BTT004_week4_lab-solution

May 1, 2024

1 Lab 4: ML Life Cycle: Modeling

Building a Logistic Regression Model From Scratch

[1]:
```
import pandas as pd
import numpy as np
import os
from sklearn.linear_model import LogisticRegression
```

In this lab, you will continue working with the modeling phase of the machine learning life cycle. You will take what you have learned about gradient descent and write a Python class from scratch to train a logistic regression model. You will implement the various mathematical functions learned in the course, such as the gradient and Hessian of the log loss.

In the course videos, we presented functions that compute the log loss, gradient and Hessian, and that implement gradient descent for logistic regression. You will do similar work here, only we'll refactor the code to improve its generality.

You will complete the following tasks:

1. Build a class that can:
    - Fit a logistic regression model given training data
    - Make predictions
2. Build your DataFrame and define your ML problem:
    - Load the Airbnb "listings" data set into a DataFrame
    - Define the label - what are you predicting?
    - Identify features
3. Create labeled examples from the data set
4. Train a logistic regression classifier using your class
5. Benchmark our class against scikit-learn's logistic regression class

2 A Logistic Regression Class

The code cell below contains the logistic regression class that we are building. Your task is to complete the logic within each specified method. Remember, a method is just a function that belongs to that particular class.

Below is a breakdown of the methods contained in the class:

1. An `__init__()` method that takes in an error tolerance as a stopping criterion, as well as max number of iterations.
2. A `predict_proba()` method that takes a given matrix of features X and predicts $P = \dfrac{1}{1+e^{-(X \cdot W+\alpha)}}$ for each entry
3. A `compute_gradient()` method that computes the gradient vector G
4. A `compute_hessian()` method that computes the Hessian. Note that the H can be broken down to the following matrix multiplications: $H = (X^T * Q) \cdot X$.
5. An `update_weights()` method that applies gradient descent to update the weights
6. A `check_stop()` method that checks whether the model has converged or the max iterations have been met
7. A `fit()` method that trains the model. It takes in the data and runs the gradient optimization

2.1 Part 1. Complete the Class

Task: Follow the steps below to complete the code in the `LogisticRegressionScratch` class.

Step A Complete the `self.predict_proba()` method. This is where we implement the inverse logit. (Note: This implementation looks a little bit different from the formula you have seen previously. This is simply because we will absorb the intercept term into our `X` matrix). Do the following:

1. Create a variable `XW`. Assign it the result of the dot product of the input `X` and `self.weights_array` variable
2. Create a variable `P`. Assign it the result of the inverse logit $(1 + e^{-XW})^{-1}$
3. Make sure the method returns the variable `P` (the `return` statement has been provided for you).

Step B Complete the `self.compute_gradient()` method. This is where we implement the log loss gradient. Do the following: 1. Create a variable `G`. Assign it the result of the gradient computation $-(y - P) \cdot X$ 2. Make sure the method returns the variable `G` (the `return` statement has been provided for you).

Step C Complete the `self.compute_hessian()` method. This is where we implement the log loss Hessian. Do the following: 1. Create a variable `Q`. Assign it the result of the following computation $P * (1 - P)$ 2. Create a variable `XQ`. Assign it the result of the following computation $X^T * Q$. Note that X is the input to the method and this is using regular multiplication 3. Create a variable called `H`. Assign it the result of the following computation $XQ \cdot X$. Note that this operation is using the dot product for matrix multiplication 4. Make sure the method returns the variable `H` (the `return` statement has been provided for you).

Step D Complete the `self.update_weights()` method. This is where we implement the gradient descent update. Do the following: 1. Create a variable `P`. Call the `self.predict_proba()` method to get predictions and assign the result to variable `P`. Note, when calling a method from within the class you need to call it using `self.predict_proba()`. 2. Create a variable `G`. Call the `self.compute_gradient()` method and assign the result to variable `G`. 3. Create a variable `H`. Call the `self.compute_hessian()` method to get the Hessian and assign the result to variable `H`. 4. Assign the `self.weights_array` variable to the `self.prior_w` variable. By doing so, the current weight values become the previous weight values. 5. Compute the gradient update-step, which is governed by $w_t = w_{t-1} - H^{-1} \cdot G$, where w_t and w_{t-1} are both the variable `self.weights_array`(You are updating the current weights and therefore want to update the values in `self.weights_array`). *Hint*: to implement the part $H^{-1} \cdot G$, use NumPy's `np.linalg.inv()` function and `dot()` method. 6. Note: this method does not return any value.

Step E Complete the `self.check_stop()` method. This is where we implement the stopping criteria. Do the following: 1. Create a variable called `w_old_norm`. Normalize `self.prior_w`. You normalize a vector `v` using the following formula $v/\|v\|$ where $\|v\|$ can be computed using the function `np.linalg.norm(v)`. Assign this result to the variable `w_old_norm`. 2. Create a variable called `w_new_norm`. Normalize `self.weights_array` following the same approach. Assign the result to the variable `w_new_norm`. 3. Create a variable called `diff` and assign it the value `w_old_norm-w_new_norm`. 4. Create a variable called `distance`. Compute $\sqrt{d \cdot d}$ where d is the variable `diff` created in the step above. Note that this uses the dot product. 5. Create a boolean variable called `stop`. Check whether `distance` is less than `self.tolerance`. If so, assign `True` to the variable `stop`. If not, assign `False` to the variable `stop`. 6. Make sure the method returns the variable `stop` (the `return` statement has been provided for you).

[2]:
```python
class LogisticRegressionScratch(object):

    def __init__(self, tolerance = 10**-8, max_iterations = 20):

        self.tolerance = tolerance
        self.max_iterations = max_iterations
        self.weights_array = None # holds current weights and intercept␣
↪(intercept is at the last position)
        self.prior_w = None # holds previous weights and intercept (intercept␣
↪is at the last position)

        # once we are done training, these variables will hold the
        # final values for the weights and intercept
        self.weights = None
        self.intercept = None


    def predict_proba(self, X):
        '''
        Compute probabilities using the inverse logit
        - Inputs: The Nx(K+1) matrix with intercept column X
        - Outputs: Vector of probabilities of length N
        '''

        ### STEP A - WRITE YOUR CODE HERE


        ### solution
        XW = X.dot(self.weights_array)
```

```
        P = (1+np.exp(-1*XW))**-1
        #### end solution


        return P



    def compute_gradient(self, X, Y, P):
        '''
        Computes the gradient vector
        -Inputs:
            - The Nx(K+1) matrix with intercept column X
            - Nx1 vector y (label)
            - Nx1 vector of predictions P
        -Outputs: 1x(K+1) vector of gradients
        '''

        ### STEP B - WRITE YOUR CODE HERE

        ### solution
        G = -1*(Y-P).dot(X)
        ### solution

        return G

    def compute_hessian(self, X, P):
        '''
        computes the Hessian matrix
        -inputs:
            - Nx(K+1) matrix X
            - Nx1 vector of predictions P
        -outputs:
            - KxK Hessian matrix H=X^T * Diag(Q) * X
        '''

        ### STEP C - WRITE YOUR CODE HERE


        ### solution
        Q = P*(1-P)
        XQ = X.T * Q
        H = XQ.dot(X)
        ### solution

        return H
```

```python
    def update_weights(self, X, y):
        '''
        Updates existing weight vector
        -Inputs:
            -Nx(Kx1) matrix X
            -Nx1 vector y
        -Calls predict_proba, compute_gradient and compute_hessian and uses the
        return values to update the weights array
        '''

        ### STEP D - WRITE YOUR CODE HERE


        ### solution
        P = self.predict_proba(X)
        G = self.compute_gradient(X, y, P)
        H = self.compute_hessian(X, P)

        self.prior_w = self.weights_array # save current weights
        self.weights_array = self.weights_array - np.linalg.inv(H).dot(G) #␣
↪update weights
        ### solution



    def check_stop(self):
        '''
        check to see if euclidean distance between old and new weights␣
↪(normalized)
        is less than the tolerance

        returns: True or False on whether stopping criteria is met
        '''

        ### STEP E - WRITE YOUR CODE HERE

        ### solution
        w_old_norm = self.prior_w / np.linalg.norm(self.prior_w)
        w_new_norm = self.weights_array / np.linalg.norm(self.weights_array)

        diff = w_new_norm - w_old_norm

        distance = np.sqrt(diff.dot(diff))
        stop = (distance < self.tolerance)
        ### solution
```

```
        return stop


    def fit(self, X, y):
        '''
        X is the Nx(K-1) data matrix
        Y is the labels, using {0,1} coding
        '''

        #set initial weights - add an extra dimension for the intercept
        self.weights_array = np.zeros(X.shape[1] + 1)

        #Initialize the slope parameter to log(base rate/(1-base rate))
        self.weights_array[-1] = np.log(y.mean() / (1-y.mean()))

        #create a new X matrix that includes a column of ones for the intercept
        X_int = np.hstack((X, np.ones((X.shape[0],1))))

        for i in range(self.max_iterations):
            self.update_weights(X_int, y)

            # check whether we should
            stop = self.check_stop()
            if stop:
                # since we are stopping, lets save the final weights and␣
↪intercept
                self.set_final_weights()
                self.set_final_intercept()
                break


    def set_final_weights(self):
        self.weights = self.weights_array[0:-1]

    def set_final_intercept(self):
        self.intercept = self.weights_array[-1]

    def get_weights(self):
        return self.weights

    def get_intercept(self):
        return self.intercept
```

2.2 Part 2. Use the Class to Train a Logistic Regression Model

Now we will test our implementation of logistic regression.

2.2.1 a. Build Your DataFrame and Define Your ML Problem

We will work with the data set `airbnbData_train`. This data set already has all the necessary preprocessing steps implemented, including one-hot encoding of the categorical variables, scaling of all numerical variable values, and imputing missing values.

[3]:
```
filename = os.path.join(os.getcwd(), "data", "airbnbData_train.csv")
```

Task: Load the data and save it to DataFrame `df`.

[4]:
```
# YOUR CODE HERE

#solution
df = pd.read_csv(filename, header=0)
```

Define the Label Your goal is to train a machine learning model that predicts whether an Airbnb host is a 'super host'. This is an example of supervised learning and is a binary classification problem. In our dataset, our label will be the `host_is_superhost` column and the label will either contain the value `True` or `False`.

Identify Features We have chosen to train the model on a subset of features that can help make with our predictive problem, that is, they can help predict with the host is a super host. Run the following cell to see the list of features.

[5]:
```
feature_list =␣
 ↪['review_scores_rating','review_scores_cleanliness','review_scores_checkin','review_scores_
feature_list
```

[5]:
```
['review_scores_rating',
 'review_scores_cleanliness',
 'review_scores_checkin',
 'review_scores_communication',
 'review_scores_value',
 'host_response_rate',
 'host_acceptance_rate']
```

2.2.2 b. Create Labeled Examples from the Data Set

Task: Our data is ready for modeling. Obtain the feature columns from DataFrame `df` and assign to `X`. Obtain the label column from DataFrame `df` and assign to `y`.

[6]:
```
# YOUR CODE HERE


# solution
y = df['host_is_superhost']
X = df[feature_list]
```

2.2.3 c. Train a Logistic Regression Model

Now that we have our labeled examples, let's test out our logistic regression class. Note: We will not be splitting our data intro training and test data sets

Task: In the code cell below, do the following: 1. Create an instance of `LogisticRegressionScratch()` using default parameters (i.e. do not supply any arguments). Name this instance `lr`. 2. Fit the model `lr` to the training data by calling `lr.fit()` with X and y as arguments.

[7]:
```
# YOUR CODE HERE



#solution
lr = LogisticRegressionScratch()
lr.fit(X, y)
```

Run the code cell below to see the resulting weights and intercept.

[8]:
```
print('The fitted weights and intercept are:')
print(lr.get_weights(), lr.get_intercept())
```

```
The fitted weights and intercept are:
[ 0.56690005  0.492255    0.201587    0.25551467 -0.00590516  1.71592957
  0.26478817] -1.829062262272181
```

2.3 Part 3: Compare with Scikit-Learn's Implementation

Now let's compare our logistic regression implementation with the `sklearn` logistic regression implementation. Note that by default scikit-learn uses a different optimization technique. However, our goal is to compare our resulting weights and intercept with those of scikit-learn's implementation, and these should be the same.

Task: In the code cell below, write code to does the following: 1. Create the scikit-learn `LogisticRegression` model object below and assign to variable `lr_sk`. Use `C=10**10` as the argument to `LogisticRegression()`.

2. Fit the model `lr_sk` to the training data by calling `lr_sk.fit()` with X and y as arguments.

[9]:
```
# YOUR CODE HERE

# solution
lr_sk = LogisticRegression(C=10**10)
lr_sk.fit(X, y)
```

[9]:
```
LogisticRegression(C=10000000000, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Run the code cell below to see the resulting weights and intercept. Compare these to our implementation.

[10]:
```python
print('The fitted weights and intercept with sklearn are:')
print(lr_sk.coef_, lr_sk.intercept_)
```

```
The fitted weights and intercept with sklearn are:
[[ 0.56691547  0.49224905  0.20150113  0.25563246 -0.005929    1.71592022
   0.26479199]] [-1.82906726]
```

Let's also check the efficiency (or run time) of both methods. We will use the magic function `%timeit` to do this

Task: Use the `%timeit` magic function to fit the logistic regression model `lr` on the training data. Hint: use `%timeit` on `lr.fit(X, y)`.

[11]:
```python
# YOUR CODE HERE

# solution
%timeit lr.fit(df[feature_list], df['host_is_superhost'])
```

```
The slowest run took 14.02 times longer than the fastest. This could mean that
an intermediate result is being cached.
182 ms ś 240 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

Task: Use the `%timeit` magic function to fit the logistic regression model `lr_sk` on the training data. Take a look and see which one is faster?

[12]:
```python
# YOUR CODE HERE



# solution
%timeit lr_sk.fit(df[feature_list], df['host_is_superhost'])
```

```
503 ms ś 227 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```