Program 1 - Perceptron

Logan Grosz

Last updated: October 30, 2020

Contents

1 Usage

1.1 Installing the module from source

In the directory with `setup.py`

```
pip install .
```

1.2 Usage

Then anywhere...

```
import <classifier> from myml
```

```
import myml.util as util
```

For examples see `examples/`

2 API Definitions

2.1 myml.util

`plot_decision_regions(X, y, classifier, resolution)` :: Will print a 2D scattery plot along with a line that separates two regions in accordance with the perceptron passed in.

`plot_decision_regions_3d(X, y, classifier, resolution)` :: Will print a 3D scattery plot along with a plane that separates two regions in accordance with the perceptron passed in.

`plot_linear_regression` :: Will print a 2D scattery plot along with a plane that separates two regions in accordance with the perceptron passed in. For any higher dimensions it will not print. (For final portfolio, this will do a 3D plot as well)

2.2 myml.Perceptron

2.2.1 Description

The perceptron follows the data fitting algorithm presented on Wikipedia closely. It initializes weights to 0. For each sample, it calculates the actual output,and then changes the weight to reflect the difference between this output and the desired output. This process repeats until the specified number of iterations is reached or there is an iteration error rate of 0. In order to preserve data, both the number of errors and weight at each step is recorded. These can be accessed through *Perceptron*`.weights` and *Perceptron*`.errors`.

2.2.2 Usage

```
pn = Perceptron(rate, epochs)
pn.fit(X, y)
pn.predict(Z)
```

`X` is a numpy.ndarray of shape ($samples$, $features$)

`y` is a numpy.ndarray of shape ($samples$)

2.2.3 Attributes

Perceptron`.rate` :: The learning rate of the classifier

Perceptron`.epochs` :: The number of epochs to run when fitting. These are not guaranteed to run if the classifier can be fit with no error at any point.

Perceptron`.errors` :: Errors at each epoch during fitting. This is a numpy.ndarray with shape (*epochs*, $features$).

Perceptron`.weights` :: Weights at each epoch during fitting. This is a numpy.ndarray with shape (*epochs*, $features$).

2.2.4 Methods

Perceptron`.__init__` :: The initilizer takes 2 arguments: the first is the learning rate, a float between 0 and 1. The second is the max number of iterations. These values can be accessed again with *Perceptron*`.rate` and *Perceptron*`.niter`.

Perceptron`.fit` :: Takes arguments `X` (numpy array with shape (*nSamples*, *nFeatures*)) and `d` (also a numpy array with shape (*nSamples*)). This function applies the steps described above and "validates" the errors and weights arrays.

Perceptron`.net_input` :: Returns the weighted values of some given input `X` (of type numpy array with shape (*nSamples, nFeatures*)) as a numpy array with shape (*nSamples*).

Perceptron`.predict` :: Returns labels (-1 or 1) for a given `X` of type numpy array with shape (*nSamples, nFeatures*) as a numpy array of shape (*nSamples*).

`Perceptron.f` :: a static method which returns the label given `w` (a numpy array of shape (*nFeatures*+1)), and `x`, a sample with shape (*nFeatures*). Note, they're not the same size but that's because `w[0]` acts as a bias.

2.3 myml.LinearRegressor

2.3.1 Description

The linear regressor fits data assumed to be linear in nature. It uses a gradient descent algorithm on the squared error between expected and actual output values. This can be done with any number of independant variables. It starts assuming the weights, $\beta = [0, ..., 0]$. It then calulates partial derivatives of the error term J, for each independant variable in X. J, the cost, is defined as the sum of the squared error for each sample point: $\sum_{i=1}^{n}(y_i - (\beta \cdot \mathbf{X_i^T}))^2$. Using the partial derivatives and the error rate, a new β is determined for the next epoch. This is repeated until the given number of epochs.

2.3.2 Usage

```
lr = LinearRegressor(rate, epochs)
lr.fit(X, y)
lr.predict(Z)
```

`X` is a numpy.ndarray of shape (n, p)

`y` is a numpy.ndarray of shape (n)

where n is the number of samples and p is the number of independant variables

2.3.3 Properties

Note: Attributes are meant to be read-only. Modify them at your own risk. I have not chosen to enforce this since anyone can break it if they wanted anyways.

LinearRegressor`.rate` :: learning rate

LinearRegressor`.epochs` :: epochs

LinearRegressor`.w` :: matrix of values of β. This is a numpy.ndarray of shape (*epochs*, p).

LinearRegressor`.errors` :: matrix of values of ϵ. ϵ is the error between each predicted y_i and its real value, d_i, such that $\epsilon + y_i = d_i$

2.3.4 Methods

LinearRegressor`.__init__(rate, epochs)` :: Simply sets the learning rate and attributes of a linear regressor so it's ready to be fitted.

LinearRegressor`.fit(X, y)` :: Takes X (numpy.ndarray of shape (n, p)) and y (numpy.ndarray of shape (n)). n is the number of samples while p is the number of independant variables.

LinearRegressor`.predict(X)` :: takes a matrix of **x**s, and spits out their predicted **y**s using the β of the last epoch. Note that the Linear Regressor must be fitted first!

2.4 Decision Stump

2.4.1 Description

The decision stump is a decision tree which only has a depth of 1. It will take in some samples, each with n features and a label. The classifier will determine which feature and value of said feature provides a threshold which gives the largest confidence gain is classifying the given samples. While the samples can have as many labels as wished for, if this is being used as a standalone classifier a binary labeling is best.

The given dataset comes with an inherit impurity (measured by gini impurity). The classifier will go through every unique value for each feature given in X, and determine the feature and value that provides the greatest information gain (the largest decrease in gini impurity).

Uniquely, this classifier will return probabilities of a label given some features. For example, for a set of features the classifier may be 10% confident that describes label "A", 70% confident it describes label "B", and 10% confident it's "C". This result would look like:

```
{
  A: 0.1,
  B: 0.7,
  C: 0.1
}
```

It's also important to note that if a feature's value is numeric, it will try a $<$ comparison, but if it's not the classifier will try an $=$ comparison.

2.4.2 Usage

```
ds = DecisionStump()
ds.fit(X, y)
ds.predict(Z)
```

2.4.3 Properties

DecisionStump`.left` :: the probability distribution of the left side of the decision stump (dict). These are the probabilities assuming the value is on the left side, not cumulative.

DecisionStump`.right` :: the probability distribution of the right side of the decision stump (dict). These are the probabilities assuming the value is on the right side, not cumulative.

DecisionStump`.q` :: this is the `Question` that is asked by the classifier. It has members `feature` and `threshold` and a method `match` which takes an array of features and returns `True` if it satisfies that threshold or `False` otherwise.

2.4.4 Methods

DecisionStump`.__init__()` :: Just creates the object. There is nothing needed to be passed in here.

DecisionStump`fit(X, y)` :: Finds the best question to ask to get the largest about of information gain. It will choose the feature and one of the values of that feature from `X`. `y` is the labelling for each row in `X`.

DecisionStump`predict(X)` :: Takes an array of features and returns the probability distribution for each one in a corresponding array.

3 Testing

Several manual tests were used in the Iris dataset. These tests included all combinations of flowers with varying features. Some converged and some did not. I also did some 3 feature learning, to test that functionality. Although there much more manual tests, two can be found in the example `prog.py` (commit tagged prog1). The first is a 2 feature test which takes almost 800 iterations to converge while the second is a 3 feature test which converges quickly. Both of these use the first two flower species as they were the easiest to get to converge.

This needs to be fleshed out with runnable tests for the final portoflio using pytest.