#h1 Chapter 14 - Learning 1 - Regression

#h2 Introduction, Concepts, and Nomenclature

#h3 14.1 - Introduction

#pg Ultimately, the most exciting part about machine learning is giving our algorithms the means to find solutions on our behalf. This has two obvious and significant advantages. First, we do not need to specify the rules and parameters ourselves. Coming up with the right set of these is tedious to do by hand, and we would have to repeat the process every time the input data changes slightly. Second, given the space of possible combinations of parameters and rules, it becomes not only tedious but, in most cases, practically impossible to have humans find and implement them. The more complex the data, the more complex the rules. Imagine you had to devise a set of rules that, when applied to raw pixels of an image, predicted whether that picture features a dog. The number of subtle little rules that we would have come up with, for each possible type of dog, for each position, size, angle, and occlusion of the dog in the image, for each pixel, is simply too vast. It would take a human beyond the end of time to iterate and test all these rules. And that is just for dogs. Now imagine doing that again for cats, bunnies, airplanes, staplers, etc.

#h3 14.2 - Why this is probably the most important chapter in this course

#pg I would argue that this chapter and the next one on classification are the key chapters of this course. Not only will you learn how to train a machine learning model to perform regression and classification, but it will also introduce you to some key features, concepts, and ideas in machine learning. We will discuss normalization, loss, error, r-squares, regularization, ordinary least squares, conventional, batch, and stochastic descent, learning rates, stopping criteria, hyperparameter tuning, overfitting, accuracy, precision, recall, F1-scores, receiver-operator-characteristics curves, to name but a few. As such, this chapter (and the next) are dense and long. And there are plenty of notebooks and playgrounds to help your master, understand, and form an intuition of these core machine learning concepts. But before we dive in, it helps to reformulate the overarching framework of what we are trying to accomplish once more. We have done so in earlier chapters, highlighting the different algorithmic objectives, different flavors of algorithms, data types, and learning methodologies. But what all of them have in coming, from a machine learning perspective, is the following. Our model seeks to find a mapping function (a parameterized machine learning model) between some available input data and an output of some interest. We want this mapping to be as informative as possible, either guided by ground truth examples (in supervised learning) or by mathematical regularities in the input data (in unsupervised learning). We set ourselves criteria to tell us about the quality or success of this mapping, given a set of parameters currently defining that mapping. Should the quality of our mapping fall short of our criteria, we will update the parameters as long as such updates yield significant improvements in the quality of our mapping function. When you feel overwhelmed with the details, try to understand what piece of the process we are currently working on. Are we defining what the mapping can look like? Are we defining what constitutes a metric of algorithmic accuracy or success? Or are we mitigating issues that make improving the model's parameters problematic? This overview-focused thinking will hopefully allow you to see similarities across many different algorithms and approaches.

#im ../assets/figures/014/014-01.png 50 256 Figure 14.1 - Infinite regression as illustrated by the Droste effect. Shown here in its original packaging, the Dutch company Droste marketed a cacao powder tin, which on it has an illustration of a woman in traditional clothing carrying a tray with the very same tin of Droste cacao powder, which also features an illustration of a woman in traditional clothing, etc. You get the point.

#h3 14.3 – Getting started with linear regression

#pg Like most texts on machine learning, we will start with linear regression. Why linear regression? For several reasons. First, it is a relatively easy algorithmic objective. There are no non-linearities or complex activation functions and transformations. And the resulting models are easy to understand. They make intuitive sense if your data 1) makes some intuitive sense and 2) contains a mapping of inputs onto an output that is at least to some degree linear. But there is a second reason for starting with linear regression. Depending on your background, you may have already been introduced to linear regression from a statistics point of view, with its unique way of solving the regression problem. Alternatively, your background might be more mathematical and algebraic, in which case you will have been introduced to linear equations and systems and perhaps practiced solving these systems. This background should be useful when introduced to the machine learning approach for building linear regression models. Given this book's target audience, we will work through all three approaches.

#im ../assets/figures/014/014-02.png 50 256 Figure 14.2 - For those not using a password manager (you should), you probably have different variations of the same password for different websites and services. In machine learning, we often have many different words for similar, if not identical concepts, due to the field being an amalgamation of different older fields, like statistics, algebra, and engineering.

#h3 14.4 - Nomenclature - Weights, parameters, and hyperparameters

#pg Given that we will discuss linear regression from the perspective of three different fields, it is inevitable that we will have nomenclature confusion. Let us take a few moments to go over the different ways certain key concepts can be and are referred to. Although we have tried our utmost not to further confuse the reader by being equally inconsistent as many different texts on the topic can be, we can't make any strong promises we will have succeeded. A common source of confusion can arise from the interchangeable use of the word 'parameter' and 'weight'. Parameters are perhaps primarily associated with models or functions, whereas 'weight' is perhaps more often used when describing neural networks. But neural networks are themselves models and implement functions or mappings between inputs and outputs. So, both terms are appropriate descriptors. We could argue, however, that the word parameter is perhaps a more general term than weight. A linear regression model will have weights representing our function's slope and intercepts. Still, in training such a model, we typically specify learning rates and convergence criteria. These parameters are not part of the model and are no longer needed after training has been completed, so calling them weights seems inaccurate. It is common to refer to them as hyperparameters. It is a good strategy to train your model using different values for your hyperparameters and iterate over possible settings in some systematic way. This is called hyperparameter search or tuning. It can be an expensive process, in both time and needed computational resources, as we need to train our algorithm on each combination of different hyperparameters. For more complex models, it is worth the effort. Not only do we regularly find a better and more stable model in this way, but it also makes us more clearly understand some of the interplays between model and data, especially when this goes awry. We will discuss hyperparameter search in Chapter 15 and beyond.

#h3 14.5 - Nomenclature - Variables, regressors, and predictors

#pg In a typical machine learning problem, our input data is a 2D array of data. Each row is a data point, and each column is an input variable. Note that input variables are sometimes called predictors or regressors. If you have ten different input variables (predictors or regressors), we consider your input data to be 10-dimensional, where each variable represents one dimension. Each dimension is associated with an axis.

#im ../assets/figures/014/014-03.png 50 256 Figure 14.3 - Euclidean geometry has three orthogonal and independent axes. Their direction is arbitrary but being orthogonal is key. Without it, the relationships we take for granted in Euclidean geometry fail. We imagine our data to live in similar N-dimensional spaces, where the dimensions/axes represent different variables, like age, sex, BMI, etc. However, these axes are not likely to be completely orthogonal, as our variables are likely not all completely statistically independent. Things in the world correlate. Two pixels next to each other in an image are usually highly correlated. Gender correlates with income and education (it shouldn't, but it does). Somebody's body temperature is correlated with time, as it increases and decreases as a function of the time of day and circadian rhythm. This doesn't mean there is always a necessary causal link between two variables, but correlations can make it hard for models to learn.

#br

#pg Thinking of dimensions in our data as axes along which your input data can vary is in line with the geometric perspective I subscribe to. If your data has three dimensions (e.g., age, height, and BMI of a patient), you can plot each patient as a point in this 3D space. As we have discussed before, our axes might not be orthogonal. In the Euclidean geometry of space, there are three orthogonal directions (x, y, and z). They are orthogonal in that you can move along a direction (or dimension or axis) without any change to the two other directions (dimensions, axes). But consider age, height, and BMI. Are they orthogonal and independent? I would argue that they are not, especially when our patients come from all age groups, from the very young to the very old. As we grow older, we also grow up and grow bigger. Therefore, moving along the age direction will also move us in the height and BMI direction. Understanding these interdependencies between variables that are likely present in your input data is key to building stable machine learning models. Exploration of your data before training your models will significantly help with that.

#h3 14.6 - Data exploration - Medical Expense Dataset

#pg We will use a data set that links total medical costs with several demographics: their age, the US region in which they reside, whether they smoke, their BMI, their sex assigned at birth, and the number of children conceived (if sex is female). The data is hosted on Kaggle for those interested in its history and collection methods: tinyurl.com/y45v3dsn. Whatever kind of data you might be using, it is always a good idea to plot a few summary statistics, distributions, and individual input data points. There will always be surprises. There will always be a need for some decision-making because of outliers, biased classes, missing values, etc. As we rely more and more on specialized toolboxes, we lose that feel for our data. Algorithms today are more and more black boxes, hiding the inner workings and our understanding behind layers of complexity. At the very least, we should know a thing or two about the data we provide to these models as input. The data we use here is already a good example of what you might learn by plotting various views of the data. The figure below plots total medical expense (in dollars x1000) as a function of age, which each dot representing a single data point. Already a few interesting effects can be observed. Across the board, it seems that medical expenses increase as a function of age, the effect we are hoping to predict. However, the data appears to be broken into three distinct bands, shifted relative to each other, see Figure 14.4. How do we explain that?

#im ../assets/figures/014/014-04.png 50 256 Figure 14.4 - Medical expenses dataset, plotting medical expenses as a function of age. Notice the distinct bands. How would you begin to explain them?

#br

#pg Could the shift in expenses be the result of tobacco use? When we color code the data points based on whether an individual self-identifies as a smoker, we see that smokers appear shifted up along the y-axis, relative to their non-smokers, see Figure 14.5A. The relationship between age and medical expenses is about the same for smokers and non-smokers, with the difference being that smokers have a higher baseline level of expenses, independent of age. This explains the differences between the upper and lower band of data well, as smokers appear almost exclusively in the upper band and non-smokers in the lower band. But what about the middle band? Perhaps we can explain other differences in medical expenses by considering sex. Figure 14.B plots the same data but is now color-coded as a function of sex. Before doing so, we should acknowledge that both being a smoker and your sex cannot explain the existence of the three bands alone, as both variables only have two levels. A moot point, it turns out, as sex does not seem to influence where your medical expenses end up on average. Specifically, data points don't cluster into bands based on sex. Used to color code the data points in Figure 14.5c, the final categorical variable is region, which has four levels. Like sex, this does not seem to explain the observed band-like structure. Thus, of these three perspectives of looking at the data, isolating the categorical variables, only smoking appears to increase the patient's medical expenses by a fixed amount, independent of the effect of age. Although, again, it cannot fully account for the pronounced three-band structure we observe.

#im ../assets/figures/014/014-05.png 50 256 Figure 14.5 - Medical expense dataset, plotting medical expenses as a function of age, with individual data points labeled according to their label. (A) Data points indicate whether patients indicated they smoked (red points) or not (green points). (B) Data points indicate the sex assigned at birth (red for males, green for females). (C) Data points indicate the region of residency of the patient, aggregated into four US zones: North-East (yellow), South-East (red), North-West (lime), and South-West (cyan).

#br

#pg Perhaps we can explain the band-like structure with the two other numerical variables: the person's BMI and the number of children. From first principles, this is unlikely. The band-like structure seems to indicate a categorical shift. If there were an effect of BMI or the number of children, we would expect a more gradient-like color coding. However, there appears to be somewhat of a gradient indicating the effect of increased BMI on medical expenses, but if it is there, it is small, see Figure 14.6A. In contrast, the number of children (Figure 14.6B) seems not to affect the amount of medical expenses accrued. However, on closer inspection of the lowest band of data, we see a pronounced effect of the number of children on the total medical expenses of the birth parent (Figure 14.7B). These observations together provide a strong argument for a close visual inspection of your data. Without it, you might miss such subtle but profound effects.

#im ../assets/figures/014/014-06.png 50 256 Figure 14.6 - Plotting medical expenses as a function of age, with individual data points color-coded according to a person's BMI (A) and the number of children they conceived (B). We can observe a faint trend toward higher medical expenses for higher BMIs. This should be an expected result: obesity is associated with many medical issues, such as diabetes type II and cardiovascular disease. It is perhaps odd that the effect is not stronger. In contrast, it doesn't seem the number of children has any clear influence on medical expenses of the birth parent. However, a closer look does reveal an effect, but only for a subset of our data, which can be seen more clearly in 14.7b. This is not all that surprising in the end, as childbearing in itself can be quite expensive.

#br

#pg In conclusion, the number of children you brought into this world, your BMI, and whether you are a smoker all influence your total amount of medical expenses. Geographic region and your sex assigned at birth do not. However, it does leave the three-band structure we observe in the data unexplained by each variable when considered in isolation. It is possible that this grouping effect is due to a variable not included in the data set, such as a difference in the type of insurance plan or some other demographic not listed. Alternatively, they could be an interaction effect between two or more variables. We can test the existence of interaction effects with the methods we outline in this chapter (to some extent). Therefore, we will return to this complete data set later. For now, we will start by explaining a single output of interest (medical expenses) from a single input variable (age).

#im ../assets/figures/014/014-07.png 50 256 Figure 14.7 - Medical expense dataset, identical to Figure 14.6, but cropped to the lower band of data points only. Within this lower band, BMI doesn't seem to influence medical expenses (14.7b). In contrast, the effect of the number of children conceived on medical expenses appears to have been obscured in Figure 14.6, as individual data points overlapped too much to see a clear shift. By focusing only on those data points found in the lower band, when we do so, we do see that each additional child conceived shifts the total medical expenditure up by some significant amount.

#h3 14.7 - A subset of the data

#pg Moving forward, we only use 20 random data points from the original data. We do this because we would like to work through the entire set of computations needed to fit a linear model without the need for many pages to detail this for thousands and thousands of data points. At the same time, we opt for selectively removing the data points that form the two smaller bands seen. We are aware of our cherry-picking tendencies, and please do not apply such filtering to any real data sets you might end up working with. We do it here, so our data has a pronounced and understandable relationship without having to resort to synthetic data. Finally, already at the raw data level, we divided medical expenses by 1000 dollars to bring it within a range that is like age. Thus, moving one unit up or down the y-axis equates to an increase or decrease of 1000 dollars.

#im ../assets/figures/014/014-08.png 50 256 Figure 14.8 - (A) All remaining data points after removing the data points that make up the smaller bands. (B) We further subset the data in (A) to a random 20 points for use throughout this chapter.

#h2 Explaining variance and quantifying performance

#h3 14.8 – Manual Tuning

#pg The key benefit of using machine learning models is that they can automatically find the best-fitting parameters that allow the model to perform with acceptable accuracy. No need for us to painstakingly set the parameters that combine our input variables to yield a model with the best possible performance. Instead, weights are adjusted through numerical optimization over many training examples, slightly changing millions of parameters on each iteration to improve accuracy by a little. This is not a human endeavor and an impossible undertaking at this point to even attempt, given the complexity and the sheer number of parameters in even modest machine learning models in production. So, you might be surprised that we will pick the parameters for our first linear regression model by hand in this section. We will get to an automated approach soon enough (in the next section). But first, we need to define precisely how we measure, numerically and objectively, the performance of our machine learning model. We can derive, compute, and compare these performance metrics without learning. So, for simplicity, we will use common sense, rather than numerical optimization, to create a few machine learning with varying performance levels.

#h3 14.9 – Defining our model

#pg We will start our discussion with a straightforward regression model to predict a patient's total medical expenses based on age. The data we will use for this can be found in Figure 14.8b, which, as discussed, is a small toy dataset based on only 20 data points out of roughly 1000 datapoint in the original dataset. Our model will have two parameters a and b, where a is the slope or directional coefficient of the function that maps x (age) to y (medical expenses), and b is the bias, intercept or offset (as discussed before, all three terms refer to the same thing, the names just originated from different disciplines). Therefore, Equation 14.1 defines our linear regression model that uses age to predict medical expenditures, assuming a linear relationship between the two variables.

#im ../assets/equations/014/png/014-01.png 50 32 Equation 14.1 - Our general linear regression model. Our prediction y is a function of x times parameters a (the slope) plus parameter b (the offset or intercept)

#h3 14.10 - Quantifying performance

#pg The usefulness of any machine learning model is a function of how well it performs across a wide range of inputs. How we quantify performance depends, amongst other things, on the algorithm we use. Whereas the performance of a classification algorithm should be reflected in the accuracy of its predictions, what constitutes an accurate prediction for a regression algorithm is less well defined. Instead, we need to rely on some measure of getting at least reasonably close to the actual output value we are trying to predict. And what constitutes reasonable isn't necessarily a given either. Therefore, a model's performance in regression is quantified in terms of how well it explains the data: the model's 'goodness-of-fit'. For regression, we typically use the r-squared value, or its more official and serious name: the coefficient of determination.

#h3 14.11 - The intuition behind the R-squared metric

#pg In the next paragraph, we will detail the formulas we need to compute the r squared. Here, we focus instead on the intuition that underlies measuring performance. First, let us briefly revisit the material discussed in chapter 13. In this chapter, we defined several key statistical metrics, like the mean, variance, and standard deviation of a random variable x. If you have read chapter 13 or have a solid knowledge of statistics, these concepts should now be familiar. For example, when we measure the height of a randomly selected group of individuals, we understand that we can compute a mean height and some variation around that height from this sampling. In statistics, these metrics will serve to compare, contrast, and estimate, with some probability or statistical significance, the difference between different groups of individuals. For example, we could test whether the Dutch are significantly taller than people from Uruguay. Not all Dutch individuals need to be taller than all Uruguayans; that is a far too stringent statistical criterium. Instead, we want to show that the mean height of the Dutch is different enough (statistically speaking) from the mean height of the Uruguayan people. In machine learning, we use highly similar metrics to quantify machine learning model performance but with a small but significant difference. In statistics, we are mainly concerned with the variance we observe within a particular random variable x. In machine learning, we want to condition that variance on how much of that variance can be explained by our model and how much variance is left over that cannot be explained by our model. In this context, we should not equate "explaining" to some conceptual scientific explanation based on our model. Rather, think of it as our model's ability to capture the natural variation we see in our outputs. The output of a high-performing model will correlate with the actual outputs because it has found some combination of inputs from which it can produce an output that correlates and varies very similarly to the actual ground truth outputs. Again, the model isn't engaged in finding a conceptual framework for how a person's IQ depends on genes, environment, access to certain resources, diet, etc. That will be the human scientific interpretation of the model. The model, mathematically speaking, just aims to have its prediction move in unison with ground truth data.

#im ../assets/figures/014/014-09.png 50 256 Figure 14.9 - There are differences in the average height of individuals between nations. The origin of the exceptional height of the Dutch has been studied extensively, suggesting it is a combination of natural selection (the Dutch value height in choosing a partner) and diet.

#br

#pg For example, imagine we measure the IQ of a modest number of people. When we discover that an individual has an IQ of 200, we ought to be surprised. After all, an IQ that high is so extremely rare (statistically speaking) that it has only been measured in a handful of people. In a way, machine learning aims at tempering this surprise. With the right model, we could explain the high IQ in some way: We would agree that an IQ of 200 is rare. But we would further point out that the person with that IQ had parents who both had high IQs. We then point out that our model has found that aspect to be highly predictive of a person's IQ. We would therefore conclude that an IQ of 200 is still quite surprising, but less so, given the circumstantial evidence. We have explained some of the variance we observed in our data. Note that we are not trying to explain these highly unusual data points. Instead, we are trying to explain the variance across all data points. If someone has an IQ very close to the population mean, our model must still get that right. In this case, it will probably rely on the absence of factors that typically push a person's IQ to be exceptionally high (or low).

#im ../assets/figures/014/014-10.png 50 256 Figure 14.10 - IQ validity, precision, and use have remained controversial ever since their inception by French psychologist and neuroscientist Binet. "IQ is what an IQ test measures" is a joke amongst psychologists and acknowledges that the test seems to measure some stable and reproducible quantity that varies across people but that it is unclear how exactly it relates to human intelligence as a broad and multifaceted quality that can take on many different forms. That doesn't mean there are some brilliant individuals, like mathematician Terence Tao and physicist Marie Curie, both with estimated IQs of around 200. Note that Marie Curie was one of 4 people ever to have been awarded the Nobel Prize twice.

#br

#pg However, suppose we can't find the right reasons to explain why we observe an exceptionally high IQ. In that case, we might be forced to conclude that the variation in IQ is unexplainable and due to some random process or that we simply did not include the things to are predictive as inputs to our model, possibly phrased as follows: "that person has an IQ of 200? That is surprising, given our model has found nothing that would have predicted that happening. Either we missed some crucial aspect, or IQ is still more or less random". We will typically find ourselves somewhere in between these two extremes. We can discover predictive things, but they rarely are entirely deterministic. We might overlook or not have measured other variables that could add predictive variance explaining power. And we are bound to have some measurement error in our data to begin with, which can never be explained. In other words, there will be residual variance in our outcome data that isn't explained by our model. To recap: in statistics, we often describe a variable's variance. In machine learning, we are interested in how much of that variance can be explained by our model. Our model, in turn, will use input data to get close to what is observed as actual output. Whatever our model cannot explain is considered residual variance. A good model will explain a substantial amount of the variance and leave little residual variance. This ratio between total, explained, and residual variance gives rise to the r-squared metric, which we will discuss next.

#h3 14.12 - SST, SSE, SSR

#pg Formalizing the above, let us work through the computations needed to arrive at the r-squared value. In the previous chapter on statistics, we defined the mean, variance, and standard deviations and outlined the formulas we need to compute them. In machine learning, it is common practice to use the intermediate precursor metrics, which carry the same information but are not standardized and normalized. In the end, the metrics computed aren't any different. They appear to be based on different conventions and disciplines brought to the table at various points during the development of machine learning algorithms. Remember our original formula for the population variance of a variable y:

#im ../assets/equations/014/png/014-02.png 50 96 Equation 14.2 - The population variance of a variable y. For generic formulas that describe specific statistical properties, such as moments, we will typically use a variable x, indexed by i. In machine learning, we typically reserve x to denote an input variable and y as an output variable. This is true for Equation 14.1, which relates input variable x to output variable y. However, here x denotes any random variable over which we can compute the variance.

#br

#pg Notice that the last step in computing the variance is to divide it by the number of elements in variable x. In other words, the variance of variable x is an average across all elements of x. Before we normalize, however, we already have a useful precursor of variance, known as the Sum of Squares. It is called that way for an apparent reason: we square the difference between each data point xi and the mean of x: (xi - x)2 and sum them all together. This Sum of Squares measures the variance in x and only x. We refer to it as the Sum of Squares Total or SST, defined in Figure 14.3, because it captures the total variation observed in x, independent of any other variable we could use to explain the variance in x. As we discussed above, in the context of IQ, we introduce a model to explain as much variance as possible in x. This allows us to define two more quantities, the Sum of Squares Regression (SSR, Figure 14.4) and the Sum of Squares Error (SSE, Figure 14.5). Theoretically, the SSR is the amount of SST we could explain using our model; the SSE is whatever is left over. From this, it naturally follows that SST = SSR + SSE.

#im ../assets/equations/014/png/014-03.png 50 64

#im ../assets/equations/014/png/014-04.png 50 64

#im ../assets/equations/014/png/014-05.png 50 16 Equations 14.3-5 - Without the normalization by the number of data points n, we get the precursors of variance of x: the sum of squares total (SST), the sum of squares regression (SSR), and the sum of squares error (SSE).

#h3 14.13 - Coefficient of determination or r-squared

#pg The ratio between the SST and SSR into SST provides us with a metric for our model's performance: the r-squared, also known as the coefficient of determination, see Equation 14.6. Figure 14.11 shows a visualization of these different metrics using synthetic data to prime you on how to interpret the various metrics we have discussed so far. Don't worry too much about fully grasping every single detail. In the next section, we will examine how these metrics change using real data and how they yield an informative r-squared metric, which measures the so-called goodness-of-fit of our model to the data. This manual tweaking of parameters allows us to present ideal and less-than-ideal models and compare the difference in performance, as measured by the r-squared.

#im ../assets/equations/014/png/014-06.png 50 64 Equation 14.6 - The r-squared value measuring the goodness-of-fit or our model is given by the ratio between the sum of squares regression and the sum of squares total.

#h3 14.14 - A quick word of caution

#pg As a quick aside, please remember that other people refer to these different quantities differently at times. Somebody might take the abbreviation SSE to mean the Sum of Squares Explained, and others will assume it is the Sum of Squares Error. Similarly, some consider SSR to mean the Sum of Squares Residual, whereas others define it as the Sum of Squares Regression. Very confusing indeed, as it comes out the exact opposite, depending on what you think the abbreviations stand for!

#im ../assets/figures/014/014-11.png 50 256 Figure 14.11 - R-squared values for two models. (A) Parameters that produce a bad fitting model explain relatively little variance in the output x (SSR), leaving most of it unexplained (SSE), resulting in a low r-squared value. (B) In contrast, parameters that yield a good fitting model will have most of the variance explained and little left unexplained, resulting in a high r-squared value.

#h2 Manual Tuning of Parameters

#h3 14.15 – Our very first regression model

#pg Without training data that links patients' age to their actual medical expenses, it will be pretty challenging to build a model with any predictive performance. Theoretically speaking, the best we could do is guess. But if we at least knew the average medical expense across individuals, we could make that guess a little bit more informed. Instead of picking a random number and hoping for the best, the best model in this situation always predicts the same value. Specifically, no matter what we might know about the individual (age, sex, etc.), we always predict that the patient's medical expenses will be equal to this mean, see Equation 14.7.

#im ../assets/equations/014/png/014-07.png 50 32 Equation 14.7 - Our first regression model simply outputs the mean of all previously observed outcomes as its prediction.

#br

#pg Technically, this is a predictive model, albeit not very good. But it is a good starting point to start understanding the r-squared metric. Figure 14.12 plots the data and the function that describes our machine learning model. And indeed, no matter what the input to our model could be (age), our prediction remains the same, as any point on the x-axis maps to the same value on the y-axis.

#im ../assets/figures/014/014-12.png 50 256 Figure 14.12 - Our first regression model simply outputs the mean of all previously observed outcomes as its prediction, utterly agnostic to the patient's age.

#pg This agnostic approach of our model is not doing us any favors. There is substantial predictive power in the data, as increases in age are paralleled by increased medical expenses. A closer look at the equations we defined for computing the total variance, explained variance, and residual variance shows our model to be a perfect edge case in computing the r squared. The sum of squares totals the difference between each output yi and its mean. By setting our model output to be the same, mean the equation for the SSR (the amount of variance in the output data we explain using our model) will always be zero, see Equation 14.8. So, the amount of variance explained by our model, relative to the total amount of variance in the output, is 0. As a result, our r-squared is at exactly 0, the worst r-square to be awarded to a machine learning model.

#im ../assets/equations/014/png/014-08.png 50 64 Equation 14.8 - A model that always predicts the mean of previously encountered outputs y will never explain any variation in y, making it quite useless.

#h3 14.16 – Improving performance - introducing a positive slope

#pg We can do better than our first attempt. By eyeballing it, it seems there is a positive relationship between expenses and age. Therefore, our slope a should be positive. Second, just by being born, you probably already have some medical expenses. So, the intercept b should also be positive, as it reflects the base predicted expenditure at age = 0. Using a = 0.05 and b = 4 (given that we standardized the medical expenses to US dollars x1000, we suggest that on day one as a newborn, you will have 4000 dollars of medical expenses to your name on average. When we run the numbers, we see that our model isn't very good, but better than predicting the mean for sure, with an r-squared of 0.11.

#im ../assets/figures/014/014-13.png 50 256 Figure 14.13 - Our second model has a positive slope (a=0.05) and positive intercept (4.00). You can interpret these parameters as follows: the amount of medical expenses predicted is 50x your age (since a=0.05, one year is 0.05 times $1000 = 50 dollars), plus a baseline 'out of pocket' of $4000 (b=4.0). By the age of 30, our model predicts you will have $50 \* 30yr + $4000 = 5500 dollars of medical expenses. Reasonable assumptions aside, our model isn't all that great performance-wise. A positive slope is a good starting point, but we must change it further for a closer fit.

#h3 14.17 - Improving performance - an empirical constraint

#pg Perhaps we were a little too negative, assuming infants have a non-zero amount of medical expenses the day they are born. So, as a subsequent attempt, we set b = 0, which means that no matter what, we will predict you will have 0 dollars in medical expenses on day 1. Whatever happens after that is solely a function of the slope of our function, which is given by our parameter a. This fit significantly improved over the previous two models, yielding an r-square of 0.5. However, some data points for older people are off quite a bit compared to their younger counterparts.

#im ../assets/figures/014/014-14.png 50 256 Figure 14.14 - Our third model uses an intercept b = 0, forcing the slope to explain the variance in our outcome data y. We get reasonably close with our fit this time round, but we can still improve the model further.

#h3 14.18 – Improving performance - close to ideal.

#pg We decided to be a little adventurous for our final manually tuned model. We increase the slope a to 0.25 while at the same time setting our intercept b to be negative (b=-2.25. An interesting choice since a negative intercept means that when you are born, the insurance company owes you money, as your medical expenses are predicted to be negative. Surely, insurance companies don't give out one-time bonuses to newborns, even if there is the expectation they will make that money back by quite a margin later during that individual's life. To be fair, this 'day one bonus' is an interpretation of the model. A negative intercept can also mean that the linear model just falls short of being able to explain the observed data. And this isn't surprising. If we look more closely at the data in 14.9A, we can see that the relationship isn't entirely linear. There is a non-linear effect that accelerates a person's medical expenses in the same way compound interest accelerates the growth in your savings. Most likely, this is an exponential effect with an exponent greater than 1 (an exponential component of 1 resolves to a linear model). This explains why a negative intercept works well. The data flattens out towards the younger individuals, intersecting close to (0,0) or assuming that at age = 0, you will have $0 worth of medical expenses. The linear model is unable to 'flatten out'. It continues unabated below $0 worth of expenses. Despite this inability of our linear model to accurately describe or predict medical expenses at an early age, overall, its performance is quite exceptional for a manually tuned set of parameters, yielding an r-squared of 0.90.

#im ../assets/figures/014/014-15.png 50 256 Figure 14.15 - Our fourth and final hand-tuned model does well. With a negative intercept (b < 0) and a positive slope (a < 0), we capture a large portion of the variance in observed outcomes.

#h3 14.19 - R-squared vs. Correlations

#pg The r-squared and correlation coefficients are often confused and used interchangeably. This is incorrect. A correlation coefficient is bound between -1 and 1 (completely negative to completely positive), whereas the R-square is bound between 0 to 1 (no variance explained to all variance explained by our model). Furthermore, we can get a high R-square even when our correlation is low. Think back to output signal y that doesn't change as a function of x. This means there should not be any significant correlation between x and y. However, our model can still be a good fit if that model would simply output the mean value of y for any x, and y shows a low amount of variation.

#h3 14.20 - It's only model

#pg As we observed, the relationship between age and medical expenses isn't linear. There is a slight quadratic or exponential component in how medical expenses increase with age. However, the performance we got out of it was robust and yielded good predictions across different inputs. It is not uncommon for edge cases to represent themselves in your data. At the extremes of your data, non-linear effects might creep up. In addition, there are usually fewer data at these extremes to drive the overall push toward finding the best parameters.

#im ../assets/figures/014/014-16.png 50 256 Figure 14.16 - In the end, our models simplify the complex interplay of causes and effects.

#br

#pg Ultimately, our models are just simplifying some more complex interplay of causes and effects. This makes models valuable tools in science. Yes, they do not always tell the complete story of everything going on at the empirical level. But in many scenarios, extra details might improve the accuracy by which they can describe and predict our observations, but at the cost of explanatory value. For example, we know now that Newton's equations that describe motion break down when we consider objects the size of planets, stars, and galaxies. At these scales, we must use the equations put forward by Einstein's theory of relativity instead. But it is sufficiently accurate in describing the trajectory of a game-winning home run. However, humans have now created technologies that operate at a scale where Einstein's equations are preferred. For example, our global position system, consisting of many earth-orbiting satellites working in unison, must factor in the effects of gravity to correctly triangulate positions on the earth's surface.

#h2 The Statistical Approach

#h3 14.21 - Estimating parameters algorithmically

#pg So far, we have used our intuition to manually set parameters that provide a close fit between our model's predictions and actual outcomes. Now, halfway into chapter 14, it is time for us to start automatically (i.e., algorithmically) estimating these optimal parameters. However, once again, we will embark on a slightly longer route before arriving at the general framework of machine learning using a method that applies to virtually all optimization problems.

#h3 14.22 - Closed Form versus Iterative Methods

#pg In machine learning, we typically iteratively improve our model's parameters based on an error we compute between our predictions and the actual outcomes. During each iteration, we carefully update the parameters by moving a fraction in the direction of where we predict they will yield better predictions. Hopefully, we will settle at some point at a set of optimal parameters. If we are fortunate, these will be the best parameters possible. These parameters are optimal in some region of the parameter space we explored. Gradient descent (GD) is the method of choice for many machine learning algorithms to systemically traverse the parameter space in pursuit of these optimal values. And we will discuss GD in detail later in this chapter. Whereas this chapter can be the backbone of this course, gradient descent is the backbone of this chapter. It provides a very elegant approach and geometric intuition on how to think about and train highly complex machine learning models.

#br

#pg However, two methods allow us to compute optimal global parameters. In addition, it does not need us to iterate and search the parameter space carefully. Instead, they will yield the optimal parameters from a single pass over the training data. They are examples of closed-form solutions, which are deterministic and derived from first principles to always produce the most optimal solution under specific conditions. Mathematically, both methods are pretty much equivalent, but their implementation may, at first, look very different. And one of them is rooted in the field of statistics, the other a product of algebra. Before we describe them, we should answer why we don't always use these methods. Why bother with a slow, non-deterministic algorithm when you can bet on a horse guaranteed to win? Not all races are equal. Specifically, the statistical and algebraic methods won't be available for more complicated, non-linear machine learning models. The truth is, the reason they work so well for linear models is that linear models, per definition, only have one minimum - one set of optimal parameters. We need a different approach when that linearity is no longer a valid assumption. Specifically, we need an iterative method like gradient descent, allowing us to search for and find a set of parameters that yield predictions worth making. To fully appreciate this, it helps to visualize these parameter spaces and outline how we define and quantify a minimum in them. This will prepare us better for our discussion on gradient descent. But we will need to first introduce two metrics commonly used in machine learning to describe and measure model error and performance.

#h3 14.23 - Error

#pg You may feel we already went through the steps necessary to compute model performance. Did we not just define the r-squared metric? Yes, in the previous chapter, we computed three related measures that allowed us to quantify our model's performance. We demonstrated that the Sum of Squares Total (SST), Regression (SSR), and Error (SSE) could tell us how much of the variance our model can explain in the variance observed in the output we are trying to predict. We would like to acknowledge that introducing two additional metrics to capture a model's performance can be slightly confusing initially. But stick with us. Hopefully, it will become clear soon enough.

#im ../assets/equations/014/png/014-09.png 50 32 Equation 14.9 - We define error associated with a data point i as the difference between the predicted and actual outcomes.

#br

#pg The first metric is simply called error and is not specific to any method described in this chapter, nor is it specific to linear regression. Error is the difference between our model's predictions and some objective ground truth, see Equation 14.9. If our model predicts you will have 5K in medical expenses, yet it is $6k, the error is -$1000. Notice that it matters what we subtract from what. If our model predicted $6k, yet the actual outcome was $5k, the error would have been +$1000. We must keep track of the sign and keep track of it correctly because updating our parameters will depend on it. If we consistently overshoot the actual medical expenses, we want to consider decreasing the slope or offset of my model and vice versa.

#h3 14.24 - Turning Error into Loss

#pg The second metric is loss, which captures how much variance the model (still) hasn't accounted for. Minimizing overall loss is the objective during optimization or learning. But why do we need a new measure when we already have the error? For reasons detailed above, the error metric is a bit ambiguous. We have already seen that our error can be both positive and negative. How should we interpret these values in terms of performance? Is an error of -1 more than an error of 1? In other cases, we have categorical outcomes, like in a binary classification problem. If the labels in our data are defined as 0 and 1 (without any mathematical relevance), what does the error mean in that case? And since the error will always be -1, 0 or 1, are these values even useful as measures of model performance to begin with? This is where loss comes in. Unlike error, the way we compute loss is specific to the type of outputs our model generates. We rely on specific statistical distributions for binary and multiclass classification problems to convert error into the loss we need to optimize our models, which we will discuss in detail in the next chapter.

#im ../assets/equations/014/png/014-10.png 50 64 Equation 14.10 - The Sum of Squares Error (SSE), defined as the squared difference between model prediction and model outcome summed across n data points.

#br

#pg For linear regression, however, we have already seen the very equation that computes loss from error before. More specifically, it is the sum of squares error (SSE). We haven't used the SSE before, relying on the ratio between SST and SSR to give us the r-squared value instead. But by plotting it once more below, we can see how it fits into our current objective. Notice how that definition consists of three steps. First, we take the difference between our model's prediction and the actual outcome. If that looks familiar, it should because that is the error we already defined above. After that, we square the error value, which removes the sign. For a good reason, because for loss, we are not interested in which way we got it wrong (over or underestimation), but rather the extent to which we got it wrong.

Equation 14.11 - We define loss as the average squared difference between model prediction and model outcome across some subset of data points n.

#br

#pg The only thing still left up to us to decide is which errors are summed together. One way is to average the loss across all training data before updating our parameters. Alternatively, we can compute error and loss for each data point, updating the parameters every time we consider a new training data point. And then there is some middle ground between these two extremes. Why we would choose one extreme over another or opt to stay somewhere in the middle depends on assumptions we make about our data and model and is something we will discuss in detail later in the chapter when we discuss different training regimes.

#h3 14.25 - Loss Landscapes

#pg Now we have defined loss, we can visualize it as a function of a range of parameters. We generated a small synthetic data set that is entirely linear, except for a bit of noise added to the outcome y. For its parameters, we set the slope a and the offset b to 1, giving us a straightforward linear model of y = ax + b, see Figure 14.17a. With only a small amount of added noise, it is safe to say that the lowest error is found at a=1 and b=1. But what about all other possible combinations of a and b? How does the landscape of loss vary as a function of varying values of and b? We can plot the loss as a 3D mesh surface, where the x and y axis are ranges of different values for a and b, respectively, with the z-axis representing the loss of the model given a particular pair of (a,b) parameters. Notice that when we do, the resulting loss landscape (shown in Figure 14.17b) for our strictly linear dataset has a single optimal lowest point of a loss close to 0. Given that that point represents the parameters used to generate the synthetic data, it is not surprising that it is associated with the lowest theoretical loss we can achieve using our linear regression model. Move away in any direction from this point (by increasing or decreasing one or more of the parameters, and the error you observe between actual outcomes and your model's prediction will always be higher. We call this a convex space of solutions: a single valley with the optimal parameters at its lowest point. Keep in mind that this convexity is a property of the machine learning model, not the data. The data can be highly non-linear, and a machine learning model that captures such non-linearity might find a very different global minimum.

#im ../assets/figures/014/014-17.png 50 256 Figure 14.17 - Synthetically created data sets (A) and (C) and their associated loss landscapes (B) and (D). The synthetic data in (A) is linear: y = ax + b. As a result, the associated loss landscape for parameters a and b is convex, with a single global optimal set of parameters (a,b) that yield the best theoretically possible performance. The data in (B) was created by adding a sinusoid with frequency a to a sinusoid with frequency b. This yields a highly non-convex loss landscape, shown in (D).

#br

#pg What if instead the data shows a non-linear relationship between inputs and outputs? Figure 14.17c plots a different synthetic dataset which we made highly non-linear by design. When we plot the corresponding loss landscape for a range of parameter combinations (a,b), we can see multiple local neighborhoods where the parameters yield some optimal solution. This is true from a local perspective as all around such minima; the loss is higher. That doesn't mean that some other, more distant combination of parameters yields an even lower loss. If we were to try and fit a linear model to this data, we would not do well. The data in Fig 14.15c can't be described as a straight line, no matter how much we try and squint. It might fit some subset of the data, but it will also be at the expense of some other data. Like a USB cable straight from Dante's inferno, no matter which way you rotate it, it never seems to be aligned with the port you want it to connect with.

#h3 14.26 - Statistical estimates for our parameters a and b

#pg Despite their limited use in machine learning, the statistical and algebraic methods still offer crucial insights into how to compute, estimate and optimize models to fit data. We will begin with the more familiar and intuitive method of the two, which you may have already come across in a statistics course where it is simply known as linear regression (with regression used as a verb, as in 'to regress'). It explicitly uses estimates of the mean, variance, and covariance of an input variable x and an output variable y to compute the slope and intercept we need as parameters for our model. If we have theoretical reasons to believe our model is indeed linear, we can use either the statistical or algebraic approach to find the optimal parameters.

52 9.75 204.49 9.79 44.74

59 12.62 453.69 36.03 127.85

32 4.65 32.49 4.23 11.73

18 2.20 388.09 19.50 87.00

54 11.51 265.69 23.94 79.75

50 9.55 151.29 8.58 36.03

34 5.12 13.69 2.24 5.53

52 11.49 204.49 23.70 69.62

18 1.62 388.09 24.98 98.46

58 12.23 412.09 31.49 113.92

19 1.74 349.69 23.84 91.31

38 5.40 0.09 1.49 -0.37

45 7.45 53.29 0.68 6.03

18 1.14 388.09 30.06 108.01

28 3.17 94.09 11.89 33.45

25 4.50 161.29 4.47 26.86

52 12.59 204.49 35.67 85.41

33 5.38 22.09 1.55 5.85

25 2.72 161.29 15.20 49.51

44 7.73 39.69 1.23 6.97

37.70 6.62 3988.20 310.55 1087.65

Table 14.1 - Steps in computing the best fitting parameters using the statistical approach to linear regression.

#br

#pg More specifically, means, variances, and covariances are all used to weigh the contribution of each data point to the overall estimate of slope and intercept based on how statistically probable that data point is. Therefore, it is useful to think that at its core, linear regression aims at answering the following statistical question: what values of a (the slope) and b (the intercept) would have been the most likely to have generated the data we observed? Answering that question relies on metrics we have already defined, which can be found in Table 14.1. The table lists the 20 data points in our small, subsampled data set related to medical expenses and age (Figure 14.9), with one column added for each subsequent value we need to compute from the original (x,y) data. These are aggregated across all data points and used by Equations 14.12 and 14.13 to compute the best slope a and intercept b. We show the resulting fit in Figure 14.16. We can compare this fit to our best efforts in the previous section to generate parameters manually. It appears our guesstimated parameters were not too far from the theoretical optimal values given by the statistical approach outlined in this section.

#im ../assets/equations/014/png/014-12.png 50 128

#im ../assets/equations/014/png/014-13.png 50 32 Equations 14.12-13 - Slope a is the ratio between the covariance between x and y and the variance in x. We compute intercept b from the means of x and y and newly found parameter a.

#h3 14.27 - Beyond a single input variable

#pg The statistical approach is easy to compute and understand when considering only a single input variable's effect on a single output variable. It seems unlikely we would ever build models this succinct. The math behind the method doesn't change by adding more input variables. However, it will become increasingly more cumbersome to iterate over all the computations. Fortunately, we can re-express our statistical take on linear regression in algebraic terms, using matrix operations to quickly estimate the optimal parameters for our model with only a fraction of the code. This approach will be the focus of the next module.

#im ../assets/figures/014/014-18.png 50 256 Figure 14.18 - Our regression model, derived using the above set of data and equations. The model found through this method aligns nicely with our expectations and our attempt to find the parameters manually through estimation.

#h2 The Algebraic Approach

#h3 14.28 - The Algebraic Approach - Ordinary Least Squares

#pg The statistical approach works well, and its computations are straightforward. You can even do them all by hand for relatively small data sets. Unfortunately, this simplicity breaks down when considering more than one input variable. There are many more things to compute in the case of 2 or more input variables. You might think that for two input variables, we would simply have to compute all the values we computed for a single input variable x twice. This, unfortunately, is not the case. As we add more input variables to the mix, we must keep track of the influence of each input variable x on y and each combination of input variables on y. For two parameters, this is still reasonably easy to manage, but beyond that, our tabular approach becomes cumbersome and inefficient. In this scenario, we can use the algebraic method of ordinary least squares (OLS). The main difference between the methods is that the first computes all necessary values by iterating over each pair of observations. It turns out that if we can represent these values not as single pairs of observations, but instead like vectors and matrices, many of the computations can be performed by straightforward matrix multiplications.

#h3 14.29 - From elements and lists to vectors and matrices

#pg Think back to the chapter on transformations. There, we used a concise representation of a particular type of transformation that could be applied in one go to a matrix of points of arbitrary size. Take rotation, for example. Imagine we have a list of (x,y) data points centered around an origin of (0,0). Let's imagine we want to rotate each (x,y) data point around the origin by 45 degrees clockwise. This is computed for a single data point, as shown in Equation 14.15. Notice that we need a little helper equation to transform our rotation from degrees to radians, as in Equation 14.14).

#im ../assets/equations/014/png/014-14.png 50 48

#im ../assets/equations/014/png/014-15.png 50 24

#im ../assets/equations/014/png/014-16.png 50 24 Equations 14.14-16 - To rotate a 2D point given by (x,y) by a certain angle (45 degrees clockwise in our current example), we use Equations 14.15 and 14.16. We will need to convert our angle in degrees to radians (alpha) using Equation 14.14.

#br

#pg Of course, our computer program is happy to loop over each datapoint tuple (x,y) in list P and apply the rotation. That means we need an iterator, like a for loop or some modern version, see Code snippet 14.1. There is nothing inherently wrong with this implementation. However, we can perform the same computation using a single matrix multiplication operation. This greatly simplifies the math that needs to be converted into code. And if we are using a numerical package like NumPy, it also cleans up our code, removing the need for a for loop. Even better, although the computations performed at the CPU level is the same in the end, optimizations within NumPy and specialized hardware solutions like GPUs also significantly speed up the computation. And for large data sets, we will use them for training deep belief neural networks; these optimizations require us to train models within a reasonable time. Let's perform the same rotation using matrix multiplication.

#br

#cc import math

#cc degrees = 45

#cc alpha = degrees / 180 \* math.pi

#cc P\_rotated = []

#cc for i in range(len(P)):

#cc x = P[i,0]

#cc y = P[i,1]

#cc x\_rot = x \\* math.cos(alpha) + y \\* math.sin(alpha)

#cc y\_rot = x \\* -math.sin(alpha) + y \\* math.cos(alpha)

#cc P\_rotated.append((x\_rot,y\_rot))

#br

#ca Code Snippet 14.1 - This code snippet uses core Python to compute the rotation specified in Equations 14.15 and 14.16.

#br

#pg First, we need to define our matrices, P and R. P is a n x 2 array, in which each row constitutes one datapoint with coordinates (xi,yi). The matrix R is a 2x2 square rotation matrix. If you want, you can work out how to construct this rotation matrix from Equation 14.15, but you can trust us that this is how one specifies a rotation around the origin in 2 dimensions, thanks to the work of great mathematicians like Euler, Laplace, LaGrange, Descartes, and many others. To rotate a 2D point given by (x,y) by a certain angle (45 degrees clockwise), we first convert the angle to radians using Equation 14.14. Second, we define our P matrix as an nx2 matrix where each row is a (x,y) coordinate (14.17), as well as our 2x2 rotation matrix (14.18). Computing the actual transformation across all points in P is done as a matrix multiplication operation (14.19). The python implementation is found in Code Snippet 14.2.

#im ../assets/equations/014/png/014-17.png 50 64

#im ../assets/equations/014/png/014-18.png 50 32

#im ../assets/equations/014/png/014-19.png 50 16 Equations 14.17-19 - We can rotate all points in one matrix operation using NumPy.

#br

#cc import NumPy as np

#cc degrees = 45

#cc alpha = degrees / 180 \* np.pi

#cc R = np.array([[np.cos(alpha),np.sin(alpha)],

#cc [np.sin(alpha),np.cos(alpha)]]);

#cc P\_rotated = np.dot(P,R)

#cc

#ca Code Snippet 14.2 - The NumPy variant of the code snippet is much more concise.

#h3 14.30 - Ordinary Least Squares

#pg In the previous example, we replaced all that tedious indexing with a for loop with a single concise line of code to implement a rotation operating on a set of points. But how does this relate to finding the optimal parameters of a linear regression model, given some input and output data? In the same we were able to reformulate an iterative method of rotating each point by some angle in turn as matrix multiplication, giving us the same result, we can reformulate the statistical method of the previous section (also operating on one datapoint at a time) in a very similar way. This is called the ordinary least squares method, and its formula is written out in equation 14.20.

#im ../assets/equations/014/png/014-20.png 50 48 Equation 14.20 - The equation for Ordinary Least Squares (OLS), an algebraic method to estimate the optimal parameters beta, given an input matrix X and output matrix Y

#br

#pg In words, you can read Equation 14.20 as follows. The parameters associated with the optimal mapping of input data X onto output data Y are given by the inverse of the inner product of X transposed and X, multiplied by the transpose of matrix X, multiplied by matrix Y. Transposing a matrix means flipping its axes. So, a matrix of n x m because a matrix of m x n. In statistics, we sometimes refer to it as a pivot. An inverse of a matrix is a matrix of the same size that yields an identity matrix when multiplied with its original. Computing the inverse of a matrix is tricky, but it helps to understand what an inverse matrix can bring to the table. For example, if our original matrix is a transformation matrix, its inverse implements the same rotation but in the opposite direction. Did you rotate the point clockwise around the origin by 45 degrees using rotation matrix R? Well, you can use R-1 (the inverse of R) to rotate them back to exactly where they were. Compared to the simplicity of its formulation, the actual derivation is a little complex and beyond the scope of this course. The complexity is mostly because the algebraic formula requires us to estimate the inverse of a matrix, which isn't easily explained or done by hand if you are not familiar with inverses and how to compute them. But let us at least understand what data it is operating on. Notice that we have two variables in the equation: X and Y. X is like our P above. It is a matrix of size n x m. Each row n is a data point, and each column m is a variable or dimension of the data points. Going back to medical expenses data sets, X has n=20 rows of data points (remember we used only a small subset of the original data for transparency into some of our computations on it) and m=6 columns of variables (aka. predictors or regressors): age, number of children, region, smoking habit, BMI, and sex assigned at birth. Y here is the output data. In this instance, we consider Y a matrix, not a vector. A vector is one-dimensional and has n values. To make the equation work in python using NumPy, we do have to transform that vector into a matrix of n x 1. It is the same data; the only difference is that it has a matrix format of n x m, even if m =1.

#h3 14.31 - OLS on the medical expense dataset

#pg We have introduced OLS as a more concise and optimized alternative to the statistical approach outlined in the previous section. Comparing Figure 14.17 with Figure 14.16 of the previous section, we can see both methods are indeed perfectly aligned in their outputs. At the very least, we should quickly verify both methods yield the same optimal parameters.

#im ../assets/figures/014/014-19.png 50 256 Figure 14.19 - Our regression model, computed using OLS. The mode is identical (as it should be) to the model found in Figure 14.18, based on fitting the data with the statistical approach to linear regression.

#h3 14.32 - More than just a concise algebraic solution

#pg There are a few surprising benefits to using the OLS implementation. First, we are not restricted to having only a single input or an outcome. We can make both X and Y as multidimensional as we want. In our examples up to this point, we have always used a single input variable (age) to predict a single outcome variable (amount of medical expenses). Using OLS, we can simultaneously map input data with multiple dimensions onto outputs with many dimensions. For example, we can include all the variables of our data set as input variables (sex, age, smoking habits, region, BMI, and the number of children). Furthermore, we can use multiple outcome variables (medical expenses, credit card score, foreclosure probability), link all the inputs to all the outputs, and see which combination of inputs is most predictive of any of the outputs. Doing this is possible using the item-by-item statistical approach, but the amount of bookkeeping and keeping track of all combinations of variables and their associated statistics because too cumbersome and tedious very quickly. While at the same time, still only requiring that single formula expressed in Equation 14.15 to derive the optimal parameters of our regression model. Ordinary Least Squares has many applications outside of machine learning. We can use it to perform dimensionality reduction, mapping a set of input variables onto a set of so-called basis functions (we will see this in the chapter on dimensionality reduction). It can be used for filtering or for finding transformations of points between spaces. We could even reformulate some more exotic machine learning mythologies like graphs into the language of ordinary least squares and obtain insights into their structure and dynamics. For a strictly linear method, that is quite impressive.

#h2 Gradient Descent

#h3 14.33 - Introduction to gradient descent

#pg Inching ever closer to a proper modern outline of a current machine learning algorithm, we finally arrive at what is perhaps the most fundamental technique used in all manner of machine learning models: gradient descent. Although our linear regression models could be estimated in closed form using a statistical or algebraic approach, any model that is marginally more complex (and therefore likely to be more interesting and useful) requires a new approach. We can no longer rely on estimating our parameters in a deterministic way. Instead, we will have to iterate, finding our way carefully through a complicated landscape of parameters and performance they offer. Framed in this way, machine learning equates to optimization: the process of adjusting the algorithm's parameters to minimize the difference between what we observe and what the algorithm predicts. To successfully optimize our parameters relative to the vast landscape of possible models and parameters, we will need to decide where to start, where to move next, how far to move, and when to stop moving. And the most important decision within this framework is where to move next: given our current parameters, our input data, and our output data, how do we change or update these parameters for them to become more optimal and yield better predictions across all the available inputs we have to our disposal?

#h3 14.34 - Getting all our ducks in a row

#pg There are a few prerequisites to start implementing gradient descent, all of which we have already discussed. Optimization means error minimization, which in turn forces us to quantify the related concepts of error and model performance. We formalized a machine learning model previously as function f, parameterized by weights theta. This function yields an optimal mapping of input data X to the output variable y. We further distinguished between the output of our model as the prediction (also sometimes called the hypothesis h) and the difference between our prediction and the objective outcome associated with the current input as the error. The metric loss was introduced to capture the performance of our model, given the current parameters theta. During training, we aim to adjust parameters theta such that the loss is minimized. Within the context of linear regression, the way we compute loss is very similar to how we computed the Sum of Squares Error, or the amount of variance left unexplained by our machine learning model. In the next chapter, we will see that we will need to compute loss differently for other types of machine learning models (e.g., classifiers). These are all the ingredients we need to implement gradient descent.

#h3 14.35 - Loss Landscapes Revisited

#pg In the previous section, we introduced the idea of a loss landscape. In theory, we can create an N-dimensional space in which each dimension represents a parameter of our model. Each point in such a space thus presents a set of parameter values that specify our model. At the same time, a particular set of parameters is associated with a particular loss value. It is impractical, if not outright impossible, to plot its loss landscape for any real-world model. Not only do we have trouble visualizing anything with three or more dimensions, but it would also take a very long time to compute the loss on each possible set of parameters. No matter how you bin each parameter into discreet values (for example a = [0.0, 0.1, 0.2, 0.3, ... 1.0], any additional parameter increases the total number of unique parameter settings exponentially. Not only does it make it hard for us to visualize that, but more importantly, this exponential growth truly limits our ability to consider all possible parameter settings and pick the best one. Instead, we will have to search for it in the space by cleverly moving through it based on an educated guess of the best direction to find parameters that yield better results, lower loss, and thus higher performance of our model compared to the parameters we have so far. Gradient descent is all about this educated guess.

#im ../assets/figures/014/014-20.png 50 256 Figure 14.20 - Finding your way down the mountain isn't all that difficult if you have an unobscured view of your surroundings at a reasonable distance. However, in machine learning, peering too far in the distance means computing too many possible parameter variations to be practical or even possible.

#h3 14.36 - Descending the Scottish Highlands

#pg Before we break out the graphs and formulas, an analogy of what gradient descent is all about might introduce the concept in a more familiar scenario. Imagine you find yourself lost in the Scottish Highlands when a heavy fog sets in. Without the fog, you could easily find a promising direction down the highlands towards the road below. You would just do a 360 around your environment (note that, with perhaps a few exceptions, it is a reasonable assumption that finding your way down in the highlands will eventually lead you to a road). However, with the fog, you can't see more than about 10 feet in front of you. How do you decide which way to go in this scenario? The best strategy now is to use a very local estimate of the overall downward direction could be.

#im ../assets/figures/014/014-21.png 50 256 Figure 14.21 - Gradient descent uses local information to find a way down the mountain, acknowledging that the loss landscape some distance away is shrouded in computational 'mist'.

#br

#pg Rather than using distant cues to determine the way down, you estimate the local orientation of the surface you are standing on. Here's how you would do that. First, look to your immediate right and note how higher or lower it is relative to you. For argument's sake, imagine it is exactly one foot higher. Now, look to your left. This time you will find it to be slightly lower, by precisely one foot at well. From this, you can estimate you are currently at a roll angle with the surface. You can do the same by looking forward and behind you, finding that looking forward, the elevation is 2 feet higher, and behind you, it is 2 feet lower. Therefore, you are also at a pitch axis angle with the surface. You can construct the plane orthogonal to the surface you are standing on from these two directions. Keep in mind that this is only true for our current location. We can't extrapolate beyond our limited viewing distance. It is possible things might look very different somewhere further ahead. Based on this information, where do we decide to go next that we feel constitutes a descent? Well, it is lower to our left by a factor of 1 and lower behind us by a factor of 2. Therefore, the biggest descent we can realize is to move 1 unit to the left and two units backwards. In other words, the biggest gradient or slope points slightly to the left and a bit more behind us, so we decide to descend in that direction. Hence: gradient descent. When we commit to a direction and bravely move in it, we move a bit further down the mountain. However, when we move a certain amount in any direction, that direction may stop the optimal direction to pursue further. We will need to estimate the direction associated with the largest gradient and thus the biggest descent again after each step. In general, gradient descent relies on a certain smoothness of the underlying loss surface, and we can argue that the highlands are smooth at a relatively small scale. A meadow of gentle curvature usually does not abruptly transition into a vertical wall often (Nature offers some interesting counterexamples of this idea of smoothness, see below). But the highlands aren't exactly completely smooth surfaces either. If you look 3 feet in some direction, it might appear to be the best direction down the mountain. Still, once we get there, we can discover we were let astray: what looked like steady descent into the valley below was nothing more than a slight local indentation in the surface. For this reason, despite the smoothness, we will need to continue to take baby steps, first picking a direction to move towards, moving in that direction for a bit, and then re-evaluating whether that direction is still the right choice. When we replace our analogy of the Highlands with a surface created by the loss associated with different parameter settings, we refer to the slope at a particular point on the surface as the gradient

#im ../assets/figures/014/014-22.png 50 256 Figure 14.22 - If the gradient is positive in every direction, indicating that the loss will increase no matter where we go, we might have found the best possible solution. It is also possible we find ourselves in a local optimum.

#h3 14.37 - Getting stuck

#pg The highlands analogy also reveals a potential pitfall in gradient descent: it can get stuck. You will appreciate the word pitfall, as it is a literal description of what can happen. Imagine we just stepped in the right direction, as indicated by the gradient. Of course, we will never retrace that step. Because from our current position, the location we came from will have a positive gradient associated with it, meaning we would have to ascend. Fine, we will just find another way forward and downward. But what is in every direction the gradient points up? In other words, what if we find ourselves at the bottom of a small convex indentation of the surface? It is problematic to the algorithm because one of two things could have happened. First, we arrived at the lowest point possible! We made it down the mountain! Pints all around! Or, in slightly more modest machine learning terms: we have found the optimal set of parameters for our model. There is no better solution; this is the best model we can theoretically get. However, it can also mean we are still very much up on the mountain but just settled into a small undulation where, from our perspective, the only way out is to go up. This is the problem of local minima in gradient descent. It cannot be solved or corrected with some magic bit of mathematics. The effects can be mitigated to some extent, however. Two approaches are typically used and combined. First, we adjust the amount of training data used on each iteration to update the weights. Using fewer data points to estimate the gradient makes the gradient less reliable. Our path through the loss landscape becomes more erratic, with bigger jumps between parameter values, although not always necessarily in the right direction. Such erratic behavior can prevent an algorithm from settling into a local minimum, as there is enough noise left in our gradient estimates that we might occasionally find ourselves jumping out of our local minimum. The downside, of course, is that it is hard to tell when we should consider not being so erratic because we may have found our global minimum already and just keep jumping out of it and back in again, like a machine learning version of Groundhog Day. This is addressed by the second approach, which changes how the algorithm makes updates over time. Based on the physics of how materials can end up with different properties based on the way they were cooled down from a hot state (like iron casting), these methods tweak the gradient descent algorithm to be more erratic early on and have more stable and smaller updates during later phases of training. We will discuss the first methodology later in this chapter when we contrast gradient descent, batch gradient descent, and stochastic gradient descent. Discussing the different training regimes will have to wait until we start working with deep belief neural networks in Chapter 22 and onward.

#h3 14.38 - Computing the gradient

#pg Translating this back to machine learning and space consisting of parameters and associated losses, rather than rolling hills and men in kilts, looking left and right means considering a slightly higher or slightly lower setting for one parameter while looking forward and behind us represents a potential increase or decrease in another parameter. In mathematics, the gradient associated with one parameter is said to be the gradient with respect to it, commonly abbreviated as wrt. In the case of 2 parameters, this gradient is with respect to both parameters. Think of it as an angle point along the direction of both parameters differently. Changing only one of them may lower the loss for our current set of parameters but changing the other has little to no effect. Or perhaps increasing parameter a while at the same time decreasing parameter b will yield the biggest decrease in loss. A gradient thus has a direction along the axis of the space formed by our parameters. For our two-parameter model, this direction can be expressed by an angle, which in turn can be decomposed to the part of the gradient that is with respect to one axis (say the axes of a parameter a and the part of the gradient that is with respect to the other axes (parameter b). In addition, each direction also has an amplitude associated with it. This amplitude conveys the amount of descent we accomplish by moving along each axis of our parameter space. Sometimes, changing one parameter causes a much more significant decrease in loss. Sometimes, a change in the other parameter will do the trick. And in some cases, no matter where we look, no direction yields a lower loss. We find ourselves at the bottom of a valley in the latter scenario. If the parameters we found there meet our criteria for a successful model, we can stop the process of trying to update our weights. But we may be stuck in some local, swallow minimum. All around us, the slope appears upward, but only as far as we can see in that direction. Beyond it are deeper valleys associated with more optimal parameter solutions. We should consider getting ourselves unstuck and continuing our search.

#h3 14.39 - Computing the gradient - 1D example

#pg Now that we have an intuition for using the gradient to guide us to a hopefully optimal set of parameters, we will need a way to compute it. It runs out, it isn't all that complicated, and we could just give you the formula and move on. Instead, we will take some time to try and convey exactly what is computed and why. And perhaps the best way to start is to consider trying to fit a single parameter: the slope of a linear function. Our function is as straightforward as they get: y = 2x + b. Using that function, we create some synthetic data completely noise free. The slope of 2 is the parameter we would like to estimate using gradient descent, given that we only have our input data x and output data y. We use gradient descent simply to figure out the best fitting function that transforms x into y, which hopefully turns out to be the function that generated it, which uses a slope of 2. Using the equations established earlier, we can compute the loss for a range of values that could be our slope. In our example, we set that range between 0 and 4. In this example, we map the entire loss landscape. When plotted in its entirety, it seems obvious what our parameter should be: the value of a at the lowest loss, which is indeed 2. Again, this stops being a way forward for more complicated models where the loss landscape is large and more importantly very highly dimensional. It simply becomes impossible to compute it in its entirety. Instead, we will have to work locally and use a guided approach for moving through the loss landscape, aiming to find a low point that yields an optimal result. In our example, we initialize our guess for the slope a to 3.5. What we need to compute at the point in our loss landscape is the gradient, or derivative of our loss function, weighted with how much that gradient contributed to the error we observed between what we predicted with a parameter at that point (a = 3.5) and what the actual outcome was. We can rephrase these two things as 1) in what direction should I move, or should I add a little or subtract a little from a to do better? and 2) how far should I move in that direction? The latter is important because if our current estimate of a doesn't cause any error in our prediction, why even move away from it, despite where the gradient might point?

#im ../assets/figures/014/014-23.gif 50 256 Figure 14.23 - Animation of gradient descent fitting a 1-parameter model using a relatively small learning rate. Optimization is somewhat slow but stable, landing us nicely at the global minimum of our function where a=2.

#br

#pg The error is easily computed in linear regression: the difference between the predicted and actual outcome for each data point we use to compute our gradient descent update this time. The derivative isn't much harder to compute, although it takes a bit more math and insight. You might remember from your math classes that the derivative of a function at a particular value gives you the slope of your function at that value. Visually it is the straight line tangential to that function. Does that mean we should take our rulers out and draw it? Not really, because we know what kind of function we must deal with for our loss function. Looking back at the equations, you will find that the loss is computed from our inputs through squaring. That makes our loss function quadratic with respect to our input data. And what is the derivative of a quadratic function? That's right, a linear one. Putting that all together means that the derivative of the loss function we need to compute the gradient is the input data x itself. Again, since the loss function is a quadratic mapping of the input data, the derivative of that loss function simply takes us back to that input data. Let us put all of that together in one iteration of updating a based on all values of x in our data. You can find this in Table 2 and Equation 14.23

#im ../assets/figures/014/014-24.gif 50 256 Figure 14.24 - The same as 14.24, but now with a larger learning rate. Note how we overshoot the global minimum on each iteration, although this overshoot eventually drops away, and the global minimum is reached.

#h3 14.40 - Computing the gradient - 2D example

#pg Let's try a model with two parameters to generalize our gradient descent to more than just a single parameter. From there, it should be clear how we can, in principle, keep adding more parameters as long as we have enough data. Figure 14.21 shows a two-parameter function consisting of a slope a and an offset b fitted using gradient descent. And again, the animation highlights how the learning rate affects our descent.

#h3 14.41 - Learning Rates and Convergence

#pg One key parameter we need to set prior to training is the learning rate. When we compute the gradients by combining our input data X with the errors they generate in predicting y, we adjust our parameters by a fraction of the computed gradients. We do this because we realize that the local gradient we compute is a very myopic estimate of what is happening at a large scale. We will get a direction we should move in, but this is only valid locally. A bit further out, the gradient might be very different. In other words, the gradient shows us the way but cannot tell us how far to go. Therefore, gradient descent uses small baby steps, updating the parameters by a fraction of the gradient. The parameter that determines this is called the learning rate and is typically denoted as alpha. If the learning rate is too small, it will take a long to converge. However, if the learning rate is too large, we might keep overshooting our target for some time and sometimes never converge. Selecting the correct learning rate is done by intuition and trial and error. As we already briefly outlined above, in more complex modern learning paradigms that train very complex deep belief neural networks, the learning rate is usually adaptative, changing over time based on how learning is progressing.

#h3 14.42 - The Gradient Descent Algorithm

#pg We have now defined all the necessary components of the gradient descent algorithm. What remains is to put them together. The following is Python code for conventional gradient descent, annotated with the corresponding equations it implements. Weights are updated on each iteration by considering the entire training data set. As we will see shortly, there are alternatives to this approach, with some benefits. We assume that the algorithm will receive as input training input data X\_train with m rows (examples) and n columns (variables/regressors), training ground truth output data y\_train (with m rows) and 1 column. Although y\_train is technically a vector, having it represented as a matrix of mx1 simplifies operating on it, especially when using other matrices. In addition, we will have a testing dataset consisting of X\_test and y\_test for objective assessment of the true objective loss for our model.

#cc //Import NumPy for all matrix operations

#cc import NumPy as np

#cc

#cc //If X\_train and X\_test do not have a constant column to //function //as a bias, add one.

#cc B\_train = np.ones((X\_train.shape[0],1))

#cc B\_test = np.ones((X\_train.shape[0],1))

#cc X\_train = np.hstack((X\_train,B\_train))

#cc X\_test = np.hstack((X\_test,B\_test)

#cc

#cc //Get and parameters

#cc no\_regressors = X\_train.shape[1]

#cc no\_training\_examples = X\_train.shape[0]

#cc learning\_rate = 0.001

#cc no\_iterations = 100

#cc loss\_history = []

#cc

#cc //Init Weights with small random weights theta. Again, making theta a nx1 matrix makes operating on it much easier.

#cc theta = np.random.random((no\_parameters,1)) \* 0.01

#cc

#cc //Loop through the maximum iterations

#cc for i in range(no\_iterations):

#cc

#cc //Compute the y\_pred using theta and X\_train

#cc y\_pred = np.dot(X\_train,theta)

#cc

#cc //Compute the error

#cc error = y\_train - y\_pred

#cc

#cc //Compute the loss

#cc loss = np.sum(error\\*\\*2) / no\_training\_examples

#cc

#cc //Combine gradient and error and normalize

#cc G = np.dot(X\_train.T, error) \* (-2/X\_train.shape[0])

#cc

#cc //Update the weights

#cc theta = theta - (G \* learning\_rate)

#cc

#cc //Store loss for plotting purposes

#cc loss\_history.append(loss)

#cc

#ca Code Snippet 14.3 - Gradient Descent in Python

#br

#pg And that is it. Things will get slightly more complicated as we move from regression to classification in the next chapter, but all the basic building blocks for training a modern machine learning model are there. And, as we will see later, most of this algorithm is still at the core of even the massive deep belief neural networks used today.

#h3 14.43 - Initial Parameters

#pg Hopefully, the initial parameters we set for our parameters all lead to the same global minimum, but there is no guarantee they will. Thinking through what good starting values are can help overcome these issues. Additionally, it is not a bad idea to train an algorithm multiple times with different initial parameters. If all yield similar parameters upon convergence, our trust in the algorithm grows substantially.

#h3 14.44 - Different Training Regimes

#pg Another critical decision that potentially impacts the parameters and the quality of the model after training is how we group different examples in our training data. At one extreme, we use all available training data to compute an aggregate gradient, which we already discussed and wrote pseudocode for. This has the benefit of being relatively stable, but since it is the average across many points, it tends to move the needle only by a little in terms of how much we update our parameters. In addition, for large data sets, computing the gradients for all data points becomes computationally expensive. Perhaps we can get away with using only a fraction of those points on each iteration?

#h3 14.45 - Stochastic Gradient Descent

#pg The other extreme, referred to as stochastic gradient descent or one-shot learning, is when we iterate over each data point individually, compute the gradient for that data point only and update the parameters accordingly. If there is any noise in the data, the updates to our parameters will look quite erratic, changing direction at each update. Such erratic behavior might seem harmful, but it does have one substantial benefit. Because of the erratic nature of the parameter updates, we explore a far bigger extent of the parameter space. This can prevent us from getting stuck in local minima, as we can jump out of them just by virtue of the noise in our data.

#h3 14.46 - Batch Gradient Descent

#pg Batch gradient descent is a good compromise, but determining the right balance requires intuition and testing. It is usually best to find some middle ground between the stability of using the entire data set or the exploratory nature of stochastic gradient descent. We can do this by grouping our data into batches, using only a percentage of the data on each iteration to update the parameters. It is traditional to call the iteration through a single batch of data an epoch.

#h3 14.47 - Hyperparameters

#pg All these settings described above require some tweaking for our data set. Our choice of the learning rate, batch size, and the number of epochs are collectively referred to as hyperparameters. Later we will learn additional hyperparameters, such as dropout and regularization constants, but the principle is the same. Modern machine learning platforms and solutions will typically train models many times over, using different values for the hyperparameters. By mapping out the space of different hyperparameter settings, we can find hyper-minima, locations where specific learning rates and batch sizes result in better-trained machine learning algorithms.

#h3 14.48 - Overfitting and Underfitting

#pg Linear regression models represent almost the most fundamental and most straightforward relationship between two variables: a linear one: the output y increases or decreases as a linear weighted fraction of the input x. However, noise in the data can still trip up a linear regression model. Most importantly, it can falsely assume that a particular noise pattern is a true signal, and the resulting function fitted on that data will reflect that. However, our model will fail when we apply that model to a new data set where that noise pattern isn't present. This is the classic case of overfitting: our model put too much faith in the data it was trained on. The way to circumvent this is by using training and testing data. We compute the loss for the training and testing data at each iteration. Initially, both will decrease as the model starts to capture the actual relationship between training and test data and all data. However, at some point, the model might start to overshoot that target and begin to fit the noise in the training data. At this point, our loss functions will diverge. For the training data, it will continue to decrease, making it seem our model is getting better and better. However, things get worse for our testing data: the model is now so tuned to the training data and its idiosyncratic patterns that it starts to fail on the testing data. It is always a good idea to closely examine the loss function for training and testing data. Typically, the point where the loss function starts to diverge is the natural stopping point: beyond that point, we start to harm our ability to generalize beyond our training data.

#h3 14.49 - Conclusions

#pg We covered a lot of ground in this chapter. Many machine learning ideas and methods are interwoven and borrow from other disciplines almost freely. Bringing all of that together, in one place, in a (hopefully) somewhat coherent story requires a fair share of words, paragraphs and pages. It also benefits greatly from you, the reader, trying out some of the methods we discussed. Working through the notebooks with the data provided, and perhaps with the data you find yourself, should give you hands-on experience and a good understanding of the fundamentals without necessarily needing the exact mathematical derivations and proofs. Many toolboxes purposely hide a lot of detail to help people quickly develop machine algorithms. But the more these algorithms are hidden from view but can be used with the click of a button, the less we eventually understand what is truly working and when it is truly not working. It is somewhat reminiscent of a famous piece of software designed specifically to allow users to run many statistical analyses on their data. If you wanted, you could select all tests and see which one gave you the desired result, despite most tests not being valid in the context of your data. There are some optimistic and pessimistic estimates of how many scientific experiments have used incorrect statistical methods as proof, rendering the result, even though peer-reviewed, essentially without any scientific merit. Suffice it to say, it is a number larger than we probably feel is ever acceptable.

#h3 14.50 - Demo Notebooks

#pg The notebooks are organized to follow closely the introduction of concepts throughout this chapter. As a result, there are quite a few of them.

#h3 14.51 – References

#bs

#be

#bp Gurnsey, R. (2017). Statistics for Research in Psychology: A Modern Approach Using Estimation. United States: SAGE Publications.

#bp Michalski, R. S., Banerji, R. B., Anderson, J. R. (1983). Machine Learning: An Artificial Intelligence Approach. Germany: Tioga Publishing Company.

#bp Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective. United Kingdom: MIT Press.

#bp Hespanha, J. P. (2018). Linear Systems Theory: Second Edition. United Kingdom: Princeton University Press.

#bp Provost, F., Fawcett, T. (2013). Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking. United States: O'Reilly Media.

#bp Rieke, F., Warland, D., Bialek, W., De Ruyter Van Steveninck, R. (1999). Spikes: Exploring the Neural Code. United Kingdom: MIT Press.

#bp Van der Plas, J. (2016). Python Data Science Handbook: Essential Tools for Working with Data. United States: O'Reilly Media.

#be