance of Object. The main API does not have a direct support for meta class, introspection and reflection. All methods starting by # are class methods (static method in Java). Instances of string, date and number are immutable objects.

Overview of the main classes (some methods are missing):

```
class Object {
      Object deepCopy();
      Object shalowCopy();
      boolean equals(Object obj);
      String toString();
}

class String {
      int length();
      String concat(String aString);
}

class Boolean {
      #boolean parse(String aString);
}

class Number {

}

class Double subclass: Number {
      #double parseAsDouble(String aString);
}

class Integer subclass: Number {
      #int parseAsInt(String aString);
}



class Long subclass: Number {
      #long parseAsLong(String aString);
}

class Date {
      booelan before(Date aDate);
      boolean after(Date aDate);
      #Date parse(String aString);
}

class Array {
      volatile int length();
}
```

```
class NumberArray subclass: Array {
      volatile int minIndex;
      volatile int maxIndex;
      volatile double stddev; //Standard Deviation
      volatile double avg; //Average
      volatile double sum; //Sum of all numbers.
}

class Matrix {
      Matrix transpose();
      Matrix add(Matrix aMatrix);
      Matrix mul(double d);
}
```

Objects are created using the new operator, and destroyed by delete operator like:

```
String str := String new;
delete str;
```

In Stella there are no primitive types. Every number is an object. Methods can be invoked directly from the number. e.g. The following statement is a valid one:

```
String number :=  3.asString();
```

There are alias and special constructions for the following basic types: String, Date, Integer (`int`), Double (`double`) and Long (`long`) (again, they are not primitive types, just aliases for their respective classes).

The following statements are equivalent:

`Integer i := 3;` and `int i := 3;`
`Boolean bool := true` and `boolean bool := true;`
`Double d := 3d` and `double d := 3d;`
`Long l := 23l` and `long l := 23l`
`String str := String new` and `String str := ""`
`Date date := Date.parse("12/22/2008")` and `Date date := {"mm/DD/yyyy"}`

Example of a user defined class:

```
class Person {
      String name;
      int age;
}

//Person usage...
Person p := Person new;
p.name := "John";
p.age := 25;

delete p;

drop class Person; //class Person is destroyed forever
```

3. Arrays

Basically there are two types of arrays: generic and numeric array. Elements can be inserted on the fly (an array is not static as in Java). All array fields are volatile. They are automatically incremented and decremented according to elements that are inserted and removed (this behavior is declared by the `volatile` keyword). An array is specified by open and close brackets after the field type (like double[] array). Matrix is represented by a basic class and it is composed by arrays of arrays. The language offers some special constructions for matrices manipulation.

Examples of array manipulation:

```
double[] doubles := {1d, 2d, 3d};
double avg := doubles.avg;
doubles[4] := 4d; //inserting a new element on the fly

double[][] matrix := {{1d,0d},{0d,1d}};

Matrix matrix := {{1d,0d},{0d,1d}}; //Special matrix construction
```

4. Rules

Rules are implemented as boolean floating statements. They are useful for assertion. They can be modified in runtime.

```
rule assertNotNull(Object obj, String msg) {
when:
      obj == null;
then:
      error msg;
}
```

In Stella there is no exception support. Errors are declared by the `error` keyword. One can think of an error as a runtime exception in Java.

5. Machine Learning API

The input of any ML model is a data set. One way to store a data set is using CSV (comma separated values) files. ARFF [9] (Attribute-Relation File Format) is a CSV file with a header describing each attribute (attribute type and possible values). An instance has values and one weight. The weight means how representative is the instance. An instance with missing values is less representative than a full one.

```
enum AttributeType { NOMINAL, NUMERIC, DATE }

class Attribute {
  String name;

  AttributeType type;
}
```

```
class Instance {
  double[] values;
  int weight;
}

class DataSet {
  String name;
  Date creationDate;
  #DataSet load(String file); //native
  Instance[] instances;
}
```

Example of ARRF file:

```
@relation weather.symbolic

@attribute outlook {sunny, overcast, rainy}
@attribute temperature {hot, mild, cool}
@attribute humidity {high, normal}
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,hot,high,FALSE,no
sunny,hot,high,TRUE,no
overcast,hot,high,FALSE,yes
```

Stella provides a SQL like language for dataset querying:

```
DataSet trainSet := DataSet.parse("aDataSet.arff");
Instance[] resultSet := trainSet where play = “yes” and windy = “TRUE”;
```

For now, Stella supports only the following simple filters: equals, different, greater than (and greater equals), lesser than (and lesser equals) and like (for strings only). Filters can be composed using the conjunction and disjunction operators.

Each model has its own base class as following:

```
/* base class for all classifiers */
class Classifier {
      String name;
      Attribute[] attributes;
      Attribute classAttribute;

      double classify(Instance instance);
}

/* base class for all clusters */
class Clusterer {

      int predictCluster(Instance instance);
}

/* base class for all association rules */
class Associatior {

      //AssociationRule is an object used to represent if then rules
      AssociationRule[] extractRules(DataSet dataSet);
}


/*Example of classifier based on Bayes's rule
  This classifier works computing and combining probabilities for all
   attributes.
  This is an incomplete code.
*/
class NaiveBayes subclass: Classifier {

      double[] counts; // All the counts for nominal attributes.

      Attribute[] attributes;

      NominalAttribute classAttribute; //Attribute to predict

      double[] distribution(Instance instance) {
        String[] categories := classAttribute.categories; //e.g. Buy Y/N
        double[] probs := double[categories.length] new;

        foreach(categories[i]) {
          probs[i] := 1;
          foreach(attributes[j]) {
            if (attribute[j].equals(classAttribute)) {
              continue;
            }

            if (instance.values[i] != NaN) {
              if (attributes[j].isNominal()) {
                probs[i] *=
                  self.counts[j][classAttribute.index][instance.values[j];
               } else {
                  error “This model doesn't support numeric attribytes yet”;
               }
              }
            }
         }
         ^ probs;
      }
}
```

The easiest way to evaluate a classifier by using a train and test set. The train set is the one you use to create your classifier. The test set is used to measure how good your model is in an previously unseen dataset. The following code shows how to execute this evaluation procedure:

```
DataSet trainingSet := DataSet.load("/home/trainset.arff");
DataSet testSet := DataSet.load("/home/testset.arff");

NaiveBayes naiveBayes := NaiveBayes.create( trainningSet.attributes );
ClassifierEvaluation eval := evaluate naiveBayes trainingSet testSet;
```

If you have limited data size, then you should use cross fold validation [10]. In cross fold validation the train set is divided in n-folds (subsets). We use n-1 folds in order to create (train) the model. The remaining folder (not used for train) is used for test. This process is repeated n times (each subset is tested). Here we have an opportunity for parallelism. The data will be stratified in order to make sure that each attribute class value is represented in the right proportion in each subset. Users define the number of folders. In a general purpose language like Java, performing a cross fold validation demands a great number of methods invocations. This can be simplified using special construction such as:

```
DataSet trainingSet := DataSet.load("/home/dataset.arff");
NaiveBayes naiveBayes := NaiveBayes.create( trainningSet.attributes );
ClassifierEvaluation eval := crossvalidation 10 naiveBayes trainingSet;
```

3. Conclusion

Stella is a dynamic (interpreted), strong typed, not fully Object Oriented language. Stella Virtual Machine (VM) will run on top of a Java VM. No ML algorithm run in complexity time less than Ω(MN) in which M is the number of attributes (columns) and N is the number of records. The Stella Virtual Machine has lots of opportunities to explore parallelism (e.g. Threads running in a multi-core machine) processing and reduce complexity time to Ω(MN/p) in which p is the number of processor. Usage of threads and concurrent algorithms should be transparent to the user.

The language offers primitive constructions like enumerations, constants and functions. These three primitives are more suitable for mathematical functions. The Object model is simple. Stella supports single inheritance only (this decision is not permanent). There is no scope delimitation (export list and design by contract from Eiffel is something under consideration). The language offers specialized ML API and language constructions for ML.

References:

[1] http://en.wikipedia.org/wiki/Bayesian_network
[2] http://en.wikipedia.org/wiki/C4.5_algorithm
[3] http://en.wikipedia.org/wiki/K-means_algorithm
[4] http://en.wikipedia.org/wiki/Apriori_algorithm
[5] http://www.sqlserverdatamining.com/ssdm/
[6] http://opsresearch.com/OR-Objects/
[7] http://www.oracle.com/technology/products/bi/odm/9idm4jv2.html
[8] http://www.cs.waikato.ac.nz/ml/weka/
[9] http://www.cs.waikato.ac.nz/~ml/weka/arff.html
[10] http://en.wikipedia.org/wiki/Cross_validation