An Introduction to Artificial Intelligence and Machine Learning Algorithms

BRANDON JOHN GRENIER

Copyright © 2018 Brandon John Grenier

All rights reserved.

ISBN: 978-1-726-60544-1

TABLE OF CONTENTS

[An Introduction to Artificial Intelligence and Machine Learning 1](#_Toc527585821)

[What is Artificial Intelligence? 1](#_Toc527585822)

[What is Machine Learning? 2](#_Toc527585823)

[Approaches to Machine Learning 2](#_Toc527585824)

[Supervised Learning: Machines that learn to recognize relationships 2](#_Toc527585825)

[Unsupervised Learning: Machines that learn to discover patterns 3](#_Toc527585826)

[Reinforcement Learning: Machines that learn from their environment 4](#_Toc527585827)

[Predictive Models in Machine Learning 5](#_Toc527585828)

[Classification: Assign membership to an object 5](#_Toc527585829)

[Regression: Predict a continuous value 5](#_Toc527585830)

[Clustering: Discover new patterns and relationships in your data 6](#_Toc527585831)

[Dimensionality Reduction: Reduce the number of variables in your data 6](#_Toc527585832)

[Linear Regression with One Feature 7](#_Toc527585833)

[Intuitive Linear Regression 8](#_Toc527585834)

[Step 1: Plot the data 9](#_Toc527585835)

[Step 2: Fit the data 10](#_Toc527585836)

[Step 3: Make predictions 11](#_Toc527585837)

[The Hypothesis Function: Model your problem space 12](#_Toc527585838)

[The Linear Regression Hypothesis Function 14](#_Toc527585839)

[The Cost Function: Get feedback on model performance 15](#_Toc527585840)

[The Cost Function 16](#_Toc527585841)

[The Cost Function in Practice 17](#_Toc527585842)

[The Gradient Descent Algorithm: Use feedback to find the best predictor 20](#_Toc527585843)

[Intuitive Gradient Descent 20](#_Toc527585844)

[The Gradient Descent Algorithm 23](#_Toc527585845)

[Convergence in Gradient Descent 26](#_Toc527585846)

[The Linear Regression Learning Rate 29](#_Toc527585847)

[Linear Regression Summary 32](#_Toc527585848)

[Linear Regression with Multiple Features 34](#_Toc527585849)

[The Hypothesis Function 35](#_Toc527585850)

[The Cost Function 36](#_Toc527585851)

[The Gradient Descent Algorithm 37](#_Toc527585852)

[Feature Scaling 41](#_Toc527585853)

[Polynomial Regression 44](#_Toc527585854)

[Polynomial Functions 45](#_Toc527585855)

[Expressing Polynomial Functions in Data 46](#_Toc527585856)

[Logistic Regression 49](#_Toc527585857)

[Classification, Part 1: Binary Classification 49](#_Toc527585858)

[Overview of Binary Classification 49](#_Toc527585859)

[Shortcomings with using Linear Regression 50](#_Toc527585860)

[The Sigmoid Function 52](#_Toc527585861)

[The Hypothesis Function 53](#_Toc527585862)

[The Decision Boundary 54](#_Toc527585863)

[The Decision Boundary in Practice 56](#_Toc527585864)

[Support Vector Machines 57](#_Toc527585865)

[K-Means Clustering 58](#_Toc527585866)

[Principal Component Analysis 59](#_Toc527585867)

[Neural Networks 60](#_Toc527585868)

[Anomaly Detection 61](#_Toc527585869)

[Recommender Systems 62](#_Toc527585870)

[Applying Machine Learning in Practice 63](#_Toc527585871)

[Applying Machine Learning at Scale 64](#_Toc527585872)

[Appendix A: Linear Functions Review 65](#_Toc527585873)

[Appendix B: Matrix Algebra Review 67](#_Toc527585874)

PREFACE

Placeholder for preface content. ACKNOWLEDGMENTS

Placeholder for acknowledgment content.

# An Introduction to Artificial Intelligence and Machine Learning

## What is Artificial Intelligence?

First coined in 1956 by John McCarthy, Artificial Intelligence (AI) broadly describes computer systems that have the ability to perform tasks that mimic human intelligence. Creating computer systems that can successfully reason, plan, represent knowledge, learn, process natural language and perceive are all major goals of active AI research. Artificial Intelligence can be categorized into two different types, or forms of intelligence:

**General AI**, also known as Strong AI or Full AI is the canonical form of intelligence you’ll likely think about when you hear the term AI – this is your Skynet and Terminators. General AI would exhibit all of the characteristics of human intelligence, but is incredibly difficult to create and does not exist in any meaningful or material form today. The creation of General AI systems is the ultimate goal for AI researchers, who reserve the term *General AI* for machines that are capable of experiencing consciousness.

**Applied AI**, also known as Weak AI or Narrow AI is the most prevalent form of AI today. Unlike General AI, Applied AI exhibits very limited and very specific characteristics of human intelligence, but can incredibly effective when applied to very specific types of problems. Concrete examples of Applied AI include facial recognition systems, personal digital assistants, spam filters and autonomous vehicles. This book covers the approaches and algorithms used to develop Applied AI systems.

## What is Machine Learning?

Machine Learning (ML) is a field of computer science that gives computers the ability to learn without being explicitly programmed, and specifically focuses on the design and construction of algorithms that can both learn from and make predictions on data. Machine Learning is an approach to *achieving* AI. A more formal definition of Machine Learning was produced by Tom Mitchell in 1998:

*A computer program is said to learn from experience* ***E*** *with respect to some task* ***T*** *and some performance measure* ***P****, if its performance on T, as measured by P, improves with experience E.*

Simply put, if a computer program can improve the performance of a task by using previous experience, you can say that program has learned from the experience.

The ability for software to learn and improve from experience is significantly different from how traditional software is written. ML enables the development new types of applications that would either be impractical or impossible if the same application were to be programmed explicitly using traditional software engineering approaches.

## Approaches to Machine Learning

There are generally considered to be three broad machine learning approaches or types that can be used depending on your objectives and the nature of problem you’re trying to solve.

### Supervised Learning: Machines that learn to recognize relationships

Supervised learning leverages existing pairs of input and output data, with an expectation that some form of relationship exists between the input and output. A supervised learning algorithm will be trained to identify *known* relationships within an existing data set, and after training is complete the algorithm should be able to identify relationships with *novel* data it has not been exposed to before.

Spam filtering is a concrete example of supervised learning in practice. In this example, a data pair consists of an *email* as an input and a *classification* (spam mail or legitimate mail) as an output. We know that there is a relationship between an email and whether or not it’s considered spam – it could be the general language or specific words used in the email, the sender address or the number or types of links included in the email. There are a number of factors which will dictate whether an email is spam or not, and this is *exactly* what we want our algorithm to learn.

To train a spam filtering algorithm, it needs to be provided with sets of emails that are labeled as spam and sets of emails that are labelled as legitimate so that the algorithm can learn to identify the relationship between an email and it’s classification. With enough training data, the algorithm should be able to reliably identify an email as either legitimate or spam.

Another way to think about supervised learning is that it helps to solve problems where *you already know the right answers*, but you don’t have a comprehensive set of rules or insights to drive the outcome you’re looking for.

It’s relatively simple for a human to look at an email and decide whether it’s spam or not, but difficult to develop software with an explicit and concrete set of rules to achieve the same outcome. Supervised learning algorithms are incredibly effective in solving these classes of problems. This book will cover a number supervised learning algorithms in detail.

### Unsupervised Learning: Machines that learn to discover patterns

Unsupervised learning is a useful approach for solving *discovery* problems. A supervised learning algorithm will be trained to *identify known relationships* in your data, whereas an unsupervised learning algorithm will attempt to *discover unknown relationships* in your data. Unsupervised learning algorithms allow you to discover patterns and structure in your data that would otherwise be impractical or impossible.

Unsupervised learning algorithms are often used for grouping, clustering or segmenting data. If we used unsupervised learning for image recognition, we would provide the algorithm with a set of *unlabeled* images – in other words, we wouldn’t tell the algorithm what the images were. We would simply supply the images, and let the algorithm attempt to figure out what they were based on common characteristics that the algorithm discovers on its own.

The algorithm wouldn’t be able to label a particular *type* of object – put simply, the unsupervised learning algorithm will not make predictions like “this is a cat”, “this is a dog” or “this is a person”. However, what the algorithm will do is recognize similar patterns in images and group them together accordingly - it will create distinct clusters for humans, cats and dogs, and will also recognize that the pattern for a cat is different from the pattern for a dog or human. After the clustering is complete, humans can inspect the results for insights and labelling.

### Reinforcement Learning: Machines that learn from their environment

Reinforcement learning is a modern approach to machine learning that makes use of goal-oriented algorithms which learn how to accomplish an objective over many steps. For example, reinforcement learning algorithms are well suited to maximize the points won in a game over many moves. These algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones, which is why these types of algorithms are called reinforcement learning algorithms.

Reinforcement learning algorithms share the same discovery characteristic as unsupervised learning algorithms; neither are not provided with training data sets (i.e. the correct answers) ahead of time. In the absence of training data, reinforcement learning algorithms learn from experience by collecting data through trial-and-error; they sense and adapt to the environment around them.

In this sense, reinforcement learning algorithms share a similar data-driven learning approach as supervised learning algorithms; while supervised learning algorithms learn from data ahead of time, reinforcement learning algorithms learn from data as it becomes available in their environment.

The ability for software to sense, learn from, and then adapt to changes in an environment makes reinforcement learning algorithms well suited for advanced problems spaces like autonomous vehicle design, robotics and supply chain optimization.

While we won’t be covering reinforcement learning in this book, we wanted to include a section on reinforcement learning so you can gain a bit of insight and appreciation around how reinforcement learning works. Reinforcement learning is an exciting and advanced area of research in Machine Learning, but still has a few practical challenges to overcome in order to successfully implement these types of algorithms in real-world applications.

## Predictive Models in Machine Learning

A number of predictive models are employed in machine learning to solve particular classes of problems. One of the first things you’ll need to do as an AI/ML designer is understand the specific problem space you’re working in, so that you can leverage the right predictive model and its associated machine learning algorithms to get the outcome you’re looking for.

The predictive model that you choose will also be driven by the ML approach you’ve taken. Supervised learning will make use of *classification* or *regression* models, and unsupervised learning will make use of *clustering* or *dimensionality reduction* models.

This section provides a brief overview of each predictive model and the problems they aim to solve; you’ll learn quite a bit more about these models in this book.

### Classification: Assign membership to an object

Classification identifies group membership of an object by predicting a discrete value from a well-defined, finite set of possible outcomes. Put simply, classification algorithms identify the best group for an object, given all of possible groups the algorithm is allowed to choose from.

Spam filtering is a concrete example of a classification problem. A spam filtering algorithm will have two possible options, or groups to choose from: a *spam email* group, or a *legitimate email* group. For every email (the *object*) we want the algorithm to choose, or identify the best group for the email out of the two options that are available – the email is either spam or legitimate.

### Regression: Predict a continuous value

Regression estimates, or predicts a continuous value. A continuous value is a *real* value, such as an integer or floating point number. Regression algorithms will often predict quantities, such as amounts and sizes. For example, a house may be predicted to sell for a specific dollar value, or a stock may be predicted to rise to a specific price.

### Clustering: Discover new patterns and relationships in your data

Clustering assigns objects to groups while ensuring that objects in different groups are not similar to each other. Clustering aims to discover the hidden patterns in, and structure of your data. Each object is described by a set features, or characteristics.

### Dimensionality Reduction: Reduce the number of variables in your data

Dimensionality reduction reduces the number of object features or *dimensions* under consideration.

Feature reduction can be achieved by *feature selection*, where a subset of features are selected from the existing features, or by *feature extraction*, where features are extracted by combining existing features.

# Linear Regression with One Feature

Linear regression is one of the most well-known and well understood algorithms in machine learning. In this chapter you will learn about the linear regression algorithm and how it works in detail. We’ll do this by going through concrete examples and demonstrate how you can start by making make simple, and then more sophisticated predictions using linear regression.

We’ll begin by making some entirely intuitive predictions – no math or algorithms. We’ll formalize this intuition by introducing key machine learning concepts, mathematics and terminology to get you ready to dive into your first machine learning algorithm.

Next, you’ll learn how to build a supervised machine learning algorithm that will be able to make predictions using a single variable, or *feature*. Linear regression that uses a single variable is referred to as *univariate linear regression.*

Going through this exercise will introduce you to a number of foundational concepts, algorithms, terminology and mathematics that are used throughout machine learning and the remainder of this book, so make sure that you have a good understanding of the material in this chapter before moving on.

## Intuitive Linear Regression

The first prediction we’re going to make will be an intuitive one – there’s no math or algorithms involved at this point, just a bit of guesswork. Our predictions will be reasonable, but not perfect and we’ll improve on this shortly.

By the end of this section, you should understand each of the steps in order to make a simple prediction, and have an appreciation of where machine learning algorithms will replace our intuition.

**Problem Space: Predicting the price of a home in Richmond**

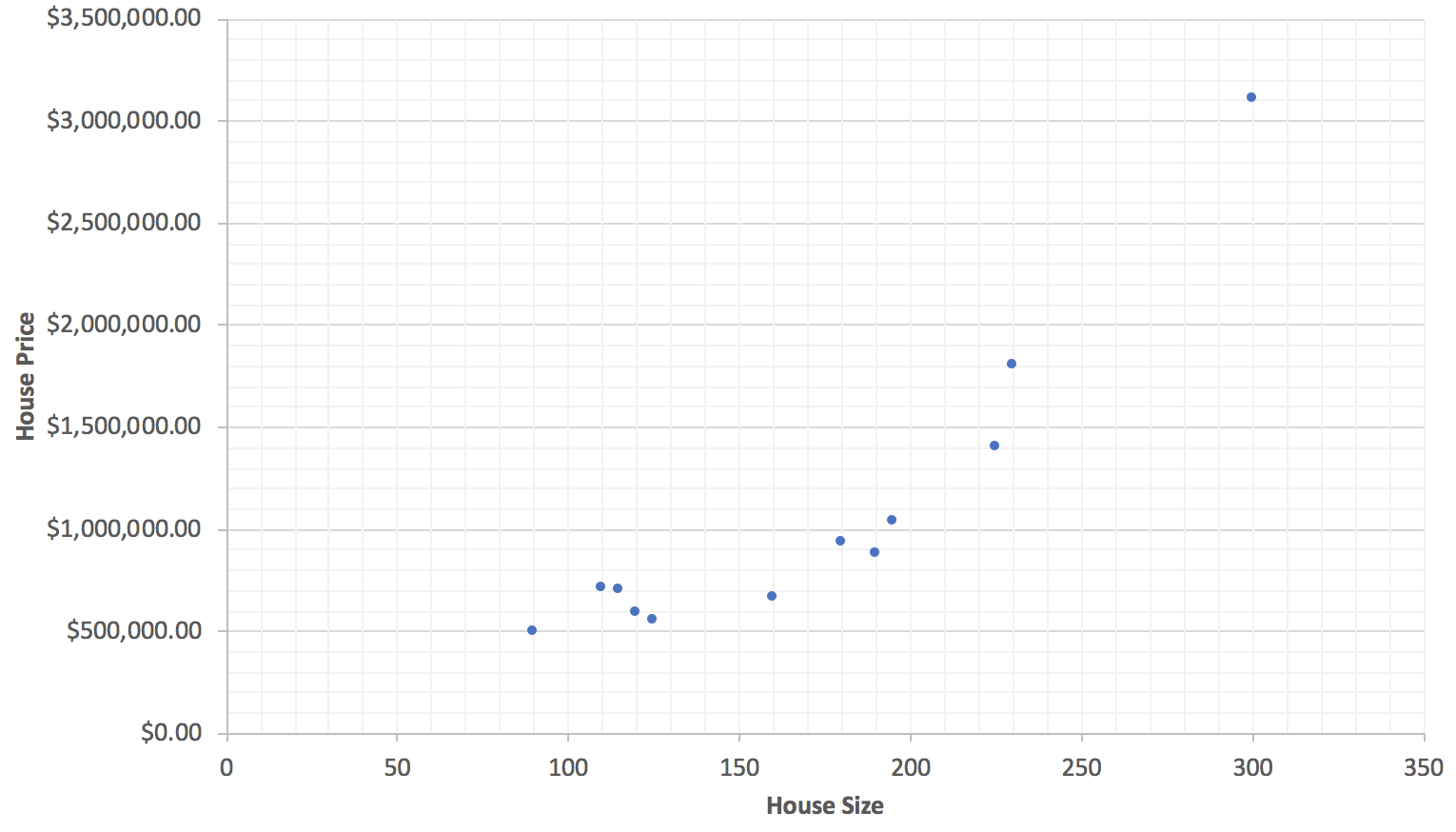
You’d like to buy a home for yourself and your family in the trendy suburb of Richmond, Melbourne. Space is a big consideration for you, so you’re looking for a home that’s at least 150 square meters (that’s around 1700 square feet) in size. You’ve done your homework and collected a few data points from recent house sales in the area. Based on this data, you want to figure out how much you might be expected to pay for a 150 square meter home in Richmond.

For the sake of this exercise, we are going to assume that the price of a house is simply dependent on the size of the house, and no other factors. We’ll also assume that the only data you were able to collect was the sale price of the house and size of the house, and no other data.

### Step 1: Plot the data

As a first step, we’ll plot all of the house sales data that you collected onto a graph. Figure 1 presents a graph of your house sales data, with the house size (in m2) on the **x axis** and the house price on the **y axis**.

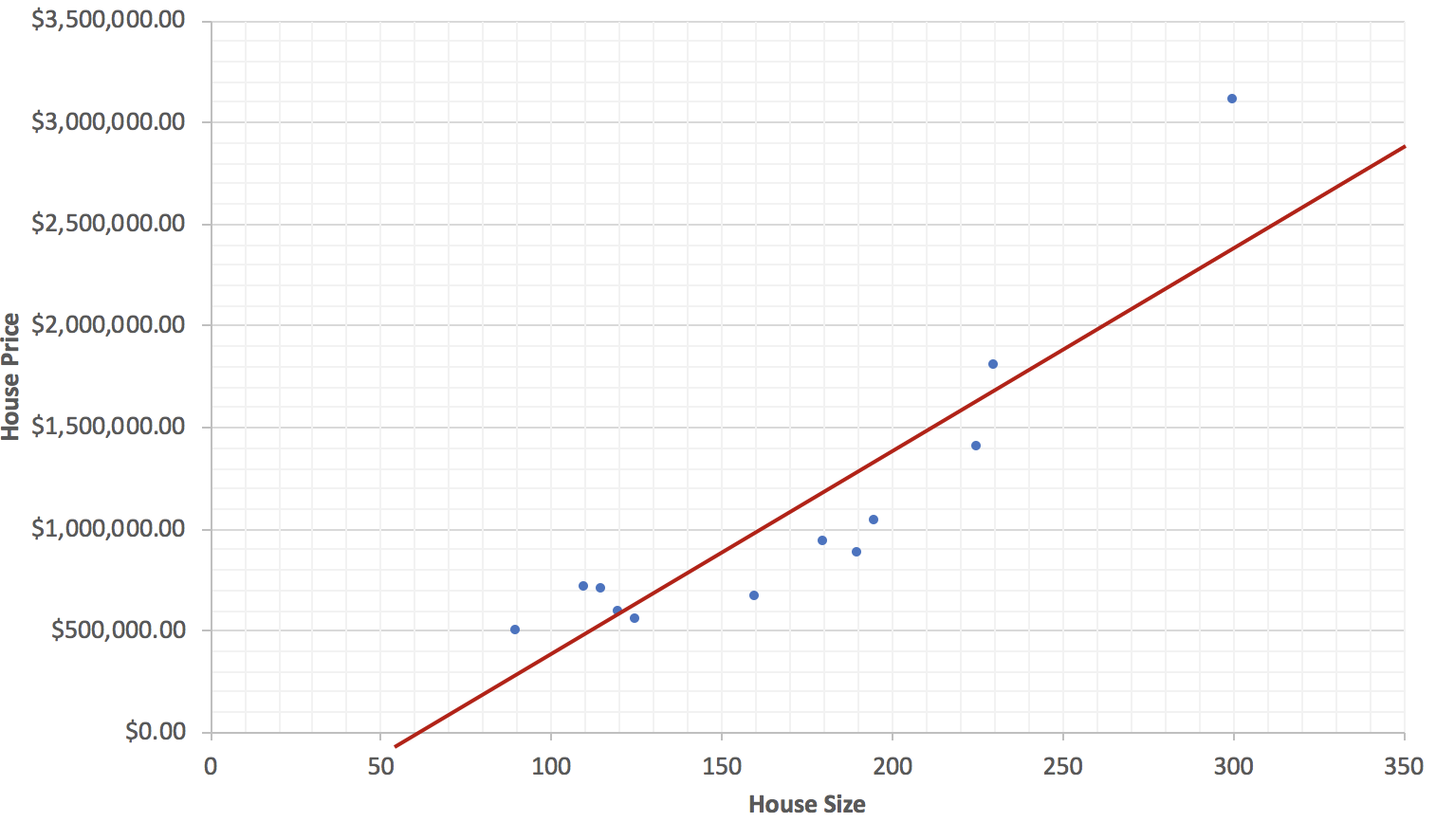
Figure 1: House price as a function of house size



### Step 2: Fit the data

With our house sales data plotted, we will draw a straight line that *fits* the data as best as possible so that we can begin making predictions. We’re going to use our best judgement here and draw a straight line that seems to fits the data reasonably well. Figure 2 displays our straight line overlaying the house sales data.

Figure 2: Using judgement to draw a line that best fits the housing data

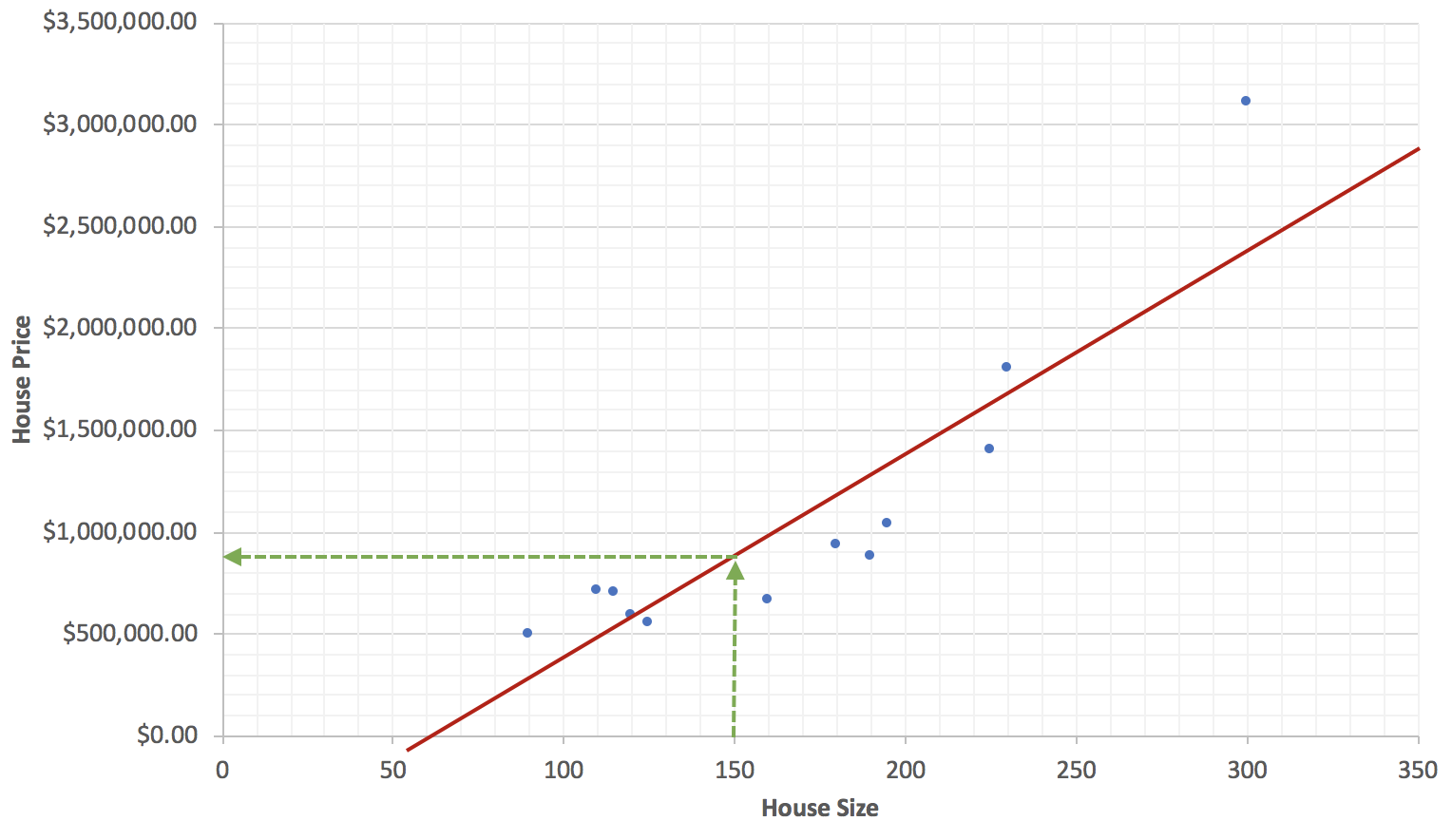


While this won’t a be perfect fit, it’s certainly within reason and at least appears to fit the data quite well. The better this line fits your data, the better your predictions will be.

### Step 3: Make predictions

To predict the price for a 150m2 house, draw a vertical line from the 150m2 value located on the **x axis** until it intersects with the line, then come across to the **y axis** to get the predicted price. Figure 3 below will provide you with a visual demonstration of our prediction.

Figure 3: Predicting the price of a 150m2 house



We can now predict that it will cost around $900,000 to buy a 150m2 house in Richmond based on the data we collected from previous home sales. As an exercise for the reader, use the figure above to predict the price of a 200m2 house and a 300m2 house.

This was our first prediction as well as an intuitive introduction to linear regression. Let’s summarize what we’ve just done to make our first prediction:

1. We collected data
2. *We plotted the data*
3. *We drew a straight line that best fits the data*
4. We used the line to make predictions for data we haven’t seen before.

Hopefully you found this to be a reasonably straightforward process and a common sense approach towards making a simple prediction from a set of data. Please note the italicized points 2 and 3 above; the goal for the remainder of this chapter will be to revise steps 2 and 3 to use an *automated, algorithmic* approach for prediction instead of a *manual, intuitive* one.

## The Hypothesis Function: Model your problem space

With our first intuitive prediction complete, it’s time to start formalizing our intuition so that we can begin making predictions algorithmically. In this section we introduce you to the concept of a hypothesis function, and define the hypothesis function that we will use in linear regression.

The hypothesis function describes *any* function that is used to fit data in order to make predictions. In other words, the hypothesis function is a function that acts as an effective model, or proxy for your data. It is the responsibility of the AI/ML designer to select the most appropriate hypothesis function to effectively model the available training data. Consequently, this will be one of the first activities an AI/ML designer will undertake when building machine learning algorithms.

Our housing data appeared to have a linear relationship, so we used a *linear function* to model this relationship. Problems spaces like population growth and infection rates may best be modelled with *exponential functions*; predicting yearly fluctuations in global temperature could potentially be modelled with *sinusoidal functions*; predicting earthquake behavior could be modelled with *logarithmic functions*.

Figure 4 below provides a few examples of functions that can be used as a hypothesis function to effectively fit the shape of your data and model your problem space.

Figure 4: Examples of functions at your disposal to model your problem space

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

### The Linear Regression Hypothesis Function

The linear regression algorithm uses a linear function as it’s hypothesis function, and is defined as:

h(x) = Θ0 + Θ1x

This symbol Θ is called *theta*. Note that the hypothesis function **h(x)** is just a slightly *rebranded* linear function; these are simply the conventions used in machine learning, so we’ll stick with them – just remember that’s it just a regular linear function.

With the linear regression hypothesis function defined, we can now define the objective of linear regression: *Given a training set, produce a hypothesis function h so that h(x) is a good predictor for y.*

More concretely, the objective of the linear regression algorithm is to *learn and discover* the most appropriate values forΘ0 and Θ1 so that the hypothesis function fits our training data as closely as possible, resulting in the most accurate predictions.

What do we mean by *good predictor* exactly? In the next section we provide a concrete definition of a good predictor, and we also introduce the cost function as a means of quantifying good predictors.

**Reader Tip: Linear Function Review**

We’ve included a review of linear functions in Appendix A. If it’s been a while since you’ve last used linear functions we recommend that you read Appendix A before you move on to the next section.

## The Cost Function: Get feedback on model performance

A good predictor is one that *most effectively minimizes the difference* between the predicted values **h(x)** and the observed values **y** for all values of **x** in a training set. A cost function quantifies this difference to estimate how models are performing.

The cost function is an incredibly important tool in machine learning, as it acts as the *feedback mechanism* for machine learning algorithms. The ability to receive and adapt to feedback is one of the foundational characteristics of machine learning algorithms.

The following cost function pseudo-algorithm is used to quantify the performance of a model (i.e. a hypothesis function):

1. For each instance in the training set, calculate the difference between the predicted value h(x) and the observed value y.
2. Sum up all of the differences.
3. Divide the total difference by the size of the training set.

The cost function outputs the average difference between all predictions and observations in a training set. The closer this value is to zero, the better a model is at making predictions. If the value is *equal* to zero, it means that a model *perfectly* predicts observations.

Now that you have an understanding of how a cost function works, we’ll derive the mathematical definition of the cost function step-by-step:

**Step 1:** For each instance in the training set, calculate the difference between the predicted value **h(x)** and the observed value **y**.

Where the superscript *(i)* refers to a specific training *instance* in the training set. In this step we’re simply subtracting the observed value **y** from the predicted value **h(x)** for a specific training instance *i*.

**Step 2:** Sum up all the differences.

Where *m* refers to the size of the training set. Please note that the summation for the training set is 0-indexed (i=0). You may see training sets 1-indexed (i=1) in some literature.

**Step 3:** Divide the total difference by the size of the training set:

What we’ve just derived in step 3 is a perfectly usable cost function. Although we could use the function as-is, we are going to make two small modifications to the function so that it’s a bit simpler to implement in practice. The next section describes the modifications and the benefits of making them.

### The Cost Function

This is the formal definition of the cost function, which may also be referred to as the squared error function or mean error function:

It’s important to note that the formal definition of the cost function shown above and the cost function derived in step 3 are functionally and materially *identical*. Using either of these functions will provide a consistent and correct outcome. The formal definition happens to be simpler to implement in software, and we like simple.

First, the error component (the difference between the predicted value and observed value) has been squared. Squaring always ensures that we have a positive number; for example 42 and -42 are both equal to 16. A programmatic implementation of the cost function only cares about the *magnitude* of error - getting rid of negatives removes a bit of complexity in dealing with both positive and negative numbers.

Second, the cost function is multiplied by ½ - this is a practical change to restore balance to the cost function. Since the error component has been squared, we needed to add a term to make sure the function remains consistent when derivatives of this function are taken. Since the derivative of x2 is 2x, so multiplying by 1/2 cancels out the effect of the exponent.

Lastly, we substitute the reference to our hypothesis function **h(xi)** with the linear regression hypothesis function that we introduced previously. This is the fully expanded cost function that we will use in the linear regression algorithm:

The cost function is conventionally defined as **J**, so J(Θ0, Θ1) is a function that will produce a performance measure for specific values of Θ0 and Θ1.

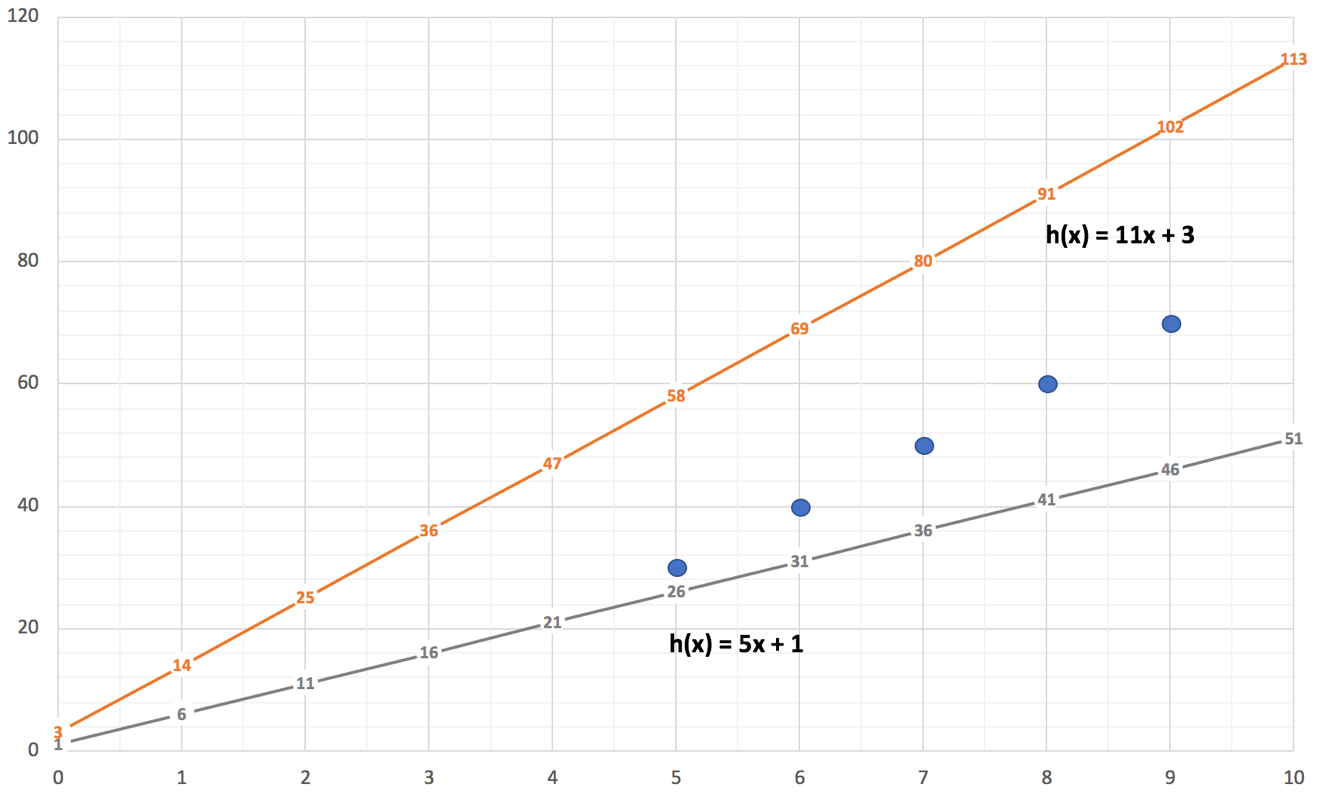
The linear regression algorithm will choose values for Θ0 and Θ1 so that J(Θ0, Θ1) produces the smallest possible output. By way of convention, we want our machine learning algorithm to *minimize* J(Θ0, Θ1).

Before we move on to minimizing J(Θ0, Θ1), let’s work through a concrete example with the cost function that we’ve just derived.

### The Cost Function in Practice

To get a better appreciation of the cost function, we’ll run through an exercise comparing the performance of two different linear regression hypothesis functions. In figure 5 below, I’ve plotted the function **h(x) = 3 + 11x**, the function **h(x) = 1 + 5x** and a training set consisting of five data points. The data points for our training set are as follows:

Figure 5: Comparing two linear regression hypothesis functions for predive performance



We can assert that the function h(x) = 1 + 5x is a better fit to the data than the function h(x) = 3 + 11x by visual inspection alone, but we’ll generate the cost for each of these hypothesis functions to get an objective measure.

Table 1 and Table 2 below show the work for calculating the cost functions. The final line item in each table (Cost) is calculated by taking the total difference and multiplying by 1/2m, or 1/10 since there are 5 instances in our training set.

Table 1: Calculating the cost for h(x) = 1 + 5x

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X | Predicted Value | Observed Value | Difference. | Squared Difference. |
| 5 | 26 | 30 | -4 | 16 |
| 6 | 31 | 40 | -9 | 81 |
| 7 | 36 | 50 | -14 | 196 |
| 8 | 41 | 60 | -19 | 361 |
| 9 | 46 | 70 | -24 | 576 |
| Sum of squared differences | | | | 1230 |
| Cost | | | | 123.0 |

Table 2: Calculating the cost for h(x) = 3 + 11x

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X | Predicted Value | Observed Value | Difference. | Squared Difference. |
| 5 | 58 | 30 | -28 | 784 |
| 6 | 69 | 40 | -29 | 841 |
| 7 | 80 | 50 | -30 | 900 |
| 8 | 91 | 60 | -31 | 961 |
| 9 | 102 | 70 | -32 | 1024 |
| Sum of squared differences | | | | 4510 |
| Cost | | | | 451.0 |

By going through this exercise, we have been able to show that J(1, 5) has a cost of **123.0** and that J(3,11) has a cost of **451.0**. We can new objectively assert that h(x) = 1 +5x is a better predictor for our observed data than h(x) = 3 + 11x.

Given a perfect predictor has a cost of 0, the data also suggests that there are more optimal values for Θ0 and Θ1. As an exercise for the reader, find values for Θ0 and Θ1 that would be a better predictor than (1, 5) and run through the exercise above to quantify how much better your function performs. Going through the exercise should demonstrate that even though the cost function might look a bit intimidating, it’s quite easy to work with.

We now have the means to define our model and get feedback on its performance. The final step is to discover the best performing model to act as the predictor for our data. In the next section we introduce the gradient descent algorithm, which will provide us with the means to minimize J(Θ0, Θ1) efficiently.

## The Gradient Descent Algorithm: Use feedback to find the best predictor

The gradient descent algorithm is a general purpose algorithm that has a number of practical applications in machine learning. It has broad applicability in machine learning as the gradient descent algorithm provides an efficient means to *minimize functions*. The cost function has provided us with the ability to *quantify the effectiveness of a model*, and with that feedback the gradient descent algorithm will find the *minimal cost for our model* – put simply, gradient descent will find the values of Θ0 and Θ1 that will act as the *best predictor* for our model.

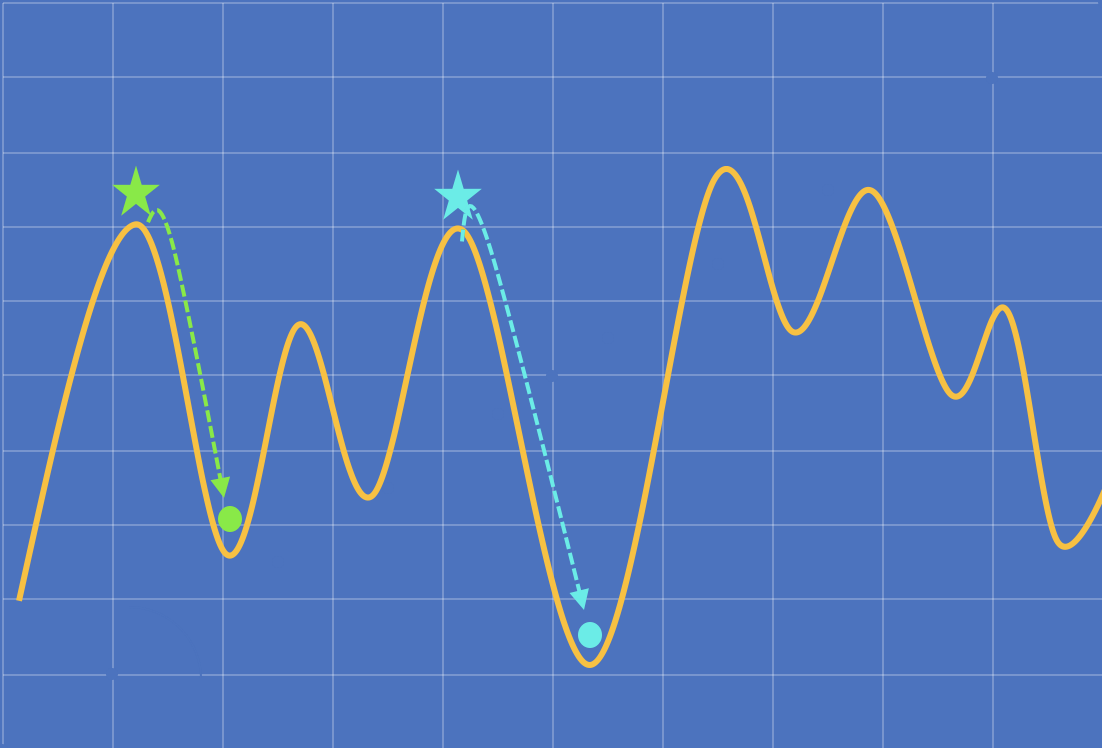
By the end of this section you should have a good understanding of how the gradient descent algorithm works, and appreciate how the algorithm solves minimization problems.

### Intuitive Gradient Descent

Before we define the gradient descent algorithm, let’s first get an intuitive understanding of how the gradient descent algorithm works and appreciate its limitations. The gradient descent algorithm is effective at finding *local minima* - the smallest values of a function, in and around a local area of a function.

Figure 6 below presents a function with a couple examples of the gradient descent algorithm in action:

Figure 6: The gradient descent algorithm - discovering minima by intuition



Imagine this function is a mountain, and you’re standing where the green star is. A storm has rolled in, and visibility is poor – you can only see a few steps in front of you. You need to head to the base of the mountain as soon as you can to ride out the storm in safety.

How will you get down? The approach described below acts as a good analogy for the behavior of the gradient descent algorithm.

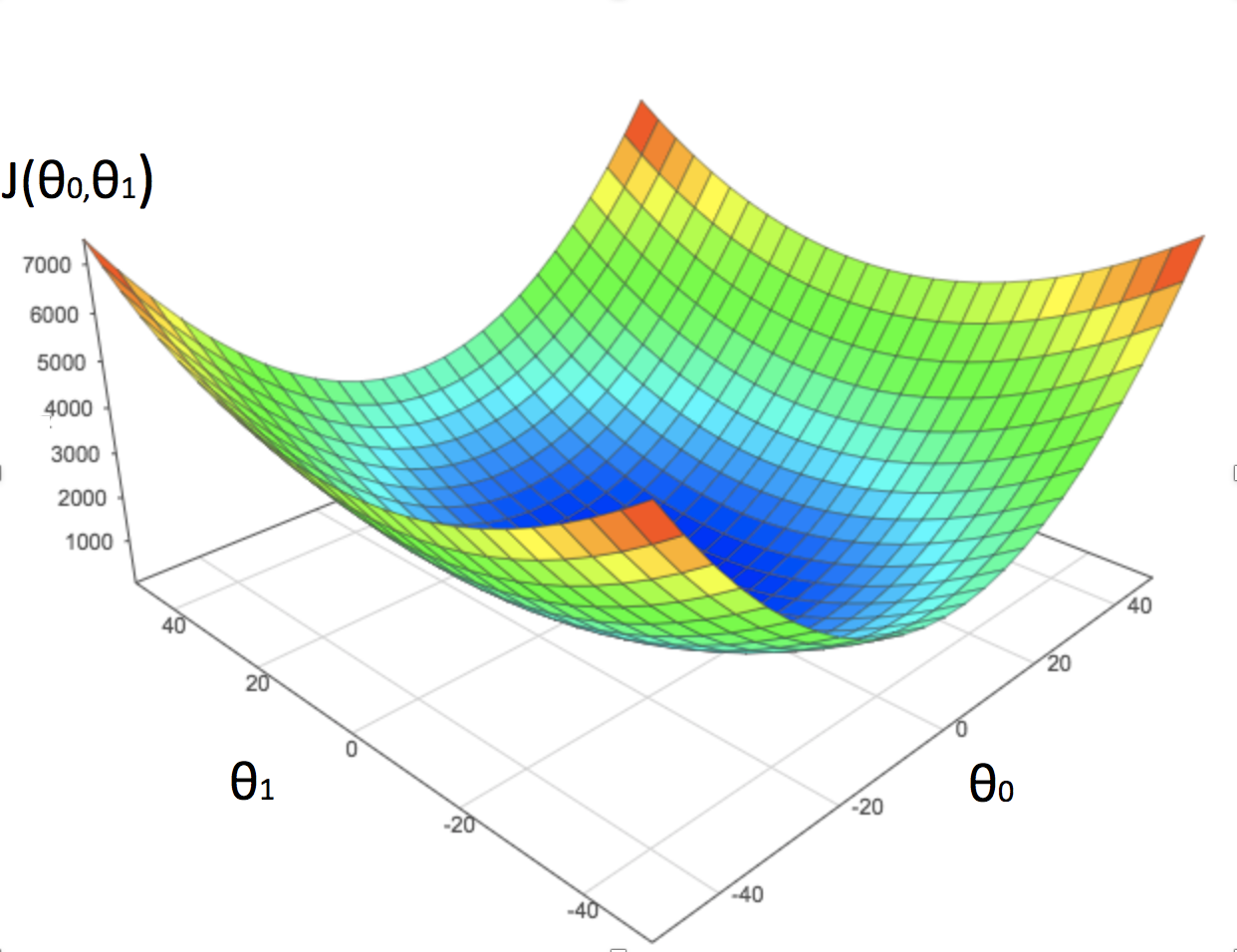
1. You look around, and find a new location that is lower than your current location.
2. You take a few steps in the direction of the new location and stop.
3. From the new location, you repeat steps 1 and 2 until you can no longer find a location that is lower; in this case you’ve reached (as far as you know) the base of the mountain – the green circle in the figure.

One thing you may have immediately noticed going through this exercise is that we did not reach the base of the mountain when we started from the green star; this is an example of a local minimum. Had we started the same exercise from the blue star, we would have reached the base of the mountain.

The conclusion we can draw from this exercise is that the gradient descent algorithm is not well suited for minimizing functions that have more than one local minima, since the result you receive would be highly dependent on your initial starting position – you would never know if you truly minimized the function. The gradient descent algorithm should only be used to minimize functions that have a single, global minima.

The good news is that the linear regression cost function does have a single global minima, so it’s well suited to be minimized by the gradient descent algorithm. We aren’t going to explain why linear regression cost functions have a single global minima, but we are going to present a 3D visualization of the cost function as a function of Θ0 and Θ1 so you can appreciate the actual terrain our gradient descent algorithm will traverse.

Figure 7: J(Θ0, Θ1) as a function of Θ0  and Θ1



This graph presents cost J(Θ0, Θ1) as a function of both Θ0 and Θ1, and represents the actual topology that our gradient descent algorithm will navigate - it’s pretty cool. In fact, all linear regression cost functions will end up with this convex shape, which is why we can use the gradient descent algorithm to minimize linear regression cost functions.

High areas shown in red represent high costs, and are consequently *bad predictors.* Low areas in shown in blue represent low costs, and are consequently *good predictors*. The gradient descent algorithm will find the lowest point on this graph, which also means that it will find the *best predictor*.

### The Gradient Descent Algorithm

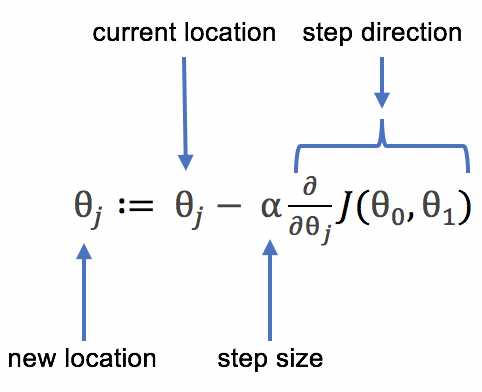
While the pseudo-algorithm for walking down a mountain was quite straightforward, transcribing this concept algorithmically is a little less straightforward; we’ll break down the gradient descent algorithm and show that it’s simply a formal representation of our intuition. The definition of the gradient descent algorithm for minimizing a cost function is as follows:

Repeat until convergence, for j = 0 and j = 1 {

}

There are a few new terms in the gradient descent algorithm to get your head around. Before we define these new terms, Figure 8 presents the gradient descent algorithm with an overlay of the concepts and intuition in our mountain analogy; this should help provide a bit of context around how the algorithm works in practice.

Figure 8: The gradient descent algorithm with an overlay of the mountain analogy



Let’s review some of the new terms introduced in the gradient descent algorithm:

|  |  |
| --- | --- |
|  | The expression **:=**  means *assignment*. The assignment ensures that we don’t lose our location as the algorithm moves towards the right solution. |
| α | This is the term *alpha*, also known as the learning rate and defines the size of the step you want to take. We will discuss the learning rate in detail shortly. |
|  | This term represents the *derivative* of the cost function. There’s a bit more to unpack here, and we’ll provide more insight about what this term does shortly when we review convergence. |

The algorithm refers to repeating until convergence, *for j = 0 and j = 1*. What does the statement *for j = 0 and j = 1* mean?

Put simply, when you take a step you’re moving some distance along the x axis (Θ0) and some distance along the y axis (Θ1). The gradient descent algorithm needs a practical means to capture both the x direction and y direction for every step so that the algorithm knows exactly where it is, and where it needs to move. We replace the original definition of the gradient descent algorithm with:

Repeat until convergence {

}

Implementers, treat this as an atomic operation! Do not compute the value of Θ0  and pass it into the calculation for Θ1. Both of these calculations need to use the same value of Θ0 and Θ1 as both functions work to represent a single, atomic, logical step.

The gradient descent algorithm that we’ve presented so far minimizes a *generic* cost function J(Θ0, Θ1). We will now substitute the generic cost function term J(Θ0, Θ1) with the actual linear regression cost function that we defined in the previous section.

Repeat until convergence {

}

The very last step requires us to take partial derivatives of these functions. Calculating the partial derivative of these two functions requires some knowledge of multivariate calculus. While you are more than welcome to work through the derivations yourself, we’ve worked through the derivations for you, with the final functions presented below. It may have taken a while to get here, but there you have it - this is our first machine learning algorithm!

Repeat until convergence {

}

To close out this chapter, we will provide some insights on convergence in gradient descent and learn how to manage the learning rate . For those readers with a programming background, these will be the last two pieces of information that you’ll need to implement this algorithm in software.

### Convergence in Gradient Descent

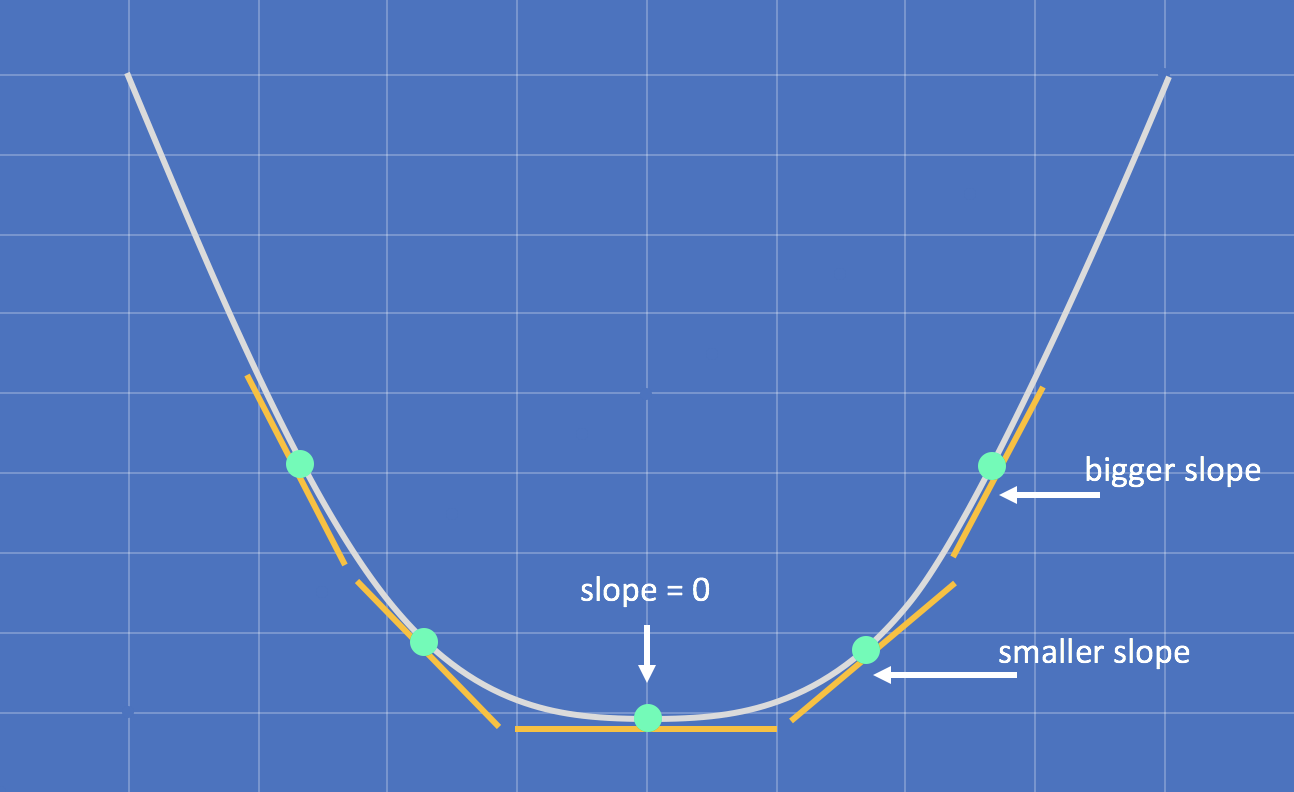
In the definition of the gradient descent algorithm, we start with the statement *repeat until convergence*. In this section, we’ll define what convergence means and provide you with some insight into how the derivative term in the gradient descent algorithm helps converge on an optimum solution. This is the term that we briefly touched on in the previous section:

The derivative of the cost function produces the *slope* of the cost function at points . Using our mountain analogy, the slope gives us information about how *steep* we are. High up on the mountain it’s going to be pretty steep; near the base of the mountain it will be nearly flat; at the *very* base of the mountain, it’s going to be *absolutely* flat.

Since the goal of the gradient descent algorithm is to find the base of the mountain, the slope gives us an indication of how close we are to finding the optimal solution. The lower the slope, the closer we are to finding (or converging on) the optimal solution.

Figure 9 below provides a visual of varying slopes as we converge on an optimal solution. Please note that that Figure 9 represents a *cross-section* of the 3D function that we presented in Figure 7 earlier (it’s a bit easier to visualize the 2D cross section than the 3D graph itself).

Figure 9: The slope approaches 0 as we converge towards the local minima.



As we approach the global minima, the derivative of the cost function *approaches* 0; once we reach the global minima, the derivative of the cost function is *equal* to 0. A very useful characteristic in the gradient descent algorithm emerges - take one of our update operations in the gradient descent algorithm as an example:

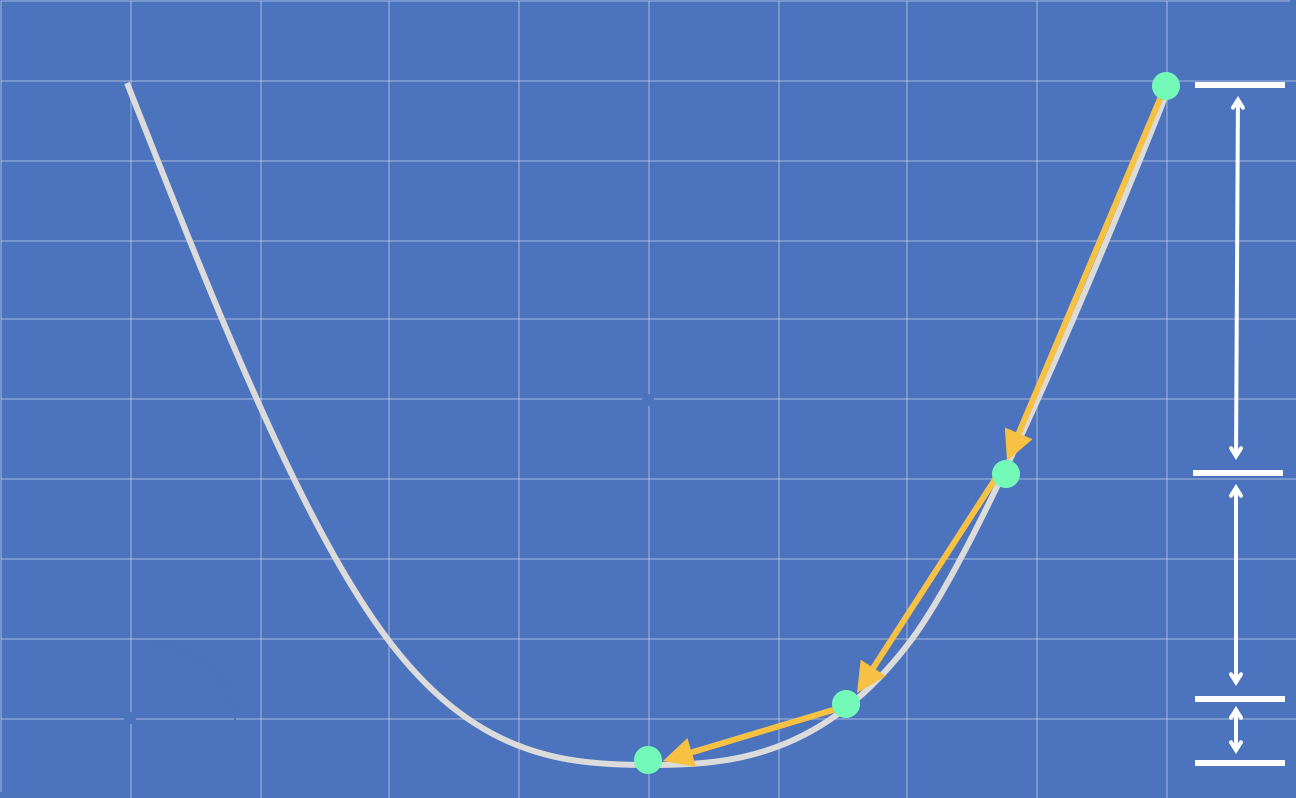
Once we’ve reached the global minima, the derivative of the cost function is equal to zero, and we end up with:

Multiplying our learning rate times 0 cancels out this term entirely, and we are left with:

The key insight here is that when the gradient descent algorithm reaches the global minima, the values will become *immutable*; they will not change with further iterations of the gradient descent algorithm. This case should be detected by algorithm implementers, and defines convergence.

In practice, you may not necessarily want to wait for this specific condition; as the gradient descent algorithm approaches the global minima, it will take smaller and smaller steps since the derivative term will get closer and closer to 0 after every iteration. The algorithm will naturally take smaller steps as it approaches the global minima, and it may take a large number of iterations to meet the exact definition of convergence. Figure 10 provides a simple visual of the deltas as the algorithm converges towards the global minima:

Figure 10: The delta between steps decreases as you approach convergence



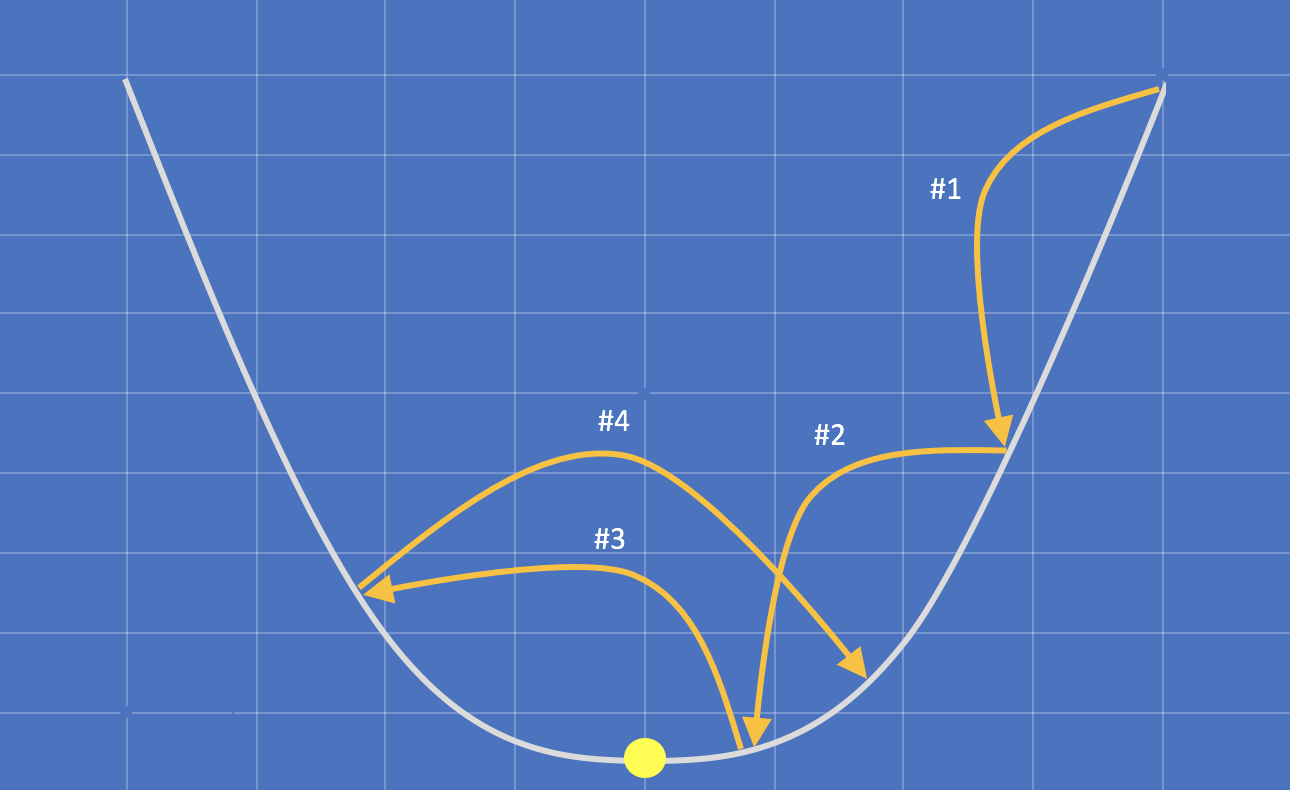
It’s a tradeoff between efficiency and accuracy, and AI/ML designers need to find a pragmatic balance given their specific goals and objectives.

### The Linear Regression Learning Rate

From an implementation perspective, he learning rate *alpha* is the only adjustable parameter in the gradient descent algorithm, and choosing an appropriate value for alpha ensures that the algorithm finds an optimal solution efficiently and accurately. If the value of alpha is too large, the algorithm may not find the most optimal solution; if the value of alpha is too small, the algorithm may a long time to find the most optimal solution.

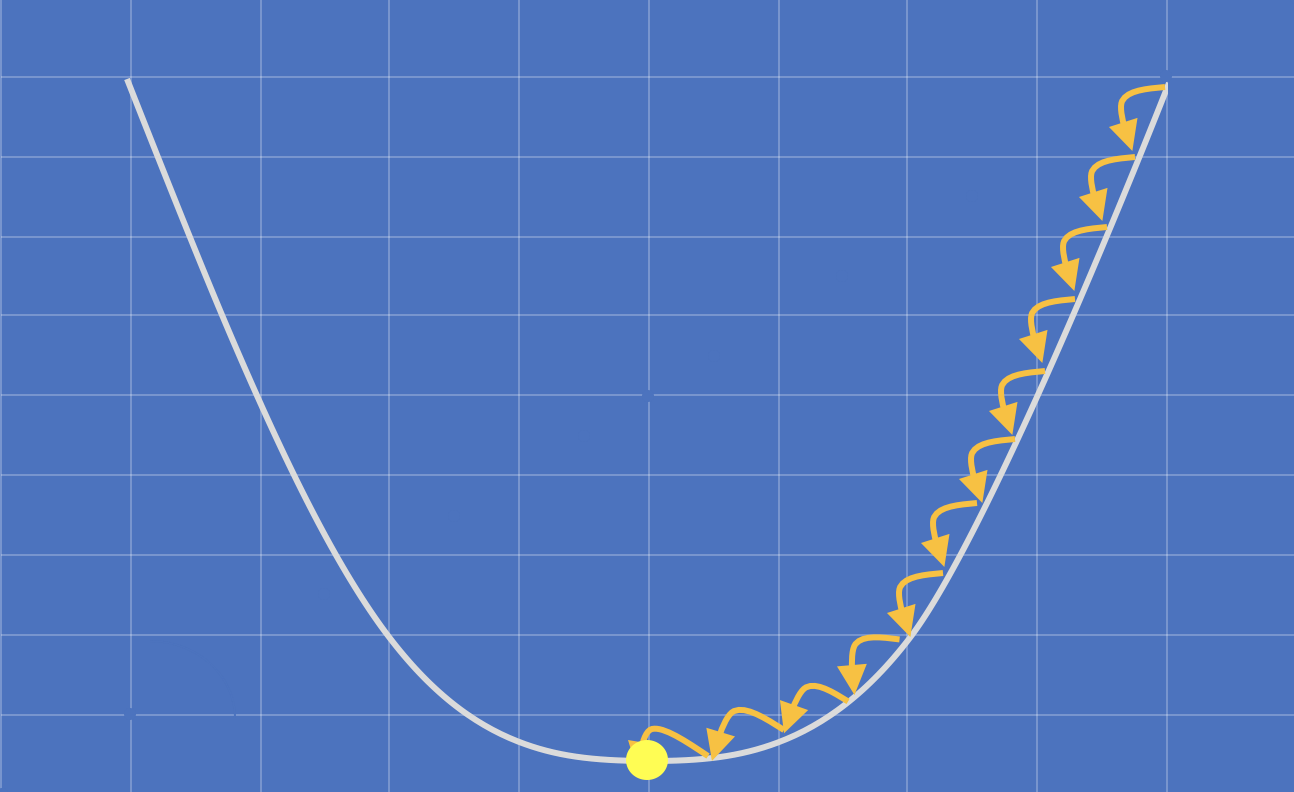
The following figures present the behavior of the gradient descent algorithm when the value of alpha is set too high or too low.

Figure 11: Linear regression behavior with an aggressive learning rate.



The larger the value of alpha, the larger the steps the gradient descent algorithm will take between each iteration. In iteration #1 and iteration #2, the algorithm takes big steps and gets closer to the minima (the yellow circle). In iteration #3 the algorithm overshoots the minima and iteration #4 begins to diverge from the right solution. When alpha is set at a high value we risk the possibility of never converging on the minima.

Figure 12: Linear regression behavior with a conservative learning rate



The smaller the value of alpha, the smaller the steps the gradient descent algorithm will take between each iteration. If the learning rate is too small, the gradient descent algorithm will make a larger number of smaller steps, which reduces the efficiency of the algorithm and increases the time it will take to find an optimal solution.

As an AI/ML designer it will be your responsibility to choose an ideal value for alpha that balances efficiency and accuracy. We recommend that you start conservatively with a smaller value of alpha, and incrementally increase the value to improve performance.

## Linear Regression Summary

This was a big chapter! This may seem like quite a bit of work to make a prediction for the price of a house based solely on its size. However, the concepts and methods that we’ve introduced in this chapter lay the groundwork for much more powerful and sophisticated prediction algorithms in the upcoming chapters.

If you understand the content in this chapter well, you will have no problem working through the next two chapters; in fact, we will only be making a few small modifications to the functions and algorithms that what we’ve learned so far. Personally, this is one of the things that I’ve found most exciting; with only a few incremental updates, we will go from predictions that we can make intuitively to predictions so sophisticated they would be impossible to reason about without the tools we’ve learned in this chapter.

We would encourage you to re-read this chapter before moving on if you’re not entirely comfortable with all of the content quite yet. Here are the three key takeaways should be confident with:

The **Hypothesis Function** is the tool that we use to model our data. The hypothesis function can take many forms (linear functions, exponential functions, logarithmic functions), and it’s the responsibility of the AI/ML designer to select the hypothesis function that most closely models the problem space. In this chapter, we introduced the linear function as a model our housing data.

h(x) = Θ0 + Θ1x

The **Cost Function** is a tool that allows us to quantify how well our model is performing. This is the critical piece of feedback that we need in order to effectively tune so that our model makes the best predictions possible.

The **Gradient Descent Algorithm** is the logic behind the linear regression algorithm. The gradient descent algorithm makes use of the quantitative feedback from the cost function to choose the best values for , and therefore the best predictor for our data.

Repeat until convergence {

}

Once the gradient descent algorithm has converged, you can go back to your hypothesis function and **start making predictions**. Simply substitute in the hypothesis function with the values generated by the gradient descent algorithm and you will have a perfectly tuned model, ready to make the best predictions possible.

# Linear Regression with Multiple Features

In the previous chapter you learned how to develop a simple linear regression algorithm to predict the price of a house using a single feature, the size of the house. Predictions based on a single feature aren’t particularly useful in practice, but we needed to start simple.

In this chapter, we will develop a more sophisticated linear regression algorithm that can predict the price of a house using *multiple* features. Linear regression that uses multiple variables is referred to as *multivariate linear regression.* Predictions using multiple features will be broadly applicable and more useful for AI/ML designers.

Our new training set will consist of four features - the house size, the number of bedrooms, the number of bathrooms, and the age of the house.

Table 3: Training data set with multiple features for house price prediction.

| **House Size (m2)** | **Bedrooms** | **Bathrooms** | **Age (Years)** | **House Price ($)** |
| --- | --- | --- | --- | --- |
| 124 | 3 | 1 | 3 | 223000 |
| 199 | 3 | 2 | 11 | 430000 |
| 334 | 5 | 3 | 22 | 900000 |
| 112 | 4 | 1 | 44 | 300000 |

## The Hypothesis Function

When we predicted the price of a house using a single feature, we used the following linear function as our hypothesis function:

The first thing we need to do is extend this hypothesis function to incorporate multiple features. This can be achieved by simply appending new terms to the function - one new term for each additional feature you want include in your model. For example, a hypothesis function supporting 4 features is defined as:

This function also introduces a new variable subscript notation, highlighted in blue above. The univariate hypothesis function only needed a single variable **x** to keep track of a single feature; x *always* referred to the house size, it couldn’t possibly have referred to anything else.

Our updated housing data set has 4 features, which introduces a problem - the variable **x** could refer to the house size, house age, bedroom count or bathroom count. With multiple features, the variable **x** on its own becomes *ambiguous*, so the variable subscript notation was introduced conventionally as a means to keep track of multiple features.

With a larger number of terms in this function, you might notice a bit of inconsistency. Every Θ parameter is paired up with a variable except for the first one - Θ0 looks a little out of place.

To make Θ0 consistent with the rest of the terms, we introduce a convention where we define a new variable **x0** which will always have the value **1**. The hypothesis function is now defined as:

The first term in the function is now consistent with the remaining terms. By defining x0 as 1 we ensure that the value of the first term remains unchanged. Introducing this convention is purely for convenience - the consistency will help simplify the algorithm implementation.

We also introduce some convenient shorthand notation for the hypothesis function so that we don’t have to write out the terms for each feature. This is the definition of the generalized hypothesis function that we will use in multivariate linear regression:

The parameter **n** is used as a convention to describe the number of features.

## The Cost Function

When we predicted the price of a house using a single feature, we used the following cost function:

We do not need to make any material changes to the cost function – we simply substitute the term with the multivariate hypothesis function definition, and we get:

This definition is a bit long, so for convenience we will introduce additional shorthand notation by substituting all of the *enumerated* terms (i.e. all 0…n terms) with a single theta term.

This is the generalized definition of the cost function that we will use in multivariate linear regression:

## The Gradient Descent Algorithm

When we predicted the price of a house using a single feature, we used the fully derived gradient descent algorithm defined below. We need to make a couple of practical changes to the gradient descent algorithm to support multiple features, and we’ll also make a couple of additional modifications to make it a bit easier to work with as well.

First, the gradient descent algorithm references the univariate hypothesis function (highlighted in red below), and needs to be updated with the multivariate hypothesis function we just developed.

Repeat until convergence {

}

When we replace the univariate hypothesis function with the multivariate hypothesis function (highlighted in blue below), we get the following definition:

Repeat until convergence {

}

Finally, we need to update the term (highlighted in red above) as this term also becomes ambiguous when we use multiple features. To fix this, we’ll use the same variable subscript notation that we introduced in our multivariate hypothesis function. Once we update the term to use the subscript notation (highlighted in blue below) we get the following definition:

Repeat until convergence {

}

Those were the only two changes that were absolutely required in order to support multiple features in the gradient descent algorithm. We have an opportunity to make our new gradient descent algorithm even more convenient to implement, so let’s continue with our derivation.

In the multivariate hypothesis function derivation we introduced a convention where we defined a variable x0 which will always had the value 1. We introduced this convention to make all terms in the hypothesis function *consistent*; let’s take the same approach with our gradient descent algorithm.

When we introduce the conventional x0 term to our update operation, and we end up with the following definition:

Repeat until convergence {

}

The newly introduced x0 term is highlighted in blue above. The update operation is now consistent with the update operation. Since x0 is conventionally defined to always equal, 1 the introduction of this term does not change the behavior or output from the operation.

Now that the update operations are consistent with each other, we have an opportunity to take a data-driven approach and collapse these update operations together. Before we do that, let’s take a quick look at what our current gradient descent algorithm would look like if we applied it to our new housing data set with 4 features.

Repeat until convergence {

🡪 constant, 1

🡪 house size

🡪 bedrooms

🡪 bathrooms

🡪 house age

}

Other than a quite a bit of typing there’s absolutely nothing wrong with this algorithm. To simplify, we can collapse all of these individual update operations into a single update operation by using the following definition of the gradient descent algorithm:

Repeat simultaneously until convergence, for j = 0 to j = n {

}

All explicit terms () are replaced with the parameterized terms and respectively. This is a data driven approach to implementing gradient descent which can find an optimal solution with an arbitrary number of features in your model.

Congratulations, you now have your second machine learning algorithm! Discuss the use of the term “simultaneously”.

## Feature Scaling

A practical challenge surfaces when we move from making simple predictions with a single feature to making more powerful predictions with multiple features. To illustrate our challenge, let’s look at a training set with two features - the size of the house and the number of bedrooms.

Table 4: Training set for feature scaling

| **House Size - m2** | **Bedrooms** | **House Price - $** |
| --- | --- | --- |
| 124 | 3 | 223000 |
| 199 | 3 | 430000 |
| 334 | 5 | 900000 |
| 112 | 4 | 300000 |

In this training set, house sizes range from **112** m2 to **334** m2, and the number of bedrooms range from **3** to **5**. The range and scale of the house size feature is significantly larger than the range and scale of the bedroom feature.

As a consequence, the gradient descent algorithm will have to make many more iterations to converge on the house size feature compared to the bedroom count feature, because we have a pretty big range to work through.

Since we have to compute the value of for *each of our features simultaneously*, we end up with a situation where we converge on the bedroom count feature quickly, but the gradient descent algorithm will need execute many more iterations to converge on the house size feature.

Imagine a scenario where you have a training set with 10 features, and only one feature had a large range. The gradient descent algorithm would spend an overwhelming amount of its time attempting to converge on that *one* feature. This isn’t a particularly efficient use of CPU cycles, and with sizeable data sets can significantly increase algorithm run time.

In this section, we introduce a technique to scale your data so that the gradient descent algorithm converges on all features as quickly as possible.

F*eature scaling* or *data normalization* is a method used to minimize and standardize the range of values for a specific feature. For a given feature x, feature scaling will attempt to normalize each *instance* of x so that all instances lie in a common range, usually within the range of -1 to 1.

-1 <= xi <= 1

This isn’t a hard requirement; it is perfectly acceptable for certain features to fall out of this range, but the closer you can scale your features to this range the more effectively the gradient descent algorithm will perform. It will be the responsibility of the AI/ML designer to select the most appropriate scaling range to ensure efficient algorithm performance given the size and scope of the training set.

Feature scaling is an incredibly effective method to normalize data, and you’ll see this method employed commonly in machine learning and data science. The technique that we will introduce for feature scaling is called *mean normalization*. Mean normalization is the most common technique used for feature scaling and is defined as:

Where  is the average value for feature *i* and  is the *range* for feature *i*, which is simply the difference between the largest instance value and the smallest instance value.

As a concrete example, let’s apply mean normalization to the **house size** feature in the training set we introduced at the beginning of this section in Table 4.

To figure out the range , we calculate the difference between the largest instance value (in this case 334 m2) and the smallest instance value (in this case 112 m2), and we end up with:

= 334 – 112 = 222

To figure out the average , we sum up all house sizes in our training set and divide by the training set size, and we end up with:

= (334 + 112 + 124 + 199) / 4 = 192.25

To scale the value of our first training instance , we substitute the values for , and into the mean normalization function, and we end up with:

= -0.31

As an exercise for the reader, use the mean normalization function to scale the remaining three training instances in the training set. You can use the following table to check your work:

Table 5: House sizes before and after feature scaling

| **House Size – m2 (Before Scaling)** | **House Size – m2 (After Scaling)** |
| --- | --- |
| 124 | -0.31 |
| 199 | -0.03 |
| 334 | 0.63 |
| 112 | -0.36 |

Once you scale a training set, machine learning algorithms will be trained on the *scaled training set*, not the *original training set*. As a consequence, your hypothesis function will be only able to make accurate predictions with *scaled inputs*. When you make predictions using an ML algorithm trained with a scaled training set, you will also need to remember to scale your inputs as well.

# Polynomial Regression

The previous two chapters have employed linear functions as models for our house sales data – a linear function is an appropriate model to use when a straight line fits your data best. We briefly introduced other types of functions that can be used as models for your data when a straight line isn’t the best fit.

In this chapter we introduce polynomial regression and polynomial functions. Polynomial functions are compelling models for *nonlinear data*; put simply, data that does not fit a straight line. For example, polynomial functions would be much better at modelling bell curves than a linear function would.

One of the most interesting things about this chapter is that we won’t be introducing new hypothesis functions, cost functions or gradient descent algorithms; if you need a bit of a break from all of the math, this is your chapter! Polynomial regression will leverage *absolutely all* of the machinery that we developed in multivariate linear regression.

Instead of developing algorithms for polynomial regression, we are going to learn about a technique to transform *data sets*. While the technique is quite simple, it will allow you to develop sophisticated predictive models for nonlinear data.

## 

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

## Polynomial Functions

Polynomial functions are versatile data modelling tools. For reference, figure 13 below presents three different degrees of polynomial functions, along with the definition of the function and a visual of the shape each function can notionally produce. While higher degree polynomials may be used for modelling, lower degree polynomials like the ones presented below are the most commonly used polynomial functions for machine learning.

Figure : An overview of lower degree polynomial functions and their graphs

|  |  |
| --- | --- |
| **Second Degree (Quadratic) Polynomial Functions**  f(x) = a + bx + cx2 |  |
| **Third Degree (Cubic) Polynomial Functions**  f(x) = a + bx + cx2 + dx3 |  |
| **Fourth Degree (Quartic) Polynomial Functions**  f(x) = a + bx + cx2 + dx3 + ex4 |  |

## Expressing Polynomial Functions in Data

As we mentioned at the beginning of this chapter, data transformation is the approach to implementing polynomial regression, not algorithm development. In this section we’ll show you how to transform your data in a way that allows a polynomial function to be *expressed as* a linear function in your data. Once the transformation is complete, you can reuse the multivariate linear regression algorithm we developed last chapter to train and make predictions on nonlinear data sets.

Let’s work through a concrete example to demonstrate how this technique works in practice. To start, we’ll define a simple house sales training set with three different features – number of bedrooms, number of bathrooms and house age.

Table : A simple training set for polynomial regression

| **Bedrooms** | **Bathrooms** | **Age** | **House Price** |
| --- | --- | --- | --- |
| 3 | 1 | 3 | 223000 |
| 3 | 2 | 11 | 430000 |
| 5 | 3 | 22 | 900000 |
| 4 | 1 | 44 | 300000 |

In the previous chapter we developed the multivariate linear regression hypothesis function; the same hypothesis function will be used in polynomial regression. As a quick refresher, a multivariate linear regression hypothesis function with three features is defined as:

Where is our conventional constant value 1, is the number of bedrooms, is number of bathrooms and is the age of the house. As a linear function, it produces a straight line on a graph.

Let’s assume that a straight line is *not* what we’re looking for; instead, we discover that our housing data would be best modelled with a third degree (cubic) polynomial function. By quickly referring to figure x above, we know that the hypothesis function for a cubic polynomial is:

So how do we go about expressing this in our data exactly? Let’s make a quick side-by-side comparison of the multivariate linear hypothesis function and the cubic polynomial hypothesis function.

// multivariate linear hypothesis function

// cubic polynomial hypothesis function

You’ll notice that the two functions are nearly identical, except for the squared term and cubed term in the polynomial function (highlighted in red). The key insight is this: a *linear function* that uses *squared and cubed data* will output the exact same results as a *polynomial function* that squares and cubes the *original data*. We’re really just shifting *when* and *where* data gets squared or cubed.

We’ll add two new features to our training set, **Bathrooms2**and **Age3**. The Bathrooms2 feature squares the existing bathroom values, and the Age3 feature cubes the existing age values.

Table 7: Training set with three features.

| **Bedrooms** | **Bathrooms2** | **Age3** | **House Price** |
| --- | --- | --- | --- |
| 3 | 1 | 27 | 223000 |
| 3 | 4 | 1331 | 430000 |
| 5 | 9 | 10648 | 900000 |
| 4 | 1 | 85184 | 300000 |

After data exponentiation, our multivariate linear regression hypothesis function is effectively:

This linear function is *functionally equivalent* to a polynomial function that uses the original training data; both functions will exhibit the same behavior and compute identical values for h(x). This data exponentiation technique can be used to map any degree polynomial function – we just happened to use a third degree polynomial function as an example.

Exponentiation will most likely increase the range of your training data, so please ensure that you use feature scaling to normalize your data *before* training the univariate linear regression algorithm.

# Logistic Regression

## Classification, Part 1: Binary Classification

To date we’ve been solving regression problems, where the values being predicted (like the price of a house) are entirely unconstrained. Classification problems are those where we want a machine learning algorithm to predict a specific, discrete result from a predefined set of data.

### Overview of Binary Classification

Binary classification is the simplest form of classification problem to solve for, but can help us answer some incredibly valuable questions. Examples of binary classification problems include spam filtering (is this email spam? yes/no) fraud detection (is this transaction legitimate? yes/no) and medicine (is this tumor malignant? yes/no).

For each of these example, the variable that we’re trying to predict will take on one of two distinct values, 0 or 1. More formally, we want to predict a value y which will take on the value 0 or 1, defined as:

y∈{0,1}

By convention, the value 0 is referred to as the “negative class”, and the value 1 is referred to as the “positive class”.

### Shortcomings with using Linear Regression

If we take what we’ve learned so far with linear regression and apply it to classification problems, we’ll learn that there are a few shortcomings with this approach.

Linear regression works well with continuous value predictions, but with classification problems we need an approach to produce discrete value predictions. With binary classification specifically, we only want to make predictions that produce 0 or 1 as outputs. In the examples below, we attempt to predict whether a tumor is malignant or not based on the size of the tumor, using linear regression.

In each example, we generate a hypothesis function that fits the data as best as possible. To determine whether a particular data point should represent a “yes” or a “no”, we take a midpoint at y = 0.5 (in other words, the halfway point between 0 and 1 on the y axis). This line extends until it intersects our hypothesis function, and then we make a simple decision - anything to the right of the intersection will be grouped into the “yes” group, and anything to the left of the intersection will be grouped into the “no” group.

In our very first graph, linear regression works out quite well, and it has correctly grouped the malignant and non-malignant tumors. The problem lies with the remaining two graphs, where we have more data points. The data is intuitively consistent with what we might expect (larger tumors tend to be more malignant), but the linear regression algorithm starts “pulling down” to fit the data points, and as a result starts predicting malignant tumors as non-malignant.

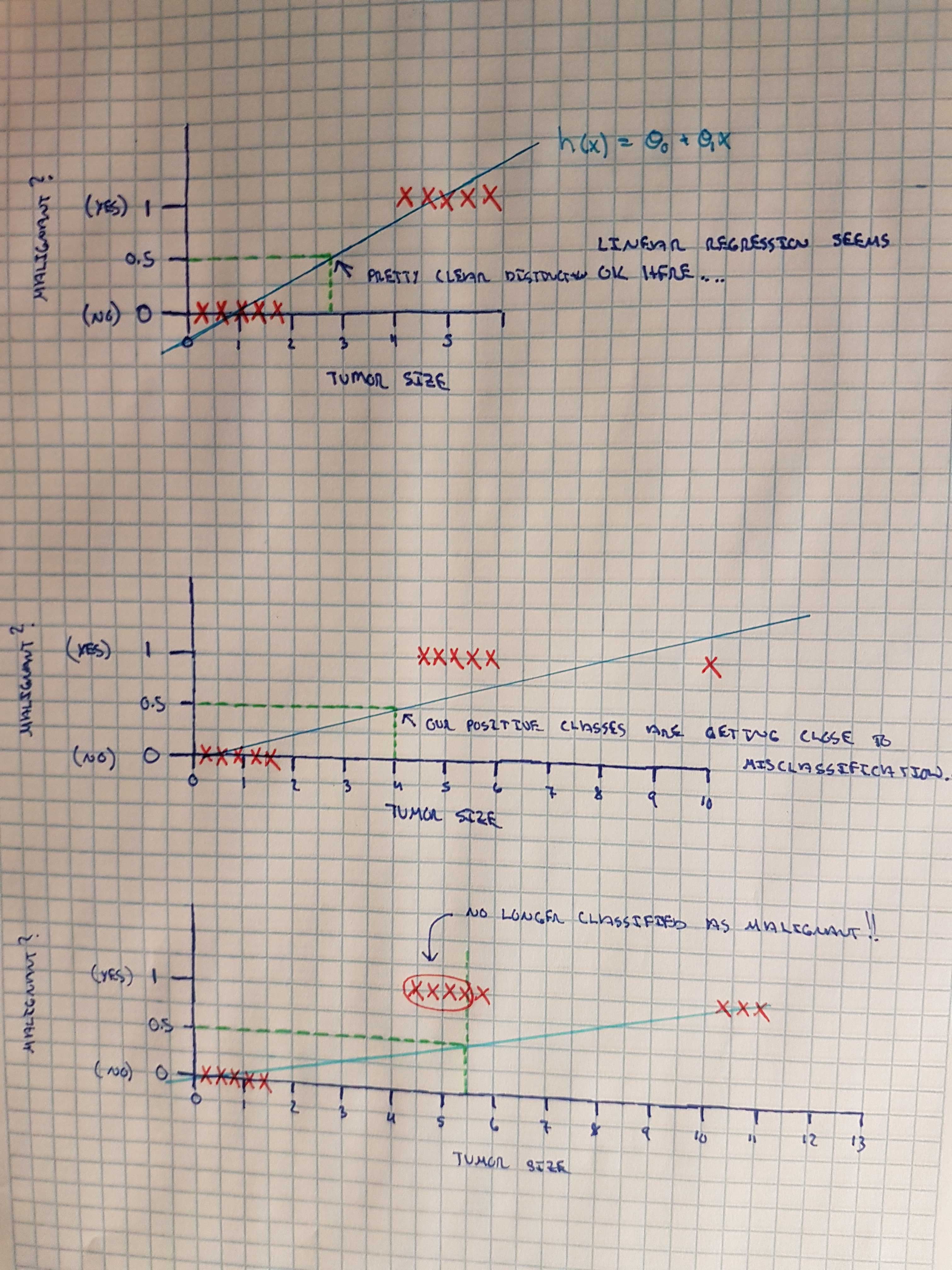


Figure 14: Shortcomings with linear regression

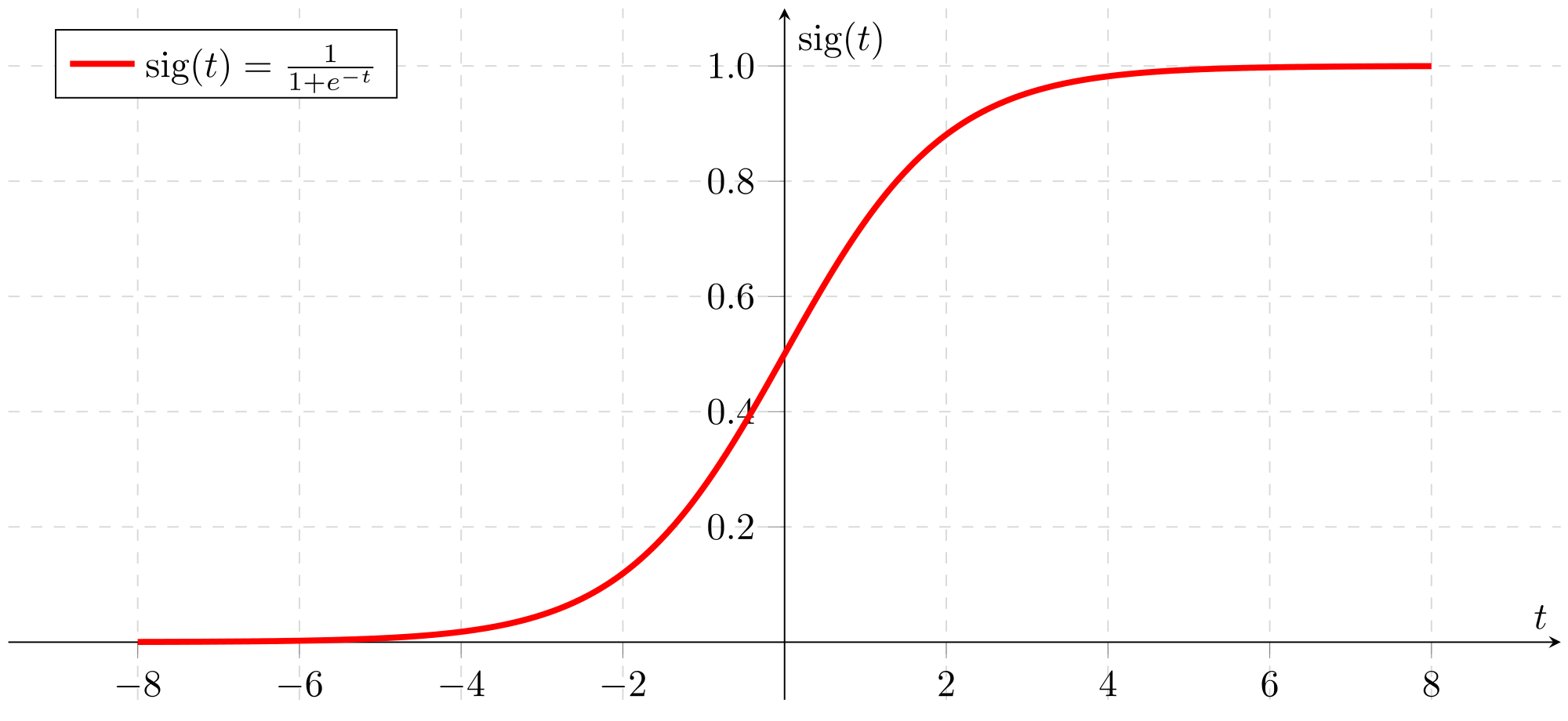
Linear functions are, well, too linear. We need a function that can shortly and sharply “cleave” our data points to cleanly place them into one of two camps.

The other practical problem that we run into is that linear regression functions (as we’ve seen before) generate predictive values y that can be less than 0 and greater than one. We can make use of feature scaling to reduce the problem, but it won’t solve the problem entirely. Ideally, we want a function that will guarantee predictive values are always bound between zero and one.

### The Sigmoid Function

The Sigmoid function (also referred to as the Logistic Regression function) is a function that neatly satisfies our requirements for classification. Don’t get confused about the terminology here - although this function is called the Logistic Regression function, it is in fact used for classification problems.

Figure 15: The Sigmoid Function



Firstly, the Sigmoid function maps any real number along the x axis to the [0, 1] interval we require for classification. For any input value x, the Sigmoid function guarantees that we will always be in our “yes”/”no” boundary. Secondly, it has a sharp transition between the [0, 1] boundary, which will help us produce a well-defined decision boundary (i.e. the point where we decide what qualifies as a yes, and what qualifies as a no).

### The Hypothesis Function

We need to “hook” our hypothesis function into the Logistic Regression function, here’s how to do it.

Our existing hypothesis function is defined as:  
h(x) = Θ0x0 + Θ1x1 + Θ2x2 + … + Θnxn

First, we temporarily assign our existing hypothesis function to a variable z, so we have:  
z = Θ0x0 + Θ1x1 + Θ2x2 + … + Θnxn

The Logistic Regression function is defined as:

(z)=11+e−z

We can formally define our new classification hypothesis function as:

h(x) = g(z)

When we substitute z with our definition above, we get:

h(x) = g(Θ0x0 + Θ1x1 + Θ2x2 + … + Θnxn)

When we expand the function g, our fully implementable hypothesis function is:

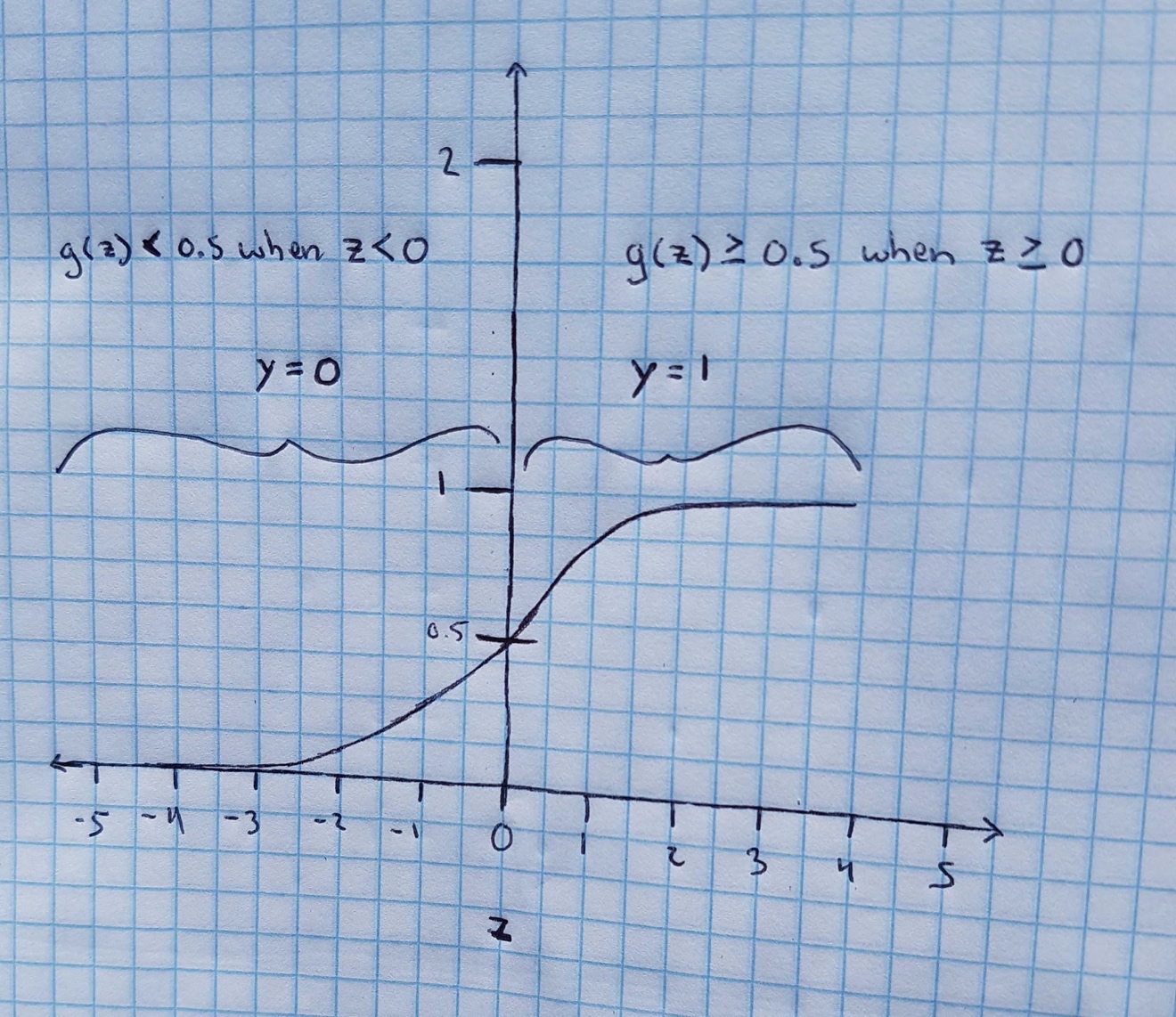
h(x)=11+e−(θ0x0+θ1X1+...+θnXn)

It’s also important to note the behavioral difference between the Sigmoid function and our linear regression hypothesis function. For any input value x, the Sigmoid function will return a probability of that y = 1 for the input value.

For example, if we found h(x) = 0.55, the Sigmoid function is telling us that h(x) as a 55% chance of being 1, and therefore a 45% chance of being 0.

### The Decision Boundary

The goal of our hypothesis function is to predict a discrete value, a 1 or a 0. In this section, we’ll understand how the hypothesis function will make these predictions with the use of a decision boundary. We’ll use the following graph to get a better sense of what the logistic regression hypothesis function is computing.



The image above shows our hypothesis function (Logistic Regression or Sigmoid function), with the function asymptoting at 0 and 1. Since our predictions can only result in one of two possible values, the first question we need to ask ourselves is when should a value of h(x) fall into the “0” camp, and when should a value of h(x) fall into the “1” camp.

We now know that if h(x) = 0.55, the hypothesis function is telling us that h(x) as a 55% chance of being 1, and therefore a 45% chance of being 0. Let’s define our first decision boundary as:

when h(x) >= 0.5, the predicted value (y) should be equal to 1.   
when h(x) < 0.5, the predicted value (y) should be equal to 0.

This is a pretty sensible definition - the arbitrary component here is that when h(x) is exactly equal to 0.5, the predicted value could really go in either camp. As a convention, we’ll make the predicted value 1. If you really wanted to, you could make the predicted value 0 - there’s no right or wrong answer.

We can start reasoning about how the algorithm will work. If you look at the value z = 0 on the graph, you’ll notice that the function intersects at y = 0.5. The value of y increases (up to 1) as the values of z get bigger.

We can generalize and say that g(z) >= 0.5 when z >= 0. Likewise, we can say that g(z) < 0.5 when z < 0.

Looking at our graph, we know that g(z) >= 0.5 when z >= 0. That’s the right hand side of the graph. Given our hypothesis function h(x) = g(z), we can also say that

h(x) >= 0.5 when z >= 0, and h(x) < 0.5 when z < 0

Of course, 0.5 is our decision boundary for deciding when a predicted value should go into the “1” camp or “0” camp, so we can also say that

h(x) should produce a predicted value of 1 when z >= 0,  
h(x) should produce a predicted value of 0 when z < 0

We also know what z is, so we can more formally say

h(x) should produce a predicted value of 1 when Θ0x0 + Θ1x1 + Θ2x2 + … + Θnxn >= 0 h(x) should produce a predicted value of 0 when Θ0x0 + Θ1x1 + Θ2x2 + … + Θnxn < 0

### The Decision Boundary in Practice

# Support Vector Machines

Placeholder for chapter content.

# K-Means Clustering

Placeholder for chapter content.

# Principal Component Analysis

Placeholder for chapter content.

# Neural Networks

Placeholder for chapter content.

# Anomaly Detection

Placeholder for chapter content.

# Recommender Systems

Placeholder for chapter content.

# Applying Machine Learning in Practice

Placeholder for chapter content.

# Applying Machine Learning at Scale

Placeholder for chapter content.

# Appendix A: Linear Functions Review

This section provides an overview of linear functions and includes a practical example that you can work through. By the end of this section you should understand what a linear function is and have the ability to graph a linear function.

A *linear function* is a simple mathematical function whose graph is a straight line. You would have certainly studied linear functions in high school, but in case it’s been a while this is a brief refresher on linear functions, which take the form:

y = ax + b

The value **a** is referred to as the slope, **b** is the intercept, **x** is the independent variable and **y** is the dependent variable.

Let’s run through a concrete example and graph a linear function where a = 5 and b = 2, so our linear function is:

y = f(x) = 5x + 2

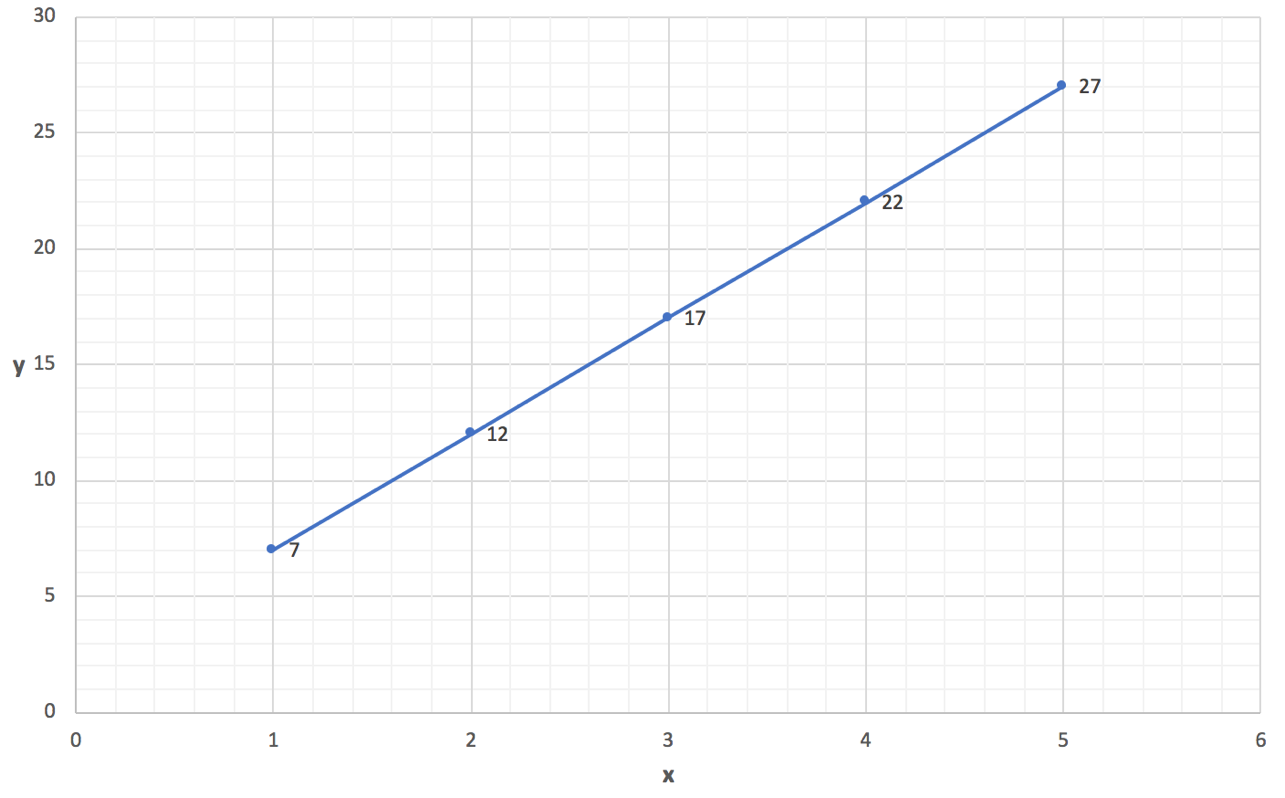
Table 1 presented below shows the calculation of the dependent variable y for various values of x, ranging from x=1 to x=5.

Table 8: Examples of running various values of x through linear function f(x) = 5x + 2

|  |  |  |
| --- | --- | --- |
| **x (Independent Variable)** | **f(x) = 5x + 2** | **y (Dependent Variable)** |
| 1 | f(1) = 5\*1 + 2 | 7 |
| 2 | f(2) = 5\*2 + 2 | 12 |
| 3 | f(3) = 5\*3 + 2 | 17 |
| 4 | f(4) = 5\*4 + 2 | 22 |
| 5 | f(5) = 5\*5 + 2 | 27 |

Now that we’ve calculated y for each value of x, we plot each of the (x,y) pairs on a graph and draw a line through each pair. Doing so produces a straight line on a graph, as shown in figure 8.

Figure 16: A graph of the linear function f(x) = 5x + 2



This simple example demonstrated the calculations to produce the (x,y) pairs and what the resulting linear function looks like when drawn on a graph.