November 12, 2013

Create a simple predictive analytics classification model in Java with Weka

James Howard, *University of Maryland Baltimore County*

Create a simple predictive analytics classification model in Java with Weka

Introduction to basic data mining and classification

James Howard (jh@jameshoward.us)
Senior Associate
Kore Federal Inc.

12 November 2013

Get an overview of the Weka classification engine and learn how to create a simple classifier for programmatic use. Understand how to store and load models, manipulate them, and use them to evaluate data. Consider applications and implementation strategies suitable for the enterprise environment so you turn a collection of training data into a functioning model for real-time prediction.

Introduction

This article introduces Weka and simple classification methods for data science. It starts with an introduction to basic data mining and classification principles and provides an overview of Weka, including the development of simple classification models with sample data. Then it will introduce the Java™ programming environment with Weka and show how to store and load models, manipulate them, and use them to evaluate data. Finally, this article will discuss some applications and implementation strategies suitable for the enterprise environment. With this basic information, a data analyst should be able to turn a collection of training data into a functioning model for real-time prediction.

> *“ Weka is an open source program for machine learning written in the Java programming language .... Weka has a utilitarian feel and is simple to operate. ”*
>
> James Howard

Real-time classification of data, the goal of predictive analytics, relies on insight and intelligence based on historical patterns discoverable in data. These patterns are presumed to be causal and, as such, assumed to have predictive power. That predictive power, coupled with a flow of new data, makes it possible to analyze and categorize data in an online transaction processing (OLTP) environment.

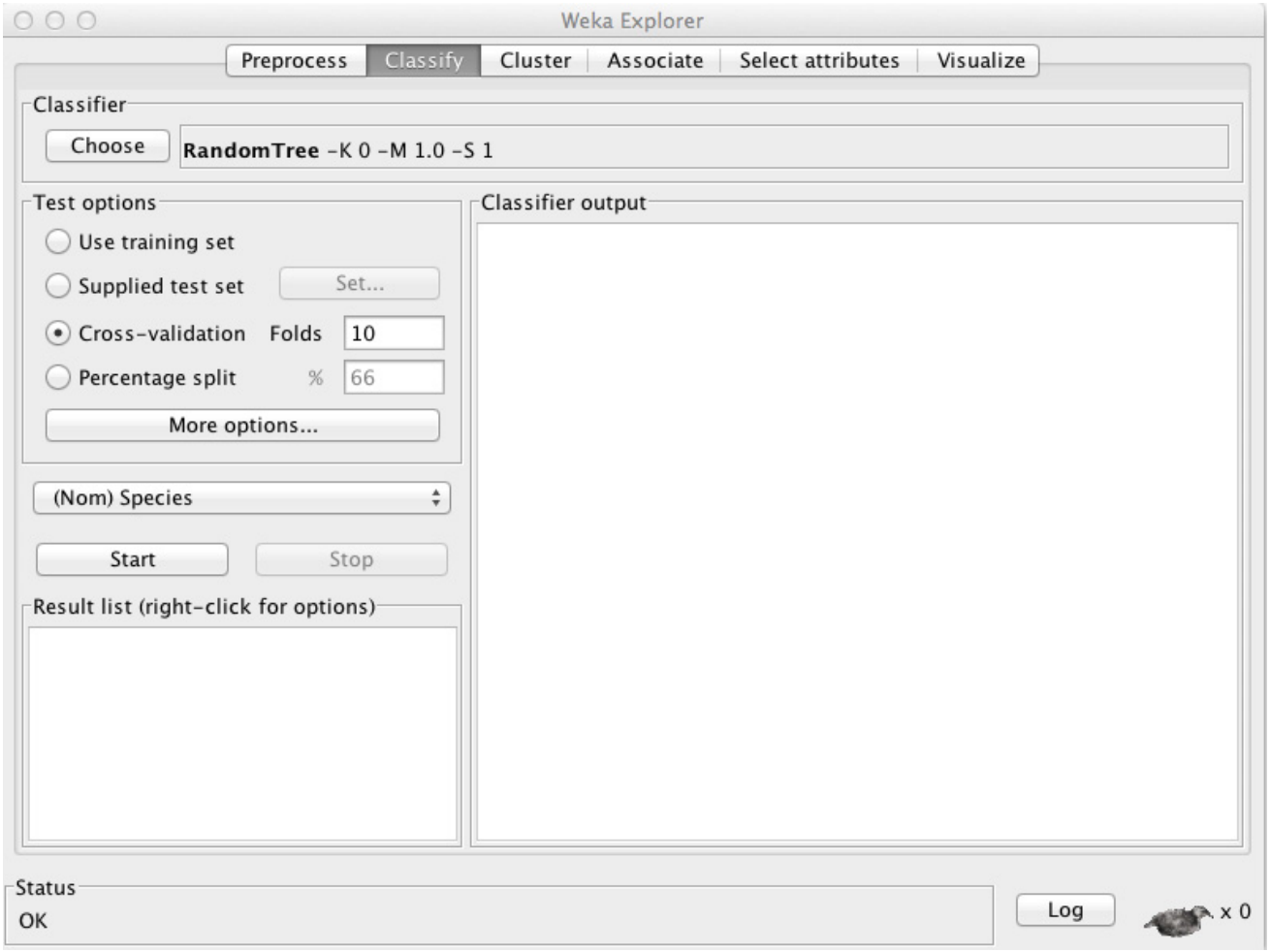
Machine learning, at the heart of data science, uses advanced statistical models to analyze past instances and to provide the predictive engine in many application spaces. These statistical models include traditional logistic regression (also known as *logit*), neural networks, and newer modeling techniques like RandomForest. These models are trained on the sample data provided, which should include a variety of classes and relevant data, called factors, believed to affect the classification. Models like this are evaluated using a variety of techniques, and each type can serve a different purpose, depending on the application. The simplest application domains use classification to turn these factors into a class prediction of the outcome for new cases. The prediction can be true or false, or membership among multiple classes.

Classification methods address these class prediction problems. The most familiar of these is probably the logit model taught in many graduate-level statistics courses. The example in this article will use the RandomTree classifier, included in Weka. The RandomTree is a tree-based classifier that considers a random set of features at each branch. It has few options, so it is simpler to operate and very fast. The RandomTree classifier will be demonstrated with Fisher's iris dataset. Fisher used a sample of 150 petal and sepal measurements to classify the sample into three species. This dataset is a classic example frequently used to study machine learning algorithms and is used as the example here.

Using Weka

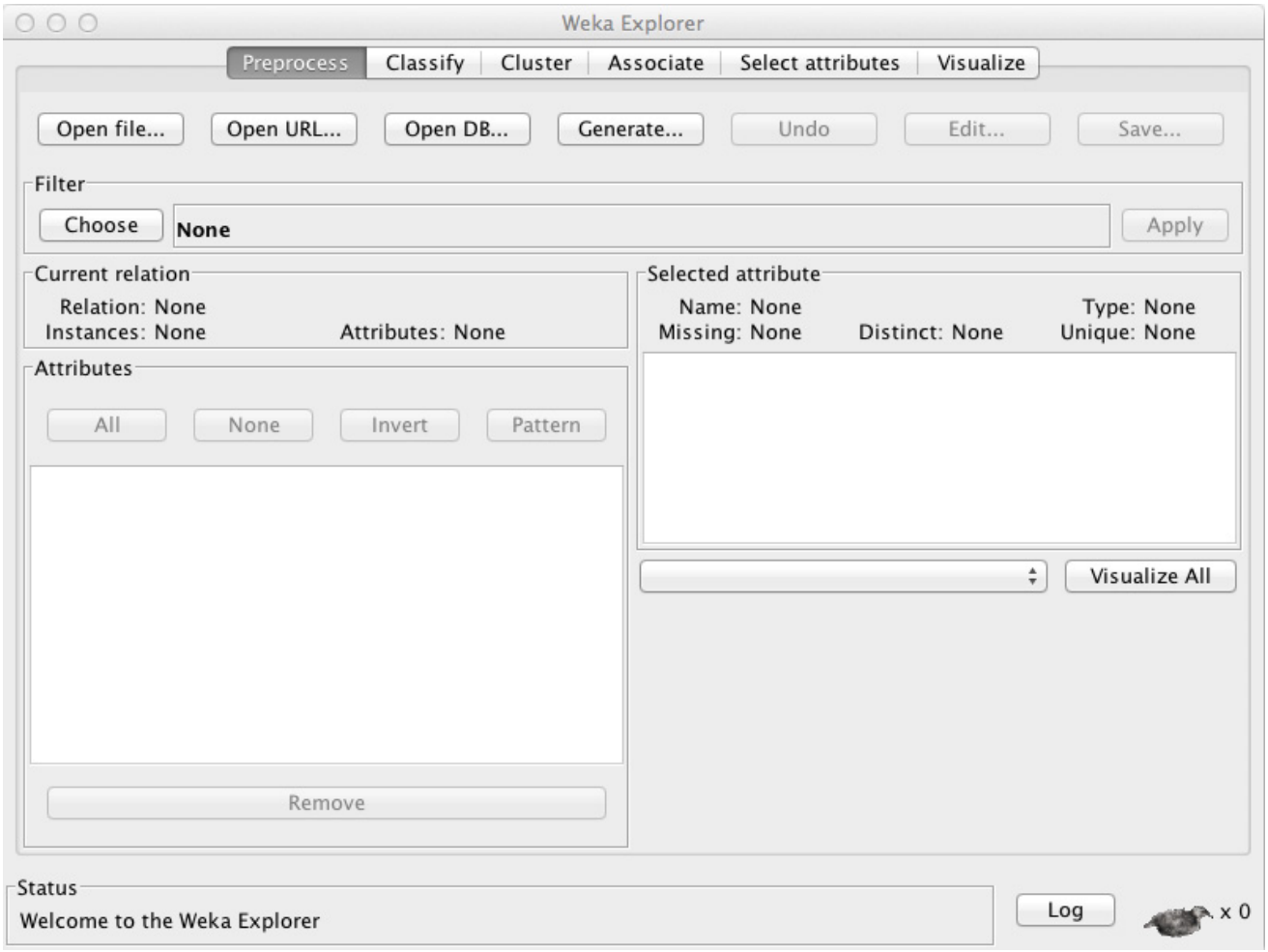
Weka is an open source program for machine learning written in the Java programming language developed at the University of Waikato. Coming from a research background, Weka has a utilitarian feel and is simple to operate. Upon opening the Weka, the user is given a small window with four buttons labeled **Applications**. Everything in this article is under **Explorer**.

Figure 1. The Weka startup box



After selecting **Explorer**, the Weka Explorer opens and six tabs across the top of the window describe various data modeling processes. In addition to the graphical interface, Weka includes a primitive command-line interface and can also be accessed from the R command line with an add-on package. Start with the **Preprocess** tab at the left to start the modeling process.

Figure 2. Opening a data file in Weka



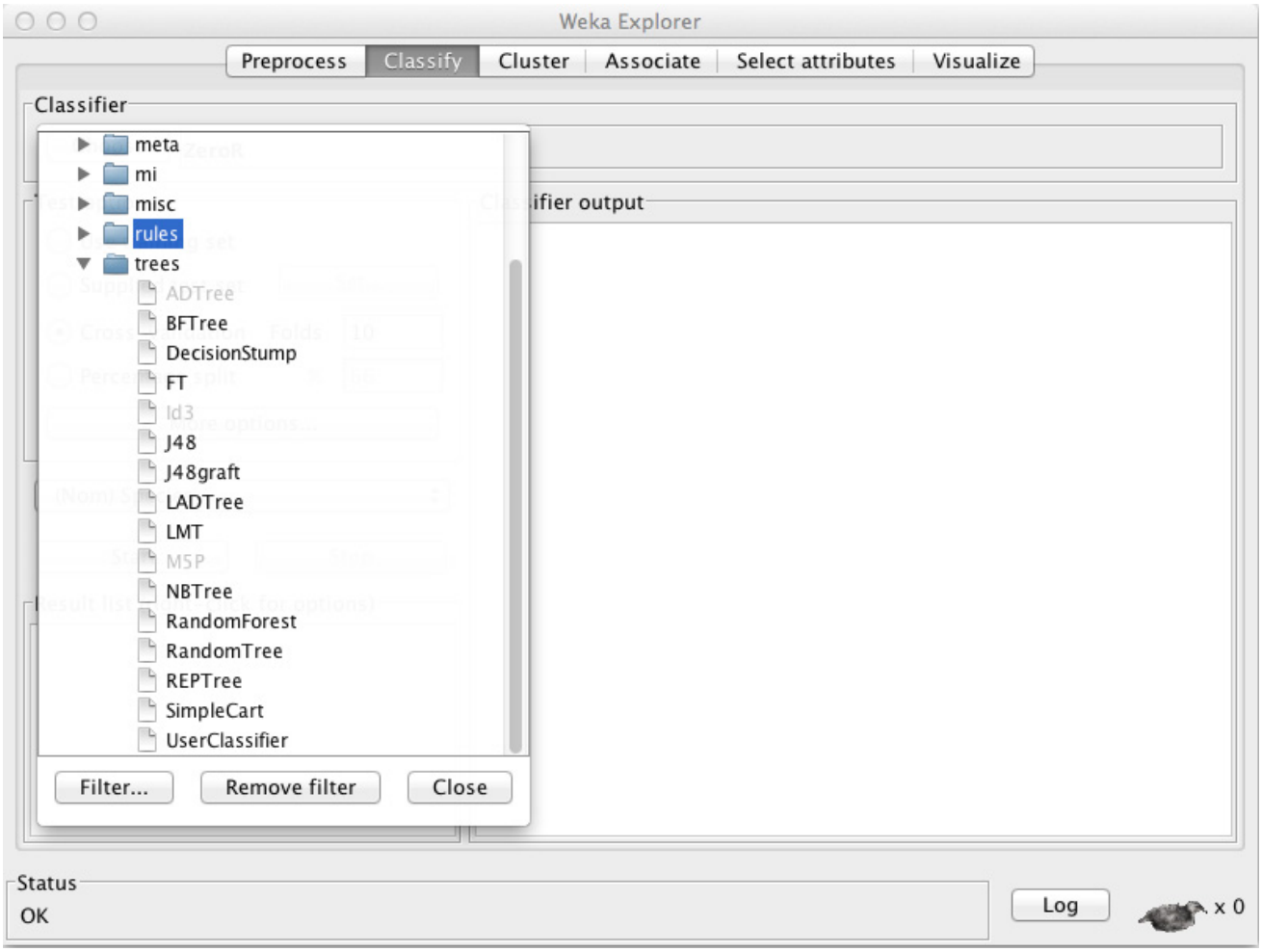
The iris dataset consists of five variables. Two describe the observed petal of the iris flowers: the length, and the width. Two describe the observed sepal of the iris flowers: also the length and the width. The last variable in the dataset is one of three species identifiers: setosa, versicolor, or virginica. There are 50 observations of each species. Generally, the setosa observations are distinct from versicolor and virginica, which are less distinct from each other. However, many machine learning algorithms and classifiers can distinguish all three with a high accuracy. The iris dataset is available from many sources, including Wikipedia, and is included with the example source code with this article.

Weka can read in a variety of file types, including CSV files, and can directly open databases. Because Weka is a Java application, it can open any database there is a Java driver available for. It also includes a simple file format, called ARFF, which is arranged as a CSV file, with a header that describes the variables (see Resources). The iris dataset is available as an ARFF file. To read in a file, start Weka, click **Explorer** and select **Open file**. This will make the file the current dataset in Weka. Only one dataset can be in memory at a time.

In the case of the iris dataset, the species is the classification of the data. Weka automatically assigns the last column of an ARFF file as the class variable, and this dataset stores the species

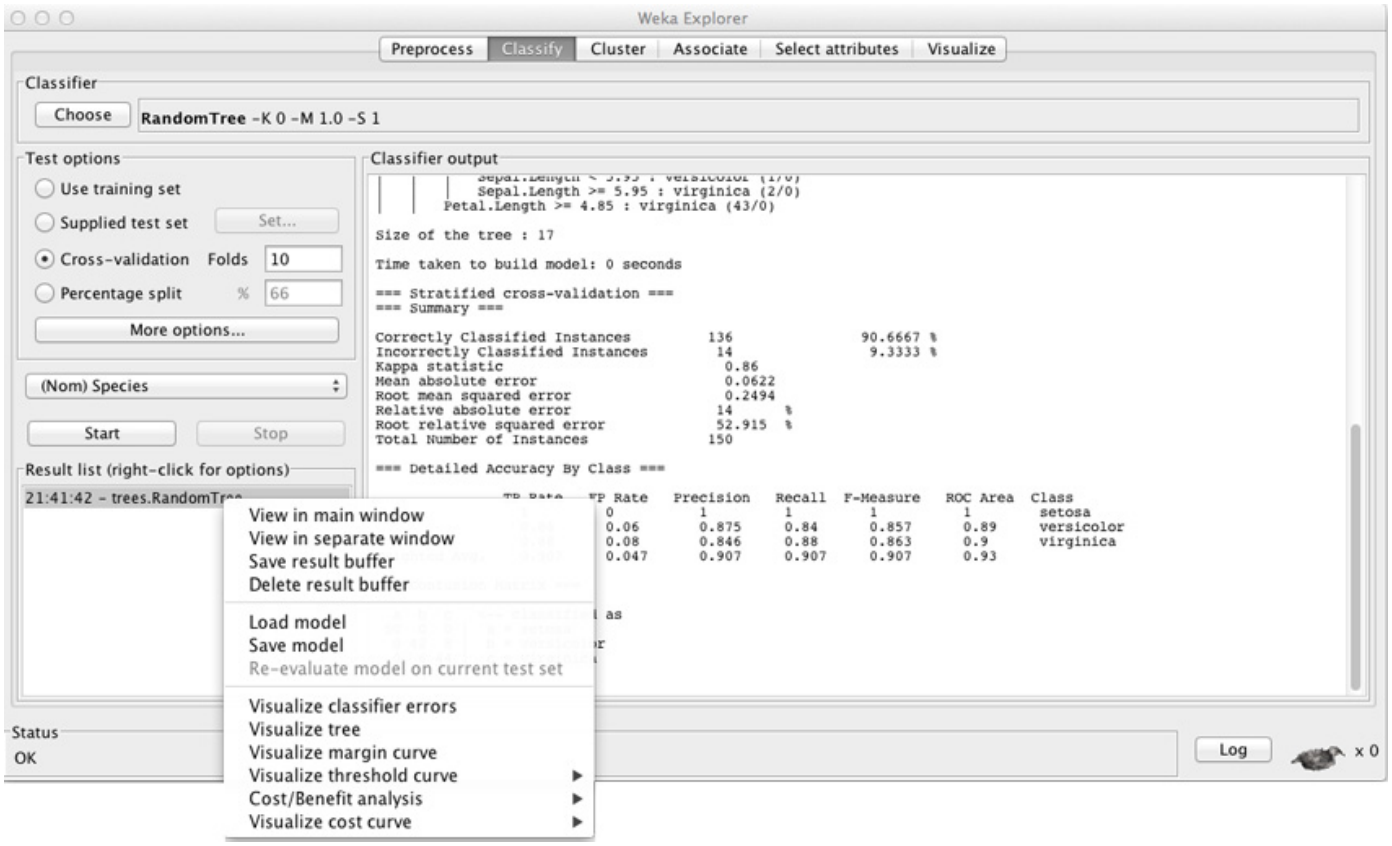
in the last column. Therefore, no adjustments need to be made initially. To train an initial model, select **Classify** at the top of the screen. The model type, by default, is ZeroR, a classifier that only predicts the base case, and is therefore not very usable. To change the model to train, click **Choose** from the top left-hand side of the screen, which presents a hierarchical list of classifier types. Choose from **trees** the type **RandomTree** to train a basic tree model. Click **Start** to start the modeling process.

Figure 3. Starting the modeling process



After a few seconds, Weka will produce a classifier. The classifier is listed under **Results List** as **trees.RandomTree** with the time the modeling process started. Weka will keep multiple models in memory for quick comparisons. It will also display in the box **Classifier output** some model performance metrics, including the area under the ROC curve and a confusion matrix for the classifier. The example uses 10-fold cross-validation for testing. Each classifier has distinct options that can be applied, but for this purpose, the model is good enough in that it can correctly classify 93 percent of the examples given. Save the model by right-clicking on the classifier result and selecting **Save model**. The default model extension is `.model` when saved. The entire process can be clicked through for exploratory or experimental work or can be automated from R through the RWeka package.

Figure 4. Saving a model



Working with the Classifier

From here, the saved model can be reloaded in Weka and run against new data. The more interesting option, however, is to load the model into Weka through a Java program and use that program to control the execution of the model independent of the Weka interface. This gives Weka a distinct advantage since Java is usually available within database and OLTP environments, such as Oracle, without modification. Additionally, Weka provides a JAR file with the distribution, called weka.jar that provides access to all of Weka's internal classes and methods.

This process begins with creating a Weka classifier object and loading the model into it. The classifier object is an abstract interface within Java, and any of the Weka model types can be loaded in to it. This advantage means the same code can execute a logistic regression, a support vector machine, a RandomForest, or any other classifier type supported by Weka. These models can also be exchanged at runtime as models are rebuilt and improved from new data. After the model is loaded into the `classifier` object, it is fully functional and ready for classification tasks.

This process is shown in the constructor for the `Iris` class. The class includes an instance variable of type `Classifier` called `classModel` to hold the `classifier` object. The class also includes an instance variable of type `string` called `classModelFile` that includes the full path to the stored model file. This model is stored as a serialized Java object. The stored model file can be deployed as a JAR file, the file is opened with `getResourceAsStream()`, and it is read using Weka's static function: `SerializationHelper.read()`. This returns the model file as a Java Object that can

be cast to `Classifier` and stored in `classModel`. At this point, the classifier needs no further initialization.

Using the Classifier

With the classifier loaded, the process for using it can depart from the general approach for programming in Java. Weka is designed to be a high-speed system for classification, and in some areas, the design deviates from the expectations of a traditional object-oriented system. It provides its own implementation of vectors (FastVector) and measurement sets for classification (Instance). These objects are not compatible with similar objects available in the Java Class Library. That complicates using them. In the provided example, the `classifySpecies()` method of the `Iris` class takes as a single argument a `Dictionary` object (from the Java Class Library) with both keys and values of type `String`. This structure allows callers to use standard Java object structures in the classification process and isolates Weka-specific implementation details within the Iris class.

The `classifySpecies()` method must convert the `Dictionary` object it receives from the caller into an object Weka can understand and process. Weka operates on objects called `Instance`s, provided within the weka.core package. The `Instance` object includes a set of values that the classifier can operate on. Alternatively, the classifier can be trained on a collection of `Instance` objects if the training is happening through Java instead of the GUI. However, the relationship between the feature metadata, such as names, and the values are not stored in the `Instance` object. The particulars of the features, including type, are stored in a separate object, called `Instances`, which can contain multiple `Instance` objects. An Instance must be contained within an `Instances` object in order for the classifier to work with it. The `Instances` object is also available in weka.core.

A major caveat to working with model files and classifiers of type `Classifier`, or any of its subclasses, is that models may internally store the data structure used to train model. However, there is no API for restoring that information. So a class working with a `Classifier` object cannot effectively do so naively, but rather must have been programmed with certain assumptions about the data and data structure the `Classifier` object is to be applied to. This caveat underlies the design of the `classifySpecies()` method in the `Iris` class. Several design approaches are possible. For instance, the class may initialize the data structure as part of the `Iris` class constructor. In this example, the setup takes place at the time of classification. This is reasonable if the implementation does not require a high-speed response and it will only be called a few times.

The `classifySpecies()` method begins by creating a list of possible classification outcomes. In the case of the iris dataset, this is a list of three species included in the original dataset: setosa, versicolor, and virginica. These are each added to a `FastVector` object by using the FastVector's `addElement()` method. Then, once the potential outcomes are stored in a `FastVector` object, this list is converted into a nominal variable by creating a new `Attribute` object with an attribute name of `species` and the FastVector of potential values as the two arguments to the `Attribute` constructor. The `FastVector` must contain the outcomes list in the same order they were presented in the training set. From the ARFF file storing the initial iris measurements, these are:

```
@attribute Species {'setosa','versicolor','virginica'}
```

And in Java, the potential species values are loaded in the same order:

```
dataClasses.addElement("setosa");
dataClasses.addElement("versicolor");
dataClasses.addElement("virginica");
```

After the species classes are prepared, the `classifySpecies()` method will loop over the `Dictionary` object and perform two tasks with each iteration:

1. It will assemble a collection of keys, which are aggregated into a second `FastVector` object, using the `FastVector's addElement()` method. That `FastVector` object contains all the feature names to be supplied to the `Classifier` object.
2. It will get the value associated with each key. The values are floating-point numbers stored as strings, so they must be converted to a floating-point type, double in this case. An array of doubles holds each value as it is returned from the `Dictionary` object.

The array needs to hold the number of elements in the `Dictionary` object, plus one that will eventually hold the calculated class. Similarly, after the loop executes, the species `Attribute`, created at the start of the function, is added as the final element of the attributes `FastVector`.

The next step is to create the final object the classifier will operate on. This is a two-step process involving the `Instances` class and `Instance` class, as described above. The process begins with creating the `Instances` object. The first argument to the constructor is the name of the relationship. For a data instance to be classified, it is arbitrary and this example calls it *classify.* The second argument to the constructor is the `FastVector` containing the attributes list. The final argument is the capacity of the dataset. In this example, the capacity is set to 0. After the `Instances` object is created, the `setClass()` method adds the `species` object as a new attribute that will contain the class of the instances.

Finally, the data should be added to the `Instances` object. This example will only classify one instance at a time, so a single instance, stored in the array of double values, is added to the `Instances` object through the `add()` method. The example adds an anonymous `Instance` object that is created inline. The first argument to the `Instance` constructor is the weight of this instance. The weight may be necessary if a weighted dataset is to be used for training. On classification tasks, the weight is irrelevant. So it is set to 1. The second and final argument to the constructor is the double array containing the values of the measurements. The class of the instance must be set to missing, using the `setClassMissing()` method to `Instance` object.

With the classifier and instance prepared and ready, the classification process is provided by two potential classification methods of the `Classifier` object. One, `classifyInstance()` returns a double representing the class of an object, either true or false, numerically. The second is `distributionForInstance()`, which returns an array of doubles, representing the likelihood of the instance being a member of each class in a multi-class classifier. Because this is a multi-class classifier, the example uses `distributionForInstance()`, which is called on the instance within the `Instances` object at index 0. With the distribution stored in a new double array, the classification is selected by finding the distribution with the highest value and determining what species that represents, returned as a `String` object.

Implementation specifics

The implementation of the classifier included herein is designed for demonstration. As such, it operates on a standard Java object type (`Dictionary`) and returns the classification in a simplistic form: a `String` object. Two drivers are provided. The class `IrisDriver` provides a command-line interface to the classifier with the feature set specified on the command line with the name followed by an equal sign and the value. In addition, a JUnit regression test is provided that looks at six combinations of iris measurements to classify them correctly. Specific examples known to predict correctly with this classifier were used. Both drivers, however, provide an opportunity to examine how one of these processes can operate in real time. These iris measurements were created at random based on the original training measurements.

However, there is no reason the `Iris` object must expect a `Dictionary` object. Since it includes a translation process as part of the classification method, the object containing the item to be classified can be any structure convenient to the implementation or the programmer, provided the internal structure of the object to be classified can be recreated from the storage form. If speed is a concern, a caller can operate with the `Classifier` object directly and pass it values directly. However, the architecture of the caller will suffer from reduced abstraction, making it harder to use different models from within Weka, or to use a different classification engine, entirely. Finding the right balance between abstraction and speed is difficult across many problem domains. This application is no exception and abstraction was selected for demonstration purposes.

Summary

This article has provided an overview of the Weka classification engine and shows the steps to take to create a simple classifier for programmatic use. This example can be refined and deployed to an OLTP environment for real-time classification if the OLTP environment supports Java technology. It can also be used offline over historical data to test patterns and mark instances for future processing. The basic example's abstraction can be reduced in favor of speed if the final application calls for it. With the information included, it is possible to create a solid classifier and make any necessary changes to fit the final application.

Downloads

Description	Name	Size
iris dataset to demonstrate RandomTree classifier	Irises.zip	16KB

Resources

Learn

- Learn more about the Weka data mining software in the Java language.
- Find resources about the simple file format ARFF.
- Data mining is the talk of the tech industry, as companies are generating millions of data points about their users and looking for a way to turn that information into increased revenue. Check out Data mining with Weka.
- Weka requests that all publications about it cite the paper titled "The Weka Data Mining Software: An Update," by Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten.
- Read the details about ARFF, so you can load your data into Weka.
- Learn more about big data in the developerWorks big data content area. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Find resources to help you get started with InfoSphere BigInsights, IBM's Hadoop-based offering that extends the value of open source Hadoop with features like Big SQL, text analytics, and BigSheets.
- Follow these self-paced tutorials (PDF) to learn how to manage your big data environment, import data for analysis, analyze data with BigSheets, develop your first big data application, develop Big SQL queries to analyze big data, and create an extractor to derive insights from text documents with InfoSphere BigInsights.
- Find resources to help you get started with InfoSphere Streams, IBM's high-performance computing platform that enables user-developed applications to rapidly ingest, analyze, and correlate information as it arrives from thousands of real-time sources.
- Stay current with developerWorks technical events and webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

- Get the iris dataset from Wikipedia. Note that it can also be downloaded from this article.
- Download InfoSphere BigInsights Quick Start Edition, available as a native software installation or as a VMware image.
- Download InfoSphere Streams, available as a native software installation or as a VMware image.
- Use InfoSphere Streams on IBM SmartCloud Enterprise.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Ask questions and get answers in the InfoSphere BigInsights forum.
- Ask questions and get answers in the InfoSphere Streams forum.
- Check out the developerWorks blogs and get involved in the developerWorks community.
- Check out IBM big data and analytics on Facebook.

About the author

James Howard

James Howard is a statistical analysis expert who provides in-depth economic, policy, and data analysis to federal agencies, public institutions, and private clients. Currently, he is a senior associate at Kore Federal and provides independent consulting as J.P. Howard & Co. Previously, he worked for the Board of Governors of the Federal Reserve System as an internal consultant on statistical computing.