Week 3

Lecture

- ERM
  - ML pipeline, loss function
  - Overfitting and generalization
  - Cross validation
- Linear Regression and Logistic Regression
  - From closed-form solution to gradient descent
- Optimization
  - GD, SGD and Minibatch SGD

Tutorial

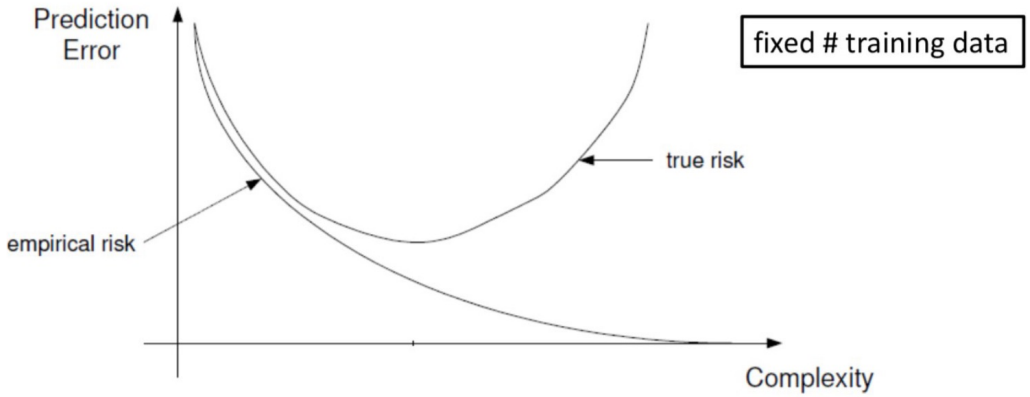
Task 1: Applying Linear Regression to predict house price. [Regression]
Task 2: Applying Logistic Regression to classify the generated synthetic data. [Classification]

Step 1: Load the training/test data.
Step 2: Preprocessing:
1. feature extraction
2. log-transform

Step 3: Closed-form solution vs. gradient descent

TODO:
1. Implement SGD and minibatch SGD
2. Evaluate RMSE on test data

Overfitting and generalization



A FIRST APPROACH

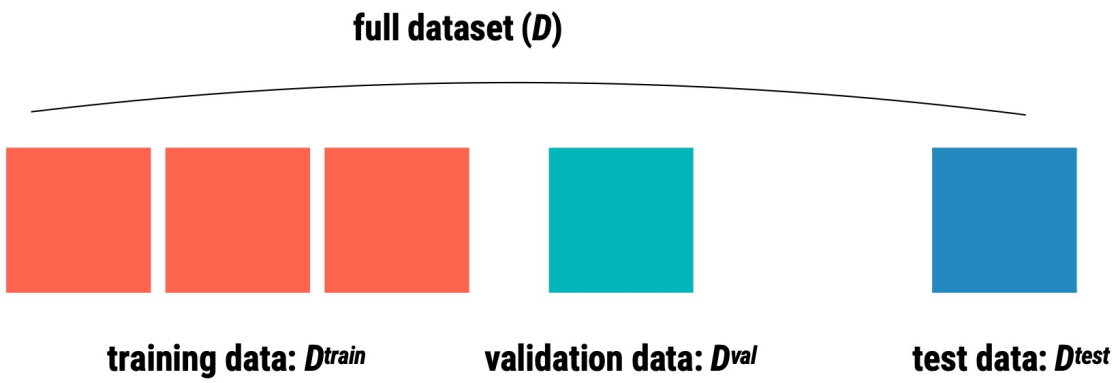
Tuning with a validation set

Training data: $D^{train} = \{(x_1, y_1), \ldots (x_n, y_n)\}$
- Used to learn the ERM predictor

Test data: $D^{test} = \{(x_1, y_1), \ldots (x_m, y_m)\}$
- Used to assess the performance of the ERM predictor

Validation data: $D^{val} = \{(x_1, y_1), \ldots (x_L, y_L)\}$
- Used to optimize hyperparameter(s)

training, validation, and test data should not overlap!



Cross Validation

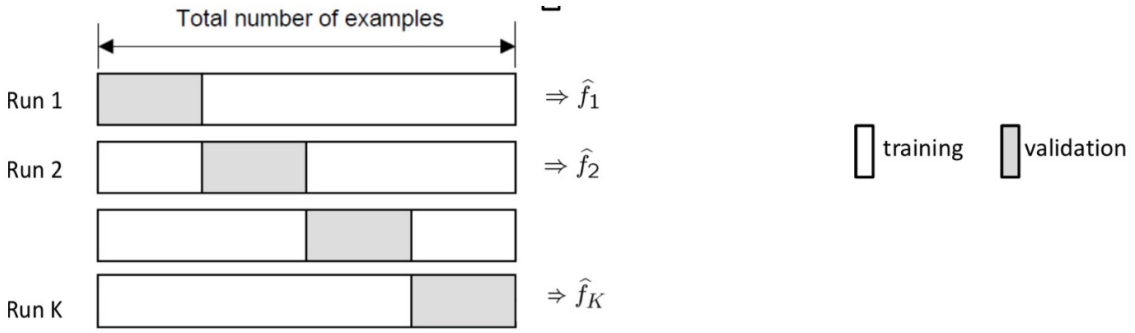
Cross-validation

Split the training data into K equal parts

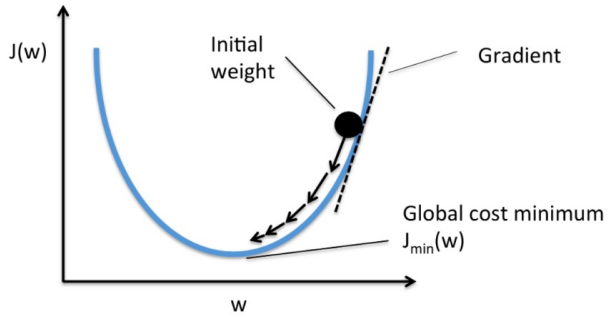
Use each part in turn as a validation dataset, with the remaining for training

Choose the hyperparameter such that the model performs the best (e.g., on average) across the K validation datasets

Special case: when K=n, leave-one-out (LOO) cross-validation



LR and GD



Cost function: Sum of Squared Errors

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

Step 1: Take partial derivative of the cost function with respect to each weight wj (gradient)

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

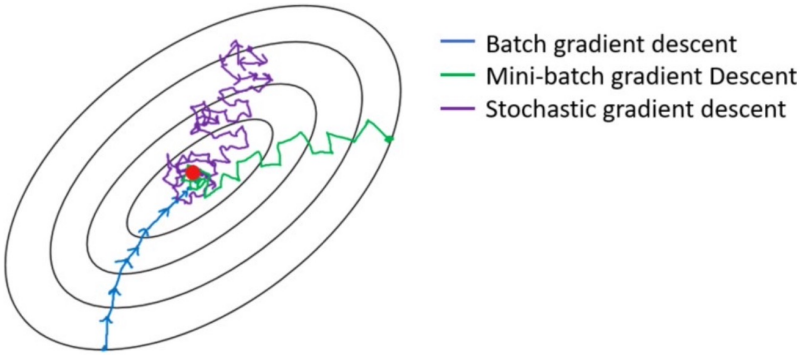
Step 2: Update weights by taking a step away from the gradient

$$w := w + \Delta w$$

Step 3: The weight change is defined as the negative gradient multiplied by the learning rate

$$\Delta w = -\eta \Delta J(w)$$

Gradient Descent



Objective we want to solve: $\min_{\mathrm{w}} f(\mathrm{w})$ where $f(\mathrm{w}) := \sum_{j=1}^{n} f_j(\mathrm{w})$

Gradient Descent Update: $\mathrm{w}_{i+1} = \mathrm{w}_i - \alpha_i \nabla f(\mathrm{w}_i)$

Stochastic Gradient Descent (SGD) Update: $\mathrm{w}_{i+1} = \mathrm{w}_i - \alpha_i \nabla f_j(\mathrm{w}_i)$

with j sampled at random

Mini-batch SGD Update: $\mathrm{w}_{i+1} = \mathrm{w}_i - \alpha_i \nabla f_{B_i}(\mathrm{w}_i)$

with mini-batch $\mathbf{B}_i \subseteq \{1,\dots n\}$ *sampled at random*

Mini-batch SGD

Pros

- In contrast to SGD: parallelizable
- Less computation than GD (might lead to faster overall convergence)
- Can tune computation vs. communication depending on batch size

Cons

- In terms of iterations: slower convergence than gradient descent
- Another parameter to tune (batch size)
- Still might be too much communication …

- Batch gradient descent:
  1. Go through the entire training set on every iteration. Cost long time on every iteration
  2. The cost function decreases on every single iteration

- Stochastic gradient descent:
  1. Cost function won't ever converge, it will always just kind of oscillate and wander around the region of the minimum. But it won't ever just head to the minimum and stay there.
  2. Lose almost all your speed up from vectorisation.

- Mini-batch gradient descent:
  1. Choose 1 < size < m can give you in practice the fastest learning. You do get a lot of vectorisation
  2. Cost function's decrease is in between of previous two.