SIN NOTES 2

BLOCK 2: AUTOMATED LEARNING

CHAPTER 1.-PROBALISTIC REASONING

1.-INTRODUCTION: UNCERTAINITY AND PROBABILITY

Currently, **probabilistic methods** prevail as the general framework to represent uncertainty. These methods enable to:

- Adequately and consistently model and combine:
  - The **inaccuracy or vagueness of a priori knowledge**.
  - The **imprecision of facts, observations or data**.
- **Automated learning** of the representation models.

2.-PROBABILITY THEORY

Let Ω be a set named **sample space**. The possible worlds of Ω are **mutually exclusive** and **exhaustive**. One item w ϵ Ω is called **simple event**, world, or simply sample.

A **probability model or probabilistic space** is a sample space along with a function $P: \Omega \rightarrow \mathbb{R}$ that assigns a real number to each w ϵ Ω such that:

$$0 \leq P(\omega) \leq 1; \sum_{\omega} P(\omega) = 1$$

An **event** A is a subset of possible worlds of Ω, its probability is:

$$P(\mathrm{A}) = \sum_{\omega \in \mathrm{A}} P(\omega)$$

A **random variable** is a function that maps the sample space to a domain. If X is a random variable, "(X = x)" denotes the event:

$$(X = x) \equiv \omega \in \Omega : \mathrm{X}(\omega) = x$$

Given a random variable X, P induces a **probability distribution**:

$$P(X = x) \equiv \sum_{\omega \in (\mathrm{X}=\mathrm{x})} P(\omega)$$

A (logic) proposition is interpreted as an event (subset of possible worlds) in which the proposition is true. When using boolean variables, the set of possible worlds are just those

worlds in which the proposition holds (propositional logic). We can simplify the notation whenever the semantics is clear.

Various axiomatic formulation to the Probability theory have been proposed. For example, the Kolmogorov's axioms:

$$0 \leq P(\omega) \leq 1$$

$$\sum_{\omega} P(\omega) = 1$$

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

We can build up the rest of probability theory from this simple foundation.

Unconditional or prior probability of a random variable X:

$$P(X = x) \equiv P(x) : \sum_{x} P(x) = 1$$

Join probability of two random variables X, Y:

$$P(X = x; Y = y) \equiv P(x, y) : \sum_{x} \sum_{y} P(x, y) = 1$$

Conditional probability:

$$P(X = x|Y = y) \equiv P(x|y) : \sum_{x} P(x|y) = 1 \; \forall y$$

3.- PROBABILISTIC REASONING: INFERENCE

The unconditional (or marginal) probability P(x) is the **marginalization** of the joint probability P(x, y):

$$P(x) = \sum_{y} P(x, y), \;\; equivalently \; P(y) = \sum_{x} P(x, y)$$

The joint probability is related to the conditional and unconditional probabilities (**product rule**):

$$P(x, y) = P(x)P(y|x) = P(y)P(x|y)$$

Bayes' rule:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} = \frac{P(x,y)}{P(y)} = \frac{P(x)P(y|x)}{\sum_{x'} P(x')P(y|x')}$$

4.-UNCERTAINITY AND OPTIMAL DECISIONS

The Utility theory can be used to represent and infer preferences or the cost of the undesirable effects of the decisions:

Probability theory + Utility theory = **Statistical decision theory**

Simplification: decisions can only be "right" or "wrong" and the costs are 0 and 1, respectively. Let x ϵ X be a fact or data and let d ϵ D be a decision for x.

Probability of error if we take decisions d:

$$P_d(error|x) = 1 - P(d|x)$$

Minimum probability of error:

$$\forall x \in X: P_*(error|x) = \min_{d \in D} P_d\,(error|x) = 1 - \max_{d \in D} P(d|x)$$

That is, for each x, the minimum probability of error is obtained if we take the decision with the highest (maximum) conditional (posterior) probability.

Minimum average probability of error:

$$P_*(error) = \sum_{x \in X} P_*(error|x)\,P(x)$$

Bayes decision rule for minimizing the probability of error (error risk):

$$\forall x \in X: d^*(x) = \operatorname*{argmax}_{d \in D} P(d|x)$$

CHAPTER 2.-LEARNING DISCRIMINANT FUNCTIONS: PERCEPTRON

1.-INTRODUCTION

The **pattern recognition goals** are:

- Modelling the process of **perception**:
  - Difficult to formalize (an expert can seldom put his/her perceptual skills into words).
  - Extraordinary **plasticity**: **learning** (unconsciously) through repeating exposition to the problems to solve (and their solutions).
- Modelling simple processes of **reasoning**.
- **Intrinsic impossibility** to reach exact results.
- Development of helpful systems to improve productivity and standard of living in general.

Design of a Pattern Recognition (PR) System:

- S(x): observable sign of an object x.
- Y(x): representation of x as a set of features describing its properties.
- **Response**: a class label, a complex structural information.
- **Learning**: on the basis of pairs input–output + domain knowledge of the task.

2.-REPRESENTATION SPACE

Representation space:

- Space: usually a feature vector in a n-dimensional feature space.
- An object representation is elicited via **pre-processing techniques** and **feature extraction**.

Feature extraction, desirable properties:

- **Continuity** and **discriminative capacity**: the similarity between the representations of two objects must be in direct correspondence with the similarity with which the objects are perceived.
- **Invariance under common transformations and distortions**: different instances of the same object should have similar representations.

Classes ($\mathbb{C}$):

- Each object (or its signal) is shown in a Primary Space or 'Universe', U.
- Let's assume that each object x ϵ U belongs to a unique class c(x) ϵ $\mathbb{C}$.
- $\mathbb{C}$ is the set of all possible identifiers or class labels.

Representation Space ($E = \mathbb{R}^D$):

- Let y = y(x) be the result of the preprocessing and feature extraction process applied to an object x ϵ U.
- E encloses all the possible results: {y : y = y(x), x ϵ U} ⊂ E.

- Given that two different objects from U can have the same representation in E, it is not guaranteed that each point in E belongs to a unique class.

Classifier ($G : E \rightarrow \mathbb{C}$):

- G is learnt with N labelled samples (y1, c1),…, (y_N, c_N) ϵ E x $\mathbb{C}$.
- For a new object x 2 U, its class is estimated as c^= c^(x) = G(y(x)). The goal is to obtain the correct class; that is, c^ = c(x), the maximum number of times as possible.

3.-DISCRIMINANT FUNCTIONS AND DECISIONS BOUNDARIES

Every classifier G into C classes can be stated in terms of C **discriminant functions** $g_c: E \rightarrow \mathbb{R}, 1 \leq c \leq C,$ and the corresponding classifying rule:

$$G = (g_1, g_2, \dots, g_C), c\text{^} = G(y) \equiv \underset{1 \leq c \leq C}{\operatorname{argmax}}\, g_c(y)$$

Any classifier partitions the representation space into C **decision regions**, R_1,…, R_C:

$$R_j = \{y \in E: g_j(y) > g_i(y)\ i \neq j, 1 \leq i \leq C\}$$

Decision boundary between two classes i, j: geometric place of the points y 2 ∈ for which $g_i(y) = g_j(y)$.

Decision boundary of a single class i: geometric place of the points y ∈ E for which:

$$g_i(y) \equiv \max_{j \neq i} g_j(y)$$

Two classifiers (g_1,…, g_C) and (g'_1,…, g'_C) are equivalent if they infer the same decision boundaries, that is:

$$g_i(y) > g_j(y) \Leftrightarrow g'_i(y) > g'_j(y)$$

Let $f: \mathbb{R} \rightarrow \mathbb{R}$ be any monotonically increasing function. Then, the following classifiers are equivalent:

(g_1,…, g_C), (f(g_1),…, f(g_C))

Linear discriminant functions:

$$g(y) = \sum_{i=1}^{d} a_i y_i + a_0 = a^t y + a_0$$

The **weight vector** is **a**, and $\mathbf{a_0}$ is the **threshold weight** or bias. The number of parameters is d+1. In general, **the decision boundaries are hyperplanes**.

Quadratic discriminant functions:

$$g(y) = \sum_{i=1}^{d}\sum_{j=1}^{d} a_{ij}y_i y_j + \sum_{i=1}^{d} a_i y_i + a_0 = y^t A y + a^t y + a_0$$

A is a matrix named **weight matrix** and $\mathbf{a_0}$, the **threshold weight**. The number of parameters has a **quadratic** relation with **d**. In general, **the decision boundaries are hyperquadratic**.

4.-LINEAR DISCRIMINANT FUNCTIONS(LDF)

A classifier is linear if its Discriminant Functions are **linear functions** of the vectors in E.
Let $y \in E \equiv \mathbb{R}^D$ be the representation of any object:

$$g_c(y) = \sum_{j=1}^{D} a_{cj}y_j + a_{c0} = a_c^t y + a_{c0}, 1 \leq c \leq C$$

Classification rule:

$$c\text{^} = G(y) \equiv \underset{1\leq c\leq C}{\operatorname{argmax}}\, a_c^t y$$

The **decision boundary** between any pair of classes i, j: $a_i^t y = a_j^t y$. These are **linear boundaries** or **hyperplanes** of dimension D.

5.-LEARNING LDF: PERCEPTRON

Given N training samples (y_1, c_1),…, (y_N, c_N), the goal is to find C weight vectors a_j, $1 \leq j \leq C, C$ that correctly classify (the most as possible) the given training samples.

Perceptron Algorithm:

// Let: $\mathbf{a}_j,\ 1 \le j \le C,\ \ C$ initial weight vectors;
// $(\mathbf{y}_1, c_1), \ldots, (\mathbf{y}_N, c_N),\ \ N$ training samples;
// $\alpha \in \mathbb{R}^{>0}$, 'learning rate';
// $b \in \mathbb{R}$, 'margin' (to tune the convergence).

do {
  $m = 0$ // number of correctly classified samples
  for $(n = 1;\ n \le N;\ n$++$)$ {
    $i = c_n$; $g = \mathbf{a}_i^t\, \mathbf{y}_n$; error=*false*
    for $(j = 1;\ j \le C;\ j$++$)$ ***if*** $(j \ne i)$
      {***if*** $(\mathbf{a}_j^t\, \mathbf{y}_n + b > g)$ $\{\mathbf{a}_j = \mathbf{a}_j - \alpha\, \mathbf{y}_n$; error=*true*$\}\}$
    if (error) $\mathbf{a}_i = \mathbf{a}_i + \alpha\, \mathbf{y}_n$; **else** $m = m + 1$
  }
} ***while*** $(m < N)$

In case of error the algorithm updates the weight vector of all the classes that produce the error, and the weight vector of the correct class.

The classification rule for C = 2, with weight vectors a_1, a_2:

$$c\text{^} = G(y) = \begin{cases} 1\ if\ a_1^t y > a_2^t y \\ 2\ else\ (a_1^t y \ge a_2^t y) \end{cases}$$

For this case, the decision boundary, the success criterion and the updating equations of the weight vectors are simplified by labelling the classes {1, 2} as {+1, -1} and by using a single weight vector a = a1 - a2:

- Classifier: $G(y) = \text{sgn}(a^t y)$.
- Decision boundary: $a^t y = 0$.
- Success criterion: $c_n a^t y_n > 0, n = 1, \ldots, N$.
- Leaning: $a = a + \alpha(c_n - G(y_n))y_n, n = 1, \ldots, N$.

Three parameters control the convergence and quality of results:

α, b(margin), M(maximum number of iterations)

α determines the size of the corrections and therefore the **learning speed**. In general, α <<, so soft convergence, but with more iterations.

With **linearly** separable training sets:

- Converges in a finite number of iterations ∀α > 0.
- b = 0: the boundary decisions might present little 'separation', that is, they might be too close to some data.
- b > 0: if b is large enough, the boundary decisions are centred across the decision regions (typically much better than with b = 0).

With **non-linearly** separable training sets:

- b = 0: no guarantee of convergence nor quality of the results.
- b>0: no convergence but with b and M large enough, good decision boundaries can be obtained; in general, (quasi-)optimal, with respect to the minimization of the classification error of the training set.

6.-EMPIRICAL ESTIMATION OF THE DECISION ERROR

Let p be the true probability of error of a decision system. An empirical estimate (p^) of p can be obtained by counting the number of decision errors, N_e, that occur in a test set of N data:

$$p\hat{} = \frac{N_e}{N}$$

If N >>, we can assume that p^ follows a normal distribution:

$$p\hat{} \sim \mathcal{N}\left(p, \frac{P(1-p)}{N}\right)$$

With a 95% of confidence interval:

$$P(p^{-\epsilon} \leq p \leq p^{+\epsilon}) = 0.95;\ \epsilon = 1.96\sqrt{\frac{p\hat{}(1-p\hat{})}{N}}$$

An automated learning system not only needs data to estimate the error but also data to learn the decision model. Given a set of labelled data, there are different methods to split it into a training set and a test set.

- **Re-substitution**: all available data are used for training as well as for testing. Inconvenient: it is (very) optimistic.
- **Hold out**: data are split into a training set and a test set. Inconvenient: it fails to use all the available data.
- **B-fold cross-validation**: data are randomly split into B equal subsets. Each subset is then used as test set for a system trained with the rest of sets. Inconvenient: the number of data in the training set can be small (specifically when the B is small) and the computation time increases in B.
- **Leaving One Out**: Each individual data (single observation) is used as test set for a system trained with the remaining n - 1 observations. It is the extreme case of B-fold cross-validation when B = n. Inconvenient: high computational cost.

CHAPTER 3.-CLASSIFICATION TREES

1.-DECISION AND CLASSIFICATION TREES (DCT)

Classification trees (also called **Decision trees** or Identification trees) is a **non-parametric approach** to Pattern Recognition; that is, the probability density of the features are not known a priori so we cannot make assumptions regarding the distribution of the features.

A classification tree is a structure that results from a recursive splitting of the representation space from a pattern or learning sample. Classification trees are a simple and effective knowledge representation method.

Method for classifying a pattern through a decision tree: ask a sequence of questions (queries), in which the next question depends on the answer to the current question. Questions are about the value of a feature (attribute or property) of the sample.

The sequence of questions is displayed in a directed decision tree, where the root node is displayed at the top, connected by successive (directional) links or branches to other nodes. These are similarly connected until we reach the terminal or leaf nodes which have no further links.

The root node and intermediate nodes contain a question about a particular feature (with a branch for each possible answer). Leaf nodes are labelled with a class. The reached leaf node will determine the class of the sample, for example, the final classification decision.

CART: partitions or splits are exclusively based on binary-valued properties, whose values are supported by solid statistical information.

The **decision regions** are formed by shaped **rectangular blocks** as the decision boundaries are always parallel to the axes.

Resubstitution error: misclassification error, the proportion of misclassified observations on the training set.

Notation:

- **Representation space**: $E \equiv \mathbb{R}^D; y = (y_1, y_2, \dots, y_D)^t \in E$.
- **Training set/N learning samples**: N feature vectors along with its correct classification: $(y_1, c_1), \dots, (y_N, c_N), \quad y_i \in E, c_i \in \mathbb{C} = \{1,2, \dots, C\}, 1 \leq i \leq N$.
- A tree is denoted by T:
  - A node is denoted by t.
  - The left child of t is denoted by t_L.
  - The right child of t is denoted by t_R.
  - The set of terminal or leaf nodes by T~.
- **T is a binary tree**: the outcomes of a query in a node t are 'yes/no'.
- A **binary split** is denoted by s and the set of admissible splits by S.

Let be:

- N the number of training samples.
- Nc the number of training samples that belong to class c.
- N(t) the number of training samples represented in node t.
- $N_c(t)$ the number of training samples in node t that belong to class c.

Prior probability of class c:

$$P\hat{}(c) = \frac{N_c}{N}$$

Conditional (posterior) probability of class c in node t:

$$P\hat{}(c|t) = \frac{N_c(t)}{N(t)}$$

Probability of a terminal node t ∈ T~:

$$P\hat{}(t) = \frac{N(t)}{N}$$

Probability of choosing the left node of t:

$$P\hat{}_t(L) = \frac{N(t_L)}{N(t)}$$

Probability of choosing the right node of t:

$$P\hat{}_t(R) = \frac{N(t_R)}{N(t)}$$

2.-LEARNING OF DCTS

Necessary elements to build a decision tree:

1. The method for splitting and selecting the best split; particularly:
    a. Query selection: decide which property test or query should be performed at each node. Without loss of generality, queries will be: $y \in B?, B \subseteq E$.
    b. Evaluation and optimization of the quality of a split.
2. Criteria that makes the data in a node t as pure (homogeneous) as possible so as to declare t as terminal node. We will define the **impurity**, rather than the purity of a node.
3. Criteria to assign a class label to a terminal node.

A query involves only one property j of E, $1 \leq j \leq D$. Queries like y ∈ B? are actually queries of the form $y_j \leq r?$. That is, a split is a pair s = (j, r) made up of a component, j ∈ {1,…,D}, and its corresponding threshold, $r \in \mathbb{R}$.

Since splits define hyperplanes parallel to the feature axes, the resulting partitions are hyper-parallelepiped blocks (B), which are rectangular in the case E = $\mathbb{R}^2$.

Since we work with a finite number of learning samples (N), there are only finitely many splits. For a node t with N(t) elements:

- We have to explore each feature $j, 1 \leq j \leq D$, from E.
- For each j, we have to analyse (at least) N(t) possible values of r.

Consequently, for each node t, we have to explore at least O(D N(t)) splits.

Entropy is a measure of the **uncertainty** associated to a decision between k classes (quantitative measure of the information not available when taking a decision):

$$H = -\sum_{i=1}^{k} P_i \log_2 P_i \qquad (0 \ \log 0 = 0)$$

In the case of two classes (k = 2) that have the same probability, the value of H is 1, the largest possible value of entropy (one bit, information associated to a binary decision). The minimum value of H is 0 and corresponds to a decision for which there is only one possible choice. The maximum value of H is +∞. This happens when taking a decision between k classes all of them with the same probability and with k -> ∞.

CHAPTER 4.- CLUSTERING UNSIPERVISED LEARNING: K-MEANS ALGORITHM

1.-INTRODUCTION

The **aim of clustering** is to group objects in classes so the objects belonging to the same class have a high degree of **natural association**, while the other classes are relatively different. The purpose is to create classes that are relatively different from each other. In other words, to find **natural groupings** of a set of input patterns so that the descriptions of these objects can be done in terms of classes or groups with strong internal similarities.

We need a distance (similarity) function to assess the quality of a clustering result (high similarity within a cluster and low similarity between clusters). There are **two types of clustering**: **partitional** and **hierarchical**.

2.-PARTITIONAL CLUSTERING

The **general problem** is that given a set of N objects or observations $(x_1, x_2, \dots, x_N)$, where each object is a d-dimensional real vector, the objective is to find a partition Π of the N objects into C classes or clusters. In order to do so, we assume there is available a **criterion function** J to evaluate the quality of a partition Π. Thus, the clustering problem can be seen as a search problem:

$$\Pi^* = \underset{\Pi=\{X_1,\dots,X_C\}}{\arg min}\ J(\Pi)$$

The best partition is the one that minimizes J for each partition.

The **difficulty** is that the number of partitions to evaluate is too high even for small values of N and C. Searching for global optimal solutions through enumeration techniques (explicitly or implicitly) is not feasible except fora few particular cases.

The **solution** is to search for suboptimal solutions obtained by approximate algorithms.

The **SSE** is a criterion to select a partition Π of N objects into C clusters. The objective is to select the partition that minimizes SSE. The SSE value of a partition Π of N objects $(x_1, x_2, \dots, x_N)$ into C clusters, $\Pi = \{X_1, \dots, X_C\}$, is computed as follows:

1. Each cluster X_C contains a subset of the N objects (disjoint sets).
2. For each cluster X_C, compute the squared errors for all $x \in X_C$: $SE_x = \|x - m_c\|^2$. m_c is the mean of points in X_C:
$$m_c = \frac{1}{|X_C|} \sum_{x \in X_C} x$$
m_c is interpreted as the natural prototype of X_C. Each object $x \in X_C$, is interpreted as a "distorted version" of m_c and the distortion of x is modelled by the error vector $x - m_c$.
3. For each cluster X_C, J_C is the function for cluster X_C. $J_C = WCSE_C$ (within-cluster squared errors)
$$J_C = WCSE_C = \sum_{x \in X_C} \|x - m_c\|^2$$

4. The SSE value of a partition Π is computed as:

$$SSE_{\Pi} = J(X_1, \dots, X_C) = \sum_C J_C = \sum_C WCSE_C$$

Now, we would have to calculate the SSE value for every partition (if they are explicitly given) and take the one that minimizes SSE (Π^*, best partition). SSE measures the sum (or average) of the squares of the magnitudes of the error vectors, and, obviously, it is a criterion to minimize. The mean of each cluster, m_c, is the point that represents the object of this cluster with the lowest SSE. m_c is the center of the cluster X_C (called centroid).

The SSE criterion is appropriate only when the objects form **hyper-spherics clusters of similar size**. If the sizes of the clusters are too different, it may happen that the natural grouping does not give the minimum SSE and so we would not find the natural clusters. Therefore, the centroid representation alone works well if the clusters are of the hyper-spherical shape. If clusters are elongated or are of other shapes, centroids are not sufficient.

3.-C-MEANS ALGORITHM

The movement of x from cluster X_i to X_j will be successful if the increment of SSE is negative. The **goal of the C-means algorithm** is that the data given by X is clustered by C-means algorithm, which aims to partition the N objects into C clusters such that the sum of the squared errors from points to the assigned cluster is minimized.

The algorithm is:

Algorithm K-*means* ("correct" version [Duda & Hart])
Input: $X = (x_1, x_2, \dots, x_N)$; C; $\Pi_0 = \{X_1, \dots, X_C\}$; (random initial partition)
Output: $\Pi^* = \{X_1, \dots, X_C\}$; $\boldsymbol{m}_1, \dots, \boldsymbol{m}_C$; $J(\Pi^*)$
for $c = 1$ **to** C **do** $\boldsymbol{m}_c = \frac{1}{n_c} \sum_{\boldsymbol{x} \in X_c} \boldsymbol{x}$ **endfor** (compute initial centroids for the clusters in input partition)
compute $J = J(\Pi_0)$
repeat
  $transfers$ = false
  forall $\boldsymbol{x} \in X$ (let $i : \boldsymbol{x} \in X_i$) **do** (for all the objects ... assume object x belongs to cluster Xi)
    if $n_i > 1$ **then** (if cluster Xi has more than one object, not only x)
      $j^* = \arg\min_{j \neq i} \frac{n_j}{n_j + 1} \|\boldsymbol{x} - \boldsymbol{m}_j\|^2$ (study x in any other cluster different from Xi; take the min. cluster X_{j^*})
      $\triangle J = \frac{n_{j^*}}{n_{j^*} + 1} \|\boldsymbol{x} - \boldsymbol{m}_{j^*}\|^2 - \frac{n_i}{n_i - 1} \|\boldsymbol{x} - \boldsymbol{m}_i\|^2$ (increment of SSE, difference bt. X_{j^*} and current cluster)
      if $\triangle J < 0$ **then**
        $transfers$ = true (move x to X_{j^*})
        $\boldsymbol{m}_i = \boldsymbol{m}_i - \frac{\boldsymbol{x} - \boldsymbol{m}_i}{n_i - 1}$ $\boldsymbol{m}_{j^*} = \boldsymbol{m}_{j^*} + \frac{\boldsymbol{x} - \boldsymbol{m}_{j^*}}{n_{j^*} + 1}$
        $X_i = X_i - \{\boldsymbol{x}\}$ $X_{j^*} = X_{j^*} + \{\boldsymbol{x}\}$
        $J^* = J + \triangle J$ $J = J^*$ (J value of the best partition so far ...)
      endif
    endif
  endforall
until $\neg transfers$ Cost per iteration: $O(N \cdot C \cdot D)$, $N = |X|$, D = dimension

Another version of the algorithm is:

Algorithm C-*means* ("popular" version)
Input: $X; C; \Pi = \{X_1, \ldots, X_C\}$;
Output: $\Pi^* = \{X_1, \ldots, X_C\}; \boldsymbol{m}_1, \ldots, \boldsymbol{m}_C$
repeat
  $transfers =$ false
  for $c = 1$ **to** C **do** $\boldsymbol{m}_c = \frac{1}{n_c} \sum_{\boldsymbol{x} \in X_c} \boldsymbol{x}$ **endfor**
  forall $\boldsymbol{x} \in X$ (let $i : \boldsymbol{x} \in X_i$) **do**
    if $n_i > 1$ **then**
      $j^* = \arg\min_{1 \le j \le C} d(\boldsymbol{x}, \boldsymbol{m}_j)$ (Euclidean distance from x to each centroid; $d(\boldsymbol{x}, \boldsymbol{m}_j) = \|\boldsymbol{x} - \boldsymbol{m}_j\|$)
      (study x in all clusters, including current cluster, and take the min. cluster)
      if $j^* \neq i$ **then** (if the min. cluster is different from the current cluster, move x to the min. cluster)
        $transfers =$ true
        $X_i = X_i - \{\boldsymbol{x}\}; X_{j^*} = X_{j^*} + \{\boldsymbol{x}\}$
      endif
    endif
  endforall
until $\neg transfers$ (Cost per iteration: $O(N \cdot C \cdot D)$, $N = |X|$, D = cost of $d(\cdot, \cdot)$)

The "popular" version does not apply increments/decrements to the mean (centroid) nor the SSE. It simply takes the Euclidean distance from x to each centroid (not the squared Euclidean distance), moves x to the minimum cluster and updates the objects in the current and minimum cluster. When all x are studied, it re-computes the mean points of all the clusters again.

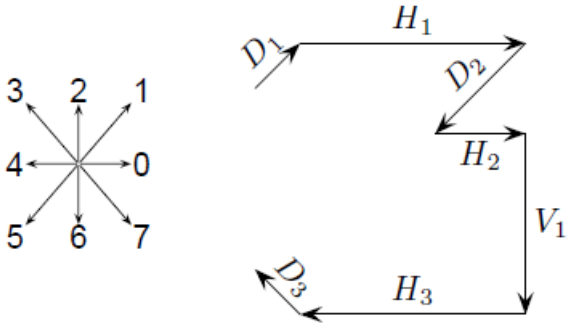
CHAPTER 5.-SYNTACTICAL/STRUCTURAL METHODS. MARKOV MODELS

1.-STRUCTURED REPRESENTATION

Syntactic or structural Pattern Recognition makes use of symbolic and structured information. The goal is to find a clear structure in the patterns. Representing patterns through numerical feature vectors may result in a **loss of information for some applications**. The **solution** is to use a structured representation like sequences of vectors or symbols of variable length. **Modelling** is done by using structural models.

2.-NEED FOR PROBABILITIES: STOCHASTIC GRAMMARS

Let's assume we have a grammar like this one, Gc, for each digit c.



$$
\begin{array}{lcl}
S & \Rightarrow & D_1 \\
D_1 & \Rightarrow & 1\,H_1 \mid H_1 \\
H_1 & \Rightarrow & 0\,H_1 \mid 0\,D_2 \\
D_2 & \Rightarrow & 5\,D_2 \mid 5\,H_2 \\
H_2 & \Rightarrow & 0\,H_2 \mid 0\,V_1 \\
V_1 & \Rightarrow & 6\,V_1 \mid 6\,H_3 \\
H_3 & \Rightarrow & 4\,H_3 \mid 4\,D_3 \\
D_3 & \Rightarrow & 3 \mid \lambda
\end{array}
$$

If $x \in \{0,1,\ldots,7\}^+$ represents a digit stroke, we can decide which class x belongs to by using a 'pure' syntactic classifier:

$$
c(x) = \begin{cases} c \text{ if } G_c \text{ is the only grammar that generates } x \\ \text{"reject" if no grammar generates } x \\ \text{"doubt" if more than one grmmar generates } x \end{cases}
$$

The need for inclusion of the classes "reject" and "doubt" is a clear drawback. Another drawback is that conventional grammar generates natural outcomes as well as "undesirable" outcomes. The common solution is to introduce probabilities in grammars, that is, **stochastic grammars**.

A **stochastic grammar** G' is a grammar G with probabilities associated to its rules. It is **consistent** if:

$$
\sum_{y \in \sum *} p(y|G') = 1
$$

Learning the rules of their "topology" is difficult to automate, so instead, we use an approach, that is, a pre-defined topology. In our case will use **Markov Models**.

Learning the probabilities (estimation) can be done in two ways:

- **Non-ambiguous context-free (or regular) grammars G'**: maximum likelihood estimation from the frequencies of use of the rules during the parsing of a sequence of training strings supposedly generated by G'. These estimations get close to the true probabilities when the number of training strings approaches infinite.
- **Ambiguous regular grammars and/or Markov models**: locally optimal estimation by using "Viterbi reestimation" or the "Backward-Forward" algorithm.

3.-MARKOV MODELS

A Markov models is a tuple $M = (Q, \Sigma, \pi, A, B)$ where:

- $\boldsymbol{Q}$ is a **set of states**: The model is in a state q_t at every instant $t = 1,2, \dots, M$. Q includes a final state F.
- $\boldsymbol{\Sigma}$ is a **set of "observable" symbols** (observations): the models emit a symbol y_t at every instant $t = 1,2, \dots, M$.
- $\boldsymbol{\pi} \in \mathbb{R}^Q$ is the ini**tial probability vector**: M chooses q_1 using π.
- $\boldsymbol{A} \in \mathbb{R}^{QxQ}$ is a **transition probability matrix** (between states): M chooses q_{t+1} using q_t and A: $A_{q,q'} = P(q_{t+1} = q' | q_t = q, A)$.
- $\boldsymbol{B} \in \mathbb{R}^{Qx\Sigma}$ is an **observation/emission probability matrix** (probability that a state emits an observable symbol):M chooses $y_t$using q_t and B: $B_{q,\sigma} = P(y_t = \sigma | q_t = q, B)$.

Normalization conditions for π, A, B:

- Probability of the initial state: $0 \leq \pi_q \leq 1, \sum_{q \in Q} \pi_q = 1,\ \pi_F = 0$.
- Probabilities of transition between states: $0 \leq A_{q,q'} \leq 1, \sum_{q' \in Q} A_{q,q'} = 1,\ A_{F,q} = 0$.
- Probabilities of emitting observable symbols: $0 \leq B_{q,\sigma} \leq 1, \sum_{\sigma \in Q} B_{q,\sigma} = 1,\ B_{F,\sigma} = 0$.

A stochastic process has the Markov property if the conditional probability distribution of future states **depends only upon the present state**, not on the sequence of states that preceded it. Example of generating a sequence of states:

Probability of generating the sequence of states 123F: P(123F) = P(F|3)*P(3|2)*P(2|1)*P(1)

To calculate the **probability of generating a string with a sequence of states**, let $M = (Q, \Sigma, \pi, A, B)$ be a Markov model with a final state q_F:

1. Choose an initial state $q \in Q$ using $P(q) \equiv \pi_q$.
2. Select an observation $\sigma \in \Sigma$ using $P(\sigma|q) \equiv B_{q,\sigma}$; emit σ.
3. Choose the next state $q' \in Q$ using $P(q'|q) \equiv A_{q,q'}$.
4. If $q = q_F$ end; else, go to step 2.

An example would be:

P(cba, 113F) = P(cba|113F)*P(113F) = P(c|1)*P(b|1)*P(a|3)*P(113F) =
P(c|1)*P(b|1)*P(a|3)*P(F|3)*P(3|1)*P(1|1)*P(1)

With a Markov Model the following properties hold:

$$0 \leq P(y|M) \leq 1, \sum_{y \in \Sigma +} P(y|M) = 1$$

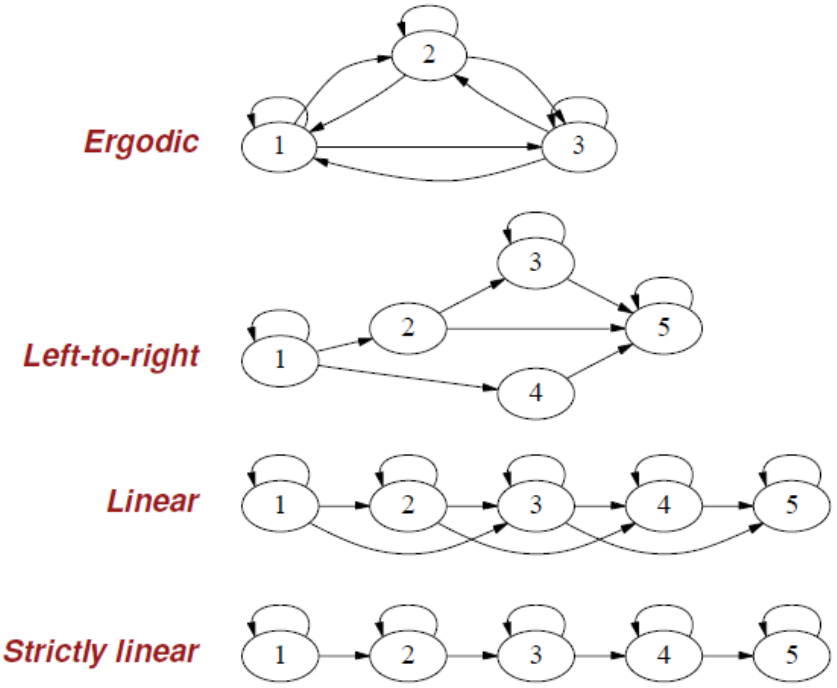
A sequence of **steps to generate a string with a Markov Model** are:

1. We have to find all possible sequences of states that generate the string.
2. We annotate the states that can generate the symbols of the string.
3. We erase the states in the final observation that doesn't reach the final state F.
4. We erase the states in the first observation that aren't initial ones.
5. We discard combinations that are not possible.
6. The probability is the sum of all combinations of the remaining states.

4.-TOPOLOGY OF A MARKOV MODEL

The topology of a Markov model is the underlying graph. It is determined by the structure (number and location of zeroes in the state-transition matrix A). The most common topologies are:

- **Ergodic**: complete graph, no zeroes in A.
- **Left-to-right**: the graph is directed and acyclic (DAG), though there can be individual loops in the states. A is triangular.
- **Linear**: the graph is a restricted DAG (possibly with loops in the states) where transitions leaving the i-th state can only reach states i + 1,…,i + k. The non-null elements in A are in k + 1 adjacent diagonals. These transitions are called skips.
- **Strictly linear**: the graph is a concatenation of states (possibly with loops in the states). The non-null elements in A are in two adjacent diagonals.



CHAPTER 6.-FORWARD AND VITERBI ALGORITHMS

1.-FORWARD ALGORITHM TO CALCULATE P(y|M)

We define α(q, t) as the probability that a Markov model M generates the sub-string $y_1 \cdots y_t$ and reaches the state q at instant t:

$$\alpha(q,t) = \sum_{\substack{q_1,\ldots,q_t \\ q_t=q}} P(y_1 \cdots y_t, q_1, \ldots, q_t)$$

In general:

$$\alpha(q,t) = \begin{cases} \pi_q * B_{q,y_1} \; if \; t = 1 \\ \sum_{q' \in Q} \alpha(q', t-1) * A_{q',q} * B_{q,y_t} \; if \; t > 1 \end{cases}$$

The temporal complexity of the forward algorithm is 0(mb), where m is the string length and b is the number of state transitions.

2.-VITERBI ALGORITHMS TO APPROXIMATE P(y|M)

Trying to find the probability of string y by means of **considering all state sequences** is **impractical**. The **solution** is to use the **Viterbi approximation** to a P(y|M), that is, to calculate the most likely sequence of states for generating y:

$$\tilde{P}(y|M) = \max_{q_1,\ldots,q_m \in Q^+} P(y, q_1, \ldots, q_m)$$

The corresponding most probable sequence of states is:

$$\tilde{q} = (\tilde{q}_1, \ldots, \tilde{q}_m) = \operatorname*{argmax}_{q_1,\ldots,q_m \in Q^+} P(y, q_1, \ldots, q_m)$$

We define $V(q,t)$ as the maximum probability that a Markov model reaches state q at instant t and emits the string $y = y_1 \ldots y_t$:

$$V(q,t) = V(q,|y|) = \max_{\substack{q_1,\ldots,q_t \\ q_t=q}} P(y_1 \ldots y_t, q_1, \ldots, q_t)$$

In general:

$$V(q,t) = V(q,|y|) = \begin{cases} \pi_q * B_{q,y_1} \; if \; t = 1 \\ \max_{q' \in Q} V(q', t-1) * A_{q',q} * B_{q,y_t} \; if \; t > 1 \end{cases}$$

Now, we can replace the calculation of P(y|M) by the Viterbi approximation:

$$\tilde{P}(y|M) = \max_{q \in Q} V(q, |y|) * A_{q,F}$$

In other words, rather than finding all the state sequences that generate the string y, when using Viterbi approximation we only consider the most probable state sequence (optimal state sequence), which is the one that maximizes the expression $\max_{q \in Q} V(q, |y|) * A_{q,F}$.

The temporal complexity of Viterbi is O(mb) where m is the length of the string and b is the number of state transitions.

3.-SYNTACTIC-STATISTICAL CLASSIFICATION

Assume that we have C classes of objects which are represented as strings from Σ^+. That is, one class of strings ($c \in C$) is characterized by a Markov model (M_C) that generates the strings of the class. The **Syntactic-statistical classification** is the most probable class c (Markov model M_C) for string y.

We can use a similar approach as the statistical classification for the feature vector case. That is:

- We are given the prior probability of each class, $P(c)$.
- We know the conditional probability of each class c, we calculate this with the Forward algorithm, or we approximate the value by Viterbi. We have to compute this for every class.

Then:

- We have to compute the posterior probability of class c, $P(c|y)$ by applying Bayes.
- We apply the classification rule that returns the most probable class.

Summary:

- **Prior probability** of a class c: $P(c), 1 \leq c \leq C$.
- **Conditional probability** of class c: $P(y|M_C)$. It is the probability of obtaining string y given that is generated by the Markov model M_C. It is a probability function that models the distribution of strings of c in $\sum*$ through the Markov model M_C.
- **Posterior probability** of a class c: $P(c|y)$. It is the probability that the string y belongs to class c:

$$P(c|y) = \frac{P(y|M_C) * P(c)}{P(y)} \; where \; P(y) = \sum_{c'=1}^{C} P(y|M_{C'}) * P(c')$$

- **Classification rule**: a string $y \in \sum +$ is assigned to a class $c\hat{}(y)$:

$$c\hat{}(y) = \underset{1 \leq c \leq C}{\operatorname{argmax}} P(c|y)$$

CHAPTER 7.-ESTIMATION OF MARKOV MODELS

1.-LEARNING: ESTIMATION OF PROBABILITIES OF A MARKOV MODEL

Basic problem:

- Estimate/learn the probabilities/parameters of a Markov model M; that is, learn A, B and π.
- Use a set of training strings $Y = \{y_1, \dots, y_n\}$ drawn independently according to the probability rule $P(y|M)$.

The **basic idea** of the estimation by Viterbi algorithm is to parse all the strings in Y, counting the **frequencies of use of transitions between states**, fr**equencies of generation of symbols in each state**, etc., and normalize to obtain the probabilities. The **problem** is how to parse a string if the model probabilities are not known and so we cannot calculate the sequence of states. A **possible solution** would be:

1. Initialize the model probabilities "properly" (we obtain an initial Markov model).
2. Parse each string in y ∈ Y using the Viterbi algorithm and obtain the corresponding sequence of states.
3. Count the required frequencies (transitions between states -A-, generation of symbols in each state -B-) for the sequence of states of each string.
4. Normalize frequencies to obtain the new model probabilities.
5. Repeat steps 2-4 until convergence (the model probabilities converge).

This is called **re-estimation**: we are given an initial Markov model M with initial values A, B, and π, and we have to **re-estimate** these values by parsing a set of input strings.

The **Viterbi re-estimation algorithm** is:

Input: $M^0 = (Q^0, \Sigma^0, \pi^0, A^0, B^0)$ /* Initial model */
$Y = \{y_1, \dots, y_n\}$ /* training sets */
Output: $M = (Q, \Sigma, \pi, A, B)$ /* Optimized model */

$M = M^0$
repeat $M' = M;\ \pi = 0;\ A = 0; B = 0$
  for $k = 1$ **to** n **do**
    $m = |y_k|$ /* most probable state sequence for y_k, */
    $\tilde{q}_1, \dots, \tilde{q}_m = \text{argmax}_{q_1,\dots,q_m} P(y_k, q_1, \dots, q_m \mid M')$ /* by Viterbi */
    $\pi_{\tilde{q}_1}$++; $B_{\tilde{q}_1, y_{k,1}}$++ /* counter update */
    for $t = 2$ **to** m **do** $A_{\tilde{q}_{t-1},\tilde{q}_t}$++; $B_{\tilde{q}_t, y_{k,t}}$++ **done**; $A_{\tilde{q}_m, F}$ ++
  done
  $s = \sum_{q \in Q} \pi_q$
  forall $q \in Q$ **do** /* counter normalization */
    $\pi_q = \pi_q / s$
    $a = \sum_{q' \in Q} A_{q,q'}$; **forall** $q' \in Q$ **do** $A_{q,q'} = A_{q,q'}/a$
    $b = \sum_{\sigma \in \Sigma} B_{q,\sigma}$; **forall** $\sigma \in \Sigma$ **do** $B_{q,\sigma} = B_{q,\sigma}/b$
  done
until $M = M'$

2.-INITIALIZATION OF VITERBI RE-ESTIMATION

If we are not given an initial Markov model M or if the model probabilities are unknown, we compute the optimal sequences of states for each string by initializing all probabilities following a uniform distribution. The problem is that this usually produces convergence problems or a convergence to inadequate local-maxima. **A useful idea for linear or left-to-right models**:

- Split each string in Y in as many segments (approximately) as states in the Markov model.
- Assign the symbol of each segment to one state.
- Count the frequencies of transition and generation.
- Normalize frequencies to obtain the required initial probabilities.

The **initialization by linear segmentation for Viterbi re-estimation** is:

Input: $Y = \{y_1, \ldots, y_n\}$, N /* training strings, number of states */
Output: $M = (Q, \Sigma, \pi, A, B)$ /* model */

$Q = \{1, 2, \ldots, N, F\}$; $\Sigma = \{y \in y_k \in Y\}$ /* states and symbols */
$\pi = 0$; $A = 0$; $B = 0$ /* initialization of counters*/

for $k = 1$ **to** n **do** /* counter updating using */
  $q = 1$; π_q++; $B_{q,y_{k,1}}$++ /* linear alignment of y_k with the states */
  for $t = 2$ **to** $|y_k|$ **do** $q' = q$; $q = \left\lfloor \frac{t}{|y_k|+1} N \right\rfloor + 1$; $A_{q',q}$++; $B_{q,y_{k,t}}$++ **done**
  $A_{q,F}$ ++
done

$s = \sum_{q \in Q} \pi_q$
forall $q \in Q$ **do** /* counter normalization */
  $\pi_q = \pi_q \,/\, s$
  $a = \sum_{q' \in Q} A_{q,q'}$; **forall** $q' \in Q$ **do** $A_{q,q'} = A_{q,q'} \,/\, a$
  $b = \sum_{\sigma \in \Sigma} B_{q,\sigma}$; **forall** $\sigma \in \Sigma$ **do** $B_{q,\sigma} = B_{q,\sigma} \,/\, b$
done