Journal of
open research software

Beckham, C et al 2016 WekaPyScript: Classification, Regression, and Filter Schemes for WEKA Implemented in Python. *Journal of Open Research Software*, 4: e33, DOI: http://dx.doi.org/10.5334/jors.108

SOFTWARE METAPAPER

WekaPyScript: Classification, Regression, and Filter Schemes for WEKA Implemented in Python

Christopher Beckham[1], Mark Hall[2] and Eibe Frank[1]

[1] Department of Computer Science, The University of Waikato, Hamilton 3240, New Zealand

[2] Pentaho Corporation, Suite 340, 5950 Hazeltine National Dr., Orlando, FL 32822, USA

Corresponding author: Christopher Beckham (cjb60@students.waikato.ac.nz)

WekaPyScript is a package for the machine learning software WEKA that allows learning algorithms and preprocessing methods for classification and regression to be written in Python, as opposed to WEKA's implementation language, Java. This opens up WEKA to its machine learning and scientific computing ecosystem. Furthermore, due to Python's minimalist syntax, learning algorithms and preprocessing methods can be prototyped easily and utilised from within WEKA. WekaPyScript works by running a local Python server using the host's installation of Python; as a result, any libraries installed in the host installation can be leveraged when writing a script for WekaPyScript. Three example scripts (two learning algorithms and one preprocessing method) are presented.

Keywords: Python; WEKA; machine learning; data mining

(1) Overview

Introduction

WEKA [1] is a popular machine learning workbench written in Java that allows users to easily classify, process, and explore data. There are many ways WEKA can be used: through the WEKA Explorer, users can visualise data, train learning algorithms for classification and regression and examine performance metrics; in the WEKA Experimenter, datasets and algorithms can be compared in an automated fashion; or, it can simply be invoked on the terminal or used as an external library in a Java project.

Another machine learning library that is increasingly becoming popular is Scikit-Learn [2], which is written in Python. Part of what makes Python attractive is its ease of use, minimalist syntax, and interactive nature, which makes it an appealing language to learn for non-specialists. As a result of Scikit-Learn's popularity the wekaPython [3] package was released, which allows users to build Scikit-Learn classifiers from within WEKA. While this package makes it easy to access the host of algorithms that Scikit-Learn provides, it does not provide the capability of executing external custom-made Python scripts, which limits WEKA's ability to make use of other interesting Python libraries. For example, in the world of deep learning (currently a hot topic in machine learning), Python is widely used, with libraries or wrappers such as Theano [4], Lasagne [5], and Caffe [6]. The ability to create classifiers in Python would open up WEKA to popular deep learning implementations.

In this paper we present a WEKA classifier and a WEKA filter,[1] PyScriptClassifier and PyScriptFilter (under the umbrella "WekaPyScript"), that are able to call arbitrary Python scripts using the functionality provided by the wekaPython package. So long as the script conforms to what the WekaPyScript expects, virtually any kind of Python code can be called. We present three example scripts in this paper: one that re-implements WEKA's ZeroR classifier (i.e., simply predicts the majority class from the training data), one that makes use of Theano in order to train a linear regression model, and a simple filter that standardises numeric attributes in the data. Theano is a symbolic expression library that allows users to construct arbitrarily complicated functions and automatically compute the derivatives of them – this makes it trivial to implement classifiers such as logistic regression or feed-forward neural networks (according to Baydin et al. [7], the use of automatic differentiation in machine learning is scant).

In our research, we used this package to implement new loss functions for neural networks using Theano and compare them across datasets using the WEKA Experimenter.

Implementation and architecture

In this section, we explain how wekaPython is implemented and how WekaPyScript makes use of it to allow classifiers and filters to be implemented in Python.

wekaPython

WekaPyScript relies on a package for WEKA 3.7 called wekaPython [3]. This package provides a mechanism that allows the WEKA software, which is running in a Java JVM, to interact with CPython – the implementation of the

Python language written in C. Although there are versions of the Python language that can execute in a JVM, there is a growing collection of Python libraries for scientific computing that are backed by fast C or Fortran implementations, and these are not available when using a JVM-based version of Python.

In order to execute Python scripts that can access packages incorporating native code, the wekaPython package uses a micro-service architecture. The package starts a small server, written in Python, and then communicates with it over local sockets. The server implements a simple protocol that allows WEKA to transfer and receive datasets, invoke CPython scripts, and retrieve the values of variables set in Python. The format for transporting datasets to and from Python is comma-separated values (CSV). On the Python side, the fast CSV parsing routine from the pandas package [8] is used to convert the CSV data read from a socket into a data frame data structure. On the WEKA side, WEKA's CSVLoader class is used to convert CSV data sent back from Python.

The two primary goals of the wekaPython package are to: a) allow users of WEKA to execute arbitrary Python scripts in a Python console implemented in Java or as part of a data processing workflow; and (b) enable access to classification and regression schemes implemented in the Scikit-Learn [2] Python library. In the case of the former, users can write and execute scripts within a plug-in graphical environment that appears in WEKA's Explorer user interface, or by using a scripting step in WEKA's Knowledge Flow environment. In the case of the latter, the package provides a "wrapper" WEKA classifier implementation that executes Python scripts to run Scikit-Learn algorithms. Because the wrapper classifier implements WEKA's Classifier API, it works in the same way as a native WEKA classifier, which allows it to be processed by WEKA's evaluation routines and used in the Experimenter framework. Although the general scripting functionality provided by wekaPython allows users to write scripts that access machine learning libraries other than Scikit-Learn, they do not appear as a native classifier to WEKA and can not be evaluated in the same way as the Scikit-Learn wrapper. The goal of the WekaPyScript package described in this paper is to provide this functionality.

WekaPyScript

The new PyScriptClassifier and PyScriptFilter components contain various options such as the name of the Python script to execute and arguments to pass to the script when training or testing. The arguments are represented as a semicolon-separated list of variable assignments. All of WekaPyScript's options are described below in **Table 1**. **Figures 1** and **2** show the GUI in the WEKA Explorer for PyScript Classifier and PyScriptFilter, respectively.

When PyScriptClassifier/PyScriptFilter is invoked, it will utilise wekaPython to start up a Python server on localhost and construct a dictionary called `args`, which contains either the training or the testing data (depending on the context) and meta-data such as the attribute names and their types. This meta-data is described in **Table 2**.

This `args` dictionary can be augmented with extra arguments by using the `-args` option and passing a semicolon-separated list of variable assignments. For instance, if `-args` is `alpha=0.01;reg='l2'` then the dictionary `args` will have a variable called `alpha` (with value 0.01) and a variable `reg` (with value `'l2'`) and these will be available for access at both training and testing time.[2]

Given some Python script, PyScriptClassifier will execute the following block of Python code to train the model:

```
import imp
cls = imp.load_source('train', <name of python
      script>)
model = cls.train(args)
```

In other words, it will try and call a function in the specified Python script called `train`, passing it the `args` object, and this function should return (in some form) something that can be used to reinstantiate the model. When the resulting WEKA model is saved to disk (e.g., through the command line or the WEKA Explorer) it is the `model` variable that gets serialised (thanks to wekaPython's ability to receive variables from the Python VM). If the

Option	Description
`-cmd` (pythonCommand)	Name of the Python executable
`-script` (pythonFile)	Path to the Python script
`-args` (arguments)	Semicolon-separated list of arguments (variable assignments) to pass to the script when training or testing
`-binarize` (shouldBinarize)*	Should nominal attributes be converted to binary ones?
`-impute` (shouldImpute)*	Should missing values be imputed (with mean imputation)?
`-standardize` (shouldStandardize)*	Should attributes be standardised? (If imputation is set then this is done after it)
`-stdout` (printStdOut)	Print any stdout from Python script?
`-save` (saveScript)	Save the script in the model? (E.g., do not dynamically load the script specified by `-script` at testing time)
`-ignore-class` (ignoreClass)**	Ignore class attribute? (See Table 2 for more information.)

Table 1: Options for PyScriptClassifier and PyScriptFilter (* = applicable only to PyScriptClassifier, ** = applicable only to PyScriptFilter). (Note that the names in parentheses are the names of the options as shown in the Explorer GUI, as opposed to the terminal).

Variable(s)	Description	Type
`X_train`, `y_train`	Data matrix and label vector for training data. If `-ignore-class` is set or the class attribute is not specified, `y_train` will not exist and will instead be inside `X_train` as an extra column	`numpy.ndarray (float 64)`, `numpy.ndarray (int 64)`
`X, y*`	Data matrix and label vector for data, when PyScriptFilter calls the `process` method (see Listing 2)	`numpy.ndarray (float 64)`, `numpy.ndarray (int 64)`
`X_test`	Data matrix for testing data	`numpy.ndarray (float 64)`
`relation_name`	Relation name of ARFF	`str`
`class_type`	Type of class attribute (e.g., numeric, nominal)	`str`
`num_classes`	Number of classes	`int`
`attributes`	Names of attributes	`list`
`class`	Name of class attribute	`str`
`attr_values`	Dictionary mapping nominal/string attributes to their values	`dict`
`attr_types`	Dictionary mapping attribute names to their types (possible values are either `nominal` or `numeric`	`dict`

Table 2: Data and meta-data variables passed into `args` (* = only applicable to PyScriptFilter).

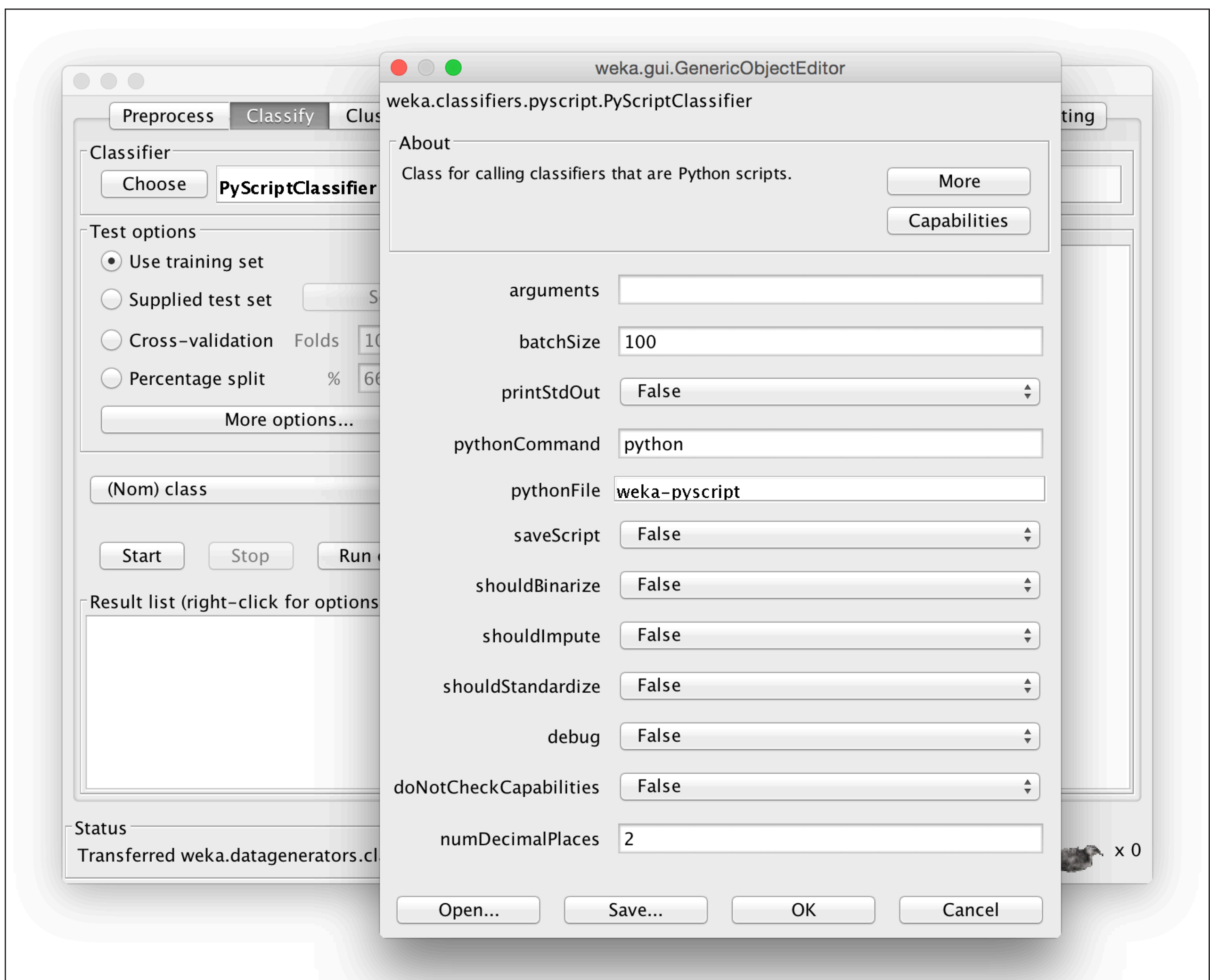

Figure 1: The graphical user interface for PyScriptClassifier.

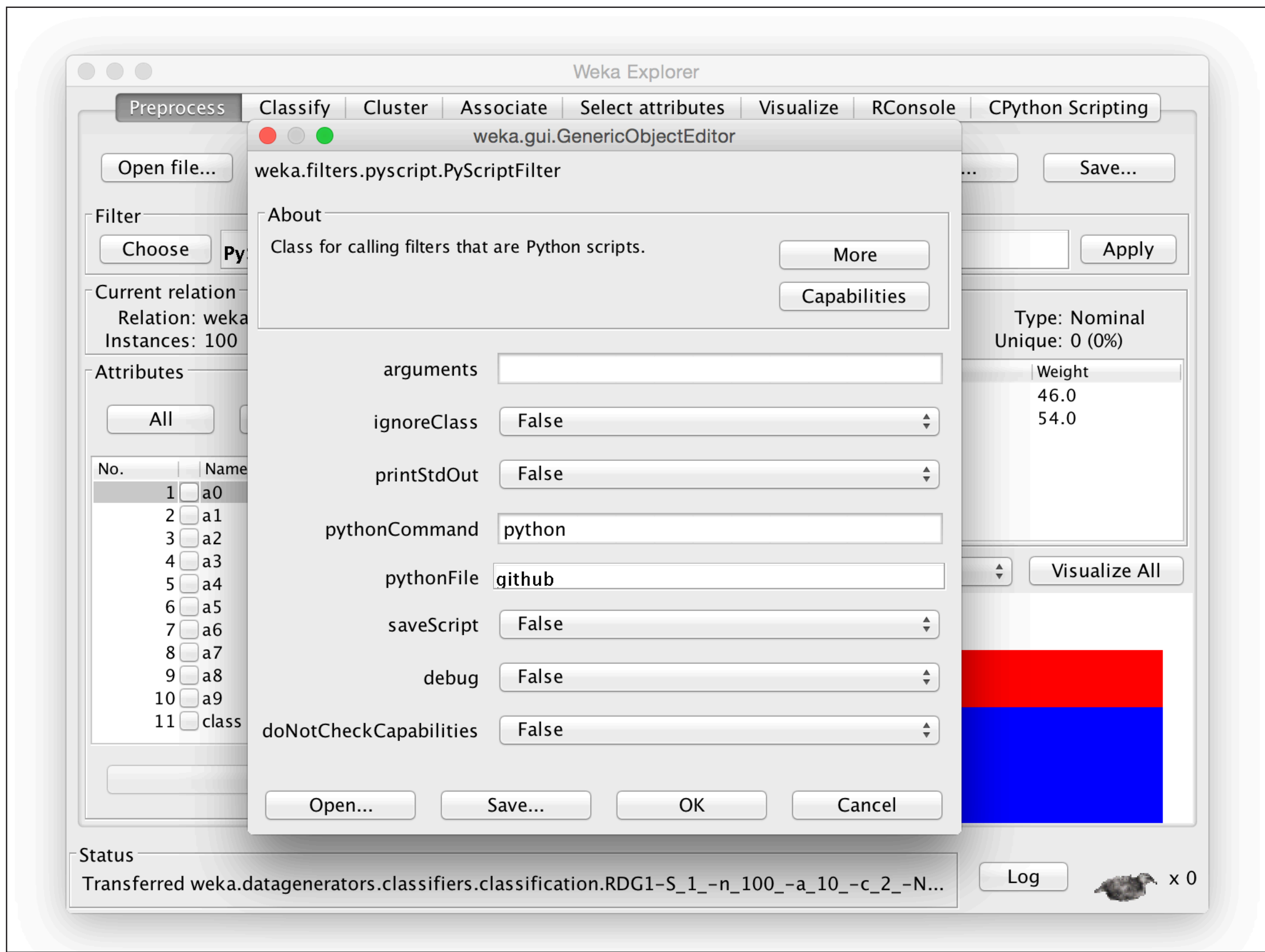

Figure 2: The graphical user interface for PyScriptFilter.

`-save` flag is set, the WEKA model will internally store the Python script so that at testing time the script specified by `-script` is not needed – this is not ideal however if the script is going to be changed frequently in the future.

When PyScriptClassier needs to evaluate the model on test data, it deserialises the model, sends it back into the Python VM, and runs the following code for testing:

```
cls = imp.load_source('test', <name of python
      script>)
preds = cls.test(args, model)
```

In this example, `test` is a function that takes a variable called `model` in addition to `args`. This additional variable is the model that was previously returned by the `train` function. The `test` function returns an $n \times k$ Python list (i.e., not a NumPy array) in the case of classification (where n_i is the probability distribution for k classes for the i'th test instance), and an n-long Python list in the case of regression.

To get a textual representation of the model, users must also write a function called `describe` which takes two arguments – the `args` object as described earlier, and the model itself – and returns some textual representation of the model (i.e. a string). This function is used as follows:

```
cls = imp.load_source('describe', <name of
      python script>)
model_description = cls.describe(args, model)
```

From the information described so far, the basic skeleton of a Python script implementing a classifier will look like what is shown in **Listing 1**.

PyScript Filter also has a train function that works in the same way.[3] Unlike a `test` function however, there is a `process(args, model)` function, which is applied to both the training and testing data. This function returns a modified version of the `args` object (this is because filters may change the structure, i.e., attributes, and contents of the data):

```
cls = imp.load_source('process', <name of python
      script>)
new_args = cls.process(args, model)
```

```
def train(args):
  # code for training model
def test(args, model):
  # code for running model on new instances
def describe(args, model):
  # textual representation of model
```

Listing 1: Skeleton of a Python script for PyScriptClassifier.

This new args object is then automatically converted back into WEKA's internal ARFF file representation, which then can be input into another filter or classifier.

The skeleton of a Python filter is shown in **Listing 2**.

Example use

In this section we present three examples: a classification algorithm that simply predicts the majority class in the training data; an excerpt of a linear regressor that uses automatic differentiation; and a filter that standardises numeric attributes in the data.

ZeroR

The first example we present is one that re-implements WEKA's ZeroR classifier, which simply finds the majority class in the training set and uses that for all predictions (see **Listing 3**).

In the `train` function we simply count all the classes in `y_train` and return the index (starting from zero) of the majority class, m (lines 5–7). So for this particular script, the index of the majority class is the "model" that is returned. In line 15 of the `test` function, we convert the majority class index into a (onehot-encoded) probability distribution by indexing into a $k \times k$ identity matrix, and in line 16, return this vector for all n test instances (i.e., it returns an $n \times k$ array, where n is the number of test instances and k is the number of classes; $n_{im} = 1$ and the other entries in n_i are zero).

Here is an example use of this classifier from a terminal session (assuming it is run from the root directory of the WekaPyScript package, which includes `zeror.py` in its `scripts` directory and `iris.arff` in the `datasets` directory)[4]:

```
java weka.Run .PyScriptClassifier \
  -cmd python \
  -script scripts/zeror.py \
  -t datasets/iris.arff \
  -no-cv
```

This example is run on the entire training set (i.e., no cross-validation is performed) since the standard `-no-cv` flag for WEKA is supplied. We have also used `-cmd` to tell WekaPyScript where the Python executable is located (in our case, it is located in the `PATH` variable so we only have to specify the executable name rather than the full path). If `-cmd` is not specified, then WekaPyScript will assume that the value is `python`. The output of this command is shown below in **Listing 4**.

Linear regression

We now present an example that uses Theano's automatic differentiation capability to train a linear regression classifier. We do not discuss the full script and instead present the gist of the example. To introduce some notation, let $x = \{x^{(1)}, x^{(2)}, \ldots, x^{(n)}\}$ be the training examples, where $x^{(i)} \in \mathbb{R}^p$, and $y = \{y^{(1)}, y^{(2)}, \ldots, y^{(n)}\}$ where $y^{(i)} \in \mathbb{R}$. Then, the sum-of-squares loss is simply:

$$L(w, b) = \frac{1}{n}\sum_{i=1}^{n}[(wx^{(i)} + b) - y^{(i)}]^2 \qquad (1)$$

where $w \in \mathbb{R}^p$ is the vector of coefficients for the linear regression model and $b \in \mathbb{R}$ is the intercept term. To fit a model, i.e., find w and b such that $L(w, b)$ is minimised,

```
def train(args):
  # code for training filter
def process(args, model):
  # code for processing instances(training or testing)
```

Listing 2: Skeleton of a Python script for PyScriptFilter.

```
from collections import Counter
import numpy as np

def train(args):
       y_train = args["y_train"].flatten()
       counter = Counter(y_train)
       return counter.most_common()[0][0]

def describe(args, model):
       return "Majority class: %i" % model

def test(args, model):
       num_classes = args["num_classes"]
       n = args["X_test"].shape[0]
       majority_cls = np.eye(num_classes)[model].tolist()
       return [majority_cls for x in range(0, n)]
```

Listing 3: Python implementation of ZeroR.

```
Options: -script scripts/zeror.py

Majority class: 0

Time taken to build model: 2.54 seconds
Time taken to test model on training data: 0.02 seconds

=== Error on training data ===

Correctly Classified Instances            50             33.3333 %
Incorrectly Classified Instances         100             66.6667 %
Kappa statistic                            0
Mean absolute error                        0.4444
Root mean squared error                    0.6667
Relative absolute error                  100      %
Root relative squared error              141.4214 %
Coverage of cases (0.95 level)            33.3333 %
Mean rel. region size (0.95 level)        33.3333 %
Total Number of Instances                150

=== Detailed Accuracy By Class ===

TP Rate        FP Rate  Precision  Recall  F-Measure   MCC     ...
1.000          1.000    0.333      1.000   0.500       0.000   ...
0.000          0.000    0.000      0.000   0.000       0.000   ...
0.000          0.000    0.000      0.000   0.000       0.000   ...
Weighted Avg.  0.333    0.333      0.111   0.333       0.167   ...

=== Confusion Matrix ===

a   b   c    <-- classified as
50  0   0  |  a = Iris-setosa
50  0   0  |  b = Iris-versicolor
50  0   0  |  c = Iris-virginica
```

Listing 4: Output from `zeror.py` script.

we can use gradient descent and iteratively update w and b:

$$w := w - \alpha \frac{\partial}{\partial w} L(w, b) \tag{2}$$

$$b := b - \alpha \frac{\partial}{\partial b} L(w, b) \tag{3}$$

We repeat above until we reach a maximum number of epochs (i.e., scans through the training data) or until we reach convergence (with some epsilon, ϵ). Fortunately, we do not need to manually compute the partial derivatives because Theano can do this for us. **Listing 5** illustrates this.

In this code, which we would place into the `train` function of the script for PyScriptClassifier, we define our parameters w and b in lines 7–9, initialising w and b to zeros. In lines 12–13, we define our symbolic matrices $x \in \mathbb{R}^{n \times p}$ and $y \in \mathbb{R}^{n \times 1}$, and in line 15, the output function $h(x) = wx+b$, where $h(x) \in \mathbb{R}^{n \times 1}$. In line 18, we finally compute the loss function in Equation 1 and in lines 20–21 we compute the gradients $\frac{\partial}{\partial w}L(w, b)$ and $\frac{\partial}{\partial b}L(w, b)$. We define our learning rate α in line 23 and in line 24, we define the parameter updates as described in Equations 2 and 3. Finally, in line 26 we define the `iter_train` function: given some x and y (which can be the entire training set, or a mini-batch, or a single example), it will output the loss (Equation 1) and automatically update the parameters as per Equations 2 and 3.

We can run this example from a terminal session by executing:

```
java weka.Run .PyScriptClassifier \
  -script scripts/linear-reg.py \
  -args "alpha=0.1; epsilon=0.00001" \
  -standardize \
  -t datasets/diabetes_numeric.arff \
  -no-cv
```

In this example we have used the `-standardize` flag to perform zero-mean unit-variance normalisation on all the numeric attributes. Also note that we did not have to explicitly specify an alpha and epsilon since the script has default values for these – this was done just to illustrate how arguments work. The output of this script is shown below in **Listing 6**.

Because we created a textual representation of the model with the `describe` function, we get the equation of the linear classifier in the output.

Standardise filter

Lastly, we present an example filter script that standardises all numeric attributes by subtracting the mean and dividing by the standard deviation. This is shown in **Listing 7**.

```
import theano
from theano import tensor as T
import numpy as np

# assume 5 attributes for this example
num_attributes = 5
w = theano.shared(
    np.zeros((num_attributes, 1)), name ='w')
b = theano.shared(0.0, name ='b')

# let x be a n*p matrix, and y be a n*1 matrix
x = T.dmatrix('x')
y = T.dmatrix('y')
# prediction is simply xw + b
out = T.dot(x, w) + b

# loss function is mean squared error
loss = T.mean((out - y)**2)
# compute gradient of cost w.r.t. w and b
g_w = T.grad(cost = loss, wrt = w)
g_b = T.grad(cost = loss, wrt = b)

alpha = 0.01
updates = [(w, w - alpha * g_w), (b, b - alpha * g_b)]

iter_train = theano.function(
    [x, y], outputs=loss, updates=updates)
```

Listing 5: Optimising sum-of-squares loss in Theano.

```
Options: -script scripts/linear-reg.py...

f(x)=
   age *0.266773099848 +
   deficit *0.289990210412 +
   4.74354333559

Time taken to build model: 8.49 seconds
Time taken to test model on training data: 1.18 seconds

=== Error on training data ===

Correlation coefficient                  0.607
Mean absolute error                      0.448
Root mean squared error                  0.5659
Relative absolute error                 82.3838 %
Root relative squared error             79.4711 %
Coverage of cases (0.95 level)           0      %
Mean rel.region size (0.95 level)        0      %
Total Number of Instances               43
```

Listing 6: Output from `linear-reg.py` script.

In lines 11–18, we iterate through all attributes in the dataset and store the means and standard deviations for the numeric attributes. The "model" that we return in this script is a tuple of two lists (the means and standard deviations). In lines 26–28, we perform the standardisation. From there, we return the args object (which has changed due to the modification of `X`).

We can run this example on the diabetes dataset:

```
java weka.Run .PyScriptFilter \
  -script scripts/standardise.py \
  -i datasets/diabetes_numeric.arff \
  -c last
```

The output of this script is the transformed dataset. An excerpt of this, in WEKA's ARFF data format, is shown in **Listing 8**.

```
from wekapyscript import \
     ArffToArgs, get_header, instance_to_string
import numpy as np

def train(args):
    X_train = args["X_train"]
    means = []
    sds = []
    attr_types = args["attr_types"]
    attributes = args["attributes"]
    for i in range(0, X_train.shape[1]):
        if attr_types[attributes[i]] == "numeric":
            means.append(np.nanmean(X_train[:,i]))
            sds.append(
                np.nanstd(X_train[:,i],ddof=1))
        else:
            means.append(None)
            sds.append(None)
    return (means, sds)

def process(args, model):
    X = args["X"]
    attr_types = args["attr_types"]
    attributes = args["attributes"]
    means, sds = model
    for i in range(0, X.shape[1]):
        if attr_types[attributes[i]]  == "numeric":
           X[:,i] = (X[:,i] - means[i]) / sds[i]
    return args
```

Listing 7: Standardise filter in Python.

```
@relation diabetes_numeric—weka.filters.pyscript.PyScriptFilter ...

@attribute age numeric
@attribute deficit numeric
@attribute c_peptide numeric

@data

—0.952771, 0.006856, 4.8
—0.057814, —1.116253, 4.1
0.364805, 1.017655, 5.2
0.389665, 0.048973, 5.5
0.339945, —2.927268, 5
...
```

Listing 8: Output from `standardise.py` script.

Because we have set the class attribute using `-c`, standardisation is not applied to it. However, if we wish to also apply standardisation to it, we can add the flag `-ignore-class` to the end of the command.

Note that we are not limited to just modifying the existing data inside `args` – we can do more complex things such as adding new attributes, and an example of this is shown in the Gaussian noise filter (`add-gauss-noise.py`) located in the `scripts` folder.

Quality control

WekaPyScript contains a collection of unit tests written for the JUnit framework. WEKA also contains an abstract test class (that includes regression tests) for classifiers and filters to implement, which we have used to ensure that WekaPyScript performs correctly with other algorithms in WEKA.

(2) Availability

Operating system

WekaPyScript has been tested on OS X Yosemite, Ubuntu 14.04.1, and Windows 10.

Programming language

Java 7+, CPython 2.7 or 3.4.

Dependencies

WEKA 3.7.13, the wekaPython package for WEKA, and Python. Python packages NumPy, Pandas, Matplotlib and Scikit-Learn are also required (the easiest way to install

these is through Anaconda).[5] To run the linear regression example, Theano should also be installed.[6]

Software location

Archive

- **Name:** WekaPyScript
- **Identifier:** http://dx.doi.org/10.5281/zenodo.50198
- **License:** GPLv3
- **Publisher:** Zenodo
- **Date published:** 04/22/2016

Code repository

- **Name:** Github
- **URL:** http://github.com/christopher-beckham/weka-pyscript
- **License:** GPLv3
- **Current version:** 0.5.0
- **Date published:** 04/22/2016

Documentation

- **Mailing list:** https://groups.google.com/forum/#!forum/weka-pyscript
- **Installation instructions:** https://github.com/christopher-beckham/weka-pyscript/blob/master/README.md
- **Wiki:** https://github.com/christopher-beckham/weka-pyscript/wiki

(3) Reuse potential

PyScriptClassifier and PyScriptFilter can be used to implement filters and classifiers without requiring the end user to know Python. For example, we created LasagneNet,[7] a classifier that wraps the deep neural network framework Lasagne. Users can easily define network architectures within WEKA, and once trained, the WekaLasagne classifier will use PyScript Classifier to generate the necessary Python code in order to train the neural network. This enables users not familiar with Python or Lasagne to train a variety of neural networks within WEKA which are relatively fast to train, thanks to the latter's utilisation of native linear algebra libraries. Within the machine learning community, researchers can use WekaPyScript to compare Python implementations of learning algorithms with ones in WEKA (or in R)[8], by using the WEKA Experimenter tool.

Author roles

- Christopher Beckham (`cjb60 at students dots waikato dot ac dot nz`)
  - Department of Computer Science, The University of Waikato, Hamilton 3240, New Zealand
  - Wrote software (WekaPyScript), co-wrote manuscript
- Mark Hall (`mhall at pentaho dot com`)
  - Pentaho Corporation, Suite 340, 5950 Hazeltine National Dr., Orlando, FL 32822, USA
  - Wrote software (wekaPython), co-wrote manuscript, software feedback and testing
- Eibe Frank (`eibe at waikato dot ac dot nz`)
  - Department of Computer Science, The University of Waikato, Hamilton 3240, New Zealand
  - Project supervisor, manuscript and software feedback and testing

Competing Interests

The authors declare that they have no competing interests.

Notes

1. Filters are used to preprocess the data before being provided to a learning algorithm. Examples of filter applications include binarisation of nominal attributes, standardisation of data, and discretisation. In the context of WEKA, a "classifier" refers to either a classification or regression learning algorithm. Additional information can be found at https://weka.wikispaces.com/Primer.
2. Since `args` is a list of Python variable assignments separated by semicolons, something like `"a=[1,2,3,4,5];b=abs(-2)"` is valid because it will result in the assignments `args['a'] = [1,2,3,4,5]` and `args['b'] = abs(-2)`, which are syntactically valid Python statements.
3. "Training" may be a confusing term depending on the filter. For example, a filter that randomly adds noise to values in the data need not be "trained", but a supervised filter (such as a discretisation algorithm) will. In the case of the former, we can simply perform no operation in this method and return anything, such as an empty string.
4. This assumes that `weka.jar` is located in the `CLASSPATH` variable. For more information, see the `README` located in the root of the package directory.
5. https://www.continuum.io/downloads
6. https://github.com/Theano/Theano
7. http://www.github.com/christopher-beckham/weka-lasagne
8. R algorithms can be called from WEKA using the RPlugin package: http://weka.sourceforge.net/packageMetaData/RPlugin/index.html

References

1. **Hall, M, Frank, E, Holmes, G, Pfahringer, B, Reutemann, R** and **Witten, I H** 2009 (November) The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1): 10–18. DOI: http://dx.doi.org/10.1145/1656274.1656278
2. **Pedregosa, F, Varoquaux, G, Gramfort, A, Michel, V, Thirion, B, Grisel, O, Blondel, M, Prettenhofer, P, Weiss, R, Dubourg, V, Vanderplas, J, Passos, A, Cournapeau, D, Brucher, M, Perrot, M** and **Duchesnay, E** 2011 Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830.
3. **Hall, M** 2015 wekaPython: Integration with CPython for WEKA. Available at: http://weka.sourceforge.net/packageMetaData/wekaPython/index.html.
4. **Bastien, F, Lamblin, P, Pascanu, R, Bergstra, J, Goodfellow, I J, Bergeron, A, Bouchard, N** and **Bengio, Y** 2012 Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

5. **Dieleman, S, Schlüter, J, Raffel, C, Olson, E, Sønderby, S K** and **Contributors** 2015 (August) Lasagne: First release. DOI: http://dx.doi.org/10.5281/zenodo.27878
6. **Jia, Y, Shelhamer, E, Donahue, J, Karayev, S, Long, J, Girshick, R, Guadarrama, S** and **Darrell, T** 2014 Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.
7. **Baydin, A G, Pearlmutter, B A** and **Radul, A A** 2015 Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767.
8. **McKinney, W** 2010 Data structures for statistical computing in Python. In: van der Walt, S and Millman, J (Eds.) *Proceedings of the 9th Python in Science Conference*, pp. 51–56.

How to cite this article: Beckham, C, Hall, M and Frank, E 2016 WekaPyScript: Classification, Regression, and Filter Schemes for WEKA Implemented in Python. *Journal of Open Research Software,* 4: e33, DOI: http://dx.doi.org/10.5334/jors.108

Submitted: 09 December 2015 **Accepted:** 01 July 2016 **Published:** 08 August 2016

Copyright: © 2016 The Author(s). This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC-BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited. See http://creativecommons.org/licenses/by/4.0/.