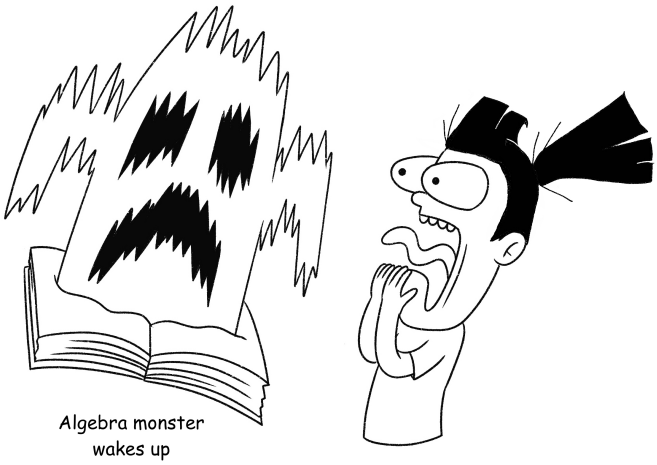
Chapter 8: Regression, Regularization, and Optimization

As we have explained in Chapter 1, *classification* algorithms, and *regressions* algorithms are two main categories of supervised learning algorithms. The focus of this chapter is on regression algorithms. Regression is used to understand how a variable's value changes when values of other corresponding variables change. Provost and Fawcett [Provost '13] stated *"classification is about whether something will happen, whereas regression predicts how much something will happen."* In another good book, James et al. [James '13] stated that classification can be used to predict categorical (qualitative) variables, but regression can be used to predict numerical (quantitative) variables. We can conclude regression is used for continuous data modeling and classification for discrete data modeling.

Nevertheless, the border is blurred, and a machine learning algorithm can belong to classification and regression groups. For example, in this chapter, we explain Logistic regression as a regression method, but it is a classification method as well.

In this chapter, first, we get familiar with the concept of 'cost', 'loss' or 'objective' function. Next, we describe five well-known regression algorithms including 'Linear Regression', 'Polynomial Regression', 'Piecewise Regression' as linear models, and 'Logistic Regression' and 'Softmax Regression' as non-linear models. Our discussion is followed by explaining ARIMA, a regression model for time series analysis. Afterward, we describe devils in model building, i.e., overfitting and underfitting, and then we switch to regression analysis and its algorithms. Next, we describe regularization methods, including Ridge, LASSO, ElasticNet, and Non-Negative Garrote regularization methods. Finally, we conclude this chapter with optimization and describe the very famous Gradient Descent and Newton methods. While learning about optimizations, we refer to some algebraic concepts we learned in high school, so brace yourself for the algebra.



Objective, Cost, and Loss

Everything we do in life is associated with a cost. For instance, you decide to learn data science and begin by reading this book. This decision comes with the cost of spending your precious time and expending energy from your brain cells. In other words, you invest time (a cost) to learn machine learning (a gain). You might have decided, or may decide, to get married; this decision costs you your free time, which will be dedicated to another person. Moreover, for a marriage to be successful, you or anyone else must make some changes to your behaviors. It's impossible to maintain the same behavior after marriage as when you were single.

Something similar exists in algorithms as well. A mathematical function that *tries to maximize or minimize a variable* (e.g., accuracy of an algorithm, energy use of an algorithm, etc.) is called an *objective function*. In the context of machine learning, we can call any function an objective function, if a model is trying to optimize this function. If the objective function seeks to minimize the cost of a variable (where the higher its value, the more its cost). We may refer to this objective function as the *cost function***.**

A *Loss score* is a score that defines the cost associated with a single prediction error and measure the cost of that single data point. In layman's terms, we can say loss functions measures how many mistakes we make. On the other hand, the cost function aggregates the losses over all data points, summing up the individual mistakes to measure the overall performance of the model.

Assuming an output of a model is presented with y, the differences between actual output values (y_{actual}) and predicted output values ($y_{predicted}$), specify the cost or loss. One single difference is the loss, and all loss values are constructing the cost.

In summary, while a loss function quantifies the error of a single prediction, a cost function encompasses the total error of the model across all data points. Although 'loss,' 'cost,' and 'objective' functions can sometimes be used interchangeably in the literature, a cost function includes the sum of loss scores across predictions, and both serve as types of objective functions that a model aims to minimize.

Epoch: A complete pass through the entire training set to calculate the cost function during the training phase is called an *epoch*. When calculating the cost function, each epoch involves using the data in the training set to compute the cost and use that cost to update the model's parameters. If a text mentions that a model was trained for 100 epochs, it means that the algorithm has completed 100 iterations over the training set, updating the model after each pass to reduce the cost function.

Linear Regressions

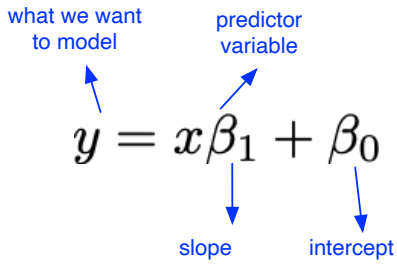
In Chapter 3, we have described that to find a relation between two variables, we use correlation analysis, which results in a single number to describe the relation. This single number is known as the correlation coefficient. This coefficient ranges from -1 to 1, where values closer to -1 or 1 indicate a strong relationship and a value of 0 indicates no relationship. However, sometimes there is a complicated relation among variables (like those relations between teenagers); in these cases, we use linear regression, which measures a change of one variable with respect to the other variable.

In summary, while correlation analysis assigns a score to describe the relationship between two variables, linear regression explains how changes in one variable are expected to affect the other variable. It's important to understand that correlation is about association, whereas regression is about prediction and causation (under certain assumptions)

In the following, we start from the simplest form of regression, which is univariate linear regression, then gradually increases the complexity of the regression models.

Univariate Linear Regression

Linear regression is the simplest linear model used for prediction. It predicts the *quantitative* variable y based on the single predictor variable x (or more than one single predictor variable). Y is also called a *response*, *dependent*, or *output* variable. X is called *predicator*, *input*, *explanatory* or *independent* variable. It is written and calculated with the following equation: $\hat{y} = \beta_0 + x\beta_1$, which we read as *regression y to x* or *y on x*. In the context of machine learning when we predict a variable we use a greek sign, called circumflex on top of that variable, e.g., $\hat{y}$ means a "predicted value of y".



Here, we have two model parameters that a simple linear regression model should identify their best optimal values, i.e., β_0 (intercept) and β_1 (slope). Note that they are not hyperparameters[1] because the model will use a cost function to identify them and are not configured by users of the algorithm. These types of parameters are called *model parameters* or called *model weights*.

Model parameters are parameters whose optimal value will be identified by the algorithm, via the cost function, and we should not give them to the algorithm (unlike hyperparameters, which the user sets). The optimization algorithm (by using cost function) tunes model parameters. Later in this chapter, we describe the optimization algorithm. It is also common to use θ, which is a vector that presents all model parameters.

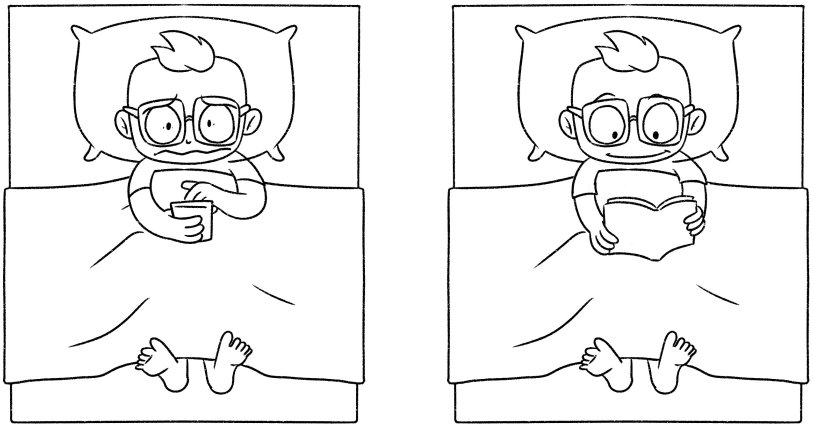
We can use linear regression or other linear models to perform prediction, describe a phenomenon, or even measure the effect size (see Chapter 3 to recall the effect size).

---

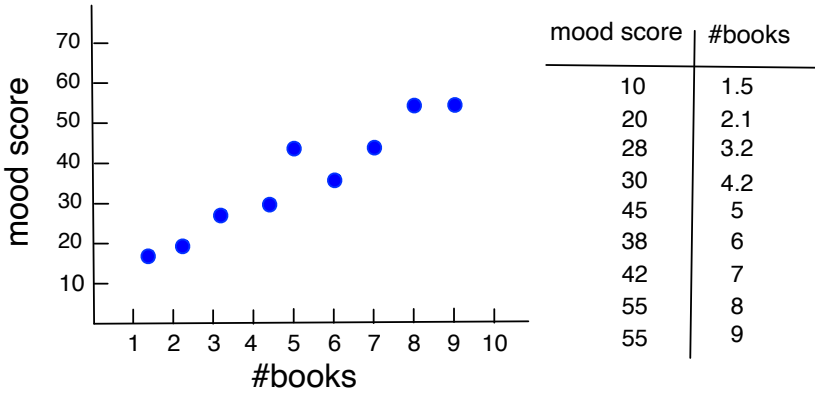
[1] Input parameters of an algorithm that need to be given by the user to the algorithm are called hyperparameters.

Let's look at an example scenario with linear regression. Recently, Mr. Nerd began to suffer from work stress. To solve his stress, he decided to make some changes in his life, and he started to read books before sleep. Previously, he has wasted his time on social media and fighting the holy war of useless discussions with trolls.

He speculates that reading the book could positively change his mood (the hypothesis). To experiment with this hypothesis, he has started to journal the number of books he read within his daily mood score. The result looks like the table on the right side of Figure 8-1. By plotting them he gets the plot on the left side of Figure 8-1.



It is clear that his mood is improving. Recently, he has finished the 9th book, he becomes curious to predict what his mood will be after the 20th book. He can use linear regression to answer his question, The question will be as follows: "what will be the mood score after reading 20 books?" A linear regression to predict his mood based on the number of read books can be written as follows: $mood - score \approx \#books \times \beta_1 + \beta_0$



mood score	#books
10	1.5
20	2.1
28	3.2
30	4.2
45	5
38	6
42	7
55	8
55	9

Figure 8-1: Mr. Nerd's mood score based on the number of books he read.

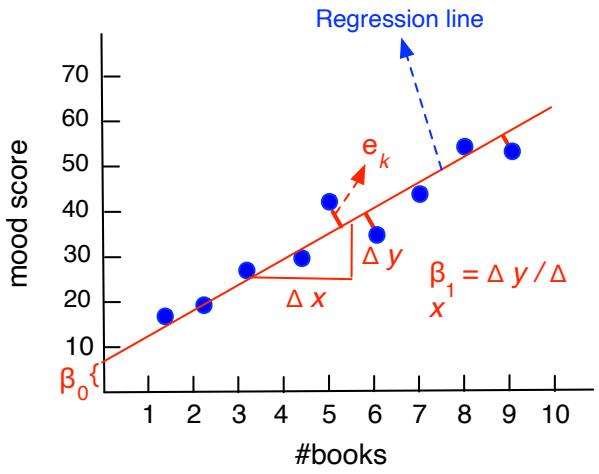


Figure 8-2: Figure 8-1 with regression line and its parameters.

The goal of linear regression is to find the optimal values for β_0 and β_1. In this example, the best value that a cost function of linear regression found is β_0 (intercept) = 7.369 and β_1(slope) = 5.58. Later in this section, we will describe how the cost function identifies these parameters. Figure 8-2 shows what we have in Figure 8-1 with a linear regression line and two optimal values for β_0 (intercept) and β_1(slope) parameters.

Therefore, assuming that there is no error, the algorithm can easily predict the mood score achieved by reading specific number books, via replacing these parameters in the linear equation. For example, it can calculate the mood score after reading 20 books, by follows:

$$mood - score_{(20th-book)} \approx 20 \times 7.369 + 5.58 = 152.96$$

Another application of regressions is to identify or describe a phenomenon in the dataset. Therefore, we can use regression to quantify the relation between input (predictor) and output variables. This will be reported as a hypothesis test and its *p*-values. Check Chapter 3 if you can't recall the use of *p*-value and hypothesis test. The hypothesis is written as follows:

H_0 = *There is no relation between input and output variables* ($\beta_1 = 0$).
H_1 = *There is a relation between input and output variables* ($\beta_1 \neq 0$).

For example in the book reading example, if the *p-value < 0.05,* we can infer there is a relation between the number of books he read and Mr. Nerd's mood score.

Assuming X (input dataset) is a $n \times m$ matrix. Linear regression will be presented as $n \times m$ matrix multiplication. The computational complexity of linear regression is close to $O(m^2 . n + m^3)$, which is calculated based on matrix decomposition and inversion. We don't go into the detail, just remember that a linear regression has *quadratic* complexity.

Model Parameters (Coefficients) Estimation

Now that we have answered his question about book reading's impact on mood score, we should also learn how the cost function identifies β_1(slope) and β_0 (intercept) parameters. In other

words, we use the cost function to fit a model to a training set. Different cost functions could be used to identify the best values for these two parameters (best coefficients to fit the model), but a basic and common one is *Residual Sum of Squares (RSS)* function. As we progress through this chapter, we explain more cost functions.

Figure 8-2 presents the described linear regression with its parameters highlighted. Check the red lines connecting each data point to the regression line in Figure 8-2, those e are called *errors* or *residuals*. RSS for n data points will be written as the sum of squared errors: $RSS = e_1^2 + e_2^2 + \ldots + e_n^2$ or $RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \ldots = \Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2$, we could summarized it as follows:

$$RSS = \Sigma_{i=1}^{n}(y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i))^2$$

The mean of x is presented as $\bar{x}$, and mean of y is presented as $\bar{y}$, the predicted values are presented with a hat sign "^", e.g., the predicted value of y is presented as $\hat{y}$. To calculate the $\hat{\beta}_0$ and $\hat{\beta}_1$ which minimize RSS, we can use the following equation.

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

Remember that our goal is to minimize change parameters to minimize the cost function, and the cost function here is RSS. The regression library wraps all details in a simple function call, but the mathematical notion of linear regression is fairly easy to learn, and it is worth understanding the rationale behind linear regression.

It is also worth mentioning, to write it accurately, we can write the linear regression equation as follows: $y = \beta_1 x + \beta_0 + e$, assuming that e is the mean of error values. Also, some use approximation sign ($\approx$) instead of the equal sign, because there is no guarantee that the created regression line is the perfect line, thus if you like to avoid generalization even in your mathematical writings and show off you are knowledge, write the linear regression as follows.

$$y \approx \beta_1 x + \beta_0 + e.$$

We can also write a linear model as a transpose of matrix x (it is a vector) times the vector of coefficients ($\hat{\beta}$), as follows: $y = x^T\hat{\beta}$ or $y = x^T\hat{\theta}$. In the near future (Chapter 10) we will write β_0 as b and other intercepts (βs) are vectors of weights w, i.e., $y = wx + b$. Most of the time, we do not write e in the equation.

Now that we understand how we have estimated model parameters (β_0 and β_1), let us repeat that model parameter estimation will be done through a cost function and not by the user of the algorithm.

Multiple Linear Regression

The linear regression we have explained is the simplest regression, which has one input or independent variable. Most of the time, we have multiple input (predictor) variables. In these cases, we can use multiple linear regression to predict the output, i.e., *multilinear regression*. The multiple linear regression equation is written as follows:

$$y = \beta_0 + x_1\beta_1 + x_2\beta_2 + \ldots + e$$

We can summarize the summation as follows:

$$y = \beta_0 + \Sigma_{k=1}^{n} x_k\beta_k + e$$

To better understand the use of multiple linear regression, let's use an example. Assume Mr. Nerd is still looking to find the symptom of his stress. Therefore, he has studied his life in a bit more detail, and he found some factors that are contributing to his daily stress, including work, being active of social media discussions, physical pain in his body, and family issues. To model this amount of information, we can use the following multiple linear regression:

$$Stress = \beta_0 + \beta_1 . work + \beta_2 . social\ media + \beta_3 . physcial\ pain + \beta_4 . family - issues + \ldots + e$$

Assuming we have two predictors, we can visualize it with a surface plot as shown in Figure 8-3. We neglect family issues for the sake of visualization because more than three predictors are hard to visualize in two-dimensional picture. Previously, we have described two questions that can be answered with simple linear regression, (i) predicting the output based on the given input and (ii) describing the relation between x and y.

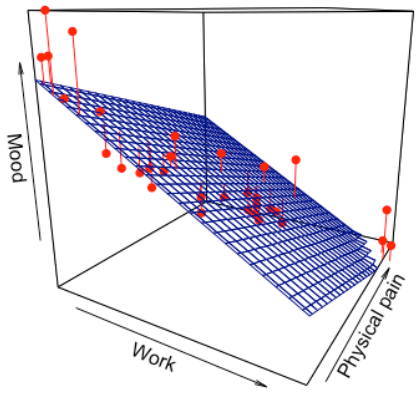
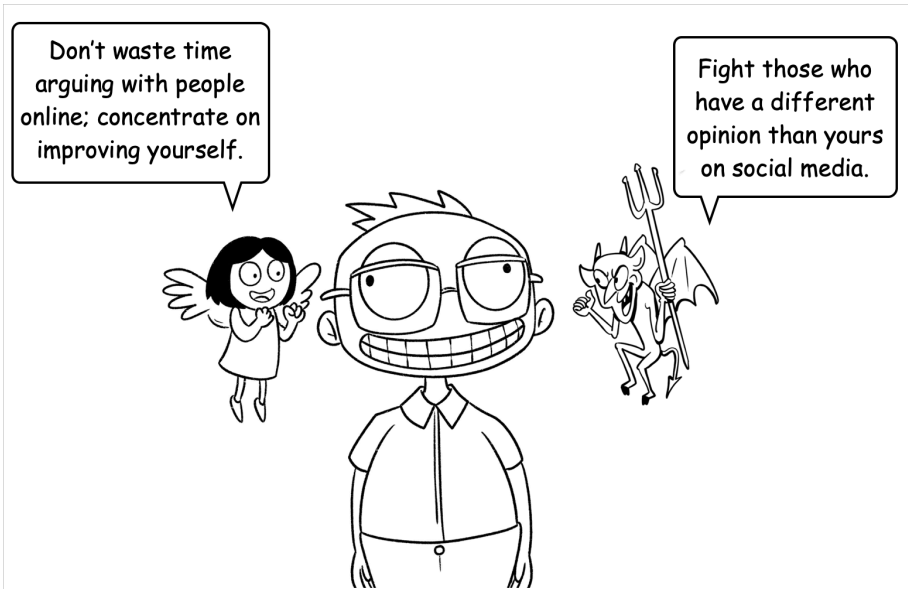


Figure 8-3: A multi linear regression example for Mr. Nerd and his mood score.

While employing multiple linear regression, usually, we need to answer some more questions [James '13]. For example, which one of the input variables x_1, x_2, … is useful for predicting the y? Do all input variables help to explain the output variable? How well does our model fit the data? To answer these questions, the software package that implements regression, provides a p-value or F-statistic test score for each input variable. Then, we can decide which input variables are useful and which are not useful.

If there is no relation between input variables, all their βs should be equal to zero. If there is a relation, at least one of the βs is not zero. Therefore, we can write the following hypothesis:

H_0 = *There is no relation between input and output variables* ($\beta_1 = \beta_2 = \ldots = \beta_n = 0$).
H_1 = *There is a relation between input and output variables (at least one* $\beta_x \neq 0$).



This hypothesis test will be performed by F-statistic test or F-test, which is written as follows:

$$F = \frac{(TSS - RSS)/p}{RSS/(n-p-1)}$$

In this equation $TSS = \Sigma_{i=1}^{n}(y_i - \bar{y})^2$, and $n - p - 1$ presents the degree of freedom, which in this case is the degree of freedom from error. The *f-value* > 1 means the null hypothesis is rejected (the alternate hypothesis is correct). The *f*-value ≤ 1 means the null hypothesis is correct, and thus the model has no predictive capability. It is recommended to check the output of regression and check *p*-value and *f*-value should be significant to accept the model. Note that the *p*-values and F-statistics are associated with the overall model and the individual coefficients, not directly with the input variables themselves.

Deciding About Model Variables?

The F-test is very useful for multiple linear regression. Assume the model did not pass the F-test and we have several input variables. Probably one or a few of them are responsible for the failure, and other ones are good to be used for building a model for multiple linear regression. Now a question arises: how can we identify which variables perform well in building the model, and which do not perform well?

For example, assume we build a model with three variables x_1, x_2, and x_3 as follows: $Y = \beta_1 . x_1 + \beta_2 . x_2 + \beta_3 . x_3 + \beta_0$. If the model fails with x_1, x_2, x_3 variables, we can remove x_1 and then test another model with x_2, x_3. Or we can remove x_2, x_3 and test another model with x_1, and so forth. In particular, for *m* numbers of variables we could have 2^m different models.

The best way to deal with model parameters is similar to the SBS and SFS which we have explained in Chapter 6, 'Wrapper Methods' section. Here, instead of features, we have model

variables, and SFS style of parameter selection is called *Forward Stepwise Selection*, and SBS style of parameter selection is called *Backward Stepwise Selection*.

In some cases, two more parameters of a model together have a very strong impact on the output variable. This phenomenon is called *synergy effect* or *interaction effect* [James '13]. For example, a popular celebrity (x_1) is advertising a product on popular social media (x_2). In this scenario, the popularity of these two variables boosts product sales, and we can extend our model to emphasize their importance by writing $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + e$ instead of $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + e$, because we want to emphasize the combination of x_1 and x_2. Usually, we need to experiment with both models (using interaction effect and not using interaction effect). Then, we can compare the *RSS* or other evaluation metrics of these two models and decide to use which one for the upcoming data (i.e., test dataset).

Linear Regression Challenges and Resolutions

There are some limitations in linear regression. We describe two of them that introduce new classes of regression. The first problem with linear regressions is that they can only predict numerical values (categorical values should be encoded). Sometimes we need to predict categorical data, e.g., if an email is spam or not spam, if the patient will die or survive the operation, etc. You might say we can map them to 0 and 1 and easily employ linear regression. That is not wrong, but what will happen if we can not draw a regression line between those binary states? The shape of the data points does not allow to draw the linear regression line. To handle this we use logistic regression, which we will explain shortly.

Another issue is what happens if we have more than one categorical predictor. For example, Mr. Nerd likes technology books, religious and science-fiction books. If we assign them numbers like *'1: technology'*, *'2: religion',* and *'3: science-fiction'.* There is an ordering enforced on book genres. It might lead to a mistake that the algorithm assumes the distance between 'technology' and 'science-fiction' books is larger than the distance between '*technology*' and '*religious*' books. As another example, assume we try to predict a medical condition of emergency room patients; it could be '*accident injury*', '*seizure*', or '*stroke*'. They are very different information, and considering them all together in a linear regression equation does not sound like a wise decision because we are imposing an ordering that does not exist. Therefore, we should look for a better method that considers input variables as binary and not continuous variables. This will be handled by *logistic regression*, which we will explain later.

The second problem is that linear regression is designed to handle linear relationships between features, and can not handle non-linear relations. In other words, the linear regression we described is useful for linear data with a fixed slope (i.e., additive slope). However, sometimes there is no straight line to be able to model the data points. A curved line can resolve this and it will be handled by *polynomial regression*.

Polynomial Regression

Sometimes, the relationship between the predictor and response variable is not linear. Assume Mr. Nerd enjoys reading books, but if he reads too few books or too many books in a time period, such as a month, his stress level increases. He likes to keep a balance on his book-reading behavior. Again, he logs the number of books he reads per month, and he plots them as shown in Figure 8-4 (a).

It is clear that we can not draw a straight line and model Mr. Nerd's stress level in this scenario. Even if we draw as we did in Figure 8-4 (b), this line is not descriptive enough, and it is too far from data points. We need a line that can model his stress level more accurately, like Figure 8-4 (c) to (e). For these scenarios a straight line (linear regression) is not enough, and instead, we can use polynomial regression.

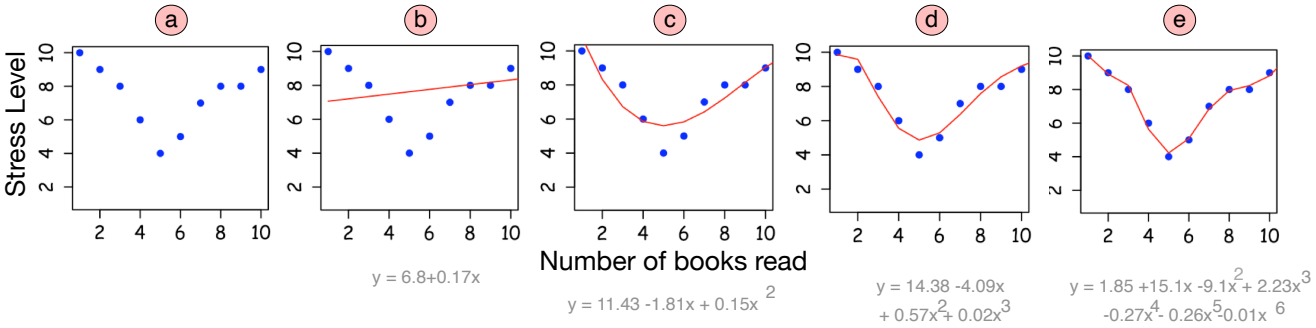


Figure 8-4: (a) plot of stress level and number of read books in a month, (b) Linear regression on the data, which is not properly representative of the data (underfitting). (c) Polynomial regression line with the degree of four, (d) Polynomial regression line with the degree of six, and (e) Polynomial regression line with the degree of six, which could cause overfitting. The equation of each plot is written at the bottom of each diagram in grey color.

A polynomial is a mathematical expression that consists of terms, where each term is formed by multiplying a coefficient (e.g., β) with a variable x raised to a non-negative integer power. These terms are then combined through addition or subtraction. Please keep this in your lovely brain: polynomial equation can produce a non-linear curve, and a polynomial regression can be formalized as follows:

$$y = \beta_0 + x\beta_1 + x^2\beta_2 + x^3\beta_3 + \ldots + \beta_d x^d + e$$

Note that here, we have only one x (input variable), but it has different powers. Having different powers means a curved line function, that can fit into the non-straight linear data points. In the above equation, d is called the degree of the polynomial, and *the larger the polynomial degree gets, the curve is getting more flexible*. In other words, x to a power larger than one gives us parabolic shapes (a sexy name for a curved or bell-shaped line), and thus, it can fit into the data with different shapes. x^2 is called quadratic, x^3 cubic, and x^4 quartic.

To better understand this phenomenon, consider Figure 8-4, which presents a regression model creation with different degrees. As we can see from this figure, the more we increase the degree

of polynomial, the more flexibility we have in the model to fit the sample data points (from Figure 8-4 (c) to Figure 8-4 (e)). Nevertheless, increasing it too much causes an unforgivable sin, i.e., overfitting, which we will explain later in this chapter.

Besides, note that increasing the degrees is associated with an increase in complexity. Assuming we have n data points and d degree, polynomial regression builds $((n+d)!)/(n! \times d!)$ features [Géron '17]. Therefore, if computational complexity is important, we should be careful and not increase the degree of polynomial regression too generously. For a polynomial of degree d and n data points, the time complexity of polynomial regression is $O(n d^2 + d^3)$.

Model Parameters (Degrees and Coefficients) Estimation

There are two types of parameters required to be identified in polynomial regression. The first is the regression "coefficients" ($\beta_0, \beta_1, \beta_2, \ldots, \beta_m$) and the second one is to identify the minimum number of "degrees" for the model ($x, x^2, x^3, \ldots, x^n$). The regression line should be the best representative of the observation (our data points) and this will be achieved through the optimal configuration of those two types of parameters. In the following, we describe each in more detail.

Coefficients

To get a good estimate for polynomial coefficients, we can use a cost function similar to the linear regression. A well-known cost function to estimate polynomial coefficients is the least square cost function or least square fitting. A polynomial regression can be written as a multiplication of matrices, and solving the equation can provide the values for βs.

For example, let's say we have a polynomial regression with degree m. y is the dependent variable and the βs are the coefficient for different nth powers of the independent variable x. By substituting the values for x and y (from our real dataset) and by solving the following m equations we can identify $\beta_0, \beta_1 \ldots \beta_m$ (we have learned in school matrix multiplication and how to resolve equations that have m unknown variables):

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^m \\ 1 & x_2 & x_2^2 & \ldots & x_2^m \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^m \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \cdots \\ \beta_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix}$$

Note that we have the values of x and y available from our observed dataset, thus, we can substitute them, then perform matrix multiplication and end up having three equations with m unknown variables, i.e., $\beta_0, \beta_1 \ldots, \beta_m$. By solving this equation, we get the values for coefficients. For example, let's assume that we end up with the following variables for *m=3*. $\beta_0 = 0.1, \beta_1 = 0.3, \beta_2 = 0.12$.

We intend to predict: what will be the mood of Mr. Nerd after reading his 20th book in a month? By using two degrees of polynomial regression we will have the following:

$y = \beta_0 + \beta_1 x + \beta_2 x^2$ and by substituting β variables we will have 54.1 as his mood score:
$y = 0.1 + 0.3 \times 20 + 0.12 \times 20^2 = 54.1$

Degrees

A polynomial regression with *n-1* degree is able to model *n* data points, but this is typically not desirable in practice. As we have explained before, increasing the number of degrees is associated with a huge computational cost. Therefore, we should try to find the least number of degrees. One way to identify a reasonable degree is to perform the following two steps:

(1) Leave out some data points from the dataset and then calculate some polynomial regressions with different degrees, e.g., with x^2, x^3, x^4 and x^5. There is a method called **'**leave-one-out' and one-by-one data points will be removed. However, this method is very resource-intensive, and it is recommended that it not be used.

(2) Add back those removed data points and check how much an error value (e.g. RSS) changes. For example, assume by using x^2, the error is 0.28, by using x^3 , the error is 0.23, by using x^4 , the error is 0.21 and by using x^5 the error is 0.46. We can conclude x^4 is the best choice, because it has the least amount of error while encountering new data.

This approach is very similar to the test and train that is being used in all supervised machine learning, but it is not exactly the same. In this case, we play with the training dataset and not the test dataset. Usually, the software that implements the polynomial regression library provides us with a parameter search function to automatize this process.

Piecewise, Segmented, or Non-Additive Regression

Despite the described limitations, linear regressions are very resource-efficient and are widely in use. Sometimes, instead of using polynomial regression, which is not as efficient as linear regression, we can use two or more linear regressions to get rid of the 'additive assumption' of linear regression. See Figure 8-5, which shows the amount of joy Mr. Nerd experiences from eating based on the calories of his food. He enjoys eating, but overeating reduces his joy as well.

The data points presented in Figure 8-5 are not additive because at around 110 calories, his joy starts to decline; before 110 calories, it was increasing. Therefore, we cannot model it with simple linear regression. Polynomial regression is also computationally complex, and it is recommended not to use it. However, by separating the dataset into two datasets, we can easily use two linear regression lines that fit the data appropriately. Figure 8-5 (b) presents the separator line, and Figure 8-5 (c) presents these two regression lines.

Sometimes, we can use more than one polynomial regression to model our dataset, and it is not just limited to linear regression. However, this doesn't seem a rational choice because when we don't care about computational complexity, we can simply use a polynomial regression.

Now, a question might arise: at what point do we need to separate the dataset? By visual inspection, we can recognize where in this small dataset, we can spot the distinctive point. For real-world datasets, which are usually too large and multidimensional, we do not have such a

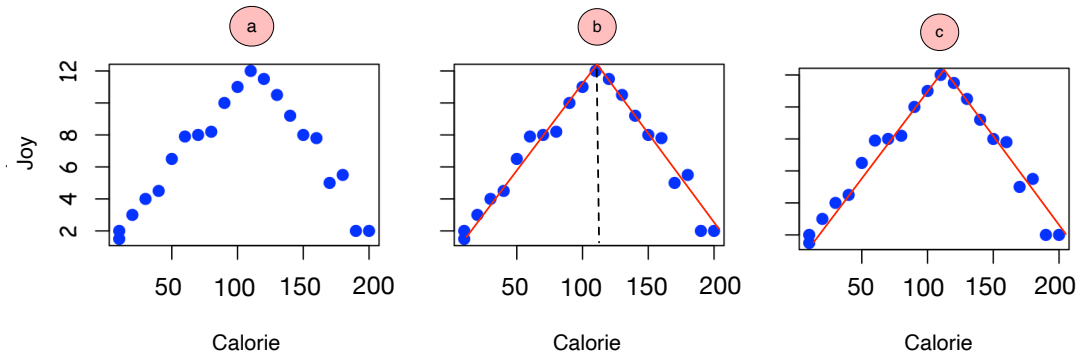


Figure 8-5: (a) A dataset that can are not additive, and thus, we can't use a simple line to fit them. (b) The knot is specified to separate the dataset into two sub-datasets. (c) The same dataset can be modeled with two linear regressions.

simple solution and can't visualize them. Therefore, we need a more subtle approach to design the distinctive border for a regression (the dotted line in Figure 8-5 (b)).

We can start by selecting a random data point and separating the dataset into two subsets from that random data point. Then, we draw a linear regression line for each subset and calculate R, SSE, F-statistics, or other evaluation metrics for each of these regression lines. We will explain these evaluation metrics later in this chapter. We change the data point, selecting a different one from the dataset, and determine various evaluation metrics. The point that yields the minimum sum of these metrics will serve as the optimal discrimination point, allowing us to divide the dataset into two subsets for which we can apply individual linear models.

For example, assuming we have a dataset and we randomly select three data points to separate it into two subsets.

$$d_1 : f\_statistics_1 + f\_statistics_2 = 3.1 + 2.5$$
$$d_2 : f\_statistics_1 + f\_statistics_2 = 3.2 + 2.1$$
$$d_3 : f\_statistics_1 + f\_statistics_2 = 3.1 + 2.6$$

Among the three following data points, i.e., d_1, d_2, d_3. The d_3 is the best data point for separation because it has the maximum F-statistics.

The point where one linear regression breaks and another starts is called a *knot*. We can also use the piecewise regression for polynomial regressions (See Figure 8-6). Assuming c is the knot, and based on the value of x_1 and c we should have two models. Therefore we write the following equation[2] for a polynomial regression with two variables x_1 and x_2.

[2] If you encounter the power inside parentheses, e.g., $x^{(k)}$, it is not x to the power of (k), it is read as kth index of x. Sometimes, the index is not written as a subscript, and if it is written as super-script, it should be inside parentheses.

$$y = \begin{cases} \beta_0^{(1)} + \beta_1^{(1)} x_1 + \beta_2^{(1)} x_2 & if \quad x_1 > c \\ \beta_0^{(2)} + \beta_1^{(2)} x_1 + \beta_2^{(2)} x_2 & if \quad x_1 \leq c \end{cases}$$

Each of these regression models in a piecewise regression is called a *spline*. We explained linear regression, but polynomial regression can have the same attribute too. For example, by looking at Figure 8-6, we see a single polynomial function can't model all the datasets properly. However, using a piecewise regression, we can use two polynomials to model the dataset properly.

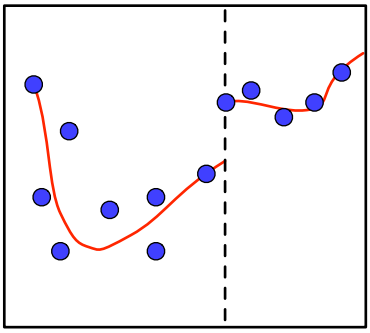


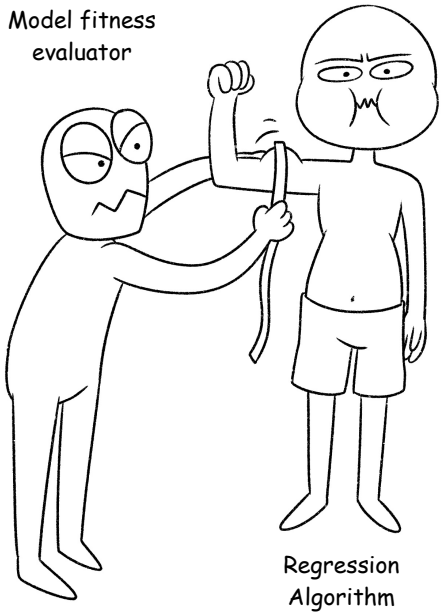
Figure 8-6: Piecewise polynomial regression example.

Although this approach seems flexible, it is useful when we have access to the entire dataset (population) and not just the sample dataset. Somehow, we are hacking a polynomial or linear regression to handle the dataset, and some mathematicians do not agree with this approach for different reasons, including giving biased regression coefficients that need shrinkage (we explain later); it yields a very high R^2 value that is badly biased, and so forth. Nevertheless, who cares? Since it works, feel free to use it. Just be careful not to use it in front of a picky mathematician.

Note in all examples we explained here, we assume there is only one knot, and thus, we separate the dataset into two subsets. There is no limitation on the number of knots and we can separate a dataset into more than two subsets with several knots [James '13].

Evaluating the fitness of training sets in linear models.

Once we have implemented our regression, we must (as with all machine learning algorithms) evaluate the accuracy of our model. Such an evaluation in regression algorithms is called *model evaluation*, a.k.a., measuring *model fitness*. In the this section, we describe common approaches used for evaluating linear models, including Residual Standard Error (RSE), Coefficient of Determination or R-squared (R^2), Root Mean Square Error (RMSE), and Mean Square Error (MSE).



These metrics are applied to both the *training set* to understand how well the model fits the data it was trained on, and the *test set* to assess how well the model generalizes to unseen data. Although the same metrics are used for both sets, the interpretation of these metrics can vary depending on whether we are evaluating model performance during training or testing.

Residual Standard Error (RSE)

RSE is an average number of data points that deviated from the regression line. It is called the *lack of fit* measures and is calculated as follows:

$$RSE = \sqrt{\frac{RSS}{n-p-1}} = \sqrt{\frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-p-1}}$$

Here, n is the number of data points, p is the number of model parameters (not including the intercept), and thus $n-p-1$ is the degree of freedom from error. If we have one input variable and one intercept, we can write the degree of freedom as $n-2$.

In fact, RSE measures the differences between y_i and $\hat{y}_i$ (y_i is the original data point and $\hat{y}_i$ is the predicted value, which is a data point located on the regression line). The closer two variablesy_i and $\hat{y}_i$ are, the better the model will be at the end. Thus, a smaller RSE is better. The *RSE* for Mr. Nerd's linear regression (Figure 8.1) is 5.074 on 7 degrees of freedom.

What is the degree of freedom in the context of regression analysis? The degrees of freedom, in this context is equal to the number of data points minus the number of parameters that will be estimated by the model. In the context of regression analysis, a parameter is estimated for every model's variable, and each parameter costs one degree of freedom. Therefore, including lots of variables in a regression model reduces the degrees of freedom available to estimate the parameters' variability. For example, if our sample size is 12 and our model has 3 parameters. The degree of freedom is $12-3=9$.

R^2

Another measure for model fitness is the *Coefficient of determination* or R^2 (R squared). RSE depends on the value of y strongly. Therefore, it is not clear what part of the linear regression equation contributes more to the RSE score. R^2 tries to mitigate this challenge by using the *total sum of squares (TSS)*, i.e., $\Sigma_{i=1}^{n}(y_i - \bar{y})^2$ divided by the *regression (or residual) sum of squares (RSS)*, i.e., $\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2$. Therefore, R^2 will be written as follows:

$$R^2 = 1 - \frac{RSS}{TSS} = 1 - \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{\Sigma_{i=1}^{n}(y_i - \bar{y})^2}$$

Recall that y is a dependent (output) variable, $\bar{y}$ is the mean of y, and $\hat{y}$ is the predicted value of y. The predicted value is presented with the hat symbol "^", e.g. $\hat{y}$.

R^2 returns a value between 0 and 1, which describes *how close the data points are to the regression line*. Higher values of R^2 indicate that our data points are closer to the regression line, and thus, the model is performing well. Mr. Nerd's R^2 is 0.9039 (based on data from Figure 8-1), and since it is near 1, which means that the R^2 evaluation shows high accuracy.

Root Mean Square Error (RMSE) and Mean Square Error (MSE)

For polynomial regressions, we can use RMSE to measure the accuracy. It is recommended to do the experiment for several different polynomial degrees and select the lowest RMSE. RMSE is the standard deviation of residuals or prediction errors. Check Figure 8-2 to recall what is residual. In short, residuals are the distance of data points to the regression line. Assuming $\hat{y}_i$ as a predicted value and y_i is the *i*th observed data points, the formula is written as follows:

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

When we encounter *Mean Square Error (MSE),* it is the same as RMSE, just its square root has been removed.

$$MSE = \frac{\Sigma_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}$$

These tests are also called *goodness-of-fit* tests, similar to the Chi-square that has been explained in Chapter 3, which is another test for goodness-of-fit.

NOTE:

* By increasing the number of labeled data or sample data for a regression algorithm, the chance of getting a wrong prediction decreases because we are increasing the likelihood of newly arrived data points being considered to be similar to already labeled data.
* Depending on the goal of our prediction, we might use more than one regression model or even algorithm to estimate the output variable. Therefore, do not hesitate to experiment with more than one model and then choose the best one. We will discuss this later in this chapter.
* Linear regression and polynomial regression are called *parametric methods*. Parametric methods assume that our prediction function has a form or shape. Non-parametric methods, such as Piecewise regression, do not make any assumption about the prediction function form or shape. The model that we choose might not correctly reflect the underlying dataset, which is the disadvantage of parametric methods. Nevertheless, since the non-parametric method does not have any assumptions about the model, we might end up using a large number of data points (train dataset) to create the correct non-parametric model, and it is hard to prepare a large number of labeled data points.
* Linear models, especially linear regression, are one of the most popular machine learning algorithms. Interestingly, in terms of quality, they are as good as non-linear models, which we will explain later. Since they are parametric, they do not need a large amount of data to train the model, and operating with a small dataset makes them very attractive algorithms.
* The simplest form of linear regression is Ordinary Least Squares (OLS), which estimates the relationship between X and Y by minimizing the sum of squares between real values (the blue dots in Figure 8-2) and predicted ones (the blue dots projections on the red line in Figure 8-2).

* Sometimes, to identify which model from a set of models can fit better into our dataset, we need to plot their residuals. Residuals are useful to identify the linearity of the model. After plotting residuals, we can check if the residuals are linear or not, and we can find a pattern in residuals. If there is a pattern in residuals, this is a sign that the dataset is non-linear. However, we described that linear regression is amazingly efficient. To enforce linearity in the data, we can apply a transformation on the data, e.g., log transformation or L_2 norm transformation. Check Chapter 6 to recall transformation. Another useful application of plotting residuals is plotting them based on time to ensure that errors are not correlated. Correlated residuals based on time is a phenomenon called *tracking* [James '13]. If there is tracking exists in our residuals, then we have no guarantee about the confidence of our model, and thus, we should think about another model. Tracking phenomena is common in time series analysis.
* Having two highly correlated variables negatively affects the accuracy of the multilinear regression and it is better to remove one of the highly correlated variables. Another recommended approach is to combine *collinear variables* (highly dependent variables) and create a new input variable, i.e., feature engineering, which we described in Chapter 6.
* Despite the attractiveness and flexibility of polynomial regression, it is prone to the overfitting problem, and therefore, it is not widely in use, unlike linear regression, which everybody loves. More about overfitting will be described later in this chapter; now, keep in mind that polynomial regression is prone to overfitting.
* Polynomial regression is more sensitive to outliers and noise in comparison to linear regression. It could be another reason that linear regression is usually favored over polynomial regression.
* Mathematicians are in a constant struggle to *estimate something that is non-linear by using linear tools*. It can be another motivation behind using linear regression significantly more than other regressions, especially among data scientists with a good understanding of math. Nevertheless, we should not forget that linear models assume the structure of the data will remain the same, which is not the case for real-world applications.
* Regression is a type of approximation. Approximation is the process of finding an efficient estimate of the value of the function at a certain unknown point by leveraging the information we have at a known point about the function.

AutoRegressive Integrated Moving Average (ARIMA)

ARIMA [Box '70] is a popular algorithm used to predict or model time series. Since we are talking about linear regression, it is good to describe this popular algorithm. If you are not interested in working with time series, feel free to skip this section. Before we describe the details of ARIMA we need to become familiar with two concepts, *extrapolation* and *interpolation*.

Extrapolation and Interpolation

Extrapolation is estimating the next change of a variable *beyond* the observation range, the value of a variable on the basis of its relationship with another variable.

For example, in Figure 8-7 (left) the red values are an extrapolation (predicted behavior) of the observed values (blue ones), which could be called forecasting time series as well.

Interpolation produces an estimate of potential values *between* observation data points, such as red dots that are generated between blue dots in Figure 8-7 (right).

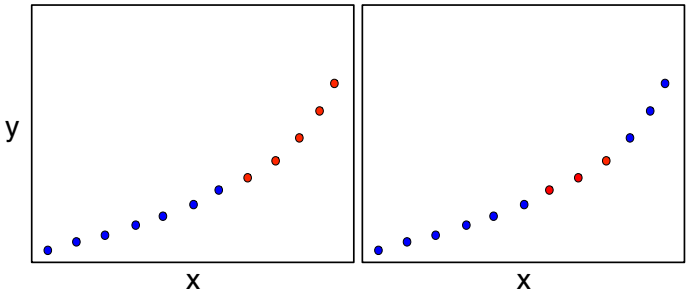


Figure 8-7: (left) A toy example of extrapolation. (right) A toy example of interpolation.

ARIMA Model

ARIMA is used for time series extrapolation (or time series forecasting) algorithm. It assumes that the future data points can be determined by using the past data points, or in other words, we can extrapolate future data points of time series based on the existing data points in the target time series. Specifically, ARIMA models the future data points as a linear combination of past data points (autoregressive part), past forecast errors (moving average part), and differences of past data (integrated part).

ARIMA receives three input parameters, p, d and q, which is written as $ARIMA(p, d, q)$. $ARIMA(1,1,0)$ means that $p=1$, $d=1$, and $q=0$. ARIMA model can be written as a linear equation to predict the next data point, $\hat{y}$ as follows:

$\hat{y}$= *constant + weighted sum of p number of lags + weight sum of q number of lag errors*.

In the following we explain the ARIMA algorithm steps and this equation in more detail.

(i) As the first step, ARIMA converts a non-stationary time series into a stationary time series, which is called *differencing* the time series. Differencing helps to stabilize the mean of the time series by removing changes in the level of a time series and thus de-trending it. The first column of Figure 8-8 presents the result of de-trending time series. This transformation is necessary because the prediction of future data points relies on the lags from a stationary time series model.

The differencing process is done by subtracting previous data (y_{t-1}) from the current data (y_t). However, just one previous data substitution is usually not enough, and the algorithm should subtract d data points ($y_{t-1}, y_{t-2}, \dots$) from the current data point. In other words, the parameter d specifies the *order of differencing* required to make the time series stationary. In other words, d specifies the minimum number of differences required for each data point to transform the non-stationary time series into the stationary time series. If the original time series is stationary, we have $d = 0$.

For example, considering y'_t is the differences between lags (the concept of lag is described in Chapter 6), and time t, we can write the following:

$d = 0: \quad y'_t = y_t$
$d = 1: \quad y'_t = y_t - y_{t-1}$ # first-order differencing
$d = 2: \quad y'_t = (y_t - y_{t-1}) - (y_{t-1} - y_{t-2})$ # second-order differencing

y'_t values are used to transform the original time series, which is not stationary into a stationary time series. The transformation through differencing is actually related to the *Integrated* component of ARIMA.

(ii) The parameter p specifies the order of autoregressive terms. It refers to the number of lags to be used as input (predictors) to predict the upcoming y'_t. A simple autoregressive (AR) model can be written as a linear model, y'_t depends on its p number of lags. Therefore, we can write the following equation for predicting y'_t based on pure autoregression:

$$\hat{y}'_t = \beta_0 + \beta_1 y'_{t-1} + \beta_2 y'_{t-2} \ldots + \beta_p y'_{t-p} + \epsilon_1$$

In this equation, βs are coefficients of their associated data points (lags) and β_0 is the intercept. Note that it is only AR and not the *moving average* (MA).

(iii) The parameter q specifies the order of *moving average (MA)* term. This means it refers to the number of lagged forecast error terms used to predict the current value of the series. The moving average part of the model is indeed a weighted sum of the past forecast errors. In other words, *moving average* is a weighted sum of q number of lags of the prediction errors. Therefore, assuming α is the intercept (β_0), ϵ is the error of the associated lag and ϕ is the coefficient of the error, a pure moving average equation (only MA and not AR) can be written with the following equation.

$$\hat{y}'_t = \alpha + \epsilon_t + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + \ldots + \phi_q \epsilon_{t-q}$$

ARIMA model combines both AR and MA models and creates the following equation to predict the upcoming data points in time series.

$$\hat{y}'_t = \alpha + \beta_1 y_{t-1} + \ldots + \beta_p y_{t-p} + \epsilon_t + \phi_1 \epsilon_{t-1} + \ldots + \phi_q \epsilon_{t-q}$$

ARIMA Parameters Estimation

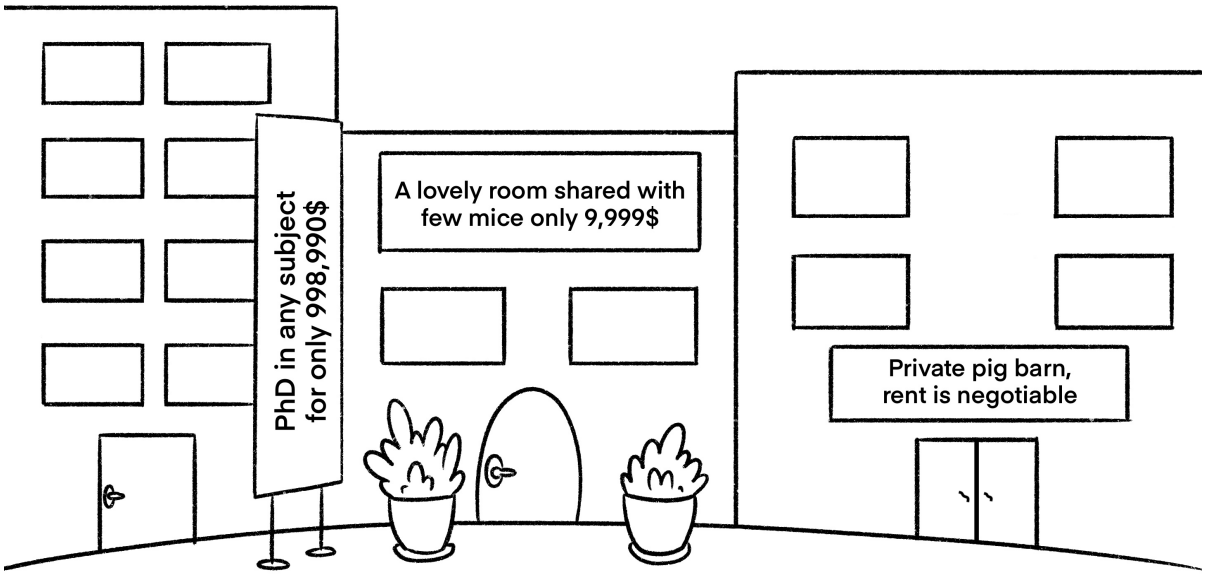
We understand the intuition behind the ARIMA equation, so now, we should explain how to choose the best values for d, p, and q. We have learned in Chapter 3 that correlation describes the relationship between two variables. In the context of time series and signals, when a correlation is calculated against lag variables (e.g., the correlation between d_t and d_{t-1}) it is called *autocorrelation* or *self-correlation*. In other words, autocorrelation means that the signal or time series is correlated with itself.

Let's use a sample; assume $X = \{1,2,4,5,6\}$ and $Y = \{2,3,5,7,7\}$. If we calculate the correlation coefficients of these two sets the Pearson correlation will be $r = 0.9834$, which means they are highly correlated. Let's say we would like to measure the correlation of X with itself, but with one lag. One shift in X will result in $X' = \{0,1,2,4,5\}$, and the correlation between X and X' will be 0.97, which means still, with one shift, X is autocorrelated. The more shifts we perform, the correlation coefficient decreases until it reaches zero. The existence of autocorrelation in errors (residuals) of a model is a sign of error.

We can identify the existence of autocorrelation by using a *correlogram* that stays for the *Auto Correlation Function plot* (ACF plot). ACF plot visualizes the auto-correlation (correlation between the original series and its lagged values). It presents how well the present data points correlate with lag data points (past data points). ACF plot allows us to determine the Auto Regressive coefficient for the ARIMA. In particular, the x-axis of the ACF plot represents the number of lags. The y-axis typically measures the autocorrelation coefficient, which ranges from -1 to 1, as we can see in the second column of Figure 8-8.

The lag value located outside the significant area (blue-shaded area) should be chosen because they are statistically significant. For example, only lag 1 is significant in 'autocorrelation of first-order differencing', the rest, which are located inside the blue area, are insignificant. Lag 0 is also located outside the blue area, despite showing a significant size.

To determine the stationarity of a time series, a statistical test called the Augmented Dickey-Fuller test or ADF test [Dickey '79] will be used. The p-value of this test determines if the time series is stationary. Its null hypothesis states that the time series is non-stationary. If p-value < 0.05, the null hypothesis is rejected, which means the time series is stationary.



ACF plot measures the correlation between observation (data) at the current time and previous observations (data); in other words, it is doing the MA part of the ARIMA. Another plot called the *Partial Autocorrelation Function plot (PACF plot)* finds a correlation of residuals. PACF plot finds a correlation of residuals. It is a bit challenging to understand, and we try to use an example to learn it.

Assume we are living in a city that has a business of selling academic degrees to international students at a huge price. It is a win-win game because academic corporations get richer, and international students who return to their countries are proud to get their degrees from well-

known universities. Let's call this imaginary city Bustom. Most of the Bustom population are students. Assume you have arrived later than the start of a new semester and you need to rent an apartment in November. November prices correlate to September (when students move to the city and the fall semester begins) and not October. December prices correlate with September prices, and not October. Therefore, PACF can resolve our need because it identifies the correlation between data at two different times (e.g. d_{t+1}, d_{t+2}) given both data are correlated to a data at other time (e.g., d_{t-1}). In other words, PACF can remove the correlation to data that is not relevant (e.g., d_t is not relevant). In our example, prices of November (d_{t+1}) and December (d_{t+2}) correlate to September (d_{t-1}) and not October (d_t).

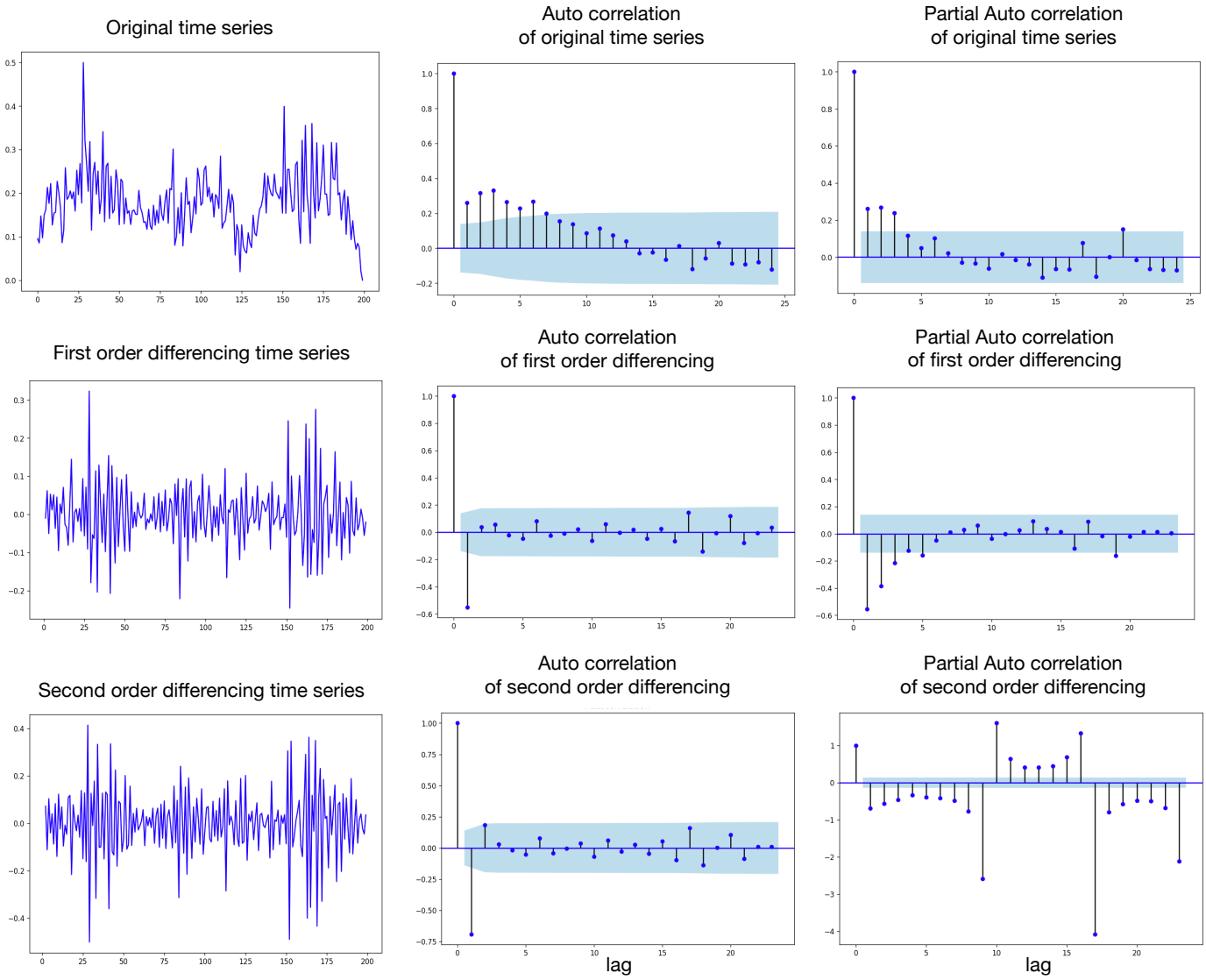


Figure 8-8: (Left) original time series, first-order differencing and second-order differencing (center) ACF plots of each time series. (Right) PCAF plot for each time series on the left side. The light blue color background is called the confidence band and tells us whether the correlation is statistically significant.

Check Figure 8-8, where we present ACF and PACF plots, along with the original and transformed time series.

Determining the best order of differencing (d)

There are a couple of rules to identify the best *d*. A good differencing is the minimum number of differencing that its ACF plot is fluctuating and decaying toward zero fast. In other words, a good order of differencing is often the order of differencing in which its standard deviation is lowest, and it indicates the time series is stationary. For example, by looking at Figure 8-8, we can see that the ACF plot for second-order differencing is moving toward zero, and compared to the original time series, its standard deviation is also closer to zero. Therefore, we choose the second-order of differencing. Keep in mind that if a time series is stationary (which is our desire), both ACF and PACF should move their tail toward zero (tail off to zero).

In the example presented in Figure 8-8, the second-order of differencing moves toward zero (better than the first-order of differencing), thus we choose *d=2*. If both the first-order and the second-order are moving toward zero, we go for the smaller *d*. Usually, we do not need to have a *d* larger than 2.

Determining the best order of Auto Regression (p)

To determine a proper value for *d* we use the ADF test. To determine a proper value for *p*, we will use the PACF plot (AR part of ARIMA). We choose a *p* based on the significance test reported by the PACF plot. For example, if we realize that d_{t-1} is not passing the significance test (located inside the blue area) but d_{t-2} passes, we choose d_{t-2} to predict d_t. The left side of the Figure 8-8 presents PCAF of each time series on their right side. To determine *p,* we use *PACF* plot and choose lag values that have its value is statistically significant, i.e. fall outside the blue area. The first-order PCAF has some lag values statistically significant. We go for the smallest one and say *p=1* is the proper value for *p*. Lags 1, 2, 3, 4, and 19 are significant in the "partial autocorrelation of first-order time series" plot, but we go for the smallest value of lag and assign *p=1*.

Determining the best order of Moving Average (q)

The same approach we have used to determine *p* from PACF could be used to determine *q*, but instead of PACF we rely on ACF. Therefore, based on our data presented in Figure 8-8, for both first-order differencing and second-order differencing lag number 1 falls outside the blue area in ACF plots, and this shows statistically significance. However, we choose our *q* from first-order differencing, because the literature recommends staying with one order differencing that has been chosen for *p,* and in our case *p* has been chosen from first-order differencing. Therefore, we can finalize our ARIMA parameters by writing *ARIMA (p=1, d=2, q=1)*.

If these parameter configurations sound hard to rationalize, we can go for cross-fold validation as well. We can test different settings of a parameter and choose the one that has the highest accuracy. However, it is recommended to use this method instead of cross fold validation.

Logistic Regression (Classifier)

To understand logistic regression, we should get familiar with the Sigmoid function. The Sigmoid function is a function that gets numerical data as input and outputs numerical vectors in a S-shaped curve, as is shown in Figure 8-9 (c) with a red line. This function brings any given number close to zero or one. It is written as $\sigma(.)$, and its equation is as follows:

$$\sigma(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}.$$

For example, assume we have three inputs $x = 0, \quad x = 1, \quad x = 2, \quad x = 3$. We will have the following sigma for each variable:

$$\sigma(0) = \frac{1}{1 + e^0} = 0.5,\ \sigma(1) = 0.73,\ \sigma(2) = 0.88,\ \sigma(3) = 0.95.$$

We are lazy, and thus, we do not show more examples here, but we recommend you use more values for x and plot them yourself. Then, you will notice that this function returns something between 0 and 1, mostly either close to zero or close to one. Its shape is similar to an S-shaped curve, as shown in Figure 8-9 (c).

Logistic (or Logit) regression (Classification) is used when our output (dependent) variable (y) is binary (e.g., yes/no, open/close, die/survive, spam email/not spam email), as opposed to linear regression which its output has a numeric range. Through a linear regression as input of a sigmoid function, we get a logistic regression. Therefore, we can formalize logistic regression as:

$$\hat{y} = \sigma(\beta_0 + x\beta_1)$$

Logistic regression is not used for predicting or describing a continuous variable. Instead, it predicts or describes a binary variable. It assigns a probability to each class output instead of directly assigning a value to y. In other words, it specifies the probability if a data point belongs to a class or not, and thus, is used to solve classification problems.

With some mathematical calculations, which we skip describing its details here, we can derive the following equation for logistic regression:

$$\hat{y} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} = \frac{e^{\beta_0 + \beta_1 x}}{1 + e^{\beta_0 + \beta_1 x}}$$

$\hat{y}$ presents what we want to predict (similar to $\hat{y}$, in linear regressions), and x is a set of the predictor variables. Linear regression is written as $y = \beta_1 x + \beta_0$, but here we call β_0 a *balance*, but in linear regression, it is called *intercept*.

Let's learn logistic regression with an example. Mr. Nerd is attending a course at the university, and due to a pandemics (Covid-19 started in 2019), the grade of the course is based on assignments and universities are closed to take a final exam. He either likes a course or doesn't like it (only two possible classes). If a course has more than ten assignments, it's very unlikely that Mr. Nerd will like it, and he doesn't subscribe to that course. Accordingly, we can formalize the Mr. Nerd course-taking procedure as follows: more than 10 assignments: don’t like, less than 10 assignments: like.

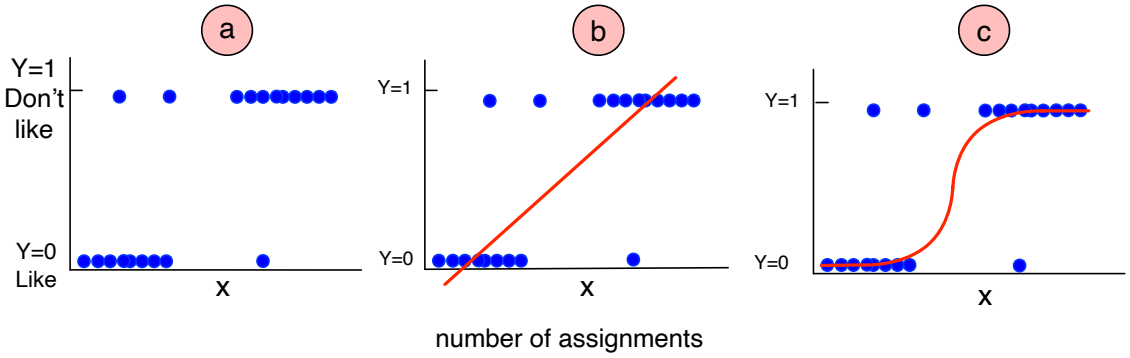


Figure 8-9: (a) Original dataset, x presents the number of pages each book has (b) linear regression plotted, (c) Logistic regression Sigmoid line plotted.

Let's assign like to 0 and don't like to 1. He has plotted his previous courses, based on like/don't like and the number of assignments he had, in Figure 8-9 (a). The *x-axis* (independent variable) represents the number of assignments, and on the *y*-axis we have two values for the independent variable, like (0) or don't like (1).

Figure 8-9 (b) presents a linear regression, which has been calculated and plotted based on given data points. We can see in Figure 8-9 (b) that the linear regression line is not a good fit for these data points. If we look at Figure 8-9 (a), we can intuitively guess that there is a correlation between data points. However, instead of fitting a straight line used in linear regression, we need a method to fit a regression line more accurately on data points, like the line Figure 8-9 (c). By using the Sigmoid function, as shown in Figure 8-9 (c) we can cover more data points (than a straight line, which is the result of a linear regression).

The output of the Logistic regression can classify input data into two categories by estimating the probability, e.g., classify any new course that is given to the algorithm as like it, or not like it, depending on the number of its assignments.

The output of our example will be provided as a probability value of *like* ($y = 0$) or *don't like* ($y = 1$), and the input variable is the number of pages.

As an example, to calculate output, we give Mr. Nerd's course preferences data to the Logistic regression algorithm, and it identifies model parameters as $\beta_0 = 0.4$ and $\beta_1 = 0.3$ coefficients. Now, we can calculate the estimated probability of Mr. Nerd liking a new course, which has eight assignments, as follows:

$$\hat{y} = \frac{e^{0.4+0.3\times 8}}{1 + e^{0.4+0.3\times 8}} = 0.94$$

It is more than 50%, and thus, we conclude he likes it. Also, the probability of not liking it is $1 - 0.94 = 0.06$. This seems a very easy example to identify, but in reality, logistic regression can solve more complex classification tasks.

Model Parameters Estimation

Similar to linear regression, the objective of logistic regression is to identify parameters that fit the sigmoid line. However, logistic regression is not a linear model, and thus, to identify its parameters, we should convert it into a linear model.

We know that the output of logistic regression is a probability value (between 0 and 1). However, the linear regression output does not have a range limitation. This flexibility in linear regression allows it to identify all linear model parameters easily. Therefore, if we make Logistic regression similar to linear regression, we can determine its model parameters.

To make Logistic regression similar to linear regression and be able to determine a value for each model parameter, the output variable is transformed from probability to *Logarithm of Odds (Logit)*. What is Odds[3]? In statistics, the likelihood of an event happening is called "odds" of that event. Assuming P is the probability, odds is calculated as follows:

$$odds = \frac{p(event)}{1 - p(event)} = \frac{probability\ \ of\ \ success}{probability\ \ of\ \ failure}$$

Odds are not probabilities, Odds provide a measure of the likelihood of a particular outcome. They are calculated as the ratio of an event happening, to that event not happening. For example, the probability of rolling a dice and getting the number 6 is $1/6$, its odds is $1/6 \div 5/6 = 1/5$, and 5/6 here presents the probability of not getting 6 (failure).

A *Logit* function, which is the *Logarithm of Odds (Log Odds)* is written as follows[4], for the sake of simplicity we write p instead of $p(event)$:

$$log(odds) = log(\frac{p}{1-p}) \text{ or } ln(\frac{p}{1-p}) \text{ for } 0 \leq p \leq 1$$

Model parameters of logistic regression are estimated by modeling the logit (the natural logarithm of the odds) as a linear combination of the input variables. This makes the output variable of logistic regression (probability value) similar to the output of linear regression (numerical value). Figure 8-10 (c) presents a logistic regression on the left, and by using the logit function, we transfer output probabilities into the Log Odds on the right side.

After using logit, the problem can be solved as a linear regression, but another problem is raising. As p approaches 1, the odds $1-p$ increase towards infinity, and hence the logit of the probability approaches $+\infty$. Conversely, as p approaches 0, the odds approach 0, and the logit of the probability approaches $-\infty$. This means that our data points are located at $-\infty$ and $+\infty$, and still, we cannot identify their coordinates.

Nevertheless, we can think of an imaginary linear regression line and project our data points on that line, the orange dotted line in Figure 8-10 (a) and (b). present that line. If we can project

---

[3] We described odds ratio in Chapter 3 for another use.

[4] $Log_e(x)$ and $ln(x)$ are equal and they both mean base e logarithm. If you are not familiar with logarithm check Chapter 6, The Magic Power of Transformation section.

our data points on this line, their coordinate will be in a range that linear regression can use to calculate all parameters (we call it θ to abbreviate all these parameters).

To solve this problem (range problem), logistic regression uses a Maximum Likelihood Estimation (MLE) **(**check Chapter 3) to project data into a small range. Here, the algorithm that implements MLE tries to identify parameter values that are closest to the output probability (0 or 1). In other words, the algorithm tries to find a value for its projected parameters $\hat{\beta}_0$, $\hat{\beta}_1$, …$\hat{\beta}_k$. Then it substitutes them to estimate $\hat{y}$, which yields a number close to 1 or close to 0. If you remember from Chapter 3, we are using L_n to denote the likelihood function, ℓ to denote the logarithm (log) of the likelihood function, and we calculate the maximum likelihood on log-likelihood. Assuming θ presents the parameters of our model, X our dataset, we have $\ell(\theta; X) = log\ L(\theta; X)$. Here. $L(\theta; X)$ represents the likelihood function, which quantifies the probability of observing the given data under a specific set of parameters denoted by θ. Now, we have $\ell(\beta_0, \beta_1, \ldots \beta_k)$, because here, the objective of this MLE is to find the optimal values for Logistic regression model parameters.

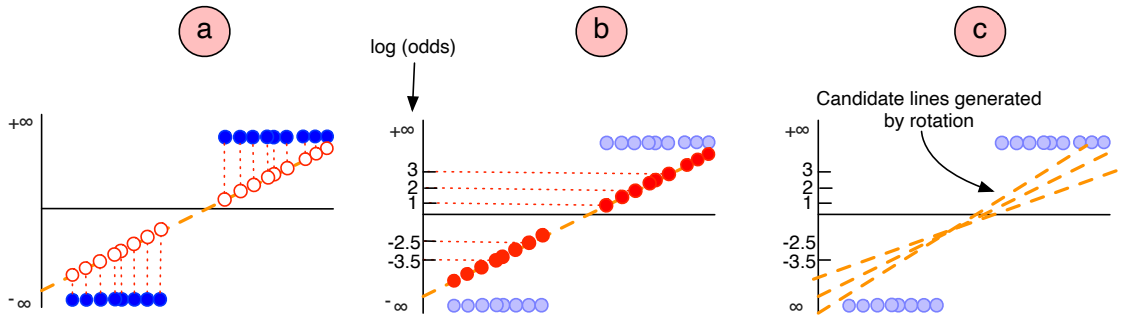


Figure 8-10: (a) Projecting original data points from negative and positive infinity to a candidate line. (b) Identify the Y-axis values of each project data point. (c) By rotation, different candidate lines are generated, and the line with the lowest error is chosen to identify the optimal model parameters.

Once again, let's review why we need MLE to project our data points on a line. Our data points are located at - ∞ and +∞. Therefore, their linear regression line will start from plus infinity to minus infinity, and thus, we cannot identify the model parameters of this linear regression. The MLE can resolve the infinity problem by projecting the original data points into an imaginary (candidate) line. See Figure 8-10 (a), the original data points are projected on a candidate line and presented as red empty dots. In Figure 8-10 (b), their coordinates to the y-axis have been calculated. These values on the Y-axis are *log(odds)* of the original data points[5].

For example, let's take a look at Figure 8-10. There is one data point with a value of -3.5, its projected probability (p) will be calculated as: $y = e^{-3.5}/(1 + e^{-3.5}) = 0.29$. This is the number that is used by MLE to sum all likelihoods together. Projected probabilities are between 0 and 1 classes. If the data point belongs to 0 classes (the probability is less than 0.5) its likelihood will

[5] We can transform these *log(odds)* back to probabilities by the following equation: $Y = e^{log(odds)}/(1 + e^{log(odds)})$.

be $1-p$. If it belongs to the 1 class (its probability is larger than 0.5), its likelihood will be the p. The likelihood of all data points on a candidate line is the product of all projected values, or the log-likelihood of all projected data points is the sum of projected data.

Keep in mind that it is better to convert a product into a sum, and the logarithm function converts the products into a sum, e.g., $log(xy) = log(x) + log(y)$. The maximum log-likelihood uses log-likelihood to turn products into sums, making the calculations easier. When optimizing, we look for the parameters that maximize the log-likelihood. By negating the result we mean that sometimes we minimize the negative log-likelihood instead of maximizing the log-likelihood.

For example, let's say MLE draws a random line at the first negative of maximum log-likelihood is 2.99. Next, the MLE rotates a bit this candidate line and then it creates a new candidate line. Afterwards, it calculates the log-likelihood for the new candidate line, e.g., 3.21, and continues this process with another rotation and thus another line, e.g., 2.19, and so forth. In the end, the candidate line with the highest likelihood (3.21 in our example) will be used as the best candidate line.

The best candidate line will be used for projecting data points onto this line, and similar to the linear regression, the algorithm can identify parameters from this line. This means that now we have a single linear line, the intercept (β_0) and slope (β_1) of this line will be reported as the best parameters for our logistic regression.

We remember earlier in the linear regression that to solve non-linear models scientists try to convert them into linear models and solve them. This is another similar scenario. We use MLE to transfer the Logistic regression into linear regression and identify optimal parameters for it.

Since it sounds complex to remember, lets summarize it as question and answer.

Question: How to estimate model parameters for logistic regression?

Answer:

1. Convert logistic regression to log(odds); using log(odds) makes a linear regression out of given logistic regression. However, now we have a problem with the range of the new linear regression. It has a range from $-\infty$ to $+\infty$.

2. The MLE resolves the range problem, but it reports a log-likelihood. Its result is negative because the logarithm of numbers between 0 and 1 is negative.

3. To formulate the maximization of the log-likelihood as a minimization problem, we use the negative log-likelihood. Therefore, the implementation of the MLE experiments with different linear regression lines (by using optimization) and choosing the one that has the highest negative likelihood.

Softmax Regression (Classifier)

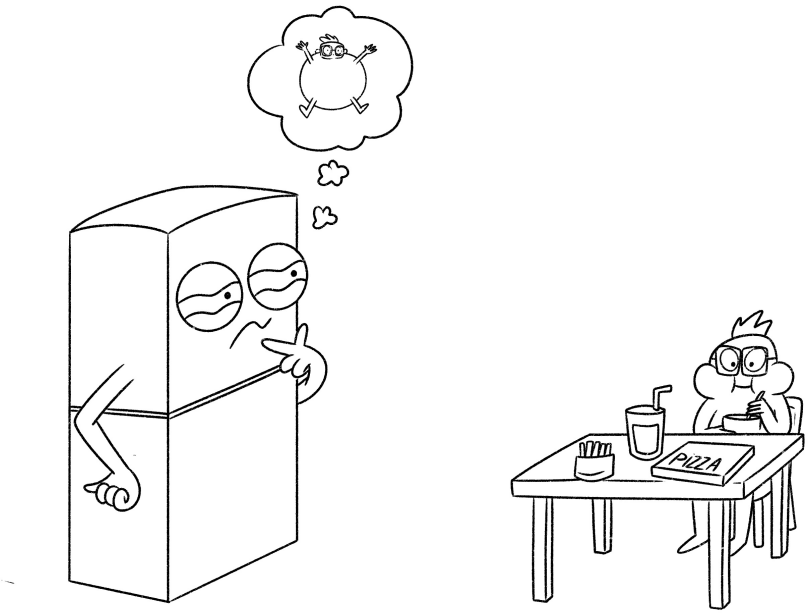
Similar to multiple linear regression and linear regression, sometimes we could have more than two possible outcomes, and the output is not just binary. It could be more than two variables,

e.g., type of animal (chicken, cat, dog, mouse, etc.). By logistic regression, we have one binary output variable (e.g., die/survive, like/dislike, true/false, etc.). To handle more than two outputs, we use multinomial logistic regression, multi-class logistic regression**,** or polytomous regression. Four known approaches are used for the multinomial logistic regression, including baseline logit model, adjacent category logit, proportional odds cumulative logit, and softmax regression.

Here, we describe *softmax regression* or *maximum entropy regression*, which is the most common multi-class logistic regression. In Chapter 10, we start to learn deep learning models, and will see that softmax is a common approach for multi-class classifiers by using neural networks, but we can use it without neural networks as well.

One easy way to deal with more than two dependent variables is to convert them into a single dependent variable and perform logistic regression. This means we convert a multinomial problem into many binomial classes, and solve binomial classes problem, with logistic regression. For example, in a mobile app that recognizes its users' transportation mode, one variable is "walk" the other variable is "not walk" (bike, public transport, and car). Our variables are "bike" and "not bike" (walk, public transport, and car), and this process continues for each of the possible categorical values. In the end, we have a set of probabilities, and the highest probability will be used to assign the class label to that particular input. However, it is recommended not to use logistic regression for non-binary classes, see Chapter 4.3.5 of [James '13].

Softmax regression is a generalization of logistic regression for K categories. Therefore, instead of two binary probabilities, we have K different probabilities. For example, the transportation mode could walk, bike, public transport & private vehicle ($K=4$), or a medical treatment type could be surgical, pharmaceuticals, diet, or none ($K=4$).



As another example, assume Mr. Nerd is developing an algorithm for a smart refrigerator he has purchased in Chapter 2. He is developing an image recognition algorithm that enables the refrigerator's camera to record the type and amount of foods he has consumed and calculate the body shape of Mr. Nerd in the future. The input of the refrigerator algorithm is a vector of foods Mr. Nerd puts them inside the refrigerator. The output is the future Mr. Nerd's body shape, i.e.,"gaining weight", "no changes" and "losing weight" (*K=3*). The softmax regression can create a model to predict Mr. Nerd's future body based on the three possible outputs.

The output variable of softmax regression is shown as $Y^{(i)} = \{1,2,...,K\}$ (in logistic regression, it was binary, i.e., $Y^{(i)} = \{0,1\}$). The softmax output is a set of probabilities (values between 0 and 1), and their sum is equal to 1. Unlike Sigmoid (S-shaped function) we can not use a specific shape or plot to visualize the softmax, which has more than two dimension space.

The softmax regression predicts one output class at a time, it is a multi-class classifier but not multi-output. Therefore, we cannot use it to predict more than one output variable, e.g., we can identify a chicken in a picture, but we cannot identify a chicken and a cat in the same image with one softmax.

To better understand the details of softmax regression, we should look at the logistic regression function, i.e., Sigmoid, and see how it can be generalized to be softmax. Let's rewrite the Sigmoid function, which is used for Logistic regression, as follows:

$$\sigma(z)_i = \frac{1}{1 + e^{-(z)}}$$

Here, $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_k x_k = \Sigma_{i=0}^{m} \beta_i x_i = \theta^T x$ and i presents its index. In many resources, $\sigma(z)_i$ is written as $h_\theta(x)$ or $\hat{P}$, which $h_\theta(x)$ stays for a hypothesis based on given input vector of x and θ presents parameters of the model.

We see a transpose sign used for θ, why? Because θ is a $1 \times n$ matrix (which is a vector) and x is $1 \times n$ matrix (which is again a vector). Therefore, to multiply these two matrices, we need to multiply $n \times 1$ matrix to $1 \times n$ matrix, and from matrix algebra, we knew that $(1 \times n)^T = (n \times 1)$.

We can write the softmax function, as a generalization of the sigmoid function. For all k classes of output it is written as follows:

$$P(y = j \mid x, \theta) = \frac{e^{\theta_j^T x}}{\sum_{i=1}^{k} e^{\theta_i^T x}}$$

Here, y is the vector we intend to predict, $P(y = j \mid x, \theta)$ is the probability that the outcome y is class j given the input features x , and θ is the model parameters. k is the number of classes. The equation is very similar to sigmoid (except having a Σ sign in its denominator). The softmax function produces a vector of probabilities, each element representing the probability that the input x belongs to one of the k classes. We can express its output as $\hat{P}_k(x)$. To better understand it, let's expand this equation as follows:

$$\hat{P}_k(x) = \begin{bmatrix} P(y=1\,|\,x,\theta) \\ P(y=2\,|\,x,\theta) \\ \cdots \\ P(y=k\,|\,x,\theta) \end{bmatrix} = \frac{1}{exp(\theta^{(1)T}x) + exp(\theta^{(2)T}x) + \ldots + exp(\theta^{(k)T}x)} \cdot \begin{bmatrix} exp(\theta^{(1)T}x) \\ exp(\theta^{(2)T}x) \\ \cdots \\ exp(\theta^{(k)T}x) \end{bmatrix}$$

Or some prefer to write it as follows:

$$\hat{P}_k(x) = \begin{bmatrix} P(y=1\,|\,x,\theta) \\ P(y=2\,|\,x,\theta) \\ \vdots \\ P(y=k\,|\,x,\theta) \end{bmatrix} = \frac{1}{\sum_{i=1}^{k} e^{\theta_i^T x}} \begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \\ \vdots \\ e^{\theta_k^T x} \end{bmatrix}$$

As the output result of softmax regression is a vector of probabilities, their sum is equal to 1. Therefore, while using a softmax function, *each output data point depends on the entire vector of input data points*. The input of the softmax function is a vector of raw scores (logits) for each class. For a given instance (data point), we have an input feature vector, and this vector represents the attributes or features of the instance relevant to the classification task.

Back to the refrigerator and Mr. Nerd's example, he went shopping and got the following items for its refrigerator: {broccoli, ice cream, chocolate bar, wheat bread, apple, orange}. His refrigerator, which is equipped with a softmax regression algorithm, can calculate the future body style of Mr. Nerd and returns the following result as a set of probabilities: {"gaining weight" = 0.4, "no changes"= 0.5, and "losing weight" = 0.1}.

Model Parameter Estimation

Similar to logistic regression, the cost function of softmax regression is the MLE. However, we cannot explain a multidimensional classification with visualization, because our brain is limited to 3D visualization only. Before explaining it, we should go back to the cost function of logistic regression, maximum log-likelihood, and formalize it in a mathematical notation.

Assuming $X = \{x_1, x_2, \ldots, x_n\}$ are input variables of the model, $\theta = \{\beta_0, \beta_1, \ldots \beta_k\}$ presents all model parameters, and $Y = \{0,1\}$ presents output classes for our input variables, the following equation presents the likelihood function for logistic regression, which we have described:

(i) $$L(\theta) = \prod_{i=1}^{n} [p(y_i = 1\,|\,x_i, \theta)]^{y_i} [1 - p(y_i = 1\,|\,x_i, \theta)]^{1-y_i}$$

The sign Π represents a set of products (multiple multiplications) similar to the Σ, which is used for summation. The goal is to find θ (or $\hat{\beta}$) that maximizes the $L(\theta)$ function.

We can extend the above equation and incorporate more than two Y values as follows:

(ii) $$L(\theta) = \prod_{n=1}^{N} P(y_n\,|\,x_n, \theta)$$

Equation (ii) is the cost function for softmax regression. We have explained that for the logistic regression, we are using the log-likelihood function, therefore, the described equation can be rewritten as follows:

$$\text{(iii) } J(\theta) = -\sum_{n=1}^{N} \log P(y_n \mid x_n, \theta)$$

Note that instead of $\ell(\theta)$ we use $J(\theta)$, which is common while using softmax refer to cost function as $J(\theta)$. We use the negative value for $J(\theta)$ as a function for model parameter estimation, which is called the *negative log-likelihood* or *cross-entropy loss*. We use the negative sign at the end to make the maximization problem as minimization and thus easier to compute. If you are good at mathematics, you can realize that the cost function, we used for softmax regression parameter estimation will be a generalization of the cost function that we used for logistic regression (i.e., maximum log-likelihood).

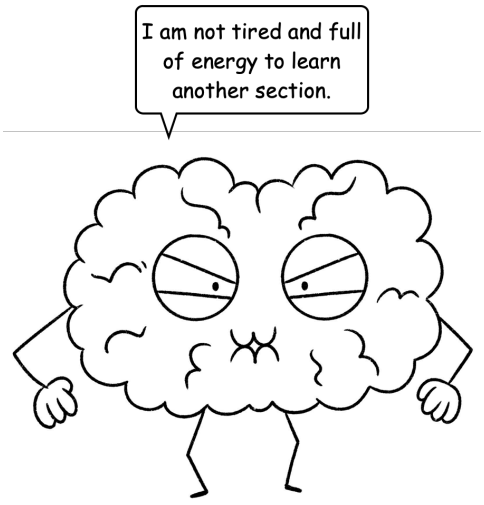
Since cross-entropy is a cost function used to measure the accuracy of the softmax regressions, our goal should be minimizing the cross-entropy loss. Assuming we have *M different classes (in logistic regression, we have only two)* and *N number of input samples,* the cross-entropy cost function is written as equation (iv).

$$\text{(iv) } J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\, log(p_{ij})$$

This equation is a generalization of equation (i) for more than two output variables. The cross-entropy objective function is to reduce the cross-entropy, thus a model that has smaller cross-entropy is favored over the model that has larger cross-entropy.

Evaluating Regression Models Fitness

The output of softmax regression is k number of distinct classes. The output of logistic regression is a binary classification result, which could be 0 or 1; unlike linear and polynomial regression, there is no traditional residual exists to compute the accuracy.



Logistic and softmax regression (classification) evaluation methods focus on assessing *how good is the model fits the data*. In this section, we list methods used to evaluate these regressions' results. Some of them are very important, not limited to regression methods, and will be used for other classification algorithms as well. Therefore, please tune your brain in full attention mode and learn them very carefully.

k-fold cross-validation

The most popular approach to evaluate classification results is the use of *k*-fold cross-validation (check Chapter 1 if you can't recall it). To implement k-fold cross-validation, we should make *k* subsets of the dataset, one set will be used as testing, and the other *k-1* sets will be used as training. We redo this process *k* times, and in each iteration, we change the testing set to another subset. Then, we evaluate the accuracy of each iteration. Next, we take an average of these accuracies and report the final accuracy.

The logistic regression has only two outputs for prediction and it could be written as the confusion matrix in Table 8.1. (left). The softmax regression has more than two predicted variables and its confusion matrix could be something like Table 8.1. (right), which has four possible outputs. We have explained confusion matrix back in Chapter 1.

		Predicted	
		0	1
Actual	0	12	2
	1	0	14

		Predicted			
		a	b	c	d
Actual	a	5	1	0	1
	b	1	7	2	3
	c	0	3	12	1
	d	2	2	1	6

Table 8-1: (Left) Confusion matrix example for binary logistic regression. (Right) A confusion matrix example for more multi-logistic regression, such as softmax, which has four possible output variables.

As we have explained in Chapter 1, some literature [Grus '19] and [Ng '18] recommend splitting datasets into three distinct sets including; 'training', 'validation', and 'test', instead of just two (training and test). The validation set is used for parameter tuning, feature selection, and more, facilitates the selection of the optimal model among various candidates or configurations. It's particularly crucial to ensure that the validation and test sets come from the same distribution to maintain the integrity of the model evaluation process.

Learning Curve

A fairly easy approach to determining the optimal model size is to study the error changes with different dataset sizes and experiment with the model. This is known as the learning curve. For example, Figure 8-11 presents a sample that by increasing the dataset size in both train and test sets, the error rate will converge at some point,

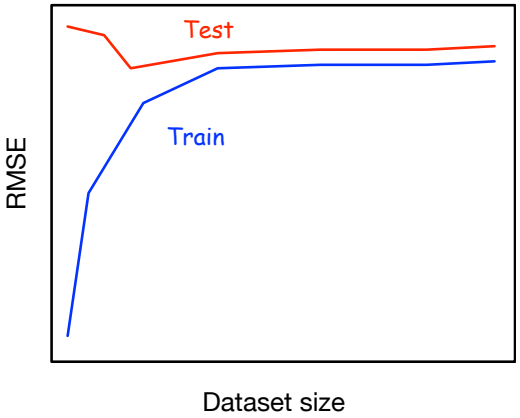


Figure 8-11: Learning curve example. By increasing the number of data objects, both training and test datasets converge. This makes the model reliable.

and the model will be stable. At the beginning, when there are few training instances (the left side of Figure 8-11) the model fits them very well, and thus, the RMSE is very low. Then, slowly, more training data arrives, and the models' RMSE increases until it gets into a constant value.

On the other hand, you can observe that for the test set at the beginning, the RMSE rate is high despite having a very low training error, but as we add more data, it decreases and reaches a steady point similar to the train set. Therefore, we can claim that the model is stable at that point because both errors converge and stay constant.

Observing a very low training error and very high test error rate at the beginning of the curve, in learning curve, present the overfitting. We will explain overfitting later in this chapter.

ROC Curve

Another widely used approach to evaluate the quality of a classifier is the *Receiver Operating Characteristic (ROC) curve*. This approach is not limited to the described regressions and can be used to measure the accuracy of any classification algorithm.

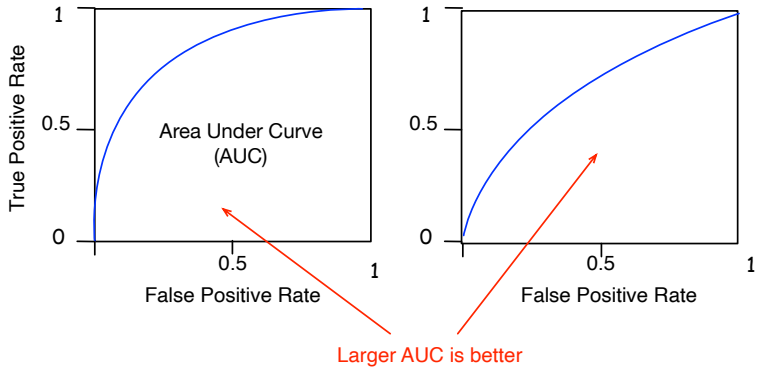


Figure 8-12: (left) an ROC curve which shows high accuracy, (right) ROC curve which has less accuracy in comparison to the left one.

The ROC curve plots True Positive Rate (TPR or sensitivity or recall) against the False Positive Rate (FPR or fall-out), at different threshold settings. Note that *FPR = 1 - TNR (True Negative Rate or Specificity)* as shown in the following:

$$FPR = \frac{FP}{FP + TN} = 1 - \frac{TN}{TN + FP}$$

ROC curve is useful for demonstrating the trade-off between TPR and FPR. The closer the curve is to the left and top borders, the more accurate is our results. The closer the curve comes toward the diagonal, the more accuracy decreases. Figure 8-12 presents two ROC curve samples, the left one is more accurate than the right one because the right one is closer to the diagonal.

The result of the ROC plot is read as the *Area Under the ROC* (AUROC) or some said Area Under Curve (AUC), which will be a value between 0 and 1. The higher AUROC (closer to the one) presents better model accuracy.

Pseudo R^2

Unlike linear regression, which typically uses the MSE to minimize the variance between the predicted and actual values and can employ methods like Ordinary Least Squares for parameter estimation, logistic and softmax regressions utilize a different approach. They are designed for classification rather than predicting continuous outcomes, and they estimate their parameters using the Maximum Likelihood Estimation (MLE) approach. This involves optimizing a cost function like cross-entropy (log loss) for logistic regression or a generalized form of it for softmax regression, focusing on maximizing the probability of the observed classifications given the model rather than minimizing variance.

To provide measurement metrics for logistic and softmax regression, we can use some *Pseudo* R^2 or *Adjusted* R^2 methods, such as *McFadden's* R^2 [Domencich '75]. McFadden's R^2 is computed as follows:

$$R^2_{McFadden} = 1 - \frac{ln\ L(\hat{\theta})}{ln\ L(\theta_0)}$$

$L(\hat{\theta})$ is the (maximized) likelihood value from the current model, $L(\theta_0)$ is the model with only an intercept (β_0) and no covariates (input variables). The high value of McFadden's Pseudo R^2 indicates a lower likelihood (the lower is better), because the ratio of $L(\hat{\theta})$ and $L(\theta_0)$ likelihoods should be close to 1. In particular, the higher Pseudo R^2 indicates a better model fit. Note that while Pseudo R^2 can theoretically range from 0 to 1, it will never reach 1 due to unpredictability in the data or model limitations.

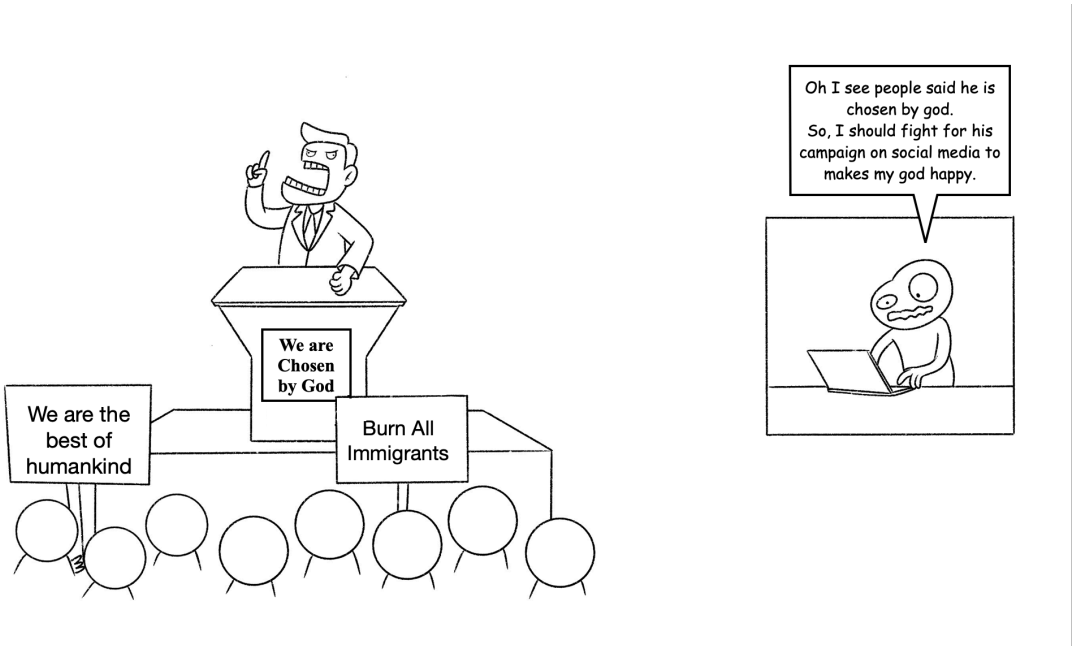
Remember, your machine learning library can compute these pseudo R^2, so there's no need to calculate them manually.

Wald Test

Wald test, also known as Wald Chi-square test [Wald '45], is a significance test that is used to assess the statistical significance of coefficients in regression models. Its null hypothesis states that model parameters do not have any effect on the model fitness. If the null hypothesis fails to get rejected (the null hypothesis is true), then removing those parameters from the model does not harm the fitness of the model. In other words, those parameters are not useful for the model. Conversely, rejecting the null hypothesis implies that the parameter significantly contributes to the model.

For example, assume a data scientist, Dr. Devil, is consulting a politician for election. He had a logistic regression that used three input variables (x_1, x_2, x_3) to predict the election result. x_1 input variable is presenting the welfare support, x_2 is using the cyber army of fake accounts and bots in social media to promote that candidate and damage the reputation of other candidates, x_3 is creating an imaginary enemy and fueling the xenophobia of that enemy in the public.

Based on previous consultations, Dr. Devil identified some parameters for these variables. He is creating a model that helps him to predict whether a candidate wins the election. He used a Wald test and realized that x_1 does not show a statistically significant result. Therefore, he can remove



it from his model and focus on the two other variables, i.e., creating fake accounts on social media and fueling the public's xenophobia of an imaginary enemy.

Information Criterion

Information criterion methods are used to compare a set of models together and identify the best model among others. There are several information criterion tests, but two of them are used more often than others, including the Akaike Information Criterion (AIC) [Alkaline '74] and the Bayesian Information Criterion (BIC), which is also called Schwarz Information Criterion (SIC) [Schwarz '78].

Assuming $log(L(\hat{\theta}))$ represents the log-likelihood of the current model, and k is the number of adjustable parameters in that model, the Akaike Information Criterion (AIC) is computed as follows:

$$AIC = -2(log(L(\hat{\theta}))) + 2k$$

Assuming we have n number of data points (sample size), it can also be used for linear regressions by using the following equation:

$$AIC = n\ log(\frac{RSS}{n}) + 2k$$

If you do not recall the RSS, check the description of linear regression earlier in this chapter. The output of AIC is a numerical score, and the lower AIC score is better. It means, that if we compare the two models' AIC scores, we should choose the model with the lowest AIC score.

Another method, the BIC or SIC, offers a distinct advantage over AIC by more severely penalizing the number of parameters in a model. While both AIC and BIC penalize model

complexity, BIC incorporates a larger penalty factor, which is particularly sensitive to the sample size. This factor is the logarithm of the number of data points, i.e., log(n), making BIC's penalty for adding parameters more substantial as the dataset grows. Consequently, BIC tends to favor simpler models compared to AIC, especially in large datasets.

One might ask, why do we penalize the number of parameters? Because a large number of parameters makes the model complex and thus inefficient (e.g., prone to overfitting, computationally more demanding).

Assuming n is the number of data points, k is the number of parameters, θ is a set of all model parameters, and $L(\hat{\theta})$ is the maximum likelihood of the model, BIC is computed as follows:

$$BIC = k\ log(n) - 2log(L(\hat{\theta}))$$

For linear types of regressions we use the following equation:

$$BIC = k\ log(n) + n\ .log(\frac{RSS}{n})$$

Similar to AIC, while comparing BIC scores of several models the lowest BIC score presents the best model.

Both BIC and AIC are based on the law of *Occam's Razor* principle or the law of briefness. It indicates that we should use only things that are necessary. Models which are obtaining this law are called 'parsimonious models', which have only parameters that are necessary for prediction, and they do not include unused parameters.

Likelihood Ratio Test

The Likelihood Ratio Test (LRT) or Likelihood Ratio Chi-square test is useful to choose the best model from two nested models. Nested models are models in which all parameters of one model exist in the other model as well. For example, we would like to model the applicants' acceptance rate in an elite university, where getting admission there is very competitive. One model has two parameters: the entrance exam score (e.g., SAT in the U.S.) and high school performance (GPA in the U.S.). The other model has five parameters: exam score, high school performance, race of the candidate, nationality of the candidate, and whether their parents are rich. The first model is nested within the second model because the second model has all the parameters of the first model and three more parameters (race, nationality, and parent-richness).

The Likelihood Ratio Test assumes that the best model is the one that maximizes the likelihood function. It uses a log-likelihood to compare two models together and identifies the one with a higher maximum likelihood function. It is written as a ratio between the model with a lower number of parameters $L(\theta_1)$ to the model with a higher number of parameters, $L(\theta_2)$, with the following equation:

$$\text{Likelihood Ratio Test} = -2log_e(\frac{L(\theta_1)}{L(\theta_2)})$$

Once again, for the hundred thousand times, we should repeat that your software package includes these tests and you don't need to implement them on your own. Unless you would like to implement specific customization on the existing method, or you are a student and your instructor asks you implement these algorithms on your own.

NOTES:

* When we encounter, $exp(x)$ it is similar to e^x, which is read as "e to the x" or "e to the power of x", and similar to π, e is a constant $e = 2.7182...$ This is called Euler constant number.
* While working with logistic regression, there could be a predictor variable that shows a negative coefficient (negative value for β). This means that there is a negative relation between that particular predictor (X) and output (Y).
* One known issue in logistic regression is the coefficient impact on two or more different input variables on each other. For example, if we increase x_1, then Y will be increased regardless of the value of x_2. To mitigate this issue and increase the interaction effect, we can define a third coefficient (β_3), which is called *interaction terms* between two other input variables.

  $Y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$
* Similar to linear regression, Logistic regression is a form of the generalized linear model. Generalized models are models which use a "link function" to link a nonlinear model to a linear model. Here, the link function is the log odds.

* A result obtained by one predictor (input variable) might be different from the result obtained by multiple predictors (input variables). This is caused by the correlation among predictors, which is called the *confounding effect*.
* We have referred to some weird terms that are -∞ and +∞. They are used in logistic and softmax regression. Nevertheless, since smoothing the data is a very common problem in machine learning, sometimes large data will be converted to +∞, which is called an *overflow* problem, and small data will be converted to zero which causes an *underflow* problem, e.g. diving by a zero will be -∞.
* We should not perform extrapolation beyond our observed data points, especially while using polynomial regression, which uses curved lines.

Devils of Model Building

Why do we use all these regression algorithms for prediction? Because by observing the past behavior (observed or training data), we would like to predict the future (unobserved or test data). The ability to match future data with observed data is called *generalization*. Therefore, an umbrella term that is used for regression algorithms is called *Generalized Linear Models (GLM)*. Note that GLM models do not necessarily assume a linear relationship between input and output variables. Instead, they posit a linear relationship between the expected value of the output variables, as transformed by a link function (e.g., Logit for logistic regression), and the input (explanatory) variables. This encompasses logistic and softmax regressions, among others. In simple terms, GLMs include a transformation of the output through the link function, rather than just modeling a linear relationship between inputs and outputs.

There are two important challenges associated with all GLM methods and also other supervised learning algorithms, underfitting and overfitting.

Overfitting and Underfitting

Overfitting refers to creating a model that performs very well on the data that we have observed (train dataset), but it performs very weakly on the newly arrived data (test dataset). In other words, overfitting makes a model too specific on the training dataset and causes poor generalization of any new data points (test dataset).

Underfitting refers to creating a model that can neither fit the train nor test data properly. In other words, underfitting refers to *too much generalization*. Therefore, the model will perform poorly on both our training and test datasets. Figure 8-13 shows three examples: an underfitted model, an overfitted model, and a proper model.

We can see in this figure that both underfitted and overfitted lines (in red) are not good representations of the data and are incapable of predicting or modeling the dataset. Overfitting is a very common mistake when we increase the degree of our polynomial regression. Usually, overfitting is a very common mistake in supervised learning.

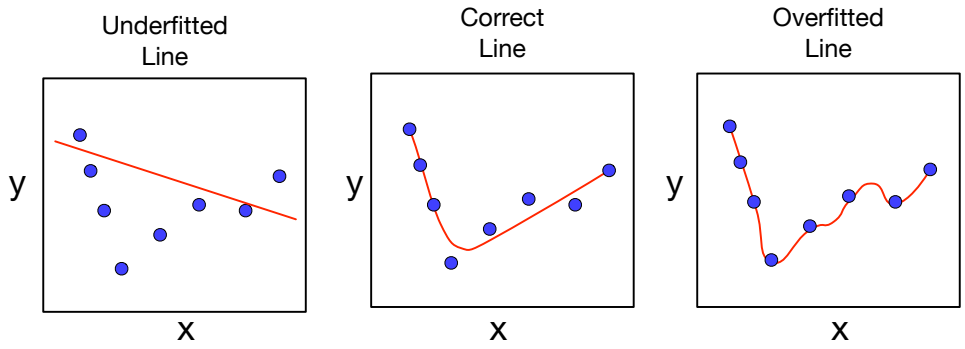
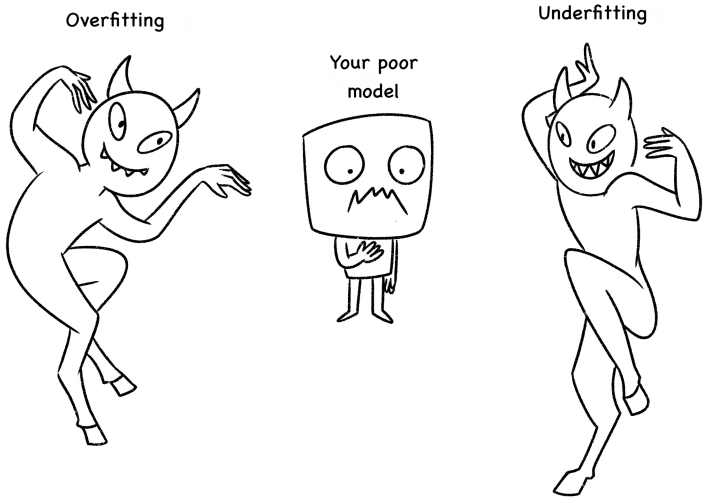


Figure 8-13: Example of underfitting and overfitting in model fitting.

Now, the question is how to avoid releasing the devils of overfitting or underfitting. Underfitting is easy to recognize because as soon as a model does not perform well both on the train and test dataset, we can find it. Nevertheless, overfitting is not that easy. To recognize overfitting, we can change the train and test dataset in every iteration, i.e., k-fold cross-validation. We hope you remember that by using the cross-validation we could change the train and test dataset. For example, we change the train and test dataset five times (5-fold cross-validation) and report the average error from those experiments. If the error is not changing significantly among each experiment, this is an indication that the model is not overfitted. On the other hand, if in one experiment we have a very good result and not in others, that particular experiment is overfitted, and thus, we should revise the model. We can summarize that any significant difference between testing and training is a sign of overfitting.



Bias-Variance Tradeoff

Overfitting is a very common error while working with supervised machine learning models. An appropriate approach to deal with overfitting is to decompose it into bias and variance problems

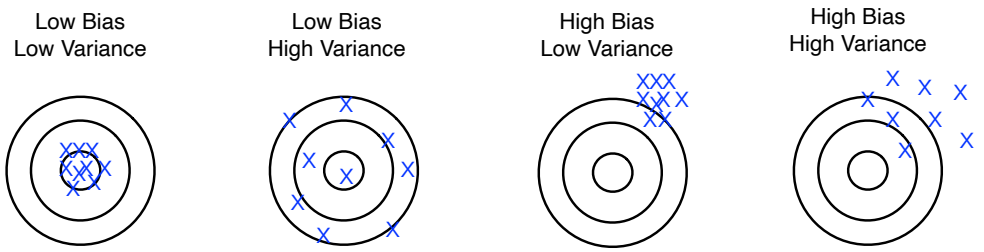


Figure 8-14: Example of underfitting and overfitting in model fitting. The target is the actual data, and the thrown darts are the predicted data.

[Domingos '12]. In other words, we use bias and variance to study whether our model accurately describes the underlying dataset or is misleading.

Variance refers to how far our observed data (or training dataset) differs from the mean of the predicted data (output variables). It is clear that different training sets result in different output variables, but we need to be sure that changes in the output variables for different training sets are insignificant, i.e., the variety of output variables is low. In other words, variance refers to how much the predictions of a model vary or fluctuate for different training datasets (Variance is varieties of the model's outputs).

Bias refers to differences between the model output and correct output (what happens in the real world). In other words, bias presents the prediction error and measures how far off the model's predictions are from the true values, on average.

In summary, Variance is about the model's sensitivity to the training data and how much its predictions vary for different training sets. Bias is about the error in the model itself, regardless of the training data. Domingos [Domingos '12] described: *"Bias is a learner's tendency to consistently learn the same wrong thing. Variance is the tendency to learn random things irrespective of the real signal."* Figure 8-14 presents the bias and variance analogy by throwing darts at the dartboard. In this example, assume the correct output is the closeness to the center of the dartboard, and the input is a set of parameters that affect the dart direction. Sometimes we will get a good distribution of training data so we predict very well and we are close to the dartboard center, while sometimes our training data might be full of outliers values resulting in poorer predictions.

A good model has both a low bias and low variance. However, there is a phenomenon known as the *Bias-Variance tradeoff*, which means an increase in one of them causes a decrease in the other one. Earlier in this chapter (Evaluating the fitness (accuracy) of linear models section), we discussed MSE while describing the fitness of a linear regression model. MSE has been explained earlier in this chapter

With some mathematics that we skip to explain, the MSE can be decomposed into the error equation as follows: $Error = bias(\hat{y}_i)^2 + variance(\hat{y}_i) + variance(e)$. Here, e presents *irreducible error*, which is an error that can not be reduced by having a good model and it is

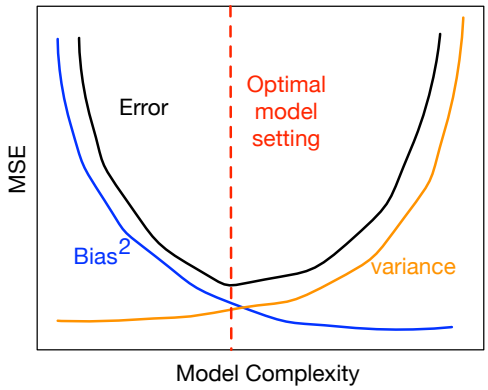


Figure 8-15: Typical bias-variance and error changes. At the red dotted line, the model has the smallest MSE error, thus, complexity at this point can be used to develop the optimal model.

always there. The *Error* on the left is called the *reducible error*. It is a sum of $bias(\hat{y}_i)^2 + variance(\hat{y}_i)$, and irreducible error.

The more we increase the complexity of a model, the bias will be reduced, but the variance increases. We should look for an optimal boundary, something like the red line in Figure 8-15, in which we have the lowest error rate, MSE is the Error in this figure. The red dotted line will be identified by experimenting with models with different complexities (e.g., more model parameters lead to an increase in the complexity).

Any classification algorithm that builds a model creates a decision boundary, which can be a line that is straight, curved, or has a complex form [Burkov '18]. The decision boundary is used to separate labels for data points. For example, in Figure 8-15, the decision boundary is depicted as a straight red dotted line.e.

As a final remark about these model-building devils, keep in mind that it is recommended [Grus '19] to mitigate high bias, we can add more features to the model, and to mitigate high variance, we can remove some features from the model and add more sample data points (not features but data points). It means that more flexible models (which cover more features) have higher variance.

NOTES:

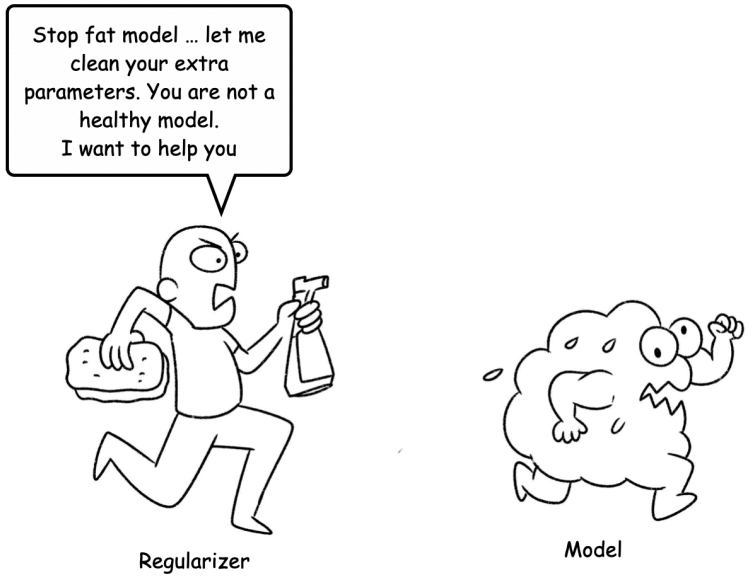
* High bias is often associated with underfitting, which occurs when a model is too simple and does not capture the complexities of the dataset. This simplicity leads to errors because the model cannot adjust to the data's underlying patterns. High variance is associated with overfitting, where a model is too complex relative to the simplicity of the data. Such a model captures random noise in the training data, leading to a model that doesn't generalize to new, unseen data.
* Flexible models, which can adapt their shape to the training data, often have lower bias. However, if too flexible, they may also capture the noise in the data, leading to high variance.

* Usually, underfitting is characterized by high bias and low variance. Keep in mind that underfitting can not be resolved by adding more training data points. To resolve the underfitting. We require to create more complex models, such as adding more parameters.
* A small sample size usually results in high variance, and the simplest approach to mitigate high variance is to increase the sample size. However, usually, the dataset is available before we start building our model, and we can not easily increase the sample size. The data is a valuable source, and usually, we use it with lots of care. There are other techniques, such as adding regularization (we will explain them shortly), reducing the number of features, and using ensemble methods that can help reduce variance without increasing the amount of data.

Regularization

Now, we are familiar with devils in model building, and one of the fiercest devils is overfitting. Due to Occam Razor's principle, we are always looking for less complex (parsimonious) models. Besides, we should always consider reducing overfitting, and for these two needs, we can use *regularization*. Regularization is defined as constraining or shrinking a model to reduce the risk of overfitting the given training set. It applies to the training phase only, not the testing phase. They make the model simpler and try to reduce model coefficients (parameters). Hence, they are also called *shrinkage* methods. In other words, regularization is the process of *shrinking the model's coefficients* and focusing on *decreasing the variance of the model*. Regularizations penalize the high power coefficients that make the polynomial model very sensitive to noise.

If a coefficient of a regression β_k is zero, its predictor will get zero effect on the model as well. Assuming x_j is a model parameter $\beta_k \,.\, x_j = 0.x_j = 0$, and thus, we can remove x_j. It means that this particular variable will be removed from the model, which is good. Why? Because we use the regularization to reduce the model parameters.



Regularization is crucial for working with regressions. There are $(2^n - 1)$ possible linear models from a combination of *n* input variables. For example, assuming we have three input variables as x_1, x_2 and x_3, we can have $(2^3 - 1)$ combinations of these parameters: x_1, x_2, x_3, x_1x_2, x_2x_3, x_1x_3, and $x_1x_2x_3$ and each is related to a model parameter. If we have 10 input variables, we can build 2^{10} different linear models. If we have 100 input variables, which is common in real-world applications, we need a supercomputer to choose the model from a set of $2^{100} - 1 = 1267650600228229401496703205375$ models. Therefore, we need a way to shrink models and reduce the number of choices as much as we can. This task will be done by regularization methods. In this section, we explain four common regularization methods, including *Ridge, LASSO, ElasticNet,* and *NonNegative Garrote*.

Ridge

Ridge regression or regularization (Tikhonov-Miller regularization, L_2 regularization, or weight decay) [Tikhonov '43] can be used when the model's *independent variables are highly correlated* (multicollinearity). Also, we use it when there is a multicollinearity among the features or we have few training data points. In other words, it makes the prediction less sensitive to the training data, by reducing the variance of the model.

Ridge penalizes the model based on the distances of predictor coefficients (weights) from zero. The higher the distance between coefficient and zero, the higher will be the penalty. It is an RSS (If you can't recall RSS, check "Evaluating the fitness of training sets in linear models" section of this chapter), plus a penalty:

$$Ridge\ \ Regression = RSS + penalty.$$

Assuming θ is a vector and presents all parameters of our model and we have *n* parameters, the penalty will be shown as $\lambda\Sigma_{i=1}^{n}\theta_i^2$, which is λ times the sum of square coefficients. λ is a hyperparameter used to define the *severity of the penalty* or *tuning parameter*. As λ increases, the flexibility of the ridge regression fit decreases, leading to a decrease in variance but an increase in bias. The Ridge regression cost function can be written as follows:

$$J(\theta) = RSS(\theta) + \lambda\Sigma_{i=1}^{n}\theta_i^2$$

Usually, when we encounter *J(something)* it presents an objective function for *something* parameters. A question might arise now: how can we choose an appropriate λ? Typically, we use 10-fold cross-validation to choose the one that provides the lowest variance. It means, we test with different λ values until we identify a λ that results in the lowest variance. The objective of Ridge regression is to minimize $J(\theta)$. In some literature, MSE [Géron '19] has been used instead of RSS, but both methods are correct (MSE is just the RSS divided by the sample size).

Figure 8-16 presents an example model with three parameters, and after increasing the λ to a specific value, all parameters converge toward zero (but do not get zero). This example shows us how increasing the penalty will reduce the impact of each coefficient on the model until they remove their associated variables because their coefficients will be zero. Keep in mind that Ridge

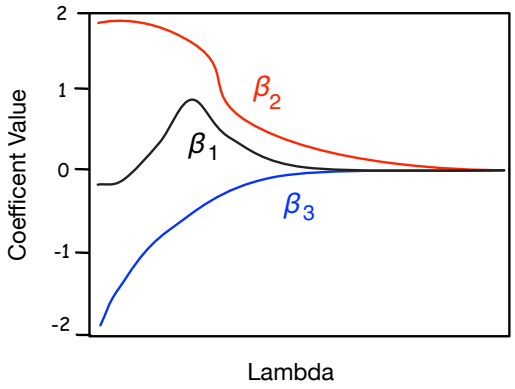


Figure 8-16: A mock model with three parameters that presents the impact of ridge regression. All parameters are converged toward zero as λ value increases.

regression introduces a small amount of bias, and increasing the bias will cause a significant decrease in the variance.

Let's use a simple example to better understand the ridge regression. Before explaining the example, note that acquiring data is an expensive process, and we should use a small number of data objects (as a training dataset) to create the regression model and then use the model on the test dataset.

Figure 8-17 (a) shows a toy dataset composed of 7 data points, and we would like to use linear regression to make a prediction model for this dataset. Figure 8-17 (b) shows two random data points that have been selected as training and marked in red. By calculating their RSS, we identify their RSS is zero, which is too low for bias, and this makes us suspicious. Figure 8-17 (c) calculates the RSS for other points (test set), which are marked with blue color. Test set data points (blue dots) stays far the line constructed by using training set (red dots). This is a sign of high variance (overfitting) and we can say that this regression line is not a good model because we observe overfitting.

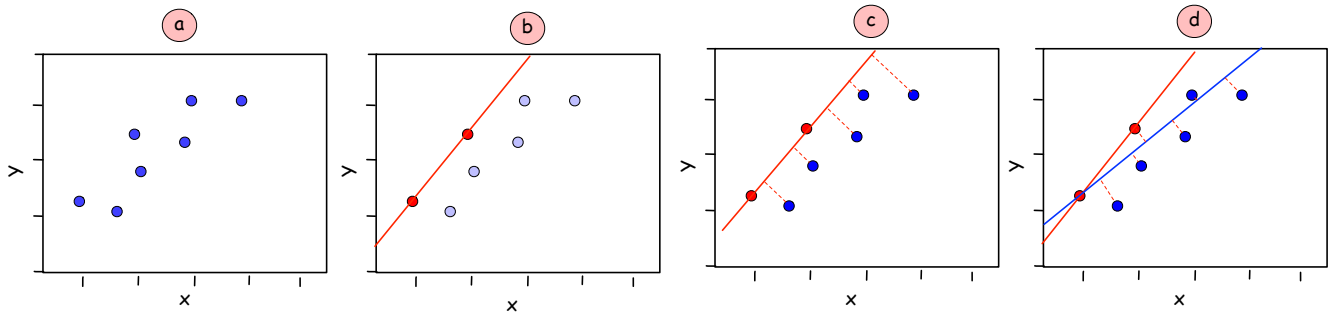


Figure 8-17: (a) Original dataset. (b) Training set data points were selected as red color. (c) The other remaining points RSS are calculated to the regression line (red regression line). (d) By introducing a Ridge penalty, the red regression line is rotated, and the blue line regression is drawn.

To solve the problem of high variance, we use a Ridge regression, and an increase in its penalty causes a rotation of the red regression line a bit toward the X-axis. As a result, the blue regression line will be created, which is shown in Figure 8-17 (d). The blue regression line has a lower variance despite having a bit more bias. In such a simple approach, Ridge regression resolves the overfitting problem. Ridge regression can be applied to logistic regressions and other regression methods as well, but we used a linear regression example for the sake of simplicity.

If you would like to learn how the math behind the Ridge regression works, you can plot some data points in 2D spaces and try them on your own with the linear regression, which is easy to calculate on a piece of paper.

Note that the shrinkage penalty is not applied to the intercept (β_0), it is applied to other parameters ($\beta_1, \beta_2, \ldots, \beta_k$), because the intercept measures the mean value of responses, and we do not want to shrink that. Increasing the λ causes a decrease in the slope. The larger the lambda gets, the prediction for y becomes less and less sensitive to x. In other words, the introduction of the penalty moves the regression line more toward the horizon line (parallel to the x-axis). When all coefficients are set to zero, the regression line will be a horizontal line parallel to the x-axis, which obviously is useless because $y = \beta_0 + 0 \times x_1 + 0 \times x_2 + \ldots = \beta_0$, and we do not have any predictor in the final model.

Assuming m is the number of our parameters, and n is our data points for the training set. The computational complexity of the ridge regression is $O(m^2 n)$. Ridge is good when we have a small dataset size, but it is not scale-invariant[6].

LASSO

LASSO (Least Absolute Shrinkage and Selection Operator) [Santosa '86, Tibshirani '96] is another regularization method in use and performs slightly better than ridge regression.

Ridge regression is very helpful, but it has a disadvantage. It keeps all coefficients in the final model and does not remove them completely. It means that Ridge reduces all coefficients toward zero but didn't set any of them to zero. In other words, Ridge does not exclude predictors that are unrelated to the model, it only reduces the magnitude of their coefficients, which is against Occam's Razor principle. Therefore, all predictors will stay in the final model, even the unrelated ones, but with a low coefficient. However, the LASSO regularization resolves this by substituting the Ridge's penalty $\Sigma_{i=1}^{n}\theta_i^2$ (L_2 norm) with $\Sigma_{i=1}^{n}|\theta_i|$ or $||\theta_i||_1$ (L_1 norm). We have the following equations for these two regularizations:

$$Ridge\ \ Regression = RSS + \lambda\Sigma_{i=1}^{n}\theta_i^2$$

$$LASSO\ \ Regression = RSS + \lambda\Sigma_{i=1}^{n}|\theta_i|$$

LASSO is very simple, it just uses an absolute value instead of squares. Nevertheless, it creates a parsimonious model (respecting Oscam's Razor principle), and even it has been recommended to use it as a feature reduction technique as well.

---

[6] If you recall, we have used this term in Chapter 6, while describing SIFT. However, scale invariant means that the feature or characteristics of the system should not change if their scale of lengths, energy, or other variables changes.

For example, assume we are creating a linear model to predict the success of a candidate in an upcoming election. The model measures success based on the following parameters as follows:

$success = \beta_0 + \beta_1.diet + \beta_2.hobby + \beta_3.climate\text{-}change + \beta_4.foreign\text{-}policy.$

If we have a dataset about this model and use Ridge regression, by increasing the λ, both personal's "diet policy" and "hobby", which do not affect on the success of the candidate will be shrunken toward zero, but they wouldn't be removed completely. However, LASSO can clean the model and get rid of both (diet and hobby) by increasing the λ, and thus the final model will only keep "climate-change policy" and "foreign-policy" as its predictors. LASSO is better than Ridge because we can say that its result models are more parsimonious. In summary, LASSO shrinks the coefficients to zero, unlike Ridge, which shrinks them toward zero but does not remove them entirely. Since LASSO can remove several features from the model, its result model is called a *sparse model.*

LASSO is better than Ridge, but it does not perform very well for highly correlated predictors, and it is also not scale invariant. Besides, when the number of parameters is larger than the number of training set data points, it does not perform well.

Assuming k is the number of variables and n is the number of sample sizes, LASSO computational complexity is $O(k^3 + k^2 n)$ [Efron '04].

Elastic Net

Both Ridge and LASSO are useful, but using them when our model includes many parameters, might be challenging. Besides, when the number of our parameters is larger than the available data points for training both LASSO and Ridge do not perform well.

Elastic Net regression [Zou '05] can mitigate these issues because it combines the strength of both Ridge and LASSO. Elastic Net regression is a combination of Ridge (L_2) and LASSO (L_1) regressions. We write the equation of Elastic Net as follows:

$Elastic\ Net\ Regression = RSS + \lambda_1 \Sigma_{i=1}^{n} \theta_i^2 + \lambda_2 \Sigma_{i=1}^{n} |\theta_i|$

λ_1 is used for Ridge and λ_2 is used for LASSO. We should use cross-validation to find the best values for both λ_1 and λ_2. By setting $\lambda_1=0$, we will have LASSO regression, and by setting $\lambda_2=0$, we will have Ridge regression.

Elastic Net uses this combination of penalties, which can better deal with correlative parameters. Therefore, Elastic Net includes advantages of LASSO and Ridge, but the best way to decide about a regularization selection is to conduct several experiments on the dataset with different regularizations and check results until a good one that addresses our concerns is identified.

Non-Negative Garrote

Another regularization method worth being familiar with is Non-Negative Garrote (NNG) [Breiman '95]. If you don't know what is Garrote, it is better not to search online for it, and we wonder how the author of this method cannot end up using a less gruesome name.

NNG is comparable to LASSO, but it is scale-invariant. It can also eliminate predictors with weak coefficients (LASSO can not do it). NNG does not require specifying the penalty function separately, as it has a fixed penalty function. It only requires specifying the tuning parameter λ, which reduces the computational complexity compared to methods that require selecting both the penalty function and the tuning parameter [Xiong '10].

Assuming n is the number of parameters and $u \geq 0$ is the shrinkage factor, its equation is written as follows:

$$\textit{Non-Negative Garrote} = RSS + 2\lambda\Sigma_{i=1}^{n} u_i^2$$

There are different approaches to computer the shrinkage factor u_i for parameter i. For example, assuming β_i is the ith parameter, and $\hat{\beta_{i,LS}}$ is the least squares estimate of the parameters, we can rewrite the Non-Negative Garrote as follows:

$$\textit{Non-Negative Garrote} = RSS + \lambda \sum_{i=1}^{P} \frac{\beta_i^2}{\hat{\beta}_{i,LS}^2}$$

This method is not as popular as previous methods and unlike Elastic Net, it cannot handle datasets whose number of predictors (or parameters) are larger than training data points. Perhaps its attractive and friendly name demotivates researchers and developers to implement it into their software packages or libraries.

NOTES:

* Keep in mind that regularization should be performed on the training set. We should use the same regularized model for both training and testing to assess the model's generalization performance accurately.
* The cost function result of the training set is usually different from the cost function result of the test set because regularization is applied to the cost function at the training phase.
* We have explained that Ridge regression can help to model data with a small training set. Even in some cases, the number of parameters is larger than the number of training data. If we have a model whose parameters are larger than its data points (the dataset is small), we can also use a regularization regression to reduce the magnitude or influence of its parameters.
* In the world of mathematics, we have a concept referred to as *Lagrangian relaxation*. It is a method to approach a difficult problem of constrained optimization by a simpler problem. It solves the optimization problem by introducing additional variables called *Lagrange multipliers*, which act as a penalty. As a result, it simplifies the optimization solution. While regularization methods use penalty terms in their objective functions, they cannot be considered as direct applications of Lagrangian relaxation. Regularization methods are designed to address the problem of overfitting and improve the generalization of models rather than solving constrained optimization problems (Lagrangian relaxation does it).

Optimization Algorithms

Any repetitive process that is performed by humans is associated with optimization. Optimization is a fundamental concept that can be observed even in everyday human behaviors, such as learning to eat efficiently. Initially, infants explore various methods, using their hands, faces, and different objects to feed themselves. This exploratory phase is messy but crucial for learning. As they grow, individuals experiment with various tools like spoons, forks, and chopsticks, gradually optimizing their technique based on feedback, effectiveness, and cultural practices. This iterative process of experimenting, learning, and refining is similar to optimization in scientific concepts. In particular, optimization tests various strategies and adjusts to achieve the best possible outcome, such as minimizing waste, maximizing efficiency, or achieving a specific goal.

Usually, identifying parameters of an unknown mathematical equation, which is the objective of many machine learning algorithms, consumes lots of computational resources. Resources are limited, and an optimization algorithm can help to search efficiently for optimal model parameters within the parameter space. In other words, we use optimization to estimate model parameters while minimizing the computational resources needed to identify optimal model parameters. Before explaining optimization we need to briefly review some algebraic concepts, which most probably we have learned in the school.

Mathematical Concepts Required for Optimization

Probably, if you are not a mathematician and reading this book, you have not encountered derivative gradients after primary/high school, and while you were in primary/high school, you might have thought they were completely useless for your life, especially if you do your high school in developing countries which emphasize on theoretical science more than practical ones. We have been in the same boat until now, and we thought the same way. At that time, we didn't know that mathematics would stick to our lives like a parasite and we could never eliminate it. Galileo, who was a decent scientist said that *"the book of nature is written in a mathematical language"*.

In this section, first, we explain these concepts very briefly. If you are familiar with these mathematical concepts feel free to skip this section.

Derivative

Lets review that a function is something that gets an input variable x, does something with it, and produces the output, i.e. $f(x)$ or y. *Derivative* is a variable that measures the sensitivity of changes in the output variable with respect to changes in the input variable. In other words, it specifies how the output will change, based on the given input. If it is not still clear, read the following: the derivative of a function describes how fast the function changes (increases or decreases).

If the function is a single straight line, the derivative is a constant variable. If the function is a horizontal line parallel to x-axis, it means there is no slope, and its derivative is zero. If the function is changing at different paces, such as in Figure 8-19, the derivative is a function itself. The derivative is written as a ratio of output changes over the ratio of input changes:

$$Derivative = \frac{Output\ Changes}{Input\ Changes}$$

The process of finding the derivative is called *differentiation*. A derivative of a function $f(x)$ is written as $f'(x)$, which is the notation introduced by *Lagrange*. A bit sexier version that is widely in use is called *Leibniz* notation, and it is written as follows:

$$f'(x) = \frac{d\ f(x)}{d\ x}$$

As you can see from the red lines in Figure 8-18, when the function is not a straight line, the derivative is changing, because, at different points, the function has a different pace of change.

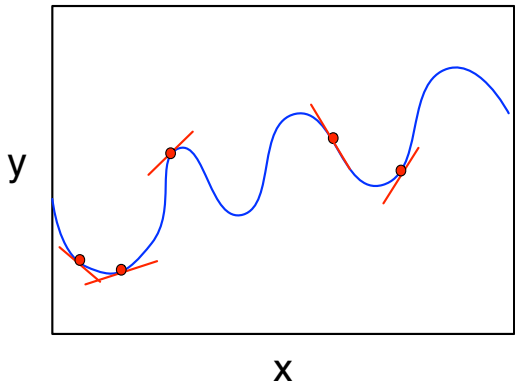


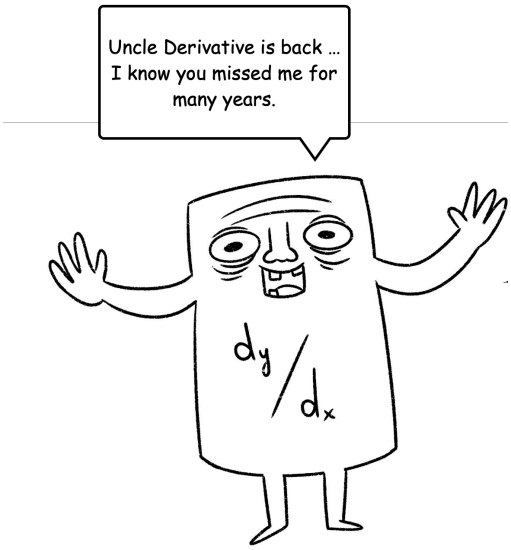
Figure 8-18: A sample function that is plotted in two dimensional spaces ($\mathbb{R}^2$) and five random tangent lines (red lines) has been drawn. You can see that each tangent line touches the curve at one point (red dots).

A constant derivative means that the function is growing constantly and a derivative of a straight line is a constant value.

To better understand the concept of derivatives, let's take a look at Figure 8-19. This figure shows a function that can be modulated with polynomial regression. A line that "just touches" the curve at each arbitrary point is called a ‘tangent' line. The particular point that connects the tangent line to the curve is called the point of tangency (red points in Figure 8-18).

The derivative is responsible for specifying the slope for the given tangent line of the function. From Figure 8-18, we can see different tangent lines and their slope changes at each point. More

formally, the derivative of a function at a certain point gives the slope of the tangent line to the function's graph at that point.



However, each tangent line has one point on the function, as shown in Figure 8-18, we need at least two points to be able to draw a line. To calculate the slope, we use the following equation:

$$slope = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

This equation shows to calculate the slope, we should select two points from the curve, and by using them, we can calculate the slope of the line connecting them. Let's say the first data point has coordinates $(x_0, f(x_0))$, the second data point is located in h distance to the first data point, thus, it has a coordinate of $(x_0 + h, f(x_0 + h))$. We can rewrite the slope equation as follows:

$$slope = \frac{f(x_0 + h) - f(x_0)}{(x_0 + h) - x_0} = \frac{f(x_0 + h) - f(x_0)}{h}$$

However, the smaller h leads to having a more accurate slope of that particular curve, but why? Let's take a look at Figure 8-19. In Figure 8-19 (a), we can see that the distance of h is the largest, and the line that describes the curve's slope is the least accurate line (in comparison to Figure 8-19 (b) and Figure 8-19 (c)). Then, in Figure 8-19 (b), the points get closer, and thus, the line gets more accurate because the direction is getting more specified. In Figure 8-19 (c), the data points get closer to each other, and the slope line gets more accurate than what we had in Figure 8-19 (b) because, from Figure 8-19 (a) to Figure 8-19 (c), the red line becomes more tangential and does not cross the function line (blue lines).

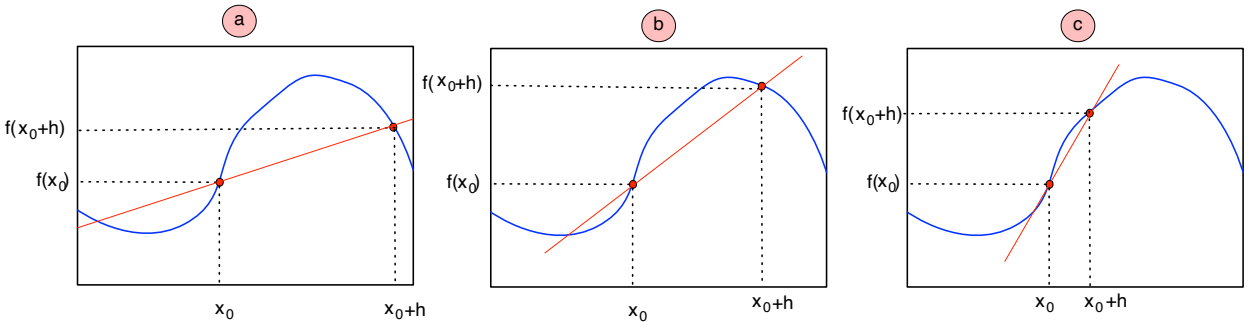


Figure 8-19: Three examples of slope line identification based on distances between two data points, which have *h* distance. These slope lines are called 'secant' lines. The more *h* gets smaller, the *secant* line gets closer to being a tangent line.

From Figure 8-19 we can conclude that, as the h distance gets smaller, the slope line to the tangent line is getting more accurate. Therefore, we can write the derivation function of x as the *limit* as h is approaching zero:

$$f'(x) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

There are some basic rules for a derivative that we need to know, assuming n is a constant number, including:

$f(x) = x^n \rightarrow f'(x) = n\,x^{n-1}$,

$f(x) = n\,x \rightarrow f'(x) = n$,

$f(x) = ln(x) \rightarrow f'(x) = 1/x$,

$f(x) = x \rightarrow f'(x) = 1$,

$f(x) = n \rightarrow f'(x) = 0$,

In other words, the function x^n has the slope of $n\,x^{n-1}$, the function $n\,x$ has the slope of n, etc.

What we have explained is a first-order derivative. It is also possible to make a derivative from a derivative, i.e. *second-order derivative***,** and it is written as $f''(x)$, e.g.,
$f(x) = x^2 \rightarrow f'(x) = 2x \rightarrow f''(x) = 2$

The second-order derivative is more stable than the first-order derivative because it captures the rate of change of the first-order derivative. However, due to its high computational cost, it is used rarely. To calculate the derivative, we should use some rules specified for the derivative including the *chain rule*, *quotient rule*, *product rule, sum rule* and *difference rule,* which we do not explain in detail here and just write down. If you would like to learn their proof, there are plenty of online resources to help you understand them.

Chain rule (outside-inside rule): $[f(g(x))]' = f'(g(x))g'(x)$ # it most important rule that is used in backpropagation.

Reciprocal rule: $(\frac{1}{x})' = \frac{-1}{x^2}$ # it is known as a specific case of Quotient rule.

Product rule: $(f(x)\,.\,g(x))' = f(x)\,.\,g(x)' + f(x)'\,.\,g(x)$

Quotient rule: $(\frac{f(x)}{g(x)})' = \frac{g(x)f(x)' - g(x)'f(x)}{g^2}$

Difference rule: $(f(x) - g(x))' = f'(x) - g'(x)$

Sum rule: $(f(x) + g(x))' = f'(x) + g'(x)$

We will learn how Chain rule is used for Backpropagation Chapter 10. For now, keep in mind that the Chain rule is useful for finding a derivative of a function when we have nested functions (a function inside another function), e.g. $g(\,.\,)$ is inside $f(\,.\,)$. We can say it is a derivative of the outside function while leaving the inside function times the derivative of the inside function, check this equation $[f(g(x))]' = f'(g(x)) \times g'(x)$ we wrote about chain rule.

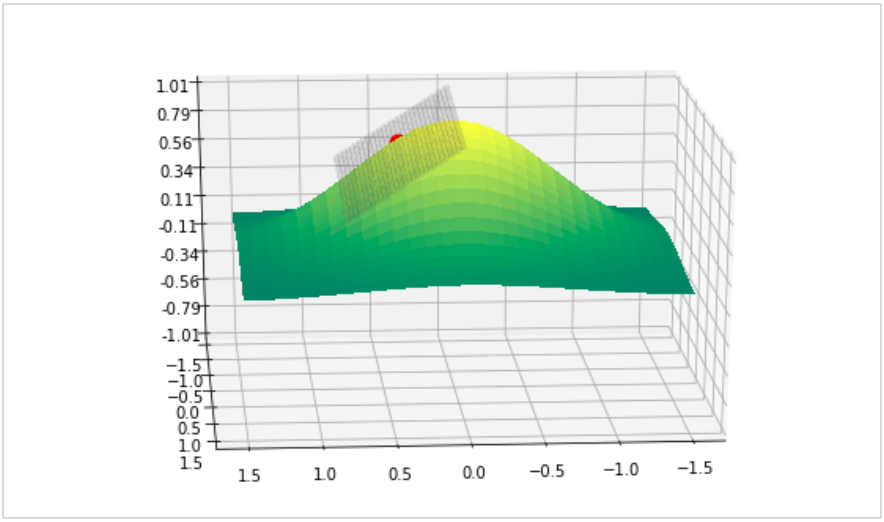


Figure 8-20: Tangent hyper plane plotted in 3D space.

Second-Order Derivative

Making a derivate from a derivative is referred to as second-order derivative. In mathematical terms, if you have a function $f(x)$ its derivative is written as $f'(x)$ or $\frac{df}{dx}$. The second-order derivative is the derivative of $f'(x)$ and it is denoted as $f''(x)$ or $\frac{d^2f}{dx^2}$.

The second-order derivative is particularly useful in various applications such as physics, engineering, and economics, where it can signify concepts like acceleration (the rate of change of velocity) or concavity/convexity of functions, among others. However, due to its high computational cost we will avoid it or in cases we must implement it, we use a trick to simulate second-order derivative.

Partial Derivative

We have discussed the derivative, which is used for functions of one variable. Sometimes, we need to deal with a function with more than one variable, e.g., $f(x, y) = x^2 + 4y$. To differentiate such a function, we use the *partial derivative*. A partial derivative once computes the derivative for x and considers the other variable (y) as constant. Then, it computes the derivative for y and considers the variable x as constant.

Instead of d we used to refer to the derivative, here, we use ∂ sign for partial derivative. To write a partial derivative for variable x and variable y, we write the following:

$$\frac{\partial}{\partial x}[f(x, y)] = 2x + 0 \text{ and } \frac{\partial}{\partial y}[f'(y)] = 0 + 4$$

Gradient

Now that we understand derivatives, let's discuss them in the context of multivariate functions. Recall that the derivative in univariate functions represents the rate of change with respect to a single variable and is represented in two-dimensional space, often visualized in $\mathbb{R}^2$. What happens when we extend this concept to functions of more than one variable, i.e., *multivariate* , which is represented in higher-dimensional spaces such as $\mathbb{R}^3$?

The generalization of derivatives for a multivariate function is referred to as *Gradient*. In simple words, the gradient is used for functions that take more than one input variable (or parameter). It is a *vector of partial derivatives,* and gives the input direction in which the function increases or decreases. In machine learning, we often deal with high-dimensional spaces, and while the derivative conceptually applies to vectors, the gradient extends to higher-dimensional structures like matrices and tensors.

While calculating the gradient, instead of the tangent line, which we had in the derivative, we have a *tangent plane* or *hyperplane* for the gradient. Figure 8-20 visualizes the gradient hyperplane on a function plotted in 3D space. Let's review what we have said: *a gradient is a vector of all partial derivatives of the target function.*

A Gradient of a function $f(X)$, assuming $X = \{x_1, x_2, \ldots x_n\}$ can be written as follows:

$$\nabla f(X) = \left[\frac{\partial f(X)}{\partial x_1}, \frac{\partial f(X)}{\partial x_2}, \ldots, \frac{\partial f(X)}{\partial x_n}\right]^{7}.$$

For example, for $f(x_1, x_2) = a\,x_1^2 + b x_2 + c$ function.

The derivative of this function with respect to x_1 is written as: $\frac{\partial f(x_1, x_2)}{\partial x_1}$. The gradient of this function is written as follows:

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = 2a\,x_1 + 0 + 0\,,\ \frac{\partial f(x_1, x_2)}{\partial x_2} = 0 + b + 0.$$

$$\nabla f(x_1, x_2) = \left[\frac{\partial f(x_1, x_2)}{\partial x_1}, \frac{\partial f(x_1, x_2)}{\partial x_2}\right] = \left[2a\,x_1, b\right]$$

In machine learning, *we calculate gradient for cost functions*, and it is used by the most popular optimization methods. Let's see another example, we choose RMSE as a cost function for a linear model, with *n* data points, we will have the following cost function:

$J(\beta_0, \beta_1) = \sqrt{\frac{\Sigma_{i=1}^{n}(\hat{y}_i - (\beta_1 x_i + \beta_0))}{n}}$, its gradient will be $\nabla J(\beta_0, \beta_1) = \left[\frac{\partial j}{\partial \beta_1}, \frac{\partial j}{\partial \beta_0}\right]$

---

[7] The ∇ sign is read as "nabla" and ∂ is a "partial" sign used for specifying a derivative.

Jacobian

When we stack the gradient of two functions, each of them is a vector of variables, into a matrix called a Jacobian matrix. For example, assume we have $\nabla f(x, y)$ and $\nabla g(x, y)$. The Jacobian matrix is written as follows:

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x, y)}{\partial x} & \frac{\partial f(x, y)}{\partial y} \\ \frac{\partial g(x, y)}{\partial x} & \frac{\partial g(x, y)}{\partial y} \end{bmatrix}$$

For example, if we have $f(x, y) = 4x^2y$ and $g(x, y) = 2x + y^3$, then their Jacobian matrix is written as follows:

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} 8x & 4x^2 \\ 2 & 3y^2 \end{bmatrix}$$

Hessian

The gradient is the first-order derivative of a multivariate function, and it can be written as a vector of variables. The Hessian matrix is a square matrix of second-order (a derivative of derivative) partial derivatives of a function. We have seen previously that the gradient can be written as $\nabla f(x_1, x_2, \ldots x_n)$. The Hessian, which is the second-order of the gradient, is a symmetric matrix, and it is written as follows:

$$\nabla^2 f \quad or \quad H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Hessian includes all possible second-order partial derivatives, or we can say it includes every possible pairing of the *n* variables.

Integral

Many physical and non-physical entities, such as acceleration, force, and velocity, are defined as a rate of change. The derivation is used to quantify the rate of change in mathematics.

However, the rate of change is not enough to force us to learn mathematics, and some devils, who named themselves mathematicians, introduced the concept of the integral. Integral has many physical and non-physical entities, such as specifying a region's weight with respect to the overall weight of a function.

The integral of a function is the 'area under the curve', which we use to 'specify the weight of the function in a certain range'. Integral is the opposite of derivative, and it is also called anti-derivative. The process of finding the integral is called *integration*. Both differentiation and integration are used to study changes in a function (integration is the opposite of differentiation). Differentiation is about finding the rate of change, while integration is about finding the initial function that was changed.

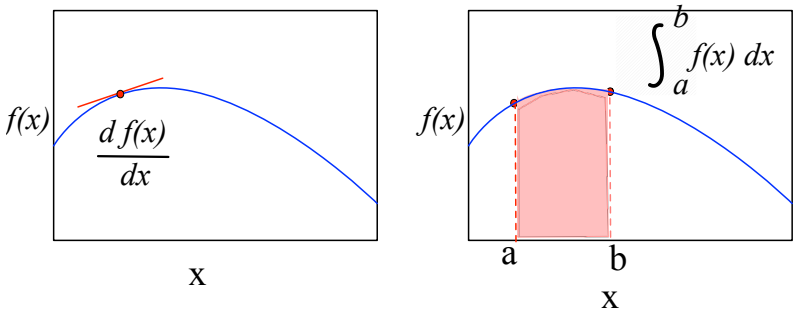


Figure 8-21: Comparison between derivative (left) and integral (right).

Figure 8-21 presents the derivative and integral of a function. The area under the curve is its integral. We have a plot that represents the speed of a car over time. The area under the curve presents the distance traveled by the car over a given period. To compute this distance, we can use integral.

Another application of integrals is finding the volume of an object with a complex shape. For example, we want to find the volume of a complexed-shaped swimming pool. We could use an integral to calculate the volume by dividing the swimming pool into smaller slices, finding the volume of each slice, and then adding them up to identify how much water this pool requires.

Assume we are filling this pool with water from a tap. The rate at which water flows from the tap can be considered the derivative of the water volume in the pool with respect to time. Thus, the total volume of water added to the pool over time can be found by integrating this rate over the relevant time period.

Assume we are filling this pool with water from a tap. The rate at which water flows from the tap can be considered the derivative of the water volume in the pool with respect to time. Thus, the total volume of water added to the pool over time can be found by integrating this rate over the relevant time period.

For example, if the tap is open and it provides a constant amount of water every second, e.g., 1 ml, the water volume inside the pool will increase at a constant rate, mathematically represented as $V(t) = t$ ml after t seconds. If the tap's rate of delivering water increases such that each second it doubles the previous second's rate—represented as $\frac{dV}{dt} = 2t$ ml per second, then the total volume of water in the pool over time will be $V(t) = t^2$ ml after t seconds.

Taylor Series

A *series* in mathematics is referred to as a group of several terms that are all about the same thing. For example, the following is a series:

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\ldots.$$

Taylor series or Taylor expansion is a function that has an infinite sum of terms. These terms, however, were calculated based on the repeated derivate at a single point. In mathematics, any function can be represented with a Taylor series.

For example, the Taylor series of functions e^x and $sin(x)$ are written as follows:

$$e^x = 1 + x + \frac{x^2}{2!} + +\frac{x^3}{3!} + \ldots \text{ or } sin(x) = x - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

Generally, the Taylor series of the function $f(x)$, that is differentiable at a real number a, is approximated as follow:

$$f(x) \approx f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots \approx \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Taylor series is used in the approximation function and approximation functions try to approximate a target function that is either hard to calculate or unknown, by a simpler function that is easy to calculate or known function.

Linear approximation refers to the first-order Taylor approximation, and quadratic approximation refers to the second-order Taylor approximation.

We learn some mathematical concepts required for optimization. There is more to explain about optimization, but we will postpone it to Chapter 10 where we describe neural networks and deep learning. To continue this section, knowing derivative, gradient, hessian, integral, and Taylor series are enough.

What is Optimization in Machine Learning?

All models we have explained in this chapter are heavily dependent on their parameters, e.g., linear regression is dependent on β_0 and β_1, and changes in these parameters change the model behavior. In the context of machine learning, optimization is the process of changing and tweaking model parameters to minimize the cost function. Why do we need an optimization algorithm, and why not experiment with all possible parameters (brute force) until we get the best result? To better understand the need for optimization, take a look at the right side of Figure 8-22. The bottom of this curve is the optimal parameter value because the cost is at its lowest. However, to get to the minima or maxima point via a brute force approach, we need to experiment with all possible points on the function curve, i.e., all blue dots on the curve. This is expensive to experiment with, and even for this simple convex function, we have too many blue points to experiment with. Recall in Chapter 4, we explained that a real-valued function is called convex if the line segment between any two distinct points on the graph of the function lies

above the graph between the two points. For convex functions, any local minimum is also a global minimum. It is efficient in optimization as it guarantees that if we find a local minimum, we have also found the global minimum. But still, it is not guaranteed that the optimization algorithm will find it.

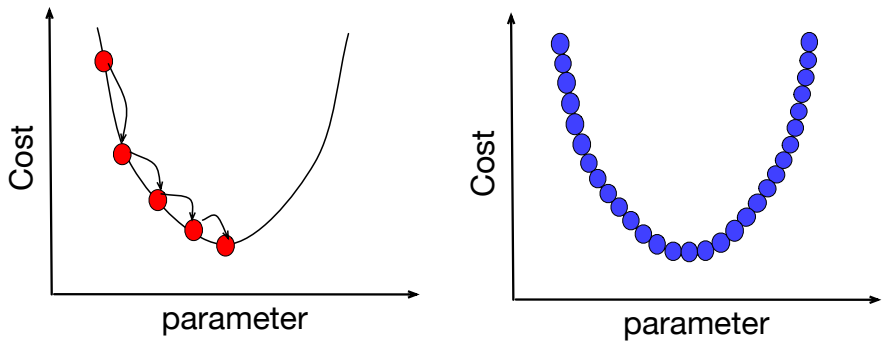


Figure 8-22: (left) Using an optimization to reach the minima, and therefore, very few data points will be processed. (right) Not using an optimization, and thus, many data points will have experimented on the function to find the minima.

However, if we use optimization, as is shown on the left side of Figure 8-22, we can skip many data points and get faster to the minimum. By faster, we mean fewer experiments.

Take a look at Figure 8-23, it is a function and we can see its local minimums (minima or sometimes called optima), have the smallest y value in each valley (or in comparison to its adjacent points), and global minima, is specified the smallest value of y in the entire function. Local minima are points where the cost function is lower than at adjacent points, while the global minima is the lowest point across the entire function.

The process of optimization in the context of the machine learning algorithm is usually identifying a good local minima. By good, we mean a number close to the global minimum or, ideally, the global minimum itself. Therefore, we can say that the ultimate goal of optimization is to find the global minimum. In reality, if we deal with non-convex shape optimization, it is not possible to find the optimal point (global minima), and thus we seek to identify strong local minima. If we have a convex shape (e.g., Figure 8-22), both local minima and global minima are the same, but still, it is not guaranteed that the optimization algorithm will find it.

Note that the blue curve in Figure 8-23 presents the cost function. We don't know the shape of this curve, and by optimization, we experiment with lots of different parameter values and measure the cost to identify this function shape (cost function shape).

By looking at the figure, we can easily identify local minima(s) or global minimum (or minima). Mathematically there are two conditions required to claim a datapoint as a local minima: (i) its first-order derivative is zero, i.e., $f'(x) = 0$ (ii) its second-order derivative is positive, i.e., $f''(x) \geq 0$.

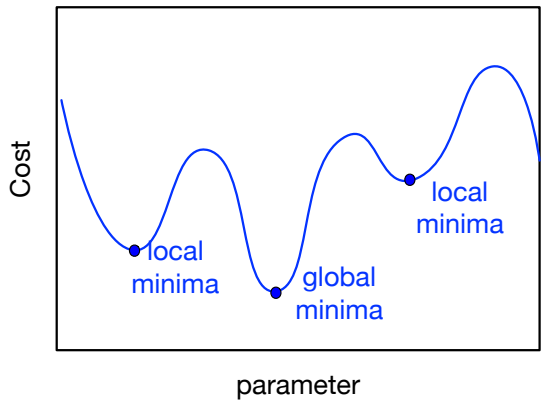


Figure 8-23: A sample function (non-convex shape) with its local minimal and global minima data points have been specified. While reading any of these optimization diagram note that the function presented in blue line is not known and the optimization objective is to find these minima.

Now, by using this background knowledge about optimization, we can say that optimization is focused on reducing the cost or loss function by choosing the right values for the model parameters. In other words, our goal is to build the best model to describe the dataset. Therefore, identifying the optimal value for model parameters reduces the error, and thus we can have the optimal model. Please accept our apologies for repeating it too many times, we want to be sure you memorize the logic behind optimization.

The algorithms we described in this chapter are associated with a cost function, e.g. RMSE is the cost function of linear regression, and changing parameters (β_0, β_1) will affect the RMSE. Therefore, an optimization algorithm is used to identify the lowest cost based on different combinations of β_0, β_1 . We do not use RMSE, because RMSE can be used for a convex function, which is usually not a case in a real-world model.

There are various methods for directing the search on the given function toward optimum, including gradient, Hessian, and directional derivatives. In the rest of this section, we explain gradient descent methods, which are widely used for directing the search toward the optimum.

Gradient Descent

Gradient descent is a first-order iterative optimization algorithm that focuses on *minimizing a function by moving its direction toward the steepest descent* (negative of gradient). We will explain Gradient descent in 2D space, which is easier to understand. Previously, we have explained that an optimization problem is the process of finding a good minima that is close to global minima. Now, assume we have a function, and we would like to optimize it, i.e., finding minima close to the global minimum. We knew from the previous section that the gradients of minimum points are zero. Also, the more we move forward toward the minimum point, the closer the gradient gets to zero. Thus, it gets a random data point and calculates its derivative (slope). Then, based on the slope value, another jump toward minima is made.

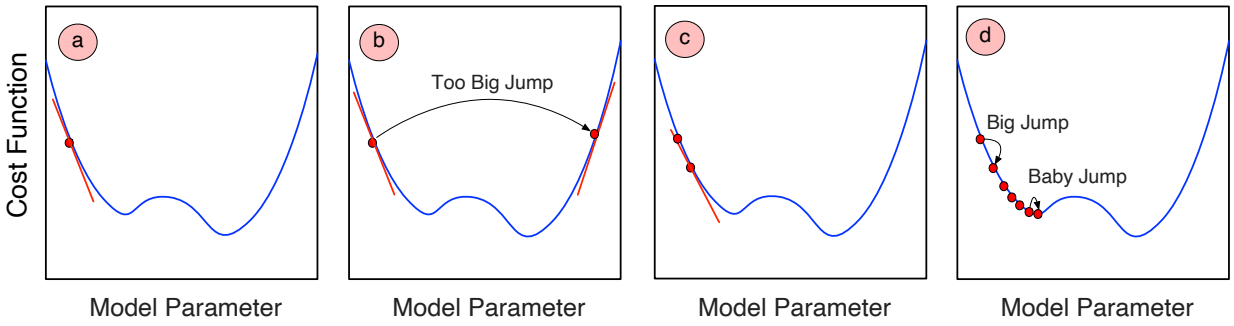


Figure 8-24: A gradient descent approach step by step in a non-convex cost function.

For example, look at Figure 8-24, which presents the cost of an imaginary function based on its model parameter. At the beginning of Figure 8-25 (a), a random point (red dot) is chosen on the function. Next, the optimization algorithm makes a jump, and another point is chosen on the function, as it has been shown in Figure 8-24 (b). Nevertheless, the sign of the derivative of the new point is changed, and this shows that the jump was too big. Thus, the optimization function discards this jump and makes another jump, but smaller, as shown in Figure 8-24 (c), and calculates its steepness. Since the absolute value of the derivative is smaller than the previous point, the algorithm realized that it was a correct jump toward minima. As the derivative of selected points gets closer to zero, the next jumps will get smaller, which means the function is close to a minima. This approach is adaptive step size computation, which is a commonly used policy optimizer. With enough repetitions, the gradient descent function can identify a good minima, which is close to the global minima or could be itself the global minima.

Gradient descent stops either (i) a specific threshold (maximum epoch) reaches, or (ii) the step size gets very small (derivative gets too small and close to zero), which means we reach the minima.

Now, a question might arise: why do we not take the step at the beginning small enough to avoid jumps like Figure 8-24 (b)? The answer is that although taking small steps makes the minima identification more accurate, it makes the gradient calculation process very time-consuming as well.

As another example, let's take a look at a surface plot in Figure 8-25, which shows a cost function of a linear regression model for both β_0 and β_1[8]. We can observe this simple surface plot has one minimum. To calculate the gradient descent, the first random point will be chosen, let's say ($\beta_0 = 0, \beta_1 = 0, RSME = 1000$). This figure shows that the first jumps are large (the size of the green line between two red dots), then they get smaller and smaller as the gradient descent function moves toward the minimum point (which in this case is global minima, but in complex real-world problems, the cost function is usually not convex and therefore it is not that simple).

[8] The visualization provided in this figure is adapted from the code provided at https://github.com/dolittle007/am207_2018/blob/master/wiki/gradientdescent.md

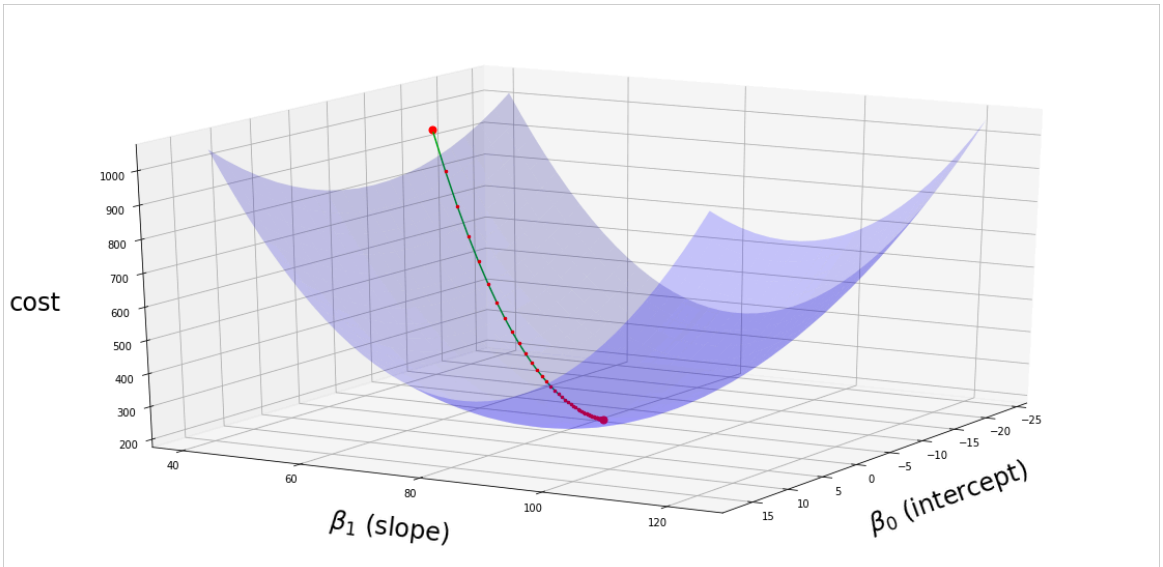


Figure 8-25: Visualization of step sizes of the gradient descent in green line. Each step starts with a red dot and stops in the next red dot. You can see that step sizes are getting smaller as we move toward the global minima in this convex function.

These step sizes were calculated by a parameter called *learning rate* times *slope* of that particular point for each parameter:

next point = current point - (learning rate $\times$ *slope (a derivative of that particular parameter).*

In particular, the learning rate specifies the step in each iteration. A learning rate is usually not constant, and it is given as a hyperparameter. Usually, the gradient descent algorithm starts with a small learning rate of 0.001 and improves it as they experiment more with the dataset. Learning rate, α, is a hyperparameter of gradient descent algorithm.

To summarize this discussion, we can take the following five steps to calculate the Gradient Descent as follows:

(1) The objective is to select a cost function for the given parameter values. For example, we chose to calculate RMSE for the given β_0 and β_1.

(2) A random value for each parameter will be selected and the cost for these parameters will be calculated.

(3) If the model has only one parameter, then its derivative, otherwise gradient, which gives us the slope of this particular cost value (from step 2) will be calculated.

(4) The algorithm calculates the next data point by using the following equation: $new\ point = current\ point - learning\ rate \times slope$. As the new data point will be specified, the algorithm uses this data point. The slope is equal to the cost function's gradient at the current point. Assuming the α is the learning rate, we can write the next point will be calculated as follows:

$$x_{t+1} = x_t - \alpha_t \,.\, \nabla f(x_t)$$

(5) The optimization algorithm checks whether a given threshold for iteration has been reached or the slope size of this new data point is zero (it means we reach the minima), or both values are converged (β_0 and β_1) and not changing anymore. If none of these two conditions were true, then again it goes to step 3 and continues to step 4 and then 5. Otherwise, it stops.

We should be careful with setting the threshold as well. If it is too small, we might never reach a good minima, and if it is too large, it could consume too much of our resources to find a good minima. Each time one training set is analyzed we say one epoch passed.

As a formalized example, let's say the linear regression has the following equation: $y = \beta_0 x + \beta_1$. We intend to use the Sum of Squared Residuals (SSR) error as a cost function.

We do not know the values for βs and both parameters should be identified to minimize the, which is written as $l = \sum_{i=1}^{n} (\hat{y}_i - (\beta_0 + \beta_1 x_i))^2$.

Gradient descent starts with calculating a partial derivative for every parameter, based on the chain rule:

$$\frac{\partial l}{\partial \beta_0} = -2(\hat{y}_i - (\beta_0 + \beta_1 x_i)) \text{ and } \frac{\partial l}{\partial \beta_1} = -2x_i(\hat{y}_i - (\beta_0 + \beta_1 x_i))$$

Instead of SSR we can use RMSE (Quadratic cost function) or other cost functions as well, including cross-entropy cost, which is used for classification, and its equation is written as follows:

$$l = -\sum_{j} (\hat{y}_j log(y_j) + (1 - \hat{y}_j) log(1 - y_j))$$

Gradient descent can use other cost functions which have been described (in Chapter 3 and Chapter 6), and we will learn more about other cost functions in upcoming chapters.

Types of Gradient Descent

There are several types of gradient descent: *batch gradient descent*, *mini-batch gradient descent*, s*tochastic gradient descent (SGD)*, *SGD with momentum*, etc. The right choice of optimization algorithm is very important because we might get a good accuracy after several days of waiting for the algorithm or get a good accuracy after a few minutes of experimenting. Here, we only explain three varieties of the Gradient Descent algorithm, which is the most popular optimization algorithm.

Batch Gradient Descent (BGD)

As a reminder, recall that the gradient descent uses the training dataset to identify the minimum value of the cost function for different values of model parameters.

A training dataset is divided into smaller segments that are referred to as batches. When all training datasets are collected in one single batch, we use gradient descent for optimization, which is called Batch Gradient Descent.

Batch Gradient Descent is the simplest form of gradient descent because it uses the entire training set to compute the gradients at every step. This means that in each epoch [9], *the entire training set* will be used, and then the gradient will be calculated to decide the next step. Despite its simplicity, since it loads the entire dataset, it is usually the most accurate gradient descent algorithm.

This approach is useful when we have a convex function such as RSME in linear regression because, with enough iterations, the algorithm can easily reach the global minima. Nevertheless, when the training set is large, it uses the entire training set and the batch gradient descent is not resource-efficient.

Stochastic Gradient Descent (SGD)

To solve the problem of using the entire training set another variety of gradient descent has been used since 1951 [Robbins '51], which is stochastic gradient descent. SGD picks *one random sample* from the training set instead of using the entire dataset and calculates its gradient in the first step. Then, again, for the next step, the algorithm takes *another single random sample* from the training set and calculates the gradient for this point. These iterations will continue until it reaches a zero gradient or the maximum threshold for epochs (iterations).

Using random samples from the training set and not the entire training set makes the algorithm very fast and efficient, especially when the training set is too large to load into the memory. Nevertheless, for non-convex functions, it is bouncing around the minima, and unlike BGD, the SGD is not moving straightly toward the minima.

If the cost function is convex, BGD can approach global minima easily. On the other hand, if the cost function is non-convex, BGD is stuck in local minima, but SGD can jump out of the local minima because of its irregular bouncing behavior.

We can observe in Figure 8-26 that the SGD is bouncing around the minima, but BGD moves toward the minima. However, there is no guarantee that this minimum is the global minima, and it could be a weak local minima. Even sometimes, when the step size is too large, the gradient descent can jump out of the minima, as we can see in Figure 8-24 (b). This problem is called *overshooting*.

You might ask why SGD is too noisy? Later in Chapter 10, we explain more about how to reduce the noise. Another simple solution to enforce SGD moves toward minima (even in cone shape functions such as Figure 8-26) is reducing the learning rate slowly and cautiously, which helps the algorithm to settle at the global minima. The process of slowly reducing the learning rate to reach global minima is called *simulated annealing,* and the function that determines a value for the learning rate is called *learning schedule*. If the learning rate reduces too fast, we might stuck

---

[9] Once again, we remind you that a complete pass through the training set is referred to as an epoch, and thus, processing each batch in BGD costs one epoch. Also, every single data point that is processed in SGD is one epoch.

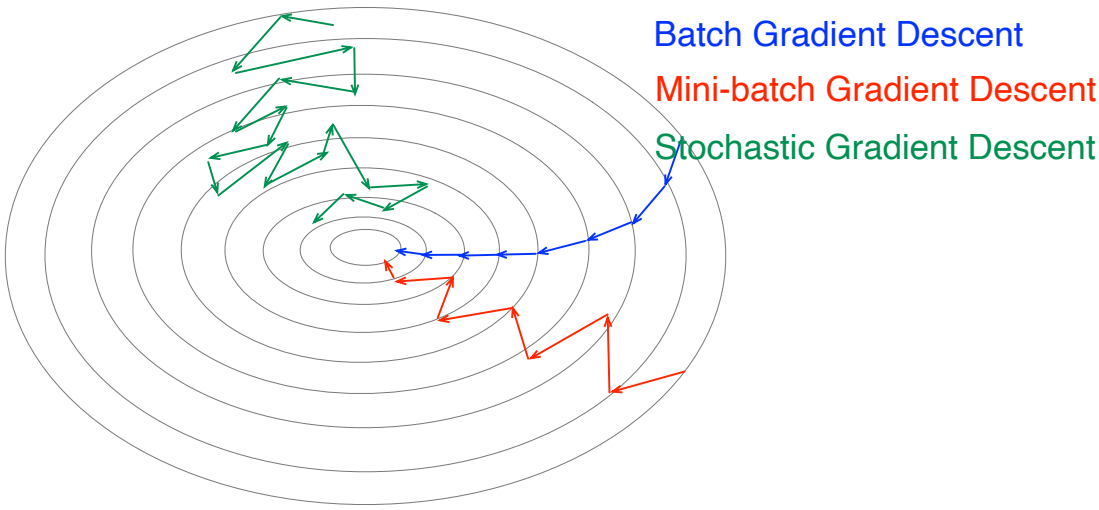


Figure 8-26: A toy example that shows a contour plot of a convex function. The minimum is located in the center. We can see that the batch gradient descent (blue colors) goes straight to the global minima. Mini-batch gradient descent (red colors) moves toward minima with some variances, and this variance significantly increases while using the stochastic gradient descent (green color).

in local minima, if it reduces too slow, we might jump to global minima and end up in a sub-optimal solution. Therefore, implementing a good learning schedule is not easy, but the underlying software library usually takes care of it. Just keep in mind that simulated annealing is useful when finding an approximate global minima is more important than finding precise local minima.

Mini Batch Gradient Descent (miniBGD)

We described that BGD uses the entire training set in each epoch to calculate the gradient, and SGD uses one single sample in each epoch. There is another variation that tries to be in the middle of these two algorithms named miniBGD. In particular, miniBGD uses *a subset of the training set*, (BGD uses the entire training set), and thus it is faster than BGD. Since it uses a subset of training data and not a single instance. However, it has less bouncing in comparison to SGD. In particular, it splits the training dataset (batch) into smaller sets (mini-batch), and in each epoch, it uses one mini-batch. A mini-batch is smaller than a batch and greater than one single data point. Similar to other types, miniBGD needs to have a dynamically changing learning rate, and thus, the learning schedule function here is also important. miniBGD is useful for parallelization of the gradient descent process across multiple GPU [Zhang ’21].

Table 8-2 summarizes the comparison between these three methods. We have learned many details about Gradient Descent; it might make sense to emphasize that gradient descent is a first-order optimization algorithm, which means it only works with the first derivation of the function, and it is iterative.

	Advantages	DisAdvantages
Batch Gardient Descent	- no bouncing and no variance - can identify global minima in convex function	- very slow - not suitable for large dataset - can stuck in local minima
Mini-Batch Gardient Descent	- fast - suitable for large dataset - reduce the variance overhead of SGD	- sensitive to the learning schedule - can stuck in local minima but better than BGD
Stochastic Gardient Descent	- very fast - suitable for large dataset - very unlikely to stuck in local minima	- sensitive to deal learning schedule - very high variance that might miss a good minima

Table 8-2: Comparison between advantages and disadvantages of different varieties of gradient descent algorithm.

Newton Method

Gradient descent algorithms and other first-order iterative algorithms (first-order means using the first-order derivative) can determine the direction toward a minima (good local or global). However, a limitation of first-order optimization algorithms is that they cannot determine the right step size. Gradient descent depends on a learning rate, and we describe how to identify the learning rate later in Chapter 10. Besides, the step size is a parameter that should be given to gradient descent algorithms. Nevertheless, it is still the best algorithm used for optimization, and no other algorithm can provide a better result than Gradient Descent.

Second-order optimization algorithms, i.e. Newton, is another iterative method that allows us to determine the 'approximate step size' toward a minima.

Gradient descent uses the derivative at a randomly chosen point to determine the direction of the steepest descent, effectively moving opposite to the gradient. On the other hand, Newton's method approximates the function by a parabola[10] at the current point, leveraging both the first and second derivatives, see Figure 8-27. This approximation is helpful

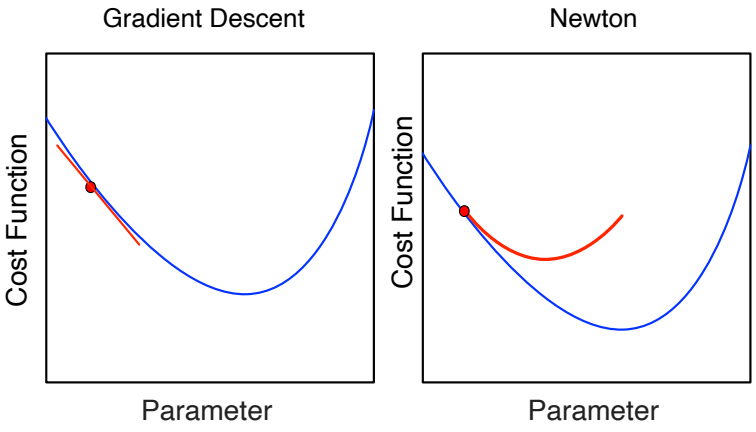


Figure 8-27: The differences between Gradient Descent and Newton method

[10] A symmetric bowl-shaped curve is called a parabola.

because it allows the method to find the zero of the derivative more accurately by considering the curvature of the function, which is reflected in the second derivative.

Since a parabola is either convex or concave, depending on its coefficients, it generally presents a clear path to a minima or maxima when it is convex. As is shown in Figure 8-28 after three iteration Newton method gets into the minima. In Newton's method, the minimum of the approximating parabola can be found at the vertex, which is derived from the roots of the first derivative using the second derivative for refinement. The newly identified x-coordinate of this vertex becomes the next point in the iterative search for a minimum.

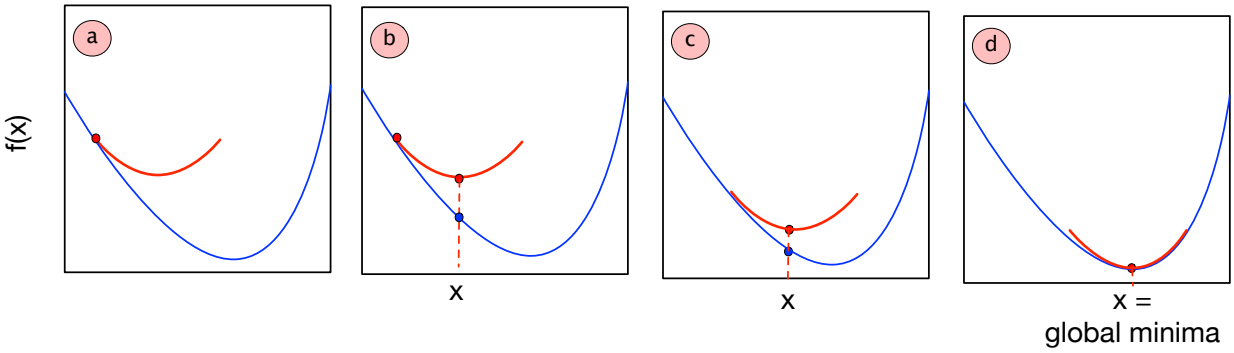


Figure 8-28: Newton method optimization example. We can see with a few steps, the global minima will be identified. Nevertheless, for the sake of simplicity, we assume the optimization function as a convex function.

To understand this concept, we go a bit deeper into its mathematical concepts. Assuming the current point is x_t how did we calculate the next point, $x_t + 1$ in gradient descent? If you remember, we use this equation:

$$x_{t+1} = x_t - \alpha_t \,.\, \nabla f(x_t)$$

Based on the Taylor series, a (quadratic) approximation function can be used to define the Newton method. Here, *f(x)* is the quadratic approximation, and with the Taylor series we can write it as follows:

$$f(x) = f(x^{(k)}) + (x - x^{(k)}) \,.\, f'(x^{(k)}) + \frac{(x - x^{(k)})^2}{2} \,.\, f''(x^{(k)}) + \dots$$

To calculate the next point, instead of using the slope, we substitute the target point with the Hessian matrix of that point. Therefore, we will have:

$$x_{t+1} = x_t - Hessian^{-1} \,.\, Gradient$$

Or we can write it as follows:

$$x_{t+1} = x_t - [\nabla^2 f(x_t)]^{-1} \,.\, \nabla f(x_t)$$

$\nabla^2 f(x_t)$ represents the Hessian matrix at x_t, which is the matrix of second-order partial derivatives. Similar to previous examples, for the sake of simplicity, we explain the univariate form of a function.

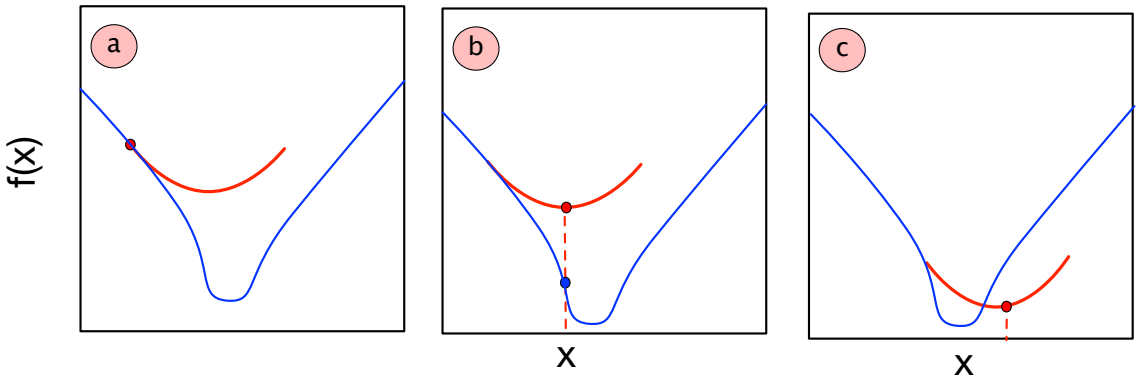
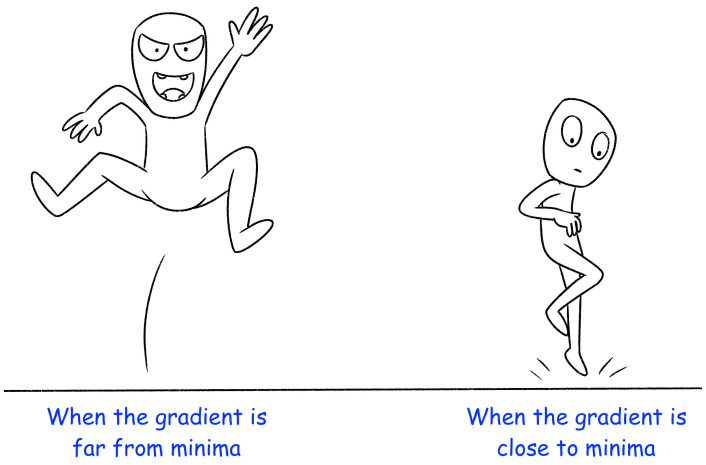


Figure 8-29: Newton method optimization example. We can see in Figure (c) that the parabola falls outside, and thus, minima will not be identified. This is a common problem that occurs a lot if we only rely on Newton's method.

The major problem with Newton method is its complexity of computing second-order derivative. Besides, sometimes the cost function is fairly flat, and the parabola takes a too large step that falls outside the function. This is another drawback of the Newton method. For example, look at Figure 8-29, in which we can see the parabola is too large for the mimina of the function. Therefore, the Newton method works when we are close to the minimum.



Some suggest using gradient descent, and when we get close to the minima, we can take some Newton steps to jump earlier toward the minima. Nevertheless, we do not see Newton's method implemented in modern algorithms because of its huge computational cost. Even algorithms that must deal with second-order optimization, such as TRPO and PPO (we will describe them in Chapter 13), use a trick to simulate Newton's method because constructing a Hessian matrix is computationally very expensive.

Early Stopping

In optimization algorithms such as gradient descent or Newton, the epoch number should be given by the user as a hyperparameter. Usually, in each epoch, the error is reduced until a threshold is reached, and then again, it starts to increase; see Figure 8-30. When such a thing happens, to preserve resources, it is better to force the model to stop its optimization and not go until the maximum number of iterations. It means that before all the requested number of epochs finishes when the error rate starts to get higher, we should stop; this phenomenon is called *early stopping*. An increase in the error rate is a sign of overfitting on the training data, and the algorithm passes the minima. Most implementations of the gradient descent objective function enable the user to specify having or not having an early stopping as a hyperparameter.

Figure 8-30 visualizes the need for early stopping; the more epoch encounters by the algorithm,

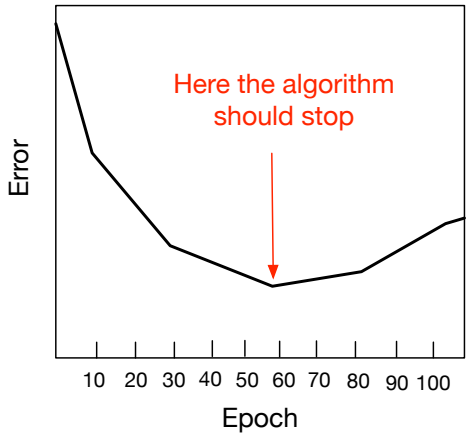


Figure 8-30: Increasing the number of iterations decreases the error until epoch 60, then the error starts to increase. This is where it is recommended to enable the algorithm stops by implementing "early stopping".

the error gets reduced until epoch number 60, in which the error starts to increase. A few iterations later, the algorithm should spot this and stop at epoch 60. This is called early stopping, and it can be categorized as a regularization method.

NOTE:

* The minima data point is where the gradient is zero, but a zero-gradient does not imply necessary optimality.

* Inflection points are when the curve of the function goes from downward to upward or vice versa. Where the derivative is zero, but there are not necessarily local or global minima points. Figure 8-31 presents some examples of inflection points that are not minima or maxima.

* According to Geron [Geron ’19], a model is composed of a set of features, and while using gradient descent, we should ensure that all features have a similar scale.

* Gradient descent for linear regression easily gets to the global minima because the cost function of linear regression is a convex function and it has a bowl shape. Therefore, there is no challenge of being stuck in local minima, never reaching global minima, etc.

* SGD is good for non-convex cost functions and reduces the chance of getting stuck in local minima.

* Gradient descent optimization algorithms assume the training dataset data objects are independent and identically distributed (i.i.d). Therefore, to ensure they are not used by the algorithm in any sorted or ordered approach, shuffling the dataset before feeding it into the gradient descent algorithm is recommended.

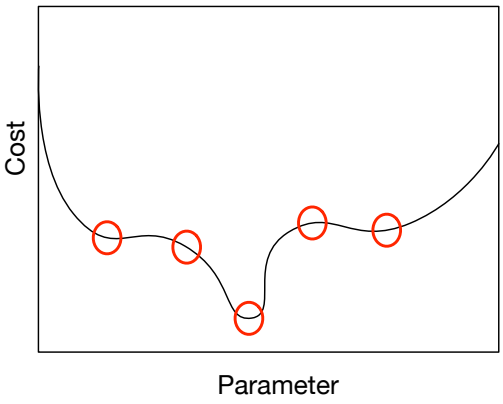


Figure 8-31: Some examples of inflicting points. At the inflicting point, the curve direction changes from upward to downward or vice versa.

* Gradient descent is a type of *hill climbing* algorithm, a popular optimization algorithm. However, gradient descent uses the slope in the local neighbors and then chooses the point with the steepest slope; hill climbing uses a cost function in the local neighbors and chooses the point with the lowest cost score.

Summary

Supervised learning involves regression and classification. In this chapter, we have explained regression algorithms. First, we introduced some concepts, including objective function, epoch, and batch. The objective function is used to maximize or minimize cost model parameters. The cost function is an objective function that tries to minimize a variable. The loss function is similar to the cost function but only for one single data point. A loss score is a quantitative value to measure the difference between the real function and our fitted function.

When an entire dataset is read by the machine learning algorithm, in the context of regression analysis and neural networks, we call each pass an epoch. The number of data points used for training is referred to as a batch. Batch size specifically refers to the size of the subsets the dataset is divided into. For example, dividing a dataset of 1000 data points into two subsets means we can have two batches, and analyzing these two batches takes one epoch to complete.

Linear regression is the simplest and most classical regression model. It is a parametric model. We generally estimate the parameters (slope and intercept) of this model based on its cost function; a basic and common cost function is RSS (Residual Sum of Squares). Whether it is a univariate linear regression model or a multivariate linear regression model, they are basically the same. However, Multiple regression involves knowledge of F-statistics, Forward Stepwise Selection, and Backward Stepwise Selection.

Polynomial regression is also a regression, except that the linear regression plot is linear, while the polynomial regression plot is non-linear. Nevertheless, due to its computational complexity, it is less popular than linear regression.

Condition	Regression Algorithm
Continous Dependent Variable	Linear Regression
Not Continous Dependent Variable	Pieceweise or Polynomial Regression
Categorical Dependent Variable with Binary output	Logistic Regression
Categorical Dependent Variable with more than two output	Multinomial Logistic Regression (e.g. Softmax regression)

Table 8-3: Regression algorithm and their use cases.

Logistic and softmax are called regression, but in reality, they are classification models. Logistic Regression and Softmax Regression are such models. "Logistic Regression" is a well-known binary classifier whose function is also frequently seen in neural networks. Parameter estimation of "Logistic Regression" generally uses the MLE method, which requires a logarithmic likelihood function. "Softmax Regression" is a multi-classification model, and its relationship with logistic Regression is similar to that between ordinary linear Regression and multiple linear Regression. In addition to these regression algorithms, we described the ARIMA model, to complete your mental shock. ARIMA is used for time series extrapolation (or time series forecasting).

Table 8-3 summarizes when to use which regression algorithm. Afterwards, we describe model parameter estimation for each algorithm. Besides, we described methods to evaluate the result of regressions, including k-fold cross-validation, ROC Curve, Pseudo R2, Wald Test, Information Criterion (AIC and BIC), and Likelihood Ratio Test.

Next, we list challenges related to training models, including bias-variance tradeoffs, overfitting, and underfitting. Then, we moved to regularization algorithms, which are used to reduce model parameters, and described Ridge, LASSO, Elastic Net, and Non-Negative Garrote regularization algorithms.

The last part of this chapter explained some algebraic concepts required for optimization and two optimization approaches: Gradient Descent and Newton. Gradient descent is used when we cannot use an algebraic optimization algorithm to calculate the best minima, and we should search for the solution (similar to the heuristic problems explained in Chapter 6). We have explained three types of gradient descent, which are listed as follows:

- Batch Gradient Descent: Batch size is equal to the training set.
- Stochastic Gradient Descent: Batch size is equal to one data point.

- Mini-Batch Gradient Descent: 1 < Batch size < Training set size.

We also explained Newton, which could be used in combination with Gradient descent. Newton is second-order derivative optimization and, unlike Gradient descent, it is independent of step size. However, due to its computational complexity, we should avoid using it directly. Some mathematical tricks could be used to approximate Newton optimization while not using second-order derivatives.

From now on most of the algorithms we learn in machine learning, operate as follows:

(i) makes some random guess about model parameters.

(ii) Use objective function to compute an error.

(iii) Until specified maximum epoch reaches or the error is less than some specific value, use a method to change guess parameters in step (i) and continue step (ii).

Further Readings and Watching

* There is a book by James et al. [James '13] that has a very detailed and easy-to-understand description of linear regressions and their variation.

* If you are interested in learning the math behind logistic regression there is a good tutorial provided under this link: https://www.hackerearth.com/practice/machine-learning/machine-learning-algorithms/logistic-regression-analysis-r/tutorial.

* Joshua Starmer has a very easy to understand detailed explanation about logistic regression and how a maximum likelihood function can be used to identify its parameter. Also, his explanation about regularization is amazing. His videos are available at: https://statquest.org, and in contrast to his songs, his tutorials about regressions are all fantastically useful.

* There is a very good example and explanation about Polynomial regression available online at http://mathforcollege.com/nm/videos; you can use it to learn how polynomial parameters were estimated.

* If you are interested in learning the ARIMA model in detail, Bob Nau has an online tutorial on his homepage https://people.duke.edu/~rnau/411home.htm

* Brandon Foltz has a good series on explaining logistic regression with examples, if you are interested in learning more detail about that. https://www.youtube.com/channel/UCFrjdcImgcQVyFbK04MBEhA

* The hundred-page machine learning book [Burkov '19] has a good introduction to the mathematics required for optimization, and we used it to construct the initial introduction of the optimization as well.

* If you would like to read more about Gradient Descent and its implementation, Geron [Geron '19] has a good summary of gradient descent with Python codes. If you are interested in delving deep into optimization techniques and their mathematical concepts, the Algorithms for

Optimization book [Kochenderfer ‘19] is a proper reference to read, but only if you intend to go deep into this topic, and be sure you are able to grasp all of its mathematical concepts.

* Another good resource for learning Gradient Descent and comparing different methods together is Jason Brownlee’s page, https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size.

* If you intend to learn more about the basics of mathematics and algebra Nancy Pi has a very good video series on that. Her explanations are easy to understand. http://youtube.com/nancypi