Lecture 1, Part 1: Linear regression

Roger Grosse

1 Introduction

Let's jump right in and look at our first machine learning algorithm, **linear regression**. In regression, we are interested in predicting a scalar-valued target, such as the price of a stock. By linear, we mean that the target must be predicted as a linear function of the inputs. This is a kind of supervised learning algorithm; recall that, in supervised learning, we have a collection of training examples labeled with the correct outputs.

Regression is an important problem in its own right. But today's discussion will also highlight a number of themes which will recur throughout the course:

- Formulating a machine learning task mathematically as an optimization problem.
- Thinking about the data points and the model parameters as vectors.
- Solving the optimization problem using two different strategies: deriving a closed-form solution, and applying gradient descent. These two strategies are how we will derive nearly all of the learning algorithms in this course.
- Writing the algorithm in terms of linear algebra, so that we can think about it more easily and implement it efficiently in a high-level programming language.
- Making a linear algorithm more powerful using basis functions, or features.
- Analyzing the generalization performance of an algorithm, and in particular the problems of overfitting and underfitting.

1.1 Learning goals

- Know what objective function is used in linear regression, and how it is motivated.
- Derive both the closed-form solution and the gradient descent updates for linear regression.
- Write both solutions in terms of matrix and vector operations.
- Be able to implement both solution methods in Python.

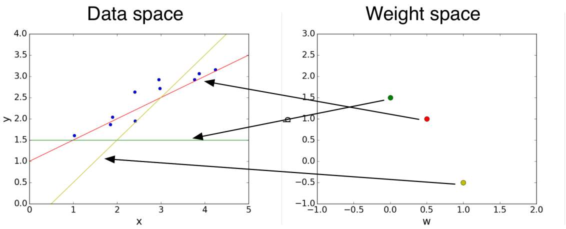

Figure 1: Three possible hypotheses for a linear regression model, shown in data space and weight space.

- Know how linear regression can learn nonlinear functions using feature maps.
- What is meant by generalization, overfitting, and underfitting? How can we measure generalization performance in practice?

2 Problem setup

In order to formulate a learning problem mathematically, we need to define two things: a model and a loss function. The **model**, or **architecture** defines the set of allowable **hypotheses**, or functions that compute predictions from the inputs. In the case of linear regression, the model simply consists of linear functions. Recall that a linear function of D inputs is parameterized in terms of D coefficients, which we'll call the **weights**, and an intercept term, which we'll call the **bias**. Mathematically, this is written as:

$$y = \sum_j w_j x_j + b. \tag{1}$$

Figure 1 shows two ways to visualize linear models. In this case, the data are one-dimensional, so the model reduces to simply $y = wx + b$. On one side, we have the **data space**, or **input space**, where t is plotted as a function of x. Three different possible linear fits are shown. On the other side, we have **weight space**, where the corresponding pairs (w, b) are plotted.

You should study these figures and try to understand how the lines in the left figure map onto the X's on the right figure. Think back to middle school. Hint: w is the slope of the line, and b is the y-intercept.

Clearly, some of these linear fits are better than others. In order to quantify how good the fit is, we define a **loss function**. This is a function $\mathcal{L}(y, t)$ which says how far off the prediction y is from the target t. In linear regression, we use **squared error**, defined as

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2. \tag{2}$$

This is small when y and t are close together, and large when they are far apart. In general, the value $y - t$ is known as the **residual**, and we'd like the residuals to be close to zero.

Why is there the factor of 1/2 in front? It just makes the calculations convenient.

When we combine our model and loss function, we get an optimization problem, where we are trying to minimize a **cost function** with respect to the model parameters (i.e. the weights and bias). The cost function is simply the loss, averaged over all the training examples. When we plug in

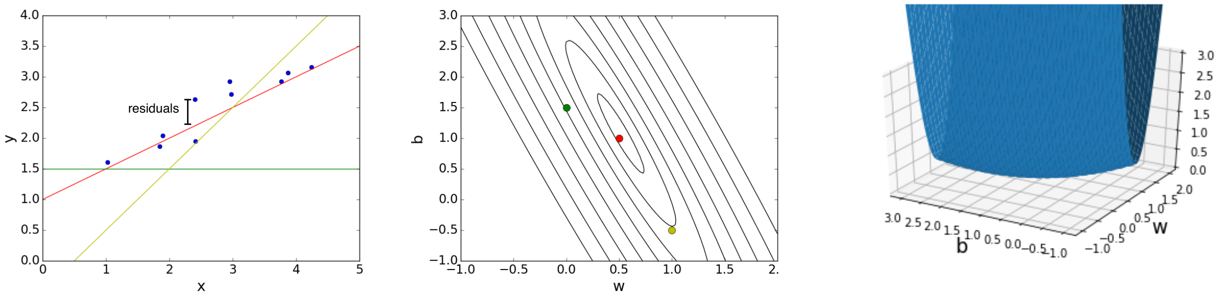

Figure 2: **Left:** three hypotheses for a regression dataset. **Middle:** Contour plot of least-squares cost function for the regression problem. Colors of the points match the hypotheses. **Right:** Surface plot matching the contour plot. Surface plots are usually hard to interpret, so we won't look at them very often.

the model definition (Eqn. 1), we get the following cost function:

$$\mathcal{J}(w_1, \ldots, w_D, b) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)}) \tag{3}$$

$$= \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right)^2 \tag{4}$$

$$= \frac{1}{2N} \sum_{i=1}^{N} \left(\sum_j w_j x_j^{(i)} + b - t^{(i)} \right)^2 \tag{5}$$

Our goal is to choose $w_1, \ldots, w_D$ and b to minimize $\mathcal{J}$. Note the difference between the loss function and the cost function. The loss is a function of the predictions and targets, while the cost is a function of the model parameters. The cost function is visualized in Figure 2.

The distinction between loss functions and cost functions will become clearer in a later lecture, when the cost function is augmented to include more than just the loss — it will also include a term called a regularizer which encourages simpler hypotheses.

3 Solving the optimization problem

In order to solve the optimization problem, we'll need the concept of partial derivatives. If you haven't seen these before, then you should go learn about them, on Khan Academy.[1] Just as a quick recap, suppose f is a function of $x_1, \ldots, x_D$. Then the partial derivative $\partial f / \partial x_i$ says in what way the value of f changes if you increase x_i by a small amount, while holding the rest of the arguments fixed. We can evaluate partial derivatives using the tools of single-variable calculus: to compute $\partial f / \partial x_i$ simply compute the (single-variable) derivative with respect to x_i, treating the rest of the arguments as constants.

Whenever we want to solve an optimization problem, a good place to start is to compute the partial derivatives of the cost function. Let's do that in the case of linear regression. Applying the chain rule for derivatives

[1] `https://www.khanacademy.org/math/calculus-home/multivariable-calculus/multivariable-derivatives#partial-derivatives`

to Eqn. 5, we get

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} \left(\sum_{j'} w_{j'} x_{j'}^{(i)} + b - t^{(i)} \right) \tag{6}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{j'} w_{j'} x_{j'}^{(i)} + b - t^{(i)} \right). \tag{7}$$

It's possible to simplify this a bit — notice that part of the term in parentheses is simply the prediction. The partial derivatives can be rewritten as:

It's always a good idea to try to simplify equations by finding familiar terms.

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} (y^{(i)} - t^{(i)}) \tag{8}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} y^{(i)} - t^{(i)}. \tag{9}$$

Now, it's good practice to do a sanity check of the derivatives. For instance, suppose we overestimated all of the targets. Then we should be able to improve the predictions by decreasing the bias, while holding all of the weights fixed. Does this work out mathematically? Well, the residuals $y^{(i)} - t^{(i)}$ will be positive, so based on Eqn. 9, $\partial \mathcal{J} / \partial b$ will be positive. This means increasing the bias will increase $\mathcal{J}$, and deceasing the bias will decrease $\mathcal{J}$ — which matches up with our expectation. So Eqn. 9 is plausible. Try to come up with a similar sanity check for $\partial \mathcal{J} / \partial w_j$.

Later in this course, we'll introduce a more powerful way to test partial derivative computations, but you should still get used to doing sanity checks on all your computations!

Now how do we use these partial derivatives? Let's discuss the two methods which we will use throughout the course.

3.1 Direct solution

One way to compute the minimum of a function is to set the partial derivatives to zero. Recall from single variable calculus that (assuming a function is differentiable) the minimum $x^\star$ of a function f has the property that the derivative $\mathrm{d}f/\mathrm{d}x$ is zero at $x = x^\star$. Note that the converse is not true: if $\mathrm{d}f/\mathrm{d}x = 0$, then $x^\star$ might be a maximum or an inflection point, rather than a minimum. But the minimum can only occur at points that have derivative zero.

An analogous result holds in the multivariate case: if f is differentiable, then all of the partial derivatives $\partial f / \partial x_i$ are zero at the minimum. The intuition is simple: if $\partial f / \partial x_i$ is positive, then one can decrease f slightly by decreasing x_i slightly. Conversely, if $\partial f / \partial x_i$ is negative, then one can decrease f slightly by increasing x_i slightly. In either case, this implies we're not at the minimum. Therefore, if the minimum exists (i.e. f doesn't keep growing as x goes to infinity), it occurs at a **critical point**, i.e. a point where the partial derivatives are zero. This gives us a strategy for finding minima: set the partial derivatives to zero, and solve for the parameters. This method is known as **direct solution**.

Let's apply this to linear regression. For simplicity, let's assume the model doesn't have a bias term. (We actually don't lose anything by getting rid of the bias. Just add a "dummy" input x_0 which always takes the value 1; then the weight w_0 acts as a bias.) We simplify Eqn. 6 to remove the bias, and set the partial derivatives to zero:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} \left(\sum_{j'=1}^{D} w_{j'} x_{j'}^{(i)} - t^{(i)} \right) = 0 \tag{10}$$

Since we're trying to solve for the weights, let's pull these out:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{j'=1}^{D} \left(\sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)} \right) w_{j'} - \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)} = 0 \tag{11}$$

The details of this equation aren't important; what's important is that we've wound up with a system of D linear equations in D variables. In other words, we have the system of linear equations

$$\sum_{j'=1}^{D} A_{jj'} w_{j'} - c_j = 0 \qquad \forall j \in \{1, \ldots, D\}, \tag{12}$$

where $A_{jj'} = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} x_{j'}^{(i)}$ and $c_j = \frac{1}{N} \sum_{i=1}^{N} x_j^{(i)} t^{(i)}$. As computer scientists, we're done, because this gives us an algorithm for finding the optimal regression weights: we first compute all the values $A_{jj'}$ and c_j, and then solve the system of linear equations using a linear algebra library such as NumPy. (We'll give an implementation of this later in this lecture.)

Note that the solution we just derived is *very* particular to linear regression. In general, the system of equations will be *nonlinear*, and except in rare cases, systems of nonlinear equations don't have closed form solutions. Linear regression is very unusual, in that it has a closed-form solution. We'll only be able to come up with closed form solutions for a handful of the algorithms we cover in this course.

3.2 Gradient descent

Now let's minimize the cost function a different way: **gradient descent**. This is an example of an **iterative algorithm**, which means that we apply a certain update rule over and over again, and if we're lucky, our **iterates** will gradually improve according to our objective function. To do gradient descent, we initialize the weights to some value (e.g. all zeros), and repeatedly adjust them in the direction that most decreases the cost function. If we visualize the cost function as a surface, so that lower is better, this is the direction of **steepest descent**. We repeat this procedure until the iterates **converge**, or stop changing much. (Or, in practice, we often run it until we get tired of waiting.) If we're lucky, the final iterate will be close to the optimum.

In order to make this mathematically precise, we must introduce the **gradient**, the direction of steepest ascent (i.e. fastest increase) of a function. The entries of the gradient vector are simply the partial derivatives with respect to each of the variables:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix} \tag{13}$$

The reason that this formula gives the direction of steepest ascent is beyond the scope of this course. (You would learn about it in a multivariable calculus class.) But this suggests that to decrease a function as quickly as possible, we should update the parameters in the direction opposite the gradient.

We can formalize this using the following update rule, which is known as **gradient descent**:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}, \tag{14}$$

or in terms of coordinates,

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}. \tag{15}$$

The symbol ← means that the left-hand side is updated to take the value on the right-hand side; the constant α is known as a **learning rate**. The larger it is, the larger a step we take. We'll talk in much more detail later about how to choose a learning rate, but in general it's good to choose a small value such as 0.01 or 0.001. If we plug in the formula for the partial derivatives of the regression model (Eqn. 8), we get the update rule:

$$w_j \leftarrow w_j - \alpha \frac{1}{N} \sum_{i=1}^{N} x_j (y^{(i)} - t^{(i)}) \tag{16}$$

In practice, we rarely if ever go through this last step. From a software engineering perspective, it's better to write our code in a modular way, where one function computes the gradient, and another function implements gradient descent, taking the gradient as given.

So we just repeat this update lots of times. What does gradient descent give us in the end? For analyzing iterative algorithms, it's useful to look for **fixed points**, i.e. points where the iterate doesn't change. By inspecting Eqn. 14, setting the left-hand side equal to the right-hand side, we see that the fixed points occur where $\partial \mathcal{J}/\partial \mathbf{w} = 0$. Since we know the gradient must be zero at the optimum, this is an encouraging sign that maybe it will converge to the optimum. But there are lots of things that could go wrong, such as divergence or local optima; we'll look at these in more detail in a later lecture.

Lecture 9 discusses optimization issues.

You might ask: by setting the partial derivatives to zero, we compute the exact solution. With gradient descent, we never actually reach the optimum, but merely approach it gradually. Why, then, would we ever prefer gradient descent? Two reasons:

1. We can only solve the system of equations explicitly for a handful of models. By contrast, we can apply gradient descent to any model for which we can compute the gradient. This is usually pretty easy to do efficiently. Importantly, it can usually be done *automatically*, so software packages like Theano and TensorFlow can save us from ever having to compute partial derivatives by hand.

2. Solving a large system of linear equations can be expensive, possibly many orders of magnitude more expensive than a single gradient descent update. Therefore, gradient descent can sometimes find a reasonable solution much faster than solving the linear system. Therefore, gradient descent is often more practical than computing exact solutions, even for models where we are able to derive the latter.

For these reasons, gradient descent will be our workhorse throughout the course. We will use it to train almost all of our models, with the exception of a handful for which we can derive exact solutions.

4 Vectorization

Now it's time to bring in linear algebra. We're going to rewrite the linear regression model, as well as both solution methods, in terms of operations on matrices and vectors. This process is known as **vectorization**. There are two reasons for doing this:

Vectorization takes a lot of practice to get used to. We'll cover a lot of examples in the first few weeks of the course. I'd recommend practicing these until they start to feel natural.

1. The formulas can be much simpler, more compact, and more readable in this form.
2. Vectorized code can be much faster than explicit `for`-loops, for several reasons.
    - High-level languages like Python can introduce a lot of interpreter overhead, and if we explicitly write a for-loop corresponding to Eqn. 16, this might be 10-100 times slower than the C equivalent. If we instead write the algorithm in terms of a much smaller number of linear algebra operations, then it can perform the same computations much faster with minimal interpreter overhead.
    - Since linear algebra is used all over the place, linear algebra libraries have been extremely well optimized for various computer architectures. Hence, they use much more efficient memory-access patterns than a naïve for-loop, even one written in C.
    - Matrix multiplication is inherently highly parallelizable and involve little control flow. Hence, it's ideal for **graphics processing unit (GPU)** architectures. We're not going to talk much about GPUs in this course, but just think "matrix multiplication + GPU = good". If you run vectorized code on a GPU using a framework like TensorFlow or PyTorch, it may run 50 times faster than the CPU version. As it turns out, most of the computation in deep learning is matrix multiplications, which is why it's been such an incredibly good match for GPUs.

First, we need to represent the data and model parameters in the form of matrices and vectors. If we have N training examples, each D-dimensional, we will represent the inputs as an $N \times D$ matrix $\mathbf{X}$. Each row of $\mathbf{X}$ corresponds to a training example, and each column corresponds to a single input dimension. The weights are represented as a D-dimensional vector $\mathbf{w}$, and the targets are represented as a N-dimensional vector $\mathbf{t}$.

In general, matrices will be denoted with capital boldface, vectors with lowercase boldface, and scalars with plain type.

The predictions are computed using a matrix-vector product

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}, \tag{17}$$

where $\mathbf{1}$ denotes a vector of all ones. We can express the cost function in vectorized form:

You should stop now and try to show that these equations are equivalent to Eqns. 3–5. The only way you get comfortable with this is by practicing.

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2 \tag{18}$$

$$= \frac{1}{2N}\left\|\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t}\right\|^2. \tag{19}$$

Note that this is considerably simpler than Eqn. 5. Even more importantly, it saves us from having to explicitly sum over the indices i and j. As our models get more complicated, we would run out of convenient letters to use as indices if we didn't vectorize.

Now let's revisit the exact solution for linear regression. We derived a system of linear equations, with coefficients $A_{jj'} = \frac{1}{N}\sum_{i=1}^N x_j^{(i)} x_{j'}^{(i)}$ and $c_j = \frac{1}{N}\sum_{i=1}^N x_j^{(i)} t^{(i)}$. In terms of linear algebra, we can write these as the matrix $\mathbf{A} = \frac{1}{N}\mathbf{X}^\top\mathbf{X}$ and $\mathbf{c} = \frac{1}{N}\mathbf{X}^\top\mathbf{t}$. The solution to the linear system $\mathbf{A}\mathbf{w} = \mathbf{c}$ is given by $\mathbf{w} = \mathbf{A}^{-1}\mathbf{c}$ (assuming $\mathbf{A}$ is invertible), so this gives us a formula for the optimal weights:

$$\mathbf{w} = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{t}. \tag{20}$$

An exact solution which we can express with a formula is known as a **closed-form solution**.

Similarly, we can vectorize the gradient descent update from Eqn. 16:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N}\mathbf{X}^\top(\mathbf{y} - \mathbf{t}), \tag{21}$$

where $\mathbf{y}$ is computed as in Eqn. 17.

5 Feature mappings

Linear regression might sound pretty limited. What if the true relationship between inputs and targets is nonlinear? Fortunately, there's an easy way to use linear regression to learn nonlinear dependencies: use a feature mapping. I'll introduce this by way of an example. Suppose we want to approximate it with a cubic polynomial. In other words, we would compute the predictions as:

$$y = w_3 x^3 + w_2 x^2 + w_1 x + w_0. \tag{22}$$

This setting is known as **polynomial regression**.

Let's use the squared error loss function, just as with ordinary linear regression. The important thing to notice is that algorithmically, polynomial regression is *no different* from linear regression. We can apply any of the linear regression algorithms described above, using (x, x^2, x^3) as the inputs. Mathematically, we define a feature mapping $\boldsymbol{\psi}$, in this case

Just as in Section 3.1, we're including a constant feature to account for the bias term, since this simplifies the notation.

$$\boldsymbol{\psi}(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}, \tag{23}$$

and compute the predictions as $y = \mathbf{w}^\top\boldsymbol{\psi}(x)$ instead of $\mathbf{w}^\top\mathbf{x}$. The rest of the algorithm is completely unchanged.

Feature maps are a useful tool, but they're not a silver bullet, for several reasons:

- The features must be known in advance. It's not always easy to pick good features, and up until very recently, feature engineering would take up most of the time and ingenuity in building a practical machine learning system.
- In high dimensions, the feature representations can get very large. For instance, the number of terms in a cubic polynomial is *cubic* in the dimension!

It's possible to work with polynomial feature maps efficiently using something called the "kernel trick," but that's beyond the scope of this course.

In this course, rather than construct feature maps, we will use neural networks to learn nonlinear predictors directly from the raw inputs. In most cases, this eliminates the need for hand-engineering of features.

6 Generalization

We don't just want a learning algorithm to make correct predictions on the training examples; we'd like it to **generalize** to examples it hasn't seen before. The average squared error on novel examples is known as the **generalization error**, and we'd like this to be as small as possible.

Returning to the previous example, let's consider three different polynomial models: (a) a linear function, or equivalently, a degree 1 polynomial; (b) a cubic polynomial; (c) a degree-10 polynomial. The linear function may be too simplistic to describe the data; this is known as **underfitting**. The degree-10 polynomial may be able to fit every training example exactly, but only by learning a crazy function. It would make silly predictions everywhere except the observed data. This is known as **overfitting**. The cubic polynomial is a reasonable compromise. We need to worry about both underfitting and overfitting in pretty much every application of machine learning.

The terms underfitting and overfitting are a bit misleading, since they suggest the two phenomena are mutually exclusive. In fact, most machine learning models suffer from *both* problems simultaneously.

The degree of the polynomial is an example of a **hyperparameter**. Hyperparameters are values that we can't include in the training procedure itself, but which we need to set using some other means. In practice, we normally tune hyperparameters by partitioning the dataset into three different subsets:

Statisticians prefer the term *metaparameter* since *hyperparameter* has a different meaning in statistics.

1. The **training set** is used to train the model.
2. The **validation set** is used to estimate the generalization error of each hyperparameter setting.
3. The **test set** is used at the very end, to estimate the generalization error of the final model, once all hyperparameters have been chosen.

We will talk about validation and generalization in a lot more detail later on in this course.