index

March 11, 2022

1 Coding Logistic Regression From Scratch - Lab

1.1 Introduction

In this lab, you'll practice your ability to translate mathematical algorithms into Python functions. This will deepen and solidify your understanding of logistic regression!

1.2 Objectives

In this lab you will:

- Build a logistic regression model from scratch using gradient descent

1.3 Overview

Recall that the logistic regression algorithm builds upon the intuition from linear regression. In logistic regression, you start by taking the input data, `X`, and multiplying it by a vector of weights for each of the individual features, which produces an output, `y`. Afterward, you'll work on using an iterative approach via gradient descent to tune these weights.

1.4 Linear regression setup

Write a simple function `predict_y()` that takes in a matrix `X` of observations and a vector of feature weights `w` and outputs a vector of predictions for the various observations.

Recall that this is the sum of the product of each of the feature observations and their corresponding feature weights:

$\hat{y}_i = X_{i1} \cdot w_1 + X_{i2} \cdot w_2 + X_{i3} \cdot w_3 + ... + X_{in} \cdot w_n$

> **Hint**: Think about which mathematical operation you've seen previously that will take a matrix (`X`) and multiply it by a vector of weights (`w`). Use NumPy!

[1]:
```python
# Your code here
import numpy as np
import pandas as pd

def predict_y(X, w):
    y = np.dot(X,w)
    return y
```

1.5 The sigmoid function

Recall that the sigmoid function is used to map the linear regression model output to a range of 0 to 1, satisfying basic premises of probability. As a reminder, the sigmoid function is defined by:

$$S(x) = \frac{1}{1 + e^{(-x)}}$$

Write this as a Python function where `x` is the input and the function outputs the result of the sigmoid function.

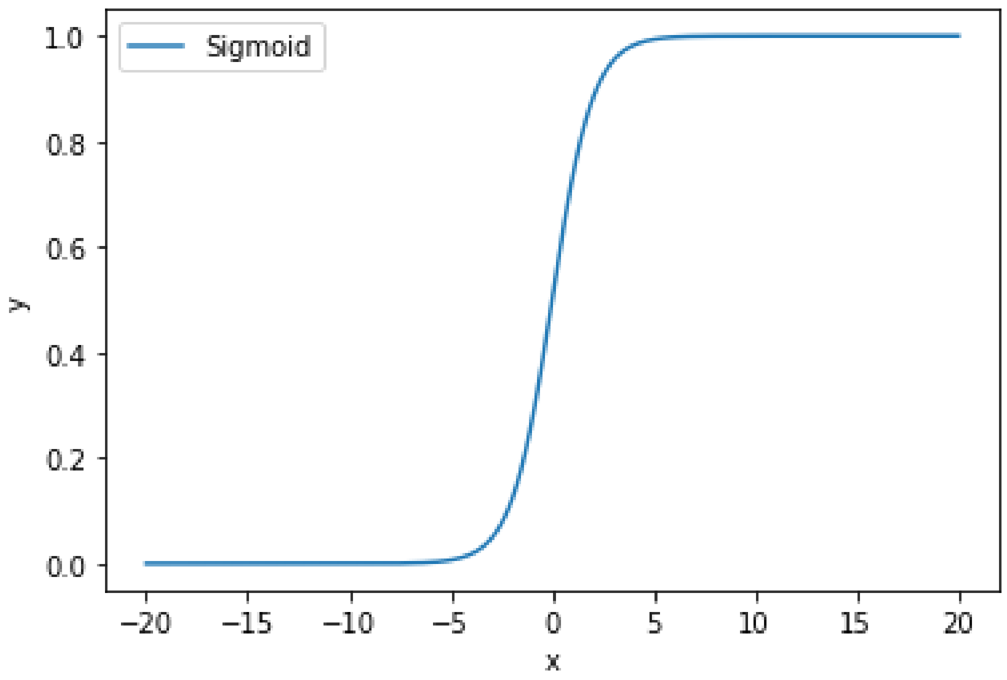
> **Hint**: Use NumPy!

[2]:
```
# Your code here
def sigmoid(y):
    s = 1 / (1 + np.exp(-1*y))
    return s
```

1.6 Plot the sigmoid

For good measure, let's do a brief investigation of your new function. Plot the output of your `sigmoid()` function using 10,000 values evenly spaced from -20 to 20.

[3]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Plot sigmoid
x = np.linspace(start = -20, stop = 20, num = 10000)
y = sigmoid(x)

sns.lineplot(x = x, y = y, label = "Sigmoid")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show();
```



1.7 Gradient descent with the sigmoid function

Recall that gradient descent is a numerical method for finding a minimum to a cost function. In the case of logistic regression, you are looking to minimize the error between the model's predictions and the actual data labels. To do this, you first calculate an error vector based on the current model's feature weights. You then multiply the transpose of the training matrix itself by this error vector in order to obtain the gradient. Finally, you take the gradient, multiply it by the step size and add this to our current weight vector to update it. Below, write such a function. It will take 5 inputs:

- `X`
- `y`
- `max_iterations`
- `alpha` (the step size)
- `initial_weights`

By default, have your function set the `initial_weights` parameter to a vector where all feature weights are set to 1.

[19]:
```python
# Your code here
def grad_desc(X, y, max_iterations, alpha, initial_weights=None):
    """Be sure to set default behavior for the initial_weights parameter."""
    if initial_weights==None:
        initial_weights = np.ones((X.shape[1],1)).flatten()
# Create a for loop of iterations
    # Generate predictions using the current feature weights
    # Calculate an error vector based on these initial predictions and the
    # correct labels
    # Calculate the gradient
    # As we saw in the previous lab, calculating the gradient is often the
    # most difficult task.
    # Here, your are provided with the closed form solution for the gradient
    # of the log-loss function derived from MLE
    # For more details on the derivation, see the additional resources
    # section below.

    w = initial_weights
    w_df = pd.DataFrame({})
    w_df["initial_weight"] = w
    for i in range(max_iterations):

        y_pred = sigmoid(predict_y(X, w))
        error_vector = y - y_pred
        gradient = np.dot(X.transpose(), error_vector)
        # Update the weight vector take a step of alpha in direction of gradient
        w += alpha * gradient

        w_df[f"{i}_iter"] = w
    # Return finalized weights
    return w, w_df
```

1.8 Running your algorithm

Now that you've coded everything from the ground up, you can further investigate the convergence behavior of the gradient descent algorithm. Remember that gradient descent does not guarantee a global minimum, only a local minimum, and that small deviations in the starting point or step size can lead to different outputs.

First, run the following cell to import the data and create the predictor and target variables:

[20]:
```
# Import data
import pandas as pd
df = pd.read_csv('heart.csv')

# Create the predictor and target variables
y = df['target']
X = df.drop(columns=['target'], axis=1)
```

```
print(y.value_counts())
X.head()
```

```
1.0    165
0.0    138
Name: target, dtype: int64
```

[20]:

```
        age  sex        cp  trestbps      chol  fbs  restecg   thalach  exang  \
0  0.708333  1.0  1.000000  0.481132  0.244292  1.0      0.0  0.603053    0.0
1  0.166667  1.0  0.666667  0.339623  0.283105  0.0      0.5  0.885496    0.0
2  0.250000  0.0  0.333333  0.339623  0.178082  0.0      0.0  0.770992    0.0
3  0.562500  1.0  0.333333  0.245283  0.251142  0.0      0.5  0.816794    0.0
4  0.583333  0.0  0.000000  0.245283  0.520548  0.0      0.5  0.702290    1.0

    oldpeak  slope   ca      thal
0  0.370968    0.0  0.0  0.333333
1  0.564516    0.0  0.0  0.666667
2  0.225806    1.0  0.0  0.666667
3  0.129032    1.0  0.0  0.666667
4  0.096774    1.0  0.0  0.666667
```

Run your algorithm and plot the successive weights of the features through iterations. Below is a dataset, with X and y predefined for you. Use your logistic regression function to train a model. As the model trains, record the iteration cycle of the gradient descent algorithm and the weights of the various features. Then, plot this data on subplots for each of the individual features. Each graph should have the iteration number on the x-axis and the value of that feature weight for that iteration cycle on the y-axis. This will visually display how the algorithm is adjusting the weights over successive iterations, and hopefully show convergence to stable weights.

[24]:

```
w.shape
```

[24]:

```
(13,)
```

[25]:

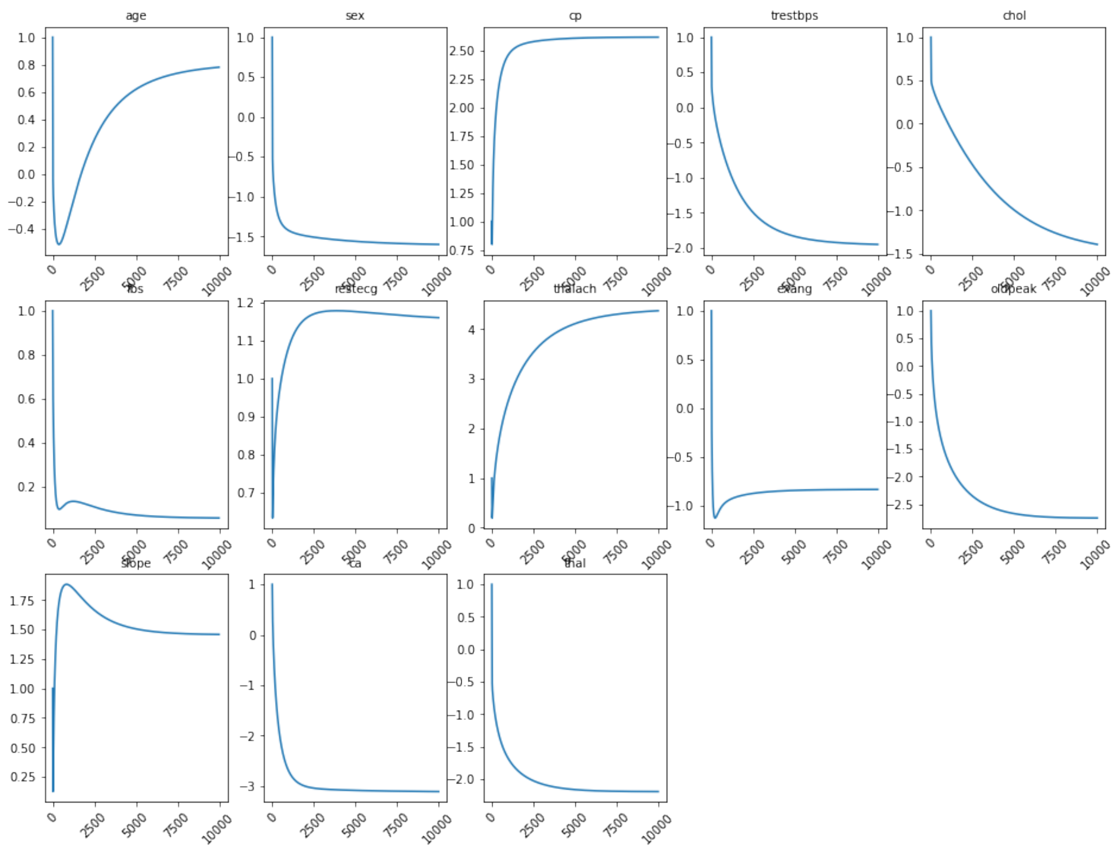
```
w_df
```

[25]:

```
    initial_weight    0_iter    1_iter    2_iter    3_iter    4_iter  \
0              1.0  0.921824  0.844091  0.766977  0.690725  0.615666
1              1.0  0.886965  0.774398  0.662523  0.551665  0.442287
2              1.0  0.978665  0.957571  0.936807  0.916498  0.896810
3              1.0  0.948142  0.896567  0.845390  0.794768  0.744918
4              1.0  0.961222  0.922681  0.884469  0.846715  0.809593
5              1.0  0.978095  0.956234  0.934437  0.912736  0.891173
6              1.0  0.969541  0.939287  0.909318  0.879747  0.850725
7              1.0  0.929711  0.859963  0.790966  0.723009  0.656483
8              1.0  0.924392  0.848973  0.773836  0.699118  0.625019
9              1.0  0.965014  0.930148  0.895453  0.861000  0.826886
10             1.0  0.921124  0.842862  0.765457  0.689245  0.614685
11             1.0  0.959973  0.920046  0.880263  0.840691  0.801422
```

```
12            1.0  0.884694  0.770039  0.656297  0.543836  0.433160

      5_iter    6_iter    7_iter    8_iter    9_iter
0   0.542248  0.471051  0.402792  0.338294  0.278409
1   0.335031  0.230757  0.130552  0.035687 -0.052503
2   0.877964  0.860242  0.843990  0.829602  0.817482
3   0.696134  0.648794  0.603371  0.560406  0.520463
4   0.773338  0.738251  0.704703  0.673119  0.643934
5   0.869810  0.848732  0.828054  0.807918  0.788492
6   0.822455  0.795203  0.769294  0.745104  0.723024
7   0.591911  0.529961  0.471451  0.417301  0.368449
8   0.551823  0.479923  0.409834  0.342183  0.277664
9   0.793248  0.760263  0.728157  0.697195  0.667662
10  0.542391  0.473156  0.407954  0.347889  0.294083
11  0.762590  0.724372  0.687002  0.650760  0.615958
12  0.324949  0.220093  0.119691  0.025007 -0.062652
```

[34]:
```
# Your code here
iteration = 10000
alpha = 0.001
w, w_df = grad_desc(X, y, iteration, alpha) # 10000

w_df.columns = np.arange(len(w_df.columns))

plt.figure(figsize=(16, 12))


for (i, j) in enumerate(w):
    plt.subplot(3, 5, i + 1)
    plt.title(list(X)[i], size='medium')
    plt.plot(w_df.iloc[i].T)
    plt.xticks(rotation = 45)
    plt.axis('tight')
```



1.9 Scikit-learn

For comparison, import scikit-learn's standard `LogisticRegression()` function. Initialize it with **no intercept** and **C=1e16** or another very high number. The reason is as follows: our implementation has not used an intercept, and you have not performed any regularization such as Lasso or Ridge (scikit-learn uses l2 by default). The high value of `C` will essentially negate this. Also, set the `random_state` to 2 and use the `'liblinear'` solver.

After initializing a regression object, fit it to `X` and `y`.

[36]:
```python
# Your code here
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(fit_intercept=False, random_state = 2,
                            C = 1e16, solver = "liblinear")
logreg.fit(X,y)
```

[36]:
```
LogisticRegression(C=1e+16, fit_intercept=False, random_state=2,
                   solver='liblinear')
```

1.10 Compare the models

Compare the coefficient weights of your model to that generated by scikit-learn.

[47]:
```python
# Your code here

print("From SKlearn:", np.round(logreg.coef_, 3))
print()

print("Our Model: ",list(np.round(w_df[len(w_df.columns)-1], 3)))
```

```
From SKlearn: [[ 0.812 -1.613  2.618 -1.969 -1.509  0.057  1.152  4.421 -0.83
-2.747
   1.456 -3.116 -2.191]]

Our Model:  [0.78, -1.602, 2.618, -1.952, -1.394, 0.058, 1.16, 4.369, -0.831,
-2.751, 1.458, -3.108, -2.192]
```

1.11 Level up (Optional)

Update the gradient descent algorithm to also return the cost after each iteration. Then rerun the algorithm and create a graph displaying the cost versus the iteration number.

[52]:
```python
# Your code here
def grad_desc(X, y, max_iterations, alpha, initial_weights=None):
    """Be sure to set default behavior for the initial_weights parameter."""

    if initial_weights==None:
        initial_weights = np.ones((X.shape[1],1)).flatten()
# Create a for loop of iterations
    # Generate predictions using the current feature weights
    # Calculate an error vector based on these initial predictions and the
    # correct labels
    # Calculate the gradient
    # As we saw in the previous lab, calculating the gradient is often the
    # most difficult task.
    # Here, your are provided with the closed form solution for the gradient
    # of the log-loss function derived from MLE
    # For more details on the derivation, see the additional resources
    # section below.
    cost = []
    w = initial_weights
    w_df = pd.DataFrame({})
    w_df["initial_weight"] = w
    for i in range(max_iterations):

        y_pred = sigmoid(predict_y(X, w))
        error_vector = y - y_pred
        gradient = np.dot(X.transpose(), error_vector)
        # Update the weight vector take a step of alpha in direction of gradient
        w += alpha * gradient

        w_df[f"{i}_iter"] = w
        cost.append(-y*np.log(y_pred) - (1-y)*log(1-y_pred))
    # Return finalized weights
    return w, w_df, cost
```

1.12 Additional Resources

If you want to see more of the mathematics behind the gradient derivation above, check out section 4.4.1 from the Elements of Statistical Learning which can be found here: https://web.stanford.edu/~hastie/ElemStatLearn//.

1.13 Summary

Congratulations! You just coded logistic regression from the ground up using NumPy! With this, you should have a fairly deep understanding of logistic regression and how the algorithm works! In the upcoming labs, you'll continue to explore this from a few more angles, plotting your data along with the decision boundary for our predictions.