Assignment: Finding the Minimum of a Function

In this exercise, you will define a function and then use three different approaches to find the value that minimizes the function. You will:

- plot the function to visually find the minimum of the function.
- try a range of values to find the one that minimizes the function.
- follow the steps demonstrated in the videos to implement the gradient descent method of finding the minimum of the function.

This exercise will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window 

3. This assignment will be auto-graded and can be resubmitted. After submission, the Jupyter Notebook will remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Education —> Mark as Uncompleted,** then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one and two.

This exercise will be auto-graded.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Watch: Using Gradient Descent in Logistic Regression Training

In these videos, Mr. D'Alessandro walks through the code implementation of using gradient descent to train a logistic regression model. The implementation consists of the gradient calculation of log loss, and the learning rate calculation using the Hessian function. Mr. D'Alessandro will also exemplify how weights evolve throughout training. Watch as he compares the results of our gradient descent implementation against the results from an implementation in scikit-learn. Keep in mind that stepsize and learning rate are synonymous here and that scikit-learn uses a different optimization algorithm known as "lbfgs".

Part 1

This video explains how to implement components that will be used in the gradient descent implementation. First, Mr. D'Alessandro will demonstrate how to implement a Python function that finds the gradient of log loss. Next, he will demonstrate how to implement a Python function that will compute the Hessian to find the learning rate.

Video Transcript

In this video, we're going to wrap up our gradient descent discussion and lay out how we can apply it to find the weights of a logistic regression model during training. This will be one of the more complex lessons in this course with more math than usual. But I want to emphasize that the math is mostly the vector and matrix operations we've been covering, which is something we'll also need to learn how to translate into code. In that vein, I'll present Python implementations of each step governed by the math. We're going to use the same gradient descent function I presented in previous lectures. We will have to make a few modifications, though. First is that we're going to run gradient descent on a full vector of weights as opposed to just one weight. Our code will have to reflect that. Then we will have to implement our gradient and hessian functions, which are more complicated than the type of gradients and hessians I showed before. One thing that makes them more complicated is they're both functions of the underlying data, so we need to incorporate that into the implementation. I will first show the math behind each and then show a corresponding Python implementation. We will start with discussing how to find the

gradient and then implement the function that finds the gradient. This is the gradient of the negative log loss. We multiplied the original log loss by negative one so that we can make this a minimization problem. Computing the gradient requires two steps. We first compute the model predictions for each example represented by a vector P. We then take the difference between the ground truth labels Y and the predictions P and take the dot product of this vector with the original feature matrix X. Our result is a vector of gradients, which has one element for each feature and an extra one for the intercept of the model. Let me show how to do this with Python. This function does exactly what I just described using the math as a reference. I made an implementation choice, though, that is specific to this demonstration. This function only takes in the initial weight vector as an input. Remember here, that a NumPy array here is equivalent to a vector. I did this so that we can use this function in gradient descent function we presented in a previous lecture. In step one here, we prepare the data. In the third line of this step, I add a constant vector of ones. This is something most packages will do for you under the hood. But this is how we get gradient descent method to learn the intercept. We can think of the intercept as the weight for a feature that has a constant value of one for all examples. From a gradient descent perspective, though, the intercept gets the same treatment as the features. Once we have the data, we get the predictions in step two, using the inverse logit, which uses the input weight vector and features in matrix X. After that, we compute the gradient of the log loss by translating the math I just displayed into Python code. While getting to the gradient equation requires multivariate calculus, which is outside the scope of this course, the implementation is straightforward. Another note about the implementation here — in this function, I hardcoded the data preparation, so this implementation will be specific to a given problem. There are definitely better ways to do this where we pass in the data instead and make it more dynamic. Next, we will implement the hessian function. Let's start with the math. Similar to how we compute the gradient, we start by getting the predictions, P at the current weight vector. After that, we compute the hessian in two steps. First, we compute this vector Q, which is just P times one minus P for each example. After that, we take the element-wise product with the Q matrix and our feature matrix X, and then we multiply the transpose of that resulting matrix against X. Our result is a K by K matrix, where again, K is the number of features plus one for the intercept. Let's see a Python function that computes this hessian for us. Similar to the gradient, we are going to first prepare the

data and compute model predictions. You can see here that we have to do these two steps in both the gradient and the hessian functions, which is a good cue that this code can probably be refactored and made more efficient. But again, I wrote it this way so that we can just pass these functions into the overall gradient descent function we've been using. Step three here is where the hessian is computed. I first compute the Q vector and I reshape it so that it can be multiplied against our matrix X. After that, I take the transpose of the left-hand matrix and take the dot product with the X matrix again.

Part 2

This video demonstrates how to implement gradient descent in Python using the gradient and Hessian functions that were just developed.

Video Transcript

Let's now put all this together and walk through a gradient descent step. I'll do this through code. The following code shows each step will run to make a full loop of the gradient descent algorithm. This represents the logic that would be part of the main loop of our gradient descent algorithm. I also show output of each step so we can better visualize what is happening. At the top here I initialize the process with a vector of zeros. This data set has three features plus an intercept, so we need a weight vector of linked four. Remember again, the intercept is just another weight the model has to learn. After that, I compute the gradient in the hessians. Notice the output of each function. I also do one extra step to the hessian, which is compute its inverse. This gives us the learning rate. Last we update our weight vector. This line is written to be compatible with arrays. The main difference is we're using the dot product between the learning rate matrix and the gradient vector. The learning rate is defined by a matrix instead of a scalar here, because we're running gradient descent on a multiple features at once. After this step, I print the updated weight vector. We can see that this is different than the zeros we started with. In a few moments we'll walk through a way to test whether this output is correct. But first, I want to present a full gradient descent function for logistic regression. This is the same full gradient descent function that was presented in an earlier video. Again, we pass in functions that compute the gradient and the hessian to determine the learning rate. These functions are defined to only input the weight factor. I have highlighted with red arrows parts of this function that were adapted to operate on arrays instead of scalar values. As an example, the last

arrow points to our convergence test. In this case, instead of checking whether one number isn't changing much, we sum up the element wise absolute differences of the current and prior weights. If this sum is below our tolerance, then we declare convergence. Running this function on our data is the equivalent of calling the fit method and circuit learn for the logistical regression package. I'd like to show an example of how the weight vectors changes with each iteration of the loop. In this example, application of gradient descent for logistic regression, we have a small data set with three features. In addition to the features, we're using Gradient Descent to learn the intercept. Each row shows the weight vector at each iteration of the process. We start at zero, but notice how quickly we get close to the converged result. Rows two through four only change by a small amount. An important question is how do we know this result is correct? We can of course use the weight vector here and evaluate our predictions on a test set and let that guide, ourselves. But I also want to test whether the gradient descent provided the right weights in the first place. One approach to do this is to compare our implementation with an implementation we trust. We do this in the following code, which compares our implementation of gradient descent with the same from second learn. This block of code I call our custom function and also fit a model using second learn. At the bottom I print the results of each. We can see these are pretty much the same with small differences in the lower decimal values. These differences are likely due to setting different tolerance levels for each process. Also second, learn uses regularization by default, which isn't used in our implementation. In this problem, it doesn't really make a difference. But for other problems it could where regularization has a stronger effect. If you're ever doing this, I'd recommend setting second learn C parameter to a very high level which minimizes the effect of regularization. Let me point out that if you're able to test a method like this against a well-supported implementation like second learn, you are generally better off using second learn. We built our own gradient descent here mostly as a learning exercise. Gradient descent is the method of choice for training advanced neural networks using applications like image classification, recommendation engines, and natural language processing. There are many variations of gradient descent. In this module, we covered a variant that uses the hessian matrix to inform learning rates, and this is a commonly used approach for logistic regression. With this week's material, you should be primed with a baseline understanding of the general gradient descent algorithm. This understanding will help you better navigate different

gradient descent variations you'll need to consider when you start your study of advanced neural networks.

Back to Table of Contents

Read: Training a Logistic Regression Model

The training process of a logistic regression model is accomplished iteratively, with the help of a loss function and an optimization algorithm. The goal is to find the model parameters (weights and intercept) that will result in the best model.

The optimization algorithm uses a loss function to evaluate a model's loss and adjusts the model parameters accordingly to reduce loss. It continues this process until an optimal model is produced.

☆ Key Points

A loss function is used to quantify the performance of a logistic regression model.

Training a logistic regression model involves finding the model parameters that result in a low training loss.

Gradient descent is an optimization algorithm that iteratively updates the model parameters until a loss function is minimized.

A common optimization algorithm used in logistic regression training is known as gradient descent. Figure 1 below demonstrates this iterative training process.

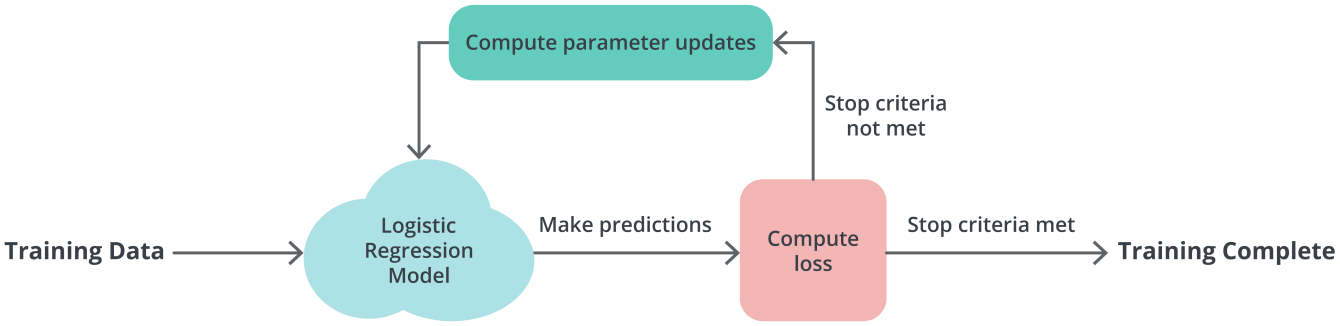


Figure 1. Iterative training process.

During model instantiation, the weights and intercept are usually set to some small random float numbers; this means the model will typically start off performing poorly since it was created at random. As it progresses iteratively throughout the training

phases, its weights and intercept become more and more optimized against the training data set and the performance improves.

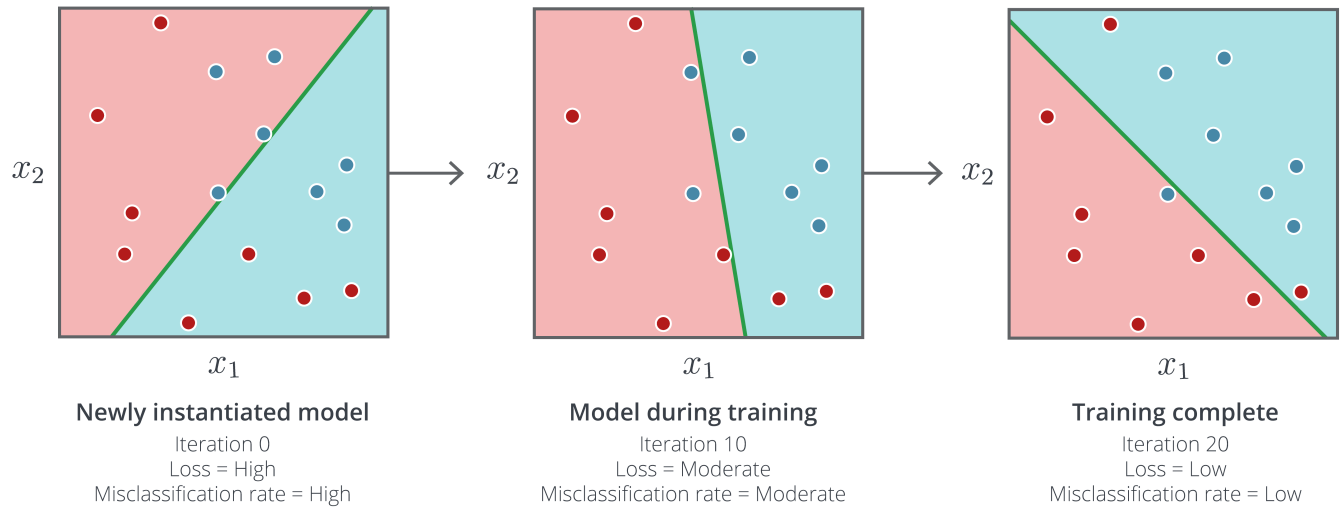


Figure 2. Training of a logistic regression model.

▼ Loss function

The loss function is a function that evaluates the performance of the model against training data at any point in time. When a model misclassifies most examples, its loss will be high, therefore the weights and intercept will need to be tweaked for optimization. When a model classifies most examples correctly, its loss will be low, and weights and intercept will no longer need to be updated, at which point training will have been completed.

The most common loss function used for logistic regression is the log loss function. The log loss function $L_{\text{LL}}^{(i)}$ for a single example i is provided below. In the function, y_i is the ground truth or the known label of the training example and P_i is the predicted probability. Keep in mind that this corresponds to the prediction probability of a given example belonging to class 1.

$$L_{\text{LL}}^{(i)} = -y_i \log(P_i) - (1 - y_i) \log(1 - P_i)$$

If an example is deeply misclassified, the loss function will yield a high value. If the example is correctly classified, it will yield a low value. Let's work out some

examples.

Example 1: Model correctly classifies an example:

Say we have a training example with a known class label of 1. The logistic regression's inverse logit function outputs a prediction probability of 0.96 that the example belongs to class 1. The probability of 0.96 is a very high probability. This means our model has correctly predicted the label for this example. Notice what log loss computes as a result. Also note we are using the natural log in our computations by default.

$$L_{\mathrm{LL}}^{(i)} = -1 \log(0.96) - (1-1) \log(1-0.96) \approx 0.04$$

It returns a value of 0.041. The loss for this training example is low because our model is successful at correctly predicting the label.

Example 2: Model incorrectly classifies an example:

Say we have a training example with a known class label of 1. The logistic regression's inverse logit function outputs a prediction probability of 0.014 that the class label is 1. This means our model has incorrectly predicted the label for this training example. Notice what log loss computes as a result:

$$L_{\mathrm{LL}}^{(i)} = -1 \log(0.014) - (1-1) \log(1-0.014) \approx 4.3$$

The loss for this training example is much higher than for the first training example because our model failed at predicting the label correctly. You are encouraged to try out various scenarios yourself to better understand the behavior of this loss function.

An actual training iteration includes many many training examples — or a batch, which is a subset of the entire training data set. For each iteration, we would evaluate the loss of our model against a set of training examples; this allows us to get a snapshot of the performance of our logistic regression model with the weights and the intercept at that particular point in time. The loss for a batch of examples is simply the sum of their individual losses divided by the number of examples; i.e., for N examples in a batch:

$$L_{LL} = -\frac{1}{N} \sum_{i=1}^{N} \Big(y_i \log(P_i) + (1-y_i) \log(1-P_i) \Big)$$

This average loss at the batch level is sometimes referred to as the "cost," so the loss function is sometimes called a cost function. If most examples are misclassified, the loss of this batch of examples will be high. If most examples are correctly classified, the loss of the batch will be low.

▼ Gradient descent

Now that we know how to evaluate the performance of our model using the loss function, we can learn how logistic regression uses log loss to determine the optimal model parameters (weights and intercept). Recall that there is one weight per feature, and so the number of weights equals the number of features.

One way to accomplish this is to apply an optimization algorithm called gradient descent. Logistic regression uses gradient descent to update the weights and the intercept of poorly performing models. We start with an arbitrary value for the model parameters; these can be a random value or, in many cases, zero. We then take the partial derivative of the loss function L_{LL} with respect to the weights vector W ($w_1, \ldots, w_n$) and the intercept α individually. This partial derivative tells us how much and toward what direction (positive or negative) to update the weights and intercept.

We then update the model parameters accordingly. The learning rate γ determines the step size, or the speed at which we update the model parameters. A high learning rate means every time we update the model parameters, we are taking a big step or making a big change to model parameters. A small learning rate means we are making a small change to the model parameters. A typical γ is between the range of 0.001 and 0.1.

Below are the formulas to compute the next value of one weight and the intercept at a given update step.

$$w_{t+1} = w_t - \gamma \frac{\partial}{\partial w} L_{LL}(w_t)$$

$$\alpha_{t+1} = \alpha_t - \gamma \frac{\partial}{\partial \alpha} L_{LL}(\alpha_t)$$

Below are the formulas to compute the next value of all weights W and the intercept at a given update step.

$W_{t+1} = W_t - \gamma \nabla L_{LL}(W_t)$, where ∇L_{LL} is the gradient vector defined by $\nabla L_{LL}(W) = \left(\frac{\partial}{\partial w_i} L_{LL}(W), \ldots, \frac{\partial}{\partial w_n} L_{LL}(W) \right)$

$\alpha_{t+1} = \alpha_t - \gamma \frac{\partial}{\partial \alpha} L_{LL}(\alpha_t)$

Figure 3 below shows how gradient descent works to adjust one weight during each update step (a similar effect goes for intercept). In the image, we are working with one weight w for the purpose of illustration. In reality, all of the weights W will be updated in a similar manner.

The curve is a plot of the log loss function. The gradient of the log loss is equal to the partial derivative $\frac{\partial L_{LL}}{\partial w}$ (slope) of the curve evaluated at the current value of the weight w_t, and it tells you in which direction and how much to move toward the bottom of the curve. The gradient descent algorithm takes a step in that direction in order to reduce loss, updating w using the update equation. The log loss function is convex (i.e., only has one minimum).

Therefore, the goal is to iteratively update the value of w to reduce loss until we reach that minimum; that is, until the slope is at or close to zero. At this point, our model will be performing as best as it can against the training data set and we have identified the value of w that will result in the lowest training loss.

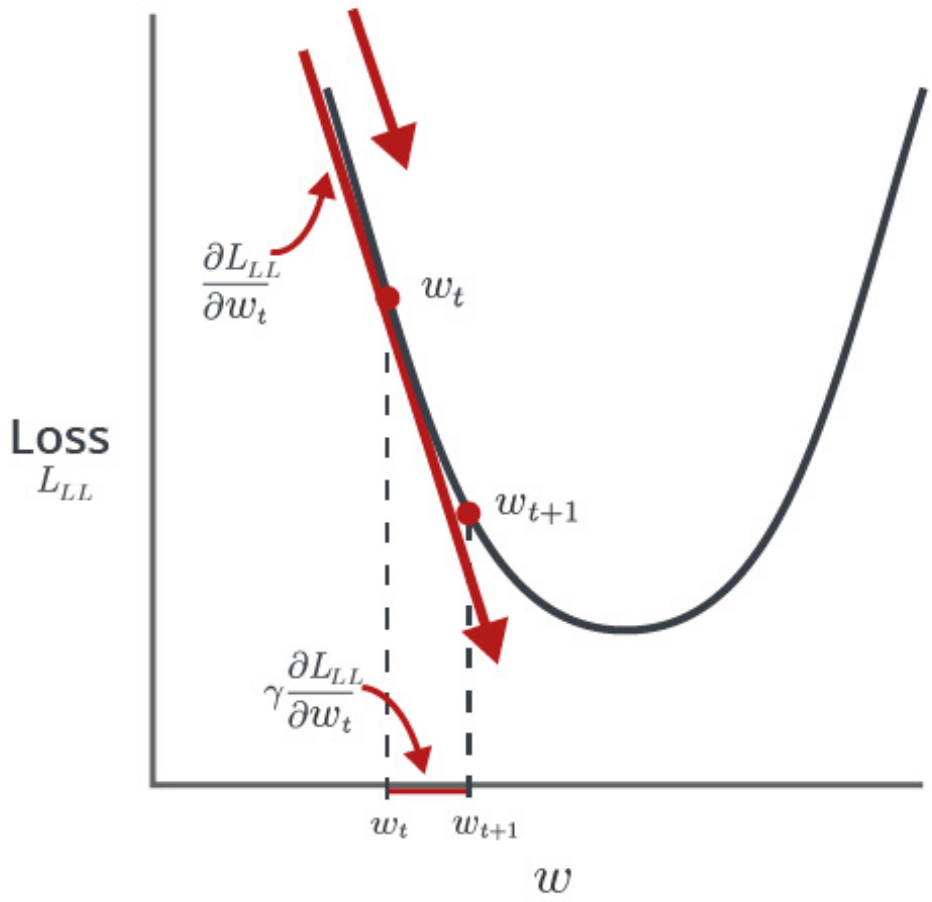


Figure 3. Gradient descent to find the minimum loss with respect to one weight w.

Summary

Logistic regression is an iterative algorithm that uses loss functions to determine how well a model is currently performing. A logistic regression model typically starts off with random weights and an intercept, which is likely to result in poor predictions, thus having high loss. We adjust the weights and the intercept with gradient descent in order to improve the model performance against a batch of training examples. We continue to do so iteratively and evaluate the loss function during each iteration. When the loss has become small enough and the performance no longer improves, we stop the training process and lock in the weights and intercept. During prediction, the weights and intercept are used in the linear step of a logistic regression model to calculate the linear combination as the output. This output is passed into a sigmoid

function to get a probability that the label is 1. For high probability, it is mapped to a label of 1, and low probability is mapped to a label of 0.

The pseudocode for training a logistic regression model is as follows:

1. Initialize a random number $\boldsymbol{\alpha}$ and set of random numbers $\boldsymbol{W}$ equaling the number of features.
2. Choose a learning rate γ.
3. For the set of training examples, calculate the log loss $\boldsymbol{L_{LL}}$.
4. Update $\boldsymbol{W}$ and $\boldsymbol{\alpha}$ using gradient descent:

$$\boldsymbol{W_{t+1} = W_t - \gamma \nabla L_{LL}(W_t)}$$

$$\boldsymbol{\alpha_{t+1} = \alpha_t - \gamma \frac{\partial}{\partial \alpha} L_{LL}(\alpha_t)}$$

5. Repeat until the change in loss is smaller than some threshold.

Back to Table of Contents

Logistic Regression Hyperparameter: Learning Rate

Just like KNN and decision trees, logistic regression has its own set of hyperparameters. By optimizing these hyperparameters, we will be able to produce a model that generalizes well. One very common hyperparameter for logistic regression is the learning rate.

☆ **Key Points**

One common hyperparameter for logistic regression is the learning rate.

The ideal learning rate is one that reaches global minima in a fast and efficient manner.

The learning rate, γ, also commonly known as the step size, is a hyperparameter that dictates the speed of gradient descent.

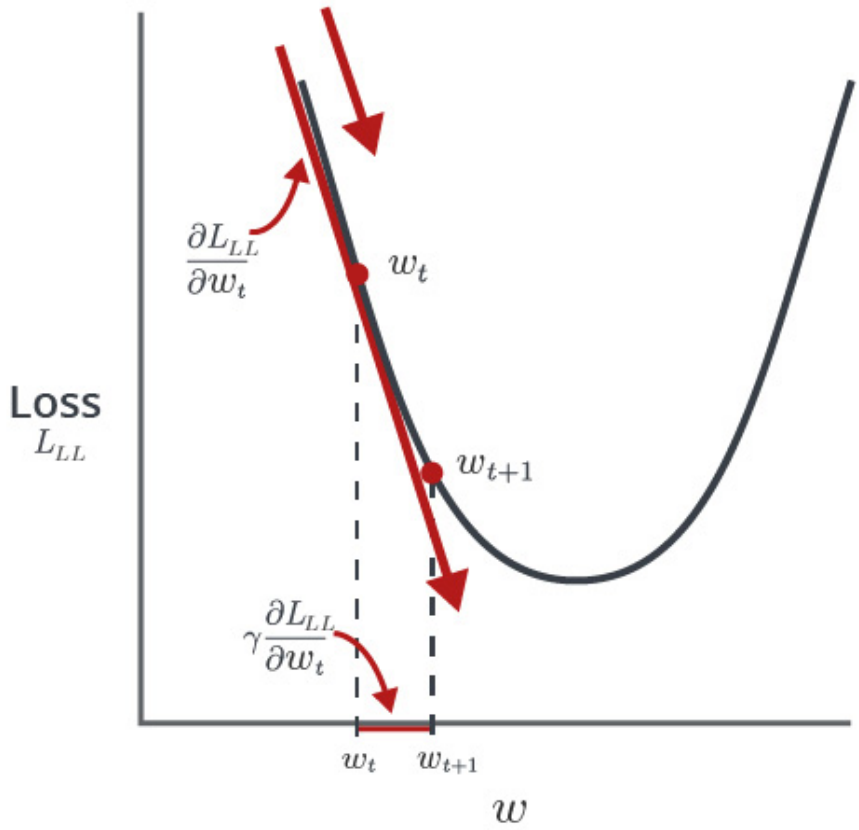


Figure 1. Gradient descent example.

A high learning rate updates the weights and intercept in big increments. A low learning rate updates the weights and intercept in small increments. Recall that in gradient descent, our goal is to reduce the loss of our model. With that in mind, let's explore the effects of various learning rates.

▼ High learning rate

When the learning rate is too high, the weights can fail to minimize. They could just bounce around back and forth near the bottom but never get close enough to the minima because the learning rate is too large.

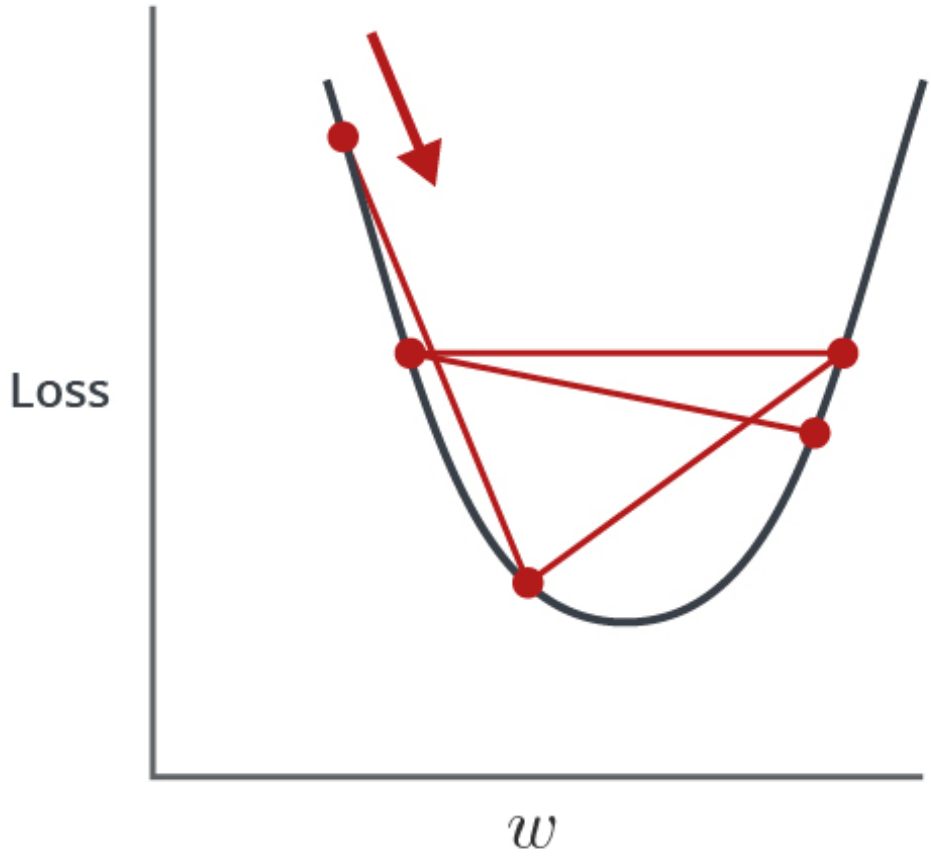


Figure 2. Gradient descent with a high learning rate

▼ **Low learning rate**

When the learning rate is low, it will take a long time to train the model since the weights and intercept change in very small increments during training.

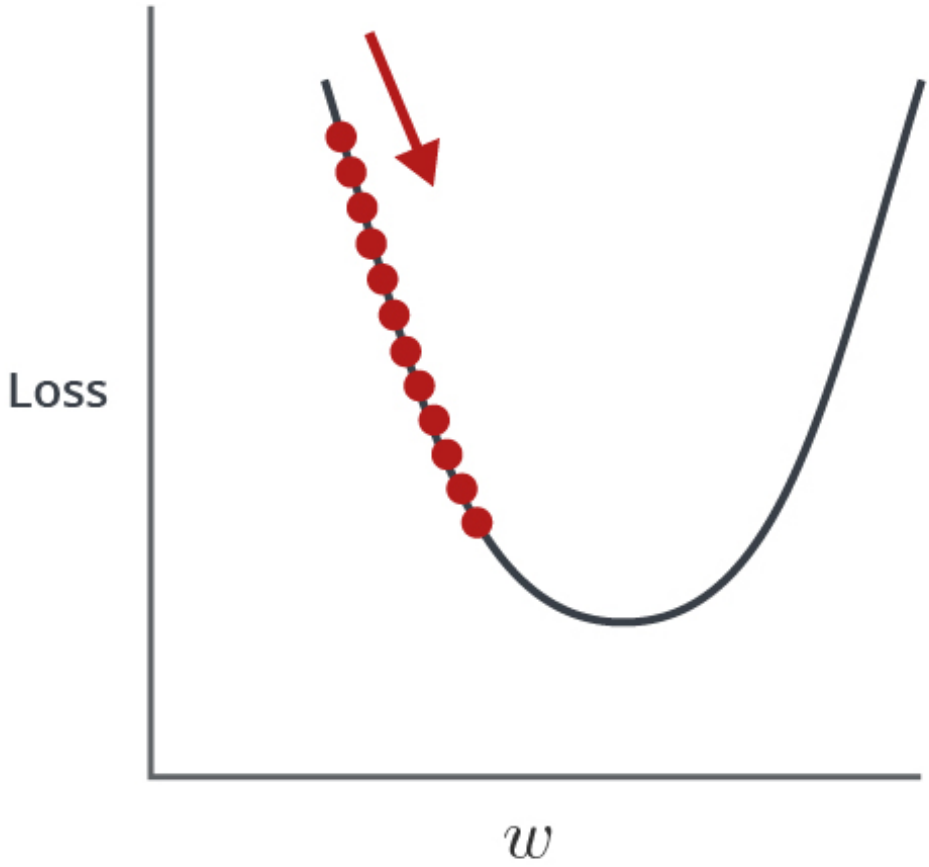


Figure 3. Gradient descent with a low learning rate.

▼ **Ideal learning rate**

An ideal learning rate allows fast convergence to the minima without overshooting.

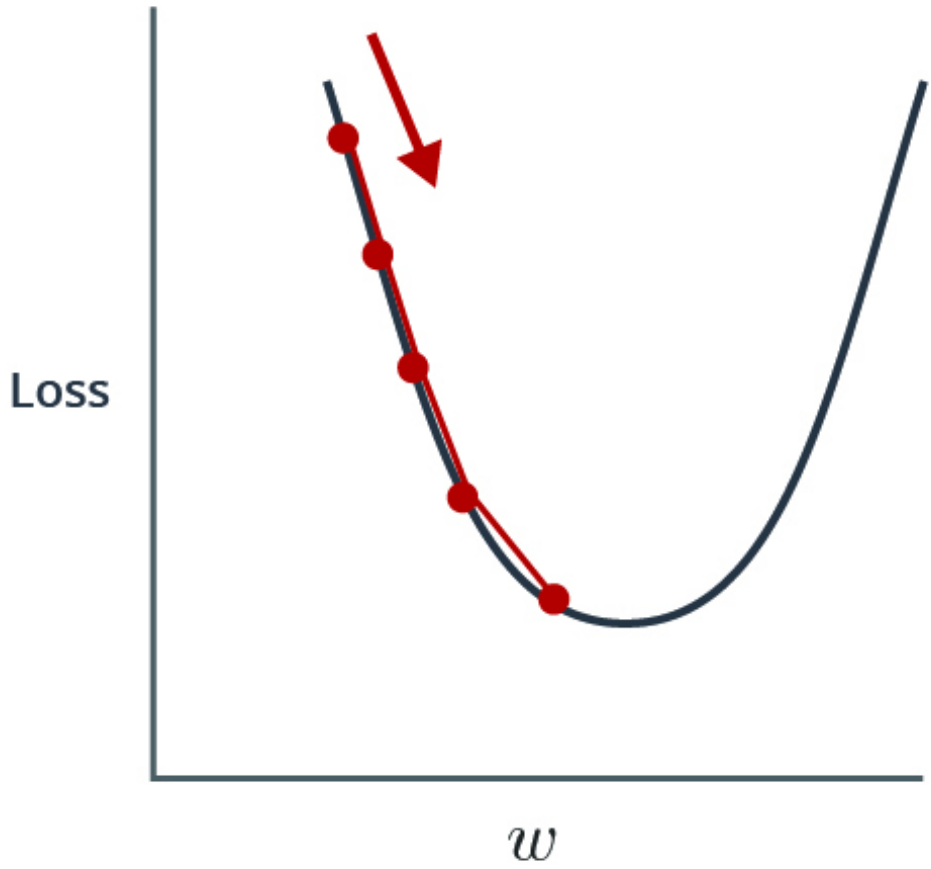


Figure 4. Gradient descent with an ideal learning rate.

Choosing the right learning rate ensures efficient training of the model, thereby reducing cost and improving model accuracy.

Back to Table of Contents

Tool: Logistic Regression Cheat Sheet

The tool linked to this page provides an overview of logistic regression for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this modeling approach.

Back to Table of Contents

Download the Tool

Use this **Logistic Regression Cheat Sheet** as a quick way to review how logistic regression works.

Regularization

Our goal during training is to minimize loss. Recall, however, that a model can have low training loss but not generalize well to new data, as seen in the graph below. This is known as overfitting and is caused by a complex model that has learned the training data so closely that it does not generalize well to new data.

☆ Key Points

Regularization controls model complexity by modifying the loss function with a penalty term to prevent weights from growing out of proportion.

The two types of regularization are L1 and L2 regularization.

One common hyperparameter for logistic regression is the regularization hyperparameter C, which controls how much regularization is applied to the model.

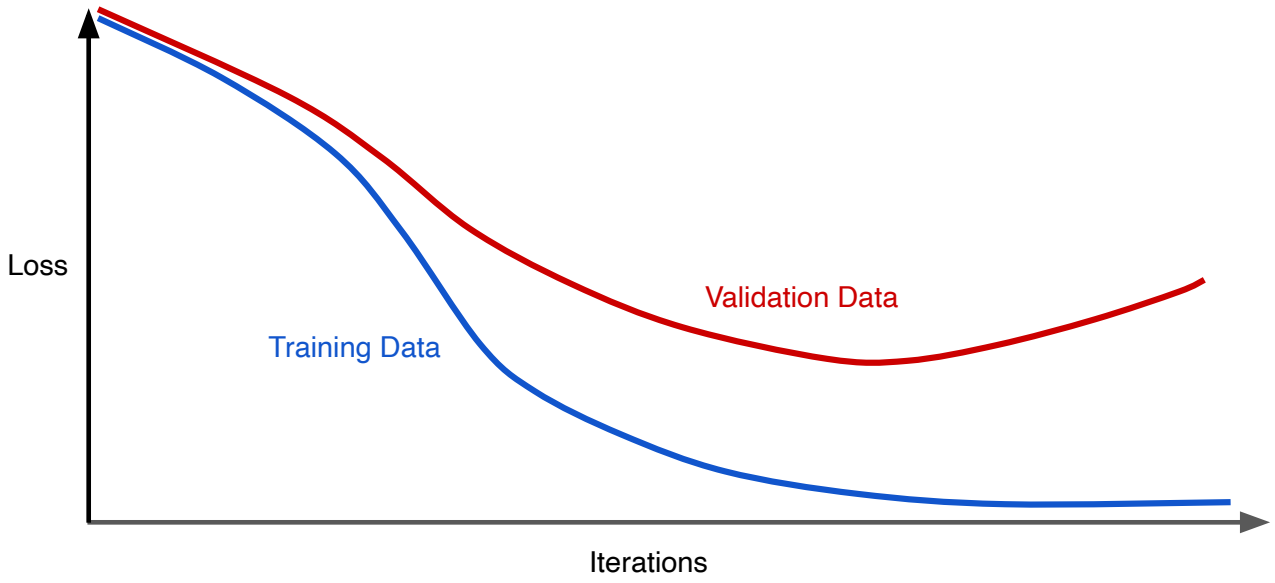


Figure 1. A model with low training loss that does not generalize well to new data.

Our goal should therefore be to minimize loss and avoid overfitting by minimizing the model's complexity. Regularization is a technique that accomplishes this goal by penalizing complex models in an attempt to prevent overfitting. By adding regularization terms, which are functions of the weights, we can mitigate overfitting by "punishing" a model with extreme values for any given weight.

There are two ways to think of model complexity:

- Model complexity as a function of the *total number of features* with non-zero weights
- Model complexity as a function of the *weights* of all the features in the model

There are two types of regularization used to address these two types of model complexity: L1 and L2 regularization. L1 (also known as Lasso) and L2 (also known as Ridge) regularization prevent weights from growing out of proportion. Oftentimes our feature values can be too large or too imbalanced. Due to such conditions, these features can overly influence the size of their corresponding weights. L1 and L2 add a penalty to the loss function for weights that have grown too large.

These penalty terms (also called regularization terms) can be seen in the following formulas, in which L represents the loss as computed by the loss function, m is the total number of weights, and C is a hyperparameter discussed below. Regularized loss involves adding the penalty term to the loss.

$L_{\text{L1 regularized}} = L + \frac{1}{C}\sum_{j=1}^{m} |w_j|$

- L1 regularization introduces a term that penalizes less-important features, reducing their coefficients to zero and effectively eliminating them. L1 penalizes the count of non-zero weights; namely, L1 *penalizes |weight|.*

$L_{\text{L2 regularized}} = L + \frac{1}{C}\sum_{j=1}^{m} w_j^2$

- L2 regularization introduces a regularization term to the loss function that is the sum of squares of all feature weights; namely, L2 *penalizes* $weight^2$.

Let's look at an example of L2 regularization. Say we have training data with four features. This means our model will have four weights. Recall that in logistic regression, the goal is to find the weights that minimize the loss. Through training, we

compute the loss for various weight values until we find the appropriate ones. Let's say that at a given point during training, the values of the four weights are:

$w_1 = 3.1, w_2 = 5.2, w_3 = 1493, w_4 = -4.3$.

Our loss function will return an extremely high value since w_3 has a very high value. We can see that w_3 is the main cause of the model complexity as it has a much higher value than the rest of the weights.

Using the formula for L2 regularization, we will determine the L2 regularization term that will be added as a penalty to the loss function:

$$\begin{aligned} w_1^2 + w_2^2 + w_3^2 + w_4^2 &= (3.1)^2 + (5.2)^2 + (1493)^2 + (-4.3)^2 \\ &\approx 2,229,104 \end{aligned}$$

The L2 regularization term that will be added to the loss function as a penalty is approximately 2,229,104. This encourages our model to reduce the model complexity.

To further solidify our understanding of L1 and L2 regularization, we will visualize their penalties in the image below.

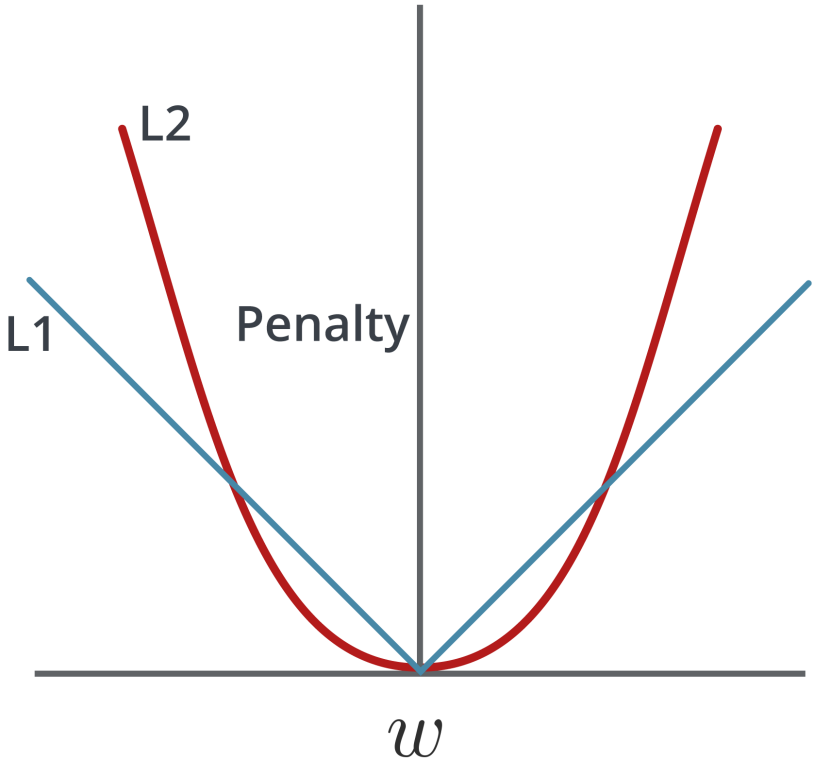


Figure 2. Penalties of L1 and L2 for possible values of one weight $\boldsymbol{w}$.

As the magnitude of $\boldsymbol{w}$ becomes larger, the penalty will be higher for both L1 and L2. L1, however, tends to exert a downward pressure regardless of the magnitude of $\boldsymbol{w}$. Even as the weight gets closer to zero, L1 will still exert pressure to attempt to bring it even closer to zero. L2, on the other hand, gradually reduces downward pressure as a weight approaches zero. This is why L1 regularization is more conducive to a sparse model in which some of the weights end up having little or no effect on the model. In that regard, L1 can almost be considered an act of feature selection, as it eliminates some of the weights of the features.

Hyperparameter C

The penalties are often accompanied by a factor $\frac{1}{C}$ that controls the strength of the penalty. This hyperparameter C controls how much regularization is applied to the

model. Along with choosing the right regularization (L1 or L2), the value of C can be fine-tuned to improve the performance of the model.

Back to Table of Contents

Assignment: Unit 4 Assignment - Optimizing Logistic Regression

In this assignment, you will use regularization to train logistic regression models using scikit-learn. You will train multiple models using different values of regularization hyperparameter C and will plot the resulting accuracy scores and log loss.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.

Back to Table of Contents

Assignment: Unit 4 Assignment - Written Submission

In this part of the assignment, you will answer 6 questions about linear models.

The questions will prepare you for future interviews as they relate to concepts discussed throughout the unit. You've practiced these concepts in the coding activities, exercises, and coding portion of the assignment.

Completion of this assignment is a course requirement.

Instructions:

1. Download the **Unit 4 Assignment document**.
2. Answer the questions.
3. Save your work as one of these file types: .doc, .docx. No other file types will be accepted for submission.
4. Submit your completed Unit 4 Assignment document for review and credit.
5. Click the **Start Assignment** button on this page, attach your completed Unit 4 Assignment document, and then click **Submit Assignment** to send it to your facilitator for evaluation and credit.

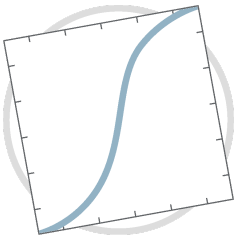
Back to Table of Contents

Module Wrap-up: Train a Logistic Regression Model

In this module, Mr. D'Alessandro explained the training process of a logistic regression model. In particular, you have learned how to use gradient descent to update the model parameters of a logistic regression model. You gained an understanding of learning rates and regularization, as well as the impacts they have on the model training process.

Back to Table of Contents

Module Introduction: Introduction to Linear Regression



In this module, you will be introduced to a popular supervised machine learning algorithm used for regression problems known as linear regression. Similar to logistic regression, this model also makes predictions based on the linear combination of model weights and features. The major difference between logistic and linear regression is that linear regression is used specifically for solving regression problems that have a continuous label such as age, temperature, distance, or salary. You will practice implementing your own linear regression model using scikit-learn.

Back to Table of Contents

Ask the Expert: Kathy Xu on a Real-World Example Using Linear Regression

In this video, Ms. Xu talks about an experience where she used linear regression to solve her ML problem.

Note: The job title listed below was held by our expert at the time of this interview.

Kathy Xu
Analytics Extensibility Lead, Pfizer

Kathy (Qingyu) Xu leads a team responsible for extending analytics capabilities across Pfizer. Her team helps create machine learning-powered products (web apps, plugins, and dashboards) for use across the enterprise and provides guidance for fellow data scientists and analysts to optimize their usage of data and analytical technologies that are a part of the enterprise analytics platform.

In particular, Ms. Xu has an interest in MLOPs and industrializing machine learning models. Outside of work, she is an avid art history lover and frequently visits the museums around NYC, such as the Cooper Hewitt and Brooklyn Museum. Ms. Xu received her Master and Bachelor of Science in Statistics from Cornell University.

Question

Can you provide an example of when you used a linear model vs. another model?

Video Transcript

I'm part of the digital organization and particularly the platform and product organizations. One of the questions we were trying to answer is we help manage a lot of the different types of analytical tools that are leveraged throughout our different teams. We wanted to see if we can infer how much RAM or CPU is being used for some of the different tools that we manage. We were able to use linear regression to infer that by inputting different types of information, such as the size of the data and time of day. In this case, the linear regression model for us was really helpful and was really easy to implement. Because most of the input data in comparison to the output had some sort of of linearity, that was really helpful. In other use cases, we've used

linear models such as linear regression as our baseline model. We've used this as a way for us to help create a baseline model to predict or forecast different types of product demand. However, we found that, in this case, other models were a little bit better just because there wasn't so much linearity between the input and output information.

Back to Table of Contents

Read: Linear Regression

In this unit, we have been focusing on logistic regression, a linear model that is best suited to solve binary classification problems by predicting the probability of a class label, such as the probability that an email is spam.

There is another very commonly used supervised learning algorithm called linear regression that is similar to logistic regression in that it makes predictions based on the linear combination of model weights and features. However, linear regression is used to solve regression problems. It is worth emphasizing that while both algorithms have the word “regression” in their names, logistic regression is intended for classification problems, and linear regression is intended for regression problems. While logistic regression is used to predict the probability of a categorical label, linear regression is used to predict a continuous label, such as a price or an age.

☆ Key Points

Linear regression is a popular supervised machine learning algorithm used for regression problems

Linear regression finds a linear relationship between one or more features and a label. It fits linear model to the data

The Ordinary Least Squares (OLS) method is used in linear regression to minimize the sum of the squared errors between the model predictions and the actual values

Linear regression finds a linear relationship between one or more features and a label. It fits a linear model to the data by assuming that the data relationship is well described by a straight line - more specifically, a straight line **is** the model of the data. It then uses that line to predict a label for a new unlabeled example.

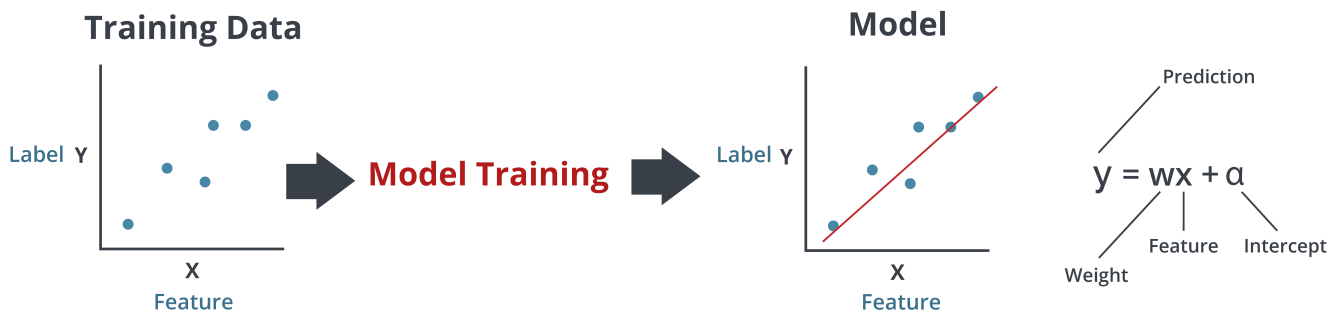


Figure 1. Linear regression

This difference between linear regression and logistic regression can be visualized in Figure 2 below where a logistic regression model is represented as a decision boundary between two classes, and a linear regression model is represented as a fitted line.

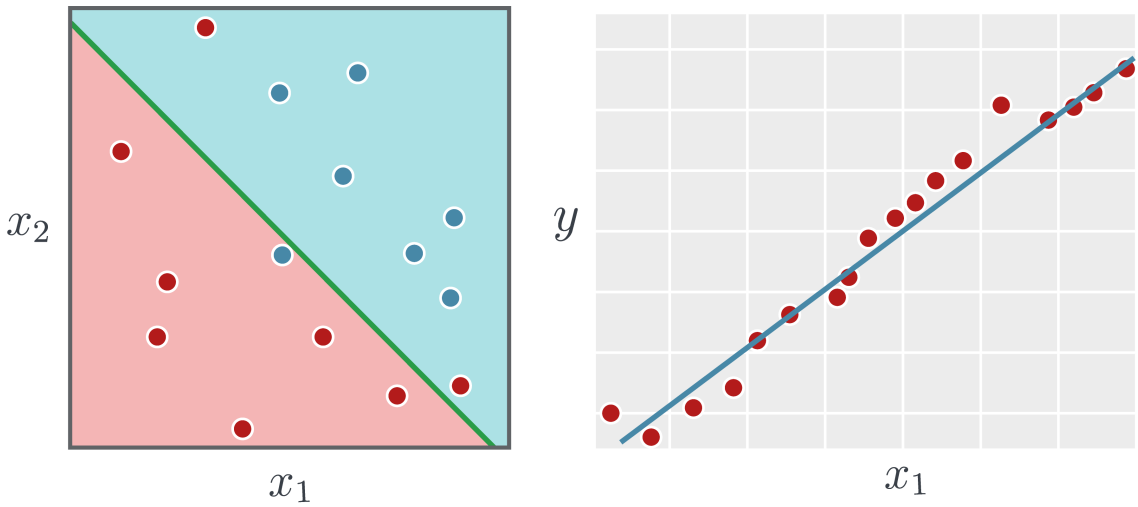


Figure 2. Logistic Regression (left) and Linear Regression (right) models visualized

Making Predictions with Linear Regression

There are two types of linear regression models. Simple linear regression finds the linear relationship between one feature and one label, and multiple linear regression finds the linear relationship between multiple features and one label. Note that a

feature is often referred to as an independent variable and a label is often referred to as a dependent variable.

To make a prediction with a linear regression model, we simply take the feature values of a new unlabeled example and input them into the fitted line (the equation). This is done mathematically simply by taking the linear combination of the feature values ($x_1, x_2, \ldots, x_n$) and the learned weights ($w_1, w_2, \ldots, w_n$) plus an intercept term (α) as shown below. The result (y) is the label, or prediction. Note that the intercept is commonly referred to as the bias β.

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \alpha$$

For a simple linear regression, the prediction for an example would be determined using $y = w_1 x_1 + \alpha$. For multiple linear regression, the model will be a fitted hyperplane. Figure 2 shows a multiple linear regression model for examples with two features where the vertical lines between the points and line/hyperplane represent the error. The prediction for an unlabeled example would be determined using $y = w_1 x_1 + w_2 x_2 + \alpha$.

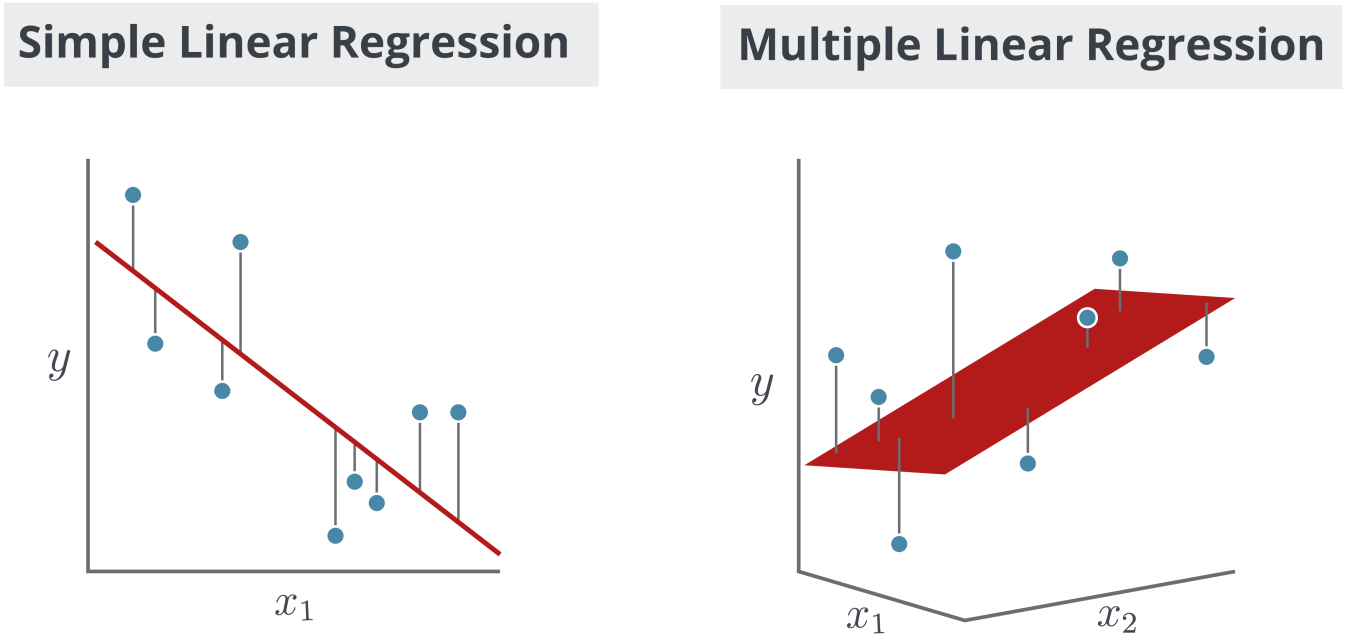


Figure 3. Linear Regression trained with a 1-dimensional feature (left) vs Linear Regression trained with 2-dimensional feature (right).

To further solidify our understanding of linear regression, let's take a look at an example in action. Let's say we want to learn a model to predict the relationship between salary and the number of years of on-the-job experience in a given field. First, let's plot the data we have.

Figure 4. Salary and years of on-the-job experience data plot

Notice that the salary increases with the number of years of experience. This shows that the relationship between the number of years of experience and the salary is a linear relationship. We can draw a straight line to approximate this relationship.

Figure 5. Linear relationship between years of experience and salary

Despite the fact that the line doesn't pass through every example, the line does clearly show the relationship between the feature and the label. We can therefore use the equation for a line ($\boldsymbol{y} = \boldsymbol{wx} + \boldsymbol{\alpha}$) to model this relationship.

Once our model is trained and we have the best fit line, we will have the intercept ($\boldsymbol{\alpha}$) and the slope ($\boldsymbol{w}$). All we have to do is substitute our unlabeled example's feature value for $\boldsymbol{x}$ and get the prediction ($\boldsymbol{y}$).

From the plotted line you can see that you can predict that someone with 20 years of experience will make 70K.

Training a Linear Regression Model

The goal of linear regression is to fit a line to the training data and use that line to make predictions for new data. Therefore, to train a linear regression model is to estimate the model parameters (weights and intercept) that best fit against the training data set.

There are different methods to accomplish this. One of these is a technique called Ordinary Least Squares (OLS). OLS is a non-iterative technique used to estimate the model parameters that minimize the sum of the squared errors (SSE), i.e. the square of the differences, between the model's predictions and the actual values. You can see in the image below that the “error” is the vertical distance between a training example and the fitted line. OLS seeks to find the line that minimizes the sum of the squared distances between each training example and the line.

In other words, the algorithm chooses the model parameters (slope and the intercept of the line) in a way that the total area of all of these squared distances is as small as possible. We can therefore also view OLS as minimizing the MSE (mean squared errors) over all of the training examples in the training data.

The math behind OLS is rather complex to demonstrate. As an MLE, you would typically perform OLS using an existing library such as NumPy’s **np.linalg.lstsq** function, or scikit-learn’s **Linear Regression** class.

Figure 6. Training a linear regression model

Note that while OLS is a non-iterative method, training a linear regression model can also be an iterative process. This iterative training process will use the same gradient descent optimization algorithm used in logistic regression but will minimize a loss function used for regression, such as the MSE loss function.

Back to Table of Contents

Linear Regression Demo

In this demo, you will see how to implement a linear regression model using scikit-learn. You will work with the World Happiness Report (WHR). The WHR is a yearly summary of various economic and social indicators for countries around the world, linked to summaries of happiness as reported by people living in those countries. This type of panel data across countries and years is typical of many real-world data sets. For more information about this data set, consult the **WHR 2018 website.**

You will compute simple and multiple linear regressions among some of the variables in the WHR using the OLS method and will analyze your models using the MSE loss function. You will also learn how to iteratively train a linear regression model using scikit-learn's implementation of gradient descent.

This activity will not be graded.

The full contents of this page cannot be rendered in the course transcript. Log in to the course to view it.

Back to Table of Contents

Module Wrap-up: Introduction to Linear Regression

In this module, you were introduced to a model used to solve regression problems that have a continuous label. This model is known as linear regression. You were able to practice implementing this model using scikit-learn and saw how it can be used to make predictions on common real-world data. You experimented with both simple and multiple linear regression and were introduced to common evaluation metrics used for regression problems.

Back to Table of Contents

Lab 4 Overview

In this lab, you will build a logistic regression model from scratch and compare it to scikit-learn's logistic regression implementation. You will be working in a Jupyter Notebook.

This three-hour lab session will include:

- **10 minutes** - Icebreaker
- **30 minutes** - Week 4 Overview and Q&A
- **20 minutes** - Breakout Groups: Big-Picture Questions
- **10 minutes -** Class Discussion
- **10 minutes - Break**
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 1
- **15 minutes** - Working Session 1 Debrief
- **30 minutes** - Breakout Groups: Lab Assignment Working Session 2
- **15 minutes** - Working Session 2 Debrief
- **10 minutes -** Concluding Remarks and Survey

By the end of Lab 4, you will:

- Load and split the data into training and test sets.
- Write a Python class that will train a logistic regression model.
- Compare your implementation to scikit-learn's implementation.

Back to Table of Contents

Assignment: Lab 4 Assignment

In this lab, you will continue working with the Airbnb NYC "listings" data set.

This assignment will be graded.

When you finish your work:

1. Save your notebook by selecting the "Save and Checkpoint" entry from the "File" menu at the top of the notebook. If you do not save your notebook, some of your work may be lost.
2. Submit your work by clicking **Education —> Mark as Completed** in the upper left of the Activity window 

3. Note: This assignment will be manually graded by your facilitator. You cannot resubmit.

This lab assignment will be graded by your facilitator.

The full contents of this page cannot be rendered in the course transcript. Please complete this activity in the course.