Machine Learning

2° BEMACS

Written by
Giorgio Micaletto

Contents

Rules

Office hours: usemotion.com/meet/andrea-celli/ML30412

Exam:

- General exam (out of 31)
- Project can be taken for a maximum of 16 points
- Project can be submitted only once
- Project grades will be kept up until September

Introduction

In a traditional programming setting, the computer is fed an input and a function and returns an output; within Machine Learning, the computer is given the data and the output and is asked to return the function that best maps the inputs to outputs. Machine Learning can extract information from data, but NOT create it.

There are two types of "learning" from data:

Supervised Learning

The algorithm is given labelled input (for example, ECG reading and "heart attack"), where the labelling is done by experts and the algorithm tries to mimic the labelling. More formally:

Given training dataset $D = (x, t)$ where x are the input, t is the target and an unknown function f the goal is to find an approximation of f that generalizes well on test data.

To do so, a loss function L must be defined, an hypothesis space H must be defined and an optimization must be done to find an approximate model h

Unsupervised Learning

In unsupervised learning the core idea is to learn a more efficient representation of a set of unknown inputs.

1 Lecture 1: Loss Function and Model Performance

1.1 Modeling Relationship Between Features and Target

Consider a feature matrix $X = [X_1, \ldots, X_n]$ and a target variable y. The relationship between them can be expressed as:

$$y = f(X) + \varepsilon \tag{1}$$

where ε represents a random error term that is independent of X and has a mean of zero.

If we assume a linear relationship, the model function f is defined as:

$$f(X) = \beta_0 + \sum_{i=1} \beta_i X_i \tag{2}$$

Given the infinite possibilities for β values, the optimal set is chosen by defining a loss function and evaluating the performance of each candidate model.

1.2 Loss Functions

A commonly used loss function in regression problems is the Mean Squared Error (MSE), defined as:

$$MSE = \frac{1}{n} \sum^{n}_{i=1} \left(y_i - \hat{f}(X_i) \right)^2 \tag{3}$$

where $\hat{f}$ represents the model's estimation of the function f. The optimal model minimizes the MSE on a set T of test observations:

$$\min_{\text{model} \in H} \frac{1}{|\mathrm{T}|} \sum_{(x,y)} \left(y - \hat{f}(x) \right)^2 \tag{4}$$

In classification problems, Categorical Log Loss is widely used, where:

$$L = - \sum_{i=1} y_i \log(p_i) \tag{5}$$

Here, L denotes the loss for a single example, C is the number of classes, y_i is a binary indicator of whether class label i is the correct classification, and p_i is the predicted probability of the observation belonging to class i.

1.3 Model Performance Issues

Underfitting occurs when a model is too simplistic, capturing insufficient patterns from the data:

- Poor performance on training data.
- The model is overly simplified.
- Low variance in predictions.

This is typically due to a lack of sufficient parameters to account for the complexity of the data, leading to high bias and low variance.

Conversely, overfitting happens when a model is excessively complex, capturing noise as patterns:

- High performance on training data but poor generalization to new data.
- The model captures noise as if it were signal.
- High variance in predictions.

This typically occurs in models with too many parameters, resulting in low bias but high variance.

2 Lecture 2: Optimization

2.1 Unconstrained Optimization

Optimizing a machine learning model typically involves identifying an optimal set of parameters. This process is guided by an objective function, assumed to be differentiable and designed for minimization. The primary goal is to solve:

$$\min_{x} f(\mathbf{x}) \tag{6}$$

where $f : \mathrm{R}^d \to \mathrm{R}$ represents the objective function.

2.1.1 Gradient Descent

To minimize said function, we use gradient descent, a first-order optimization algorithm. It seeks a local minimum by updating parameters in the direction opposite to the gradient of the objective function. Given a suitable small step-size $\eta \geq 0$, the update rule is:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \, (\nabla f(\mathbf{x}_{t-1}))^T \tag{7}$$

This method converges to a local minimum, although it may slow near the minimum due to its suboptimal asymptotic convergence rate.

The choice of step-size η is critical in gradient descent:

- A large step-size can cause the algorithm to overshoot the minimum.
- A small step-size can render the convergence time impractically long.

Adaptive methods get away with this by adjusting the step-size dynamically by:

- Increasing the step-size when the function value decreases after a step.
- Decreasing the step-size and revert the last update when the function value increases, ensuring monotonic convergence.

When utilizing gradient descent, thus solving systems such as $A\mathbf{x} = \mathbf{b}$, convergence can be slow if the matrix A is poorly conditioned. The convergence rate is influenced by:

$$\kappa = \frac{\max \sigma(A)}{\min \sigma(A)} \tag{8}$$

where κ is the condition number. For ill-conditioned systems, applying a preconditioner P that simplifies $P^{-1}(A\mathbf{x} - \mathbf{b}) = 0$ can enhance performance.

Another variant of gradient descent is gradient descent with momentum. This method incorporates a momentum term to stabilize updates:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \, (\nabla f(\mathbf{x}_{t-1}))^T + \alpha \Delta \mathbf{x}_{t-1} \tag{9}$$

where $\Delta \mathbf{x}_{t-1} = \mathbf{x}_{t-1} - \mathbf{x}_{t-2}$ and α is the momentum coefficient, which lies between 0 and 1.

2.1.2 Stochastic Gradient Descent (SGD)

For large datasets, computing the exact gradient becomes computationally prohibitive. SGD approximates the gradient using a random subset of data, thus efficiently estimating the gradient by:

$$L(\boldsymbol{\theta}) = \sum_{i=1} L_n(\vartheta_i) \tag{10}$$

This approach leverages the fact that a mini-batch gradient is an unbiased estimate of the full dataset gradient, making it a practical choice for large-scale optimization.

2.2 Constrained Optimization

In contrast to the unconstrained optimization problems discussed earlier, constrained optimization introduces limits or conditions that the solution must satisfy. The general form of such a problem is:

$$\min_{x} f(\mathbf{x})$$
$$\text{subject to } g_i(\mathbf{x}) \leq 0 \quad \forall i = 1, \ldots, m$$

We will focus on the convex case, although f and g_i may be non-convex in other contexts.

To handle the constraints, we use Lagrange multipliers, introducing a multiplier $\lambda_i \geq 0$ for each constraint. This leads to the Lagrangian function:

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1} \lambda_i g_i(\mathbf{x}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{g}(\mathbf{x}) \tag{11}$$

where $\mathbf{g}(\mathbf{x})$ is a vector of all constraint functions $g_i(\mathbf{x})$.

Associated with the original problem is its dual problem, which is formulated as:

$$\max_{\boldsymbol{\lambda} \in \mathrm{R}^m} D(\boldsymbol{\lambda})$$
$$\text{subject to } \boldsymbol{\lambda} \geq 0$$

The dual function $D(\boldsymbol{\lambda})$ is defined as:

$$D(\boldsymbol{\lambda}) = \min_{\mathbf{x} \in \mathrm{R}^d} L(\mathbf{x}, \boldsymbol{\lambda}) \tag{12}$$

which turns each specific $\boldsymbol{\lambda}$ into an unconstrained optimization problem.

The minimax inequality relevant to duality states:

$$\max_{\mathbf{y}} \min_{\mathbf{x}} \phi(\mathbf{x}, \mathbf{y}) \leq \min_{\mathbf{x}} \max_{\mathbf{y}} \phi(\mathbf{x}, \mathbf{y}) \tag{13}$$

This inequality allows us to understand the dual problem as:

$$\max_{\boldsymbol{\lambda} \geq 0} \min_{\mathbf{x} \in \mathrm{R}^d} L(\mathbf{x}, \boldsymbol{\lambda}) \tag{14}$$

It's important to note that although the functions $f(\cdot)$ and $g_i(\cdot)$ might be non-convex, the dual function $D(\boldsymbol{\lambda})$ is concave. This is due to the linear (affine) nature of $L(\mathbf{x}, \boldsymbol{\lambda})$ in terms of $\boldsymbol{\lambda}$, ensuring that the maximization over $\boldsymbol{\lambda}$ involves a concave function.

An equality constraint, such as $h_j(\mathbf{x}) = k$, can be represented using two inequality constraints:

- $h_{j1}(\mathbf{x}) \geq k$
- $h_{j2}(\mathbf{x}) \leq k$

This dual formulation allows the use of Lagrange multipliers for equalities similarly to inequalities.

3 Lecture 3: Ordinary Least Squares

Regression is one of the two fundamental tasks of supervised learning. We will now introduce the linear regression model, which is (historically) the most used model. In a more mathematical framework, regression is about learning a model f such that

$$\min_{|\varepsilon|} \mathbf{y} = f(X) + \boldsymbol{\varepsilon} \tag{15}$$

Where $\mathbf{y}$ is the vector of outputs of dimension n, and X is our feature matrix of dimension $n \times d$ and $\boldsymbol{\varepsilon}$ is a random vector (noise) with mean 0 and is independent of X.

The linear regression model assumes that $\mathbf{y}$ can be described as a combination of the d input columns $X_1, \ldots, x_d$, in other words,

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{16}$$

where $\boldsymbol{\beta}$ is the vector of parameters of the model.

The goal in supervised machine learning is making predictions $\hat{f}(X^*)$, where X^* is data previously unseen and $\hat{f}$ is the learned values for $\boldsymbol{\beta}$. Since we assume that $\boldsymbol{\varepsilon}$ is random with zero mean and independent, it makes sense to replace it with 0 in the prediction, that is, in a prediction $\hat{\mathbf{y}}$ is equivalent to

$$\hat{\mathbf{y}} = \hat{f}(X^*) = X^*\hat{\boldsymbol{\beta}} \tag{17}$$

To find the minimum amount of errors, we have to define a loss function. For regression error usually we use the Mean Squared Error (MSE), which is defined as

$$L(\hat{\boldsymbol{\beta}}) = \frac{1}{n} \left\| X\hat{\boldsymbol{\beta}} - \mathbf{y} \right\|^2 = \frac{1}{n} ||\boldsymbol{\varepsilon}||^2 \tag{18}$$

From the perspective of linear algebra, the challenge of linear regression can be viewed as finding the vector in the column space of X that is closest to $\mathbf{y}$ in the Euclidean sense. This task is accomplished through the orthogonal projection of $\mathbf{y}$ onto the column space of X. The parameters $\hat{\boldsymbol{\beta}}$ that achieve this are determined by solving the equation:

$$X^T X\hat{\boldsymbol{\beta}} = X^T\mathbf{y} \tag{19}$$

This equation is known as the normal equation. Here, $X^T X\hat{\boldsymbol{\beta}}$ represents the projection of $\mathbf{y}$ onto the column space of X, ensuring that the difference between $\mathbf{y}$ and the linear combination of columns of X (i.e., the residuals) is orthogonal to the column space of X, which results in the least squared error between $\mathbf{y}$ and the predicted values. If $X^T X$ is invertible, then $\hat{\boldsymbol{\beta}}$ has the closed form expression

$$\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \mathbf{y} \tag{20}$$

The fact that this closed-form solution exists is the reason for why the linear regression is so common, as other loss functions lead to optimization problems that lack this type of closed-form solution.

4 Lecture 4: Maximum Likelihood, Categorical Data

4.1 Maximum Likelihood

To get another perspective on the least squared error, we will now redefine our initial problem as

$$\max_{\boldsymbol{\beta}} p(\mathbf{y} \mid X; \boldsymbol{\beta}) \tag{21}$$

Here $p(\mathbf{y} \mid X; \boldsymbol{\beta})$ is the probability density of all observed outputs $\mathbf{y}$ in the training data given X and parameters $\boldsymbol{\beta}$, however we need to make another assumption about the distribution of $\mathbf{y}$. A common assumption is $\boldsymbol{\varepsilon} \sim N(\mathbf{0}, \sigma_\varepsilon^2 I)$, which is equivalent to say that the errors follow a normal distribution with mean zero. This in turns means that

$$p(\mathbf{y} \mid X; \boldsymbol{\beta}) = N(\mathbf{y}; X\boldsymbol{\beta}, \sigma_\varepsilon^2 I) \tag{22}$$

Where $N(\mathbf{y}; X\boldsymbol{\beta}, \sigma_\varepsilon^2 I)$ is the probability of $\mathbf{y}$ under the model $N(X\boldsymbol{\beta}, \sigma_\varepsilon^2 I)$. If we drop within the normal distribution the part that doesn't depend on $\boldsymbol{\beta}$ we have that our original problem becomes

$$\hat{\boldsymbol{\beta}} = \arg\max_{\boldsymbol{\beta}} \frac{1}{n} \cdot -||X\boldsymbol{\beta} - \mathbf{y}||^2 = \arg\min_{\boldsymbol{\beta}} \frac{1}{n} \cdot ||X\boldsymbol{\beta} - \mathbf{y}||^2 \tag{23}$$

Which is exactly the same as the least square error function.

4.2 Categorical Data

Categorical inputs are variables that represent categories or groups, such as gender or color, and must be encoded numerically to be used in regression analysis. If a categorical column i in the feature matrix X has only two categories, it can be

encoded using binary or dummy variables. This encoding assigns one category a value of 0 and the other a value of 1. For example, if the column represents gender, males might be coded as 0 and females as 1.

However, if the categorical variable can take on more than two values, the encoding becomes slightly more complex. This scenario requires one-hot encoding. In one-hot encoding, each category of the variable is transformed into a new binary column, ensuring that for each instance, only one of these columns contains a 1 (indicating the presence of that category), while all others contain 0. Specifically, if the i-th column of X can take k different categories, the column is expanded into an $n \times k$ matrix. Each row l in this matrix contains a 1 in the column corresponding to the category of the original input and 0s elsewhere.

For instance, if a color variable includes three categories—red, green, and blue—three new columns are created: one for each color. An observation with the color green would be represented as $(0, 1, 0)$, indicating that the middle column (green) is the observed category.

5 Lecture 5: Polynomial Regression, Bias, Variance

5.1 Polynomial Regression

Linear regression might appear to be rigid due to its reliance on a straight line model. To increase flexibility, we can employ polynomial regression by transforming the input variables into higher-degree terms. Mathematically, polynomial regression can be expressed as:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \varepsilon \tag{24}$$

Here, the inclusion of $x^2, x^3, \ldots$ allows the model to fit a wider range of data shapes, thus increasing model complexity. However, adding more terms also raises the risk of overfitting, as the model's complexity increases.

One effective method to control overfitting in polynomial regression is through regularization. By introducing a penalty term, $\lambda \|\boldsymbol{\beta}\|^2$ (known as L2 regularization), we alter the optimization problem to:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \quad \frac{1}{n} \cdot ||X\boldsymbol{\beta} - \mathbf{y}||^2 + \lambda \, ||\boldsymbol{\beta}||^2 \tag{25}$$

Selecting an appropriate value for λ is crucial; too small a value has minimal impact, whereas too large a value drives all coefficients towards zero, overly simplifying the model.

5.2 Understanding Bias and Variance

When training models, it is essential to consider both bias and variance, which are defined as:

$$\text{Bias: } \bar{y} - y \tag{26}$$

$$\text{Variance: } \text{Var}(\bar{y} - y) \tag{27}$$

Here, $\bar{y}$ is the prediction of our model and y is the actual output. Bias measures the error introduced by approximating a real-world problem with a simplified model, while variance measures how much the model's predictions vary between different training sets.

5.3 The Generalization Gap and Model Complexity

Model complexity can lead to discrepancies between performance on training data and unseen testing data, often referred to as the *generalization gap*. In supervised learning, particularly in regression, this gap can be quantified through the Mean Squared Error (MSE) on new data:

$$\text{MSE Test} = \mathrm{E}^* \; \mathrm{E}_{\text{TD}} \left[\hat{f}(X^*) - f_0(X^*) - \varepsilon \right]^2 \tag{28}$$

Where E^* represents the average error across all possible testing data sets X^*. This can further be broken down into:

$$\text{MSE Test} = \underbrace{\mathrm{E}^* \left[\hat{f}(X^*) - f_0(X^*) \right]^2}_{\text{Bias}^2} + \underbrace{\mathrm{E}^* \; \mathrm{E}_{\text{TD}} \left[\hat{f}(X^*) - \bar{f}(X^*) \right]^2}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible error}}$$

High model complexity typically results in low bias but high variance, indicating a model that fits the training data well but may not generalize effectively. Conversely, a model with low complexity might not capture the underlying patterns adequately, leading to high bias. Balancing these factors is key to minimizing the generalization gap and improving the robustness of model predictions.

6 Lecture 6: Regularization and Cross-Validation

6.1 L2 Regularization

L2 regularization, also known as Ridge, plays a crucial role in controlling the complexity of the model to prevent overfitting. Ridge regularization shrinks the coefficients

towards zero but typically does not set them exactly to zero, regardless of the value of λ, unless it is infinitely large. This characteristic ensures that all features are included but with reduced influence, making Ridge less effective for variable selection compared to Lasso.

The optimization problem for Ridge regularization within a linear regression framework is given by:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \quad \frac{1}{n} \cdot ||X\boldsymbol{\beta} - \mathbf{y}||^2 + \lambda \, ||\boldsymbol{\beta}||_2^2 \tag{29}$$

where $||\boldsymbol{\beta}||_2^2$ denotes the L2 norm of $\boldsymbol{\beta}$, which is the sum of the squares of the coefficients.

Ridge regularization affects all coefficients by applying a penalty proportional to the square of the magnitude of coefficients. This squared term ensures that the penalty increases significantly as the coefficients grow, making it very effective at controlling large values of coefficients, thus leading to more stable and generalizable models. The L2 norm, visualized as spherical contours in parameter space, ensures that the solution to the optimization problem lies within these spherical bounds, gently pulling all coefficients towards zero as λ increases, but never exactly setting any to zero.

This form of regularization is especially beneficial in situations where model prediction is more critical than interpretation, as it includes all variables but regulates their impact through shrinkage.

6.2 L1 Regularization

As we previously explored, incorporating L2 regularization, also known as Ridge, reduces the risk of overfitting. However, Ridge has a notable limitation: it never sets any coefficient to zero unless $\lambda = \infty$. This behavior might not significantly affect prediction accuracy but can complicate model interpretation, particularly when the dimensionality d (the number of predictor variables in X) is substantial. On the other hand, L1 regularization, commonly referred to as Lasso, addresses this issue effectively. Within a linear regression framework, the optimization problem for Lasso is formulated as:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \quad \frac{1}{n} \cdot ||X\boldsymbol{\beta} - \mathbf{y}||^2 + \lambda \, ||\boldsymbol{\beta}||_1 \tag{30}$$

where $||\boldsymbol{\beta}||_1$ denotes the L1 norm of $\boldsymbol{\beta}$, which is the sum of the absolute values of the coefficients.

Like L2, the L1 penalty shrinks the coefficients towards zero. However, a distinctive feature of L1 regularization is its ability to set some coefficients exactly to zero when λ is sufficiently large. This phenomenon occurs because the L1 norm is

not differentiable at zero, which introduces points of non-differentiability in the objective function. As a result, during the optimization process, some coefficients can converge precisely to zero. Consequently, Lasso not only helps in reducing overfitting by regularization but also performs variable selection. This attribute allows Lasso to produce sparse models — models that involve only a subset of the available variables, enhancing model simplicity and interpretability.

The reason why L1 regularization can result in some coefficients being exactly zero lies in its geometric interpretation and the properties of the L1 norm. The L1 norm penalty, $\lambda ||\boldsymbol{\beta}||_1$, encourages sparsity because it imposes a constant penalty for any non-zero coefficient. This form of regularization can be visualized as bounding the coefficients within a diamond-shaped (in two dimensions) or a rhomboid-shaped (in higher dimensions) contour, centered at the origin.

As the value of λ increases, the size of these contours decreases, pulling the coefficient estimates towards the origin. In higher dimensions, the corners of these contours lie along the axes, and it is more probable for the optimization solution to hit these corners where some of the coefficients are exactly zero. In contrast, the L2 norm creates circular (or spherical in higher dimensions) contours, which do not promote sparsity as their boundaries never touch the axes unless the radius is zero.

Thus, by employing L1 regularization, we can achieve a model that is not only less prone to overfitting but also more interpretable due to its simplicity in involving fewer variables.

6.3 Bayesian Interpretation of L1 and L2 Regularization

Regularization techniques can also be interpreted through a Bayesian lens, where they correspond to introducing specific prior distributions on the regression coefficients. This perspective connects regularization with Bayesian inference, where regularization parameters are viewed in terms of prior beliefs about the values of the coefficients.

6.3.1 Bayesian Interpretation of L2 Regularization (Ridge)

L2 regularization, or Ridge regression, can be understood as placing a Gaussian prior on the coefficients $\boldsymbol{\beta}$. Specifically, this is equivalent to assuming that each coefficient β_i is drawn from a normal distribution centered at zero with a variance inversely proportional to the regularization parameter λ:

$$\beta_i \sim N(0, \sigma^2) \tag{31}$$

where $\sigma^2 = \frac{1}{\lambda}$. Under this framework, the Ridge penalty $\lambda ||\boldsymbol{\beta}||_2^2$ corresponds to the log of the Gaussian prior probability. The effect of this prior is to shrink the coefficients

towards zero, with stronger shrinkage as λ increases, reflecting a stronger belief that the coefficients are small.

6.3.2 Bayesian Interpretation of L1 Regularization (Lasso)

L1 regularization, or Lasso, implies a Laplace prior distribution on the coefficients $\boldsymbol{\beta}$. This corresponds to assuming that each coefficient β_i is drawn from a Laplace distribution centered at zero, which is characterized by a probability density function:

$$p(\beta_i) = \frac{\lambda}{2} \exp(-\lambda|\beta_i|) \tag{32}$$

Here, the regularization parameter λ controls the diversity of the distribution; higher values of λ create a sharper peak at zero, encouraging stronger sparsity. This Laplacian prior leads to a probability distribution with heavier tails and a sharp peak at zero, which makes it more likely for the coefficients to be exactly zero, thereby promoting sparsity.

6.3.3 Implications of Bayesian Priors

The choice between Ridge and Lasso in a regression model can thus be viewed as a choice between believing that the true coefficients are small but non-zero (Gaussian prior) versus believing that many coefficients are exactly zero with some potentially large outliers (Laplace prior).

6.4 Cross-Validation

Cross-validation is a statistical method used to estimate the generalization error of a model, especially when a large dataset for testing is not available. The process involves holding out a subset of the training data from the fitting process, then applying the statistical model to these held-out observations to evaluate performance.

6.4.1 Validation Set Approach

A straightforward method of cross-validation is the validation set approach, where the dataset is randomly divided into a training set and a validation set. The model is trained on the training set and evaluated on the validation set. This method provides a rough estimation of model performance. However, the performance can vary significantly with different splits of the data, indicating the potential instability and bias of this method.

6.4.2 Leave-One-Out Cross-Validation (LOOCV)

Leave-One-Out Cross-Validation (LOOCV) is a more intensive approach compared to the validation set method. In LOOCV, the model is fitted n times for a dataset with n entries, each time leaving out one data point. The omitted data point is used as the test set for that iteration. The LOOCV estimate of the model's performance is given by:

$$\text{LOOCV} = \frac{1}{n} \sum_{i=1}^{n} \text{MSE}_i \tag{33}$$

where MSE_i is the mean squared error of the prediction for the left-out data point in the i-th iteration. Although LOOCV is excellent for reducing bias in model evaluation, it is computationally expensive as it requires fitting the model n times.

6.4.3 k-Fold Cross-Validation

An alternative to LOOCV is k-fold cross-validation, which balances computational efficiency with model evaluation bias. In k-fold cross-validation, the dataset is divided into k equal parts. The model is then trained k times, each time using $k-1$ subsets as the training data and the remaining subset as the test set. This approach is expressed mathematically as:

$$k\text{-Fold CV Error} = \frac{1}{k} \sum_{i=1}^{k} \text{MSE}_i \tag{34}$$

where MSE_i is the mean squared error on the left-out subset during the i-th iteration. Choosing $k < n$ results in a process that is less computationally demanding than LOOCV, but it may introduce more bias into the estimation of the model's error.

7 Lecture 7: Classification

If we have a classification problem, we need a way of defining $\text{P}[Y \mid X]$. Unlike regression methods which can (and will!) output negative values, classification models can provide meaningful estimates for $\text{P}[Y \mid X]$, particularly important in multi-class settings.

7.1 Logistic Regression

Logistic regression is a fundamental classification method. It differs from typical regression as it estimates the probability of an observation belonging to a class rather

than directly assigning a class. The probability that observation X_i belongs to a certain class is given by the logistic function:

$$p(X_i) = \frac{e^{\beta_0+\beta_1 X_i}}{1 + e^{\beta_0+\beta_1 X_i}} \tag{35}$$

Through some algebraic manipulation, we can express this as:

$$\log \frac{p(X_i)}{1 - p(X_i)} = \beta_0 + \beta_1 X_i \tag{36}$$

This transformation is known as the *logit* of $p(X_i)$, which interestingly shows that the logit is linear with respect to X.

Estimation via Maximum Likelihood: To determine the best parameters for our logistic model, we employ the maximum likelihood estimation method. The likelihood function for a set of parameters β_0 and β_1, considering a binary classification scenario, is formulated as:

$$L(\beta_0, \beta_1) = \prod_{i:y_i=1} p(X_i) \prod_{j:y_j=0} (1 - p(X_j)) \tag{37}$$

Maximizing this function with respect to the parameters yields the estimates $\hat{\beta}_0$ and $\hat{\beta}_1$, which are used in predicting new data points.

7.2 Multivariable Logistic Regression

Multivariable logistic regression extends the simple logistic regression model to accommodate multiple predictors. This model is particularly useful for scenarios where the outcome depends on more than one explanatory variable.

7.2.1 Model Formulation

In the multivariable logistic regression, we model the probability that an observation belongs to a particular class as a function of several input variables. If $X_i = (1, X_{i1}, X_{i2}, \ldots, X_{ip})^T$ is a vector representing the predictors for the i-th observation and $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2, \ldots, \beta_p)^T$ is the vector of coefficients, the probability is given by:

$$p(X_i) = \frac{e^{\boldsymbol{\beta}^T X_i}}{1 + e^{\boldsymbol{\beta}^T \mathbf{X}_i}} \tag{38}$$

This can be rearranged using the logit transformation:

$$\log \frac{p(X_i)}{1 - p(X_i)} = \boldsymbol{\beta}^T X_i \tag{39}$$

This equation shows that the logit (the logarithm of the odds) is a linear combination of the predictors.

7.2.2 Parameter Estimation

The estimation of parameters in a multivariable logistic regression is typically performed using maximum likelihood estimation (MLE). The likelihood function for the logistic model, given binary outcomes across observations, is:

$$L(\boldsymbol{\beta}) = \prod_{i:y_i=1} p(X_i) \prod_{j:y_j=0} (1 - p(X_j)) \tag{40}$$

Maximizing this likelihood function with respect to $\boldsymbol{\beta}$ provides estimates $\hat{\boldsymbol{\beta}}$, which best explain the observed relationships between predictors and the outcome variable.

Note: The optimization of this likelihood function is typically performed numerically using iterative methods such as Newton-Raphson or gradient descent, as analytic solutions are not feasible due to the complexity of the model.

7.3 Multivariate Logistic Regression

Multivariate logistic regression, also known as multinomial logistic regression when each outcome is a category, allows for the modeling of scenarios where there are multiple dependent categorical variables. This is particularly useful for understanding complex relationships where outcomes influence each other.

7.3.1 Model Formulation

In multivariate logistic regression, we aim to model multiple responses simultaneously. Let Y_i be the vector of responses for the i-th observation and X_i be the corresponding vector of predictors. If each element of Y_i can take on values from a set of categories, the model provides probabilities for each category for each response variable.

The model uses a set of coefficients for each response variable, creating a matrix of coefficients $\boldsymbol{\beta}$. Each row of $\boldsymbol{\beta}$ corresponds to a different response variable, and each column corresponds to a predictor. The probability of observing a particular category for each response variable is modeled similarly to the binary logistic regression but extended across multiple equations:

$$p(Y_{ij} = k \mid X_i) = \frac{e^{\boldsymbol{\beta}_{jk}^T X_i}}{\sum_{l} e^{\boldsymbol{\beta}_{jl}^T X_i}} \tag{41}$$

where $\boldsymbol{\beta}_{jk}$ is the coefficient vector for the j-th response in category k, and the denominator ensures that the probabilities sum to one.

7.3.2 Parameter Estimation

The parameters of a multivariate logistic regression model are usually estimated using maximum likelihood estimation. The likelihood function considers the joint probability distribution of the response vector given the predictors, which is the product of the probabilities for each category of each response:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} \prod_{j=1}^{m} \prod_{k=1}^{K_j} (p(Y_{ij} = k \mid X_i))^{\mathbb{1}(Y_{ij}=k)} \tag{42}$$

where n is the number of observations, m is the number of response variables, K_j is the number of categories in the j-th response, and $\mathbb{1}(Y_{ij} = k)$ is an indicator function that is 1 if $Y_{ij} = k$ and 0 otherwise.

Note: This model is more computationally intensive due to the larger number of parameters and the complexity of the likelihood function. Techniques such as Expectation-Maximization (EM) or specialized optimization algorithms may be necessary to find the best-fit parameters.

This extension of logistic regression allows for a nuanced analysis of complex datasets with interdependent outcomes, providing a deeper insight into the underlying processes that generate the data.

8 Lecture 8: GLMs and Non-Parametric Methods

8.1 Generalized Linear Models

Generalized Linear Models (GLMs) are a versatile class of models that extend traditional linear regression to accommodate response variables, y, that are not necessarily continuous, such as counts or categorical data. This is achieved by specifying a suitable probability distribution for the response variable, $p(y \mid X_i; \vartheta)$, and linking it to a linear predictor through a function.

8.1.1 Components of a GLM

GLMs consist of three primary components:

- **Random Component:** Specifies the probability distribution of the response variable, y, which typically belongs to the exponential family (e.g., Gaussian, Binomial, Poisson).
- **Systematic Component:** The linear combination of predictors, X_i, denoted as $z = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n$.

- **Link Function:** A function, ϕ, that relates the expected value of the response variable to the linear predictor, z. The relationship is given by $\mathrm{E}[y \mid x_i; \vartheta] = \phi^{-1}(z)$.

8.1.2 Example: Logistic Regression

A common example of a GLM is logistic regression, where the response variable is categorical (e.g., binary). The probability distribution of y is modeled using the binomial distribution, and the link function is the logistic function. This function is defined as:

$$\phi(\mu) = \log \frac{\mu}{1 - \mu} \tag{43}$$

where μ is the probability of one of the outcomes (e.g., success). The inverse of this link function, $\phi^{-1}(z)$, which is the logistic function, transforms the linear predictor into a probability:

$$\mu = \frac{e^z}{1 + e^z} \tag{44}$$

8.2 k-Nearest Neighbours

k Nearest Neighbors (k-NN) is a non-parametric method that can be used for both classification and regression. This method assumes that similar inputs have similar outputs, making predictions based on the outputs of the nearest training examples.

8.2.1 The Algorithm

The steps involved in the k-NN algorithm are as follows:

1. **Distance Calculation:** Compute the Euclidean distance between the test point x^*_j and each training point x_i, represented as:
$$\left\| x_i - x^*_j \right\|_2$$
2. **Identify Nearest Neighbors:** Sort all points by their distance to x^*_j and select the top k closest points.
3. **Aggregate Neighbors' Outputs:** For regression, predict by averaging the outputs y_i of these k neighbors. For classification, use the mode of y_i. The neighborhood of x^*_j is defined by:
$$\mathrm{N}^* = \{i : x_i \text{ is one of the } k \text{ nearest training datapoints to } x^*_j\}$$
and the prediction is given by:
$$\hat{f}(x^*_j) = \mathrm{aggregate}(\{y_i : i \in \mathrm{N}^*\})$$

8.2.2 Choice of k

The hyperparameter k plays a crucial role in the performance of the k-NN algorithm:

- A smaller k makes the algorithm sensitive to noise in the data.
- A larger k provides smoother predictions but may include less similar points, which can reduce accuracy.

Optimal k is usually selected via cross-validation.

8.2.3 Data Normalization

Due to the reliance on distance calculations, it is crucial to normalize the data so that each feature contributes equally to the distance. This is particularly important when features vary in scale and range.

Min-Max Scaling This technique scales each feature to a [0, 1] range independently, enhancing the uniformity of influence among features. The transformation for each feature column j is computed as:

$$x'_{ij} = \frac{x_{ij} - \min(x_j)}{\max(x_j) - \min(x_j)}$$

where x_{ij} is the original value, and $\min(x_j)$ and $\max(x_j)$ are the minimum and maximum values of the feature column j, respectively.

8.3 Decision Trees

Decision trees are rule-based models that explicitly partition the input space using a set of decision rules. These models are structured with various nodes and branches: the terminal points of each branch are called leaf nodes, and the decision points leading to further branches are referred to as internal nodes. The lines connecting the nodes are known as branches, and when an internal node divides into two branches, it is described as binary.

8.3.1 Regression Tree

We will start by discussing how to train a decision tree in a regression problem. The prediction $\hat{f}(x_j^*)$ is a piecewise constant function of x_j^* and can be written as:

$$\hat{f}(x_j^*) = \sum_{\ell=1} \hat{f}_\ell 1_{\{R_\ell\}}(x^*) \tag{45}$$

where L is the total number of regions (leaf nodes), R_ℓ is the ℓth region, $\hat{f}_\ell$ is the constant prediction for said region, and $1_{\{R_\ell\}}(X^*_j)$ is the indicator function, which equals 1 if $X^*_j \in R_\ell$ and 0 otherwise. Training the tree involves finding suitable values for the parameters defining the function, namely the regions R_ℓ and the constant predictions.

8.3.2 Classification Tree

Classification trees, similar to regression trees, use a set of rules to determine the class label for a given input X^*_j. In classification, the prediction $\hat{f}(X^*_j)$ is typically the class that occurs most frequently within the leaf node:

$$\hat{f}(X^*_j) = \text{mode}\{y_i : X_i \in R_\ell \text{ and } 1_{\{R_\ell\}}(X^*_j) = 1\} \tag{46}$$

where R_ℓ is the region corresponding to the leaf node that contains X^*_j, and y_i are the class labels of the training instances in R_ℓ. Training a classification tree involves creating splits that maximize the purity of each node, using criteria such as Gini index or entropy, to ensure that each leaf is as homogeneous as possible in terms of class distribution.

8.3.3 Criteria for Splitting in Decision Trees

In decision trees, the choice of the best split at each node is crucial to reduce the complexity and increase the accuracy of the model. Several criteria can be used to measure the quality of a split, including the misclassification rate, Gini index, and entropy. These measures help determine which feature and threshold should be used to divide the node into child nodes.

Misclassification Rate The misclassification rate is the simplest criterion for evaluating splits. It is calculated as the proportion of the most common class in a node minus one:

$$\text{Misclassification Rate} = 1 - \max(\text{proportion of class } k) \tag{47}$$

where the proportion of class k is the number of instances of class k in the node divided by the total number of instances in the node. This criterion seeks to minimize the error directly but may not be sensitive enough for trees with more than two classes or imbalanced class distributions.

Gini Index The Gini index measures the impurity of a node. A node is pure (Gini = 0) when all its cases belong to a single class. The Gini index for a node is computed as:

$$\text{Gini Index} = 1 - \sum_{k=1} (p_k)^2 \tag{48}$$

where p_k is the proportion of class k instances within the node, and K is the number of classes. The Gini index is a measure of the total variance across the classes in the node. It is particularly effective for categorical targets where the variable does not have a huge number of categories.

Entropy Entropy, a concept borrowed from information theory, measures the randomness or uncertainty within a node. The entropy for a dataset is zero when it contains instances of only one class. It is calculated using the formula:

$$\text{Entropy} = -\sum_{k=1} p_k \log_2(p_k) \tag{49}$$

where p_k is the proportion of class k instances in the node. A split that results in the largest decrease in entropy is considered the best split. Entropy is particularly useful for training classification trees because it gives the best possible reduction in entropy after each split.

Each of these splitting criteria aims to optimize different aspects of the tree's structure and classification power. Typically, the choice of splitting criterion can have a significant impact on the performance of the decision tree model.

9 Lecture 9 - 10: Support Vector Machines

Support Vector Machines (SVMs) are a sophisticated development of a simpler classifier known as the Maximal Margin Classifier. This simpler form is often impractical as it requires that all classes be linearly separable with a linear boundary.

9.1 Maximal Margin Classifier

A Maximal Margin Classifier identifies a type of hyperplane in a d-dimensional space. A hyperplane is a flat, $d - 1$ dimensional subspace defined by the equation:

$$\mathbf{w}^T \mathbf{x} = 0 \tag{50}$$

where $\mathbf{w}$ and $\mathbf{x}$ are a vector in R^d. If $\mathbf{x}$ satisfies the equation, it lies on the hyperplane. Values greater than or less than 0 indicate the vector's position relative to either side of the hyperplane.

9.1.1 Classification using a Hyperplane

Consider a feature matrix X of dimensions $n \times d$, with labels $\mathbf{y}_i \in \{-1, 1\}$ for $i = 1, \ldots, n$. If it's possible to perfectly separate the classes with a hyperplane, such a hyperplane must satisfy:

$$\mathbf{y}_i(\mathbf{w}^T X_i) > 0 \quad \forall i = 1, \ldots, n \tag{51}$$

A natural classifier then assigns a class based on the sign of:

$$\hat{f}(X_i) = \mathbf{w}^T X_i \tag{52}$$

The magnitude of $\hat{f}(X_i)$ can also indicate the confidence in its classification and if $\|\mathbf{w}\| = 1$ then the margin is called the geometric margin

9.1.2 The Maximal Margin Classifier

If data are linearly separable, many separating hyperplanes are possible. The optimal choice is the maximal margin hyperplane, which maximizes the distance (margin) from the nearest training observations. This hyperplane is computed by solving the following optimization problem:

$$\max_{\mathbf{w}} \min_i \frac{1}{\|\mathbf{w}\|} y_i(\mathbf{w}^T X) \tag{53}$$

$$\text{subject to } y_i(\mathbf{w}^T X_i) \geq 1, \quad \forall i = 1, \ldots, n. \tag{54}$$

where $\min_i$ is the minimal distance between the hyperplane and any observation of the feature matrix X. This is called the maximal margin classifier or hard margin classifier. This problem is very difficult, as we have a nested max-min, and can be reformulated as

$$\max_{\mathbf{w}:\|\mathbf{w}\|=1} M \tag{55}$$

$$\text{such that } y_i \mathbf{w}^T X_i \geq M \quad \forall i = 1, \ldots, n. \tag{56}$$

or equivalently

$$\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{w}\|^2 \tag{57}$$

$$\text{such that } y_i \mathbf{w}^T X_i \geq 1 \quad \forall i = 1, \ldots, n. \tag{58}$$

9.2 Support Vector Classifiers

The support vector classifier, also called soft margin classifier, rather than seeking the largest possible margin so that every observation is on the correct side of the margin (and of the hyperplane) we allow some observation to be on the incorrect side of the margin (and even hyperplane).

The support vector classifier is the hyperplane that separates most of the training observations but may misclassify a few observations. It is the solution to the optimization problem

$$\min_{\mathbf{w},\xi} \frac{\lambda}{2}\|\mathbf{w}\|^2 \frac{1}{n} + \sum_{i=1} \xi_i \tag{59}$$

$$\text{such that } y_i(\mathbf{w}^T X_i) \geq 1 - \xi_i, \quad \forall i = 1, \ldots, n \tag{60}$$

$$\text{and } \xi \geq 0, \quad \forall i = 1, \ldots, n. \tag{61}$$

Where $\xi_1, \ldots, \xi_n$ are slack variables that allow individual observations to be on the wrong side and λ is the tradeoff parameter; if λ is very small we want as few points in the margin as possible, while on the contrary if λ is very large, we are comfortable with more and more observations within the margin.

Note: misclassification only happens for $\xi_i \geq 1$.

The maximization problem can be rewritten as

$$\min_{\mathbf{w}} \frac{\lambda}{2}\|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \underbrace{1 - y_i \mathbf{w}^T X_i}_{\text{hinge loss}}{}_{+} \tag{62}$$

where $[\alpha]_+ = \max\{\alpha, 0\}$. This is a form of regularized empirical loss minimization. We can derive (61) using the risk minimization framework; given X, $\mathbf{y} \sim D$ we want to find the classifier $g : X \to \mathbf{y}$ with the lowest possible risk, which is

$$L(g) = \mathrm{P}_D(\mathbf{y} = g(X)) \tag{63}$$

we don't know D and for this reason we use empirical risk minimization,

$$\min_{g:X\to\mathbf{y}} L_{TR}(g) = \frac{1}{n} \sum_{i=1}^{n} 1(\mathbf{y}_i g(X_i) \leq 0) \tag{64}$$

However this problem is not convex; to circumvent this, we take a convex function $\phi : \mathrm{R} \to \mathrm{R}$ which is always greater than (or equal) than the 0-1 loss and minimize that function. In this case we take the hinge loss, which penalises more if it's not confident enough or if it's far from the border. Then once we have the surrogate we compute

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1} \phi(\mathbf{y}_i \mathbf{w}^T X_i) \tag{65}$$

9.2.1 How to compute w

We have now a convex problem that, however, is not smooth (as the hinge loss is not differentiable at 1). To circumvent this problem we can either use subgradient descent (see the next subsection) or define an auxiliary function $g(\mathbf{w}, \mathbf{a})$ such that

$$\min_{\mathbf{w}} L(\mathbf{w}) = \min_{\mathbf{w}} \max_{\mathbf{a}} g(\mathbf{w}, \mathbf{a}) \tag{66}$$

But first we have to define a good g, in this case we take it to be

$$g(\mathbf{w}, \mathbf{a}_i) = \max_{\mathbf{a}_i \in [0,1]} \mathbf{a}_i (1 - \mathbf{y}_i \mathbf{w}^T X_i \tag{67}$$

Then the problem becomes

$$\min_{\mathbf{w}} L(\mathbf{w}) = \min_{\mathbf{w}} \max_{\mathbf{a} \in [0,1]^n} \frac{1}{n} \sum_{i=1}^{n} \mathbf{a}^i (1 - \mathbf{y}^i \mathbf{w}^T X^i) + \frac{\lambda}{2} \|\mathbf{w}\|^2 \tag{68}$$

If g is convex in $\mathbf{w}$ and concave in $\mathbf{a}$ and the domains of $\mathbf{w}$, $\mathbf{a}$ are convex and compact then

$$\min_{\mathbf{w}} \max_{\mathbf{a}} g(\mathbf{w}, \mathbf{a}) = \max_{\mathbf{a}} \min_{\mathbf{w}} g(\mathbf{w}, \mathbf{a}) \tag{69}$$

where we say that a domain is *convex* if, for every pair of points within the domain, every point on the straight line segment that joins them is also within the domain. A domain is *compact* if it is closed and bounded; that is, it contains all its limit points, and its size is limited. As this condition is satisfied in Support Vector Machines we can exchange the min and max.

We can now solve the inner problem by taking its gradient, namely

$$\nabla_{\mathbf{w}} G(\mathbf{w}, \mathbf{a}) = -\frac{1}{n} \sum_{i=1}^{n} \mathbf{a}_i \mathbf{y}_i X_i + \lambda \mathbf{w} \tag{70}$$

setting it to 0 we can express $\mathbf{w}$ as a function of $\mathbf{a}$,

$$\mathbf{w}(\mathbf{a}) = \frac{1}{\lambda n} \sum_{i=1}^{n} \mathbf{a}_i \mathbf{y}_i \mathbf{x}_i = \frac{1}{\lambda n} X^T \mathbf{y} \mathbf{a} \tag{71}$$

where $\mathbf{a}$ is a row vector and $\mathbf{y}$ is a column vector. Now the dual problem becomes

$$\max_{\mathbf{a} \in [0,1]^n} \frac{1}{n} \sum \mathbf{a}_i \left(1 - \frac{1}{\lambda N} \mathbf{y}_i X_i^T X^T \mathbf{y} \mathbf{a}\right) + \frac{1}{2\lambda n^2} \| X^T \mathbf{y} \mathbf{a} \|_2 \tag{72}$$

which is equal to

$$\max_{\mathbf{a} \in [0,1]^n} \frac{\mathbf{1}^T \mathbf{a}}{n} - \frac{1}{2\lambda n^2} \mathbf{a}^T \mathbf{y} X X^T \mathbf{y} \mathbf{a} \tag{73}$$

And this problem is differentiable and concave and can be solved either through quadratic programming solvers or coordinate ascent methods. The loss function depends on the data only through the kernel matrix $K = XX^T \in \mathrm{R}^{n \times n}$ which does not depend on the dimensionality of the feature matrix, d.

$\mathbf{a}$ is tipically sparse and $= 0$ only for the training observation necessary for determining the decision boundary. There are three cases:

- $\mathbf{a}_i = 0$, which means that the X_i is a non-support vector
- $\mathbf{a}_i \in [0, 1]$ which means that X_i is an essential support vector (right side but on the margin)
- $\mathbf{a}_i = 1$ which means that X_i is a bound support vector (wrong side or strictly inside the margin)

Explanation of Subgradient Descent

Subgradient descent is a variant of the gradient descent method used for minimizing non-differentiable functions. It is particularly useful in optimization problems where the objective function does not have a derivative at some points, as is the case with the hinge loss function at 1.

In gradient descent, we update the variable $\mathbf{x}$ using the formula:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} - \alpha \nabla f(\mathbf{x}_{\text{old}})$$

where α is the learning rate and $\nabla f(\mathbf{x}_{\text{old}})$ is the gradient of the function f at $\mathbf{x}_{\text{old}}$.

For non-differentiable functions, the gradient ∇f at certain points does not exist. Instead, we use a subgradient, which is any vector $\mathbf{g}$ that satisfies:

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{g}^T(\mathbf{y} - \mathbf{x}), \quad \text{for all } \mathbf{y}$$

The subgradient is not unique, and any vector that satisfies the above condition can be used as a subgradient.

The update rule in subgradient descent then becomes:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} - \alpha \mathbf{g}$$

where $\mathbf{g}$ is a subgradient of f at $\mathbf{x}_{\text{old}}$. The choice of the subgradient and the learning rate α significantly impacts the convergence properties and the solution's quality.

Subgradient methods are typically slower in convergence compared to gradient descent for smooth problems, and careful tuning of the learning rate is necessary to achieve good performance.

11 Lecture 11: Kernel Methods

11.1 Kernel Ridge Regression

Sometimes, as we have seen in lecture 5, it makes sense to transform the input data with some function ϕ, for example we can take a feature matrix X where $X_i = [x_{i1}, x_{i2}]$ and transform it into $\Phi_i = [x_{i1}, x_{i2}, x_{i1}x_{i2}]$.

A carefully engineered Φ may perform extremely well for a specific machine learning problem, however we would like Φ to contain a lot of transformations that could possibly be of interest for most problems, letting $d \to \infty$, where d is the dimensionality of Φ.

First of all, however we have to use some kind of regularisation if we are going to increase d in order to avoid overfitting. Recall that the equation for L2 linear regression is

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta} \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{\beta}^T \Phi - y)_i^2 + \lambda \|\boldsymbol{\beta}\|_2^2 = (\Phi^T \Phi + n\lambda \mathbf{I})^{-1} \Phi^T \mathbf{y} \tag{74}$$

The problem is that for d very large, if we want to compute a prediction, we have to learn d parameters and store the d-dimensional vector $\hat{\boldsymbol{\beta}}$. The first step is to realise that the predictions can be rewritten as

$$\hat{f}(X_i^*) = \underbrace{\mathbf{y}^T}_{1\times n} \underbrace{\Phi}_{n\times d} \underbrace{\Phi^T \Phi + n\lambda \mathbf{I})^{-1}}_{d\times d} \underbrace{\phi(X_i^*)}_{d\times 1} \tag{75}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{n\times 1}$$

This expression suggests that for each test input we could compute the n-dimensional vector $\Phi(\Phi^T \Phi + n\lambda \mathbf{I})^{-1}\phi(X_i^*)$; by doing so, we avoid storing a d-dimensional vector but we would still have to invert a $d \times d$. As the push-through matrix identity states that $A(A^T A + \mathbf{I})^{-1} = (AA^T + \mathbf{I})^{-1}A$ for any matrix A we could rewrite our equation to be

$$\hat{f}(X_i^*) = \underbrace{\mathbf{y}^T}_{1\times n} \underbrace{(\Phi\Phi^T + n\lambda \mathbf{I})^{-1}}_{n\times n} \underbrace{\Phi\phi(X_i^*)}_{n\times 1} \tag{76}$$

It appears that we can compute $\hat{f}(X_i^*)$ without having to deal with any d-dimensional vector or matrix, provided that the multiplications can be computed. Examining $\Phi\Phi^T$ we can see that

$$\Phi\Phi^T = \begin{bmatrix} \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_1)^T\phi(\mathbf{x}_n) \\ \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_2)^T\phi(\mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_1) & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_2) & \cdots & \phi(\mathbf{x}_n)^T\phi(\mathbf{x}_n) \end{bmatrix} \tag{77}$$

each entry is an inner product, which is a scalar. If we are able to compute that inner product directly, without explicitly computing each $\phi(\mathbf{x})$, we have reached our goal.

11.2 The Kernel Trick

The kernel trick is a powerful technique in machine learning that facilitates the handling of high-dimensional feature spaces without the need to explicitly compute the coordinates of the data points in that space. This is achieved by defining a kernel function $k(\mathbf{x}, \mathbf{x}')$ that computes the inner products between the images of the data in the feature space:

$$k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}') \tag{78}$$

where ϕ is a mapping from the input space to the feature space.

Mercer's Theorem plays a crucial role in the kernel trick. It states that a kernel function $k(\mathbf{x}, \mathbf{x}')$ corresponds to an inner product in some feature space if and only if the corresponding kernel matrix $\mathbf{K}$ is symmetric and positive semi-definite. This matrix $\mathbf{K}$, with elements $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, is known as the Gram matrix.

The **Representer Theorem** provides a theoretical justification for the kernel trick. It asserts that any function that minimizes a regularized risk functional over a reproducing kernel Hilbert space can be expressed as a linear combination of kernel functions evaluated at the training data points:

$$\hat{f}(\mathbf{x}) = \sum_{i=1} \alpha_i k(\mathbf{x}_i, \mathbf{x}) \tag{79}$$

where α_i are coefficients determined through the learning process. This theorem demonstrates that the complexity of the model is inherently controlled by the number of training samples, rather than the dimensionality of the feature space, facilitating computation in very high-dimensional spaces.

In the context of **Kernel Ridge Regression**, the kernel trick allows us to reformulate the regression function as follows:

$$\hat{f}(\mathbf{X}^*) = \mathbf{y}^T (\mathbf{K} + n\lambda \mathbf{I})^{-1} \mathbf{k}(\mathbf{X}^*) \tag{80}$$

where $\mathbf{k}(\mathbf{X}^*)$ is a vector consisting of $k(\mathbf{x}_i, \mathbf{X}^*)$ for each training point $\mathbf{x}_i$. This model avoids the explicit computation of the feature vectors $\phi(\mathbf{x}_i)$, using instead the kernel matrix $\mathbf{K}$ to achieve efficient computation. The kernel approach provides a practical way to handle situations where the dimensionality of the feature space is extremely large or even infinite.

By exploiting properties of kernel functions and the structure of data, the kernel trick significantly simplifies computations in machine learning algorithms and enables the application of linear methods to solve nonlinear problems.

12 Lecture 12: Gaussian Processes

A Gaussian Process (GP) is a specific type of stochastic process that generalizes the concept of multivariate Gaussian distributions to infinite dimensions. A stochastic process corresponds to a collection of random variables $\{z(t) : t \in \mathbb{R}\}$ indexed by time t. In this context, each time point t corresponds to a random variable $z(t)$, and the values at different time points are correlated. This correlation depends on the difference between time points, x, and can be generalized to functions of random variables, $\{f(\mathbf{x}) : \mathbf{x} \in X\}$, where X is the input space of dimension d.

In the Gaussian process model, we begin by considering the case where the input variable $\mathbf{x}$ is discrete and can only take q different values. In this setting, the function f can be characterized by a q-dimensional vector $\mathbf{f} = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_q)]^T$. This vector $\mathbf{f}$ is modeled as a random function by assigning it a joint probability distribution which is multivariate Gaussian:

$$p(\mathbf{f}) = \mathrm{N}(\mathbf{f}; \boldsymbol{\mu}, \Sigma) \tag{81}$$

However, for a Gaussian process on a continuous input space, we extend this by letting the index set be continuous and replacing the vector $\mathbf{f}$ with a random function f. The Gaussian process is then fully specified by its mean function $\mu(\mathbf{x})$ and a covariance function $\kappa(\mathbf{x}, \mathbf{x}')$. We will use then the notation

$$f \sim \mathrm{GP}(\mu, \kappa) \tag{82}$$

For a new, unseen point X_i^*, the Gaussian process prescribes that the joint distribution of the observed outputs $f(X)$ and the output at the new point $f(X_i^*)$ is given by:

$$p \begin{bmatrix} f(X_i^*) \\ f(X) \end{bmatrix} \sim \mathrm{N} \left(\begin{bmatrix} f(X_i^*) \\ f(X) \end{bmatrix}; \begin{bmatrix} \mu(X_i^*) \\ \mu(X) \end{bmatrix}, \begin{bmatrix} \kappa(X_i^*, X_i^*) & \kappa(X_i^*, X)^T \\ \kappa(X, X_i^*) & \kappa(X, X) \end{bmatrix} \right) \tag{83}$$

where X represents the set of all input points corresponding to the outputs in f. The covariance between $f(X_i^*)$ and $f(X)$ is governed by the kernel functions, reflecting the underlying assumptions about the function's smoothness and variability.

The predictive distribution for $f(X_i^*)$ given the observed data is a normal distribution with mean and variance specified by conditioning on the observed data. The mean and covariance for the prediction at X_i^* are computed using the formulas:

$$\mu_{\mathrm{pred}}(X_i^*) = \mu(X_i^*) + \kappa(X_i^*, X)\kappa(X, X)^{-1}(f(X) - \mu(X)) \tag{84}$$

$$\sigma^2_{\mathrm{pred}}(X_i^*) = \kappa(X_i^*, X_i^*) - \kappa(X_i^*, X)\kappa(X, X)^{-1}\kappa(X, X_i^*). \tag{85}$$

13 Lecture 13-14: Ensemble Methods

13.1 Bagging

Bagging, which stands for *Bootstrap Aggregating*, is an ensemble method designed to improve the stability and accuracy of machine learning algorithms. It reduces variance and helps to avoid overfitting. Although it is usually applied to decision tree methods, it can be used with any type of method. Bagging involves creating multiple versions of a predictor and using these to get an aggregated predictor.

The process of bagging involves:

1. Creating multiple subsets of the original dataset with replacement, known as bootstrap samples.
2. Training a model on each of these subsets.
3. Combining the models using the average of predictions from all models for regression tasks or a majority vote for classification tasks.

The mathematical representation of bagging for classification can be expressed as:

$$\hat{f}_{\text{bag}}(\mathbf{x}) = \text{mode}\{\hat{f}^{(1)}(\mathbf{x}), \hat{f}^{(2)}(\mathbf{x}), \ldots, \hat{f}^{(B)}(\mathbf{x})\} \tag{86}$$

where $\hat{f}^{(b)}(\mathbf{x})$ is the prediction of the b-th model trained on the b-th bootstrap sample. For regression problems, the aggregate prediction is typically the average of the predictions:

$$\hat{f}_{\text{bag}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{(b)}(\mathbf{x}) \tag{87}$$

where B is the number of bootstrap samples (models).

The bias (or average) stays the same, but the variance is reduced by

$$\frac{1-\rho}{B}\sigma^2 + \rho\sigma^2 \tag{88}$$

Where ρ is how correlated the base models are and σ^2 is their variation

13.2 Random Forest

The idea behind random forest is similar to bagging, however instead of considering all possible p variables as splitting variables, we only consider $q \leq p$, with the subset p changing at every node split

13.3 Adaboost

AdaBoost (Adaptive Boosting) is an ensemble technique that combines weak learners (typically decision stumps) into a strong learner in a sequential manner. Each weak learner is trained on the entire dataset but with different sample weights, which are adjusted to focus more on previously misclassified instances.

Given: Training data $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i \in \{-1, 1\}$

Initialize: $D_1(i) = \frac{1}{n}$ for $i = 1, \ldots, n$

For $t = 1$ to T :

1. Train weak learner h_t using weights D_t
2. Compute error $\epsilon_t = \sum_{i=1} D_t(i) I(y_i \neq h_t(x_i))$
3. Compute weight $\alpha_t = \frac{1}{2} \log \frac{1 - \epsilon_t}{\epsilon_t}$
4. Update weights $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$ where Z_t is a normalization factor

Output: $H(x) = \text{sign} \sum_{t=1}^{T} \alpha_t h_t(x)$

13.4 Gradient Boosting

Gradient Boosting is an additive model that fits new predictors to the residual errors made by the previous predictors. Unlike AdaBoost, it does not tweak the weights of training instances but instead fits the new model to the residual errors.

Given: Training data $(x_1, y_1), \ldots, (x_n, y_n)$

Initialize: $F_0(x) = \arg\min_{\gamma} \sum_{i=1} L(y_i, \gamma)$

For $t = 1$ to T :

1. Compute residuals $r_{ti} = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}$ for $i = 1, \ldots, n$
2. Fit a learner $h_t(x)$ to residuals r_{ti} to predict r_{ti}
3. Compute $\gamma_t = \arg\min_{\gamma} \sum_{i=1} L(y_i, F_{t-1}(x_i) + \gamma h_t(x_i))$
4. Update model $F_t(x) = F_{t-1}(x) + \gamma_t h_t(x)$

Output: $F_T(x)$

15 Lecture 15: Introduction to Neural Networks

We start with the description of the neural network model

$$\hat{f}(\mathbf{x}) = \boldsymbol{\beta}^T \mathbf{x} + b \tag{89}$$

Where the weights are $\beta_1, \ldots, \beta_d$ offset by a term b and the input vector $\mathbf{x} \in \mathrm{R}^d$. To describe non-linear relationships between $\mathbf{x}$ and $\hat{f}(\mathbf{x})$ we introduce a non-linear scalar function called activation function $h : \mathrm{R} \to \mathrm{R}$, where the linear regression model is modified into a generalised linear regression model where the linear combination of the inputs is transformed by h

$$\hat{f}(\mathbf{x}) = h(\boldsymbol{\beta}^T \mathbf{x} + b) \tag{90}$$

with common choices being the logistic function $h(x) = \frac{1}{1 + e^{-z}}$ and rectified linear unit (ReLU) $h(x) = \max(0, x)$.

The Generalised Linear Model is very simple and unable to describe the complicated relationship between $\mathbf{x}$ and $\hat{f}(\mathbf{x})$, for this reason we use several parallel generalised linear regression models to build a layer and then stack these layers in a sequential construction.

In the equation (90), the output is constructed by one scalar regression. To increase its flexibility, we instead let its output be a sum of U generalised linear regression models. The parameters for the k-th regression model are $b_k^{(1)}$, $W_k^{(1)}$ =

$\beta_1, \ldots, \beta_d$ and denote its output by q_k, which is reached by using the various betas and the bias b in equation (90). The outputs of this first layer is then fed into the second layer, where the output is

$$\hat{f}(\mathbf{x}) = W_1^{(2)} q_1 + \cdots + W_U^{(2)} q_U + b^{(2)} \tag{91}$$

In vector notation,

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & \cdots & W_{1d}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & \cdots & W_{2d}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{U1}^{(1)} & W_{U2}^{(1)} & \cdots & W_{Ud}^{(1)} \end{bmatrix} \qquad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_U^{(1)} \end{bmatrix} \tag{92}$$

$$W^{(2)} = \begin{bmatrix} W_1^{(2)}, \ldots, W_U^{(2)} \end{bmatrix} \qquad \mathbf{b}^{(2)} = [b^{(2)}] \tag{93}$$

The two-layer neural network is a useful method on its own, however the real descriptive power of a neural network comes when multiple of such layers are stacked together, achieving a deep neural network. Each layer maps a hidden layer

$$\mathbf{q}^{(L)} = h(W^{(L)} \mathbf{q}^{(L-1)} + \mathbf{b}^{(L)}) \tag{94}$$

If we use classification, we simply set h to be the softmax, defined as

$$h(\mathbf{x}) = \frac{1}{\sum_{j=1}^{M} e^{z_j}} \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_M} \end{bmatrix} \tag{95}$$

16 Lecture 16: Training a neural network

In a neural network, we find the suitable values for the parameters $\boldsymbol{\theta}$ by solving the following optimization problem

$$\hat{\boldsymbol{\theta}} = \arg\min_{\theta} J(\boldsymbol{\theta}) = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, \boldsymbol{\theta}) \tag{96}$$

Where $J(\boldsymbol{\theta})$ is the cost function. The functional form of the los function depends on the problem (MSE for classification, Cross-entropy for multiclass logistic regression), and the optimization is usually done through gradient descent.
The backpropagation algorithm is an important ingredient in almost all procedures,

which is how the gradients and the cost function are calculated with repsect to all the parameters. To summarize, we want to find

$$d\mathbf{W}^{(l)} = \nabla_{\partial \mathbf{W}^{(l)}} J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial W^{(l)}_{1,1}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial W^{(l)}_{1,U(l-1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial W^{(l)}_{U(l),1}} & \cdots & \frac{\partial J(\boldsymbol{\theta})}{\partial W^{(l)}_{U(l),U(l-1)}} \end{bmatrix} \tag{97}$$

19 Lecture 19: CNNs and Regularizers

19.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special kind of neural network, originally tailored for problems where the input data has a grid structure. We will focus to images, however CNNs can also be used for any input data on a grid, be it one dimensional or n dimensional.

In contrast to a dense layer, a convolutional layer leverages two important concepts, sparse interactions and parameter sharing. By sparse we mean that most of the parameters are forced to be zero, more specifically, a hidden unit only depends on the pixels in a small region of the image, not all pixels, and in border cases, zero padding is used for the regions located outside the image.

Instead of learning separate sets of parameters for every position, we only learn one set of a few parameters, and use it for all links between input layer and the hidden units. We call this set of parameters a filter. the mapping between the input variables and the hidden units can be interpreted as a convolution between the input variables and the filter. Mathematically it can be written as

$$q_{ij} = h\left(\sum_{i=1}^{F}\sum_{j=1}^{} x_{i+k-1,j+l-1} W_{kl}\right) \tag{98}$$

where $x_{i,j}$ denotes the zero-padded input to the layer, $q_{i,j}$ is the output of the layer and $W_{k,l}$ is the filter with shape $F \times F$.

If we add more layers, we would like to reduce the number of hidden units and only store the most important information, we can use a stride parameter s, which essentially means that we jump s pixels. Mathematically, it is

$$q_{ij} = h\left(\sum_{i=1}^{F}\sum_{j=1}^{} x_{s(i+k-1),s(j+l-1)} W_{kl}\right) \tag{99}$$

Another way of summarising information is through pooling. A pooling layer acts as an additional layer after the convolutional one,it only depends on a region of pixels, however in contrast to convolutional layers, the pooling layer doesn't come with trainable parameters. Two common versions of pooling are average pooling and max pooling. In average pooling the average of the units in the corresponding regions is computed, which in other words means that the output is:

$$q_{ij} = \frac{1}{F^2} h \left(\sum_{i=1}^{F} \sum_{j=1}^{F} x_{s(i+k-1),s(j+l-1)} \right) \tag{100}$$

When extending to multiple channels, we have that the output from layer l at position i, j in channel m is

$$q_{ijm}^{(l)} = h \left(\sum_{k=1}^{F_l} \sum_{l=1}^{F_l} \sum_{m=1}^{U_{l-1}} q_{s_l(i-1)+k-1,s_l(j-1)+l,m}^{(l-1)} \; W_{k,l,m,n}^{(l)} \right) \tag{101}$$

where $q_{i,j,m}$ is the input to the layer and U_{l-1} is the number of input channels.
In a full CNN architecture, there are multiple convolutional layers, were usually the number of rows and columns decrease and the number of channels increase, allowing for the model to encode high level features. And at the end there is one, or multiple, dense layers.

19.2 Dropout

Dropout is a bagging-like technique that allows us to combine many neural networks without the need to train them separately. The trick is to let the different models share parameters with each other, which reduces the computational cost and memory requirement.
Mathematically we can write this as sampling a mask, $\mathbf{m}^{(l-1)} = [m_1^{(l-1)}, \ldots, m_{U_{l-1}}^{(l-1)}]$ where

$$m_i^{(l-1)} = \begin{cases} 1 \text{ with probability } r \\ 0 \text{ with probability } 1 - r \end{cases} \quad \forall i = 1, \ldots, U_{l-1} \tag{102}$$

so the output of layer l becomes

$$\mathbf{q}^{(l)} = h(\mathbf{W}^{(l)} \tilde{\mathbf{q}}^{(l-1)} + \mathbf{b}^{(l)}) \tag{103}$$

with

$$\tilde{\mathbf{q}}^{(l-1)} = \mathbf{m}^{(l-1)} \odot \mathbf{q}^{(l-1)} \tag{104}$$

At evaluation for unseen data points, we simply set the mask to always be equal to 1.

Note: when the model is over-parametrized a new phenomenon occurs, where after the test error explodes, it goes down again, in many cases larger overparametrized models always lead to a better test performance, and the test error peaks around $N \approx d$ (number of training points almost equal to the number of parameters). This is possibly because commonly-used optimizers provide an implicit regularization effect. For example, stochasticity in the optimization process seem to help the optimizer to find flat global minima, which tend to give more Lipschitz models and better generalization.

20 Lecture 20 - 21: Unsupervised Learning

One type of (simple) but very frequent application of Unsupervised learning is clustering. The idea is to partition the data points into clusters, where each point belongs exactly in one class

20.1 K-Means

The simplest form of this problem is to divide the space into k distinct clusters (with k chosen by the user).
a way of separating points is by using the euclidian distance, that is

$$d(\mathbf{x}, \mathbf{x}') = ||x - x'||_2^2 \tag{105}$$

And the optimization problem becomes

$$\arg \min_{R_1,\dots,R_k} \sum_{i=1} \frac{1}{|R_i|} \sum_{\mathbf{x},\mathbf{x}' \in R_i} ||\mathbf{x} - \mathbf{x}||_2^2 \tag{106}$$

It can be shown that this is equivalent to select clusters minimizing the overall distances to cluster centers

$$\arg \min_{R_1,\dots,R_k} \sum_{i=1} \sum_{\mathbf{x} \in R_i} ||\mathbf{x} - \boldsymbol{\mu}_i||_2^2 \qquad \text{where } \boldsymbol{\mu}_i = \frac{1}{|R_i|} \sum_{\mathbf{x} \in R_i} \mathbf{x} \tag{107}$$

To find an approximate solution, we set cluster centers to some initial values; for each input we find the cluster with the closest center; update centers as the average of all points belonging to that cluster.
The objective is highly non-convex, with many local minima; k-means will converge to a stationary point, which may not be the best one. The solution is to run it multiple times and select the best version. It is very sensitive to the normalisation of

the input values and clusters are forced to be spherical (due to the distance chosen) and each point can belong only to one cluster.
To choose k, there is a heuristic method called the elbow method, where methods are fitted with some values, from $k = 1$ to $k = K_{\max}$, the loss is plotted as a function of k, and you select k such that going from k to $k + 1$ yields insignificant changes.

20.2 Gaussian Mixture Model

The idea behind the GMM is that for any class (or in this case cluster), $\mathbf{x}$ follows a gaussian. In other words,

$$f(\mathbf{x}, y) = f(\mathbf{x} \mid y)\pi(y) \tag{108}$$

where $\pi(y)$ is the marginal distribution of y. Within unsupervised learning, you want to know $f(\mathbf{x})$, which is

$$\sum_{y=1}^{M} \pi(y)\mathrm{N}(\mathbf{x} \mid \boldsymbol{\mu}_y, \Sigma_y) \tag{109}$$

with $\pi(y) \geq 0 \quad \forall y = 1, \ldots, M$ and $\sum_{y} \pi(y) = 1$. In our case ys are latent variables (they exist but never observed). We can thus compute the conditional distribution of each y_i given $\mathbf{x}$

$$w_i(j) = p(y^i = j \mid \mathbf{x}^i) = \frac{\pi(j)\mathrm{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j \Sigma_j)}{\sum_{v=1}^{M} \pi(v)\mathrm{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_v, \Sigma_v)} \tag{110}$$

These conditionals allow for soft clustering, which means that we assign a probability of $\mathbf{x}_i$ to belong to any cluster.
The log likelihood is

$$\sum_{i=1}^{N} \log(f(\mathbf{x}_i)) = \sum_{i=1}^{N} \log \left(\sum_{j=1}^{M} \pi(j)\mathrm{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \Sigma_j) \right) \tag{111}$$

For $M > 1$ there is no closed solution, so for this reason we use the Expectation-Maximization (EM) algorithm.

1. **Initialization:** Start with initial estimates for the parameters, denoted as $\boldsymbol{\theta}^{(0)}$.
2. **E-Step:** Compute the expected value of the log-likelihood function with respect to the conditional distribution of the latent variables given the observed data and current parameter estimates:

$$Q(\boldsymbol{\theta}^{(t)}) = \mathrm{E}[\log f(X, \mathbf{y} \mid \boldsymbol{\theta}^{(t)}) \mid X, \boldsymbol{\theta}^{(t)}]$$

3. **M-Step:** Update the parameters by maximizing the expected log-likelihood function obtained in the E-step:

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\vartheta} Q(\boldsymbol{\theta}^{(t)})$$

4. **Check for Convergence:** Repeat the E-step and M-step until the changes in parameters ϑ are below a certain threshold.

Where

- $\boldsymbol{\theta}^{(t)}$ is the set of parameters to be estimated, usually probability of each class, the mean vector for that class and the variance-covariance matrix of that class.
- $\mathrm{E}[\log f(X, \mathbf{y} \mid \boldsymbol{\theta}^{(t)}) \mid X, \boldsymbol{\theta}^{(t)}] = \sum_{\mathbf{y}} \log\left(f(X, \mathbf{y} \mid \boldsymbol{\theta}) p(\mathbf{y} \mid X, \boldsymbol{\theta})\right)$ where the first term can be rewritten as $\log f(X, \mathbf{y} \mid \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_{y_i}, \Sigma_{y_i}) + \log \pi_{y_i}$ which means that $Q(\boldsymbol{\theta})$ can be rewritten as

$$Q(\boldsymbol{\theta}) = \sum_{i=1}^{N} \sum_{j=1}^{M} w_i(j) \log \mathrm{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_{y_j}, \Sigma_{y_j}) + \log \pi_{y_j} \tag{112}$$

If w_i were fixed, there would exist a global maxima where

$$\pi(j) = \frac{1}{N} \sum_{i=1}^{N} w_i(j)$$

$$\boldsymbol{\mu}_j = \frac{1}{\sum_{i=1}^{N} w_i(j)} \sum_{i=1}^{N} w_i(j) \mathbf{x}_i$$

$$\Sigma_j = \frac{1}{\sum_{i=1}^{N} w_i(j)} \sum_{i=1}^{N} w_i(j)(\mathbf{x}_i - \boldsymbol{\mu}_j)(\mathbf{x}_i - \boldsymbol{\mu}_j)^T$$

The GMM, as for k-means has a non-convex problem, and it can only guarantee a point of stationarity, which means that poor initialization means poor local optimum. For this reason you run GMM multiple times with random initialization.

20.3 Principal Component Analysis

The idea behind PCA is to shrink the number of dimensions from d to k with $k < d$. If we have an orthonormal basis $\{\mathbf{u}_i\}_{i=1}^{k}$ then we can approximate a d-dimensional

vector as

$$\hat{\mathbf{x}}_i = \hat{\mathbf{x}} + \sum_{j=1}^{k} a_j^i \mathbf{u}_j \tag{113}$$

where $\bar{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i$. To understand how good or bad it is we can take the mse, that is $(\mathbf{x}_i - \hat{\mathbf{x}}_i)^2$ which by orthonormality of the basis becomes

$$\sum_{j=k+1}^{d} (a_j^i)^2 \tag{114}$$

The cumulative loss will then be

$$\sum_{i=1}^{N} \sum_{j=k+1}^{d} (\mathbf{x}_i - \bar{\mathbf{x}}_i)^T \mathbf{u}_j \tag{115}$$

Which can be rewritten as

$$N \sum_{j=k+1}^{d} \mathbf{u}_j^T \left(\frac{1}{N} \sum_{i=1}^{N} (\mathbf{x}_i - \bar{\mathbf{x}}_i)(\mathbf{x}_i - \bar{\mathbf{x}}_i)^T \right) \mathbf{u}_j \tag{116}$$

The middle part is the sample covariance matrix, Σ. The optimization problem thus becomes

$$\min \sum_{j=k+1}^{d} \mathbf{u}_j^T \Sigma \mathbf{u}_j \qquad \text{subject to } \mathbf{u}_j^T \mathbf{u}_j = 1 \tag{117}$$

Using Lagrange multipliers, differentiating with respect to $\mathbf{u}_j$ and setting the gradient to 0 we get

$$\Sigma \mathbf{u}_j = \lambda_j \mathbf{u}_j \tag{118}$$

For this reason, the eigenvectors of Σ are the solution. To minimize the loss, we have to find (and discard) $d - k$ eigenvectors with the smalles eigenvalues.

An alternative view of PCA is that it's a variance-preserving method, where the variance is maximized by keeping the highest eigenvectors, so we recursively take k times the highest eigenvector.

To decide how many eigenvector we need, we use the elbow criterion, just like with k-means.

SVD is like PCA but we don't have to compute Σ. Assuming you have data centered at 0, SVD computes the following factorization

$$X = U \Gamma V^T \tag{119}$$

where U has orthonormal columns, Γ is diagonal with non-negative entries and V is orthonormal. The columns of V are the eigenvectors we're looking for and the square root in Γ give us the eigenvalues of Σ

20.4 Autoencoders

The idea is to use a feed-forward neural network where the data is projected in a much smaller dimensional space and then projected back in the original space.
If the autoencoder is linear (only one hidden layer), then setting $W_1 = U^T$ (from PCA, is the collection of the d-dimensional span that forms R^d) and $W_2 = U$ is the best setting. Deep, non linear autoencoders learn to project the data, not onto a subspace but onto a nonlinear manifold. This manifold is the image of the decoder, and this is a kind of nonlinear dimensionality reduction, which can learn more powerful codes for a given dimensionality when compared with linear autoencoders (PCA).

22 Lecture 22: Introduction to Bandits and Reinforcement Learning

22.1 Online Gradient Descent

Online learning is the process of answering a sequence of questions given (possibly partial) knowledge of the correct answers to previous questions and additional available info.
If in supervised (offline) learning, we have that predictions are based on a large dataset and we want to predict a new point, in online learning, data is collected sequentially, we have to predict labels one-by-one, and only after is the true label revealed. Here predictions can influence the data collection process, so we want to collect it in a smart way, to optimize some criterion (for example in Reinforcement Learning - RL - to maximize some cumulated reward).
Can we extend models to online learning? For the most part, yes. For example we can rewrite the OLS model to

$$\hat{\beta} = \left(\sum_{i=1}^{n} \mathbf{x}_t \mathbf{x}_t^T \right)^{-1} \sum_{t=1}^{n} y_t \mathbf{x}_t \tag{120}$$

This easy trick can't be done with logistic regression however, as there doesn't exist a closed form solution. What we can do however is to minimize the cumulated loss. At each step we observe $\mathbf{x}_t \in \mathbb{R}^d$, we predict label $\hat{y}_t \in Y$ and observe true label y_t incurring loss $\ell(y_t, \hat{y}_t)$. The idea is that we want to minimize regret, whic is defined as

$$R_T = \sum_{t=1}^{T} \ell(y_t, \hat{y}_t) - \min_{g \in \mathcal{G}} \sum_{t=1}^{T} \ell(y_t, g(\mathbf{x}_t)) \tag{121}$$

where $\mathcal{G}$ is the best predictor up to time T, i.e. the whole dataset treated as an offline problem.

The first algorithm designed for Online Convex Optimization is Online Gradient Descent (OGD). Given a convex loss function $L(\vartheta; x, y)$ where ϑ are the parameters, and (x, y) is a data point:

1. Initialize parameters ϑ_0.
2. For each iteration $t = 1, 2, \ldots, T$:
   - Receive a data point (x_t, y_t).
   - Compute the gradient $\nabla L(\vartheta_{t-1}; x_t, y_t)$.
   - Update the parameters:

$$\vartheta_t = \vartheta_{t-1} - \eta_t \nabla L(\vartheta_{t-1}; x_t, y_t)$$

     where η_t is the learning rate at time t.
3. Output ϑ_T.

For bounded gradients ($\| \nabla L(\vartheta; x_t, y_t) \| \leq L$) and parameters within a bounded domain ($\|\vartheta - \vartheta'\| \leq R$), the regret at time T is bounded by:

$$R_T \leq RL\sqrt{T}$$

This bound indicates that the average regret per round diminishes as $O(1/\sqrt{T})$, showcasing the efficiency of OGD in converging towards the performance of the best possible fixed decision.

22.2 Stochastic Multi-Armed Bandits

In Stochastic Multi-Armed Bandits (MAB), the learner chooses an action $a \in A$, the environment chooses a loss function $\ell_t : A \to [0, 1]$, the learner suffers a loss $\ell_t(a_t)$ and then observes some feedback. The environment can be stochastic or adversarial, and the action set can be continuous or finite and the feedback may be partial.
We will explore a multi-armed bandit problem, where we have k slot machine, with some probability of a reward (which is the only thing we observe if we interact with it). A bandit algorithm is a sequential sampling strategy

$$a_{t+1} = F_t(a_1, r_1, \ldots, a_t, r_t) \tag{122}$$

Since there are k arms, there are k probability distributions, with mean μ_a for each arm. At each round t the learner chooses an arm a_t and receives a reward $X_{a,t} \sim F_{a_t}$.

The goal is to maximize $\mathrm{E}\left[\sum_{t=1}^{T} r_t\right]$. The expectation is taken with respect to the probability measure on outcomes induced by the stochasticity of the algorithm and the distributions F_a of each arm.
The optimal arm, $a^* \in \arg\max_a \mu_a$ with the best mean, $\mu^* = \max_a \mu_a$. For any arm a, we define the sub-optimality gap as

$$\Delta_a = \mu^* - \mu_a \tag{123}$$

The idea is to play a^* as much as possible, minimizing the pseudo-regret defined as

$$R_F^T = T\mu^* - \mathrm{E}\left[\sum_{t=1}^{T} r_t\right] \tag{124}$$

Which is basically

$$R_F^T = \sum_{a \in [K]} \Delta_a \mathrm{E}[N_a(T)] \tag{125}$$

where $N_a(t)$ is defined as

$$N_a(t) = \sum_{i=1} 1[a_i = a] \tag{126}$$

If we just do uniform exploration, we draw each arm T/K times,

$$R_F^T = T\left[\frac{1}{K}\sum_{a \in \mathrm{A}} \Delta_a\right] \tag{127}$$

Otherwise we can use a greedy strategy, where we estimate the mean of arm a as

$$\hat{\mu}_t(a) = \frac{1}{N_a(t)} \sum_{j=1} r_{a,j} 1[a_j = a] \tag{128}$$

The next action is the bet according to current estimates

$$a_{t+1} = \arg\max_a \hat{\mu}_t(a) \tag{129}$$

with it being estimated as ∞ if $N_a(t) = 0$. The problem is that with a greedy setup we may follow a suboptimal strategy and incur regret at each round, so you suffer regret $\Omega(T)$.
A better idea is Explore-Then-Commit (ETC), where you explore uniformly for $KT_0 < T$ rounds and then commit to the arm with the best empirical mean. If

we choose $T_0 = \left(\frac{T}{K}\right)^{\frac{2}{3}} (\log T)^{\frac{1}{3}}$ guarantees $R_F^T = O(K^{\frac{1}{3}} T^{\frac{2}{3}})$. At the same time, fix a time horizon T and a number of arms K, for any bandit algorithm there exists a problem instance D such that $R_D^T \geq \Omega(\sqrt{KT})$.

To achieve the optimal regret bounds we need to explore and exploit at the same time. A simple idea is the ε-greedy rule, where at round t with probability ε you sample $a_t \sim U(\{1, \ldots, K\})$ and with probability $1 - \varepsilon$ $a_t = \arg\max_{a \in K} \hat{\mu}_a(t)$. Here $R^T \geq \frac{K-1}{K} \Delta_{\min} T$ with $\Delta_{\min} = \min_{a:\mu_a \neq \mu^*} \Delta_a$. We could obtain regret upper bound $O\left(\frac{K \log(T)}{d^2}\right)$ by setting $\varepsilon_t = \min\left(1, \frac{K}{d^2 t}\right)$ for $0 < d \leq \Delta_{\min}$, but this requires knowledge of $\Delta_{\min}$. We can be greedy but use optimism to incentivize exploration, this is the idea of UCB1, where we build a confidence interval on the mean, $\mu_a : [LCB_a(t), UCB_a(t)]$ and we play the arm with the highest UCB. The UCB is defined as

$$UCB_t(a) = \underbrace{\hat{\mu}_t(a)}_{\text{exploitation term}} + \underbrace{\sqrt{\frac{2 \log T}{N_t(a)}}}_{\text{exploration term}} \tag{130}$$

With $UCB_t(a) = \infty$ if $N_t(a) = 0$

ASTRA
BOCCONI