Working Paper Series
ISSN 1177-777X

CLASSIFICATION AND REGRESSION ALGORITHMS FOR WEKA IMPLEMENTED IN PYTHON

Christopher J. Beckham

Working Paper: 02/2015
October 2015

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Classification and Regression Algorithms for WEKA Implemented in Python

Christopher J. Beckham

October 6, 2015

1 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 classifiers 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 command-line 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 ScikitLearnClassifier [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 (in the form of a package) that is able to call arbitrary Python scripts. So long as the script conforms to what the classifier expects, virtually any kind of Python code can be called.[1] 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 one that trains a convolutional neural network on MNIST digits using a library that abstracts Theano. 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.

An example application of the new Python script classifier in WEKA would be implementing new loss functions using Theano and comparing them using

[1] Note the term "classifier" is used in WEKA to refer to both classification and regression schemes.

the WEKA Experimenter and, in fact, this was actually one of the motivations for writing this package.

The package can be downloaded on Github [7], with instructions on how to install it and the relevant libraries.

2 Representation

Once installed, the classifier is located in the Java package `pyscript.PyScriptClassifier` in WEKA and contains various options such as the name of the Python script and arguments to pass to the script when training or testing. The arguments are represented as a semicolon-separated list of variable assignments; depending on the Python script that is called, it may require arguments to be passed, or none at all (more on this later). All of the classifier's options are described below in Table 1. Figure 1 also shows the GUI for the classifier in the WEKA Explorer.

Table 1: Parameters for PyScriptClassifier

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?

When PyScriptClassifier is invoked, it will 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.

Table 2: Data and meta-data variables passed into `args`

Variable(s)	Description	Type
X_train, y_train	Data matrix and label vector for training data	numpy.ndarray (float64), numpy.ndarray (int64)
X_test	Data matrix for testing data	numpy.ndarray (float64)
relation_name	Relation name of ARFF	string
class_type	Type of class attribute (e.g. numeric, nominal)	string
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	dict

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.

When PyScriptClassifier 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:

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

[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.

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:

```
import imp
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.

Listing 1: Skeleton of a Python script

```
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
```

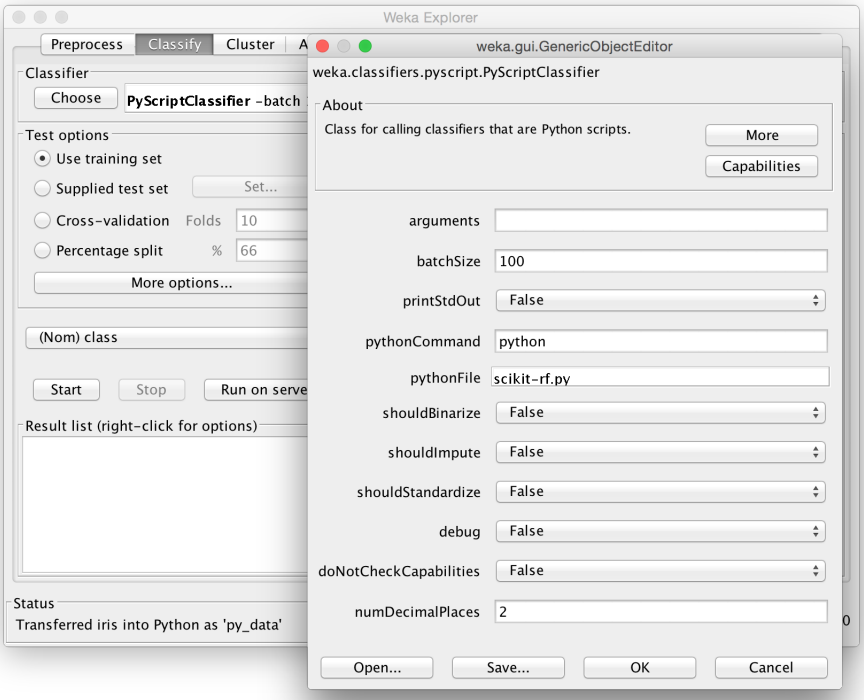

Figure 1: PyScriptClassifier within WEKA Explorer

3 Examples

We now present three examples: one that uses ZeroR (i.e., simply predict the majority class), one that implements linear regression, and one that trains a convolutional neural network on MNIST digits. [3]

3.1 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. 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. So for this particular script, the index of the majority class is the "model" that is returned. The `test` function returns an $n \times k$ array (where n is the number of test instances and k is the number of classes), where $n_{im} = 1$ and the other entries in n_i are zero.

[3]The latter example, MNIST, currently only works on Python 2.7

Listing 2: Python implementation of ZeroR

```
from collections import Counter
import numpy as np

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

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

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

Here is an example use of this classifier from the terminal (assuming it is run from the root directory of the PyScriptClassifier package, which includes `zeror.py` in its `scripts` directory):

```
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 `-no-cv` flag is supplied.

3.2 Linear regression

The next example (Listing 3) uses Theano to train a linear regression model using batch gradient descent. This example is only for illustrative purposes – the main idea is that we are able to symbolically define our parameters w and b, the loss function $\frac{1}{N}\sum_{i=1}^{N}((wx^{(i)}+b)-y^{(i)})^2$, automatically compute the gradients with respect to w and b, and then update the parameters in each loop (epoch) of gradient descent.

The `train` function can take two arguments: `alpha` (the learning rate), and `epsilon` (an early stopping criterion). It then returns a list consisting of the coefficients of the model and the intercept (which we will collectively call the "weights"). The `describe` function uses the weights of the linear model and the names of the attributes to construct a neat textual representation of the output. `test` takes the weights of the linear model, creates a prediction function, and then evaluates the test instances `X_test`, which are inside `args`.

Listing 3: Python implementation of linear regression using Theano

```
import theano
from theano import tensor as T
import numpy as np
import gzip
import cPickle as pickle

def train(args):

    X_train = args["X_train"]
    y_train = args["y_train"]

    # let w be a p*1 vector, and b be the intercept
    num_attributes = X_train.shape[1]
    w = theano.shared( np.zeros( (num_attributes, 1) ), name='w')
    b = theano.shared( 1.0, 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

    # cost function is mean squared error
    num_instances = X_train.shape[0]
    cost = (T.sum((out - y)**2)) / num_instances
    # compute gradient of cost w.r.t. w and b
    g_w = T.grad(cost=cost, wrt=w)
    g_b = T.grad(cost=cost, wrt=b)

    alpha = 0.01 if "alpha" not in args else args["alpha"]
    epsilon = 1e-6 if "epsilon" not in args else args["epsilon"]

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

    train = theano.function([x, y], outputs=cost, updates=updates)

    prev_loss = train(X_train, y_train)
    for epoch in range(0, 100000):
        this_loss = train(X_train, y_train)
        print this_loss
        if abs(this_loss - prev_loss) < epsilon:
            break
        prev_loss = this_loss

    return [ w.get_value(), b.get_value() ]

def describe(args, weights):
    coefs = weights[0].flatten()
    intercept = weights[1]
    st = "f(x) = \n"
    for i in range(0, len(coefs)):
        st += "  " + args["attributes"][i] + "*" + str(coefs[i]) + " +\n"
    st += "  " + str(intercept)
    return st

def test(args, weights):
    X_test = args["X_test"]
    num_attributes = X_test.shape[1]

    w = theano.shared( np.zeros( (num_attributes, 1) ), name='w')
    b = theano.shared( 1.0, name='b' )
    w.set_value( weights[0] )
    b.set_value( weights[1] )

    x = T.dmatrix('x')
    out = T.dot(x, w) + b
    pred = theano.function([x], out)

    X_test = args["X_test"]
    return pred(X_test).tolist()
```

We can run this example on the command line by running:

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

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. We have also omitted `-cmd python` since `python` is the default Python executable.

Because we created a textual representation of the model with the `describe` function, we get a textual representation of the model in the output:

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

3.3 Deep neural network

In the last example, we train a deep convolutional neural network on an MNIST [8] database of handwritten digits (Figure 2). To do this, we use the "nolearn" library [9], which abstracts a neural network library called Lasagne [5] and provides an interface similar to Scikit-Learn. Note that this example is part of a separate Github repository that aims to provide interesting PyScriptClassifier examples. [10] For the sake of brevity, we will only explain the most interesting parts.

We do not include the full source code for `mnist-nolearn.py` (located in the `mnist` folder) in this document. However, it is easy to define neural network architectures with nolearn. For example, if we wanted to train a basic neural net that took MNIST digits (i.e., 28 x 28 images), with a hidden layer consisting of 100 units, it could be done as shown in Listing 4.

Listing 4: Initialising a basic neural network in nolearn

```
...
net = NeuralNet(
    layers = [
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
        ('output', layers.DenseLayer),
    ],
    input_shape = (None, 1, 28, 28),
    hidden_num_units = 100,
    output_nonlinearity = softmax,
    output_num_units = 10,
    update=nesterov_momentum,
    update_learning_rate=0.01,
    update_momentum=0.9,
    verbose=1,
    max_epochs=10
)
...
X, y = load_data()
net.fit(X,y) # train network
```

In `mnist-nolearn.py` we train a convolutional neural network instead, which is a very popular type of neural network in image recognition and deep learning. Because nolearn is based on Lasagne (which, in turn is based on Theano), this network can be trained using a GPU, giving WEKA the (indirect) ability to train deep neural networks very efficiently.

Figure 2: MNIST digits [11]

In this example, we have a folder that contains 10,000 MNIST digits as .gif images. For the sake of efficiency, rather than store the pixel values for these digits in an ARFF file, we simply store their relative locations and their class values. This can be seen in the file `mnist.meta.arff` in the `mnist` folder from the PyScriptExamples repository, and an excerpt is shown in Listing 5:

Listing 5: mnist.meta.arff

```
@relation mnist
@attribute filename string
@attribute class {0,1,2,3,4,5,6,7,8,9}
@data
0_c7.gif,7
1000_c9.gif,9
1001_c0.gif,0
1002_c2.gif,2
1003_c5.gif,5
...
```

When we run this example through PyScriptClassifier and use `mnist-nolearn.py`, it will get the filenames, and load these from a specified directory in an incremental fashion to train a convolutional neural network using mini-batch stochastic gradient descent.

To train the network (with no cross-validation), we run:

```
java weka.Run .PyScriptClassifier \
    -script mnist-nolearn.py \
    -args "'dir'='data'" \
    -t mnist.meta.arff -no-cv
```

This script takes an argument called `dir`, which specifies where the images are located (the filenames in `mnist.meta.arff` are relative to this directory). The resulting output from this example is shown below in Listing 6. In this listing the textual representation of the model (i.e., the output of the `describe` function) is a summary of the network such as the number of learnable parameters, the architecture of the network, and training statistics after one epoch (which is the default number of epochs).

Listing 6: Output from training convolutional neural network on MNIST

```
Options: -cmd python -script mnist-nolearn.py -args 'dir'='data'

# Neural Network with 109295 learnable parameters

## Layer information

  #  name       size
---  ---------  --------
  0  l_in       1x28x28
  1  l_conv1    10x24x24
  2  l_pool1    10x12x12
  3  l_conv2    25x8x8
  4  l_pool2    25x4x4
  5  l_hidden   250
  6  l_out      10

  epoch    train loss    valid loss    train/val    valid acc  dur
-------  ------------  ------------  -----------  -----------  ------
      1       1.43549       0.65860      2.17962      0.80595  28.50s

Time taken to build model: 31.94 seconds
Time taken to test model on training data: 10.83 seconds

=== Error on training data ===

Correctly Classified Instances        8539               85.39   %
Incorrectly Classified Instances      1461               14.61   %
Kappa statistic                          0.8376
Mean absolute error                      0.05
Root mean squared error                  0.1502
Relative absolute error                 27.763  %
Root relative squared error             50.0786 %
Coverage of cases (0.95 level)          97.73   %
Mean rel. region size (0.95 level)      24.613  %
Total Number of Instances           10000

...
```

4 Miscellany

Having to run WEKA to debug a Python script is inconvenient. PyScriptClassifier provides a helper class called `ArffToPickle`, which will convert any ARFF file to a Python pickle (i.e., a serialised Python object) so that scripts can be

tested independently of WEKA if the user so desires. The class is invoked like so:

```
java weka.Run .ArffToPickle \
    -i <input arff> \
    -o <output pickle> \
    -c <class index> \
    [-impute] [-standardize] [-binarize] [-debug]
```

More conveniently, however, PyScriptClassifier comes with a Python module called `pyscript` which can be used to convert ARFF files to an `args` object within a Python script.

Listing 7 shows how to use the `ArffToArgs` class from the `pyscript` module to test out a classifier within Python. In this example we create an `args` object from the Iris dataset and call the `train` function, to emulate what WEKA does when it trains a classifier.

Listing 7: Testing a classifier independently of WEKA using the `pyscript` module

```
from pyscript.pyscript import ArffToArgs

def train(args):
    pass

def describe(args, model):
    pass

def test(args, model):
    pass

if __name__ == '__main__':
    f = ArffToArgs()
    f.set_input("datasets/iris.arff")
    args = f.get_args()
    f.close()
    # train model
    model = train(args)
```

Notice how we used the `__name__ == '__main__'` if statement to ensure that the block of code following this statement only gets executed when we invoke the script from the command-line – when we execute a script like this in WEKA, it will not be executed.

Serialised PyScriptClassifier models maintain a reference to a Python script on the file system, and if the script that the model is referring to is moved then it will no longer work. To remedy this issue, PyScriptClassifier comes with another helper class called `ChangeScriptPath` that can deserialise the model and change the path name of the Python script inside. This helper class is invoked as follows:

```
java weka.Run .ChangeScriptPath \
    -i <input model> \
    -o <output model> \
    -script <new path to script>
```

Listing 8 shows an example usage of this.

Listing 8: Example usage of helper class `ChangeScriptPath`

```
# create a zeror model from iris.arff and save the model to
# /tmp/zeror.model
java weka.Run .PyScriptClassifier \
    -script scripts/zeror.py \
    -t datasets/iris.arff -no-cv -d /tmp/zeror.model

# use the model to predict on the same dataset (iris.arff)
java weka.Run .PyScriptClassifier \
    -l /tmp/zeror.model \
    -T datasets/iris.arff

# make a copy of the script somewhere else
cp scripts/zeror.py /tmp/zeror.py

# make a new model that references /tmp/zeror.py instead of
# scripts/zeror.py
java weka.Run .ChangeScriptPath \
    -i /tmp/zeror.model -o /tmp/zeror_2.model \
    -script "/tmp/zeror.py"

# use new model to predict iris.arff
java weka.Run .PyScriptClassifier \
    -l /tmp/zeror_2.model \
    -T datasets/iris.arff
```

To ensure that users pass the sensible arguments to a script, the `train()` function can be decorated using the `uses` decorator. This decorator takes a list of strings that are valid arguments to the script. An example is shown in Listing 9.

Listing 9: Example usage of the `uses` decorator

```
@uses(["foo","bar"])
def train(args):
  foo = args["foo"]
  bar = args["bar"]
  ...
def describe(args, model):
  ...
def test(args, model):
  ...
```

If an argument is passed that is not `foo` or `bar` then an exception will be thrown by PyScriptClassifier.

Lastly, the `output-debug-info` flag may prove useful when issues occur with a Python script – this will print to standard error all the commands that get executed on the Python VM (if WEKA is run from the command-line) or in the error log (in WEKA's Explorer). Likewise, the `-stdout` flag can be useful as it prints any standard out from the invoked Python script.

5 Conclusion

In this paper, we have presented PyScriptClassifier, a generic classifier that is able to call any Python script so long as it conforms to the required structure. This opens up WEKA to many exciting new possibilities, such as quickly prototyping classifiers or training state-of-the-art deep neural networks. We have presented three examples: a majority-class classifier in less than 15 lines of code,

a linear regressor trained using automatic differentiation, and a convolutional neural network trained on the MNIST dataset. Future work entails allowing WEKA filters to be written in Python as well.

6 Acknowledgements

Mark Hall for development of the wekaPython package (which this package depends on) and fixes/changes to WEKA (to facilitate smooth operation of PyScriptClassifier), and Eibe Frank for changes/fixes to WEKA and proof-reading of this paper.

References

[1] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[3] Mark Hall. wekaPython: Integration with CPython for WEKA. `http://weka.sourceforge.net/packageMetaData/wekaPython/index.html`, 2015.

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[5] Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, and contributors. Lasagne: First release. `http://dx.doi.org/10.5281/zenodo.27878`, August 2015.

[6] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[7] Christopher Beckham. Github: PyScriptClassifier package for WEKA. `http://github.com/chrispy645/weka-pyscript`, 2015.

[8] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[9] Daniel Nouri. Github: nolearn – Abstractions around neural net libraries, most notably Lasagne. `http://github.com/dnouri/nolearn`, 2015.

[10] Christopher Beckham. Github: PyScriptClassifier examples. `http://github.com/chrispy645/weka-pyscript-examples`, 2015.

[11] Leif Johnson. Theanets 0.7.0pre documentation, 2015. File: `mnist-digits-small.png`.