

CSC321 Lecture 2: Linear Regression

Roger Grosse

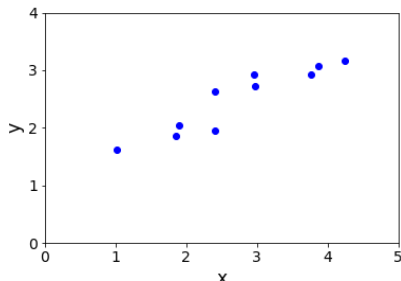
Overview

- First learning algorithm of the course: **linear regression**
 - **Task:** predict scalar-valued targets, e.g. stock prices (hence “regression”)
 - **Architecture:** linear function of the inputs (hence “linear”)

Overview

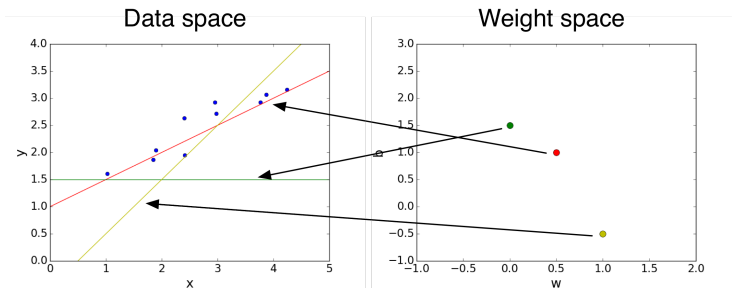
- First learning algorithm of the course: **linear regression**
 - **Task:** predict scalar-valued targets, e.g. stock prices (hence “regression”)
 - **Architecture:** linear function of the inputs (hence “linear”)
- Example of recurring themes throughout the course:
 - choose an **architecture** and a **loss function**
 - formulate an **optimization problem**
 - solve the optimization problem using one of two strategies
 - **direct solution** (set derivatives to zero)
 - **gradient descent**
 - **vectorize** the algorithm, i.e. represent in terms of linear algebra
 - make a linear model more powerful using **features**
 - understand how well the model **generalizes**

Problem Setup



- Want to predict a scalar t as a function of a scalar x
- Given a dataset of pairs $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^N$
- The $\mathbf{x}^{(i)}$ are called **inputs**, and the $t^{(i)}$ are called **targets**.

Problem Setup



- **Model:** y is a linear function of x :

$$y = wx + b$$

- y is the **prediction**
- w is the **weight**
- b is the **bias**
- w and b together are the **parameters**
- Settings of the parameters are called **hypotheses**

Problem Setup

- **Loss function:** squared error

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

- $y - t$ is the **residual**, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.

Problem Setup

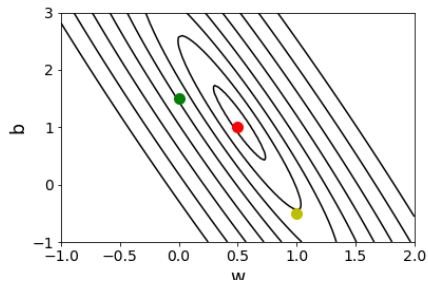
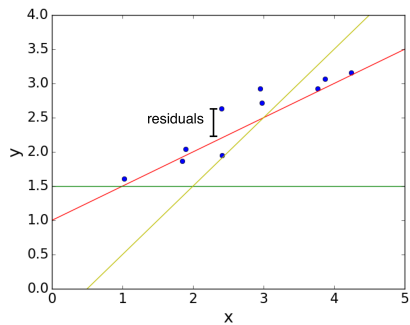
- **Loss function:** squared error

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

- $y - t$ is the **residual**, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- **Cost function:** loss function averaged over all training examples

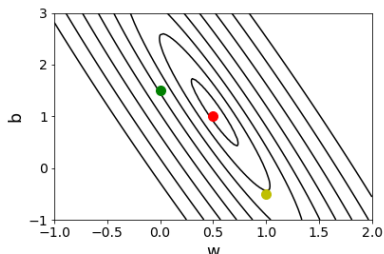
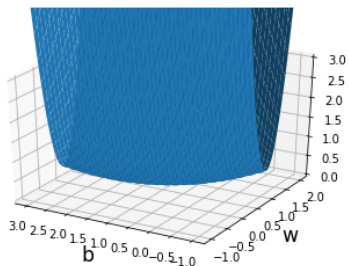
$$\begin{aligned}\mathcal{E}(w, b) &= \frac{1}{2N} \sum_{i=1}^N \left(y^{(i)} - t^{(i)} \right)^2 \\ &= \frac{1}{2N} \sum_{i=1}^N \left(wx^{(i)} + b - t^{(i)} \right)^2\end{aligned}$$

Problem Setup



Problem Setup

Surface plot vs. contour plot



Problem setup

- Suppose we have multiple inputs x_1, \dots, x_D . This is referred to as **multivariable regression**.
- This is no different than the single input case, just harder to visualize.
- Linear model:

$$y = \sum_j w_j x_j + b$$

Vectorization

- Computing the prediction using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- For-loops in Python are slow, so we **vectorize** algorithms by expressing them in terms of vectors and matrices.

$$\mathbf{w} = (w_1, \dots, w_D)^\top \quad \mathbf{x} = (x_1, \dots, x_D)$$

$$y = \mathbf{w}^\top \mathbf{x} + b$$

- This is simpler and much faster:

```
y = np.dot(w, x) + b
```

Vectorization

Why vectorize?

Vectorization

Why vectorize?

- The equations, and the code, will be simpler and more readable. Gets rid of dummy variables/indices!
- Vectorized code is much faster
 - Cut down on Python interpreter overhead
 - Use highly optimized linear algebra libraries
 - Matrix multiplication is very fast on a Graphics Processing Unit (GPU)

Vectorization

- We can take this a step further. Organize all the training examples into a matrix \mathbf{X} with one row per training example, and all the targets into a vector \mathbf{t} .

one feature across
all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training
example (vector)

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^\top \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

Vectorization

- Computing the squared error cost across the whole dataset:

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

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

- In Python:

```
y = np.dot(X, w) + b
cost = np.sum((y - t) ** 2) / (2. * N)
```

- **Example in tutorial**

Solving the optimization problem

- We defined a cost function. This is what we'd like to minimize.
- Recall from calculus class: minimum of a smooth function (if it exists) occurs at a **critical point**, i.e. point where the derivative is zero.
- Multivariate generalization: set the partial derivatives to zero. We call this **direct solution**.

Direct solution

- **Partial derivatives:** derivatives of a multivariate function with respect to one of its arguments.

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \rightarrow 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- To compute, take the single variable derivatives, pretending the other arguments are constant.
- Example: partial derivatives of the prediction y

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j} \left[\sum_{j'} w_{j'} x_{j'} + b \right]$$

$$= x_j$$

$$\frac{\partial y}{\partial b} = \frac{\partial}{\partial b} \left[\sum_{j'} w_{j'} x_{j'} + b \right]$$

$$= 1$$

Direct solution

- Chain rule for derivatives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial w_j} \\ &= \frac{d}{dy} \left[\frac{1}{2}(y - t)^2 \right] \cdot x_j \\ &= (y - t)x_j \\ \frac{\partial \mathcal{L}}{\partial b} &= y - t\end{aligned}$$

- We will give a more precise statement of the Chain Rule in a few weeks. It's actually pretty complicated.
- Cost derivatives (average over data points):

$$\begin{aligned}\frac{\partial \mathcal{E}}{\partial w_j} &= \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} \\ \frac{\partial \mathcal{E}}{\partial b} &= \frac{1}{N} \sum_{i=1}^N y^{(i)} - t^{(i)}\end{aligned}$$

Direct solution

- The minimum must occur at a point where the partial derivatives are zero.

$$\frac{\partial \mathcal{E}}{\partial w_j} = 0 \quad \frac{\partial \mathcal{E}}{\partial b} = 0.$$

- If $\partial \mathcal{E} / \partial w_j \neq 0$, you could reduce the cost by changing w_j .
- This turns out to give a system of linear equations, which we can solve efficiently. **Full derivation in tutorial and the readings.**
- Optimal weights:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

- Linear regression is one of only a handful of models in this course that permit direct solution.

Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: **gradient descent**.
- Gradient descent is an **iterative algorithm**, which means we apply an update repeatedly until some criterion is met.
- We **initialize** the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the **direction of steepest descent**.

Gradient descent

- Observe:
 - if $\partial\mathcal{E}/\partial w_j > 0$, then increasing w_j increases \mathcal{E} .
 - if $\partial\mathcal{E}/\partial w_j < 0$, then increasing w_j decreases \mathcal{E} .
- The following update decreases the cost function:

$$\begin{aligned}w_j &\leftarrow w_j - \alpha \frac{\partial\mathcal{E}}{\partial w_j} \\ &= w_j - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)}\end{aligned}$$

- α is a **learning rate**. The larger it is, the faster \mathbf{w} changes.
 - We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001

Gradient descent

- This gets its name from the **gradient**:

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

- This is the direction of fastest increase in \mathcal{E} .

Gradient descent

- This gets its name from the **gradient**:

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

- This is the direction of fastest increase in \mathcal{E} .
- Update rule in vector form:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{E}}{\partial \mathbf{w}} \\ &= \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)} \end{aligned}$$

- Hence, gradient descent updates the weights in the direction of fastest *decrease*.

Gradient descent

Visualization:

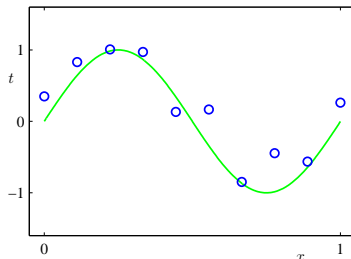
http://www.cs.toronto.edu/~guerzhoy/321/lec/W01/linear_regression.pdf#page=21

Gradient descent

- Why gradient descent, if we can find the optimum directly?
 - GD can be applied to a much broader set of models
 - GD can be easier to implement than direct solutions, especially with automatic differentiation software
 - For regression in high-dimensional spaces, GD is more efficient than direct solution (matrix inversion is an $\mathcal{O}(D^3)$ algorithm).

Feature mappings

- Suppose we want to model the following data



-Pattern Recognition and Machine Learning, Christopher Bishop.

- One option: fit a low-degree polynomial; this is known as **polynomial regression**

$$y = w_3x^3 + w_2x^2 + w_1x + w_0$$

- Do we need to derive a whole new algorithm?

Feature mappings

- We get polynomial regression for free!
- Define the feature map

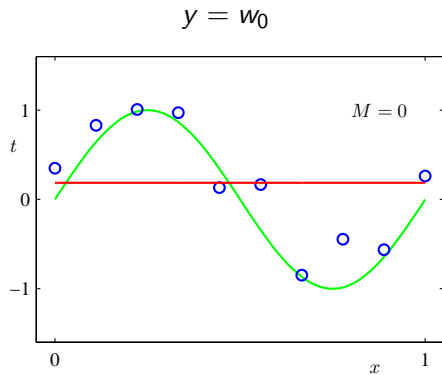
$$\phi(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

- Polynomial regression model:

$$y = \mathbf{w}^\top \phi(x)$$

- All of the derivations and algorithms so far in this lecture remain exactly the same!

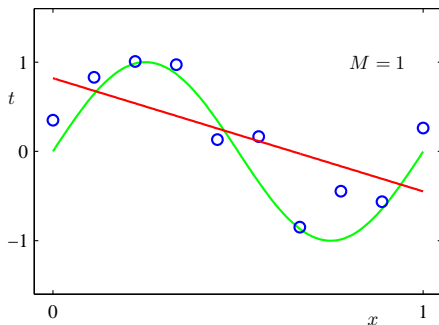
Fitting polynomials



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

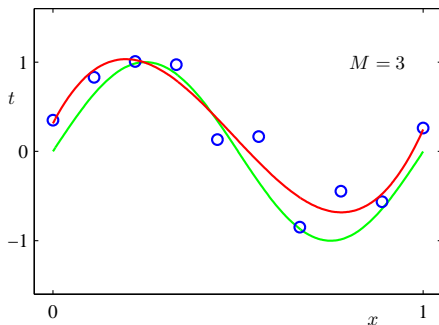
$$y = w_0 + w_1x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

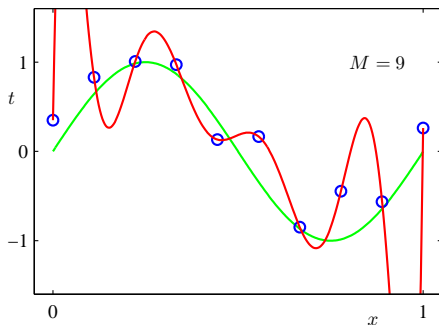
$$y = w_0 + w_1x + w_2x^2 + w_3x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

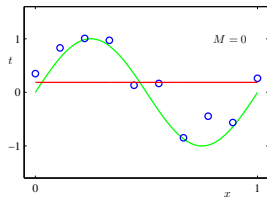
$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$



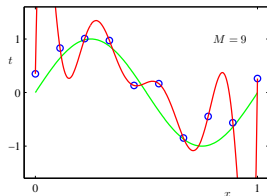
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

Underfitting : The model is too simple - does not fit the data.

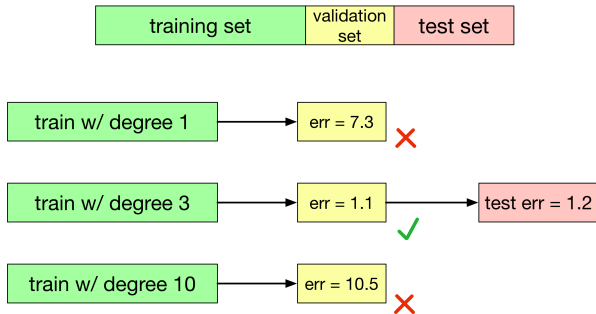


Overfitting : The model is too complex - fits perfectly, does not generalize.



Generalization

- We would like our models to **generalize** to data they haven't seen before
- The degree of the polynomial is an example of a **hyperparameter**, something we can't include in the training procedure itself
- We can tune hyperparameters using a **validation set**:



Foreshadowing

- Feature maps aren't a silver bullet:
 - It's not always easy to pick good features.
 - In high dimensions, polynomial expansions can get very large!
- Until the last few years, a large fraction of the effort of building a good machine learning system was feature engineering
- We'll see that neural networks are able to learn nonlinear functions directly, avoiding hand-engineering of features