COMP 432 Machine Learning

Probability Density Estimation
Part 2

Computer Science & Software Engineering
Concordia University, Fall 2023

Summary of the last episode….

We have seen many **supervised learning** techniques:

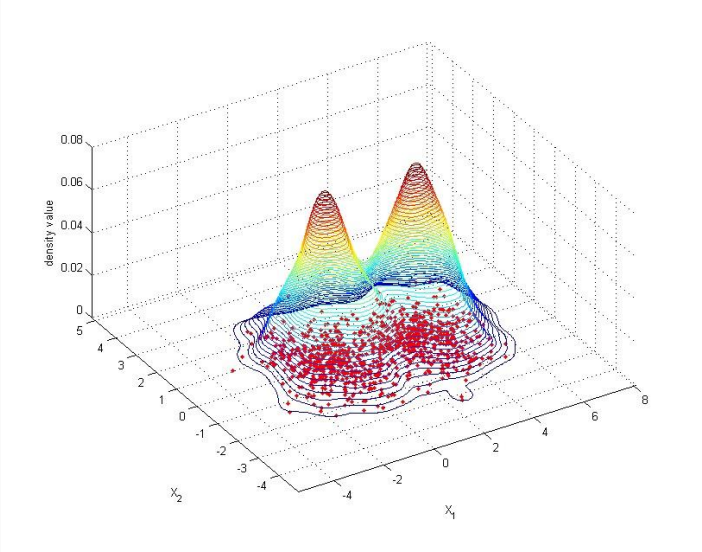
- Clustering (K-means)
- Density Estimation

Today we are going to continue our discussion on **unsupervised learning:**

- Gaussian Mixture Models (GMMs)
- Parzen Windows
- Feature Selection and Transformation

Density Estimation

- In the last lecture, we introduced the problem of **density estimation**.
- **Goal**: given a dataset **X**, we want to estimate p(**x**).



- One approach consists to choose a **type of probability distribution** (e.g. Gaussian).
- Then, we can use **Maximum Likelihood Estimation** to find the parameters **θ** that best fit the input data **X**:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ p(\mathbf{X} \mid \boldsymbol{\theta})$$

Density Estimation

- The maximum likelihood estimation for the parameters of a **1D Gaussian** is the following:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})^2$$

- Similarly, for a **D-dimensional Gaussian** we have:

$$\hat{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i \qquad \hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})^T$$

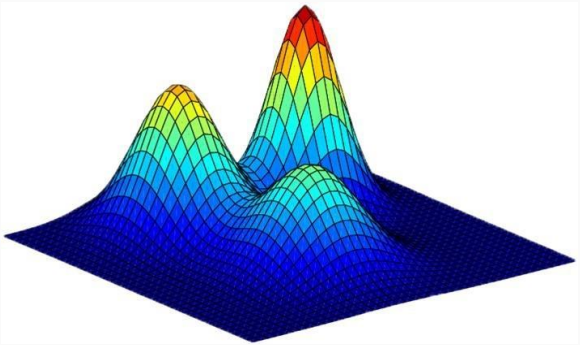
D-dimensional vectors

(DxD) (Dx1) (1xD)

Gaussian Mixture Models

- Despite many phenomena can be modeled with a Gaussian (due to the central limit theorem), many others cannot be described with such a simple model.

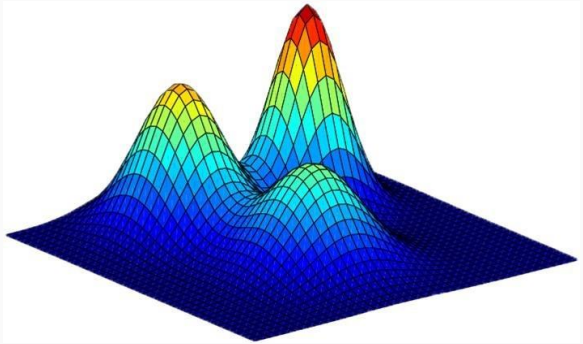
What about using multiple Gaussians?



This model is called "**Gaussian Mixture Model**" (GMM)

- With multiple Gaussians, we can model more **complex** probability density functions!

Gaussian Mixture Models



The analytical expression for a Gaussian Mixture Model is the following:

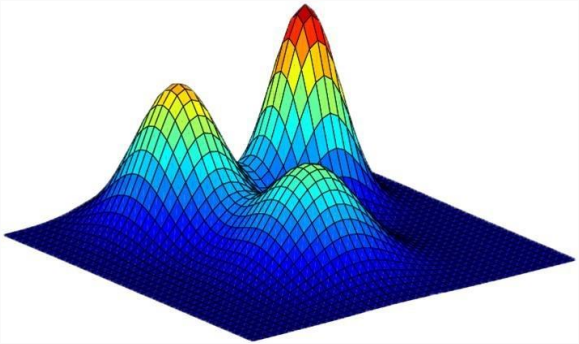
$$p(\mathbf{x} \mid \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

"Height" — Mean — Covariance Matrix

$$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi)^{D/2}|\Sigma_k|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right]$$

- We sum up **K Gaussians** with different height, mean, and covariance matrices.

Gaussian Mixture Models



The analytical expression for a Gaussian Mixture Model is the following:

$$p(\mathbf{x} \mid \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

"Height" — Mean — Covariance Matrix

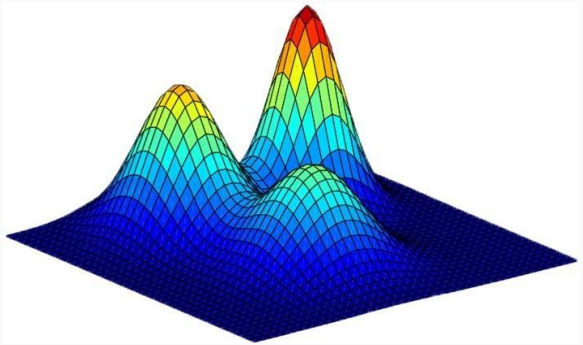
- To turn the expression above into a probability density function, the following **constraints** must hold:

$$\pi_k \geq 0 \text{ for all k in 1,..,K}$$

$$\sum_{k=1}^{K} \pi_k = 1$$

This derives from the fact that the probability must be **non-negative** and must **sum up to one**.

Gaussian Mixture Models



The analytical expression for a Gaussian Mixture Model is the following:

$$p(\mathbf{x} \mid \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

$$\mathbf{x} = [x_1, x_2, \ldots, x_D]^T$$ *(D x 1)* ⟶ Single input vector

$$\boldsymbol{\pi} = [\pi_1, \pi_2, \ldots, \pi_K]^T$$ *(K x 1)* ⟶ Height vector

Mean Matrix

$$\boldsymbol{\mu} = [\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K]^T = \begin{bmatrix} \mu_{1,1} & \cdots & \mu_{1,D} \\ \cdots & \cdots & \cdots \\ \mu_{K,1} & \cdots & \mu_{K,D} \end{bmatrix}$$

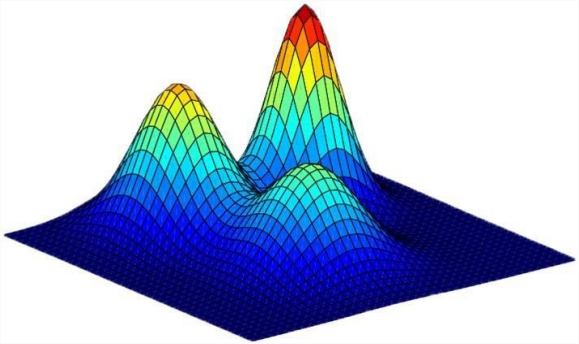
(K x D)

Covariance Tensor

$$\boldsymbol{\Sigma} = [\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_K]^T$$

(D x D x K) ⟶ Tensor collecting all the covariance matrices

Gaussian Mixture Models



Considering the N samples of the dataset **X** (iid), we can write the likelihood in the following way:

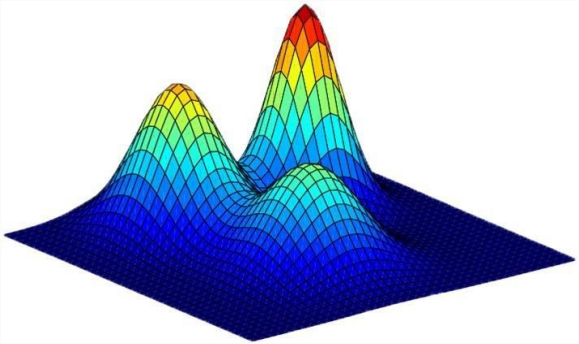
$$p(\mathbf{X} \mid \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

(N x D) Matrix

- As done before, we have to find the parameters that maximize the likelihood:

$$\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}}{\operatorname{argmax}} \prod_{i=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Gaussian Mixture Models



$$\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\Sigma}}{\operatorname{argmax}} \prod_{i=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

↓ Apply logarithm

$$\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Let's try to solve this problem!

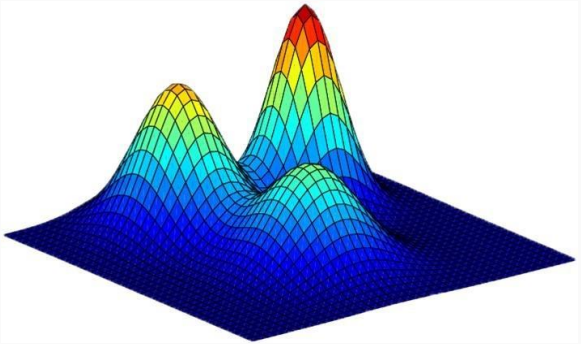
- *Is the objective function **convex**?* NO
- Does a ***close-form*** *(direct) solution exist?* NO
- Is the objective **differentiable** wrt to the parameters? YES

Do you think we can use gradient descent then?

YES

This is possible!

Gaussian Mixture Models



$$\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

- Even though it is possible to solve this problem with **gradient descent**, we normally use the **Expectation-Maximization** Algorithm.

- Because it turned out to be more **efficient** and reach the local minimum **faster.** Moreover, it does not introduce “annoying” hyperparameters such as the **learning rate**.

Gaussian Mixture Models

$$\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

- Similar to the k-means algorithm, the “trick” here is to optimize **one variable at a time** (and freeze the others).
- For GMMs, we have three types of optimization variables and we have to solve the following problems:

Problem 1 (fix Σ π, solve for μ)

$$\hat{\boldsymbol{\mu}} = \underset{\boldsymbol{\mu}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Problem 2 (fix π, μ solve for Σ)

$$\hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Problem 3 (fix Σ, μ solve for π)

$$\hat{\boldsymbol{\pi}} = \underset{\boldsymbol{\pi}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Such that:

$$\sum_{k=1}^{K} \pi_k = 1 \quad \pi_k \geq 0 \text{ for all k in 1,..,K}$$

Constrained optimization!

Gaussian Mixture Models

Problem 1 (fix Σ π, solve for μ)

$$\hat{\boldsymbol{\mu}} = \underset{\boldsymbol{\mu}}{\operatorname{argmax}} \underbrace{\sum_{i=1}^{N} \ln \Big(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \Big)}_{LL}$$

⟶ A direct solution exists!

- We can compute the gradient of LL:

$$\nabla_{\boldsymbol{\mu}_k} LL = \nabla_{\boldsymbol{\mu}_k} \Big[\sum_{i=1}^{N} \ln \Big(\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \Big) \Big]$$

$$= \sum_{i=1}^{N} \nabla_{\boldsymbol{\mu}_j} \ln \Big(\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \Big)$$

(gradient of sum = sum of gradients)

Gaussian Mixture Models

$$\nabla_{\boldsymbol{\mu}_k} LL = \sum_{i=1}^{N} \nabla_{\boldsymbol{\mu}_j} \ln \Big(\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \Big)$$

$$= \sum_{i=1}^{N} \frac{\nabla_{\boldsymbol{\mu}_k} [\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)]}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

(gradient of the logarithm)

$$= \sum_{i=1}^{N} \frac{\pi_k \nabla_{\boldsymbol{\mu}_k} [\mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

(The only gradient which is non zero is when j=k)

$$\nabla_{\boldsymbol{\mu}_k} \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k) \longrightarrow$$

Derivative of the Gaussian wrt $\boldsymbol{\mu}_k$

Gaussian Mixture Models

$$\nabla_{\boldsymbol{\mu}_k} LL = \sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k)$$

(Replacing Gaussian Derivatives)

$$\sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k) = 0$$

(Set gradient to zero)

$$\sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} (\mathbf{x}_i - \boldsymbol{\mu}_k) = 0$$

(Multiply by $\boldsymbol{\Sigma}_k$)

γ_{ik}

This term is called **responsibility** (we will discuss more in detail later)

Gaussian Mixture Models

$$\sum_{i=1}^{N} \gamma_{ik}(\mathbf{x}_i - \boldsymbol{\mu}_k) = 0$$

$$\sum_{i=1}^{N} \gamma_{ik}\mu_k = \sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i$$

(Algebraic Manipulation)

$$\mu_k \sum_{i=1}^{N} \gamma_{ik} = \sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i$$

$\sum_{i=1}^{N} \gamma_{ik}$ → N_k

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik}\mathbf{x}_i$$

Gaussian Mixture Models

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i$$

This is the same as a single Gaussian, except that the **mean** is **weighted**.

The weight γ_{ik} depends on "how far" the gaussian is from the point $\mathbf{x}_i$

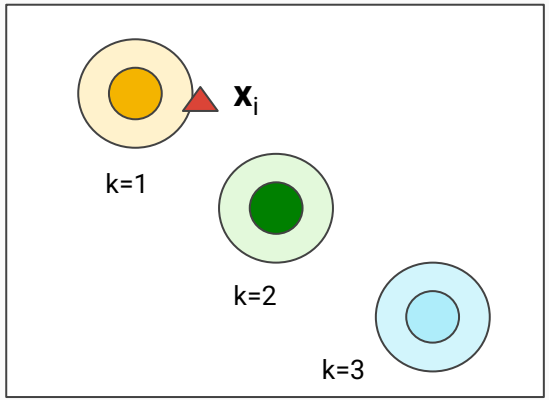
Points that are **far** from $\boldsymbol{\mu}_k$ give a **little contribution** to the mean.

Points that are **close** to $\boldsymbol{\mu}_k$ give a **large contribution** to the mean.

This term γ_{ik} is called r**esponsibility**.

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

It represents the probability that the sample $\mathbf{x}_i$ has been generated by the Gaussian k



γ_{i1}= 0.70 → This is the highest because $\mathbf{x}_i$ is closer to gaussian 1

γ_{i2}= 0.25

γ_{i3}= 0.05

Sum up to 1

Gaussian Mixture Models

- If we solve the other two problems, we have the following solutions(*):

Problem 2 (fix π, μ solve for Σ)

$$\hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\Sigma}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

solution

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T$$

This is the same as a single Gaussian, but again each point is weighed by γ_{ik}

Problem 3 (fix Σ, μ solve for π)

$$\hat{\boldsymbol{\pi}} = \underset{\boldsymbol{\pi}}{\operatorname{argmax}} \sum_{i=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

Such that:

$$\sum_{k=1}^{K} \pi_k = 1 \quad \pi_k \geq 0 \text{ for all k in 1,..,K}$$

solution

$$\pi_k = \frac{N_k}{N}$$

The k gaussian is high if many points are close to it.

* More details on the Bishop's Book

Gaussian Mixture Models

- If we put everything together, we obtain the following **Expectation-Maximization** algorithm:

1. **Random Initialization** of **Σ π μ**
2. **E step**: evaluate each responsibility γ_{ik}.
3. **M step**: update model parameters for all k.
4. **Repeat** E step and M step until **convergence**.

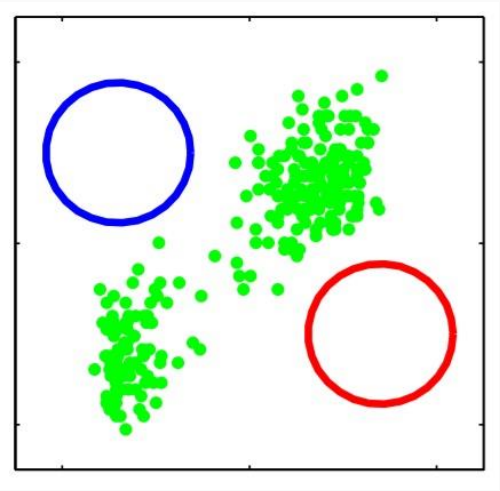
$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i \qquad \pi_k^{new} = \frac{N_k}{N}$$

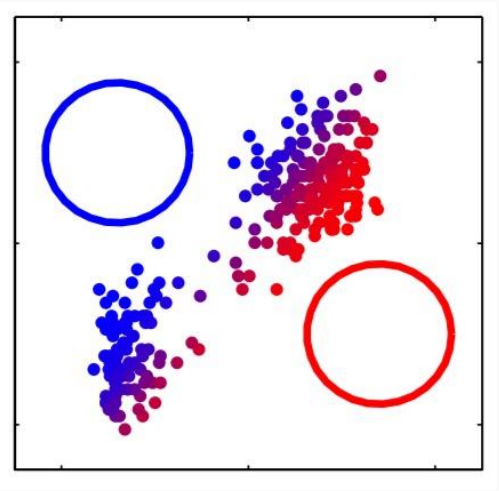
$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N} \sum_{i=1}^{N} \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T$$

Gaussian Mixture Models

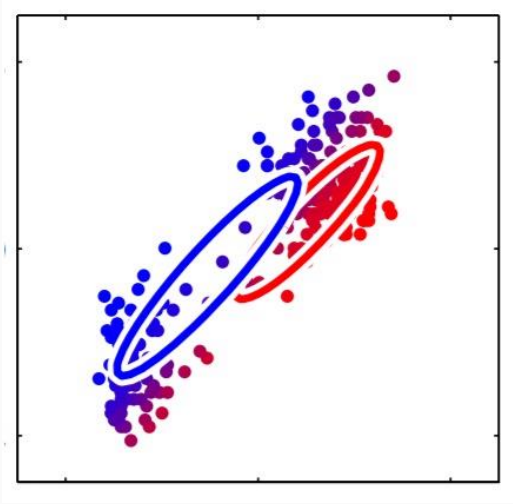
Initialization



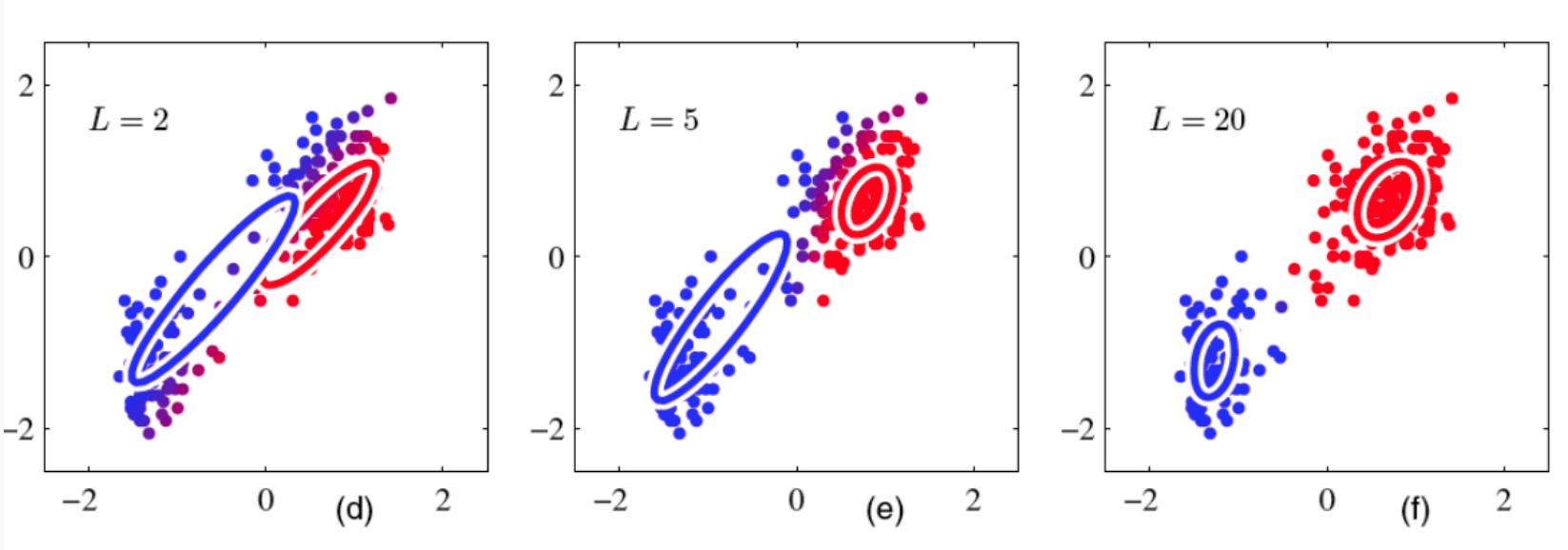
E-Step (compute γ_{ik})



M-Step (update **Σ π μ**)



Gaussian Mixture Models



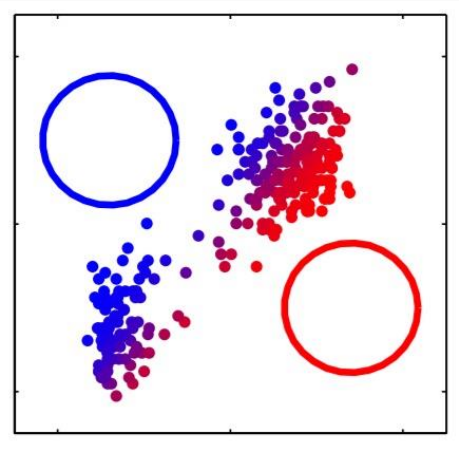
- We can repeat E-step and M step over and over until the parameters do not change significantly anymore.

GMM versus K-means

- *Does this algorithm remind you of something?*

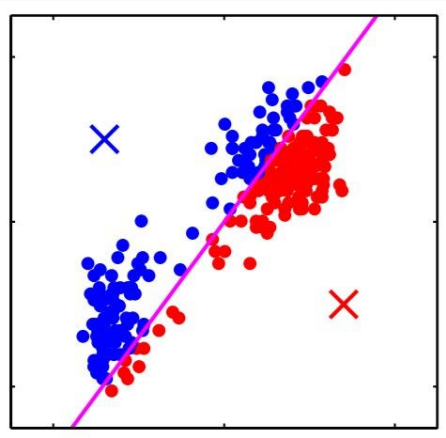
E-step for GMMs

$$\gamma_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$



E-step for k-means

$$r_{ik} = \begin{cases} 1 \text{ if } k = \underset{\mathbf{j}}{\operatorname{argmin}} ||\mathbf{x}_i - \boldsymbol{\mu}_j||^2 \\ 0 \text{ otherwise} \end{cases}$$



Key difference

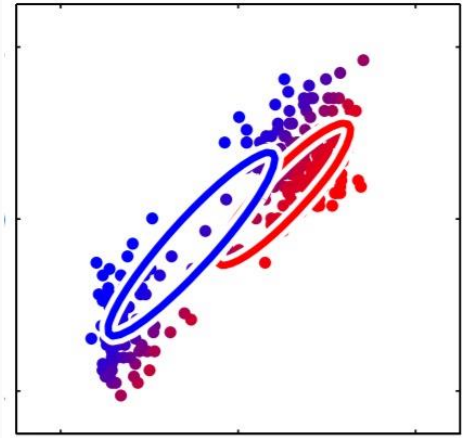
- In k-means we perform a **hard assignment** of data points to the cluster.
- In GMMs we perform a **soft assignment** based on γ_{ik}.

GMM versus K-means

- *Does this algorithm remind you of something?*

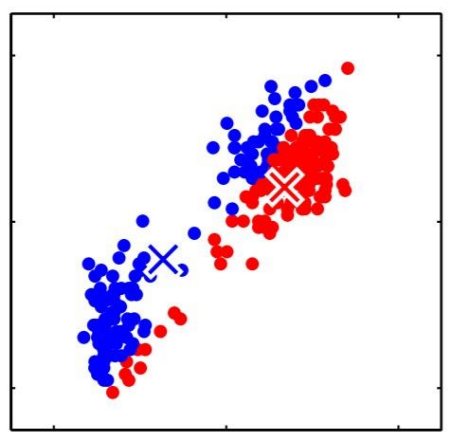
M-step for GMMs

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} \mathbf{x}_i$$



M-step for k-means

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} r_{ik} \mathbf{x}_i}{\sum_{i=1}^{N} r_{ik}}$$



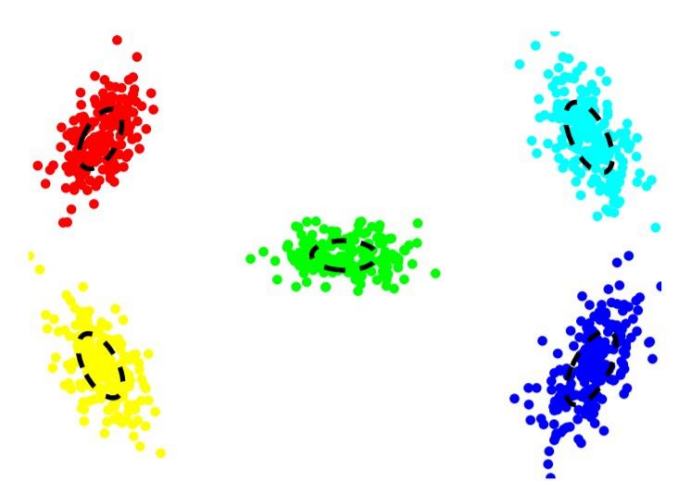
Key difference

- In k-means the mean is the average of samples assigned to the cluster k.
- In GMMs, the mean is a weighted average of all samples.

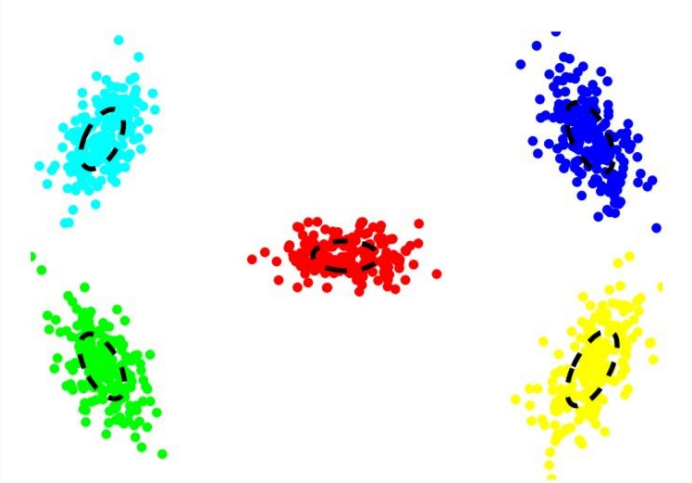
GMM versus K-means

- Both work well for non-overlapping “*ball-like*” clusters:

EM algorithm for GMMs



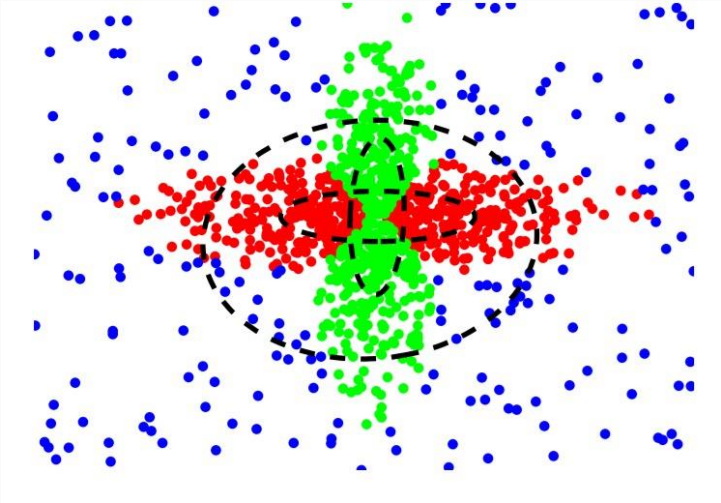
Weighted elliptical K-means algorithm



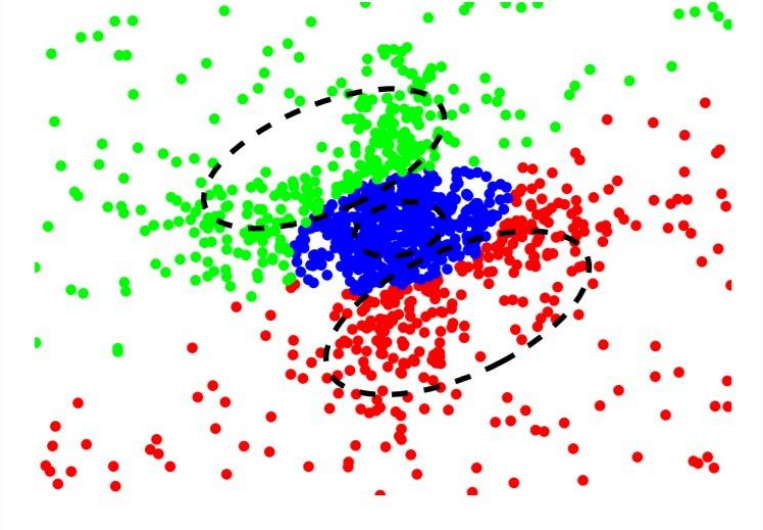
GMM versus K-means

- GMMs work better for overlapping structures:

EM algorithm for GMMs

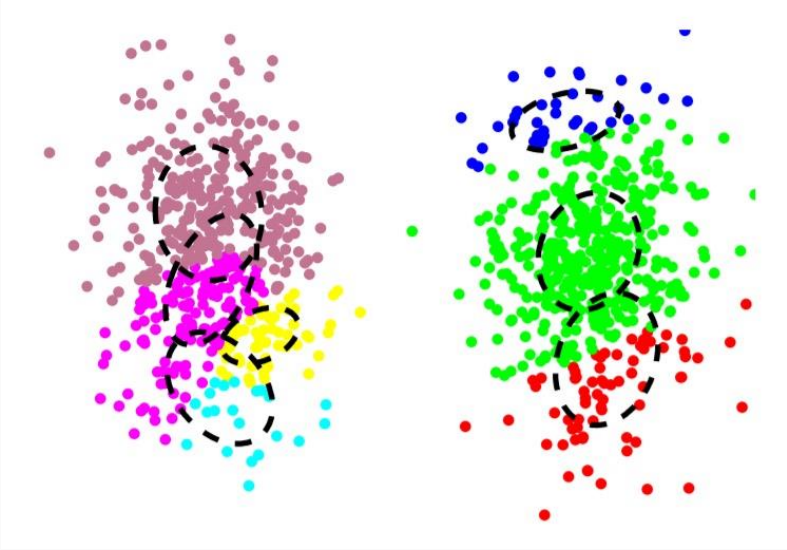


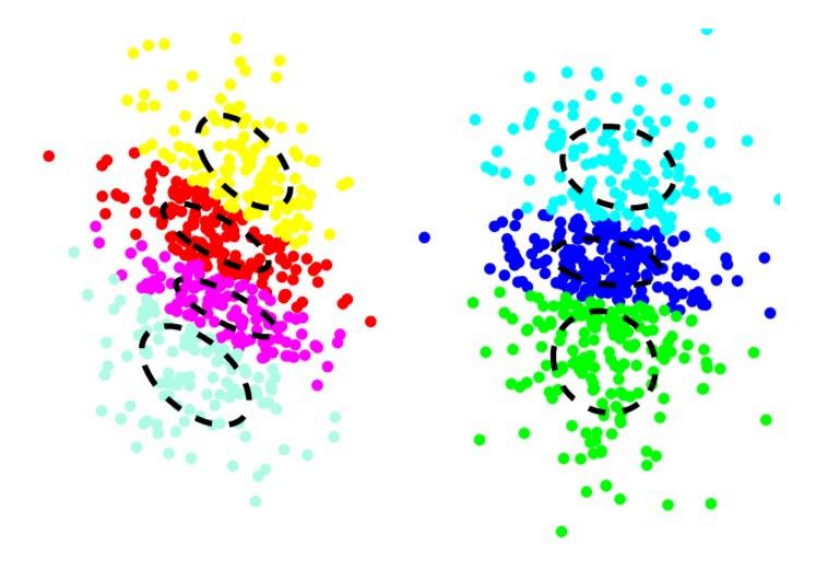
Weighted elliptical K-means algorithm



GMM versus K-means

- Both get stuck in **local minima**.
- Both sensitive to **guessing** number of components **K**.





GMM versus K-means

- K-means tends to converge much **faster**.

- Common practice:
  1. Run K-means to provide “smart” initialization to GMM.
  2. Run EM algorithm to refine GMM parameters.

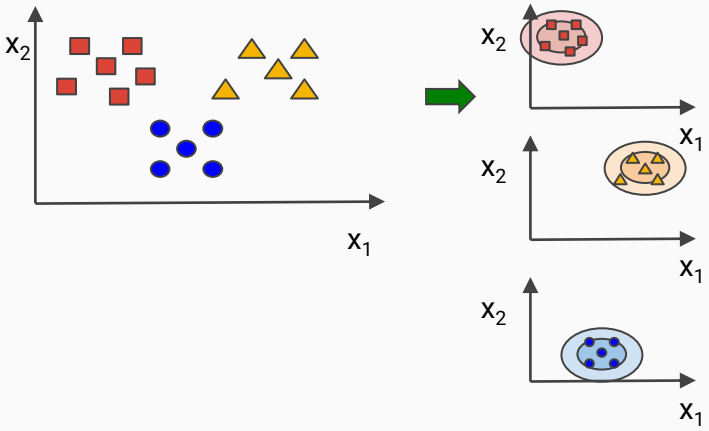
- K-means can be used for **clustering** only, while GMMs are **generative models** that can be used for:
  - Probability Density Estimation
  - Clustering
  - Classification (supervised learning)
  - Data Generation

GMMs for Classification

Prior Probability

Likelihood

$$k^* = \underset{k}{\operatorname{argmax}}\ P(y_k)p(x_1, \ldots, x_D \mid y_k)$$



$p(\mathbf{x}|y_1)\ P(y_1) = 0.01$

$p(\mathbf{x}|y_2)\ P(y_2) = 0.05$

$p(\mathbf{x}|y_3)\ P(y_3) = 0.02$

argmax

Class 2

1. We perform a different **pdf estimation** for all the classes.

2. We estimate the **prior probability** by computing the fraction of samples that belong to class K.

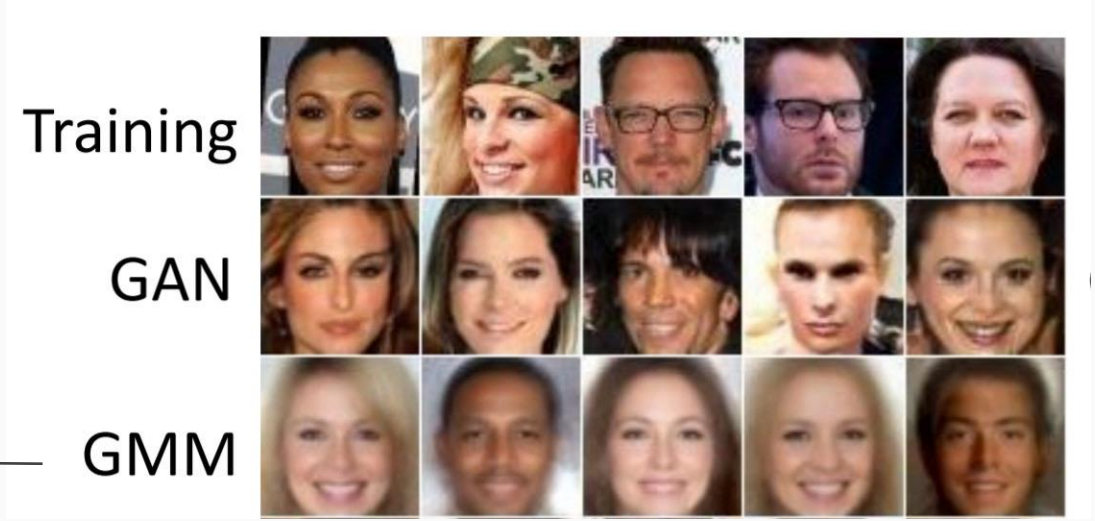
3. At test/inference time, we vote for the class with the **highest joint probability** (product between the prior probability and the likelihood).

GMMs for Data Generation

- After estimating the probability density function, we can easily **sample** from it!

1. *Sample integer k in {1,.., K} with probability π_k of being selected*
2. *Sample a data point according to:*

$$\mathbf{x} \sim \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$



Despite its simplicity, a GMM can capture complex high-dimensional structure

Scikit-learn

`sklearn.mixture`.GaussianMixture

class sklearn.mixture.GaussianMixture(*n_components=1*, ***, *covariance_type='full'*, *tol=0.001*, *reg_covar=1e-06*, *max_iter=100*, *n_init=1*, *init_params='kmeans'*, *weights_init=None*, *means_init=None*, *precisions_init=None*, *random_state=None*, *warm_start=False*, *verbose=0*, *verbose_interval=10*) [source]

Gaussian Mixture.

Representation of a Gaussian mixture model probability distribution. This class allows to estimate the parameters of a Gaussian mixture distribution.

Read more in the User Guide.

New in version 0.18.

Parameters:

n_components : *int, default=1*
The number of mixture components.

covariance_type : *{'full', 'tied', 'diag', 'spherical'}, default='full'*
String describing the type of covariance parameters to use. Must be one of:

- 'full': each component has its own general covariance matrix.
- 'tied': all components share the same general covariance matrix.
- 'diag': each component has its own diagonal covariance matrix.
- 'spherical': each component has its own single variance.

tol : *float, default=1e-3*
The convergence threshold. EM iterations will stop when the lower bound average gain is below this threshold.

reg_covar : *float, default=1e-6*
Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter : *int, default=100*
The number of EM iterations to perform.

n_init : *int, default=1*
The number of initializations to perform. The best results are kept.

init_params : *{'kmeans', 'random'}, default='kmeans'*

Bayesian Estimation

- With Maximum Likelihood Estimation we find the parameters that maximize p(**X**|**θ**):

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\mathbf{X} \mid \boldsymbol{\theta})$$

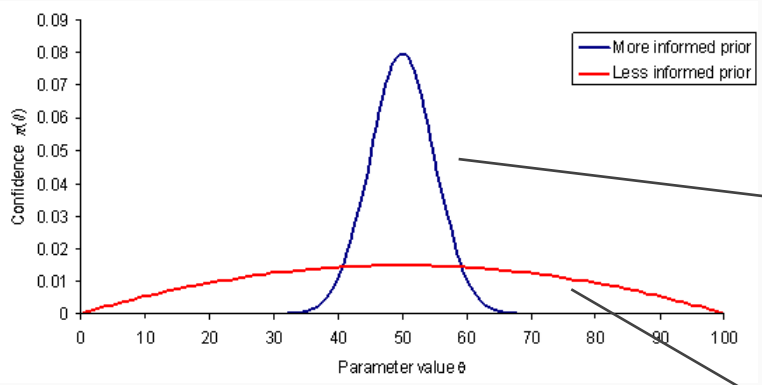
- What about if we have some **prior knowledge** on which set of parameters is good and which is bad?

- **Bayesian Estimation** is a statistical framework that allows one to encode prior knowledge on the parameters to estimate.

- This prior knowledge on the parameters can derive from our **experience** or from the previous related experiments we have done.

Bayesian Estimation

- The prior knowledge is encoded with a **prior distribution** p(**θ**):



With a prior distribution, we inform the estimator that some parameters are more likely than others.

This prior distribution gives **strong** “hints” on the set of parameters (because only a small set of parameters has a high probability).

This prior distribution gives **weak** “hints” on the set of parameters (because all the parameters are approximatively equally likely).

Intuitively, adding prior knowledge is a way to perform **regularization** and counteract **overfitting**.

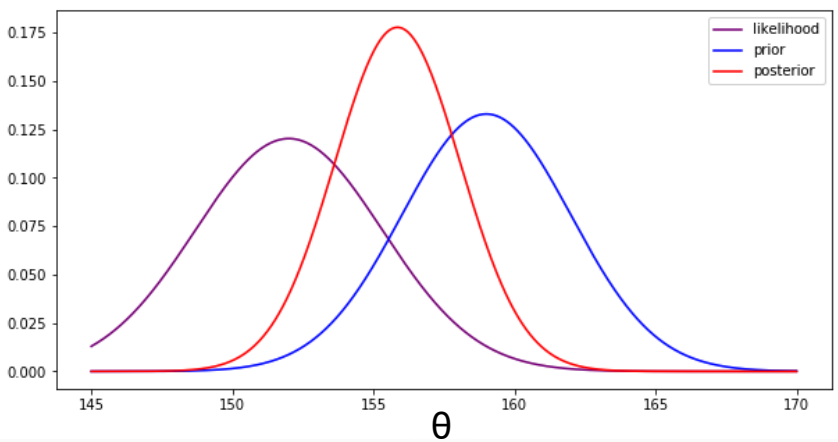
Bayesian Estimation

- Maximum likelihood estimates the **likelihood function**: $p(\mathbf{X} \mid \boldsymbol{\theta})$
- Bayesian estimation estimates the **posterior probability**:

Likelihood Prior

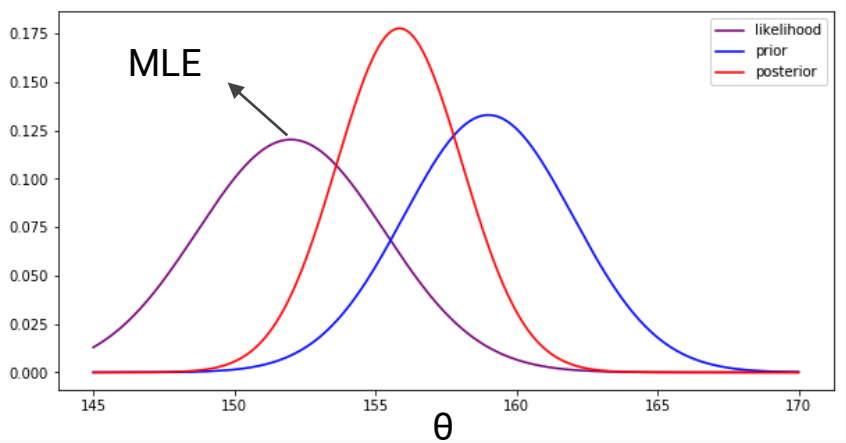
$$p(\boldsymbol{\theta} \mid \mathbf{X}) = \frac{p(\mathbf{X} \mid \boldsymbol{\theta}) \cdot p(\boldsymbol{\theta})}{p(\mathbf{X})}$$

Posterior Evidence



Bayesian Estimation

- *Once we have the posterior probability, how can we use them to compute the probability p(**x**) of new data?*



MLE Prediction:

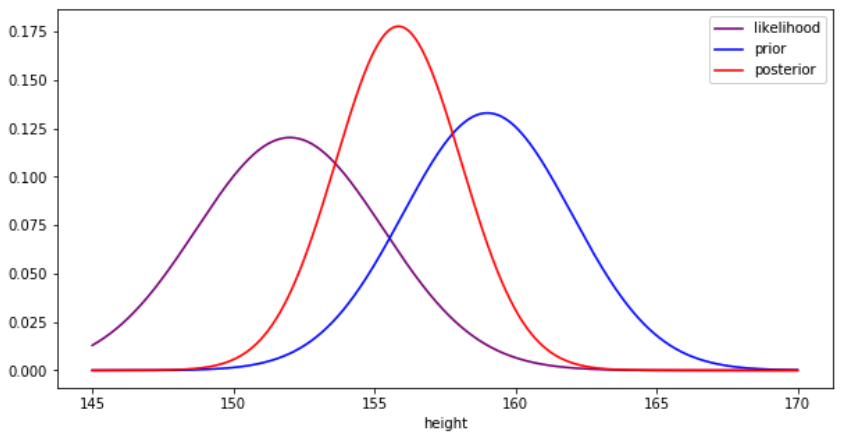
$$p(\mathbf{x} \mid \mathbf{X}) = p(\mathbf{x} \mid \hat{\boldsymbol{\theta}})$$

To be precise we have to write p(**x**) as p(**x**|**X**) because the prediction on the new sample x depends on the training dataset **X**.

We compute the probability of the new input x by using the best parameters found at training time.

- *What can we do at prediction time in the context of Bayesian estimation?*

Bayesian Estimation



θ

One reasonable approach would be to take the parameters that **maximize the posterior probability**.

Instead, we perform inference in the following way:

$$p(\mathbf{x} \mid \mathbf{X}) = \int p(\boldsymbol{\theta} \mid \mathbf{X}) p(\mathbf{x} \mid \boldsymbol{\theta}) d\theta$$

Marginalization

Posterior probability

Probability of observing **x** with the current parameters

- Bayesian predictions are a little more complex than MLE ones because we are not considering a single model, but **all the possible models**.
- The final prediction is a **weighted sum of all the models**, where weights correspond to the posterior probabilities.

Bayesian Estimation vs MLE

Maximum Likelihood Estimation

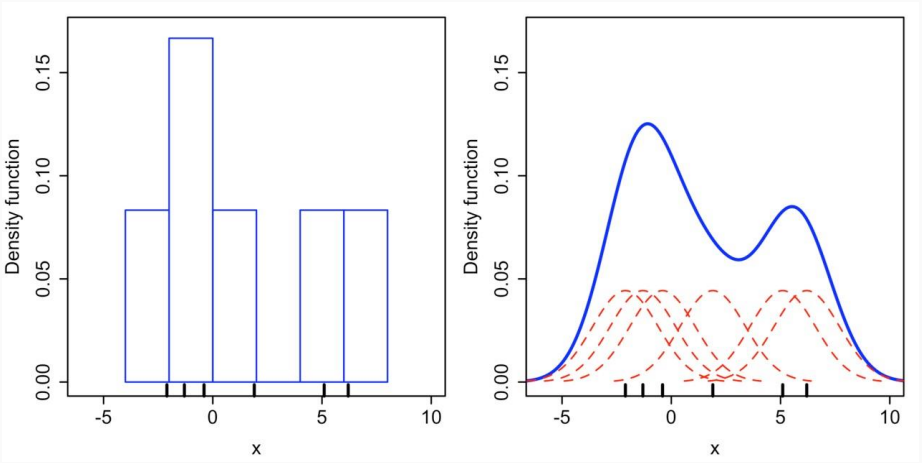
- Does not use prior knowledge of parameters
- Point estimation of the parameters obtained after solving an optimization problem.
- Easier and less computationally expensive.

Bayesian Estimation

- Exploits prior knowledge of parameters
- If the prior distribution is uniform (i.e, no information), it becomes MLE.
- No Point estimation of the parameters. We estimate a posterior probability.
- Computational expensive (including inference).

Parzen Windows

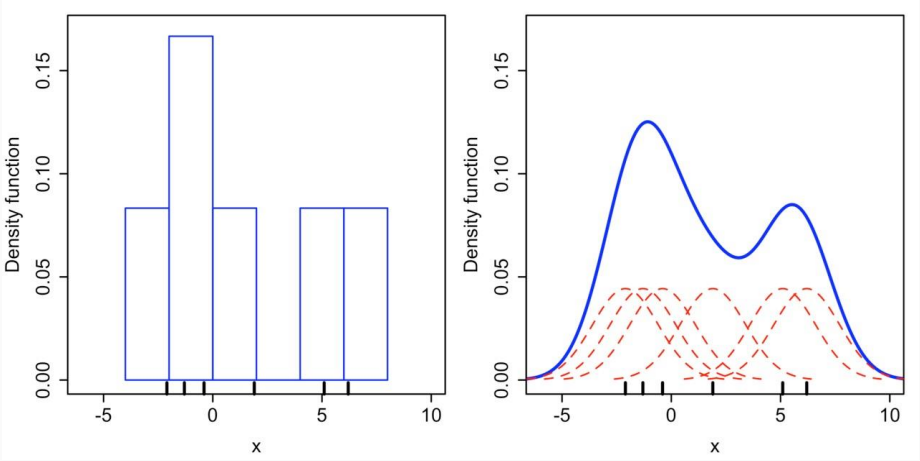
- Another way to perform density estimation is by using **Parzen windows** (kernel density).
- This is an example of a **non-parametric estimation** where we don't have a fixed number of parameters and we do not perform assumptions on the probability distribution (as in the parametric estimation techniques seen so far).
- **Idea**: Estimate p(**x**) by "smoothing" the data itself with some "kernel."



On top of each datapoint, we superimpose a kernel (e.g. Gaussian). The final density function is the sum of all kernels centered in each data point.

Parzen Windows

- **Idea**: Estimate by “smoothing” the data itself with some “kernel.”



- The density p(**x**) is computed as follows:

$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N} k(\mathbf{x} - \mathbf{x}_i)$$

- A valid “kernel function” must satisfy:

$$k(\mathbf{x}) \geq 0 \qquad \int_{\mathbb{R}^D} k(\mathbf{x})d\mathbf{x} = 1$$

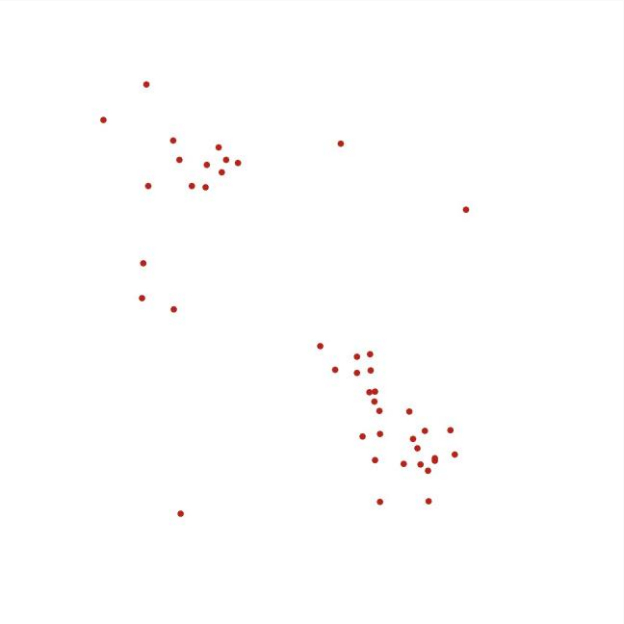
- The complexity of the density estimate can scale with the amount of data (more data => more terms in p(**x**)).

- The density p(**x**) is expressed directly in terms of **data** (an not in terms of parameters as for parametric estimation).

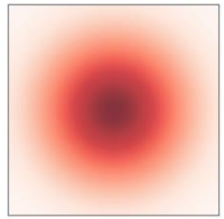
Parzen Windows

- We can see this estimation method as a convolution that “smooths” the data points with a kernel.

f(x,y)



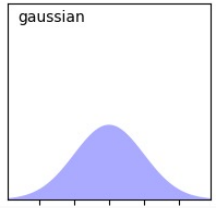
k(x,y)



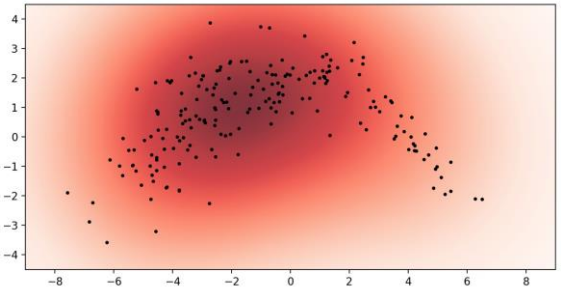
f(x,y) * k(x,y)

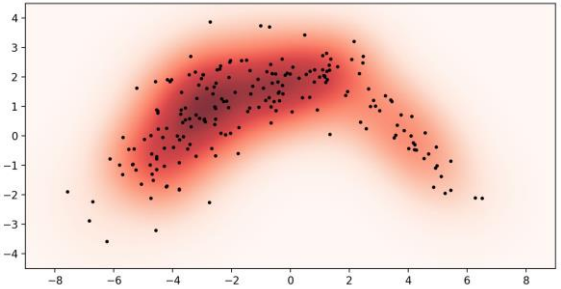
“f is zero everywhere except at the data points”

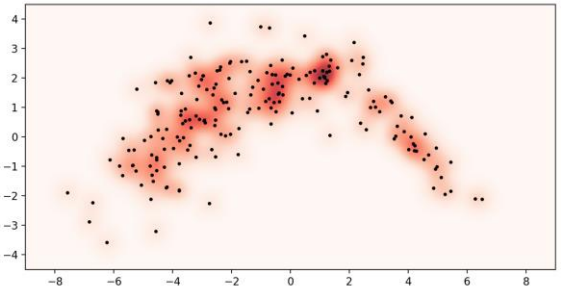
Parzen Windows



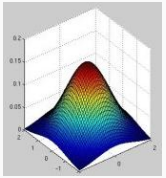
We have to properly set the kernel size to avoid overfitting/underfitting



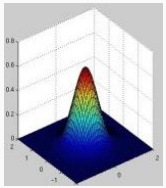




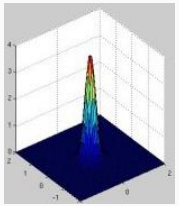
“Large” variance



“Medium” variance

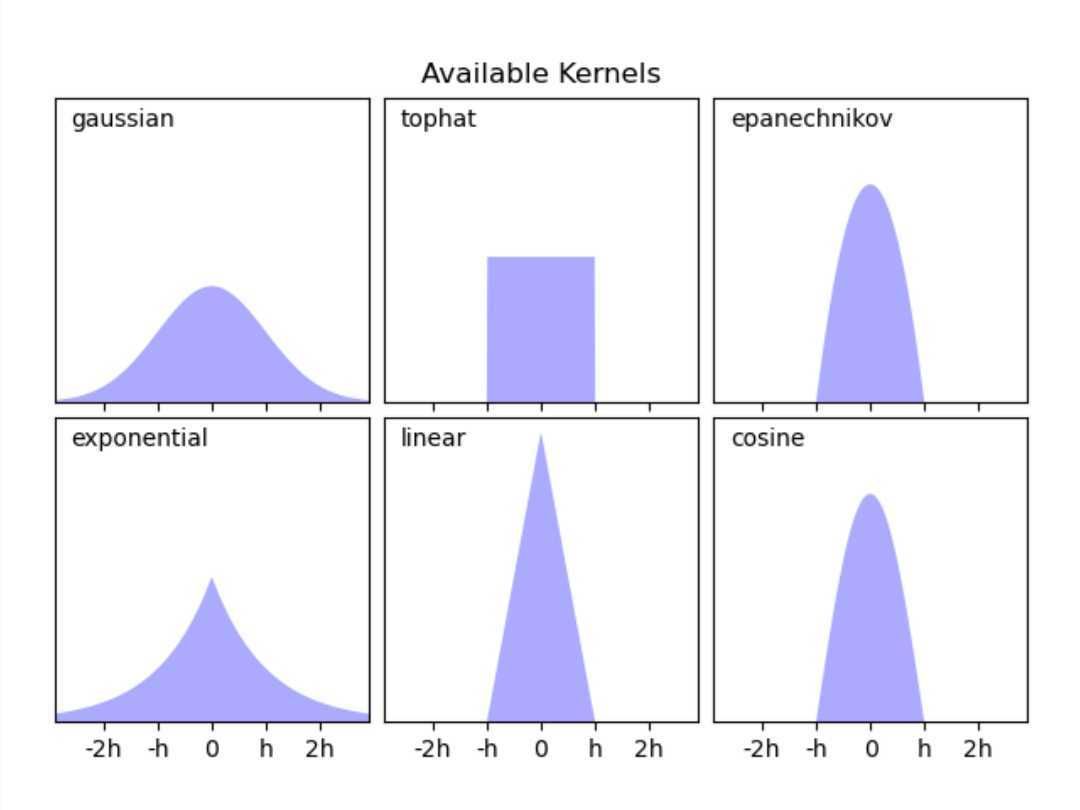


“Small” variance



Parzen Windows

- We can impose different types of kernels on top of the data points. Popular examples are:



Parametric vs Non-parametric Estimation

	Parametric	**Non-parametric**
Model Assumptions		
What do we have to store?		
Memory requirements		
Inference speed		
PDF estimation algorithm		
Hyperparameters		

Parametric vs Non-parametric Estimation

	Parametric	**Non-parametric**
Model Assumptions	Yes	No
What do we have to store?	Parameters	All data points
Memory requirements	Low	High (if we have many data points)
Inference speed	Fast	Slow (if we have many data points)
PDF estimation algorithm	Direct solution (if possible) Expectation Maximization	We just store the data points.
Hyperparameters	They might depend on the PDF type. For instance, for GMM the number of gaussian K is a hyperparameter.	Kernel type Window size

Scikit-learn

`sklearn.neighbors`.KernelDensity

class sklearn.neighbors.KernelDensity(*, *bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=0*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*) [source]

Kernel Density Estimation.

Read more in the User Guide.

Parameters:

bandwidth : *float, default=1.0*
The bandwidth of the kernel.

algorithm : *{'kd_tree', 'ball_tree', 'auto'}, default='auto'*
The tree algorithm to use.

kernel : *{'gaussian', 'tophat', 'epanechnikov', 'exponential', 'linear', 'cosine'}, default='gaussian'*
The kernel to use.

metric : *str, default='euclidean'*
The distance metric to use. Note that not all metrics are valid with all algorithms. Refer to the documentation of `BallTree` and `KDTree` for a description of available algorithms. Note that the normalization of the density output is correct only for the Euclidean distance metric. Default is 'euclidean'.

atol : *float, default=0*
The desired absolute tolerance of the result. A larger tolerance will generally lead to faster execution.

rtol : *float, default=0*
The desired relative tolerance of the result. A larger tolerance will generally lead to faster execution.

breadth_first : *bool, default=True*
If true (default), use a breadth-first approach to the problem. Otherwise use a depth-first approach.

leaf_size : *int, default=40*
Specify the leaf size of the underlying tree. See `BallTree` or `KDTree` for details.

metric_params : *dict, default=None*
Additional parameters to be passed to the tree for use with the metric. For more information, see the documentation of `BallTree` or `KDTree`.

Attributes:

n_features_in_ : *int*
Number of features seen during fit.

COMP 432 Machine Learning

Dimensionality Reduction

Computer Science & Software Engineering
Concordia University, Fall 2023

Dimensionality Reduction

- In some cases, it is convenient to **reduce** the dimensionality of the features that we feed into a machine learning algorithm.

- As we have seen, some machine learning algorithms are sensitive to the **curse of dimensionality** issues.
- We don't want to **waste computational resources** to process unnecessary features.
- Sometimes, we want to **visualize** the D-dimensional features in a 2D or 3D plot.

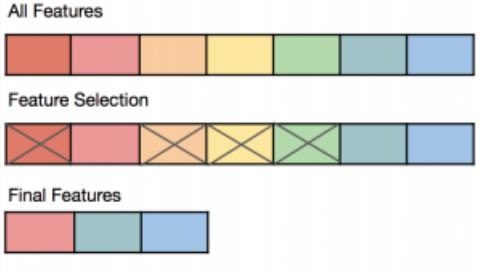
↓

This is often done also for the "internal" features (latent representations) learned by a neural network in the different layers.

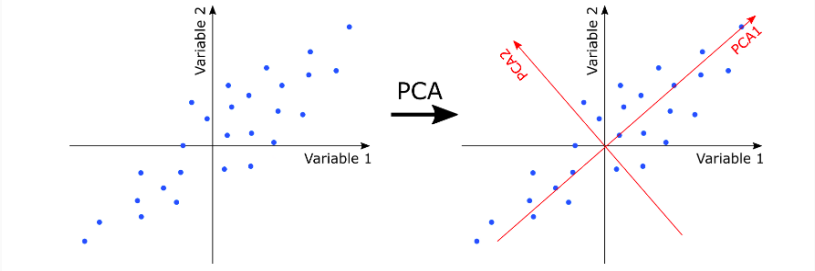
Dimensionality Reduction

- Different techniques can be applied:

- **Feature Selection**: we start from an D-dimensional feature vector and we remove the features that are not relevant for our machine learning problem.



- **Feature Transformation** (feature extraction): we transform the D-dimensional feature vector into a M-dimensional feature vector using a linear or non-linear transformation.



- Moreover, each technique can be **supervised** (if it exploits labels) or **unsupervised** (otherwise).

Feature Selection

Feature Selection

- **Feature Selection**: we want a subset of M "relevant" features from the original D features.
- The problem can be formalized as follows:

$$\mathbf{F}^* = \underset{\mathbf{F}' \subset \mathbf{F}}{\operatorname{argmin}} J(\mathbf{F}')$$

$\mathbf{F} = \{x_1, x_2, .., x_D\} \rightarrow$ Original set of features

$\mathbf{F}'$ is a subset of the features in $\mathbf{F}$

$\mathbf{F}^*$ is the optimal subset of features.

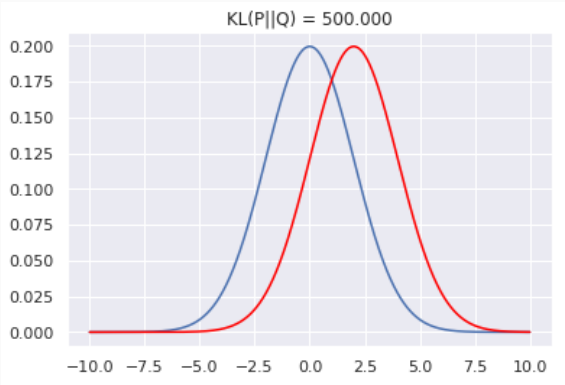
- As often done in this course, we can formalize the problem as an optimization problem where:

1. We have to define an **objective function** that quantifies "how good" is a feature set.
2. We need an optimization algorithm that attempts to find the solution that minimizes (or maximizes) the objective.

Feature Selection

Different **objective functions** can be used:

- One way is to use the **final performance** of the system (e.g, Classification Error Rate) measured on the **validation set** as an objective function for feature selection.
- For classification problems, an alternative is to estimate a **probability density function** for each class and select the set of features that maximizes the “distance” across these PDFs.
- The latter approaches rely on **divergence** metrics:



→ The idea is to measure the separability between two classes in a considered feature subspace by computing the **overlap degree** between the two related densities.

KL Divergence

- One popular measure of divergence is called **Kullback–Leibler** (KL) divergence:

$$\mathrm{KL}[p||q] = \int_{\mathbf{x}} p(\mathbf{x}) \ln \left(\frac{p(\mathbf{x})}{q(\mathbf{x})} \right) d\mathbf{x}$$

p(**x**) → Density function for class 1

q(**x**) → Density function for class 2

- For simple distributions (e.g. Gaussian) it can be computed analytically.
- If we have a limited number of data points in **X**, it is computed as:

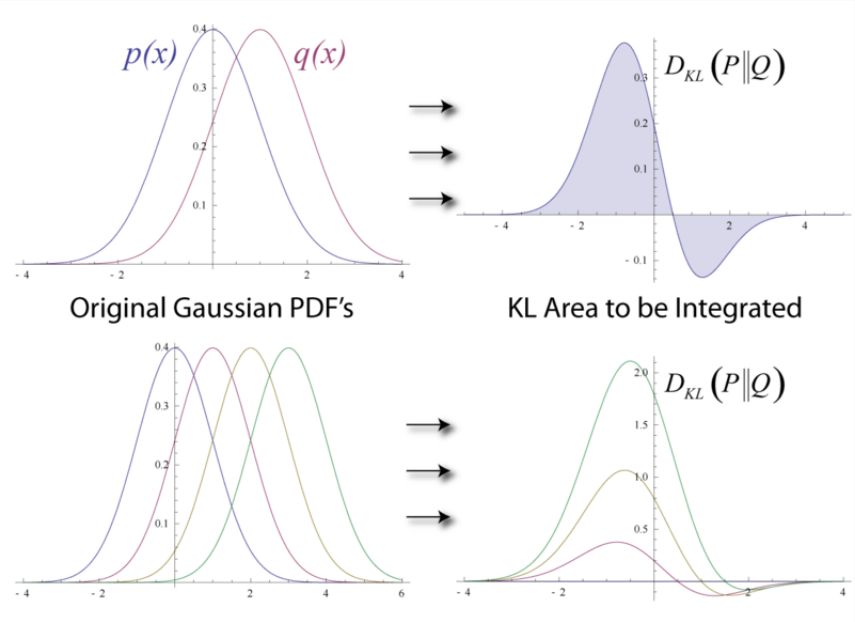
$$\mathrm{KL}[p||q] = \sum_{i=1}^{N} p(\mathbf{x}_i) \ln \left(\frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)} \right)$$

The KL divergence is the **expectation** of the **logarithmic difference** between two probabilities density functions.

KL Divergence

$$\mathrm{KL}[p||q] = \sum_{i=1}^{N} p(\mathbf{x}_i) \ln \left(\frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)} \right)$$

The KL divergence is the **expectation** of the **logarithmic difference** between two probabilities density functions.



- If the two distributions are **similar**, the **KL divergence** is **low**.
- If the two distributions are **dissimilar**, the **KL divergence** is **high**.
- The KL divergence ranges between 0 and ∞.
- It is not symmetric:

$$\mathrm{KL}[p||q] \neq \mathrm{KL}[q||p]$$

KL Divergence

- The KL divergence is useful in machine learning applications.
- In the context of feature selection, we are interested in the **KL divergence between two classes** in a considered feature subspace.
- We can thus write the KL in this way:

$$D_{ij} = \sum_{n=1}^{N} p(\mathbf{x}_n \mid y_i) \ln \left(\frac{p(\mathbf{x}_n \mid y_i)}{p(\mathbf{x}_n \mid y_j)} \right)$$

$$D_{ji} = \sum_{n=1}^{N} p(\mathbf{x}_n \mid y_j) \ln \left(\frac{p(\mathbf{x}_n \mid y_j)}{p(\mathbf{x}_n \mid y_i)} \right)$$

- The divergence used for feature selection is computed as follows:

$$d_{ij} = D_{ij} + D_{ji}$$

- The divergence between class i and class j used for feature selection is the sum of the two KL divergences D_{ij} and D_{ji}.
- This is a "*symmetrization*" of the Kullback-Leibler distance between the two distributions.

Feature Selection

- In the case of multiple classes, we can compute the distance of all the couples of classes:

$$d = \sum_{i=1}^{K} \sum_{j=1}^{K} d_{ij} P(y_i) P(y_j)$$

Each term d_{ij} is weighted by the product of the prior probabilities $p(y_i)p(y_j)$. This gives more "*importance*" to the classes with more samples.

This is the objective function J that we want to maximize.

- As an alternative, we can compute the minimum "distance":

$$d_{min} = \min_{i \neq j} d_{ij}$$

In the literature, many other measures have been proposed. Examples are:

- *Chernoff distance*
- *Bhattacharyya distance*

Feature Selection

- Once we have an objective function J, we need an optimization algorithm that finds the best subset of M features by minimizing (or maximizing) the objective:

$$\mathbf{F}^* = \underset{\mathbf{F}' \subset \mathbf{F}}{\operatorname{argmin}} J(\mathbf{F}')$$

$\mathbf{F} = \{x_1, x_2, .., x_D\} \rightarrow$ Original set of features

$\mathbf{F}'$ is a subset of the features in $\mathbf{F}$

$\mathbf{F}^*$ is the optimal subset of features.

- *Is there a way to make this objective convex?* NO
- *Is there a way to make this objective differentiable?* NO

Feature Selection

$$\mathbf{F}^* = \underset{\mathbf{F}' \subset \mathbf{F}}{\operatorname{argmin}}\, J(\mathbf{F}')$$

$\mathbf{F} = \{x_1, x_2, .., x_D\}$ → Original set of features

$\mathbf{F}'$ is a subset of the features in $\mathbf{F}$

$\mathbf{F}^*$ is the optimal subset of features.

- *Can we explore all the possible feature sets and take the best one?*

An **exhaustive search** requires testing the following number of combinations:

$$\frac{D!}{M!(D-M)!}$$

D=10, M=2 → 45 combinations feasible

D=10, M=4 → 210 combinations feasible

D=40, M=15 → $4.02 * 10^{10}$ impossible

Most of the times, we cannot explore all the combinations

Feature Selection

- We often accept the idea of finding a **suboptimal solution**.
- An example of a suboptimal optimization techniques is called **Sequential Forward Selection:**

This is a "*bottom-up*" approach that consists of progressively adding the **single feature** that optimizes the objective:

1. Create an empty feature set **F**′={ }
2. Select the best remaining feature that maximizes the objective:

$$x^* = \underset{x \notin \mathbf{F}'}{\operatorname{argmax}} J(\mathbf{F} + \{x\})$$

3. Update the feature set: **F**′= **F**′+ {x*}
4. Repeat M times

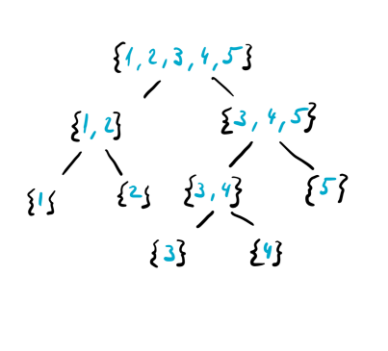
One can also start from the full set of features and remove the less important ones.

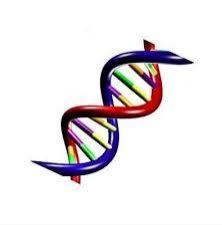
This is called **Sequential Backward Selection**.

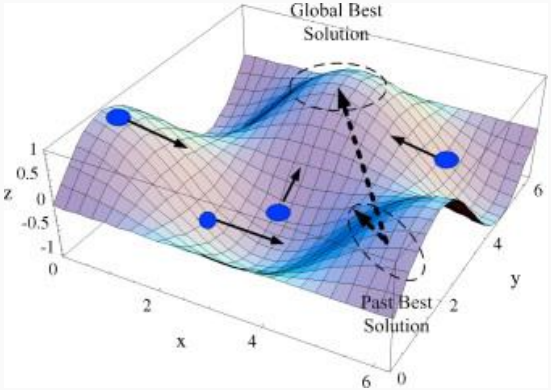
Feature Selection

- Alternatively, we can employ optimization techniques that attempt to find a global solution:

  - **Branch & Bound Search Strategy**
  - **Genetic Algorithms**
  - **Particle Swarm Optimization**
  - **Simulated Annealing**







Scikit-learn

1.13. Feature selection

The classes in the `sklearn.feature_selection` module can be used for feature selection/dimensionality reduction on sample sets, either to improve estimators' accuracy scores or to boost their performance on very high-dimensional datasets.

1.13.1. Removing features with low variance

`VarianceThreshold` is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e. features that have the same value in all samples.

As an example, suppose that we have a dataset with boolean features, and we want to remove all features that are either one or zero (on or off) in more than 80% of the samples. Boolean features are Bernoulli random variables, and the variance of such variables is given by

$$\mathrm{Var}[X] = p(1-p)$$

so we can select using the threshold `.8 * (1 - .8)`:

```
>>> from sklearn.feature_selection import VarianceThreshold
>>> X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
>>> sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
>>> sel.fit_transform(X)
array([[0, 1],
       [1, 0],
       [0, 0],
       [1, 1],
       [1, 0],
       [1, 1]])
```

As expected, `VarianceThreshold` has removed the first column, which has a probability $p = 5/6 > .8$ of containing a zero.

1.13.2. Univariate feature selection

Univariate feature selection works by selecting the best features based on univariate statistical tests. It can be seen as a preprocessing step to an estimator. Scikit-learn exposes feature selection routines as objects that implement the `transform` method:

- `SelectKBest` removes all but the k highest scoring features
- `SelectPercentile` removes all but a user-specified highest scoring percentage of features
- using common univariate statistical tests for each feature: false positive rate `SelectFpr`, false discovery rate `SelectFdr`, or family wise error `SelectFwe`.
- `GenericUnivariateSelect` allows to perform univariate feature selection with a configurable strategy. This allows to select the best univariate selection strategy with hyper-parameter search estimator.

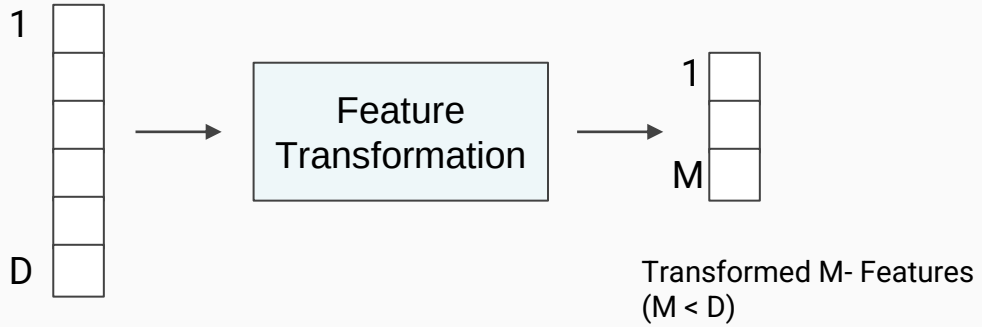
For instance, we can perform a χ^2 test to the samples to retrieve only the two best features as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import chi2
>>> X, y = load_iris(return_X_y=True)
>>> X.shape
(150, 4)
>>> X_new = SelectKBest(chi2, k=2).fit_transform(X, y)
>>> X_new.shape
```

Feature Transformation

Feature Transformation

- In **feature selection**, we select the subset of features that are more relevant to the task we want to solve.
- As an alternative, we can **transform** the D-dimensional feature vector into an M-dimensional feature vector using a linear or non-linear transformation.
- This approach is known as **feature transformation** (sometimes called feature extraction).

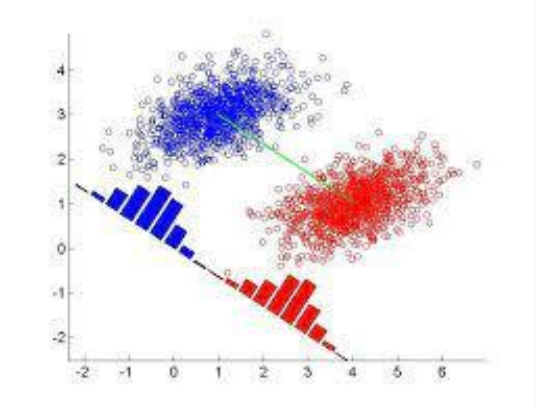


The transformation can be linear or non-linear.

It can use the labels (supervised transformation) or not (unsupervised transformation)

Linear Discriminant Analysis (LDA)

- One popular **supervised linear transformation** is called **Linear Discriminant Analysis** (LDA).



The main idea of LDA is to find a linear projection **W** such that the classes are **as separated as possible**.

When two classes are well separated?

If we assume the two classes to follow a Gaussian probability distribution, a “reasonable” distance measure is the following:

$$d = \frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2}$$

→ The classes are distant if their means are far and their variance is low.

This problem can find the **W** that maximizes this distance by computing **eigenvalues** and **eigenvectors**.

Principal Component Analysis (PCA)

- One popular **unsupervised linear transformation** is called **Principal Component Analysis** (PCA)

- **Goal**: project the D-dimensional data to an M-dimensional subspace, in a way that approximates the original data ("preserves most information"):

Reconstructed Error

$$\hat{\mathbf{W}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{W}\mathbf{z}_i||^2$$

Reconstructed Input

Main idea: Given a dataset $\mathbf{X} = [\mathbf{x}_1,...,\mathbf{x}_i,...,\mathbf{x}_N]^T$ with $\mathbf{x}_i$ in $\mathbb{R}^D$ find the features $\mathbf{Z} = [\mathbf{z}_1,...,\mathbf{z}_i,...,\mathbf{z}_N]^T$ with $\mathbf{z}_i$ in $\mathbb{R}^M$ and an **orthonormal basis** $\mathbf{W} \in \mathbb{R}^{DxM}$ such that the **reconstruction error** is **minimized**.

Principal Component Analysis (PCA)

Reconstructed Error

$$\hat{\mathbf{W}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{W}\mathbf{z}_i||^2$$

Reconstructed Input

In PCA, the matrix **W** is composed of **orthonormal** vectors:

$$\mathbf{w}_i^T \cdot \mathbf{w}_j^T = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{cases}$$

- Moreover, in an **orthonormal matrix**:

$$\mathbf{W}^T = \mathbf{W}^{-1}$$

- This means that once we found the best **W**, we can simply transform our features in this way:

$$\mathbf{x}_i = \mathbf{W}\mathbf{z}_i$$

$$\mathbf{W}^{-1}\mathbf{x}_i = \mathbf{W}^{-1}\mathbf{W}\mathbf{z}_i$$

$$\mathbf{z}_i = \mathbf{W}^T\mathbf{x}_i$$

Principal Component Analysis (PCA)

Reconstructed Error

$$\hat{\mathbf{W}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{W}\mathbf{z}_i||^2$$

Reconstructed Input

How can we solve this problem?

- *Is this function convex?* YES
- *Is this function differentiable?* YES

- Potentially, we can solve this problem with **gradient descent**.
- However, we normally do not use gradient descent because a **closed-form** solution exists.
- It can be shown that the best solution is based on finding the **eigenvectors** and **eigenvalues** of the **covariance matrix Σ**.

Principal Component Analysis (PCA)

- More precisely, the algorithm to compute the PCA is the following:

1. Compute the **mean vector**:

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$$

2. Compute the **Covariance Matrix**:

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

3. Compute the **Eigenvectors** and **Eigenvalues of Σ**

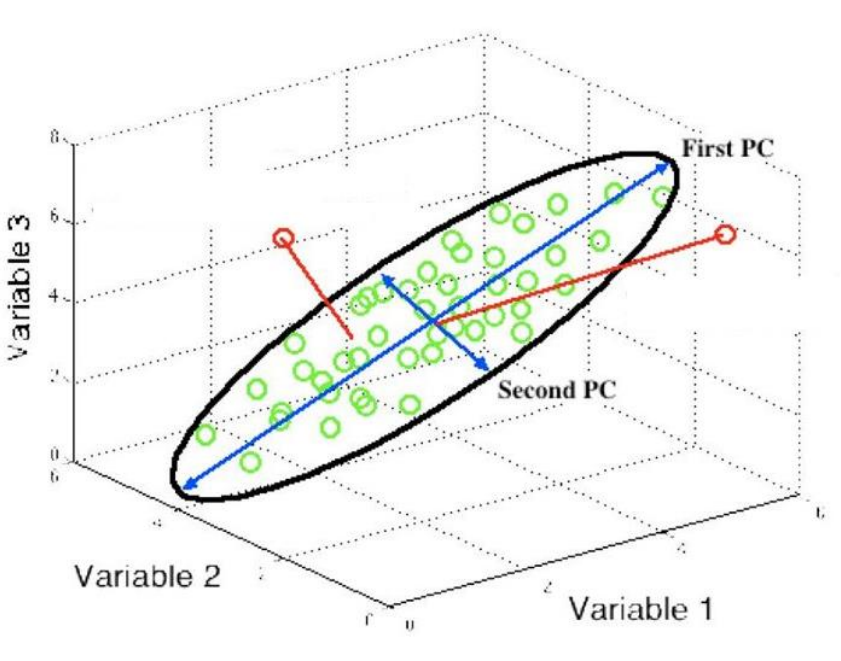
Eigendecomposition:

$$\boldsymbol{\Sigma} = \mathbf{V}\mathbf{L}\mathbf{V}^T$$

- Covariance matrix (squared)
- Matrix of eigenvectors (each column is an eigenvector)
- Diagonal matrix with eigenvalues (decreasing order)

4. Compose the matrix **W** by adding the M Eigenvectors corresponding to the M largest Eigenvalues.
5. Transform the input features: $\mathbf{z} = \mathbf{W}^T\mathbf{x}_i$

Principal Component Analysis (PCA)



Interpretation:

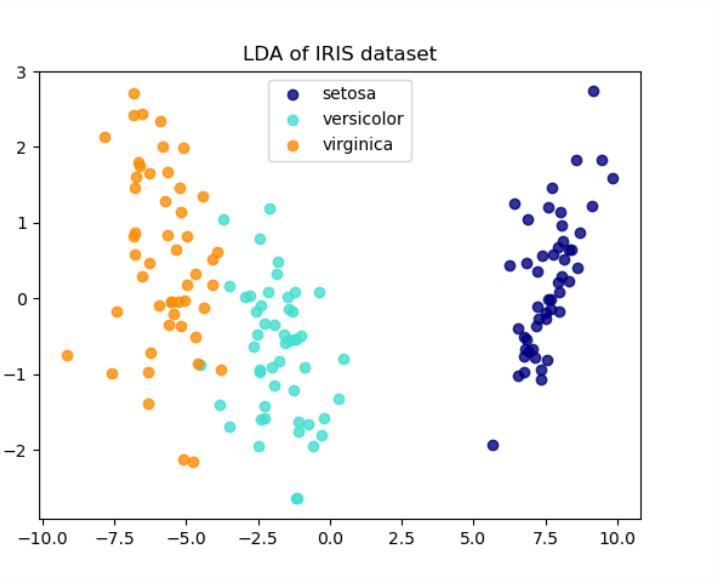
- The **first eigenvector** (i.e., the one with the highest eigenvalue) points in the direction of maximum variance.
- The **second eigenvector** points in the direction orthogonal to the first one where the variance is maximum.

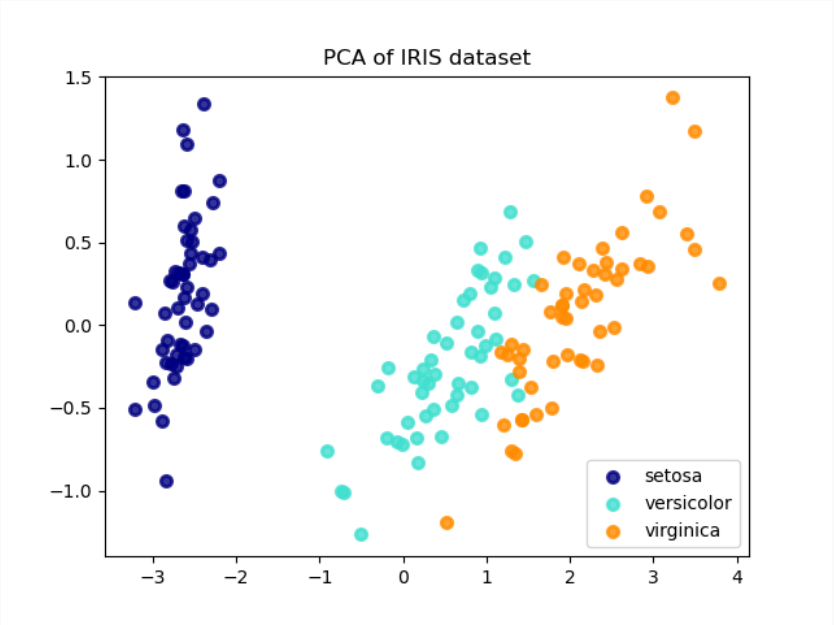
Note that there are many orthogonal vectors, we choose that with max variance.

- The **third eigenvector** points in the direction orthogonal to the second one where the variance is maximum.

...and so on...

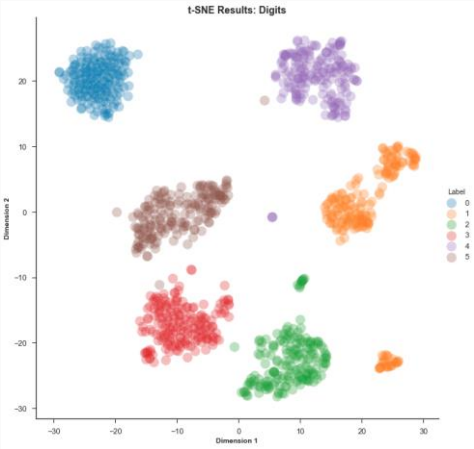
Principal Component Analysis (PCA)





t-SNE

- Feature transformation can be performed with **non-linear transformation** as well.
- A popular non-linear unsupervised transformation is called **t-distributed Stochastic Neighbor Embedding** (t-SNE).
  - t-SNE is mainly used to **visualize** high-dimensional data (such as internal features of neural networks).



Tutorial:

https://distill.pub/2016/misread-tsne/

Blog post:

https://towardsdatascience.com/an-introduction-to-t-sne-with-python-example-5a3a293108d1

Scikit-learn

sklearn.discriminant_analysis.LinearDiscriminantAnalysis

class sklearn.discriminant_analysis.LinearDiscriminantAnalysis(*solver='svd'*, *shrinkage=None*, *priors=None*, *n_components=None*, *store_covariance=False*, *tol=0.0001*, *covariance_estimator=None*) [source]

Linear Discriminant Analysis.

A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix.

The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions, using the `transform` method.

New in version 0.17: LinearDiscriminantAnalysis.

Read more in the User Guide.

sklearn.decomposition.PCA

class sklearn.decomposition. **PCA** (*n_components=None*, *copy=True*, *whiten=False*, *svd_solver='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*) [source]

Principal component analysis (PCA)

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

sklearn.manifold.TSNE

class sklearn.manifold. **TSNE** (*n_components=2*, *perplexity=30.0*, *early_exaggeration=12.0*, *learning_rate=200.0*, *n_iter=1000*, *n_iter_without_progress=300*, *min_grad_norm=1e-07*, *metric='euclidean'*, *init='random'*, *verbose=0*, *random_state=None*, *method='barnes_hut'*, *angle=0.5*) [source]

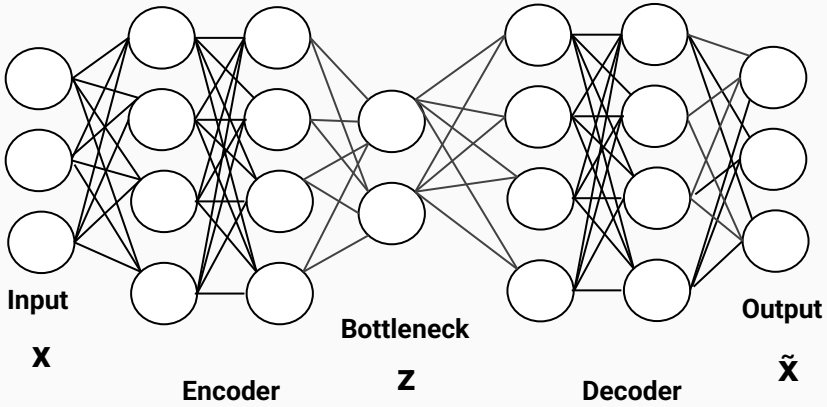
t-distributed Stochastic Neighbor Embedding.

t-SNE [1] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high. This will suppress some noise and speed up the computation of pairwise distances between samples. For more tips see Laurens van der Maaten's FAQ [2].

Autoencoders

- We can perform non-linear unsupervised feature transformation with **neural networks** as well.



- An autoencoder is a neural network composed of an **encoder** and a **decoder**.
- The **encoder** compresses the D features **x** into an M features **z**.
- The **decoder** reconstructs the original input x from the compressed features **z**.

- The **autoencoder** is trained to minimize the mean squared error (MSE) between the **original input** and the **reconstructed one**:

$$MSE = \frac{1}{N}\sum_{i=1}^{N} ||\mathbf{x}_i - \tilde{\mathbf{x}}_i||^2$$

Just like any other network, it is trained with gradient descent.

Could you guess why the bottleneck is needed?

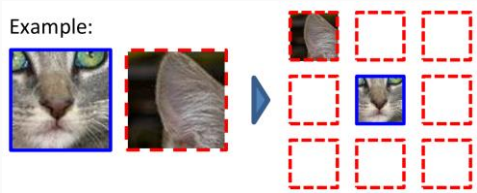
Self-Supervised Learning

- A modern way to perform non-linear unsupervised feature transformation is by using **self-supervised learning**.

Self-supervised Learning = the supervision is extracted from the signal itself.

- In general, this is performed by applying **known transforms** to the input data and using the resulting outcomes as targets.

Relative Positioning



(Doersch et. al. , ICCV 2015)

Colorization

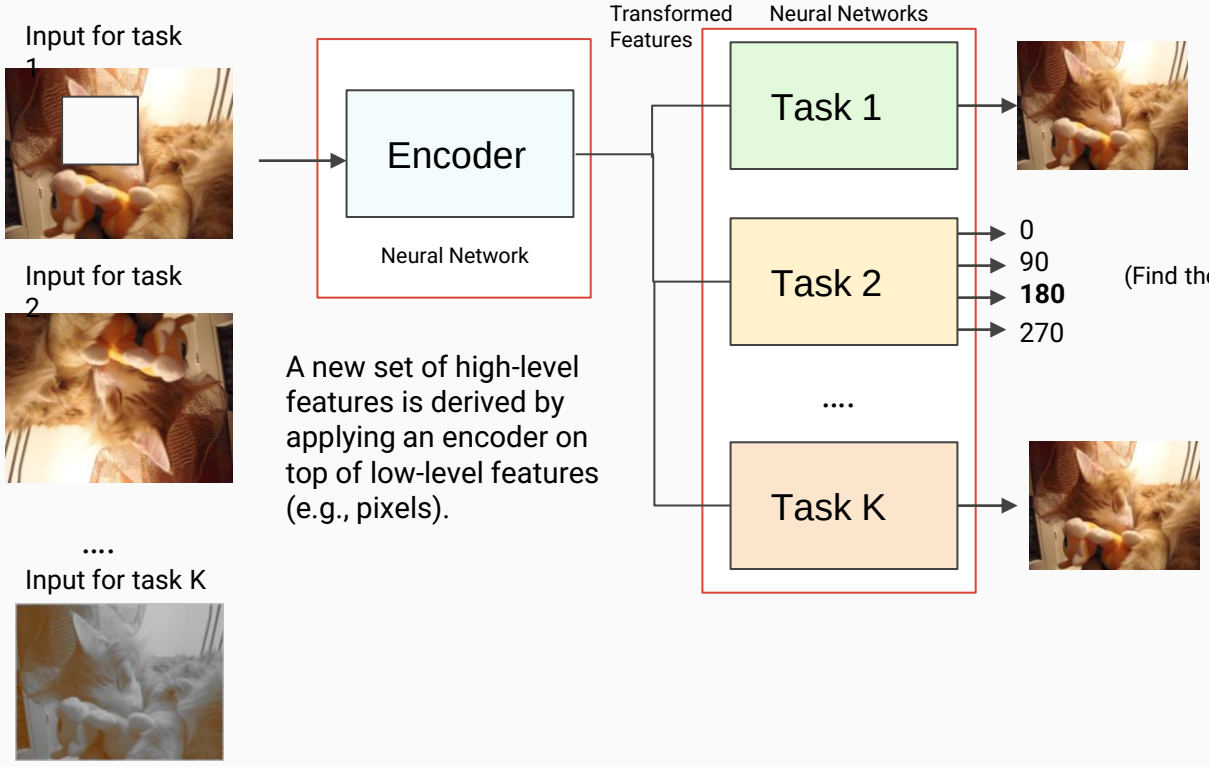
(Zhang et al., ECCV 2016)

Correct Rotation

(Gidaris et al., ICLR 2018)

Self-Supervised Learning

Self-supervised Learning = the supervision is extracted from the signal itself.



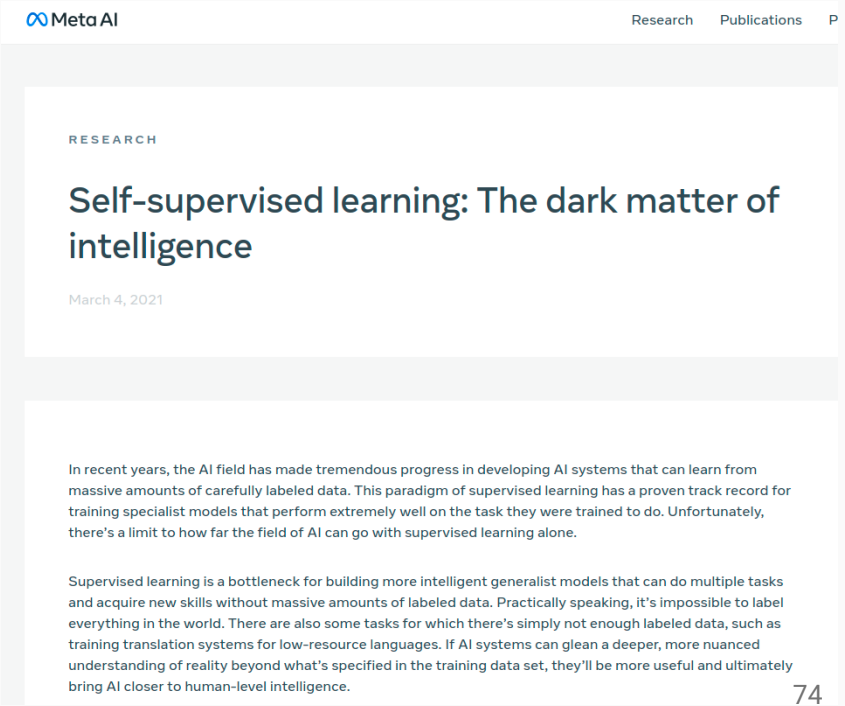
A new set of high-level features is derived by applying an encoder on top of low-level features (e.g., pixels).

The encoded features feed other neural networks that solve self-supervised tasks.

The encoder and neural networks solving the self-supervised tasks are **jointly trained**.

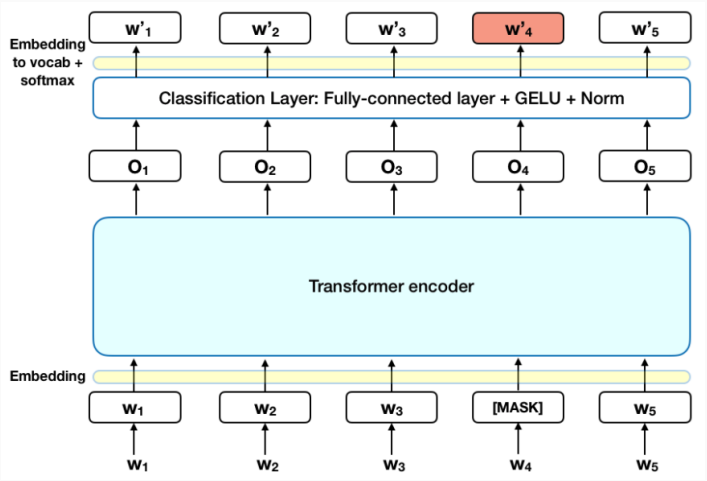
Self-Supervised Learning

- We normally train the model with tons of **unsupervised data**.
- After training, we can use the learned features as input to a **supervised** machine learning model.
- This approach has been proved **extremely effective**.
- Some top researchers in the field (e.g., Yann LeCun) defined self-supervised learning as the "**dark matter of intelligence**".
- State-of-the-art models for **computer vision**, **speech processing**, and **Natural Language Processing** are often based on a combination of self-supervised and supervised learning.

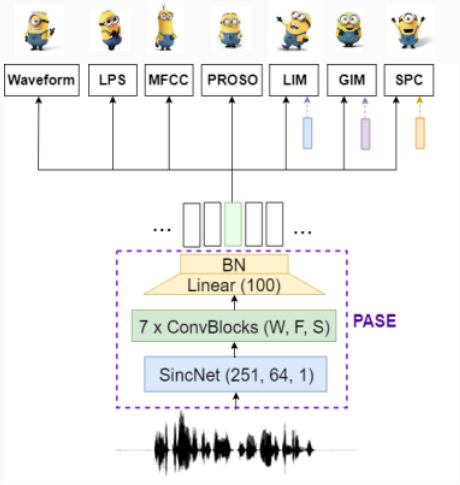


Self-Supervised Learning

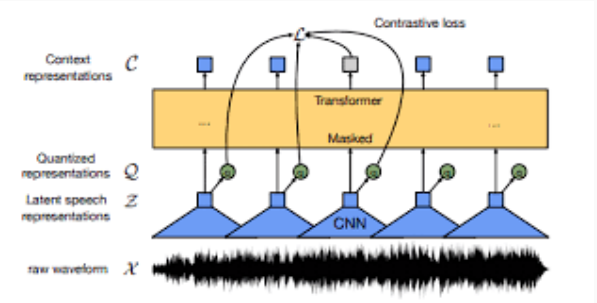
- The approach introduced in the previous slides for images can be extended to many signals, such as **text** and **audio**.



BERT (for computing word embeddings)



PASE (Problem Agnostic Speech Encoder)



Wav2vec 2.0 (for computing speech embeddings)

Final Remarks

What we learned

- In this course, we have seen several **machine learning algorithms**:
- Linear Models (Linear Regression, Logistic Regression, Multi-Class logistic regression)
- Neural Networks (MLPs, CNNs, RNNs)
- Support Vector Machines (Linear and Non-Linear)
- Decision Trees, Random Forest, Boosting
- K-nearest neighbor, Naive Bayes
- Clustering (K-means)
- PDF estimation (Gaussians, GMMs, Parzen Windows)
- Dimensionality Reduction (Feature Selection and Feature Transformation)

Which is the best machine learning algorithm?

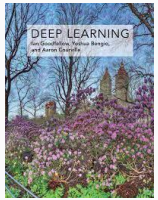
No Free Lunch Theorem

- The **“No free lunch”** theorem states that “*no machine learning is universally better than any other*”

- Every classification algorithm has the same error rate when classifying new data points if we average the performance over **all possible data-generation distributions**.

This result is valid only if we average the performance over all the data generation processes.

We can still find algorithms that work better than others for most real-world applications.



5.2.1 The No Free Lunch Theorem

Other Machine Learning Algorithms

- With this course, you got an overview of many popular machine learning algorithms.
- However, machine learning is much more than that.
- There are many other techniques that you can now study on your own:

Reinforcement Learning	*Transfer Learning*	*Continual Learning*
Federated Learning	*Attention Models*	*Transformers*
Hidden Markov Models	*Generative Adversarial Networks*	*Variational Autoencoders*
		.....

Deep Learning Related Courses @CSSE/Concordia

- CS Department @Concordia offers COMP499/691

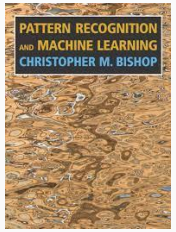
  “Deep Learning”
  “Conversational AI”
  “Deep Learning for Computational Pathology”

Final Suggestion

- Now, it is your turn!

Suggestion: Do your best to understand the material. Study it a lot and **in detail** for the final exam!

Additional Material

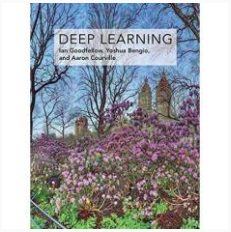


9.0.0 Mixture Models and EM

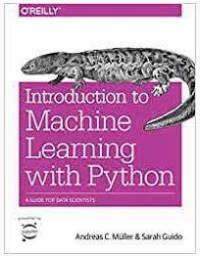
2.5.0 Nonparametric methods

2.5.1 Kernel density estimators

12.1 Principal Component Analysis.



5.2.1 The No Free Lunch Theorem



3.4 Dimensionality Reduction


That's all Folks!