Linear Regression

Linear Regression

Consider a (training) dataset with attributes X and target function y

where X is an n-by-a array and y is an n-by-m array (usually m=1)

Let's suppose there are constants W and b such that

$XW + b = y$

If we append a column containing only 1's to X, we can simplify to:

$X_1W = y$

From linear algebra, if X_1 is an invertible matrix

$(X_1)^{-1} X_1W = (X_1)^{-1} y$

$W = (X_1)^{-1}\ y$

The we can use W to predict y for test examples

Linear Regression

From linear algebra, if X_1 is an invertible matrix

$(X_1)^{-1} X_1W = (X_1)^{-1} y$

$W = (X_1)^{-1} y$

The we can use W to predict y for test examples

However, for most (all?) training sets, X_1 will not be invertible

Nevertheless, there is an approximation called the pseudo-inverse of X1, denoted by $(X_1)^*$

$(X_1)^* = (X_1^TX_1)^{-1}X_1^T$

where X_1^T is the transpose of X_1

then, for we can predict y in the test set as:

$y = X_{1test} W$

Linear Regression for Classification

In order to use a regressor (any regressor, not just linear regression) we use a one-hot representation of the output.

Suppose y is a 1D array of length n of integers containing the classes in a dataset

As usual, we assume classes are integers in the 0,...,c-1 range

Then the one-hot representation of y is an n-by-c array, where row i consists of 0 everywhere except at position y[i].

For example if y = [4,2,3,1,0,4] and classes are 0,..,4, onehot(y) is given by:

```
[[0., 0., 0., 0., 1.],
  [0., 0., 1., 0., 0.]
  [0., 0., 0., 1., 0.],
  [0., 1., 0., 0., 0.],
  [1., 0., 0., 0., 0.],
  [0., 0., 0., 0., 1.]])
```

Linear Regression for Classification

Thus we train the model to predict not y, but the one-hot representation of y:
`model.fit(X_train,onehot(y_train))`

When we predict, the model will return the predicted one-hot representation of y_test.

```
p = model.predict(X_test)
```

`p` will be a 2D array of size `X_train.shape[0]` by `onehot(y_train).shape[1]`

How do we convert to a single prediction?

We could find the Euclidean distance from each row in `p[i]` in `p` to the one-hot representation of each of the classes and assign example `i` to the class with the most similar representation.

This is equivalent to assigning example `i` to the class corresponding to the position of the largest item in `p[i]` (that is, `pred[i] = argmax(p[i])`.

Linear Regression for Classification

For example, the following array would result in predictions [4,2,3,1,0,4]:

[[0.79115577, 0.63147976, 0.39390119, 0.54309383, **1.07130847**],
    [0.62835492, 0.02026977, **1.81213046**, 0.0330209 , 0.38278168],
    [0.53767076, 0.77982095, 0.91745407, **1.84388367**, 0.04652723],
    [0.85631156, **1.76228783**, 0.91840274, 0.15162574, 0.0680549],
    [**1.29367029**, 0.01124145, 0.63691936, 0.01732445, 0.05142839],
    [0.16763345, 0.83554417, 0.7718447 , 0.17421716, **1.51564816**]])

Examining the models learned by linear regression

```
X = np.load('particles_X.npy')
y = np.load('particles_y.npy')
X_train, X_test, y_train, y_test = split_train_test(X,y)
model = linear_regression()
model.fit(X_train, y_train)
print(model.W)
[[ 0.01351843]
 [-0.04888286]
 [ 0.06925198]
 [-0.16791558]
 [ 0.70958567]
 [-0.72275243]]
```

Examining the models learned by linear regression

W is a 1x6 vector

The predicted target value of training example x is:

p = w[0]*x[0] + ... + w[4]*x[4] + w[5]

The prediction is a linear combination of the attributes

Which attribute is the most important?

```
print(model.W)
[[ 0.01351843]
 [-0.04888286]
 [ 0.06925198]
 [-0.16791558]
 [ 0.70958567]    – Coefficient with largest absolute value – Is attribute 4 the most important?
 [-0.72275243]]   – Constant value added to all predictions
```

Examining the models learned by linear regression

- Is the attribute with the largest absolute coefficient the most important?
- Not necessarily. Attributes with small values will tend to have larger coefficients and vice versa
- Linear regression is independent to scaling of its features. Thus we can scale the features to make them have unit standard deviation, then fit a linear model
- Then the coefficient with the largest absolute value will normally correspond to the most important attribute (excluding the last element of W, which is the constant or bias term)

Beyond Linear Features with Linear Regression

Is it possible to learn a quadratic function such as

$y = 3x^2 - 5x + 10$

using linear regression?

Beyond Linear Features with Linear Regression

For the particle dataset, what if *y* depends on the square of one or some of the attributes in *X*?

Beyond Linear Features with Linear Regression

For the particle dataset, what if *y* depends on the square of one or some of the attributes in *X*?

Can we learn a function such as:

p = w[0]*x[0] + ... + w[4]*x[4] + w[5]*x[0]**2 + ... + w[9]*x[4]**2 +w[10]

Beyond Linear Features with Linear Regression

For the particle dataset, what if *y* depends on the product(s) of two of the attributes in *X*?

Can we learn a function such as:

p = w[0]*x[0]*x[0] + w[1]*x[0]*x[1] + w[2]*x[0]*x[2] + ...+

w[14]*x[4]*x[4] + w[15]

Beyond Linear Features with Linear Regression

For the particle dataset, what if *y* depends on the product(s) of two of the attributes in *X*?

The linear regression algorithm can only learn linear combinations of the attributes

However, we can create new attributes to augment the original dataset with non-linear combinations of the original attributes

Beyond Linear Features with Linear Regression

We can create new attributes to augment the original dataset with non-linear combinations of the original attributes

What does the following function do?

```
X = np.load('particles_X.npy')
y = np.load('particles_y.npy')
X = np.hstack((X,X**2))
X_train, X_test, y_train, y_test = split_train_test(X,y)
model = linear_regression()
model.fit(X_train, y_train)
pred = model.predict(X_test)
```