in3050_2023_w7_solutions

February 26, 2023

1 Week 07: Linear and logistic regression

1.0.1 Introduction

This week, we will get some first-hand experience with regression. We will implement gradient descent for linear regression. Then we will proceed to classification, first by using linear regression and then logistic regression.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
```

1.1 NumPy

We will first familiarize ourselves a little with NumPy. A function which we will use over again is `linspace(x1,x2,N)` which makes a vector of length N splitting the interval $[x1, x2]$ into equally sized intervals.

[2]:
```
xx = np.linspace(-5,5,101)
xx[:10]
```

[2]:
```
array([-5. , -4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1])
```

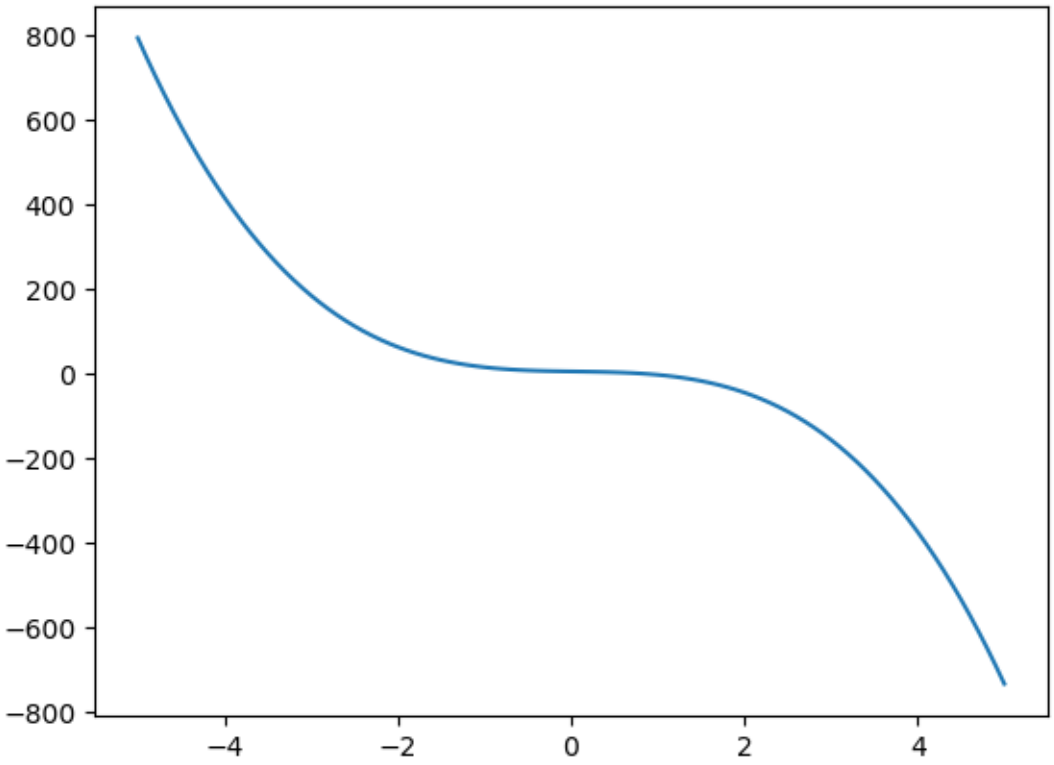
One of the major improvements from using NumPy is the possibilty of computing many values by applying a function to a numpy array.

[3]:
```
y1 = -6*xx**3 + xx**2 -3*xx + 5
```

`y1`will contain the corresponding function values for each element `x`in `xx`. We may plot the result.

[4]:
```
plt.plot(xx,y1)
```

[4]:
```
[<matplotlib.lines.Line2D at 0x1975b525240>]
```

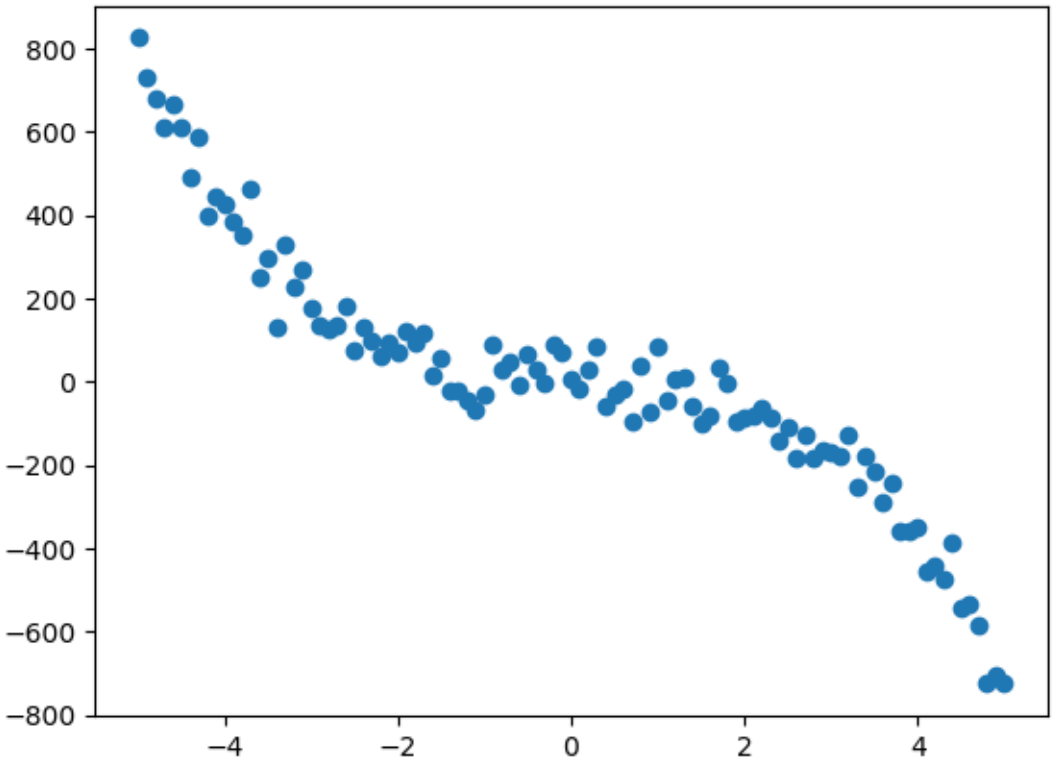


1.2 Dataset for linear regression

We will start with a smooth curve and add some "noise". The underlying idea is that the smooth curve represents the function we are looking for, and that this is the best we can hope to learn. A solution which does better on the training material than the smooth curve is probably overfit and will not generalize as well to new data as the smooth curve. We are using a normal distribution to generate noise. The numpy function `normal` will generate a vector of `size` many random points around `loc` from a distribution with standard deviation `scale`.

[5]:
```
from numpy.random import normal
t = y1 + normal(loc=0, scale=50, size=101)
plt.scatter(xx, t)
```

[5]:
```
<matplotlib.collections.PathCollection at 0x1975be1eda0>
```



Our training data now consists of pairs (xx[i], t[i]), where xx[i] is the datapoint and t[i] the target value. So far, both `xx` and `t` are vectors. Check their shapes, e.g., `xx.shape`. The goal is to make an implementation for linear regression which works with an arbitrary number of input features and not just one. We will therefore transform `xx` to a matrix of dimension $N \times m$ where each row represents one datapoint, and m is the number of input variables (or features). Check the shape of `X` after the transform.

[6]:
```
X = xx.reshape(-1,1)
```

Solution

[7]:
```
xx.shape
```

[7]:
```
(101,)
```

[8]:
```
X.shape
```

[8]:
```
(101, 1)
```

End of solution

1.3 Part A: Linear regression

We will implement our own linear regression model. Our aim is to find an approximate function that fits the data generated above.

Since we are dealing with only one input variable, we start with a simple linear function, $f(x_1) = w_0 + w_1 x_1$.

1.3.1 Exercise 1: MSE

We wonder if our f fits the data well, and what parameters will give us the best approximation. We will estimate this using the Mean Squared Error:

$\frac{1}{N}\sum_{j=1}^{N}(t_j - \sum_{i=0}^{m} w_i x_{ji})^2$

Write a function calculating MSE of our approximation.

[9]:
```
# Your code here
```

solution

[10]:
```
# Pure python solution
def mse_1(y, y_pred):
    sum_errors = 0.
    for i in range(0,len(y)):
        sum_errors += (y[i] - y_pred[i])**2
    mean_squared_error = sum_errors/len(y)
    return mean_squared_error
```

[11]:
```
# Vector form solution
def mse_2(x,y):
    return np.mean((x - y)**2)
```

end of solution

1.3.2 Exercise 2: Testing the MSE

To test our implementation, we can take the function $f(x_1) = 0$ as a baseline and calculate the MSE for this f. Also calculate the Root Means Square Error which provides a more natural measure for how good the fit is.

Solution

[12]:
```
hypothesis = np.array([0 for x in xx])
mse = mse_1(hypothesis, t)
print("MSE: ", mse)
```

```
MSE:  92493.0749194188
```

[13]:
```
hypothesis = np.array([0 for x in xx])
mse = mse_2(hypothesis, t)
```

```
print("MSE: ", mse)
```

```
MSE:  92493.0749194188
```

[14]:
```
print("RMSE: ", np.sqrt(mse))
```

```
RMSE:  304.1267415394753
```

End of solution

1.3.3 Exercise 3: Adding bias

We will implement linear regression with gradient descent and test it on the data. To make it simple, we will add a $x_0 = 1$ to all our datapoints, and consider $f(x_1) = w_0 + w_1 x_1$ as $f(x_0, x_1) = w_0 x_0 + w_1 x_1$. Make a procedure that does this.

[15]:
```
def add_bias(X):
    """X is a Nxm matrix: N datapoints, m features
    Return a Nx(m+1) matrix with added bias in position zero"""
    pass
```

solution There are many ways that can be done in numpy. This is one of them.

[16]:
```
def add_bias(X):
    """X is a Nxm matrix: N datapoints, m features
    Return a Nx(m+1) matrix with added bias in position zero"""
    N = X.shape[0]
    bias = np.ones((N,1)) # Make a N*1 matrix of 1-s
    # Concatenate the column of bias in front of the columns of X.
    return np.concatenate((bias, X), axis  = 1)
```

end of solution

1.3.4 Exrcise 4: Gradient Descent

We will implement the linear regression in a class as we did with the classifiers earlier. The fit method will run the gradient descent step a number of times to train the classifier. The predict method should take a matrix containing several data points and predict the outcome for all of them. Fill in the methods.

Assume that the matrix of training data is not extended with bias features. Hence, make adding bias a part of your methods.

After training there should be an attribute with learned coeffecients (weights) which is applied by the predict method.

[17]:
```
class NumpyLinReg():

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
```

```
        """X_train is a Nxm matrix, N data points, m features
        t_train is avector of length N,
        the targets values for the training data"""


    def predict(self, X):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""
        pass
```

solution

[18]:
```
class NumpyLinReg():

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
        """X_train is a Nxm matrix, N data points, m features
        t_train is avector of length N,
        the targets values for the training data"""

        (N, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        for e in range(epochs):
            weights -= eta / N *  X_train.T @ (X_train @ weights - t_train)

    def predict(self, X):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""
        Z = add_bias(X)
        return Z @ self.weights
```

end of solution

1.3.5 Exercise 5: Train and test the model

Fit the model to the training data. Report the coefficients. Plot the line together with the observations. Calculate the RMSE. Is the result a better fit than the baseline constant function $f(x) = 0$?

solution

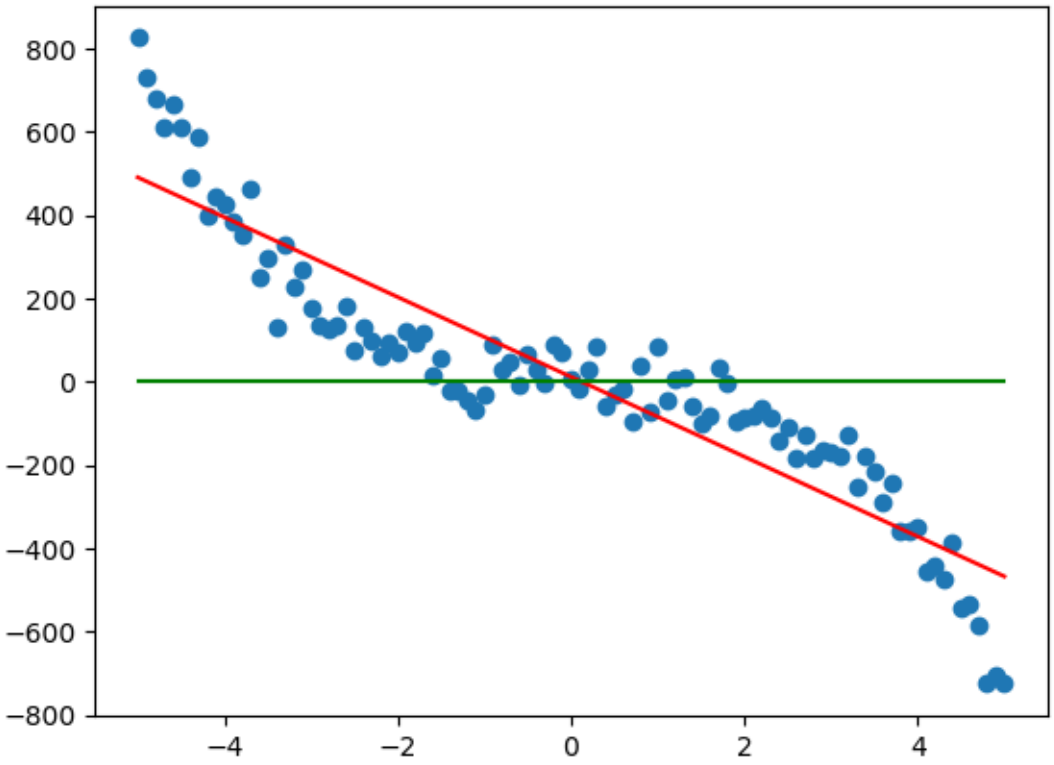
[19]:
```
reg = NumpyLinReg()
reg.fit(X,t, epochs=100)
print("The coefficients: ", reg.weights)
```

```
The coefficients:  [ 11.43131192 -95.77449346]
```

[20]:
```
plt.plot(X, reg.weights[0]+reg.weights[1]*X, color="r")
plt.plot(X, [0 for i in X], 'g')

# The generated dataset
plt.scatter(X, t)
```

[20]: <matplotlib.collections.PathCollection at 0x1975becb640>



[21]:
```
error = mse_2(t, add_bias(X) @ reg.weights)
print("MSE: ", error)
```

```
MSE:  14393.987508996268
```

[22]:
```
print("RMSE :", np.sqrt(error))
```

```
RMSE : 119.9749453385842
```

It has improved

end of solution

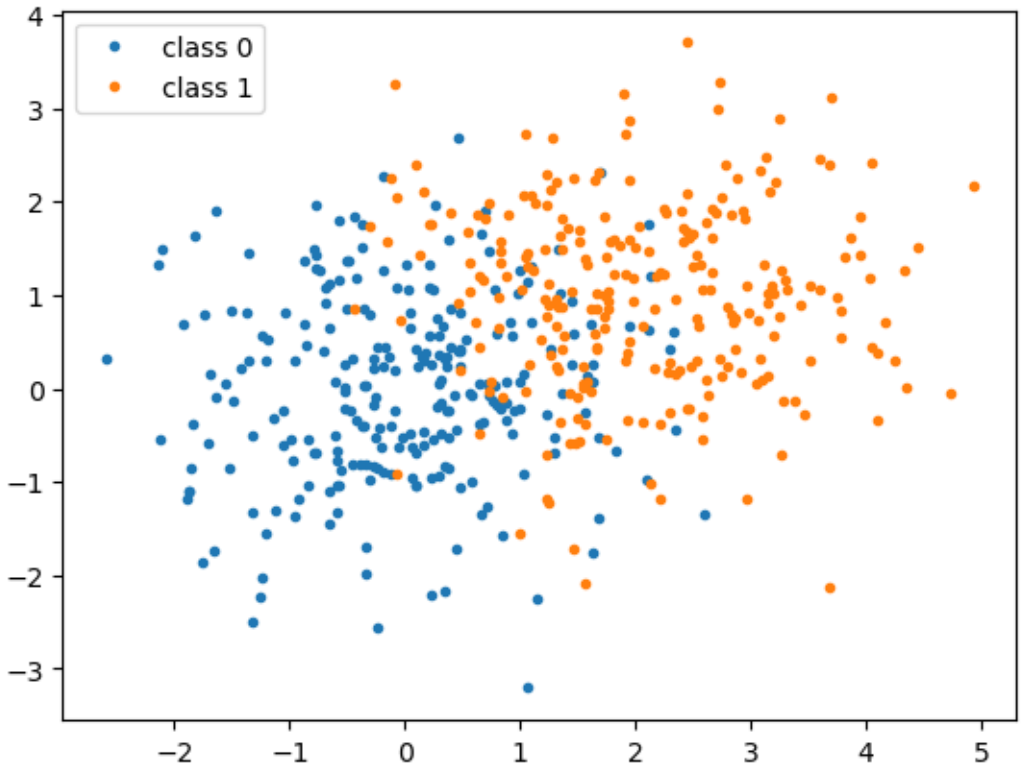
1.4 Dataset for classification

We will use simple synthetic data similarly to week_05, but we will make the set a little bigger to get more reliable results.

[23]:
```
from sklearn.datasets import make_blobs
X_train, y_train = make_blobs(n_samples=500, centers=[[0,0],[1,2]],
                   n_features=2, random_state=2019)
X_test, y_test = make_blobs(n_samples=500, centers=[[0,0],[1,2]],
                   n_features=2, random_state=2020)
```

[24]:
```
def show(X, y, marker='.'):
    labels = set(y)
    for lab in labels:
        plt.plot(X[y == lab][:, 1], X[y == lab][:, 0],
                 marker, label="class {}".format(lab))
    plt.legend()
```

[25]:
```
show(X_train, y_train)
```



1.5 Linear Regression classifier

This is also called Ridge Classifier in the literature when it is smoothed. We will consider the simple unsmoothed version here and return to smoothing and regularization in a later lecture.

1.5.1 Exercise 6: Coding the classifier

Make a linear regression classifier. Make it as a Python class with methods for 'fit' and 'predict', similarly to the linear regression above and the kNN classifier from week05 exercises.

Solution

[27]:
```
class NumpyClassifier():
    """Common methods to all numpy classifiers --- if any"""
```

[28]:
```
class NumpyLinRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
        """X_train is a Nxm matrix, N data points, m features
        t_train is avector of length N,
        the targets values for the training data"""

        (N, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        for e in range(epochs):
            weights -= eta / N *  X_train.T @ (X_train @ weights - t_train)

    def predict(self, x, threshold=0.5):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""
        z = add_bias(x)
        score = z @ self.weights
        return score>threshold
```

End of solution

1.5.2 Exercise 7: Experiment

We will conduct repeated testing. We therefore need a development test set different from the final test set. Make such a set X_dev, y_dev, similarly to X_test, y_test using random_state=2021. Train the classfier on X_train, y_train and test for accuracy on X_dev, y_dev.

You also need a procedure for accuracy here.

Solution

[29]:
```
X_dev, y_dev = make_blobs(n_samples=500, centers=[[0,0],[1,2]],
                          n_features=2, random_state=2021)
```

[30]:
```
# The week05 solution

def accuracy_1(y, t):
    """Calculate the accuracy"""
    equal = len([(p, g) for (p,g) in zip(y, t) if p==g])
    return equal / len(t)
```

[31]:
```
# Vector solution

def accuracy(y, t):
    return np.mean(y == t)
```

[32]:
```
lin_cl = NumpyLinRegClass()
lin_cl.fit(X_train, y_train)
accuracy(lin_cl.predict(X_dev), y_dev)
```

[32]:
```
0.864
```

End of solution

1.6 Logistic Regression

1.6.1 Exercise 8: The logistic function

Implement the logistic function. Sometimes called only the sigmoid.

[33]:
```
def logistic(x):
    # fill in the rest
    pass
```

Solution

[34]:
```
def logistic(x):
    return 1/(1+np.exp(-x))
```

End of solution

1.6.2 Exercise 9: Code for the classifier

Write code for the logistic regression classifier. Compared to linear regression classifier you have to make adaptions to both fit and predict taking the logistic into consideration

Solution

[35]:
```
class NumpyLogReg(NumpyClassifier):

```

```
    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
        """X_train is a Nxm matrix, N data points, m features
        t_train is avector of length N,
        the targets values for the training data"""

        (N, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        for e in range(epochs):
            weights -= eta / N *  X_train.T @ (self.forward(X_train) - t_train)

    def forward(self, X):
        return logistic(X @ self.weights)

    def predict(self, x, threshold=0.5):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""
        z = add_bias(x)
        return (self.forward(z) > threshold).astype('int')
```

End of solution

1.6.3 Exercise 10: Initial experiments

Train the classfier on X_train, y_train and test for accuracy on X_dev, y_dev.

Solution

[36]:
```
lr_cl = NumpyLogReg()
lr_cl.fit(X_train, y_train)
accuracy(lr_cl.predict(X_dev), y_dev)
```

[36]:
```
0.752
```

End of solution

1.6.4 Exercise 11: Repeated experimentation

Did you get better results than with the linear regression classifier? That does not necessary have to be the case for this data set. But, if your result is much inferior to the linear regression classifier, the reason might be the parameter settings. Experiment with the parameter values for the learning rate and the number of epochs to get an optimal result.

Solution

[37]:
```
for e in [1, 2, 5, 10, 50, 100, 1000, 10000, 100000]:
    lr_cl = NumpyLogReg()
    lr_cl.fit(X_train, y_train, epochs=e, eta=0.1)
    print("Learning rate: {}  Epochs: {:7}  Accuracy: {}".format(
        0.1, e, accuracy(lr_cl.predict(X_dev), y_dev)))
```

```
Learning rate: 0.1  Epochs:       1  Accuracy: 0.74
Learning rate: 0.1  Epochs:       2  Accuracy: 0.742
Learning rate: 0.1  Epochs:       5  Accuracy: 0.744
Learning rate: 0.1  Epochs:      10  Accuracy: 0.752
Learning rate: 0.1  Epochs:      50  Accuracy: 0.818
Learning rate: 0.1  Epochs:     100  Accuracy: 0.842
Learning rate: 0.1  Epochs:    1000  Accuracy: 0.87
Learning rate: 0.1  Epochs:   10000  Accuracy: 0.874
Learning rate: 0.1  Epochs:  100000  Accuracy: 0.874
```

[38]:
```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    lr_cl = NumpyLogReg()
    lr_cl.fit(X_train, y_train, epochs=1000, eta = eta)
    print("Learning rate: {:7}  Epochs: {:4}  Accuracy: {}".format(
        eta, 1000, accuracy(lr_cl.predict(X_dev), y_dev)))
```

```
Learning rate:       1  Epochs: 1000  Accuracy: 0.874
Learning rate:     0.1  Epochs: 1000  Accuracy: 0.87
Learning rate:    0.01  Epochs: 1000  Accuracy: 0.842
Learning rate:   0.001  Epochs: 1000  Accuracy: 0.76
Learning rate:  0.0001  Epochs: 1000  Accuracy: 0.742
Learning rate:   1e-05  Epochs: 1000  Accuracy: 0.74
```

We see that the best accuracy is 0.874. We also see that it is faster to reach the solution with a learning rate of 1.0 than with smaller learning rates. #### End of solution

1.7 End of week 07