

# Part II Exam Cheatsheets: Statistics and Machine Learning

Eric Ordoñez

April 2025

## 1 Multivariate statistics

**Calculus:** For a function  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ , the *gradient* of  $f$  is

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_d} \right)$$

The *Hessian* of  $f$  is the  $d \times d$  matrix

$$(H_f)_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}, \quad 1 \leq i, j \leq d$$

$f$  is convex (concave) if  $H_f$  is positive (negative) semi-definite. Strict convexity/concavity follows from strict definiteness.

Let  $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ . The gradient of  $f$  is the  $d \times k$  matrix

$$(\nabla f)_{ij} = \frac{\partial f_j}{\partial x_i}$$

whose transpose is called the *Jacobian*  $J_f$ .

**Convergence:** A sequence of multivariate r.v.s  $(X_n)$  in  $\mathbb{R}^d$  converges in probability to  $X$  if  $X_n^{(k)} \xrightarrow{\mathbb{P}} X^{(k)}$  for all  $k \in [d]$ .

**Covariance:** Given a random variable  $X = (X^{(1)}, X^{(2)}, \dots, X^{(d)})$ , the covariance matrix of  $X$  is defined  $(\Sigma_X)_{ij} = \text{Cov}(X^{(i)}, X^{(j)})$ .

Let  $\mu = \mathbb{E}[X]$  be the entry-wise mean of  $X$ . The covariance matrix is equivalently defined  $\Sigma_X = \mathbb{E}[(X - \mu)(X - \mu)^\top]$ .

Properties of multivariate covariance:  $\Sigma_{AX+B} = \Sigma_{AX} = A\Sigma_X A^\top$

**Gaussian distribution:** The  $d$ -dimensional Gaussian distribution of  $X$  is completely specified by its mean  $\mathbb{E}[X] = (\mathbb{E}[X^{(1)}], \dots, \mathbb{E}[X^{(d)}])$  and covariance matrix  $\Sigma$ . If  $\Sigma$  is invertible, then

$$f(x) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left(-\frac{1}{2}(x - \mu)^\top \Sigma^{-1}(x - \mu)\right)$$

**Central limit theorem:** A sequence  $(T_n)$  of r.v.s in  $\mathbb{R}^d$  converges in distribution to a random vector  $T \in \mathbb{R}^d$  if  $v^\top T_n \xrightarrow{d} v^\top T$  for all  $v \in \mathbb{R}^d$ .

**Delta method:** Let  $(T_n)$  be a sequence of random vectors such that  $\sqrt{n}(T_n - \theta) \xrightarrow{d} T$ . If  $g : \mathbb{R}^d \rightarrow \mathbb{R}^k$  is continuously differentiable at  $\theta$ , then

$$\sqrt{n}(g(T_n) - g(\theta)) \xrightarrow{d} \nabla g(\theta)^\top T = J_g T$$

## 2 Linear regression

### 2.1 Assumptions

Suppose that  $\{(X_i, Y_i)\}_{i \in [n]}$  are sampled i.i.d. from an unknown joint distribution on  $X$  and  $Y$ . The *regression function* of  $Y$  with respect to  $X$  is  $\mu(x) = \mathbb{E}[Y | X = x]$ . Linear regression assumes that  $\mu$  is linear in  $x$ .

From here on, assume that  $Y \in \mathbb{R}$ . In the univariate case  $X \in \mathbb{R}$ , the *theoretical linear regression* of  $Y$  on  $X$  is the line  $Y = a^* + b^* X$ , where

$$b^* = \frac{\text{Cov}(X, Y)}{\text{Var } X}, \quad a^* = -b^* \mathbb{E}[X]$$

minimize the square loss  $\sum_{i=1}^n |Y_i - aX_i - b|^2$ .

Assume  $\text{Var } X \neq 0$ ; otherwise, there is no line that predicts  $Y$  given  $X$  with probability 1, regardless of their distribution.

**Ordinary least squares estimator:** Generalizing the setup to  $\mu(X) = X\beta^*$ , we have  $\hat{\beta} = \text{argmin}_{\beta} \|Y - X\beta\|^2$ .

**Noise:** A r.v.  $\epsilon$  that satisfies  $Y = a^* + b^* X + \epsilon$  with  $\mathbb{E}[\epsilon] = 0$  and  $\text{Cov}(X, \epsilon) = 0$ . In the multivariate case,  $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$  for some known or unknown value of  $\sigma^2 > 0$ .

### 2.2 Properties of the OLS estimator

- Under the above assumptions,  $\hat{\beta}$  is also the MLE.
- Distribution:  $\hat{\beta} \sim \mathcal{N}(\beta^*, \sigma^2(X^\top X)^{-1})$

- Quadratic risk:  $\mathbb{E} [\|\hat{\beta} - \beta^*\|_2^2] = \sigma^2 \text{tr}((X^\top X)^{-1})$
- Prediction error:  $\mathbb{E} [\|Y - X\hat{\beta}\|_2^2] = \sigma^2(n - p)$
- An unbiased estimator of  $\sigma^2$  is  $\hat{\sigma}^2 = \|Y - X\hat{\beta}\|_2^2/(n - p)$ .
- $(n - p) \frac{\hat{\sigma}^2}{\sigma^2} \sim \chi_{n-p}^2$

Note that we can always perform linear regression and obtain an OLS estimate, no matter the true underlying distribution!

## 3 Linear classifiers

### 3.1 Perceptron algorithm

Data is *linearly separable* if there exists a hyperplane that can perfectly classify the data. The perceptron algorithm is guaranteed to work for linearly separable data, but it will never converge if the data is not separable.

```

initialize  $\theta = 0, \theta_0 = 0$ 
for  $i = 1, \dots, T$ 
  if  $y^{(i)}(\theta x^{(i)} + \theta_0) \leq 0$ 
     $\theta = \theta + y^{(i)}x^{(i)}$ 
     $\theta_0 = \theta_0 + y^{(i)}$ 

```

Run the algorithm  $T$  times or as many times needed for convergence, i.e., the perceptron makes no more misclassifications.

### 3.2 Maximum margin classifiers

A decision boundary is a hyperplane  $\theta x + \theta_0 = 0$  with margin equal to  $1/\|\theta\|$ . Define *agreement* by  $z = y(\theta x + \theta_0)$ . *Hinge loss* is defined

$$\text{Loss}_h(z) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{if } z < 1 \end{cases}$$

*Regularization* seeks a balance between separation (overfitting) and margin maximization (generalizability) by minimizing

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \text{Loss}(z^{(i)}; \theta, \theta_0) + \frac{\lambda}{2} \|\theta\|^2$$

where  $\lambda > 0$  is a hyperparameter.

### 3.3 Kernels

Let  $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$  be a feature mapping where  $p \gg d$ . A *kernel function* is a function  $K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $K(x, x') = \Phi(x) \cdot \Phi(x')$ .

Composition rules:

- $K(x, x') = 1$  is a kernel.
- Let  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  and  $K$  be a kernel. Then  $f(x)K(x, x')f(x')$  is also a kernel.
- If  $K_1, K_2$  are kernels, then so are  $K_1 + K_2$  and  $K_1 K_2$ .

**Kernel perceptron:** Data that is not linearly separable in one space might be in a higher-dimension feature space. In the perceptron algorithm, recall that  $\theta = \sum_{j=1}^n \alpha_j y^{(j)} \Phi(x^{(j)})$ , where  $\alpha_j$  is the number of times the perceptron has misclassified  $y^{(j)}$ . Alter these two things for the kernel perceptron algorithm:

- The condition to check is  $y^{(i)} \sum_{j=1}^n \alpha_j y^{(j)} K(x^j, x^i) \leq 0$ .
- The update rule is  $\alpha_j = \alpha_j + 1$ .

## 4 Gradient descent

### 4.1 Newton's method of minimization

Let  $f$  be a convex analytical function and  $w_0$  the initial guess for its minimizer. Newton's method updates our guess according to the law

$$w_{t+1} = w_t - (H_f(w_t))^{-1} \nabla f(w_t).$$

This will not work for all convex loss functions because a positive semi-definite Hessian is not necessarily invertible!

### 4.2 Quadratic minimization

Fix a step size  $\alpha > 0$  and approximate the Hessian by  $H_f(w_t) \approx \alpha^{-1}I$ . Applying the same logic from Newton's method, gradient descent updates  $w_t$  according to the law

$$w_{t+1} = w_t - \alpha \nabla f(w_t).$$

Step size is a hyperparameter, and choosing the right step size to achieve convergence is an important part of calibrating a model.

### 4.3 Stochastic gradient descent

Take the loss of  $w_t$  averaged over  $N$  uniformly chosen data points:

$$f(w_t) \approx \frac{1}{N} \sum_{i=1}^N f_i(w_t).$$

Use this to estimate the gradient and update  $w_t$  like in regular gradient descent:

$$w_{t+1} = w_t - \frac{\alpha}{N} \sum_{i=1}^N \nabla f_i(w_t).$$

## 5 Unsupervised learning

### 5.1 Clustering

*Clustering* attempts to partition a set  $\{x^{(i)}\}_{i \in [n]}$  of feature vectors in  $\mathbb{R}^d$  into  $K$  distinct clusters. It outputs a  $K$ -partition of the  $n$  indices and a representative of each set in the partition.

Define a measure of homogeneity within cluster assignments and compare the measure across clustering scenarios. We want to minimize the total cost of a scenario,  $\text{Cost}(C_1, \dots, C_K) = \sum_{j=1}^K \text{Cost}(C_j)$ , with some ways here to define cost:

- Greatest distance between any two points in a cluster, i.e., diameter.
- Average distance between points inside a cluster.
- Sum of the distances between the representative and all other points in a cluster.

***K*-means algorithm:** Given the number  $K$  of clusters to label:

```
randomly select  $z_1, \dots, z_K$ 
iterate
  assign each  $x^{(i)}$  to the closest  $z_j$  such that
```

$$\text{Cost}(z_1, \dots, z_K) = \sum_{i=1}^n \min_{j \in [K]} \|x^{(i)} - z_j\|^2.$$

given  $C_1, \dots, C_K$ , find the best representatives  $z_1, \dots, z_K$

$$z_j = \underset{z}{\operatorname{argmin}} \sum_{i \in C_j} \|x^{(i)} - z\|^2.$$

$z_j$  is the centroid of the  $j$ -th cluster (this means  $z_j$  need not be one of the points!). Each update of  $z_j$  depends only upon points in  $C_j$ , thus the algorithm's output is sensitive to its random initialization. Complexity:  $\mathcal{O}(ndK)$ .

***K*-medioids algorithm:** Proceed the same way as in the  $K$ -means algorithm, but ensure  $\{z_1, \dots, z_K\} \subseteq \{x^{(1)}, \dots, x^{(n)}\}$ . Complexity:  $\mathcal{O}(n^2dK)$ .

### 5.2 Mixture models

A *Gaussian mixture model (GMM)* is defined by the following parameters, collectively called  $\theta$ :

- $K$  mixture components;
- A  $d$ -dimensional Gaussian  $\mathcal{N}(\mu^{(j)}, \sigma_j^2)$  for every  $j \in [K]$ ;
- Mixture weights  $p_1, \dots, p_K$ .

The likelihood of a point  $x$  in a GMM is

$$p(x | \theta) = \sum_{j=1}^K p_j \mathcal{N}(\mu^{(j)}, \sigma_j^2)(x)$$

**Expectation maximization algorithm:** The EM algorithm finds a locally optimum solution  $\hat{\theta}$  to the GMM likelihood maximization problem.

- **E step:** Find the posterior probability that point  $x^{(i)}$  was generated by cluster  $j$ , for each  $i \in [n]$  and  $j \in [K]$ :

$$p(j | i) = \frac{p_j \mathcal{N}(\mu^{(j)}, \sigma_j^2)(x^{(i)})}{p(x^{(i)} | \theta)}$$

- **M step:** Maximize a proxy function  $\hat{\ell}$  of the log-likelihood over  $\theta$ :

$$\hat{\ell}(x^{(1)}, \dots, x^{(n)} | \theta) = \sum_{i=1}^n \sum_{j=1}^K p(j | i) \log \left( \frac{p(x^{(i)}, x^{(i)} \text{ generated by cluster } j | \theta)}{p(j | i)} \right)$$

Solve the first-order conditions of the M step for  $\hat{\theta}$ :

$$\begin{aligned} \widehat{\mu^{(j)}} &= \frac{\sum_{i=1}^n p(j | i) x^{(i)}}{\sum_{i=1}^n p(j | i)} \\ \widehat{p_j} &= \frac{1}{n} \sum_{i=1}^n p(j | i) \\ \widehat{\sigma_j^2} &= \frac{\sum_{i=1}^n p(j | i) \|x^{(i)} - \widehat{\mu^{(j)}}\|^2}{d \sum_{i=1}^n p(j | i)} \end{aligned}$$

Repeat until there is no change in the likelihood.

## 6 Reinforcement learning

### 6.1 Markov decision processes

A *Markov decision process* is a collection  $\langle S, A, T, R \rangle$ , where:

- $S$  and  $A$  are the state and action spaces, respectively;
- $T(s, a, s')$  is the transition probability of ending in state  $s'$  by taking action  $a$  from state  $s$ ;
- $R(s, a, s')$  is the state-specific reward for taking action  $a$ .

All transition probabilities are memoryless:  $\mathbb{P}(s_{t+1} | s_t, s_{t-1}, \dots, s_0) = \mathbb{P}(s_{t+1} | s_t)$ .

Optimal behavior maximizes the expectation of a *utility function*  $U$  that measures accumulated rewards. Suppose rewards depend only upon the current state. Then we maximize:

- *Finite horizon*: The sum of rewards after acting for a fixed number of steps  $n$ :  $U(s_0, s_1, \dots, s_n) = \sum_{k=0}^n R(s_k)$ .
- *Infinite horizon*: The infinite sum of rewards that are exponentially discounted by a factor  $\gamma$ :  $U(s_0, s_1, \dots, s_n) = \sum_{k=0}^{\infty} \gamma^k R(s_k)$ .

## 6.2 Policy and value functions

**Policy**: A function  $\pi : S \rightarrow A$  that assigns an action  $\pi(s)$  for any state  $s$ .

**Optimal policy**: A policy  $\pi^*$  that maximizes expected utility.

**Value function**: The expected reward  $V^*(s)$  from acting optimally at state  $s$ .

**Q-function**: The expected reward  $Q^*(s, a)$  from starting at state  $s$ , taking action  $a$ , then acting optimally afterwards.

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

**Bellman equation**:  $V^*(s) = \max_a Q^*(s, a)$

## 6.3 Iteration

**Value function iteration**: Updated value of state  $s$  after acting optimally for  $k$  steps is:

$$V_{k+1}^*(s) = \max_a \left\{ \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma V_k^*(s')) \right\}$$

Complexity:  $\mathcal{O}(|S|^2|A|)$

**Q-value iteration**: Updated Q-value of the state-action pair  $(s, a)$  for the  $k$ -th step is:

$$Q_{k+1}^*(s, a) = \sum_{s'} T(s, a, s') \left( R(s, a, s') + \gamma \max_{a'} Q_k^*(s', a') \right)$$

An  *$\epsilon$ -greedy* algorithm is a training algorithm that randomly samples a new action with probability  $\epsilon$  or chooses the best currently available option with probability  $1 - \epsilon$ .  $\epsilon$  should eventually decay during training.