JAVA-ML Developers'Manual

Thu, 11/27/2008 - 14:47 — Thomas Abeel

This tutorial shows the very first step in using Java-ML. It will show you how to create Instances which are the default way to represent a sample of real world data.

The main way to represent data is the DenseInstance which requires a value for each attribute of an Instance. In most scenarios this representation of the data will suffice. In case your data is sparse, you can also put your data in a SparseInstance which requires less memory in case of sparse data (less than 10% attributes set).

How to create a DenseInstance

[Documented source code]

The block of code below will create an Instance with ten attributes. The values of these attributes are one through ten. The class label of this Instance is not set.

```
1.    /* values of the attributes. */
2.    double[] values = new double[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
3.    /*
4.     * The simplest incarnation of the DenseInstance constructor will
      only
5.     * take a double array as argument an will create an instance
      with given
6.     * values as attributes and no class value set. For unsupervised
      machine
7.     * learning techniques this is probably the most convenient
      constructor.
8.     */
9.    Instance instance = new DenseInstance(values);
```

To create an Instance with ten attributes, with values one through ten and with the class label "positive" we can use the following code. Note that the class label can be any object.

```
1.    /* values of the attributes. */
2.    double[] values = new double[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
3.    /* Create and an Instance with the above values and its class
      label set to "positive" */
4.    Instance instance = new DenseInstance(values,"positive");
```

How to create a SparseInstance

[Documented source code]

To create a SparseInstance you just have to specify the number of attributes that the Instance has and then you set the particular attributes values. The example below creates

an Instance with 10 attributes. Next, we set the value of attributes with index one, three and seven to particular values.

```
1.    /*
2.     * Here we will create an instance with 10 attributes, but will
   only set
3.     * the attributes with index 1,3 and 7 with a value.
4.     */
5.    /* Create instance with 10 attributes */
6.    Instance instance = new SparseInstance(10);
7.    /* Set the values for particular attributes */
8.    instance.put(1, 1.0);
9.    instance.put(3, 2.0);
10.    instance.put(7, 4.0);
```

Creating a Dataset

Thu, 11/27/2008 - 15:06 — Thomas Abeel

This tutorial show how to create a Dataset. Simply put, a Dataset is a collection of Instances. This tutorial assumes you know how to create an Instance, either DenseInstance or SparseInstance will work as both are implementations of the Instance interface.

For the purpose of this tutorial we will use a method of InstanceTools to create Instances with random values for its attributes. In reality you will either create the Instances yourself or you will load them from a file (see next page in this tutorial trail). The method *InstanceTools.randomInstance(25);* will create a DenseInstance with 25 attributes that all have values between zero and one.

Now on to creating a Dataset. Dataset is an interface which defines a number of operations on a data set. The default implementation of Dataset is DefaultDataset. At the moment no other implementations of Dataset are available. In the following example we will create a DefaultDataset and populate it with 10 Instances that have been randomly generated as described in the previous paragraph.

```
1. Dataset data = new DefaultDataset();
2. for (int i = 0; i < 10; i++) {
3.    Instance tmpInstance = InstanceTools.randomInstance(25);
4.    data.add(tmpInstance);
5. }
```

Load data from file

Fri, 09/05/2008 - 09:29 — Thomas Abeel

This tutorial shows how you can read data from a number of file types. At the moment the library supports any type of field formatted file, i.e. one sample on a line and the attributes separated by a symbol (commonly , (comma) ;(semi-colon) or \t (tab)). Common formats include CSV and TSV and this type of file is further denoted by 'CSV-like file'. Java-ML also supports ARFF formatted files, with the limitation that only the class label can be non-numeric.

Loading data from CSV-like files

[Documented source code]

```
1. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
```

The first parameter of loadDataset is the file to load the data from. The second parameter is the index of the class label (zero-based) and the final parameter is the separator used to split the attributes in the file.

Data for the above example.
Iris data set

Loading sparse data from CSV-like files

[Documented source code]

```
1. Dataset data =  FileHandler.loadSparseDataset(new
   File("sparse.tsv"), 0, " ",":");
```

The first parameter of loadSparseDataset is the file to load the data from. The second parameter is the index of the class label (zero-based), the third parameter is the attribute separator and the final parameter is the separator used to split the index and value of the attribute.

Sample data file for sparse data
Sample sparse data

Loading data from an ARFF formatted file

Loading from an ARFF formatted file is much like loading from a CSV or TSV file.
[Documented source code]

```
1. Dataset data = ARFFHandler.loadARFF(new File("iris.arff"), 4);
```

The loadARFF method of the ARFFHandler only has two arguments, the first one to indicate the file that should be loaded and the second one to indicate the index of the class

label. The ARFFHandler also has a method with only a File as argument to load files that do not contain a class label.

There are some caveats with the ARFF loader: (i) all the header information is ignored by the loader, and (ii) Java-ML only supports numeric attributes, data sets that have attributes that are not numeric will crash the loader.

Sample data file for ARFF loader
Sample Iris ARFF data

Note that the FileHandler works with flat files or GZIP compressed files, while the ARFFHandler only works with flat files.

Store data to file

Thu, 11/27/2008 - 16:10 — Thomas Abeel

Java-ML can store Datasets back to file using the FileHandler class. At the moment it is not possible to store data in the ARFF format.

In the previous article in the tutorial trail we have shown how to load data from a file, we will use this mechanism again to load our data and then store it to a different file.

Storing data to a file
[Documented source code]

```
1. /* Load the iris data set from file */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. System.out.println(data);
4. /* Store the data back to another file */
5. FileHandler.exportDataset(data,new File("output.txt"));
```

The above example will load the iris data set and will store it back to another file (output.txt). Each instance is written to the file separately with the class label on position 0. The fields are delimited with a tab character.

The data that was outputted can be read again with the following code:

```
1. Dataset data = FileHandler.loadDataset(new File("output.txt"),
   0,"\t");
```

Caveats: Note that data sets with mixed sparse and dense instances may result in files that cannot be loaded with the methods in FileHandler. Exporting data sets where some

Instances have their class label set and some don't, will result in unwanted behaviour when loading the file with the methods in FileHandler.

Normalization

Tue, 10/12/2010 - 12:59 — Thomas Abeel

Can be done using the filter classes for normalization.

```
1. /* Create data set with random instances */
2. Dataset data=new DefaultDataset();
3. for(int i=0;i<100;i++){
4.   Instance rgA=InstanceTools.randomInstance(5);
5.   data.add(rgA);
6. }
7. NormalizeMidrange nmr=new NormalizeMidrange(0.5,1);
8. /* Instanciate new filter */
9. nmr.build(data);
10.
11. Instance rgB=InstanceTools.randomInstance(5);
12. /* Filter another instances */
13. nmr.filter(rgB);
```

Tue, 05/04/2010 - 07:41 — Thomas Abeel

Sampling can be done with the Sampling class.

The most interesting methods are the different signatures of 'sample'

```
1. public Pair<Dataset, Dataset> sample(Dataset data)
2. public Pair<Dataset, Dataset> sample(Dataset data, int size)
3. public Pair<Dataset, Dataset> sample(Dataset data, int size, long
   seed)
```

The data set is the one from which to sample. The size is the number of items that should be in the sample. The seed is for the random generator that is used in the sampling.

The methods return a pair of data sets. The first part is the actual sample, the second part of the pair is a data set containing the out-of-bag samples.

A working example:

```
1. Dataset data=FileHandler.loadDataset(new
   File("devtools/data/iris.data"),4,",");
```

```
2. Sampling s=Sampling.SubSampling;
3. for(int i=0;i<5;i++){
4.  Pair<Dataset, Dataset> datas=s.sample(data,
   (int)(data.size()*0.8),i);
5.  Classifier c=new LibSVM();
6.  c.buildClassifier(datas.x());
7.  Map pms=EvaluateDataset.testDataset(c, datas.y());
8.  System.out.println(pms);
9. }
```

Tue, 12/16/2008 - 15:55 — Thomas Abeel

A clustering algorithm creates a division of the orginal dataset. In Java-ML this is done with the method *cluster* of the Clusterer interface.

Creating and running a clustering algorithm

```
1. /* Load a dataset */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create a new instance of the KMeans algorithm, with no options
4.   * specified. By default this will generate 4 clusters. */
5. Clusterer km = new KMeans();
6. /* Cluster the data, it will be returned as an array of data sets,
   with
7.   * each dataset representing a cluster. */
8. Dataset[] clusters = km.cluster(data);
```

[Documented source code]

The code above will load the example iris data set. Next it creates an instance of the K-means algorithms and uses it to cluster the data. The results are returned in an array of Datasets where each Dataset represents a cluster.

Note that there is no guarantee that all original Instances will occur in the clusters or that each Instance occurs only once. Some algorithms allow overlapping clusters, some algorithms allow that 'noisy' datapoints are removed. This is algorithm specific and you can find more information on the API page for each algorithm.

Clustering basics

Tue, 12/16/2008 - 15:55 — Thomas Abeel

A clustering algorithm creates a division of the orginal dataset. In Java-ML this is done with the method *cluster* of the Clusterer interface.

Creating and running a clustering algorithm

```
1. /* Load a dataset */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create a new instance of the KMeans algorithm, with no options
4.   * specified. By default this will generate 4 clusters. */
5. Clusterer km = new KMeans();
6. /* Cluster the data, it will be returned as an array of data sets,
   with
7.   * each dataset representing a cluster. */
8. Dataset[] clusters = km.cluster(data);
```

[Documented source code]

The code above will load the example iris data set. Next it creates an instance of the K-means algorithms and uses it to cluster the data. The results are returned in an array of Datasets where each Dataset represents a cluster.

Note that there is no guarantee that all original Instances will occur in the clusters or that each Instance occurs only once. Some algorithms allow overlapping clusters, some algorithms allow that 'noisy' datapoints are removed. This is algorithm specific and you can find more information on the API page for each algorithm.

Cluster evaluation

Tue, 12/16/2008 - 16:19 — Thomas Abeel

Java-ML provides a large number of cluster evaluation measures that are provided in the package net.sf.javaml.clustering.evaluation. All scores are measures for the quality of the clustering, i.e. how well it reflects the properties of the data. Mostly they try to quantify how well the data is separated in logical units by the clustering algorithm.

All scores implement the *double score(Dataset[]clusters)* method. This method returns a score for an array of datasets that is returned from a clustering algorithm. Typical usage is illustrated in the code snippet below.

```
1. /* We load some data */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* We create a clustering algorithm, in this case the k-means
4.  * algorithm with 4 clusters. */
5. Clusterer km=new KMeans(4);
6. /* We cluster the data */
```

```
7. Dataset[] clusters = km.cluster(data);
8. /* Create a measure for the cluster quality */
9. ClusterEvaluation sse= new SumOfSquaredErrors();
10. /* Measure the quality of the clustering */
11. double score=sse.score(clusters);
```

Weka clustering

Fri, 01/16/2009 - 09:17 — Thomas Abeel

Clustering algorithms from Weka can be accessed in Java-ML through the WekeClusterer bridge. This class makes it easy to use a clustering algorithm from Weka in Java-ML.

In the example below, we load the iris dataset, we create a clusterer from Weka (XMeans), we wrap it in the bridge and use the bridge to do the clustering.

```
1. /* Load data */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create Weka classifier */
4. XMeans xm = new XMeans();
5. /* Wrap Weka clusterer in bridge */
6. Clusterer jmlxm = new WekaClusterer(xm);
7. /* Perform clustering */
8. Dataset[] clusters = jmlxm.cluster(data);
9. /* Output results */
10. System.out.println(clusters.length);
```

Feature scoring

Thu, 01/15/2009 - 15:46 — Thomas Abeel

All feature scoring algorithms implements the following method. Higher scores are better.

```
1. public double score(int attIndex);
```

This method will return the score for the supplied feature index.

Typical usage of a feature scoring algorithm is shown in the snippet below

```
1. /* Load the iris data set */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create a feature scoring algorithm */
```

```
4. GainRatio ga = new GainRatio();
5. /* Apply the algorithm to the data set */
6. ga.build(data);
7. /* Print out the score of each attribute */
8. for (int i = 0; i < ga.noAttributes(); i++)
9.     System.out.println(ga.score(i));
```

Feature ranking

Thu, 01/15/2009 - 15:46 — Thomas Abeel

All feature ranking algorithms provide the following method to determine the rank of a features. Lower ranks are better.

```
1. public int rank(int attIndex);
```

The method will return the rank of the feature with the provided index.
Typical usage of a feature ranking algorithm is very similar to the use of the feature scoring algorithms.

```
1. /* Load the iris data set */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create a feature ranking algorithm */
4. RecursiveFeatureEliminationSVM svmrfe = new
   RecursiveFeatureEliminationSVM(0.2);
5. /* Apply the algorithm to the data set */
6. svmrfe.build(data);
7. /* Print out the rank of each attribute */
8. for (int i = 0; i < svmrfe.noAttributes(); i++)
9.     System.out.println(svmrfe.rank(i));
```

Feature subset selection

Thu, 01/15/2009 - 15:47 — Thomas Abeel

Subset selection algorithms differ with the scoring and ranking methods in that they only provide a set of features that are selected without further information on the quality of each feature individually.
Subset selection algorithms provide the method

```
1. public Set<Integer> selectedAttributes();
```

which will return a set of feature indices that have been selected by the algorithm.

The basic use of a feature subset selection algorithm is depicted in the snippet below.

```
1.  /* Load the iris data set */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Construct a greedy forward subset selector */
4. GreedyForwardSelection ga = new GreedyForwardSelection(1, new
   PearsonCorrelationCoefficient());
5. /* Apply the algorithm to the data set */
6. ga.build(data);
7. /* Print out the attribute that has been selected */
8. System.out.println(ga.selectedAttributes());
```

[Documented source code]
This examples create a greedy forward selection algorithm that will select one ('the best') feature. To determine the quality of the feature in this example the Pearson correlation is used.

Ensemble feature ranking

Thu, 01/15/2009 - 15:45 — Thomas Abeel

The ensemble feature ranking algorithm is another form of feature ranking and as such provides the following method to determine the rank of a feature. Lower ranks are better.

```
1. public int rank(int attIndex);
```

The method will return the rank of the feature with the provided index.
Typical usage of an ensemble feature ranking algorithm is very similar to the use of a single feature ranking algorithm.

```
1. /* Load the iris data set */
2. Dataset data = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
3. /* Create a feature ranking algorithm */
4. RecursiveFeatureEliminationSVM[] svmrfes = new
   RecursiveFeatureEliminationSVM[10];
5. for (int i = 0; i < svmrfes.length; i++)
6.     svmrfes[i] = new RecursiveFeatureEliminationSVM(0.2);
7. LinearRankingEnsemble ensemble = new
   LinearRankingEnsemble(svmrfes);
```

```
8. /* Build the ensemble */
9. ensemble.build(data);
10. /* Get rank of i-th feature */
11. int rank=ensemble.rank(i)
```

Weka attribute selection

Sun, 05/10/2009 - 14:57 — Thomas Abeel

This article provides a brief introduction to the concepts of
using Weka attribute selection through the Java-ML feature selection interfaces.

In Weka, attribute selection searches through all possible combination of attributes in the data to
find which subset of attributes works best for prediction. It employs
two objects which include an attribute evaluator and and search method.

Any combination of these attribute evaluator and search algorithms can be used to determine the score and rank attribute in a data set. Currently, only
Java-ML feature scoring and feature ranking are available through the wrapper, subset selection is not yet implemented.

Typical use of the Weka feature selection wrapper is shown in the snippet below:

```
1. /* Load the iris data set */
2. Dataset data = FileHandler.loadDataset(new File [1]("iris.data"),
   4, ",");
3. /*Create a Weka AS Evaluation algorithm */
4. ASEvaluation eval = new GainRatioAttributeEval();
5. /* Create a Weka's AS Search algorithm */
6. ASSearch search = new Ranker();
7. /* Wrap WEKAs' Algorithms in bridge */
8. WekaAttributeSelection wekaattrsel = new
   WekaAttributeSelection(eval,search);
9. /* Apply the algorithm to the data set */
10. wekaattrsel.build(data);
11. /* Print out the score and rank  of each attribute */
12. for (int i = 0; i &lt; wekaattrsel.noAttributes(); i++)

   System [2].out.println("Attribute  " +  i +  "  Ranks  " +
wekaattrsel.rank(i) + " and Scores "  + wekaattrsel.score(i) );
```

Tue, 12/16/2008 - 14:47 — Thomas Abeel

This tutorial explains the basics of setting up a classifier, training the algorithm and evaluating its performance. First we need to initialize a classifier, next we can train it with some data, and finally we can use it to classify new instances.

Creating a classifier

The following sample loads data from the iris data set, next we construct a K-nearest neighbor classifier and we train it with the data.

```
1. /* Load a data set */
2. Dataset data = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
3.  /* Contruct a KNN classifier that uses 5 neighbors to make a
4.   *decision. */
5. Classifier knn = new KNearestNeighbors(5);
6. knn.buildClassifier(data);
```

[Documented source code]

Note that the build method of a classifier may modify the Dataset that is provided as parameter.

Evaluating the performance of a classifier

Now that we have constructed and trained a classifier, we can use it to classify new Instances. In this example we will reload the iris data set and use the trained classifier to predict the class label for each Instance.

```
1.  Dataset dataForClassification = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
2. /* Counters for correct and wrong predictions. */
3. int correct = 0, wrong = 0;
4. /* Classify all instances and check with the correct class values
   */
5. for (Instance inst : dataForClassification) {
6.     Object predictedClassValue = knn.classify(inst);
7.     Object realClassValue = inst.classValue();
8.     if (predictedClassValue.equals(realClassValue))
9.         correct++;
10.     else
11.         wrong++;
12. }
```

13. This example will go over all instances in the iris data set and try to predict its class by majority voting on its 5 neighbors. In this example this will result in 145 correct predictions and 5 wrong ones.

14. Note that this is not the proper way to do validation of a classifer. For the proper technique, look at cross validation.

Evaluate classifier on a dataset

Mon, 12/01/2008 - 13:16 — Thomas Abeel

This tutorial shows you how you can test the performance of a classifier on a data set. This tutorial will introduce two classes. EvaluateDataset, which allows you to test a classifier on a data set and it will also introduce PerformanceMeasure. This class is used to store information regarding the performance of a classifier.

Evaluate a classifier on a dataset

```
1. Dataset data = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
2. Classifier knn = new KNearestNeighbors(5);
3. knn.buildClassifier(data);
4. Dataset dataForClassification = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
5.
6. Map<Object, PerformanceMeasure> pm =
   EvaluateDataset.testDataset(knn, dataForClassification);
7. for(Object o:pm.keySet())
8.     System.out.println(o+": "+pm.get(o).getAccuracy());
```

[Documented source code]
This sample loads the iris data set, constructs a 5-nearest neighbor classifier and loads the iris data again.

The testDataset method will use the trained classifier to predict the labels for all instances in the supplied data set. The performance of the classifier is returned as a map that contains for each class a performance measure. A PerformanceMeasure is a wrapper around the values for the true positives, true negatives, false positives and false negatives. This class also provides a number of convenience method to calculate a number of aggregate measures like accuracy, f-score, recall, precision, sensitivity, specificity, etc.

Classification cross validation

Thu, 11/27/2008 - 16:33 — Thomas Abeel

In this tutorial we discuss how you can perform cross-validation with Java-ML.

In this tutorial we assume that you know how to load data from a file, how to create a classifier and how to work with the PerformanceMeasure.

Cross validation in Java-ML can be done using the CrossValidation class. The code below shows how to use this class.

[Documented source code]

```
1. /* Load data */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Construct KNN classifier */
4. Classifier knn = new KNearestNeighbors(5);
5. /* Construct new cross validation instance with the KNN classifier
   */
6. CrossValidation cv = new CrossValidation(knn);
7. /* Perform cross-validation on the data set */
8. Map<Object, PerformanceMeasure> p = cv.crossValidation(data);
```

This example first loads the iris data set and then constructs a K-nearest neighbors classifier that uses 5 neighbors to classify instances.
In the next step we create a cross-validation with the constructed classifier. Finally we instruct the cross-validation to run on a the loaded data. By default a 10-fold cross validation will be performed and the result for each class will be returned in a Map that maps each class label to its corresponding PerformanceMeasure.

Using the same folds for multiple runs

```
1. /* Load data */
2. Dataset data = FileHandler.loadDataset(new
   File("devtools/data/iris.data"), 4, ",");
3. /* Construct KNN classifier */
4. Classifier knn = new KNearestNeighbors(5);
5. /* Construct new cross validation instance with the KNN classifier,
   */
6. CrossValidation cv = new CrossValidation(knn);
7. /* 5-fold CV with fixed random generator */
8. Map<Object, PerformanceMeasure> p = cv.crossValidation(data, 5, new
   Random(1));
9. Map<Object, PerformanceMeasure> q = cv.crossValidation(data, 5, new
   Random(1));
10. Map<Object, PerformanceMeasure> r = cv.crossValidation(data, 5,
   new Random(25));
```

[Documented source code]

The example above performs three rounds of cross-validation on the data set. The first two are using exactly the same folds as the random generator used to create the folds is initialized with the same seed. The third CV will be run on different folds as it uses a different seed.

While in this example we have used the same classifier, one can exchange the classifier with a different one and test different classifiers on exactly the same folds.

Weka classifier

Fri, 01/16/2009 - 09:24 — Thomas Abeel

Classification algorithms from Weka can be accessed from within Java-ML and used the same way as the native algorithms by using the WekaClassification bridge. This class can be wrapped around Weka classifiers and makes them transparently available to Java-ML based programs.

In the example below, we first load the iris data set. Next, we create a SMO support vector machine from Weka with default settings. Then, we wrap the SMO in the WekaClassifier bridge. Finally, we perform cross-validation on the classifier and write out the results.

```
1. /* Load data */
2. Dataset data = FileHandler.loadDataset(new File("iris.data"), 4,
   ",");
3. /* Create Weka classifier */
4. SMO smo = new SMO();
5. /* Wrap Weka classifier in bridge */
6. Classifier javamlsmo = new WekaClassifier(smo);
7. /* Initialize cross-validation */
8. CrossValidation cv = new CrossValidation(javamlsmo);
9. /* Perform cross-validation */
10. Map<Object, PerformanceMeasure> pm = cv.crossValidation(data);
11. /* Output results */
12. System.out.println(pm);
```