ACES ML Project 2021
Introduction to Some Basic Algorithms

Ethan Stovall

Goal: This document will hopefully act as a quick intro to some of the basic algorithms you'll encounter in Machine Learning. The goal is to give enough intuition about their derivation so that the code implementations make sense in the applied problems. Note that all the topics we'll be working are examples of supervised learning. (This document heavily draws from the Stanford course CS229, which can be found at this link: http://cs229.stanford.edu/syllabus-fall2020.html)

I included the first three notes from the class syllabus in the project package. I wasn't able to finish past my own notes on Logistic Regression, but I included where to find the topics in the Stanford notes. My notes are taken from the class notes anyway, but I hope they can be helpful to clear up some of the questions or confusion that I myself had while going through them the first time. All notes for this project can be found in the `ACES_ML_Project/notes` folder.

Outline: This introduction will be divided into a 2-part introduction portion, and 3 coding portions, each one dealing with a respective dataset that we are trying to learn.

i. Notation and Terminology

ii. Linear Regression

iii. Dataset 1: Binary Classification of Tumors:

   (a) Discuss Logistic Regression and code it to predict whether tumors are cancerous or not.

   (b) Discuss the Perceptron Algorithm. **(cs229-notes1.pdf page 17)**

   (c) Discuss Gaussian Discriminant Analysis, and its connection to Logistic Regression. **(cs229-notes2.pdf page 2)**

iv. Dataset 2: Binary Classification of Mushrooms:

   (a) Discuss the Naive Bayes model and Laplace Smoothing, and implement them by hand. **(cs229-notes2.pdf page 8)**

   (b) Discuss Decision Trees and try out some built-in algorithms from scikit-learn. Compare their performance with Naive Bayes.

v. Dataset 3: Support Vector Machines (SVM) on a Non-Linearly-Separable Dataset:

   (a) Quick discussion on kernels and their use. **(cs229-notes3.pdf page 1 for kernels and page 11 for SVM)**

   (b) Use a built in SVM with RBF and compare to standard Logistic Regression.

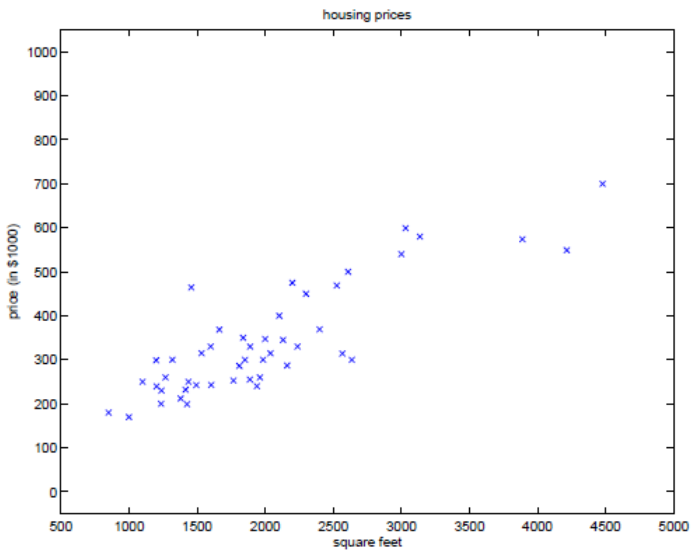
vi. Summary of Coding Portion

i. Introduction to Notation

A good example to start with is to try to predict housing prices based on their living area. Consider this dataset of 47 houses:

Living Area (sq. ft.)	Price (1000$)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

If we plot this data, it looks like this:



(a) House Prices by Living Area

The goal is to learn this data and then output a prediction for a house price given a new living area. Before we do this, we'll establish some notation that is usually pretty standard in ML:

- **Features:** We'll use $x^{(i)}$ to denote the input features of our ith training example. In this case, we only have one input feature, the living area, so then

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \end{bmatrix}$$

However, if we had another input feature, like the number of bedrooms, we would represent $x^{(i)}$ like this:

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

We can go ahead and extend this to when we have a lot of features, say n, which would look like:

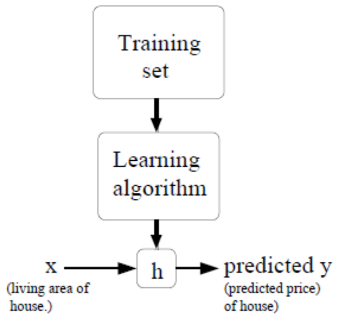
$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix}$$

- **Target:** The target variable is our output. We denote it $y^{(i)}$. For our purposes, $y^{(i)}$ will be a scalar, but there are datasets that have multiple target variables, and models that give a corresponding output.
- **Training Example:** Each "input"/"output" pair will be written $(x^{(i)}, y^{(i)})$. In this case, $x^{(i)}$ is the living area, and $y^{(i)}$ is the housing price.
- **Training Set:** The list of m training examples $\{(x^{(i)}, y^{(i)}), i = 1, 2, ..., m\}$ is called our training set.
- **Design Matrix:** The design matrix is often denoted X and is an $m \times n$ matrix representation of every training example and its features. We can write the target variable vector $\vec{y}$ as an $m \times 1$ vector:

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \dots & x_n^{(m)} \end{bmatrix} = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

  Note that in the design matrix, each row is the transpose of our feature vector for the corresponding training example.

- **Hypothesis:** The function $h(x)$ that we use to predict y from a given x is called the hypothesis function. Our goal is ultimately to get an accurate prediction by the route in this chart:



- **Regression Problem**: The y that we're trying to predict is a continuous variable (e.g. house price).

- **Classification Problem**: The y that we're trying to predict is a discrete variable (e.g. cancerous or benign tumor).

ii. Intro to Linear Regression

To make the earlier housing example a little bit more robust, we'll add in the number of bedrooms as a feature:

Living Area (sq. ft.)	# Bedrooms	Price (1000$)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
⋮	⋮	⋮

In this new problem, every $x^{(i)}$ is a two-dimensional vector such that

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

with $x_1^{(i)}$ as the living area and $x_2^{(i)}$ as the number of bedrooms. Now we want to define our hypothesis function $h(x)$, which will allow us to make a prediction about the price of a house based on these features. A common choice for $h(x)$ is just a linear function of our x values, like so:

$$h(x^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}$$

Here, the θ values are called **parameters**, or **weights**. It is conventional to let x_0 be the intercept term in our x vector, and set it equal to 1. Keep in mind that $x^{(i)}$ and θ are vectors:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

We can then write the hypothesis function like this:

$$h(x^{(i)}) = \sum_{j=0}^{n} \theta_i x_j^{(i)} = \theta^T x^{(i)}$$

Here, n is the number of features in our x vector (excluding x_0), and the rightmost part of the equation is in vector notation.

As a quick note: I find it helpful to think of the parameters θ more as weights. That is, you're trying to figure out how to add up the features x so that they give you y. If the square footage of a home is more important in determining its price than the number of bedrooms, then we want the x_1 value to have more weight than the x_2 value in our sum. So we will make the parameter we're multiplying it with, θ_1, larger than the parameter θ_2 for x_2.

Now, the trick is to figure out how our program will "learn" our θ values to be the best ones for an accurate prediction. To start, we'll define a way to measure the "goodness" of our prediction

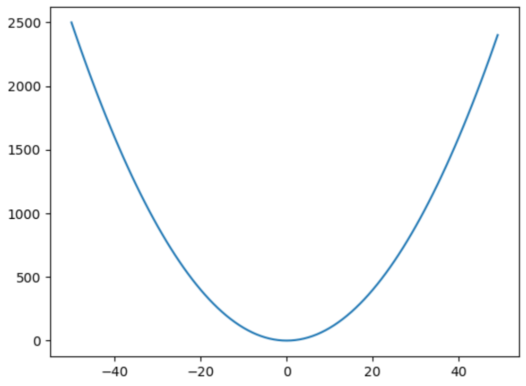
$h(x)$. Naturally, we'd imagine that we want $h(x)$ to be as close to the true y as possible, so we define the **cost function** to measure this:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2$$

This is called the Least Squares cost function, and it's the central piece of the **ordinary least squares model**. More specifically, this is the Average Least Squares cost function. Notice that it is the average of sum of the squared differences between our prediction and the true y value for every training example $(x^{(i)}, y^{(i)}), i = \{1, ..., m\}$.

The Least Mean Squares (LMS) Algorithm

For the best prediction, we want to minimize $J(\theta)$. We can take a look at the function and notice that it is quadratic. Note that for some one-dimensional x, a simple quadratic function (x^2) looks like this:



There is a clear global minimum. It turns out that even when x has many dimensions (features), the LMS cost function $J(\theta)$ always has a global minimum due to its quadratic form. That said, we can now perform **gradient descent** to move along the steepest slope towards this minimum by modifying our θ values. To do this, we can initialize θ as a vector of random values. Then, we will find the partial derivative (slope) of $J(\theta)$ with respect to each parameter θ_j. We want to take a step down this slope in the direction of every θ_j, so that we get closer and closer to the global minimum of $J(\theta)$. That is, we will apply the following update rule to θ:

For every θ_j in $\theta = \begin{bmatrix}\theta_0 \\ \vdots \\ \theta_n\end{bmatrix}$,

$$\theta_j := \theta_j - \alpha\frac{\partial J}{\partial \theta_j}$$

In the above equation, α is the **learning rate**. Think of this like the size of the step we are taking down. If it's too big, we might overshoot the minimum. If it's too small, our algorithm might never converge. The last thing we need is the expression for $\frac{\partial J}{\partial \theta_j}$. I'll write the expression for the derivation here, but feel free to skip it.

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2 \\
&= \left(2 \cdot \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})\right) \frac{\partial}{\partial \theta_j} (h(x^{(i)}) - y^{(i)}) \\
&= \left(\frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})\right) \frac{\partial}{\partial \theta_j} \left((\theta_0 x_0 + ... + \theta_j x_j + ... + \theta_n x_n) - y^{(i)}\right) \\
&= \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)}) x_j
\end{aligned}$$

The purpose of the factor $\frac{1}{m}$ is to normalize the data based on the number of examples. To give an example of why this is important, if we had $m = 10$ examples, all with a squared error of 1, our $J(\theta)$ would be 10 without division by m. If we had 100 examples, $J(\theta)$ would be 100. In the second case, our step size would be 10 times larger. Dividing by m makes sure that our algorithm works regardless of how many examples we have.

(**NOTE:** Since we'll be working with numpy a lot, it helps to be able to think about these updates in terms of vectors (e.g. θ), rather than the individual values in the vector (e.g. θ_j). So when we're talking about updating the vector θ, we need a vector representation for $\frac{\partial J}{\partial \theta_j}$ too. Remember that $J(\theta)$ is a function of a vector. The "derivative" of a such a function is called the **gradient**, shown here:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

Keep this in mind for the following steps.)

This gives us the update rule by which we minimize $J(\theta)$. So in code, our procedure would look like:

1. Prepare our training set. Create an $m \times n$ numpy 2d array for our design matrix X and an $m \times 1$ numpy array for our target variable $\vec{y}$.

2. Initialize an $n \times 1$ array of parameters θ.

3. Calculate $h(x^{(i)}) = \theta^T x^{(i)}$ for all training examples. **Note:** Here, since we're using the design matrix X, rather than a single training example, we will instead write $X\theta$. This product of an $m \times n$ array and an $n \times 1$ array will produce an $m \times 1$ array of our predictions $h(x^{(i)})$ for every example. Refer to the definition of the design matrix X to see that the multiplication of each row with θ is the same as $\theta^T x^{(i)}$ for any training example i.

4. Calculate the gradient of the cost function,

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)}) x_j$$

In vector notation, this is:

$$\begin{aligned}\nabla_\theta J(\theta) &= \frac{1}{m}\left(\left(X\theta - \vec{y}\right)^T X\right)^T \\ &= \frac{1}{m}X^T\left(X\theta - \vec{y}\right)\end{aligned}$$

5. Complete the update on θ:

$$\theta_j := \theta_j - \alpha\frac{\partial J}{\partial \theta_j}$$

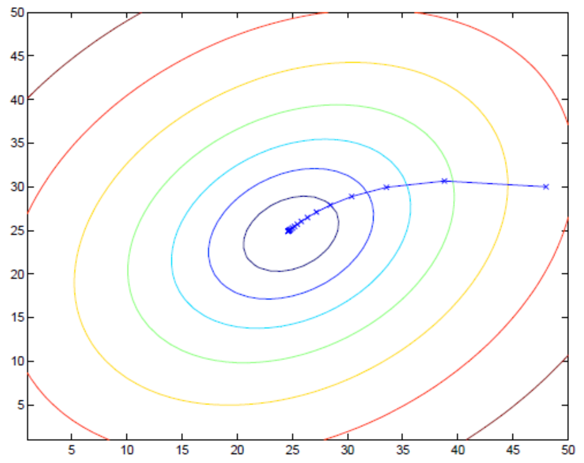
In vector notation, this is:

$$\theta := \theta - \alpha\nabla_\theta J(\theta)$$

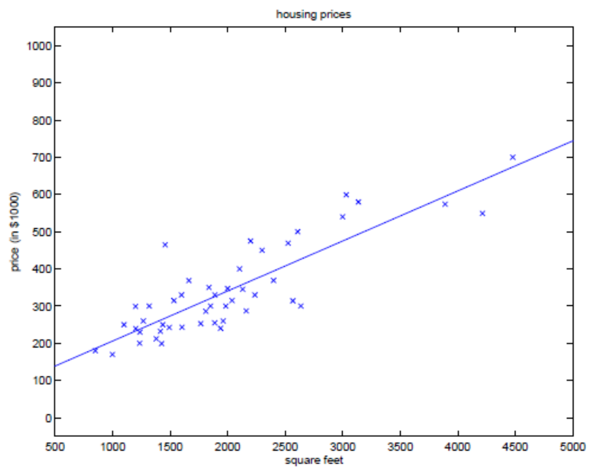
6. Repeat steps 3-5 until convergence.

As a note, step 4 can be difficult to get straight mentally. I find it's helpful to write out all the matrix multiplication on paper so that I can see how it condenses to the summand expression. In the end though, you'll just want to multiply your numpy arrays the same way it's shown here. The numpy package is really fast and efficient at that, so it won't take any for loops or many lines of code. We won't be coding this one anyway, so you can just try to get a feel for it mentally.

The end goal is for θ to step down to the global minimum of $J(\theta)$, as seen in the contour plot of this (unrelated) quadratic where $\theta = \begin{bmatrix}\theta_0 \\ \theta_1\end{bmatrix}$ is initialized to $\begin{bmatrix}48 \\ 30\end{bmatrix}$:



In the first example, with the price as a function of the living area, this updated θ gives us a line of best fit for the data, allowing us to predict y with $h(x^{(i)}) = \theta^T x^{(i)}$:



If you notice the update rule above, you'll see that we sum the squared difference for all our training examples before we subtract out the update to our θ. This sort of allows our algorithm to take in a lot of info about the the best direction to go before updating, but it can be very slow. This is called **batch gradient descent**. An alternate approach, called **stochastic gradient descent**, makes an update to θ after calculating the squared difference for one example. This is faster to converge, but doesn't step to the global minimum of $J(\theta)$ as directly, and you're not guaranteed it will reach it exactly. Here are the basic procedures for both:

- **Batch Gradient Descent:**

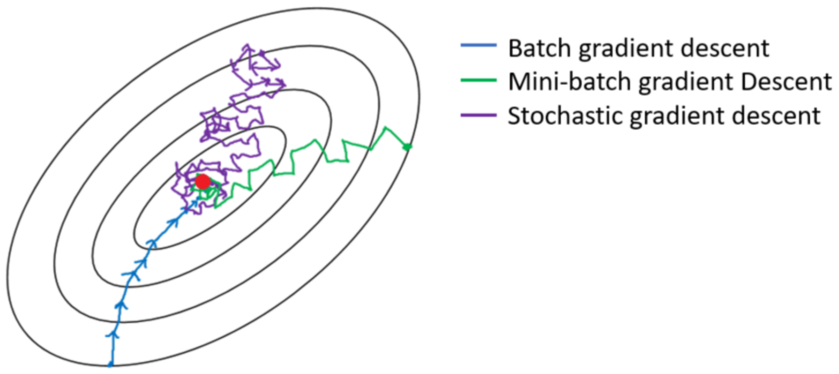
```
Repeat until convergence {
    $\theta_j := \theta_j + \frac{\alpha}{m} \sum_{i=1}^{m} \big(y^{(i)} - h(x^{(i)})\big) x_j$ (for every $j$)
}
```

- **Stochastic Gradient Descent:**

```
Repeat until convergence {
    for $i = 1$ to $m$ {
        $\theta_j := \theta_j + \alpha\big(y^{(i)} - h(x^{(i)})\big) x_j$ (for every $j$)
    }
}
```

Note: In the above update, we're adding the update, rather than subtracting, because we've negated the $\big(h(x^{(i)}) - y^{(i)}\big)$ expression from earlier. We're still subtracting the update, though, and descending $J(\theta)$. Also, we don't divide by m in the stochastic descent rule because each step is produced by only one example.

The image below shows the difference in convergence between batch and stochastic, along with mini-batch gradient descent, which is a mix of the two approaches.


Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent

iii. Binary Classification of Tumors

Here, we'll be working with the **binary classification** problem, in which our y variable can only take two values, 0 or 1, based on the features in our training example. For the dataset we'll be using logistic regression, $x^{(i)}$ will be the features of a tumor, and $y^{(i)}$ will be 1 if the tumor is cancerous and 0 otherwise. 0 is called the **negative class**, and 1 is called the **positive class**, sometimes denoted "-" and "+", respectively. Given $x^{(i)}$, the corresponding $y^{(i)}$ is called the **label** for the training example.

Logistic Regression

For this section, we'll be thinking more in terms of probabilities. Rather outputting a value for y as a linear function of x, here we'll be trying to figure out the **probability** of the y label (0 or 1), given our features x.

To reiterate, now that we're trying to predict a value that is 0 or 1, it doesn't make sense to use the same hypothesis function

$$h(x^{(i)}) = \theta^T x^{(i)}$$

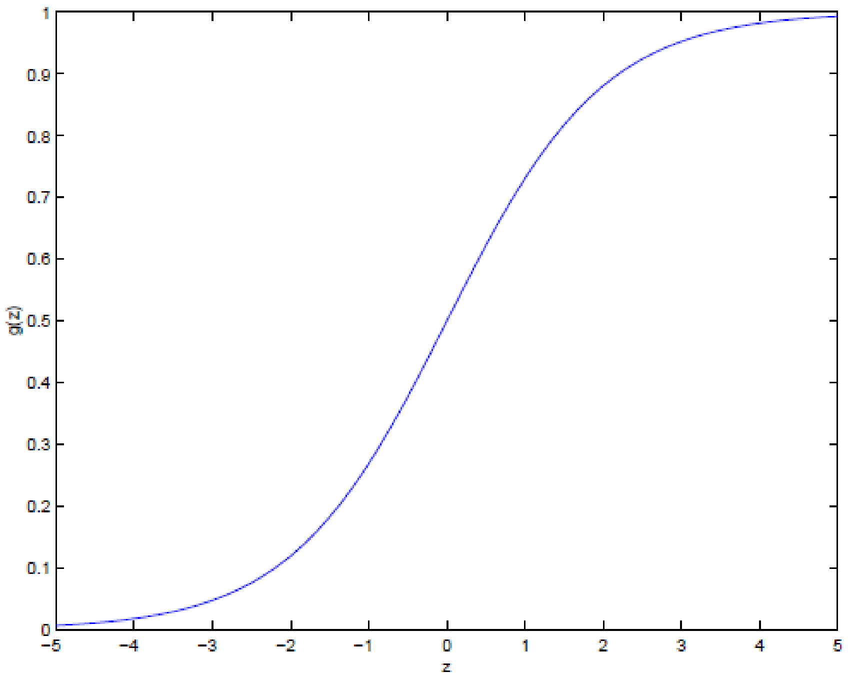
to predict our labels, since that is a linear function that can take values larger or smaller than this range. Instead, we'll pick the following function as our hypothesis:

$$h(x^{(i)}) = g(\theta^T x^{(i)}) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

where

$$g(z) = \frac{1}{1 + e^{-z}}.$$

Note that $g(z)$ is a special function called the **logistic** or **sigmoid** function. Here is a plot of it:



You can see that $g(z)$ tends toward 0 as $z \to -\infty$ and toward 1 as $z \to \infty$. So $g(z)$, and therefore $h(x)$, is bounded by 0 and 1. So then when we have large $\theta^T x^{(i)}$, our hypothesis function will be close to 1, and when we have small $\theta^T x^{(i)}$, it will be close to 0. You can probably imagine that the value of the hypothesis function will act as a probability that our example has a positive label. That is, $h(x^{(i)}) = P(y^{(i)} = 1|x^{(i)};\theta)$. You can read this as "the probability of the label $y^{(i)}$ being 1, given your features $x^{(i)}$ and parameterized by θ." We don't condition on θ, like this: $P(y^{(i)} = 1|x^{(i)}, \theta)$, because θ is not a random variable here. So just like that, we now have a function to predict the probability of a label, given some features. Let's go ahead and define the probability of y for both positive and negative labels:

$$P(y^{(i)} = 1|x^{(i)};\theta) = h(x^{(i)})$$

$$P(y^{(i)} = 0|x^{(i)};\theta) = 1 - h(x^{(i)})$$

We can shorten this to a single equation:

$$p(y^{(i)}|x^{(i)};\theta) = \big(h(x^{(i)})\big)^{y^{(i)}} \big(1 - h(x^{(i)})\big)^{1-y^{(i)}}$$

Keep in mind that $y^{(i)}$ is either 0 or 1, and a any number raised to the power 0 is just 1.

If you're curious about why the sigmoid function is a good choice for our hypothesis, I would definitely recommend checking out Andrew Ng's notes on Generalized Linear Models from the Stanford CS229 course **(cs229-notes1.pdf page 20)**. They explain how if you make certain assumptions about the probability distributions of your data, the sigmoid function will naturally fall out as the true expression for $p(y^{(i)}|x^{(i)};\theta)$.

So how do we fit the θ parameters for our logistic regression model? Doing this will take a little more work than linear regression. For linear regression, we were able to pick the convenient Least Squares cost function, and then apply gradient descent to it to reach the best θ. For logistic regression we need to find a new measure of "goodness" for our predictions.

We've already decided we can represent $p(y^{(i)}|x^{(i)};\theta)$ as the sigmoid function, $g(\theta^T x^{(i)})$. Let's now also assume that $y^{(i)}|x^{(i)};\theta$ is an IID (identically and independently distributed) variable. Imagine we have a training set of 2 examples. This independence means if we want the joint probability of $y^{(1)}$ and $y^{(2)}$ being our labels for training examples $x^{(1)}$ and $x^{(2)}$, respectively, we can just multiply their probabilities together. If we generalize this, then the probability of our entire training set could be written as

$$p(\vec{y}|X;\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta)$$

This already seems like a great starting point. If the probability of our labels given the features is low, our θ needs tweaking. If it's high, our θ is well-fitted. If we could make this expression a function of θ instead of y, we could take the gradient with respect to θ just like before. As it turns out, the **probability** of the training set is actually equivalent to the **likelihood** of θ. That is:

$$L(\theta) = L(\theta;\vec{y},X) = p(\vec{y}|X;\theta)$$

So we can go ahead and write the likelihood of θ as:

$$\begin{aligned}L(\theta) &= p(\vec{y}|X;\theta)\\ &= \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta)\\ &= \prod_{i=1}^{m} \big(h(x^{(i)})\big)^{y^{(i)}} \big(1-h(x^{(i)})\big)^{1-y^{(i)}}\end{aligned}$$

In this case, we're trying to maximize $L(\theta)$. Those exponents would make taking the derivative a hassle, though, so the common thing is to take the logarithm of $L(\theta)$. Maximizing the log of $L(\theta)$ will give us the same θ as maximizing $L(\theta)$, and now we get the bonus that these exponents become coefficients instead. It also changes the product to a summation:

$$\begin{aligned}\ell(\theta) &= log L(\theta)\\ &= log\Bigg(\prod_{i=1}^{m} \big(h(x^{(i)})\big)^{y^{(i)}} \big(1-h(x^{(i)})\big)^{1-y^{(i)}}\Bigg)\\ &= \sum_{i=1}^{m} log\Bigg(\big(h(x^{(i)})\big)^{y^{(i)}}\Bigg) + log\Bigg(\big(1-h(x^{(i)})\big)^{1-y^{(i)}}\Bigg)\\ &= \sum_{i=1}^{m} y^{(i)} log\big(h(x^{(i)})\big) + (1-y^{(i)}) log\big(1-h(x^{(i)})\big)\end{aligned}$$

This is called the **log likelihood**. Now we can maximize this function. To do this, we'll find its gradient, and then we'll apply the update rule:

$$\theta := \theta + \alpha \nabla_\theta \ell(\theta)$$

Since we're maximizing instead of minimizing this time, we add the update instead of subtracting it. This is **gradient ascent**.

Before we find the gradient of this function, here's a useful property of the sigmoid function, $g(z)$:

$$\begin{aligned}g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}}\\ &= -\frac{1}{(1+e^{-z})^2}\cdot\frac{d}{dz}e^{-z}\\ &= -\frac{1}{1+e^{-z}}\cdot\frac{-e^{-z}}{1+e^{-z}}\\ &= \frac{1}{1+e^{-z}}\cdot\frac{e^{-z}}{1+e^{-z}}\\ &= \frac{1}{1+e^{-z}}\cdot\frac{e^{-z}+1-1}{1+e^{-z}}\\ &= \frac{1}{1+e^{-z}}\cdot\Big(\frac{1+e^{-z}}{1+e^{-z}}-\frac{1}{1+e^{-z}}\Big)\\ &= \frac{1}{1+e^{-z}}\cdot\Big(1-\frac{1}{1+e^{-z}}\Big)\\ &= g(z)\big(1-g(z)\big)\end{aligned}$$

Now that we have a shortened form for the derivative our hypothesis function, it will be easier to calculate the gradient of $\ell(\theta)$. Remember that

$$\nabla_\theta \ell(\theta) = \begin{bmatrix} \frac{\partial \ell(\theta)}{\partial \theta_0} \\ \frac{\partial \ell(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial \ell(\theta)}{\partial \theta_n} \end{bmatrix},$$

and

$$h(x^{(i)}) = g(\theta^T x^{(i)}).$$

Let's just calculate the derivative for one element j of the gradient vector. Remember that even though there's a lot of visual clutter from the xs and ys, those are just constants when we're taking the derivative w.r.t. θ_j:

$$\begin{aligned}
\frac{\partial \ell(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} \sum_{i=1}^{m} y^{(i)} log\big(g(\theta^T x^{(i)})\big) + (1-y^{(i)}) log\big(1-g(\theta^T x^{(i)})\big) \\
&= \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} \Big(y^{(i)} log\big(g(\theta^T x^{(i)})\big)\Big) + \frac{\partial}{\partial \theta_j} \Big((1-y^{(i)}) log\big(1-g(\theta^T x^{(i)})\big)\Big) \\
&= \sum_{i=1}^{m} \left(y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1-y^{(i)}) \frac{1}{1-g(\theta^T x^{(i)})} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x^{(i)}) \\
&= \sum_{i=1}^{m} \left(y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1-y^{(i)}) \frac{1}{1-g(\theta^T x^{(i)})} \right) g(\theta^T x^{(i)})\big(1-g(\theta^T x^{(i)})\big) \frac{\partial}{\partial \theta_j} (\theta^T x^{(i)}) \\
&= \sum_{i=1}^{m} \Big(y^{(i)}\big(1-g(\theta^T x^{(i)})\big) - (1-y^{(i)}) g(\theta^T x^{(i)}) \Big) \frac{\partial}{\partial \theta_j} (\theta_0 x_0^{(i)} + ... + \theta_j x_j^{(i)} + ... + \theta_n x_n^{(i)}) \\
&= \sum_{i=1}^{m} \big(y^{(i)} - g(\theta^T x^{(i)})\big) x_j^{(i)} \\
&= \sum_{i=1}^{m} \big(y^{(i)} - h(x^{(i)})\big) x_j^{(i)}
\end{aligned}$$

To make sure that the step size is independent of the number of examples, we'll redefine the gradient with the $\frac{1}{m}$ factor:

$$\frac{\partial \ell(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \big(y^{(i)} - h(x^{(i)})\big) x_j^{(i)}$$

And after all that we've got our batch gradient ascent rule for logistic regression. Let's write out the rules like last time:

- **Batch Gradient Descent:**

```
Repeat until convergence {
    $\theta_j := \theta_j + \alpha \frac{1}{m} \sum_{i=1}^{m} \big(y^{(i)} - h(x^{(i)})\big) x_j$ (for every $j$)
}
```

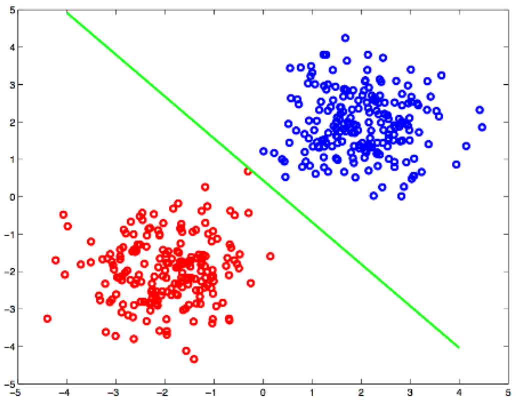
- **Stochastic Gradient Descent:**

```
Repeat until convergence {
    for $i = 1$ to $m$ {
        $\theta_j := \theta_j + \alpha\big(y^{(i)} - h(x^{(i)})\big)x_j$ (for every $j$)
    }
}
```

Note: You'll notice that these appear to be the exact same update rules as in linear regression. There are 2 key differences though:

1. Our hypothesis function $h(x)$ is the sigmoid here, so the values these update rules produce will be significantly different.
2. Whereas the Least Squares cost function had a global minimum, the log likelihood $\ell(\theta)$ has a global maximum, so adding the slope $\big(y^{(i)} - h(x^{(i)})\big)x_j$ to our θ is equivalent to ascending rather than descending $\ell(\theta)$. As a result, we maximize the likelihood of our parameters given the data.

Note on Linear Separability: Another way to think about what this algorithm is doing is to consider this image:



In this example we have two features (x_1 is the x-axis and x_2 is the y-axis). For any training example i,

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$$

(remember $x_0^{(i)} = 1$ is the intercept term). Depending on these values, each training example is labeled either red (0) or blue (1). The green line is the linear separator of these two groups. That is, the exact line at which $P(y^{(i)} = 1|x^{(i)};\theta) = \frac{1}{2}$. Since we know $p(y^{(i)} = 1|x^{(i)};\theta) = \frac{1}{1+e^{-\theta^T x^{(i)}}}$, we can find this line by solving $\frac{1}{1+e^{-\theta^T x^{(i)}}} = \frac{1}{2}$. By looking at the equation, we can see we need the lefthand denominator to equal 2:

$$\begin{aligned} 1 + e^{-\theta^T x^{(i)}} &= 2 \\ e^{-\theta^T x^{(i)}} &= 1 \\ \implies \theta^T x^{(i)} &= 0 \end{aligned}$$

So we now see that the dividing line defined by θ is given by:

$$\theta^T x^{(i)} = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + ... + \theta_n x_n^{(i)} = 0$$

Every time we update θ, this line more accurately separates the two classes. For data that is not linearly separable, we'll have to use a different approach (like a nonlinear kernel, for example). The final, fitted θ is the argument that maximizes $L(\theta)$, and we can write it as $\hat{\theta} = argmax L(\theta)$. This is called the **Maximum Likelihood Estimation (MLE)**.

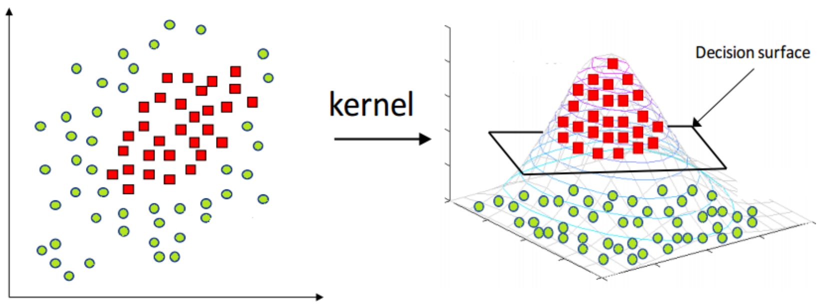
iv. Binary Classification of Mushrooms

Here, we're still working with the **binary classification** problem, but now our features are categorical rather than continuous. In these cases, other approaches like a Naive Bayes or Decision Tree Model are likely to be more effective than a logistic regression model. I wasn't able to write out my full notes on these models, but you can find the explanations I would be summarizing here:

- Naive Bayes: **(cs229-notes2.pdf page 8)**
- Decision Trees: See if you can find some good online sources for this. It hasn't been something I've seen widely implemented by hand, so I just used scikit-learn.

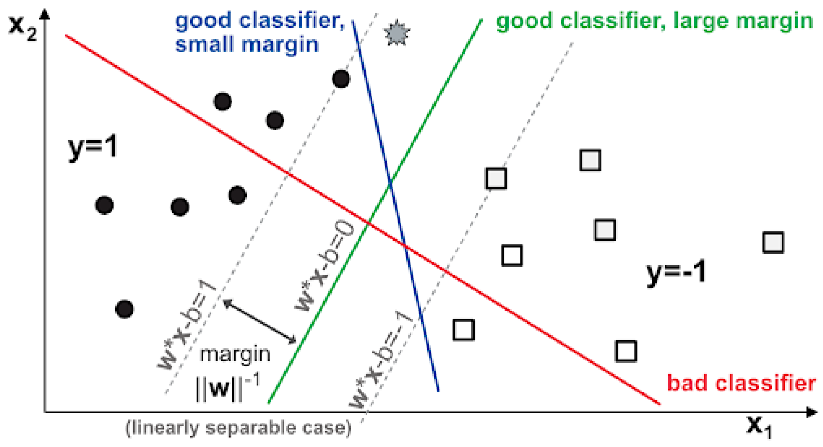
v. Non-Linearly Separable Data

When your data is not linearly separable, like it was in the logistic regression examples, you'll need to find some other method to classify it. One method is the use of kernels, which sort of add new dimensions to your data by projecting it into a higher dimension so that you can see the separations in the new space. Then you're able to find the separating hyperplane, and collapse the data back down to its original form with this knowledge. Here's a picture of this concept in action:



On the left, you can see there's no line in 2 dimensions that we can use to separate the data. But if we map our data to 3 dimensions using a special kernel function, now we can see very clearly a plane that separates the classes. Collapsing it back down to two dimensions, it will look like an ellipse. You can find a detailed discussion of kernels on page 1 of **cs229-notes3.pdf**.

Support Vector Machines (SVM) also come up frequently when discussing kernels, as they often employ powerful kernel functions to separate data. The basic idea behind SVM is to find the separator with the greatest possible distance on both sides to the nearest data point. Here's an illustration:


x2
good classifier, small margin
good classifier, large margin
y=1
y=-1
w*x-b=1
w*x-b=0
w*x-b=-1
margin ||w||^-1
bad classifier
(linearly separable case)
x1

You can see that of the two lines that separate the data, the green one is ideal, because it maximizes the distance between the line and the datapoints, thus maximizing the algorithm's certainty in its prediction. You can read more about SVM on page 11 of **cs229-notes3.pdf**.

vi. Coding Portion

Here I'll include a summary of the coding portions of the project so you know how to navigate and change everything.

README

- The README file will explain how to set up all the tools like Pycharm and Jupyter Lab on your computer, along with installing the necessary python packages you'll need.

Data folder

- Logistic Regression Intro: the two files `logreg_intro_train.csv` and `logreg_intro_test.csv` are for the intro notebook. There are only two features so you'll be able to plot the linear separator over the data. I wound up generating my own data because I couldn't get my own logistic regression to converge on them, but they will definitely work with Newton's method for Logistic Regression if you ever try that.

- Breast cancer datasets: There are three csv files. One is the full csv of the data, and the other two are the test and train splits I made with the file `split_train_test.py`. You can run this file from the command line too to make your own splits, but honestly if you're using scikit-learn, it's often easier to process all your data in one csv, then split it up after with the train_test_split module. `fraction_xy.py` is just the original file from which I wrote `split_train_test.py`.

- Mushroom datasets: There are two files for the mushroom train and test splits.

- Non-Linearly Separable datasets: `nonlinsep_train.csv` and `nonlinsep_test.csv` are files that would be perfect for trying kernel methods and SVM, since they're 2d and allow you to see the nonlinear separator.

Python Files

- `util.py`: This file contains several functions that are specific to data loading and plotting for the hand-coded portions of `logreg_intro.ipynb` They are largely from the CS229 Stanford course.
- `linear_model.py`: Base class for creating linear and logistic regression models by hand.
- `LogisticRegression.py`: Empty template to extend the LinearModel class in `logreg_intro.ipynb`.
- `LogisticRegression_key.py`: Key for the Logistic Regression model. I wrote this to give a guide of how to write your file. I think it pretty consistently converges on the generated data.

Jupyter Notebook Files

- These files will be where you'll spend most of your time. This is where you bring in all the previously mentioned files/classes through module importing and data loading to actually examine your data and run the algorithms.
- `logreg_intro.ipynb`: Here, you'll be able to hand-implement your own Logistic Regression algorithm, and compare it to the scikit-learn version.
- `breast_cancer_logreg.ipynb`: Here, I loaded our breast cancer dataset and ran scikit-learn's logistic regression on it. Feel free to follow through and see if you can get the hand-made model to work on it (though I bet it will require some more optimized code with more advanced methods).
- `mushroom_visualization.ipynb`: This is a really neat file I got from Kaggle and modified slightly for our dataset (they're only slightly different). It shows how you can visualize your data to give you an idea what features to pay attention to. I credited the author in the file.
- `mushroom_algorithms.ipynb`: This is a basic version of another Kaggle file I found in which we learn the mushroom data with scikit-learn's Gaussian Naive Bayes model, Decision Tree, and Logistic Regression. I credited the author in the file.
- I didn't get around to making a notebook for the kernel and SVM topic. You can maybe look up some SVM implementations to try them out on the `nonlinsep_train/test.csv` dataset though.

Closing Note: *This isn't a comprehensive tutorial of ML or anything, but I hope it gives you a good place to start, and maybe helps to centralize and discuss some topics that you've heard mentioned here and there but never had much context for. Again, most all the information comes from Stanford's CS229 course, so feel free to look into the class page and Andrew Ng's course on Coursera. Thanks for checking this out; I learned a lot putting it together!*