TOPICS:
Dimension reduction methods
Non parametric techniques for density estimation

Slides compiled by : Sanghamitra De

Dimensionality Reduction

- In **machine learning** we are having too many factors on which the final classification is done.
- These factors are basically, known as variables.
- The higher the number of features, the harder it gets to visualize the training set and then work on it.
- Sometimes, most of these features are correlated, and hence redundant.
- This is where dimensionality reduction algorithms come into play.

Dimensionality Reduction

- Basically, dimension reduction refers to the process of converting a set of data.
- That data needs to be converted from having vast dimensions into data with lesser dimensions.
- Also, it needs to ensure that it conveys similar information concisely.
- We use these techniques to solve machine learning problems.
- And the problem here is to obtain better features for a classification or regression task.

Motivation

- When we deal with real problems and real data we often deal with high dimensional data that can go up to millions.
- In original high dimensional structure, data represents itself. Although, sometimes we need to reduce its dimensionality.
- We need to reduce the dimensionality that needs to associate with visualizations. Although, that is not always the case.

Components of Dimensionality Reduction

- There are two components of dimensionality reduction:
  - ✓ Feature selection

    In this, we need to find a subset of the original set of variables. Also, need a subset which we use to model the problem. It usually *involves three ways*:
    - Filter
    - Wrapper
    - Embedded
  - ✓ Feature Extraction

We use this, to reduces the data in a high dimensional space to a lower dimension space, i.e. a space with lesser no. of dimensions.

Dimensionality Reduction Methods

- The various methods used for dimensionality reduction include:
  - ✓ Principal Component Analysis (PCA)
  - ✓ Linear Discriminant Analysis (LDA)
  - ✓ Generalized Discriminant Analysis (GDA)
- Dimensionality reduction may be both linear or non-linear, depending upon the method used.

Importance of Dimensionality Reduction

- Dimension Reduction is important in machine learning predictive modelling
- The problem of unwanted increase in dimension is closely related to fixation of measuring/recording data at a far granular level then it was done in past.
- This is not a recent problem, but has started gaining more importance lately due to a surge in data.
- Lately, there has been a tremendous increase in the way sensors are being used in the industry.
- These sensors continuously record data and store it for analysis at a later point.
- In the way data gets captured, there can be a lot of redundancy.

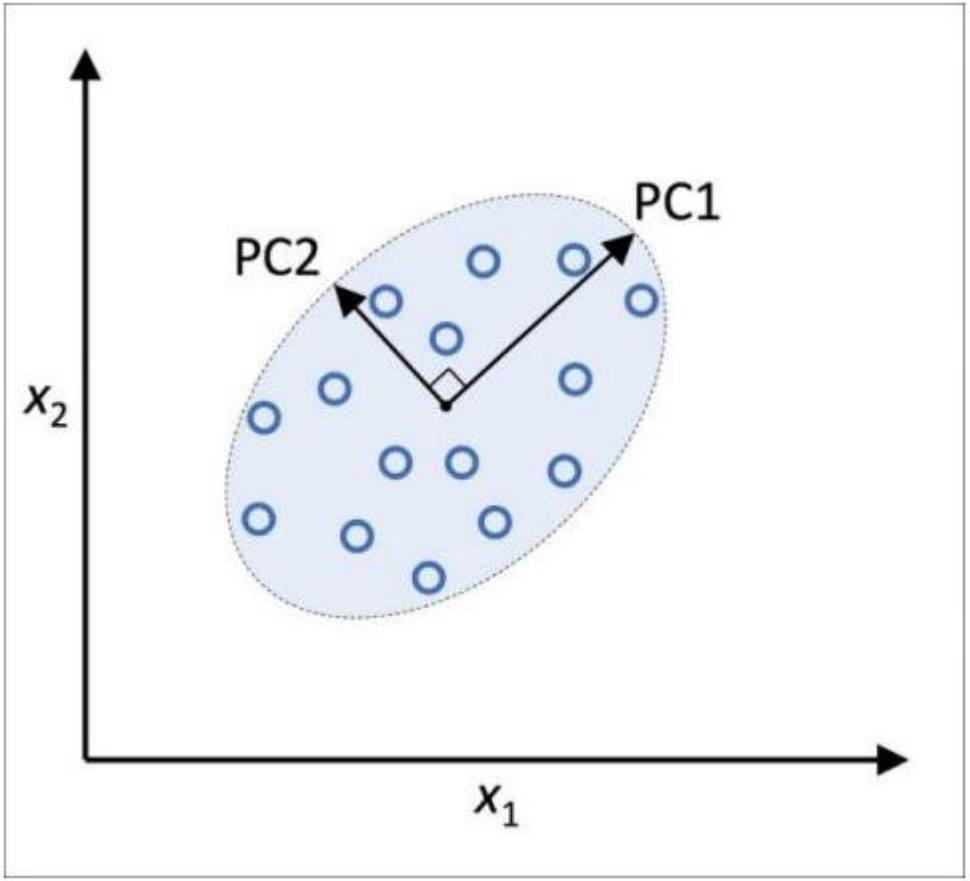
Principal Component Analysis (PCA)

- This method was introduced by **Karl Pearson**.
- It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.
- **Principal Component Analysis (PCA)** is an *unsupervised linear transformation technique* that is widely used across different fields, most prominently for feature extraction and dimensionality reduction.
- Other popular applications of PCA include exploratory data analyses and de-noising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics.

PCA

- PCA helps us to identify patterns in data based on the correlation between features.
- In a nutshell, *PCA aims to find the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one*.
- The orthogonal axes **(principal components)** of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other, as illustrated in the following figure:

PCA



PCA

- In the preceding figure, x_1 and x_2 are the original feature axes, and PC1 and PC2 are the principal components.
- If we use PCA for dimensionality reduction, we construct a d * k–dimensional **transformation matrix W** that allows us to map a sample vector x onto a new k–dimensional feature subspace that has fewer dimensions than the original d–dimensional feature space:

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d$$
$$\downarrow \boldsymbol{x}\boldsymbol{W}, \quad \boldsymbol{W} \in \mathbb{R}^{d \times k}$$
$$\boldsymbol{z} = [z_1, z_2, \ldots, z_k], \quad \boldsymbol{z} \in \mathbb{R}^k$$

PCA

- As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k \ll d$), the first principal component will have the largest possible variance, and all consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other principal components — even if the input features are correlated, the resulting principal components will be mutually orthogonal (uncorrelated).
- Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features prior to PCA if the features were measured on different scales and we want to assign equal importance to all features.

PCA

- Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:
  1) Standardize the d-dimensional dataset.
  2) Construct the covariance matrix.
  3) Decompose the covariance matrix into its eigenvectors and eigen values.
  4) Sort the eigen values by decreasing order to rank the corresponding eigenvectors.
  5) Select k eigenvectors which correspond to the k largest eigen values, where k is the dimensionality of the new feature subspace ($k \leq d$).
  6) Construct a projection matrix W from the "top" k eigenvectors.
  7) Transform the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

Data Pre-Processing

- The preprocessing to be performed is mean normalization or feature scaling. We are doing this for unlabeled data x(1) through x(m).
- For the mean normalization process, we first calculate the mean of each feature, then construct a new feature by subtracting the mean from each feature, which will make the mean of each feature just zero.
- Then, if there is a large difference between the features, then the feature needs to be scaled to a relative value range.

Data PreProcessing

Training set : $x^{(1)}$, $x^{(2)}$, ... , $x^{(m)}$

Preprocessing (feature scaling / mean normalization) :

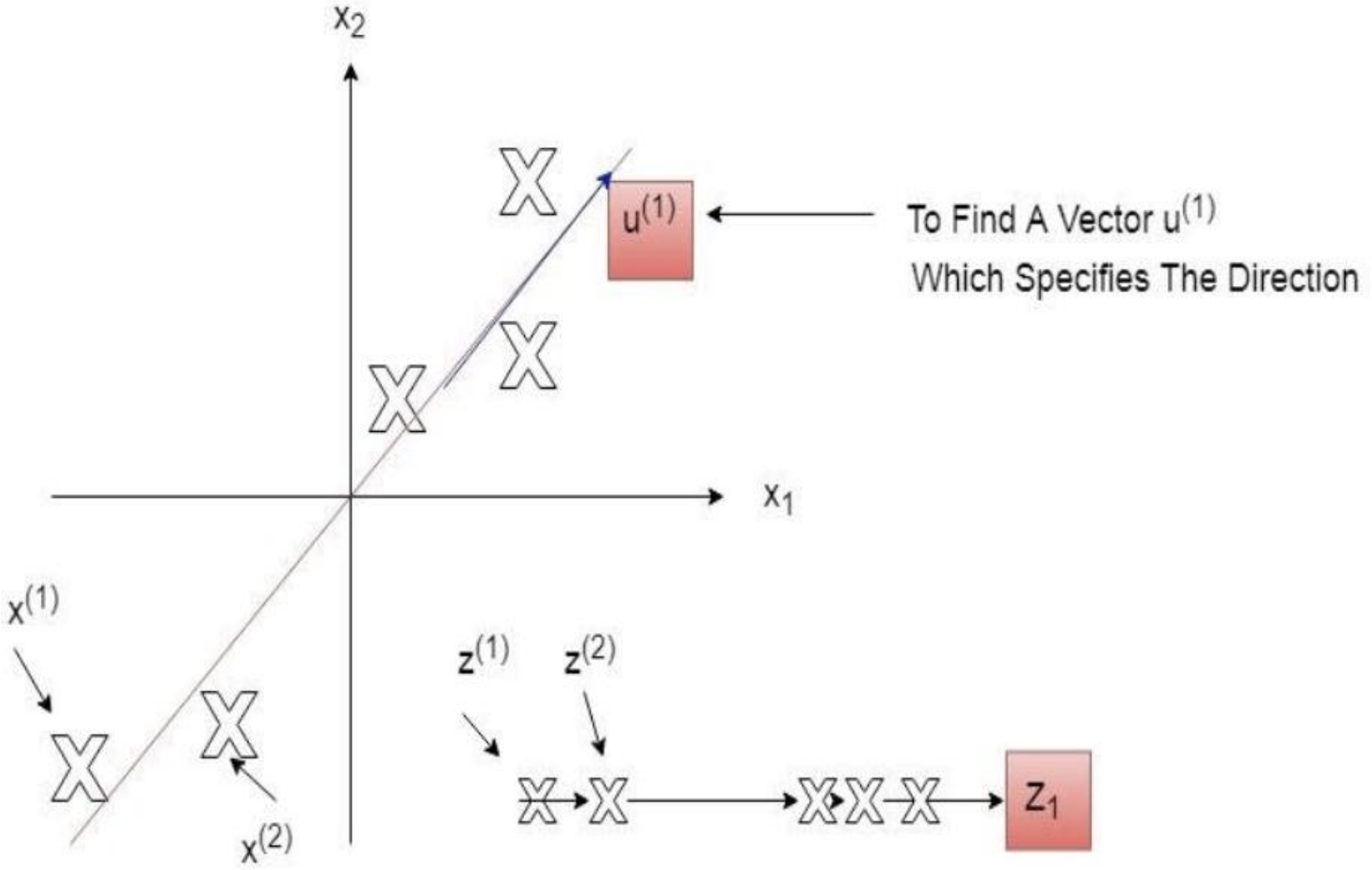
$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

Replace each $x_j^{(i)}$ with $x_j - \mu_j$

If different features on different scales (e.g. x_1 size of house, x_2 = number of bedrooms) scale features to have comparable range values

PCA

- PCA will find a low dimensional subspace on to project the data to minimize the square of the projection error and to minimize the square of the distance between each point and it’s projection point.

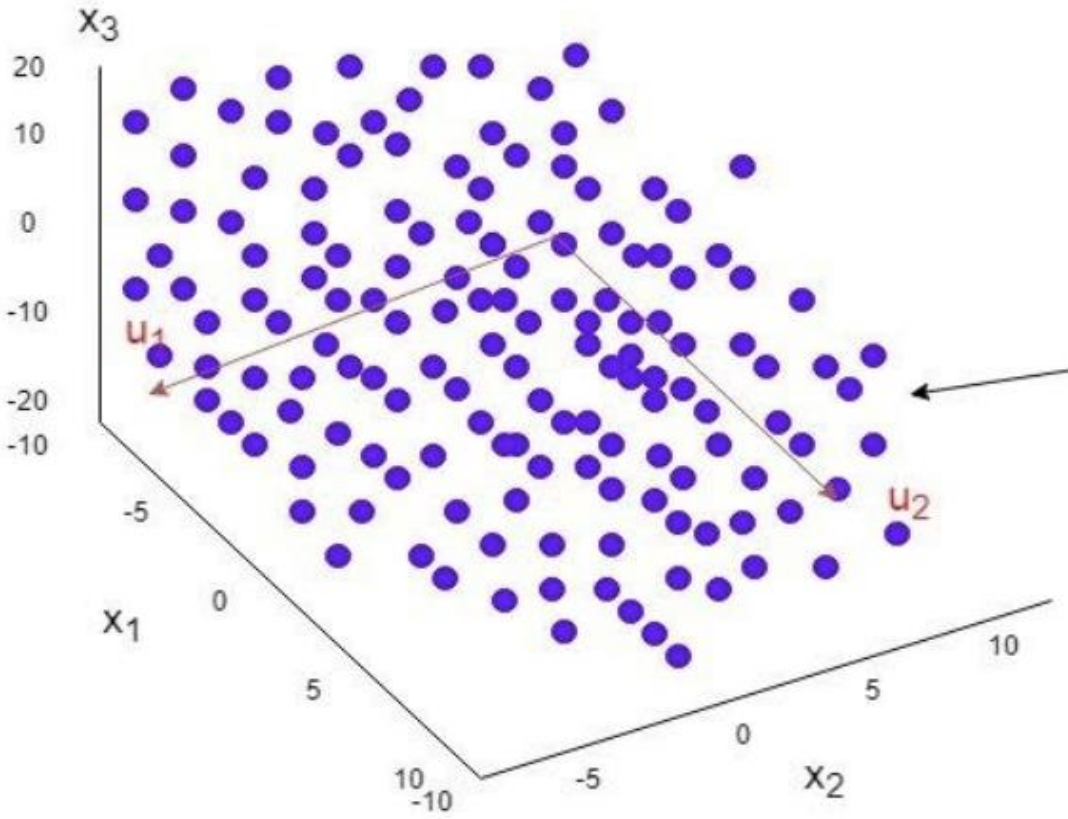

x_2
$u^{(1)}$
To Find A Vector $u^{(1)}$
Which Specifies The Direction
x_1
$x^{(1)}$
$x^{(2)}$
$z^{(1)}$
$z^{(2)}$
z_1

PCA

- In the Figure 1 it is given the examples x(i), which are in R2 and what we're going to do is to find a set of numbers Z(i) in R with which we wish to represent our data.
- So, that's what a reduction from 2d to 1d means and so specifically by projecting on to the red line we need only one number to specify the position of the points on the line.
- So, we will call that number Z1. There on the line it will represent all the Xs on the Z1 with denoting X with Z(1) and so on.
- So, what PCA has to do is we need to come up with a way to compute two things.
- One is to compute the vectors u1 and then the bottom picture to compute u1 and u2.
- And the other is how to compute the number Z. So, this means we are reducing the data from 2D to 1D.

Reduce Data From 2D To 1D

$X^{(i)} \leftarrow R^2 \rightarrow Z^{(i)} \leftarrow R$ ← Real Number



Uses two vectors U_1 and U_2

Reduce Data From 2D to 3D

$X^{(i)} \leftarrow R^3 \rightarrow Z^{(i)} \leftarrow R^2$

$$z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

PCA

- In the figure 2, we would be reducing data from 3 dimensional as in R^3 to Zi which is now two dimensional. So, these z vectors would now be two dimensional.
- So, it would be z1 and z2, and we need to give away to compute these new representations, the z1 and z2 of the data as well
- But how do you calculate all of these quantities?
- This turns out to be a mathematical derivation seen what is the right value for U1, U2, Z1, Z2.

Procedure

- To implement all of these things, the vectors U1, U2 and the vector Z, we use the below procedure.
- Here's the procedure. Let's assume that we want to reduce the data of a dimension n to a dimension k.
- What we're going to do is first compute something called the covariance matrix, and the covariance matrix is commonly denoted by the Greek alphabet which is the capital Greek alphabet sigma.

$$\sum = \frac{1}{m}\sum_{i=1}^{n}\left(x^{(i)}\right)\left(x^{(i)}\right)^{T}$$

Procedure

- Next is how do you compute this matrix.
- So, what we need to do is compute the eigenvectors of the matrix sigma.

NOTE: By using octave, this is done by using the command u,s,v equals s v d of sigma. S V D stands for singular value decomposition.

$$[U, S, V] = \text{svd}(\text{Sigma})$$

Procedure

- So, this covariance matrix sigma will be an n by n matrix.
- One way to see this is to look at the definition of the covariance matrix where x(i) is n by 1 vector and the other x(i) is 1 by N and the product of such two things is going to be N by N matrix.
- And what the SVD outputs is the three matrices U, S, V.
- U matrix is the thing you really need out of the SVD.
- Here the U matrix will also be a N x N matrix.
- And if we look at the columns of the U matrix it turns out the columns of the U matrix that will be exactly containing the vectors u1, u2 and so on.

Procedure

- If you want to reduce the data from n dimensions down to k dimensions, you need to take the first k vectors from u1 to uk that give us the K path to which we want to project the data.

Principal Component Analysis Algorithm

Reduce data from n - dimensions to k - dimensions
Compute "convariance matrix" :

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n} \underbrace{\left(x^{(i)}\right)}_{n*1} \underbrace{\left(x^{(i)}\right)^{T}}_{1*n}$$

Combining $(x^{(i)})$ $(x^{(i)})^{T}$ → n *n matrix

Compute "eigenvectors" of matrix Σ :

```
[U,S,V] = svd (Sigma);
```

n * n matrix

Single Value Decomposition

$U = [\, U^{(1)} \; U^{(2)} \; U^{(3)} \; \ldots \; U^{(m)} \,]$

K

$U \leftarrow R^{n*n}$

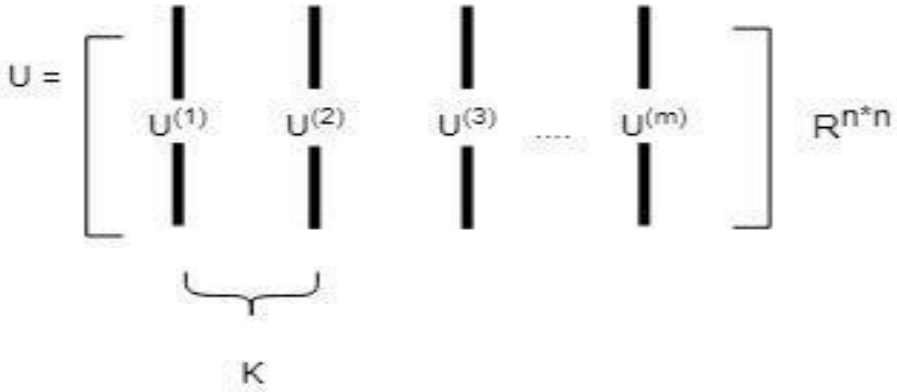
$U^{1}, \ldots, U^{(k)}$

Procedure

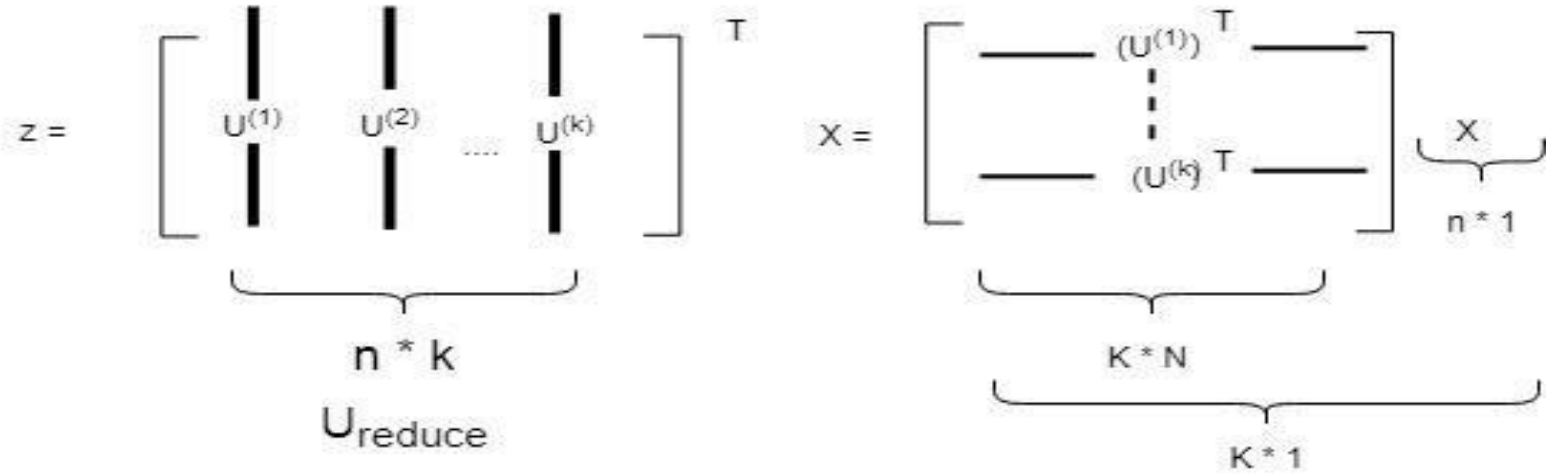
- Here U is a matrix of n*n , we get the first k column of U and get a matrix of n*k, which we call ***U reduce matrix***.
- Thereafter we will use this matrix to reduce data.
- U reduce is an n * k dimensional matrix, and x(i) is an n*1 dimensional matrix, so z is a k*1 dimensional matrix, which is the dimensionally reduced data we want to get.

Principal Component Analysis Algorithm

From [U,S,V] = svd (Sigma), we get :



$$x \in R^n \longrightarrow z \in R^k$$



Principal Component Analysis (PCA) Algorithm Summary

After mean normalization (ensure every feature has zero mean) and optionally feature scaling :

Sigma : $\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T$

$X = \begin{bmatrix} — (X^{(1)})^T — \\ \vdots \\ — (X^{(m)})^T — \end{bmatrix}$

[U,S,V] = svd (Sigma);

Sigma = (1/m) *X' * X ;

Ureduce = U(:,1:k) ;

z= Ureduce'*X ;

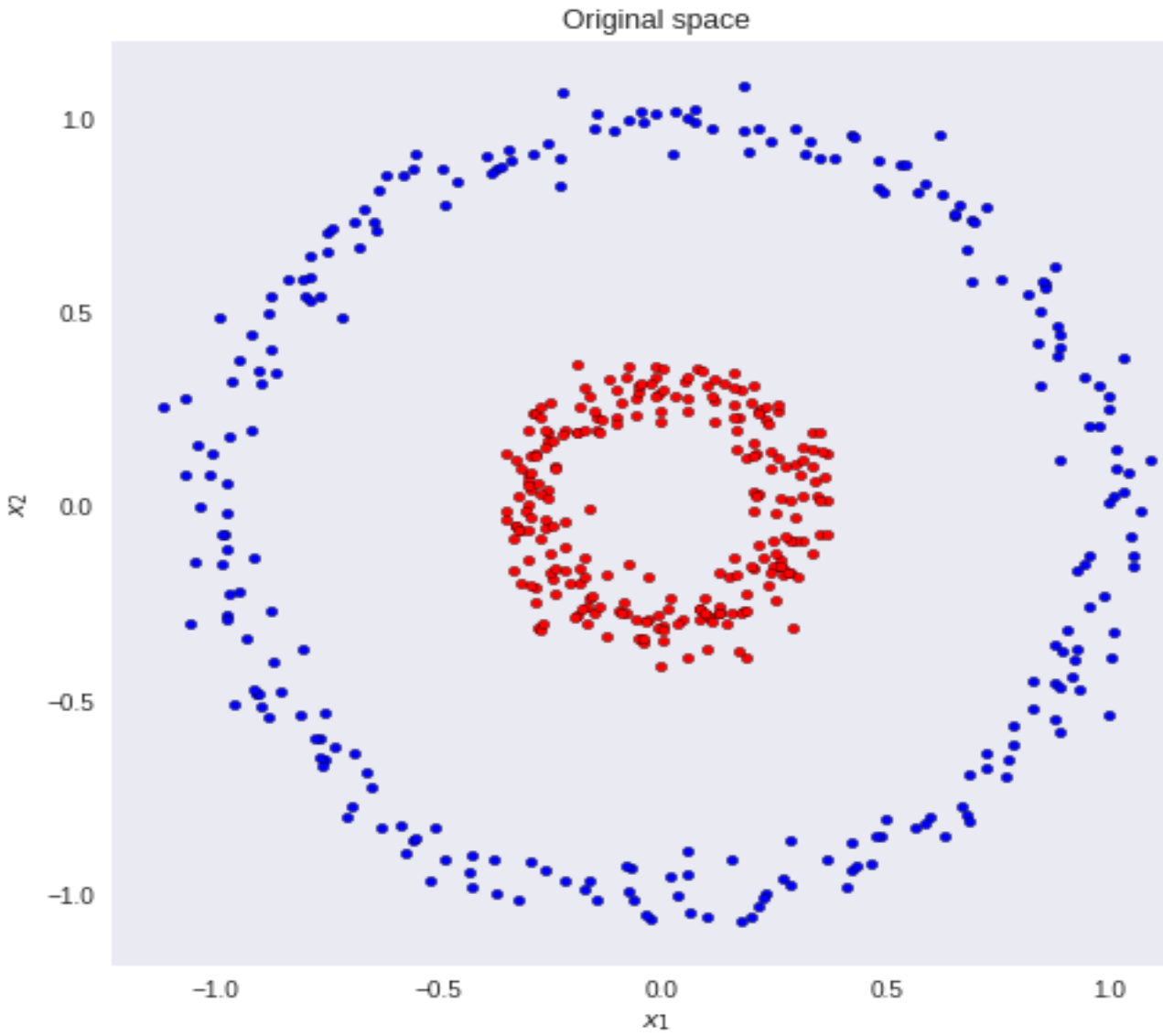
$X \in R^n$ $X_0 = 1$

In PCA, we obtain $z \in \mathbb{R}^k$ from $x \in \mathbb{R}^n$ as follows:

$$z = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \dots & u^{(k)} \\ | & | & & | \end{bmatrix}^T x = \begin{bmatrix} --- & (u^{(1)})^T & --- \\ --- & (u^{(2)})^T & --- \\ & \vdots & \\ --- & (u^{(k)})^T & --- \end{bmatrix} x$$

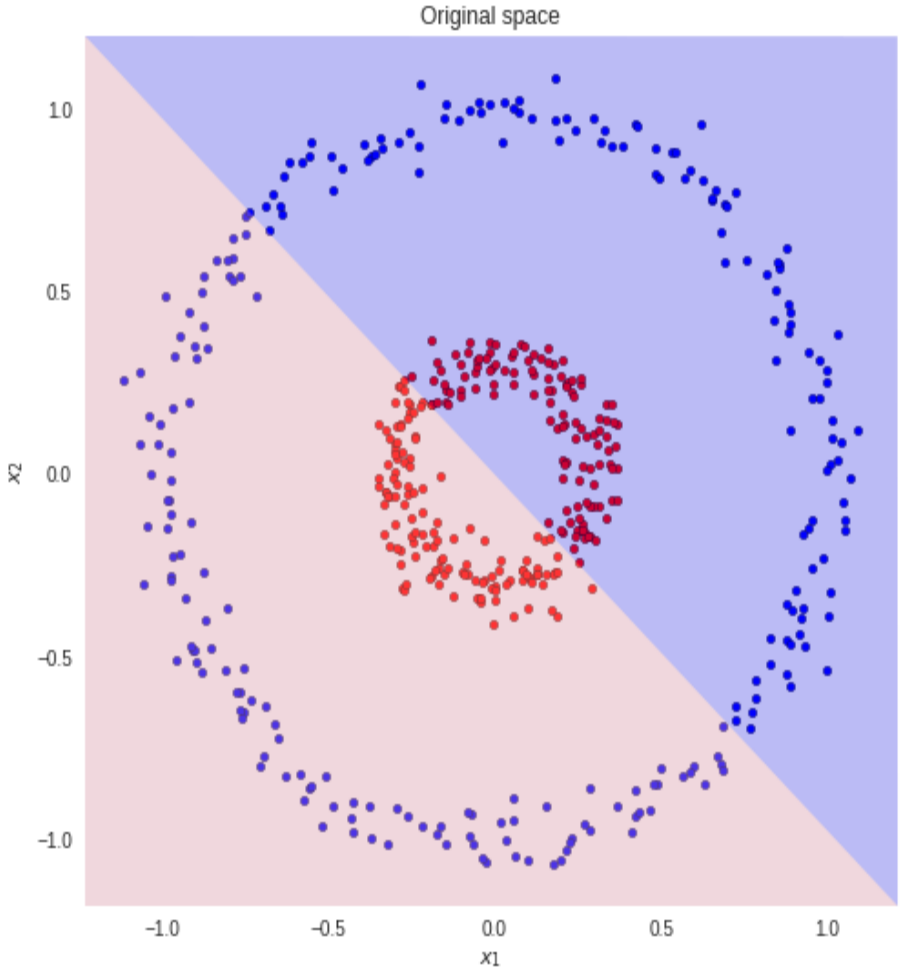
Fisher's Linear Discriminant - Introduction

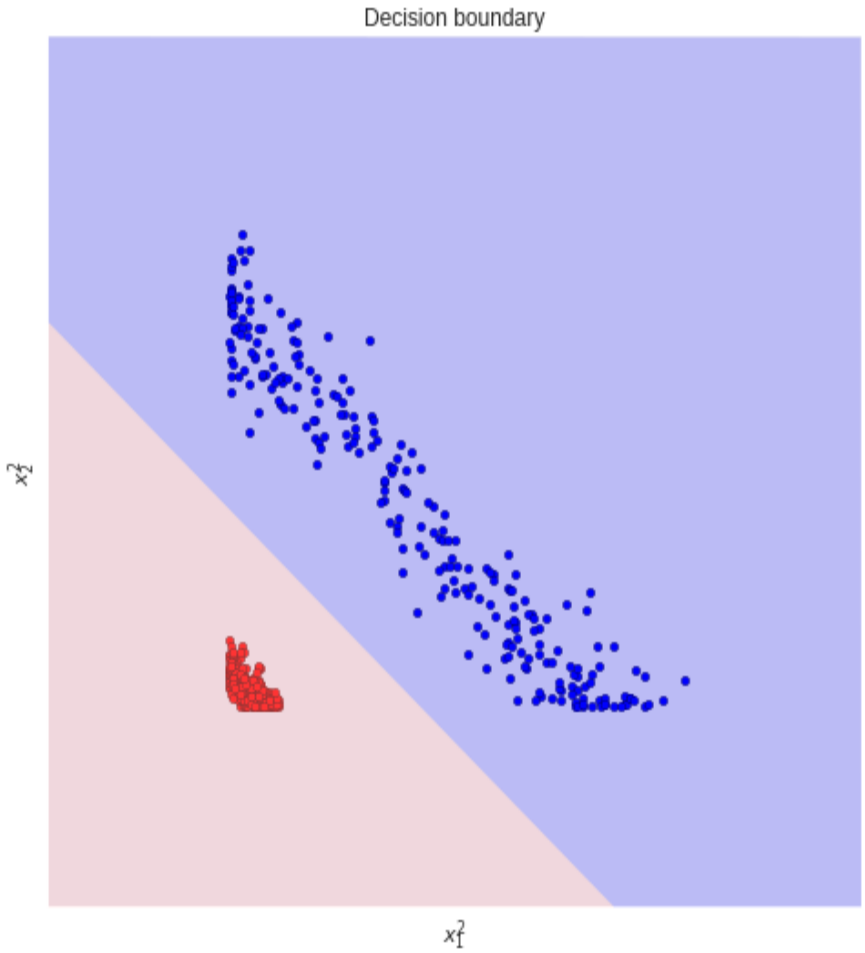
- To deal with classification problems with 2 or more classes, most Machine Learning (ML) algorithms work the same way.
- Usually, they apply some kind of transformation to the input data with the effect of reducing the original input dimensions to a new (smaller) one.
- The goal is to project the data to a new space. Then, once projected, they try to classify the data points by finding a linear separation.
- For problems with small input dimensions, the task is somewhat easier. ***Take the following dataset as an example.***


Original space
1.0
0.5
0.0
−0.5
−1.0
x2
−1.0
−0.5
0.0
0.5
1.0
x1

FLD - Introduction

- Suppose we want to classify the red and blue circles correctly.
- It is clear that with a simple linear model we will not get a good result.
- There is no linear combination of the inputs and weights that maps the inputs to their correct classes.
- But what if we could transform the data so that we could draw a line that separates the 2 classes?
- ***That is what happens if we square the two input feature-vectors. Now, a linear model will easily classify the blue and red points.***


Original space
1.0
0.5
0.0
−0.5
−1.0
x_2
−1.0
−0.5
0.0
0.5
1.0
x_1


Decision boundary
x_2^2
x_1^2

FLD - Introduction

- However, sometimes we do not know which kind of transformation we should use.
- Actually, to find the best representation is not a trivial problem.
- There are many transformations we could apply to our data.
- Likewise, each one of them could result in a different classifier (in terms of performance).
- One solution to this problem is to learn the right transformation.
- This is known as ***representation learning.***

FLD - Introduction

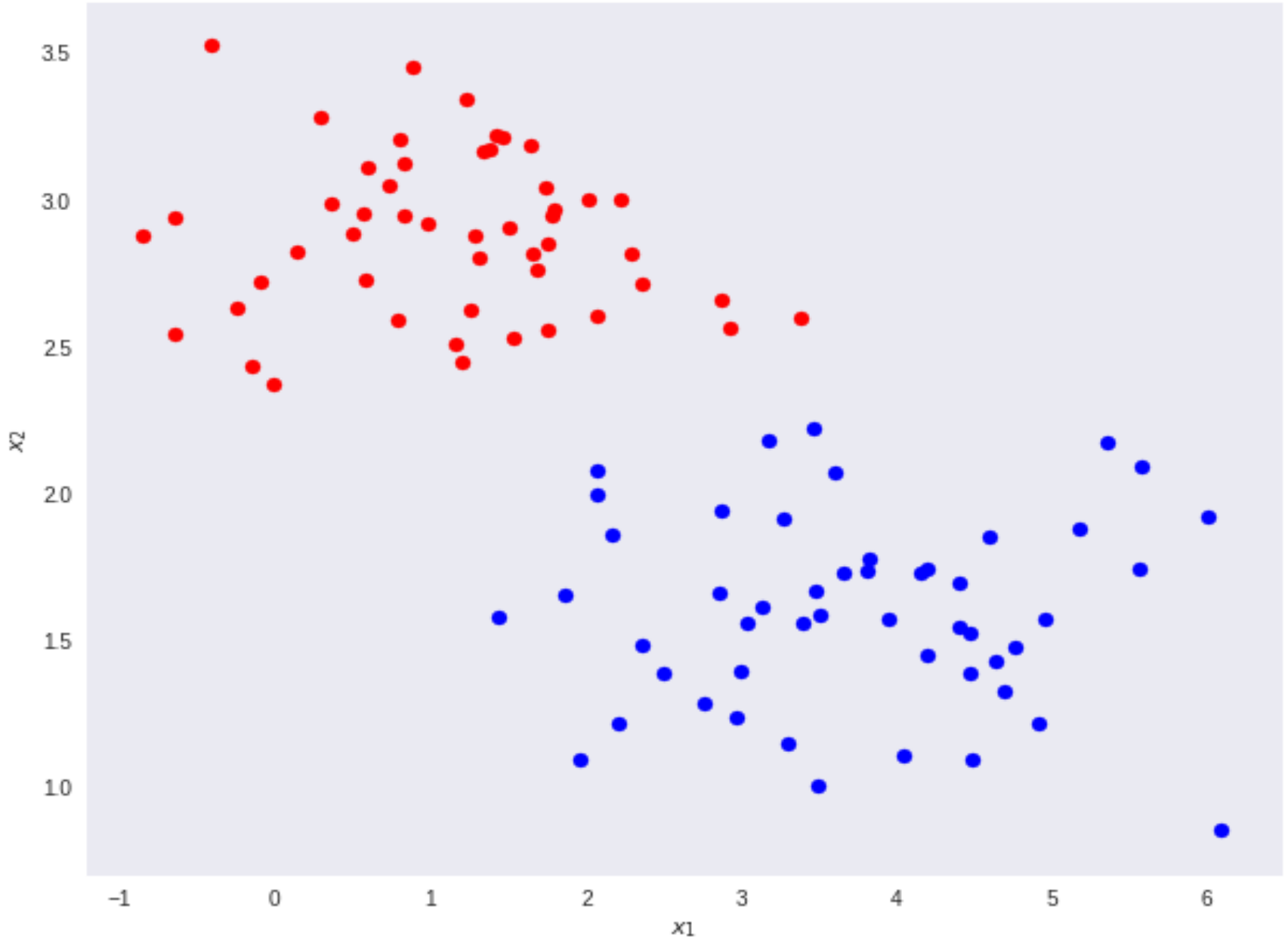
- Here, we do not need to "guess" what kind of transformation would result in the best representation of the data. The algorithm will figure it out.
- However, keep in mind that regardless of representation learning or hand-crafted features, the pattern is the same.
- We need to change the data somehow so that it can be easily separable.
- Let's take some steps back and consider a simpler problem.
- In this piece, we are going to explore how Fisher's Linear Discriminant (FLD) manages to classify multi-dimensional data.

Fisher's Linear Discriminant

- We can view linear classification models in terms of dimensionality reduction.
- To begin, consider the case of a two-class classification problem (K=2). Blue and red points in R^2.
- In general, we can take any D-dimensional input vector and project it down to D'-dimensions.
- Here, D represents the original input dimensions while D' is the projected space dimensions. Throughout this article, consider D' less than D.
- In the case of projecting to one dimension (the number line), i.e. D'=1, we can pick a threshold t to separate the classes in the new space.
- Given an input vector x:
  - if the predicted value y >= t then, x belongs to class C1 (class 1) - where $y = W^t x$
  - otherwise, it is classified as C2 (class 2).

FLD

- Take the dataset below as a toy example.
- We want to reduce the original data dimensions from D=2 to D'=1.
- In other words, we want a transformation T that maps vectors in 2D to 1D=>$T(v) = \mathbb{R}^2 \rightarrow \mathbb{R}^1$.
- First, let's compute the mean vectors m1 and m2 for the two classes.


35
30
25
20
15
10
x_2
−1
0
1
2
3
4
5
6
x_1

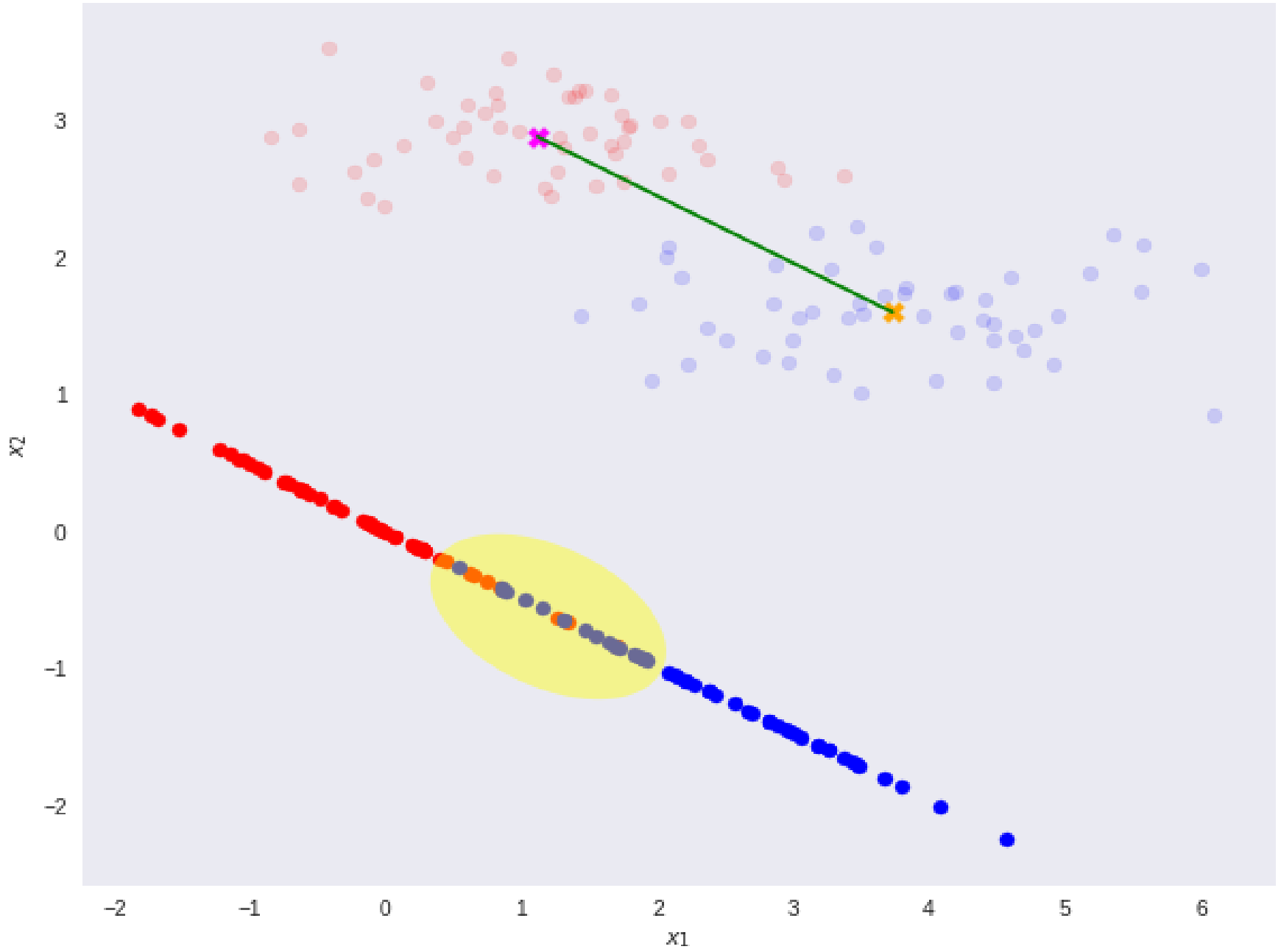
$$m_1 = \frac{1}{N_1} \sum_{n \in C1} x_n \qquad m_2 = \frac{1}{N_2} \sum_{n \in C2} x_n \qquad (1)$$

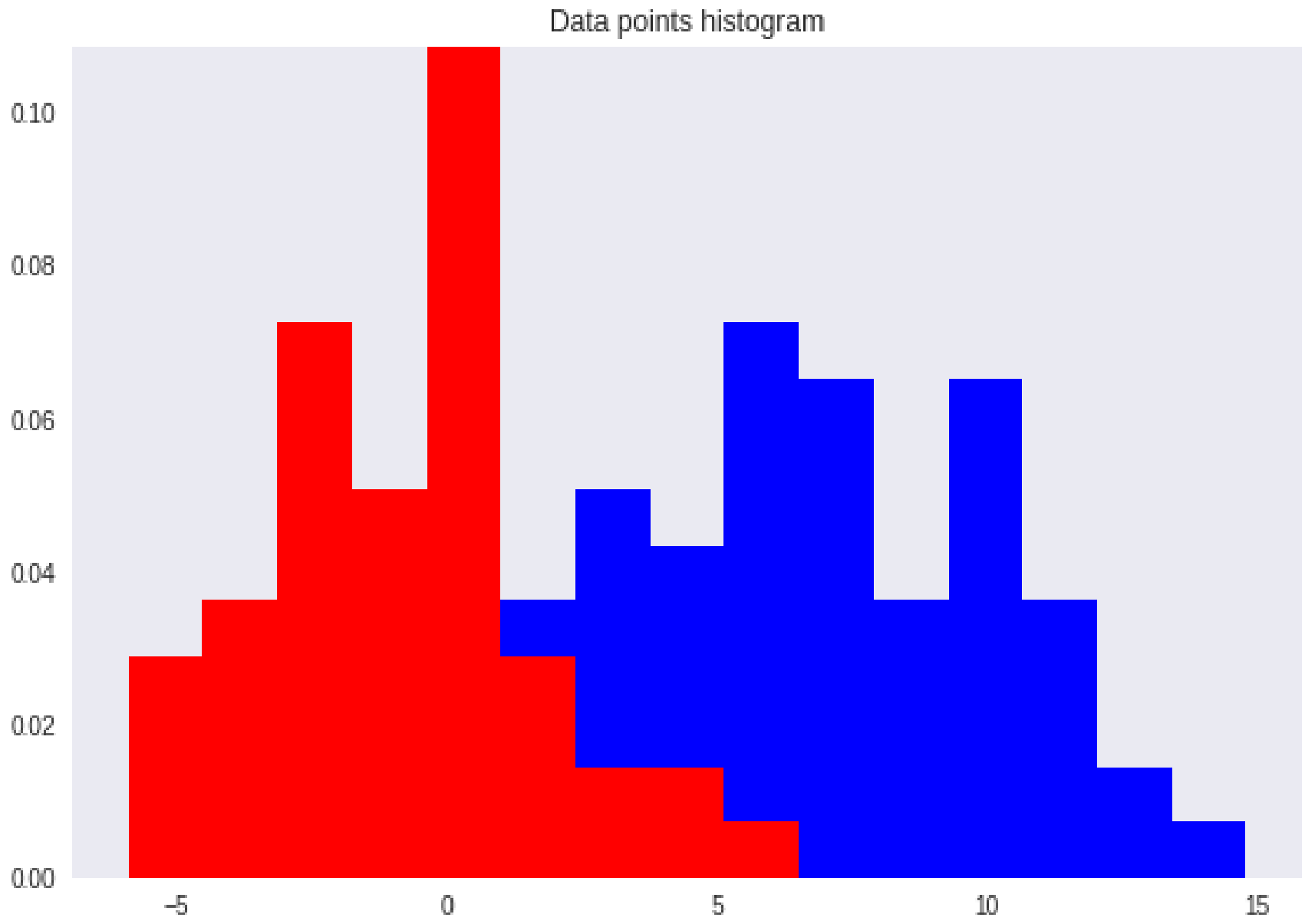
- Note that N1 and N2 denote the number of points in classes C1 and C2 respectively.
- Now, consider using the class means as a measure of separation. In other words, we want to project the data onto the vector W joining the 2 class means.


35
30
25
20
15
10
x2
−1
0
1
2
3
4
5
6
x1

FLD

- It is important to note that any kind of projection to a smaller dimension might involve some loss of information.
- In this scenario, note that the two classes are clearly separable (by a line) in their original space.
- However, after re-projection, the data exhibit some sort of class overlapping-shown by the yellow ellipse on the plot and the histogram below.


3
2
1
0
−1
−2
x2
−2
−1
0
1
2
3
4
5
6
x1

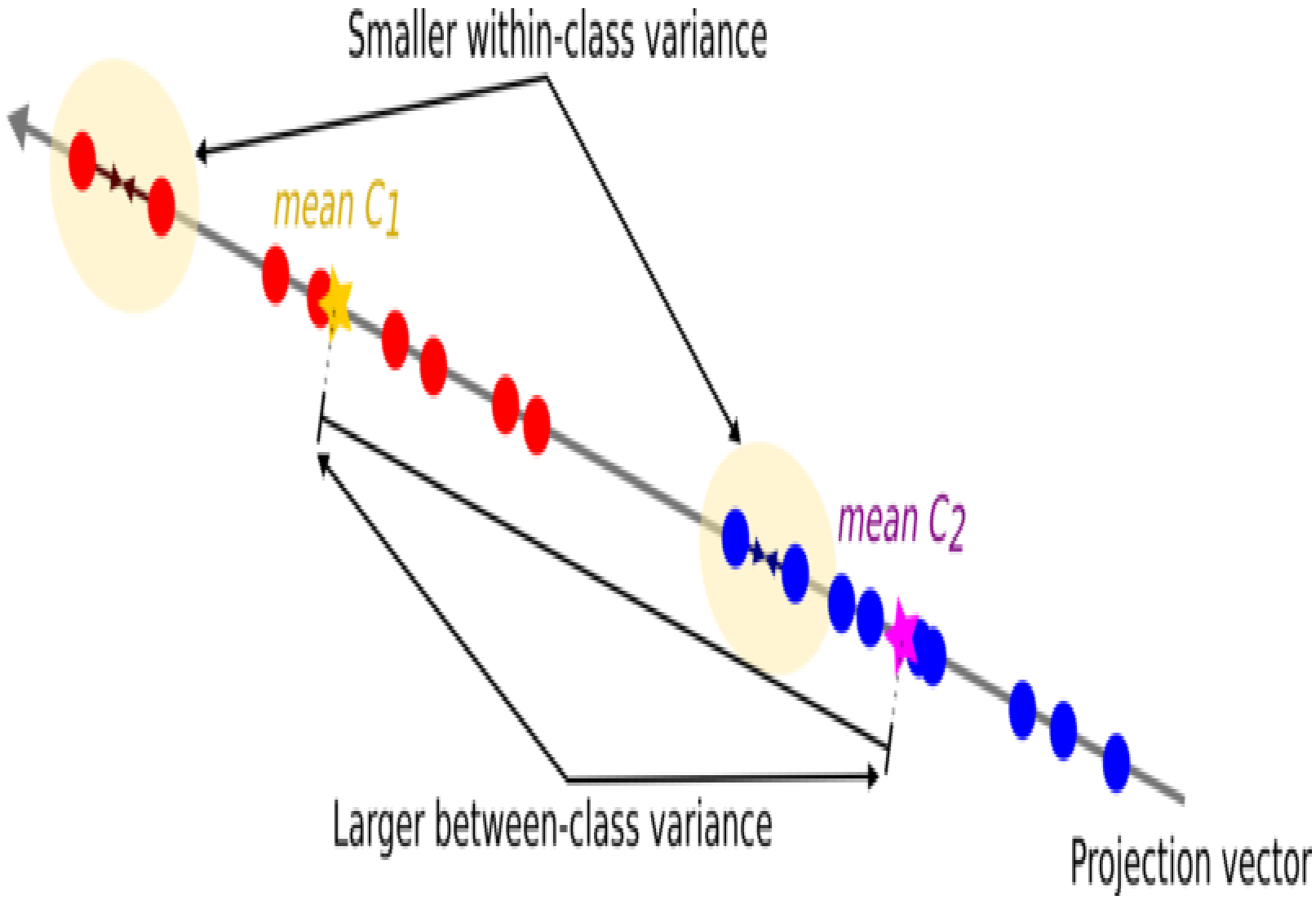

Data points histogram
0.10
0.08
0.06
0.04
0.02
0.00
−5
0
5
10
15

FLD

- That is where the Fisher's Linear Discriminant comes into play.
- The idea proposed by Fisher is to maximize a function that will give a large separation between the projected class means while also giving a small variance within each class, thereby minimizing the class overlap.
- In other words, **FLD selects a projection that maximizes the class separation.** To do that, it maximizes the ratio between the between-class variance to the within-class variance.

FLD

- In short, to project the data to a smaller dimension and to avoid class overlapping, FLD maintains 2 properties.
  - ✓ A large variance among the dataset classes.
  - ✓ A small variance within each of the dataset classes.
- Note that **a large between-class variance means that the projected class averages should be as far apart as possible. On the contrary, a small within-class variance has the effect of keeping the projected data points closer to one another.**


Smaller within-class variance
mean C_1
mean C_2
Larger between-class variance
Projection vector

To find the projection with the following properties, FLD learns a weight vector W with the following criterion.

$$J(\boldsymbol{W}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} \qquad (1)$$

Between-class variance

Within-class variance

$$s_k^2 = \sum_{n \in C_k} (y_n - m_k)^2 \qquad y_n = \boldsymbol{W}^T x_n \tag{2}$$

$$J(\boldsymbol{W}) = \frac{\boldsymbol{W}^T S_B \boldsymbol{W}}{\boldsymbol{W}^T S_W \boldsymbol{W}} \tag{3}$$

$$\boldsymbol{W} \propto S_W^{-1}(m_2 - m_1) \tag{4}$$

Per-class mean

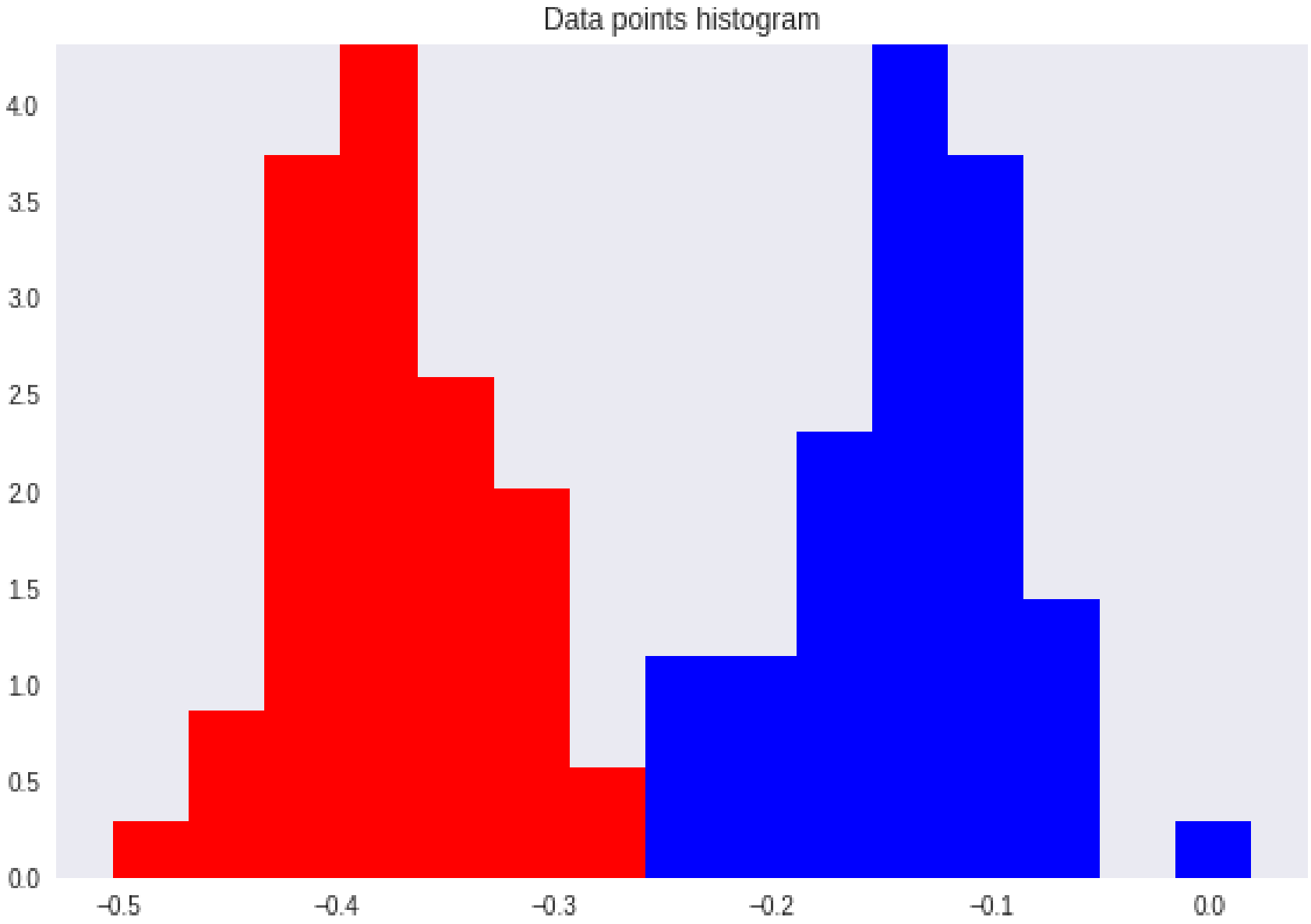
Within-class variance

projection equation

FLD

- If we substitute the mean vectors m1 and m2 as well as the variance s as given by equations (1) and (2) we arrive at equation (3).
- If we take the derivative of (3) w.r.t W (after some simplifications) we get the learning equation for W (equation 4).
- **That is, W (our desired transformation) is directly proportional to the inverse of the within-class covariance matrix times the difference of the class means.**

As expected, the result allows a perfect class separation with simple thresholding.


Data points histogram
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.3
−0.2
−0.1
0.0

Fisher's Linear Discriminant for Multiple Classes

- We can generalize FLD for the case of more than K>2 classes.
- Here, we need generalization forms for the within-class and between-class covariance matrices.

$$S_W = \sum_{k=1}^{K} S_k \tag{5}$$

$$S_k = \sum_{n \in C_k} (x_n - m_k)(x_n - m_k)^T \tag{6}$$

$$S_B = \sum_{k=1}^{K} N_k (m_k - m)(m_k - m)^T \tag{7}$$

$$\boldsymbol{W} = max_{D'}(eig(S_W^{-1} S_B)) \tag{8}$$

Within-class covariance

Between-class covariance

Fisher's Linear Discriminant for Multiple Classes

- For the within-class covariance matrix S_W, for each class, take the sum of the matrix-multiplication between the centralized input values and their transpose. Equations 5 and 6.
- For estimating the between-class covariance S_B, for each class k=1,2,3,…,K, take the outer product of the local class mean m_k and global mean m. Then, scale it by the number of records in class k - equation 7.
- The maximization of the FLD criterion is solved via an eigen decomposition of the matrix-multiplication between the inverse of S_W and S_B. **Thus, to find the weight vector "W", we take the D' eigenvectors that correspond to their largest eigenvalues (equation 8).**
- In other words, if we want to reduce our input dimension from D=784 to D'=2, the weight vector W is composed of the 2 eigenvectors that correspond to the D'=2 largest eigen values. This gives a final shape of W = (N,D'), where N is the number of input records and D' the reduced feature dimensions.

Building a linear discriminant

- Up until this point, we used Fisher's Linear discriminant only as a method for dimensionality reduction.
- To really create a **discriminant**, we can model a multivariate Gaussian distribution over a D-dimensional input vector x for each class K as:

$$\mathcal{N}(\mathbf{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\Sigma|^{1/2}}\exp\left\{-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right\} \qquad (9)$$

Building a linear discriminant

- Here μ (the mean) is a D-dimensional vector. Σ (sigma) is a DxD matrix - the covariance matrix. And |Σ| is the determinant of the covariance.

```python
# Returns the parameters of the Gaussian distributions
def gaussian(self, X):
  means = {}
  covariance = {}
  priors = {}  # p(Ck)
  for class_id, values in X.items():
    proj = np.dot(values, self.W)
    means[class_id] = np.mean(proj, axis=0)
    covariance[class_id] = np.cov(proj, rowvar=False)
    # estimate the priors using fractions of the training set data points in each of the classes.
    priors[class_id] = values.shape[0] / self.N
  return means, covariance, priors

# model a multi-variate Gaussian distribution for each class' likelihood distribution P(x|Ck)
def gaussian_distribution(self, x, u, cov):
  scalar = (1. / ((2 * np.pi) ** (x.shape[0] / 2.))) * (1 / np.sqrt(np.linalg.det(cov)))
  x_sub_u = np.subtract(x, u)
  return scalar * np.exp(-np.dot(np.dot(x_sub_u, inv(cov)), x_sub_u.T) / 2.)
```

gaussian_model.py hosted with ♥ by GitHub view raw

Building a linear discriminant

- The parameters of the Gaussian distribution: μ and Σ, are computed for each class k=1,2,3,…,K using the projected input data.
- We can infer the priors $P(C_k)$ class probabilities using the fractions of the training set data points in each of the classes (line 11).
- Once we have the Gaussian parameters and priors, we can compute class-conditional densities $P(x|C_k)$ for each class k=1,2,3,…,K individually. To do it, we first project the D-dimensional input vector x to a new D' space.
- Keep in mind that D < D'. Then, we evaluate equation 9 for each projected point. Finally, we can get the posterior class probabilities $P(C_k|x)$ for each class k=1,2,3,…,K using equation 10.

$$P(C_k|\boldsymbol{x}) = p(\boldsymbol{x}|C_k)P(C_k) \qquad (10)$$

Equation 10 is evaluated on line 8 of the score function below.

```python
def score(self,X,y):
  proj = self.project(X)
  gaussian_likelihoods = []
  classes = sorted(list(self.g_means.keys()))
  for x in proj:
    row = []
    for c in classes:  # number of classes
      res = self.priors[c] * self.gaussian_distribution(x, self.g_means[c], self.g_covariance[c])  # Compute t
      row.append(res)

    gaussian_likelihoods.append(row)

  gaussian_likelihoods = np.asarray(gaussian_likelihoods)
  # assign x to the class with the largest posterior probability
  predictions = np.argmax(gaussian_likelihoods, axis=1)
  return np.sum(predictions == y) / len(y)
```

score_function.py hosted with ♥ by GitHub view raw

Building a linear discriminant

- We then can assign the input vector x to the class $k \in K$ with the largest posterior.

Building a linear discriminant

- All parametric densities are unimodal (have a single local maximum), whereas many practical problems involve multi-modal densities (subgroups)
- Nonparametric procedures can be used with arbitrary distributions and without the assumption that the form of the underlying densities is known
- Two types of nonparametric methods:
  - ✓ Estimating class conditional densities, $P(x \mid w_j)$
  - ✓ Bypass class-conditional density estimation and directly estimate the a-posteriori probability

Density Estimation

- Basic idea: Probability that a vector x will fall in region R is:

$$P = \int_{\Re} p(x')dx' \qquad (1)$$

- P is a smoothed (or averaged) version of the density function p(x). How do we estimate P?
- If we have a sample of size n (i.i.d from p(x)), the probability that k of the n points fall in R is (binomial law):

$$P_k = \binom{n}{k} P^k (1-P)^{n-k} \qquad (2)$$

and the expected value for k is: E(k) = nP (3)

i.i.d = Independent and identically distributed

- ML estimation of unknown $P = \theta$

$\underset{\theta}{Max}(P_k \mid \theta)$ is achieved when $\hat{\theta} = \frac{k}{n} \cong P$

- Ratio k/n is a good estimate for the probability P and can be used to estimate density fn. p (x)

- Assume $p(x)$ is continuous and region R is so small that p does not vary significantly within it. Then

$$\int_{\Re} p(x')dx' \cong p(x)V \qquad (4)$$

where x is a point in R and V the volume enclosed by R

Combining equations (1) , (3) and (4) yields:

$$p(x) \cong \frac{k/n}{V}$$

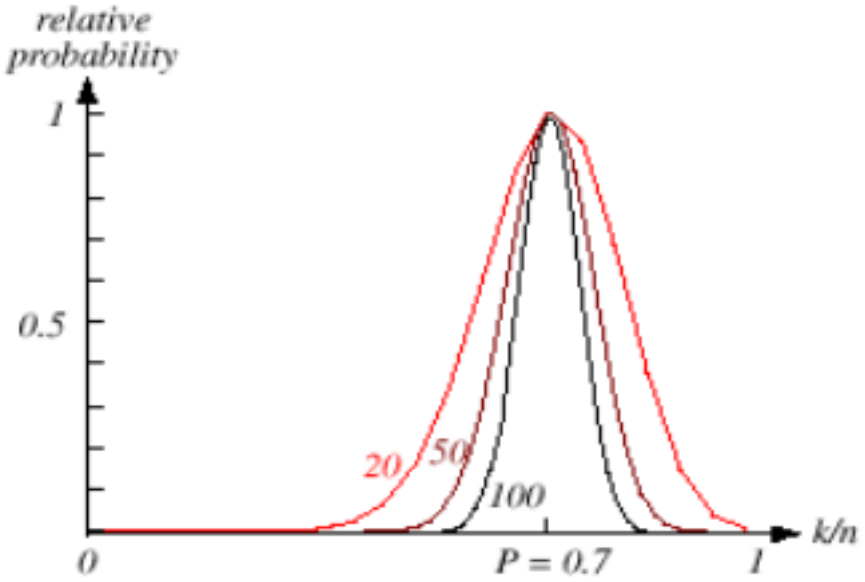


FIGURE 4.1. The relative probability an estimate given by Eq. 4 will yield a particular value for the probability density, here where the true probability was chosen to be 0.7. Each curve is labeled by the total number of patterns n sampled, and is scaled to give the same maximum (at the true probability). The form of each curve is binomial, as given by Eq. 2. For large n, such binomials peak strongly at the true probability. In the limit $n \rightarrow \infty$, the curve approaches a delta function, and we are guaranteed that our estimate will give the true probability. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

- Conditions for convergence

The fraction $k/(nV)$ is a "space averaged" value of $p(x)$. Convergence to true $p(x)$ is obtained only if V approaches zero.

$$\lim_{V \to 0, k=0} p(x) = 0 \quad (\text{if } n = \text{fixed})$$

This is the case where no samples are included in R

$$\lim_{V \to 0, k \neq 0} p(x) = \infty$$

In this case, the estimate diverges

The volume V needs to approach 0 if we want to obtain p(x) rather than just an averaged version of it.

- In practice, V cannot be allowed to become small since the number of samples , n, is always limited
- We have to accept a certain amount of averaging in p(x)
- Theoretically, if an unlimited number of samples is available, we can circumvent this difficulty as follows:

  To estimate the density of x, we form a sequence of regions $R_1, R_2, \ldots$ containing x: the first region contains one sample, the second two samples and so on.

  Let V_n be the volume of R_n, k_n be the number of samples falling in R_n and $p_n(x)$ be the n^{th} estimate for $p(x)$:

$$p_n(x) = (k_n/n)/V_n \qquad (7)$$

Necessary conditions for $p_n(x)$ to converge to $p(x)$:

$$1)\ \lim_{n\to\infty} V_n = 0$$

$$2)\ \lim_{n\to\infty} k_n = \infty$$

$$3)\ \lim_{n\to\infty} k_n / n = 0$$

Two ways of defining sequences of regions that satisfy these conditions:

(a) Shrink an initial region where $V_n = 1/\sqrt{n}$; it can be shown that

$$p_n(x) \underset{n\to\infty}{\longrightarrow} p(x)$$

This is called the Parzen-window estimation method

(b) Specify k_n as some function of n, such as $k_n = \sqrt{n}$; the volume V_n is grown until it encloses k_n neighbors of x. This is called the k_n-nearest neighbor estimation method

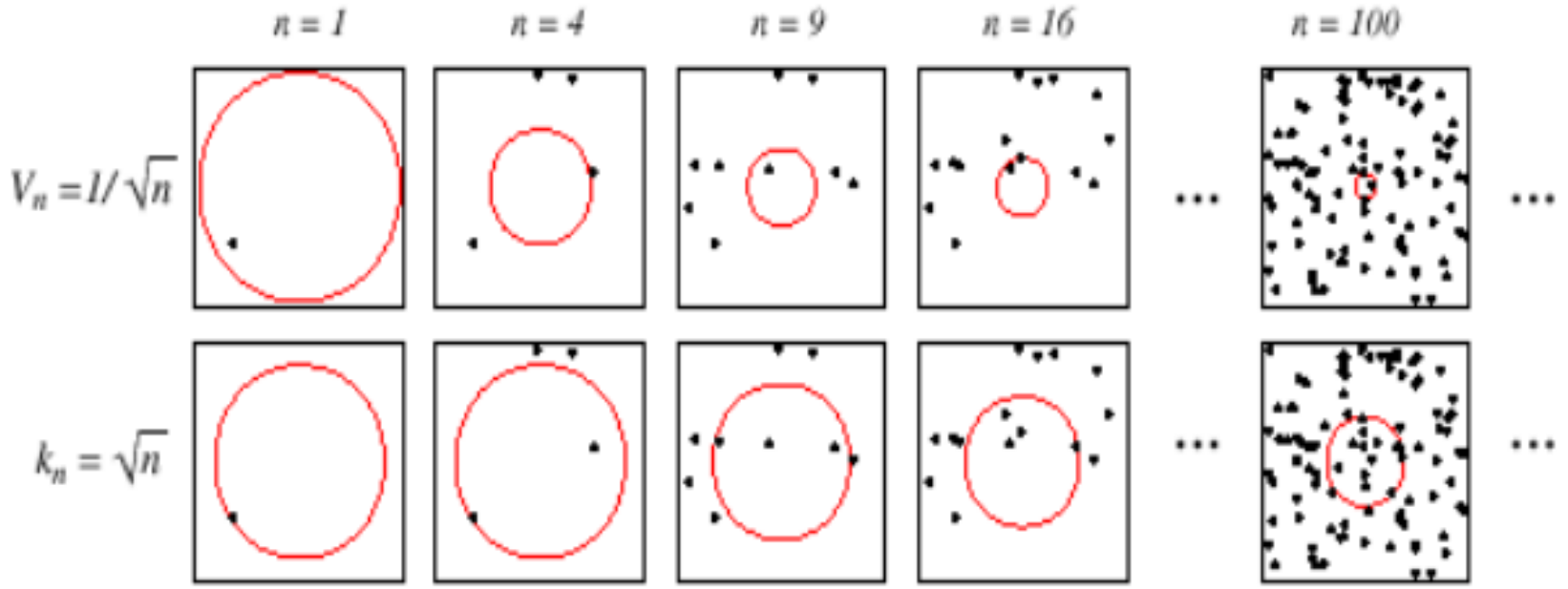


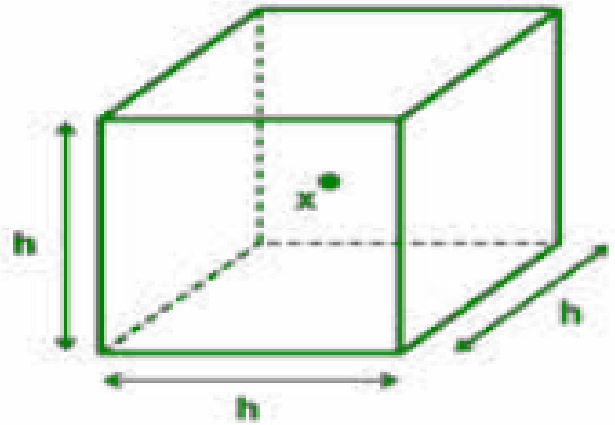
FIGURE 4.2. There are two leading methods for estimating the density at a point, here at the center of each square. The one shown in the top row is to start with a large volume centered on the test point and shrink it according to a function such as $V_n = 1/\sqrt{n}$. The other method, shown in the bottom row, is to decrease the volume in a data-dependent way, for instance letting the volume enclose some number $k_n = \sqrt{n}$ of sample points. The sequences in both cases represent random variables that generally converge and allow the true density at the test point to be calculated. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Parzen Windows density estimation technique

- Parzen Window is a non-parametric density estimation technique.
- Density estimation in Pattern Recognition can be achieved by using the approach of the Parzen Windows
- Parzen window density estimation technique is a kind of generalization of the histogram technique.
- It is used to derive a density function, f(x).
- f(x) is used to implement a Bayes Classifier.
- When we have a new sample feature and when there is a need to compute the value of the class conditional densities, f(x) is used.
- f(x) takes sample input data value and returns the density estimate of the given data sample.

Parzen Windows density estimation technique

- An d-dimensional hypercube is considered which is assumed to possess k-data samples.
- The length of the edge of the hypercube is assumed to be h_n.



Hence the volume of the hypercube is: $V_n = h_n^d$

Parzen Windows density estimation technique

- We define a hypercube window function, **φ(u)** which is an indicator function of the unit hypercube which is centered at origin:
  φ(u) = 1 if $|u_i| <= 0.5$
  φ(u) = 0 otherwise
  Here, u is a vector, $\mathbf{u} = (\mathbf{u_1}, \mathbf{u_2}, \ldots, \mathbf{u_d})^T$.
- **φ(u)** should satisfy the following:

1. $\varphi(u) >= 0; \forall u$
2. $\int_{R^d} \varphi(u).du = 1$

Let $V = \int_{R^d} \varphi\frac{u}{h})\mathrm{du} = \int_{R^d} \varphi\ (\frac{u-u_0}{h})\mathrm{du}$

Φ pronounced as phi

Since, φ(u) is centered at the origin, it is symmetric.

φ(u) = φ(-u)

- $\varphi \frac{(u-u_0)}{h}$ is a hypercube of size h cenetered at $\mathbf{u_0}$
- Let D = $\mathbf{\{x_1, x_2, ..., x_n\}}$ be the data samples.
- For any x, $\varphi \frac{(x-x_i)}{h}$ would be 1 only if x_i falls in a hypercube of side h centered at x.
- Hence the number of data points falling in a hypercube of side h centered at x is $k = \sum_{i=1}^{n} \varphi \frac{(x-x_i)}{h}$

Hence the estimated density function is :

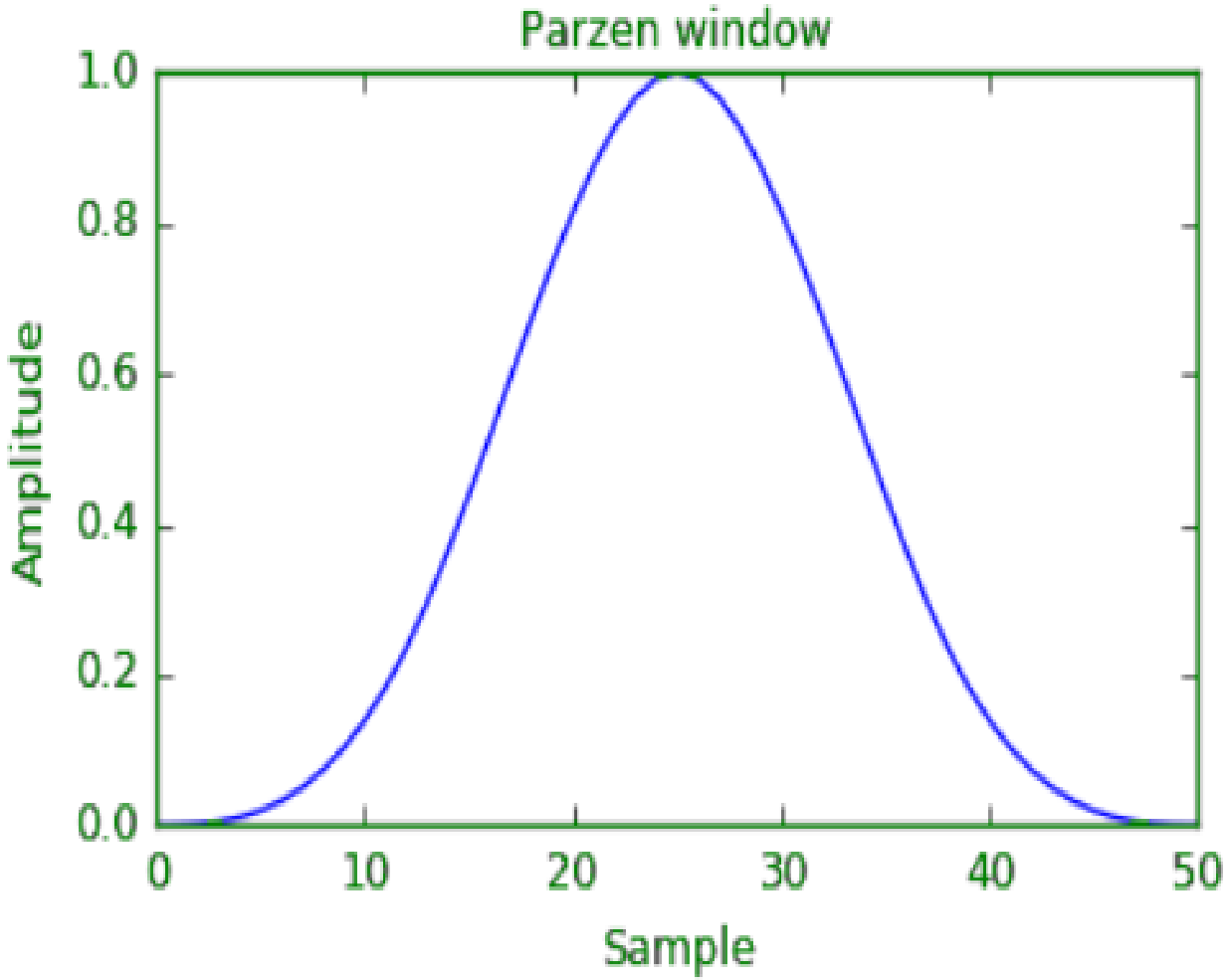
$$f(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{h^d} \varphi \frac{(x - x_i)}{h}$$

Also Since, $V_n = h_n^d$, Density Function becomes :

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{V} \varphi \frac{(x - x_i)}{h}$$

$f(x)$ would satisfy the following conditions:

1. $f(x) >= 0; \forall x$
2. $\int f(x).dx = 1$


Parzen window
1.0
0.8
0.6
0.4
0.2
0.0
Amplitude
0
10
20
30
40
50
Sample

Reading list for Parzen Window

- Parzen window method and classification by Chiho Choi

K-nearest neighbor method

- The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems.
- A **supervised machine learning algorithm** (as opposed to an unsupervised machine learning algorithm) is one that relies on labelled input data to learn a function that produces an appropriate output when given new unlabeled data.
- Supervised machine learning algorithms are used to solve classification or regression problems.

K-nearest neighbor method

- A **classification problem** has a discrete value as its output. For example, “likes pineapple on pizza” and “does not like pineapple on pizza” are discrete. There is no middle ground.
- The analogy of teaching a child to identify a pig is another example of a classification problem.

Age	Likes Pinapple on Pizza
42	1
65	1
50	1
76	1
96	1
50	1
91	0
58	1
25	1
23	1
75	1
46	0
87	0
96	0
45	0
32	1
63	0
21	1
26	1
93	0
68	1
96	0

This image shows a basic example of what classification data might look like. We have a predictor (or set of predictors) and a label. In the image, we might be trying to predict whether someone likes pineapple (1) on their pizza or not (0) based on their age (the predictor).

K-nearest neighbor method

- It is standard practice to represent the output (label) of a classification algorithm as an integer number such as 1, -1, or 0.
- In this instance, these numbers are purely representational.
- Mathematical operations should not be performed on them because doing so would be meaningless.
- A **regression problem** has a real number (a number with a decimal point) as its output.

For example, we could use the data in the table below to estimate someone's weight given their height.

Height(Inches)	Weight(Pounds)
65.78	112.99
71.52	136.49
69.40	153.03
68.22	142.34
67.79	144.30
68.70	123.30
69.80	141.49
70.01	136.46
67.90	112.37
66.78	120.67
66.49	127.45
67.62	114.14
68.30	125.61
67.12	122.46
68.28	116.09

Image showing a portion of the SOCR height and weights data set

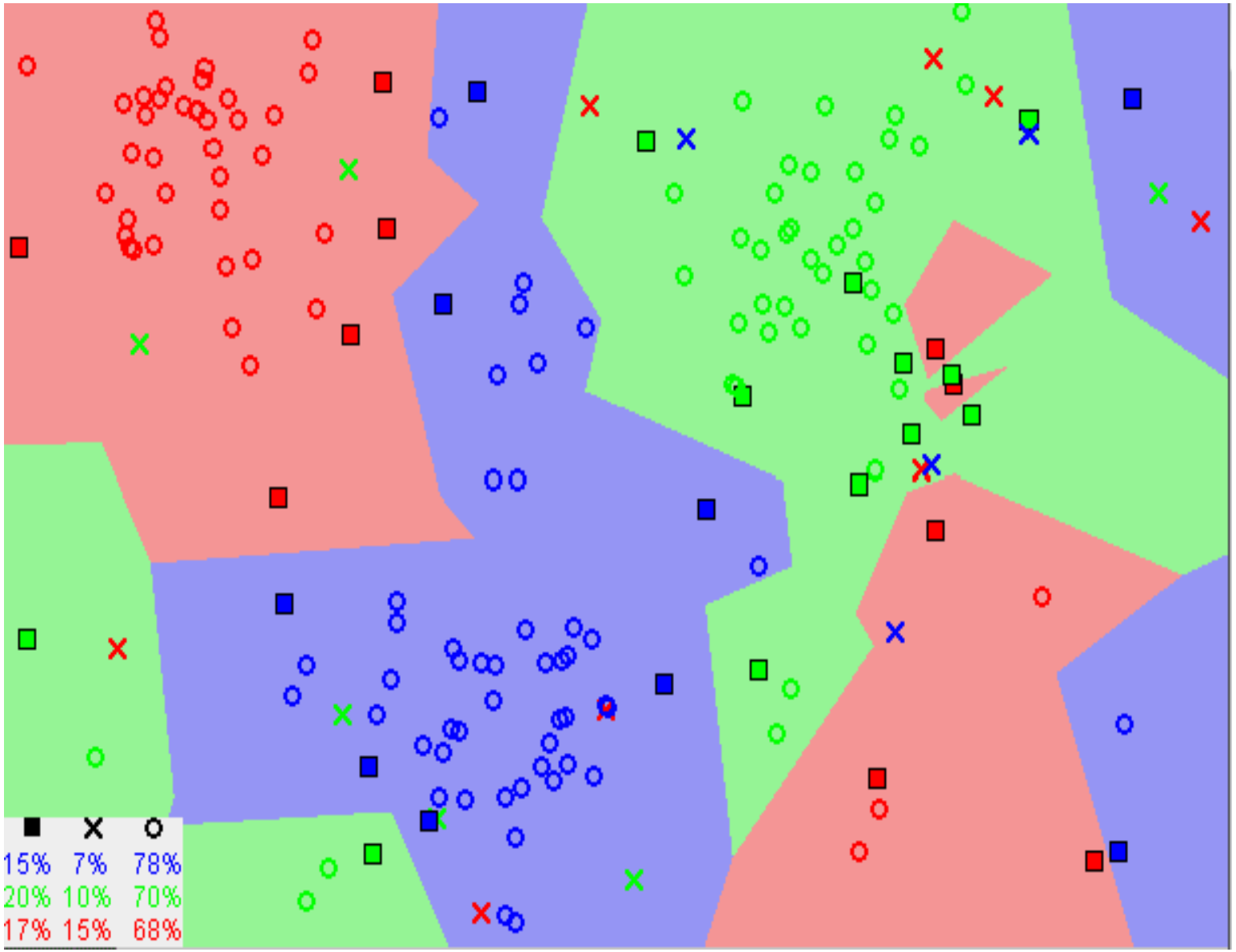
Data used in a regression analysis will look similar to the data shown in the image above. We have an independent variable (or set of independent variables) and a dependent variable (the thing we are trying to guess given our independent variables). For instance, we could say height is the independent variable and weight is the dependent variable.

K-nearest neighbor method

- Also, each row is typically called an **example, observation, or data point**, while each column (not including the label/dependent variable) is often called a **predictor, dimension, independent variable, or feature.**
- An **unsupervised machine learning** algorithm makes use of input data without any labels —in other words, no teacher (label) telling the child (computer) when it is right or when it has made a mistake so that it can self-correct.
- Unlike supervised learning that tries to learn a function that will allow us to make predictions given some new unlabeled data, unsupervised learning tries to learn the basic structure of the data to give us more insight into the data.

K-Nearest Neighbors

- The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.


15% 7% 78%
20% 10% 70%
17% 15% 68%

K-nearest neighbor method

- Notice in the image above that most of the time, similar data points are close to each other.
- The KNN algorithm hinges on this assumption being true enough for the algorithm to be useful.
- KNN captures the idea of similarity (sometimes called distance, proximity, or closeness) with some mathematics we might have learned in our childhood— calculating the distance between points on a graph.
- There are other ways of calculating distance, and one way might be preferable depending on the problem we are solving.
- However, the straight-line distance (also called the Euclidean distance) is a popular and familiar choice.

KNN Algorithm

1) Load the data
2) Initialize K to your chosen number of neighbors
3) For each example in the data
   i. Calculate the distance between the query example and the current example from the data.
   ii. Add the distance and the index of the example to an ordered collection
4) Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
5) Pick the first K entries from the sorted collection
6) Get the labels of the selected K entries
7) If regression, return the mean of the K labels
8) If classification, return the mode of the K labels

Choosing the right value for K

- To select the K that's right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

Choosing the right value for K

Some things to keep in mind:

- As we decrease the value of K to 1, our predictions become less stable. Just think for a minute, imagine K=1 and we have a query point surrounded by several reds and one green (I'm thinking about the top left corner of the colored plot above), but the green is the single nearest neighbor. Reasonably, we would think the query point is most likely red, but because K=1, KNN incorrectly predicts that the query point is green.
- Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point). Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.
- In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

KNN Advantages

- The algorithm is simple and easy to implement.
- There's no need to build a model, tune several parameters, or make additional assumptions.
- The algorithm is versatile. It can be used for classification, regression, and search.

KNN Disadvantages

- The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase.

KNN in practice

- KNN's main disadvantage of becoming significantly slower as the volume of data increases makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.
- However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, KNN can still be useful in solving problems that have solutions that depend on identifying similar objects. An example of this is using the KNN algorithm in **recommender systems**, an application of KNN-search.

Home Work

- Go through the example of Recommender System using KNN from:

https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761#:~:text=Summary-,The%20k%2Dnearest%20neighbors%20(KNN)%20algorithm%20is%20a%20simple,that%20data%20in%20use%20grows.

References

- https://data-flair.training/blogs/dimensionality-reduction-tutorial/
- https://towardsdatascience.com/principal-component-analysis-for-dimensionality-reduction-115a3d157bad
- https://priyalwalpita.medium.com/dimensionality-reduction-in-machine-learning-using-pca-e714ebddc032
- https://sthalles.github.io/fisher-linear-discriminant/
- https://www.geeksforgeeks.org/parzen-windows-density-estimation-technique/
- https://quarizmiadtech.medium.com/a-full-introduction-to-the-linear-fisher-discriminant-analysis-848530dce336
- https://www.geeksforgeeks.org/dimensionality-reduction/
- https://www.cse.msu.edu/~cse802/S17/slides/Lec_12_13_Feb22.pdf

End.