ML0120EN-1.4-Review-LogisticRegressionwithTensorFlow

October 25, 2020

LOGISTIC REGRESSION WITH TENSORFLOW

Objective for this Notebook

1. What is different between Linear and Logistic Regression?
2. Utilizing Logistic Regression in TensorFlow.
3. Training the model

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:

Table of Contents

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:

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

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:

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

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

We begin by installing TensorFlow version 2.2.0 and its required prerequistes.

[ ]:
```
!pip install grpcio==1.24.3
!pip install tensorflow==2.2.0
```

Restart kernel for latest version of TensorFlow to be activated

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

[2]:
```
if not tf.__version__ == '2.2.0':
    print(tf.__version__)
    raise ValueError('please upgrade to TensorFlow 2.2.0, or restart your␣
↪Kernel (Kernel->Restart & Clear Output)')
```

IMPORTANT! => Please restart the kernel by clicking on "Kernel"->"Restart and Clear Outout" and wait until all output disapears. Then your changes are beeing picked up

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

[21]:
```
iris = load_iris()
iris_X, iris_y = iris.data, iris.target   # Split 2-D array into X and y.
iris_y= pd.get_dummies(iris_y).values     # Turn y into a categorical value.

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 variables will hold our iris data (both the features and label matrices) We also need to give them shapes which correspond to the shape of our data.

[23]:
```
# numFeatures is the number of features in our input data. In the iris dataset,␣
 ↪this number is '4'.
numFeatures = trainX.shape[1]
print('numFeatures is : ', numFeatures )

# numLabels is the number of classes our data points can be in. In the iris␣
 ↪dataset, this number is '3'.
numLabels = trainY.shape[1]
print('numLabels is : ', numLabels )
```

```
numFeatures is :  4
numLabels is :  3
```

[25]:
```
# Iris has 4 features, so X is a tensor to hold our data.
X = tf.Variable( np.identity(numFeatures), tf.TensorShape(numFeatures),␣
 ↪dtype='float32')

# This will be our correct answers matrix for 3 classes.
yGold = tf.Variable(np.array([1,1,1]), shape=tf.TensorShape(numLabels),␣
 ↪dtype='float32')
```

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. TensorFlow variables need to be initialized with values, e.g. with zeros.

[29]:
```
W = tf.Variable(tf.zeros([numFeatures, numLabels]))  # 4-dimensional input and␣
 ↪3 classes
b = tf.Variable(tf.zeros([numLabels])) # 3-dimensional output␣
 ↪[0,0,1],[0,1,0],[1,0,0]
```

[32]:
```
#Randomly sample from a normal distribution with .01 standard deviation

weights = tf.Variable(tf.random.normal([numFeatures,numLabels],
                                       mean=0.,
                                       stddev=0.01,
                                       name="weights"),dtype='float32')


bias = tf.Variable(tf.random.normal([1,numLabels],
                                    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:

$$= \sigma(XW + b)$$

However, for the sake of clarity, we can have it broken into its three main components:

- a weight times features matrix multiplication operation, $X \in \mathbb{R}^{numSamples \times numFeatures}$, $W \in \mathbb{R}^{numFeatures \times numLabels}$
- a summation of the weighted features and a bias term, $b \in \mathbb{R}^{numLabels}$
- and finally the application of a sigmoid function $\sigma(\cdot)$.

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

[33]:
```
def logistic_regression(x):
    apply_weights_OP = tf.matmul(X, weights, name="apply_weights")
    add_bias_OP = tf.add(apply_weights_OP, bias, name="add_bias")
    activation_OP = tf.nn.sigmoid(add_bias_OP, name="activation")
    return activation_OP
```

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.

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.

[34]:
```
numEpochs = 700
learningRate = tf.keras.optimizers.schedules.
 ↪ExponentialDecay(initial_learning_rate=0.0008,
                                                          decay_steps=trainX.shape[0],
                                                          decay_rate= 0.95,
                                                          staircase=True)
```

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

[35]:
```
loss_object = tf.keras.losses.MeanSquaredLogarithmicError()
optimizer = tf.keras.optimizers.SGD(learningRate)
```

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

[36]:
```
def accuracy(y_pred, y_true):
    print('y_pred : ',y_pred)
    print('y_true : ',y_true)
    correct_prediction = tf.equal(tf.argmax(y_pred, -1), tf.argmax(y_true, -1))
    return tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

we first wrap computation inside a GradientTape for automatic differentiation. Then we compute gradients and update W and b.

[37]:
```
def run_optimization(x, y):
    with tf.GradientTape() as g:
        pred = logistic_regression(x)
        loss = loss_object(pred, y)
    gradients = g.gradient(loss, [weights, bias])
    optimizer.apply_gradients(zip(gradients, [weights, bias]))
```

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).

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

[39]:
```
display_step = 100
epoch_values = []
accuracy_values = []
loss_values = []
loss = 0
diff = 1

for i in range(numEpochs):
    if i > 1 and diff < .000001:
        print("change in loss %g; convergence."%diff)
        break
    else:
        # Run training step
        run_optimization(X, yGold)

        # Report occasional stats
        if i % display_step == 0:
            # Add epoch to epoch_values
            epoch_values.append(i)

            pred = logistic_regression(X)
```

```
            newLoss = loss_object(pred, yGold)
            # Add loss to live graphing variable
            loss_values.append(newLoss)

            # Generate accuracy stats on test data
            acc = accuracy(pred, yGold)
            accuracy_values.append(acc)


            # Re-assign values for variables
            diff = abs(newLoss - loss)
            loss = newLoss

            #generate print statements
            print("step %d, training accuracy %g, loss %g, change in loss␣
 ↪%g"%(i, acc, newLoss, diff))


print("final accuracy on test set: %s" %str(acc))
```

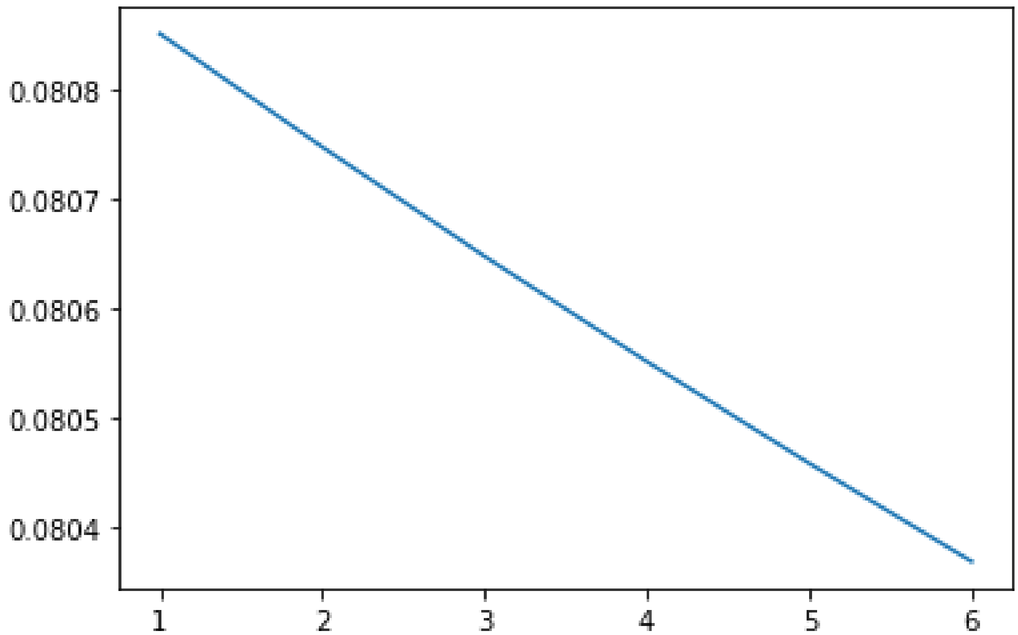
```
y_pred :  tf.Tensor(
[[0.49993765 0.5061819  0.5073769 ]
 [0.5026681  0.50406975 0.50978297]
 [0.5037266  0.5075008  0.5048494 ]
 [0.50453365 0.5052603  0.50443   ]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 0, training accuracy 0, loss 0.0808507, change in loss 0.0808507
y_pred :  tf.Tensor(
[[0.5004914  0.50672936 0.50792277]
 [0.50322086 0.504618   0.510328  ]
 [0.504279   0.5080479  0.5053962 ]
 [0.50508577 0.50580806 0.5049769 ]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 100, training accuracy 0, loss 0.0806435, change in loss 0.000207208
y_pred :  tf.Tensor(
[[0.50101656 0.50724864 0.50844055]
 [0.50374514 0.5051379  0.5108449 ]
 [0.50480294 0.5085667  0.5059148 ]
 [0.50560945 0.5063276  0.50549567]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 200, training accuracy 0, loss 0.0804472, change in loss 0.000196226
y_pred :  tf.Tensor(
[[0.5015147  0.5077411  0.50893164]
 [0.5042424  0.50563115 0.51133513]
```

```
 [0.5052999  0.5090587  0.5064067 ]
 [0.50610614 0.50682044 0.50598776]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 300, training accuracy 0, loss 0.0802614, change in loss 0.00018584
y_pred :  tf.Tensor(
[[0.5019872  0.5082083  0.50939745]
 [0.50471413 0.506099   0.51180017]
 [0.50577134 0.5095255  0.50687337]
 [0.5065773  0.5072879  0.5064545 ]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 400, training accuracy 0, loss 0.0800854, change in loss 0.000176027
y_pred :  tf.Tensor(
[[0.5024355  0.5086515  0.50983936]
 [0.50516164 0.5065428  0.5122413 ]
 [0.50621855 0.5099683  0.507316  ]
 [0.5070243  0.5077314  0.5068973 ]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 500, training accuracy 0, loss 0.0799186, change in loss 0.000166751
y_pred :  tf.Tensor(
[[0.50286084 0.50907195 0.51025856]
 [0.5055862  0.50696385 0.51265985]
 [0.5066428  0.5103883  0.50773597]
 [0.5074483  0.5081521  0.50731736]], shape=(4, 3), dtype=float32)
y_true :  <tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([1.,
1., 1.], dtype=float32)>
step 600, training accuracy 0, loss 0.0797606, change in loss 0.00015799
final accuracy on test set: tf.Tensor(0.0, shape=(), dtype=float32)
```

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

[40]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.plot([np.mean(loss_values[i-50:i]) for i in range(len(loss_values))])
plt.show()
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/numpy/core/fromnumeric.py:3373: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/numpy/core/_methods.py:170: RuntimeWarning: invalid value encountered
in double_scalars
  ret = ret.dtype.type(ret / rcount)
```



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: Romeo Kienzler , Saeed Aghabozorgi , Walter Gomes de Amorim Junior , Victor Barros Costa

Updated to TF 2.X by Samaya Madhavan

0.3 Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-09-21	2.0	Srishti	Migrated Lab to Markdown and added to course repo in GitLab

##