Table of Contents / Outline

1. Problem: Perceptron

[Note to Erin] - This tag marks sections that are written for Erin. They contain "behind-the-scenes" information and notes about how I hope to implement incomplete sections.

[Text for Students] - This tag marks sections that are written for an audience of students.

[Video Script Summary] - This tag marks sections that will ideally become walkthrough videos. The scripts are by no means final. Instead, they're meant to outline the core of the ideas that should be covered. Here's a sample of how we hope to deliver:
http://neuralblocks.strikingly.com/

2. [Text for students] TL;DR

Construct the equivalent of a human neuron in code! Aka: build a simple logistic regression model, a perceptron.

3.1. [Text for students] Background: What is Machine Learning?

Machine learning (a type of artificial intelligence or AI) is a set of algorithms that learn to do tasks without being explicitly programmed. In this problem set, you will create the building blocks for your own machine learning model. You can then apply your model in new ways you dream up.

Okay, but what are some examples of how machine learning plays a role in our lives?

Well, have you ever chatted with Siri?

Watched a self-driving car navigate a road?

Or consider this: you're on a mission to help folks in a remote area. You use your phone's camera to examine someone's eye and a machine learning app accurately spots the presence of diseases like diabetic retinopathy.

Machine learning is a rapidly expanding field with profound impacts and lots of opportunities to stake out creative new uses. It is transforming industries ranging from healthcare to education to transportation to finance.

This problem set will enable you to code your own algorithm for assessing the malignancy of breast cancer tumors. You can then apply the concepts you learn and the model you master to many other challenges.

All right! You're ready to head over to the cs50 machine learning workspace where we will use Python.

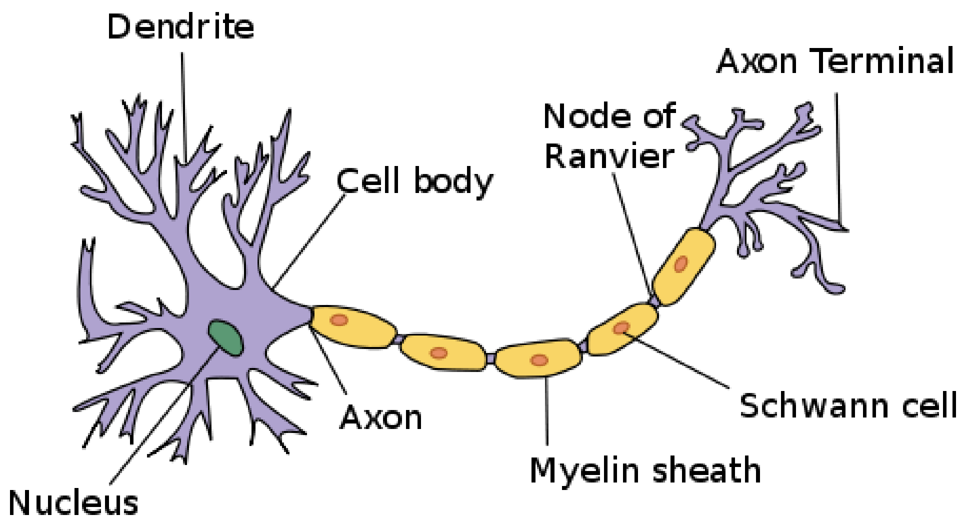
Python is a very popular programming language for implementing machine learning models. Part of its beauty stems from the fact that it's very high level. Since we're aspiring machine learning and data scientists, though, we'll start at a slightly lower level of abstraction. This way, you'll get a taste of the first-principle mathematics behind how these systems work . . . and you'll appreciate libraries like TensorFlow at a much deeper level.

3.2. [Walkthrough video] The Human Neuron: How It Works

Video Script Ideas:

Since our goal is to build a “neuron” in code, it’s gonna be helpful to understand what a human neuron is in the first place, and how it works.

So here’s the classic human neuron:



[image from creative commons]

At its heart, the function of a neuron is very simple. If it receives a signal above a certain threshold, it fires; otherwise, it doesn’t.

[animate this process visually]

While this functionality is very simple--neurons are very simple pattern recognizers--when you collect a bunch of these neurons and put them into a network, they can actually process information in surprisingly complex ways, and they can learn to recognize complex patterns (for example, handwritten digits).

[Show image of a neural network, and animate this to illustrate how information will flow through.]

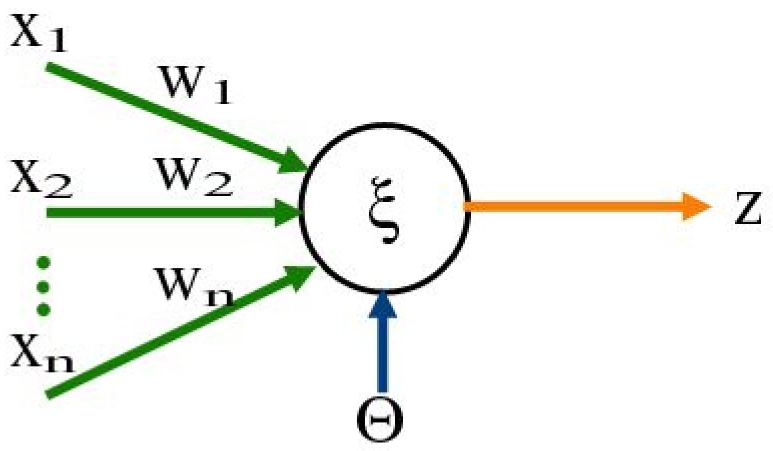
Fun (crazy) ideas for spicing up the video:

- Is there a Harvard professor of neuroscience who would want to make a guest appearance? ML applications are cool because they represent cross-disciplinary challenges, and since Harvard is so strong in all subjects, it could be cool to bring in Harvard experts in various fields as guests in these walkthroughs . . .

3.3. [Walkthrough video] How a Perceptron Emulates a Neuron

<u>Video script ideas:</u>

A perceptron looks similar to a neuron:



[Image from creative commons]

Perceptrons take input data that you feel are relevant to your machine learning challenge (represented by X1, X2, X3 . . .) and then make output **predictions.**

The output prediction can take one of two main forms. On the one hand, the perceptron can output a **continuous value**. For example, a perceptron may predict that tomorrow's temperature will reach 25.7 degrees Celsius at noon, and then drop to 16.1 degrees Celsius at nightfall. Or, it can output a **discrete value**, like a "1" to represent that it will be hot during the day, or a "0" to represent that the day will be cold. This latter case, in which the network outputs either a one or a zero to indicate whether the temperature will be hot or cold, is called a "classification task". With classification tasks, the job of the perceptron is to give the probability that the input data belongs to a certain **class** or type.

As a classic example of a classification task, consider a machine learning algorithm that takes in a picture of a fruit as an input, and then outputs the probability that the fruit is an apple, an orange, or neither. In this case, the algorithm is classifying the fruit by assigning it one of three **labels**, namely "apple", "orange", or "neither".

In classification tasks, perceptrons behave very similarly to human neurons. Namely, if the input they receive is above a certain threshold, they “fire” and output a value close to one. Otherwise, they output a value close to zero.

Let’s step through an example.
//Erin, I’ll flesh this out, but here is a quick summary:

We have a perceptron that’s supposed to classify an image as being either an “apple” or “not an apple”. In this case, the input X values are pixel data, and we want the perceptron to output a “1” if it’s 100% sure the image is an apple or a “0” if it’s 100% sure the image is not an apple.

If we were to plot the data, we’re trying to define the ideal line, represented by a mathematical function, that represents “apple” or “non-apple”.

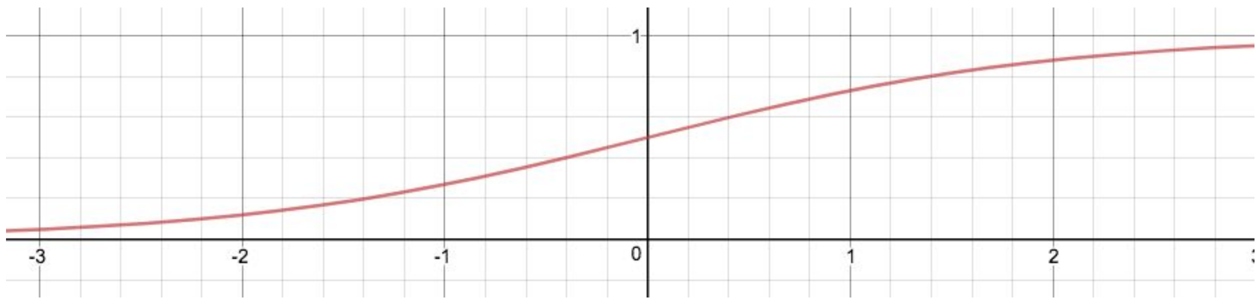
When we feed the input data into the network, each X value (pixel value) is multiplied by a coefficient or **weight** and summed together into a single real number. Weights allow the perceptron to determine the importance of each piece of input data. For example, a higher weight would give the associated piece of input data more influence over the perceptron’s final prediction.

[Use visual animation to make this clearer → light up parts of the diagram or something like that]

Now, the neuron needs to decide whether or not to fire based on the value of this real number that we’ve calculated. But how do we simulate the human neuron’s firing process?

Researchers have found that a certain mathematical function, called the sigmoid function, does a nice job of simulating the neuron’s firing process.

Here’s a graph of the sigmoid function:



$$\frac{1}{\left(1+e^{-x}\right)}$$

[thanks, Desmos]

When the input value to this function is a small number (less than zero), the sigmoid function gives an output close to zero. On the other hand, when the input value to the function is a large number (greater than zero) the sigmoid function yields an output close to one.

We can feed the real number that we generated before into this function! Then, we can interpret the function's output as the probability that the input image is an apple (i.e. an output of 0.75 would mean that there's a 75% chance--or that the perceptron is 75% confident--that the input image represents an apple).

All right, so we understand how input data would be converted into output predictions. But how do we make those predictions better over time?

We mentioned something called the "weights", or the coefficients by which the input X values are multiplied. These weights start off as random values, but the job of the perceptron is to learn which weights enable it to make the most accurate predictions.

In other words, the perceptron's job is to learn a nonlinear function that maps input data to the correct output predictions.

It figures this out over time by continuing to update its weights as it encounters more training examples. The perceptron makes guesses, generates errors (because it won't always make correct guesses in the beginning), and then learns over time to minimize these errors by appropriately adjusting its weights. This whole process is referred to as **backpropagation**.

[Note to Erin: the following writing is quite dense, so I will work on making it more digestible . . .]

In backpropagation, we calculate error using the cost function (which for logistic classification tasks is usually defined as the sigmoid cross entropy function [show image of the equation]) and then we use multivariate calculus to determine the derivative of this function with respect to each of the weights in the network. Each weight in the network is then updated based on these derivative values. Visually, if the cost function graph represents a multidimensional landscape, then the back propagation process involves mathematically taking "steps" across that landscape by updating the network's weight values in order to find the lowest valley. Behold, a classic optimization problem: we're trying to minimize the network's error.

3.4. [Walkthrough video] Diagnosing Breast Cancer as a Doctor

Video script ideas:

When assessing whether a tumor is malignant or benign, doctors look at factors like:
- Cell clump thickness
- Uniformity of cell size

- Uniformity of cell shape

And some other factors in order to make their prediction of whether or not a tumor is malignant. In the days before machine learning, the decision was made heuristically--doctors would study the samples and make diagnoses based on their medical experience and training.

But now . . . you will have the power to design an algorithm that makes diagnoses in seconds, based on hundreds of other data points.

That leads us to our next video, diagnosing breast cancer as a data scientist.

Fun ideas to spice things up:

- Dress up as a doctor in the video
- Maybe bring in a guest medical expert

3.5. [Walkthrough video] Diagnosing Breast Cancer as a Data Scientist

Video script ideas:
As a data scientist, our goal is to design an algorithm that learns how to classify whether or not a potential tumor is malignant. This means we need to find a whole bunch of training data for our algorithm.

Check out this open-source repository of datasets.

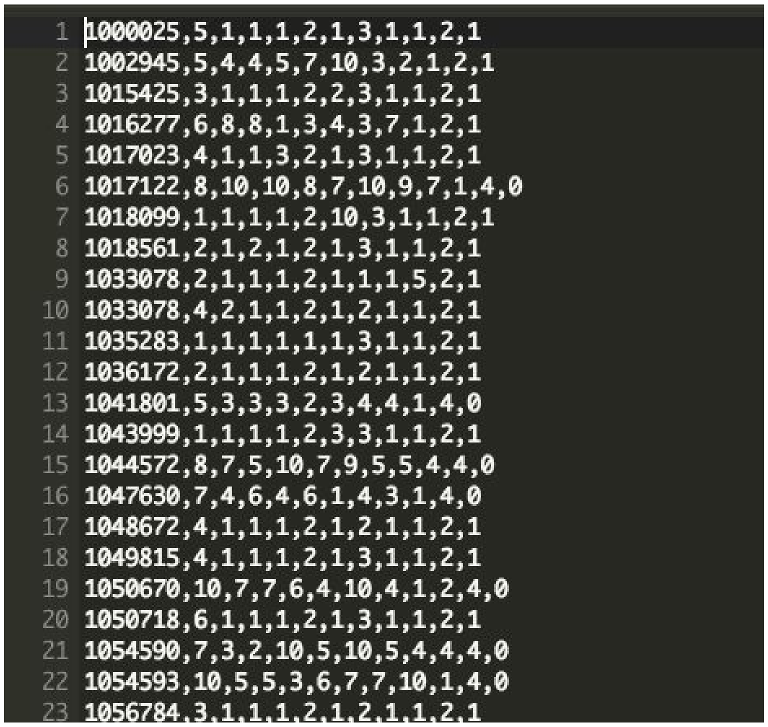
[Show footage of navigating to the UC Irvine dataset repository]

Behold, a breast cancer dataset! This information [gesturing with mouse] tells us that the dataset represents a classification task, and all of the data is in integer form. Oooh and look at all of these citations!

Digging into the details, this dataset has nine input parameters that represent the important factors mentioned above in diagnosing breast cancer. Each factor is represented by a number between 1 and 10 . . . how kind of the dataset curator to scale this for us (each input parameter has numbers in the same range). The output value is essentially binary--according to this description, a ‘2’ marks a benign case and a ‘4’ marks a malignant case. Finally, the bias is reasonable: benign cases make up 65.5% of the dataset, and malignant cases make up 34.5% of the dataset.

We’ve downloaded this dataset for you and tweaked some of it. For one, we noticed some missing values in the data, so we removed those entries. Also, we converted the output values so that a ‘0’ means the tumor is benign and a ‘1’ means that it’s malignant (instead of the ‘2’ and ‘4’ they use) so that this will match the output of the sigmoid function.

[Show an image of the downloaded dataset → navigate in cs50 ide and open it up there]



```
1000025,5,1,1,1,2,1,3,1,1,2,1
1002945,5,4,4,5,7,10,3,2,1,2,1
1015425,3,1,1,1,2,2,3,1,1,2,1
1016277,6,8,8,1,3,4,3,7,1,2,1
1017023,4,1,1,3,2,1,3,1,1,2,1
1017122,8,10,10,8,7,10,9,7,1,4,0
1018099,1,1,1,1,2,10,3,1,1,2,1
1018561,2,1,2,1,2,1,3,1,1,2,1
1033078,2,1,1,1,2,1,1,1,5,2,1
1033078,4,2,1,1,2,1,2,1,1,2,1
1035283,1,1,1,1,1,1,3,1,1,2,1
1036172,2,1,1,1,2,1,2,1,1,2,1
1041801,5,3,3,3,2,3,4,4,1,4,0
1043999,1,1,1,1,2,3,3,1,1,2,1
1044572,8,7,5,10,7,9,5,5,4,4,0
1047630,7,4,6,4,6,1,4,3,1,4,0
1048672,4,1,1,1,2,1,2,1,1,2,1
1049815,4,1,1,1,2,1,3,1,1,2,1
1050670,10,7,7,6,4,10,4,1,2,4,0
1050718,6,1,1,1,2,1,3,1,1,2,1
1054590,7,3,2,10,5,10,5,4,4,4,0
1054593,10,5,5,3,6,7,7,10,1,4,0
1056784,3,1,1,1,2,1,2,1,1,2,1
```

<u>Fun ideas to spice things up:</u>

- Wear aloha outfit; dressed like this to let the algos do the work for me…

4. [Note to Erin] Getting Started

Students will download a zip file containing starter Python code to their CS50.io environment.

Files in the zip will include:

- Script_01 (for downloading libraries, inspiration from here: https://medium.com/@tenzin_ngodup/installing-TensorFlow-in-cloud9-436043d75ce9)
- Script_02 (for downloading libraries, inspiration from here: https://medium.com/@tenzin_ngodup/installing-TensorFlow-in-cloud9-436043d75ce9)
- Perceptron_student.py (for building the TensorFlow logistic regression model)
- Sigmoid_student.py (for implementing the math behind the sigmoid function) *note: not yet written*

“Behind-the-scenes” files:

- Perceptron_staff.py (staff code with the correct solutions for the TensorFlow logistic regression model)
- Sigmoid_staff.py (staff code with the correct solutions for the sigmoid function implementation) *note: not yet written*

[Text for students]
Run the two scripts labeled Script_01 and Script_02 in that order. These will install TensorFlow and NumPy. In order to run the file labeled perceptron_student.py, first navigate to the directory in which that file is stored, and then type “python perceptron_student” into the terminal. Voilà! The code written in that file should execute.

[Note to Erin]
You may find it handy to put the code document in one tab and this write-up in the other. This is designed for you to check as you work your way through the code documents.

5. [Text for Students] Implement the Key Mathematical Function Behind the Perceptron: Translating the Sigmoid Function into Code

[Note to Erin: I haven’t officially implemented this in the distribution code yet]

Now that we understand how a perceptron model works and how we expect it to diagnose breast cancer based on our dataset, it’s time to implement the key mathematical function behind the model: the sigmoid function!

This is the function that enables the perceptron to behave like a human neuron by firing only if its input is above a certain threshold value.

As a refresher, here’s the function written out in mathematical notation:

$$\frac{1}{\left(1+e^{-x}\right)}$$

Let’s translate this math into code!

Sig_output =

We just translated a mathematical function (the sigmoid function) into working code. This is a great skill to practice because as you go out and research the exciting new developments in AI you’ll find that many authors simply mention the math. Its left to you to translate that math into functioning code.

6. [Text for Students] Construction Phase: Building the Perceptron in TensorFlow

Now that we have reviewed the essential components of a perceptron, let's dive in and create one…

6.1. Taking a look at the dataset

It's generally a good idea to visualize the data at this stage in order to understand its key features.

It's important to keep in mind that your model is only as good as the data you feed it . . . garbage in, garbage out as they say. In this case, we've provided some clean data for you. (As an up-and-coming data scientist, you will spend a good amount of time cleaning and preprocessing your data to make sure it's complete and representative of what you want to analyze.)

Here's a recap of the relevant dataset specifics:

- There are 9 input parameters or "X values", each ranging from 1 to 10 in value. These parameters represent things like the area of the cell nucleus and the number of concave portions in its contour.
    - If you'd like to know what each parameter represents, check this out: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
- The output value (Y value) is either 0 or 1 for benign or malignant, respectively
    - Note that this dataset contained missing values, but since there were only a few instances, we opted to straight out delete them. There are different methods for dealing with missing values. More info in this helpful article: [LINK]
- In terms of bias, benign cases make up 65.5% of the dataset, and malignant cases make up 34.5% of the dataset. This is not a perfect 50%-50% split, but it is not egregiously dominated by one class, so it will work fine for our purposes.

Feel free to open up the bc.csv file in your cs50 environment in order to view the raw data.

6.2. Set up TensorFlow placeholders: creating the "pipe system" that will allow us to later feed data into the perceptron

In TensorFlow, the first stage of building a perceptron algorithm is the "construction" phase. This means you must build what is called in the lingo of TensorFlow a computational "graph"-- you're essentially outlining what mathematical functions will be applied to the data you want to analyze.

Or, if you'd like a physical metaphor: if the data is like water, then you're designing the pipe system (aka the mathematical functions) that the data will flow through.

We are using placeholders to establish the points in the computational graph where we will feed the data into our model. These are like the “nozzles” in the pipe system into which water can be poured.

We will be sending two types of data into our model: 1) the input parameters and 2) the expected output data, or "targets" for the model. We use the targets to help the model learn more effectively by comparing the perceptron’s “guesses” to the correct labels.

With all of this in mind, it's time to set up placeholders in TensorFlow. You might find this function helpful: [LINK]

6.3. Set up the TensorFlow variables: these will store the values that our perceptron model will manipulate

So now we have “pipes” through which we can eventually feed the data. The next important task is creating and initializing TensorFlow variables.

Like the other programming variables you know and love, TensorFlow variables store values. The values that these variables store can be manipulated by TensorFlow functions (also called TensorFlow operations). For example, if you use a TensorFlow variable to store the value ‘1’, and then you use tf.add() to add another ‘1’ to the value of the variable, the variable’s new value will be--you guessed it--’2’. Simple, right?

Awesome.

The whole point of training this machine learning model will be to optimize a set of parameters. These are the weights that we’ve mentioned in the walkthrough videos. Remember this diagram?

[insert diagram with weights labeled]

We’ll implement these weights by creating a TensorFlow variable.

By setting up TensorFlow variables, we’ll be creating matrices of random numbers. The random numbers represent an arbitrary starting point, but over time, the neural network will learn which weight values enable the network to make the best predictions.

Let’s make the first TensorFlow variable a tensor (basically an array) that holds the initial weights.

In order to work with the set of weights, we need to use matrix math. If feeling a bit unfamiliar with this linear algebra, please feel free to consult this short walkthrough video:

[video script: short explanation of how to take the dot product between two matrices]

Remember that there are 9 ‘X’ parameters. Also remember that to perform matrix multiplication, the following must hold true:

[insert diagram of dot product with dimensions]

Engineer the dimensions of the weight variable accordingly so that it can be multiplied in the manner above with the array (matrix) of input values.

Weights =

Last but not least, implement a TensorFlow variable for the bias unit.

Bias =

6.4. Building the model: code the functions that will transform the input data into output predictions

This time around, we’ll be using a fairly simple model. We’re basically building one neuron of a neural network--aka we’re building a perceptron network that’s a logistic regression classifier.

If we were to draw a diagram of this perceptron model, it would look like this:

[Diagram of perceptron model]

If we were to translate this diagram into mathematical notation (specifically linear algebra notation), it would look like this:

[Image of linear algebra notation]

Now it’s time to implement this in code . . . don’t forget to add in the bias unit! If you want a hint, here’s a TensorFlow function you may find useful for carrying out this task: [LINK]

6.5. Backpropagation: help the perceptron model learn from its mistakes

The perceptron model needs to learn. As explained in the walkthrough above, it does this by making guesses, quantifying how wrong those guesses are by using a cost function, and then updating its weights based on the mistakes it made.

6.5.1. Cost function: quantifying the mistakes the perceptron model makes

A cost function is a way to measure how far off the perceptron's predictions are from the correct outputs. Here it is written mathematically:

[Insert another image of the equation for the cost function]

So how do we write up this learning process in code?

We'll give this one to you easy. Well, actually, TensorFlow is giving this one to you easy because this is the beauty of the library: it abstracts messy math for you!

Since this is a logistic classification task, we probably want to use the sigmoid cross entropy function as a cost function.

[Insert another image of the equation for the cost function]

Feel free to check out these goodies in the TensorFlow reference [LINK TO HELPFUL FUNCTIONS], and then complete this line of code:

Cost =

6.5.2. Optimizer: defining a function for updating the weights of the perceptron in order for it to improve and minimize its mistakes

In one of our previous walkthrough videos, we talked about the mathematics behind the backpropagation process, the fancy name for the process of updating the perceptron's weights based on its errors or mistakes. With TensorFlow, the crazy (but, granted, somewhat cool) derivative calculations and optimization procedure are all abstracted. We just need to intelligently choose an optimizer, the function that will minimize the error or cost of the perceptron. So let's do that now. Complete this line:

Optimizer =
Here's a hint if that feels a bit too open-ended: [LINK TO OPTIMIZER]

6.5.3. Defining the training step: telling the optimizer what to optimize

Finally, we need to explicitly tell TensorFlow our end goal with each training step, which is to minimize the perceptron's cost.

Train_step =
Here's a hint [LINK TO TENSORFLOW REFERENCE WITH THE APPROPRIATE FUNCTION]

6.6. Calculating accuracy

The whole point of creating this model is to create a system that effectively diagnoses the malignancy of breast cancer. But how can we tell if the model is doing a good job?

This is where the accuracy score comes into play.

Finding the accuracy of the model means counting up the number of times the model makes correct predictions and dividing that by the total number of predictions the model makes.

If you are interested in fancier metrics for determining how well the model is doing, then check this out: [LINK]

The first part of our process will be observing the model's prediction. To get the model's prediction, we have to send the output from our model through the sigmoid function. Complete this line of code:

Prediction =

Now we want to figure out which predictions are correct. Ultimately, we want to produce one long vector of ones and zeros. Each element should represent one training example--the element will be a one if the model's prediction matches with the target value. Otherwise, the element will be a zero.

Let's make this happen:

Correct =
Hint: this might be a helpful function [LINK TF.EQUAL], along with this [LINK TF.CAST]

We're almost there! Now we just need to calculate the accuracy: in other words, sum over all the correct predictions and divide by the total number of predictions. Here we go . . .

Accuracy =

Hint [LINK TO TF.REDUCE_MEAN]

7. [Text for Students] Execution Phase: Sending Data Through the Model

Great job! We’ve completed the construction phase. Now we enter the execution phase. We’ve finished building the metaphorical “pipe system”, so now it’s time to send the data through it.

Usually this is characterized by a loop because we will be sending multiple batches of input data through the model.

Start by setting up a loop to step through the total number of epochs, or training time periods. The number of epochs (training periods) represents how many times we want to send a new batch of input training data through our model.

The batch size is really an arbitrary choice made in an earlier line of code. Feel free to come back and experiment with optimal batch size. Right now it's set at 30, so 30 training examples will be sent through the model at a time. These training examples will be randomly chosen. That means we want to find a random set of indices and then extract values from the test and training sets corresponding to those indices.

Check out this potentially useful function: [LINK TO numpy.random.choice]

Now that we have randomly selected a batch of training examples, we must feed them into the computational "pipe system" (graph) that we created during the construction phase. This can be accomplished with the help of a feed_dict (the name reflects the action of "feeding in" a dictionary of data to the model). Take a look at the documentation for sess.run() and write a line of code that sends data through the model in order to optimize the "train_step". In other words, run the chunk of the computational graph that we’ve labeled “train_step”.

Here’s a hint: [LINK TO TF.EVAL() FUNCTION]

It can be enlightening to keep track of the model's cost. This measurement should, if all is ideal, continuously decrease as the model trains because the model is expected to learn how to make better predictions over time.

Use sess.run to store the cost in curr_cost:
Curr_cost =

Finally, it's helpful to track the accuracy of the model. This is our measurement for how well the model is performing. We'll want to calculate both the training set accuracy and the test set accuracy.

Use sess.run to do this:
Test_acc =
Train_acc =

The last part of the code in this file is written for you. After every 5 epochs, it prints out the cost, training accuracy, and testing accuracy of the model.

8. [Text for students] Congrats!

You now know how to construct algorithms using Google's TensorFlow library, one of the leading machine learning platforms in the world. This is the same library that powers many of Google's own cutting-edge machine learning products.

What's more, you understand one of the core types of machine learning algorithms: the perceptron model. You can do loads with it, and it is the fundamental building block to more complex artificial intelligence algorithms.

Oh, and you managed to create an algorithm for detecting breast cancer tumor malignancy with around 90% accuracy along the way.

Really, not bad for a day's work!

9. [Note to Erin] Parting Challenge

Now that students have these new tools on their belts, I'd like to urge them to hunt down datasets that interest them from friendly sources:

UC Irvine: https://archive.ics.uci.edu/ml/datasets.html
Or Kaggle: https://www.kaggle.com/datasets

Then, students can tweak their logistic regression models (or if they're very ambitious, look towards exploring other models) in order to make predictions based on datasets of their liking. This "capstone project" would probably have to be evaluated by teachers in-person (or maybe it's possible to set up a cs50 help forum or some sort of human communication channel for that).

10. [Note to Erin] Evaluation

Metrics such as cost and accuracy on a test data set can be used to determine if the students' models have been implemented correctly.

Sigmoid function section:

- Input a matrix of random numbers (with a set seed or something like that to keep it consistent) and ensure that the student's sigmoid function outputs the correct value.

Logistic model section:

- Check to make sure that the student's accuracy is around 90% on the test set and 92% on the training set
- It may be worth brainstorming some intermediate checks so that students will have an easier time debugging . . . oh yeah! I remember check50 having a list of smiley/neutral/sad faces describing which parts of the algorithm were working and which were not