
(https://www.bigdatauniversity.com)

LOGISTIC REGRESSION WITH TENSORFLOW

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 n0t 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):

$$ Y = W X + b $$

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.

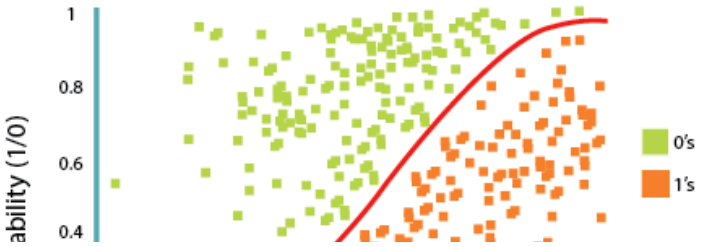
Despite the name logistic *regression*, it is actually a **probabilistic classification** model. Logistic regression fits a special s-shaped curve by taking the linear regression and transforming the numeric estimate into a probability with the following function:
$$ ProbabilityOfaClass = \theta(y) = \frac{e^y}{1 + e^y} = exp(y) / (1 + exp(y)) = p $$

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 (http://en.wikipedia.org/wiki/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

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.

In [1]:

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorf
low/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) o
r '1type' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

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 (https://archive.ics.uci.edu/ml/datasets/iris)

Attributes Independent Variable

- petal length
- petal width
- sepal length
- sepal width

Dependent Variable

- Species
  - Iris setosa
  - Iris virginica
  - Iris versicolor

In [2]:

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?

1. 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.
2. 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.

In [3]:

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.

In [4]:

In [5]:

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:
$$ ŷ =sigmoid(WX+b) $$

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

- a weight times features matrix multiplication operation,
- 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.

In [6]:

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 (${\bf w}$). 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 (http://en.wikipedia.org/wiki/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.

In [7]:

In [8]:

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.

In [9]:

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

In [10]:

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

In [11]:

```
step 0, training accuracy 0.353535, cost 34.6577, change in cost 34.65
77
step 10, training accuracy 0.535354, cost 29.9523, change in cost 4.70
534
step 20, training accuracy 0.646465, cost 28.0384, change in cost 1.91
39
step 30, training accuracy 0.646465, cost 26.4183, change in cost 1.62
013
step 40, training accuracy 0.646465, cost 25.069, change in cost 1.349
35
step 50, training accuracy 0.646465, cost 23.9524, change in cost 1.11
654
step 60, training accuracy 0.646465, cost 23.0272, change in cost 0.92
5182
step 70, training accuracy 0.646465, cost 22.2559, change in cost 0.77
1322
step 80, training accuracy 0.646465, cost 21.6073, change in cost 0.64
8619
step 90, training accuracy 0.646465, cost 21.0565, change in cost 0.55
0783
step 100, training accuracy 0.666667, cost 20.5841, change in cost 0.4
72424
step 110, training accuracy 0.666667, cost 20.1748, change in cost 0.4
09229
step 120, training accuracy 0.666667, cost 19.817, change in cost 0.35
7843
step 130, training accuracy 0.666667, cost 19.5013, change in cost 0.3
15695
step 140, training accuracy 0.666667, cost 19.2205, change in cost 0.2
80825
step 150, training accuracy 0.676768, cost 18.9688, change in cost 0.2
51726
step 160, training accuracy 0.686869, cost 18.7415, change in cost 0.2
27245
step 170, training accuracy 0.686869, cost 18.535, change in cost 0.20
6486
step 180, training accuracy 0.69697, cost 18.3463, change in cost 0.18
8753
step 190, training accuracy 0.717172, cost 18.1728, change in cost 0.1
73496
step 200, training accuracy 0.717172, cost 18.0125, change in cost 0.1
60284
step 210, training accuracy 0.737374, cost 17.8637, change in cost 0.1
48769
step 220, training accuracy 0.747475, cost 17.725, change in cost 0.13
868
step 230, training accuracy 0.757576, cost 17.5953, change in cost 0.1
2978
step 240, training accuracy 0.757576, cost 17.4734, change in cost 0.1
21901
step 250, training accuracy 0.787879, cost 17.3585, change in cost 0.1
1488
step 260, training accuracy 0.787879, cost 17.2499, change in cost 0.1
08599
step 270, training accuracy 0.787879, cost 17.1469, change in cost 0.1
02957
step 280, training accuracy 0.787879, cost 17.0491, change in cost 0.0
```

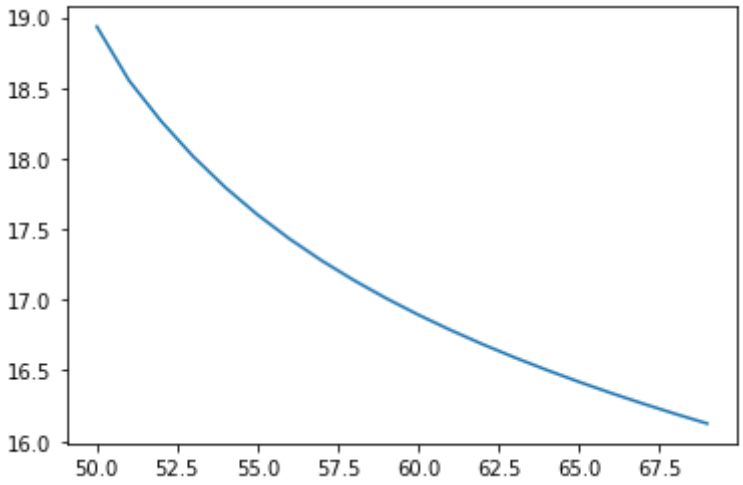
978642
step 290, training accuracy 0.79798, cost 16.9558, change in cost 0.09
32484
step 300, training accuracy 0.79798, cost 16.8668, change in cost 0.08
90541
step 310, training accuracy 0.79798, cost 16.7815, change in cost 0.08
52261
step 320, training accuracy 0.79798, cost 16.6998, change in cost 0.08
17146
step 330, training accuracy 0.79798, cost 16.6213, change in cost 0.07
84969
step 340, training accuracy 0.818182, cost 16.5458, change in cost 0.0
755234
step 350, training accuracy 0.828283, cost 16.473, change in cost 0.07
27825
step 360, training accuracy 0.838384, cost 16.4028, change in cost 0.0
702362
step 370, training accuracy 0.838384, cost 16.3349, change in cost 0.0
678749
step 380, training accuracy 0.838384, cost 16.2692, change in cost 0.0
6567
step 390, training accuracy 0.848485, cost 16.2056, change in cost 0.0
636158
step 400, training accuracy 0.848485, cost 16.1439, change in cost 0.0
616875
step 410, training accuracy 0.848485, cost 16.0841, change in cost 0.0
598831
step 420, training accuracy 0.848485, cost 16.0259, change in cost 0.0
581856
step 430, training accuracy 0.858586, cost 15.9693, change in cost 0.0
565863
step 440, training accuracy 0.868687, cost 15.9142, change in cost 0.0
550795
step 450, training accuracy 0.868687, cost 15.8605, change in cost 0.0
536528
step 460, training accuracy 0.878788, cost 15.8082, change in cost 0.0
523005
step 470, training accuracy 0.878788, cost 15.7572, change in cost 0.0
510216
step 480, training accuracy 0.878788, cost 15.7074, change in cost 0.0
498037
step 490, training accuracy 0.878788, cost 15.6588, change in cost 0.0
48646
step 500, training accuracy 0.878788, cost 15.6112, change in cost 0.0
475416
step 510, training accuracy 0.878788, cost 15.5647, change in cost 0.0
464907
step 520, training accuracy 0.888889, cost 15.5193, change in cost 0.0
454836
step 530, training accuracy 0.89899, cost 15.4747, change in cost 0.04
45213
step 540, training accuracy 0.89899, cost 15.4311, change in cost 0.04
3602
step 550, training accuracy 0.89899, cost 15.3884, change in cost 0.04
27189
step 560, training accuracy 0.89899, cost 15.3465, change in cost 0.04
18701

```
step 570, training accuracy 0.89899, cost 15.3055, change in cost 0.04
10566
step 580, training accuracy 0.909091, cost 15.2652, change in cost 0.0
402746
step 590, training accuracy 0.909091, cost 15.2257, change in cost 0.0
395193
step 600, training accuracy 0.909091, cost 15.1869, change in cost 0.0
387926
step 610, training accuracy 0.909091, cost 15.1488, change in cost 0.0
380945
step 620, training accuracy 0.909091, cost 15.1114, change in cost 0.0
374174
step 630, training accuracy 0.909091, cost 15.0746, change in cost 0.0
367632
step 640, training accuracy 0.909091, cost 15.0385, change in cost 0.0
361338
step 650, training accuracy 0.909091, cost 15.003, change in cost 0.03
55225
step 660, training accuracy 0.909091, cost 14.968, change in cost 0.03
49312
step 670, training accuracy 0.909091, cost 14.9337, change in cost 0.0
343599
step 680, training accuracy 0.909091, cost 14.8999, change in cost 0.0
338049
step 690, training accuracy 0.909091, cost 14.8666, change in cost 0.0
33267
final accuracy on test set: 0.9
```

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

In [12]:

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/numpy/c
ore/fromnumeric.py:3335: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/numpy/c
ore/_methods.py:161: RuntimeWarning: invalid value encountered in doub
le_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?

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 (https://cocl.us/ML0120EN_PAI).

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 (https://cocl.us/ML0120EN_DSX).This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

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 (https://br.linkedin.com/in/walter-gomes-de-amorim-junior-624726121) , Walter Gomes de Amorim Junior (https://br.linkedin.com/in/walter-gomes-de-amorim-junior-624726121) , Victor Barros Costa

---