Chapter 8: Regression, Regularization, and Optimization

We have explained in Chapter 1 that there are two main categories of supervised learning algorithms, **classification** algorithms, and **regressions** 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] state: *"classification is about whether something will happen, whereas regression predicts how much something will happen."* In another good book, James et al. [James '13] state that classification can be used to predict categorical (qualitative) variables, but regression can be used to predict numerical (quantitative) variables.

Nevertheless, the border is blurred and a machine learning algorithm can belong to both 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 have to get familiar with "cost", "loss" and "objective" functions. 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. 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 we describe the famous Gradient Descent and Newton-Raphson methods.

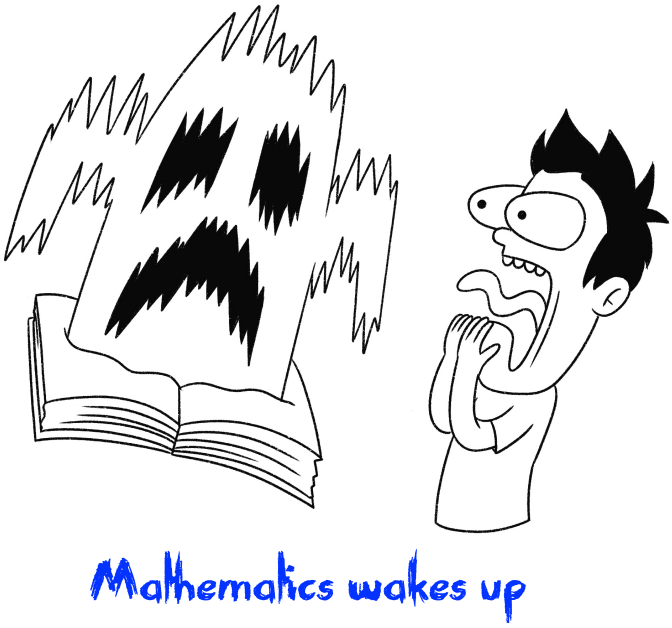
Objective, Cost, and Loss

Anything we do in our life is associated with a cost. For example, you decided to be an expert in data science and then make a wise decision by starting to reading this book. There are costs associated with spending your precious time and enforcing your brain cells to consume energy. In other words, you spend time (cost) for learning machine learning (gain). You may have decided or will decide to get married, the cost is your free time, which will be dedicated to

another person. Also, to have a successful marriage, we (and everybody else) should make some changes to our behaviors as well, and it is impossible to keep the same behavior as we were single.

Something similar existed in algorithms as well. A mathematical function that *tries to maximize or minimize a variable* (e.g. accuracy of an algorithm, battery use of an algorithm, …) is called an **objective function**. In the context of machine learning, we can call any function objective function, if a model is trying to optimize this function. A simple mathematical function is written as $y = f(x)$, in which x is our input data and y is the output data.

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.**



Loss score is a score that defines the cost of a *single data point* and measures the cost of that single data point. In layman's terms, we can say loss functions measures how many mistakes we made. The cost function is used to measure the sum of the losses (mistakes) of all data points, but the loss function focuses on the costs of a single data point.

In simple words, the differences between actual output values (y_{actual}) and predicted output values ($y_{predicted}$), specify the cost. One single difference is the loss and all loss values are constructing the cost.

In summary, we can say that a loss function is a subset of a cost function, and a cost function is a type of objective function. However, costs, losses, and objective functions can all be used interchangeably.

Every iteration through the entire training set for calculating the cost function is called **epoch**. If you encounter a text that said after 100 epochs, this means that the algorithm reports the result of cost function after 100 times it has iterated the entire training dataset.

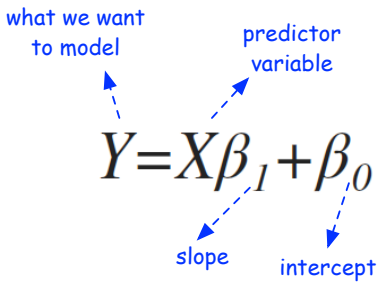
Linear Regression

We hope you remember in Chapter 3 we have described that to find a relation between two variables we use correlation analysis. However, sometimes there is a relation among variables but it is complicated (like those relations between teenagers), in these cases, we use linear regression. There is a significant difference between correlation and regression: correlation

assigns a score that specifies the relationship between two variables, but regression describes how do changes in one variable affect the other variable.

In other words, instead of assigning a correlation score (what we do in correlation co-efficient estimation methods), we study how changes in one variable, x, causes changes on another variable, y. We start from the simplest form of regression, which is 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 formula, 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 for example $\hat{y}$ 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 (slope) and β_1 (intercept). Note that they are not hyperparameters[1], because the model will use a cost function to identify them and they are not configured by users of the algorithm. These types of parameters are called **model parameters,** or it is also called **model weights**.

In other words, model parameters are parameters whose optimal value will be identified by the algorithm itself and not the user of the algorithm. Therefore, we do not need to worry about tunning these parameters, it is the job of the optimization algorithm (by using cost function) to tune model parameters. Later in this chapter, we describe the optimization algorithm.

It is common to use θ, which is a vector, to present all model parameters.

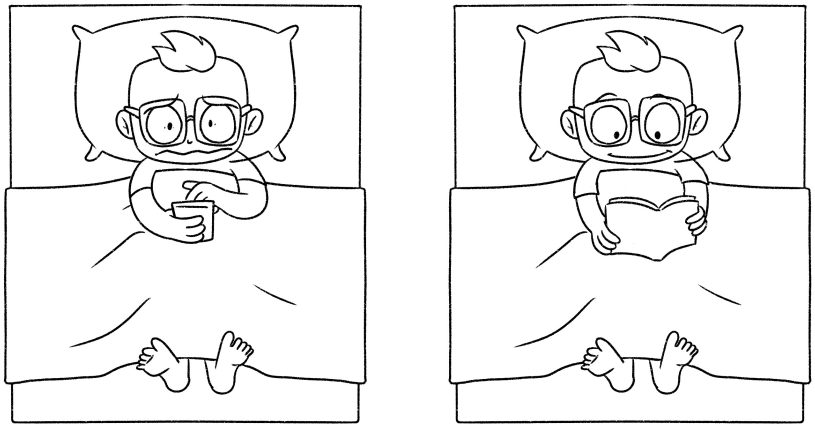
We can use linear regression or other linear models to perform prediction, describe a phenomenon, or even measure the effect size (see Chapter 3 to recall the effect size).

Let's look at an example scenario with linear regression. Recently, Mr. Nerd begins to suffer from work stress. To solve his stress, he decided to make some changes in his life, and he starts to read books, before bed, instead of wasting his time on social media and fighting the holy war with online trolls.

He speculates that reading the book could positively change his mood, i.e. 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, so he decides to read 20 books to have a fantastic mood. But, now he has finished the 9th book, he becomes curious to predict what his mood will be after

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



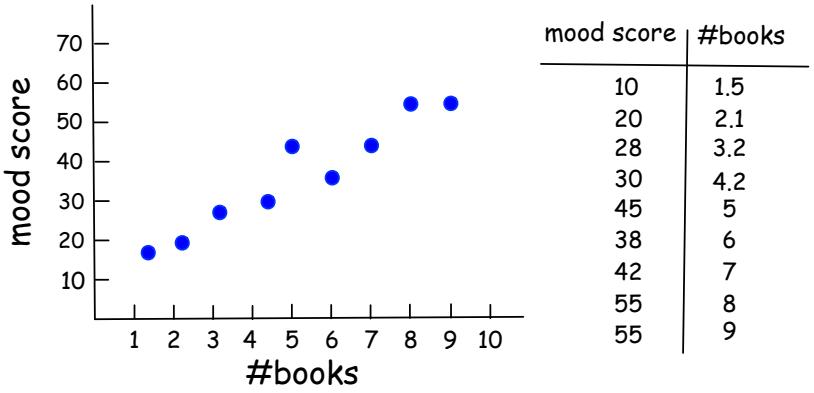
the 20th book? He can use linear regression to answer his question, The question will be as follows: "what will be the mood score after reading 20 books?" A linear regression of his prediction model can be written as follows: $mood - score \approx \#books \times \beta_1 + \beta_0$

The objective of linear regression is to find the optimal values for β_0 and β_1. 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 20 books, via replacing these parameters in the linear equation. It will be calculated as 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 identify the correlation between input (predictor) and output variables. This will be reported as a hypothesis test and its p-values. Please, 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 case described for Mr. Nerd, the p-value < 0.05 and thus we claim there is a relation between the number of books he read and his mood score.



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.

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 $O(m^2(n+m))$, which is calculated based on matrix decomposition and inversion. We don't need to know the detail, just remember that a linear regression has *quadratic* complexity.

Model Parameters (Coefficients) Estimation

Now we have answered his question, 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.

Figure 8-2 presents the described linear regression with its parameters highlighted. Check the red lines connected each data point to the regression line in Figure 8-2 those e are called **errors** or

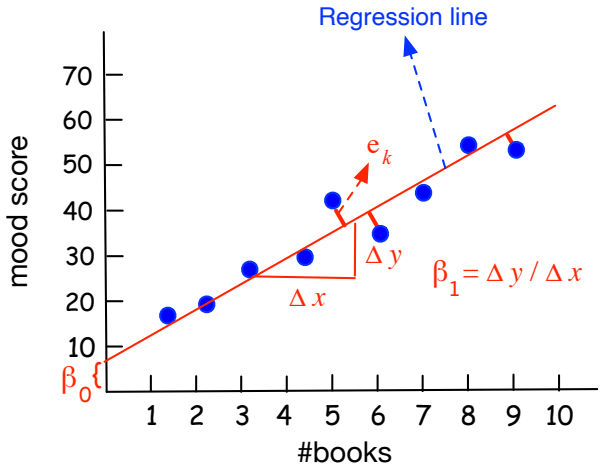


Figure 8-2: Figure 8-1 with regression line and its parameters.

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 it could be written as follows:

$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2)^2 + \ldots = \Sigma_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The mean of x is $\bar{x}$ and mean of y is $\bar{y}$, 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.

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

$$\hat{\beta}_0 = \bar{y} - \beta_1 \bar{x}$$

Remember that we should minimize the cost function and the cost function here is RSS.

The mathematical notion of linear regression is fairly easy to learn and it is worth reading them and understanding the rationale behind linear regression, despite your regression library wrapping all details in a simple function call.

Since we have learned what is error, it is also worth mentioning to be more accurate 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, it is better to use approximation instead of the equal sign, because there is no guarantee that that created regression line is the perfect line, so if you like to avoid generalization even in your mathematical writings and show-off you are knowledge, write the linear equation 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: $y = x^T \hat{\beta}$. In the near future (Chapter 10) β_0 will be written as b and other intercepts (βs) are vectors of weights *w, i.e.,* $y = wx + b$. Because mapping them is easier than the neural network concept.

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**. Now, we describe only one cost function, but as we progress through this chapter we explain more cost functions.

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:

$$Y = \beta_0 + X_1\beta_1 + X_2\beta_2 + \ldots + \epsilon$$

or we can summarize the summation as follows:

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

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 are contributing to his daily stress, including work, use of social media, 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 it has been 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. While employing multiple linear regression, usually, we need to answer some more questions [James '13] Chapter 3. For example,

is at least one of the input variables, i.e., X_1, X_2, … useful for predicting the *Y?* Do all *Xs* (input, independent, or predictor variables) help to explain *Y* (output or dependent variable)? Or, how well does our model fit the data? To answer these questions we need to rely on the output of our software application which implements linear regression, including the *p*-value given for each input variable or F-statistic test.

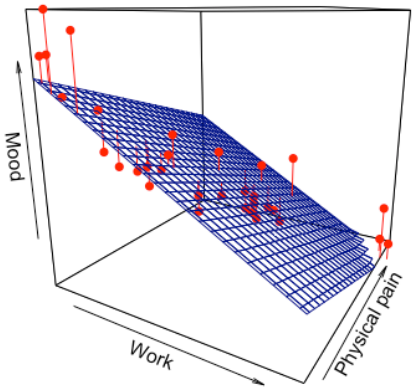


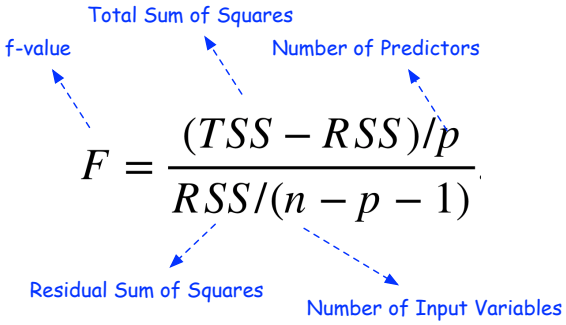
Figure 8-3: A multi linear regression example for Mr. Nerd and his mood score.

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

H_0 = There is no relation between input and output variables ($\beta_1 = \beta_2 = \ldots = \beta_p = 0$).

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:



$TSS = \Sigma_{i=1}^{n}(y_i - \bar{y}_i)^2$, The *f-value larger than 1 means the null hypothesis is rejected (the alternate hypothesis is correct)*. Otherwise, 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 both *p*-value and *f*-value should be significant to accept the model.

Deciding About Model Variables?

F-test is very useful for multiple linear regression. Assume the f-test failed 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 will be raised: how to 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, 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 test another model x_2, x_3. Or we can remove x_2, x_3 and test with x_1, or … 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 boost the product sale 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 their *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 model and predict numerical values. However, 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 employs linear regression. That is not wrong, but what will happen if we can not draw a regression line in between those binary states? The shape of the data points does not allow the algorithm to draw the linear regression line.

Another issue is that 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. This 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 patients in an emergency room, 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 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. For now, just keep in mind for the categorical variable we use logistic regression.

The second problem is that linear regression is designed for one feature, and it can not handle more than one feature. The version of 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 this will be handled by *polynomial regression*.

Polynomial Regression

Sometimes the relationship between predictor and response variable is not linear. Assume Mr. Nerd loves reading books, but either he should read too few books or many books each month, otherwise his stress level increases. He likes to keep a balance on his book reading behavior. Again he logs on the number of books he is reading per month, and he plots them as it has been 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 (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. For these scenarios that a straight line (linear regression) is not enough and instead we can use polynomial regression.

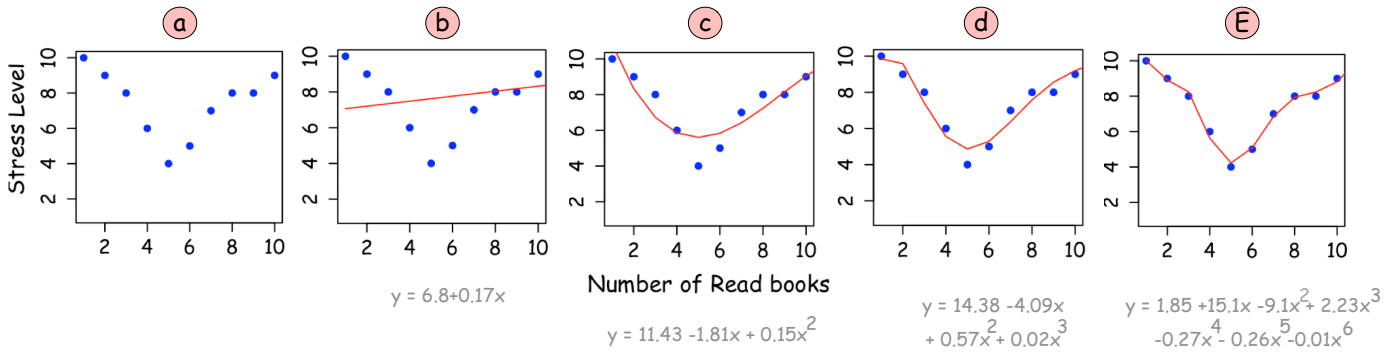
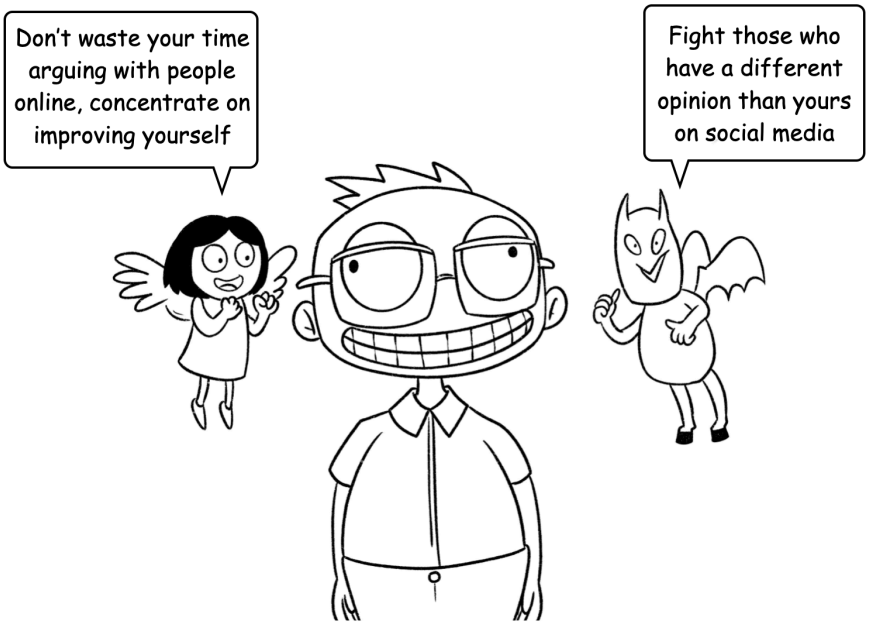


Figure 8-4: (a) plot of stress level and number of read books in a month, (b) linear regression on the data, which is not properly representative of the data (under fitting). (c) polynomial regression line with the degree of four (d) polynomial regression line with the degree of six, (e) polynomial regression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

A series of mathematical terms (coefficient, e.g. βs, multiplied by a variable) which are joined together by addition or subtraction is called a **polynomial**. Please keep this in your lovely brain: *polynomial regression can produce a non-linear curve.* The polynomial equation for linear regression is written 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 enables us to draw a curved line that fits into the data points. In this 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 shape 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 you can see the more we increase the degree of regression, 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 will be explained 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 $((n+d)!)/(n!.d!)$ features [Géron '17]. Therefore, if computational complexity is important, we should be careful and not increase the degree of polynomial regression too generously.

Model Parameters (Degrees and Coefficients) Estimation

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

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

For example, let's say we have a four degree polynomial regression. By substituting the values for x and y (from our real dataset) and by solving the following matrix we can identify $\beta_0, \beta_1 \; and \; \beta_2$ (we learned in school how to resolve three equations that have three unknown variables):

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} x_i y_i^2 \end{bmatrix}$$

Note that we have the values of x and y available from our observed dataset, so we can substitute them, then perform matrix multiplication and end up having three equations with three unknown variables, i.e., $\beta_0, \beta_1 \; and \; \beta_2$. By solving this we understand the values for coefficients.

For example, let's assume that we end up with the following variables
$\beta_0 = 0.1, \beta_1 = 0.3, \beta_2 = 0.12.$

We like 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:

$y = \beta_0 + \beta_1 \times x + \beta_2 \times x^2$ 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. As we have explained before, increasing the number of degrees is associated with a huge computational cost, so we should find a reasonable degree. 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 not to use it.

(2) Add back those removed data points and check how much an error value (e.g. RSS) changes. For example, assume x^2 error = 0.28, x^3 error = 0.23, x^4 error = 0.21 and x^5 error = 0.46. Then we can say x^4 or quadratic polynomial regression is good, because it has the least amount of error while encountering the new data.

This approach is very similar to 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 train dataset and not the test dataset.

Usually, the software which implemented polynomial regression provides us a parameter search function to automatize this process.
page break:

Piecewise, Segmented, or Non-Additive Regression

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

The data points presented in Figure 8-5 are not additive, i.e. around 110 calories his joy increases, but then it starts to decrease. Therefore, we can not model it with simple linear regression. However, by separating the dataset into two datasets, we can easily use two linear regression lines that fit the data appropriately. Figure Figure 8-5 b presents the separator line, and Figure 8-5 c presents these two regression lines.

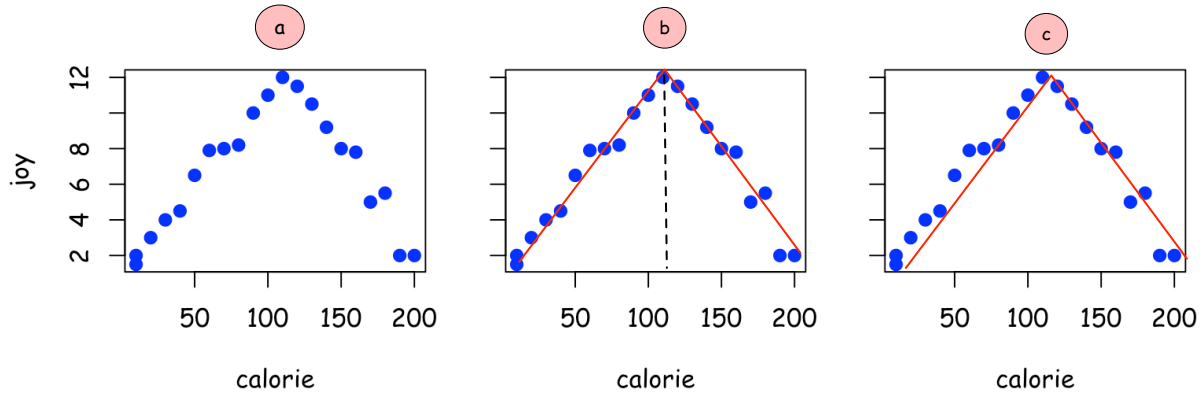


Figure 8-5: (a) A dataset that can are not additive and thus we can't use a simple line to fit them (b) the knot is specified that separates the dataset into two sub datasets (c) The same dataset that can be modeled with two linear regressions.

Even sometimes, we can use more than one polynomial regression to model our dataset and it is not just limited to linear regression. Now, the question is "at what point do we need to separate the dataset?" By visual inspection (a sexy term for using our eyes) we can recognize where in this small dataset we can spot the distinctive point. For real-world dataset, which is usually too large and multidimensional, however, we do not have such a luxury. 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 or F-statistics, or other evaluation metrics for each of these regression lines. We change the data point to another data point from the dataset and identify some evaluation metric. The

minimum sum of an evaluation metric will be the best discriminatory point to separate the dataset into two subsets and use two linear models.
For example, assuming we have a dataset and we select randomly three data points of it to separate it into two subsets.

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

Among the three following data points, i.e., d_1, d_2, d_3, d_2 is the best data point for separation because it has the minimum sum of F-statistics.
The point at which one linear regression is broken and another one is started is called *knot*. We can also use the piecewise regression for polynomial regressions as well (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.

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

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

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, because of different reasons including giving biased regression coefficients that need shrinkage (we explain it 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.

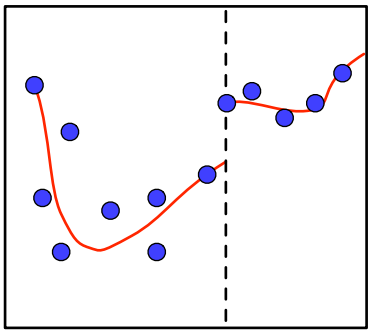


Figure 8-6: Piecewise polynomial regression example.

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], Chapter 7. Also, to identify the optimal number of knots we could rely on F-statistics as well.

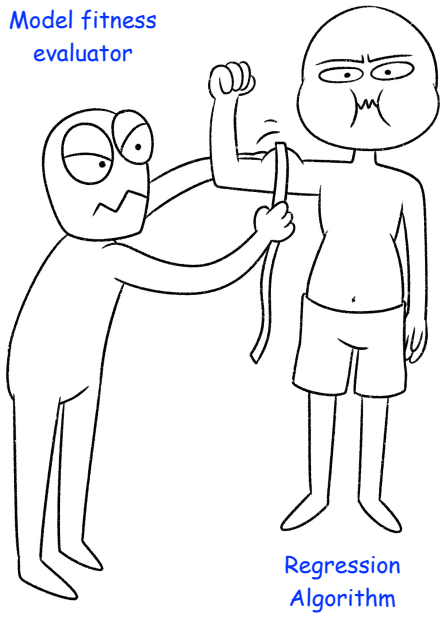
Besides, keep in mind a quote from Mr. Obvious: two polynomial regression splines, usually perform better than a single polynomial regression.
page break:

---

[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. Some times the index is not written as subscript and if it is written as super script it should be inside parentheses.

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 fitness** or **fitting a model**.



In the following we describe some approaches used for evaluating linear models' fitness, Residual Standard Error (RSE), Coefficient of Determination or R-squared (R^2) and Root Mean Square Error (RMSE),.and Mean Square Error (MSE).

These tests are good for the *training set*, and to evaluate the *test set* we need more subtle approaches, which we will explain later. There are many more methods available, such as F-score, but we explain them for the evaluation of linear types of regressions.

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

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

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 predicted value which is a data point located on the regression line). The closer these two variables (y_i and $\hat{y}_i$) are, will result in having better model at the end. Thus, a smaller RSE is better. The *RSE* for Mr. Nerd 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 it 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. $\Sigma_{i=1}^{n}(y_i - \bar{y}_i)^2$ divided by **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:

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

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, this means that the R^2 evaluation shows high accuracy.

Root Mean Square Error (RMSE): 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 ith 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 formula as RMSE, just its square root has been removed. In some implementation libraries, MSE is supported and not RMSE.

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 to be 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 more 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, and this is the disadvantage of parametric methods. Nevertheless, since the non-parametric method does not have any assumption 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 them later. Since they are parametric they do not need a large amount of data to train the model and operating with small dataset makes them very attractive algorithms.

* There are many other cost functions that can be used for linear regressions, we describe the F-Statistics, RSE, RMSE, and R^2 costs. These are called cost functions, which we will explain in this chapter.

* 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).

* Some time to identify which model, from a set of models, can fit better to 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 the sign that there is a non-linearity in the dataset. However, we told you 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 a 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 variables that are highly correlated 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 has been recommended 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 it. More about overfitting will be described later in this chapter, now just keep in mind that polynomial regression is prone to overfitting.

* Polynomial regression is more sensitive to outlier and noise in comparison to linear regression. This could be another reason that usually linear regression is favored over polynomial regression.

* Nancy Pi[3] has interesting quotes in her online videos, *"much of the calculus is to estimate something non-linear by using something linear"*. This could be another motivation behind using linear regression significantly more than other regressions, especially among data scientists who have 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.
* A more general name for regression is an approximation. Approximation is the process of finding a good estimate about a value of the function at a certain unknown point, by leveraging the information we have at a known point about the function.

page break:

[3] http://nancypi.com

ARIMA (Auto Regressive Lag Integrated Moving Average)

ARIMA is a very popular machine learning algorithm used for predicting or modeling time series. Since we are talking about linear regression it is good to describe this popular algorithm as well. If you are not interested in working with time series, feel free to skip this section. Before we explain ARIMA we need to explain some concepts that are used by the ARIMA algorithm.

Extrapolation and Interpolation

Extrapolation is the process of estimating the next change of a variable, beyond the original 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 of the observed values (blue ones), which could be called forecasting time series as well.

Another term is an **interpolation**, which produces an estimate of potential values between observations data points, such as red dots that are generated between blue dots in Figure 8-7 (right).

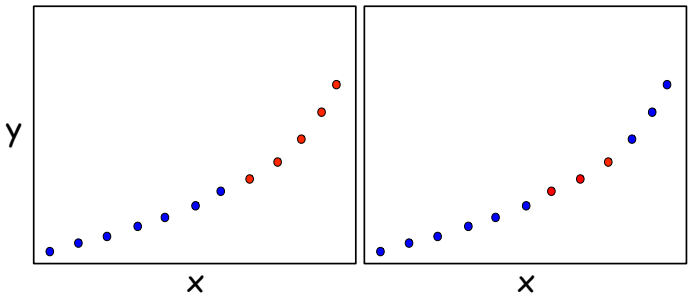


Figure 8-7: (Left) toy example of extrapolation, (Right) toy example of interpolation.

ARIMA algorithm

ARIMA is used for time series extrapolation (or time series forecasting) algorithm. It assumes that the later data points can be determined by using the previous 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.

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*. The rest of this subsection explains this equation in more detail.

As the first step, ARIMA converts a none-stationary time series into a stationary time series, which is called *differencing* the time series. In other words, we use differencing to de-trend a time series. This conversion is required because there should be a stationary model to use lags for predicting the future data points. This refers to AutoRegressive (AR) characteristic of ARIMA. The differencing process will be 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 more data points ($y_{t-1}, y_{t-2}, \ldots$) from the current point, this number is specified by d. In short, parameter d specifies the *order of differencing* required to make the time series stationary.

In other words, d specifies the minimum number of differencing required by each data point to transform the non-stationary time series to stationary time series. If the original time series is stationary we have $d = 0$.

For example, considering y'_t is the differences between lags (check Chapter 6 to recall lag), time t, we can write the following:

$d = 0: \quad y'_t = y_t$

$d = 1: \quad y'_t = y_t - y_{t-1}$

$d = 2: \quad y'_t = y_t - y_{t-1} - (y_{t-1} - y_{t-2})$

y'_t values are used to transform the original time series, which is not stationary into a stationary time series.

Parameter p specifies the order of autoregressive term. 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, which y'_t *depends on its p number of lags*. Therefore, we can write the following equation for predicting y'_t based on pure autoregression, only AR and not moving average (MA). $\hat{y}'_t = \beta_0 + \beta_1 y'_{t-1} + \beta_2 y'_{t-2} \ldots + \beta_p y'_{t-p} + \epsilon_1$.

In this equation, βs are coefficients of their associated data points (lags) and β_0 is the intercept.

The parameter q specifies the order of moving average (MA) term. As we have learned in Chapter 6, a "moving average" depends on lagged data points. We can say "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:

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

We just understand the intuition behind the ARIMA equation, now we should explain how to choose the best values for d, p, and q.

ARIMA Parameters Estimation

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 (check Chapter 3 to recall correlation) is calculated against lag variables (e.g. correlation between d_t and d_{t-1}) it is called **autocorrelation** or **self-correlation**. In simple words, autocorrelation means that the signal or time series is correlated with itself.

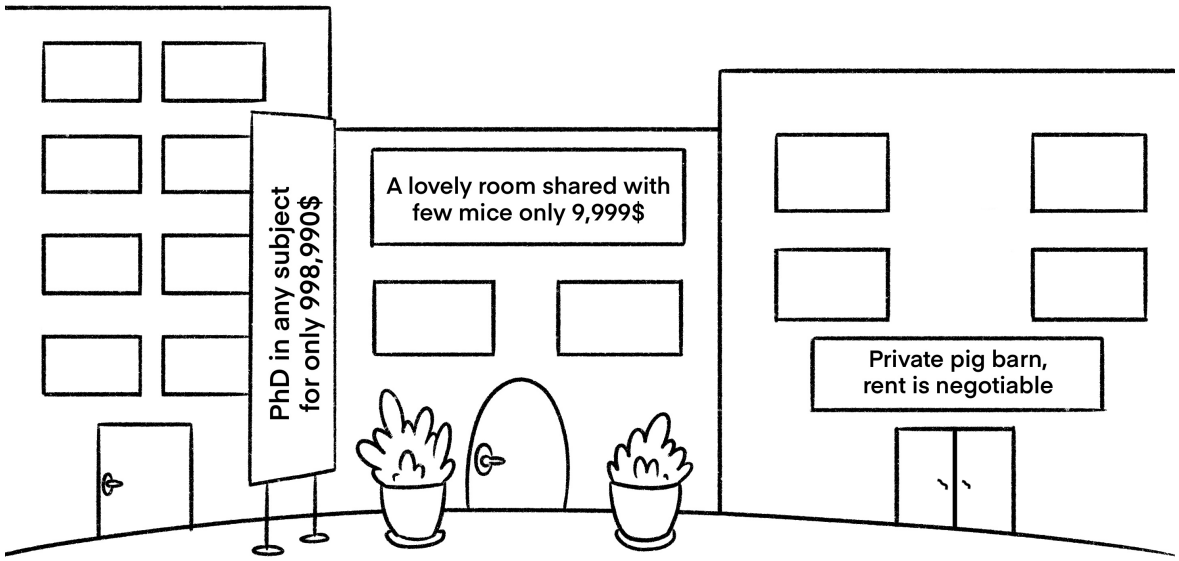
To better understand this let's use a sample, we have $X = \{1,2,4,5,6\}$and $Y = \{2,3,5,7,7\}$ if we calculate the correlation coefficients of these two sets its Pearson correlation will be $r = 0.9834$, which means they are highly correlated. Now let's say we would like to measure the correlation of X with itself, but with one lag. So 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 shift 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 *Auto Correlation Function plot* (ACF plot), and your software package usually has it. ACF plot visualizes the auto-correlation (correlation between the original series and its lagged values). It presents how well does 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 presents the lag and you can see in Figure 8-8. The lag value that is located outside the significant area (blue area) should be chosen because they are statistically significant. For example, only lag 1 is significant in "autocorrelation of first-order differencing", the rests which are located inside the blue area are insignificant, lag 0 is also located outside the blue area, despite it is showing a significant size.

To determine the stationarity of a time series a statistical test called the Augmented Dickey-Fuller test or ADF test 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 none-stationary. If the p-value > 0.05 then we need to increase the order of differences until the ADF test shows that it is stationary (p-value < 0.05).

ACF plot measures the correlation between observation (data) at current time and previous observations (data), in other words, it is doing the MA part of the ARIMA. There is another plot, called *Partial Autocorrelation Function plot* (**PACF plot**). PACF plot finds a correlation of residuals. It might sound a bit hard 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 went back to their country are proud to get their degree from well-known universities. Let's call this imaginary city Bustom.



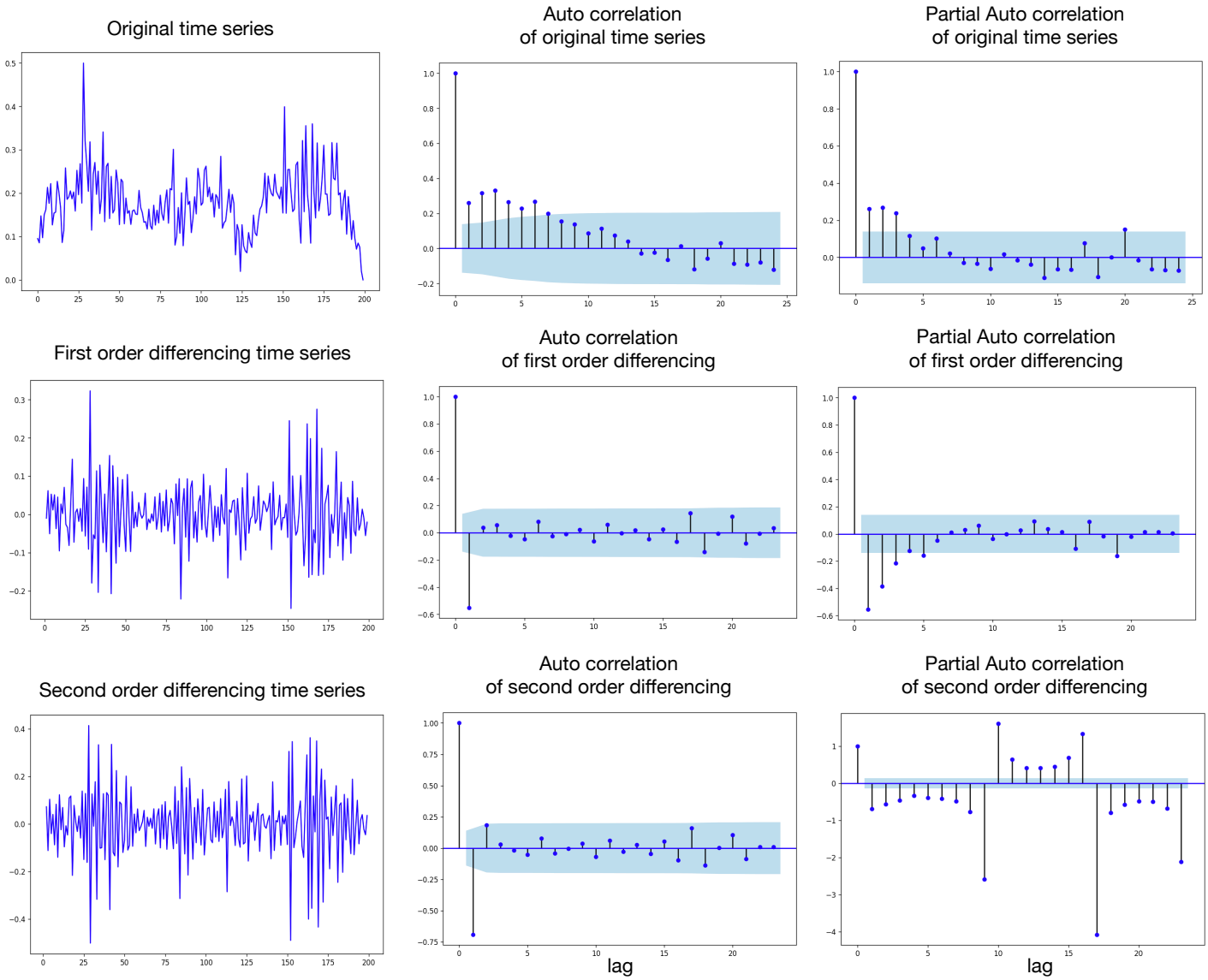


Figure 8-8: (Left) original time series, first order differencing and second order differencing (center) ACF plots of each time series. (Right) PCAF plot for each time series on the left side. The light blue color background is called confidence band and tell us whether the correlation is statistically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

Most of the Bustom population are students. God forbids, assume you have arrived late 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. Prices of December correlates to September too, and not October. Therefore, in this case, 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).

Take a look now at Figure 8-8, there 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, looking at Figure 8-8 you can see that the ACF plot for second-order differencing is moving toward zero, and in comparison to the original time series its standard deviation is also closer to zero, so 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 second order of differencing moves toward zero (better than first order of differencing), so we choose *d=2*. If both first order and 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 have used ADF test. To determine a proper value for *p*, we will use PACF plot (AR part of ARIMA).

We choose a *p* based on the significance test reported by 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} is significant we chose d_{t-2} to predict d_t. The left side of 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 can 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 numbers 1 fall outside the blue area in ACF plots and this means it is statistically significant. 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 favor using this method instead of cross fold validation. page break:

Logistic Regression

Before we explain Logistic regression, we should be familiar with Sigmoid function.

Sigmoid function or sigmoid curve is a function that gets data as input and output data in a S-shaped curve. It is written as $\sigma(\ )$ and it calculates 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:

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

We are lazy and thus we do not show more example here, but we recommend you to use more values for x and plot them yourself, then you will notice that this function returns something between 0 and 1 and its shape is very similar to an S-shaped curve.

Logistic (or Logit) regression (Classification): It is used when our output (dependent) variable (Y) is binary. Logistic regression has a binary output (e.g. yes/no, open/close, die/survive, spam email/not spam email), as opposed to linear regression which has a numeric range.

When we write linear regression inside a sigmoid function it presents a logistic regression, therefore we can write $\hat{y} = \sigma(\beta_0 + x\beta_1)$.

Instead of directly modeling the value of Y (identifying a numeric value for Y), logistic regression assigns a probability to each class (output). Therefore, this it is not used for predicting or describing a continuous variable, instead, it predicts or describes a binary variable. In other words, it specifies the probability if a data point belongs to a class or not and thus it can be used to solve the classification problem.

With some mathematical calculations, which we skip describing here, we can derive the following equation of 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 in the university and due to pandemics (Covid-19 started in 2019), the grade of the course is based on assignments. He either likes a course or doesn't like it (only two possible classes). If a course has more than 10 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:

$=>$ 10 *assignments: don't like*

$<$ 10 *assignments: like*

Let's assign "like" to 0 and "don't like" to 1. In particular, as it is shown in Figure 8-9 a he has plotted his previous courses, based on like/don't like and the number of assignments he had. *X*-

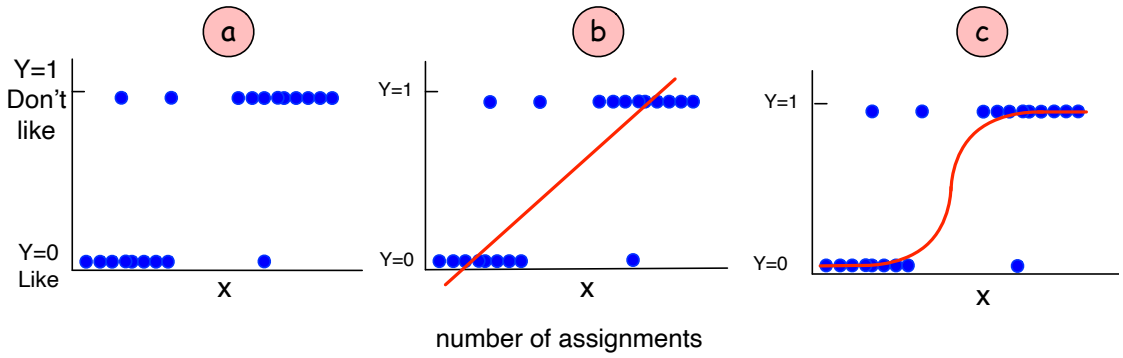


Figure 8-9: (a) Original dataset, x presents the number of pages each book has (b) linear regression plotted, (c) Logistic regression Sigmoid line plotted.

axis (independent variable) represents the number of assignments and on *Y*-axis we have two values *like* (*0)* or *don't like* (*1)* (independent variable).

Figure 8-9 b. presents a linear regression line, 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 say 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, something like the line Figure 8-9 c. By using an S-shaped line (the result of Sigmoid function), as it is shown in Figure 8-9 c., it can cover more data points than a straight line (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 "Mr. Nerd will like it", or "Mr. Nerd will 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 consider we give data for 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 8 assignments, as follows:

$\hat{Y} = \frac{e^{0.4+0.3\times8}}{1+e^{0.4+0.3\times8}} = 0.94$, 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 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 β_0 (intercept) and other β parameters that fit the sigmoid line.

Logistic regression uses a Maximum Likelihood Estimation (MLE), which we have introduced back in Chapter 3, to estimate the optimal model parameters. Here the algorithm that implements MLE tries to identify parameter values, that are closest to the output probability (either 0 or 1). In other words, the algorithm tries to find a value for $\hat{\beta}_0$, $\hat{\beta}_1$, $\ldots\hat{\beta}_k$. 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 use 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 parameters of our model we have $\ell(\theta; X) = log\ L(\theta; X)$. 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.

Let's delve a bit deeper into the details of parameter estimation. 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[4]? 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 is a ratio of an event happening, to that event not happening. For example, the probability of rolling a dice and getting the number 6 is $1/6$, but the 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** or **Log-Odds** is written as follows[5], for the sake of simplicity we write p instead of $p(event)$:

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

In other words, to transform Logistic regression output into a linear model, the output of Logistic Regression $((e^{\beta_0+\beta_1 X})/(1 + e^{\beta_0+\beta_1 X}))$, which is in probability will be fed into the Logit function.

This makes the output variable of logistic regression (probability value) similar to the output of linear regression (numerical value). Figure 8-10 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 such a transformation, we will have a range between -∞ and +∞, instead of a range between 0 and 1. Now, our data points are located at -∞ and +∞ and still we cannot identify their

---

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

[5] $Log_e(x)$ 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.

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 8-11 a. and b. present that line.

If we can project our data points on this line, then, their coordinate can be in a range that linear

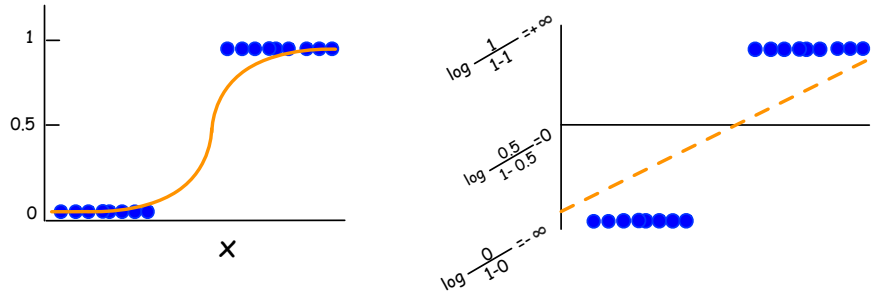


Figure 8-10: (Left) The logistic regression line and the sigmoid function is shown as an orange line. (Right) with the assistance of the logit function, we transfer data (blue) points, from 0 to 1 values, into -∞ (negative infinity) to +∞ (positive infinity). Now they are in a linear space and an imaginary line such as the dotted orange line is used to identify optimal value for coefficients, similar to the approach linear regression uses.

regression can use to calculate θ parameters (remember we refer to all β parameters as θ while working with MLE approach).

Once again let's review why do we need such a projection. Our data points are located at - ∞ and +∞. Therefore, their regression line will start from infinity to infinity and thus we cannot identify model parameters. The MLE can resolve the infinity problem, by projecting the original data points into an imaginary (candidate) line. See Figure 8-11 a, the original data points are projected

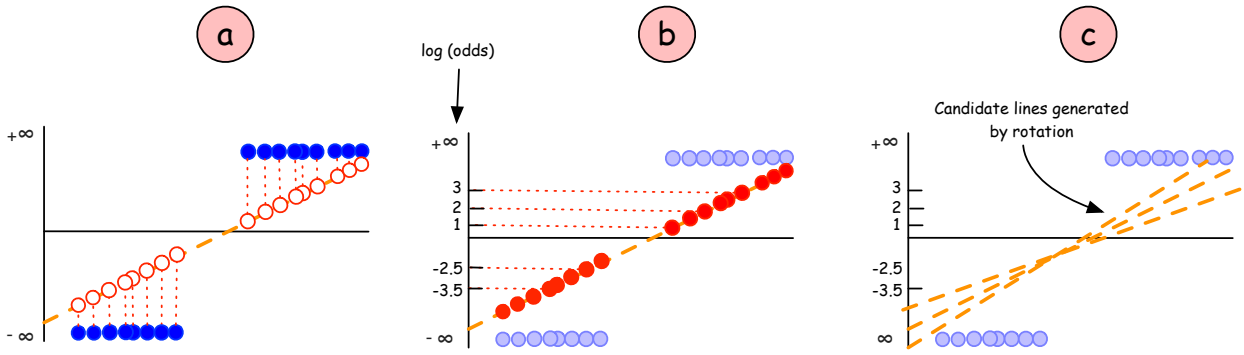


Figure 8-11: (a) projecting original data points from negative and positive infinity to a candidate line. (b) identify the Y-axis values of each project data point. (c) by rotation different candidate lines are generated and the line which has the lowest error will be chosen to identify the optimal model parameters.

on a candidate line and presented as red empty dots. In Figure 8-11 b, their coordinates to the Y-axis have been calculated. These values on Y-axis are *log(odds)* of the original data points. Then, we can transform these *log(odds)* back to probabilities by the following equation:

$$Y = \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

For example, let's take a look at Figure 8-11 there is one data point that has the 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 to 1 classes. If the data point belongs to 0 classes (the probably is less than 0.5) its likelihood will 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. $log(xy) = log(x) + log(y)$. Nevertheless, since the logarithm of numbers smaller than one is negative, the maximum log-likelihood negates the result at the end.

Check the following example:

$$log\ likelihood\ = -(\ldots + 0.73 + 0.56 + 0.92... + (1 - 0.35) + (1 - .42) + \ldots) = 3.21$$

This means that the log-likelihood of this particular candidate line is 3.21. Then, the MLE rotates a bit this candidate line and creates a new candidate line, then it calculates the log-likelihood for the new candidate line, e.g. 2.99, and 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.

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

Page break:

Softmax Regression (Classifier)

Similar to the 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 the animal (chicken, cat, dog, mouse, etc.). In other words, by using logistic regression, we have one output variable that is binary (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 being used for the multinomial logistic regression, including baseline logit model, adjacent category logit, proportional odds cumulative logit, and softmax regression.

One easy way to deal with more than two dependent variables is to convert them into two dependent variables and perform logistic regression. This means we convert a multinomial problem into many binomial classes and to solve binomial classes we can use 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). Again, one variable is "bike" and another variable is "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].

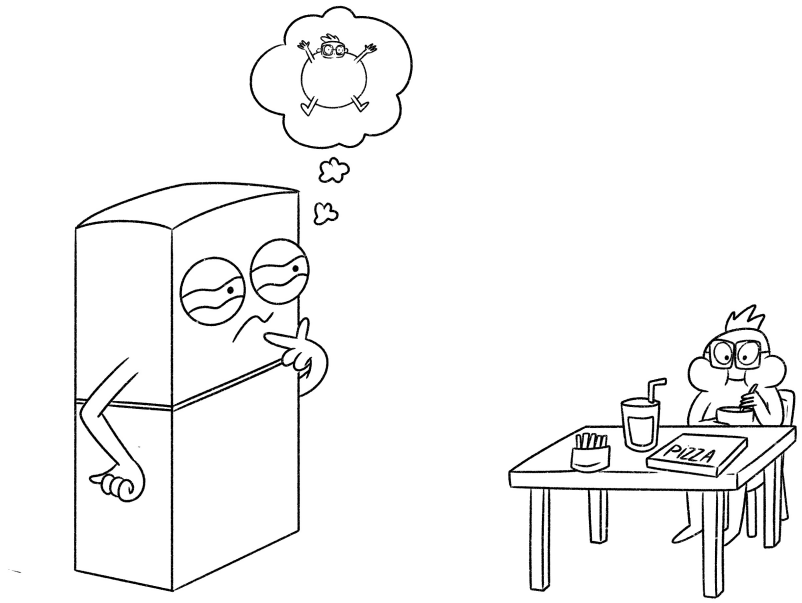
We describe **softmax regression** or **maximum entropy regression**, which is the most important one. We should be familiar with it because when in Chapter X we start to learn deep learning algorithms we will come back to softmax again.

Softmax regression is a generalization of logistic regression for K categories, so 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, 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 put 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 to 1), and their sum is equal to 1.

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



To better understand the detail of softmax regression we should take a look again at logistic regression function, sigmoid, and see how it can be generalized to be softmax. Let's get a bit more civilized and write again the logistic regression function (sigmoid function) as follows:

$\sigma(z)_i = \frac{1}{1+e^{-(z)}}$ assuming that $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_k x_k = \Sigma_{i=0}^{m} \beta_i x_i = \theta^T x$

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 (recall that θ presents all parameters of the model).

We see a transpose sign used for θ, why? Because θ is an $1 \times n$ 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 sigmoid function. For all k classes of output it is written as follows:

It is very similar to Sigmoid (except having a Σ sign in the denominator) but to better understand it, let's expand this equation as follows:

k = number of possible classes

vector of transpose of parameters (θ) multiplied by input values (x)

$$P(y=j|x,\theta) = \frac{e^{\theta^T x_j}}{\Sigma_{i=1}^{k} e^{\theta^T x_k}}$$

A vector of what we want to predict

$$\hat{P}_k \; or \; h_\theta(x) = \begin{bmatrix} P(y=1|x,\theta) \\ P(y=2|x,\theta) \\ \cdots \\ P(y=k|x,\theta) \end{bmatrix} = \frac{1}{exp(\theta^{(1)T}x) + exp(\theta^{(2)T}x) + \ldots + exp(\theta^{(k)T}x)} \cdot \begin{bmatrix} exp(\theta^{(1)T}x_1) \\ exp(\theta^{(2)T}x_2) \\ \cdots \\ exp(\theta^{(k)T}x_j) \end{bmatrix}$$

The output result of softmax regression is a vector of probabilities and their sum is equal to 1. The input of the softmax function is a vector of data objects and not an individual data object. Therefore, while using Softmax function *each output data object depends on the entire vector of input data objects*.

Unlike Sigmoid (S-shaped function) we can not use a specific shape or plot to visualize the softmax in more than two dimension space.

Let's get back to the refrigerator and Mr. Nerd's example. Mr. Nerd went into 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 linear regression, the cost function of softmax regression is the MLE. Before, explaining it we should go back to the cost function of logistic regression, maximum log-likelihood, and formalize it in a mathematical notation, because, unfortunately, we can not explain a multidimensional classification with visualization.

Assuming $X = \{x_1, x_2, \ldots, x_n\}$ are input variables of the model, $\theta = \{\beta_0, \beta_1, \ldots \beta_n\}$ 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) $L(\theta) = \Pi_{i=1}^{n} p(x_i)(1 - p(x_i))$

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. Since there are only two possible variables for Y we can write the cost function of logistic regression as follows:

(ii) $L(\theta) = \Pi_{i=1}^{n} p(Y = 1 \mid x_i)(p(Y = 0 \mid 1 - x_i))$

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

(iii) $L(\theta) = P(Y \mid X, \theta) = \Pi_{n=1}^{N} P(y_n \mid x_n, \theta)$

Equation (iii) is the cost function for softmax regression. We have explained for the logistic regression, we are using the log-likelihood function, therefore the described equation can be written as follows, note that instead of $\ell(\theta)$ we use $J(\theta)$, which is common while using softmax refer to cost function as $J(\theta)$:

(iv) $J(\theta) = -\, log\ P(Y \mid X, \theta) = -\, \Sigma_{n=1}^{N}\ log\ P(y_n \mid x_n, \theta)$

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**. Because the logarithm of any variable smaller than one is negative, we use the negative sign at the end to make it positive. If you are good in mathematic 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).

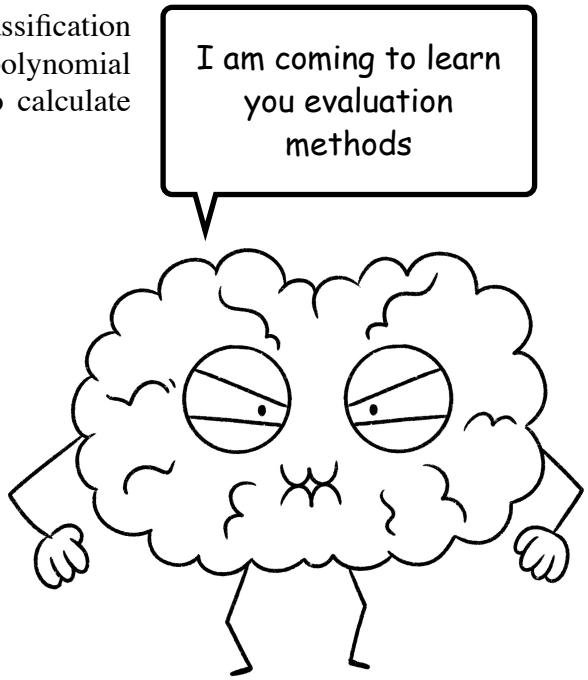
Since cross-entropy is a cost function used to measure the accuracy of the softmax regressions, our goal should be minimizing the cross-entropy loss. Assuming we have ***M*** **different output classes (in logistic we have only two)**, and ***N*** **number of input variables** the cross-entropy cost function is written as equation (v). This equation is a generalization of equation (i) for more than two output variables:

(v) $J(\theta) = -\Sigma_{j=1}^{M}\Sigma_{i=1}^{N} y_{ij}\, log(p_{ij})$

The cross-entropy objective function is to reduce the cross-entropy, thus a model that has ش smaller cross-entropy is favored over the model that has larger cross-entropy.
page break:

Evaluating Regression Models Fitness

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



Logistic and softmax regression (classification regressions) evaluation methods focus on evaluating *how good is the model fits the data*. Here, we list methods used to evaluate classification regressions. However, some of them are very important 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.

		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 variable.

Some literature [Grus '19] (in Chapter 11), and [Ng '18] recommended having three different sets, instead of two sets (test and train). They recommend using the third part of the dataset as a **validation** or **development** set. The validation set is responsible for parameter tuning, feature selection, etc. In particular, it can be used to choose the best possible model among different models or different model settings. While selecting the validation set, it is recommended to have a test set and validation set have the same distribution.

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. For example Figure 8-12 presents a sample that by increasing the dataset size in both train and test sets the error rate will converge at some point and the model will be stable. It can be seen that at the beginning when there are few training instances (the left side of Figure 8-12) 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.

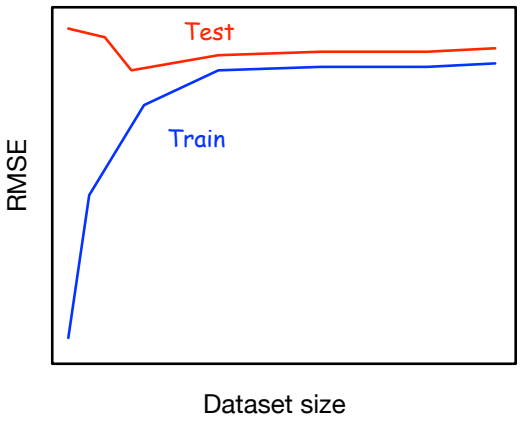


Figure 8-12: Learning Curve example. By increasing the number of data objects both training and test dataset converge. This makes the model reliable.

On the other hand, you can observe that for the test set at the beginning the RMSE rate is high, despite having 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.

While we are plotting the learning curve, a very low training error and very high test error rate at the beginning of the curve is presenting the overfitting. We will explain overfitting later in this chapter.

ROC Curve

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

The ROC curve is plotting 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 - Specificity (or True Negative Rate)* as it has been shown in the following:

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

ROC curve is useful to demonstrate 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 45

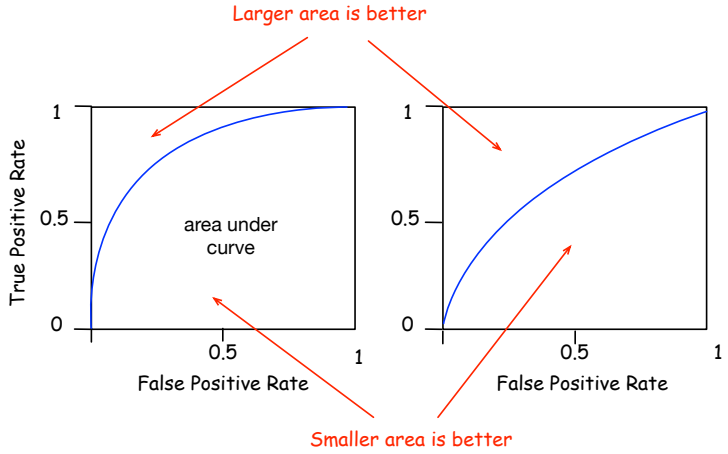


Figure 8-13: (Left) a ROC curve which shows high accuracy, (Right) ROC curve which has less accuracy in comparison to the left one.

degrees, the accuracy is decreasing. Figure 8-13 presents two ROC curve samples, the left one is more accurate than the right one because the right one is closer to 45 degrees.

The result of the ROC plot is read as the Area Under the ROC area (AUROC), 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 for logistic and softmax regression, we do not have R^2. Because their parameters are not calculated to minimize variance (linear regressions). In logistic and softmax regressions parameters were estimated based on the MLE approach, through its iterative process and not minimizing the variance.

Nevertheless, to provide measurement metrics there are some **Pseudo R^2** or **Adjusted R^2** methods available such as **McFadden's R^2** [Domencich '75].
McFadden's R^2 is written 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 fitted model, $L(\theta_0)$ is the model with only an intercept (β_0) and no covariates (input variables). The high value of McFadden's 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.

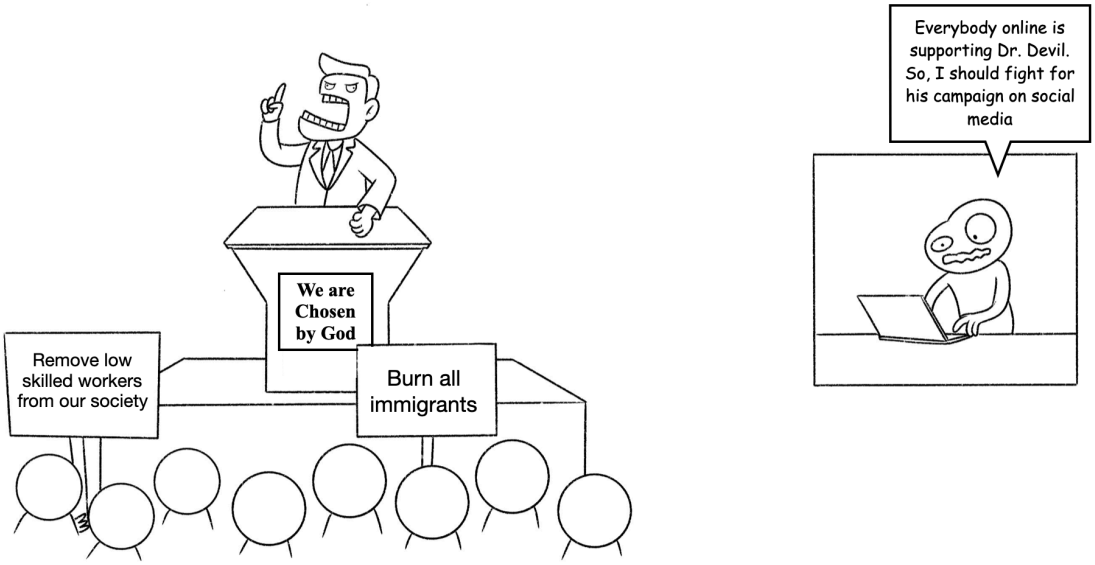
Your software will have pseudo R^2 and again you do not need to calculate them on your own. Therefore, don't worry if implementing an MLE and Pseudo R^2 sounds complicated.

Wald Test

Wald test is also known as **Wald Chi-square test**. It is used to measure whether model parameters are statistically significant [Wald '45]. Wald test null hypothesis states model

parameters do not have any effect on the model fitness. If the null hypothesis failed 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 and we need to look for better parameters. If the output of the Wald test shows that a parameter for a certain input variable is zero, we can remove that particular input parameter from our model.

For example, assume a data scientist, i.e. Dr. Devil, is consulting politicians for election. He had a logistic regression that uses 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 the public.



Based on previous consultations, Dr. Devil identifies 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 on 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 **Akaike Information Criterion (AIC)** [Alkaline '74] and **Bayesian Information Criterion (BIC)**, which is also called **Schwarz Information Criterion (SIC)** [Schwarz '78].

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

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

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 don't recall the RSS, check the description of linear regression. The output of AIC is a score and the lower AIC score is better. Therefore, if we compare the two models' AIC scores, we should choose the model which has the lowest AIC score.

Another method is BIC or SIC, which has an advantage over AIC, because it considers the number of parameters as a penalty. As the number of model parameters increases, the BIC score increases as well, and a good model is the one with the lowest BIC score. If we intend to penalize a model for having too many parameters it is recommended to favor BIC over AIC. Why do we use this? Because a large number of parameters makes the model complex and thus inefficient (from a resource usage perspective).

Assuming n is the number of data objects, 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 written as follows:

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

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

For linear types of regressions we use the following equation:

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

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 that all parameters of one model that existed 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, the exam score, high school performance, race of the candidate, nationality of the candidate, and if his/her parents are

rich. The first model is nested within the second model because the second model has all 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:

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

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.

NOTE:

* 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 funciton 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 and it 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 note not performing extrapolation beyond our observed data points, especially while using polynomial regression which uses curved lines.

page break:

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 be able to predict the future (unobserved or test data). The ability to match the future data into observed data is referred to as **generalization**. Therefore, an umbrella term that can be used for regression algorithms is called Generalized Linear Models (GLM). Note that GLM models do not necessarily assume that there is a linear relationship between input and output variables. Instead, they assume there is a linear relationship between output variables (a.k.a. transformed responses), in terms of link function (e.g. logit), and input (a.k.a explanatory) variables, which covers logistic and softmax regressions as well. In simple words, they incorporate link function as well, and not just linear relationship between input and output)

There are two 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 *it performs very well good 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 *too specific model* on the training dataset and poor generalization on 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 have a poor performance on both our training and test datasets.

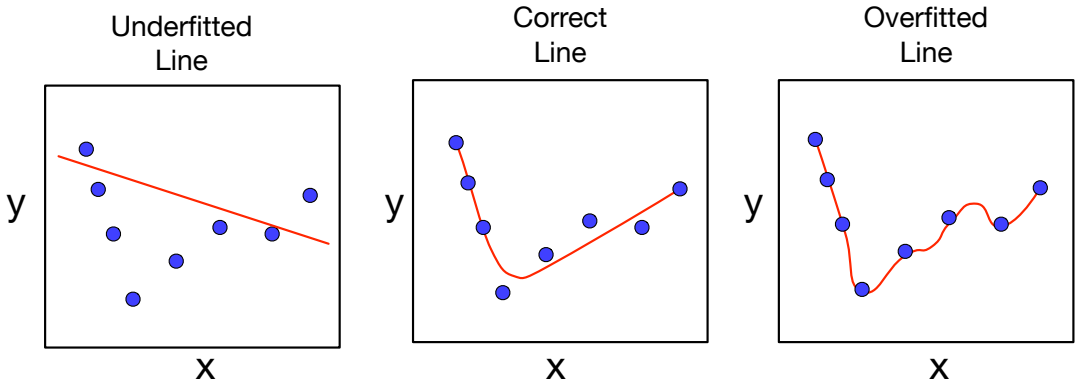


Figure 8-14: Example of underfitting and overfitting in model fitting.

Figure 8-14 shows three examples, an underfitted model, an overfitted model, and a proper model.

You can see 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.

Now, the question is how to avoid releasing the devils of overfitting or underfitting? Underfitting is easy to recognize because as soon 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 were able to 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 fantastic result and not in others, that particular experiment is overfitted and thus we should revise the model.

Keep in mind 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. A good approach to deal with overfitting is to decompose it into bias and variance problems [Domingos '12]. In other words, we use bias and variance to study if our model is accurately describing the underlying dataset or it is misleading.

Variance refers to how far is our observed data (or training dataset) differs from the mean of the predicted data (output variables). In other words, variance presents the *amount of changes in output variables*, *in another training set*. It is clear that different training set results 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.*

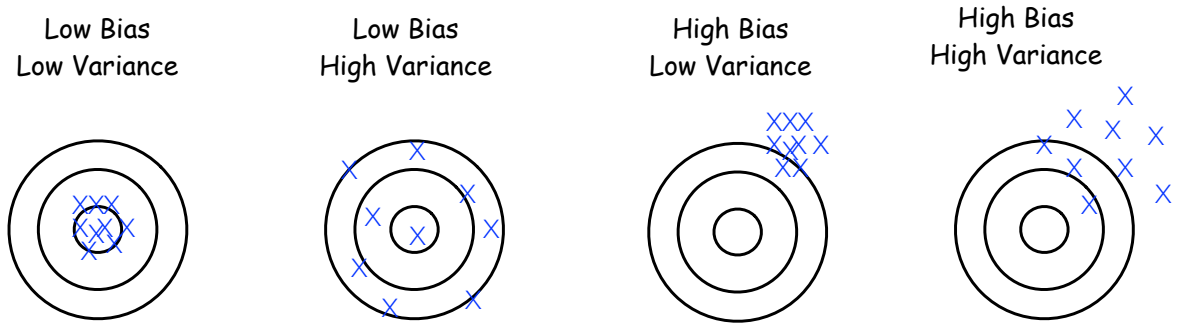


Figure 8-15: Example of underfitting and overfitting in model fitting. The target is the actual data and thrown dart are predicted data.

Bias refers to differences between the model output and correct output (what happens in the real world). In other words, bias *presents the differences between the model estimation and the true value of the parameter being estimated by the model*.

How do we get the real-world result? again by our lovely cross-fold validation and using the ground truth dataset. Domingos [Domingos '12] provides a good clarification on bias and variance, as follows:

"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 8-15 presents the bias and variance analogy by throwing darts at the dartboard. In this example, assume the output is the closeness to the center of the dartboard, and the input is a set of parameters that affect the dart direction.

To summarize keep in mind: *Bias is about the Model's incorrectness, Variance is varieties of the model's outputs*. A good model has both a low bias and low variance. However, there is a phenomenon known as **Bias-Variance trade off**, which means an increase in one of them, causes a decrease in the other one. In particular, it is easy to create a model with low bias, but its variance will be high, or it is easy to create a model with a low variance but its bias will be very high.

Earlier in this chapter in the "Evaluating the fitness (accuracy) of linear models" section, we have discussed MSE while describing the fitness of a linear regression model. MSE is written as follows:

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

and the expected MSE can be decomposed into the error equation as follows, which we don't go deep into its math.

$$Error = bias(\hat{y}_i)^2 + variance(\hat{y}_i) + variance(e).$$

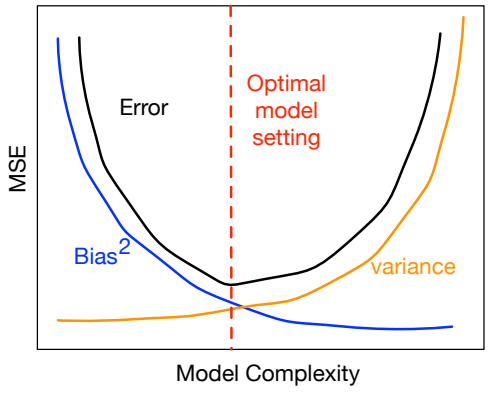
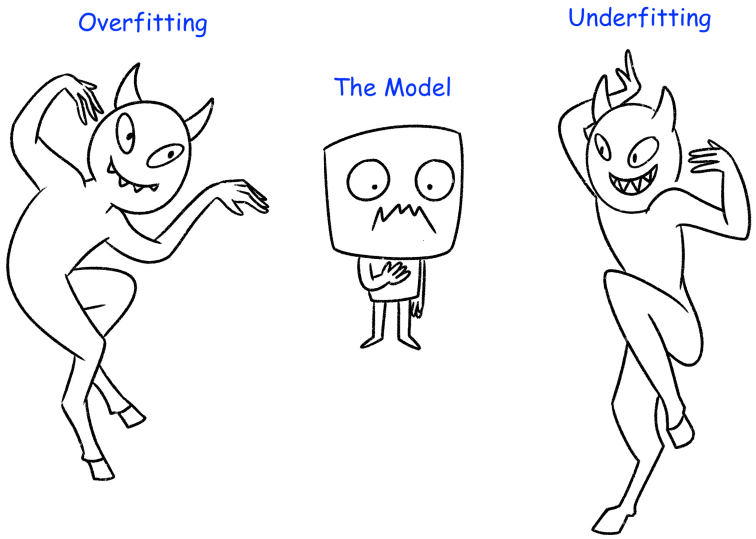


Figure 8-16: Typical bias-variance and error changes. At the red dotted line, the model has the smallest MSE error, thus complexity at this point can be used to develop the optimal model.

Note that e presents **an irreducible error**, which is an error that can not be reduced by having a good model and it is 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 an algorithm the bias will be reduced but the variance increases. We should look for an optimal boundary something like the red line in Figure 8-16, 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 or regression algorithm that builds a model, creates a decision boundary. The decision boundary is a line that could be straight, curved, or have a complex form [Burkov '18]. In the simple example of Figure 8-16 the boundary is a straight line.

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. It means that more flexible models (cover more features) have higher variance.

NOTES:

* High bias presents underfitting and high variance presents overfitting. Usually, the more flexible the method is, the less the bias will be.

* Underfitting usually causes 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 so easily increase the sample size. The data is a valuable source and usually, we use it with lots of care.

Page break:

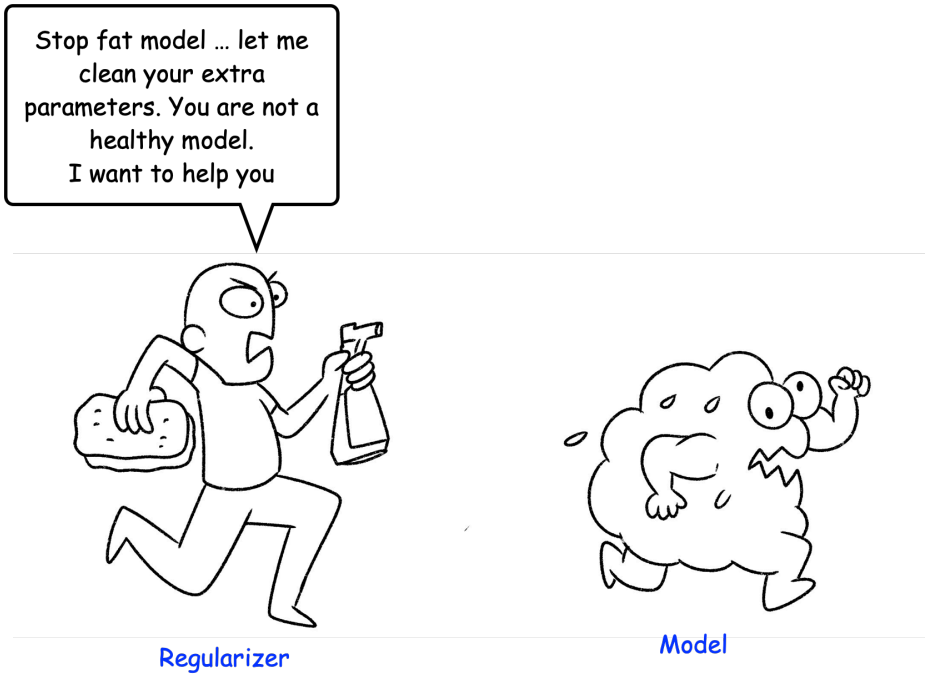
Regularization

Now we are familiar with devils in model building and one of the fiercest devils is overfitting. Due to Oscam 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, on the given training set.*

It applies to the training phase only, not the testing phase. Besides, we can use regularization to penalize the high power coefficients that make the polynomial model very sensitive to noise.

Regularization methods make the model simpler and try to reduce model coefficients (parameters), thus they are also called **shrinkage** methods. In simple words, regularization is the process of *shrinking the model's coefficients* and focusing on *decreasing the variance of the model*.

Regularization is crucial for working with regressions. There are $(2^n - 1)$ possible linear models from a combination of n input variables. For example, assume we have three model parameters 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 $x_1x_2x_3$. If we have 10 input variables, we can build 2^{10} linear models. If we have 100 input variables, which is very common in real-world applications, we need a super powerful computer 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. Here we explain four important regularization methods that are common, 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 the sample size is small and 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.*

In particular, it 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. If a coefficient 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 rid of x_j. It means that this particular variable will be removed from the model, which is a good thing. Why? Because we use regularization to reduce the model parameters.

Ridge regression is nothing more than 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 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. One question might arise now: how to choose an appropriate λ? Typically we use 10-fold cross validation to choose the one that provides the lowest variance. In other words, we test with different λ values until we identify a λ that results in the lowest variance. The Ridge regression cost function can be written as follows:

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

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. Usually, when we encounter *J(something)* it is presenting an objective function for *something* parameters.

Figure 8-17 presents an example model with three parameters and after increasing the λ to a specific value all parameters are converged toward zero (but not get into zero). This example shows us how increasing the penalty will reduce the impact of each coefficient on the model, until they will remove their associated variables because their coefficients will be zero. Keep in

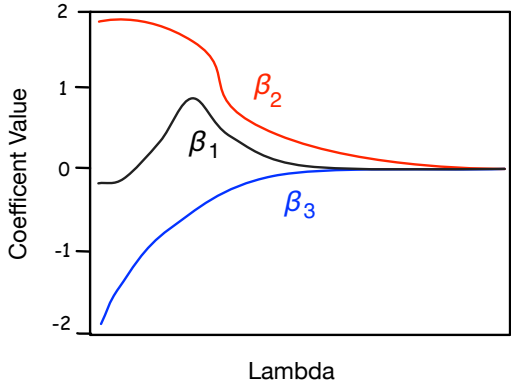


Figure 8-17: A mock model with three parameters that presents the impact of ridge regression. All parameters are converged toward zero as λ value increases.

mind that Ridge 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 8-18 (a) shows a dataset composed of 7 data points and we would like to use linear regression to make a prediction model for this dataset. Figure 8-18 (b) shows two random data points that have been selected as a training marked as red. By calculating their RSS we identify their RSS is zero, which is too low for a bias, and it makes us suspicious. Figure 8-18 (c) calculates the RSS for other points (test set), which are marked with blue color. Test set data points (blue dots) are very different from the training set data points (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.

To solve the problem of high variance, we use a Ridge regression and increase in its penalty cause a rotation of the red regression line a bit toward X-axis, as a result, the blue regression line will be created, which has been shown in Figure 8-18 (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.

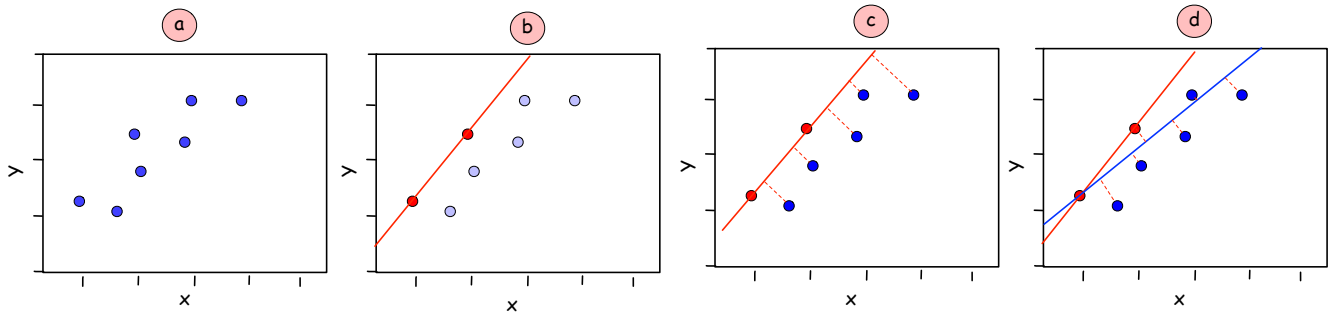


Figure 8-18: (a) Original dataset (b) Training set data points were selected as red color. (c) the other remaining points RSS are calculated to the regression line (red regression line) (d) by introducing a Ridge penalty the red regression line is rotated and the blue line regression is drawn.

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

Note that the shrinkage penalty will not be applied on the intercept (β_0), it will be applied on other parameters ($\beta_1, \beta_2, \ldots$), 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 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 $y = \beta_0 + 0.x_1 + 0.x_2 + \ldots = \beta_0$ 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 exactly to zero. In other words, it 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:

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

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

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

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

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

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 have an effect 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. Therefore, by increasing the λ, they will be zero, and thus the final model will only keep "climate-change policy" and "openness 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 completely*. Since LASSO can remove several features from the model, its result model is called a *sparse model.*

---

[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.

LASSO is better than Ridge, but it is not performing 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 when we have a good understanding of our parameters and they are very well known. However, when our model includes many parameters, like deep learning models which include millions of parameters (due to their black-box nature), it is not possible to know all of the parameters of the model. 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:

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

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 is usually preferred over LASSO and Ridge, but the best way to decide about them is to conduct several experiments on the dataset with different regularizations and check results until a good one that addresses our concerns will be identified.

NonNegative Garrote

Another regularization method is called NonNegative Garrote (NNG) [Breiman '95]. If you don't know what is Garotte, 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 Ridge, but it is scale-invariant. It can also eliminate predictors with weak coefficients (Ridge can't do it). Regression methods require two steps (i) specifying the penalty function, and (ii) specifying the λ (the tuning parameter). NNG can combine these two steps, which reduces the computational complexity significantly [Xiong '10].
Assuming n is the number of parameters and $u \geq 0$ is the shrinkage factor, its equation is written as follows:

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

We do not explain the detail of calculating the shrinkage factor u_i, you can read its detail in Xiong's article [Xiong '10]. 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 smooth 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 and not the test set. When the model is trained then we use the non-regularized model to measure the accuracy of our model.

* The cost function result we get for the training set is usually different from the cost function result we get for 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 sometimes the number of parameters is larger than the number of training data. If we have a model that its parameters are larger than its data points (the dataset is small), we can also use a regularization regression to reduce the number of its parameters.

page break:

Optimization Algorithms

Any repetitive process that is performed by humans is associated with optimization. For example, as a human grows she optimizes her eating attitude by using spoon, fork, or chopsticks, and while the human is an infant she experiments her first eating performance by using her hand and face. It means we learn a process and the more we learn, the more we optimize the process.

Usually, identifying parameters of an unknown mathematical equation, which is the objective of many machine learning algorithms, consumes lots of computational resources. Since resources are limited the optimization process helps the algorithm to search for an appropriate parameter in the environment with limited resources. In other words, we use optimization to estimate model parameters while reducing the resource required to search for the best model parameters.

Before starting to explain optimization we need to briefly discuss some mathematical concepts.

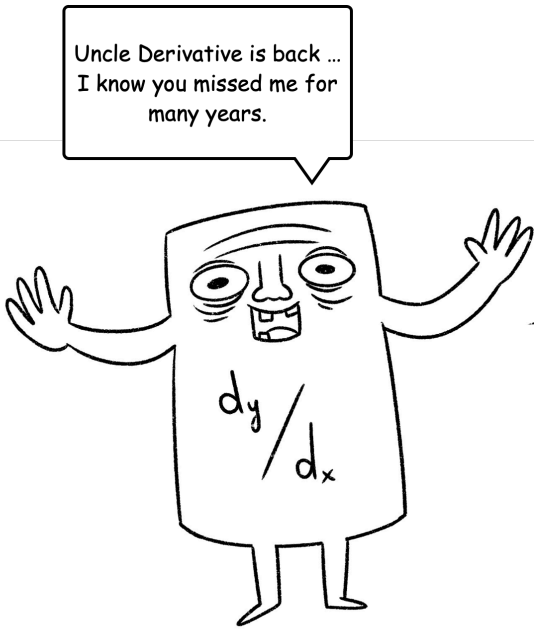
Mathematical Concepts Required for Optimization

Probably, if you are not mathematicians and reading this book, you have not encountered derivative, gradients after primary/high school and while you were in primary/high school you thought they are completely useless for your life. We thought the same as well. But, at that time we didn't know that math stuck to our life like a parasite, and we can never get rid of mathematics. Galileo, who was a decent scientist said that *"the book of nature is written in a mathematical language"*. In the following, first, we explain these concepts very briefly. If you are familiar with these mathematical concepts feel free to skip this section.

Derivative:

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, the *derivative specifies how the output will change, based on the given input*. If it is not still clear read the

following: *derivative of a function describes how fast does 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 thus its derivative will be zero. If the function is changing at different paces, such as Figure 8-19, the derivative will be a function itself. Derivative can be 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:

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

As you can see from the red lines in Figure 8-19, when the line is not straight the derivative is changing, because at different points the function has a different pace of change. A constant derivative means that the function is growing constantly and a derivative of a straight line is a constant value.

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

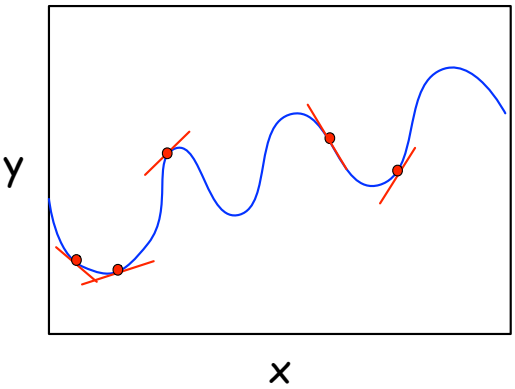


Figure 8-19: A sample function that is plotted in two dimensional spaces ($\mathbb{R}^2$) and five random tangent lines (red lines) has been drawn. You can see that each tangent line touches the curve at one point (red dots).

The derivative is responsible to specify the slope for the given tangent line of the function. From Figure 8-19 we can see we have different tangent lines and their slope is changing at each point.

However, each tangent line has one point on the function as it has been shown in Figure 8-19, we need at least two points to be able to draw a

line. To draw the slope, we use the following equation:

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

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 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 formula 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 8-20. In Figure 8-20 a, you can see that the distance of h is largest and the line that describes the curve's slope is the least accurate line (in comparison to Figure 8-20 b and Figure 8-20 c). Then in Figure 8-20 b. the points get closer and thus the line gets more accurate because the direction is getting more specified. In Figure 8-20 c, the data points get

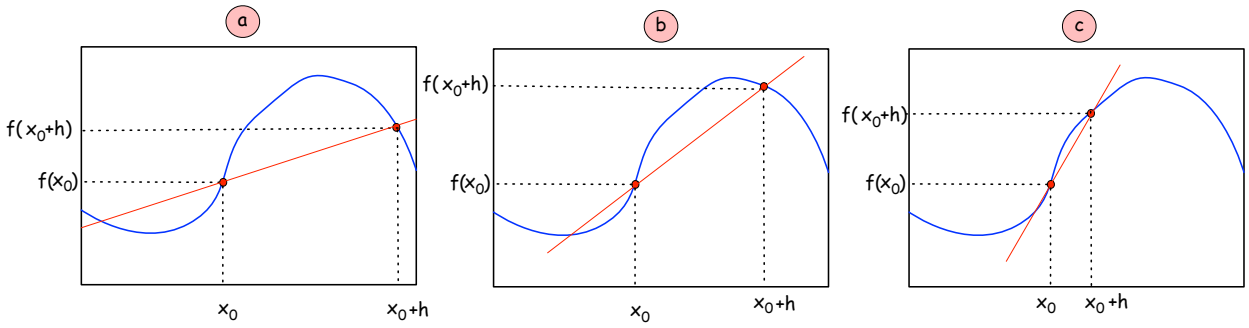


Figure 8-20: Three examples of slope line identification based on distances between two data points, which have h distance. These slope lines are called **secant** lines. The more h gets smaller the *secant* line gets closer to be a tangent line.

closer to each other and the slope line gets more accurate than what we had in Figure 8-20 b. Because, from Figure 8-20 a to Figure 8-20 c, the red line gets more tangential and does not cross the function line (blue lines).

What we can conclude from Figure 8-20, that as the h distance is getting smaller the slope line to the tangent line is getting more accurate. Therefore, we can write the derivation function of x is 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 derivative that we need to know, assuming n is a constant number, including:

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

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

$f(x) = ln(x) \rightarrow f'(x) = 1/x,$
$f(x) = x \rightarrow f'(x) = 1,$
$f(x) = n \rightarrow f'(x) = 0,$
etc.

In other words, we can say the function x^n has the slope of $n\,x$, the function $n\,x$ has the slope of n, etc.

What we have explained is 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.
$f(x) = x^2 \rightarrow f'(x) = 2x \rightarrow f''(x) = 2$
To calculate the derivative we should use some rules specified for derivative including "chain rule", "quotient rule", "product rule" which we do not explain in detail here and just write them down. If you like to learn their proof, there are plenty of online resources to explain them.
Reciprocal rule: $(\frac{1}{x})' = \frac{-1}{x^2}$
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: $f(x) - g(x) = f'(x) - g'(x)$
Chain rule (outside-inside rule): $[f(g(x))]' = f'(g(x))g'(x)$
Chain rule in particular is very important to learn and the Backpropagation algorithm, which we explain in Chapter 11. Keep in mind that the *Chain rule is useful to find a derivative of a function when we have nested functions (a function inside another function), e.g. function g is inside function f.* We can say it is a derivative of the outside function while leaving the inside function times the derivative of the inside function. $[f(g(x))]' = f'(g(x)) \times g'(x)$

Partial Derivative:

We have discussed the derivate which is used for one variable. Sometimes in an equation, we need to deal with more than one variable, e.g., $f(x, y) = x^2 + 4y$. To write a derivative of such a function once we need to write derivative for x and consider the other variable (y) as constant, then we need to write derivative for y and consider the variable x as constant. This is called a partial derivative.

To write a partial derivative for variable x and variable y we write the following:
$\frac{\partial}{\partial x}[f(x, y)] = 2x + 0$ and $\frac{\partial}{\partial y}[f'(y)] = 0 + 4$
Instead of d that we use to refer to the derivative here we use ∂ sign for partial derivative.

Gradient:

Congratulation now you understand or recall what is derivative. You can see that the derivative is defined in two dimensional space (it can be shown as $\mathbb{R}^2$) or a *univariate* function. How about having a *multivariate* function, e.g., $\mathbb{R}^3$?

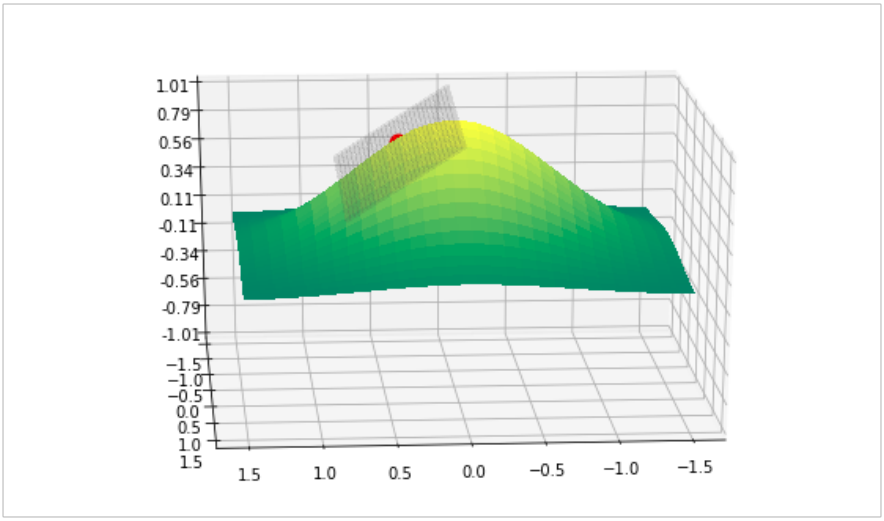


Figure 8-21: Tangent hyper plane plotted in 3D space.

The generalization of derivatives for a multivariate function is called ***Gradient***. In the context of machine learning, we can say derivative is used for vectors, but Gradient is the deviate for matrix and tensors. In simple words, Gradient is used for functions that take more than one input variables (or parameters). It is a **vector of partial derivatives** and it gives the input direction, in which the function increases or decreases.

While calculating Gradient, instead of the tangent line, which we had in the derivative, we have **tangent-plane** or **hyperplane** in gradient. Figure 8-21 visualizes the gradient hyperplane on a function that is plotted in 3D space.

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

For example for our $f(x_1, x_2) = a\,x_1^2 + b x_2 + c$ function, the derivative of this function with respect to x_1 is written as $\frac{\partial f(x_1, x_2)}{\partial x_1}$. Its gradient will be 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 the context of machine learning, *we calculate gradient for cost functions*.

---

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

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 $\nabla J(\beta_0, \beta_1) = \left[\frac{dj}{d\beta_1}, \frac{dj}{d\beta_0}\right]$

Jacobian:

When we stack the gradient of two functions, which each is a vector of variables, into a matrix, it is called 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 second order of gradient (a derivative of derivative) is called Hessian (read it as heshian) and it is written as a matrix. We have seen previously that gradient can be written as $\nabla f(x_1, x_2, \ldots x_n)$. The Hessian which is the second-order of the gradient will be 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 enforce us to learn mathematics and some devils, who named themselves mathematicians, introduce the concept of 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* and we use it 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.

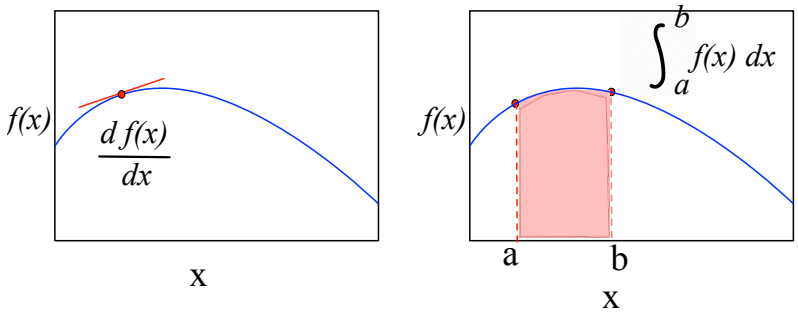


Figure 8-22: Comparison between derivative (left) and integral (right).

A good example to understand integral is that it is the area under the curve. Assume we are filling a tank with water from a tap. The pace of water coming out of the tap is derivative and the amount of water inside the tank is integral. For example, if the tap is open and it gives a constant amount of water every second, e.g. 1 *ml*, the water inside the tank will increase with the constant pace, i.e. $f(x) = x$. If the tap is open and each second its pace of pumping water increases $2x$ (which is a derivative of x^2) times, then the amount of water inside the tank will increase with $f(x) = x^2$. Figure 8-22 presents the derivative and integral of a function.

If still the purpose of learning these concepts is not clear, please be a bit more patient in a few more paragraphs we will learn how mathematics is used in optimization functions and we will drown you into mathematics.

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:

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

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:

$$e^x = 1 + x + \frac{x^2}{2!} + + \frac{x^3}{3!} + \dots$$

$$sin(x) = x - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

Given *a* as a variable of the function *f(x)*, a Taylor series for *f(x)* will be calculated by using the following equation:

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

Taylor series is used in the approximation function. 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.

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

Page break:

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. O*ptimization is the process to change and tweak model parameters to minimize the cost function.* Why do we need optimization and why not experiment with all possible parameters until we get the best result? To better understand the need for optimization, take a look at the right side of Figure 8-23. The bottom of this curve is the optimal parameter value,

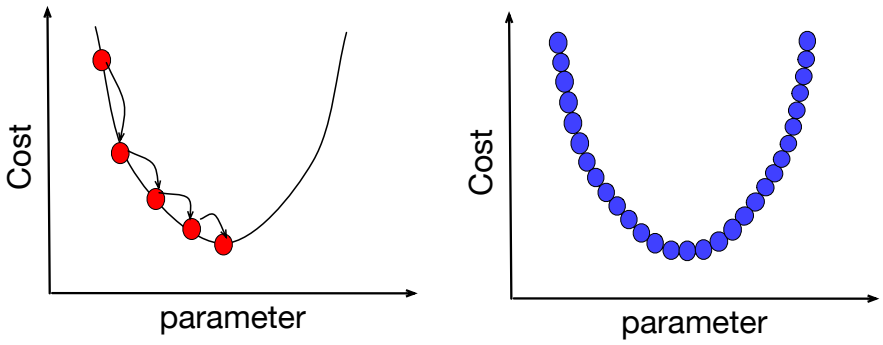


Figure 8-23: (left) using an optimization to reach the minima and therefore very few data points will be processed. (right) not using an optimization and thus many data points will be experimented on the function to find the minima.

because the cost is at it lowest. However, to this get point via a brute force approach, we need to experiment with all possibles points on the function curve, i.e., all blue dots on the curve. This is expensive to experiment and even for this simple convex function, we have too many blue points to experiment. However, if we use optimization as it is shown on the left side of Figure 8-23, we can skip many data points and get faster to the minimum. By faster, we mean fewer experiments.

Take a look at Figure 8-24, it is a function and we can see its local minimums (minima or sometimes it is called optima), which are the smallest y in each valley (or in comparison to its adjacent points), and global minima, which is the smallest amount of y in the entire function. The process of optimization in the context of the machine learning algorithm is usually identifying strong local minima. By strong, we mean a number close to the global minima number, or ideally global minima itself. Therefore, we can say that *the ultimate goal of optimization is to find the global minima*. In reality, if we deal with non-convex[8] shape optimization it is not always 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 8-23), the local minima could be the global minima as well, but still it is not guaranteed that the optimization algorithm will find it.

---

[8] To understand if a function is convex, if we we select any data points and a direct line that joins them together *do not crosses* the function this function is a **convex function**. We referred to the convex function back in Chapter 4 as well.

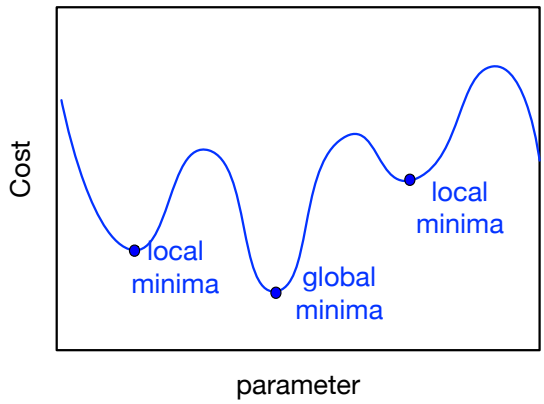


Figure 8-24: A sample function (non-convex shape) that its local minimal and global minima data points has been specified. While reading any of these optimization diagram note that the function (here presented in blue line) is not known and the optimization objective is to find these minima.

Note that the blue curve in Figure 8-24 did not exist at the beginning of the optimization process, we should experiment with lots of different parameter values and measure the cost function to be able to identify the cost function shape.

By looking at the figure we can easily identify local minima(s) or global minima. However, mathematically there are two conditions required to claim a datapoint as a local minima: (i) its derivative is zero, i.e. $f'(x) = 0$ (ii) it's second order derivative is not negative $f''(x) \geq 0$.

Now, by using our background knowledge about optimization we can say that *the process of optimization is focused on reducing the cost or loss function by choosing the right values for the model parameters*. In other words, our goal in building a model is to find the best model that is able to describe the dataset. Therefore, identifying the optimal value for model parameters reduces the error, and thus we can have the optimal model. Apologies for repeating it too many times, we would like to enforce you to memorize the logic behind optimization.

Any of 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 can be 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 very easy to optimize and usually not a case in a real-world model.

There are various methods 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 in use 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 it in 2D space, which is easier to understand. Previously, we have explained that an optimization problem is the process of finding a minima which is close to global minima. Now, assume we have a function and we would like to optimize it, i.e. finding a minima close to the global minima. We knew from the previous description that the gradients of minimum points are zero, also the more we are moving forward toward the minimum point the gradient is getting closer to zero. Thus, it gets a random data point and calculate its derivative (slope). Then based on the value of slope it makes another jump toward minima.

For example, take a look at Figure 8-25 which presents a cost of an imaginary function, based on its model parameter. At the beginning (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

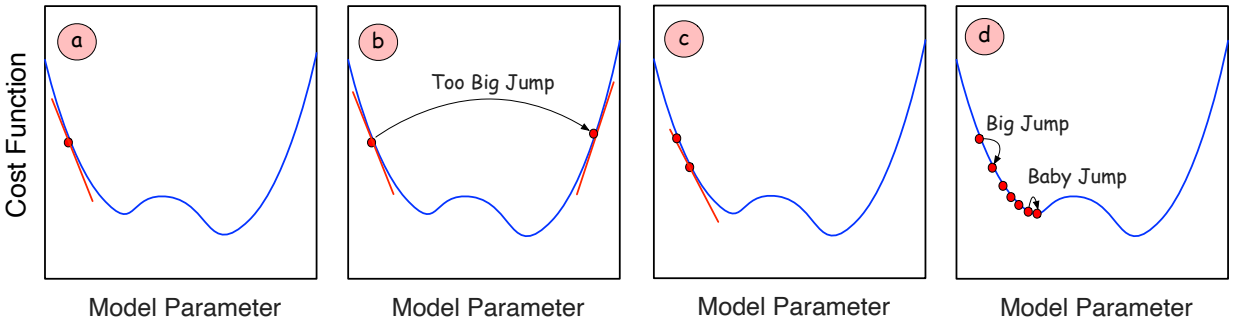


Figure 8-25: A gradient descent approach step by step in a non-convex cost function.

function as it has been shown in Figure 8-25 b. Nevertheless, *the sign of the derivative of the new point is changed and this is a sign that the function made a too big jump*. Thus, the optimization function discard this jump and makes a smaller jump, as shown in Figure 8-25 c., and calculates its steepness. Since *the absolute value of the derivative is smaller than the previous point, thus it the algorithm realized that it was a correct jump*. 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 called *Adaptive Gradient Descent (ADAM)* [Kingma '14], which is the most used optimizer used for neural networks. With enough repetitions, the gradient descent function can identify a good minima, which is close to global minima or it could be itself the global minima.

Gradient descent stops either (i) a specific threshold (maximum epoch) reaches, (ii) the step size is getting very small (derivative is getting 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 8-25 b? Taking small steps make the minima identification more accurate, but it makes the gradient calculation process very time consuming as well.

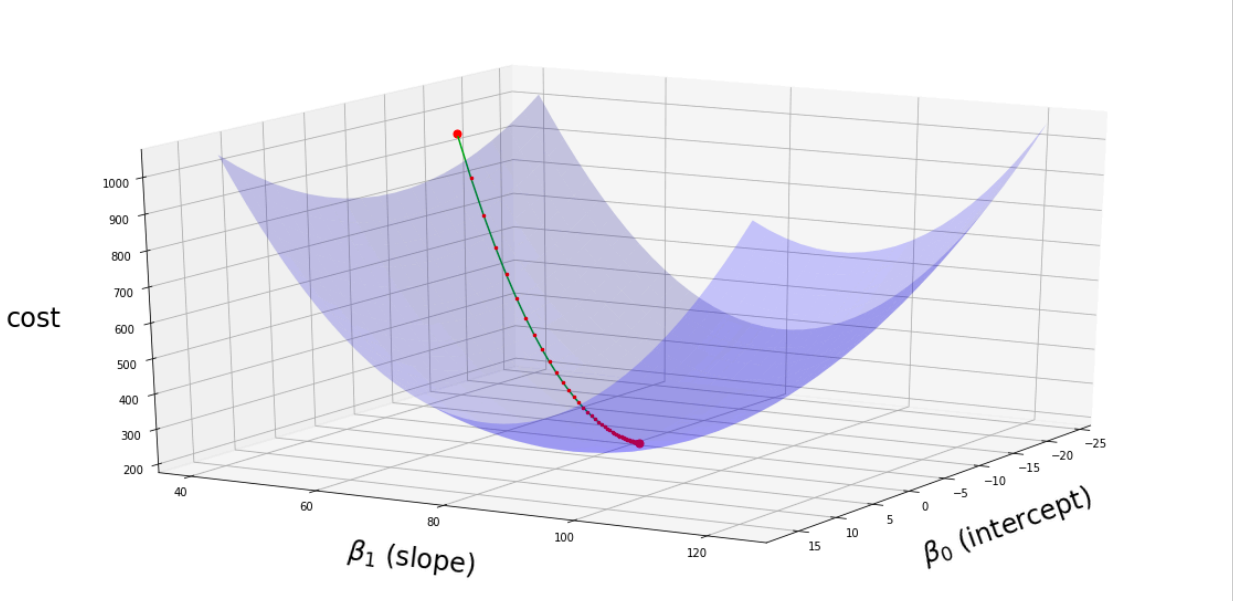


Figure 8-26: Visualization of step sizes of the gradient descent in green line, Each step starts with a red dot and stops in the next red dot. You can see that step sizes are getting smaller as we move toward the global minima in this convex function.

To summarize let's take a look at a surface plot in Figure 8-26, which shows a cost function of a linear regression model for both β_0 and β_1 [9]. You can observe this simple surface plot has one minimum. To calculate the gradient descent the first a 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 are getting smaller and smaller as the gradient descent function moves toward the minimum point (which in this case is global minima, but in real-world, the cost function is usually not convex and therefore it is not that simple).

These **step sizes** were calculated by a parameter called **learning rate** times **slope** of that particular point, for each parameter, i.e. *next point = current point - (learning rate × slope (a derivative of that particular parameter).*

In particular, *the learning rate specifies the step in each iteration.* The good learning rate will be usually determined by the algorithm and we do not need to care about its value. 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.

In general, we can take the following five steps to calculate the Gradient Descent as follows:

(1) Calculate the cost for the given parameter values. For example, calculate for RMSE for the given β_0 and β_1. Then, by combining these two derivatives the gradient of this loss function will be identified.

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

---

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

(3) The gradient (if the model has an only parameter, then its derivative), 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 gradient of the function 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 to 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 by gradient descent we can say one epoch passed.

As a formalized example, let's say the linear regression has the following equation: $y = \beta_0 x + \beta_1$ by using 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 $l = \sum_{i=1}^{n} (\hat{y}_i - (\beta_0 + \beta_1 x_i))$. Gradient descent starts with calculating a partial derivative for every parameter, based on the chain rule:
$\frac{\partial l}{\partial \beta_0} = -2(\hat{y}_i - (\beta_0 + \beta_1 x_i))$ and $\frac{\partial l}{\partial \beta_1} = -2x_i(\hat{y}_i - (\beta_0 + \beta_1 x_i))$
In addition to 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 has been described (in Chapter 3 and Chapter 6), and we will learn more about cost functions in upcoming chapters, as well.

Types of Gradient Descent

There are several types of gradient descents, batch gradient descent, mini-batch gradient descent, stochastic gradient descent (SGD), SGD with momentum, etc. The right choice of the 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 Gradient Descent and post-pone other methods for later chapters. More optimization algorithms will be introduced in Chapter 10.

Batch Gradient Descent (BGD): As a reminder recall that the gradient descent is using the training dataset to identify the minimum value of the cost function for different values of model

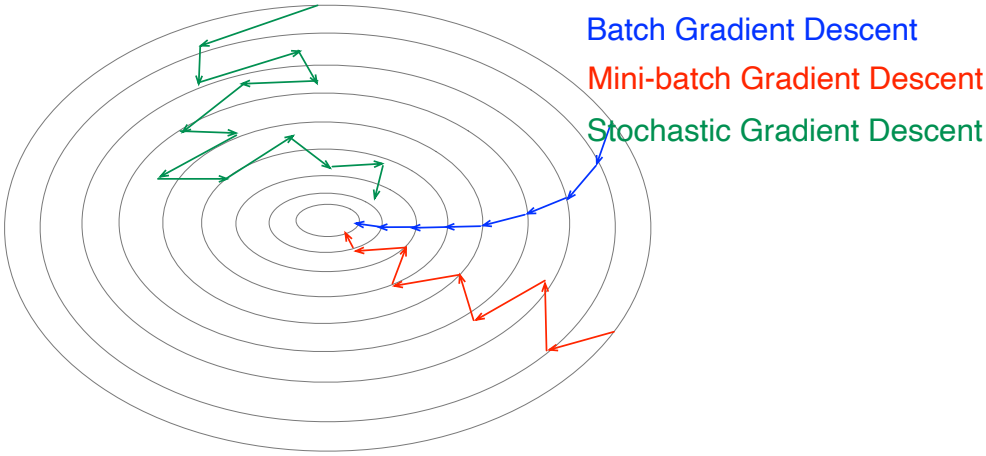
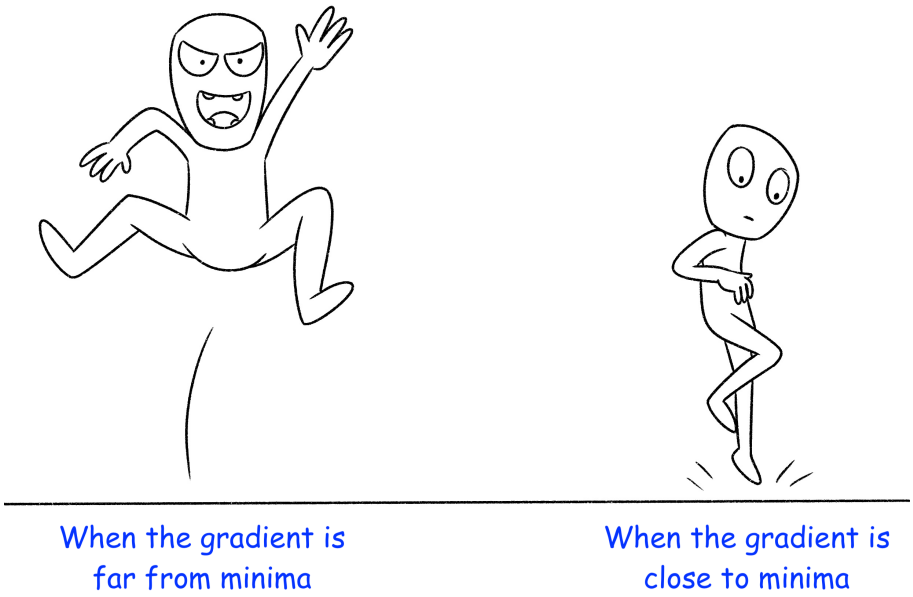


Figure 8-27: A toy example that shows a contour plot of a convex function, the minimum is located in the center. We can see that the batch gradient descent (blue colors) goes straight to the global minima. Mini-batch gradient descent (red colors) move toward minima with some variances and this variance significantly increases while using the stochastic gradient descent (green color).

parameters.

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



Batch Gradient Descent is the most simple form of gradient descent, because it uses the entire training set to compute the gradients at every step. This means in each epoch[10], *all training datasets* will be used and then the gradient will be calculated to decide on the next step.

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, it uses the entire training set, when the training set is large, the batch gradient descent is not resource efficient.

Stochastic Gradient Descent (SGD): To address the problem of using the entire training set another variety of gradient descent is 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 again calculates their gradient. These iterations will continue, until it reaches zero gradients or reaches the threshold for iteration.

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, it is bouncing around the minima and unlike BGD the SGD is not moving straightly toward the minima. Besides, due to its bouncing behavior, if the cost function is non-convex, it will rarely approach the global 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 8-27 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 you can see in Figure 8-25 b. This problem is called **overshooting**.

One solution to enforce SGD moves toward minima (even in cone shape functions such as Figure 8-27) 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 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.

---

[10] Once again we remind you that a complete pass through the training set is referred to as epoch, and thus processing each batch in BGD costs one epoch.

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 and it is miniBGD. In particular, miniBGD uses *a subset of the training set*, (GBD uses the entire training set), and thus it is faster than GBD. Since it is using 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 larger than one single instance. Similar to other types, miniBGD needs to have a dynamically changing learning rate and thus the learning schedule function here is also important.

	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.

Table 8-2 summarizes the comparison between these three methods. Now, that 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. There are optimization algorithms that are working with second-order derivation, such as Newton-Raphson method and we explain that algorithm here as well.

Newton-Raphson Method

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

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

Gradient descent draws a tangent line (at the given random point), which is identified by a derivative. Newton-Raphson's method draws a parabola[11] (at the given random point), which is identified by the second-order derivative, see Figure 8-28.

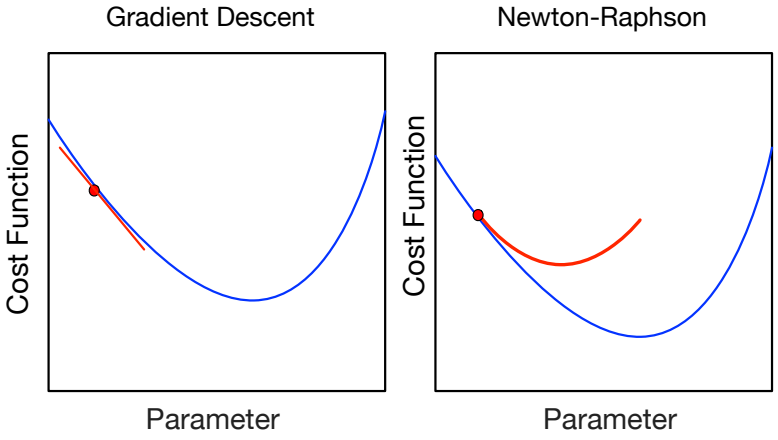


Figure 8-28: differences between Gradient Descent and Newton method

Since parabola is a convex function, it is very easy to find the single minimum in the parabola. The minimum of parabola represents $f(x)$ and by having $f(x)$ its associate x will be calculated as well. The next point to continue the iterative search for minima will be the recently identified x

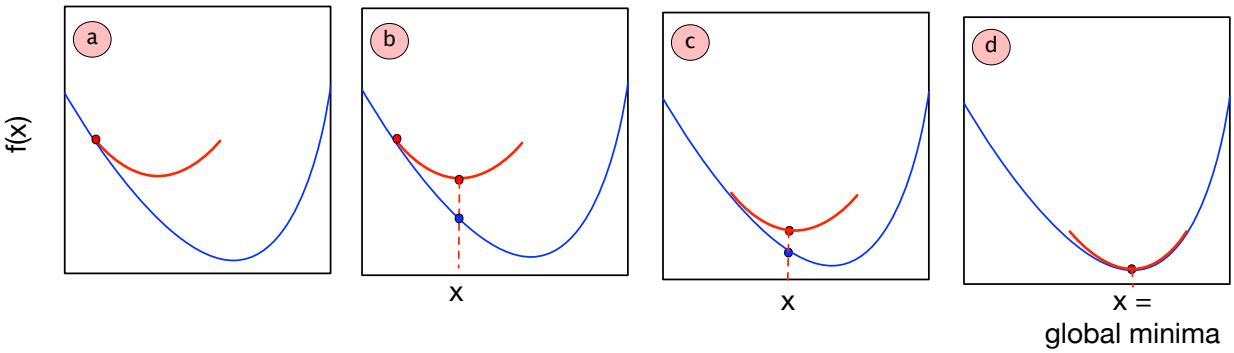


Figure 8-29: Newton method optimization example. We can see with few steps the global minima will be identified. Nevertheless for the sake of simplicity we assume the optimization function as a convex function.

(which is extracted from the center of the recently created parabola). Figure 8-29 shows that the Newton algorithm reaches the minima in three iterative steps.

To digest 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 is being used to define the Newton-Raphson method. You do not need to remember the Taylor series for that, but for your information, *q(x)* is the quadratic approximation for the Taylor series:

[11] A symmetric bowl shape curve is called parabola.

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

To calculate the next point instead of using the slope we substitute it with the Hessian of that point. Therefore, we will have $x_{t+1} = x_t - Hessian^{-1} \,.\, Gradient.$

Or we can write it as follows: $x_{t+1} = x_t - [\nabla^2 f(x_t)]^{-1} \,.\, \nabla f(x_t)$

Similar to previous examples, for the sake of simplicity we explain the univariate form of a function.

Sometimes the cost function is fairly flat and the parabola takes a too large step that falls outside the function. This is a big drawback of the Newton-Raphson method. For example, take a look at Figure 8-30 which you can see the parabola is too large for the mimina of function. Therefore, the Newton-Raphson method is working when we are close to the minimum.

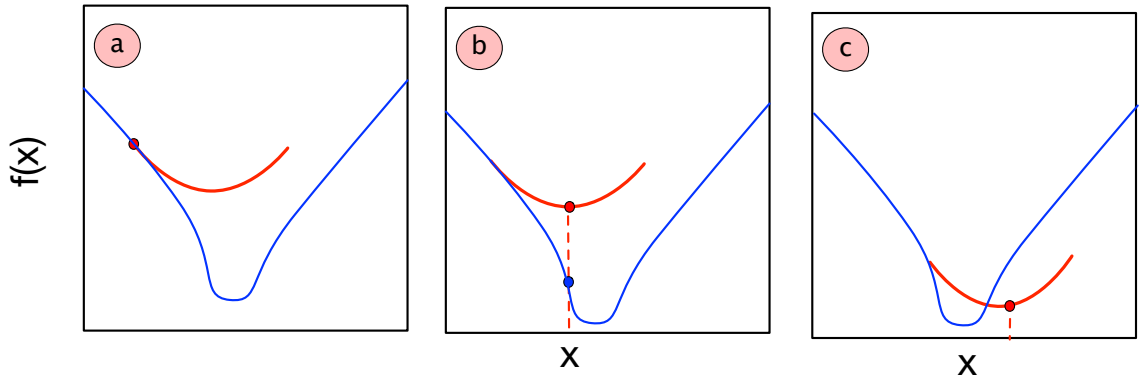


Figure 8-30: Newton method optimization example. We can see in Figure c that the parabola fall outside and thus minima will not identified. This is a common problem occurs a lot if we only rely on Newton method.

Some optimization experts suggest using gradient descent and when we get close to the minima, then take some Newton-Raphson steps to jump earlier toward the minima.

Early Stopping

In an iterative learning optimization algorithm such as gradient descent or Newton-Raphson the epoch number, i.e., the number of iterations, should be given by the user as a hyperparameter. Usually in each epoch, the error gets reduced, until a specific point, and then again it starts to increase, see Figure 8-31. When such a thing happens, to preserve resources it is better to enforce the algorithm stops its iterations. It means before finishing all requested number of epochs 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 implementation of gradient descent objective function, enables the user to specify having or not having an early stopping as a hyperparameter.

Figure 8-31 visualizes the need for early stopping, the more epoch encounters by the algorithm the error gets reduced, until a epoch=60, in which the error starts to increase. The algorithm should stop here and not continue until all epochs, this is called early stopping. This method could 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.

* Where the derivative is zero, but there is no local or global minima point, this type of point is called an **inflection point**.

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

* 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 using in non-convex cost function and reduces the chance to stuck in a 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, it is recommended to shuffle the dataset before feeding it into the gradient descent algorithm.

* Gradient descent is a type of "hill climbing" algorithm, which are popular optimization algorithm. However, gradient descent uses the slope in the local neighbors and then chooses the point which has the steepest slope, hill climbing look uses a cost function in the local neighbors and choose the point which has 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 network, we call each pass an epoch. The number of data points used for training is referred as a batch, so we can call the training dataset size is called batch size. 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 base 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 common Regression, except that the linear Regression image is linear, while the Polynomial Regression image is non-linear. Nevertheless, due to its computational complexity it is not as popular as linear regression.

In regression models, there are often some models, called "regression", but in reality they are

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.

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 MLE method, which requires logarithmic likelihood function. ”Softmax Regression“ is a multi-classification model,

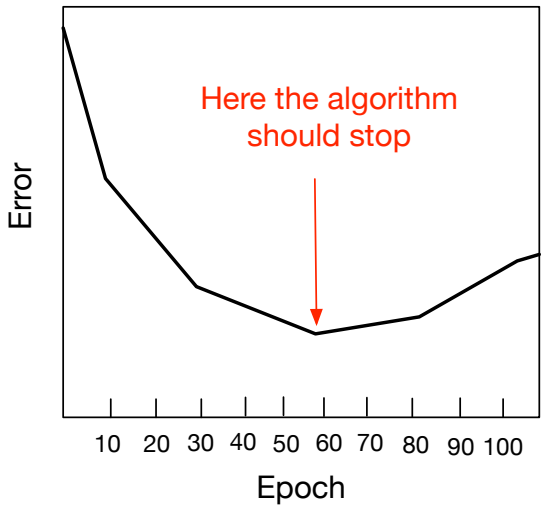


Figure 8-31: Increasing the number of iteration decreases the error, until epoch 60, then the error starts to increase. This is the place that it is recommended to enable the algorithm stops by implementing “early stopping”.

and its relationship with logistic Regression is similar to that between ordinary linear Regression and multiple linear Regression.

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.

In addition to these regression algorithms, to bring you a mental shock we described ARIMA model as well, which is used for time series extrapolation (or time series forecasting). Afterwards, challenges related to train models, including bias-variance trade off, over fitting and under fitting. Next we moved to regularization algorithms which are used to reduce model parameters and described Ridge, LASSO, Elastic Net and Non-Negative Garrote regularization algorithms. Table 8-4 summarizes regularization approaches we have explained.

The last part of this chapter explained some mathematical concepts and two optimization approaches Gradient Descent and Newton-Raphson. 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 that we have explained in Chapter 6). We have explained three types of gradient descent, which are listed as follows:

- Batch Gradient Descent: Batch size = Training set size
- Stochastic Gradient Descent: Batch size = 1
- Mini-Batch Gradient Descent: 1 < Batch size < Training set size

We also explained Newton-Raphson, which could be used in combination to Gradient descent. Newton-Raphson is second order derivative optimization and unlike Gradient descent is independent to step size.

Further Readings

* There is a book by James et al. [James '13] which has a very detailed and easy to understand description about linear regressions and its variation.

* There is an interactive visualization

* If you are interested in learning the math behinds logistic regression there is a good tutorial provided by hackerearth.com 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 are 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 in http://mathforcollege.com/nm/videos/, you can use it to learn how polynomial parameters were estimated.

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

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

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

* If you like to read more about Gradient Descent and in its implementation, Geron [Geron '19] has a good summary about gradient descent with python codes. If you are interested to delve deep into the optimization techniques and its mathematical concepts, the Algorithms for Optimization book [Kochenderfer '19] is a good 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 the Jason Brownlee's web log, https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/

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