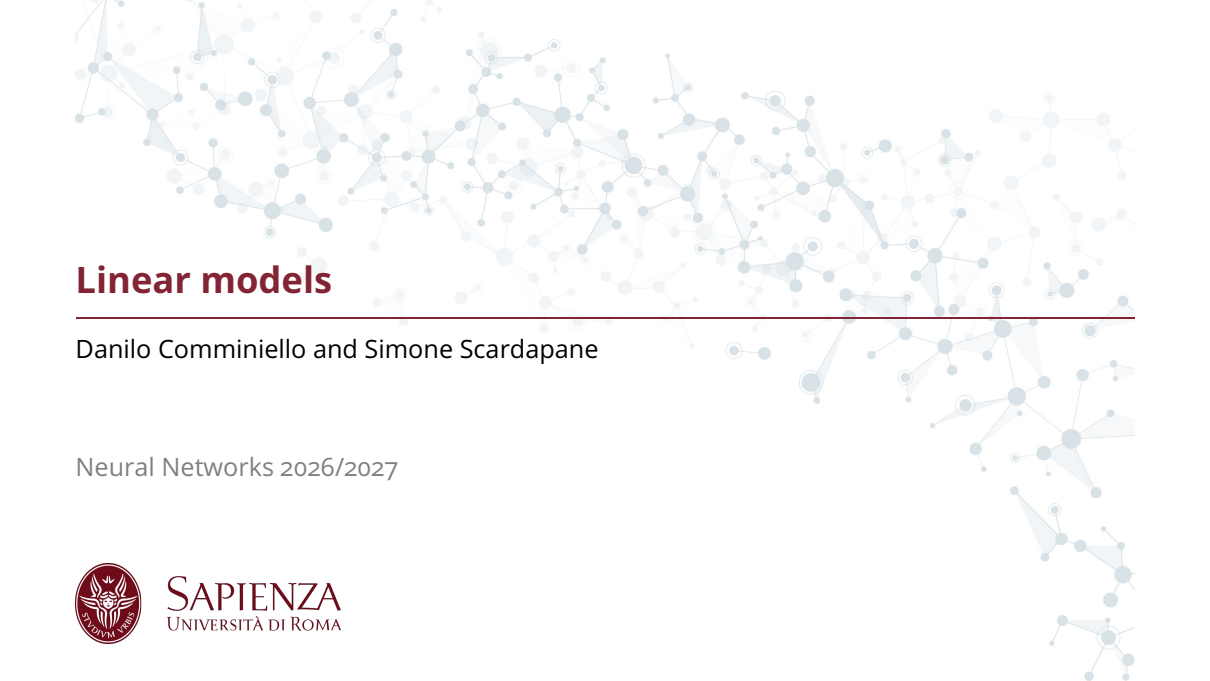




Linear models

Danilo Comminiello and Simone Scardapane

Neural Networks 2026/2027



SAPIENZA
UNIVERSITÀ DI ROMA

1 Supervised learning

Setup and data splits

Loss functions

Supervised learning

Dataset

A set of N input-target pairs:

$$\mathcal{S} = \{(x_i, y_i)\}_{i=1}^N . \quad (1)$$

Model

A map from inputs to predictions:

$$\hat{y} = f_{\theta}(x) . \quad (2)$$

We fit θ on $\mathcal{S}$ and evaluate the model on examples that were not used for training.

Training, validation, and test data

We split the available examples into three disjoint sets:

$$\mathcal{S} = \mathcal{S}_{\text{train}} \dot{\cup} \mathcal{S}_{\text{val}} \dot{\cup} \mathcal{S}_{\text{test}} . \quad (3)$$

Training

Fit parameters θ .

Validation

Model selection.

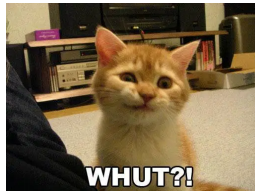
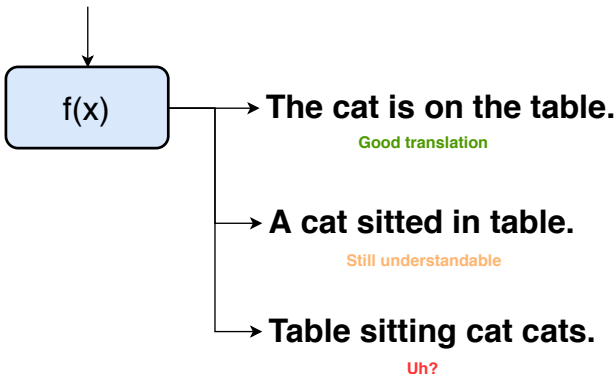
Test

Final evaluation.

We usually assume i.i.d. samples from the same unknown distribution $p(x, y)$. If the deployment distribution differs, we have a **distribution shift**.

What is a good approximation?

"Il gatto è sul tavolo."



Introducing loss functions

A **loss function** assigns a scalar cost to a prediction–target pair:

$$\ell(\hat{y}, y) \in \mathbb{R}, \quad \hat{y} = f_{\theta}(x). \quad (4)$$

A smaller loss means a better prediction. Learning amounts to choosing θ so that predictions have low loss on the training data.

For gradient-based learning, we usually choose losses that are **differentiable** with respect to the model parameters.

Expected risk and empirical risk

What we care about

The loss over the data distribution:

$$\mathcal{R}(\theta) = \mathbb{E}_{p(x,y)}[\ell(f_{\theta}(x), y)] . \quad (5)$$

This is the **expected risk**, but $p(x, y)$ is unknown.

What we can optimize

The average training loss:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f_{\theta}(x_i), y_i) . \quad (6)$$

This is the **empirical risk**; training computes $\theta^* = \arg \min_{\theta} \mathcal{L}(\theta)$.

A small empirical risk does not guarantee a small expected risk. Their difference is the **generalization gap**.

Overfitting

Consider a model that memorizes the training set:

$$f(x) = \begin{cases} y_i & \text{if } x = x_i \text{ for some } (x_i, y_i) \in \mathcal{S}_{\text{train}}, \\ \bar{y} & \text{otherwise.} \end{cases} \quad (7)$$

It predicts every training target exactly, and returns the same default value $\bar{y}$ for a new input.

Zero training risk does not imply small expected risk:

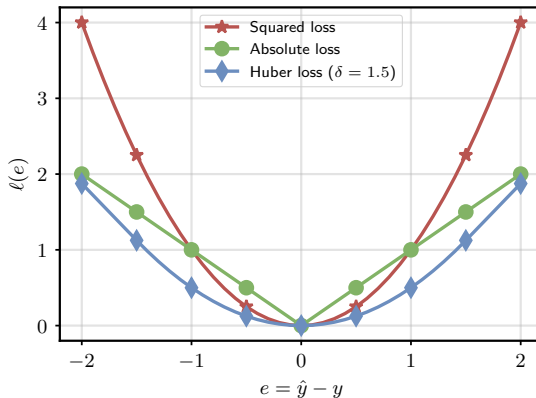
$$\mathcal{L}(\theta) = 0 \quad \nRightarrow \quad \mathcal{R}(\theta) \text{ is small.}$$

This is **overfitting**: fitting the training data without learning a rule that generalizes.

Losses for regression

For **regression** we have $y, \hat{y} \in \mathbb{R}$, and we can write the prediction error as $e = \hat{y} - y$.

- **Squared loss:** $\ell(e) = e^2$. Smooth, but gives large errors more weight.
- **Absolute loss:** $\ell(e) = |e|$. Less sensitive to outliers, but not differentiable at $e = 0$.
- **Huber loss:** quadratic near zero and linear for large $|e|$.



Maximum likelihood gives squared loss under Gaussian noise and absolute loss under Laplace noise, see the book.

2 Linear models

Linear models for regression

Linear models for classification

Fixed nonlinear features

A linear model

For an input $\mathbf{x} \in \mathbb{R}^D$, a scalar linear model is defined as:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{w}^{\top} \mathbf{x} + b = \sum_{j=1}^D w_j x_j + b. \quad (8)$$

The parameters are $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$, with $\boldsymbol{\theta} = \{\mathbf{w}, b\}$. The scalar b is the **bias**; we will sometimes omit it to simplify the notation. We call each component x_j a **feature**, and datasets of this form **tabular**.

Least squares

With the bias omitted, we collect all N training examples as rows of a matrix $\mathbf{X}$ and the targets as a vector $\mathbf{y}$:

$$\mathbf{X} \in \mathbb{R}^{N \times D}, \quad \mathbf{y} \in \mathbb{R}^N, \quad X_{ij} = x_{ij}. \quad (9)$$

A linear model with squared loss gives the **least-squares** objective:

$$\mathcal{L}_{\text{LS}}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 = \frac{1}{N} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2. \quad (10)$$

For gradient descent we have $\nabla \mathcal{L}_{\text{LS}}(\mathbf{w}) = \frac{2}{N} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$.

Solving least squares

Normal equations

For least squares, setting $\nabla \mathcal{L}_{LS}(\mathbf{w}) = \mathbf{0}$ gives a linear system:

$$\mathbf{X}^\top \mathbf{X} \mathbf{w} = \mathbf{X}^\top \mathbf{y}. \quad (11)$$

Closed-form solution

If $\mathbf{X}$ has full column rank (i.e. $\mathbf{X}^\top \mathbf{X}$ is invertible):

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (12)$$

Solving the $D \times D$ system directly costs roughly $\mathcal{O}(D^3)$. Gradient methods avoid the explicit solution, scale to larger problems, and still work when no closed-form solution is available.

Regularizing least squares

With strongly correlated features, the fitted weights can be unstable.

Regularization modifies the training objective to favour solutions with useful properties.

Ridge regression is a common type of regularization that favours smaller weights:

$$\mathcal{L}_{\text{ridge}}(\mathbf{w}) = \frac{1}{N} \left(\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda \|\mathbf{w}\|^2 \right). \quad (13)$$

At $\lambda = 0$ this is ordinary least squares. Larger values produce smaller, more stable weights at the cost of training fit. Since the penalty acts on the coefficients, features should be on comparable scales.

The normal equations and their solution become:

$$(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) \mathbf{w}^* = \mathbf{X}^\top \mathbf{y}, \quad \mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (14)$$

Linear models for classification

In classification, each target $y_i \in \{1, \dots, C\}$ identifies one of C classes. These numbers are labels, not quantities: class 2 does not mean twice class 1.

A linear classifier produces one score for each class:

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b} \in \mathbb{R}^C, \quad \mathbf{W} \in \mathbb{R}^{C \times D}, \quad \mathbf{b} \in \mathbb{R}^C. \quad (15)$$

The components of $\mathbf{z}_i$ are called the **logits**.

The largest logit gives the predicted class. Since $\arg \max$ is not differentiable, training replaces the $\arg \max$ with a continuous approximation (**softmax**) that turns the logits into probabilities.

From logits to probabilities

The **softmax** maps the logits to a vector $\mathbf{p}_i \in \mathbb{R}^C$:

$$p_{ik} = \text{softmax}(\mathbf{z}_i)_k = \frac{\exp(z_{ik})}{\sum_{j=1}^C \exp(z_{ij})} . \quad (16)$$

Every component is non-negative and the components sum to one:

$$\mathbf{p}_i \in \Delta_C = \left\{ \mathbf{p} \in \mathbb{R}^C : p_k \geq 0, \sum_{k=1}^C p_k = 1 \right\} . \quad (17)$$

We can therefore read p_{ik} as the predicted probability of class k . Softmax preserves the ordering of the logits, so $\arg \max_k p_{ik} = \arg \max_k z_{ik}$.

Softmax and temperature

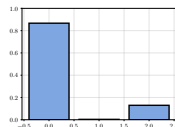
The **temperature** $\tau > 0$ controls the sharpness of the distribution:

$$\text{softmax}(\mathbf{z}/\tau)_k = \frac{\exp(z_k/\tau)}{\sum_{j=1}^C \exp(z_j/\tau)} . \quad (18)$$

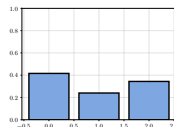
Smaller values produce sharper distributions; larger values produce flatter ones. The same temperature controls the sharpness of token distributions in language models.



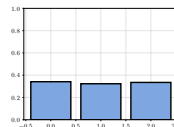
(a) Logits



(b) $\tau = 1$



(c) $\tau = 10$



(d) $\tau = 100$

Multiclass logistic regression

Let $\mathbf{p}_i = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$, with $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$. The **cross-entropy loss** is the negative log-probability assigned to the correct class:

$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = -\log p_{i, y_i} . \quad (19)$$

With one-hot targets, this is the usual cross-entropy between two vectors.

Training minimizes the empirical risk:

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) . \quad (20)$$

There is no closed-form solution. The equivalent maximum-likelihood interpretation is discussed in the book.

Binary classification

For two classes, with $y_i \in \{0, 1\}$, one scalar logit is enough:

$$s_i = \mathbf{w}^\top \mathbf{x}_i + b, \quad p_i = \sigma(s_i) = \frac{1}{1 + \exp(-s_i)}. \quad (21)$$

Here p_i is the probability of class 1, while $1 - p_i$ is the probability of class 0.

The cross-entropy becomes the **binary cross-entropy**:

$$\ell(p_i, y_i) = -y_i \log p_i - (1 - y_i) \log(1 - p_i). \quad (22)$$

The most probable class is 1 whenever $p_i \geq 0.5$, equivalently whenever $s_i \geq 0$.

What makes the classifier linear?

Binary case

At threshold 0.5, the predicted class changes when the scalar logit is zero:

$$\mathbf{w}^\top \mathbf{x} + b = 0. \quad (23)$$

This hyperplane is normal to $\mathbf{w}$.

Multiclass case

The boundary between classes k and j is where their logits are equal:

$$(\mathbf{w}_k - \mathbf{w}_j)^\top \mathbf{x} + (b_k - b_j) = 0. \quad (24)$$

This is again a hyperplane.

Softmax and sigmoid are nonlinear, but the resulting decision boundaries are linear in the input.

Visualizing the sigmoid function

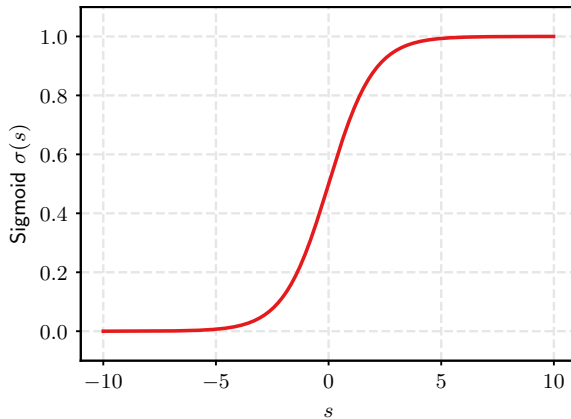


Figure 1: The sigmoid approaches 0 and 1 asymptotically.

Cross-entropy from logits

Computing softmax first and then taking its logarithm can be numerically unstable.

Substituting the softmax definition gives the equivalent logits-based loss:

$$\ell(\mathbf{z}_i, y_i) = -z_{i,y_i} + \text{logsumexp}(\mathbf{z}_i). \quad (25)$$

Writing $m_i = \max_k z_{ik}$, we can evaluate the second term safely as:

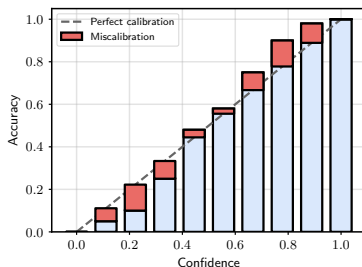
$$\text{logsumexp}(\mathbf{z}_i) = m_i + \log \sum_{k=1}^C \exp(z_{ik} - m_i). \quad (26)$$

Use logits-based framework losses; apply softmax only when probabilities are needed.

Can we trust these probabilities?

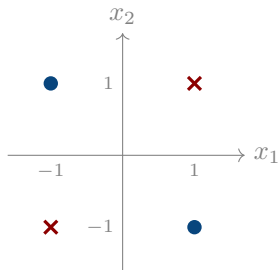
Softmax scores are not automatically calibrated probabilities.

A classifier is **calibrated** if predictions made with confidence q are correct about a fraction q of the time.



This is a **reliability plot**, see the book for more details.

Linear in what?



XOR: no single straight line separates the two classes.

Suppose we add the interaction x_1x_2 as a feature:

$$f_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2.$$

This model is **nonlinear in $\mathbf{x}$** , but **linear in $\mathbf{w}$** .

For example, $\mathbf{w} = (0, 0, 0, -1)^\top$ gives

$$f_{\mathbf{w}}(\mathbf{x}) = -x_1x_2 = \begin{cases} 1 & \text{blue circles,} \\ -1 & \text{red crosses.} \end{cases}$$

Thresholding this score at zero solves XOR.

A linear model on transformed inputs can produce a nonlinear decision boundary in the original input space.

Fixed nonlinear features

Generalizing this, we can fix a nonlinear feature map ϕ and train only the coefficients:

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

However, **polynomial features grow quickly**. For d inputs, degree-two terms alone give $d(d+1)/2$ features. Higher degrees can make explicit enumeration infeasible.

Random Fourier features

$$\phi_j(\mathbf{x}) = \sqrt{2/m} \cos(\mathbf{a}_j^\top \mathbf{x} + b_j).$$

Sample m frequencies and phases once, then keep them fixed.

Kernel basis functions

$$\phi_j(\mathbf{x}) = k(\mathbf{x}, \mathbf{c}_j).$$

Similarities to fixed centers $\mathbf{c}_j$, e.g., with a Gaussian kernel.

Next lecture: can we learn the features themselves?

Random Fourier features: Rahimi and Recht (2007), Random Features for Large-Scale Kernel Machines.

- **Chapters 3 and 4** of the book, along with the suggested exercises in Chapter 4.