Solving Linear Regression & Polynomial Regression

CMPUT 296: Basics of Machine Learning

Textbook §7.2-7.4

Recap: Linear Regression

A **linear predictor** has the form

$$f(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d = \sum_{j=0}^{d} w_j x_j = \mathbf{w}^T \mathbf{x}$$

Probabilistic approach:

1. Assume **i.i.d. Gaussian noise**: $Y \sim \mathcal{N}(\omega^T \mathbf{x}, \sigma^2)$
2. Use MLE to estimate model from resulting **parametric family**
   $\mathcal{F} = \left\{ p(\cdot \mid \mathbf{x}) = \mathcal{N}(\mathbf{w}^T \mathbf{x}, \sigma^2) \mid \mathbf{w} \in \mathbb{R}^{d+1} \right\}$
3. Use the **optimal predictor** for the estimated model $\mathbf{w}^*$:
   $f^*(\mathbf{x}) = \mathbb{E}[Y \mid X = \mathbf{x}] = \mathbf{w}^T \mathbf{x}$

Outline

1. Recap & Logistics
2. Solving Linear Regression
3. Polynomial Regression

Linear Regression: Analytical Solution

For a small enough dataset, we can find $\mathbf{w}_{\text{MLE}}$ **analytically**.

$$\mathbf{w}_{\text{MLE}} = \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} c(\mathbf{w}) = \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \sum_{i=1}^{n} (y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

$$= \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

Constant doesn't change argmin

$$= \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

Keep total in same range as size of $\mathscr{D}$ grows

$$= \arg\min_{\mathbf{w}\in\mathbb{R}^{d+1}} \frac{1}{n}\sum_{i=1}^{n} c_i(\mathbf{w}), \text{ where } c_i(\mathbf{w}) = \frac{1}{2}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

We will optimize c_i separately

Stationary Points (1)

We can compute the gradient for each datapoint separately:

$$\nabla c(\mathbf{w}) = \nabla \left[\frac{1}{n} \sum_{i=1}^{n} c_i(\mathbf{w}) \right] = \frac{1}{n} \sum_{i=1}^{n} \nabla c_i(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \begin{bmatrix} \boxed{\dfrac{\partial c_i(\mathbf{w})}{\partial w_0}} \\ \vdots \\ \dfrac{\partial c_i(\mathbf{w})}{\partial w_d} \end{bmatrix}$$

Recall that the gradient is just a vector of partial derivatives (one for each $\mathbf{w}_j$), so we can actually compute **each element** of the gradient separately.

Partial Derivatives of c_i

$$\frac{\partial c_i(\mathbf{w})}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}(\mathbf{x}_i^T\mathbf{w} - y_i)^2$$

$$= \frac{\partial}{\partial w_j}\frac{1}{2}u^2 \quad \text{where } u = (\mathbf{x}_i^T\mathbf{w} - y_i)$$

$$= \frac{\partial \frac{1}{2}u^2}{\partial u}\frac{\partial u}{\partial w_j} \quad \text{by the chain rule}$$

$$= u\frac{\partial u}{\partial w_j}$$

$$= u\frac{\partial}{\partial w_j}\mathbf{x}_i^T\mathbf{w} - y_i$$

$$= u\frac{\partial}{\partial w_j}\sum_{m=0}^{d} x_{im}w_m - y_i$$

$$= u\sum_{m=0}^{d}\frac{\partial x_{im}w_m}{\partial w_j} \qquad \frac{\partial y_i}{\partial w_j} = 0$$

$$= u x_{ij}$$

$$= (\mathbf{x}_i^T\mathbf{w} - y_i)x_{ij}$$

What is $\frac{\partial x_{im}w_m}{\partial w_j}$ for $m \neq j$?

Stationary Points (2)

$$\frac{\partial c_i(\mathbf{w})}{\partial w_j} = (\mathbf{x}_i^T \mathbf{w} - y_i) x_{ij}$$

$$\nabla c(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \begin{bmatrix} \frac{\partial c_i(\mathbf{w})}{\partial w_0} \\ \vdots \\ \frac{\partial c_i(\mathbf{w})}{\partial w_d} \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum_{i=1}^{n} \frac{\partial c_i(\mathbf{w})}{\partial w_0} \\ \vdots \\ \frac{1}{n} \sum_{i=1}^{n} \frac{\partial c_i(\mathbf{w})}{\partial w_d} \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i^T \mathbf{w} - y_i) x_{i0} \\ \vdots \\ \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i^T \mathbf{w} - y_i) x_{id} \end{bmatrix}$$

So to set $\nabla c(\mathbf{w}) = 0$, we must solve a system of $d + 1$ equations:

$$\frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i^T \mathbf{w} - y_i) x_{ij} = 0 \quad \forall 0 \leq j \leq d$$

Stationary Points (3)

$$\frac{1}{n}\sum_{i=1}^{n} x_{ij}(\mathbf{x}_i^T\mathbf{w} - y_i) = 0 \quad \forall 0 \le j \le d$$

(recall that $\mathbf{x}_i = \begin{bmatrix} x_{i0} \\ x_{i1} \\ \vdots \\ x_{id} \end{bmatrix}$)

$$\implies \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i(\mathbf{x}_i^T\mathbf{w} - y_i) = \mathbf{0}$$

$$\implies \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i\mathbf{x}_i^T\mathbf{w} - \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i y_i = \mathbf{0}$$

$$\implies \underbrace{\boxed{\left(\frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i\mathbf{x}_i^T\right)}}_{\mathbf{A}}\mathbf{w} = \underbrace{\boxed{\frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i y_i}}_{\mathbf{b}}$$

$$\implies \mathbf{A}\mathbf{w} = \mathbf{b}$$

$$\implies \mathbf{w} = \mathbf{A}^{-1}\mathbf{b}$$ (if $\mathbf{A}$ is invertible)

Analytical Solution Drawback

In practice, we don't usually solve for $\mathbf{w}$ analytically (**why?**)

$$\mathbf{A} = \frac{1}{n}\underbrace{\sum_{i=1}^{n}}_{n \text{ times}} \underbrace{\boxed{\mathbf{x}_i\mathbf{x}_i^T}}_{\substack{(d+1)\times(d+1) \\ \text{element matrix}}}$$ costs $O(nd^2)$ operations to construct (**why?**)

Matrix inversion costs $O(d^3)$ operations.

So analytic solution costs $O(nd^2 + d^3)$.

Question: Why not just $O(d^3)$?

Linear Regression: Numerical Solution

First-order gradient descent update:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla c(\mathbf{w}_t)$$

- We can use **gradient descent** to find $\arg\min_{\mathbf{w}} c(\mathbf{w})$
- Each gradient descent update costs $O(nd)$:

$$\nabla c(\mathbf{w}) = \frac{1}{n} \underbrace{\sum_{i=1}^{n}}_{n \text{ times}} \mathbf{x}_i (\underbrace{\boxed{\mathbf{x}_i^T \mathbf{w}}}_{d+1 \text{ multiplications and additions}} - y_i)$$

- So k iterations of **gradient descent** costs $O(knd)$
  - For large d (or small n), this is a lot cheaper than $O(d^3 + nd^2)$
  - Can be very fast to find an approximate solution
  - Still very costly for **large** n

Linear Regression
Stochastic Gradient Descent

First-order gradient descent:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla c(\mathbf{w}_t) \quad \text{with} \quad \nabla c(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i (\mathbf{x}_i^T \mathbf{w} - y_i)$$

Stochastic gradient descent (SGD):
Do the gradient update using an estimate of the true batch gradient:

1. Uniformly sample a datapoint index i from $\{1, \ldots, n\}$
2. Do an estimated gradient step $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t g(i)$
   where $g(i) = \mathbf{x}_i (\mathbf{x}_i^T \mathbf{w} - y_i)$

Question: What is the time complexity of k iterations of SGD?

Unbiased Estimator of Gradient

$$\nabla c(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i(\mathbf{x}_i^T\mathbf{w} - y_i) \qquad I \sim \text{Unif}\{1,\ldots,n\}$$

$$\begin{aligned}\mathbb{E}[g(I)] &= \sum_{i=1}^{n} p(i)g(i) \;= \sum_{i=1}^{n} \frac{1}{n} g(i)\\ &= \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i(\mathbf{x}_i^T\mathbf{w} - y_i)\\ &= \nabla c(\mathbf{w}).\end{aligned}$$

$g(I)$ is an **unbiased estimator** of $\nabla c(\mathbf{w})$.

Linear Regression for Nonlinear Predictors

- Linear regression is useful for more than just linear models
- Can obtain nonlinear functions by **transforming** the observation vector before fitting
  - Then do linear regression on the transformed vector
- We write this as $\phi(\mathbf{x}) = \Big(\phi_0(\mathbf{x}), \ldots, \phi_p(\mathbf{x})\Big)$
  - Note that each ϕ_j takes the entire observation vector $\mathbf{x}$
  - p need not equal d
- **Question:** Have we seen an example of this already?

$$\phi\left(\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}\right) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}$$

1D Polynomial Regression

- In the one-dimensional case, OLS would learn

$$f(x) = w_0 + w_1 x$$

- We can do **polynomial regression** instead:

$$\mathbf{w} = \begin{bmatrix} w_0 \\ \vdots \\ w_p \end{bmatrix}$$

$$f(x) = \sum_{j=0}^{p} w_j x^j \quad = \sum_{j=0}^{p} w_j \phi_j(x) = \mathbf{w}^T \phi(\mathbf{x})$$

- But notice that this is just **linear regression** with a particular $\phi(x) = \left(x^0, x^1, \ldots, x^p\right)$!

$$\phi(\mathbf{x}) = \begin{bmatrix} \phi_0(\mathbf{x}) \\ \vdots \\ \phi_p(\mathbf{x}) \end{bmatrix}$$

- **Question:** Can a linear model learn anything that a polynomial model cannot?

Multivariate Polynomial Regression

- We can also do **polynomial regression** in the **multi-dimensional** case
- Example: For $\mathscr{X} = \mathbb{R}^2$ and $p = 2$:

$$\phi(\mathbf{x}) = \begin{bmatrix} \phi_0(\mathbf{x}) = 1.0 \\ \phi_1(\mathbf{x}) = x_1 \\ \phi_2(\mathbf{x}) = x_2 \\ \phi_3(\mathbf{x}) = x_1 x_2 \\ \phi_4(\mathbf{x}) = x_1^2 \\ \phi_4(\mathbf{x}) = x_2^2 \end{bmatrix}$$

Question:

What do we need to do differently to train **nonlinear models** (like polynomial regression) with **linear regression**?

Summary

A **linear predictor** has the form $f(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_d x_d = \sum_{j=0}^{d} w_j x_j = \mathbf{w}^T \mathbf{x}$

- **Linear regression** is the process of finding a vector $\mathbf{w}$ of weights that minimizes the expected cost of the prediction
- This can be solved **analytically** by solving a system of linear equations
  - But this can be very expensive for large d: $O(nd^2 + d^3)$
- More common solved **numerically** by **first-order gradient descent**
  - But this can also be very expensive for large n: $O(ndk)$ for k iterations
  - We can get around this using **stochastic gradient descent**
- Linear regression can be straightforwardly extended to **nonlinear regression**
  - Just do linear regression on a bunch of nonlinear features