Desh Iyer | 500081889 | Experiment - 10

April 10, 2023

1 Experiment 10 - Implementing Non Parametric Locally Weighted Regression on Non-linear Data

1.1 Theory

1.1.1 Parametric vs Non-Parametric Learning Algorithms

Parametric — In a Parametric Algorithm, we have a fixed set of parameters such as theta that we try to find(the optimal value) while training the data. After we have found the optimal values for these parameters, we can put the data aside or erase it from the computer and just use the model with parameters to make predictions. Remember, the model is just a function.

Non-Parametric — In a Non-Parametric Algorithm, you always have to keep the data and the parameters in your computer memory to make predictions. And that's why this type of algorithm may not be great if you have a really really massive dataset.

1.1.2 Need for NPLW Regression

We specifically apply this regression technique when the data to fit is non-linear. In Linear Regression we would fit a straight line to this data but that won't work here because the data is non-linear and our predictions would end up having large errors. We need to fit a curved line so that our error is minimized.

1.1.3 How NPLW Regression Works

In Locally weighted linear regression, we give the model the `x` where we want to make the prediction, then the model gives all the `x(i)`'s around that `x` a higher weight close to one, and the rest of `x(i)`'s get a lower weight close to zero and then tries to fit a straight line to that weighted `x(i)`'s data.

This means that if want to make a prediction a point on the x-axis, the model gives higher weight to the input data i.e. `x(i)`'s near or around the circle above the green point and all else `x(i)` get a weight close to zero, which results in the model fitting a straight line only to the data which is near or close to the circle. The same goes for the purple, yellow, and grey points on the x-axis.

`x` is the point where we want to make the prediction. `x(i)` is the `ith` training example.

The value of this function is always between 0 and 1.

So, if we look at the function, we see that

```
If |x(i)-x| is small, w(i) is close to 1.
If |x(i)-x| is large, w(i) is close to 0.
```

The `x(i)`'s which are far from `x` get `w(i)` close to zero and the ones which are close to `x`, get `w(i)` close to 1.

1.1.4 Calculating Error

In the loss function, it translates to error terms for the `x(i)`'s which are far from `x` being multiplied by almost zero and for the `x(i)`'s which are close to `x` get multiplied by almost 1. In short, it only sums over the error terms for the `x(i)`'s which are close to `x`.

2 Code

2.1 Importing Libraries

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import warnings

%matplotlib inline

warnings.filterwarnings('ignore')
```

2.2 Generating Non-linear Data

A good way to create a non-linear dataset is to mix sines with different phases. The dataset we will work with in this experiment is created with the following Python script and exported to a CSV file.

Create a variable to set the number of samples to create.

[ ]:
```
numberSamples = 500
```

Now, creating a lambda function that takes the input and convolves it as a mixture of two `cos` functions.

[ ]:
```
deLinearise = lambda X: np.cos(1.5 * np.pi * X) + np.cos(5 * np.pi * X)
```

Now, creating `X` and `y` using the above function and `np.random.rand`.

[ ]:
```
X = np.sort(np.random.rand(numberSamples)) * 2
y = deLinearise(X) + np.random.randn(numberSamples) * 0.1
```

Here's what `X` looks like.

[ ]:
```
X, X.shape
```

[ ]:
```
(array([1.61626173e-03, 2.16638046e-03, 4.39592108e-03, 9.78783148e-03,
        1.24444625e-02, 1.37327151e-02, 1.42952923e-02, 1.51459055e-02,
        2.00655885e-02, 2.19776641e-02, 2.58197537e-02, 2.61190460e-02,
        2.99786683e-02, 3.39833109e-02, 3.70190329e-02, 4.56369025e-02,
```

```
       1.57816518e+00, 1.57893477e+00, 1.57909490e+00, 1.57977261e+00,
       1.58304093e+00, 1.58362397e+00, 1.58428691e+00, 1.59272623e+00,
       1.59541334e+00, 1.60462124e+00, 1.60724459e+00, 1.60758362e+00,
       1.60976101e+00, 1.61100869e+00, 1.61192806e+00, 1.61936480e+00,
       1.62483045e+00, 1.63051002e+00, 1.63433863e+00, 1.63714579e+00,
       1.64345559e+00, 1.64625529e+00, 1.64665438e+00, 1.64919143e+00,
       1.65251662e+00, 1.65323536e+00, 1.65554095e+00, 1.65722484e+00,
       1.65809136e+00, 1.66034720e+00, 1.66260762e+00, 1.66811312e+00,
       1.67403874e+00, 1.67597510e+00, 1.68418806e+00, 1.68853737e+00,
       1.69149999e+00, 1.69189753e+00, 1.69809080e+00, 1.70310609e+00,
       1.71248393e+00, 1.72442935e+00, 1.72488999e+00, 1.73033559e+00,
       1.73314726e+00, 1.73423355e+00, 1.74433032e+00, 1.74614983e+00,
       1.74886706e+00, 1.74952616e+00, 1.75624607e+00, 1.75759715e+00,
       1.75898765e+00, 1.76163789e+00, 1.76674461e+00, 1.76993810e+00,
       1.77334010e+00, 1.77538726e+00, 1.77759022e+00, 1.78531637e+00,
       1.78786782e+00, 1.79045010e+00, 1.79702990e+00, 1.79921204e+00,
       1.80023752e+00, 1.80474168e+00, 1.80531188e+00, 1.81023073e+00,
       1.82051647e+00, 1.82052047e+00, 1.82493166e+00, 1.83437807e+00,
       1.84008779e+00, 1.85138943e+00, 1.85380701e+00, 1.85719613e+00,
       1.87105206e+00, 1.87450616e+00, 1.87711921e+00, 1.87839073e+00,
       1.88241657e+00, 1.88274731e+00, 1.88607804e+00, 1.89210205e+00,
       1.89286409e+00, 1.89435333e+00, 1.89746541e+00, 1.89840951e+00,
       1.89864700e+00, 1.90759482e+00, 1.90791997e+00, 1.91584686e+00,
       1.92322833e+00, 1.92549289e+00, 1.92711008e+00, 1.92810280e+00,
       1.93033446e+00, 1.94041640e+00, 1.95362281e+00, 1.97026957e+00,
       1.97197376e+00, 1.97436249e+00, 1.97539102e+00, 1.98191997e+00,
       1.98693283e+00, 1.99091012e+00, 1.99317942e+00, 1.99836875e+00]),
 (500,))
```

And now, y.

[ ]:
```
y, y.shape
```

[ ]:
```
(array([ 2.13346440e+00,  1.85221200e+00,  2.10689493e+00,  1.99324770e+00,
         1.99423445e+00,  2.04486596e+00,  2.10670623e+00,  2.07939754e+00,
         2.13084138e+00,  1.87983846e+00,  1.93843490e+00,  1.87408931e+00,
         1.88246544e+00,  1.94848495e+00,  1.96075724e+00,  1.61121467e+00,
         1.48212616e+00,  1.58837442e+00,  1.39832761e+00,  1.41424745e+00,
         1.21278770e+00,  1.28603287e+00,  1.27195535e+00,  1.37745028e+00,
         1.34547911e+00,  8.04957894e-01,  7.50657194e-01,  9.42086174e-01,
         7.52440554e-01,  8.17966250e-01,  4.99543256e-01,  4.53930244e-01,
         4.47722567e-01,  4.15866962e-01,  5.88946626e-01,  2.64582220e-01,
         3.43015342e-01,  9.13748316e-02,  2.19710912e-02, -1.47364238e-01,
        -6.75288399e-02, -3.80161331e-01, -3.90658691e-02, -2.76883791e-01,
        -2.07050375e-01, -1.76450839e-01, -2.32084937e-01, -2.64716160e-01,
        -2.72687048e-01, -3.18142736e-01, -5.44253042e-01, -5.16057077e-01,
        -3.30821410e-01, -3.75822296e-01, -3.06798011e-01, -3.82691202e-01,
```

```
        -6.97380151e-01, -7.20170276e-01, -4.49416710e-01, -8.07957537e-01,
        -8.84928643e-01, -8.58005019e-01, -9.80591487e-01, -1.17802273e+00,
        -1.03523019e+00, -9.22642166e-01, -1.22336514e+00, -1.10633587e+00,
        -1.11453395e+00, -1.30245002e+00, -1.34343433e+00, -1.32871856e+00,
        -1.42320452e+00, -1.52920797e+00, -1.29953232e+00, -1.61201982e+00,
        -1.53990361e+00, -1.48428795e+00, -1.70381647e+00, -1.54680170e+00,
        -1.56083643e+00, -1.58838453e+00, -1.63903193e+00, -1.54433995e+00,
        -1.69398649e+00, -1.52528449e+00, -1.60560809e+00, -1.46904605e+00,
        -1.60443062e+00, -1.49949205e+00, -1.19033402e+00, -1.38183831e+00,
        -1.39719004e+00, -1.20110428e+00, -1.06754351e+00, -1.17888606e+00,
        -1.13977748e+00, -1.11719976e+00, -1.17169964e+00, -1.09981540e+00,
        -9.31507067e-01, -9.41957914e-01, -9.00338830e-01, -7.91777666e-01,
        -8.09522914e-01, -7.56190452e-01, -7.58360208e-01, -5.26270226e-01,
        -6.00025527e-01, -3.66131619e-01, -4.84235173e-01, -5.58304061e-01,
        -4.16574467e-01, -4.49775054e-01, -2.30266290e-01, -2.38521132e-01,
         5.12563421e-02, -1.97718422e-02, -6.19207878e-02, -1.81074868e-01,
        -1.90562189e-02, -2.36026793e-02,  7.52465215e-02,  9.22098126e-02]),
 (500,))
```

2.2.1 Reshaping Arrays

[ ]:
```
X = X.reshape(X.shape[0], 1)
X.shape
```

[ ]:
```
(500, 1)
```

[ ]:
```
y = y.reshape(y.shape[0], 1)
y.shape
```

[ ]:
```
(500, 1)
```

2.3 Calculating Predictions

Defining a function to calculate the diagonal weight matrix. This function takes in the test point, the training data and the value of `tau` which corresponds to the radius of the circle surrounding a point. All the points laying in the circle are considered for the regression problem.

[ ]:
```
# Weight Matrix in code. It is a diagonal matrix.def wm(point, X, tau):
def calculateWeightMatrix(point, X, tau):
    '''
    The parameters of this function are,
                tau --> bandwidth
                X --> Training data.
                point --> the x where we want to make the prediction.
    '''

    # m is the No of training examples .
```

```
    m = X.shape[0]

    # Initialising W as an identity matrix.
    w = np.mat(np.eye(m))

    # Calculating weights for all training examples [x(i)'s].
    for i in range(m):
        xi = X[i]
        d = (-2 * tau * tau)
        w[i, i] = np.exp(np.dot((xi - point), (xi - point).T) / d)

    return w
```

Now, defining a function to predict the y value for a point which uses the previously defined function to calculate the weight matrix.

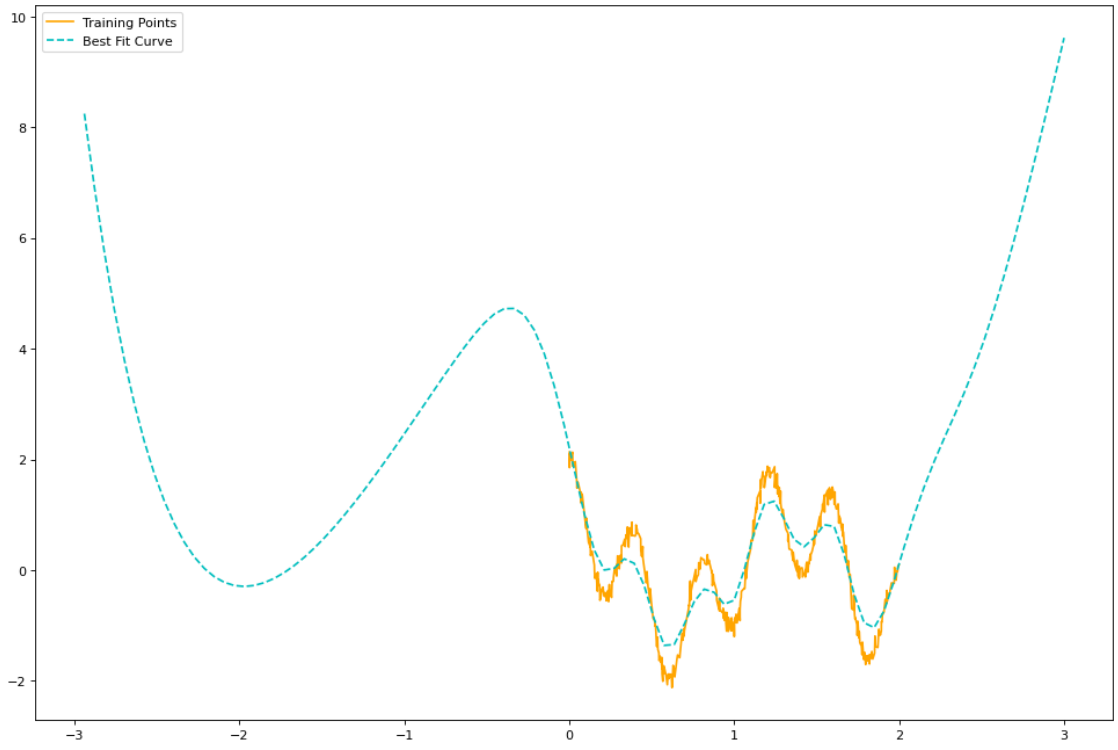
[ ]:
```
def predict(X, y, point, tau):
    # m = number of training examples.
    m = X.shape[0]

    onesColumn = np.ones(m).reshape(m, 1)

    # Appending a column of ones in X to add the bias term. Just one parameter:␣
 ↪theta, that's why adding a column of ones to X and also adding a 1 for the␣
 ↪point where we want to predict.
    X_ = np.append(X, onesColumn, axis=1)

    # point is the x where we want to make the prediction.
    point_ = np.array([point, 1])

    # Calculating the weight matrix using the wm function we wrote      #  #␣
 ↪earlier.
    w = calculateWeightMatrix(point_, X_, tau)

    # Calculating parameter theta using the formula.
    theta = np.linalg.pinv(X_.T * (w * X_)) * (X_.T * (w * y))

    # Calculating predictions.
    pred = np.dot(point_, theta)

    # Returning the theta and predictions
    return theta, pred
```

2.4 Plotting Predictions

[ ]:
```
def plot_predictions(X, y, tau, numberPredictionValues):
    '''
    The values for which we are going to predict.
    X_test includes numberPredictionValues evenly spaced values in the domain␣
 ↪of X.

          X --> Training data.
         y --> Output sequence.
         numberPredictionValues --> number of values/points for which we are␣
 ↪going to
         predict.
         tau --> the bandwidth.
    '''
    X_test = np.linspace(-3, 3, numberPredictionValues)

    # Empty list for storing predictions.
    predictions = []

    # Predicting for all numberPredictionValues values and storing them in␣
 ↪predictions.
    for point in X_test:
        theta, pred = predict(X, y, point, tau)
        predictions.append(pred)

    # Reshaping X_test and predictions
    X_test = np.array(X_test).reshape(numberPredictionValues, 1)
    predictions = np.array(predictions).reshape(numberPredictionValues, 1)

    # Plotting
    plt.figure(figsize=(15, 10), dpi=80)

    plt.plot(X, y, 'orange')
    plt.plot(X_test, predictions, 'c--')

    plt.legend(['Training Points', 'Best Fit Curve'])

    plt.show()
```

2.5 Putting it All Together

[ ]:
```
plot_predictions(X, y, 0.08, 100)
```



3 References

1. Medium Article
2. Creating Random Non-Linear Data