References


57


Appendix A: Attribute-Relation File Format (ARFF)

ARFF files have two sections. The first section is the Header information and the next section is the Data information.

The Header of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types. Header section for IRIS data set will be as follows.

```
@RELATION Iris

@ATTRIBUTE sepal_length NUMERIC
@ATTRIBUTE sepal_width NUMERIC
@ATTRIBUTE petal_length NUMERIC
@ATTRIBUTE petal_width NUMERIC
@ATTRIBUTE class {Iris-setosa, Iris-versicolor, Iris-virginica}
```

Data section will be as follows.

```
@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.5,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

Comments begin with %. The @RELATION, @ATTRIBUTE and @DATA declarations are case insensitive.

The format for the @attribute declaratio is

```
@attribute <attribute-name> <datatype>
```
Data types supported by Weka

- numeric
- integer is treated as numeric
- real is treated as numeric
- nominal
- string
- date [<date-format>]
- relational for multi-instance data

The keywords numeric, real, integer, string and date are case insensitive.

**Numeric attributes**

Numeric attributes can be real or integer numbers.

**Nominal attributes**

Nominal values are defined by providing an <nominal-specification> listing the possible values: {<nominal-name1>, <nominal-name2>, <nominal-name3>, ...}

**String attributes**

String attributes allow us to create attributes containing arbitrary textual values.

**Date attributes**

Date attribute declarations take the form:
```
@attribute <name> date [<date-format>]
```

where <name> is the name for the attribute and <date-format> is an optional string specifying how date values should be parsed and printed (this is the same format used by SimpleDateFormat). The default format string accepts the ISO-8601 combined date and time format: yyyy-MM-dd'T'HH:mm:ss.

**Relational attributes**

Relational attribute declarations take the form:
```
@attribute <name> relational
    <further attribute definitions>
@end <name>
```
Appendix B: Introduction to JGAP Framework

JGAP (pronounced "jay-gap") is a Genetic Algorithms and Genetic Programming component provided as a Java framework. It provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions.

JGAP has been available as an open source genetic algorithm package since year 2000. The programming language used to develop JGAP is Java and the current version available is 3.3.

Steps to use JGAP are as follows.

1. Plan the Chromosome
2. Implement the fitness function
3. Setup Configuration object
4. Create a population of appropriate size.
5. Evolve the population

B.1 Plan the Chromosome
At the heart of the genetic algorithm is the Chromosome. The Chromosome represents a potential solution and is divided into multiple genes. Genes in JGAP represent distinct aspects of the solution as a whole. The first step is to decide on the makeup of the chromosomes, which includes how many genes needed what those genes will represent.

For example consider the case of having to find optimum parameters weight vector and value for K for data set with 5 attributes. The chromosome representation would be as follows.

| W1 | W2 | W3 | W4 | W5 | K |

B.2 Implementing a Fitness Function
JGAP is designed to do almost all of the evolutionary work for an application in a relatively generic fashion. However, it has no knowledge of the specific problem
actually trying to solve, and hence has no intrinsic way of deciding if one potential solution is any better than another potential solution for the specific problem. That's where the fitness function comes in: it's a single method that the user must implement that accepts a potential problem solution and returns an integer value that indicates how good (or "fit") that solution is relative to other possible solutions. The higher the number, the better the solution. The lower the number (1 being the lowest legal fitness value), the poorer the solution. JGAP will use these fitness measurements to evolve the population of solutions toward a more optimal set of solutions.

B.3 Setup Configuration Object
JGAP uses a Configuration object that must be setup with all of the settings the user wants prior to using the genetic engine. There is a DefaultConfiguration class that comes already setup with the most common settings. User will need to provide three extra pieces of information: what fitness function to use, how to setup Chromosomes, and how many Chromosomes needed population.

A Chromosome is made up of genes. JGAP lets the user choose what Gene class to use to represent each gene in the Chromosome. Also user has the flexibility to use his own implementation of gene class. The gene classes provide by JGAP are BooleanGene, IntegerGene, DoubleGene, StringGene and MapGene.

B.4 Create a Population
A population of Chromosomes is called a Genotype, and that is the class needs to construct to create the population. If the user wants, can construct each Chromosome individually and then pass them all into a new Genotype. Also JGAP provides a much quicker and easier way of creating a random population with the method randomInitialGenotype().

B.5 Evolve Population
The final step is to evolve the population until it contains a satisfactory potential solution. If an acceptable solution is not obtained population can be evolved again.
Appendix C: Customizing gaKnn

To use gaKnn to optimize KNN the user has to extend following classes.

1. Evaluator
2. Predictor
3. AbstractSimilarity
4. FitnessFunction

How these classes are extended and used for Abalone data set is given below. For more details refer to gaKnn.sourceforge.net.

C.1 Evaluator

Main function of the Evaluator class is to define the evaluation criteria for the gaKnn optimization. Sample code for SimpleWeightEvaluator class is given below.

```java
public class SimpleWeightEvaluator extends Evaluator {
    public SimpleWeightEvaluator(Predictor predictor, Instance[] testSet) {
        super(predictor, testSet);
    }

    public double evaluate() {
        double fitness = 0.0;
        int testSetSize = testSet.length;
        double fit = 0.0;
        for (int i = 0; i < testSetSize; i++) {
            fit += predictor.predict(testSet[i]);
            fitness += (1 - fit) * (1 - fit);
        }

        if (fitness > Double.MAX_VALUE)
            fitness = Double.MAX_VALUE;
        else if (fitness < Double.MIN_VALUE)
            fitness = Double.MIN_VALUE;
        return Math.sqrt(fitness);
    }
}
```

C.2 Predictor

The main function of the Predictor class is to define how to carry out the class prediction of the generated solutions. Sample implementation of the Predictor class with the main method Predict is given below.

```java
public class SimplePredictor extends Predictor {
    public double predict(Instance instance) {
        return Math.sqrt(fitness);
    }
}
```
public class Predictor1 extends Predictor {
    public Predictor1(AbstractSimilarity sim, Instance[] trSet)
    {
        super(sim, trSet);
    }

    public double Predict(Instance instance)
    {
        int dataLength = trainSet.length;
        Pair[] simList = new Pair[dataLength];
        for (int i = 0; i < dataLength; i++)
        {
            double simValue =
                similarityMeasure.GetSimilarity(instance.GetElements(),
                trainSet[i].GetElements());
            simList[i] = new Pair(i, simValue);
        }
        Arrays.sort(simList);
        double[] vote = new double[m_ClassList.length];
        int ClassIndex = 0;
        for (int i = 0; i < m_K; i++)
        {
            int index = ((Pair) simList[i]).index;
            ClassIndex = trainSet[index].GetClassIndex();
            vote[ClassIndex] += ((Pair) simList[i]).value;
        }
        int clsid = (int) instance.GetClassIndex();
        double val = CalculateClassConf(vote, clsid);
        if (val < Double.MIN_VALUE)
            val = 0.0;
        else if (val > Double.MAX_VALUE)
            val = Double.MAX_VALUE;
        else if (!Double.isNaN(val))
            val = 0.0;
        return val;
    }
}

C.3 AbstractSimilarity

AbstractSimilarity is the extendable class to determine similarity between two instances.
```java
public class BasicSimilarity extends AbstractSimilarity{

    public BasicSimilarity(FastVector attributes) {
        super(attributes);
    }

    public double getSimilarity(double[] attributeSet1, double[] attributeSet2) {
        double simValue = 0.0;
        double diff = 0.0;
        int weightPos = 0;
        int numAttributes = attributeSet1.length;
        for (int i = 0; i < numAttributes; i++) {
            Attribute attribute = (Attribute) m_Attributes.elementAt(i);
            switch (attribute.Type()) {
                case Attribute.NOMINAL:
                    if (attributeSet1[i] == attributeSet2[i])
                        diff = 0.0;
                    else
                        diff = m_Weights[weightPos];
                    break;
                default:
                    diff = m_Weights[weightPos] * Math.abs(attributeSet1[i] - attributeSet2[i]);
            }
            weightPos++;
            simValue += diff;
        }

        if (simValue < Double.MIN_VALUE)
            simValue = Double.MIN_VALUE;
        else
            simValue = 1.0 / simValue;

        if (!Double.isInfinite(simValue))
            simValue = Double.MAX_VALUE;

        return simValue;
    }
}
```

### C.4 FitnessFunction

FitnessFunction is the main extendable class from the JGAP framework. This defines how the fitness is measured. A sample implementation is given below.
public class EvaluatePredictions extends FitnessFunction
{
    int m_ChromosomeLength;
    AbstractSimilarity m_Similarity;
    Predictor m_Predictor;
    Evaluator m_Evaluator;
    int m_Fit;
    boolean m_FindK = true;

    public EvaluatePredictions(abstract Similarity sim, Predictor pred, Evaluator eval)
    {
        super();
        m_Similarity = sim;
        m_Predictor = pred;
        m_Evaluator = eval;
    }

    public double evaluate(Chromosome a_chrom) {
        int weightVecSize;
        double fitness = 0.0;
        
        if (m_FindK)
        {
            weightVecSize = a_chrom.size() - 1;
            m_F = (Integer) a_chrom.getGene(weightVecSize).getAllele();
        }
        else
        {
            weightVecSize = a_chrom.size();
        }

        double[] weights = new double[weightVecSize];
        for (int i = 0; i < weightVecSize; i++)
        {
            weights[i] = (Double) a_chrom.getGene(i).getAllele();
        }

        m_Predictor.setF(i);
        m_Similarity.setWeights(weights);
        m_Predictor.setSimilarity(m_Similarity);
        m_Evaluator.setPredictor(m_Predictor);
        try
        {
            fitness = m_Evaluator.Evaluate();
        }
        catch (Exception e)
        {
            System.out.println("\"+e.toString()\";)
        }
    }
}