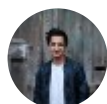


Machine Learning for Humans, Part 2.1: Supervised Learning

The two tasks of supervised learning: regression and classification. Linear regression, loss functions, and gradient descent.



Vishal Maini [Follow](#)

Aug 19, 2017 · 13 min read

This series is available as a full-length e-book! **Download here.** Free for download, contributions appreciated (paypal.me/ml4h)

How much money will we make by spending more dollars on digital advertising? Will this loan applicant pay back the loan or not? What's going to happen to the stock market tomorrow?

In supervised learning problems, we start with a data set containing **training examples** with associated correct **labels**. For example, when learning to classify handwritten digits, a supervised learning algorithm takes thousands of pictures of handwritten digits along with labels containing the correct number each image represents. The algorithm will then learn the relationship between the images and their associated numbers, and apply that learned relationship to classify completely new images (without labels) that the machine hasn't seen before. This is how you're able to deposit a check by taking a picture with your phone!

To illustrate how supervised learning works, let's examine the problem of **predicting annual income** based on the number of years of higher education someone has completed. Expressed more formally, we'd like to build a model that approximates the relationship f between the number of years of higher education X and corresponding annual income Y .

$$Y = f(X) + \epsilon$$

X (input) = years of higher education
Y (output) = annual income
f = function describing the relationship between X and Y
ε (epsilon) = random error term (positive or negative) with mean zero

Regarding epsilon:

(1) ϵ represents **irreducible error** in the model, which is a theoretical limit around the performance of your algorithm due to inherent noise in the phenomena you are trying to explain. For example, imagine building a model to predict the outcome of a coin flip.

(2) Incidentally, mathematician Paul Erdős referred to children as “epsilons” because in calculus (but not in stats!) ϵ denotes an arbitrarily small positive quantity. Fitting, no?

One method for predicting income would be to create a rigid rules-based model for how income and education are related. For example: “I’d estimate that for every additional year of higher education, annual income increases by \$5,000.”

```
income = ($5,000 * years_of_education) + baseline_income
```

This approach is an example of **engineering** a solution (vs. **learning** a solution, as with the linear regression method described below).

You could come up with a more complex model by including some rules about degree type, years of work experience, school tiers, etc. For example: “If they completed a Bachelor’s degree or higher, give the income estimate a 1.5x multiplier.”

But this kind of explicit rules-based programming doesn’t work well with complex data. Imagine trying to design an image classification algorithm made of if-then statements describing the combinations of pixel brightnesses that should be labeled “cat” or “not cat”.

Supervised machine learning solves this problem by getting the computer to *do the work for you*. By identifying patterns in the data, the machine is able to form heuristics. The primary difference between this and human learning is that machine learning runs on computer hardware and is best understood through the lens of computer science

and statistics, whereas human pattern-matching happens in a biological brain (while accomplishing the same goals).

In supervised learning, the machine attempts to learn the relationship between income and education *from scratch*, by running **labeled training data** through a **learning algorithm**. This learned function can be used to estimate the income of people whose income Y is unknown, as long as we have years of education X as inputs. In other words, we can apply our model to the **unlabeled test data** to estimate Y .

The goal of supervised learning is to **predict Y as accurately as possible** when given new examples where X is known and Y is unknown. In what follows we'll explore several of the most common approaches to doing so.

The two tasks of supervised learning: regression and classification

Regression: predict a continuous numerical value. *How much will that house sell for?*

Classification: assign a label. *Is this a picture of a cat or a dog?*

The rest of this section will focus on regression. In [Part 2.2](#) we'll dive deeper into classification methods.

Regression: predicting a continuous value

Regression predicts a **continuous target variable Y** . It allows you to estimate a value, such as housing prices or human lifespan, based on input data X .

Here, **target variable** means the unknown variable we care about predicting, and **continuous** means there aren't gaps (discontinuities) in the value that Y can take on. A person's weight and height are continuous values. **Discrete** variables, on the other hand, can only take on a finite number of values—for example, the number of kids somebody has is a discrete variable.

Predicting income is a classic regression problem. Your **input data X** includes all relevant information about individuals in the data set that can be used to predict income, such as years of education, years of work experience, job title, or zip code. These attributes are called **features**, which can be **numerical** (e.g. years of work experience) or **categorical** (e.g. job title or field of study).

You'll want as many training observations as possible relating these features to the target output Y, so that your model can learn the relationship f between X and Y.

The data is split into a **training data set** and a **test data set**. The training set has labels, so your model can learn from these labeled examples. The test set does *not* have labels, i.e. you don't yet know the value you're trying to predict. It's important that your model can generalize to situations it hasn't encountered before so that it can perform well on the test data.

Regression

$$Y = f(X) + \epsilon, \text{ where } X = (x_1, x_2, \dots, x_n)$$

Training: machine learns f from labeled training data

Test: machine predicts Y from unlabeled testing data

*Note that X can be a **tensor** with an any number of dimensions. A 1D tensor is a vector (1 row, many columns), a 2D tensor is a matrix (many rows, many columns), and then you can have tensors with 3, 4, 5 or more dimensions (e.g. a 3D tensor with rows, columns, and depth). For a review of these terms, see the first few pages of this linear algebra review.*

In our trivially simple 2D example, this could take the form of a .csv file where each row contains a person's education level and income. Add more columns with more features and you'll have a more complex, but possibly more accurate, model.

| Supervised Learning: Regression | | | |
|---------------------------------|---------------|-------------------------------|------------|
| training set | Observation # | Years of Higher Education (X) | Income (Y) |
| | 1 | 4 | \$80,000 |
| | 2 | 5 | \$91,500 |
| | 3 | 0 | \$42,000 |
| | 4 | 2 | \$55,000 |
| | ... | ... | ... |
| | N | 6 | \$100,000 |
| test set | 1 | 4 | ??? |
| | 2 | 6 | ??? |

Machine Learning for Humans 🧠💡

So how do we solve these problems?

How do we build models that make accurate, useful predictions in the real world? We do so by using **supervised learning algorithms**.

Now let's get to the fun part: **getting to know the algorithms**. We'll explore some of the ways to approach regression and classification and illustrate key machine learning concepts throughout.

Linear regression (ordinary least squares)

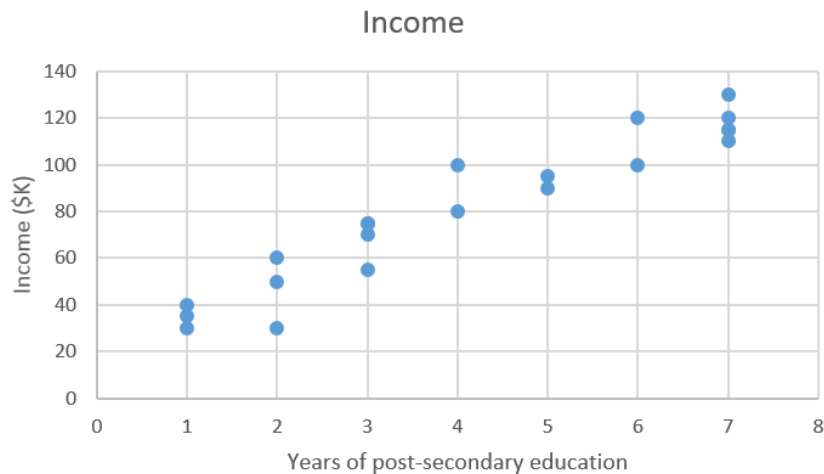
"Draw the line. Yes, this counts as machine learning."

First, we'll focus on solving the income prediction problem with linear regression, since linear models don't work well with image recognition tasks (this is the domain of deep learning, which we'll explore later).

We have our data set X , and corresponding target values Y . The goal of **ordinary least squares (OLS)** regression is to learn a linear model that we can use to predict a new y given a previously unseen x with as little error as possible. We want to guess how much income someone earns based on how many years of education they received.

```
X_train = [4, 5, 0, 2, ..., 6] # years of post-secondary
education

Y_train = [80, 91.5, 42, 55, ..., 100] # corresponding annual
incomes, in thousands of dollars
```



Linear regression is a **parametric method**, which means it makes an assumption about the form of the function relating X and Y (we'll cover examples of non-parametric methods later). Our model will be a function that predicts \hat{y} given a specific x :

$$\hat{y} = \beta_0 + \beta_1 * x + \epsilon$$

In this case, we make the explicit assumption that there is a **linear relationship** between X and Y—that is, for each one-unit increase in X, we see a constant increase (or decrease) in Y.

β_0 is the y-intercept and β_1 is the slope of our line, i.e. how much income increases (or decreases) with one additional year of education.

Our goal is to learn the **model parameters** (in this case, β_0 and β_1) that minimize error in the model's predictions.

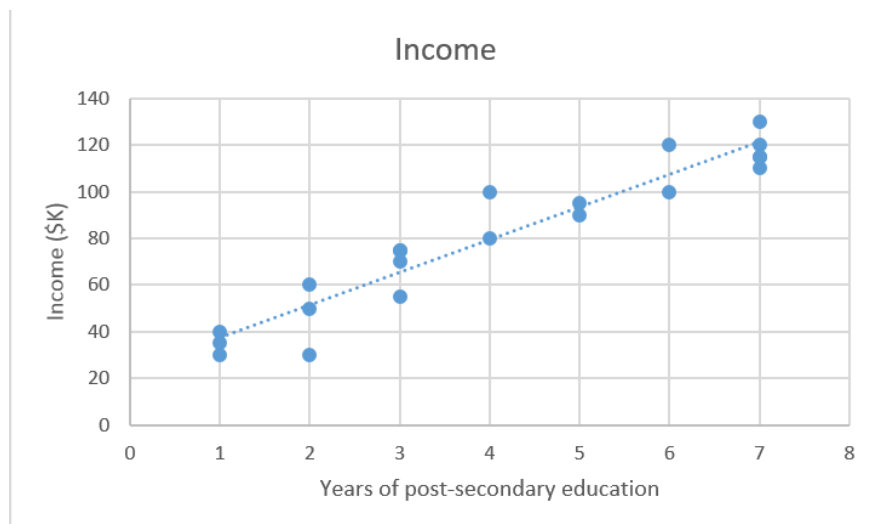
To find the best parameters:

1. Define a **cost function**, or **loss function**, that measures how inaccurate our model's predictions are.

2. Find the parameters that **minimize loss**, i.e. make our model as accurate as possible.

Graphically, in two dimensions, this results in a line of best fit. In three dimensions, we would draw a plane, and so on with higher-dimensional hyperplanes.

A note on dimensionality: our example is two-dimensional for simplicity, but you'll typically have more features (x 's) and coefficients (betas) in your model, e.g. when adding more relevant variables to improve the accuracy of your model predictions. The same principles generalize to higher dimensions, though things get much harder to visualize beyond three dimensions.



Mathematically, we look at the difference between each real data point (y) and our model's prediction (\hat{y}). Square these differences to avoid negative numbers and penalize larger differences, and then add them up and take the average. This is a measure of how well our data fits the line.

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n}$$

n = # of observations. Using $2*n$ instead of n makes the math work out more cleanly when taking the derivative to minimize loss, though some stats people say this is blasphemy. When you start having opinions on this kind of stuff, you'll know you are all the way in the rabbit hole.

For a simple problem like this, we can compute a closed form solution using calculus to find the optimal beta parameters that minimize our loss function. But as a cost function grows in complexity, finding a closed form solution with calculus is no longer feasible. This is the motivation for an iterative approach called **gradient descent**, which allows us to minimize a complex loss function.

Gradient descent: learn the parameters

“Put on a blindfold, take a step downhill. You’ve found the bottom when you have nowhere to go but up.”

Gradient descent will come up over and over again, especially in neural networks. Machine learning libraries like scikit-learn and TensorFlow use it in the background everywhere, so it’s worth understanding the details.

The goal of gradient descent is to find the minimum of our model’s loss function by iteratively getting a better and better approximation of it.

Imagine yourself walking through a valley with a blindfold on. Your goal is to find the bottom of the valley. How would you do it?

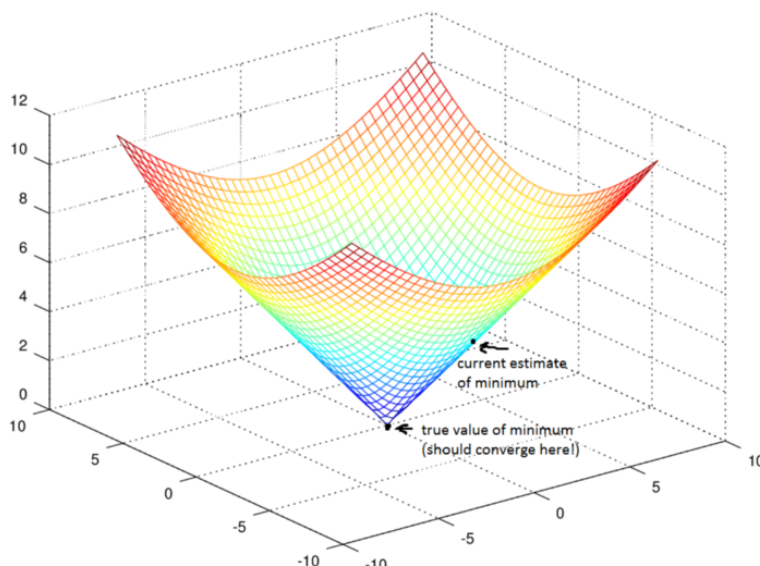
A reasonable approach would be to touch the ground around you and move in whichever direction the ground is sloping down most steeply. Take a step and repeat the same process continually until the ground is flat. Then you know you’ve reached the bottom of a valley; if you move in any direction from where you are, you’ll end up at the same elevation or further uphill.

Going back to mathematics, the ground becomes our loss function, and the elevation at the bottom of the valley is the minimum of that function.

Let’s take a look at the loss function we saw in regression:

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n}$$

We see that this is really a function of two variables: β_0 and β_1 . All the rest of the variables are determined, since X , Y , and n are given during training. We want to try to minimize this function.



The function is $f(\beta_0, \beta_1) = z$. To begin gradient descent, you make some guess of the parameters β_0 and β_1 that minimize the function.

Next, you find the partial derivatives of the loss function with respect to each beta parameter: $[dz/d\beta_0, dz/d\beta_1]$. A **partial derivative** indicates how much total loss is increased or decreased if you increase β_0 or β_1 by a very small amount.

Put another way, how much would increasing your estimate of annual income assuming zero higher education (β_0) increase the loss (i.e. inaccuracy) of your model? You want to go in the *opposite* direction so that you end up walking *downhill* and minimizing loss.

Similarly, if you increase your estimate of how much each incremental year of education affects income (β_1), how much does this increase loss (z)? If the partial derivative dz/β_1 is a *negative* number, then *increasing* β_1 is good because it will reduce total loss. If it's a *positive* number, you want to *decrease* β_1 . If it's zero, don't change β_1 because it means you've reached an optimum.

Keep doing that until you reach the bottom, i.e. the algorithm **converged** and loss has been minimized. There are lots of tricks and exceptional cases beyond the scope of this series, but generally, this is how you find the optimal **parameters** for your **parametric** model.

Overfitting

Overfitting: “Sherlock, your explanation of what just happened is too specific to the situation.” **Regularization:** “Don’t overcomplicate things, Sherlock. I’ll punch you for every extra word.” **Hyperparameter (λ):** “Here’s the strength with which I will punch you for every extra word.”

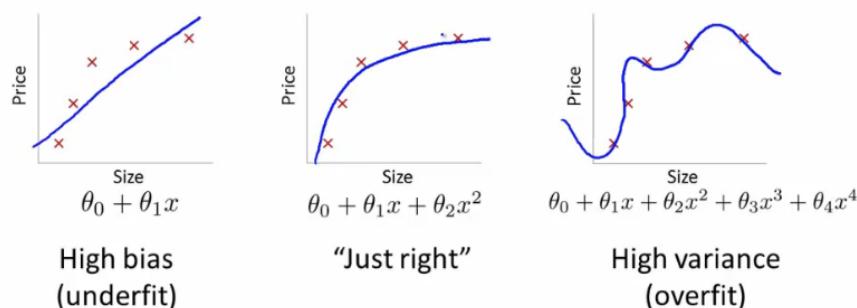
A common problem in machine learning is **overfitting**: learning a function that perfectly explains the training data that the model learned from, but doesn’t generalize well to unseen test data. Overfitting happens when a model *overlearns* from the training data to the point that it starts picking up idiosyncrasies that aren’t representative of patterns in the real world. This becomes especially problematic as you make your model increasingly complex. *Underfitting* is a related issue where your model is not complex enough to capture the underlying trend in the data.

Bias-Variance Tradeoff

Bias is the amount of error introduced by approximating real-world phenomena with a simplified model.

Variance is how much your model's test error changes based on variation in the training data. It reflects the model's sensitivity to the idiosyncrasies of the data set it was trained on.

As a model increases in complexity and it becomes more wiggly (**flexible**), its bias decreases (it does a good job of explaining the training data), but variance increases (it doesn't generalize as well). **Ultimately, in order to have a good model, you need one with low bias and low variance.**



Source: Coursera’s ML course, taught by Andrew Ng

Remember that *the only thing we care about is how the model performs on test data*. You want to predict which emails will be marked as spam before they’re marked, not just build a model that is 100% accurate at

reclassifying the emails it used to build itself in the first place. Hindsight is 20/20—the real question is whether the lessons learned will help in the future.

The model on the right has zero loss for the *training data* because it perfectly fits every data point. But the lesson doesn't generalize. It would do a horrible job at explaining a new data point that isn't yet on the line.

Two ways to combat overfitting:

1. **Use more training data.** The more you have, the harder it is to overfit the data by learning too much from any single training example.
2. **Use regularization.** Add in a penalty in the loss function for building a model that assigns too much explanatory power to any one feature or allows too many features to be taken into account.

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n} + \lambda \sum_{i=0}^1 \beta_i^2$$

The first piece of the sum above is our normal cost function. The second piece is a **regularization term** that adds a penalty for large beta coefficients that give too much explanatory power to any specific feature. With these two elements in place, the cost function now balances between two priorities: explaining the training data and preventing that explanation from becoming overly specific.

The **lambda** coefficient of the regularization term in the cost function is a **hyperparameter**: a general setting of your model that can be increased or decreased (i.e. **tuned**) in order to improve performance. A higher lambda value will more harshly penalize large beta coefficients that could lead to potential overfitting. To decide the best value of lambda, you'd use a method called **cross-validation** which involves holding out a portion of the training data during training, and then seeing how well your model explains the held-out portion. We'll go over this in more depth

Woo! We made it.

Here's what we covered in this section:

- How **supervised machine learning** enables computers to learn from labeled training data without being explicitly programmed
- The tasks of supervised learning: **regression** and **classification**
- **Linear regression**, a bread-and-butter **parametric** algorithm
- Learning **parameters** with **gradient descent**
- **Overfitting** and **regularization**

In the next section—[Part 2.2: Supervised Learning II](#)—we'll talk about two foundational methods of classification: **logistic regression** and **support vector machines**.

Practice materials & further reading

2.1a—Linear regression

For a more thorough treatment of linear regression, read chapters 1–3 of [An Introduction to Statistical Learning](#). The book is available for free online and is an excellent resource for understanding machine learning concepts with accompanying exercises.

For more practice:


- Play with the [Boston Housing dataset](#). You can either use software with nice GUIs like Minitab and Excel or do it the hard (but more rewarding) way with [Python](#) or [R](#).
- Try your hand at a [Kaggle challenge](#), e.g. [housing price prediction](#), and see how others approached the problem after attempting it yourself.

2.1b—Implementing gradient descent

To actually implement gradient descent in Python, check out [this tutorial](#). And [here](#) is a more mathematically rigorous description of the same concepts.

In practice, you'll rarely need to implement gradient descent from scratch, but understanding how it works behind the scenes will allow you to use it more effectively and understand why things break when they do.

. . .


Enter your email below if you'd like to stay up-to-date
with future content 

Send me the latest from Machine Learning for Humans!

Sign up





I agree to leave Medium and submit this information,
which will be collected and used according to
[Upscribe's privacy policy](#).

On Twitter? So are we. Feel free to keep in touch —
Vishal and Samer 

. . .

More from *Machine Learning for Humans*  

- [Part 1: Why Machine Learning Matters](#) 
- [Part 2.1: Supervised Learning](#) 
- [Part 2.2: Supervised Learning II](#)
- [Part 2.3: Supervised Learning III](#)
- [Part 3: Unsupervised Learning](#)
- [Part 4: Neural Networks & Deep Learning](#)
- [Part 5: Reinforcement Learning](#)
- [Appendix: The Best Machine Learning Resources](#)