Pattern Analysis - Lecture notes

Sebastian Rietsch

July 27, 2017

Contents

[1] Bishop, Pattern Recognition: 8.3.3

1 Density Estimation

Let $p(\vec{x})$ denote a probability density function (pdf) then:

1. $p(\vec{x}) \geq 0$
2. $\int\limits_{-\infty}^{+\infty} p(\vec{x}) d\vec{x} = 1$
3. $p(\vec{a} \leq \vec{x} \leq \vec{b}) = \int\limits_{\vec{a}} p(\vec{x}) d\vec{x}$

The task of density estimation is to obtain from a set of discrete samples/measurements a continuos representation of the underlying pdf.

1.1 Parametric density estimation

Make an assumption about the underlying distribution and determine the best fitting distribution parameters from the data. (MLE, Maximum a Positriori Estimation)

1.2 Non-parametric density estimation: Parzen-Rosenblatt estimator

The Parzen window estimator interpolates the pdf from the observations in the neighborhood of a position $\vec{x}$, using an appropriate kernel (or window) function.

Short derivation

Let p_R denote the probabilty that $\vec{x}$ lies within region R: $p_R = \int_R p(\vec{x}) d\vec{x}$.
Now assume that $p(\vec{x})$ is approximately constant in R.

$$\Rightarrow p_R = p(\vec{x}) \underbrace{\int_R d\vec{x}}_{\text{volume of R}}$$

For example let R be a d-dimensional hypercube with side length h, then its volume is h^d and $p_R \approx p(\vec{x}) \cdot V_R$.
Let $p_R = \frac{k_R}{N}$, i.e. we determine the probability of making an observation in region R by counting the samples in R (k_R) and dividing by the total number of samples. (p_R is also called the "relative frequency")

$$\Rightarrow p(\vec{x}) = \frac{p_R}{V_R} = \frac{k_R}{V_R \cdot N}$$

Let us write the Parzen window estimator as a function of a kernel $k(\vec{x}, \vec{x_i})$, then

$$p(\vec{x}) = \frac{1}{Nh^d} \cdot \sum_{i=1}^{N} k(\vec{x}, \vec{x_i})$$

and

$$k(\vec{x}, \vec{x_i}) = \begin{cases} 1 & \frac{|\vec{x}_{i,k} - \vec{x}_k|}{h} \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$

.
In any dimension k, $\vec{x_i}$ and $\vec{x}$ are not farther apart than $0.5 \cdot h$.

Equivalently, if we use a (multivatiate) Gaussian kernel:

$$k(\vec{x}, \vec{x_i}) = \frac{1}{\underbrace{(2\pi)^d|\Sigma|}_{\text{Volume of Gaussian}}} \quad e^{-(\vec{x}-\vec{x_i})^T\Sigma^{-1}(\vec{x}-\vec{x_1})} \quad \Rightarrow \quad p(\vec{x}) = \frac{1}{N}\sum_{i=1}^{N} k(\vec{x}, \vec{x_i})$$

A note on application

- General remark: we obtain a continous pdf, i.e. density estimation converts a list of measurements to a statistical model
- One specific example: we can sample from a pdf. This means that we have a principled way of generating new/more ... data that behaves/looks similarly to the observations

Question: How can we (practically) sample from a pdf?

1. Convert pdf into cumulative density function (cdf) ($cdf[i] = cdf[i-1] + pdf[i]$)
2. Draw a uniformly distributed number (r) between 0 and 1
3. Our sampled value is x, where $cdf[x] = r$

How can we determine good window size?

Let's do a Maximum Likelihood with cross validation. e.g. leave-one-sample-out cross-validation (CV).

$$p^j_{h,N-1}(\vec{x}) = \frac{1}{Nh^d} \sum_{i=1, i\neq j}^{N} k(\vec{x}, \vec{x_i})$$

We estimate the pdf from all samples except $\vec{x_j}$. $\vec{x_j}$ will be used to evaluate the quality of the pdf using window size h.

$$\begin{aligned} \hat{h} &= \underset{h}{\operatorname{argmax}}\, \mathcal{L}(h) = \underset{h}{\operatorname{argmax}} \prod_{j=1}^{N} p^j_{h,N-1}(\vec{x_j}) \\ &= \underset{h}{\operatorname{argmax}} \sum_{j=1}^{N} log\ p^j_{h,N-1}(\vec{x_j}) \end{aligned}$$

The position of the maximum (when using log-likelihood) does not change, because the logarithm is a strictily monotonic function.

2 Mean Shift Algorithm (Comaniciu, Meer)

Purpose: Find maxima in a pdf without actually performing a full density estimation (e.g. for Clustering (maximum is cluster center), segmentation, ...)

Assume that we have a full density estimator. Then

$$p(\vec{x}) = \frac{1}{N} \sum_{i=1}^{N} k_h(\vec{x}, \vec{x_i})$$

.

Idea: Maxima can be found where the gradient of the pdf is zero.

Problems:

- It may be the case, that a zero gradient is just between two finer maxima
- The kernel size indirectly controls the number of identified maxima

Applications:

- Clustering: Find local maxima, create one cluster per maxima and assign every vector to the maxima it "walks up to" (gradient points in direction of maxima)
- Image smoothing: define features as $x_i = (x, y, L, u, v)^T$, assign pixels forming a cluster to the mean Luv-value

3 Mean Shift Algorithm (continued)

Let

$$p(x) = \frac{1}{N} \sum_{i=1}^{N} K_h(x_i, x)$$

denote the multivariate kernel density estimate.

A local maximum of the PDF can be assumed where the gradient $\triangledown p(x) = 0$ ("the gradient vanished", local minimum not possible because we perform a grandient asscend).

$$\triangledown p(x) = \triangledown(\frac{1}{N} \sum_{i=1}^{N} K_h(x_i, x)) = \frac{1}{N} \sum_{i=1}^{N} \triangledown K_h(x_i, x)$$

Let us assume that K_h is a radially symmetric kernel, i.e.

$$K_h(x_i, x) = c \cdot k_h(\|x_i - x\|^2)$$

where c is a constant that adjusts this quantity towards d dimensions.

$$\frac{\partial k_h(s)}{\partial s} = k_h'(s)$$

$$\frac{\partial k_h(s)}{\partial x} = \frac{\partial (x_i - x)^T (x_i - x)}{\partial x} = -2(x_i - x)$$

$$\Rightarrow \triangledown p(x) = \frac{1}{N} \cdot \sum_{i=0}^{N} c_d \cdot k_h'(\|x_i - x\|^2) \cdot (-2(x_i - x)) = \vec{0}$$

(drop $\frac{1}{N}, c_d$), multiply)

$$\Leftrightarrow \sum_{i=1}^{N} k_h'(\|x_i - x\|^2) \cdot x_i - \sum_{i=1}^{N} k_h'(\|x_i - x\|) \cdot x = \vec{0}$$

Mean shift vector:

$$\frac{\sum_{i=1}^{N} k_h'(\|x_i - x\|^2) \cdot x_i}{\sum_{i=1}^{N} k_h'(\|x_i - x\|^2)} - x = \vec{0}$$

Mean Shift Algorithm:

1. Compute mean shift vector
2. Update x:

$$\begin{aligned} x^{(t+1)} &= x^{(t)} + m(x^{(t)}) \\ &= x^{(t)} + \frac{\sum_{i=1}^{N} k_h'(\|x_i^{(t)} - x^{(t)}\|^2) \cdot x_i^{(t)}}{\sum_{i=1}^{N} k_h'(\|x_i^{(t)} - x^{(t)}\|^2)} - x^{(t)} \end{aligned}$$

Why is it called "Mean Shift"? Because if we plug in for k_h the Epanechnikov kernel, the whole computation breaks down to computing for the update the mean of the samples in a circular or (hyperspherical, resp.) neighborhood around $x^{(t)}$.

Epanechnikov kernel: $k_E(x) = \begin{cases} c \cdot (1 - x^T x) & x^T x \leq 1 \\ 0 & otherwise \end{cases}$

Example applications:

1. (Color) quantization
2. Segmentation: similar in result to a superpixel segmentation: operate locally in the image ⇒ incorporate the position of each pixel. Thus, the feature vector becomes 5-dimensional, consisting of (x, y, r, g, b). Properly scale each feature dimension, such that distances are comparable between them.

Remarks on the found maxima:

1. Different trajectories typically converge only to almost the same peak, thus we will have to postprocess the peaks and (somewhat) reduce them
2. We do not have a guarantee to sit on top of a maximum when reaching a 0-gradient. This is due to the finite window size and the discrete representation of our density
3. If the amount of data is larger, then it may become extremely costly to iteratively evaluate the "neighborhood finder." In that case, you have to help yourself either with a smart data structure or LSH (Locality-Sensitive Hashing)

Note: RGB color space is not perceptually uniform. This means the same Euclidean distance between two points looks very different for different point pairs in RGB space. Standard way out: Lab or Luv color space.

4 Clustering

Summary: Elements of Statiscial Learning, Section 14

4.1 Cluster Analysis

Grouping or segmenting a collection of objects into subsets or "clusters", such that those within each cluster are more closely related to one another than objects assigned to different clusters. Sometimes the goal also is to arrange the clusters into a natural hirarchy.

All clustering methods attempt to group the objects based on the definition of similarity supplied to it → choice of distance or dissimilarity measure between two objects.

4.1.1 Proximity Matrices

Sometimes the data is represented directly in terms of the proximity (alikeness or affinity) between pairs of objects. These can either be *similarities* or *dissimilarities*. This type of data can be represented by an $N \times N$ matrix D, where N is the number of objects, and each element $d_{ii'}$ records the proximity between the ith and i'th objects.

Most algorithms presume a symmetric matrix of dissimilarities with nonnegative entries and zero diagonal elements. If the original data were collected as similiarities, a suitable monotone-decreasing function can be used to convert them to dissimilarities.

4.1.2 Dissimilarities Based on Attributes

Most often we have measurements x_{ij} for $i = 1, \dots, N$, on variables $j = 1, \dots, p$ (also called *attributes.* We first have to construct pairwise dissimilarities between the observations. In the common case, we define a dissimilarity $d_j(x_{ij}, x_{i'j})$ (e.g. squared distance $(x_{ij} - x_{i'j})^2$, even though not appropriate for nonquantitive/categorial attributes) between values of the jth attribute, and then define

$$D(x_i, x_i') = \sum_{j=1}^{p} d_j(x_{ij}, x_{i'j})$$

- Ordinal variables: Convert to numbers on a scale
- Categorial(nominal) variables: Dissimialrity has to be described explicitly

4.1.3 Object Dissimilarity

For comparing objects, often a weighted sum of distances is computed:

$$D(x_i, x_i') = \sum_{j=1}^{p} w_j d_j(x_{ij}, x_{i'j}); \sum_{j=1}^{p} w_j = 1$$

Note that giving every variable the same weight w_j doesn't necessarily give all attributes equal influence. The relative influence of the jth variable is given by $w_j \cdot \bar{d}_j$, where $\bar{d}_j = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} d_j(x_{ij}, x_{i'j})$ is the average dissimilarity on the jth attribute. To achieve this the weights have to be set to $w_j = \frac{1}{\hat{d}_j}$, which doesn't always make sense.

Specifying an appropriate dissimilarity measure is far more important in obtaining success with clustering than choice of clustering algorithm.

4.1.4 Clustering Algorithm

Three types of clustering algorithms:

- **Combinatorial:** work directly on the observed data with no direct reference to an unterlying probability model.
- **Mixture modeling:** suposes that the data is a sample from some population describe by an probability density function. This density function is characterized by a parameterized model taken to be a mixture of component density functions; each component density describes one of the clusters.
- **Mode seeking("bump hunting"):** take a nonparamteric perspective, attempting to directly estimate distinct modes of the pdf. Observations "closest"to each respective mode then define the individual clusters.

4.1.5 Combinatorial Algorithms

Each observation is uniquely labeled by an integer $i \in \{1, \dots N\}$. A prespecified number of clusters $K < N$ is postulated, and each one is labeled by an integer $k \in \{1, \dots, K\}$. Each observation is assigned to one and only one cluster by an "encoder" $C(i) = k$, that assigned the ith observation to the kth cluster. One seeks the particular encoder $C^*(i)$ that achieves the required goal, based on the dissimilarities between every pair of observations.

One approach is to directly specify a mathematical loss function and attempt to minimize it through some combinatorial optimization algorithm. Since the goal is to assign close points to the same cluster, a natural loss function would be

$$W(C) = \frac{1}{2}\sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(i')=k} d(x_i, x_{i'})$$

which is also called the *within-cluster scatter*. One can equivalently maximize the *between-cluster scatter*

$$B(C) = \frac{1}{2}\sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(i')\neq k} d(x_i, x_{i'})$$

One simple way to solve this is to minimizes W or maximized B over all possible assignments of the N data points to K clusters. Unfortunally, such optimization by complete enumeration is feasible only for very small data sets.

For this reason, practical clustering algorithms are able to maximize only a very small fraction of all possible encoders C. The goal is to identify a small subset that is likely to contain the optimal one, or at least a good suboptimal partition. Such feasible stratgies are based on iterative greedy descent (initial partition is specified, at each step cluster assignments are changed such that value of criteroen is improved, stop when no more improvement possible).

4.1.6 K-means

Squared Euclidean distance:

$$d(x_i, x_{i'}) = \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2 = \|x_i - x_{i'}\|^2$$

The within-point scatter can be written as

$$W(C) = \sum_{k=1}^{K} N_k \sum_{C(i)=k} \|x_i - \bar{x_k}\|^2$$

where $\hat{x_k}$ is the mean vector associated with the kth cluster, and N_k is the number of observations assigned to a cluster. Thus, the critereon is minimized by assigning the N observations to the K clusters in such a way that within each cluster the average dissimilarity of the observations from the cluster mean, as defined by the points in that cluster, is minimized.

Algorithm:

1. For a given cluster assignment C, compute each clusters mean
2. Assign each observation to the closest cluster mean
3. Iterate 1 and 2 until the assignments do not change

Each iteration reduces the value of the critereon, so that convergence is assured, but the solution may be suboptimal. In addition, on should start the algorithm with many different random choices for the starting means, and choose the solution having smallest value of the objective function.

4.1.7 Gaussian Mixtures as Soft K-means Clustering

The K-means clustering procedure is closely related to the EM algorithm for estimating a certain Gaussian mixture model. Suppose we specify K mixture components, each with a Gaussian density having a scalar covariance matrix $\sigma^2 I$. Then the relative density under each mixture component is a monotone function of the Euclidean distance between the data point and the mixture center. Hence in this setup EM is a "soft" version of K-means clustering, making probabilistic assignments of points to cluster centers. As the variance $\sigma^2 \to 0$, these probabilities become 0 and 1, and the two methods coincide (responsibility of mixture component $i : \frac{g_i(x)}{\sum_m g_m(x)}$).

4.1.8 Vector quantization

The K-means clustering algorithm represents a key tool in the apparently unrelated area of image and signal compression, particular in *vector quantization* or *VQ*(Gersho and Gray, 1992).

Algorithm:

1. Convert to grayscale
2. Break image into small blocks, e.g. 2 × 2 blocks of pixels
3. Each block is regarded as a vector in $\mathcal{R}^4$
4. Apply K-means
5. Each of the blocks is approximated by its closest cluster centroid, known as codeword. The clustering process is called the *encoding* step, and the collection of centroids is called the *codebook.*

4.1.9 K-medioids

$K-$means stuggels with outliers, because using *squared* Euclidean distance places the highest influence on the larger distances.

Instead of taking means as cluster center, one can identify the sample inside each cluster, which is nearest to all other samples inside it. One then assigns this sample to be the new cluster center.

The downside is that K-medioids is far more computationally intensive.

4.1.10 Practical Issues

In order to use K-means or -medioids one must select the number of clusters K^* as initialization. A choice for the number of clusters K depends on the goal. For data segmentation K is usually defined a part of the problem.

Data-based methods for estimating K^* typically examine the within cluster dissimilarity W_k as a function of the number of clusters K. Seperate solutions are obtained for $K \in \{1, 2, \ldots, K_{max}\}$. The corresponding values generally decrease with increasing K.

Intuition:

- $K < K^*$ will partition actual groups into subsets. The solution criterion value will tend to decrease substantially with each successive increase in the number of specified clusters, $W_{K+1} \ll W_K$, as the natural groups are successively assigned to seperate clusters.
- $K > K^*$ will fuse natural groups into one. This will tend to provide a smaller decrease in the criterion as K is further increased.

$\Rightarrow$ Splitting a natural group, within which the observations are quite close to each other, reduces the criterion less than partitioning the union of two well-separated groups into their proper constituents.

To the extent this scenario is realized, there will be a sharp decrease in successive differences in critereon value, $W_K - W_{K+1}$, at $K = K^*$. An estimate for K^* is then obtained by identifying a "kink" in the plot of W_K as a function of K. As with outher aspects of clustering procedures, this approach is somewhat heuristic.

Recently proposed *Gap statistic* (Tibshirani et al., 2001b):

- Compare the curve $log W_K$ to the curve obtained from data uniformily distributed over a rectangle containing the data.
- Optimal K is where the gap between the two curves is largest

4.1.11 Hirarchical Clustering

Hirarchical clustering methods do not require specifications like for K-means. Instead, the require the user to specify a measure of dissimilarity between (disjoint) *groups* of observations, based on the pairwise dissimilarities among the observartions in the two groups.

At the lowest level, each cluster contains a single observation. At the highest level there is only one cluster containing all of the data.

Two basic paradigms: *aglomerative* (bottom-up, every step up there is one less cluster) and *devisive* (top-down, every step down there is one new cluster). There are $N - 1$ levels in the hirarchy.

Agglomerative Clustering

Let G, H represent two groups. The dissimilarity $d(G, H)$ is computed from the set of pairwise observation dissimilarities $d_{ii'}$.*Single linkage* (SL) agglomerative clustering takes the intergroup dissimilarity to be that of the closest (least dissimilar) pair. This is also often called the *nearest-neighbor* technique. *Complete linkage* (CL) agglomerative clustering (*furthest-neighbor* technique) takes the intergroup dissimilarity to be that of the furthest pair. *Group average* (GA) clutering uses the average dissimilarity between the groups (trade-off between compactness/diameter of clusters).

5 Dirichlet Process

We have several options for clustering the data. $k-$means variants are straight forward, easy to understand, but if the proper number of clusters is part of the unknowns, it may be a suboptimal choice. One alternative is mean shift clustering: here, the number of clusters is determined implicitly by the kernel type and size, and the type of bump post processing (non-maximum supression).

Today, we will look into a **Dirichlet Process** to model infinite Gaussian mixtures.

Short reminder: Gaussian mixture models.

1. What is a GMM? $\Rightarrow \sum_{i=1}^{k} \beta_i \mathcal{N}(\mu_i, \Sigma_i)$
2. What parameters? $\Rightarrow \beta_i, \mu_i, \Sigma_i$
3. How do I determine these parameters? $\Rightarrow EM - Algorithm$

   Start with random initialization, expectation (how much does each component contribute to each point), maximization (update component parameters), repeat until convergence

   From a more abstract perspective, EM performs two tasks simultaneously: segmentation/clustering assignment, and model parameters estimation/fitting

Idea for infinite mixture models: Instead of fitting a specific distribution to the data (the GMM), we define a meta-distribution from which the actual distribution is drawn. The infinite mixture model can be used for clustering.

Specifically, we can draw a GMM from a Dirichlet Process. However, this is a top-down persepectiv, i.e., if we randomly draw a GMM, we will certainly not draw one the nicely fits to our data. From a bottom-up perspective, we need a fitting algorithm that works with the available data points and finds a GMM in such a way that it could also have been drawn from a distribution of distribtions (the Dirichlet Proess).

To illustrate the model behind the fitting algorithm, we usually talk about the **Chinese Restaurante Process**.

- There are tables in a restaurant (our mixture components)
- Whenever a new customer comes in (a sample), it chooses the table it most relates to
- When the customer is unsatisfied with current offering, he can open up a new table

Extension to a "normal" GMM:

1. If a customer prefers to sit on a new table, this is possible: we can add arbitrarily many tables (this will vary the k in our mixture model)
2. "Rich get richer": The more people sit on a table, the more attractive it is to new customers ("this must be really tasty")

The Chinese Restaurant Process (CRP) provides a constructive way of sampling from a Dirichlet Process.

Here is a straight forward clustering algorithm based on the CRP:

1. Init: assign each sample to a cluster (e.g. randomly, or do $k-$means)
2. Randomly select a sample x_i from the data
3. Compute a affinity of x_i to each table: $t_i = \frac{N_i}{N+\alpha}\mathcal{N}(x_i, \mu_i, \Sigma_i)$ (Gaussian distance of x_i to (μ_i, Σ_i)), where N_i is the number of customers on the table (number of samples assigned to $\mathcal{N}(\mu_i, \Sigma_i)$ and N is the total number of samples, α is an "expansion parameter"

4. Compute affinity to a new table $t_0 = \frac{\alpha}{N+\alpha} \cdot \mathcal{N}(x_i, \mu_0, \Sigma_0)$
5. Collect all affinities $t_0, \dots, t_T$ in a list, normalize sum to one
6. Sample from that list a table assignment (all affinities are interpreted as a PDF from which we sample)
7. Recompute μ_i, Σ_i for the assigned table
8. Goto 2, stop after certain number of iterations

This type of algorithm is called **Gibbs sampling** (take one out, modify, put it back). This is a probabilistic algorithm (we sample from the list of affinities).

6 Dirichlet Process (continued)

Criteria for the affinity of a sample x to a cluster ϑ_i:

- Proximity: $\mathcal{N}(x, \mu_i, \Sigma_i)$
- Number of samples already assigned to that cluster: $\frac{N_i}{N+\alpha}$, where α is the expansion parameter (corresponds to β in Gaussian Mixture, $G = \sum_{i=0}^{k} \beta_i \mathcal{N}(\mu_i, \Sigma_i)$

Back to our top-down view: using the *CRP*, we are effectively drawing a fully parameterized GMM using a Dirichlet Process. Implicitly, the mixture weights β_i match a Beta(α) distribution and the Normal-distributions are drawn from hyper-distribution i.e. $\vartheta_i \sim H(\lambda)$.

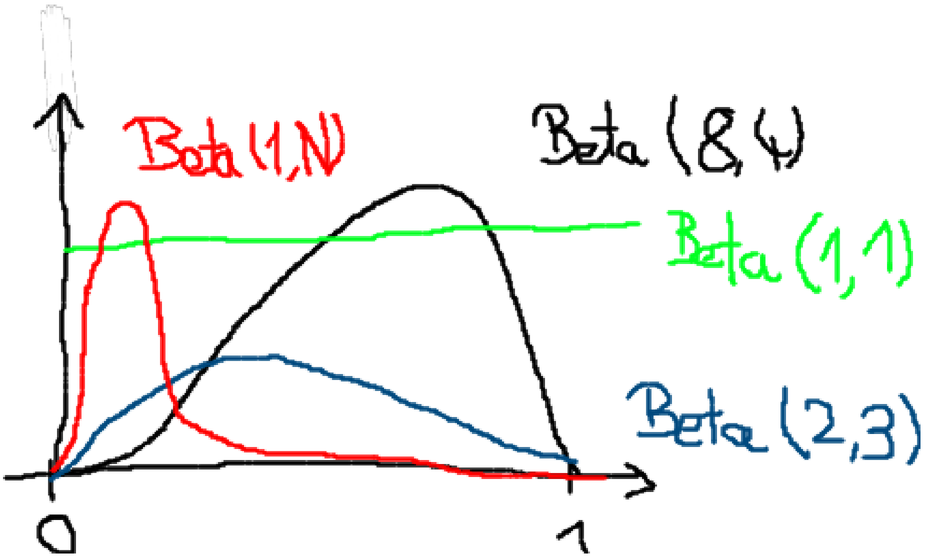
Illustration of a Beta-distribution: Stick breaking process:

1. Stick of length 1
2. We draw a random number between 0 and 1
3. We break this length off from the stick
4. Repeat 2,3 infinitly, while scaling the random number down to new sick length
5. The length of the stick parts are our cluster weights

We draw from a $Beta(1, \alpha) = \frac{\Gamma(a) \cdot \Gamma(\alpha)}{\Gamma(1+\alpha)}$ distribution, where $\Gamma(x) = \int_0^\infty u^{x-1} e^{-u} du$ is a Gamma-distribution.

We are expressing the probability distribution for flipping a coin, but the coin doesn't need to be fair (if the two parameters are far apart, it's a unfair coin. If they are close, it's more fair).

Qualitatively, this is what Beta-distribution looks like:



The expansion parameter α is a prior, that influences the number of clusters that will be created.

Observation 1: The larger the expansion parameter α is, the more likeli it is to draw a number that is close to zero from $Beta(1, \alpha)$.

Observation 2: The number that has been drawn from $Beta(1, \alpha)$ is used in the stick breaking process to determine the weight of the i-th mixture component β_i.

Stick breaking process:

- $b_i \sim Beta(1, \alpha)$

- $\beta_i = b_i \cdot \prod_{l=1}^{i-1}(1 - b_l) = b_i \cdot (1 - \sum_{l=1}^{k-1} \beta_l$ (remaining stick length)

7 Manifold Learning

7.1 Curse of Dimensionality

"Human intuition breaks down in high dimensional spaces." (Bellman, 1961)

Let us illustrate this: Consider d-dimensional feature vectors $x_1, x_2, \dots, x_n \in \mathcal{R}^d$ where $0 \leq x_{i,k} \leq 1$ uniformly distributed in a d-dimensioanl hypercube of volume 1. Let's say we would like to cover enough volume of that cube to collect 1% of the data. Let's say we also use a cube for this task. What is the required edge length of the cube to obtain that 1% of space?

- For example, for a 10-dimensional hypercube:

$$V = s^d \Rightarrow s = V^{\frac{1}{d}} = 0.01^{\frac{1}{10}} = 0.63$$

Another perspective: We have to draw a 10-dimensional vector, where no value is allowed to exceed 0.63 (has probabiliy of 1%).

Another way of thinking about this is thtat in a high dimensional space, virtually every feature point is located at the boundary of the space, because in at least one dimension we draw a very low or very high value. This leads to the effect that common distance measures loose their effectivity.

This can be seen by looking at the median distance of the nearest neighbor to the origin, given N samples in d dimensions:

$$medianDist(d, N) = (1 - \frac{1}{2}^{\frac{1}{N}})^{\frac{1}{d}}$$

$$\Rightarrow N = 5000, d = 10 : medianDist(10, 5000) \approx 0.52$$

This shows that distance measures start losing their effectiveness to measure dissimilarity in high dimensional spaces. Because of this, we need methods for reducing the dimensionality of our data, witout breaking its structure to hard.

"Notorious" example for high dimensional data: *hyperspectral remote sensing image classification*

Satellite image: perfom e.g. aggricultural monitoring: classify type of vegetation via hyperspectral vector (dozens of color channels).

In such classification pipelines, dimensionality reduction is often times one integral component.

7.2 (k)PCA

Our known approach to dimensionality reduction: **PCA** (principle component analysis): We search for an orthogonal basis that is aligned with the "maximum spread" (w.r.t to the covariance of the data). It is a global and unsupervised methode, operating on all datapoints (no labels, finds on globally optimal solution).

PCA is a linear method, which is problematic in certain cases ⇒ Kernel PCA: perform a non-linear mapping of the data, then perfom stadard (linear) PCA on the result. For radial data (inner and outer circle of two classes) we can for example perform a radial transform (space consisting of radius and angle). With the kernel trick, the non-linear mapping and PCA can be merged into one step.

Objective fuction:

$$J = \sum_{i,j=1}^{N} (\Phi(x_i) - \Phi(x_j))^T (\Phi(x_i) - \Phi(x_j)) - \lambda(\Phi^T \Phi - 1)$$

where nonlinear mapping is part of the Φ.

The **kPCA** idea had some successfull offsprings:

1. perform some preprocessing/mapping that operates non-linearly
2. the dimensionality reduction itself is an operation that operates linearly (linear algebra)

PCA operates on explicitly given feature vectors x_i. What if we only have access to differences between feature vectors, such as the Euclidean distance $\|x_i - x_j\|_2^2$?

⇒ This is a closely related problem, PCA on distances is called **MDS**.

7.3 Multidimensional Scaling (MDS)

Task: Reconstruct a set of points (potentially in a lower dimension) from their differences. MDS computes the "best" (in a least square sense) coordinates up to rotation, translation and mirroring (axis reversal).

Mathmatical problem formulation:

- Let $S = \{x_1, \dots, x_N\}, x_i \in \mathbb{R}^d$.
- Let X denote a matrix consisting of all samples, $X = [x_1, \dots, x_N] \in \mathbb{R}^{d \times N}$.
- Let $D^2 = [d_{ij}^2]_{i,j \in \{1,\dots,N\}}$, where $d_{ij}^2 = (x_i - x_j)^T (x_i - x_j)$.
- Goal: Given D^2, compute X.
- We assume that $x_1, \dots, x_N$ have zero mean i.e. $\sum_{i=1}^N x_i = 0$

Let us consider the distance matrix in terms of x:

$$d_{ij}^2 = (x_i - x_j)^T (x_i - x_j) = x_i^T x_i + x_j^T x_j - 2x_i x_j$$

In matrix notation,

$$D^2 = diag(X^T X) \cdot \vec{1}^T + \vec{1} \cdot diag(X^T X)^T - 2X^T X$$

where $diag(A)$ denotes a vector diagonal entries of A, i.e. $diag(A) = (a_{11}, a_{22}, \dots, a_{NN})^2$, and $\vec{1}$ denotes a vector of ones, i.e. $\vec{1} = (1, 1, \dots, 1)^T \in \mathbb{R}^d$.

Multiply D^2 from left and right with a centering matrix $C = (I - \frac{1}{N}\vec{1}\,\vec{1}^T)$, and weight the result by $-\frac{1}{2}$:

$$\begin{aligned} -\frac{1}{2} C D^2 C &= -\frac{1}{2}(I - \frac{1}{N}\vec{1}\,\vec{1}^T)(diag(X^T X) \cdot \vec{1}^T + \vec{1} \cdot diag(X^T X)^T - 2X^T X)(I - \frac{1}{N}\vec{1}\,\vec{1}^T) \\ &= \dots \text{(see lecture, too lazy)} \\ &= X^T X \end{aligned}$$

This is a matrix factorization problem. If you you compute $SVD(X^T X) \Rightarrow U\Sigma V \Rightarrow X = \Sigma^{\frac{1}{2}} V$.

MDS is essentially the same thing as PCA, but it operates on distances. We can reduce the dimensionality of our data by:

- Determining the m largest eigennvalues and their corresponding eigenvectors
- Dropping the remaining eigenvalue and eigennvectors inside the multiplication

7.4 ISOMAP algorithm

A non-linearity "patch" for MDS.

Idea: nearby points have their "usual" Euclidean distance. If a pair of points is not within a local neighborhood, the distance between these points is a graph distance (shortest path). We have an all pairs shortest path problem, which can be solved with certain algorithms (Dijstra, A^*, BFS, DFS, ...). Then, run MDS on the resulting distance matrix.

This procedure is useful if you want to reduce the dimensionality of "swirly" datasets for example.

ISOMAP operates on geodesic distances (like a distance on earths surface). This means that

7.5 Locally Linear Embedding

Idea: Treat the local neighborhood of a point x as linear. This makes the mapping an overall non-linear mapping consisting of small linear patches.

We can linearily interpolate a point x_i from its neighbors,

$$x_i = \sum_{x_j \in N(x_i), j \neq i} w_{ij} x_j, \text{ subject to } \sum_j w_{ij} = 1$$

and search points x' in a lower dimensional space, such that

$$x'_i = \sum_{j \in N(x'_i)} w_{ij} x'_j.$$

Algorithm:

1. Define the neighborhood (k-nearest neighbors, distance thresholding)
2. Solve for w_{ij} in the high dimensional space

$$min \sum_i \|x_i - \sum_{j \in N(x_i)} w_{ij} x_j\|_2^2 \text{ subject to } \sum_{j \in N(x_i)} w_{ij} = 1$$

3. Solve for $x'_i \in \mathbb{R}^{d'} (d' \ll d)$:

$$\sum_i \|x'_i - \sum_{j \in N(x_i)} w_{ij} x'_j\|_2^2, \text{ subject to } \frac{1}{N} \cdot \sum_i x'_i \cdot x'^T_i = I, \sum_i x'_i = \vec{0}$$

The first constraint says that the covariance is the identity.

Nearer point in the original space will result in higher weights.

Caveats:

1. The idea of LLE is really nice, but the original formulation is sometimes a bit unstable if applied on real data. Therefore there exists a number of variants to LLE in the literature that aims to make it more robust.
2. We will now deviate from the original formulation, and modify the constraint in step 2 to $\sum_{jinN(x_i)} w_{ij}^2 = 1$, to obtain a closed solution.

Modified step 2 (weight computation): Note tha the objective function is invariant to translation:

$$\sum_{i=1}^{N}\|x_i - \sum_j w_{ij}x_j\|_2^2 = \sum_{i=1}^{N}\|(x_i - \vec{t}) - \sum_j w_{ij}(x_j - \vec{t})\|_2^2$$

$$\Rightarrow \text{set } x_i = \vec{t} \Rightarrow \sum_{i=1}^{N}\|\sum_j w_{ij}(x_j - x_i)\|_2^2$$

$$= \|M_i\vec{w_i}\|_2^2, \text{ where } \vec{w_i} = \begin{pmatrix} w_{i1} \\ w_{i2} \\ \dots \\ w_{iN} \end{pmatrix}, M_i = \begin{pmatrix} x_1 - x_i & x_2 - x_i & \dots & x_N - x_i \end{pmatrix}$$

Note: differences $x_j - x_i = 0$ for x_j outside the neighborhood of x_i

$$\Rightarrow minimize (M_i w_i)^T(M_i w_i) + \lambda(1 - w_i^T w_i), w.r.t\ w_i$$

where the latter is the constraint that the squared weights should sum up to one. We see why we have to take the squaered weights critereon: if we would compute the derivative of $\lambda(1-w_i)$, the constraint would vanish to 0. We would have to use linear programming or something.

Compute

$$\frac{\partial}{\partial w_i}(w_i^T(M_i^T M_i)w_i + \lambda(1 - w_i^T w_i) = 2 \cdot M_i^T M_i w_i - 2\lambda \cdot w_i = 0$$

$$\Rightarrow M_i^T M_i w_i = \lambda w_i$$

This is an eigenvector eigenvalue problem and can solve this easily.

7.6 Laplacian Eigenmaps

Graph Laplacian: Matrix representation of a graph. There are many other ways to do this.

$$L = (D - W),$$

where D is a diagonal matrix with entries

$$d_{ij} = \begin{cases} \sum_{k=1}^{N} w_{ik} & if\ i = j \\ 0 & if\ i \neq j \end{cases}$$

and W is the adjacency matrix

$$w_{ij} = \begin{cases} \text{weight between nodes i and j} & if\ i \neq j \\ 0 & if\ i = j \end{cases}$$

We exclude self-loops here. D is there for normalization. This results in a matrix, which row sum equals 0 ($\sum_{k=1}^{N} l_{ik} = 0$).

The graph Laplacian is a popular tool to operate on graphs with methods from linear algebra (→ this is a topic in "spectral graph theory").

Idea:

1. Build an adjacency graph on the set of feature points $S = \{x_1, \ldots, x_n\}$
2. Choose weights w_{ij} for the edges of the graph
3. Perform eigendecomposition of the graph Laplacian
4. Obtain low-dimensional embedding

Choose the weights as affinities, meaning that closer point pairs have higher weights → points that are closely together in high-dimensional space shall remain close in the lower-dimensional embedding.

One common choice for the weights is the so-called heat kernel $w_{ij} = e^{-\|x_i - x_j\|_2^2}$, or just binary affinity $w_{ij} = \begin{cases} 1 & if\ \|x - x_j\|_2^2 \leq t \\ 0 & otherwise \end{cases}$.

Ojective function:

$$minimize \sum_{i=1}^{N} \sum_{j=1}^{N} \|x'_i - x'_j\|^2 w_{ij}$$

where $x' \in \mathbb{R}^{d'}, d' \ll d$.

Contraint for optimization: $x'^T D x' = 1$ (squared distances should add up to 1).

Relationship of the objective function to the graph Laplacian:

$$\begin{aligned}\sum_{i,j} \|x'_i - x'_j\|_2^2 w_{ij} = \sum_{i,j} (x_i'^T x'_i + x_j'^T x'_j - 2x_i'^T x'_j) w_{ij} = (2 \cdot \sum_{i,j=1}^{N} (x_i'^T x'_i) \cdot w_{ij}) - (2 \cdot \sum_{i,j} x_i'^T x'_j w_{ij}) \\ = (2 \cdot \sum_{i=1}^{N} x_i'^T x'_i \cdot (\sum_{j=1}^{N} w_{ij})) - (2 \cdot \sum_{i,j} x_i'^T x'_j w_{ij}) \\ = 2x'^T D x' - x'^T W x' = 2 \cdot x'^T (D - W) x'\end{aligned}$$

$$\text{minimize } x'^T L x' \text{ subject to } x'^T D x' = 1 :$$

$$\begin{aligned} & \frac{\partial}{\partial x'} (x'^T L x + \lambda(1 - x'^T D x')) \\ & = 22Lx' - \lambda D x' = 0 \\ & = 22Lx' - \lambda D x' = \vec{0} \\ Lx' & = \lambda D x' \\ \Rightarrow D^{-1} L x' & = \lambda x' \end{aligned}$$

On 4.) The lower-dimensional embedding is obtained by selecting a ordered subset of eigenvectors from that decomposition.

- Sort eigenvectors by the magnitude of their associated eigenvalues
- Discard the eigenvalue/eigenvector pair for the lowest eigenvalues (the 0 eigenvalue) (number of values depends on connected components in graph)
- The d' next smallest eigenvalue/eigenvector pairs $(e_1, e_2, \ldots)$ form the lower -dimensionanl embedding: ith row of $(e_1, e_2, ...e_d)$ represents the lower dimensioanl embedding of x_i^T.

Addendum to our Section on clustering: "Spectral clustering" is essentially executing Laplacian Eigenmaps + k-Means on the lower dimensional space.

How to choose a appropriate d'? The distribution of the eigenvalues gives an indication about the "intrinsic" dimensionality of the data. If there is a certain gap between eigenvalue, this is a good indication. We don't have this available when using LLE.

8 Random Forests

What is a random forest? An ensemble of decision trees.

What is a decision tree? A binary tree with a "decision" at every internal and the root node. It is a learning based approach: A good decision functions at the internal nodes are the result of training.

What is a decision? For example the answer to the question: "Is this sample on the right side of a hyperplane?"

Why an ensemble of trees? Experience showed that it is complicated to train a single, highly accurate decision tree. The idea of random forests is therefore to train a large number of individually less accurate trees in a ranomized fashion, and to report then an averaged result of this forest. If we have a trained decision tree, we can test it by evaluating at the root node a function

$$h(\vec{x}, \vec{\vartheta}_j) : \mathbb{R}^d \times \underset{\text{tree parameters}}{\mathcal{T}} \rightarrow \{0, 1\}.$$

Depending on the result we evaluate the test function at either the left successor or the right successor of the root node, and continue down the tree recursively until a terminal/leaf node is reached. The leaf node performs an application-specific action. For example, if the task is to perfom classification, it assigns a label to the sample.

Training: compute the tree parameters ϑ, consisting of

- the tree height/depth (*Note:* deeper trees tend to overfitt, must be complemented with increased number of trees, trade-off)
- the splitting function at each internal node
- if necessary, the "action" in the leaf node

It is important to note that this "binary tree paradigm" essentially performs a partitioning of the feature space. More specifically, each internal node subdivides the incoming samples into two parts.

8.1 Specific task: classification

Training of a single tree in a forest of size T

- decide for a set of splitting function prototypes, e.g. hyperplanes or conics, ... (simpler functions are typically prefered. Simplest function: "axis aligned split")
- (decide for randomization)
- to find the parameters for the split function, select a suitable objective function.

"Solid choice": Information Gain

$$I = H(S_j) - \sum_{i \in \{L,R\}} \frac{|S_j^i|}{|S_j|} H(S_j^i)$$

where S_j is the data that flows into the node, $S_j^L S_j^R$ is the data that flows to the left/right successor and $H(\dots)$ denotes the entropy. (*Note:* $I = H_{\text{before}} - H_{\text{after, weighted}}$)

$$H(S_j) = -\sum_{c \in C} p(c) \cdot log(p(c)))$$

where c denotes the class label, and $p(c)$ the empirical distribution computed from S_j.

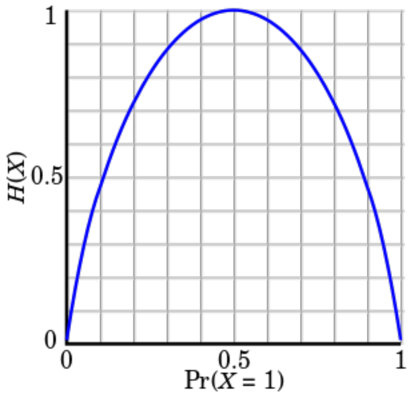


Figure 1: Binary entropy function

The candidate functions, out of which the best on is chosen with the information gain, are randomly drawn. Typically, one decides:

- how many candidate functions are drawn
- if also linear projection of the data shall be drawn (e.g. consider only dimensions $\{d_{i_1}, d_{i_2}, \ldots, d_{i_n}\}$
- how the splitting parameters are sampled

⇒ Note that a sparser sampling leads to more "noise"/less optimal results. (*Note:* might be desired, e.g. prevents overfitting)

Choice of tree depth:

- set maximum depth ("mandatory")
- optional: set minimum number of samples for split
- depending on the application, stop if for example 99% of features in a node belong to one class

Final classifier output:

- at a leaf node, report the relative frequencies of the class labels in that node (e.g., 15%: class 1, 85% class 2)
- combine al trees by averaging the individual tree outputs. If a single discrete label is required, decide for the class with maximum probability.

9 Random Forests (continued)

Example: Classification

1. Randomly select a number of splitting functions
2. Evaluate the information gain for each splitting function
3. Set the function with maximum information gain as the current nodes' decision function
4. Recursively repeat for the child nodes, until max tree depth (or some other criterium) is reached

9.1 Regression Forests

Goal: predict a continous label $p(y|\vec{x})$. (Remark: choice of the model complexity is related to the bias/variance trade off)

"Leaf prediction model": a base function that is fitted to the samples. The leaf prediction model could be

- constant
- linear
- polynomial
- ...

To faithfully represent all of the data with a single function, it would certainly make sense to use a polynomial model, or something even more complex. However, the random idea implies to subdivide/partition the space, and to fit simpler models to the individual partitions. As a specific example, let's split up our input data points:

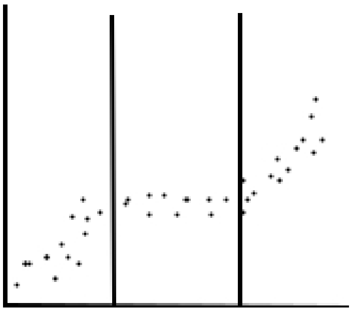


Figure 2: Regression split

The decision criterion for the splitting function works analogously to the classification case. The only difference ist that we need to define the entropy $H(S_j)$ on continous values:

$$H(S_j) = -\frac{1}{|S_j|} \cdot \sum_{\vec{x} \in S_j} \int_y p(y|x) \cdot log(y|x) dy$$

where $p(y|x)$ can, e.g. be chosen as a Gaussian distribution $p(y|x) = \mathcal{N}(y; \overline{y}(x), \sigma_y^2(x))$, where $\overline{y}(x)$ is a linear function and $\sigma_y(x)$ is the conditional variance computed from a linear fit.

Probabilistic linear fit:

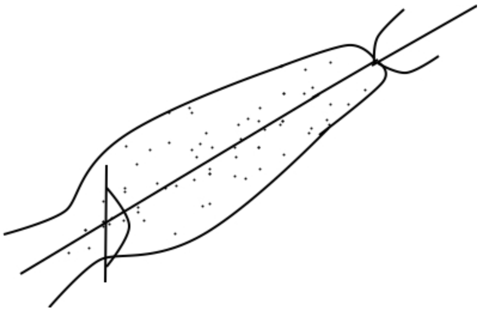


Figure 3: Probabilistic linear fit

Combining the expression for $p(y|x)$ into $H(S_j)$ yields

$$H(S_j) = \frac{1}{|S_j|} \cdot \sum_{\vec{x} \in S_j} \frac{1}{2} \cdot log((2\pi e)^2 \sigma_y^2(\vec{x}))$$

$$\Rightarrow I(S_j, \vartheta) = \sum_{\vec{x} \in S_j} log(\sigma_y(\vec{x})) - \sum_{i \in \{L,R\}} (\sum_{x \in S_j^i} log(\sigma_y(\vec{x})))$$

9.2 Density Forests

Very same idea, adapted to unlabelled data ⇒ learning-baed density estimator.

Each leaf node is modeled as a multivariate Gaussian distribution. The information gain metric can again be reused, i.e. $I(S_j, \vartheta) = H(S_j) - \sum_{i \in \{L,R\}} \frac{|S_j^i|}{|S_j|} \cdot H(S_j^i)$ but let us choose $H(S_j)$ as

$$H(S_j) = \frac{1}{2} \cdot log((2\pi e)^d |\underset{\nearrow}{\Lambda} S_j|)$$

Determinant of the covariance matrix of the data,
"volume of a cluster"

Plugging $H(S_j)$ back into $I(S_j, \vartheta)$ yields

$$I(S_j, \vartheta) = log(|\Sigma(S_j)|) - \sum_{i \in \{L,R\}} \frac{|S_j^i|}{|S_j|} \cdot log(|\Lambda(S_j^i)|)$$

.

In each leaf, fit a multivariate Gaussian distribution to the data in that leaf using, e.g. MLE.

Note that the fitted densities have discontinuities at the splitting boundaries (this is the same type of discontinuity that we observed for regression forests or classification forests). However, remember that the result of a random forest is averaged over all individual trees. Because of randomization, each tree splits at a slightly different location, and thus the discontinuities are "averaged out" in the forest.

10 Random Forests (continued)

An ensemble of trees that partition the feature space.

Each tree is trained randomly. The amount of randomness is one design decision that we have to make.

We greedily optimize the information gain at each level. At each internal node, we seek a splitting function for the incoming features that achieves the highest information gain from a set of alternative splitting functions.

Density estima1tion: From a set of discrete samples X, predicte the pdf $p(x$.

Parzen window estimator: $p(x_0) = \frac{k}{N \cdot V}$

10.1 Manifold Learning with Random Forests: Manifold Forest

Idea: Learn a partitioning of the feature space by training density forest.[2] These partitions define a local neighborhood of the samples, which can be used to compute affinities, and then to apply Laplacian Eigenmaps on them.

Affinity intuition: Decreases with increasing distance.

Task Define affinities on a readily trained density forest.

Lets $w_{ij} = e^{-Q(x_i, x_j)}$ denote the affinity between samples x_i, x_j, where $Q(x_i, x_j)$ denotes a distance function.

The "speciaility" of a Manifold Forest (in contrast, e.g. to standard Laplacian Eigenmaps) is that these distances are defined w.r.t. the leafs of the trees (i.e. the partitions).

Example choices of Q: Let $d_{ij} = x_i - x_j$, then

$$\text{Mahalanobis: } Q(x_i, x_j) = \begin{cases} d_{ij}^T (\Lambda_{l(x_i)})^{-1} d_{ij} & \text{if in the same leaf } l(x_i) \\ \infty & otherwise \end{cases}$$

Gaussian:...

$$\text{Binary: } Q(x_i, x_j) = \begin{cases} 0 & \text{if in the same leaf} \\ \infty & otherwise \end{cases}$$

Within a single tree, only the samples with the same leaf have a non-zero affinity. Thus, a single tree produces a number of disconnected neighborhoods. However if the affinities are averaged over the whole forest, all points become connected.

⇒ We have a full adjacency matrix $A = \frac{1}{T} \sum_{t=1}^{T} w_i j^t$, where T denotes the total number of trees.

⇒ Compute LE on A.

Note that the graph Laplacian in Ciminisi/Shotton are differently normalized, but it is the same algorithm:

$$L = I - \Gamma^{-\frac{1}{2}} A \Gamma^{-\frac{1}{2}}$$

where $\Gamma = D = \sum A_{ij}$ denotes the sum of the weights for sample x_i.

[2] Note that we don't explicitly use the density

$\Rightarrow$: analogous to LE (Eigenvalue decomposition, arrange d' eigenvectors that are associated with the lowest eigenvalues in a matrix.

$$E = (e_1, e_2, \dots, e_{d'})$$

The projection of x_i onto a d' dimensional space just corresponds to the i-th row of E.

11 Hidden Markov Models and Markov Random Fields

Generative vs. Discriminative Model: let x denote the input, y the hidden variable/prediction/...

$$p(x,y) \Leftrightarrow p(y|x)$$

In a generative model, the available information is "more complete", i.e. we can for example also generate new samples x by sampling from $p(x,y)$ by marginalizing over y.

A discriminative model is slimmer (which is particularly useful for robust training with limited training data (the situation we're almost always in)); however, it only allows to discriminate between different y's, but we can not generate new values x.

11.1 Remarks about HMMs

- Probabilistic, graphical model
- the directed edge in the graph can be understood as a statistical dependency:

$$S_1 \rightarrow S_2 \Rightarrow p(S_2|S_1)$$

- Generative approach
- For many tasks, including speech processing, we oftentimes only allow for state transitions a_{ij} with $i \leq j$. (no backward links) Those restricted HMMs are called "Left-right HMMs", as opposed to fully connected ("ergodic") HMMs.

How can we generate observations with an HMM?

1. Sample from $\pi = (\dots)$ (this gives you starting state)
2. Let S_1 denote the starting state. Sample from b_1 (column or row from production probability matrix B) a symbol
3. Sample from $\vec{a}_{S_1}$ (column or row from state transition matrix A) the next state
4. Repeat 2.-3. until (randomly drawn/desired) word length reached

11.2 Markov Random Fields (MRF)

1985, Geman/Geman: introduced MRFs to image processing: Consider the pixel grid as a lattice (Gitter) of random variables.

More specifically, let us assume that the image F is given by the random matrix $[f_{ij}]$.

Assumption: limited statistical dependency:

$$p(f_{ij}|f_{i-1,j-1}, f_{i,j-1}, f_{i-1,j})$$

where $f_{i-1,j-1}, f_{i,j-1}, f_{i-1,j})$ form a dependency of the neighbors to $f_{i,j}$ (This will be our Markov property).

$$p([f_{ij}]) = \prod_{i,j} p(f_{i,j}|f_{i-1,j-1}, f_{i,j-1}, f_{i-1,j})$$

11.2.1 Definition of MRF

Lets us consider the features/observations $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N$

1. Positivity: $p(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N) > 0$
2. Markov property: $p(\vec{x}_k|\vec{x}_1, \ldots, \vec{x}_{k-1}, \vec{x}_{k+1} \ldots, \vec{x}_N) = p(\vec{x}_k|\mathcal{N}(\vec{x}_k))$ where $\mathcal{N}(\vec{x}_k)$ denoted the neighborhood of $\vec{x}_k$.

Definition of the neighborhood:

1. $\vec{x}_k \notin \mathcal{N}(\vec{x}_k)$
2. $\vec{x}_i \in \mathcal{N}(\vec{x}_k) \Rightarrow \vec{x}_k \in \mathcal{N}(\vec{x}_i)$
3. $\mathcal{N}(\vec{x}_k) = \{\vec{x}_i | 0 < dist(\vec{x}_i, \vec{x}_k) \leq t\}$

Example: Pixel grid: $\mathcal{N}(\vec{x}_{i,j}) = \{x_{k,l}|(i-k)^2 + (j-l)^2 \leq c^2, i \neq k \text{ or } j \neq l\}$

- NESW as neighbors, 4-Neighborhood ($c = 1$)
- All around as neighbors, 8-Neighborhood ($c = \sqrt{2}$)

12 Markov Random Fields (continued)

(Graph) clique: Complete subgraph

Gibbs Random Field (GRF): A GRF is given by the PDF $p(x) = \underset{\nearrow}{\frac{1}{Z}}e^{-H(x)}$, where $H(x)$ is an energy

$Z = \sum_{x'} H(x')$ is called a Partition Function

function, i.e. a sum of potential functions. For a given PDF $p(x)$, the choice of energy function is not unique. Consider for example:

$$H(x) = -logp(x) - logZ$$

$$p(x) = \frac{1}{Z}e^{-H(x)} = \frac{1}{Z}e^{logp(x)}e^{logZ} = p(x)$$

$$\Rightarrow \text{we can chose Z arbitratily}$$

The interesting theoretical property is that GRF and MRF are equivalent. This is called the *Hammersley-Clifford Theorem.*

12.1 Example: image denoising

Given: Observed noisy image $[g_{ij}]$

Hidden variables are the ideal (noiseless) images $[f_{ij}]$.

Assumption 1: the ideal image is spatially smooth

$$p([f_{ij}]) = \frac{1}{Z} \cdot e^{-H([f_{ij}])}$$

where $H([f_{ij}]) = \sum_{ij} \|\Delta f_ij\|_2^2$ (sum of squared gradients, computed over a neighborhood) (Note: Smoother image produces smaller H, which maximizes $p([f_{ij}])$.

Assumption 2: $[g_{ij}]$ is similar to f_{ij}, but corrupted by additive Gaussian noise.

$$p([g_{ij}]|[f_{ij}]) = \prod_{i,j} \frac{1}{\sqrt{2\pi}\sigma_{ij}} \cdot exp(-\frac{1}{2\sigma_{ij}^2} \cdot \underset{\text{energy function H}}{(f_{ij} - g_{ij})^2})$$

With these two functions defined, we can solve for a MAP estimte for f:

$$\begin{aligned} \underset{\text{estimated ideal image}}{[\hat{f_{ij}}]} &= \underset{[f_{ij}]}{\text{argmax}}\, p([f_{ij}]|[g_{ij}]) \\ &= \underset{[f_{ij}]}{\text{argmax}}\, p([g_{ij}]|[f_{ij}]) \cdot p([f_{ij}]) \\ &= \ldots \text{(take log)} \\ &= \underset{[f_{ij}]}{\text{argmin}}\{\sum_{ij} \|\Delta f_{ij}\|_2^2 + \sum_{ij} \lambda_{ij}(f_{ij} - g_{ij})^2\} \end{aligned}$$

$\|\Delta f_{ij}\|_2^2$: function of clique of size 2

Lecture slides: `https://www.slideshare.net/ykwang/markov-random-field-mrf`

13 Markov Random Fields

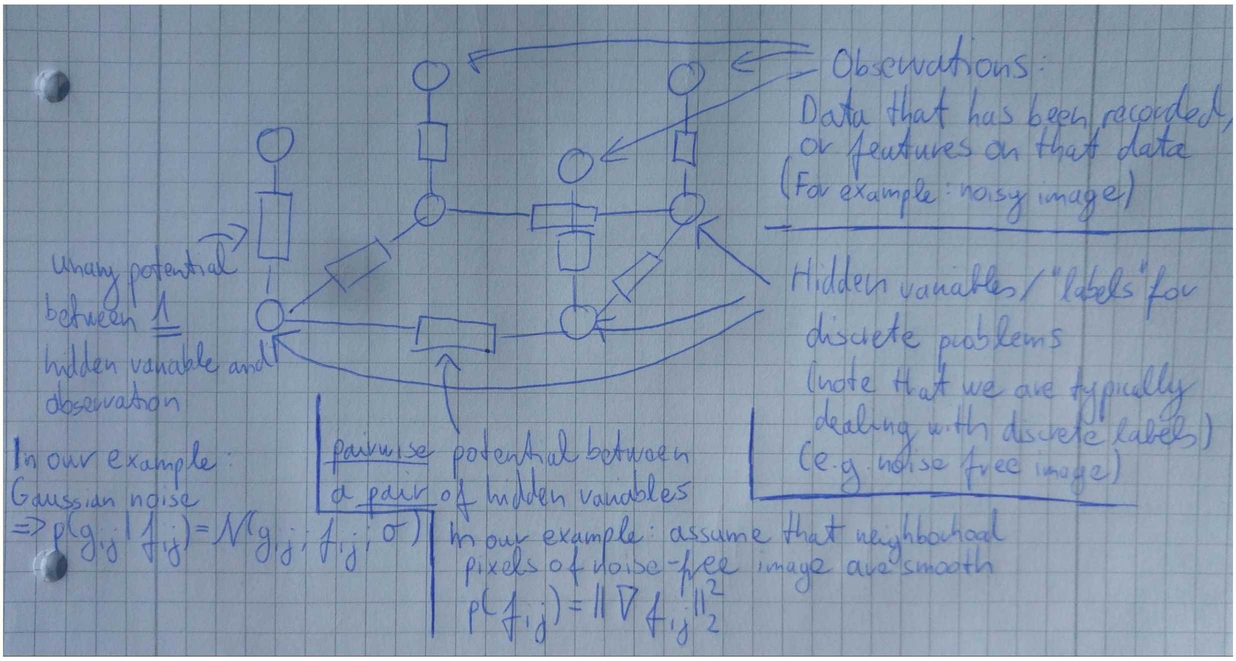

Observations:
Data that has been recorded, or features on that data
(For example: noisy image)
Hidden variables / "labels" for discrete problems
(note that we are typically dealing with discrete labels)
(e.g. noise free image)
Unary potential between 1 hidden variable and observation
In our example: Gaussian noise
=> p(g_ij | f_ij) = N(g_ij; f_ij; σ)
pairwise potential between a pair of hidden variables
In our example: assume that neighborhood pixels of noise-free image are smooth
p(f_ij) = ||∇f_ij||_2^2

Figure 4: MRF for image denoising

From a engineering perspective, we model the hidden variables by defining the unary and pairwise potentials. These are local relationships in the graph, i.e. they don't require complicated non-local interactions to be considered.

Remember from last week: a MRF and a GRF (Gibbs Random Field) are equivalent (*Hammersley-Clifford* theorem). This means that we can exchange the probabilities (defining the PDFs between the nodes in the MRF model might be impossible/hard) that describe the node relationships by potentials.

$$p(\vec{x}) = \prod_C p_C(\vec{x}_C) = \frac{1}{Z}\prod_C \psi_C(x_C)$$

Such potentials are much easier to design: We are typically interested in potential functions that assume a minimum value in a specific situation, e.g. $\|\Delta f_{ij}\|_2^2$ is minimal if $f_{i,j} = f_{i,j+1} = \ldots$, that depends on what we would like to achieve with the MRF.

$p(x_1, \ldots, x_N) = \prod \ldots$ independent segments of the random variables $x_1, \ldots, x_N$

General form of a Gibbs potential:

$$p(\vec{x}) = \frac{1}{Z}\prod_C \phi_C(\vec{x}_C) = \frac{1}{Z}\prod_C e^{-H(\vec{x}_C)} = \frac{1}{Z}e^{-\sum_C H(\vec{x}_C)}$$

where $H(\vec{x}_C) = \|\Delta f_{ij}\|_2^2$ or $H(\vec{x}_C) = \mathcal{N}(g_{ij}, f_{ij}, \sigma))$.

$H(\vec{x}_C)$ is an energy function that is defined over clique potentials. Solvers for Markov Random Fields seek to directly minimize $H(\vec{x}_C)$ instead of maximizing $\psi_C(x_C)$.

A modern way of solving an MRF is via graph cuts[3].

[3] Kolmogorov, Zabih: "What Energy Functions Can Be Minimized via Graph Cuts?"

13.1 Maximum Flow and Minimum Cut

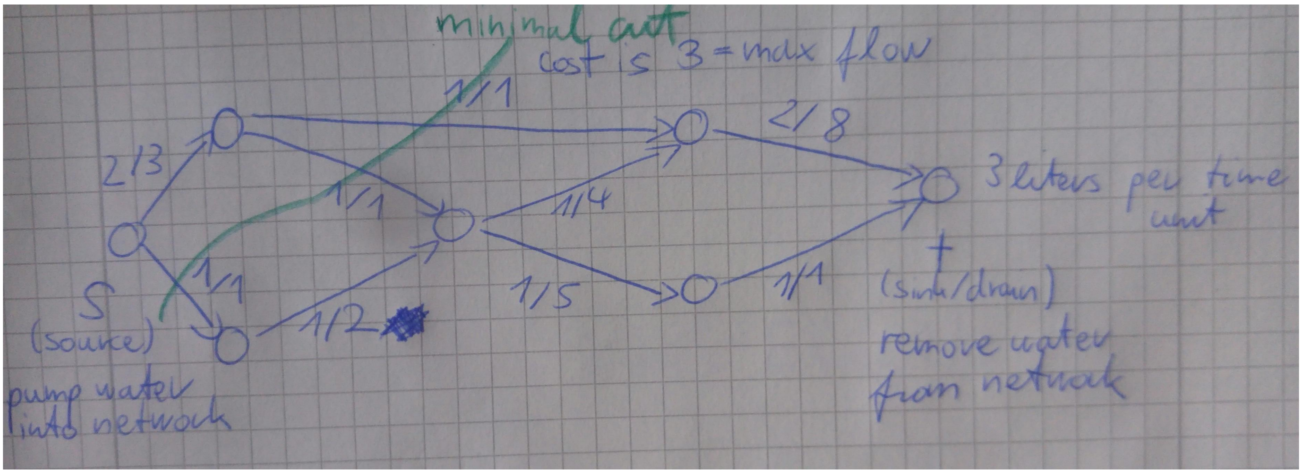


Figure 5: Max Flow Min Cut

There are a number of classic algorithm that solve the max flow problem in polynomial time.

A minimum cut divides a network into two parts s, t, such that the sum of the edge weights that are cut are minimal.

A solution to the maximum flow problem is a solution to the minimum cut problem: the minimum cut consists of edges that are fully used in the solution to the max flow problem.

Appendices

A Lawrence R. Rabiner, Fellow: A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition

A.1 Discrete Markov Processes

Consider a system being in one of a set of N states $S_1, \dots, S_N$. We denote the actual state at time t as q_t. State changes accure according to a set of probabilities associated with each state. In the case of first order Markov chains, this probabilistic description depends only on the predecessor state:

$$P(q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, \dots) = P(q_t = S_j | q_{t-1} = S_i)$$

It is assumed that the transition probabilities are independent of time, leading to a set of state transition probabilities $a_{ij} = P(q_t = S_j | q_{t-1} = S_i)$ with properties $a_{ij} \geq 0$ and $\sum_{j=1}^{N} a_{ij} = 1$. Additionally the process starts in a specific state i with probability $\pi_i = P(q_1 = S_i)$.

A.2 Extension to Hidden Markov Models

In a Markov Process each state corresponds to an observable (physical) event. This model is too restrictive to be appicable to many problems of interest. We extend the concept of Markov models to include the case where the observation a probabilistic function of the state - i.e. the resulting model (which is called a hidden Markov model) is a doubly embedded stochastic process with an underlying stochastic process that is not observable, but can be observed through another set of stochastic processes that produce the sequence of observations.

A.3 Elements of an HMM

1. N, the number of states in the model. We denote the states as $S = \{S_1, \dots, S_N\}$
2. M, the number of distinct observation symbols per state. We denote the individual symbols as $V = \{v_1, \dots, v_M\}$
3. The state transition probability distribution $A = \{a_{ij}\}$ where
$$a_{ij} = P(q_{t+1} = S_j | q_t = S_i)$$
4. The observation symbol probability distribution in state $j, B = \{b_j(k)\}$, where
$$b_j(k) = P(v_k \ at \ t | q_t = Sj)$$
5. The initial state distribution $\pi = \{\pi_i\}$, where
$$\pi_i = P(q_1 = S_i)$$

Given appropriate values of $N, M, A, B, \ and \ \pi$ the HMM can be used as a generator to generate an observation sequence

$$O = O_1, O_2, \dots, O_T$$

For convenience, we use the compact notation

$$\lambda = (A, B, \pi)$$

to indicate the complete parameter set of the model.

A.4 The Three Basic Problems of HMMs

1. Given the observation sequence $O = O_1, O_2, \ldots, O_T$, and a model $\lambda = (A, B, \pi)$, how do we efficiently compute $P(O|\lambda)$, the **probability of the observation sequence**, given the model?
2. Given the observation sequence $O = O_1, O_2, \ldots, O_T$, and the model λ, how do we choose a corresponding state sequence $Q = q_1, q_2, \ldots, q_t$ which is optimal in some meaningful sense (i.e., **best "explains" the observations**)?
3. How do we adjust model parameters $\lambda = (A, B, \pi)$ to **maximize** $P(O|\lambda)$?

Notes:

1. Evaluation problem, but also: We can view the problem as one of scoring, how well a given model matches a given observation sequence (useful for comparing models).
2. Attempt to uncover the hidden part, i.e. to find "correct" state sequence. Note: there never really is a "correct" state sequence to be found.
3. Important for training!

Examples: Isolated word recognizer

We design a sperate N-state HMM for each word in its vocabulary W. We represent each word as a time sequence of coded spectral vectors (M many). Thus for each word, we have a training sequence of a number of repetitions of sequences of spectral vectors.

1. Problem 3: estimate optimal model parameters for each word model
2. Problem 2: segement each of the word training sequences into states, try to understand the physical meaning of the model states. Make refinements (more states, different codebook size, etc.) so as to improve the capability of modeling the spoken work sequence.
3. Problem 1: Recognize an unknown word by comparing the scores of the individual models

A.5 Solution for Problem 1

We wish to calculate the probability of the observation sequence $O = O_1, O_2, \ldots, O_T$ given the model $\lambda, i.e., P(O|\lambda)$.

Straight forward way: Enumerate every possible state sequence of length T.

The probability of the observation sequence O for the state sequence $Q = q_1, \ldots, q_T$ is

$$\begin{aligned} P(O|Q,\lambda) &= \prod_{i=1}^{T} P(O_t|q_t,\lambda) \\ &= b_{q_1}(O_1) \cdot b_{q_2}(O_2) \ldots b_{q_T}(O_T) \end{aligned}$$

given the assumed statistical independence of observations.

The probability of such a state sequence Q can be written as

$$P(Q|\lambda) = \pi_{q_1} \cdot a_{q_1 q_2} \ldots a_{q_{T-1} q_T}$$

The joint probability of O, Q given λ is given by

$$P(O, Q|\lambda) = P(O|Q,\lambda) P(Q|\lambda)$$

The probability of O is obtained by summing this joint pdf over all possible state sequences q giving

$$\begin{aligned} P(O|\lambda) &= \sum_{all\ Q} P(O|Q,\lambda)P(Q|\lambda) \\ &= \sum_{q_1,q_2,\dots,q_T} \pi_{q_1} b_{q_1}(O_1)\dots a_{q_{T-1}q_T} b_{q_Y}(O_T) \end{aligned}$$

This calculation of $P(O|\lambda$ requires about $2 \cdot T \cdot N^T$ calcuations ⇒ BAD!

A.5.1 Forward-Backward Algorithm

We define the forward variable as $\alpha_t(i) = P(O_1, O_2, \dots, O_t, q_t = S_i|\lambda)$, i.e. the probability of the partial observation sequence and the state S_i at t.

1. Initialization: $\alpha_1(i) = \pi_i b_i(O_1)$
2. Induction: $\alpha_{t+1}(j) = (\sum_{i=1}^N \alpha_t(i) \cdot a_{ij}) b_j(O_{t+1})$
3. Termination: $P(O|\lambda) = \sum_{i=1}^N \alpha_T(i)$

The computation of $\alpha_t(i)$ requires N^2T calculations.

In a similar manner, we can consider backward variable $\beta_t(i) = P(O_{t+1}, O_{t+2}, \dots, O_T|q_t = S_i, \lambda)$ i.e. the probability of the partial observation sequence from $t+1$ to T, given state S_i at time t.

- Initialization: $\beta_T(i) = 1$ (choosen arbitrarily)
- Induction: $\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)$

Again, $\beta_t(i)$ requires N^2T calculations.

A.6 Solution for Problem 2

One possible optimality critereon: choose the states q_t which are *individually* most likely. We define

$$\begin{aligned} \gamma_t(i) &= P(q_t = S_i|O,\lambda) \\ &= \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \end{aligned}$$

i.e., the probability of being in state S_i at time t. The denominater makes $\gamma_t(i)$ a probability measure by normalizes $\sum_{i=1}^N \gamma_t(i) = 1$.

We can solve for the individually most likely state q_t via

$$q_t = \operatorname*{argmax}_{1 \le i \le N} \gamma_t(i)$$

Problem: Although this maximized the expected number of correct states (by choosing the most likely state for each t), it could be the case that such a state sequence isn't even possible (i.e. $a_{ij} = 0$).

Another optimaility critereon: find single best state sequence (path), i.e. maximize $P(Q|O,\lambda)$, which is equivalent to maximizing $P(Q,O|\lambda) \rightarrow$ *Viterbi-Algorithm*

We define the quantity

$$\delta_t(i) = \max_{q1,q2,\dots q_{t-1}} P(q_1, \dots, q_t = i, O_1, \dots, O_t|\lambda)$$

i.e., $\delta_t(i)$ is the best score (highest probability) along a single path, at time t, which accounts for the first t observations and ends in state S_i. By induction we have

$$\delta_{t+1}(j) = \max_i(\delta_t(i)\alpha_{ij}) \cdot b_j(O_{t+1})$$

To retrieve the state sequence, we need to keep track of the argument which maximizes this expression. We do this via the array $\psi_t(j)$

1. Initialization:

$$\begin{aligned}\delta_1(i) &= \pi_i b_i(O_1) \\ \psi_1(i) &= 0\end{aligned}$$

2. Recursion:

$$\begin{aligned}\delta_t(j) &= \max_i(\delta_{t-1}(i)\alpha_{ij}) \cdot b_j(O_{t+1}) \\ \psi_t(j) &= \operatorname*{argmax}_i(\delta_{t-1}(i)\alpha_{ij})\end{aligned}$$

3. Termination:

$$\begin{aligned}P^* &= \max_i \delta_T(i) \\ q_T^* &= \operatorname*{argmax}_i \delta_T(i)\end{aligned}$$

4. Path (state sequence) backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), t = T-1, \ldots, 1$$

A.7 Solution for Problem 3

There is no known way to analytically solve this problem: Baum-Welch Algorithm, iterative approach to choose $\lambda = (A, B, \pi)$ such that $P(O|\lambda$ is locally maximalize using an iterative procedure.

We define $\xi_{i,j}$ the probability of being in state S_i at time t, and state S_j at time $t+1$, given the model an the observations, i.e.

$$\begin{aligned}\xi_t(i,j) &= P(q_t = S_i, q_{t+1} = S_j|O, \lambda) \\ &= \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} \\ &= \frac{\alpha_t(i) \cdot a_{ij} \cdot b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}\end{aligned}$$

We have previously defined $\gamma_t(i)$ as the probability of being in state S_i at time t, given the observation sequence and the model; hence we can relate $\gamma_t(i)$ to $\xi_t(i,j)$ by summing over j, giving

$$\gamma_t(i) = \sum_{j=1}^N \xi_t(i,j)$$

If we sum $\gamma_t(i)$ over the time index t, we get a quantity which can be interpreted as the expected (over time) number of times that S_i is visited, or equivalently, the expected number of transitions made from state S_i (if we exluce T). Similarily, summation of $\xi_t(i,j)$ can be interpreted as the expected number of transitions from state S_i to S_j. That is:

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{ expected number of transitions from } S_i$$

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{ expected number of transitions from } S_i \text{ to } S_j$$

Given these formulas, we can give a method for reestimation of the parameters of an HMM. A set of reasonable reestimation fomulas for π, A, B are

$$\begin{aligned}
\bar{\pi_i} &= \text{ expected frequency in state } S_i \text{ at time } t = 1, \gamma_1(i) \\
\bar{a_{ij}} &= \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } s_i} \\
&= \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}
\end{aligned}$$

$$\begin{aligned}
\bar{b_j(k)} &= \frac{\text{expected number of time in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \\
&= \frac{\sum_{t=1}^{T} \gamma_t(j) 1(v_t = k)}{\sum_{t=1}^{T} \gamma_t(j)}
\end{aligned}$$

If we define the current model as $\lambda = (A, B, \pi)$ and use that to compute these new esimates, then it has been proven that the new $\bar{\lambda}$ is at least as likely as the old model in the sense that $P(O|\hat{\lambda}) > P(O|\lambda)$. If we use this procedure iteratively we obtain a maximum likelihood estimate of the HMM.

B Criminisi, Shotton, Konukoglu: "Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manyfold Learning and Semi-Supervised Learning", Foundations and Trends in Computer Graphics and Visio

Random Decision Forests

B.1 Background and notation

B.1.1 Decision tree basics

A tree is a collection of nodes (internal/split and terminal nodes) and edges organized in a hirarchical structure.

A *decision* tree is a tree used for making decisions. Each internal node corresponds to a *test*. The leaf contains the answer. A decision tree can be interpreted as a technique for splitting complex problems into a hirarchy of simpler ones.

B.1.2 Training and testing decision trees

Testing:

Bli Bla Blub

Training:

When discussing tree training it is convenient to think of subsets of training points associated with different tree branches, e.g. S_1 denotes the subset of training points reaching node 1 (numbering in breadth-first order starting from 0 for the root) and S_1^L, S_1^R denote the subsets going to the left and to the right children of node 1 and $S_j^L = S_{2j+1}$.

B.1.3 Entropy and information gain

Shannon entropy: $H(S) = -\sum_{c \in C} p(c) log(p(c))$, where c is a class.

Information gain: $H(S) = H(S) - \sum_{i \in \{1,2\}} \frac{|S^i|}{|S|} H(S^i)$

Differential entropy of a d-variate Gaussian density: $H(S) = \frac{1}{2} log((2\pi e)^d |\Lambda(S)|)$

In case of unlabled data:

- Fit Gaussian to data
- Split in a way where the entropy of the Gaussians, fitted to the splitted sets is the lowest (information gain is highest)

B.2 The decision forest model

A random decision forest is an ensemble of randomly trained decision trees.

Components:

- Family of split function ("weak learners")

- Type of leaf predictors
- Randomness model

B.2.1 The weak learner model

Each split node j is associated with a binary split function $h(v, \theta_j) \in \{0, 1\}$. The weak learner model is characterized by its parameters $\theta = (\phi, \psi, \tau)$ where

- ψ defines the goemetric primitive used to separate the data (e.g. an axis-aligned hyperplane, an oblique hyperplane, a general surface, etc.)
- τ captures thresholds for the inequalities used in the binary test
- ϕ is a filter function, which selects some features of choice out of the entire vector v

Example:

Linear data separation: $h(v, \theta_j) = [\tau_1 > \phi(v) \cdot \psi > \tau_2]$, where [.] is the indicator function.

B.2.2 The training objective function

Objective function:

$$\theta_j^* = \underset{\theta_j}{\operatorname{argmax}} I_j$$

with

$$I_j = I(S_j, S_j^L, S_j^R, \theta_j)$$

Can be found by extensive search.

B.2.3 The randomness model

Drawing the component trees of the forest leads to de-correlation between the individual tree predictions and, in turn, to improve generalization.

Randomness is injected into the trees during the training phase:

- random training data set sampling
- randomized node optimization

Randomized node optimization

If $\mathcal{T}$ is the entire set of all possible parameters $\mathcal{T}$ then when training the jth jode we only make available a small subset $\mathcal{T}_j \subset \mathcal{T}$ of such values. Thus under the randomness model training a tree is achieved by optimizing each split node j by

$$\theta_j^* = \underset{\theta_j \in \mathcal{T}}{\operatorname{argmax}} I_j$$

. The amount of randomness is controlled by the ratio $|\mathcal{T}_j|/|\mathcal{T}|$.

We define $p = |\mathcal{T}_j|$.

B.2.4 The leaf prediction model

In the case of classification, each leaf may store the empirical distribution over the classes associated to the subset of training data that has reached that leaf. The probabilistic leaf predictor model for the tth tree is then $p_t(c|v)$. In more conventional decision trees the leaf output may be $c^* = \operatorname{argmax}_c p_t(c|v)$.

B.2.5 The ensemble model

In a forest with T trees we have $t \in \{t, \dots, T\}$. All trees are trained independently. During testing, each test point v is simultaneously pushed through all trees. Combining tree predictions:

$$p(c|v) = \frac{1}{T} \sum_{t=1}^{T} p_t(c|v)$$

or (even though trees are not statistically independent)

$$p(c|v) = \frac{1}{Z} \prod_{t=1}^{T} p_t(c|v)$$

with the partition function Z ensuring probabilistic normalization.

B.2.6 Stopping criteria

It is common to stop creating new branches when a maximum number of level D has been reached. Alternatively one can impose a minimum informatioin gain or a node contains less than a defined number of training points. Avoiding growing full trees has shown to help generalization.

B.2.7 Summary of key model parameters

Parameters that influence forest behavior:

1. Forest size T
2. Maximum allowed tree depth D
3. Amount of randomness (controlled by p) and its type
4. The choice of weak learner model
5. The training objective function
6. The choice of features

Classification Forests Classification forests enjoy a number of useful properties:

- they naturally handle problems with more than two classes
- they provide a probabilistic output
- the generalize well
- they are efficient

B.3 Specializing the decision forest model for classification

Task: *Given a labelled traning set learn a general mapping which associates previously unseen data with their corret classes*

More formally, during testing we are given an input test data v and we wish to infer a class label c such that $c \in C$, with $C = \{c_k\}$. More generally we wish to compute the whole distribution $p(c|v)$. Training happens by optimizing an energy over a training set S_0 and associated ground truth labels.

The training objective function

Forest training happens by optimizing the parameters of the weak learner at each split node j via:

$$\theta_j^* = \underset{\theta_j \in \mathcal{T}_j}{\operatorname{argmax}} I_j$$

For classification the objective function I_j takes the form of a classical information gain defined for discrete distribtutions:

$$I_j = H(S_j) - \sum_{i \in \{L,R\}} \frac{|S_j^i|}{|S_j|} H(S_j^i)$$

The entropy for a generic set S is defined as

$$H(s) = -sum_{c \in C} p(c) log(p(c))$$

where $p(c)$ is calculated as normalized empirical histogram of labels corresponding to the training points in S. This leads to increasing confidence of prediction when going from root toward the leaves.

Randomness

We define $p = |\mathcal{T}_j|$ indication the amount of randomness. We randomly sample p parameter values out of possibly billions or infinite.

The leaf and ensemble prediction models

Classification forests produce probabilistic output:

$$p(c|v) = \frac{1}{N} \sum_1^T p_t(c|v)$$

B.4 Effects of model parameters

Here, effects of the forest model parameters on classiciation accuracy and generalization are studied.

B.4.1 The effect of forest size on generalization

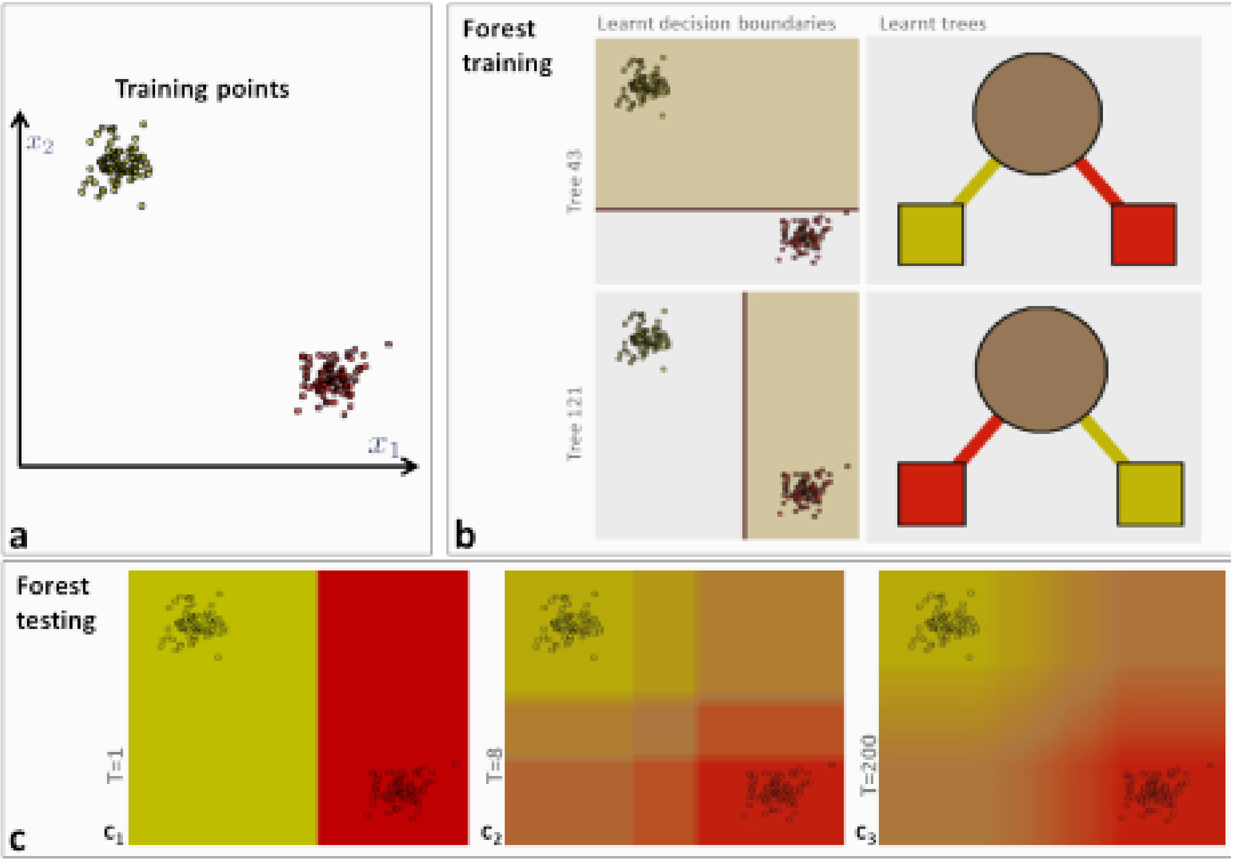


Fig. 3.3: **A first classification forest and the effect of forest size** T**.** **(a)** Training points belonging to two classes. **(b)** Different training trees produce different partitions and thus different leaf predictors. The colour of tree nodes and edges indicates the class probability of training points going through them. **(c)** In testing, increasing the forest size T produces smoother class posteriors. All experiments were run with $D = 2$ and axis-aligned weak learners. See text for details.

B.4.2 Multiple classes and training noise

Major advantage of decision forests: the same classification model can handle both binary and multi-class problems.

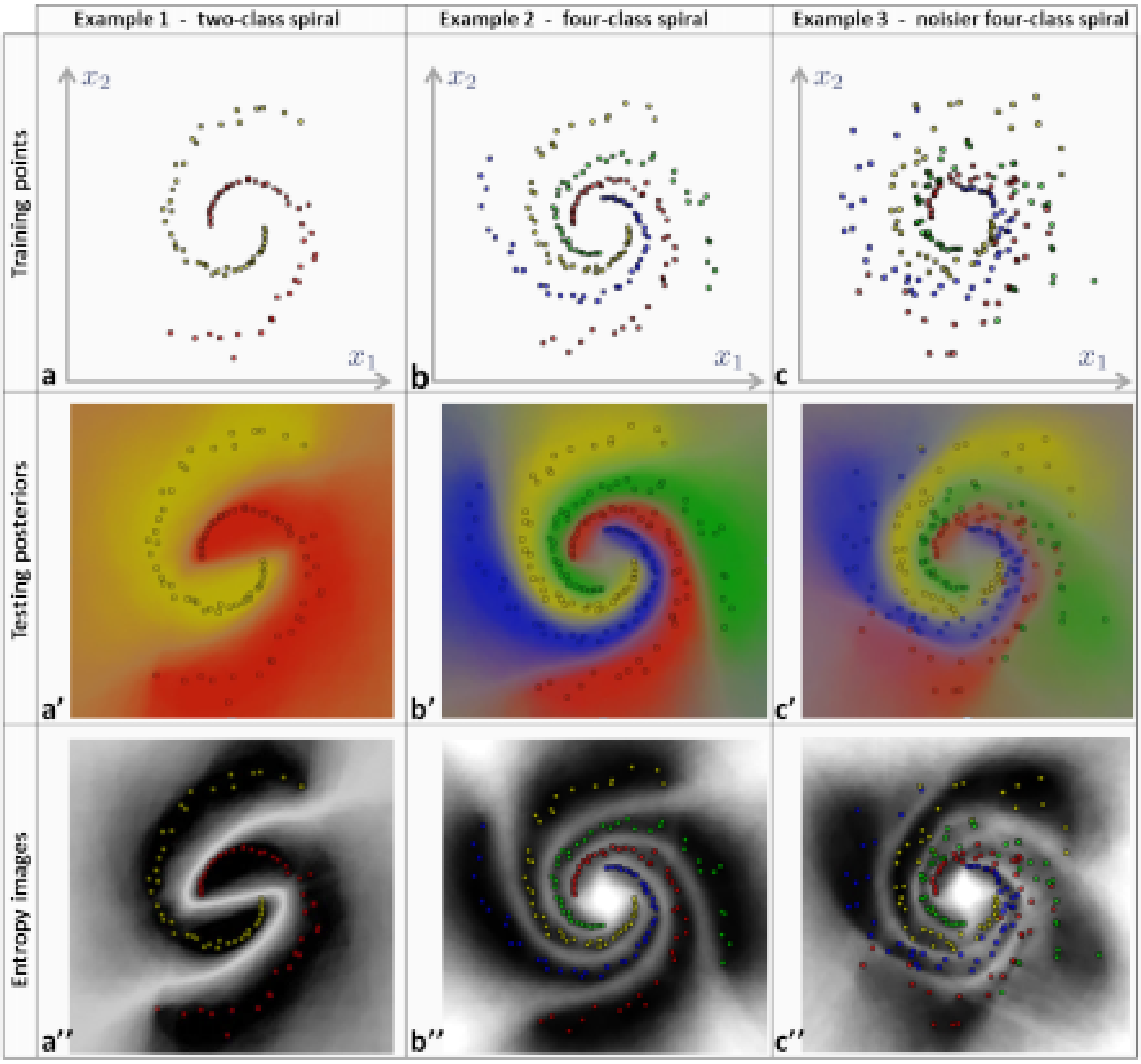


Fig. 3.4: **The effect of multiple classes and noise in training data. (a,b,c)** Training points for three different experiments: 2-class spiral, 4-class spiral and another 4-class spiral with noisier point positions, respectively. **(a',b',c')** Corresponding testing posteriors. **(a",b",c")** Corresponding entropy images (brighter for larger entropy). The classification forest can handle both binary as well as multi-class problems. With larger training noise the classification uncertainty increases (less saturated colours in c' and less sharp entropy in c"). All experiments in this figure were run with $T = 200$, $D = 6$, and a conic-section weak-learner model.

B.4.3 Sloppy lables and the effect ot the tree depth

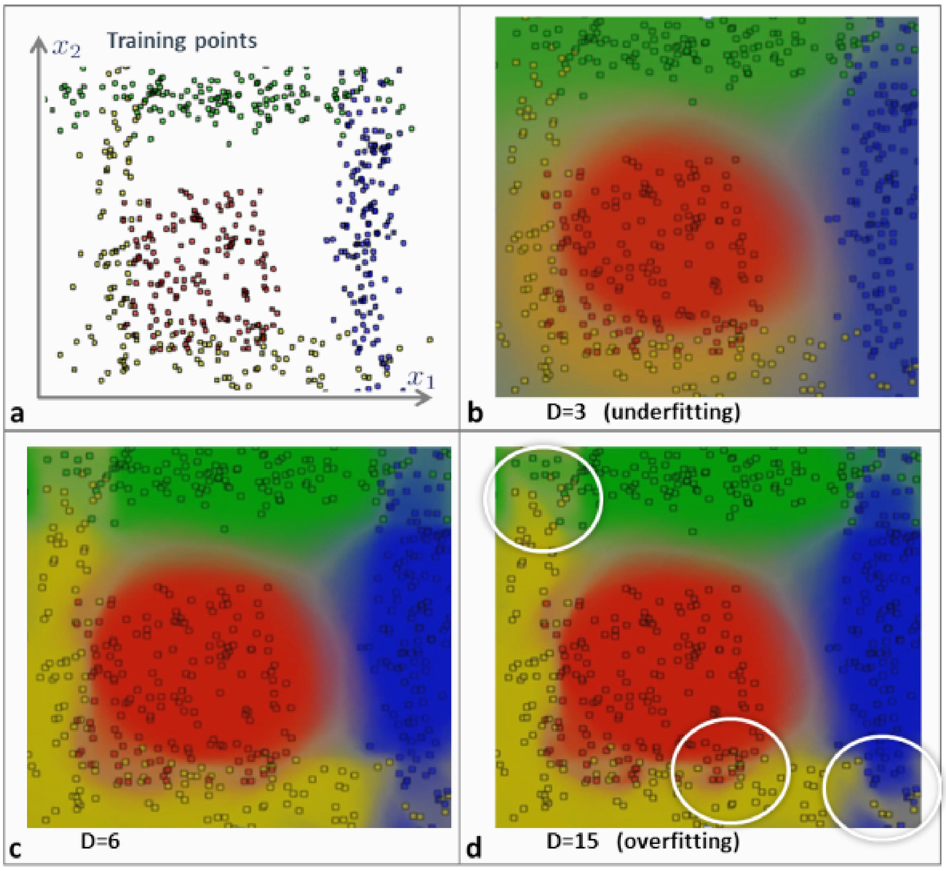


Fig. 3.5: **The effect of tree depth.** A four-class problem with both mixing of training labels and large gaps. **(a)** Training points. **(b,c,d)** Testing posteriors for different tree depths. All experiments were run with $T = 200$ and a conic weak-learner model. The tree depth is a crucial parameter in avoiding under- or over-fitting.

B.4.4 The effect of the weak learner

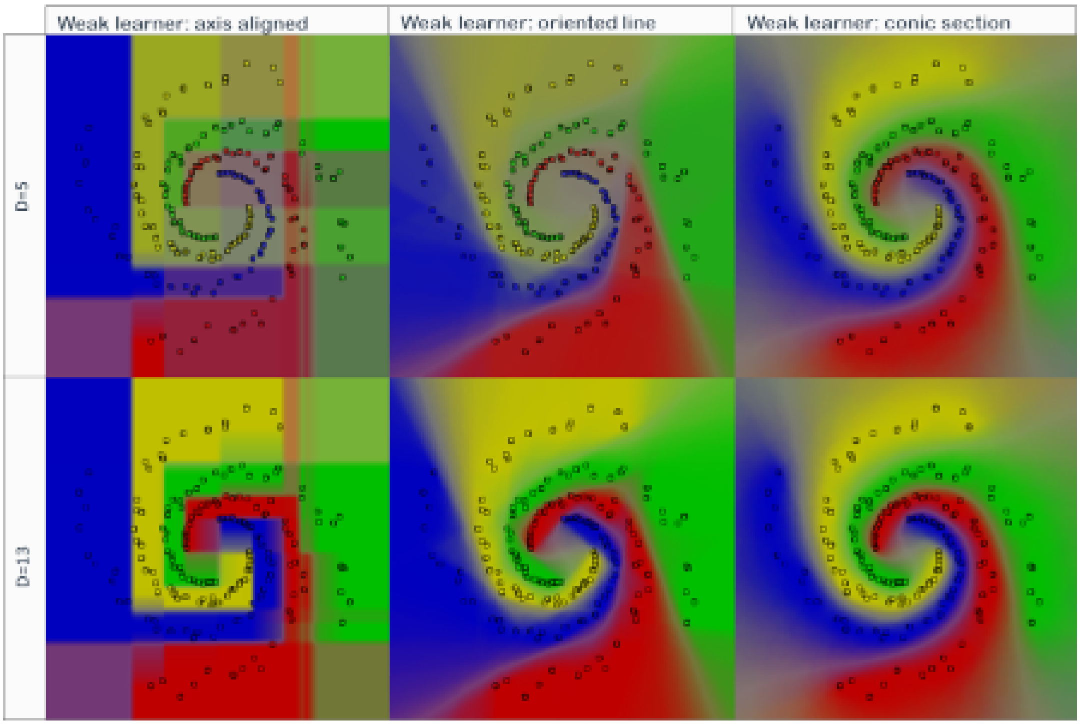


Fig. 3.6: **The effect of weak learner model.** The same set of 4-class training data is used to train 6 different forests, for 2 different values of D and 3 different weak learners. For fixed weak learner deeper trees produce larger confidence. For constant D non-linear weak learners produce the best results. In fact, an axis-aligned weak learner model produces blocky artifacts while the curvilinear model tends to extrapolate the shape of the spiral arms in a more natural way. Training has been achieved with $\rho = 500$ for all split nodes. The forest size is kept fixed at $T = 400$.

We should consider that axis aligned tests are extremely efficient to compute. So the choice of the weak learner has to be based on considerations of both accuracy and efficiency and depends on the specific application at hand.

B.4.5 The effect of randomness

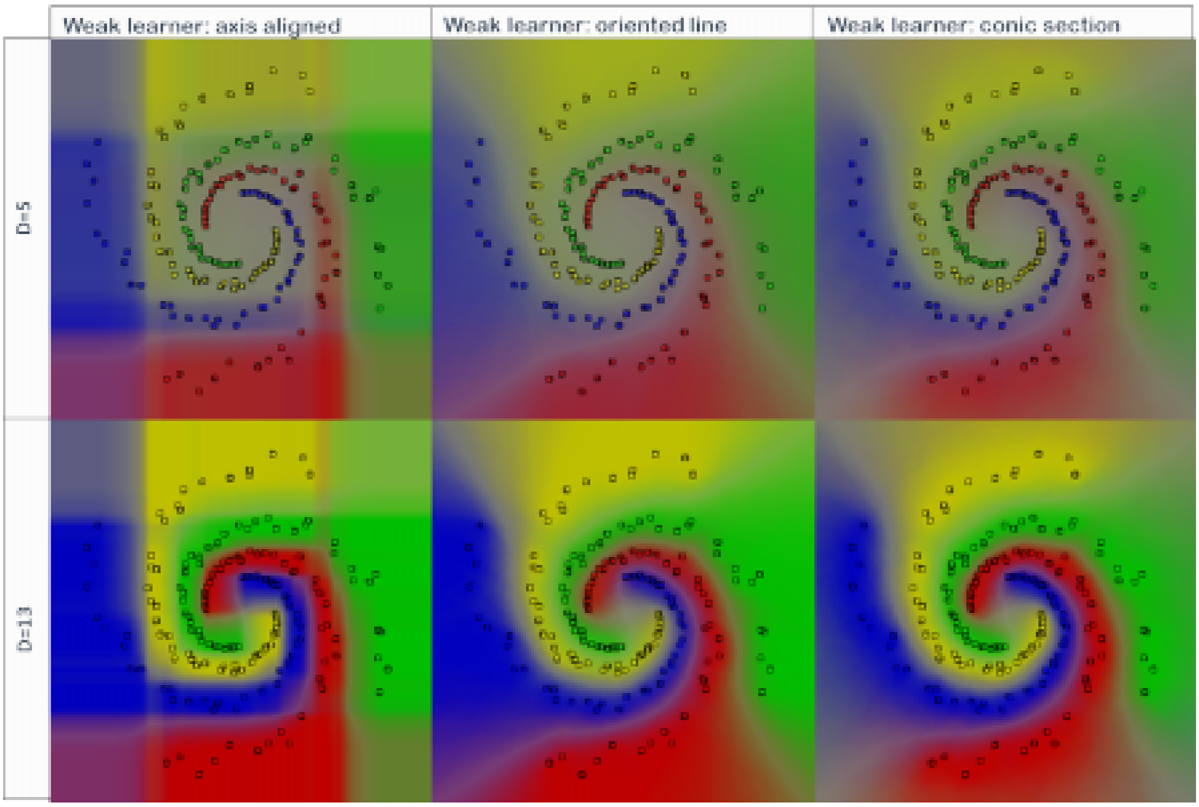


Fig. 3.7: **The effect of randomness.** The same set of 4-class training data is used to train 6 different forests, for 2 different values of D and 3 different weak learners. This experiment is identical to that in fig. 3.6 except that we have used much more training randomness. In fact $\rho = 5$ for all split nodes. The forest size is kept fixed at $T = 400$. More randomness reduces the artifacts of the axis-aligned weak learner a little, as well as reducing overall prediction confidence too. See text for details.

Larger randomness helps reduce a little the blocky artifacts of the axis-aligned weak-learner as it produces more rounded decision boundaries. Also more randomness reduces the overall confidence.

B.5 Maximum-margin properties

Imagine we have linearly separable 2-class training data set ($d = 2$), an axis aligned weak learner model and $D = 2$. As usual randomness is injected via randomized node optimization.

When training the root node of the frist tree, if we specify p large enough, the selected separating line tends to be placed somewhere within the gap. Any position within the gap is associated with exactly the same, maximum information gain. Thus, a collection of randomly trained trees produces a set of separating lines randomly placed within the gap.

If the candidate separating lines are sampled from a uniform distribution (as it is usually the case) then this would yield forest classification posteriors that vary within the gap as a linear ramp. If we are interested in a hard separation then the optimal separating surface is such that the posteriors for the two classes are identical.

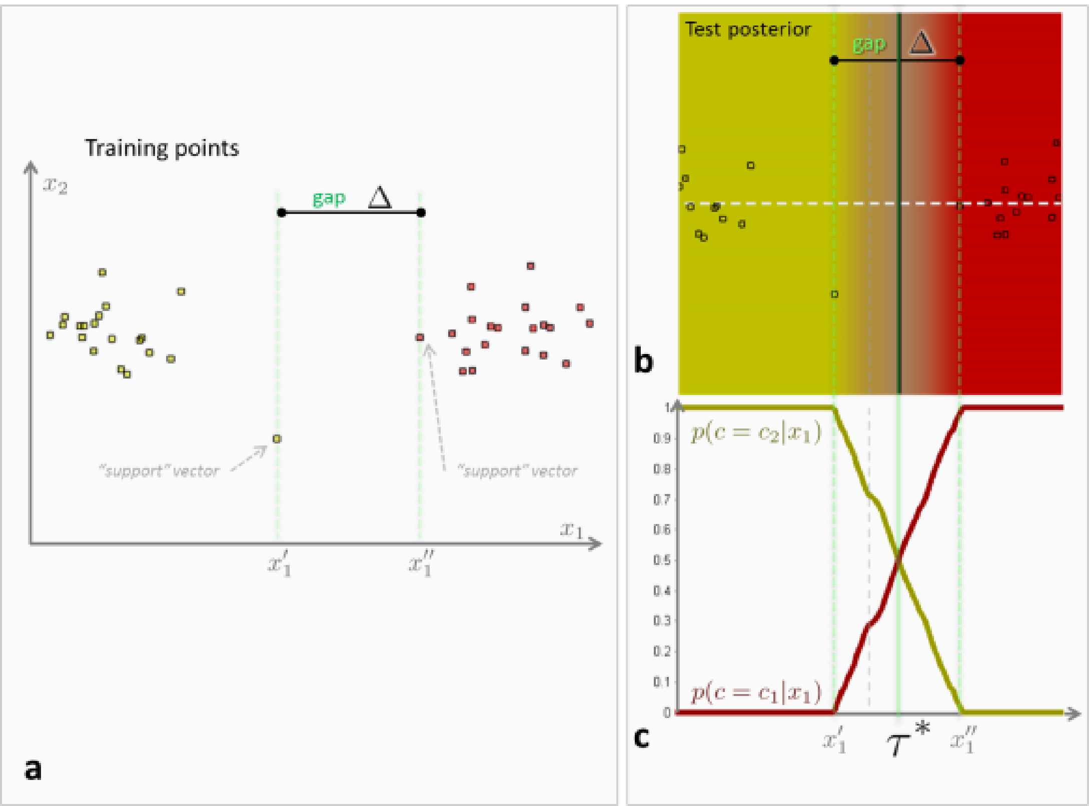


Fig. 3.8: **Forest's maximum-margin properties. (a)** Input 2-class training points. They are separated by a gap of dimension Δ. **(b)** Forest posterior. Note that all of the uncertainty band resides within the gap. **(c)** Cross-sections of class posteriors along the horizontal, white dashed line in (b). Within the gap the class posteriors are linear functions of x_1. Since they have to sum to 1 they meet right in the middle of the gap. In these experiments we use $\rho = 500, D = 2, T = 500$ and axis aligned weak learners.

Regression Forests

B.6 Specializing the decision forest model for regression

Task: *Given a labelled training set learn a general mapping which associates previously unseen independent test data with their correct continuous prediction.*

Formally, given a multi-variate input v we wish to associate a continuous multi-variate label $y \in Y \subseteq \mathbb{R}^n$. More generally, we wish to estimate the probability density function $p(y|v)$.

Why regression forests?

A regression tree splits a complex nonlinear regression problem into a set of smaller problems which can be more easily handled by simpler models (e.g. linear ones).

The prediction model

In classification we have used the pre-stored empirical class posterior as model. In regression forests we have a few alternatives (e.g. constant, linear, polynomial).

Though in this paper we use a probability density function over the continuous variable y. So, given the tth tree in a forest and an input v, the associated output takes the form $p_t(y|v)$.

The ensemble model

$$p(y|v) = \frac{1}{T}\sum_{1}^{T} p_t(y|v)$$

Randomness model

See classification forests.

The training objective function

As usual, the optimal parameter set is given by

$$\theta_j^* = \underset{\theta_j \in \mathcal{T}_j}{\operatorname{argmax}} I_j$$

where the main difference between classification and regression forests are in the objective function I_j.

Idea:

- Apply probabilistic linear fit to both subsets, yielding a best fitting line, and an additional probability density function $p(y|x)$ with $\sigma_y(x)$
- Take the split that minimizes the variance $\sigma_y(x)$ forall x in the subsets compared to before the split

---

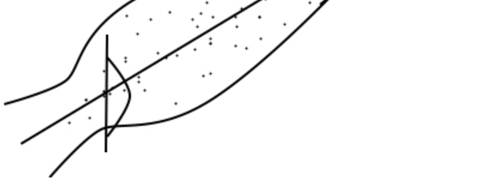
From lecture

The decision criterion for the splitting function works analogously to the classification case. The only difference ist that we need to define the entropy $H(S_j)$ on continous values:

$$H(S_j) = -\frac{1}{|S_j|} \cdot \sum_{\vec{x} \in S_j} \int_y p(y|x) \cdot log(y|x) dy$$

where $p(y|x)$ can, e.g. be chosen as a Gaussian distribution $p(y|x) = \mathcal{N}(y; \overline{y}(x), \sigma_y^2(x))$, where $\overline{y}(x)$ is a linear function and $\sigma_y(x)$ is the conditional variance computed from a linear fit.

Probabilistic linear fit:



Combining the expression for $p(y|x)$ into $H(S_j)$ yields

$$H(S_j) = \frac{1}{|S_j|} \cdot \sum_{\vec{x} \in S_j} \frac{1}{2} \cdot log((2\pi e)^2 \sigma_y^2(\vec{x}))$$

$$\Rightarrow I(S_j, \vartheta) = \sum_{\vec{x} \in S_j} log(\sigma_y(\vec{x})) - \sum_{i \in \{L,R\}} (\sum_{x \in S_j^i} log(\sigma_y(\vec{x})))$$

By comparison, the error or fit objective function for single-variate output y and constant predictors is

$$\sum_{v \in S_j} (y - \bar{y}_y)^2 - \sum_{i \in \{L,R\}} (\sum_{v \in S_j^i} (y - \bar{y_j})^2)$$

The weak learner model

See classification forest.

Density forests

B.7 Specializing the forest model for denisty estimation

Task: *Given a set of unlabelled observations we wish to estimate the probability density function from which such data has been generated.*

The desired output is the entire pdf $p(v) \geq 0, s.t. \int p(v)dv = 1$, for any generic input v.

What are density forests? A density forest is a collection of randomly trained clustering trees. The tree leaves contain simple prediction models such as Gaussians. So, loosely speaking a density forest can be thought of as a generalization of Gaussian miaxture models with two differences:

1. multiple hard clustered data partitions are created, one by each tree. This is in contrast to the single "soft" clustering generated by the EM algorithm.
2. the forest posterior is a combination of tree posteriors. So, each input data point is explained by multiple clusters (one per tree). This is in contrast to the single linear combination of Gaussians in a GMM.

The training objective function

Given a collection of unlabelled points $\{v\}$ we train each individual tree in the forest independently. As always:

$$\theta_j^* = \underset{\theta_j \in \mathcal{T}}{\operatorname{argmax}} I_j$$

with the generic information gain I_j defined as

$$I_j = H(S_j) - \sum_{i=L,R} \frac{|S_j^i|}{|S_j|} H(S_j^i)$$

No ground-truth labels ⇒ *unsupervised* entropy. As with a GMM we use the working assumption of multi-variate Gaussian distributions at the nodes. Then, the differential (continous) entropy of an d-variate Gaussian can be shown to be

$$H(S) = \frac{1}{2} log((2\pi e)^d |\Lambda(S)|)$$

where Λ is the associated $d \times d$ convariance matrix. Consequently, the information gain reduces to

$$I_j = log(|\Lambda(S_j)|) - \sum_{i \in \{L,R\}} \frac{|S_i^i|}{|S_j|} log(|\Lambda(S_j^i)|)$$

Motivation: Determinant of covariance matrix is a function of the volume of the ellispoid corresponding to that cluster. Maximizing the mentioned information gain trens to split the data into a number of compact clusters. The centers of those clusters tend to be placed in areas of high data density, while the separating surfaces are placed along regions of low density.

Prediction model Each v reaches one leave denoted as $l(v)$. The statistics of all training points arriving at each leaf node are summarized by a single multi-variate Gaussian distribution $N(v, \mu_{l(v)}, \Lambda_{l(v)})$. Then, the output of the tth tree is:

$$p_t(v) = \frac{\pi_{l(v)}}{Z_t} N(v; \mu_{l(v)}, \Lambda_{l(v)})$$

In order to ensure probabilistic normalization we need to incorporate the partition function

$$Z_t = \int_v \pi_{l(v)} N(v; \mu_{l(v)}, \Lambda_{l(v)}) dv$$

(The volume of the whole pdf, weighted by number of samples of training data inside each density region).

B.8 Effect of model parameters

Tree depth: Deeper trees tend to overfit heavily.

Forest size: The addition of further trees tends to produce smoother densities. This is thanks to the randomness of each tree density estimation. Bigger forests are always better (also for classification and regression).

B.9 Sampling from the generative model

1. Draw uniformily a random tree index $t \in \{1, T\}$ to select a single tree in the forest
2. Descend the tree
   - Starting at the root node, for each split node randomly generate the child index with probability proportional to the number of training points in edge (proportional to edge thickness)
   - Repeat step 2 until a leaf is reached
3. At the leaf draw a random sample from the *domain bounded* Gaussian stored at that leaf

---

Manifold forest

Advantage with respect to other techniques:

1. Computational efficiency
2. Automatic selection of discriminative features via information-based energy optimization
3. Being part of a more general forest model and, in turn code re-usability
4. Automatic estimation of the optimal dimensionality of the target space

B.10 Specializing the forest model for manifold learning

Task: *Given a set of k unlabelled observations $v_1, \ldots, v_k$ with $v_i \in R^d$ we wish to find a smooth mapping $f : R^d \rightarrow R^{d'}, f(v_i) = v'_i$ such that $d' \ll d$ and that preserves the observations' relative geodesic distance.*

C Undirected Graphical Models (Markov Random Fields) [4]

C.1 Introduction

- Consist of vertices (nodes) and edges joining some pairs of vertices
- Each vertex represents a random variable
- No edge between two vertices ⇒ conditionally independent, given the other variables
- Edges are parameterized by values or *potentials* that encode the strength of the conditional dependence between the random variables
- Main challenges: model selection (structure), estimation of edge parameters from data (learning), and computation of marginal vertex probabilities and expectations, from their joint distribution (inference)

C.2 Markov Graphs and Their Properties

A graph $\mathcal{G}$ consists of a pair (V, E), where V is a set of vertices and E the set of edges (defined by pairs of vertices). Two vertices X and Y are called adjacent if there is a edge joining them; this is denoted by $X \sim Y$. A *complete graph* is a graph with every pair of vertices joined by an edge.

In a Markov graph $\mathcal{G}$, the absence of an edge implies that the corresponding random variables are conditionally independent given the variables at the other vertices (known as *pairwise Markov independencies* of $\mathcal{G}$). Notation:

$$\text{No edge joining } X \text{ and } Y \Leftrightarrow X \perp Y | \text{rest}$$

If A, B and C are subgraphs, then C is said to *separate* A and B if every path between A and B intersects a node in C. Separators have the nice property that they break the graph into conditionally independent pieces (known as *global Markov properties* of $\mathcal{G}$). Notation:

$$\text{if } C \text{ separates } A \text{ and } B \text{ then } A \perp B | C$$

Pairwise and global Markov properties of a graph are equivalent. This is, the set of graphs with associated probability distributions that satisfy the pairwise Markov independencies and global Markov assumptions are the same. This is useful for inferring global independence relations from simple pairwise properties.

The global Markov property allows us to decompose graphs into smaller more manageable pieces and thus leads to essential simplifications in computation and interpretation. A *clique* is a complete subgraph - a set of vertices that are all adjacent to one another (*maximal* if no other vertices can be added to it).

A probability density function f over an Markov graph $\mathcal{G}$ can be represented as

$$f(x) = \frac{1}{Z} \prod_{C \in \mathcal{C}} \psi_C(x_C)$$

where $\mathcal{C}$ is the set of maximal cliques, and the postitive functions $\psi_C(\cdot)$ are called *clique potentials.* These are not in general density function, but rather are affinities that capture the dependence in X_c by scoring certain instances x_c higher than other. The quantity

$$Z = \sum_{x \in \mathcal{X}} \prod_{C \in \mathcal{C}} \psi_C(x_C)$$

[4] The Elements of Statistical Learning: Chapter 17, Undirected Graphical Models

is the normalizing constant, also known as *partition* function.

This definition of f implies a graph with independence properties defined by the cliques in the product. This holds for Markov networks $\mathcal{G}$ with positive distributions, and is known as the *Hammersley-Clifford* theorem.

Because we are restricted to potential functions which are stritly positive it is convenient to express them as exponentials, so that

$$\psi_C(x_C) = exp(-E(x_C))$$

where $E(x_C)$ is called an *energy function*, and the exponential representation is called the *Boltzmann distribution*. The joint distribution is defined as the product of potentials, and so the total energy is obtained by adding the energies of each of the maximal cliques.

C.3 Image de-noising [5]

Let the observed noisy image be described by an array of binary pixel values $y_i \in \{-1, +1\}$, where the index $i = 1, \dots, D$ runs over all pixels. We shall suppose that the image is obtained by taking an unknown noise-free image, described by binary pixel values $x_i \in \{-1, +1\}$ and randomly flipping the sign of pixels with some small probability.

Because the noise level is small, we know that there will be a strong correlation between x_i and y_i. We also know that neighbouring pixels x_i and x_j in an image are strongly correlated.

TODO: Random field model picture

This graph has two types of cliques, each of which contains two variables. The cliques $\{x_i, y_i\}$ have an associated energy function that expresses the correlation between these variables $(-\eta x_i y_i)$, where η is a positive constant) $\Rightarrow$ lower energy (higher probability) when x_i and y_i have the same sign.

The remaining cliques comprise pairs of variables $\{x_i, x_j\}$ where i, j are indices of neighbouring pixels. Again, we want the energy to be lower when the pixels have the same sign than when they have the opposite sign, and so we choose an energy given by $-\beta x_i x_j$, where β is a positive constant.

Additionally, we add a term hx_i for each pixel i in the noise-free image. Such a term has the effect of biasing the model towards pixel values that have one particular sign in preference to the other.

The complete energy function for the model then takes the form

$$E(x, y) = h \sum_i x_i - \beta \sum \{i, j\} x_i x_j - \eta \sum_i x_i y_i$$

which defines a joint distribution over x and y given by

$$p(x, y) = \frac{1}{Z} exp\{-E(x, y)\}$$

Note:

$$\begin{aligned} f(x) &= \frac{1}{Z} \prod_{C \in \mathbb{C}} \psi_C(x_C) \\ &= \frac{1}{Z} \prod_{C \in \mathbb{C}} exp\{-E(x_C)\} \\ &= \frac{1}{Z} exp\{-\sum_{C \in \mathbb{C}} E(x_C)\} \end{aligned}$$

[5] Bishop, Pattern Recognition: 8.3.3

We now fix the elements of y to the observed values given by the pixels of the noisy image, which implicitly defines a conditional distribution $p(x|y)$ over noise-free images (see lecture).

D Density Estimation

D.1 Parzen windows: Duda, Hart, Stork: "Pattern Classification", Section 4.3

Common parametric density functions rarely fit the densities actually encountered in practice.

Nonparametric procedures: can be used without the assumption that the forms of the underlying densities are known.

D.1.1 Density estimation

The proability P that a vector x will fall in a region $\mathcal{R}$ is given by

$$P = \int_{\mathcal{R}} p(x')\ dx'.$$

where P is a smoothed or averaged version of the denisty function $p(x)$.

Suppose that n samples $x_1, \dots, x_n$ are drawn independently from $p(x)$. We expect that the ratio k/n, where k is the number of samples inside the region, will be a very good esimate for P. This estimate is especially accurate when n is very large. If we now assume that $p(x)$ is continous and that the region $\mathcal{R}$ is so small that p does not vary consideratly within it, we can write

$$\int_{\mathcal{R}} p(x')dx' \simeq p(x)V,$$

where x is a point within $\mathcal{R}$ and V is the volume enclosed by $\mathcal{R}$. Combining all the equations:

$$\begin{aligned} P &\simeq p(x)V \\ k/n &\simeq p(x)V \\ p(x) &\simeq \frac{k/n}{V} \end{aligned}$$

D.1.2 Parzen Windows (Kernel density estimation)

For now: R_n is a d-dimensional hypercube. If h_n is the length of an edge of that hypercube, then its volume is given by

$$V_n = h_n^d$$

In order to obtain an analytic expression for k_n, we define the window function

$$\varphi(u) = \begin{cases} 1 & |u_j| \leq 1/2, \quad j = 1, \dots, d \\ 0 & otherwise \end{cases}$$

where $\varphi(u)$ indicates if u lies within a unit hypercube centered at the origin.

It follows that $\varphi((x - x_i)/h_n)$ equals 1 if x_i falls within the hypercube of volume V_n centered at x, and is zero otherwise.

The number of samples in this hypercube is therefore given by

$$k_n = \sum_{i=1}^{n} \varphi(\frac{x - x_i}{h_n})$$

and we obtain

$$p_n(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{V_n}\varphi(\frac{x - x_i}{h_n})$$

We don't have to restrict ourselfs to the hypercube window function (we can use for example Gaussian windows etc.). In essence, the window function is being used for **interpolation** - each samples contribution to the estimate in accordance with its disntace from x.

We can rewrite this by defining a function

$$\delta_n(x) = \frac{1}{V_n}\varphi(\frac{x}{h_n})$$

and then obtain $p_n(x)$ as the average

$$p_n(x) = \frac{1}{n}\sum_{i=1}^{N}\delta_n(x - x_i)$$