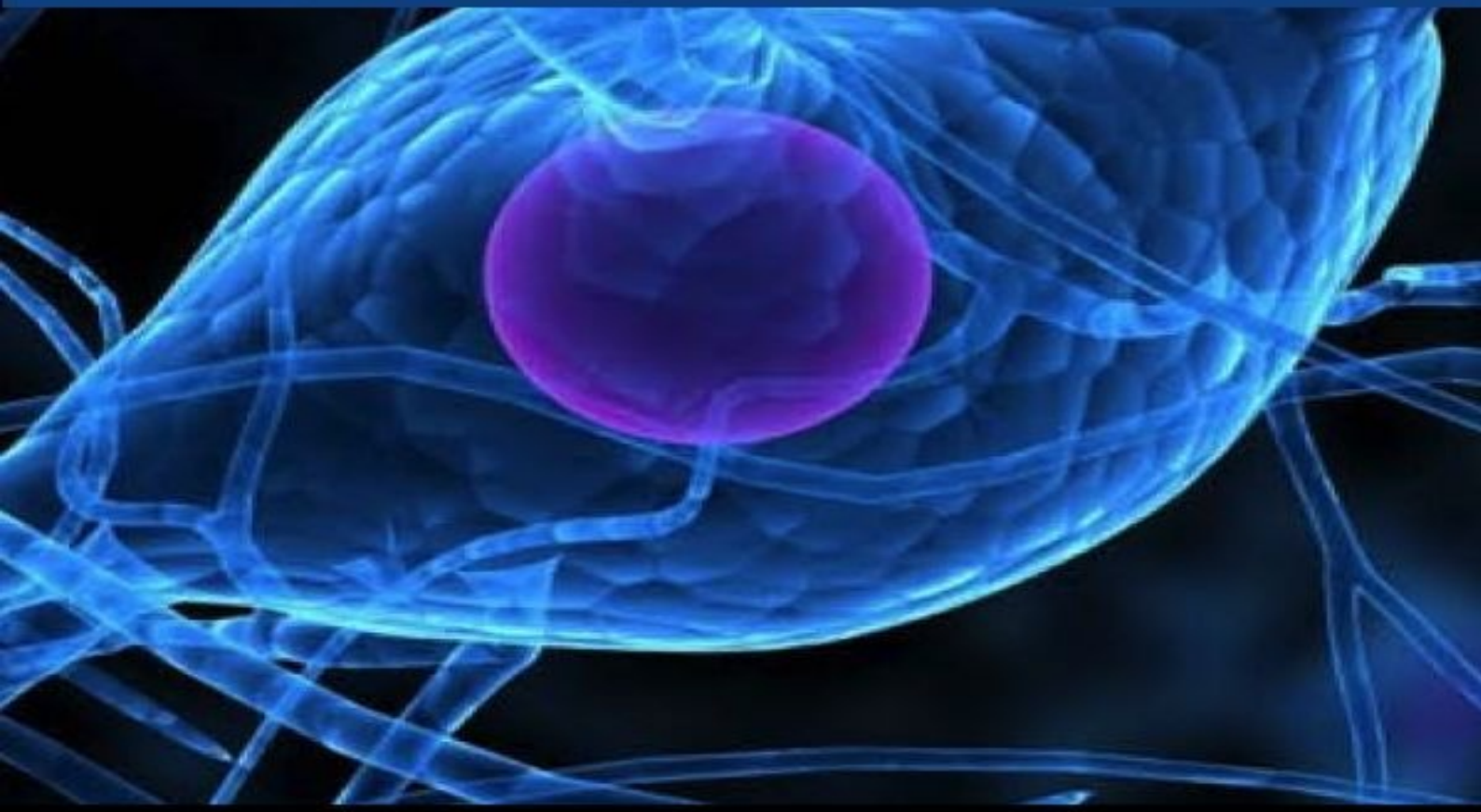
Deep Learning in Python Prerequisites

Master Data Science and Machine Learning with Linear Regression and Logistic Regression in Python



By: LazyProgrammer
(http://lazyprogrammer.me)

Deep Learning in Python Prerequisites

Master Data Science and Machine Learning with Linear Regression and Logistic Regression in Python

By: The LazyProgrammer (http://lazyprogrammer.me)

Introduction

So you want to learn about deep learning and neural networks, but you don't have a clue what machine learning even is. This book is for you.

Perhaps you've already tried to read some tutorials about deep learning, and were just left scratching your head because you did not understand any of it. This book is for you.

Believe the hype. Deep learning is making waves. At the time of this writing (March 2016), Google's AlghaGo program just beat 9-dan professional Go player Lee Sedol at the game of Go, a Chinese board game.

Experts in the field of Artificial Intelligence thought we were 10 years away from achieving a victory against a top professional Go player, but progress seems to have accelerated!

While deep learning is a complex subject, it is not any more difficult to learn than any other machine learning algorithm. I wrote this book to introduce you to the prerequisites of neural networks, so that learning about neural networks in the future will seem like a natural extension of these topics. You will get along fine with undergraduate-level math and programming skill.

All the materials in this book can be downloaded and installed for free. We will use the Python programming language, along with the numerical computing library Numpy.

Unlike other machine learning algorithms, deep learning is particularly powerful because it *automatically learns features*. That means you don't need to spend your time trying to come up with and test "kernels" or "interaction effects" - something only statisticians love to do. Instead, we will eventually let the neural network learn these things for us. Each layer of the neural network is made up of logistic regression units.

Do you want a gentle introduction to this "dark art", with practical code examples that you can try right away and apply to your own data? Then this book is for you.

This book was designed to contain all the prerequisite information you need for my next book, Deep Learning in Python: Master Data Science and Machine Learning with Modern Neural Networks written in Python, Theano, and TensorFlow.

There are many techniques that you should be comfortable with before diving into deep learning. For example, the “backpropagation” algorithm is just gradient descent, which is the same technique that is used to solve logistic regression.

The error functions and output functions of a neural network are exactly the same as those used in linear regression and logistic regression. The training process is nearly identical. Thus, learning about linear regression and logistic regression before you embark on your deep learning journey will make things much, much simpler for you.

Chapter 1: What is Machine Learning?

Computer programs typically follow very deterministic processes.

IF THIS

THEN THAT

This is desired behavior for most programs. You wouldn't want a human doing arithmetic for you, or your operating system to make "human errors" when you're trying to get your work done.

One very useful application of computer programs is modeling or simulation. You can write physics simulations and models and use them in video games to produce realistic graphics and motion. We can do simulations using the equations of fluid mechanics to determine how an airplane with a new design would move through the air, without actually building it.

This leads us to an interesting question: Can we model the brain?

The brain is a complex object but we have decades of research that tells us how it works. The brain is made up of neurons that send electrical and chemical signals to each other.

We can certainly do electrical circuit simulations. We can do simulations of chemical reactions. We have circuit models of the neuron that simulate its behavior pretty accurately. So why can't we just hook these up and make a brain?

Realize that there is a heavy assumption here - that there is no "soul", and that your consciousness is merely the product of very specifically organized biological circuits.

Whether or not that is true remains to be seen.

This is a very high-level view that kind of disappears when you study machine learning. The study of machine learning involves lots of math and optimization (finding the minimum or maximum of a function).

Another way to think of machine learning is it's "pattern recognition".

In human terms you would think of this as "learning by example".

You learn that 1 + 1 = 2. 2 + 2 = 4. 1 + 2 = 3. And so on. You begin to figure out the pattern and then you learn to "generalize" that pattern to new problems.

You don't need to re-learn how to add 1000 + 1000, you just know how to add.

This is what machine learning tries to achieve.

In less abstract terms, machine learning very often works as follows:

You have a set of training samples:

$X = \{ x_1, x_2, \ldots, x_N \}$

$Y = \{ y_1, y_2, \ldots, y_N \}$

We call X the "inputs" and Y the "outputs". Sometimes we call Y the "target" or "labels" and name it T instead of Y.

These come in pairs, so y_1 is the output when the input is x_1, and so on.

We hope that, given enough examples of x's and y's, our machine learning algorithm will learn the pattern.

Then, when we later input a new x_{NEW} into our model, we hope that the y_{NEW} that it outputs is accurate.

Note that there are other types of learning, but what I described above, where we are given X and try to make it so that we can predict Y accurately, is called "supervised learning".

There is a type of machine learning called "unsupervised learning" where we try to learn

the distribution of the data (we are just given X). Clustering algorithms are an example of unsupervised learning. Principal components analysis is another example. While deep learning and neural networks can be used to do unsupervised learning and they are indeed very useful in that context, unsupervised learning doesn't really come into play with linear regression or logistic regression.

Another way to view machine learning is that we are trying to accurately model a system.

As an example, think of your brain driving a car. The inputs are the environment. The outputs are how you steer the car.

X ———[real world system] ——— Y

An automated system to drive the car would be a program that outputs the best Ys.

X ———[machine learning model] ——— $Y_{prediction}$

We hope that after "training" or "learning" or "fitting", $Y_{prediction}$ is approximately equal to Y.

To look at this from an API perspective, all supervised machine learning algorithms have 2 functions:

train(X, Y) where the model is adjusted to be able to predict Y accurately.

predict(X) where the model makes a prediction for each input it is given.

Chapter 2: Classification and Regression

Within supervised learning there are 2 distinct tasks: classification and regression.

Classification is making predictions that are categories.

For example, the famous MNIST dataset is a set of images that are labeled 0 to 9.

A similar example is character recognition. This task is harder because you not only have to classify all the digits from 0 to 9, but all the uppercase and lowercase letters as well.

Another example is binary classification: given some measurements taken from a blood test, determine whether or not a person has a disease.

Binary classification, as its name suggests, always outputs 1 of only 2 categories.

Regression is making real-valued predictions, i.e. a number.

You might think that because the MNIST labels are numbers that the task is regression. This is not true!

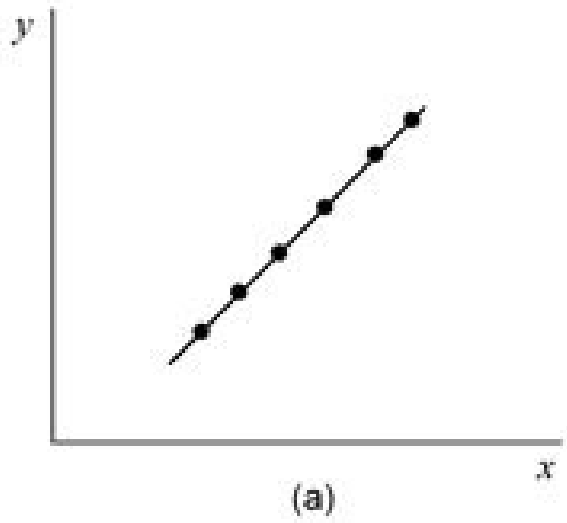
In the MNIST problem the numbers are just labels. 8 is not “closer to” 9 than 7 is. They are all just distinct, unrelated labels.

In regression, 8 IS closer to 9 than 7 is.

Chapter 3: Linear Regression

Linear regression, as its name suggests, is the prediction of a line.

Suppose we have the following set of Xs and Ys:



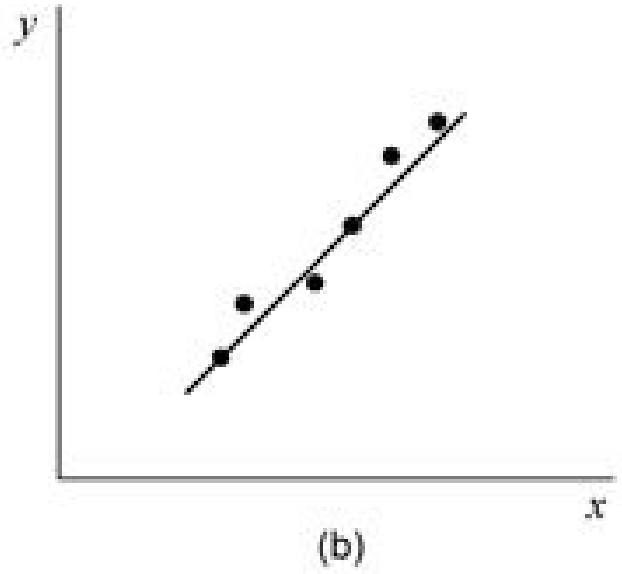
You can see that they form a perfect line.

Recall from high school geometry that a line is defined as:

$y = mx + b$

If we were to do this by hand, we could draw a line that goes through all the points, divide the “rise” over “run” to get the slope, and look at the y-intercept to find b. The prediction error would be 0 because the points fall directly on the line.

What if all the points do not fall on the line? Then we’d want to find the “line of best fit”.



How do we define such a line?

One way to write this is:

$y = mx + b + noise$

Where “noise” is a Gaussian-distributed random variable.

Visually, we would like a line that looks like it passes through the middle of where most of the points reside.

However, to solve this problem, we need to find the solution mathematically.

Let’s call our targets T and our predictions Y.

Remember that our “training data is”:

$X = \{ x_1, x_2, \ldots, x_N \}$

$T = \{ t_1, t_2, \ldots, t_N \}$

For a particular sample, we have:

$y_i = mx_i + b$

We want all the t_i to be close to the y_i.

To accomplish this, we'll define an objective function:

$J = sum\_from\_i=1..N \{ (t_i - y_i)^2 \}$

If T is exactly equal to Y, this should be 0.

If T is far away from Y, J will be large.

So we want to minimize J. You can see J takes the form of a quadratic, thus it has a unique solution.

How do we minimize quadratics? Recall from your high school calculus days that the solution is to take the derivative of J and set it to 0 and solve for m and b.

$dJ/dm = 0$

$dJ/db = 0$

I would recommend doing this at home on your own. You should arrive at the solution:

$m = [N*sum(x_iy_i) - sum(x_i)*sum(y_i)] / [sum(x_i^2) - sum(x_i)^2]$

$b = mean(y) - m*mean(x)$

Where sum() is the sum over all i from i=1 to N. And mean() is the sample mean (sum all the items and divide by N).

Extending linear regression to multiple dimensions

In real machine learning problems we of course have more than one input feature, so each x_i becomes a vector.

When x is 1-D, we get a line. When x is 2-D, we get a plane. When x is 3-D or higher, we get a hyperplane.

When you're coding in MATLAB or Python, it is more efficient to use "vectorized" operations, so understanding vectors and matrices is vital.

As an example, suppose I wanted to do the dot product between w = [1, 2, 3], and x = [4, 5, 6]

As you know from linear algebra, the answer is 1*4 + 2*5 + 3*6.

You might write it like this in code:

```
answer = 0
for i in xrange(3):
  answer += w[i]*x[i]
```

This is slow!

It is much better to use a library like numpy and call the dot function:

```
import numpy as np
w = np.array([1,2,3])
x = np.array([4,5,6])
answer = w.dot(x)
```

For this reason, we usually look at the entire training set at the same time, instead of considering individual input vectors x_i and individual outputs y_i.

So instead of saying the output of our model is:

$y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_Dx_D$

We instead say:

$y = w_0 + w^Tx$

Usually, we use a dummy variable $x_0 = 1$, and combine w_0 with $[w_1, \ldots, w_D]$ so that we can just make the entire thing one dot product.

Another way of stating this is, without loss of generality, we can always consider a model without the bias term w_0.

If we take multiple input vectors (a 1-dimensional object), and look at them simultaneously, we get a 2-dimensional object, which is a matrix.

Usually we use the convention that each row is a sample. So the dimensionality of the input matrix is N x D, where D is the number of input features, and N is the number of samples.

Similarly, the dimensionality of T is N x 1.

We can turn our old objective function:

$J = \text{sum\_from\_i=1..N } \{ (t_i - w^Tx_i)^2 \}$

Into matrix form:

$J = |T - Xw|^2$

Note that the parameter weights w move over to the other side, because the matrix X has each sample along the rows. When we talk about individual vectors, we usually mean they are column vectors.

My goal is not to teach you linear algebra in this book, but I would highly recommend learning it yourself so you can solve these problems on your own.

What you want to do here is take the gradient, or vector derivative, of J with respect to w, and set it to 0 to solve for w.

Solving for w should give you the answer:

$w = (X^T X)^{-1} X^T T$

In numpy you could use:

```
w = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(T)
```

A better way would be:

```
w = np.linalg.solve(X.T.dot(X), X.T.dot(T))
```

Because np.linalg.solve solves equations of the form Ax = b.

This type of solution is called “closed-form” because we can solve for the answer directly using algebra. No other problem concerning deep learning will be this “nice” from this point onward.

Putting it all together

```
class LinearRegression(object):
  def train(self, X, Y):
    self.w = np.linalg.solve(X.T.dot(X), X.T.dot(T))

  def predict(self, X):
    return X.dot(self.w)
```

Exercise

Many statisticians like to apply linear regression to economic data. Visit a government website of your choice, and download some census data in an area that you are interested in. See what patterns you can find (income vs. some other variable is often a popular choice).

Where to learn more

I do a full online course on linear regression you can find at: https://www.udemy.com/data-science-linear-regression-in-python. In this course I teach you how to measure how good of a fit your model is, a measure called the R-squared. I also provide some examples of how linear regression can be used. I go over polynomial regression, which allows you to predict non-linear functions. My goal in this book is not to teach you the full gamut of linear regression, but rather the parts that are relevant to logistic regression and neural networks, such that you are able to later “connect the dots”.

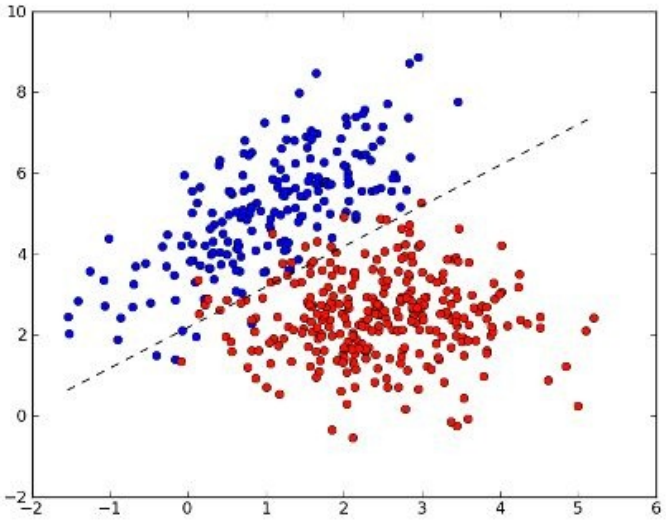
Is that all there is to linear regression?

Believe me, there is much more to linear regression analysis than what is contained in this chapter. Just ask a statistician. You can take an entire semester-long graduate course purely on linear regression. Are those things useful in deep learning? Perhaps. But not essential. It would be more productive to power ahead and look at linear classification.

Chapter 4: Linear Classification

Now that you know how to create a line / plane / hyerplane, we are ready to learn about classification.

The image below shows pictorially what we are doing when we do classification.



All of our 2-D x's are plotted in a scatterplot. Each x belongs to 1 of 2 classes - "red" or "blue".

You can see that there is a nice separation between the red class and a blue class - we can draw a line where most of the blue is on one side and most of the red is on the other side.

This would yield a "low classification error" or a "high classification rate".

Note that the "separating boundary" is a line - $y = mx + b$.

The same form we studied in the previous chapter!

The question now is - how do we find this line? What is our objective function? We will answer this in the next chapter.

For now, let's think about what would happen had we already found this ideal line.

Remember, the point of machine learning is to make predictions on new unseen data.

So let's say we have a new test point, (x_0, y_0), and we would like to know whether we should classify it as blue or red.

First, let's consider what would happen if the point were to fall directly on the line. Then the equation would work out perfectly, $y_0 = mx_0 + b$. We would be just as certain the test point is blue as we are that it is red.

What if $y_0 > mx_0 + b$? Then this point falls "above" the line and we say it's blue.

What if $y_0 < mx_0 + b$? Then this point falls "below" the line and we say it's red.

Usually, we "move the y over to the other side" and say $h(x, y) = y - mx - b$. We can now use a threshold of 0 to say whether or not a test point is blue or red.

More generally, we could say:

$h(x_1, x_2, \ldots) = w_0 + w_1x_1 + w_2x_2 + \ldots$

So when $h(x) > 0$, we predict the "positive" class, and when $h(x) < 0$, we predict the "negative" class.

I call the function h() because it's our "hypothesis" of what class x belongs to.

Sometimes, with older models like support vector machines and the perceptron, we use (+1, -1) to denote the 2 classes in binary classification.

With deep learning and neural networks, we typically use (0, 1).

Although the visualizations used in this section were necessarily in 2 dimensions, note that

all the techniques used do not require knowing what the dimensionality of the input is.

In real-world problems, we come across data that is hundreds or even thousands of dimensions. A dataset like MNIST contains 28 x 28 images. That’s 784 features per sample. Those are very tiny images in only one color. Another dataset, the street view house number dataset, contains color images of house numbers that are 32 x 32. While still a very tiny image, the fact that we need to account for the 3 color dimensions (R, G, B) means that each input vector is 3 x 32 x 32 = 3072 dimensions!

Exercise

Download the MNIST dataset from kaggle.com.

Chapter 5: Logistic Regression

Alright, onto the real stuff!

Logistic regression is a very simple extension of the linear classifier we studied in the previous chapter. Note the odd name - it has “regression” in it, but is actually a classification algorithm.

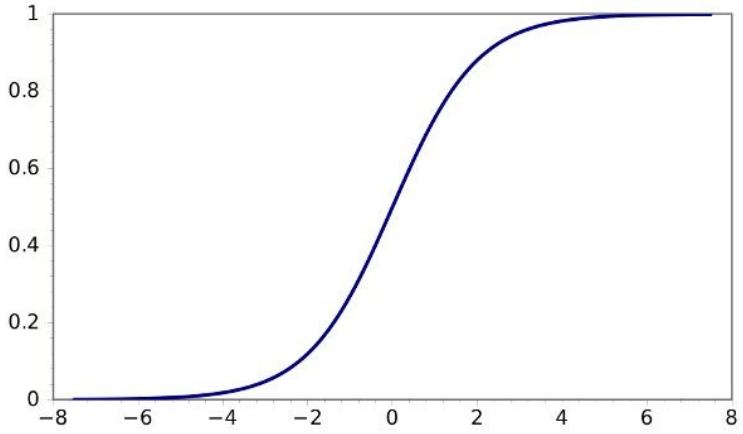
Let’s take the functional form from the previous section:

$a = w^T x$

Since a is the dot product between the weights w and the input vector x, it can be any real number between negative infinity and positive infinity.

In logistic regression we pass a through the “logistic function” or the “sigmoid”, which outputs values between 0 and 1.

Graphically, a sigmoid looks like this:



Mathematically, the output of our model is now:

$y = \text{sigmoid}(w^T x)$

Where the sigmoid is defined as:

$\text{sigmoid}(a) = 1 / [\ 1 + \exp(-a)\ ]$

In numpy and Python, this would be:

```
def sigmoid(a):
  return 1 / (1 + np.exp(-a))
```

One interpretation of the output of the sigmoid is that it's the probability of being the positive class. Or in other words:

$p(y=1 \mid x) = \text{sigmoid}(w^T x)$

The left side is read as "the probability that y equals 1 given x".

What happens when we fall exactly on the boundary of the 2 classes? From the previous section, we know that $w^T x = 0$. We can see from the figure above that sigmoid(0) = 0.5. It makes sense that the probability that y = 1 is 0.5 if we fall directly on the line. It means "we could go either way".

tanh output

One can also use the hyperbolic tangent instead of the sigmoid. It has the same shape as a sigmoid, but a different scale. The output ranges between -1 and 1, thus it cannot be considered a probability.

The definition of tanh is:

tanh(a) = [exp(a) - exp(-a)] / [exp(a) + exp(-a)]

Numpy as a built in tanh function: np.tanh()

softmax output

What happens when our output is more than 2 classes? Ex. MNIST or character recognition. With binary classification, we only needed 1 output node, because P(y=0 | x) = 1 - P(y = 1 | x).

i.e. The probability of all possibilities must sum to 1.

Let's consider what would happen if we used 2 output nodes.

We would have:

$a_1 = w_1^T x$

$a_0 = w_0^T x$

Recall that a_0 and a_1 can be either negative or positive. Since probabilities must be 0 or positive, we can enforce positivity by exponentiating a.

So we have 2 outputs, $\exp(a_1)$ and $\exp(a_0)$. How do we ensure these sum to 1?

Simply divide by $\exp(a_1) + \exp(a_0)$.

So now:

$p(y=1 \mid x) = \exp(a_1) / [\exp(a_1) + \exp(a_0)]$

$p(y=0 \mid x) = \exp(a_0) / [\exp(a_1) + \exp(a_0)]$

You can see that it would be very easy to extend this to any number of classes.

Note that we can "vectorize" a, so that, after also combining the individual weight vectors into a weight matrix, $W = [w_1\ w_2\ \ldots\ w_K\ ]$ we can simply write:

$$A = XW$$

$$Y = \text{softmax}(A)$$

Neurons

Sometimes, logistic regression is referred to as the “logistic unit” or neuron. Why? It has a few properties in common with the biological neuron. Appropriately, when you hook up a bunch of neurons / logistic units together, you get a neural network.

I discuss the similarity between digital neurons and biological neurons more in depth in my next book, Deep Learning in Python: Master Data Science and Machine Learning with Modern Neural Networks written in Python, Theano, and TensorFlow.

Exercise

Load the data from MNIST (or another dataset of your choice) into two numpy arrays, the inputs X and the targets T. X should be an N x D array, where N is the number of samples, and D is the dimensionality (D = 784 if you are looking at MNIST). T should be an N x K array (where K = 10 if you are looking at MNIST). The raw data will be in the form of an N x 1 array where the elements are values from 0..9. You will need to turn it into an indicator matrix of size N x K where the values are 0 or 1.

Write the code to initialize the weights W to come from a Gaussian-distributed array of samples, and write a function that takes in X and W and outputs a prediction Y.

It should look something like this (notice I've added the bias term for your convenience):

```
W = np.random.randn(D, K)
b = np.random.randn(K)

def softmax(a):
  expA = np.exp(A)
  return expA / expA.sum(axis=1, keepdims=True)

def predict(X, W):
  return softmax( X.dot(W) + b )
```

Chapter 6: Maximum Likelihood Estimation

You already know that to find the sample mean of a random variable, you simply sum up all the samples and divide by the number of samples.

The formal way of deriving this solution is called maximum likelihood estimation.

Suppose there is some parameter you want to estimate, let's call it "w", which can be the mean, variance, or any other parameter. The likelihood is given by:

$L = p(X \mid w)$

Where X is the data samples.

For instance, suppose we were looking at a Gaussian with unit variance.

$L = \text{product\_from\_i=1..N } \{ (1 / \text{sqrt}(2\text{pi})) * \exp(-(1/2) (x_i - \text{mu})^2) \}$

We are able to take the product of the probability of each sample because each sample is assumed to be independent.

So $L = p(x_1 \mid \text{mu})\ p(x_2 \mid \text{mu}) \ldots p(x_N \mid \text{mu})$

We would like to maximize L with respect to mu, i.e. maximize the likelihood over the entire training set.

To do this we go back to our old friend calculus. Take the derivative of L with respect to mu, set it to 0, and solve for mu.

You should arrive at the expected answer.

Note that before you do an actual calculation, you'll want to take the log of the likelihood (usually just called the log-likelihood) and maximize that. These functions are usually

easier to optimize after taking the log.

Now it is easy to see why we use objective functions like the squared error.

With linear regression, we assumed $y \sim N(w^T x, some\_variance)$. (In English, this means y is normally distributed with mean equal to $w^T x$ and variance equal to some_variance).

Another way of writing that is $y = w^T x + noise$, where $noise \sim N(0, some\_variance)$.

What happens when you set up the likelihood and take the log? You simply get the squared error!

(Technically, you get the negative of the squared error)

Maximizing the likelihood is thus the same as minimizing the squared error objective.

Sigmoid

We do something a little different for the sigmoid / logistic regression. The error isn't really normally distributed, since the output can only be between 0 and 1. The likelihood here is more like a coin toss.

$$L = \text{product\_from\_i=1..N } \{ p^{t(i)} (1-p)^{(1 - t(i))} \}$$

For the output of a logistic:

$$L = \text{product\_from\_i=1..N } \{ y(i)^{t(i)} (1-y(i))^{(1 - t(i))} \}$$

If you took the negative-log of this you would get the cross-entropy error:

$$J = -\text{sum\_from\_i=1..N } \{ t_i \log(y_i) + (1 - t_i) \log(1 - y_i) \}$$

Where I'm using $t(i) = t_i$ interchangeably due to limitations in the output format of this book.

Softmax

Whereas the sigmoid is like a coin toss, softmax is like rolling a die.

Since there are K possibilities (K output classes) in softmax, we need to consider all of them. Usually the target variables are represented by an indicator matrix, so t[i, k] = 1 if the ith sample belongs to the kth class.

Note that this means both the targets and the model output, considering all data point simultaneously, would be matrices of size N x K.

Performing the same process as in the previous 2 sections, we would arrive at the objective function:

J = -sum_from_i=1..N { sum_from_k=1..K { t[i,k] * log(y[i,k]) } }

Exercise

Write a function to calculate the cost function for our MNIST example. It should look like this:

```
def cost(T, Y):
  return -( T * np.log(Y) ).sum()
```

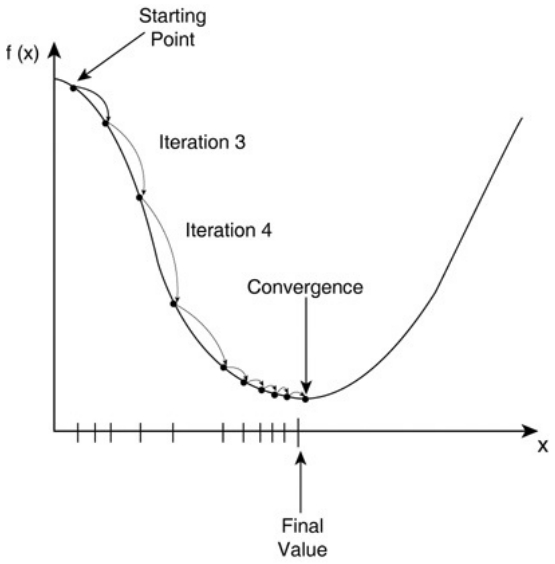
Chapter 7: Gradient Descent

Now that we have our likelihood functions, what comes next? Recall that for linear regression, we were able to solve for w directly in terms of X and Y.

Because of the “nonlinearity” (the sigmoid) we used in logistic regression, this is no longer possible. Instead, we use a more general numerical optimization technique called “gradient descent”.

We start by initializing w to a random value (usually Gaussian-distributed).

In a picture, gradient descent looks like this.



Convince yourself that by going along the direction of the gradient, we will always end up at a “lower” J than where we started.

Again, since my objective in this book is not to teach you calculus and linear algebra, I am simply going to provide you with the solution, but I would highly recommend teaching yourself how to arrive at the solution yourself.

$dJ / dw = X^T(Y - T)$

These are the full data matrices and the same formula works for both sigmoid and softmax. Convince yourself that the right-side outputs a vector of size D x 1 when the output is a sigmoid, and a matrix of size D x K when the output is softmax.

Once you find the gradient, you want to take small steps in that direction.

You can imagine that if your steps are too large, you'll just end up on the "other side" of the canyon, bouncing back and forth!

Thus we do our weight updates like so:

$$\text{weight} = \text{weight} - \text{learning\_rate} * \text{gradient\_of\_J\_wrt\_weight}$$

In a more "mathy" form:

$$w = w - \text{learning\_rate} * dJ/dw$$

Where the learning rate is a very small number, i.e. 0.01. (Note: if the number is too small, gradient descent will take a very long time. I show you how to optimize this value in my Udemy course - https://www.udemy.com/data-science-logistic-regression-in-python).

That is all there is to it!

If you want to convince yourself that this works, I would recommend trying to optimize a function you already know how to solve, such as a quadratic.

For example, your objective would be J = x**2 + x, and the gradient of J is 2x + 1, so the minimum can be found at -1/2.

Exercise

Write a complete logistic regression classifier that can do both learning and prediction, and use it on a dataset like MNIST to see what accuracy you can get.

It should look something like this:

```
def grad(Y, T, X):
  return X.T.dot(Y - T)

for i in xrange(epochs):
  Y = predict(X, W)
  W -= learning_rate * grad(Y, T, X)
```

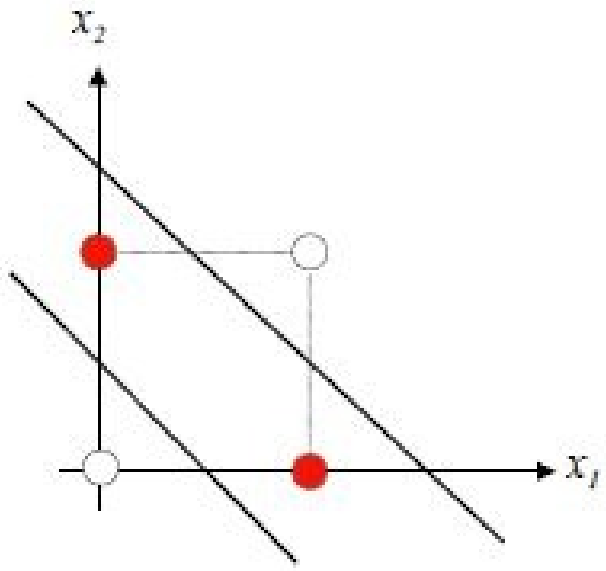
Also keep track of the cost at each iteration and plot it using matplotlib at the end. You should notice a steep decrease at the beginning but it should level out quickly.

Write a function to do linear regression with gradient descent. You should be able to find the derivative of the squared error quite easily since it’s just a quadratic.

Chapter 8: The XOR and Donut Problems

Logistic regression is really great for problems that have a linear boundary, since the weights define a line or a plane. There are some classical problems that linear classifiers can't solve in their basic form, but I will show you how to modify logistic regression in order to do so.

First, let's look at the XOR problem:



XOR is a logic gate like AND and OR. The outputs are defined as follows:

0 XOR 0 = 0

1 XOR 0 = 1

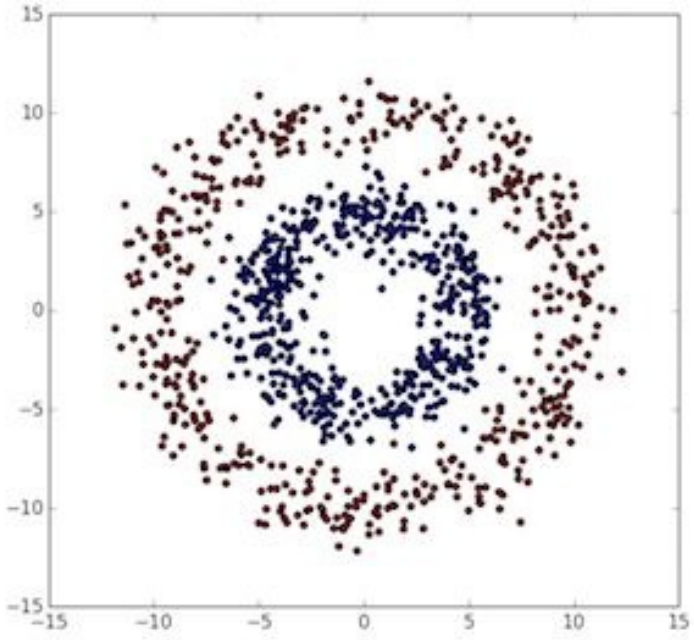
0 XOR 1 = 1

1 XOR 1 = 0

As you can see, there is no line that separates the two classes.

Now let us look at the donut problem:

As you can see, this problem also contains no linear boundary between the two classes.



So what do we do in these two cases?

For the XOR problem, create a third dimension that is derived from the first two, $x_3 = x_1x_2$. Try to draw a 3-D plot to see how a plane could separate the two classes now.

For the donut problem, we see that the radius is a discriminating feature. So if we created a third dimension $x_3 = \text{sqrt}(x_1^2 + x_2^2)$, we would be able to draw a plane between the two classes.

What is the disadvantage of this?

We have to manually come up with features!

In practice, there are just way too many to consider.

Think about the street view house numbers dataset, where D = 3072. We would consider x_1x_2, then x_1x_3, then x_1x_4, etc…

This can also lead to overfitting.

The great advantage of deep learning and neural networks is that they automatically find features for us.

Exercise

Write code to generate the data for the XOR problem and the donut problem.

Use the logistic regression classifier with no hand-crafted features to prove to yourself that this yields a low classification rate.

Next, add the features I've described in this chapter and show that you can achieve almost-perfect (or perfect in the case of XOR) discrimination.

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

Was the material in this book easy? Do you want to dive straight into neural networks and deep learning? Check out my Udemy course:

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to this book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

This next course contains very similar material to this book, but I derive everything step-by-step on video.

In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

This next course was the basis for Chapter 3 in this book on linear regression.

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don't forget to hit "like"!)