Data mining with WEKA, Part 3: Nearest Neighbor and server-side library

Michael Abernethy

June 08, 2010

Data mining can be used to turn seemingly meaningless data into useful information with rules, trends, and inferences that can be used to improve your business and revenue. This article covers the last common data mining technique, "Nearest Neighbor," and will show you how to use the WEKA Java™ library in your server-side code to integrate data mining technology into your Web applications.

View more content in this series

Introduction

In the previous two articles in this "Data mining with WEKA" series, I introduced the concept of data mining. If you have not yet read Part 1 and Part 2, please read them first because they cover key concepts you should know before moving forward. More importantly, I talked about three techniques used in data mining that can turn your confusing and useless data into meaningful rules and trends. The first of these was regression, which can be used to predict a numerical output (like house value) based on other example data. The second was classification (also known as classification tree or decision tree), which can be used to create an actual branching tree to predict the output value of an unknown data point. (In our example, we predicted the response to a BMW promotion.) Third, I introduced clustering, which can be used to create groups (clusters) of data from which you can identify trends and other rules (BMW sales in our example). These things all were similar in that they could transform your data into useful information, but each did it differently and with different data, which is one of the important aspects of data mining: The right model has to be used on the right data.

This last article will discuss the last of the four common data mining techniques: Nearest Neighbor. You'll see that it is like a combination of classification and clustering, and provides another useful weapon for our mission to destroy data misinformation.

In our previous articles, we use WEKA as a stand-alone application. How useful would that be in the real world? It's not ideal, obviously. Since WEKA is a Java-based application, it has a Java library you can use in our own server-side code. This will likely be the more common use for most people, as you can write code to constantly analyze your data and make adjustments on the fly, rather than rely on someone to extract the data, convert to a WEKA format, then run it through the WEKA Explorer.

Nearest Neighbor

Nearest Neighbor (also known as Collaborative Filtering or Instance-based Learning) is a useful data mining technique that allows you to use your past data instances, with known output values, to predict an unknown output value of a new data instance. So, at this point, this description should sound similar to both regression and classification. How is this different from those two? Well, first off, remember that regression can only be used for numerical outputs. That differentiates it from Nearest Neighbor immediately. Classification, as we saw from the example in the previous article, uses *every data instance* to create a tree, which we would traverse to find our answer. This can be a serious problem with some data. Think about a company like Amazon and the common "Customers who purchased X also purchased Y" feature. If Amazon were to create a classification tree, how many branches and nodes could it have? There are maybe a few hundred thousand products. How big would that tree be? How accurate do you think a tree that big would be? Even if you got to a single branch, you might be shocked to learn that it only has three products. Amazon's page likes to have 12 products on it to recommend to you. It's a bad data mining model for this data.

You'll find that Nearest Neighbor fixes all those problems in a very efficient manner, especially in the example used above for Amazon. It's not limited to any number of comparisons. It's as scalable for a 20-customer database as it is for a 20 million-customer database, and you can define the number of results you want to find. Seems like a great technique! It really is — and probably will be the most useful for anyone reading this who has an e-commerce store.

Let's delve into the math a little bit, so you can understand the process and also see some of the limitations of this technique.

Math behind Nearest Neighbor

You will see that the math behind the Nearest Neighbor technique is a lot like the math involved with the clustering technique. Taking the unknown data point, the distance between the unknown data point and every known data point needs to be computed. Finding the distance is really quite trivial with a spreadsheet, and a high-powered computer can zip through these calculations nearly instantly. The easiest and most common distance calculation is the "Normalized Euclidian Distance." It sounds much more complicated than it really is. Let's take a look at an example in action and try to figure out what Customer No. 5 is likely to purchase.

Listing 1. Nearest Neighbor math

```
Customer     Age     Income     Purchased Product
1            45       46k       Book
2            39       100k      TV
3            35       38k       DVD
4            69       150k      Car Cover
5            58       51k       ???

Step 1:  Determine Distance Formula
Distance = SQRT( ((58 - Age)/(69-35))^2) + ((51000 - Income)/(150000-38000))^2 )

Step 2:  Calculate the Score
Customer     Score     Purchased Product
1            .385         Book
2            .710         TV
3            .686         DVD
4            .941         Car Cover
5            0.0          ???
```

To answer the question "What is Customer No. 5 most likely to buy?" based on the Nearest Neighbor algorithm we ran through above, the answer would be a book. This is because the distance between Customer No. 5 and Customer No. 1 is less (far less, actually) than the distance between Customer No. 5 and any other customer. Based on this model, we say that the customer most like Customer No. 5 can predict the behavior of Customer No. 5.

However, the positives of Nearest Neighbor don't end there. The Nearest Neighbor algorithm can be expanded beyond the closest match to include any number of closest matches. These are termed "N-Nearest Neighbors" (for example, 3-Nearest Neighbors). Using the above example, if we want to know the two most likely products to be purchased by Customer No. 5, we would conclude that they are books *and* a DVD. Using the Amazon example from above, if they wanted to know the 12 products most likely to be purchased by a customer, they would want to run a 12-Nearest Neighbor algorithm (though Amazon actually runs something more complicated than just a simple 12-Nearest Neighbor algorithm).

Further, the algorithm shouldn't be constrained to predicting a product to be purchased. It can also be used to predict a Yes/No output value. Considering the above example, if we changed the last column to the following (from customers 1-4), "Yes,No,Yes,No," a 1-Nearest Neighbor model would predict Customer No. 5 to say "Yes" and a 2-Nearest Neighbor would predict a "Yes" (both customer nos. 1 and 3 say "Yes"), and a 3-Nearest Neighbor model would say "Yes." (Customer nos. 1 and 3 say "Yes," customer No. 2 says "No," so the average value of these is "Yes.")

The final question to consider is "How many neighbors should we use in our model?" Ah — not everything can be easy. You'll find that experimentation will be needed to determine the best number of neighbors to use. Also, if you are trying to predict the output of a column with a 0 or 1 value, you'd obviously want to select an odd number of neighbors, in order to break ties.

Data set for WEKA

The data set we'll use for our Nearest Neighbor example should look familiar — it's the same data set we used for our classification example in the previous article. It's about our fictional BMW

dealership and the promotional campaign to sell a two-year extended warranty to past customers. To review the data set, here are the specifics I introduced in the last article.

There are 4,500 data points from past sales of extended warranties. The attributes in the data set are Income Bracket [0=$0-$30k, 1=$31k-$40k, 2=$41k-$60k, 3=$61k-$75k, 4=$76k-$100k, 5=$101k-$150k, 6=$151k-$500k, 7=$501k+], the year/month their first BMW was bought, the year/month the most recent BMW was bought, and whether they responded to the extended warranty offer in the past.

Listing 2. Nearest Neighbor WEKA data

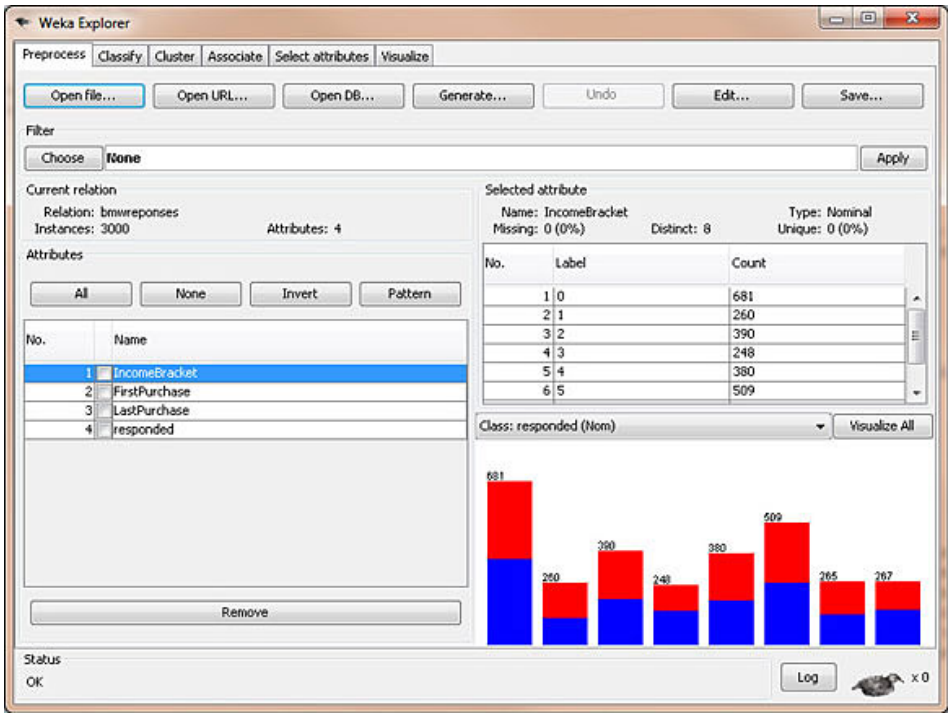
```
@attribute IncomeBracket {0,1,2,3,4,5,6,7}
@attribute FirstPurchase numeric
@attribute LastPurchase numeric
@attribute responded {1,0}

@data

4,200210,200601,0
5,200301,200601,1
...
```

Nearest Neighbor in WEKA

Why are we using the same data set we used in the classification example? Because, if you remember the results of that model, it was only 59-percent accurate, which wasn't acceptable at all (barely better than guessing). We're going to improve it and give this fictional dealership some useful information.

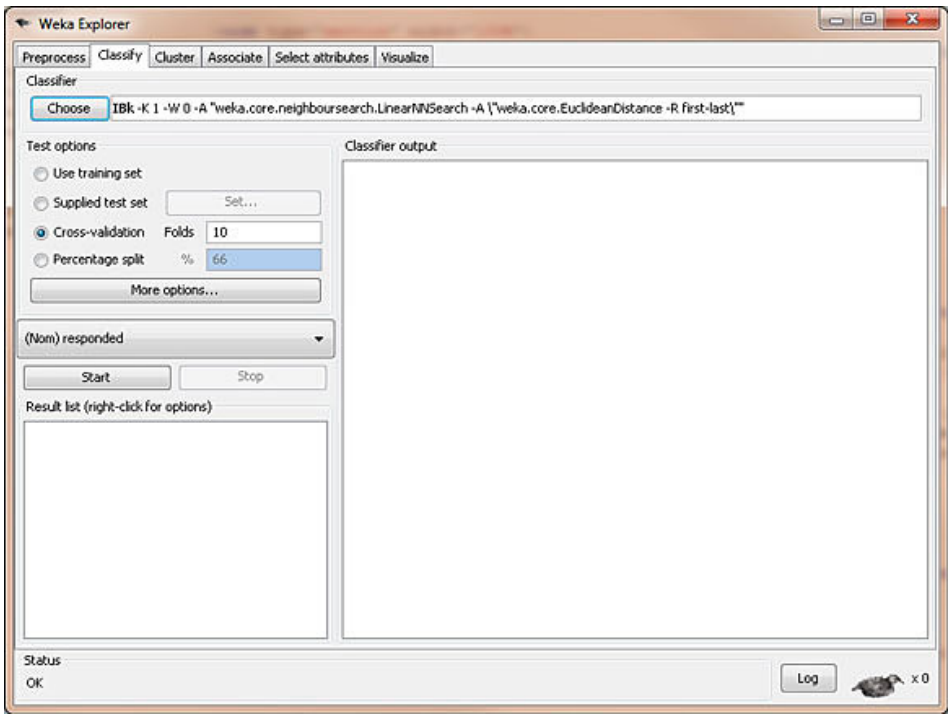
Load the data file bmw-training.arff into WEKA using the same steps we've used to this point in the Preprocess tab. Your screen should look like Figure 1 after loading in the data.

Figure 1. BMW Nearest Neighbor data in WEKA



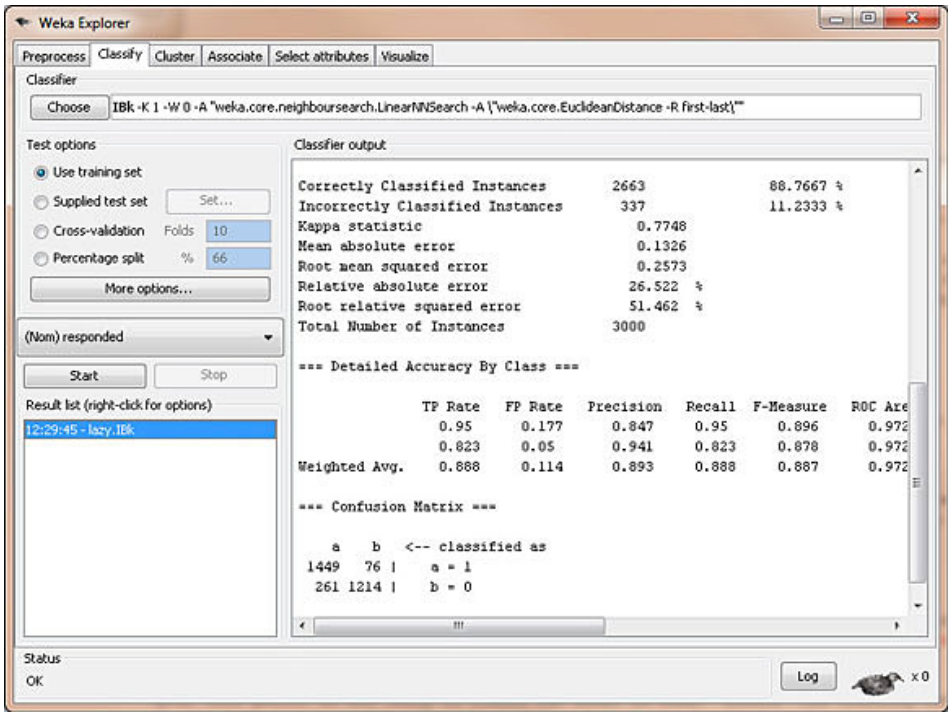
Like we did with the regression and classification model in the previous articles, we should next select the Classify tab. On this tab, we should select **lazy**, then select **IBk** (the *IB* stands for Instance-Based, and the *k* allows us to specify the number of neighbors to examine).

Figure 2. BMW Nearest Neighbor algorithm



At this point, we are ready to create our model in WEKA. Ensure that **Use training set** is selected so we use the data set we just loaded to create our model. Click **Start** and let WEKA run. Figure 3 shows a screenshot, and Listing 3 contains the output from this model.

Figure 3. BMW Nearest Neighbor model



Listing 3. Output of IBk calculations

```
=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances        2663               88.7667 %
Incorrectly Classified Instances       337               11.2333 %
Kappa statistic                          0.7748
Mean absolute error                      0.1326
Root mean squared error                  0.2573
Relative absolute error                 26.522  %
Root relative squared error             51.462  %
Total Number of Instances             3000

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.95      0.177      0.847     0.95      0.896      0.972    1
                 0.823     0.05       0.941     0.823     0.878      0.972    0
Weighted Avg.    0.888     0.114      0.893     0.888     0.887      0.972

=== Confusion Matrix ===

    a    b   <-- classified as
 1449   76 |    a = 1
 261 1214 |    b = 0
```

How does this compare with our results when we used classification to create a model? Well, this model using Nearest Neighbor has an 89-percent accuracy rating, while the previous model only had a 59-percent accuracy rating, so that's definitely a good start. Nearly a 90-percent accuracy

rating would be very acceptable. Let's take this a step further and interpret the results in terms of false positives and false negatives, so you can see how the results from WEKA apply in a real business sense.

The results of the model say we have 76 false positives (2.5 percent), and we have 261 false negatives (8.7 percent). Remember a false positive, in this example, means that our model predicted the customer would buy an extended warranty and actually didn't, and a false negative means that our model predicted they wouldn't buy an extended warranty, and they actually did. Let's estimate that the flier the dealership sends out cost $3 each and that the extended warranty brings in $400 profit for the dealer. This model from a cost/benefit perspective to the dealership would be $400 - (2.5% * $3) - (8.7% * 400) = $365. So, the model looks rather profitable for the dealership. Compare that to the classification model, which had a cost/benefit of only $400 - (17.2% * $3) - (23.7% * $400) = $304, and you can see that using the right model offered a 20-percent increase in potential revenue for the dealership.

As an exercise for yourself, play with the number of nearest neighbors in the model (you do this by right-clicking on the text "IBk -K 1...." and you see a list of parameters). You can change the "KNN" (K-nearest neighbors) to be anything you want. You'll see in this example, that the accuracy of the model actually decreases with the inclusion of additional neighbors.

Some final take-aways from this model: The power of Nearest Neighbor becomes obvious when we talk about data sets like Amazon. With its 20 million users, the algorithm is very accurate, since there are likely many potentail customers in Amazon's database with similar buying habits to you. Thus, the nearest neighbor to yourself is likely very similar. This creates an accurate and effective model. Contrarily, the model breaks down quickly and becomes inaccurate when you have few data points for comparison. In the early stages of an online e-commerce store for example, when there are only 50 customers, a product recommendation feature will likely not be accurate at all, as the nearest neighbor may in fact be very distant from yourself.

The final challenge with the Nearest Neighbor technique is that it has the potential to be a computing-expensive algorithm. In Amazon's case, with 20 million customers, each customer must be calculated against the other 20 million customers to find the nearest neighbors. First, if your business has 20 million customers, that's not technically a problem because you're likely rolling in money. Second, these types of computations are ideal for the cloud in that they can offloaded to dozens of computers to be run simultaneously, with a final comparison done at the end. (Google's MapReduce for example.) Third, in practice, it wouldn't be necessary to compare *every* customer in Amazon's database to myself if I'm only purchasing a book. The assumption can be made that I can be compared to only other bookbuyers to find the best match, narrowing the potential neighbors to a fraction of the entire database.

Remember: Data mining models aren't always simple input-output mechanisms — the data must be examined to determine the right model to choose, the input can be managed to reduce computing time, and the output must be analyzed and accurate before you are ready to put a stamp of approval on the entire thing.

Further reading: If you're interested in learning additional things about the Nearest Neighbor algorithm, read up on the following terms: distance weighting, Hamming distance, Mahalanobis distance.

Using WEKA on the server

One of the coolest things about WEKA is that it is not only a stand-alone application but it also is a self-contained Java JAR file that you can throw into your server's lib folder and call from your own server-side code. Think about how many interesting and important things this can bring to your applications. You can add reports that take advantage of all the data mining techniques we've learned so far. You can create a "Product Recommendation" widget for your e-commerce stores similar to the one that Amazon has on its site (because there's no way you'd be able to do this on-demand for each customer, running it through the stand-alone application). The WEKA stand-alone application itself just calls the underlying WEKA Java API, so you've seen the API in action already. Now it's time to see how to integrate it into your own code.

In fact, you've already downloaded the WEKA API JAR; it's the same JAR file you've been invoking to start the WEKA Explorer. To get access to the code, point your Java environment to include this JAR file in the classpath. All the usual steps to using a third-party JAR file in your own code.

As you can imagine, the central building block in the WEKA API is going to be the data. Data mining revolves around the data and, of course, all the algorithms that we've learned about have revolved around the data. So let's see how to get our data into a format that the WEKA API can use. Let's start easy though, let's start with the data from the first article in the series about house values.

NOTE: I would warn you ahead of time that the WEKA API can be difficult to navigate at times. First and foremost, double-check the version of WEKA you're using and the version of the API you're browsing. The API has changed enough between releases that the code can be totally different. Also, while the API is complete, there aren't very many good examples to get started (but that's why you're reading this of course). I am using WEKA V3.6.

Listing 4 shows how the data is formatted to be consumed by WEKA.

Listing 4. Loading data into WEKA

```
// Define each attribute (or column), and give it a numerical column number
// Likely, a better design wouldn't require the column number, but
// would instead get it from the index in the container
Attribute a1 = new Attribute("houseSize", 0);
Attribute a2 = new Attribute("lotSize", 1);
Attribute a3 = new Attribute("bedrooms", 2);
Attribute a4 = new Attribute("granite", 3);
Attribute a5 = new Attribute("bathroom", 4);
Attribute a6 = new Attribute("sellingPrice", 5);

// Each element must be added to a FastVector, a custom
// container used in this version of Weka.
// Later versions of Weka corrected this mistake by only
// using an ArrayList
```

```
FastVector attrs = new FastVector();
attrs.addElement(a1);
attrs.addElement(a2);
attrs.addElement(a3);
attrs.addElement(a4);
attrs.addElement(a5);
attrs.addElement(a6);

// Each data instance needs to create an Instance class
// The constructor requires the number of columns that
// will be defined.  In this case, this is a good design,
// since you can pass in empty values where they exist.
Instance i1 = new Instance(6);
i1.setValue(a1, 3529);
i1.setValue(a2, 9191);
i1.setValue(a3, 6);
i1.setValue(a4, 0);
i1.setValue(a5, 0);
i1.setValue(a6, 205000);

....

// Each Instance has to be added to a larger container, the
// Instances class.  In the constructor for this class, you
// must give it a name, pass along the Attributes that
// are used in the data set, and the number of
// Instance objects to be added.  Again, probably not ideal design
// to require the number of objects to be added in the constructor,
// especially since you can specify 0 here, and then add Instance
// objects, and it will return the correct value later (so in
// other words, you should just pass in '0' here)
Instances dataset = new Instances("housePrices", attrs, 7);
dataset.add(i1);
dataset.add(i2);
dataset.add(i3);
dataset.add(i4);
dataset.add(i5);
dataset.add(i6);
dataset.add(i7);

// In the Instances class, we need to set the column that is
// the output (aka the dependent variable).  You should remember
// that some data mining methods are used to predict an output
// variable, and regression is one of them.
dataset.setClassIndex(dataset.numAttributes() - 1);
```

So now we have the data loaded into WEKA. That's probably a little harder than it should be, but you can see that it would be trivial and very beneficial to write your own wrapper classes to quickly extract data from a database and place it into a WEKA instances class. In fact, I highly recommend if you get involved in using WEKA on your server, you spend some time doing that, since working with data in this way is tedious. Once you get your data in the instances object, you are free to use whatever data mining you want on the data, so you want this step to be as easy as possible.

Let's put our data through the regression model and make sure the output matches the output we computed using the Weka Explorer. It's actually quite easy to put our data through the regression model using the WEKA API, far easier than actually loading the data.

Listing 5. Creating a regression model in WEKA

```
// Create the LinearRegression model, which is the data mining
// model we're using in this example
```

```
LinearRegression linearRegression = new LinearRegression();

// This method does the "magic", and will compute the regression
// model.  It takes the entire dataset we've defined to this point
// When this method completes, all our "data mining" will be complete
// and it is up to you to get information from the results
linearRegression.buildClassifier(dataset);

// We are most interested in the computed coefficients in our model,
// since those will be used to compute the output values from an
// unknown data instance.
double[] coef = linearRegression.coefficients();

// Using the values from my house (from the first article), we
// plug in the values and multiply them by the coefficients
// that the regression model created.  Note that we skipped
// coefficient[5] as that is 0, because it was the output
// variable from our training data
double myHouseValue = (coef[0] * 3198) +
                      (coef[1] * 9669) +
                      (coef[2] * 5) +
                      (coef[3] * 3) +
                      (coef[4] * 1) +
                      coef[6];

System.out.println(myHouseValue);
// outputs 219328.35717359098
// which matches the output from the earlier article
```

And that's it! Running a classification, clustering, or Nearest Neighbor isn't quite as easy as a regression model, but they aren't much harder. It's much easier to run the data mining model than to load the data into it.

Ideally, this little section should greatly interest you into looking how to integrate WEKA into your own server-side code. Whether you run an e-commerce shop and want to better recommend products to them, or you have a coupon promotion you want to improve, or you want to optimize your AdWords campaign, or you want to optimize your landing page, these data mining techniques can all improve your results in those areas. Taking advantage of the built-in nature of the WEKA API, you can go so far as to write server-side code to rotate your landing page and constantly analyze the results using data mining to find the most effective landing page. Combine that with data mining analysis on your AdWords, and you can quickly find the best route for getting customers to your site, and converting them into sales.

Conclusion

This article wraps up the three-article series introducing you to the concepts of data mining and especially to the WEKA software. As you've seen, WEKA can do many of the data mining tasks that were previously available only in commercial software packages. WEKA is powerful and 100-percent free to use. You can't beat a deal like that, since you can quickly get WEKA up and running and crunching your data in no time.

This article went over the fourth-common data mining algorithm, "Nearest Neighbor." This algorithm is ideal for finding data points that are close to an unknown data point and using the known output from those values to predict the output for the unknown. I showed how this situation is ideal for a situation you see every time you shop online, the Recommend Products section.

Through some data mining, sites like Amazon can quickly (for them at least, with its thousands of computers) tell you what customers like you purchased.

The final section of the article showed that you shouldn't be constrained to using WEKA with the Explorer window as a stand-alone application. WEKA can be used as a stand-alone Java library, which you can drop into your server-side environment and call its API like any other Java library. I showed you how you can load data into the WEKA API (and recommended you spend some time to write a nice wrapper around your database, to make this overly complex process easier). Finally, I showed you how easy it was to create a regression model and get the same results from the API that we got from the stand-alone application.

My final recommendation when working with the API is to read through the documentation and spend some time reading all the available functions offered. I find the API somewhat difficult to work with, so reading it thoroughly can be the difference from using it successfully to throwing into the recycle bin.

Hopefully, after reading this series, you will be inspired to download WEKA and try to find patterns and rules from your own data.

Downloadable resources

Description	Name	Size
Dealer information and Java code	os-weka3-Example.zip	17KB

Related topics

- 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.
- Check out the WEKA Web site for all the documentation and an FAQ about the software.
- Download WEKA to run it on your own system.
- Check out the WEKA Java API.
- Read the details about ARFF, so you can get load your data into WEKA.
- IBM has its own data mining software, and "Integrate InfoSphere Warehouse data mining with IBM Cognos reporting, Part 1" is a good starting point.