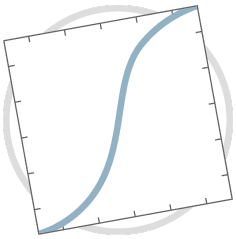
Machine Learning Foundations Unit 4

Table of Contents

- Watch: Using Gradient Descent in Logistic Regression Training
- Read: Training a Logistic Regression Model
- Logistic Regression Hyperparameter: Learning Rate
- Tool: Logistic Regression Cheat Sheet
- Regularization
- Assignment: Unit 4 Assignment - Optimizing Logistic Regression
- Assignment: Unit 4 Assignment - Written Submission
- Module Wrap-up: Train a Logistic Regression Model

Module Introduction: Introduction to Linear Regression

- Read: Linear Regression
- Module Wrap-up: Introduction to Linear Regression
- Assignment: Lab 4 Assignment

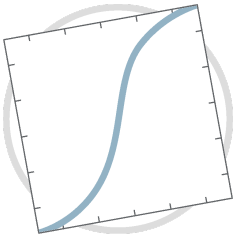


Unit 4 Overview
Video Transcript

In this week, we're going to continue learning fundamental machine learning algorithms, and focus specifically on a method called logistic regression. We are going to approach this method a little differently, though. After first presenting the foundations and practical considerations for logistic regression, we are going to take a deep dive into the algorithms that are used to train a logistic regression under the hood. We are going to learn a method called gradient descent, which is a more general purpose numerical optimization technique you use when you are trying to find model weights that minimize a loss function during training. We can think of logistic regression as the starting place for more advanced AI algorithms such as neural networks. By diving so deeply into logistic regression, both in terms of how to use and understand the method, but also on how it is trained, you will gain a solid foundation for learning machine learning functions from data.

What You'll Do:

- **Analyze the mechanics of logistic regression**
- **Understand the purpose of using gradient descent and loss functions**
- **Explore common hyperparameters for logistic regression**
- **Define the core math concepts required to solve common machine learning problems**
- **Use NumPy to perform vector and matrix operations**
- **Explore how linear regression works to solve real world regression problems**



Unit Description

Linear models are a class of supervised learning models that are represented by an equation and use a linear combination of features and weights to compute the label of an unlabeled example. Linear models are simple to implement, fast to train and have lower

---

complexity. In this unit, you will discover one of the most powerful linear models used in classification known as logistic regression. Logistic regression is used to predict the probability of an outcome. While the focus of the unit will be on logistic regression, you will also be introduced to a common linear model used to solve regression problems: linear regression.

Mr. D'Alessandro will introduce important concepts specific to the training of linear models. These include the optimization algorithm, gradient descent, and the evaluation tool, the loss function. Implementing a linear model from scratch requires a basic understanding of math using vectors and matrices. You will be given the opportunity to implement a logistic regression model from scratch by using NumPy. You will also see a demonstration of how a linear regression model can be used to solve real world regression problems.

Back to Table of Contents

Tool: Unit 4 Glossary

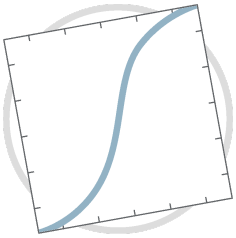
Most of the new terms you will learn about throughout the unit are defined and described in context, but there are a few vital terms that are likely new to you but undefined in the course videos. While you won't need to know these terms until later in the unit, it can be helpful to glance at them briefly now. Click on the link to the right to view and download the new terms for this unit.

Download the Tool

Download and save this **Unit 4 Glossary Tool** to use as a reference as you work through the unit and encounter unfamiliar terms.

Back to Table of Contents

Module Introduction: Introduction to Logistic Regression



Logistic regression is a powerful supervised learning algorithm used in classification problems to predict the probability of a binary outcome (e.g. the probability that an email is spam). Logistic regression belongs to a class of models known as linear models. In this module, Mr. D'Alessandro introduces linear models and the loss function. He explains how the loss function is used in the training process of a linear model to evaluate its performance so as to improve the model.

After you familiarize yourself with logistic regression and loss functions, you will implement your own logistic regression model using scikit-learn.

Back to Table of Contents

Watch: Linear Models

In these videos, Mr. D'Alessandro introduces the linear model. Linear models are supervised learning models that use the linear combination of features and weights to compute the label of an unlabeled example. These models attempt to improve themselves during training by using an optimization technique that minimizes training loss. The optimization technique uses a loss function to quantify the amount of error the model makes against the training dataset.

Watch as Mr. D'Alessandro describes the linear model and its advantages and disadvantages, and explains which loss functions are commonly used for classification and regression problems.

Part 1

In this video, Mr. D'Alessandro describes how the loss function is used to evaluate the performance of linear models. A common loss function used for classification is log loss, and a common loss function for regression is mean squared error. Watch as Mr. D'Alessandro further explains loss functions and how they can be used to guide the model training process for specific models.

Please note that Mr. D'Alessandro mentions "negative" at 2:38, it should actually be exponentially more positive instead.

Video Transcript

One class of supervised learning algorithms involves learning a mathematical function from data. Just like a decision tree can be represented by a tree graph that makes Boolean comparisons, these models can be represented by a single equation. Model training involves initializing weights in the equation with a random guess and then slowly adjusting the weights in a principled way to improve the model's prediction performance. This class includes logistic regression, neural networks, and support vector machines. Each method uses different equations, but what they all have in common is they start with a loss function, and the loss function guides the process for adjusting the model's weights. Also, these methods are different from KNN and decision trees because the loss functions are used explicitly to guide the training process. At a very high level, a loss function is a mathematical way of representing a model's prediction error. Every supervised learning algorithm is designed to produce a prediction. That prediction, again, can be a class label, a probability, or a regular number. We naturally need to know whether these predictions were correct and the loss function enables us to exactly quantify that correctness. There are several loss functions to choose from. Each one, of course, has a mathematical form, but here, I'm going to introduce them as code instead. Each loss function takes in two inputs, which are the predicted value and the ground truth label. A popular loss function used for training classification models is called the log loss. This loss function is used for both logistic regression and neural networks. NumPy allows us to do element-wise mathematical operations on entire arrays with just one line, like you can see here. The bottom

function that I'm showing would be much faster than looping through the top, which does element-wise operations. Now, one major advantage of log loss is because it compares prediction probabilities to ground truth, it enables us to measure different shades of correctness. To understand how log loss works, let's focus just on the actual formula. Here we have a plot of the log loss as a function of the prediction probability. Since the ground truth is either 1 or 0, the value we're using for our loss is the log of either the prediction or 1 minus the prediction, again, with the prediction being a probability. When the ground truth is 1, our loss is driven by the red line shown here. If our predicted probability is close to 1, we incur close to 0 loss. But as the predicted probability goes to 0, the loss gets exponentially more negative. The trend is flipped when the truth is 0, which is represented here by the black line. This plot demonstrates how we can do better than just determining absolute correctness. With log loss, we can again quantify varying degrees of correctness. Now, for aggression problems, the most common loss function you'll encounter is called the mean squared error. The pictures here show both the code for computing it, as well as a plot of how it looks. As we can see when the prediction exactly equals the ground truth, we get a loss value of 0. There will be more loss functions we'll learn when we discuss evaluation. Many loss functions are used just for evaluation purposes because they are not suitable for numeric optimization techniques, which is a core requirement for using loss functions when doing model training. But when we consider using a loss function to specifically guide the training algorithm to update model weights, we will focus just on log loss and mean squared error.

Part 2 This video explains what it means for a model to be linear and non-linear and the significance that this has on model performance. Mr. D'Alessandro compares the pros and cons of linear models and non-linear models and introduces the logistic regression linear model intended to for binary classification problems.

Video Transcript

Now that we've learned about loss functions, we're going to introduce logistic regression. Logistic regression is a linear classification method that is trained by iteratively tuning a set of weights to optimize the log loss. When we say a

classification model is linear, we mean the function that maps the features to the output is a straight line of the form Y equals A plus X times B, where B is the weight. When we look at it geometrically, a linear model is one that uses a straight line as the classification boundary. On the left-hand side here, points on either side of the red line would be predicted to be in different classes. The opposite of a linear model is just a model that is called nonlinear. Nonlinear models can take on many forms, but when abstract into a plot like this, it is the same as taking the straight line from the left and bending it to a more suitable shape. Nonlinear models have more complexity, meaning they can draw more sophisticated curves to fit arbitrary patterns in the data. Here is an example of distribution where the positive cases are in a spherical cluster sitting inside a larger cluster of negative cases. No linear model can fit this shape in a way that gets good classification accuracy. A nonlinear model might be able to fit it perfectly, on the other hand. For instance, the pattern on the right can be fit with a two-dimensional parabola, but not a straight line. The most common nonlinear model that uses log loss is a neural network. Neural networks generally fit data better, but with their increased complexity, they can lead to massive overfitting and poor generalization in certain cases. Thus, it is important to learn to embrace even the simpler techniques like logistic regression. Given these comparisons of linear to nonlinear models, one may be even asking, why would we ever use linear models? Choosing a linear versus a nonlinear model is what we would call a design decision. The fundamental rule to observe in machine learning is that the best design decisions depend on the problem. There are certainly many cases where simpler, linear models will be your best option. First of all, linear models have lower complexity. This makes them often better when you have a lot of constraints to consider. Some example constraints might be if you have a smaller data set and a lot of features, or if you need to be able to explain your model to your stakeholders, or if your system requires you to make very fast predictions. These are all cases where a linear model may be better for your problem. But like all design decisions we have to make, the best way to settle on the right design decision is to empirically test them.

Back to Table of Contents

Read: Loss Functions

Loss functions are used to evaluate a model on the training data and tell us how bad its performance is. Loss functions determine how far the predicted labels are from the actual labels in the training data. The higher the loss, the worse it is. A loss of zero means the model makes perfect predictions.

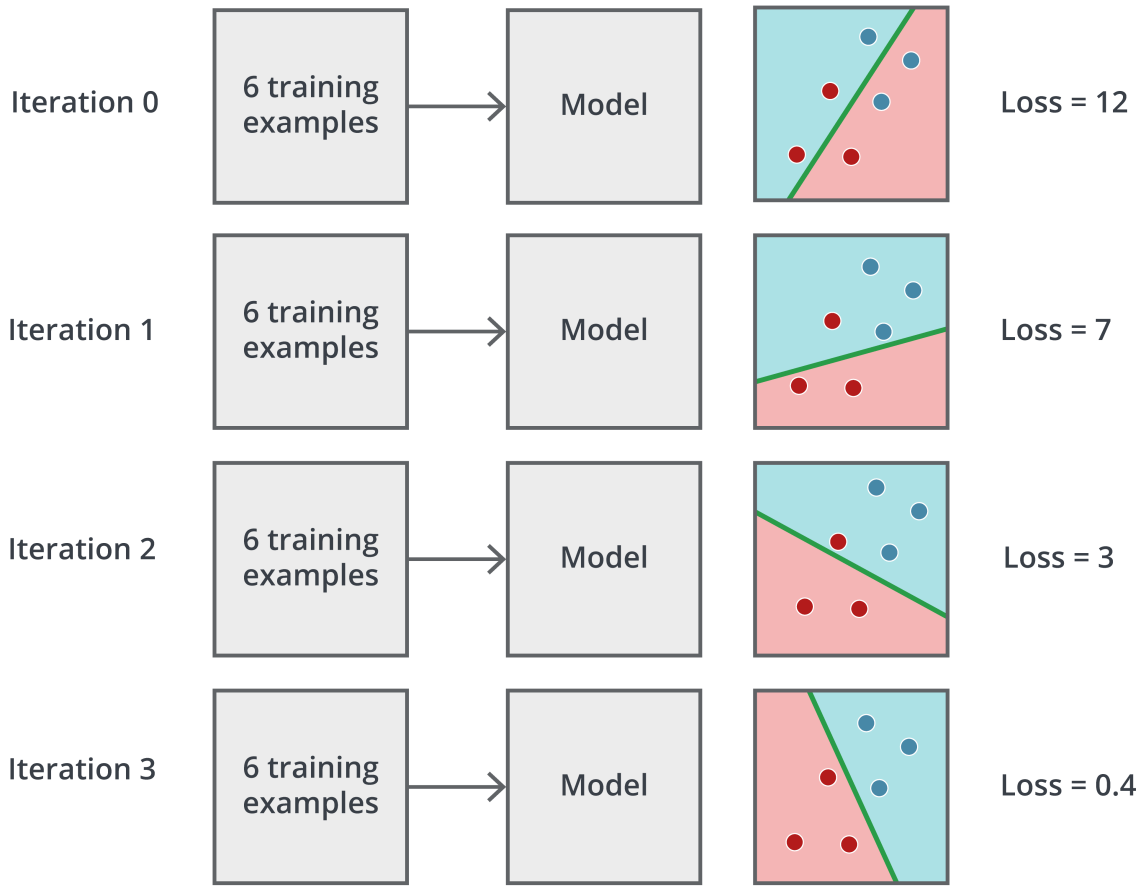
☆ Key Points

Loss functions are used to evaluate a model's performance. The higher the loss, the worse the model.

The log loss function is used to evaluate a classification model.

The mean squared error loss function is used to evaluate a regression model.

A linear model's training process involves optimization. During training, a linear model uses a loss function to determine its performance. A loss function would produce a high value if the model makes largely inaccurate predictions and a low value if the predictions are largely accurate. For the training examples that result in inaccurate prediction, the model would adjust its internal data structures in order to perform better against those examples. Let’s solidify our understanding of how loss functions are used in the training process with a demonstration.



As the image above shows, when we first initialize the model, it performs poorly as it classifies most of the examples incorrectly, hence having a high loss. Note that the loss values used here are for illustration purposes only. As we go through each iteration, the model adjusts its internal data structures and is able to make better and better predictions each iteration against the training data. The loss value gradually lowers as the model improves. Eventually, the loss value converges and stops decreasing once the model has optimized to the training data.

Examples of Loss Functions

It is common practice to divide the overall training loss by the total number of training examples N, so that the output can be the average loss per example. It is worth

noting that a function that computes this average loss across all of the individual examples is called a cost function, whereas a function that computes the loss of one example is called a loss function. However, the terms “loss function” and “cost function" are often used interchangeably. The formulas below yield the average loss across all examples in the training data.

Log Loss

Log loss, also referred to as binary cross-entropy loss, is commonly used to measure the performance of a binary classification model such as logistic regression. The equation for log loss is given below where y_i is the label of a training example and p_i is the probability prediction value for the same training example.

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

Mean Squared Error

MSE is commonly used to measure the performance of a regression model such as linear regression. For every example, you take the difference between the label and the prediction and square it. You then find the average for all examples in the training data.

The equation for MSE is given below. Just as is the case with log loss, y_i is the label of a training example and $\hat{y}_i$ is the prediction value for the same training example.

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Zero-One Loss

While loss functions are typically used in the model training phase, some are better suited for evaluation of models after training. One such loss function is the zero–one loss that is used in classification. The zero-one loss is often used to evaluate classifiers in multiclass/binary classification settings but is rarely useful to guide optimization during training.

Zero-one literally counts how many mistakes a model makes when making a prediction. For every single example that is predicted incorrectly, it suffers a loss of 1. The normalized zero–one loss returns the fraction of misclassified training examples, also referred to as the training error.

You have essentially been using zero-one loss to evaluate your classification models since zero-one loss produces a value that is equivalent to the accuracy evaluation metric used in classification.

$$L_{0/1} = \frac{1}{N} \sum_{i=1}^{N} \delta \hat{y}_i \neq y_i, \text{ where } \delta \hat{y}_i \neq y_i = \begin{cases} 1, & \text{if } \hat{y}_i \neq y_i \\ 0 & \text{otherwise} \end{cases}$$

Back to Table of Contents

Watch: Logistic Regression

In this video, Mr. D'Alessandro will demonstrate how to implement a logistic regression model in scikit-learn. He will further explain the concept of weights by explaining how they make up the underlying data representation of logistic regression as well as the effects of regularization and how it impacts the model. Mr. D'Alessandro will also describe the steps taken for a logistic regression to perform binary classification on unlabeled examples. You will see how the output of logistic regression is the probability that an example belongs to a class.

Video Transcript

Let's get tactical with logistic regression. I'll show how to train the model using scikit-learn. I'll describe the data representation of the model and then I'll spend some time discussing the coefficients, or weights, that the model returns. Let's assume we've decided that logistic regression is an appropriate candidate for our problem. Remember that logistic regression is one of the simplest supervised learning algorithms we can leverage. We'll want to use it when low model complexity is a desired feature. One area where low complexity is often desired is when making predictions on medical data, for instance. It is common for data sets in this domain to be collected from patient data, and for them to be really small and also the need for interpretability to be very high. The main point is the simplicity of the algorithm is the motivation for choosing it. Once we want to start, the first thing we'll do is prep our data. After that, training the model will require the same basic steps we've already seen in scikit-learn. I'll call out two particular elements here. First, the main hyper parameter to choose to consider with logistic regression is simply called C. There is this process under the hood called regularization. We'll learn more about how that exactly works later. The important thing to know now is that this type of parameter C controls the complexity of the model. A higher value leads to less regularization, which gives the model more complexity. In general, when you have small data and/or a lot of features, you'll likely want to use lower values of C, which increases the regularization. When building a model in practice, you'll want to test different values of C, as we usually don't know in advance which one is best. Another point I'll call out here is that for logistic regression, we usually want to use the predict_proba method for predictions. Logistic regression is explicitly designed to return probabilities, so we'll

want to take advantage of that and that is what the predict underscore problem I think does. Now that we have a model, let's take a quick peek under the hood. The model is a set of weights that belong to each individual feature, as well as a constant that we call the intercept. The table here shows an example set of weights on 11 features from some data set. We can ignore the meaning of the data for now and just focus on the model. One convention is that the weights are represented by the Greek letter Beta and the intercept by the Greek letter Alpha. This is more of a classical statistics convention. Instead, we will follow a machine learning convention and represent the weights by the letter W. When scikit-learn finishes training, the weights and intercept would be stored in the model object. This is all you need for the prediction step. The code sample here shows how the model can be used in the prediction step. When you call the model's predict_proba method, it is actually just doing these three steps. Three inputs we need are the feature values, here represented by X, the weights represented by W, and the intercept represented by Alpha. The first step is to compute the linear component of the prediction. Once we have that, we convert that linear component to a probability using the inverse logit function. Here is an illustration that I hope helps to better understand the processes behind logistic regression. Each plot here represents a hypothetical feature X and its relationship to a binary outcome Y. First, plotting a numeric feature against a binary outcome produces a funny looking plot, but we can still gain insight from it. If we look at the leftmost plots, we can see that the Y values cluster at opposite ends of the X values. This is a good indication that the particular feature X has some predictive value. The rightmost plots are a good contrast. The Y values do cluster, but the pattern is harder to decipher. The red curves are the output of the inverse logit. The linear portion of the model (or X times W plus Alpha) can take the values of negative infinity to positive infinity. When we put this into the inverse logit function, the values get squeezed to remain in the range of 0-1. This is why we can use this method for computing probabilities. Each of these red curves have somewhat of an S shape. The slope of the middle portion of the curve is determined by the feature weight. Very high weights will produce S curves that are nearly vertical. We can see examples of this in the left-most plots. The direction of the curve is driven by the sign of the weight. The middle plot here shows a moderate weight and we can see the slope is more gradual. If a feature has no predictive value, the weight will be zero or close to it. This is represented by the right-most plots. These slopes are so small that we don't even see the S curve here, but if we extended the x-axis to negative and positive

infinity, we would likely eventually see it. In real world problems, we would have many features, each with its own slope. This is naturally more difficult to visualize, but we can think of these dynamics here happening for each individual feature.

Back to Table of Contents

Read: Using Logistic Regression to Make Predictions

Logistic regression is used to solve classification problems (note that the word *regression* refers to the linear form of the model rather than the problem type; this is a common point of confusion for many). It is used to estimate the probability that a new, unlabeled example belongs to a given class. Logistic regression is best suited for binary classification. Note that it can be used for multi-class classification by making minor tweaks to the algorithm, but this will not be covered in this course.

☆ Key Points

Logistic regression is a linear model used in supervised learning for binary classification problems.

Logistic regression outputs the probability prediction of a binary outcome.

Three steps are involved when making predictions with a logistic regression mode: a linear step, an inverse logit step, and a mapping step.

Logistic regression belongs to a class called the general linear model that uses the linear combination of features and a set of weights to compute the label of an unlabeled example. When using a logistic regression model to make a class prediction, we take the feature values of an unlabeled example and pass them through our logistic regression model, which performs three steps to output a label:

1. Linear step: generating an output by taking the sum of feature values times their learned weights.
2. Inverse logit (Sigmoid): using the inverse logit function to transform the output of step 1 into a probability prediction between 0 and 1. For example, there is a .67 chance that an email is spam.
3. Mapping (Thresholds): using a probability threshold to output the class label from the probability prediction generated in step 2. For example, if the probability is above .73, the class label is "spam."

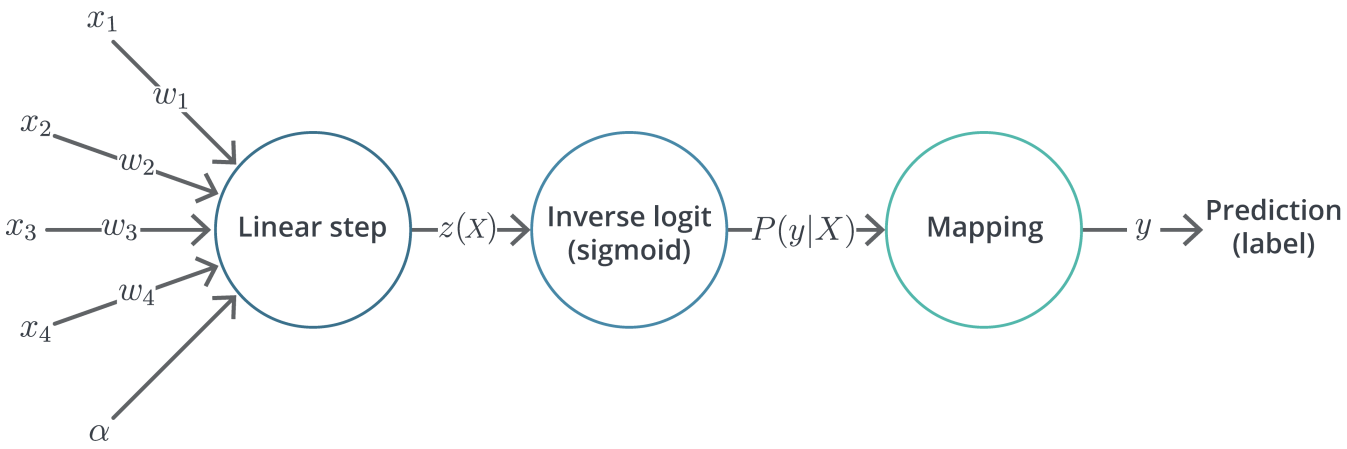


Figure 1. Anatomy of a Logistic Regression model for binary classification.

Let's go through each step in detail.

▼ Linear Step

The linear step computes a value $\boldsymbol{z(X)}$ by taking the linear sum of feature values $\boldsymbol{X}$ with model weights $\boldsymbol{W}$ and an intercept term $\boldsymbol{\alpha}$ (also known as beta, $\boldsymbol{\beta}$, in some literature). If the examples only have one feature, the linear step of the logistic regression model is basically the function of a line. If the examples have multiple features ($\boldsymbol{n}$ number of features), then the linear step becomes the function of a hyperplane.

$$\boldsymbol{z(X) = \alpha + w_1 x_1 + w_2 x_2 + \ldots w_n x_n}$$

The weights and the intercept are called "model parameters." These are different from hyperparameters. Model parameters are part of the model and are learned during the training phase from the training data. Notice that there is a weight associated with each feature. We are interested in the model parameters because those parameters tell us how an individual feature $\boldsymbol{x}$ affects the likelihood of the example belonging to a given class.

▼ Inverse Logit

Inverse Logit

The inverse logit step transforms the output of the linear step into a probability prediction $\boldsymbol{P(y|X)}$ between 0-1 based on the formula below:

$$\boldsymbol{P(y|X)} = \frac{1}{1+e^{-z(X)}}$$

A large negative output from the linear step will yield a number close to 0 and a large positive output from the linear step will yield a number close to 1. When the output of the linear step is 0, the result of the inverse logit step will be 0.5. If you plot out various values for $\boldsymbol{z(X)}$ vs $\boldsymbol{P(y|X)}$, you will end up with a plot as shown below. You are encouraged to perform this exercise yourself!

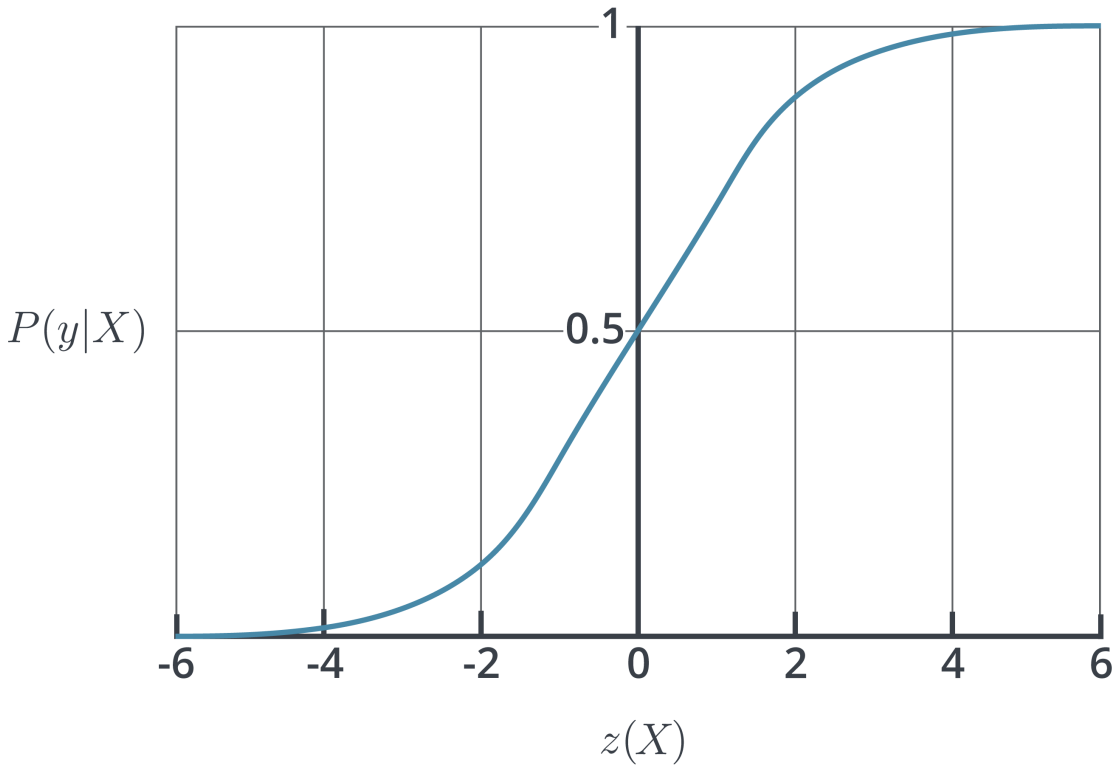


Figure 2. Inverse Logit Function graph

Let's look at an example of calculations using the inverse logit. Let's say we have three examples, each containing two features. The table below shows the feature values, their weights, their intercepts, and example calculations using inverse logit. If this were a spam email classifier, the value for $\boldsymbol{P(y|X)}$ would be

the probability that an email is a spam. If p(spam|email) is close to 1, then we are very certain that it is a spam, if it is 0.5, then we have an equal chance that it is a spam or non-spam. If it is close to 0, then we have high confidence that it is not a spam.

Table 1. Example inverse logit calculation

Example	a	w1	X1	w2	X2	z(X)	P(y|X)
1	-9	4	10	-5	0	31	1
2	3	-6	3	-4	-4	1	.73
3	10	6	0	-5	4	-10	0

▼ Mapping

Mapping

The inverse logit function returns a probability prediction that the class label is 1. The probability is a floating point value between 0 and 1. For example, the model may return a probability of .45 that a new email is a spam (class label 1).

We can map the probability values generated by a logistic regression model to binary class labels. For example, we can map a probability value to the class label 0 to represent “not spam” or the class label 1 for “ spam”.

We do this by choosing a threshold. For example, we can say that if the logistic regression model returns a value greater than 0.75, a new email is spam. Otherwise, it is not. There are different techniques you can use to determine the threshold for your problem. You will learn about these techniques later in the course. Once you’ve chosen a threshold you can then simply use a conditional statement with the threshold to map the probability to a class label.

$$y = \begin{cases} 1 & \text{if } P(y|X) > .75 \\ 0 & \text{else} \end{cases}$$

Here, the class label y is 1 (spam) if the returned probability is greater than the threshold 0.75, and 0 (not spam) otherwise.

Putting It All Together

In summary, in a logistic regression model, a weighted sum is passed through an inverse logit function that outputs a prediction probability value between 0 and 1. These probabilities can then be mapped into binary class labels using classification thresholds.

These three steps essentially make up a decision boundary for making predictions. Figure 3 below shows an example of a trained logistic regression model. The green line is the decision boundary. The dots shown represent training examples and the light blue and pink regions are the expected predictions for new, previously unseen examples (e.g., if an unlabeled example falls in the pink region, it will be predicted to be red). Notice that some of the training data points are misclassified since this training dataset is not perfectly separable by a linear line.

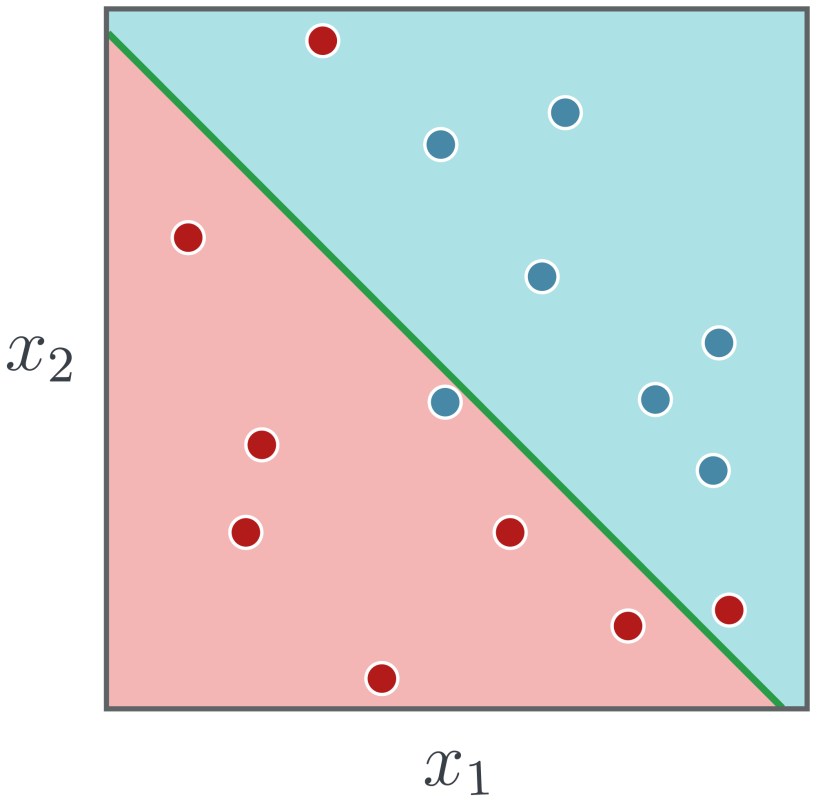


Figure 3. A trained logistic regression model.

Let's say our blue label is represented as 1 and the red label is represented as 0. Mathematically speaking, those points that are deep in the blue zone will be calculated as having a high inverse logit value, meaning its probability will be something like ~0.99, a number that is close to 1. Likewise, for a point deep in the red zone, the inverse logit value will be close to 0. For points that are close to the green line, the inverse logit value will be closer to the base rate of the label (e.g. 0.2 if 20% of the training labels are positive). This probabilistic nature of logistic regression is very powerful as it not only classifies the points (with the help of thresholding) but also gives confidence in how well a point is likely to be correctly classified.

Back to Table of Contents

Assignment: Implementing a Logistic Regression Model

In this exercise, you will implement a logistic regression model using scikit-learn. You will evaluate the model's loss and accuracy, and will learn how to use classification thresholds to map the model's probability predictions to class labels.

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. After submission, the Jupyter Notebook will always 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. Log into the course to view.

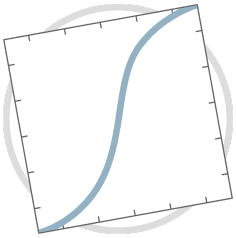
Back to Table of Contents

Module Wrap-up: Introduction to Logistic Regression

In this module, Mr. D'Alessandro described the difference between a linear and non-linear model while comparing the pros and cons of each. He also explained the idea of log loss and the use of it when evaluating the performance of a model. You discovered that logistic regression is a linear model intended for classification problems. These topics provided the foundation for you to complete the coding exercises where you implemented your own logistic regression model using scikit-learn.

Back to Table of Contents

Module Introduction: Implement the Inverse Logit and Log Loss



In this module, you will delve into the core math concepts and practical coding skills required to implement components of logistic regression from scratch. Mr. D'Alessandro will demonstrate how to perform vector and matrix operations using NumPy to achieve higher efficiency. You will put this knowledge into practice by implementing the inverse logit function to make predictions and the log loss function to evaluate those predictions.

Back to Table of Contents

Watch: Introduction to Mathematical Computation

Many math concepts are foundational to common machine learning applications. In this video, Mr. D'Alessandro gives an overview of the motivation behind learning these math concepts.

Video Transcript

In this module, we're going to take a slight detour from pure machine learning topics. We want to start building some knowledge around a few important mathematical concepts that are used in many machine learning applications. These concepts pull mainly from linear algebra, which involves operations on vectors and matrices. In this module, I'll cover a few core operations from this field, and most importantly, how to implement them in NumPy. Most of the Python methods we'll cover should be familiar to you by now. I am re-presenting them now with a more formal mathematical lens. These operations are very foundational for most of your machine learning work. A lot of data manipulation in pandas relies on some of these concepts, too. But also, when we go deeper into how to actually optimize a loss function, we'll see that a few basic operations on vectors and matrices form the core of most common approaches.

Back to Table of Contents

Watch: Vector Operations with NumPy

In this video, Mr. D'Alessandro introduces the idea of vectors and arrays in the context of machine learning and Python. There are three crucial mathematical operations: elementwise arithmetic, summation, and dot product. These are all very useful when translating mathematical notation into code, and can be used when implementing a logistic regression. Mr. D'Alessandro then illustrates the power of NumPy by demonstrating how you can use it to dramatically speed up the summation operations from the traditional loop approach.

Video Transcript

In this video, we're going to revisit NumPy arrays, but focus more on specific mathematical operations that we apply to these arrays. These operations are core to many machine learning methods under the hood. The three operations we'll cover are element-wise arithmetic, summation, and dot product. In the context of logistic regression, these three operations alone are sufficient for building a predict function for computing the log loss. In Python, we use the array data type from NumPy. In traditional mathematics, though, we call these arrays vectors. It is important to realize the equivalency between a vector and an array. When you read textbooks or papers on machine learning, usually you'll see logistic regression represented in mathematical notation. Understanding how to translate this mathematical notation into code is very useful for truly understanding the topic. The focus of this module, again, is to translate the mathematics into code so that we can be comfortable implementing it. Let's start with a vector. A vector is a set of numbers that usually represent coordinates in a geometric space. The following graph represents a vector from a geometric point of view. This vector has three dimensions and is represented by three points. Of course, a vector can have any arbitrary dimension, but visualizing them beyond three isn't very feasible. In the context of machine learning, the vectors we usually deal with are from data. Given a data matrix, we can think of a single data column or row as a vector, as we have illustrated in this hypothetical data set. Since data is usually represented as a Python object, this is where vectors become arrays. Before I show how these arrays are used in the core vector operations I mentioned in the opening here, let me first share the context in which we're going to use them. The following equation is what we call the inverse logit. The inverse logit is the name of the equation we will use to get

probability scores from a trained logistic regression. Here, X is a vector of feature values, W is the weight we learned from training the logistic regression, and Alpha is the intercept. In this representation, we can think of this as the prediction for a single example. And remember, there is one weight associated with each feature used to train the model. The one element of this equation that is an operation unique to vectors is the part in the exponent where X and W are used. This is an operation called the dot product. I break this operation down in this particular image. Here, both X and W are vectors of size N. In Python, again, these would be NumPy arrays. The dot product is a special operation where we take the element-wise product of each vector and then sum them together. It is typically represented with what looks like a period or a dot floating between the two vectors as shown here. When looking at the formula for the dot product, I also want to emphasize the use of the Greek letter Sigma, which represents summation. This is very commonly used, so it is recommended to understand it. We can think of the sigma command as a for loop where we are incrementing the sum as we iterate over the particular sequence. In NumPy, there are a few ways we can implement this dot product. The following code block shows three approaches we can take. These all produce equivalent results, but we can think of each one is representing different mathematical notation. The first approach is the most verbose way to do it. Here, I am implementing the sum as a loop and adding the product of array elements to a running sum. This approach works, but a general rule with NumPy is to avoid explicitly writing loops as much as possible. The second approach is more compact for sure. Here, we are taking the element-wise product, taking advantage of NumPy's vectorization functionality, and then summing using the sum method. This last approach is even more compact. NumPy has a specific method that lets us take the dot product of two vectors. Even though all of these produce the same answer, they are not equivalent from an efficiency point of view. The following code block shows a time test comparing each of the three methods. The biggest standout is the first one that uses a loop. This is nearly 100 times slower than the other two approaches. The bottom two use NumPy-specific methods and take advantage of the speed which is built into NumPy's code. We still see a difference between them, though. Using the dot product specific method speeds up computation about five times compared to the one above it. Not only is this approach more concise from a code perspective, it is also much faster than the other options. When you see machine learning math that requires the use of a dot product, you now know how simple this can be. The trick is to first remember that the dot product is

different than element-wise multiplication. The output of element-wise multiplication is an array of the same length as the input arrays. The output of the dot product is a scalar or a single number. Additionally, the dot product only works when the two vectors are the same size. Bringing this back to logistic regression, when we make a prediction, we first take the dot product of an examples' features and then the feature weights. This returns a scalar value that then gets added to the intercept and input into the exponent to ultimately get a probability value.

Back to Table of Contents

Watch: Matrix Operations

Matrices play a critical role in machine learning. In this video, Mr. D'Alessandro lists the most commonly used matrix operations in machine learning such as addition, elementwise multiplication, matrix multiplication, and inverse. He will explain the importance of matrix dimension and how NumPy's reshape function can be used to ensure matrices can be multiplied together without error. Watch as Mr. D'Alessandro demonstrates with examples how each of these operations are implemented in NumPy.

Video Transcript

In this video, we're going to cover common operations that we do on matrices. A matrix is just a set of vectors, as the operations we do on matrices are very similar to vector operations. But there are a few nuance differences which motivates having a dedicated lecture on this subject. We want to study matrix operations for the same reason we do this for vectors. Many of the equations that define machine learning methods use matrices in their notation. As an example, let's revisit the inverse logit function used in logistic regression. We can treat X here as either a vector or a matrix. When we are making predictions for a single example, X would be a vector of features, but when we implement this function in Python, we can treat X as a matrix, which corresponds to making predictions for all examples in the data at the same time. The matrix operations we'll cover are addition, element-wise multiplication, matrix multiplication, and the inverse. First, let's start from scratch and introduce what a matrix is from a NumPy perspective. A matrix is just a multidimensional array as shown here. This code sample shows a 4x4 NumPy matrix. We can also view this as an array of arrays. Once we create the matrix, let's look at how we might select rows, columns, or specific elements. The code block here shows examples of each. Selecting rows, columns, or elements is a simple extension of how we normally select elements of a list or an array. Let's get directly into the math now. Similar to vectors, we can do element-wise addition and multiplication on matrices. The implementation and output are shown in the following code block. The rules here are exactly the same as for vectors. First, we need to make sure the dimensions are the same for each matrix. Next, let's say we want to add a vector to a matrix. This is a little trickier because we have the option to add the vector to the matrix rows or the columns. The following

code block shows how we might do either. First, on the left side, I show that to get NumPy to treat an array like a column vector, we have to reshape it. This is shown in the bottom portion where the array is reshaped to explicitly have four rows and one column. Next, in the right-hand side block, adding vectors to the matrix is done using the same standard addition operator. Multiplication works the same way. Notice in the right-hand side that when we add a vector to a matrix this way, the addition is happening to each row or column. Now let's look at matrix multiplication. This is separate from the element-wise multiplication we just covered. This is actually an extension of the dot product of vectors. The following math shows the standard notation for matrix multiplication and the formula for computing the values of the new matrix. First, let's focus on the top portion that defines the matrices. When we multiply matrices, we do not need the matrices to be the same size. What matters is that the inner dimensions of the matrices are the same. If A is multiplied on the left, its inner dimension will be its column count. This is different from element-wise addition or multiplication, where we want the two matrices to share the exact same dimensions. For the right-hand matrix B, the inner dimension would be its row count. Now, to get the new matrix C, we iterate through each row vector of A and column vector of B and take the dot product. This is shown in dot product notation by Equation 2 here. Equation 3 just expands this further as a summation. The following picture shows two matrices, A and B, whose inner dimensions line up. In Matrix A, I've highlighted the first row; in B, the first column. The code on the bottom shows the dot product of these two highlighted vectors. This would become the element in the 0-0 position of the resulting matrix. Once we've looped over all rows and columns and applied the dot product, we'd get the final matrix. Luckily, this can all be done in NumPy with a simple one-line command. Here, we are just using the dot method, which is the same method we used for arrays. One last point before we move on: the matrix B can also be a vector, so long as its length matches the column dimension of matrix A. After all, a vector is just a matrix where one of the dimensions is one. In our inverse logit function where we compute X times the W portion, X is the data matrix whose column dimension is the number of features, and W is the vector of weights that lines up with each feature. Our next and last topic is the inverse of a matrix. When we cover the mathematical techniques that enable us to train a logistic regression, we'll see that one step in the process is to take the inverse of a matrix. We can think of a matrix inverse as similar to the idea of a reciprocal of a number where the reciprocal is just one divided by the number. The inverse of a matrix is a matrix that when

multiplied by the original matrix, it produces something called the identity matrix. The identity matrix is a special matrix whose diagonal elements are one, and all other elements are zero. The following picture shows the simple formula for the inverse and an example of a three by three identity matrix. Computing a matrices inverse is actually fairly complicated, so I'm not going to show any formulas for how this is done. Instead, I'm going to show how it's done with NumPy. The following code block defines a random matrix, computes its inverse, and then verifies that the two multiplied together produces the identity matrix. This is a standard operation, so unsurprisingly, there is a one-lane approach to doing it. Again, when we look more deeply into how logistic regression models are fit, we'll see the use of a matrix inversion. Further, we'll apply many of the vector and matrix operations we've just covered in the process of training a logistic regression model from scratch.

Back to Table of Contents

Watch: Logistic Regression: Predicting from Scratch

In this video, Mr. D'Alessandro demonstrates how to implement the prediction step of logistic regression from scratch. The first part is training a standard logistic regression model in scikit-learn. Once the model is trained, you will use the weights from the model and NumPy to compute the prediction of an unlabeled example. The goal here is to better understand how the prediction method works under the hood and how NumPy can be used to implement matrix and array operations in an efficient manner.

Video Transcript

In this lecture, we're going to take what we've learned about vector and matrix operations in Python and implement the inverse logit, which is how we get probability estimates from a logistic regression model. We are going to use scikit-learn to train a logistic regression model and then implement our own version of scikit-learn's predict_proba function to make predictions. In most applications of logistic regression, you will use scikit-learn's built-in methods for computing these values. Building from scratch is a great way, though, to really understand how these methods work. In order to apply and test our functions, we'll need real data and a model, so let's start by importing a data set and training a logistic regression. Our data looks like this and has 11 features. We are predicting customer churn, which is represented in the last column. Next, we'll follow the standard steps for training a logistic regression using scikit-learn. In this case, our goal isn't to fine tune the hyper-parameter, so we'll just use a simple implementation. The following code block runs our imports, splits the data, and trains the model. Now we have a data and a model. Let's start with implementing the inverse logit from scratch to make the predictions. This will be equivalent again to using scikit-learn's built-in predict_proba method. Here, again, is the formula which guides the code we need to write. Walking through this, the first step should be to compute the X times W portion. We're going to treat X as a matrix, so that means we'll have to implement matrix multiplication. Then we'll add Alpha the intercept, put the output into the exponent, and take the inverse of all of this. Starting with the X times W operation, let's check the dimensions of our data matrix and weight vector to make sure the dot product is allowable. In the top cell, we first convert our data frame to a matrix using the too NumPy method. Then we pull the weights from the logistic regression model as shown. I have printed the shapes of

both of these objects. We can see that they don't align here, but there is an easy fix to that. When you run into this problem, you can take the transpose of the matrix or the vector. The transpose just reverses the shape, so if the vector is 1 by 11, the transpose switches the shape to 11 by 1. We take the transpose in the second cell by adding the dot T to the vector. The code shows that this works. Now let's check the shape of the resulting matrix. We show that the result conforms to our expectations, it has the same row count as the input matrix in only one column. Now when we look at the output, we can see that this doesn't look like a traditional array. This is actually a two-dimensional column array where the column dimension is one. My opinion is we should simplify this structure and use just a one-dimensional array. We can do this using the NumPy ravel method. The following code block shows how we use that and what the output looks like. Notice that the shape has only one dimension and the output looks like a standard array. This is the most complicated part of computing the inverse logit. Now that we know how it works, let's encapsulate it all in a function that computes the full inverse logit. To start, let's review the inputs. We need the data matrix X which has the shape and examples and K features. Then the model is represented by a K by one weight factor W, which again has one weight per feature and a scalar and intercept called Alpha. Both the weight vector and intercept can be taken from the scikit-learn logistic regression model. Our first step is to compute the X times W using the dot product. This is the matrix operation we just reviewed; the output again is an array of length N; the remaining steps are performed in one line using the vectorization properties of NumPy. First we add the scalar Alpha to X times W multiplied by negative one, and then take the exponent of that resulting array. Remember when we put an array to a function, we apply the function to every element of the array and get an array in return. Then we add one to this and take the inverse of that resulting array. I've added the inverse logit formula to the image. Notice how similar the code is to the actual formula. One of my favorite aspects of NumPy is how vectorization enables us to implement formulas almost exactly as they're represented as math, our function is now complete. Let's call it with some actual data and see the output. The first code block is our implementation of the inverse logit, where the weights in intercept are pulled from the scikit-learn logistic regression model. Now, looking at the results alone, it's hard to tell if we are correct. I've also computed probabilities using scikit-learn's implementation of the same function. We can visually inspect that the first five elements returned are the same for each approach. This is a good sign. To be thorough, though, let's write a test to make sure

all values are the same. The last block here is such a test that checks whether any elements of the two arrays are different. We pass the test and learn there are no elements that are different, which we establish here by summing up the element-wise comparisons. Now that we know that our math was correctly implemented, let's check whether our version was efficient. To test this, I'm going to use Jupyter notebook's magic function called timeit. This test is shown in the following code block. I have highlighted the key output that we care about from this test, which is the average time to compute the result. We can see here that our implementation was actually faster than scikit-learn's implementation. This is great, but it doesn't mean we should always build our own functions from scratch. Scikit-learn's approach is on the same order of magnitude and is a little slower because scikit-learn is also doing various checks in exception handling to make sure the inputs conform to expectations. This extra bit of overhead is usually worth it to have a code that works in all situations.

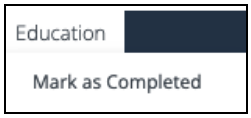
Back to Table of Contents

Assignment: Implementing the Prediction Step and Evaluating the Model

In this exercise, you will practice working with NumPy to perform matrix operations and then use that knowledge to implement a logistic regression prediction function and a log loss function. You will use your log loss function to evaluate the predictions and compare your results with the results of scikit-learn's log loss implementation.

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. After submission, the Jupyter Notebook will always 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. Log into the course to view.

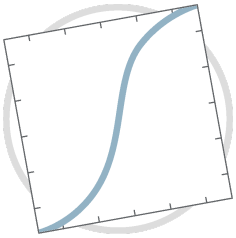
Back to Table of Contents

Module Wrap-up: Implement the Inverse Logit and Log Loss

Now that you've completed this module, you have the foundation of various math concepts and coding skills required to tackle common machine learning problems. You've seen Mr. D'Alessandro demonstrate how to perform vector and matrix operations. You also gained an understanding of how they are used to carry out computations more efficiently. This led you to the exercise where you practiced implementing the logistic regression prediction step and the log loss function using NumPy, and compared your results with scikit-learn's log loss implementation.

Back to Table of Contents

Module Introduction: Train a Logistic Regression Model



In this module, Mr. D'Alessandro will demonstrate how the training of a logistic regression model works. In particular, you will discover how it employs gradient descent, which is an optimization algorithm used to find the minimum of a function. First, Mr. D'Alessandro will explain how gradient descent is used to find a minimum of a function, and then you will see how it can be used to iteratively train and optimize a logistic regression model by minimizing a loss function. Mr. D'Alessandro will also list some of the most common hyperparameters for logistic regression and the effects they have on the training of a logistic regression model. Additionally, you will see how to apply regularization to the model training process, preparing you for the assignment where you will implement logistic regression models using regularization.

Back to Table of Contents

Watch: Finding the Minimum of a Function Using Gradient Descent

In these videos, Mr. D'Alessandro introduces the concept of gradient descent, which is an iterative optimization method used to train a logistic regression model. In the training process, the gradient descent algorithm finds the model weights that minimize the log loss function, resulting in an optimal model with low training loss. While gradient descent is used in training a logistic regression model, the gradient descent algorithm can also be used to find the minimum of any function. Before diving into the training process of a logistic regression model, these videos will show you how gradient descent works to find the minimum of a function, and how to implement this in code.

Part 1 In this video, Mr. D'Alessandro will list the essential components that make up a gradient descent algorithm, such as loss function, finding the gradient of a loss function, and stopping criteria.

Video Transcript

In this lecture, we're going to focus on what goes on under the hood when we train a logistic regression. We're going to cover a special algorithm used in the training process called gradient descent. This particular algorithm is different than other algorithms we've covered in this course. Gradient descent is a numeric optimization algorithm that is used to find the minimum or maximum of an arbitrary mathematical function. Gradient descent is used in machine learning to find optimal weights of logistic regression. It does this by minimizing a loss function. It finds the weights that result in the lowest training loss. The goal of this module is to show how we find the optimal weights of logistic regression using the gradient descent algorithm. The function it will minimize is the log loss. But before covering gradient descent on log loss, I'm going to cover the gradient descent algorithm on a simpler function first. As I mentioned, gradient descent is an algorithm we use to find the minimum or maximum of a function. The following graph shows an example function we might want to minimize. Here, I represent the function three different ways. At the top, we have the underlying equation. Below that, we have a Python implementation of it, and at the bottom, a plot of what the function looks like. The goal with the gradient descent algorithm is to find the value of X that results in the lowest value of F of X. Getting to the minimum is pretty straightforward, fortunately. I'll first walk through a visual example and then I'll show the underlying math behind it. The first step is to just make a guess on where the minimum lies. A common starting guess for most problems is X equals zero so let's start there. Our next step is to then compute what is called the gradient of the function at the value of our initial guess. If you're familiar with calculus, the gradient is the first derivative of the function you are trying to minimize. In its most simplest mathematical terms, the gradient is the slope at a given point as illustrated here. The gradient itself is also a function, which means we can compute a value for it at our initial guess. The most important aspect we care about is the direction, or the sign of the gradient. In this example, the gradient arrow is pointing upwards, which means it has a positive value when we evaluate it at our initial guess. The gradient guides us by showing us whether the function is increasing or decreasing when we

move in a specific direction. Starting at our initial guess, if we want to increase the function, we would move in the direction of the gradient because it is positive. But here, we want to minimize the function, which means we need to move in the opposite direction of the gradient. The next consideration is how far should we move in the opposite direction of the gradient. There are many different ways to answer this question. As a matter of fact, most variations of gradient descent in practice differ in how they answer this particular question. I'll address this point more later, but in the meantime, here is an illustration of gradient descent being applied to this example function. The gradient line at the minimum point is a horizontal line, which has a slope of zero. When the slope is zero, there is no obvious direction to move. When this condition is met, that means we've actually found our minimum. Now, in practice, you likely won't get to the point where the gradient is exactly zero. To compensate for that, we can define a rule such that if the gradient is close enough to zero, we can stop the procedure. It will be up to you in the implementation to define that stopping criteria. If this example were logistic regression, we can think of X as a feature weight and the function is log loss. The gradient descent procedure would then help us find the feature weight that gives us the lowest value of log loss. This example was for a single variable, but in most cases of logistic regression, we'll be running gradient descent for multiple variables at the same time.

Part 2

This video demonstrates how to implement gradient descent in code to find the minimum of a function. Mr. D'Alessandro walks through the algorithm while explaining each part, including the initial guess, selection of a learning rate, and optimization of the loss function using the gradient. Note that the learning rate is also called the step size. In the video, the learning rate is represented by the variable `stepsize`.

Video Transcript

In this video, we'll show how to implement gradient descent and find the minimum value of a function. Remember, we're building up our knowledge to be able to apply gradient descent to train a logistic regression from scratch. The general gradient descent algorithm is agnostic to the underlying function we are trying to minimize. For logistic regression, this function would be log loss, and we apply gradient descent to find the model weights and intercept that minimize log loss. But before digging into log loss, we'll cover the general gradient descent algorithm. First, I'll cover the mathematical form here of gradient descent, and specifically what we call the update step. Walking through this equation, let's assume we have a function F that takes an input W. Our goal is to search over different values of W to find the one that minimizes the function. Gradient descent is an iterative process. We'll start with an initial guess, compute the gradient at that guess, and then update our guess by moving some amount in the opposite direction of the gradient. The amount we move is governed by what we call the step size, or learning rate, which is a multiplier we apply to the gradient. This learning rate is a value that can be constant or variable, and choosing it is usually up to the model developer. We then repeat this process until we observe that our value of W isn't changing much, which is equivalent to the gradient getting closer to zero. Here is everything I just represented as code. I have labeled different lines of code with numbers so I can walk us through it. We start with a function signature. We pass in our initial guess because we need to see the process with the starting point. You can use any value. For logistic regression, we often use zero as our initial guess. Let me also add that in this implementation, we are optimizing a function with only one input W. Later, we will adapt this to multiple variables. Next, we input the learning rate. The learning rate is the topic of a lot of research and will be the topic of our next video. Intuitively, it needs to be not too small and not too large. For now, we'll use the value 0.1. Then we have the gradient. Of all the inputs, this one is an actual function. We call this function using the current value of W. Step 4 is what we've already seen. This is the main step where the update is taking place. Last, for steps 5 and 6, we check whether the update was smaller than our chosen tolerance. If that condition is met, we end the process; otherwise, we set the prior value of W to the current value and we repeat the loop. This gradient descent implementation is complete. We can generalize to any function that takes in a single value input. To use this with logistic regressions log loss, we'll need to modify the function a bit to operate on a vector of weights, and we'll apply a special technique to getting learning rates.

Back to Table of Contents

Watch: How to Choose a Learning Rate

In this video, Mr. D'Alessandro will discuss the effect the learning rate has on gradient descent convergence. He will differentiate the trade-offs between having a large learning rate versus a small learning rate. Mr. D'Alessandro will also differentiate the pros and cons of using the Hessian matrix, a way to dynamically assign the learning rate during the training process.

Video Transcript

One of the more nuanced aspects of the gradient descent algorithm is choosing an appropriate learning rate. The rule of thumb is that if you're learning rate is either too big or too small. The iterative gradient descent algorithm will converge. The following chart shows an illustration of a learning rate for a single gradient descent update step. The yellow, red and green markers show the results of updating the weight with different learning rates. The yellow marker is what a two small learning rate looks like, and the red is one that is too large. The green marker is a learning rate and that is just right. The learning rate impacts how long it takes for the algorithm to converge. In this video, I'll provide some examples of how the learning rate impacts conversion rates and introduce one technique for gaining step sizes. One of the more nuanced aspects of the gradient descent algorithm is choosing an appropriate learning rate. The rule of thumb is that if your learning rate is either too big or too small. The iterative gradient descent algorithm won't converge. The following chart shows an illustration of the learning rate for a single gradient descent update step. The yellow, red and green markers show the results of updating the weight with different learning rates. The yellow marker is a two small learning rate looks like, and the red is one that is too large. The green marker is a learning rate and that is just right. The learning rate impacts how long it takes for the algorithm to converge. In this video, I'll provide some examples of how the learning rate impacts convergence rates and introduce one technique for getting learning rates that is guaranteed to work. First, let me start with an illustration. The following plot shows four different calls of the gradient descent algorithm on the same function. The only difference between each series here is the learning rate that is used. The x axis is the iteration number of the gradient descent process, and the y axis is the is the value of the way at each iteration. We can see that each series starts at zero, but there are big differences in the outcome. Let's start with

a blue line that has a learning rate of 0.001. This learning rate is too small. The series is headed in the right direction, but after 50 iterations, it never gets close to the optimal value. At the other extreme, we have the red line, which has a learning rate of 0.4. This is characteristic of a two large learning rate. The gradient descent method is moving the weight in the right direction at each step, but it never gets close enough to converge. The orange and green lines show the right behavior though. Each one converges to the minimum. One just happens to do it faster. The biggest challenge is how do we know what is the right learning rate? The red line is learning rate is the same order of magnitude as the green line. The effect is dramatically different. One of the most classic approaches to solving this problem is to use properties of the underlying function we are trying to minimize. The following image shows the gradient descent algorithm again, but I've replaced the generic learning rate with a special function. Instead of choosing a constant learning rate like my earlier illustration showed, we use a function to determine learning rates at each point in the process. The learning rate here is determined by computing the second derivative of the function we are trying to optimize. This is also the first derivative of the gradient. I use the notation H of W because when W is a vector, this second derivative is called the Hessian matrix. These concepts of first and second derivatives come from calculus. The first derivative is the slope of the function at a given point. The second derivative, helps understand the curvature of a function. A large second derivative means the function gets steep very quickly. When this is true, we would usually want a smaller learning rate. Intuitively, this is why we use the inverse of the second derivative to dynamically set learning rates. The following Python function shows the full gradient descent algorithm, but updated to use the Hessian and function to determine these learning rates. This is virtually the same set of steps that was presented in a prior video. The only difference is we're computing the learning rate dynamically at each step, similar to how we would compute the gradient at each step. This is done in the highlighted portion. Similar to the gradient the Hessian is a function that we pass in as an argument. In the next video, I'm going to show how to build with a gradient in Hessian function so that we can apply this gradient descent algorithm to logistic regression. The reason we use the Hessian function here is that it takes a lot of the guesswork out of choosing a learning rate. This method mathematically yields the fastest and most accurate conversion speeds. This is ideal to use for problems like logistic regression. Remember that if the learning rate is too small, we never converge. If too large, we may bounce around the minimum or even diverge towards infinity.

The convergent speed is also sensitive to the initial starting point. With a constant learning rate the farther away you start from the optimal point, the longer it takes to converge. The following plot shows the convergent speed for both the Hessian and constant learning rate methods. We define convergent speed here as the number of iterations required to reach our desired tolerance. The x axis shows the initial starting point I used on different runs. There is a clear difference. We can see the hashing method is about four times faster and we can see there is also less dependency on the starting point. When feasible, this is the best approach for computing learning rates because of these two reasons. The main reason we wouldn't use the Hessian method, is when you have a problem, machine learning problem that has many ways to optimize. In this particular case, the Hessian becomes very expensive to compute. This is typical with neural networks and logistic regression with millions of features. For smaller scale problems though, this is typically the technique that most machine learning training packages are using under the hood.

Back to Table of Contents

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. After submission, the Jupyter Notebook will always 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. Log into the course to view.

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 then adjusts the model parameters accordingly to reduce loss. It continues this process until an optimal model is produced. A common optimization algorithm used in logistic regression training is called gradient descent. Figure 1 below demonstrates this iterative training process.

☆ 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

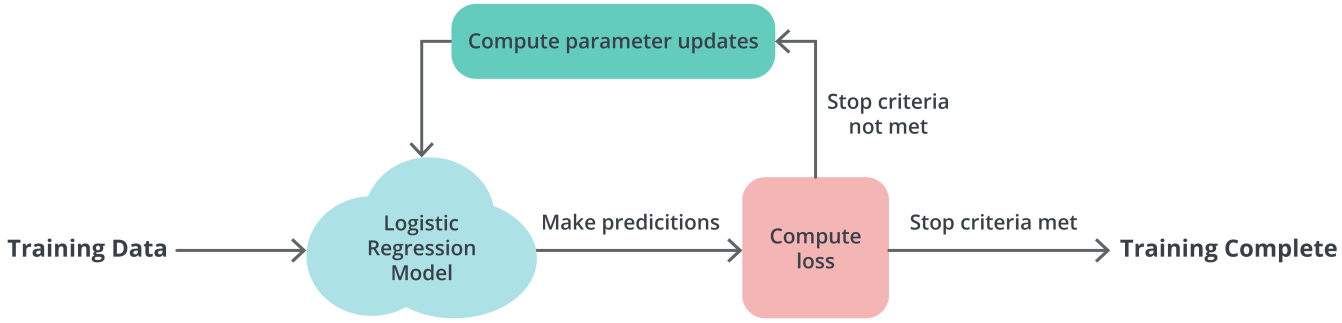


Figure 1. Iterative Training Process

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.

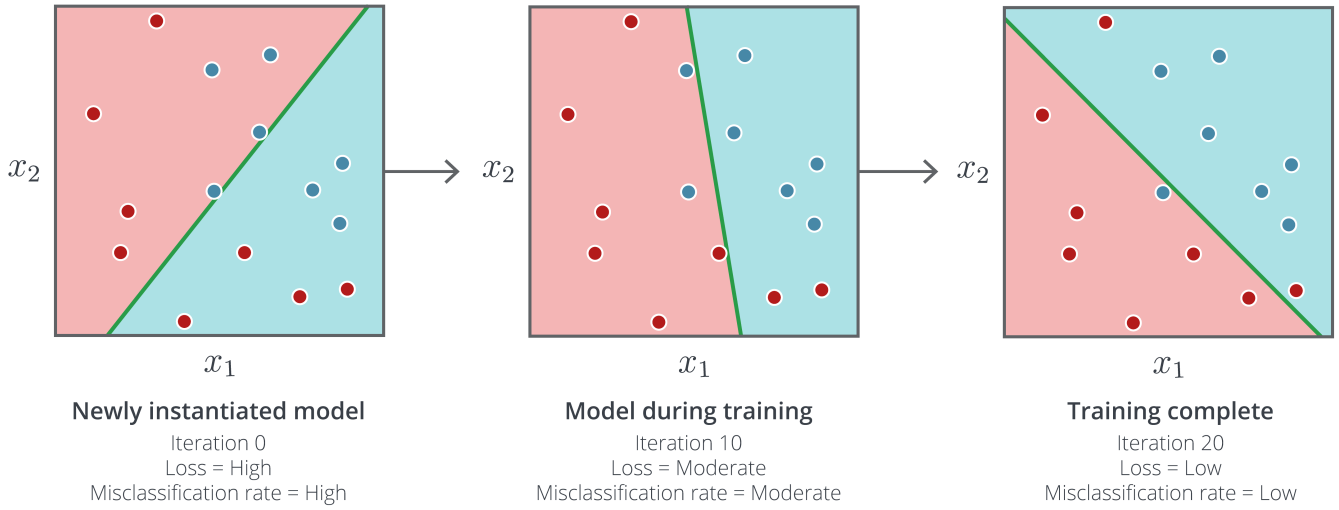


Figure 2. Training of a Logistic Regression Model.

▼ 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 points, its loss will be high, and thus the weights and intercept will need to be tweaked for optimization. When a model classifies most points 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, also known as cross-entropy. The log loss function for a single example i is provided below where y_i is the ground truth or the known label of the training example, X_i is the feature value(s) and p is the probability $P(y_i|X_i)$.

$$L(X_i, y_i) = -y_i * log(p_i) - (1 - y_i) * log(1 - p_i)$$

If an example is deeply misclassified, the 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:

$$L\left(X_i, y_i\right) = -1\log(0.96) - 1 - 1\log(1 - 0.96) = 0.0177$$

It returns a value of 0.0177. 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:

$$L\left(X_i, y_i\right) = -1\log(0.014) - 1 - 1\log(1 - 0.014) = 1.82$$

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 dataset. 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. This average loss at the batch level is sometimes referred to as 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.

$$L(\overline{X}, \overline{y}) = -\frac{1}{n}\sum_{i=1}^{n}(y_i * log(p_i) + (1 - y_i) * log(1 - p_i))$$

▼ 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 gradient descent to update the weights and the intercept on poorly performing models. As stated, gradient descent is an optimization algorithm. 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 with respect to the weights $\overline{W}$ and the intercept α individually. This partial derivative tells us how much and towards what direction (positive or negative) to update the weights and intercept. We then update the model parameters accordingly. The learning rate γ determines the stepsize, 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. 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 weights and the intercept at a given update step.

$$\overline{W}_{t+1} = \overline{W}_t - \gamma \frac{\partial L}{\partial W_t}$$

$$\alpha_{t+1} = \alpha_t - \gamma \frac{\partial L}{\partial \alpha_t}$$

Figure 3 below shows how gradient descent works to adjust a 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 will be updated in a similar manner. We are essentially plotting the log loss function; the curve represents all possible values of w for that one weight. The gradient of the loss is equal to the partial derivative (slope) of the curve, and it tells you in which direction and how much to move towards the bottom of the curve. The gradient descent algorithm takes a step in that direction in order to reduce loss. The log loss function is convex (i.e., only one minimum). Therefore, the goal is to iteratively update the value of W to reduce loss until we reach some global 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 dataset, and we have therefore identified the value of W that will result in the lowest training loss.

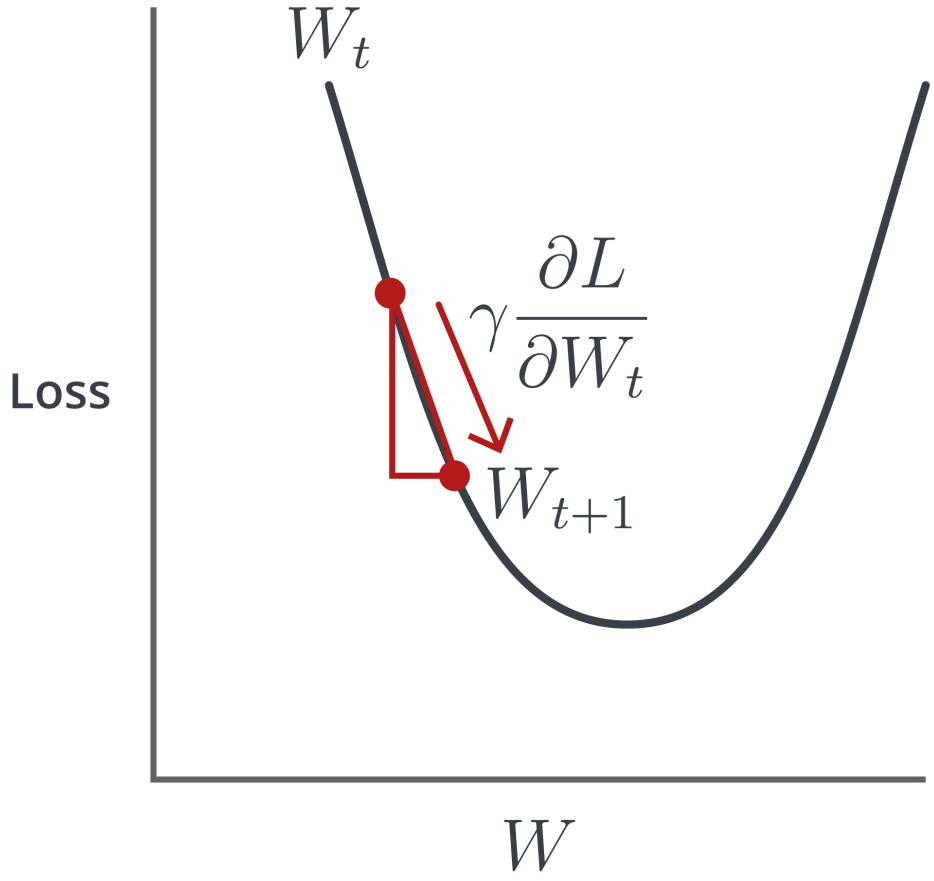


Figure 3. Gradient descent of W.

Summary

To sum things up, 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 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 then as follows:

1. Initialize a random number $\boldsymbol{\alpha}$ and set of random numbers $\overline{\boldsymbol{W}}$ equaling the number of features $\boldsymbol{X}$.
2. Choose a learning rate γ.
3. For a subset of training examples, calculate the loss $\boldsymbol{L(X, y)}$
4. Update $\overline{\boldsymbol{W}}$ and $\boldsymbol{\alpha}$ using gradient descent:
   $\overline{\boldsymbol{W}}_{t+1} = \overline{\boldsymbol{W}}_t - \gamma \frac{\partial L}{\partial \overline{W}_t}$
   $\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \gamma \frac{\partial L}{\partial \alpha_t}$
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 tree, 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.

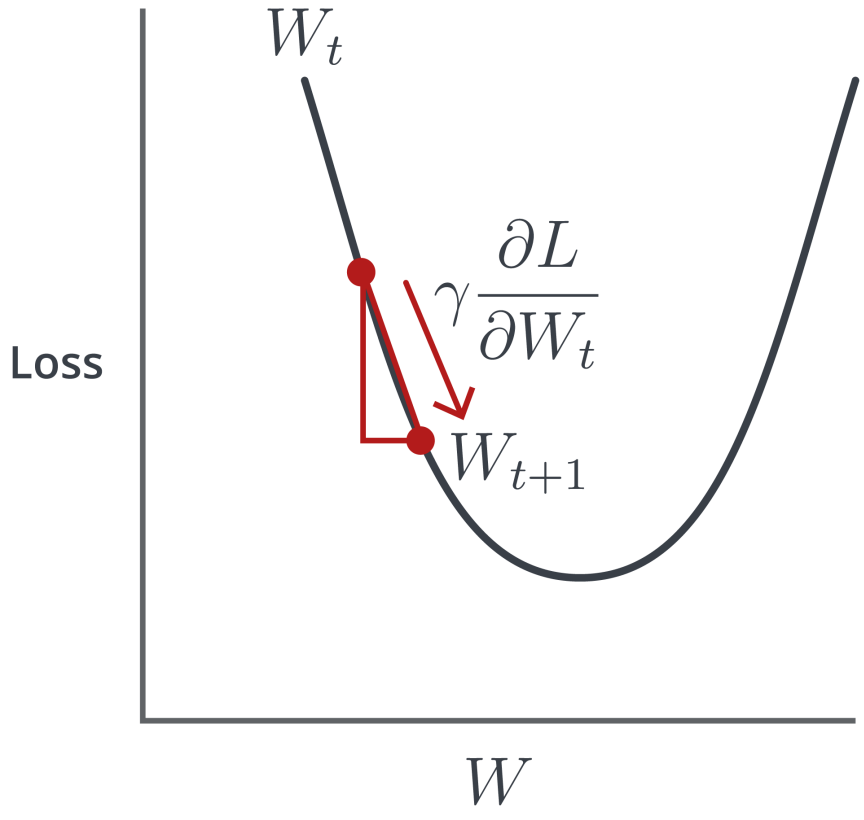


Figure 1. Gradient descent example

A high learning rate updates the weights and intercept in big increments. A low learning rate updates the weights and intercept in small increments. Let's explore different examples of learning rates. Recalling in gradient descent, our goal is to reduce the loss function of our model. With that in mind, let us explore the effect 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.

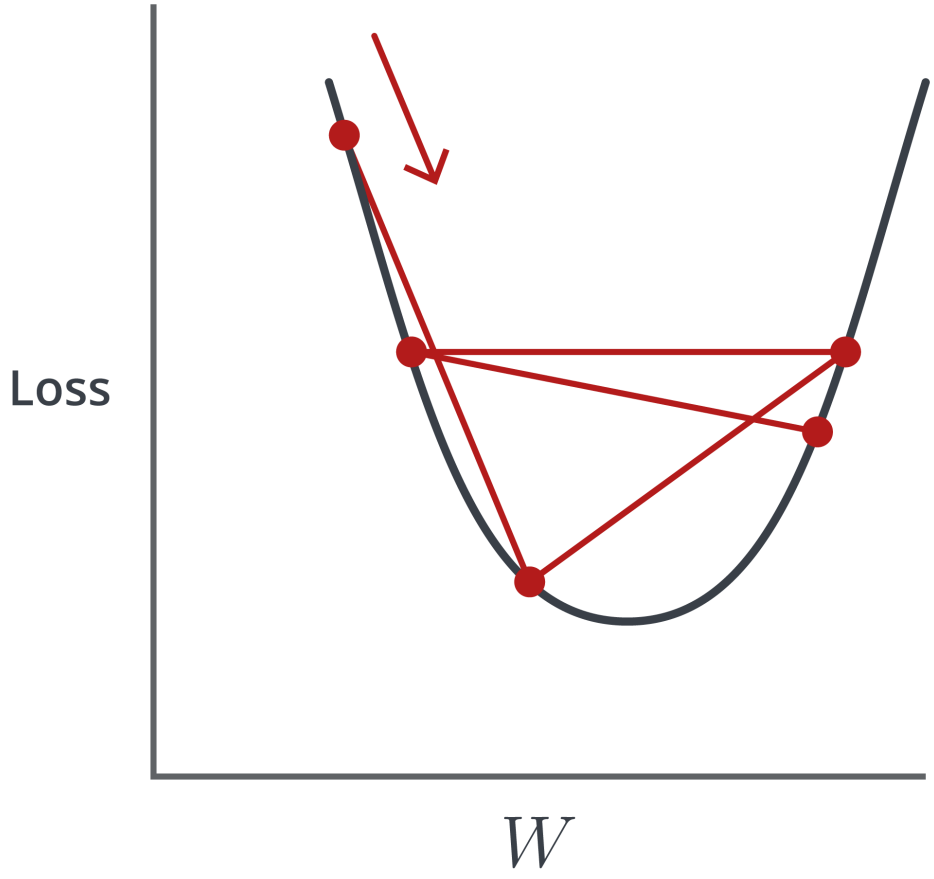


Figure 2. Gradient descent with a high learning rate

▼ 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.

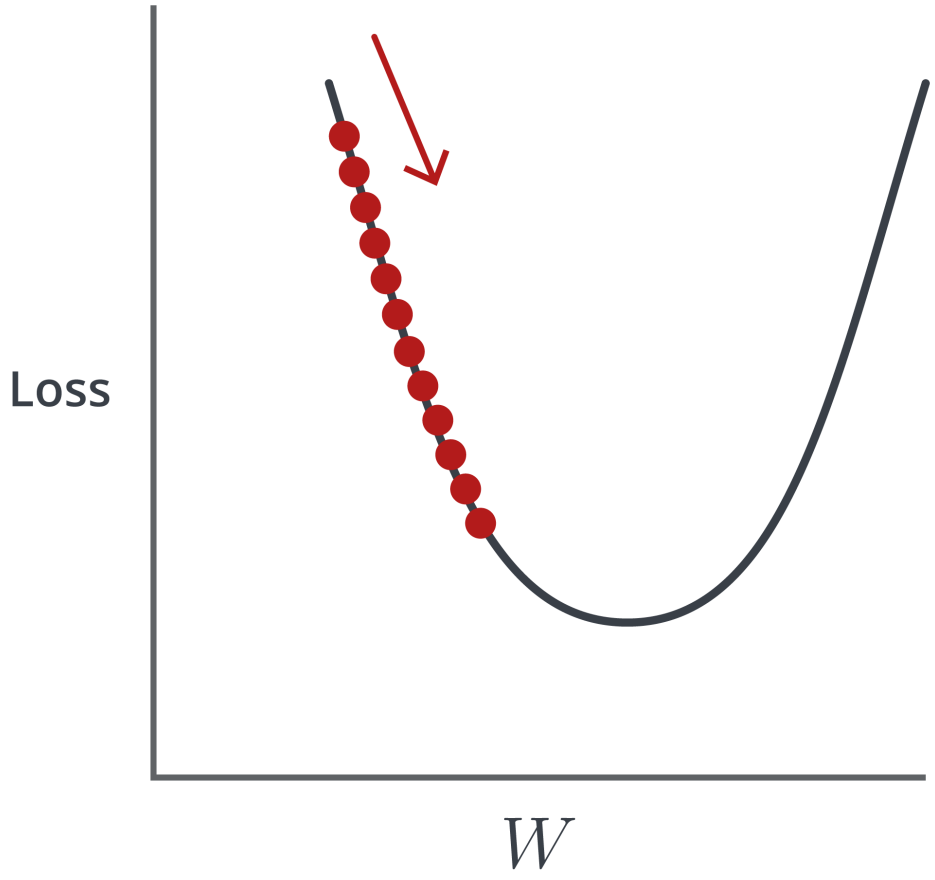


Figure 3. Gradient descent with a low learning rate

▼ Ideal Learning Rate

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

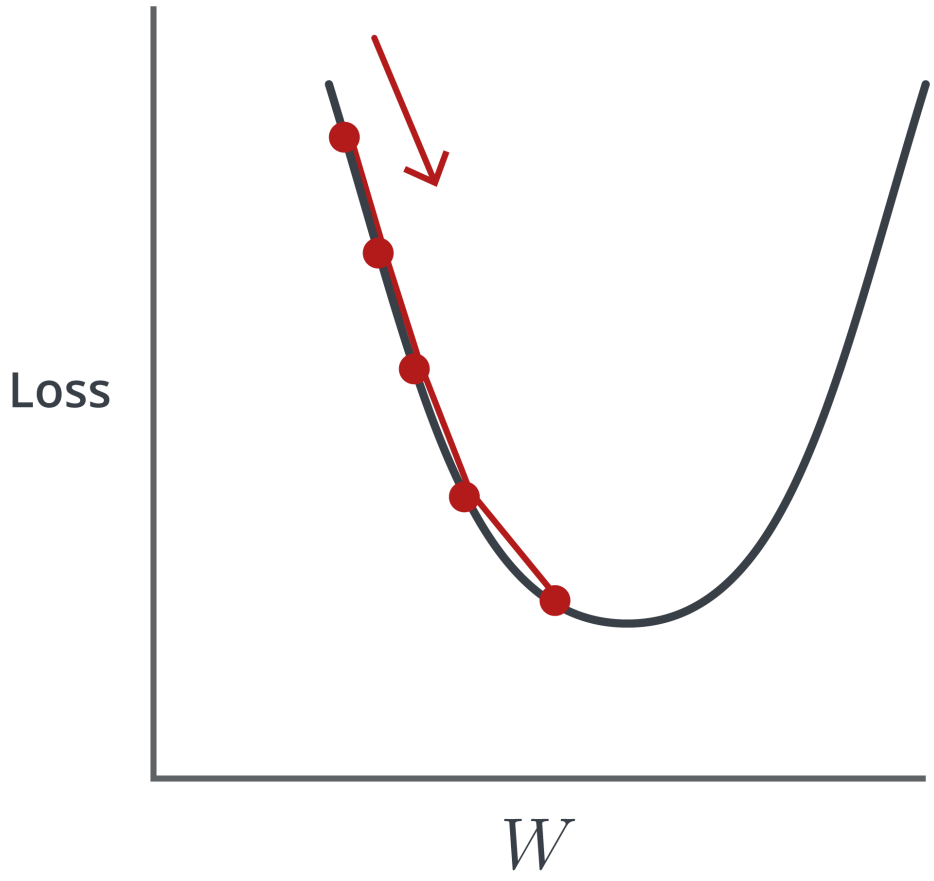


Figure 4. Gradient descent with an ideal learning rate

Choosing the right learning rate ensures efficient training of the model, 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 the details of how logistic regression works.

Regularization

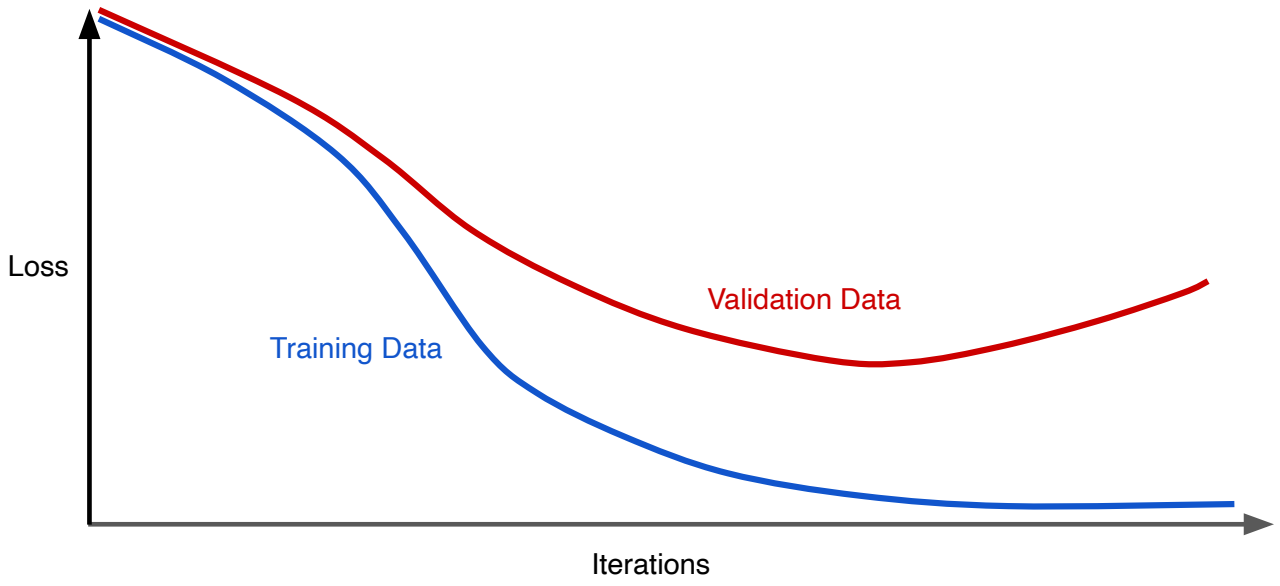
Our goal during training is to minimize loss. However, recall that a model can have low training loss but not generalize well to new data, as seen in the graph below. This is known as overfitting and is caused by a complex model that has learned the training data so closely that it does not generalize 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 L_1 and L_2 regularization

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



Our goal should therefore be to minimize loss plus 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: L_1 and L_2 regularization.

L_1 (also known as Lasso) and L_2 (also known as Ridge) regularization prevents weights from growing out of proportion. Oftentimes our input feature values can be too large or too imbalanced. Due to conditions like such, these features can overly influence the size of their corresponding weights. L_1 and L_2 add a penalty to the loss function for weights that have grown too large.

These penalty terms can be shown in the formulas below, in which $L_{Notregularized}$ represents the loss as computed by the loss function.

$$L_{L1regularized} = L_{Notregularized} + \frac{1}{C}\sum_{j=0}^{m} |w_j|$$

- L_1 regularization introduces a term that penalizes less-important features, reducing their coefficients to zero and effectively eliminating them. L_1 penalizes the count of non-zero weights. L_1 *penalizes |weight|.*

$$L_{L2regularized} = L_{Notregularized} + \frac{1}{C}\sum_{j=0}^{m} {w_j}^2$$

- L_2 regularization introduces a regularization term to the loss function that is the sum of squares of all feature weights. L_2 *penalizes* $weight^2$.

Let's look at an example of L_2 regularization. Say we have training data with 4 features. This means our model will have four weights. 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.

Below we will determine the L_2 regularization term that will be added as a penalty to the loss function.

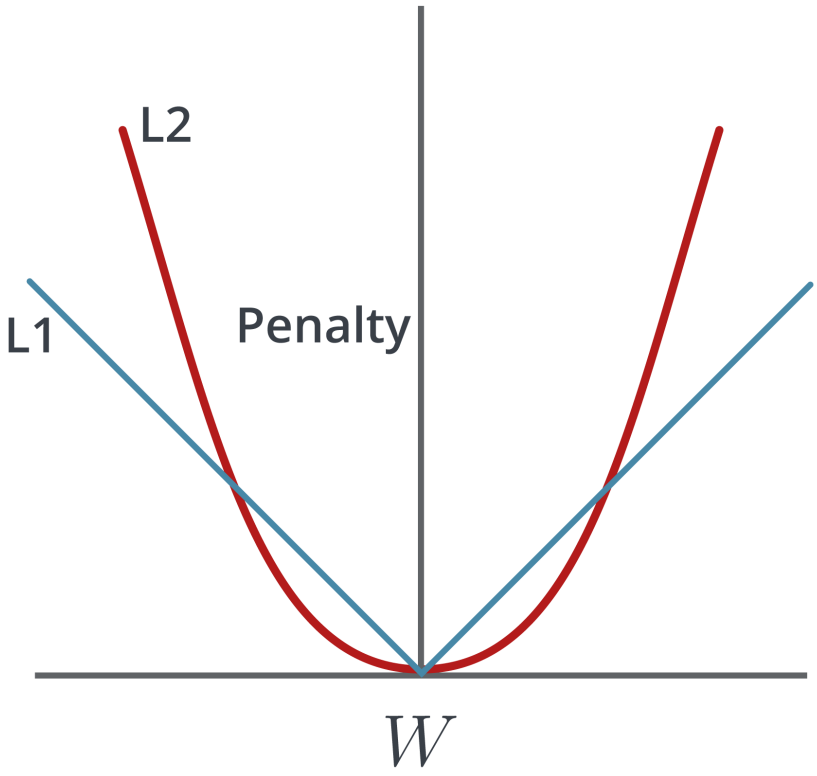
$$w_1^2 + w_2^2 + w_3^2 + w_4^2$$
$$3.1^2 + 5.2^2 + 1493^2 + (-4.3)^2$$
$$9.61 + 27.04 + 2,229,049 + 18.49$$
$$= 2,229,104.14$$

The L_2 regularization term that will be added to the loss function as a penalty is 2,229,104.14. This encourages our model to reduce the model complexity.

Hyperparameter C

The penalties are often accompanied by a factor 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 (L_1 or L_2), the value of C can be fine-tuned to improve the performance of the model. To further solidify our understanding of L_1 and L_2 regularization, we will visualize their penalties in the image below.



Penalties of L_1 and L_2

As W becomes larger, the penalty will be higher for both L_1 and L_2 . L_1, however, tends to exert a downward pressure regardless of the magnitude of W. Even as the weights are getting closer to zero, it will still exert pressure to attempt to bring it even closer to 0. L_2, on the other hand, gradually reduces downward pressure as weights approach 0. This is why L_1 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, L_1 can almost be considered an act of feature selection, as it eliminates some of the weights of the features.

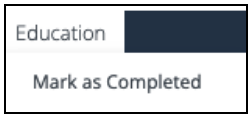
Back to Table of Contents

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. After submission, the Jupyter Notebook will always 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 assignment will be graded by your facilitator.

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

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.

Before you begin:

Please review **eCornell's policy regarding plagiarism** (the presentation of someone else's work as your own without source credit).

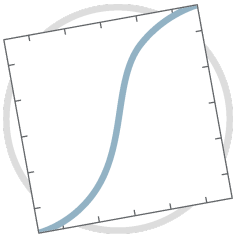
Back to Table of Contents

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

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.

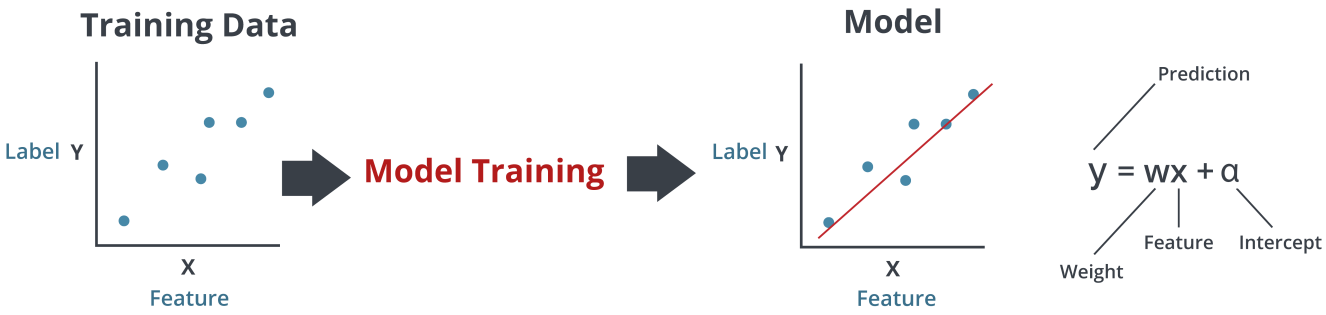
☆ Key Points

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.



This difference between linear regression and logistic regression can be visualized in figure 1 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.

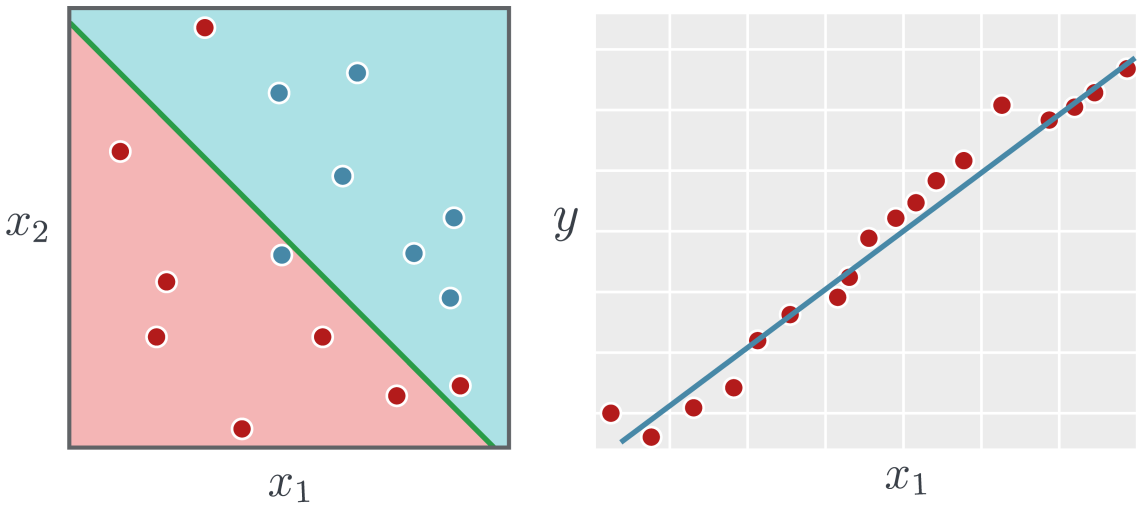


Figure 1. Logistic Regression (left) and Linear Regression (right) models visualized

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.

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

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. The prediction for an unlabeled example would be determined using $y = w_1 x_1 + w_2 x_2 + \alpha$.

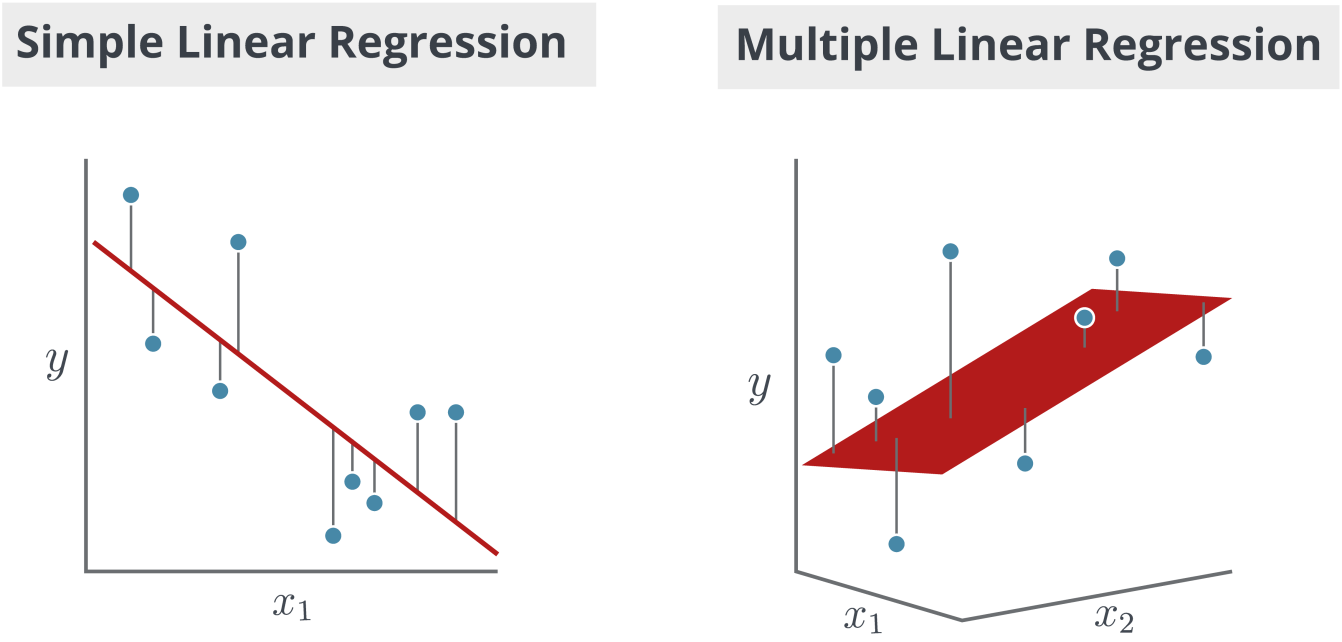
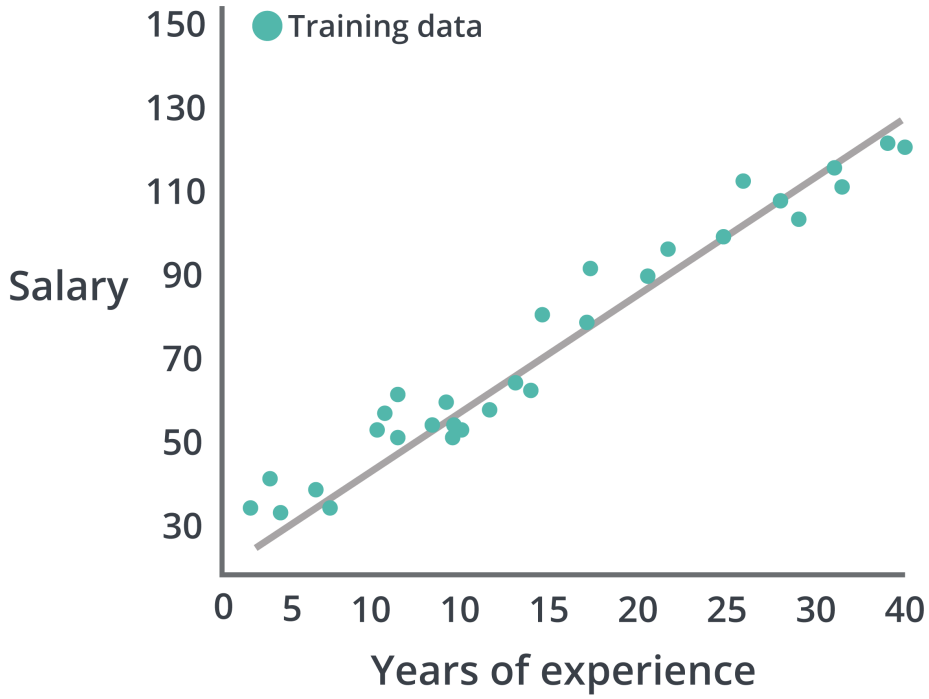


Figure 2. Linear Regression trained with a 1-dimensional feature (left) vs Linear Regression trained with 2-dimensional feature (right).

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.

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.



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 ($\boldsymbol{y = wx + \alpha}$) 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.

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.

This content cannot be rendered properly in the Course Transcript, please log into the course to view.

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 the scikit-learn'slogistic regression implementation.

You will be working in a Jupyter Notebook.

This 3-hour lab session will include:

- **5 minutes** - Icebreaker
- **30 minutes** - Concept Overview + Q&A
- **30 minutes** - Breakout Groups (Big Picture Questions)
- **15 minutes** - Sharing of Big Picture Group Responses
- **15 minutes** - Break
- **85 minutes** - Breakout Groups (Lab Assignment)
- **5 minutes -** 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

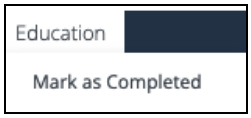
Back to Table of Contents

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. After submission, the Jupyter Notebook will always 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 lab assignment will be graded by your facilitator.

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