Program 1 - Perceptron

Logan Grosz

Last updated: October 30, 2020

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 uses fits assumed linear data with a line using a gradient descent algorithm. It starts assuming $\beta = [0, ..., 0]$ instead of random numbers. It then calulates partial derivatives of the error term J, for each independant variable in X. J is defined as $\sum_{i=1}^{n}(y_i - (\beta \cdot \mathbf{X_i^T}))^2$. 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!

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.