Pattern Discrimination: **(1) feature extraction**, mapping between raw data space and the feature space; **(2) classification,** mapping between feature space and interpretation space. We do classification based on similarity (distance between feature vectors. **Stages of RP system** include measurement/sensing, raw data preprocessing (optional), feature extraction, feature preprocessing (optional), feature selection/reduction (optional), classification; postprocessing; performance evaluation (testing.) **Main concerns:** difficult to known if we have a sufficiently large and representative dataset; features should be discriminative and not redundant, there should be an adequate parameter search for any method chosen; evaluate model with different feature/parameter sets, for robustness; consider complexity vs performance trade-off. **Precision: TP/TP+FP**, **Acc = TP+TF/all, Recall/Sensitivity = TP/TP+FN, Specificity = TF/TF+FP, F1 = 2*Prec*Recall/Prec+Recall**

Decision functions: maps between feature space and interpretation space, splits IS into decision regions, if we apply a monotonic function to it we get exact same classification; also decision function can be a linear combination of transformations to features. This is helpful to turn complex non-linear problems into linearly separable problems in higher dimension or transformed spaces. Polynomial d(x) requires setting **(k+d)! / (k! d!)** parameters, with d=n_features and k=max_pol_degree. **Multi-class classification:** divide into binary problems. Can be **absolute separation** (one-vs-all, we develop C classifiers, classifier i tells us whether sample belongs to i or not, and then we make hierarchical decisions) or **pairwise separation** (one-vs-one, we develop (c! / (2!(n-2)!) classifiers, each tells us what class in that pair the point most likely belongs to, final decision through majority voting. **Covariance Matrix:** indicates how variables vary and co-vary. It's symmetric. It's related to the **Mahalanobis distance:** adjusts to many cluster shapes, ie, data distribution, it's *scale invariant* (e. circle -> ellipsis). The mahalanobis decision surface between class 1 and 2 is linear if C1=C2 and quadratic otherwise. **PCA:** find a reduced set of new variables with less redundancy, minimizing loss. Project data into directions (eigenvectors of C) that maximize variance. The eigenvalues represent the variance along those directions. **Steps:** standardize data [(f-m)/s], compute C; find eigenvalues y such that **| y I - C | = 0** and then eigenvalue matrix such that **C V = y V.** Project data according to important PCs. **Kaiser Criterion:** keep eigenvectors with eigenvalues > 1; **Scree test:** discard eigenvalues after plot stabilizes (subjective.). PCA is **unsupervised** (doesn't need labels) which can be good but also means we can cut features because they don't contribute to variance but they actually contribute to overall discrimination. Also PCs are linear transformations, so they struggle with non-linear relationships. PCs have no physical/semantic meaning.

- $\breve{\mathbf{w}} = \sum_{i=1}^{n} \breve{\alpha}_i y_i \mathbf{x}_i$,
- $\breve{b} = -\frac{1}{2}\breve{\mathbf{w}}^T(\mathbf{x}_p + \mathbf{x}_m)$. With $\mathbf{x}_p$ and $\mathbf{x}_m$ being **any support vector** from +1 and -1 classes, respectively.
- $K(\mathbf{x}_i, \mathbf{x}) = f(\mathbf{x}_i)^T f(\mathbf{x})$

Visualization: graphical inspection of features helps see their discriminative power. Histograms let us know where data is located (mean/median), how spread it is (std/quartiles), whether it's symmetric or skewed and whether there are outliers. We should do a distribution model assessment to know wether we can apply parametric tests to it. **Kolmogorov-Smirnov (K-S):** quantifies the difference between the empirical cumulative distribution function (CDF) of the data and the theoretical CDF of a normal standard distribution. The null hypothesis states that the data is drawn from a normal distribution. **Shapiro-Wilk** is the same but for n<25. **Techniques to avoid overfitting:** Cross-validation, regularization to constrain complexity, pruning (in decision trees, ensemble methods (bagging/random forest) **Kruskal-Wallis:** non-parametric statistical test to assess features discriminative power. Formula: **(1)** sort feature values; **(2)** assign ordinal rank; **(3)** compute the H statistics: as high as H is, the more discriminative the feature. **H0:** samples from different classes are drawn from the same population (no discriminative power.) **Feature Redundancy:** use correlation matrix. **Dimensionality Ratio problem:** dr = n / d, where n is the number of samples and d number of dimensions. If we have low n, we have poor representation of problem, however, with too many dimensions we can overfit data. This is the **curse of dimensionality:** there exists a critical feature dimension (CFD) above which performance degrades (we can't just add more dimensions while keeping the number of training samples the same). In geometric pov, if we have d features each divided in m buckets, then we have m**d hypercubes dividing space – grows exponentially w d and can lead to very sparse space, representing the problem poorly. CFD not theoretically ascertainable. **MDC:** consider class means as class prototypes (representatives) and assign to each new sample the class of the nearest prototype. **Euclidean MDC:** the decision function forms hyperplanes perpendicular to the segment linking the means and passing at half-distance.

- $g_k(\mathbf{x}) = \mathbf{m}_k{}^T\mathbf{C}^{-1}\mathbf{x} - 0.5\mathbf{m}_k{}^T\mathbf{C}^{-1}\mathbf{m}_k$ $\quad d_{i,j}(\mathbf{x}) = (\mathbf{m}_i - \mathbf{m}_j)^T\mathbf{C}^{-1}[\mathbf{x} - 0.5(\mathbf{m}_i + \mathbf{m}_j)]$

- $R(\%) = \frac{\sum_{i=1}^{D}\lambda_i^2}{\sum_{i=1}^{M}\lambda_i^2}$x100

- The H statistic is:
  - $H = \frac{12}{n(n+1)}\sum_{i=1}^{c} n_i(R_i - \bar{R})^2$
  - R_i → Average of ranks for the samples belonging to **class *i***
  - $\bar{R}$ → Average of all ranks for all classes
  - n_i → Number of samples belonging to **class *i***
  - n →Total number of samples

For Mahalanobis MDC: we can use as C the pooled covariance matrix (mean of Cs of classes.) This MDC equals the Euclidean MDC when C is proportional to I. **Fisher LDA:** goal is to separate samples of diff classes by placing them in a space maximizing between-class separability and minimizing within-class variability. We can use this for dimensionality reduction and linear classifier (Fisher LDA). LDA assumes implicitly that the C of each class is equal, because the same within-class scatter matrix is used for all the classes considered. We find the optimal projection vector, w and project data according to that direction, then use euclidean MDC. LDA reduces data to at most C-1 dimensions, and it has trouble with some layouts, for example when centroids of classes are the same (ex. Concentrical) or when there's a big overlap between classes in any direction.

- $\mathbf{w} = \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$; $\mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2 = (N_1 - 1)\,\mathbf{C}_1 + (N_2 - 1)\,\mathbf{C}_2 = 2(N-1)\,\mathbf{C}$ $\quad d(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - \frac{\mathbf{w}^T\mathbf{m}_1 + \mathbf{w}^T\mathbf{m}_2}{2}$

$\mathbf{S}_W^{-1} = \frac{1}{2(N-1)}\mathbf{C}^{-1}$ thus $\mathbf{w} = \frac{1}{2(N-1)}\mathbf{C}^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$

Bayesian Classifier: takes a probabilistic approach, makes decisions based on probabilities and their associated costs. **Training** involves getting a priori probabilities and class conditional probabilities. **A priori probability P(wi):** probability of class wi without knowing any specific measurements or data distribution. **Class conditional probability density function p(x | wi):** likelihood of measurement x to occur when the class is wi, we can estimate this using training data. **A posteriori probability, P(wi | x):** probability of class wi when the measurement value is x. **Minimum Error Classification Starting Point:** we want to choose the most probable class, ie, choose w1 if P(w1 | x) > P(w2 | x).

Decision boundary: we can say decision boundary is d(x) = P(x | w1)*P(w1) – P(x | w2)*P(w2), or d(x) = g1(x) – g2(x), with gi(x)=ln(P(x|wi)*P(wi)) = ln(P(x|wi) + ln(P(wi). This is helpful when we equal P(x|wi) to the normal distribution. If we develop the function d(x) = g1(x) – g2(x) using the formula to the left, and assume C1 and C2 are equal (like Mahalanobis MDC) then we achieve a linear decision surface with the formula basically as the same as Mahalanobis MDC but with an extra factor (ln of prevalence ratio). If the prevalence ratio is 1 (same prevelance) then ln(1)=0 and Bayesian behaves exactly like Mahalanobis. **If equal prevalence and likelihood:** the threshold is half distance from means, incorrect classification equal for both classes, identical to MDC. **If unequal prevalence and equal likelihood:** decision threshold displaced towards the class with least prevalence in order to reduce number of incorrect classifications for the higher prevalence class.

- $\Lambda(\mathbf{x}) = \underbrace{\frac{p(\mathbf{x}|\omega_1)}{p(\mathbf{x}|\omega_2)}}_{\text{Likelihood ratio}} > \underbrace{\frac{P(\omega_2)}{P(\omega_1)}}_{\text{Prevalence Threshold}}$ then $\mathbf{x} \in \omega_1$ else $\mathbf{x} \in \omega_2$ or $\Lambda(\mathbf{x}) = \frac{p(\mathbf{x}|\omega_1)}{p(\mathbf{x}|\omega_2)} \underset{\omega_1}{\overset{\omega_2}{\lessgtr}} \frac{P(\omega_2)}{P(\omega_1)}$ — Likelihood ratio test (LRT)

- $g_i(\mathbf{x}) = \ln(p(\mathbf{x}|\omega_i)P(\omega_i)) = \ln(p(\mathbf{x}|\omega_i)) + \ln(P(\omega_i)) = -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_i)^T\mathbf{C}_i^{-1}(\mathbf{x} - \boldsymbol{\mu}_i) - \frac{d}{2}\ln(2\pi) - \frac{1}{2}\ln(|\mathbf{C}_i|) + \ln(P(\omega_i))$
- $d_{ij}(\mathbf{x}) = g_i(\mathbf{x}) - g_j(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \mathrm{w}_0$,

Where,

$\mathbf{w} = \mathbf{C}^{-1}(\boldsymbol{\mu}_i - \boldsymbol{\mu}_j)$; $\mathrm{w}_0 = -\frac{1}{2}(\boldsymbol{\mu}_i^T\mathbf{C}^{-1}\boldsymbol{\mu}_i - \boldsymbol{\mu}_j^T\mathbf{C}^{-1}\boldsymbol{\mu}_j) + \ln\left(\frac{P(\omega_i)}{P(\omega_j)}\right)$

Minimum Risk Classification Starting Point: decide w1 if l1 < l2. Li is the loss function for choosing I, aka, describes how much expected loss is associated with that decision. Se desenvolvermos **l1<l2** vai dar algo semelhante ao LRT. If **equal loss** then the righthand side is simply the prevalence ratio, aka it would equal the **minimum error classification.** **Loss function:** λ(ai,wj) cost of taking action ai when the true state is wj **Risk/Total Risk:** expected loss for a specific decision given an input x (loss over all possible true states weighted by probabilities: R(ai|x) = sum{i}λ(ai,wj)P(wj|x) **Bayes Risk:** Minimum possible risk achievable for a given problem (depends on the data itself, not the classifier, obtained when bayes rule is perfectly followed). **Bayes Rule:** Select action a with lowest Total Risk R(ai | x). **Strategies 4 obtaining decision rule:** **Generative**: Estimate P(I|w) and P(w) then apply Bayes. **Discriminative**: Learn decision boundary or the posterior prob. directly w/o underlying distributions. **Generalization:** Good in unseen data, **Memorization:** Overfit.

- $l_i = \sum_{k=1}^{c}\lambda(\alpha_i \mid \omega_k)P(\omega_k|\mathbf{x}) = \sum_{k=1}^{c}\lambda(\alpha_i \mid \omega_k)p(\mathbf{x}|\omega_k)P(\omega_k)$

- $c = 2$ (two class problem):
  - $R = \lambda_{12}\int_{\mathcal{R}_1} p(\mathbf{x}|\omega_2)P(\omega_2)d\mathbf{x} + \lambda_{21}\int_{\mathcal{R}_2} p(\mathbf{x}|\omega_1)P(\omega_1)d\mathbf{x}$
- Decide ω_1 if $\frac{p(\mathbf{x}|\omega_1)}{p(\mathbf{x}|\omega_2)} > \frac{\lambda_{12}P(\omega_2)}{\lambda_{21}P(\omega_1)}$

- $N(\boldsymbol{\mu}, \mathbf{C}) \approx p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\mathbf{C}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{C}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$,

Resubstitution: whole set X used for training and testing. Corresponds to setting the error estimate on the training set (apparent error). This is an optimistic and low variance error estimate. **Holdout:** total dataset randomly or not divided into two disjoint sets, one for training and one for testing .Improve testing error estimate by doing different runs. **Cross-validation:** divide data into k>1 subsets of randomly selected patterns, keep one fold with n/k patterns for testing and the remaining (n-1) parts for training, rotate k times so each fold can be testing fold and average results. **Stratified cross-validation:** ensure class prevalence along folds to handle imbalanced data. **Bootstrap:** generate artificial data samples by randomly selecting existing samples with uniform distribution. **Three set validation:** we use training and valiation subsets to choose the best model (ex. Set of parameters) and then test that on a never-before-used testing subset. **ROC:** plots the proportion of correct responses (TP or sensitivity) against the FP rate as a threshold changes. Represents trade-off between sensitivity and specificity. A perfect classifier would have AUC=1. Random classifier would have a diagonal ROC (45º degree.). **Parzen Window:** kernel density estimation. Creates histograms. **Non-parametric.** **KNN:** consider k points that are the nearest neighbors of x using a certain distance metric; the label of x depends on the class label that is found in majority among the k neighbors. It's **non-parametric** (doesn't assume about data distribution) and a **lazy** classifier because training only implies storing the feature vectors and class labels of training samples. This is good because it requires less computation for training. Also it's instance based therefore adapts locally to data structure (good for non linearly separable problems).

$\mathbf{w} = \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$ $\quad \hat{p}(\mathbf{x}) = \frac{1}{N}\frac{1}{V}\sum_{i=1}^{n}\varphi\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$

Filters: pre-processing methods independent from classifier; simple features assume independence between features and only look at relevance (discriminative power.) Ideally it would be combined with a redundancy analysis (eg. minimum redundancy maximum relevancy algorithm (mRMR). **Advantages:** fast and computationally efficient, may provice a generic selection independent of classifier. **Disadvantages:** assume independence between features. Based just on data, not in classification performance. **Wrappers:** based on classifier performamnce: we train a model with different subsets of features and choose the one that yields best performance. **Advantages:** high accuracy, overfitting control associated with classifier. **Disadvantages:** computationally expensive and lacks generality. **Embedded:** use all features and define their relevance internally during learning. More important features affect the internal parameters of the classifier more.

VC dimension, h: largest number of samples that can be correctly separated (shattered) by a classifier during training, considering all possible ways to label these samples. **Structural risk:** Structural risk refers to minimizing a bound on the true (generalization) error by balancing empirical risk with model complexity, selecting among nested function classes so as to control overfitting and achieve the best trade-off between fit to data and capacity.

SVM: define an optimal hyperplane that minimizes the training error and maximizes the separation margin among classes (minimizing structural risk.) It can extend this definition to non-linearly separable problems by having a penalty for misclassifications (soft-margin.) It can also use kernel trick to map data to high dimensional space where it's easier to classify with linear decision surfaces.

Margin, r. K is usually 1. Support vectors are those closest to decision boundary, aka **wt * x + b = +- k.** O SVM resolve o problema convexo que equivale a maximizar a margem entre as duas classes sob a restrição de classificar corretamente com margem pelo menos 1. Resolve-se com o Lagrange multipliers method: apenas support vectors ficam com lagrange multipliers não nulos.

SVM soft-margin: this method handles non linearly separable data by accounting the deviations from the ideal separable problem, it allows patterns to fall within the margin and in the opposite region but penalizes them. It adds a **free parameter, C,** that influences margin: **higher C means smaller margins**, aka hard penalization of wrong classifications, but can overfit. **Smaller C means larger margins**, soft penalization of the wrong classification. **Kernel Trick:** we can use functions to transform data to higher dimensional spaces to make problems linearly separable. But this can be computationally expensive to compute exhaustively. Kernel functions allow us to use SVMs without explicitly mapping the data; in the svm formulas we simply substitute inner products xiTxj for the kernel function K(xi, xj) = f(xi)^T * f(xj). **Gaussian kernel parameter:** this y parameter controls the locality of influence of support vectors: a large y makes each support vector affect only a small neighborhood (complex, wiggly boundary) whereas a small y yields broader influence and a smoother decision surface. **Multi-class SVM:** decompose problem in binary problems using one vs all or one vs one. We can use the **Error-Correcting Output Codes (ECOC) model**, which requires a **coding design matrix** to determine the classes that the binary classifiers will try to discriminate and set a base against which will decode the final design. To decode, the most straightforward approach is compute **hamming distance** between classifiers output and each line of the coding design matrix and attribute the class associated to the minimum distance. Hamming distance is count the number of different cells. **Get non-linear SVM w and w0 given support vectors without lagrange multipliers:** Identify SVs x1 and x2 in the original plane (on top of boundaries) -> map them to z1 and z2 with kernel -> bias = (z1_1 + z2_1)/2 -> weights vector according to the slope of the line in the middle of them. That will give us w1z1 + w2z2 + b, change to w1(x1^2)+w2(x2^2) + b if the kernel transformation is f(x) = (x1^2 x2^2)^T. **Hybrid Classifier:** combines two or more different classification methods to exploit their complementary strengths and improve overall performance.

- Univariate normal pdf:
  - $N(\mu, \sigma^2) \approx p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

$$d(x) = \sum_{i=1}^{N_{SV}} \alpha_i y_i K(x_i, x) + b$$

$$K(\mathbf{x}_i, \mathbf{x}) = e^{-\frac{(\mathbf{x}-\mathbf{x}_i)^2}{2\sigma^2}} = e^{-\gamma(\mathbf{x}-\mathbf{x}_i)^2}$$

$$\text{Maximize } L_D(\alpha) = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i, x_j)$$

Occam's Razor: when you have two competing theories which make exactly the same predictions, the simpler is the better. Aka, we should select the simplest model because it minimizes the number of incorrect assumptions. **No Free Lunch Theorem:** no algorithm is superior to any other over all possible classification problems. **Bias:** error rate due to the average performance of the classifier, how far is the average performance from optimal?. **Variance:** error rate due to training set changes, how does does it vary for different training sets? **Bagging:** draw n'<N samples with replacement from D several times, use each set to train a classifier (parallel), combine results of each classifier to the final classification usually by majority voting. This decreases variance (use case: random forests.) **Boosting:** learn a strong classifier by combining weak classifiers (classifiers with low bias and variance, only slightly better than random guessing.) We learn weak classifiers iteratively (not parallel) and reweigh data points. The final decision is a weighted sum of all the weak learners. (use case: AdaBoost.) **Stacking:** combine several heterogeneous models by training an additional classifier with the individual decision function values. Learning from diverse predictions may lead to improved overall classification. **Generalized Probabilistic Classifier Combination:** combine results of R classifiers where each classifier k outputs the probabilities of each class Pk(Wj | x.) We can apply the **Product Rule (product of every likelihood, sensitive to very low values)** or the **sum rule (sum of every likelihood),** sum rule found to perform well in practice. If prod(k=1->K)(P(wj)|x) > prod(k=1->K)(P(wi)|x) assign wj, else assign wi.\ **Advantages vs Disadvantages LDA:** advantages: computationally efficient, noise reduction, enhance classifier performance by maximizing separability, can be easily interpretable. Disadvantages: assumes data has the same covariance matrix, works best when linearly separable, difficult with some layouts (ex. Above), can produce at most C-1 discriminants. **Clustering.** **Types:** *Hierarchical/Tree:* Use linkage rules to create hierarchical sequence of clustering solutions. *Centroid adjustment algorithms / partitioning*: Based on an initial proposal for the prototypes (centroids), the algo iteratively evaluates and adjusts centroid position. *Density-based:* Join areas of high pattern density into clusters. Good for arbitrarily-shaped clusters because it can be formed as long as dense areas can be connected. *Distribution-based:* Estimate parametric data distributions from data, e.g., gaussian. *Others!* **Hierarchical Clustering:** *Agglomerative:* Start w/ *N* clusters where each cluster is composed of a single sample -> Merge two nearest clusters -> repeat until all clusters are part of a single cluster -> output dendrogram. *Divisive*: Start with all in 1 cluster -> split in two -> … **Linkage rules:** What will be the new distance between the join clusters and other clusters? *Single:* Dissimilarity given by the closest elements (good for clusters with filamentary shape. promote connectedness). *Complete:* Dissimilarity given by the furthest elements (good for compact clusters. original spherical-shaped clusters). *Average:* Pair-group using arithmetic average (effective for several types of clusters). *Centroid:* Euclidean distance between centroids of the two clusters (effective for several types of clusters). *Ward's method:* Sum of the squared within-cluster distances (minimize intra-cluster variability and consequently maximize cluster separability). *Lance-Williams formula:* update the distance matrix in agglomerative methods after merging clusters (the d in the formula can be the distance for any of the previously mentioned methods).

$$\text{10: Final classifier: } H(\mathbf{x}) = \text{sign}\left(\sum_{j=1}^{M}\alpha_j H_j(\mathbf{x})\right).$$

Centroid adjustment algorithms (K-MEANS): Assign data point to clusters -> move centroid describing the clusters. Number of clusters need to be defined a-priori (we can use hierarchical clustering in a small portion of data for initialisation. Minimize and overall within cluster distance from the patterns to the centroids (loss function to the right). K-means works if: Clusters are spherical -> clusters are well separated -> clusters are of similar volumes -> clusters have similar number of points ***K-means Alg***: assign each datapoint to the cluster represented by the nearest centroid -> compute, for the previous partition, the new centroids -> repeat until max num of iteration exceeded or there are no more changes on the centroids. **Density Algorithms (DBSCAN):** Clusters are dense areas separated by areas of lower density. Defined by two parameters: ε (radius of neighborhood with respect to some point) minPts (min number of points required to consider a region dense. *Advantages:* robust to outliers (points in very low populated areas); do not need an initial guess for the number of clusters; deal with arbitrary cluster shapes. *Point Types:* Core Point: If at least minPts points are withing a distance ε of it (including the point itself). Border Point: a point that has at least one core point at a distance ε and if the number of points inside the volume defined by ε is less than minPoint. Noise Point: Not core/border. *Reachability:* If a data point can be accessed from another datapoint (directly or indirectly). *Connectivity:* If two datapoints belong to the same cluster or not. *Two points in DBSCAN can be:* Directly density-reachable if d(y,x) < ε and y is core point, density-reachable if they can be reached indirectly, density connected if both x and y are density-reachable from O. ***DBSCAN Alg.***: Pick a point not assigned to a cluster or classified as outlier -> compute its neighborhood to determine if it's a core point -> if core point: start cluster around this point, assign label to point and all neighbours, implement neighborhood jumps to find all density-reachable points and add them to the cluster, if outlier is added, change status to border point; if not core point: label point as outlier -> repeat until all points are labeled as cluster or outlier. ***Setting hyperparameters:*** *minPts:* minPts >= numdimensions+1. This prevents the alg. from forming clusters that do not have enough density relative to num of dimensions. If we set minPts to <=2, The algorithm effectively becomes single-link agglomerative hierarchical clustering (any two points within distance ε are connected, cutting the dendrogram at height ε rather than filling dense regions). *ε:* When MinPts >= 3, develop a k-distance graph -> define k = minPts-1. For every data point in the set, calculate distance to the k-th nearest neighbor. Sort these calculated distances and plot them. the elbow is the optimal ε (threshold where points stop behaving like noise and start being part of a cluster). **Cluster evaluation:** *Unsupervised/internal:* No labels, and in general, evaluate if clusters are compact (small within cluster distances) and if clusters are well separated (large between clusters distances) (e.g. Davies-Bouldin idx, Silhouette, Dunn Index, …). *Supervised/external:* Need labels to evaluate. (e.g. Purity / Accuracy, Precision, Recall, …).

Single Link	$a_i = a_j = 0.5; b = 0; c = -0.5$
Complete Link	$a_i = a_j = 0.5; b = 0; c = 0.5$
Average Link	$a_i = \frac{n_i}{n_i+n_j}; a_j = \frac{n_j}{n_i+n_j}; b = c = 0$
Centroid	$a_i = \frac{n_i}{n_i+n_j}; a_j = \frac{n_j}{n_i+n_j}; b = -\frac{n_i n_j}{(n_i+n_j)^2}; c = 0$
Ward's Method	$a_i = \frac{n_i+n_k}{n_i+n_j+n_k}; a_j = \frac{n_j+n_k}{n_i+n_j+n_k}; b = -\frac{n_k}{n_i+n_j+n_k}; c = 0$

- $d(\omega_{i+j}, \omega_k) = a_i d(\omega_i, \omega_k) + a_j d(\omega_j, \omega_k) + b d(\omega_i, \omega_j) + c|d(\omega_i, \omega_k) - d(\omega_j, \omega_k)|$

Performance Evaluation. Confusion Matrix with TP, TN, FP, FN. Accuracy = (TP+TN)/(TP+TN+FP+FN). Sensitivity/Recall = TP/(TP+FN). Specificity = TN/(TN+FP). Precision = TP/(TP+FP). F1-score = 2·(Precision·Recall)/(Precision+Recall). Macro averaging computes metrics per class and averages them (class-balanced); Micro averaging computes metrics globally and is dominated by majority classes (≈ accuracy). For class imbalance: Binary → use Recall, Specificity, F1 or Balanced Accuracy; Multiclass → use BACC/BACCw. **Decision Trees.** Classification through recursive if-then rules; each root-to-leaf path is a conjunction, the full tree is a disjunction. Advantages: interpretable, fast, handles mixed data. Disadvantages: greedy search, high variance, sensitive to data changes. **Entropy and Information Gain (ID3).** Entropy measures class heterogeneity: $H(S)=-\sum_i p_i \log_2 p_i$. Information Gain: $Gain(S,A)=H(S)-\sum_v(|S_v|/|S|)H(S_v)$. ID3 selects the feature with highest IG recursively until entropy is zero. IG is biased toward features with many values and sensitive to noise and class imbalance. **CART.** Uses Gini impurity $G=1-\sum p^2$, binary splits only, supports classification and regression. Overfitting controlled by pruning, depth limits, minimum samples per leaf, cross-validation or Random Forests.**Naïve Bayes.** Assumes conditional independence between features given the class. Likelihood Ratio Test: $p(x|\omega_1)/p(x|\omega_2) > P(\omega_2)/P(\omega_1) \Rightarrow$ choose ω_1. Capital P denotes discrete probabilities, lowercase p denotes probability density functions; p(x) is the evidence.**Bias–Variance Decomposition.** Total error = bias + variance. High bias → underfitting; high variance → overfitting. Overfitting corresponds to very low training error and high testing error, often due to high model complexity (high VC dimension).**Feature Selection Additions.** ROC/AUC can be used to assess individual feature relevance; higher AUC implies better discriminative power. Regularization as embedded selection: LASSO (L1) forces irrelevant feature weights to zero; Ridge (L2) shrinks weights to reduce variance.**Distance Metrics.** A valid metric satisfies non-negativity, identity, symmetry and triangle inequality. A norm also satisfies d(ax,ay)=|a|d(x,y). **k-NN Theory.** As N→∞, k→∞ and k/N→0, k-NN approaches the Optimal Bayes classifier. Small k yields low bias and high variance (overfitting).

```python
class Node:
    def __init__(self, f=None, t=None, l=None, r=None, v=None):
        self.f, self.t, self.l, self.r, self.v = f, t, l, r, v
def _gini(y):
    if len(y) == 0: return 0
    p = np.bincount(y) / len(y)
    return 1 - np.sum(p**2)
def _fit_tree(X, y, depth=0, max_d=5):
    if depth == max_d or len(np.unique(y)) == 1:
        return Node(v=Counter(y).most_common(1)[0][0])
    # Random Subspace: Select sqrt(D) features
    n_feat = X.shape[1]
    feats = np.random.choice(n_feat, int(np.sqrt(n_feat)), replace=False)
    best = {'g': float('inf'), 'f': None, 't': None}
    for f in feats:
        thresholds = np.unique(X[:, f])
        for t in thresholds:
            mask = X[:, f] <= t
            if np.sum(mask) == 0 or np.sum(~mask) == 0: continue

            # Weighted Gini Impurity
            g = (len(y[mask]) * _gini(y[mask]) + len(y[~mask]) * _gini(y[~mask
            if g < best['g']: best = {'g': g, 'f': f, 't': t}
    if best['f'] is None: return Node(v=Counter(y).most_common(1)[0][0])
    mask = X[:, best['f']] <= best['t']
    return Node(f=best['f'], t=best['t'],
                l=_fit_tree(X[mask], y[mask], depth+1, max_d),
                r=_fit_tree(X[~mask], y[~mask], depth+1, max_d))
def _predict(n, x):
    return n.v if n.v is not None else _predict(n.l if x[n.f] <= n.t else n.r, x)
```

```python
def rf_training(Xtr, Ttr):
    # Xtr is D x P, Transpose to P x D for processing
    X, y = Xtr.T, Ttr.flatten().astype(int)
    trees = []
    # Bagging: Train 10 trees on bootstrap samples
    for _ in range(10):
        idx = np.random.choice(len(y), len(y), replace=True)
        trees.append(_fit_tree(X[idx], y[idx]))
    return {'trees': trees}
def rf_testing(Xte, Tte, model):
    X, y = Xte.T, Tte.flatten().astype(int)
    # Majority Voting
    preds = [_predict(t, x) for x in X for t in model['trees']]
    preds = np.array(preds).reshape(len(X), len(model['trees']))
    out = np.array([Counter(row).most_common(1)[0][0] for row in preds])
    # Metrics (1=Pos, 2=Neg)
    TP = np.sum((out == 1) & (y == 1))
    TN = np.sum((out == 2) & (y == 2))
    ss = TP / np.sum(y == 1) if np.sum(y == 1) else 0.0
    sp = TN / np.sum(y == 2) if np.sum(y == 2) else 0.0
    return out, ss, sp
```

```python
def agg_clustering_avg(dist_matrix, labels) -> list[list[Any]]:
    D = np.array(dist_matrix, dtype=float)
    np.fill_diagonal(D, np.inf)
    clusters = labels[:]
    sizes = [1] * len(labels)
    history = [list(clusters)]
    while len(clusters) > 1:
        # 1. Find the closest pair of clusters
        i, j = np.unravel_index(np.argmin(D), D.shape)
        if i > j: i, j = j, i  # Ensure i < j for consistent deletion
        # 2. Update Distances (Average Linkage: Weighted average)
        # Formula: d(u, k) = (n_i * d(i, k) + n_j * d(j, k)) / (n_i + n_j)
        n_i, n_j = sizes[i], sizes[j]
        D[i, :] = (n_i * D[i, :] + n_j * D[j, :]) / (n_i + n_j)
        D[:, i] = D[i, :]       # Maintain symmetry
        D[i, i] = np.inf        # Reset diagonal
        # 3. Merge Labels and Sizes
        clusters[i] = f"{clusters[i]} {clusters[j]}"
        sizes[i] += sizes[j]
        # 4. Remove cluster j (delete row and column)
        D = np.delete(np.delete(D, j, 0), j, 1)
        clusters.pop(j)
        sizes.pop(j)
        history.append(list(clusters))
    return history
# --- Test Case ---
dists = [
    [0, 754, 564, 138],
    [754, 0, 219, 869],
    [564, 219, 0, 669],
    [138, 869, 669, 0]
]
lbls = ['Milano', 'Napoli', 'Rome', 'Torino']
```

TEST LDA FISHER

```
def predict(self, X_test):
    X_test = np.asarray(X_test)
    X_test_proj = X_test @ self.projection_vector
    # Classify based on minimum Euclidean distance in 1D
    predictions = [0 if abs(x - self.m0_proj) < abs(x - self.m1_proj) else 1 for x in X_test_proj]
    return np.array(predictions)
```

Test MDCs

```
def min_dist_testing(Xte, Tte, model):
  Xte = np.asarray(Xte),  Tte = np.asarray(Tte)
  D, Nte = Xte.shape
mu1 = model['mu1'], mu2 = model['mu2']
  metric = model['metric']
  preds = np.empty(Nte, dtype=int)
  if metric == 'euc':
    for j in range(Nte):
      x = Xte[:, j]
      d1 = np.sum((x - mu1)**2), d2 = np.sum((x - mu2)**2)
      preds[j] = 1 if d1 < d2 else 2
  else:  # 'mah'
    invC = model['inv_cov']
    for j in range(Nte):
      x = Xte[:, j], diff1 = x - mu1, diff2 = x - mu2
      d1 = diff1 @ invC @ diff1
      d2 = diff2 @ invC @ diff2
      preds[j] = 1 if d1 < d2 else 2
  tp, tn, fp, fn = get_confusion(preds, Tte, pos_label=1)
```

Find C for SVM:

```
X = np.asarray(Xtr)
  y = np.asarray(Ttr)
  mean_f1 = []
  std_f1 = []
  skf = StratifiedKFold(n_splits=n_runs, shuffle=True)
  for C in Cs:
    f1_scores = []
    for train_idx, test_idx in skf.split(X, y):
      X_train, X_val = X[train_idx], X[test_idx]
      y_train, y_val = y[train_idx], y[test_idx]
      clf = LinearSVC(C=C, max_iter=5000, dual=False)
      clf.fit(X_train, y_train)
      y_pred = clf.predict(X_val)
      f1 = f1_score(y_val, y_pred, pos_label=1)
      f1_scores.append(f1)
    mean_f1.append(np.mean(f1_scores))
    std_f1.append(np.std(f1_scores, ddof=1))
  return np.array(mean_f1), np.array(std_f1)
```

Train Bayes for 1 and 2 labels

```
  X = np.asarray(Xtr)
  y = np.asarray(Ttr)
  # assuma rótulos exatamente 1 e 2
  mu1 = X[y==1].mean(axis=0)
  mu2 = X[y==2].mean(axis=0)
  cov1 = np.cov(X[y==1], rowvar=False) + eps*np.eye(X.shape[1])
  cov2 = np.cov(X[y==2], rowvar=False) + eps*np.eye(X.shape[1])
  prior1 = np.mean(y==1)
  prior2 = np.mean(y==2)
  # pré-calc inversas e determinantes para pdf
  inv1 = np.linalg.pinv(cov1); det1 = np.linalg.det(cov1)
  inv2 = np.linalg.pinv(cov2); det2 = np.linalg.det(cov2)
  return {
    'mu1': mu1, 'inv1': inv1, 'det1': det1, 'prior1': prior1,
    'mu2': mu2, 'inv2': inv2, 'det2': det2, 'prior2': prior2,
    'dim': X.shape[1]
```

```
def svm_testing(Xte, Tte, model) -> tuple[NDArray[Any],
    X, y = Xte.T, Tte.flatten()
    # Decision function: w^T * x + b [Slide 15]
    decision = np.dot(X, model['w']) + model['b']
    out = np.where(decision >= 0, 1, 2)
    TP = np.sum((out == 1) & (y == 1))
    TN = np.sum((out == 2) & (y == 2))
    ss = TP / np.sum(y == 1) if np.sum(y == 1) else 0.0
    sp = TN / np.sum(y == 2) if np.sum(y == 2) else 0.0
    return out, ss, sp
```

Train MDC, can be euc or mah

```
def min_dist_training(Xtr, Ttr, metric='euc', eps=1e-6):
  Xtr = np.asarray(Xtr)
  Ttr = np.asarray(Ttr)
  D, Ntr = Xtr.shape
  # separar as duas classes
  mask1 = (Ttr == 1)
  mask2 = (Ttr == 2)
  # calcular médias (vetores de dimensão D)
  mu1 = Xtr[:, mask1].mean(axis=1)
  mu2 = Xtr[:, mask2].mean(axis=1)
  model = {'mu1': mu1, 'mu2': mu2, 'metric': metric}
  if metric == 'mah':
    # covariâncias amostrais com regularização eps*I
    X1 = Xtr[:, mask1]
    X2 = Xtr[:, mask2]
    # np.cov com rowvar=True: linhas=variáveis, colunas=amostras
    cov1 = np.cov(X1, rowvar=True) + eps*np.eye(D)
    cov2 = np.cov(X2, rowvar=True) + eps*np.eye(D)
    n1 = X1.shape[1]
    n2 = X2.shape[1]
    # covariância pooled (estimador de máxima verossimilhança ou amostral)
    pooled = ((n1 - 1)*cov1 + (n2 - 1)*cov2) / (n1 + n2 - 2)
    inv_pooled = np.linalg.pinv(pooled)
    model['inv_cov'] = inv_pooled
  return model
```

TRAIN LDA FISHER

```
def train(self, X_train, Y_train):
    X_train = np.asarray(X_train)
    Y_train = np.asarray(Y_train)
    # Compute class means and scatter matrices
    class0 = X_train[Y_train == 0]
    class1 = X_train[Y_train == 1]
    self.m0 = np.mean(class0, axis=0)
    self.m1 = np.mean(class1, axis=0)
    # Compute within-class scatter matrix Sw
    S0 = np.cov(class0, rowvar=False)
    S1 = np.cov(class1, rowvar=False)
    Sw = S0 + S1
    # Compute LDA projection vector
    self.projection_vector = np.linalg.inv(Sw).dot(self.m1 - self.m0)
    self.m0_proj = np.dot(self.m0, self.projection_vector)
    self.m1_proj = np.dot(self.m1, self.projection_vector)
```

Test Bayes

```
  y = np.asarray(Tte)
  n, d = X.shape
  m = model
  # função inline de pdf multivariada Gaussiana
  def pdf(x, mu, invC, detC):
    diff = x - mu
    return np.exp(-0.5*(diff @ invC @ diff)) / np.sqrt((2*np.pi)**d * detC)
  # calcular posteriors para todas amostras de uma vez
  # mas pdf é por ponto; fazemos em loop curto
  out = np.empty(n, dtype=int)
  for i in range(n):
    x = X[i]
    p1 = pdf(x, m['mu1'], m['inv1'], m['det1']) * m['prior1']
    p2 = pdf(x, m['mu2'], m['inv2'], m['det2']) * m['prior2']
    out[i] = 1 if p1 > p2 else 2
  # sensibilidade e especificidade vetorizadas
  ss = np.mean(out[y==1] == 1) if np.any(y==1) else 0.0
  sp = np.mean(out[y==2] == 2) if np.any(y==2) else 0.0
  return out, ss, sp
```

```python
def svm_training(Xtr, Ttr) -> dict[str, Any]:
    # Transpose to PxD; Labels: 1 -> 1, 2 -> -1
    X, y = Xtr.T, np.where(Ttr.flatten() == 1, 1, -1)
    n_samples, n_features = X.shape
    # Initialize Lagrange Multipliers (alpha)
    alpha = np.zeros(n_samples)
    b = 0
    C = 1.0  # Regularization parameter (Slide 27)
    # Simplified SMO (Coordinate Descent on Dual)
    # Iterates to satisfy KKT conditions (Slide 393)
    for _ in range(50):
        for i in range(n_samples):
            # Calculate margin: f(x) - y
            # w = sum(alpha * y * x) [Slide 21]
            w = np.dot(alpha * y, X)
            error = (np.dot(X[i], w) + b) - y[i]
            # Update alpha if strict KKT conditions are violated
            if (y[i] * error < -0.01 and alpha[i] < C) or \
               (y[i] * error > 0.01 and alpha[i] > 0):
                # Gradient update for alpha (simplified)
                step = 0.01
                alpha[i] = np.clip(alpha[i] - step * y[i] * error, (
    # Compute final w from alphas [Slide 21]
    w = np.dot(alpha * y, X)
    # Save model with Support Vectors (where alpha > 0)
    return {'w': w, 'b': b, 'alpha': alpha}
```

```python
def adaboost_training(Xtr, Ttr) -> dict[str, list[Any]]:
    # Transpose D x P -> P x D. Map labels: 1->1, 2->-1
    X, y = Xtr.T, np.where(Ttr.flatten() == 1, 1, -1)
    n_samples, n_features = X.shape
    # Initialize weights uniformly
    w = np.full(n_samples, 1 / n_samples)
    models = []
    # Train 50 Decision Stumps (Weak Learners)
    for _ in range(50):
        stump = {'err': float('inf'), 'alpha': 0}
        # Greedy search for best feature/threshold minimizing weighted error
        for f in range(n_features):
            thresh = np.unique(X[:, f])
            for t in thresh:
                p = 1 # Polarity
                preds = np.ones(n_samples)
                preds[X[:, f] < t] = -1
                # Calculate weighted error
                err = np.sum(w[preds != y])
                # Invert polarity if error > 0.5
                if err > 0.5:
                    p, err = -1, 1 - err
                if err < stump['err']:
                    stump.update({'f': f, 't': t, 'p': p, 'err': err})
        # Calculate Alpha (Vote Power)
        EPS = 1e-10
        stump['alpha'] = 0.5 * np.log((1 - stump['err'] + EPS) / (stump['err'] + EPS))
        # Update Weights: Increase weight of misclassified samples
        preds = stump['p'] * np.where(X[:, stump['f']] < stump['t'], -1, 1)
        w *= np.exp(-stump['alpha'] * y * preds)
        w /= np.sum(w) # Normalize
        models.append(stump)
    return {'models': models}
```

```python
def adaboost_testing(Xte, Tte, model) -> tuple[NDArray[Any], An
    X, y = Xte.T, Tte.flatten()
    # Aggregate predictions: sign(sum(alpha * prediction))
    final_preds = np.zeros(len(X))
    for m in model['models']:
        pred = m['p'] * np.where(X[:, m['f']] < m['t'], -1, 1)
        final_preds += m['alpha'] * pred
    out = np.where(np.sign(final_preds) >= 0, 1, 2)
    TP = np.sum((out == 1) & (y == 1))
    TN = np.sum((out == 2) & (y == 2))
    ss = TP / np.sum(y == 1) if np.sum(y == 1) else 0.0
    sp = TN / np.sum(y == 2) if np.sum(y == 2) else 0.0
    return out, ss, sp
```

```python
# Read sheets
description = pd.read_excel(file_name, sheet_name="Description")
data = pd.read_excel(file_name, sheet_name="Data")

# ----------------------------------------------------
# Extract features and labels
# ----------------------------------------------------
# Assume last two columns are Apgar1 and Apgar5
X = data.iloc[:, :-2].values
apgar1 = data["Apgar1"].values

# Binary labels: bad prognosis if Apgar1 <= 6
y = (apgar1 <= 6).astype(int)

# ----------------------------------------------------
# (b) Feature ranking using AUC
# ----------------------------------------------------
auc_scores = []

for i in range(X.shape[1]):
    auc = roc_auc_score(y, X[:, i])
    auc_scores.append(auc)

auc_scores = np.array(auc_scores)

# Sort features by descending AUC
sorted_idx = np.argsort(auc_scores)[::-1]

# Select two most discriminative features
best_features_idx = sorted_idx[:2]
X_best = X[:, best_features_idx]

print("Selected feature indices:", best_features_idx)
print("Corresponding AUCs:", auc_scores[best_features_idx]

# ----------------------------------------------------
# (c)(d)(e) k-NN classification with 10 trials
# ----------------------------------------------------
errors = []

for trial in range(10):
    X_train, X_test, y_train, y_test = train_test_split(
        X_best, y, test_size=0.5, shuffle=True
    )

    # k-NN classifier with k = 8
    knn = KNeighborsClassifier(n_neighbors=8)
    knn.fit(X_train, y_train)

    y_pred = knn.predict(X_test)

    # Classification error
    error = np.mean(y_pred != y_test)
    errors.append(error)

# ----------------------------------------------------
# Results
# ----------------------------------------------------
errors = np.array(errors)
```

KNN PDF

```
def knn_pdf(data, k, eps=1e-8):
  x = np.asarray(data).ravel()
  n = x.size
  pdf = np.empty(n, dtype=float)
  for i in range(n):
    xi = x[i]
    # obtém índices e distâncias dos k vizinhos mais próximos a xi
    idxs, dists = knn1d_search(x, xi, k)
    # A distância r ao k-ésimo vizinho é:
    if len(dists) < k:
      # se houver menos pontos que k, usamos a maior distância disponível
      r = np.max(dists) if dists.size>0 else eps
    else:
      # supondo dists ordenadas ou não: garantimos pegar o k-ésimo menor
      # se não estiver ordenado:
      r = np.partition(dists, k-1)[k-1]
    if r < eps:
      r = eps
    pdf[i] = k / (n * 2 * r)
  return pdf
```