Consolidation Week: WEKA

Dr. Sharu Theresa Jose

University of Birmingham

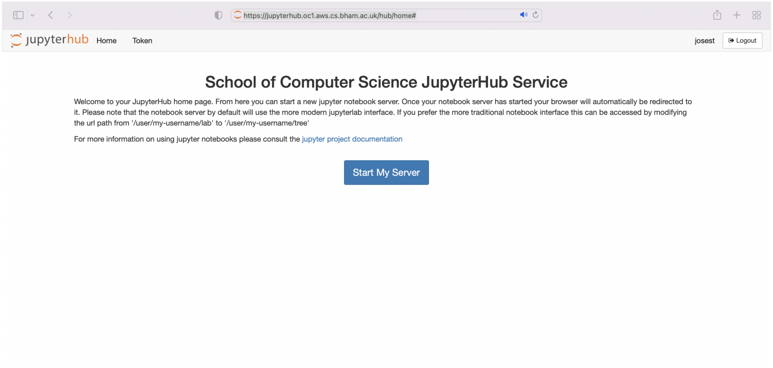
February 19, 2024

Learning Outcomes

- Learn how to run Weka in School's Jupyter server
- Familiarize with Java-based Weka
- Run supervised and unsupervised learning algorithms

Starting Weka in School's Jupyter Server

- Access the jupyter server through the following link: `https://jupyterhub.oc1.aws.cs.bham.ac.uk`
- This will take you to the page as shown below:



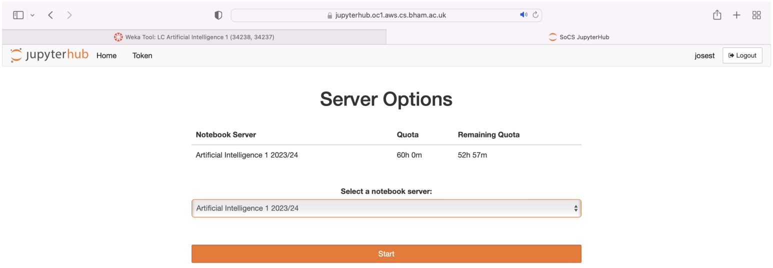
School of Computer Science JupyterHub Service

Welcome to your JupyterHub home page. From here you can start a new jupyter notebook server. Once your notebook server has started your browser will automatically be redirected to it. Please note that the notebook server by default will use the more modern jupyterlab interface. If you prefer the more traditional notebook interface this can be accessed by modifying the url path from '/user/my-username/lab' to '/user/my-username/tree'

For more information on using jupyter notebooks please consult the jupyter project documentation

- Click on Start My Server.

- You will have the following page open:



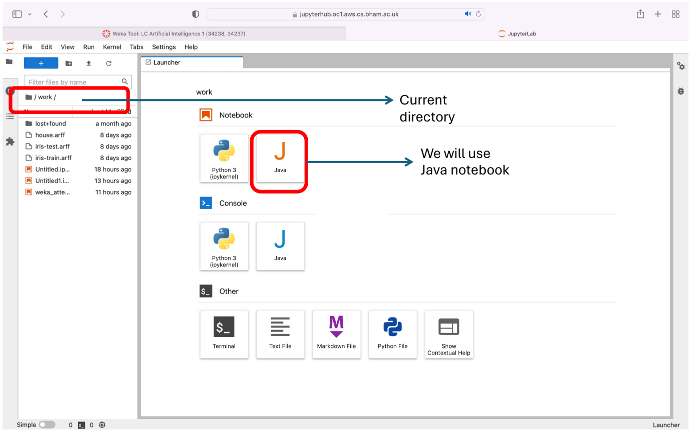
jupyterhub.oc1.aws.cs.bham.ac.uk

Server Options

Notebook Server	Quota	Remaining Quota
Artificial Intelligence 1 2023/24	60h 0m	52h 57m

Select a notebook server: Artificial Intelligence 1 2023/24

- Click on Start to get to the following page:



- The starting working directory in the notebook is `/jupyter/work` (this will appear as a breadcrumb trail near the top of the file explorer window on the left). This is the only directory backed by persistent storage so only files saved to this directory will persist between notebook server sessions.
- The weka data set is stored in `/jupyter/data`. Click on the breadcrumb trail links to navigate the directions.

Introduction to Weka in Java Notebook

- Version of Weka in Jupyter server: Weka 3.8.5
- We will be running Weka using Java-based code. Hence, a particular learning algorithm is encapsulated in a **class**, which may depend on other classes.
- Classes are organized into **packages**. Note that a package is just a directory containing a collection of related classes.
  - For example, `trees` package contains the classes that implement decision trees.
- Packages are organized in a hierarchy.
  - `trees` is a sub-package of the `classifiers` package, which itself is a sub-package of the overall `weka` package.
- To access details of weka packages, go to: `https://waikato.github.io/weka-wiki/documentation/` and click on `Javadoc for Weka 3.8`. The list of all weka packages can be found at `https://weka.sourceforge.io/doc.stable-3-8/`.

Points We Will Cover in this Lecture

- Loading Data
- Pre-processing via Filters
- Running classification and regression algorithms
- Running clustering algorithms

For more details, refer WekaManual (can be found in the canvas page).

Loading Data

Weka enables one to load data from files (in various formats) and also from databases. The following classes are used to **store data in memory**:

1. `weka.core.Instances`: Holds a complete dataset.
   - Row-based data structure
   - Single rows can be accessed via the *instance* method using a 0-based index.
2. `weka.core.Instance`: Encapsulates a single row.
3. `weka.core.Attribute`: Hold the type information about a single column in the dataset. It stores the type of the attribute, as well as the labels for nominal attributes, the possible values for string attributes or the datasets for relational attributes.

Loading Data From Files

When loading data from files, either let Weka choose from available loaders (in `weka.core.converters` package) or use the correct loader directly.

- `DataSource` class (inner class of the `weka.core.converters.ConverterUtils`) can be used to **read** data from **files that have the appropriate file extension**.
- Example code snippet:

```
import weka.core.converters.ConverterUtils.DataSource;
import weka.core.Instances;
...
Instances data = DataSource.read("/some/where/dataset.arff");
Instances data1 = DataSource.read("/some/where/dataset.csv");
Instances data2 = DataSource.read("/some/where/dataset.xrff");
```

- Recall that to run supervised learning algorithms, we must set the class attribute.
- To do this, use `setClassIndex(int)` method.
- Example code snippet:

```
// uses the first attribute as class attribute
if (data.classIndex() == -1)
    data.setClassIndex(0); ...
// uses the last attribute as class attribute
if (data.classIndex() == -1)
    data.setClassIndex(data.numAttributes() - 1);
```

Pre-processing via Filters

- In Weka, filters are used to preprocess data. They can be found inside the package `weka.filters`. Each filter falls into one of the following two categories:
  - *supervised*: The filter requires a class attribute to be set
  - *unsupervised*: A class attribute is not required to be present.
- And into one of the two sub-categories:
  - *attribute-based*: Columns are processed, e.g., added or removed
  - *instance-based*: Rows are prcoessed, e.g., added or removed.

Unsupervised attribute filter: Remove first attribute

We will see how to remove first attribute from a dataset. To do this,

- Use `Remove` filter located in the package `weka.filters.unsupervised.attribute`.
- For setting options, the `setOptions(String[])` method is used.
- Example code:

```
import weka.core.Instances;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Remove;
...
String[] options = new String[2];
options[0] = "-R";  // "range"
options[1] = "1"; // first attribute
Remove remove = new Remove(); // new instance of filter
remove.setOptions(options); // set options
remove.setInputFormat(data); // inform filter about dataset
                            // **After** setting options
Instances newData = Filter.useFilter(data, remove); // apply filter
```

Unsupervised attribute filter: Standardizing Dataset

- Standardization transforms all numeric attributes to have zero mean and unit variance.
- Use `Standardize` filter (package `weka.filters.unsupervised.attribute`)
- Example code snippet:

```
Instances train = ... // from somewhere
Standardize filter = new Standardize();
// initializing the filter once with training set
filter.setInputFormat(train);
// configures the Filter based on train instances
//and returns filtered instances
Instances newTrain = Filter.useFilter(train, filter);
```

Unsupervised attribute filter: StringToWordVector

- Use `StringToWordVector` filter (package `weka.filters.unsupervised.attribute`)
- Example code snippet:

```
import weka.filters.unsupervised.attribute.StringToWordVector;
Instances train = ... // from somewhere
StringToWordVector filter = new StringToWordVector();
// initializing the filter once with training set
filter.setInputFormat(train);
// configures the Filter based on train instances
//and returns filtered instances
Instances newTrain = Filter.useFilter(train, filter);
```

Classification

Classification and regression algorithms in Weka are called "classifiers", and are located below the `weka.classifiers` package. We will now look into the following three aspects:

- Building a classifier
- Evaluating a classifier
- Classifying instances

Building a Classifier

- Classifiers in Weka are either batch-trainable, i.e, they get trained on the whole data set, or are incremental, i.e., using each instance incrementally.
- To build a batch classifier,
  - Set options - either using the `setOptions(String[])` method or the actual set-methods.
  - Train the model - calling the `buildClassifier(Instances)` method with the training set.
  - Example code snippet to build a J48 Decision Tree Classifier:

```
import weka.core.Instances;
import weka.classifiers.trees.J48;
...
Instances data = ... // from somewhere
String[] options = new String[1];
options[0] = "-U"; // unpruned tree
J48 tree = new J48(); // new instance of tree
tree.setOptions(options); // set the options
tree.buildClassifier(data); // build classifier
```

- Will skip incremental classifier here. (Details can be found in the references listed in the last slide)

Evaluating the Classifier

Weka supports two types of evaluation:

- Cross-validation
- Dedicated test set

The evaluation step, including collection of statistics, is performed by the `Evaluation` class (package `weka.classifiers`).

Cross-Validation

- `crossValidateModel` method of the `Evaluation` class is used to perform cross-validation with **an untrained classifier and a single dataset**.
- Before cross-validation is performed, the data gets **randomized** using the supplied random number generator (`java.util.Random`). It is recommended to 'seed' the random number generator with a specified seed value.
- Example code snippet for performing 10-fold cross validation with a J48 decision tree algorithm on a dataset `newData`, with random number generator seeded with "1".

```
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
import weka.core.Instances;
import java.util.Random;
...
Instances newData = ... // from somewhere
Evaluation eval = new Evaluation(newData);
J48 tree = new J48();
eval.crossValidateModel(tree, newData, 10, new Random(1));
System.out.println(eval.toSummaryString("\nResults\n\n", false));
```

Train/Test Set

- Use **dedicated test set** to evaluate a classifier.
- Provide a **trained classifier**
- Use `evaluateModel` method from `weka.classifiers.Evaluation` class.
- Example code snippet for training a J48 with default options on a training set and evaluating it on a test set before outputting the summary of the collected statistics:

```
import weka.core.Instances;
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
...
Instances train = ... // from somewhere
Instances test = ... // from somewhere
// train classifier
Classifier cls = new J48();
cls.buildClassifier(train);
// evaluate classifier and print some statistics
Evaluation eval = new Evaluation(train);
eval.evaluateModel(cls, test);
System.out.println(eval.toSummaryString("\nResults\n\n", false));
```

Output Statistics

To get output statistic summary, in the previous slide we used `toSummaryString`. Other summary methods include:

- `toMatrixString` : outputs the confusion matrix
- `toClassDetailsString` : outputs TP/FP rates, precision, recall, F-measure, AUC (per class).
- `toCumulativeMarginDistributionString`: outputs the cumulative margins distribution

Classifying an Instance Using Trained Classifier

A built classifier can be used to predict the label of an unlabelled data instance using `classifyInstance` method.

- Example code snippet where a trained classifier tree is used to label all the instances in an unlabelled dataset.

```
// load unlabeled data and set class attribute
Instances unlabeled = DataSource.read("/some/where/unlabeled.arff");
unlabeled.setClassIndex(unlabeled.numAttributes() - 1);
// create copy
Instances labeled = new Instances(unlabeled);
// label instances
for (int i = 0; i < unlabeled.numInstances(); i++) {
double clsLabel = tree.classifyInstance(unlabeled.instance(i));
labeled.instance(i).setClassValue(clsLabel); }
// save newly labeled data
DataSink.write("/some/where/labeled.arff", labeled);
```

Clustering Algorithms

Step 1: Building a (batch) clusterer. Happens in two stages:

- Set options - either calling the `setOptions(String[])` method or the appropriate set- methods of the properties
- Build the model calling the `buildClusterer(Instances)` method
- Example code snippet of building the EM clusterer with a maximum of 100 iterations.

```
import weka.clusterers.EM;
import weka.core.Instances;
...
Instances data = ... // from somewhere
String[] options = new String[2];
options[0] = "-I"; // max. iterations
options[1] = "100";
EM clusterer = new EM(); // new instance of clusterer
clusterer.setOptions(options); // set the options
clusterer.buildClusterer(data); // build the clusterer
```

Evaluating a Clusterer

Step 2: Evaluating the Clusterer

- The class used for evaluating cluster algorithms is `ClusterEvaluation` from package `weka.clusterers`.
- Example code snippet as follows:

```
import weka.clusterers.ClusterEvaluation;
import weka.clusterers.EM;
import weka.core.Instances;
...
Instances data = ... // from somewhere
EM cl = new EM();
cl.buildClusterer(data);
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(cl);
eval.evaluateClusterer(new Instances(data));
System.out.println(eval.clusterResultsToString());
```

Classes to Clusters Evaluation

If your data contains a class attribute and you want to check how well the generated clusters fit the classes, you can perform the classes to clusters evaluation.

This type of evaluation is performed as follows:

- Create a copy of the labelled dataset and remove the class attribute, using the `Remove` filter (under `weka.filters.unsupervised.attribute`). Example code snippet:

```
Instances data = ... // from somewhere
Remove filter = new Remove();
filter.setAttributeIndices("" + (data.classIndex() + 1));
filter.setInputFormat(data);
Instances dataClusterer = Filter.useFilter(data, filter);
```

- Build the clusterer with this new data

```
EM clusterer = new EM();// set further options for EM, if necessary...
clusterer.buildClusterer(dataClusterer);
```

- Evaluate the clusterer now with the **original** data.

```
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(clusterer);
eval.evaluateClusterer(data);
// print results
System.out.println(eval.clusterResultsToString());
```

- Finally, to cluster an unknown instance, use `clusterInstance(Instance)` method that determines the cluster the `Instance` would belong to.

Homework Exercises

Try running the Weka GUI examples studied in Week 5 on Jupyter server. Verify the results obtained match.

References

- Weka Manual, Chapter 18