1 WEKA data format

An ARFF (Attribute-Relation File Format) file is an ASCII text file that describes a list of instances sharing a set of attributes. ARFF files have two distinct sections. The first section is the *Header* information, which is followed the *Data* information.

- The *header* part - contains the name of the relation, a list of the attributes (the columns in the data), and their types.
  - You can put comments at the top of the file by making the first character a "%".
  - A dataset has to start with a declaration of its name:

    ```
    @relation name
    ```

    followed by a list of all the attributes in the dataset, including the class attribute.
  - Attribute declarations have the form

    ```
    @attribute attribute_name specification
    ```

  - Nominal attributes are specified as a list of the possible attribute values in curly brackets:

    ```
    @attribute nominal_attribute {first_value,
    second_value, third_value}
    ```

  - Numeric attributes are specified using one of the following forms:

    ```
    @attribute numeric_attribute numeric
    @attribute integer_attribute integer
    @attribute real_attribute real
    ```

  - String declarations have the form

    ```
    @attribute string_name string
    ```

    but beware that these attributes will be have to be filtered into one of the above forms. Thus this type is most useful as a shortcut to a nominal, or as a comment or ID field which is not otherwise used.
- The *body* part
  - After the attribute declarations, the actual data is introduced by a

    ```
    @data
    ```

tag, which is followed by a list of all the instances. The instances are listed in comma-separated format, with a question mark representing a missing value.

- An example header on the standard *IRIS* dataset looks like this:

```
% Iris Plants Database
@RELATION iris

@ATTRIBUTE sepallength  NUMERIC
@ATTRIBUTE sepalwidth   NUMERIC
@ATTRIBUTE petallength  NUMERIC
@ATTRIBUTE petalwidth   NUMERIC
@ATTRIBUTE class        {Iris-setosa,
  Iris-versicolor,Iris-virginica}

@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
5.4,3.9,1.7,0.4,Iris-setosa
...
```

- An example header on the standard *TITANIC* dataset looks like this:

```
% Titanic Database
@relation titanic

@attribute C {crew,first,second,third}
@attribute age {adu,enf}
@attribute sex {f,m}
@attribute Class {mort,surv}

@data

crew,adu,m,surv
crew,adu,m,surv
crew,adu,f,surv
third,adu,m,mort
...
```

2 Important classes in WEKA

2.1 weka.core.Attribute

- The *Attribute* class is for handling an attribute.

  Three attribute types are supported:

  - *numeric* - this type of attribute represents a floating-point number (e.g. 1, 1.534 ...).
  - *nominal* - this type of attribute represents a fixed set of nominal values (e.g. High, Medium, Low)
  - *string* - this type of attribute represents a dynamically expanding set of nominal values (e.g. "John Smith", "Bill Gates").

- An example of definition of an *Attribute* object:

```
...
// Create numeric attributes
// ``length'' and ``weight''
Attribute length = new Attribute(``length'');
Attribute weight = new Attribute(``weight'');

// Create vector to hold nominal values
// ``first'', ``second'', ``third''
// FastVector implements a
// fast vector without synchronized methods.
// Replaces java.util.Vector
FastVector my_nominal_values = new FastVector(3);

my_nominal_values.addElement(``first'');
my_nominal_values.addElement(``second'');
my_nominal_values.addElement(``third'');

// Create nominal attribute ``position''
Attribute position = new Attribute(``position'',
my_nominal_values);
...
```

- Some important methods:

  - *enumerateValues()* - returns an enumeration of all sttribute's values if the attribute is nominal or a string; null otherwise
  - *index()* - returns index of this attribute
  - *name()* - returns the name of an attribute
  - *isNominal() , isNumeric() , isString()*

- *numValues()* - number of attribute values; 0 for numeric attributes
- *value(int index)* - for string & nominal attribute returns a values at a given position; for other attributes - empty string
- *type()* - type of an attribute (as int)
- *indexOfValue(String value)* - index of a given attribute value; -1 if it is a numerical attribute or the value can not be found
- *addStringValue(String value)* - adds a string value to the list of valid strings for attributes of type STRING

2.2 weka.core.Instance

- This is the class for handling an instance. All values (numeric, nominal or string) are internally stored as floating-point numbers. If an attribute is nominal (or a string), the stored value is the index of the corresponding nominal (or string) value in the attribute's definition.
- An example of definition of an *Instance* object:

```
...
// Create an empty instance with three attribute values
Instance inst = new Instance(3);

// Set instance's values for the attributes ``length'', ``weight''
// and ``position''
inst.setValue(length, 5.3);
inst.setValue(weight, 300);
inst.setValue(position, ``first'');

// Set instance's dataset to be the dataset ``race''
inst.setDataset(race);
...
```

- Some important methods:
  - *attribute(int index)* - returns an attribute with the given index
  - *classAttribute()* - returns the class attribute
  - *classIndex()* - returns the class attribute's index
  - *classValue()* - return class value in internal format
  - *dataset()* - returns dataset (Instances object) this instance is part of
  - *enumerateAttributes()* - returns an enumeration of all the attributes
  - *isMissing(int attrIndex)* - tests if a specific value is "'missing"
  - *numAttributes()* - number of all the attributes

- *value(int index)* - attribute value in internal format
- *setValue(int attIndex, double value)* - sets a specific value in the instance to the given value

2.3 weka.core.Instances

- This is the class for handling an ordered set of weighted instances.
- An example of definition of an *Instances* object:

```
...
// Read all the instances in the file
reader = new FileReader(filename);
Instances instances = new Instances(reader);

// Make the last attribute be the class
instances.setClassIndex(instances.numAttributes() - 1);
...
```

- Some important methods:
  - *add(Instance i)* - adds one instance to the end of the set
  - *attribute(int index)* - returns an attribute at a given position
  - *instance(int index)* - returns the instance at the given position
  - *classAttribute()* - returns the class attribute
  - *classIndex()* - returns the class attribute's index
  - *numAttributes()* - returns the number of attributes
  - *numInstances()* - returns the number of instances in the dataset
  - *enumerateAttributes()* - returns an enumeration of all the attributes
  - *enumerateInstances()* - returns an enumeration of all instances in the dataset
  - *setClass(Attribute att)* - sets the class attribute
  - *deleteStringAttributes()* - deletes all string attributes in the dataset
  - *deleteWithMissing(int index)* - deletes all instances with missing values for a given attribute

2.4 weka.classifiers.Classifier

- This is the abstract classifier. All schemas for numeric and nominal prediction in WEKA should extend this class.
- Methods to be implemented in your classifier (i.e. in class which extends weka.classifiers.Classifier):

- *buildClassifier(Instances data)* - generates a classifier; must initialize all fields of the classifier that are not being set via options;
- *classifyInstance(Instance instance)* - classifies a given instance; returns index of the predicted class as a double if the class is nominal, otherwise predicted value

- Example ZeroR (Majority Class):

```
public class ZeroR extends Classifier {

    /** The class value to predict. */
    private double m_ClassValue;

    /** The number of instances in each class (null if class numeric). */
    private double [] m_Counts;

    /** The class attribute. */
    private Attribute m_Class;

    public void buildClassifier(Instances instances)
    throws Exception {

      double sum = 0.0;
      m_Class = instances.classAttribute();
      m_ClassValue = 0;
      if (instances.classAttribute().isNumeric()) {
       m_Counts = null;
      } else if (instances.classAttribute().isNominal()) {
       m_Counts = new double [instances.numClasses()];
       for (int i = 0; i < m_Counts.length; i++) {
        m_Counts[i] = 1;
       }
       sum = instances.numClasses();
      } else {
       throw new Exception(
        "ZeroR can only handle nominal and numeric class attributes.");
      }

      Enumeration enum = instances.enumerateInstances();
      while (enum.hasMoreElements()) {
       Instance instance = (Instance) enum.nextElement();
       if (!instance.classIsMissing()) {
        if (instances.classAttribute().isNominal()) {
         m_Counts[(int)instance.classValue()] += 1;
        } else {
         m_ClassValue += instance.classValue();
        }
       }
      }
```

```
      if (instances.classAttribute().isNumeric()) {
       if (Utils.gr(sum, 0)) {
        m_ClassValue /= sum;
       }
      } else {
       m_ClassValue = Utils.maxIndex(m_Counts);
      }
    }

    public double classifyInstance(Instance instance) {
      return m_ClassValue;
    }
}
```

3 Exercises

- Write a java program which will take as an argument an .arff file and will output simple statistics for each attribute. The output should be the following:

  ```
  ...
  Attribute : name_of_nominal_attribute
  - mode : attr_value
  Attribute : name_of_numeric_attribute
  - mean : mean_value
  - variance : variance_value
  - stdDev :  stddev_value
  ...
  ```

  Remarks:

  - Do not take missing values into account.
  - *Mode* is the most common value obtained in a set of observations.
  - The *standard deviation* is the square root of the *sample variance*, which is calculated as following: substract the mean from each value, square the result, sum them together, and then divide by one less than number of values.

4 More information

- WEKA homepage: http://www.cs.waikato.ac.nz/ml/weka/