ML0120EN-1.4-Review-LogisticRegressionwithTensorFlow

March 31, 2020

LOGISTIC REGRESSION WITH TENSORFLOW

0.1 Table of Contents

Logistic Regression is one of most important techniques in data science. It is usually used to solve the classic classification problem.

This lesson covers the following concepts of Logistics Regression:

What is different between Linear and Logistic Regression?

While Linear Regression is suited for estimating continuous values (e.g. estimating house price), it is not the best tool for predicting the class in which an observed data point belongs. In order to provide estimate for classification, we need some sort of guidance on what would be the most probable class for that data point. For this, we use Logistic Regression.

Recall linear regression: Linear regression finds a function that relates a continuous dependent variable, y, to some predictors (independent variables x1, x2, etc.). Simple linear regression assumes a function of the form:

$$y = w0 + w1 \times x1 + w2 \times x2 + \cdots$$

and finds the values of w0, w1, w2, etc. The term w0 is the "intercept" or "constant term" (it's shown as b in the formula below):

$$Y = WX + b$$

Logistic Regression is a variation of Linear Regression, useful when the observed dependent variable, y, is categorical. It produces a formula that predicts the probability of the class label as a function of the independent variables.

Despite the name logistic regression, it is actually a probabilistic classification model. Logistic regression fits a special s-shaped curve by taking the linear regression and transforming the numeric estimate into a probability with the following function:

$$P(SomeClass) = \theta(y) = \frac{e^y}{1 + e^y} = exp(y)/(1 + exp(y)) = p$$

which produces p-values between 0 (as y approaches minus infinity $-\infty$) and 1 (as y approaches plus infinity $+\infty$). This now becomes a special kind of non-linear regression.

In this equation, y is the regression result (the sum of the variables weighted by the coefficients), exp is the exponential function and $\theta(y)$ is the logistic function, also called logistic curve. It is a common "S" shape (sigmoid curve), and was first developed for modeling population growth.

You might also have seen this function before, in another configuration:

$$P(SomeClass) = \theta(y) = \frac{1}{1 + e^{-y}}$$

So, briefly, Logistic Regression passes the input through the logistic/sigmoid function but then treats the result as a probability:

Utilizing Logistic Regression in TensorFlow

For us to utilize Logistic Regression in TensorFlow, we first need to import the required libraries. To do so, you can run the code cell below.

[1]:
```
import tensorflow as tf
import pandas as pd
import numpy as np
import time
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Next, we will load the dataset we are going to use. In this case, we are utilizing the iris dataset, which is inbuilt – so there's no need to do any preprocessing and we can jump right into manipulating it. We separate the dataset into xs and ys, and then into training xs and ys and testing xs and ys, (pseudo)randomly.

Understanding the Data

Iris Dataset:

This dataset was introduced by British Statistician and Biologist Ronald Fisher, it consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). In total it has 150 records under five attributes - petal length, petal width, sepal length, sepal width and species. Dataset source

Attributes Independent Variable

petal length

petal width

sepal length

sepal width

Dependent Variable

Species

Iris setosa

Iris virginica

Iris versicolor

</li>

[2]:
```
iris = load_iris()
iris_X, iris_y = iris.data[:-1,:], iris.target[:-1]
iris_y= pd.get_dummies(iris_y).values
trainX, testX, trainY, testY = train_test_split(iris_X, iris_y, test_size=0.33,␣
 ↪random_state=42)
```

Now we define x and y. These placeholders will hold our iris data (both the features and label matrices), and help pass them along to different parts of the algorithm. You can consider placeholders as empty shells into which we insert our data. We also need to give them shapes which correspond to the shape of our data. Later, we will insert data into these placeholders by "feeding" the placeholders the data via a "feed_dict" (Feed Dictionary).

Why use Placeholders?

This feature of TensorFlow allows us to create an algorithm which accepts data and knows something about the shape of the data without knowing the amount of data going in.

When we insert "batches" of data in training, we can easily adjust how many examples we train on in a single step without changing the entire algorithm.

[3]:
```
# numFeatures is the number of features in our input data.
# In the iris dataset, this number is '4'.
numFeatures = trainX.shape[1]

# numLabels is the number of classes our data points can be in.
# In the iris dataset, this number is '3'.
numLabels = trainY.shape[1]


# Placeholders
# 'None' means TensorFlow shouldn't expect a fixed number in that dimension
X = tf.placeholder(tf.float32, [None, numFeatures]) # Iris has 4 features, so X␣
 ↪is a tensor to hold our data.
yGold = tf.placeholder(tf.float32, [None, numLabels]) # This will be our␣
 ↪correct answers matrix for 3 classes.
```

Set model weights and bias

Much like Linear Regression, we need a shared variable weight matrix for Logistic Regression. We initialize both W and b as tensors full of zeros. Since we are going to learn W and b, their initial value does not matter too much. These variables are the objects which define the structure of our regression model, and we can save them after they have been trained so we can reuse them later.

We define two TensorFlow variables as our parameters. These variables will hold the weights and biases of our logistic regression and they will be continually updated during training.

Notice that W has a shape of [4, 3] because we want to multiply the 4-dimensional input vectors by it to produce 3-dimensional vectors of evidence for the difference classes. b has a shape of [3] so we can add it to the output. Moreover, unlike our placeholders above which are essentially empty shells waiting to be fed data, TensorFlow variables need to be initialized with values, e.g. with zeros.

[4]:
```
W = tf.Variable(tf.zeros([4, 3]))  # 4-dimensional input and  3 classes
b = tf.Variable(tf.zeros([3])) # 3-dimensional output [0,0,1],[0,1,0],[1,0,0]
```

[5]:
```
#Randomly sample from a normal distribution with standard deviation .01

weights = tf.Variable(tf.random_normal([numFeatures,numLabels], # dx3
                                       mean=0,
                                       stddev=0.01,
                                       name="weights"))

bias = tf.Variable(tf.random_normal([1,numLabels], # 1x3
                                    mean=0,
                                    stddev=0.01,
                                    name="bias"))
```

Logistic Regression model

We now define our operations in order to properly run the Logistic Regression. Logistic regression is typically thought of as a single equation:

$$y = sigmoid(WX + b)$$

However, for the sake of clarity, we can have it broken into its three main components: - a weight times features matrix multiplication operation (OP), - a summation of the weighted features and a bias term, - and finally the application of a sigmoid function.

As such, you will find these components defined as three separate operations below.

[6]:
```
# Three-component breakdown of the Logistic Regression equation.
# Note that these feed into each other.
apply_weights_OP = tf.matmul(X, weights, name="apply_weights") # nx3
add_bias_OP = tf.add(apply_weights_OP, bias, name="add_bias") # nx3
activation_OP = tf.nn.sigmoid(add_bias_OP, name="activation") # nx3
```

As we have seen before, the function we are going to use is the logistic function $(\frac{1}{1+e^{-Wx}})$, which is fed the input data after applying weights and bias. In TensorFlow, this function is implemented as the nn.sigmoid function. Effectively, this fits the weighted input with bias into a 0-100 percent curve, which is the probability function we want.

Training

The learning algorithm is how we search for the best weight vector (**w**). This search is an optimization problem looking for the hypothesis that optimizes an error/cost measure.

What tell us our model is bad?
The Cost or Loss of the model, so what we want is to minimize that.

What is the cost function in our model?
The cost function we are going to utilize is the Squared Mean Error loss function.

How to minimize the cost function?
We can't use least-squares linear regression here, so we will use gradient descent instead. Specifically, we will use batch gradient descent which calculates the gradient from all data points in the data set.

Cost function

Before defining our cost function, we need to define how long we are going to train and how should we define the learning rate.

[7]:
```
# Number of Epochs in our training
numEpochs = 700

# Defining our learning rate iterations (decay)
learningRate = tf.train.exponential_decay(learning_rate=0.0008,
                                          global_step= 1,
                                          decay_steps=trainX.shape[0],
                                          decay_rate= 0.95,
                                          staircase=True)
```

[8]:
```
#Defining our cost function - Squared Mean Error
cost_OP = tf.nn.l2_loss(activation_OP-yGold, name="squared_error_cost")

#Defining our Gradient Descent
training_OP = tf.train.GradientDescentOptimizer(learningRate).minimize(cost_OP)
```

Now we move on to actually running our operations. We will start with the operations involved in the prediction phase (*i.e.* the logistic regression itself).

First, we need to initialize our weights and biases with zeros or random values via the inbuilt Initialization Op, tf.initialize_all_variables(). This Initialization Op will become a node in our computational graph, and when we put the graph into a session, then the Op will run and create the variables.

[9]:
```
# Create a tensorflow session
sess = tf.Session()

# Initialize our weights and biases variables.
init_OP = tf.global_variables_initializer()

# Initialize all tensorflow variables
sess.run(init_OP)
```

We also want some additional operations to keep track of our model's efficiency over time. We can do this like so:

[10]:
```
# argmax(activation_OP, 1) returns the label with the most probability
# argmax(yGold, 1) is the correct label
correct_predictions_OP = tf.equal(tf.argmax(activation_OP,1),tf.argmax(yGold,1))

# If every false prediction is 0 and every true prediction is 1, the average␣
↪returns us the accuracy
accuracy_OP = tf.reduce_mean(tf.cast(correct_predictions_OP, "float"))

# Summary op for regression output
activation_summary_OP = tf.summary.histogram("output", activation_OP)

# Summary op for accuracy
accuracy_summary_OP = tf.summary.scalar("accuracy", accuracy_OP)

# Summary op for cost
cost_summary_OP = tf.summary.scalar("cost", cost_OP)

# Summary ops to check how variables (W, b) are updating after each iteration
weightSummary = tf.summary.histogram("weights", weights.eval(session=sess))
biasSummary = tf.summary.histogram("biases", bias.eval(session=sess))

# Merge all summaries
merged = tf.summary.merge([activation_summary_OP, accuracy_summary_OP,␣
↪cost_summary_OP, weightSummary, biasSummary])

# Summary writer
writer = tf.summary.FileWriter("summary_logs", sess.graph)
```

Now we can define and run the actual training loop, like this:

[11]:
```
# Initialize reporting variables
cost = 0
diff = 1
epoch_values = []
accuracy_values = []
cost_values = []

# Training epochs
for i in range(numEpochs):
    if i > 1 and diff < .0001:
        print("change in cost %g; convergence."%diff)
        break
    else:
        # Run training step
        step = sess.run(training_OP, feed_dict={X: trainX, yGold: trainY})
        # Report occasional stats
        if i % 10 == 0:
            # Add epoch to epoch_values
            epoch_values.append(i)
            # Generate accuracy stats on test data
            train_accuracy, newCost = sess.run([accuracy_OP, cost_OP],␣
 ↪feed_dict={X: trainX, yGold: trainY})
            # Add accuracy to live graphing variable
            accuracy_values.append(train_accuracy)
            # Add cost to live graphing variable
            cost_values.append(newCost)
            # Re-assign values for variables
            diff = abs(newCost - cost)
            cost = newCost

            #generate print statements
            print("step %d, training accuracy %g, cost %g, change in cost␣
 ↪%g"%(i, train_accuracy, newCost, diff))


# How well do we perform on held-out test data?
print("final accuracy on test set: %s" %str(sess.run(accuracy_OP,
                                                     feed_dict={X: testX,
                                                                yGold: testY})))
```

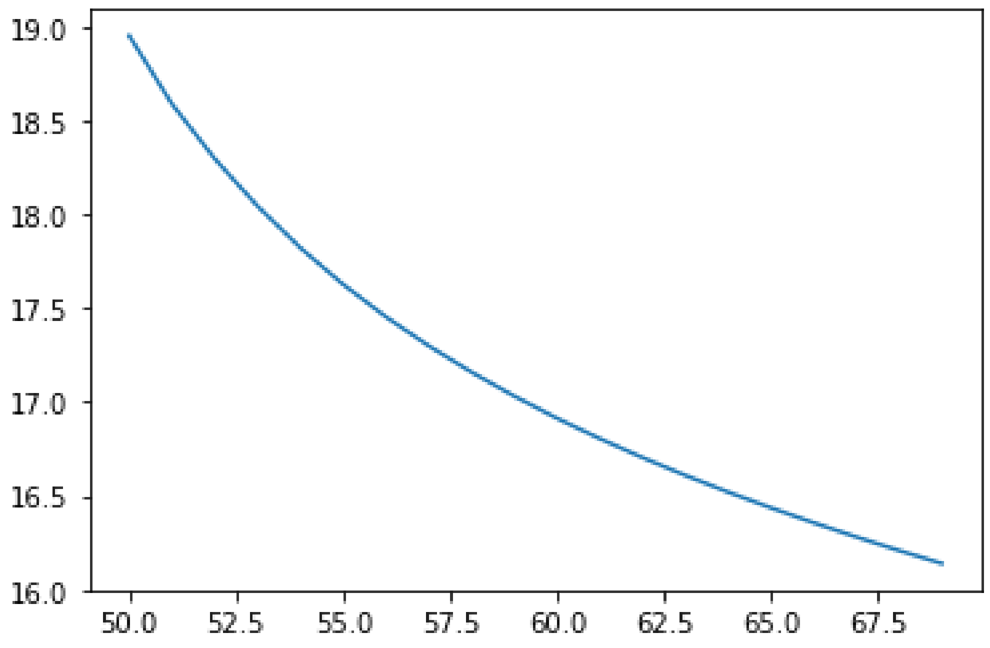
```
step 0, training accuracy 0.353535, cost 33.7687, change in cost 33.7687
step 10, training accuracy 0.545455, cost 30.1521, change in cost 3.61659
step 20, training accuracy 0.646465, cost 28.2019, change in cost 1.95015
step 30, training accuracy 0.646465, cost 26.5519, change in cost 1.65006
step 40, training accuracy 0.646465, cost 25.1793, change in cost 1.37252
step 50, training accuracy 0.646465, cost 24.045, change in cost 1.13433
step 60, training accuracy 0.646465, cost 23.1061, change in cost 0.938871
step 70, training accuracy 0.646465, cost 22.3242, change in cost 0.781935
step 80, training accuracy 0.646465, cost 21.6673, change in cost 0.65695
step 90, training accuracy 0.646465, cost 21.1099, change in cost 0.557404
step 100, training accuracy 0.656566, cost 20.6321, change in cost 0.477753
step 110, training accuracy 0.666667, cost 20.2185, change in cost 0.413572
step 120, training accuracy 0.666667, cost 19.8571, change in cost 0.361431
step 130, training accuracy 0.666667, cost 19.5384, change in cost 0.318691
step 140, training accuracy 0.666667, cost 19.255, change in cost 0.283356
step 150, training accuracy 0.666667, cost 19.0012, change in cost 0.253885
step 160, training accuracy 0.686869, cost 18.7721, change in cost 0.229105
step 170, training accuracy 0.686869, cost 18.564, change in cost 0.208103
step 180, training accuracy 0.69697, cost 18.3738, change in cost 0.190168
step 190, training accuracy 0.717172, cost 18.199, change in cost 0.174746
step 200, training accuracy 0.717172, cost 18.0376, change in cost 0.161394
step 210, training accuracy 0.737374, cost 17.8879, change in cost 0.149763
step 220, training accuracy 0.737374, cost 17.7483, change in cost 0.13957
step 230, training accuracy 0.747475, cost 17.6177, change in cost 0.130587
step 240, training accuracy 0.757576, cost 17.4951, change in cost 0.122633
step 250, training accuracy 0.777778, cost 17.3795, change in cost 0.115545
step 260, training accuracy 0.787879, cost 17.2703, change in cost 0.109215
step 270, training accuracy 0.787879, cost 17.1668, change in cost 0.103519
step 280, training accuracy 0.787879, cost 17.0684, change in cost 0.0983849
step 290, training accuracy 0.787879, cost 16.9747, change in cost 0.0937309
step 300, training accuracy 0.79798, cost 16.8852, change in cost 0.0895042
step 310, training accuracy 0.79798, cost 16.7996, change in cost 0.0856438
step 320, training accuracy 0.79798, cost 16.7174, change in cost 0.0821095
step 330, training accuracy 0.79798, cost 16.6386, change in cost 0.0788612
step 340, training accuracy 0.818182, cost 16.5627, change in cost 0.0758705
step 350, training accuracy 0.828283, cost 16.4896, change in cost 0.0731087
step 360, training accuracy 0.828283, cost 16.4191, change in cost 0.0705452
step 370, training accuracy 0.838384, cost 16.3509, change in cost 0.0681648
step 380, training accuracy 0.838384, cost 16.2849, change in cost 0.0659504
step 390, training accuracy 0.848485, cost 16.2211, change in cost 0.063879
step 400, training accuracy 0.848485, cost 16.1591, change in cost 0.0619411
step 410, training accuracy 0.848485, cost 16.099, change in cost 0.0601254
step 420, training accuracy 0.848485, cost 16.0406, change in cost 0.0584164
step 430, training accuracy 0.858586, cost 15.9838, change in cost 0.0568085
step 440, training accuracy 0.858586, cost 15.9285, change in cost 0.0552893
step 450, training accuracy 0.868687, cost 15.8746, change in cost 0.0538568
step 460, training accuracy 0.878788, cost 15.8221, change in cost 0.0524998
step 470, training accuracy 0.878788, cost 15.7709, change in cost 0.0512075
step 480, training accuracy 0.878788, cost 15.7209, change in cost 0.0499868
step 490, training accuracy 0.878788, cost 15.6721, change in cost 0.0488234
step 500, training accuracy 0.878788, cost 15.6244, change in cost 0.0477123
step 510, training accuracy 0.878788, cost 15.5777, change in cost 0.0466576
step 520, training accuracy 0.878788, cost 15.5321, change in cost 0.0456438
step 530, training accuracy 0.888889, cost 15.4874, change in cost 0.0446777
step 540, training accuracy 0.89899, cost 15.4437, change in cost 0.0437536
step 550, training accuracy 0.89899, cost 15.4008, change in cost 0.0428648
step 560, training accuracy 0.89899, cost 15.3588, change in cost 0.0420151
step 570, training accuracy 0.89899, cost 15.3176, change in cost 0.0411949
step 580, training accuracy 0.89899, cost 15.2772, change in cost 0.0404091
step 590, training accuracy 0.909091, cost 15.2375, change in cost 0.0396528
step 600, training accuracy 0.909091, cost 15.1986, change in cost 0.0389233
step 610, training accuracy 0.909091, cost 15.1604, change in cost 0.0382185
step 620, training accuracy 0.909091, cost 15.1228, change in cost 0.0375414
step 630, training accuracy 0.909091, cost 15.086, change in cost 0.0368843
step 640, training accuracy 0.909091, cost 15.0497, change in cost 0.036252
step 650, training accuracy 0.909091, cost 15.0141, change in cost 0.0356369
step 660, training accuracy 0.909091, cost 14.979, change in cost 0.0350447
step 670, training accuracy 0.909091, cost 14.9446, change in cost 0.0344706
step 680, training accuracy 0.909091, cost 14.9106, change in cost 0.0339127
step 690, training accuracy 0.909091, cost 14.8773, change in cost 0.0333729
```

```
final accuracy on test set: 0.9
```

Why don't we plot the cost to see how it behaves?

[12]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.plot([np.mean(cost_values[i-50:i]) for i in range(len(cost_values))])
plt.show()
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/numpy/core/fromnumeric.py:2957: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/numpy/core/_methods.py:80: RuntimeWarning: invalid value encountered in
double_scalars
  ret = ret.dtype.type(ret / rcount)
```



Assuming no parameters were changed, you should reach a peak accuracy of 90% at the end of training, which is commendable. Try changing the parameters such as the length of training, and maybe some operations to see how the model behaves. Does it take much longer? How is the performance?

0.2 Want to learn more?

Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises.

The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud.

Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.___Watson Studio___ is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

0.2.1 Thanks for completing this lesson!

This is the end of **Logistic Regression with TensorFlow** notebook. Hopefully, now you have a deeper understanding of Logistic Regression and how its structure and flow work. Thank you for reading this notebook and good luck on your studies.

Created by: Saeed Aghabozorgi , Walter Gomes de Amorim Junior , Victor Barros Costa