ML-MDS - Machine Learning

Jose Antonio Lorencio Abril

Spring 2023

Professor: Marta Arias

Student e-mail: jose.antonio.lorencio@estudiantat..upc.edu

This is a summary of the course *Machine Learning* taught at the Universitat Politècnica de Catalunya by Professor Marta Arias in the academic year 22/23. Most of the content of this document is adapted from the course notes by Arias, [2], so I won't be citing it all the time. Other references will be provided when used.

Contents

List of Figures

List of Algorithms

1 Introduction

I gave ChatGPT the index of these notes, and asked him for an introduction for them. Here is the result:
Welcome to the comprehensive study notes for your course in Machine Learning. This document is designed to provide you with a thorough overview of the primary topics in machine learning, to supplement your coursework and study.
Our journey will begin with an overview of Machine Learning and the foundational elements that build this exciting field. The aim is to provide you with a sound understanding of the core principles and techniques involved in the development and application of Machine Learning algorithms.
We will kick-off with 'Linear Regression,' a cornerstone of machine learning, going in-depth into topics like least squares method, issues encountered in linear regression, basis functions, and probabilistic approaches. It will also focus on Bias-Variance decomposition, ridge regression, LASSO regression, and finally, a full-Bayesian perspective.
Our exploration then continues with 'Clustering' where we cover important clustering techniques like k-Means, k-Means++, and ways to determine the optimal number of clusters. We delve deeper into the realm of Gaussian Mixtures and the Expectation-Maximization technique.
In the 'Linear Classifiers' chapter, we inspect the decision boundary in probabilistic models, generative classifiers, Naive Bayes, and methodologies like Perceptron and Logistic Regression.
Following this, we introduce 'Nearest Neighbor Prediction'—a different way of approaching classification problems. The section will equip you with tools to understand the concept of locality and choosing the best 'k', and how to combine outputs to make predictions.
We also explore the use of 'Trees and Random Forests' in machine learning. It will demystify decision trees and random forests, dealing with imbalanced data in classification, and providing insights into the interpretability of random forests.
Next, we journey into the world of 'Neural Networks' with the 'Multi-Layer Perceptron (MLP)', learning about the different error functions used for regression, binary classification, and multi-class classification. We also cover the critical process of training the MLP using Backpropagation and some common activation functions.
Finally, the notes conclude with an appendix section, 'Notes on probability theory, Bayes theorem and Bayesian learning.' This section is critical for understanding the probability theory basics, Bayes rule in the context of learning, maximum likelihood estimation, properties of estimators, maximum a posteriori estimation, and finally, Bayesian learning.
These notes aim to serve as a comprehensive reference guide for you as you explore machine learning, providing both a high-level understanding of the field and the mathematical and algorithmic details that underlie each concept. Happy learning!

2 Linear regression

2.1 Introduction

In Figure 1, we can observe a dataset of the population of different states plotted against the number of fatal accidents in each of the states. Here, each blue circle corresponds to a row of our data, and the coordinates are the $(population, \#accidents)$ values in the row. The red line is the linear regression model of this data. This means it is the line that 'best' approximates the data, where best refers to minimizing some kind of error: the squared error between each point to its projection on the y axis of the line, in this case. This approach is called the **least squares method**.

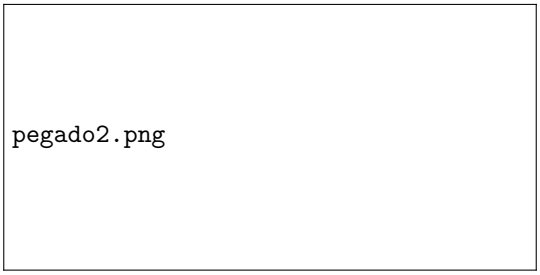


Figure 1: Matlab's accidents dataset and best linear fit.

2.2 Least squares method

2.2.1 Least squares in 2D

In 2D, we have a dataset $\{(x_i, y_i), i = 1, ..., n\}$ and we want to find the line that best approximates y as a function of x. As we want a line, we need to specify its slope, θ_1, and its intercept, θ_0. So, our estimations are:

$$\hat{y}(x_i) = \hat{y}_i = \theta_0 + \theta_1 x_i.$$

The least squares linear regression method chooses θ_0 and θ_1 in such a way that the **error function**

$$J(\theta_0, \theta_1) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \theta_0 - \theta_1 x_i)^2$$

is minimized.
Note that this function only depends on the parameters θ_0 and θ_1, since the data is assumed to be fixed (they are observations).
To compute them, we just need to find the minimum of J, by taking partial derivatives and setting them to 0. Let's do this optimization. First, we can develop the square:

$$J(\theta_0, \theta_1) = \sum_{i=1}^{n} y_i^2 + \theta_0^2 + \theta_1^2 x_i^2 - 2\theta_0 y_i - 2\theta_1 x_i y_i + 2\theta_0 \theta_1 x_i.$$

Thus:

$$\frac{\partial J}{\partial \theta_0} = \sum_{i=1}^{n} 2\theta_0 - 2y_i + 2\theta_1 x_i = 2n\theta_0 - 2\sum_{i=1}^{n} y_i + 2\theta_1 \sum_{i=1}^{n} x_i$$

and

$$\frac{\partial J}{\partial \theta_1} = \sum_{i=1}^{n} 2\theta_1 x_i^2 - 2x_i y_i + 2\theta_0 x_i = 2\theta_1 \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i y_i + 2\theta_0 \sum_{i=1}^{n} x_i.$$

We have now to solve the system given by

$$\begin{cases} 2n\theta_0 - 2\sum_{i=1}^{n} y_i + 2\theta_1 \sum_{i=1}^{n} x_i = 0 \\ 2\theta_1 \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_i y_i + 2\theta_0 \sum_{i=1}^{n} x_i = 0 \end{cases}$$

which is equivalent to

$$\begin{cases} n\theta_0 - \sum_{i=1}^n y_i + \theta_1 \sum_{i=1}^n x_i = 0 \\ \theta_1 \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i y_i + \theta_0 \sum_{i=1}^n x_i = 0 \end{cases}.$$

We can isolate θ_0 from the first equation:

$$\theta_0 = \frac{\sum_{i=1}^n y_i - \theta_1 \sum_{i=1}^n x_i}{n},$$

and substitute it in the second one

$$\theta_1 \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i y_i + \frac{\sum_{i=1}^n y_i - \theta_1 \sum_{i=1}^n x_i}{n} \sum_{i=1}^n x_i = 0,$$

which is equivalent to

$$\theta_1 \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i y_i + \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i - \theta_1 \left[\sum_{i=1}^n x_i\right]^2}{n} = 0$$

or

$$\theta_1 \left[\sum_{i=1}^n x_i^2 - \frac{\left[\sum_{i=1}^n x_i\right]^2}{n}\right] - \sum_{i=1}^n x_i y_i + \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n} = 0.$$

At this point, we can divide everything by n, yielding:

$$\theta_1 \left[\frac{\sum_{i=1}^n x_i^2}{n} - \frac{\left[\sum_{i=1}^n x_i\right]^2}{n^2}\right] - \frac{\sum_{i=1}^n x_i y_i}{n} + \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n^2} = 0.$$

If we now assume that the observations are equiprobable, i.e., $P(x_i) = \frac{1}{n}$, and we call X the random variable from which observations x_i are obtained and the same for the observations y_i, obtained from Y, then:

$$\frac{\sum_{i=1}^n x_i^2}{n} = E\left[X^2\right], \ \frac{\left[\sum_{i=1}^n x_i\right]^2}{n^2} = E[X]^2, \ \frac{\sum_{i=1}^n x_i y_i}{n} = E[XY], \ \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i}{n^2} = E[X] E[Y].$$

This means that the previous equation can be rewritten as:

$$\theta_1 \left(E\left[X^2\right] - E[X]^2\right) - (E[XY] - E[X] E[Y]) = 0 \iff \theta_1 Var[X] - Cov[X, Y] = 0$$

So

$$\theta_1 = \frac{Cov[X, Y]}{Var[X]},$$
$$\theta_0 = E[Y] - \theta_1 E[X].$$

2.2.2 Least squares regression: multivariate case

Now, we can assume that we have m independent variables $X_1, ..., X_m$ which we want to use to predict the dependent variable Y. Again, we have n observations of each variable. Now, we want to construct an hyperplane in $\mathbb{R}^{m+1}$, whose predictions would be obtained as

$$\hat{y}(X_i) = \theta_0 + \theta_1 x_{i1} + ... + \theta_m x_{im} = \theta_0 + \sum_{j=1}^m \theta_j x_{ij} = \sum_{j=0}^m \theta_j x_{ij},$$

where we define $x_{i0} = 1$, for all i. We can represent

$$X = (x_{ij})_{i=1,...,n;\ j=1,...,m}, \ Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \ \theta = \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_m \end{bmatrix},$$

so that we can write

$$\hat{Y} = X\theta.$$

The error function is defined as in the simple case

$$J(\theta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2,$$

but now we can rewrite this as

$$J(\theta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \left(Y - \hat{Y}\right)^T \left(Y - \hat{Y}\right) = (Y - X\theta)^T (Y - X\theta).$$

Again, to obtain θ we need to optimize this function using matrix calculus.

Lemma 2.1. *If* $A = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix} \in \mathcal{M}_{n\times m}(\mathbb{R})$, $\theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix} \in \mathbb{R}^m$ *and* $B = \begin{bmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nn} \end{bmatrix} \in \mathcal{M}_{m\times m}(\mathbb{R})$ *is a symmetric matrix, it holds:*

$$\frac{\partial A\theta}{\partial \theta} = A,$$

$$\frac{\partial \theta^T A^T}{\partial \theta} = A,$$

and

$$\frac{\partial \theta^T B\theta}{\partial \theta} = 2\theta^T B^T.$$

Proof. First, notice that $A\theta = \begin{bmatrix} \sum_{j=1}^m a_{1j}\theta_j \\ \vdots \\ \sum_{j=1}^m a_{nj}\theta_j \end{bmatrix} \in \mathbb{R}^n$,so it is

$$\frac{\partial A\theta}{\partial \theta} = \begin{bmatrix} \frac{\partial \sum_{j=1}^m a_{1j}\theta_j}{\partial \theta_1} & \dots & \frac{\partial \sum_{j=1}^m a_{1j}\theta_j}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sum_{j=1}^m a_{nj}\theta_j}{\partial \theta_1} & \dots & \frac{\partial \sum_{j=1}^m a_{nj}\theta_j}{\partial \theta_m} \end{bmatrix} = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix} = A.$$

For the second result, the procedure is the same.
Lastly, notice that $\theta^T B\theta = \sum_{k=1}^m \sum_{j=1}^m \theta_k b_{kj}\theta_j \in \mathbb{R}$, so

$$\frac{\partial \theta^T B\theta}{\partial \theta} = \begin{bmatrix} \frac{\partial \sum_{k=1}^m \sum_{j=1}^m \theta_k b_{kj}\theta_j}{\partial \theta_1} & \dots & \frac{\partial \sum_{k=1}^m \sum_{j=1}^m \theta_k b_{kj}\theta_j}{\partial \theta_m} \end{bmatrix} = \begin{bmatrix} 2\sum_{j=1}^m b_{1j}\theta_j & \dots & 2\sum_{j=1}^m b_{mj}\theta_j \end{bmatrix} = 2[B\theta]^T = 2\theta^T B^T.$$

□

Now, we can proceed and minimize J:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta} =& \frac{\partial (Y - X\theta)^T (Y - X\theta)}{\theta} \\ =& \frac{\partial}{\partial \theta}\left[Y^T Y - Y^T X\theta - \theta^T X^T Y + \theta^T X^T X\theta\right] \\ =& 0 - Y^T X - Y^T X + 2X^T X\theta \\ =& -2Y^T X + 2\theta^T X^T X, \end{aligned}$$

setting this to be 0, we get

$$\theta^T X^T X = Y^T X \iff X^T X\theta = X^T Y \iff \theta = \left(X^T X\right)^{-1} X^T Y.$$

Thus, the 'best' linear model is given by

$$\theta_{lse} = \left(X^T X\right)^{-1} X^T Y.$$

Once we have this model, if we have an observation of X, $x' = (x'_1, ..., x'_m)$ and we want to make a prediction, we compute

$$y' = x'\theta_{lse}.$$

The approach that we have followed here is the **optimization** view of learning, which basically consists of the steps:

1. Set up an error function as a function of some parameters.
2. Optimize this function to find the suitable values for this parameters, assuming the data as given.
3. Use incoming values to make predictions.

2.2.3 Computation of least squares solution via the singular values decomposition (SVD)

Inverting $X^T X$ can entail numerical problems, so the SVD can be used instead.

Theorem 2.1. *Any matrix $A \in \mathbb{R}^{m\times n}$, $m > n$, can be expressed as*

$$A = U\Sigma V^T,$$

where $U \in \mathbb{R}^{m\times n}$ has orthonormal columns $\left(U^T U = I\right)$, $\Sigma \in \mathbb{R}^{n\times n}$ is diagonal and contains the singular values in its diagonal and $V \in \mathbb{R}^{n\times n}$ is orthonormal $\left(V^{-1} = V^T\right)$.

Proof. Let $C = A^T A \in \mathbb{R}^{n\times n}$. C is square, symmetric and positive semidefinite. Therefore, C is diagonalizable, so it can be written as

$$C = V\Lambda V^T,$$

where $V = (v_i)_{i=1,...,n}$ is orthogonal and $\Lambda = diag\,(\lambda_1, ..., \lambda_d)$ with $\lambda_1 \geq ... \geq \lambda_r > 0 = \lambda_{r+1} = ... = \lambda_n$ and r is $rank\,(A) \leq n$.
Now, define $\sigma_i = \sqrt{\lambda_i}$, and form the matrix

$$\Sigma = \begin{bmatrix} diag\,(\sigma_1, ..., \sigma_r) & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{bmatrix}.$$

Define also

$$u_i = \frac{1}{\sigma_i} X v_i \in \mathbb{R}^m, i = 1, ..., r.$$

Then, this vectors are orthonormal:

$$u_i^T u_j = \left(\frac{1}{\sigma_i} X v_i\right)^T \left(\frac{1}{\sigma_j} X v_j\right) = \frac{1}{\sigma_i\sigma_j} v_i^T X^T X v_j = \frac{1}{\sigma_i\sigma_j} v_i^T C v_j \overset{*}{=} \frac{1}{\sigma_i\sigma_j} v_i^T \left(\lambda_j v_j\right) \overset{\left(\lambda_j = \sigma_j^2\right)}{=} \frac{\sigma_j}{\sigma_i} v_i^T v_j \overset{**}{=} 0,$$

where $(*)$ is because V is formed with the eigenvectors of C, and $(**)$ is because V is orthonormal.
Now, we can complete the base with $u_{r+1}, ..., u_n$ (using Gram-Schmidt) in such a way that

$$U = [u_1, ..., u_r, u_{r+1}, ..., u_n] \in \mathbb{R}^{n\times n}$$

is column orthonormal.
Now, if it is the case that $XV = U\Sigma$, then

$$X = XVV^T = U\Sigma V^T,$$

so it is only left to see that indeed this holds. Consider two cases:

- $1 \leq i \leq r : Xv_i = u_i\Sigma$ by the definition of u_i.

- $r+1 \leq i \leq n$: It is $Xv_i = 0$, because $X^T X v_i = C v_i = \lambda_i v_i \overset{i \geq r}{=} 0$. As $X, v_i \neq 0$, it must be $Xv_i = 0$. On the other side of the equation we also have 0 because $u_i \Sigma = u_i \sigma_i = 0$ as $i > r$.

□

This, added to the fact that if X has full rank, $X^T X$ is invertible and all its eigenvalues non null, gives us:

$$\begin{aligned}
\theta_{lse} &= \left(X^T X\right)^{-1} X^T y \\
&= \left(\left(U\Sigma V^T\right) U\Sigma V^T\right)^{-1} \left(U\Sigma V^T\right)^T y \\
&= \left(V\Sigma U^T U\Sigma V^T\right)^{-1} V\Sigma U^T y \\
&= \Sigma^{-2} V\Sigma U^T y = V\Sigma^{-1} U^T y \\
&= V \cdot diag\left(\frac{1}{\sigma_1}, ..., \frac{1}{\sigma_n}\right) \cdot U^T y.
\end{aligned}$$

Intuitive interpretation

The intuition behind the SVD is summarized in Figure 2. Basically, every linear transformation can be decomposed into a rotation, a scaling and a simpler transformation (column orthogonal).

SVD.png

Figure 2: SVD visual.

The intuition behind SVD lies in the idea of finding a low-rank approximation of a given matrix. The rank of a matrix is the number of linearly independent rows or columns it contains. A high-rank matrix has many linearly independent rows or columns, which makes it complex and difficult to analyze. On the other hand, a low-rank matrix has fewer linearly independent rows or columns, which makes it simpler and easier to analyze. SVD provides a way to find the best possible low-rank approximation of a given matrix by decomposing it into three components. The left singular vectors represent the **direction of maximum variance** in the data, while the right singular vectors represent the **direction of maximum correlation** between the variables. The singular values represent the **magnitude of the variance or correlation** in each direction.
By truncating the diagonal matrix of singular values to keep only the top-k values, we can obtain a low-rank approximation of the original matrix that retains most of the important information. This is useful for reducing the dimensionality of data, compressing images, and solving linear equations, among other applications.

Example 2.1. How to use SVD in Python and Matlab.

```python
import numpy as np

U, d, Vt = np.linalg.svd(X, full_matrices=False)
D = np.diag(1/d)
theta = Vt.T @ D @ U.T @ y
```

Algorithm 1: SVD in Python.

```matlab
[U, d, V] = svd(X)
D = diag(diag(1./d))
theta = V'*D*U'*y
```

Algorithm 2: SVD in Matlab.

2.3 Things that could go wrong when using linear regression

2.3.1 Our independent variable is not enough

It is possible that our variable X does not provide enough information to predict Y.

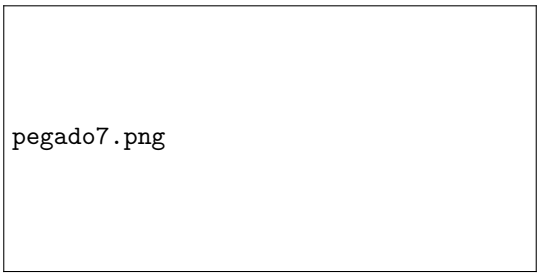


Figure 3: The independent variable does not provide enough information.

2.3.2 The relationship between the variables is not linear (underfitting)

It is also possible that the variables are related in non-linear ways.

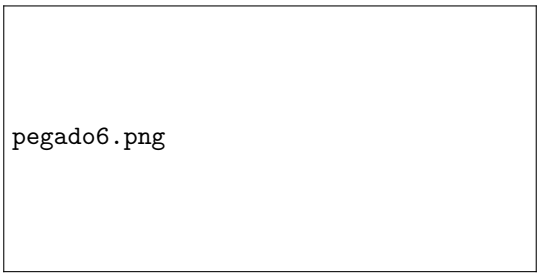


Figure 4: The variables are not linearly related.

2.3.3 Outliers affect the fit

In the presence of outliers, the model obtained can be distorted, leading to bad results.

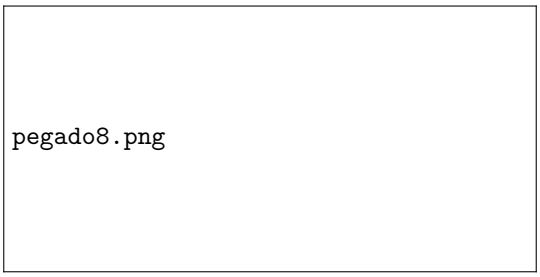


Figure 5: The outlier distort the fit.

2.4 Basis Functions

In order to fix the second problem (Subsection 2.3.2), we can make use of basis functions. The idea is to apply different transformations to the data, so that we can extend the expressive power of our model.

Definition 2.1. A **feature mapping** is a non-linear transformation of the inputs $\phi : \mathbb{R}^m \to \mathbb{R}^k$.
The resulting **predictive function** or model is $y = \phi(x)\,\theta$.

Example 2.2. For example, we can consider the **polynomial expansion of degree** k, which is a commonly used feature mapping that approximates the relationship between the independent variable x and the dependent variable y to be polynomial of degree k, i.e.:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + ... + \theta_k x^k.$$

The feature mapping is $\phi(x) = \begin{pmatrix} 1 & x & x^2 & ... & x^k \end{pmatrix}$.

Note that the idea is to transform the data so that the fit is still linear, even if the relationship is not. Of course, this requires to apply the same transformation whenever we receive an input for which we want to make predictions. Also, the resulting model is more complex, so **complexity control** is necessary to avoid overfitting.
When we apply ϕ to the input matrix X, we get a new input matrix, given by

$$\Phi = \begin{pmatrix} \phi(x_1) \\ \phi(x_2) \\ \vdots \\ \phi(x_n) \end{pmatrix} = \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \dots & \phi_m(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \dots & \phi_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \dots & \phi_m(x_n) \end{pmatrix},$$

and we obtain the optimal solution as before:

$$\theta_{min} = \arg\min_{\theta} (y - \Phi\theta)^T (y - \Phi\theta) = \left(\Phi^T \Phi\right)^{-1} \Phi^T y.$$

Example 2.3. A MATLAB example

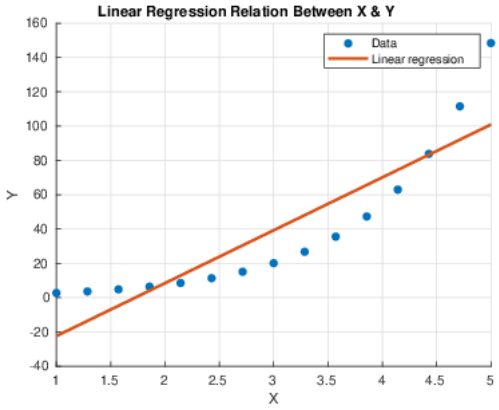
Example 2.3

First, we define the dataset. In this case $y = e^x$.

```
x=linspace(1,5,15)';
y=exp(x);
```

Now, we first see what happens with linear regression:

```
b1 = X\y;
yCalc1 = X*b1;
figure;

scatter(x,y,'filled')
hold on
plot(x,yCalc1,'LineWidth',2)
xlabel('X')
ylabel('Y')
title('Linear Regression Relation Between X & Y')
legend('Data','Linear regression')
grid on
hold off
```

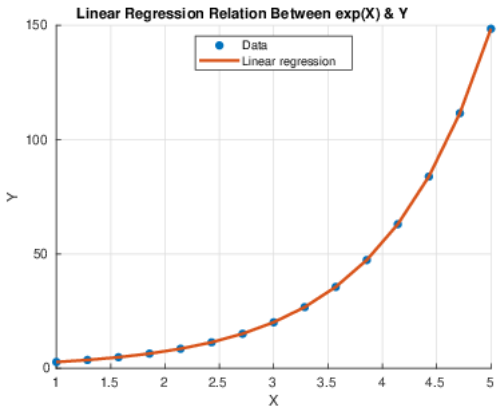


As we can see, the model does not fit the data addecuately. We can use a feature mapping and repeat the process with the transformed input:

```
X = feat_map(x)
```

```
X = 15x3
    1.0000    1.0000    2.7183
    1.0000    1.2857    3.6173
    1.0000    1.5714    4.8135
    1.0000    1.8571    6.4054
    1.0000    2.1429    8.5238
    1.0000    2.4286   11.3427
    1.0000    2.7143   15.0938
    1.0000    3.0000   20.0855
    1.0000    3.2857   26.7281
    1.0000    3.5714   35.5674
```

```
b2 = X\y;
yCalc2 = X*b2;

scatter(x,y,'filled')
hold on
plot(x,yCalc2,'LineWidth',2)
xlabel('X')
ylabel('Y')
title('Linear Regression Relation Between exp(X) & Y')
legend('Data','Linear regression','Location','north')
grid on
hold off
```



As we can see, the model is now perfectly fitting the data!

```
function X=feat_map(x)
    X = [ones(size(x)) x exp(x)];
end
```

2.5 Probabilistic approach

A review on probability theory, Bayes theorem and Bayesian Learning is in Appendix A.

2.5.1 Least squares regression from a probabilistic perspective

We are noe going to derive the linear regression estimates using the principle of maximum likelihood and the univariate Gaussian distribution, whose probability density function is given by

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}.$$

Given a sample $D = \{x_1, ..., x_n\}$, where $x_i \sim \mathcal{N}(\mu, \sigma)$, we define its likelihood as the function

$$\mathcal{L}(\mu, \sigma, D) = P(D; \mu, \sigma) = \prod_i p(x_i; \mu, \sigma).$$

In Figure 6, we can see how the likelihood relates to '*how likely it is that our points have been created froma certain distribution*', because the red outcomes are more likely to appear from the blue distribution than the green ones.

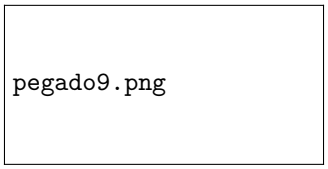


Figure 6: Likelihood example.

Now, this can be used to select the distribution that best matches our data. As an easy approach, suppose we want to decide between two distributions F_1 and F_2, and we have a dataset D. To decide, we can compute $\mathcal{L}(F_1, D)$ and $\mathcal{L}(F_2, D)$ and select the distribution whose likelihood is greater. This is visually exemplified in Figure 7, where we can see that given the three red outcomes and the two distributions (blue and green), the blue one should be preferred, because it mazimizes the likelihood.

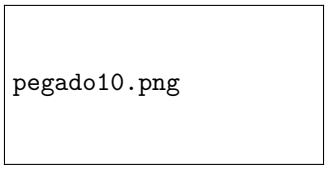


Figure 7: Using likelihood to select the distribution.

This way, we can think of the likelihood of a function of the unknown parameters of the distribution, with the dataset fixed, and we can maximize this function to obtain the parameters that best describe our data.
In the probabilistic setting of linear regression, we assume that each label y_i we observe is normally distributed, with mean $\mu = x_i\theta$ and variance σ^2:

$$y_i = x_i\theta + \varepsilon_i,$$

where $\varepsilon_i \sim \mathcal{N}\left(0, \sigma^2\right)$. This way, we seek to obtain θ and σ^2 that best describe our data. Remember that

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \; X = \begin{pmatrix} 1 & x_{11} & ... & x_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & ... & x_{nd} \end{pmatrix},$$

so that the likelihood of the parameter vector θ is given by

$$\mathcal{L}\left(\theta, \sigma\right) = P\left(y|X; \theta, \sigma^2\right) = \prod_{i=1}^{n} p\left(y_i|x_i; \theta, \sigma^2\right).$$

It is usual to maximize the log-likelihood instead, basically because the likelihood tends to give values too close to zero (we may be multiplying thousands of small values), so numerical problems may arise. Thus:

$$\begin{aligned} l\left(\theta, \sigma^2\right) = \log \mathcal{L}\left(\theta, \sigma^2\right) &= \log \prod_{i=1}^{n} p\left(y_i|x_i; \theta, \sigma^2\right) \\ &= \sum_{i=1}^{n} \log p\left(y_i|x_i; \theta, \sigma^2\right) \\ &= \sum_{i=1}^{n} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\left(y_i - x_i\theta\right)^2}\right] \\ &= \sum_{i=1}^{n} \left(\log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\left(y_i - x_i\theta\right)^2\right) \\ &= -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_i - x_i\theta\right)^2 \\ &= -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y - X\theta\right)^T\left(y - X\theta\right). \end{aligned}$$

At this point, we differentiate and set equal to 0:

$$\frac{\partial}{\partial\theta} l\left(\theta, \sigma^2\right) = -\frac{1}{2\sigma^2}\left(-2X^T y + 2X^T X\theta\right) = 0$$

$$\frac{\partial}{\partial\sigma^2} l\left(\theta, \sigma^2\right) = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\left(y - X\theta\right)^T\left(y - X\theta\right) = 0,$$

obtaining

$$\theta_{ML} = \left(X^T X\right)^{-1} X^T y$$

and

$$\sigma^2_{ML} = \frac{1}{n}\left(y - X\theta\right)^T\left(y - X\theta\right) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - x_i\theta_{ML}\right)^2 = MSE.$$

It is noticeable that the maximum likelihood estimates concide with the estimates we found minimizing the squared error. This is a consequence of assuming gaussian noise, and other types of distribution would give us the same estimates as minimizing a different error function.

2.6 Bias-Variance decomposition

Let $f : \mathbb{R}^d \to \mathbb{R}$ be the true function that we are trying to approximate and $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ a finite training dataset, where $y_i = f(x_i) + \varepsilon_i$ and $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$. Let $x \in \mathbb{R}^d$ be a test data point. The setup is using D to train a model $\hat{f}$ that we want to use to make predictions

$$\hat{y}_D = \hat{f}(x).$$

We are going to see how the expected squared error $(y - \hat{y}_D)^2$, where y is the real value, can be decomosed as a sum of the following components:

- **Irreducible error**: given by σ^2.
- **Bias**: the systematic limitation that the modelling assumptions impose. For example, if we choose linear approximations, we will never be able to model non-linear data well enough.
- **Variance**: refers to the sensitivity of the model to the training set D. The more the model varies when the training set is changed, the higher variance it has.

Let's do this:

$$\begin{aligned}
E\left[(y - \hat{y}_D)^2\right] =& E\left[(f(x) + \varepsilon - \hat{y}_D)^2\right] = E\left[(f(x) + \varepsilon - \hat{y}_D + E[\hat{y}_D] - E[\hat{y}_D])^2\right] \\
=& E\left[((f(x) - E[\hat{y}_D]) + \varepsilon + (E[\hat{y}_D] - \hat{y}_D))^2\right] \\
=& E\left[(f(x) - E[\hat{y}_D])^2\right] + E\left[\varepsilon^2\right] + E\left[(E[\hat{y}_D] - \hat{y}_D)^2\right] \\
&+ 2E[(f(x) - E[\hat{y}_D])\varepsilon] + 2E[(f(x) - E[\hat{y}_D])(E[\hat{y}_D] - \hat{y}_D)] \\
&+ 2E[\varepsilon(E[\hat{y}_D] - \hat{y}_D)] \\
=& E\left[(f(x) - E[\hat{y}_D])^2\right] + \sigma^2 + E\left[(E[\hat{y}_D] - \hat{y}_D)^2\right] \\
&+ 2E[(f(x) - E[\hat{y}_D])]\,\cancelto{0}{E[\varepsilon]} + 2E[(f(x) - E[\hat{y}_D])]\,\cancelto{0}{E[(E[\hat{y}_D] - \hat{y}_D)]} \\
&+ 2E[(E[\hat{y}_D] - \hat{y}_D)]\,\cancelto{0}{E[\varepsilon]} \\
=& E\left[(f(x) - E[\hat{y}_D])^2\right] + \sigma^2 + E\left[(E[\hat{y}_D] - \hat{y}_D)^2\right].
\end{aligned}$$

And we now define

$$\begin{aligned}
Bias[\hat{y}_D] &= E\left[(f(x) - E[\hat{y}_D])^2\right], \\
Variance[\hat{y}_D] &= E\left[(E[\hat{y}_D] - \hat{y}_D)^2\right].
\end{aligned}$$

The Bias reflects the expected difference between our assumed model and the real function, while the variance reflects the difference between the assumed model and the obtained model. In Figure 8, we can see:

- The linear model has high bias and low variance.
- The polynomial of degree 3 has low bias and moderate variance.
- The polynomial of degree 8 has low bias but high variance.

pegado14.png

Figure 8: Visual representation of the Bias-Variance errors.

A summary of commonly used errors used for regression is shown in Table 1.

Name	Abbreviation	Formula
mean squared error	MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i\theta)^2$
root mean squared error	RMSE	$\sqrt{MSE}$
normalized root mean squared error	NRMSE	$\sqrt{\frac{MSE}{Var(y)}}$
coefficient of determination	R^2	1-NRMSE^2
mean absolute error	MAE	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i - x_i\theta\rvert$

Table 1: Common error functions.

2.7 Ridge Regression from Gaussian prior

We are going to consider MAP estimates in this section, which are explained in the Annex A.
Assume isotropic Gaussian prior on d-dimensional θ, i.e., $\theta \sim \mathcal{N}\left(\mu = 0, \Sigma = \tau^2 I\right)$, so that $\Sigma^{-1} = \frac{1}{\tau^2} I$ and $\det \Sigma = \tau^{2d}$. Then, on one hand, we have

$$\begin{aligned} P\left(\theta; \mu = 0, \Sigma = \tau^2 I\right) =& \frac{1}{\det\left(\Sigma\right)^{\frac{1}{2}} \left(2\pi\right)^{\frac{d}{2}}} \exp\left\{-\frac{1}{2}\left(y-\mu\right)^T \Sigma^{-1}\left(y-\mu\right)\right\} \\ =& \frac{1}{\left(2\pi\tau^2\right)^{\frac{d}{2}}} \exp\left\{-\frac{1}{2\tau^2}\theta^T\theta\right\} \\ =& \frac{1}{\left(2\pi\tau^2\right)^{\frac{d}{2}}} \exp\left\{-\frac{\|\theta\|^2}{2\tau^2}\right\}. \end{aligned}$$

On the other hand, it is

$$\begin{aligned} P\left(\theta|y,X\right) \propto& P\left(y|X,\theta\right) P\left(\theta\right) \\ \propto& \exp\left\{-\frac{1}{2\sigma^2}\left(y-X\theta\right)^T\left(y-X\theta\right)\right\} \exp\left\{-\frac{\|\theta\|^2}{2\tau^2}\right\} \\ =& \exp\left\{-\frac{1}{2\sigma^2}\left(y-X\theta\right)^T\left(y-X\theta\right) - \frac{\|\theta\|^2}{2\tau^2}\right\}. \end{aligned}$$

To obtain the MAP, we maximize the log of this expression:

$$\begin{aligned} \theta_{MAP} =& \arg\max_\theta \log\left[P\left(y|X,\theta\right)P\left(\theta\right)\right] \\ =& \arg\max_\theta \left[-\frac{1}{2\sigma^2}\left(y-X\theta\right)^T\left(y-X\theta\right) - \frac{\|\theta\|^2}{2\tau^2}\right] \\ =& \arg\min_\theta \left[\left(y-X\theta\right)^T\left(y-X\theta\right) + \frac{\sigma^2}{\tau^2}\|\theta\|^2\right] \\ =& \arg\min_\theta \left[\left(y-X\theta\right)^T\left(y-X\theta\right) + \lambda\|\theta\|^2\right] \end{aligned}$$

which is the **ridge regression** estimate with $\lambda = \frac{\sigma^2}{\tau^2}$. We can now differenciate this expression to find its minimum:

$$\begin{aligned} \frac{\partial}{\partial\theta}\left[\|y-X\theta\|^2 + \lambda\|\theta\|^2\right] =& -y^TX - X^Ty + 2X^TX\theta + 2\lambda\theta = -2X^Ty + 2X^TX\theta + 2\lambda\theta = 0 \\ \iff& \left(2X^TX + 2\lambda I\right)\theta = 2X^Ty \\ \iff& \theta_{MAP} = \theta_{ridge} = \left(X^TX + \lambda I\right)^{-1}X^Ty. \end{aligned}$$

Remark 2.1. There are a few remarks here:

- λ controls the complexity of the solution θ, the bigger λ is, the smaller θ tends to be, leading to simpler solutions.

- $X^TX + \lambda I$ is guaranteed to be non-singular and behaves better numerically than X^TX, specially if there is high correlation in the columns of X, or if there are few observations relative to the amount of variables.
- A general approach when we have a regularized objective function is to use models that are potentially more complex than needed, and then adjust λ until obtaining good result and simpler models.

2.7.1 Tuning λ

Cross-validation

Given a dataset D, the k-cross-validation approach starts by separating D into two subsets:

- The **training dataset**, D_{train}.
- The **test dataset**, D_{test}.

These are obtained in such a way that:

1. $D_{train} \cap D_{test} = \emptyset$.
2. $D_{train} \cup D_{test} = D$.

Now, we divide D_{train} into k subsets of equal size, $\{D_{train,i}\}_{i=1}^{k}$, called **folds**, and imagine we want to decide on a set of values for our model's parameter $\lambda \in \Lambda = \{\lambda_1, ..., \lambda_l\}$.

1. For each $\lambda \in \Lambda$:
   (a) For each $i = 1, ..., k$:
      i. Train the model in $D_{train} \setminus D_{train,i}$.
      ii. Evaluate the model in $D_{train,i}$.
   (b) Average the k evaluations to obtain an estimation of the performance of the model.
2. Select λ that gives us the best estimation.

Leave-one-out cross-validation (LOOCV)

Is a special case of cross-validation, in which $k = n$, i.e., each data point is a fold.

2.7.2 LOOCV for Ridge regression

As a particularity of linear and ridge regression, for a given value of λ, only one training is necessary for LOOCV, so we can proceed as follows:

1. For each $\lambda \in \Lambda$:
   (a) Compute the optimal solution
   $$\hat{\theta}_\lambda = \left(X^TX + \lambda I\right)^{-1} X^T y.$$
   (b) Compute the *hat matrix* or smoothing matrix
   $$H_\lambda = (h_{ij})_{ij} = X\left(X^TX + \lambda I\right)^{-1} X^T.$$
   (c) Compute LOOCV directly for each λ, without folding
   $$LOOCV\,(\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - f\,(x_i)\,\theta_\lambda}{1 - h_{ii}}\right)^2.$$
2. Return λ with minimum LOOCV.

2.7.3 Generalized Cross-Validation (GCV)

Generalized Cross-Validation (GCV) is a model selection technique used to estimate the performance of a model in terms of prediction accuracy. GCV is particularly useful for choosing the optimal parameters in regularization or smoothing methods, where the goal is to balance model complexity and goodness of fit. Examples of such methods include ridge regression, LASSO, and smoothing splines.
The main idea of GCV is to approximate the leave-one-out cross-validation (LOOCV) error without actually performing the computationally expensive process of fitting the model to all but one data point multiple times. Also, it is more computationally stable than the previous approach.
The GCV score is defined as follows:

$$GCV(\lambda) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - f(x_i)\,\theta_\lambda}{1-\frac{Tr(H_\lambda)}{n}}\right)^2 = \frac{MSE_\lambda}{\left(1-\frac{Tr(H_\lambda)}{n}\right)},$$

where $Tr(H_\lambda)$ is the trace of H_λ.

2.8 LASSO regression

Definition 2.2. The ***Lp*−norm** of a vector θ is

$$\|\theta\|_p = \left(\sum_{i=1}^{n}|\theta_d|^p\right)^{\frac{1}{p}}.$$

Remark 2.2. As we have seen, assuming an isotropic Gaussian prior on the parameters leads to ridge regression, which minimizes the L2-norm of θ and the squared error.

Another very common choice is $p = 1$, which leads to **LASSO regression**. Thus, LASSO regression minimizes the L1-norm of parameters and squared error:

$$\theta_{LASSO} = \arg\min_\theta \left[\|y - X\theta\|_2^2 + \lambda\|\theta\|_1\right].$$

In fact, LASSO regression arises assuming a Lapace distribution prior over the parameters.
Some characteristics:

- LASSO regularized cost function is not quadratic anymore, and it has no close solution, so an approximation procedure is used: the **least angle regression**, which provides an efficient way to compute the solutions for a list of possible values for $\lambda > 0$, giving the **regularization path**.
- LASSO regression gives sparse solutions, in which many coefficients/coordinates of θ might be 0. This means that LASSO performs feature selection automatically.

2.9 The full-Bayesian perspective

Both maximum likelihood (ML) and maximum a priori (MAP) produce point estimates of the parameters, while in Bayesian Learning[1] we want the full posterior distribution of the parameters. The idea is that if we know the posterior distribution of the parameters, then we can use all the information provided by this distribution for our predictions, instead of just a single point-estimate that summarizes it. For instance, if the probability function of the posterior is $p(\theta|D)$ and we receive a new input point x, then we can compute the probability of $f(x) = y$ by

$$p(y|x,D) = \int_\Theta p(y|x,\theta,D)\,p(\theta|D)\,d\theta.$$

Now, when we do this for all possible values of y, we obtain the full distribution of the predictions, instead of just one estimation. Nonetheless, computing this integral is usually too hard, so it needs to be approximated, but in the context of linear regression all these expressions have close-form formulas[2].

[1] Refer to Appendix A for more details.
[2] For computational speed we may use approximations as well.

Technically, ML and MAP assume that $Y \sim \mathcal{N}\left(x^T\theta, \sigma^2\right)$, so a prediction for a new test point $\overline{x}$ is going to have a distribution $\mathcal{N}\left(\overline{x}^T\hat{\theta}, \sigma^2\right)$. Note now the lack of flexibility of this approach, since the width of the normal distribution is going to be the same for any new test point, which may be a dangerous assumption.
Let $D = \{(x_i, y_i)\}_{i=1}^n$ be our dataset, with $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, and assume:

- $Y_i, ..., Y_n$ are independent given θ.
- $Y_i \sim \mathcal{N}\left(x_i^T\theta, \frac{1}{a}\right)$, with $a > 0$ being the prevision of the noise in the observations $\left(a = \frac{1}{\sigma^2}\right)$.
- A spherical or isotropic Gaussian for the parameter's prior, $p(\theta) \sim \mathcal{N}\left(0, b^{-1}I\right)$.
- a and b are known.
- The only parameter variables are the coefficients $\theta = (\theta_0, ..., \theta_d)^T$.

Then, the likelihood function is

$$p(D|\theta) \propto \exp\left\{-\frac{a}{2}(y - X\theta)^T(y - X\theta)\right\}.$$

As usual, using Bayes, we can derive the posterior distribution

$$\begin{aligned} p(\theta|D) \propto & p(D|\theta)\, p(\theta) \\ & \propto \exp\left\{-\frac{a}{2}(y - X\theta)^T(y - X\theta)\right\} \exp\left\{-\frac{b}{2}\theta^T\theta\right\} \\ & \propto \exp\left\{-\frac{a}{2}(y - X\theta)^T(y - X\theta) - \frac{b}{2}\theta^T\theta\right\}. \end{aligned}$$

Notice here that the exponent of this expression is quadratic on θ, so it is going to be a multivariate Gaussian[3]. We need to turn the exponent into something resembling

$$(\theta - \mu)^T Q(\theta - \mu),$$

so that we can derive what the mean μ and the precision Q of the posterior density is. For this, we are going to complete squares so that we can 'match the terms' between $(\theta - \mu)^T Q(\theta - \mu)$ and $a(y - X\theta)^T(y - X\theta) + b\theta^T\theta$. On one hand:

$$\begin{aligned} (\theta - \mu)^T Q(\theta - \mu) =& \theta^T Q\theta - \theta Q\mu - \mu^T Q\theta + \mu^T Q\mu \\ =& \theta^T Q\theta - 2\theta^T Q\mu + const. \end{aligned}$$

We don't mind about constant terms with respect to θ, since we only care about proportionality. On the other hand, we have:

$$\begin{aligned} a(y - X\theta)^T(y - X\theta) + b\theta^T\theta =& ay^Ty - ay^TX\theta - a\theta^TX^Ty + a\theta^TX^TX\theta + b\theta^T\theta \\ =& ay^Ty - 2a\theta^TX^Ty + \theta^T\left(aX^TX + bI\right)\theta. \end{aligned}$$

From here, we obtain that

$$Q = aX^TX + bI,$$

and we now match $-2a\theta^TX^Ty$ with $-2\theta^TQ\mu$[4]:

$$aX^Ty = Q\mu \iff \mu = aQ^{-1}X^Ty.$$

Thus, the posterior probability is $p(\theta|D) \sim \mathcal{N}\left(\mu, Q^{-1}\right)$ with

$$Q = aX^TX + bI,\ \mu = aQ^{-1}X^Ty.$$

The MAP estimate can be directly obtained from here, since the maximum density is obtained at the mean in any Gaussian distribution. Additionally, in ridge regression we let $\lambda = \frac{b}{a}$ and turn it into a parameter that we can tune to control complexity against training error.

[3] The idea is that the product of two Gaussians is also Gaussian. Think of it as if you are measuring height and IQ: these variables can be assumed independent, and have a normal distribution. You can multiply them and obtain a combined normal distribution in 2-D. The idea here is basically the same.

[4] Q is invertible because it is positive definite, thanks to the $+bI$, with $b > 0$.

2.9.1 Using the posterior distribution for predictions

Let's now see how to compute the **predictive distribution**, i.e.,

$$p\left(y|x,D\right)=\int_{\Theta}p\left(y|x,\theta,D\right)p\left(\theta|D\right)d\theta.$$

For this, we substitute the densities

$$\begin{aligned}p\left(y|x,D\right)&=\int_{\Theta}\mathcal{N}\left(y|x^{T}\theta,\frac{1}{a}\right)\mathcal{N}\left(\theta|Q^{-1}\right)d\theta\\&\overset{w.r.t.\ y}{\propto}\int_{\Theta}\exp\left\{-\frac{a}{2}\left(y-x^{T}\theta\right)^{2}\right\}\exp\left\{-\frac{1}{2}\left(\theta-\mu\right)^{T}Q\left(\theta-\mu\right)\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{a}{2}\left(y^{2}-2\left(x^{T}\theta\right)y+\left(x^{T}\theta\right)^{2}\right)-\frac{1}{2}\left(\theta^{T}Q\theta-2\theta^{T}Q\mu+\cancel{\mu^{T}Q\mu}\right)\right\}d\theta\\&\propto\int_{\Theta}\exp\left\{-\frac{a}{2}\left(y^{2}-2\left(x^{T}\theta\right)y+\left(x^{T}\theta\right)^{2}\right)-\frac{1}{2}\left(\theta^{T}Q\theta-2\theta^{T}Q\mu\right)\right\}d\theta.\end{aligned}$$

Now, our objective is to set this integral to equate (or be proportional to) another of the form

$$\int_{\Theta}\mathcal{N}\left(\theta|...\right)g\left(y\right)d\theta=g\left(y\right)\int_{\Theta}\mathcal{N}\left(\theta|...\right)d\theta=g\left(y\right),$$

and finally to see that $g\left(y\right)\propto\mathcal{N}\left(y|...\right)$. We then have

$$\begin{aligned}p\left(y|x,D\right)&\propto\int_{\Theta}\exp\left\{-\frac{a}{2}\left(y^{2}-2\left(x^{T}\theta\right)y+\left(x^{T}\theta\right)^{2}\right)-\frac{1}{2}\left(\theta^{T}Q\theta-2\theta^{T}Q\mu\right)\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[ay^{2}-2ax^{T}\theta y+a\theta^{T}xx^{T}\theta+\theta^{T}Q\theta-2\theta^{T}Q\mu\right]\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\theta^{T}\left(axx^{T}+Q\right)\theta-2\theta^{T}\left(xya+Q\mu\right)+ay^{2}\right]\right\}d\theta.\end{aligned}$$

Again, we want to match to something of the form $\left(\theta-m\right)^{T}L\left(\theta-m\right)=\theta^{T}L\theta-2\theta^{T}Lm+m^{T}Lm$, so

$$L=axx^{T}+Q$$

and

$$Lm=xya+Q\mu\iff m=L^{-1}\left(xya+Q\mu\right),$$

assuming L^{-1} exists for now. Then:

$$\begin{aligned}p\left(y|x,D\right)&\propto\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\theta^{T}\left(axx^{T}+Q\right)\theta-2\theta^{T}\left(xya+Q\mu\right)+ay^{2}\right]\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\theta^{T}L\theta-2\theta^{T}Lm+m^{T}Lm-m^{T}Lm+ay^{2}\right]\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\left(\theta-m\right)^{T}L\left(\theta-m\right)-m^{T}Lm+ay^{2}\right]\right\}d\theta.\end{aligned}$$

Notice here that m is independent of θ so that we have

$$\begin{aligned}p\left(y|x,D\right)&\propto\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\left(\theta-m\right)^{T}L\left(\theta-m\right)-m^{T}Lm+ay^{2}\right]\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\left(\theta-m\right)^{T}L\left(\theta-m\right)\right]\right\}\exp\left\{-\frac{1}{2}\left\{ay^{2}-m^{T}Lm\right\}\right\}d\theta\\&=\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\left(\theta-m\right)^{T}L\left(\theta-m\right)\right]\right\}g\left(y\right)d\theta\\&=g\left(y\right)\int_{\Theta}\exp\left\{-\frac{1}{2}\left[\left(\theta-m\right)^{T}L\left(\theta-m\right)\right]\right\}d\theta\\&\propto g\left(y\right)=\exp\left\{-\frac{1}{2}\left\{ay^{2}-m^{T}Lm\right\}\right\}.\end{aligned}$$

And now... we complete squares again :D

$$\begin{aligned} m^T L m =& (ayx + Q\mu)^T L^{-1} \cancel{L L^{-1}} (ayx + Q\mu) \\ =& ayx^T L^{-1} ayx + 2ayx^T L^{-1} Q\mu + \cancel{\mu^T Q^T L^{-1} Q\mu} \quad \text{indep of } y \\ =& \left(a^2 x^T L^{-1} x\right) y^2 + 2\left(ax^T L^{-1} Q\mu\right) y + const. \end{aligned}$$

So

$$ay^2 - m^T L m = \left(a - a^2 x^T L^{-1} x\right) y^2 - 2\left(ax^T L^{-1} Q\mu\right) y + const$$

If $g(y)$ is a Gaussian, then this should look something like

$$\lambda (y - u)^2 = \lambda y^2 - 2\lambda u y + \lambda u^2,$$

so

$$\lambda = a - a^2 x^T L^{-1} x$$

and

$$\lambda u = a x^T L^{-1} Q\mu,$$

so

$$u = \frac{1}{\lambda} a x^T L^{-1} Q\mu.$$

And then, we have

$$\begin{aligned} p(y|x, D) \propto & g(y) \\ \propto & \exp\left\{-\frac{\lambda}{2}(y - u)^2\right\}, \end{aligned}$$

so we have that $p(y|x, D) = \mathcal{N}\left(y|u, \frac{1}{\lambda}\right)$.
Not only this, but[5]

$$u = \mu^T x$$

and

$$\frac{1}{\lambda} = \frac{1}{a} + x^T Q^{-1} x.$$

We note now that the predictive distribution's mean prediction equals the point-prediction of the MAP. However, the variance of the prediction does depend on x, which is good, since the unvertainty of out predictions depends on how far observed samples are:

- If observed samples are near from our new inputs, then we should be more certain.
- If they are far, then we should be less certain.

[5] The derivation for these values is a bit involved. It can be consulted in https://www.youtube.com/watch?v=LCISTY9S6SQ&t=287s.

3 Clustering

The goal of clustering is to partition a dataset into groups called **clusters**, in such a way that observations in the same cluster tend to be more similar than observations in different clusters. The input data is embedded in a d-dimensional space with a similarity/dissimilarity function defined among elements in the space, which should capture relatedness among elements in this space. Two elements are understood to be related when they are close in the space. Thus, a cluster is a compact group that is separated from other groups or elements outside the cluster.
There is a large variety of clustering algorithms, such as hierarchical bottom-up or top-down clustering, probabilistic clustering, possibilistic clustering, algorithmic clustering, sprectral clustering or density-based clustering.
The problem of clustering is quite complex, as if we have N data points which we want to separate into K clusters, then there are

$$S\left(N,K\right)=\frac{1}{K!}\sum_{i=1}^{K}\left(-1\right)^{i}\binom{K}{i}\left(K-i\right)^{N}$$

possibilities. This is the **stirling number of the second king**.
If in addition we don't know how many clusters we want to use, we have to add all possible $K=1,...,N$, summing up to

$$B\left(N\right)=\sum_{K=1}^{N}S\left(N,K\right)$$

possibilites. This number is the **Bell number**, which is really gigantic[6].

3.1 k-Means

The k-Means clustering algorithm takes a dataset $D=\{x_1,...,x_n\}$ and an integer $k>1$ as input, and separates D into k disjoint clusters. It is a **representative-based clustering**, meaning that each cluster is represented by one single point. In the case of k-means, the representative is the **cluster center**, $\mu_k\in\mathbb{R}^d, k=1,...,K$, i.e., the average of all the points in that cluster. Each point in thus assigned to its closest representative point. If C_k is the k-th cluster, then we consider it a better cluster when the value

$$\sum_{x\in C_k}\left\|x-\mu_k\right\|^2$$

is smaller.
Now, let's formalize all this. First, we introduce an indicator variable

$$r_{ik}=\begin{cases}1 & if\ x_i\in C_k\\ 0 & otherwise\end{cases},$$

and the objective function

$$\mathcal{J}\left(\mu,r\right)=\sum_{k=1}^{K}\sum_{i=1}^{n}r_{ij}\left\|x_i-\mu_k\right\|^2,$$

which we aim to minimize by selecting appropriate $\{\mu_k\}_k$ and $\{r_{ik}\}_{ik}$. The issue is that this problem is NP-hard, so we use an heuristic method that is only guaranteed to find local minima. This method relies in two facts:

1. For fixed cluster centers μ_k, it is easy to optimize cluster assignments r_{ik}.

   Proof. Assume fixed $\{\mu_k\}_k$, then we assign x_i to the closest μ_k, because if we assign it to a different center, μ_j, then we can minimize the sum as

   $$\left\|x_i-\mu_j\right\|^2>\left\|x_i-\mu_k\right\|^2.$$

   □

[6] For example, $B\left(71\right)\approx 4\times 10^{71}$.

2. For fixed cluster assignments r_{ik}, it is easy to optimize cluster centers μ_k.

   Proof. Assume fixed $\{r_{ij}\}_{ij}$, then

$$\begin{aligned}
\frac{\partial}{\partial \mu_j}\mathcal{J}\left(\mu_1,...,\mu_K\right) &= \sum_{k=1}^{K}\sum_{i=1}^{n}\frac{\partial}{\partial \mu_j} r_{ij}\left\|x_i-\mu_k\right\|^2 \\
&\overset{\frac{\partial}{\partial \mu_j}\mu_k=0,\forall k\neq j}{=} \sum_{i=1}^{n} r_{ij}\frac{\partial}{\partial \mu_j}\left(x_i-\mu_j\right)^T\left(x_i-\mu_j\right) \\
&= \sum_{i=1}^{n} r_{ij}\frac{\partial}{\partial \mu_j}\left(x_i^T x_i - 2\mu_j^T x_i + \mu_j^T\mu_j\right) \\
&= \sum_{i=1}^{n} r_{ij}\left(-2x_i+2\mu_j\right) \\
&= -2\sum_{i=1}^{n} r_{ij}x_i + 2\mu_j\sum_{i=1}^{n} r_{ij}.
\end{aligned}$$

   Thus, the minimum is obtained at

$$\mu_j = \frac{\sum_{i=1}^{n} r_{ij}x_i}{\sum_{i=1}^{n} r_{ij}} = \frac{\sum_{x\in C_j} x}{card\left(C_j\right)}.$$

   This is the average of the points of the cluster. □

The pseudocode is illustrated in Algorithm 3.

```
Initialize cluster centers μ_1,...,μ_K

repeat until convergence
  assign each point to the cluster with closest center
```

$$r_{ik} = \begin{cases} 1 & if\ \ k = \arg\min_j ||x_i-\mu_j||^2 \\ 0 & otherwise \end{cases}$$

```
  recompute cluster centers for all k=1,...,K
```

$$\mu_k = \frac{\sum_{x\in C_j} x}{card\left(C_j\right)}$$

Algorithm 3: k-Means.

The characteristics of k-Means are:

Advantages :

(a) Easy implementation.
(b) Fast, even for large datasets.

Limitations :

(a) Can converge to local minimum.
(b) Needs the number of clusters, K, as input.
(c) Hard cluster assignments: meaning that each point corresponds to a single cluster, which may not always be what we want.
(d) Sensitive to ourliers and clusters of different sizes and densities.
(e) Sensitive to initialization, so it is usual to run it many times and keep the best run.
(f) Biased towards rounded clusters, because it uses the Euclidean distance.

3.2 k-Means++

k-Means++ is a variant of k-Means that uses a heuristic for initializing cluster centers as in Algorithm 4.

```
choose first center μ_1 uniformly at random from all available examples

for k=2,...,K do
  choose next center μ_k at random, with probability proportional to ||x_i − μ_l||
  here, μ_l is the closest center picked so far

proceed with standard k-Means
```

Algorithm 4: k-Means++.

3.3 Choosing the number of cluster K

The number of clusters is a hyper-parameter that has to be set by the user, and there is no obvious way to choose an optimal K, since it may not even exist. This is due to the fact that there is no such thing as a true clustering against to compare. Nevertheless, there are reasonable cluster quality criteria, that can be used to select K. These criteria measure a balance between separation of clusters and their compactness. Depending on the problem, one criterion or another should be chosen.

3.3.1 Calinski-Harabasz index

The **CH index** uses Euclidean distances to measure cluster quality, so it is usually used with k-means. It measures the ratio between:

- Separation of cluster centers: sum of distances of cluster centers to the overall mean.
- Cluster compactness: sum of distances from each point to its assigned cluster center.

Thus, it is:

$$CH = \frac{N-K}{K-1} \frac{\sum_{k=1}^{K} n_k \left\|\mu_k - \overline{x}\right\|^2}{\sum_{k=1}^{K} \sum_{i=1}^{n} r_{ik} \left\|x_i - \mu_k\right\|^2},$$

where $\overline{x} = \frac{\sum x_i}{n}$. Notice that the quantities are normalized by $\frac{N-K}{K-1}$ to avoid larger K having better values. The usual approach is run k-Means with different values of K, and then select the K that maximizes the index.

Example 3.1. A k-Means example in Matlab.

k-Means algorithm example

Introduce the data

```
X1 = [2,10];
X2 = [2,5];
X3 = [8,4];
X4 = [5,8];
X5 = [7,5];
X6 = [6,4];
X7 = [1,2];
X8 = [4,9];

D = [X1;X2;X3;X4;X5;X6;X7;X8];
Y = [X5; X6; X8];
```

Compute the distance between all points

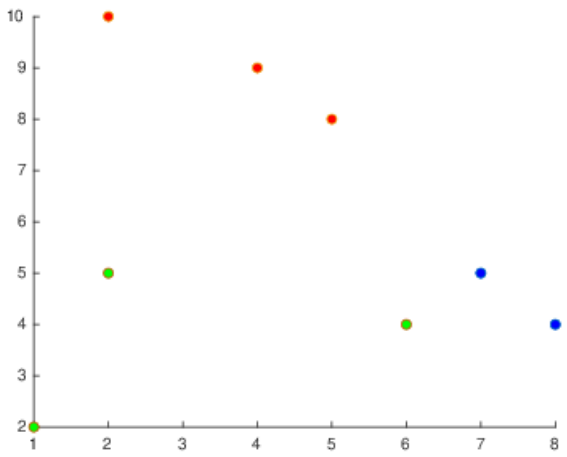
```
dist = zeros(3);

for j=1:3
    for i=1:8
        dist(i,j)=distance(D(i,:),Y(j,:),2);
    end
end
```

Select, for each point, the point that minimizes the distance

```
[v, idx] = min(dist, [], 2);
D = [D,idx];
```

Plot the clusters:

```
c1 = D(:,3) == 1;
c2 = D(:,3) == 2;
c3 = D(:,3) == 3;
scatter(D(c1,1),D(c1,2), 'MarkerFaceColor', 'b');
hold on
scatter(D(c2,1),D(c2,2), 'MarkerFaceColor', 'g');
scatter(D(c3,1),D(c3,2), 'MarkerFaceColor', 'r');
hold off
```



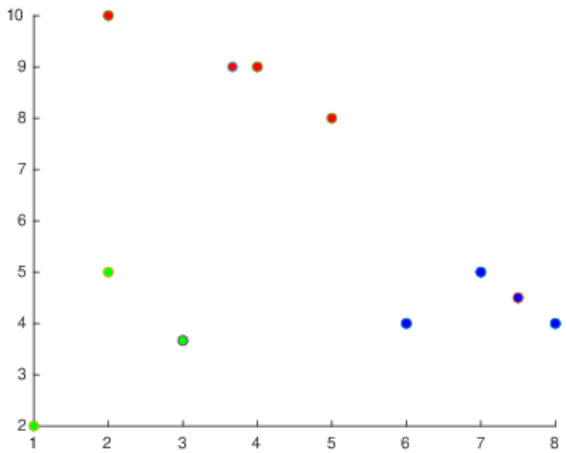
Compute the new `Y`:

```
for i=1:3
    ci = D(:,3) == i;
    x = mean(D(ci,1));
    y = mean(D(ci,2));
    Y(i,:)=[x,y];
end
Y
```

```
Y = 3x2
    7.5000    4.5000
    3.0000    3.6667
    3.6667    9.0000
```

Repeat the process:

```
%Distances
for j=1:3
    for i=1:8
        dist(i,j)=distance(D(i,1:2),Y(j,:),2);
    end
end
%Assign
[v, idx] = min(dist, [], 2);
D(:,3) = idx;
%Plot
c1 = D(:,3) == 1;
c2 = D(:,3) == 2;
c3 = D(:,3) == 3;
scatter(D(c1,1),D(c1,2), 'MarkerFaceColor', 'b');
hold on
scatter(Y(1,1),Y(1,2), 'MarkerFaceColor', 'b');
scatter(D(c2,1),D(c2,2), 'MarkerFaceColor', 'g');
scatter(Y(2,1),Y(2,2), 'MarkerFaceColor', 'g');
scatter(D(c3,1),D(c3,2), 'MarkerFaceColor', 'r');
scatter(Y(3,1),Y(3,2), 'MarkerFaceColor', 'r');
hold off
```



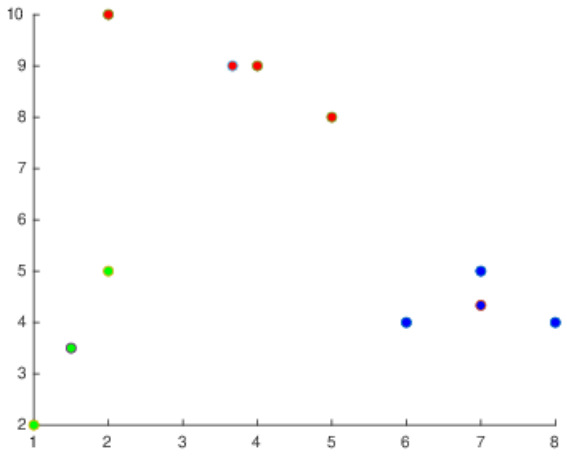

```
%Optimize
for i=1:3
    ci = D(:,3) == i;
    x = mean(D(ci,1));
    y = mean(D(ci,2));
    Y(i,:)=[x,y];
```

```
end
Y
```

```
Y = 3x2
    7.0000    4.3333
    1.5000    3.5000
    3.6667    9.0000
```

Again:

```
%Distances
for j=1:3
    for i=1:8
        dist(i,j)=distance(D(i,1:2),Y(j,:),2);
    end
end
%Assign
[v, idx] = min(dist, [], 2);
D(:,3) = idx;
%Plot
c1 = D(:,3) == 1;
c2 = D(:,3) == 2;
c3 = D(:,3) == 3;
scatter(D(c1,1),D(c1,2), 'MarkerFaceColor', 'b');
hold on
scatter(Y(1,1),Y(1,2), 'MarkerFaceColor', 'b');
scatter(D(c2,1),D(c2,2), 'MarkerFaceColor', 'g');
scatter(Y(2,1),Y(2,2), 'MarkerFaceColor', 'g');
scatter(D(c3,1),D(c3,2), 'MarkerFaceColor', 'r');
scatter(Y(3,1),Y(3,2), 'MarkerFaceColor', 'r');
hold off
```



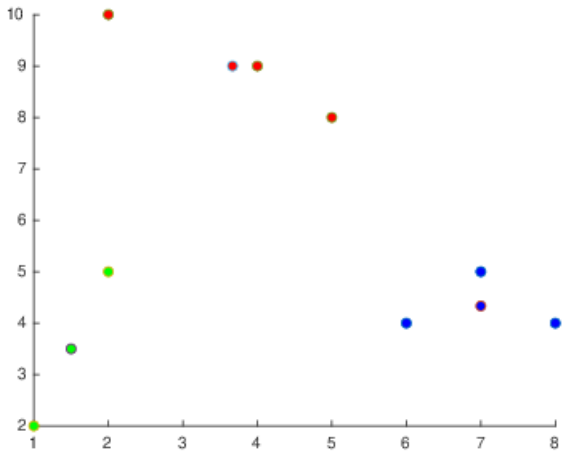

```
%Optimize
for i=1:3
    ci = D(:,3) == i;
    x = mean(D(ci,1));
    y = mean(D(ci,2));
```

```
    Y(i,:)=[x,y];
end
Y
```

```
Y = 3x2
    7.0000    4.3333
    1.5000    3.5000
    3.6667    9.0000
```

At this point we see how the clusters are the ones that we see naturally with our eyes. If we execute it again, we can see that the changes are very slight now:

```
%Distances
for j=1:3
    for i=1:8
        dist(i,j)=distance(D(i,1:2),Y(j,:),2);
    end
end
%Assign
[v, idx] = min(dist, [], 2);
D(:,3) = idx;
%Plot
c1 = D(:,3) == 1;
c2 = D(:,3) == 2;
c3 = D(:,3) == 3;
scatter(D(c1,1),D(c1,2), 'MarkerFaceColor', 'b');
hold on
scatter(Y(1,1),Y(1,2), 'MarkerFaceColor', 'b');
scatter(D(c2,1),D(c2,2), 'MarkerFaceColor', 'g');
scatter(Y(2,1),Y(2,2), 'MarkerFaceColor', 'g');
scatter(D(c3,1),D(c3,2), 'MarkerFaceColor', 'r');
scatter(Y(3,1),Y(3,2), 'MarkerFaceColor', 'r');
hold off
```



```
%Optimize
for i=1:3
    ci = D(:,3) == i;
    x = mean(D(ci,1));
    y = mean(D(ci,2));
    Y(i,:)=[x,y];
end
Y
```

```
Y = 3x2
    7.0000    4.3333
    1.5000    3.5000
    3.6667    9.0000
```

In fact, Y is not changing anymore!

Functions

```
function d = distance(X, Y, m)
    n = length(X);
    d = 0;
    for i=1:n
        d = d + (X(i)-Y(i))^m;
    end
    d = sqrt(d);
end
```

3.4 Gaussian Mixtures

A **mixture of Gaussians** is a distributions that is built using a convex sum of Gaussians, making it more flexible than a single Gaussian distribution. If the **components** of the mixture are $\mathcal{N}(\mu_k, \Sigma_k), k = 1, ..., K$ and $\pi_k, k = 1, ..., K$ are the **mixing coefficients**, with $0 \leq \pi_k \leq 1, \sum_k \pi_k = 1$, then, the density function of the mixture is given by

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k),$$

where $\theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1,...,K}$ represents the parameters of the distribution.
The key assumption is that each data point has been generated from only one component, we just don't know which one.
In Figure 9, we can see different Gaussian Mixtures that arise from the same components, choosing different mixing coefficients.

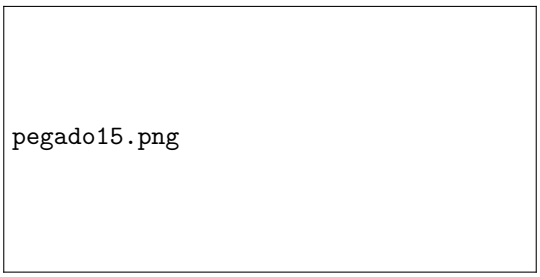


Figure 9: Different Gaussian mixtures with the same components.

3.4.1 Clustering with a Gaussian mixture

The idea is to identify each component with a cluster, so we want to determine the component from which each point in the dataset is more likely to have been created, as well as the parameters of each of the distribution. Thus, to cluster a dataset $D = \{x_1, ..., x_n\}$ into K clusters, the approach is the following:

1. Use Expectation-Maximization (EM) to estimate the mixture, obtaining approximations $\hat{\pi}_k, \hat{\mu}_k$ and $\hat{\Sigma}_k$ for each $k = 1, ..., K$.
2. Find assignments for each x_i to the clusters.

In this case, the clustering is **soft**, in opposition to the hard clustering of k-Means. This means that we will obtain the probability for each point belonging to each cluster.

3.4.2 A generative mixture of Gaussians

To sample from a mixture of Gaussians, we use a **generative model** that uses a latent variable $z = (z_1, ..., z_K)$ whose components are all 0, except one which denotes the component from which we sample, and we do:

1. Pick component k with probability π_k. This means that we set $z_k = 1$ with probability π_k.
2. Generate a sample x according to $\mathcal{N}(\mu_k, \Sigma_k)$.

The probability of generating a sample x using this generative model is

$$p(x) = \sum_z p(x, z) = \sum_k p(x, z_k = 1) = \sum_k p(x|z_k = 1)\, p(z_k = 1) = \sum_k \pi_k \mathcal{N}(x; \mu_k, \Sigma_k),$$

the joint distribution of x and z is given by

$$p(x, z) = \prod_k \pi_k^{z_k} \mathcal{N}(x; \mu_k, \Sigma_k)^{z_k},$$

and the marginal distribution over x is

$$\begin{aligned} p(x) &= \sum_k \pi_k \mathcal{N}(x; \mu_k, \Sigma_k) \\ &= \sum_z p(x, z) = \sum_z \prod_{k'} \pi_{k'}^{z_{k'}} \mathcal{N}(x; \mu_{k'}, \Sigma_{k'})^{z_{k'}}. \end{aligned}$$

Therefore, we can use Bayes to compute the conditional distribution of z given x:

$$\begin{aligned} p(z_k = 1|x) &= \frac{p(x|z_k = 1)\, p(z_k = 1)}{p(x)} \\ &= \frac{p(x|z_k = 1)\, p(z_k = 1)}{\sum_{k'} \pi_{k'} \mathcal{N}(x; \mu_{k'}, \Sigma_{k'})} \\ &= \frac{\pi_k \mathcal{N}(x; \mu_k, \Sigma_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(x; \mu_{k'}, \Sigma_{k'})} \qquad = \gamma_k(x). \end{aligned}$$

The quantity $\gamma_k(x)$ indicates how probable it is that a particular data point x has been generated by the mixture component k. Or, in the context of clustering: *how probable it is that x belongs to cluster k.* We use these quantities as the **soft membership** to each cluster. If a hard membserhip is needed, then we assign x to cluster j, where $j = \arg\max_{k'} \gamma_{k'}(x)$.

3.4.3 Learning Gaussian mixtures with Expectation-Maximization

We have a dataset of unlabelled observations $D = \{x_1, ..., x_n\}$ and we want to model it as a Gaussian mixture, with unknown parameters $\theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1,...,K}$, with a fixed K. For this we use the maximum likelihood

approach. First, we compute the loglikelihood of θ:

$$\begin{aligned} l(\theta) = \log \mathcal{L}(\theta) &= \log \prod_{i=1}^{n} p(x_i;\theta) \\ &= \log \prod_i \sum_k \pi_k \mathcal{N}(x;\mu_k,\Sigma_k) \\ &= \sum_i \log \sum_k \pi_k \mathcal{N}(x;\mu_k,\Sigma_k). \end{aligned}$$

This is hard to optimize... so we use the Expectation-Maximization approach. First, we can differenciate it to see what conditions must hold for local maxima, with $\frac{\partial}{\partial \mu_k} l(\theta) = 0$ leading to

$$\hat{\mu}_k = \frac{\sum_i \gamma_k(x_i)\, x_i}{\sum_i \gamma_k(x_i)} = \frac{\sum_i p(z_k = 1|x_i)\, x_i}{\sum_i p(z_k = 1|x_i)},$$

which is a weighted average of the points in our data, with weights being the soft assignments of each point to cluster k.
The problem now, is we cannot know $\gamma_k(x)$ without μ_k, Σ_k and π_k. Now, $\frac{\partial}{\partial \Sigma_k} l(\theta) = 0$ gives us

$$\hat{\Sigma}_k = \frac{\sum_i \gamma_k(x_i)(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{\sum_i \gamma_k(x_i)} = \frac{\sum_i p(z_k = 1|x_i)(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{\sum_i p(z_k = 1|x_i)},$$

which is the sample covariance matrix of all x_i, weigthed by the soft assignments of each point to cluster k. We have the same problem!
Since we have the constraint $\sum_k \pi_k = 1$, we now maximize the Lagrangian

$$\mathcal{L} = l(\theta) - \lambda \left(\sum_k \pi_k - 1 \right),$$

obtaining

$$\hat{\pi}_k = \frac{1}{n} \sum_i \gamma_k(x_i),$$

which is the average of all soft assignments for each point x. Again the same problem is present.
Therefore, we are in a situation in which we can estimate π_k, Σ_k and μ_k if we know γ_k, and we can compute γ_k from the estimates $\hat{\pi}_k, \hat{\Sigma}_k$ and $\hat{\mu}_k$; and we can use this in our benefit by following the pseudocode depicted in Algorithm 5. Commonly, the initializations are done using the result of k-Means in the following manner:

- Run k-Means with $k = K$.
- Set $\hat{\mu}_k$ to the mean of cluster k.
- Set $\hat{\Sigma}_k$ to the sample covariance of cluster k.
- Set $\hat{\pi}_k$ as the fraction of examples assigned to cluster k.

Example 3.2. An example of the EM algorithm using Matlab

Gaussian Mixture: EM algorithm

1- Create the mixture

```
mu = [1; 9];
sigma = zeros(1,1,2);
sigma(1,1,1) = 0.5;
sigma(1,1,2) = 0.5;

x = linspace(-2,15,300)';
p1 = [0.3,0.7];
```

Initialize parameters $\hat{\pi}_k, \hat{\Sigma}_k, \hat{\mu}_k$

repeat until convergence

E-step: recompute soft assignments $\gamma_k(x_i)$

$$\gamma_k(x_i) = \frac{\pi_k \mathcal{N}(x_i; \mu_k, \Sigma_k)}{\sum_{k'} \pi_{k'} \mathcal{N}(x_i; \mu_{k'}, \Sigma_{k'})}$$

M-step: recompute ML estimates

$$\hat{\mu}_k = \frac{\sum_i \gamma_k(x_i) x_i}{\sum_i \gamma_k(x_i)}$$

$$\hat{\Sigma}_k = \frac{\sum_i \gamma_k(x_i)(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{\sum_i \gamma_k(x_i)}$$

$$\hat{\pi}_k = \frac{1}{n} \sum_i \gamma_k(x_i)$$

Algorithm 5: EM algorithm.

```
M1 = gmdistribution(mu,sigma,p1);
MM1 = M1.pdf(x);
```

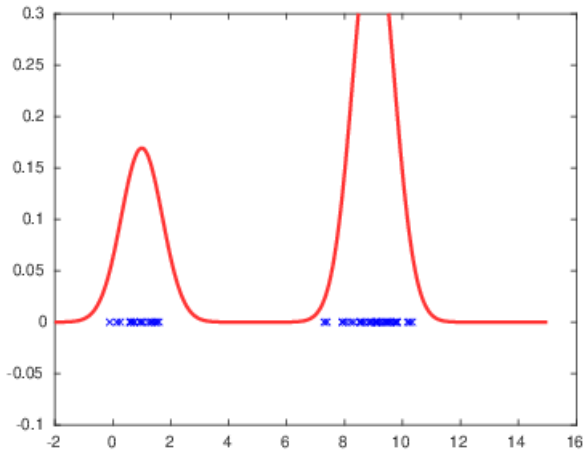
2- Sample the mixture

```
rng(3)
Y = random(M1,50)'
```

```
Y = 1x50
    8.2075    9.8171    0.1442    8.8333    9.4084    9.0987    0.7269    1.0454   -0.1108    9.4905
```

3- See what we got

```
figure(1)
plot(x,MM1,'LineWidth',2,'Color',[1 0.2 0.2 1])
hold on
ylim([-0.1 0.3])
scatter(Y,zeros(size(Y)), 'xb')
```



4- Initialize variables.

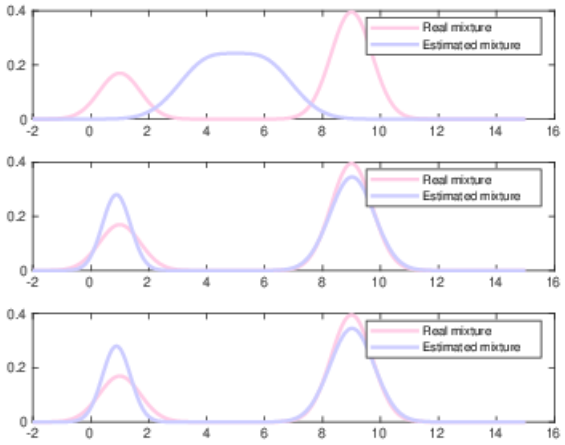
```
muhat = [4; 6];
sigmahat = zeros(1,1,2);
sigmahat(1,1,1) = 1;
sigmahat(1,1,2) = 1;
phat = [0.5 0.5]
```

```
phat = 1x2
    0.5000    0.5000
```

5- Iterate the E-step and M-step (see functions below)

```
niter = 6;
rem = 3;

figure(2)

subplot(niter/rem + 1, 1, 1)
Mhat = gmdistribution(muhat,sigmahat,phat);
MMhat = Mhat.pdf(x);
plot(x,MM1,'LineWidth',2,'Color',[1 0.8 0.9 1])
hold on
plot(x,MMhat, 'LineWidth', 2, 'Color', [0.8 0.8 1 1])
legend('Real mixture','Estimated mixture')
hold off

for i=1:niter
    g = e_step(Y,muhat,sigmahat,phat);
    [muhat,sigmahat,phat] = m_step(g,Y);

    Mhat = gmdistribution(muhat,sigmahat,phat);
    MMhat = Mhat.pdf(x);

    if mod(i,rem) == 0
        subplot(niter/rem+1,1,i/rem+1)
        plot(x,MM1,'LineWidth',2,'Color',[1 0.8 0.9 1])
```

```
        hold on
        plot(x,MMhat, 'LineWidth', 2, 'Color', [0.8 0.8 1 1])
        legend('Real mixture','Estimated mixture')
        hold off
    end
end
```




```
muhat
```

```
muhat = 2x1
    0.8838
    9.0175
```

```
sigmahat(1,1,:)
```

```
ans =
ans(:,:,1) =

    0.2352


ans(:,:,2) =

    0.5825
```

```
phat
```

```
phat = 1x2
    0.3400    0.6600
```

```
function g = e_step(Y,muhat,sigmahat,pihat)
    g = zeros(length(Y),length(muhat));
    denom = zeros(length(Y),1);
    for k=1:length(muhat)
        g(:,k) = pihat(k)*normpdf(Y(:),muhat(k),sigmahat(1,1,k));
        denom = denom + pihat(k)*normpdf(Y(:),muhat(k),sigmahat(1,1,k));
    end
    g = g./denom;
end

function [muhat,sigmahat,pihat] = m_step(g,Y)
    s = size(g);
    muhat = zeros(s(2),1);
    for k=1:s(2)
        muhat(k) = Y*g(:,k) / sum(g(:,k));
    end

    sigmahat = zeros(1,1,s(2));
    for k=1:s(2)
        sigmahat(1,1,k) = (Y-muhat(k)).^2*g(:,k) / sum(g(:,k));
    end

    pihat = zeros(1,s(2));
    for k=1:s(2)
        pihat(k) = sum(g(:,k)) / length(Y);
    end
end
```

3.4.4 Special cases

The shape of the Gaussians can be restricted, obtaining special cases of mixtures:

- No restrictions on Σ_k: the general case, in which each cluster can have general Gaussian shape.
- Σ_k are diagonal: each Gaussian component is forced to have no correlation among input dimensions.
- $\Sigma_k = \sigma^2 I$ are isotropic or spherical: each Gaussian component is forced to be spherical, so no correlation among input variables and same scaling across each input variable.

4 Linear Classifiers

In classification, we have a **labelled** dataset, $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ where $x_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y} = \{l_1, ..., l_K\}$ are **labels**. Thus, the tuple (x_i, y_i) means that the vector x_i is associated to the label y_i. With this setup, we aim at producing a **classification model**, meaning a function that, given a new input x', predicts the label y' that it should be associated with. Usually, we distinguish between two kinds of classification, attending to the number of labels considered:

- In **binary classification**, there are only two possible labels, $|\mathcal{Y}| = 2$.
- In **multi-class classification**, there are more than two labels, $|\mathcal{Y}| > 2$.

Now, we introduce some useful terms:

Definition 4.1. The **decision regions** are a partition of the feature space, $\{P_1, ..., P_K\} \subset \mathbb{R}^d$, such that $\bigcap_{j=1}^{K} P_j = \emptyset$, $\bigcup_{j=1}^{K} P_j = \mathbb{R}^d$. Intuitively, they are the regions in which we divide the feature space, so that all elements inside a region have the same label.
The **decision boundaries** are the points in the frontier between decision regions.
A **classifier** can then be understood as the creation of the decision regions.
A **linear classifier** is a classifier in which the decision boundaries are $d-1$-dimensional hyperplanes.

Example 4.1. A visualization.
In Figure 10, we can see different classification models and the regions they generate for the iris dataset. The left model correspond to a linear classifier, and we can see how the decision boundaries are linear. The other two models are not linear.

pegado16.png	pegado17.png	pegado18.png

Figure 10: Different classifiers. Linear (left). Quadratic (middle). knn (right).

Many useful classifiers don't just predict the input's class, but also the probabilities of the input belonging to each class. This is desirable, as it enables us to express uncertainty about the prediction that we make. For example, in binary classification it is a common approach to map the target labels to be 0 or 1, and then make predictions as a continuous value in $[0, 1]$. More explicitly, we can have labels $\mathcal{Y} = \{'sick', 'healthy'\}$ and encode $'sick' = 1, 'healthy' = 0$, so that given a patient x', we obtain a prediction $y' \in [0, 1]$, indicating the probability of the patient being sick.
In classification with $K > 2$ classes, it is common to encode the labels using **one-hot encoding**, meaning that we map the labels into the set $\{0, 1\}^K$. For example, if we have three labels, then they are encoded as $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$. When in this scenario, predictions are usually points in the $(K-1)$-simplex, i.e., a prediction $y' = (y'_1, ..., y'_K)$ must be $0 \leq y'_k \leq 1, \forall k$ and $\sum_k y'_k = 1$. Each y'_k represents the probability of the input belonging to class k.

4.1 Decision boundary in probabilistic models

From a probabilistic perspective, we can think of the joint probability of examples and labels, $p(x, y)$. When we are building a classifier, we then want to minimize the expected loss, or expected error. Note, nonetheless, that loss here is a bit different from the regression loss. A natural way to think about this loss is through loss or cost matrices. A cost matrix is a table as the following:

Real / Predicted	$y = l_1$	$y = l_2$	...	$y = l_K$
$y' = l_1$	0	$c_{1,2}$	...	$c_{1,K}$
$y' = l_2$	$c_{2,1}$	0	...	$c_{2,K}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$y' = l_K$	$c_{K,1}$	$c_{K,2}$	...	0

This matrix indicates the cost of each error. Note that not all errors need to have the same cost. For example, in a medical context, it has a higher cost to predict that a sick patient is healthy (this person could potentially die), than to predict that a healthy person is sick (in which case further tests would probably correct the mistake). A cost matrix for this case could look like the following:

Real / Predicted	$y = healthy$	$y = sick$
$y' = healthy$	0	60
$y' = sick$	10	0

We will focus in the case of all errors having the same impact, which is called the 0-1 loss.
Let's now see how, given a new example x, we can use a 'rule' to choose the label that x should have. We have random variables X and Y with joint distribution $p(X, Y)$. We can compute the expected loss of assigning the label $c \in \mathcal{Y}$ to x. For this, we first define the loss function as the 0-1 loss:

$$L(a, b) = \begin{cases} 0 & if\ a == b \\ 1 & otherwise \end{cases}.$$

Therefore:

$$\begin{aligned} E_Y[L(Y, c)] &= \sum_{y \in \mathcal{Y}} L(y, c)\, p(Y = y|x) \\ &= \sum_{y \neq c} p(Y = y|x) \\ &= p(Y \neq c|x) \\ &= 1 - p(Y = c|x). \end{aligned}$$

Of course, we want to minimize the expected loss, so we aim at predicting the class y' minimizing $E_Y[L(Y, y')]$:

$$y' = \arg\min_y E_Y[L(Y, y)] = \arg\min_y \{1 - p(y|x)\} = \arg\max_y p(y|x).$$

This is called the **Bayes classifier**, and it is optimal when we use 0-1 loss. Its error is given by the so-called **Bayes error rate**:

$$BER = 1 - E_X[p(y'|x)] = 1 - \int_x p(y'|x)\, p(x)\, dx = 1 - \int_x p(x|y')\, p(y')\, dx.$$

Of course, we can use this classifier to partition the feature space into regions $\mathcal{R}_c, c \in \mathcal{Y}$, and we can compute the BER summing over all regions:

$$BER = 1 - \sum_c \int_{x \in \mathcal{R}_c} p(x|c)\, p(c)\, dx.$$

Before, we claimed that the Bayes classifier is optimal. However, in practice we don't know the distribution $p(y, x)$, so it cannot be implemented exactly. Therefore, $p(y|x)$ is estimated from data, and this estimates are used for classification, incurring in additional errors. To learn $p(y|x)$, there are two basic approaches, namely discrimintative classifiers and generative classifiers.

4.2 Generative classifiers

Generative classifiers learn $p(y|x)$ through the Bayes rule.

4.2.1 Discriminant analysis

Discriminant analysis is the result of implementing a Bayes classifier assuming that the class-conditional distributions $p(x|y)$ are gaussian. This means that, having $\mathcal{Y} = \{c_1, ..., c_K\}$, then it is

$$p(x|y = c_k) \sim \mathcal{N}(\mu_k, \Sigma_k).$$

If we also assume that the prior distributions are

$$p\left(y=c_k\right)=\pi_k,$$

with $\sum_k \pi_k = 1$, then we define the **discriminant functions**

$$\begin{aligned} g_k\left(x\right) &= \log\left(P\left(y=c_k\right)P\left(x|y=x_k\right)\right) \\ &= \log\left(\pi_k \cdot \frac{1}{\det\left(\Sigma_k\right)^{\frac{1}{2}}\left(2\pi\right)^{\frac{d}{2}}} \exp\left\{-\frac{1}{2}\left(x-\mu_k\right)^T\Sigma^{-1}\left(x-\mu_k\right)\right\}\right) \\ &= \log\pi_k - \frac{1}{2}\log\left(\det\Sigma_k\right) - \frac{d}{2}\log\left(2\pi\right) - \frac{1}{2}\left(x-\mu_k\right)^T\Sigma^{-1}\left(x-\mu_k\right) \\ &= \log\pi_k - \frac{1}{2}\left[\log\left(\det\Sigma_k + \left(x-\mu_k\right)^T\Sigma_k^{-1}\left(x-\mu_k\right)\right)\right] + const. \end{aligned}$$

This is a **quadratic discriminant function**, and the corresponding classifier is implemented by predicting

$$y' = c_{k'},\ where\ k' = \arg\max_k g_k\left(x\right).$$

This corresponds to chossing the label with maximum probability a posteriori.
The **decision boundaries** in this case are those regions in which there exist k_1, k_2 with

$$g_{k_1}\left(x\right) = g_{k_2}\left(x\right).$$

These corresponds to hyper-quadrics in the feature space, and this is a quadratic method, usually called quadratic discriminant analysis (QDA).
Of course, we can further simplify our assumptions, by assuming that all labels have the same covariance matrix, $\Sigma_k = \Sigma$ for all $k = 1, ..., K$. In this simpler case, the discriminant functions end up being

$$g_k\left(x\right) = \log\pi_k + \mu_k^T\Sigma^{-1}x - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k,$$

because now $\det\Sigma_k = \det\Sigma$ is constant for all k, so we can remove it. Furthermore, the term $x^T\Sigma_k^{-1}x = x^T\Sigma^{-1}x$ is also constant with respect to k, so it will not affect the k chosen. Therefore, we end up with **linear discriminant functions**, in which the **decision boundaries** correspond to hyperplanes in the feature space. This is a linear method, usually called linear discriminant analysis (LDA).
In Figure 10, the left diagram corresponds to a LDA partitioning of the feature space for the iris dataset, while the one in the center is a QDA partitioning.

Further assumptions

Of course, we can make more simplifying assumptions for our model, such as:

- $\Sigma = diag\left(\sigma_1^2, ..., \sigma_d^2\right)$ is diagonal. In this case, we obtain

$$g_k\left(x\right) = \log\pi_k - \frac{1}{2}\sum_{j=1}^{d}\frac{\left(\mu_{kj}-x_j\right)^2}{\sigma_j^2}.$$

- Σ is an isotropic Gaussian, i.e., $\Sigma = \sigma^2 I$. In this case,

$$g_k\left(x\right) = \log\pi_k - \frac{1}{2\sigma^2}\left\|\mu_k - x\right\|^2.$$

- $\pi_k = \frac{1}{K}, \forall k = 1, ..., K$, all priors are equal. In this case

$$g_k\left(x\right) = -\frac{1}{2}\left\|\mu_k - x\right\|^2.$$

Distance-based learning perspective

In all seen cases, we have a minimum-distance classifier in $\mathbb{R}^d$:

- The general QDA case corresponds to using different Mahalanobis distance from x to each class center μ_k.
- The LDA case uses the same Mahalanobis distance from x to each class center μ_k.
- In the case of all covariance matrix being equal and diagonal, the distance is the weighted Euclidean distance.
- In the isotropic Gaussians case, the distance corresponds to the usual Euclidean distance.

Implementation

It is usual to use MLE and estimate the centers and covariance matrices using the training dataset. If we define the sets

$$S_k = \{x_i | y_i = c_k, (x_i, y_i) \in \mathcal{D}\}$$

and $n_k = card\,(S_k)$, then, the estimates are

$$\hat{\pi}_k = \frac{n_k}{n},\ \hat{\mu}_k = \frac{1}{n_k} \sum_{x \in S_k} x,$$

and the covariance matrix is:

- In QDA:
$$\hat{\Sigma}_k = \frac{1}{n_k - 1} \sum_{x \in S_k} (x - \mu_k)\,(x - \mu_k)^T\,.$$
- In LDA
$$\hat{\Sigma} = \sum_{k=1}^{K} \frac{n_k - 1}{n - n_k} \hat{\Sigma}_k.$$

Final remarks on discriminant analysis

Bayesian classifiers are optimal when the class-conditional densities and priors are known. This means that, if know the underlying distributions, then these classifiers are our best choice. Of course, this is not realistic, and estimations need to be made. However, normal distributions appear in a wide range of real scenarios, and even if we have to estimate the centers and covariance matrices, QDA and LDA are a very good choice when data resembles Gaussian distributions. In addition, they are well-principled, having a solid mathematical theory behind them, they are fast and reliable.
Of course, if the real distribution is very far from being a Gaussian, then the model obtained will be poor, so one should take care of this.
Also, it is important to ensure that we are correctly estimating the parameters of the Gaussians, because otherwise the model will not work, not even with underlying Gaussian data.
And it is clear that once we are relying on sample estimates instead of population parameters, we loose the optimality of the method.
In practice, it is really hard to assess which assumptions hold and which ones do not, so we can be limited to use a trial and error approach.

4.2.2 Regularized discriminant analysis (RDA)

When data is scarce, some problems can arise while using discriminant analysis. For example:

- If there are more dimensions than samples with some label, i.e. $d > n_k$, for some k, then QDA cannot be applied, because $\hat{\Sigma}_k$ is singular. The reason is that each of the sample is adding a rank-1 matrix. Therefore, to get a rank-d matrix, we need at least n_k samples. Not only this, but in fact, as we are subtracting the sample mean, the last value is linearly dependant of the previous $n-1$ values. Therefore, we need at least $n_k + 1$ samples to be able to construct a rank-d covariance matrix.

- If $d > n$, then we cannot use QDA nor LDA, as all covariance matrices are singular.

RDA computes the covariance matrices as

$$\hat{\Sigma}_k(\alpha) = \alpha\hat{\Sigma}_k + (1-\alpha)\hat{\Sigma},$$

where $\alpha \in [0,1]$ is the regularization parameter. The method is QDA when $\alpha = 1$ and is LDA when $\alpha = 0$. In any other case, it is something in between.
A further way to regularize the matrices is by

$$\hat{\Sigma}_k(\alpha,\gamma) = (1-\gamma)\hat{\Sigma}_k(\alpha) + \gamma\hat{\sigma}^2 I,$$

where $\hat{\sigma}^2 = \frac{Tr\left[\hat{\Sigma}_k(\alpha)\right]}{d}$, and the diagonal term improves the well-conditioning of the method.

4.3 Naïve Bayes

The Naïve Bayes Classifier is a Bayesian classifier that assumes that the features are pair-wise independent in the class-conditional distribution. This means that the probability can be written as

$$p(x|y) = \prod_{j=1}^{d} p(x_j|y).$$

This assumptions does not hold in general, but this approach can provide a good approximation in many cases. Also, it is practical, as the amount of parameters to estimate is small.
As before, we classify the input record x in the class c_k that maximizes the discriminant function:

$$g_k(x) = \log\pi_k + \sum_{j=1}^{K} \log p(x_j|y=c_k).$$

Therefore, we need to

1. Estimate the class priors as the sample frequency, $\pi_k = \frac{n_k}{n}$.
2. Estimate the class-conditional densities for each input feature independently.

Naïve Bayes can also be used in the case of categorical variables. To model binary features, we can use, for example, the Bernoulli distribution

$$P(x|p) = p^x(1-p)^{1-x},$$

where $x \in \{0,1\}$ and $p \in [0,1]$ is the probability of the event happening. For a binary feature, we would need to estimate K parameters, one for each class, so that

$$P(x|y=c_k) = p_k^x(1-p_k)^{1-x}.$$

If we had all our features as binary, then the discriminant functions would be

$$\begin{aligned} g_k(x) &= \log\pi_k + \sum_{j=1}^{d} \log P(x_j|y=c_k) \\ &= \log\pi_k + \sum_{j=1}^{d} \left[x_j \log p_{k,j} + (1-x_j)\log(1-p_{k,j})\right], \end{aligned}$$

where $p_{k,j}$ is the Bernoulli parameter for the j^{th} feature and the k^{th} class. Note that this is a linear function with respect to to x.
If instead we have categorical features with more values, we can then use the Categorical distribution

$$g_k(x) = \log\pi_k + \sum_{j=1}^{d}\sum_{v} [x_j = v] \log p_{k,j,v},$$

where $[expr]$ is 1 if $expr$ is True and 0 otherwise, and $p_{k,j,v}$ is the Categorical parameter for the value v of the j^{th} feature and the k^{th} class.
Now, we need to estimate these parameters, for which we can use the sample frequencies. Note how 0-frequencies can be a problem, so it is a common approach to utilize **Laplace smoothing**

$$\hat{p}\left(v|y=c_k\right)=\frac{n_{k,v}+p}{n_k+pV}.$$

Here:

- $p \in \mathbb{R}^+$ is a weight parameter assigned to the prior distribution of observing values v. It is typically set to 1, just to avoid 0 values.
- V is the number of modalities (number of distinct values) of the feature modelled.

Example 4.2. Naïve Bayes in MATLAB

Naïve Bayes Classifier Example

```
Outlook = categorical({'Sunny'; 'Sunny'; 'Overcast'; 'Rainy'; 'Rainy'; 'Rainy'; 'Overcast'; 'Sunny';
Temperature = categorical({'Hot'; 'Hot'; 'Hot'; 'Mild'; 'Cool'; 'Cool'; 'Cool'; 'Mild'; 'Cool'; 'Mild
Humidity = categorical({'High'; 'High'; 'High'; 'High'; 'Normal'; 'Normal'; 'Normal'; 'High'; 'Normal
Wind = categorical({'Weak'; 'Strong'; 'Weak'; 'Weak'; 'Weak'; 'Strong'; 'Strong'; 'Weak'; 'Weak'; 'We
PlayBall = categorical({'No'; 'No'; 'Yes'; 'Yes'; 'Yes'; 'No'; 'Yes'; 'No'; 'Yes'; 'Yes'; 'Yes'; 'Yes
dataset = table(Outlook, Temperature, Humidity, Wind, PlayBall);
dataset
```

	Outlook	Temperature	Humidity	Wind	PlayBall
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rainy	Mild	High	Weak	Yes
5	Rainy	Cool	Normal	Weak	Yes
6	Rainy	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rainy	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rainy	Mild	High	Strong	No

We compute the priors:

```
num_yes = sum(dataset.PlayBall == "Yes");
num_no = sum(dataset.PlayBall =="No");
n = num_yes + num_no;
prior_yes = num_yes / n;
prior_no = num_no / n;
```

We create a new record $x = (Sunny, Hot, Normal, Weak)$ and predict its class using our functions:

```
new_record = table(categorical({'Sunny'}), categorical({'Hot'}), categorical({'Normal'}), categorical
                   'VariableNames', {'Outlook', 'Temperature', 'Humidity', 'Wind'});
[predicted_class, conf] = naive_bayes_classify(new_record, {'Outlook', 'Temperature', 'Humidity', 'Wi
fprintf('Predicted class for new record: %s with confidence %f\n', char(predicted_class), conf);
```

```
Predicted class for new record: Yes with confidence 0.672948
```

```
[predicted_class, conf] = naive_bayes_classify(new_record, {'Outlook', 'Temperature', 'Humidity', 'Wi
fprintf('Predicted class for new record using Laplace Smoothing: %s with confidence %f\n', char(predi
```

```
Predicted class for new record using Laplace Smoothing: Yes with confidence 0.634538
```

Now with the built-in classifier:

```
X = table(Outlook, Temperature, Humidity, Wind);
Y = PlayBall;

mdl = fitcnb(X, Y, 'ClassNames', {'Yes', 'No'});
[predicted_class, conf, cost] = predict(mdl, new_record);

% Display the predicted class and confidence
fprintf('Predicted class for new record: %s with confidence %f\n', char(predicted_class(1)), conf(1))
```

```
Predicted class for new record: Yes with confidence 0.664913
```

```
function prob = cond_prob(data, feature, target_class, value)
    subset = data(data.PlayBall == target_class, :);
    count = sum(subset.(feature) == value);
    prob = count / size(subset, 1);
end

function prob = laplace_sm(data, feature, target_class, value, p)
    subset = data(data.PlayBall == target_class, :);
    count = sum(subset.(feature) == value);
    modalities = numel(unique(subset.(feature)));
    prob = (count + p) / (size(subset, 1) + p*modalities);
end

function [predicted_class, conf] = naive_bayes_classify(record, features, training_data, priors, lap)
    classes = categorical({'Yes', 'No'});
    probs = zeros(size(classes));

    for c = 1:length(classes)
        p = priors(c);

        for f = 1:length(features)
            feature = features{f};
            value = record.(feature);
            if lap == 0
                p = p * cond_prob(training_data, feature, classes(c), value);
            else
                p = p * laplace_sm(training_data, feature, classes(c), value, lap);
```

```
            end
        end

        probs(c) = p;
    end

    [~, idx] = max(probs);
    predicted_class = classes(idx);
    conf = probs(idx) / sum(probs);
end
```

4.3.1 Gaussian Naïve Bayes

If we have numericla features, the usual approach is to assume they follow Gaussian distributions, then we estimate their mean and variance using MLE. If all features are assumed Gaussian, this approach is equivalent to QDA with diagonal covariance matrices.
Other approaches are:

- Discretize numerical values and proceed with Categorical NB.
- Assume a different distribution and estimate its sample parameters from data.

Note that when the data is mixed, we can assume a different distribution for each feature, and then add the log-likelihoods altogether.

4.4 Perceptron and Logistic Regression

The **perceptron** is a mathematical model of the functioning of a neuron in our brain. In Figure 11 we can see a diagram of a neuron. Other neurons transmit signals into our neuron via the dendrites, then 'something' happens inside the neuron, and it transmits or not signals through its axon towards the neurons to which it is connected.

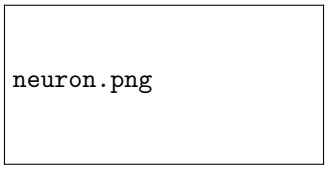


Figure 11: A neuron.

This behavior was mathematically modeled by Roosenblatt in his pioneer paper [4]. He did this with the concept of perceptron:

- A perceptron is a function $F : \mathbb{R}^d \to \{-1, 1\}$, where
  - d is the number of inputs. In the neuron analogy, it is the number of dendrites of the neuron.
  - $\{-1, 1\}$ represents that a neuron can send a signal, $F = 1$, or not, $F = -1$.
- This function operates in the following maner:
  - For each input x_i, there is a weight w_i, for $i = 1, ..., d$. This models somehow the strength of the connection between two neurons.
  - There is an aritificial input called **bias**, which is always set to 1 and serves the purpose of being able to center the inputs. It is usually depicted by $x_0 = 1$ and goes with its weight w_0, used to determine the true bias.
  - The inputs are weighted and sum, to form a state of the neuron

$$S = \sum_{i=0}^{d} w_i x_i = W^T X.$$

– Finally, we apply the **activation function**

$$f(S) = \begin{cases} 1 & if\ S > 0 \\ -1 & otherwise \end{cases}.$$

Therefore,

$$F(x) = f(S(x)).$$

This is depicted in Figure 12. Notice

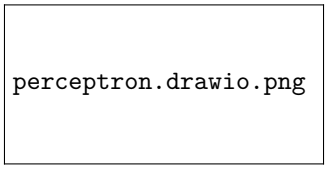


Figure 12: A simple perceptron.

This way, we can classify the input variables $X \in \mathbb{R}^d$ into two classes, 1 or -1. But... what are the weights?

Example 4.3. Imagine we have the following classification function

$$class(x) = \begin{cases} 1 & if\ x > 0.5 \\ -1 & otherwise \end{cases},$$

then it is easy to construct a perceptron that can classify the instances. We have:

- $d = 1$.
- $w_0 = 0.5, w_1 = 1$.
- Therefore, we have

$$F(X) = f(w_0 + w_1 x) = f(x - 0.5) = \begin{cases} 1 & if\ x - 0.5 > 0 \\ -1 & otherwise \end{cases} = \begin{cases} 1 & if\ x > 0.5 \\ -1 & otherwise \end{cases}.$$

And we are good!

In this previous example we can grab some intuition on how the weights are chosen, but doing this by hand is not scalable at all. When we have several input variables, this process is complicated quite a lot. Luckily, the **perceptron algorithm** helps us estimate the weigths.
This algorithm is an on-line algorithm, meaning that it process one training example at a time, updating W incrementally. The training examples are pairs (X, y), where $X \in \mathbb{R}^d$ and $y \in \{-1, 1\}$. Also, we can scale all X to lie in the unit sphere, because this also scales the hyperplanes of classification and classes remain the same. The algorithm goes as follows:
The updates are made like that because we can consider the error function

$$E(W) = -W^T X y$$

only when X is misclasified. In that case, $W^T X y < 0$ and thus the minus sign. We want to minimize this error, for which we follow a gradient descent approach (we will deepen on this later), so we update W as

$$W' = W - \eta \nabla E = W + \eta X y.$$

The basic idea is that if we now utilize this new weights in the same input, we would get

$$W'^T X = (W + \eta X y)^T X = W^T X + \eta y X^T X.$$

Now, say $y = 1$, then it is

$$W'^T X = W^T X + \eta \|X\|^2 > W^T X,$$

```
init W=0

for i in range(1,e):
  for each (X,y):

    compute prediction=F(S(W^T X))

    if prediction != y:
      # update W
      W = W + lr*X*y
return W
```

Algorithm 6: Perceptron algorithm (input X, classes y, learning rate lr, epochs e) -> weights W

so we have made the input to be closer to be positive, and thus correctly classified. If $y = -1$ the same logic applies.
In fact, under some conditions, convergence is ensured:

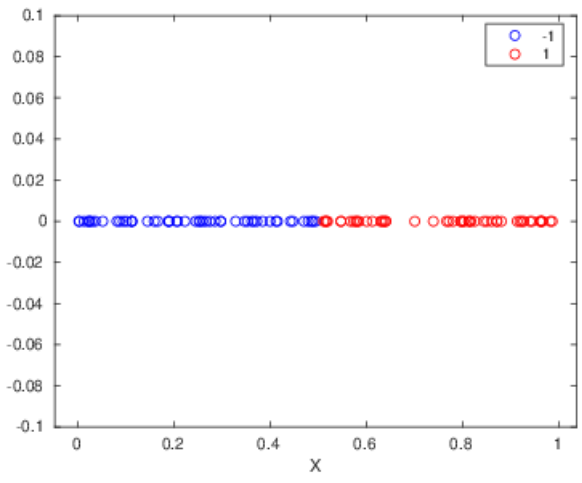
Theorem 4.1. ***Perceptron convergence theorem***
For any finite set of linearly separable labeled examples, the Perceptron Learning Algorithm will halt after a finite number of iterations. In other words, after a finite number of iterations, the algorithm yields a vector W that classifies perfectly all the examples.

The proof can be read in https://web.mit.edu/course/other/i2course/www/vision_and_learning/perceptron_notes.pdf.

Example 4.4. A simple perceptron in MATLAB

Perceptron Example

```
rng(5)
X = rand([100,1]);
y = 2*(double(X>0.5)-0.5*ones(size(X)));

f=gscatter(X,zeros(size(X)),y,'br','o');
hold on
ylim([-0.1,0.1])
hold off
```

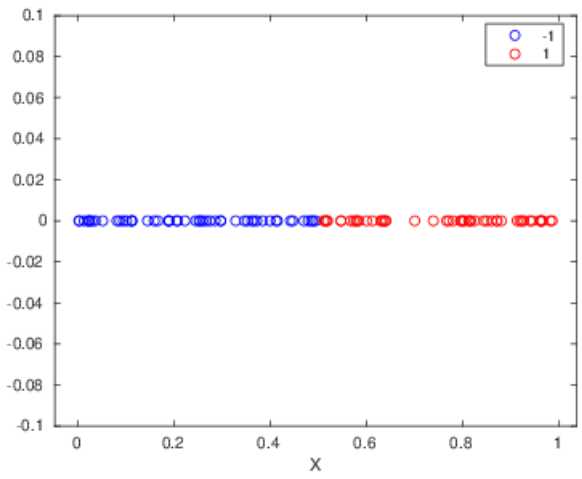


Now, let's manually use our perceptron:

```
W = [-0.5; 1]
```

```
W = 2x1
   -0.5000
    1.0000
```

```
% Add the bias
X_bias = [ones(size(X)) X];

pred = zeros(size(X));

for i=1:length(X)
    x = X_bias(i,:)';
    S = state(x,W);
    pred(i) = my_sign(S);
end

f2=gscatter(X,zeros(size(X)),pred,'br','o');
hold on
ylim([-0.1,0.1])
hold off
```

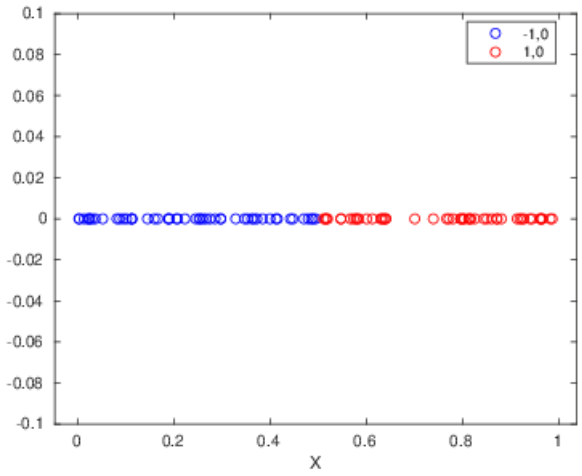


It classified perfectly! Now let's train the algorithm:

```
% Add the bias
X_bias = [ones(size(X)) X];

% Train the model
W2 = train(X_bias,y,0.1,10)
```

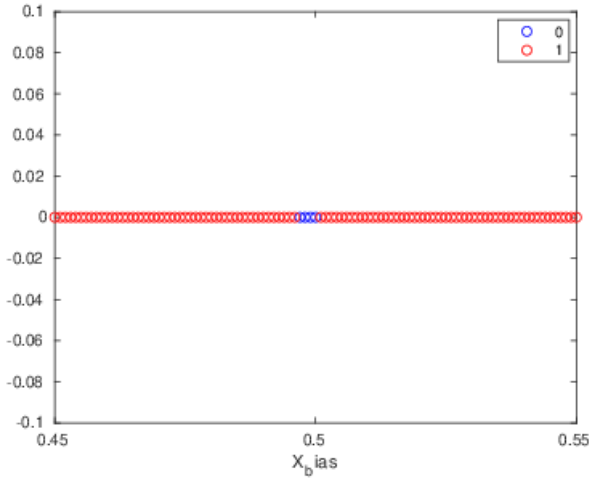
```
W2 = 2x1
   -0.1000
    0.2013
```

```
pred2 = zeros(size(X_bias));

for i=1:length(X)
    x = X_bias(i,:)';
    S = state(x,W2);
    pred2(i) = my_sign(S);
end

f3=gscatter(X,zeros(size(X)),pred2,'br','o');
hold on
ylim([-0.1,0.1])
hold off
```



This is quite interesting... it found different weights, but it classified eveything well. This must break somewhere. Let's try to determine where:

```
X = 0.45:0.001:0.55;
X = X';
y = 2*(double(X>0.5)-0.5*ones(size(X)));

X_bias = [ones(size(X)) X];

pred3 = zeros(size(X));

for i=1:length(X)
    x = X_bias(i,:)';
    S = state(x,W2);
    pred3(i) = my_sign(S);
end

f4=gscatter(X_bias,zeros(size(X)),pred3==y,'br','o');
hold on
ylim([-0.1,0.1])
xlim([0.45,0.55])
hold off
```




```
y(pred3~=y)
```

```
ans = 4x1
    -1
    -1
    -1
    -1
```

This means that our model is actually setting the bound a little bit before 0.5 (as can be seen in the graph). This should be improved by increasing the size of X.

```
function S = state(X, W)
    S = W'*X;
end

function pred = my_sign(S)
    if S > 0
        pred = 1;
    else
        pred = -1;
    end
end

function W = train(X, y, lr, epochs)
    [~, d] = size(X);
    W = zeros(d, 1);
    for epoch = 1:epochs
        for i = 1:length(X)
            x = X(i,:)';
            pred = my_sign(state(x, W));
            if pred ~= y(i)
                W = W + lr*x*y(i);
            end
        end
    end
```

```
end
```

We can improve the approach by choosing a different activation function, which would be differentiable ideally. A usual alternative is the **logistic function**

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

which maps $\mathbb{R} \to [0,1]$, so that this input can be interpreted as a probability, allowing us to represent uncertainty in the prediction. This function has the following properties:

- It verifies a pseudo-symmetry
$$\sigma(-z) = 1 - \sigma(z).$$
- It is differentiable, with derivative
$$\sigma'(z) = \sigma(z)(1-\sigma(z)).$$
- Its inverse is the logit function
$$logit(p) = \log\left(\frac{p}{1-p}\right).$$

When we use the logistic function, we are performing what is called logistic regression for binary classification. Again, we have a dataset of pairs (X_i, y_i) where $X_i \in \mathbb{R}^d$ and $y_i \in [0,1]$, associating label $y_i = 1$ with a positive example, and $y_i = 0$ with a negative example. In logistic regression we model

$$P(y=1|x) = \sigma\left(W^T X\right),$$

where, again, W is the weight vector. Therefore, we are modelling $y_i \sim Ber(p_i)$, where $p_i = \sigma\left(W^T X_i\right)$. Notice that we don't assume anything about the distribution of X, though.

Why is logistic regression a linear classifier?

When we classify, we need to set a threshold above which we classify the label to be 1. Usually, this threshold is set to be 0.5. Now,

$$\sigma(z) > 0.5 \iff \frac{1}{1-e^{-z}} > 0.5 \iff 1 > 0.5\left(1-e^{-z}\right) \iff 2 > 1 - e^{-z} \iff 1 > e^{-z}$$

$$\iff 0 > -z \iff z > 0.$$

And remember that $z = W^T X$, so, in fact, we are classifying using the same criteria as before.
At this point, we can try to use maximum likelihood in the search of an optimal W:

$$\begin{aligned}\mathcal{L}(W) &= \prod_{i=1}^{n} P(y_i|X_i, W) = \prod_{i=1}^{n} Ber\left(y_i|\hat{y}_i = \sigma\left(W^T X_i\right)\right) \\ &= \prod_{i=1}^{n} \hat{y}_i^{y_i}\left(1-\hat{y}_i\right)^{1-y_i} \\ &= \prod_{i=1}^{n} \sigma\left(W^T X_i\right)^{y_i}\left(1-\sigma\left(W^T X_i\right)\right)^{1-y_i}.\end{aligned}$$

As usual, we take the log-likelihood, but in this case we minimize the negative log-likelihood instead, which can be interpreted as an error function:

$$\begin{aligned}E(W) = -\log\mathcal{L}(W) &= -\sum_{i=1}^{n} \log P(y_i|X_i, W) \\ &= -\sum_{i=1}^{n} y_i \log\hat{y}_i + (1-y_i)\log\left(1-\hat{y}_i\right).\end{aligned}$$

This error expression is known as **log loss** or **binary cross-entropy**, and it is widely used in classification schemes.
Now, as we aim to find its minimum, we compute its gradient. For this, recall that

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)),$$

and notice that

$$\frac{\partial W^T X_i}{\partial W} = X_i.$$

Recall also that

$$(f \circ g)'(x) = [g(f(x))]' = g'(f(x)) \cdot f'(x).$$

Now, let's go for this derivative:

$$\begin{aligned}
\nabla E(W) = &\frac{\partial E(W)}{\partial W} = -\sum_{i=1}^{n} \frac{\partial y_i \log \sigma\left(W^T X_i\right)}{\partial W} + \frac{\partial (1 - y_i) \log\left(1 - \sigma\left(W^T X_i\right)\right)}{\partial W} \\
= &-\sum_{i=1}^{n} \frac{y_i}{\cancel{\sigma\left(W^T X_i\right)}} \cancel{\sigma\left(W^T X_i\right)} \left(1 - \sigma\left(W^T X_i\right)\right) X_i - \frac{1 - y_i}{\cancel{1 - \sigma\left(W^T X_i\right)}} \sigma\left(W^T X_i\right) \cancel{\left(1 - \sigma\left(W^T X_i\right)\right)} X_i \\
= &\sum_{i=1}^{n} (1 - y_i)\, \sigma\left(W^T X_i\right) X_i - y_i \left(1 - \sigma\left(W^T X_i\right)\right) X_i \\
= &\sum_{i=1}^{n} \left[(1 - y_i)\, \sigma\left(W^T X_i\right) - y_i \left(1 - \sigma\left(W^T X_i\right)\right)\right] X_i \\
= &\sum_{i=1}^{n} \left[\sigma\left(W^T X_i\right) - \cancel{y_i \sigma\left(W^T X_i\right)} - y_i + \cancel{y_i \sigma\left(W^T X_i\right)}\right] X_i \\
= &\sum_{i=1}^{n} \left[\sigma\left(W^T X_i\right) - y_i\right] X_i \\
= &\sum_{i=1}^{n} \left[\hat{y}_i - y_i\right] X_i.
\end{aligned}$$

In this case, it is not possible to find a close-form solution, so we use iterative methods to approximate local minima. We are going to use gradient descent, which is one of the simplest approaches, but it is widely used.

4.4.1 Gradient descent

Gradient descent is a general numerical method to find local minima of a differentiable function $F(Z)$. The idea is to use the fact that the gradient of a vector function points in the direction of maximum growth, and the negative gradient points in the direction of maximum decrease. Therefore, if we 'follow' this direction, we should approach a minimum of the function, although it might not be the global minimum.
The approach works as follows. To approximate a local minimum of $F(W)$:

1. We start at a weight vector W_0 and set $k = 0$.
2. Compute $\nabla F(W_k)$.
3. Update the weight vector
$$W_{k+1} = W_k - \gamma \nabla F(W_k).$$
Also update $k = k + 1$. The parameter γ is called learning rate, and it quantifies 'how much' we advance in the direction of the gradient. It has to be chosen beforehand and it is critical, since:
   (a) If γ is too big, we might jump over the minimum and the method might never converge.
   (b) If γ is too small, the method might converge too slowly.

   Related to the learning rate is the importance of feature scaling, as the same learning rate impacts all features, so if they have different scales, the method might be good for some of them, but bad for others.
4. Repeat until convergence or maximum number of steps is reached.

4.4.2 Newton's algorithm

Newton's algorithm[7] is an algorithm used to find roots of a function which converges faster than gradient descent and does not need a learning rate parameter. As this algorithm finds roots (i.e. zeros), we don't apply directly to F, but rather to its gradient. Thus, in this case, we need F to be twice differentiable and to be able to afford computing its second derivatives, which can sometimes be costly.
In this case, the algorithm works as follows:

1. We start at a weight vector W_0 and set $k = 0$.
2. Compute the Hessian of F:
$$H\left(F\left(W\right)\right) = \nabla^2 F\left(W\right) = \begin{pmatrix} \frac{\partial^2 F(W)}{\partial W_1^2} & \frac{\partial F(W)}{\partial W_1 \partial W_2} & \cdots & \frac{\partial F(W)}{\partial W_1 \partial W_d} \\ \frac{\partial F(W)}{\partial W_2 \partial W_1} & \frac{\partial F(W)}{\partial^2 W_2} & \cdots & \frac{\partial F(W)}{\partial W_2 \partial W_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F(W)}{\partial W_d \partial W_1} & \frac{\partial F(W)}{\partial W_d \partial W_2} & \cdots & \frac{\partial F(W)}{\partial^2 W_d} \end{pmatrix}.$$
3. Update the weight vector
$$W_{k+1} = W_k - H\left(F\left(W_k\right)\right)^{-1} \nabla F\left(W_k\right),$$
and set $k = k + 1$.
4. Repeat until convergence or maximum number of steps is reached.

Remark 4.1. Notice that:

- The update rule is obtained by finding the minimum of the second order Taylor series of F around W_k.
- No learning rate is needed this time, but we need to compute the Hessian.

Iterated Reweighted Least Squares (IRLS)

When we apply Newton's method to the log-likelihood of the logistic regression, it is called the IRLS method. The gradient of the log-likelihood is
$$\nabla E\left(W\right) = \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right) X_i = X^T \left(y - \sigma\left(W^T X\right)\right)$$
and the Hessian
$$H\left(F\left(W\right)\right) = \sum_{i=1}^{n} \hat{y}_i \left(1 - \hat{y}_i\right) X_i X_i^T = X^T diag\left(\hat{y}_i \left(1 - \hat{y}_i\right)\right) X.$$

Example 4.5. Logistic regression example using MATLAB

Logistic Regression example

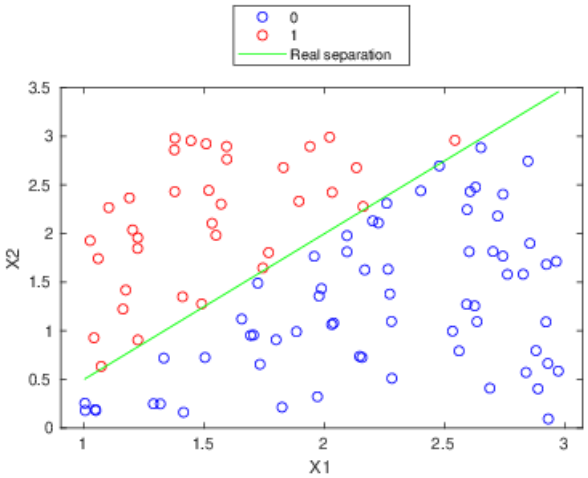
```
rng(5)
X1 = rand([100,1])*2.+1;
X2 = rand([100,1])*3.;
y = double(X2>X1.*1.5-1)
```

```
y = 100x1
     1
     0
     0
     0
```

[7] There is an infinite amount of Newton's algorithms haha, this is one of them :D

```
     0
     0
     0
     0
     1
     1
```

```
X = [X1 X2];

f=gscatter(X1,X2,y,'br','o');
hold on
plot(X1,X1.*1.5-1,'g')
legend('0','1','Real separation','Location','northoutside')
hold off
```



Let's train using gradient descent:

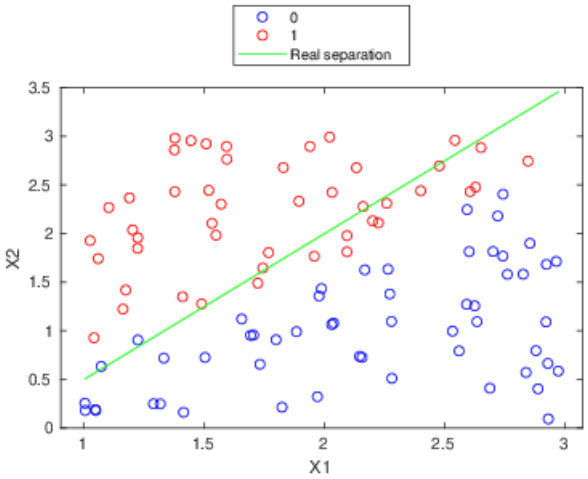
```
[n, d] = size(X);
X_bias = [ones(n,1) X]
```

```
X_bias = 100x3
    1.0000    1.4440    2.9554
    1.0000    2.7415    2.4042
    1.0000    1.4134    0.1619
    1.0000    2.8372    0.5714
    1.0000    1.9768    1.3573
    1.0000    2.2235    2.1088
    1.0000    2.5318    0.9961
    1.0000    2.0368    1.0799
    1.0000    1.5936    2.7644
    1.0000    1.3754    2.8609
```

```
[W,niter] = train_GD(X_bias,y,0.1,1e-3,20)
```

```
W = 3x1
    3.9647
  -13.2924
   13.2435

niter = 20
```

```
y_hat = my_predict(X_bias,W);

f2=gscatter(X1,X2,y_hat,'br','o');
hold on
plot(X1,X1.*1.5-1,'g')
legend('0','1','Real separation','Location','northoutside')
hold off
```

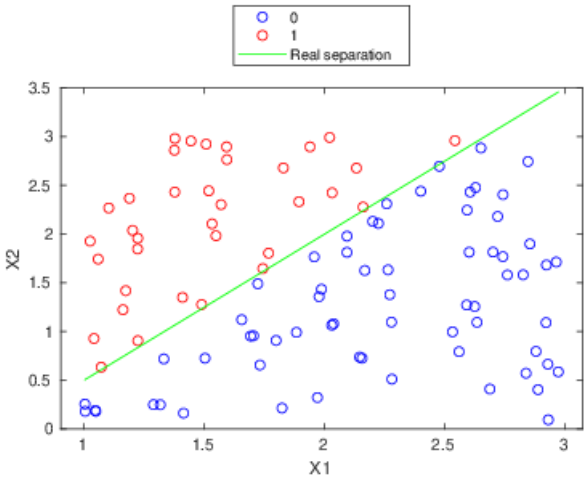


We can observe how it is quite ok... Let's increment the max_iter parameters:

```
[W,niter] = train_GD(X_bias,y,0.1,1e-3,1000)
```

```
W = 3x1
   20.9642
  -27.7530
   17.3205

niter = 1000
```

```
y_hat = my_predict(X_bias,W);

f3=gscatter(X1,X2,y_hat,'br','o');
hold on
plot(X1,X1.*1.5-1,'g')
legend('0','1','Real separation','Location','northoutside')
hold off
```



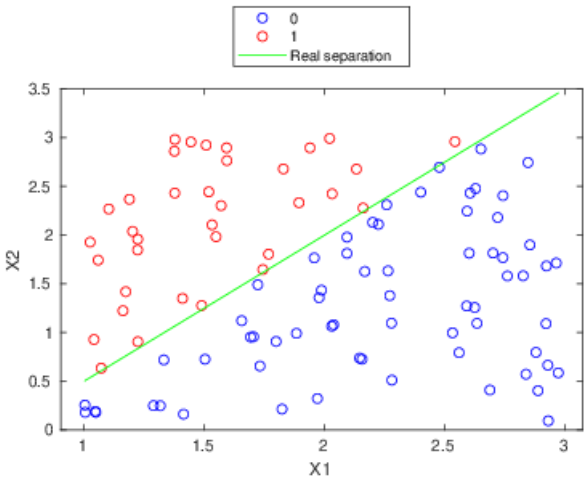
And in fact it is much better! :D

Let's see how IRLS performs:

```
[W,niter_NR] = train_NR(X_bias,y,1e-3,1000,1e-3)
```

```
W = 3x1
1.0e+05 *

    1.6554
   -2.0991
    1.2912

niter_NR = 227
```

```
y_hat = my_predict(X_bias,W);

f4=gscatter(X1,X2,y_hat,'br','o');
hold on
plot(X1,X1.*1.5-1,'g')
legend('0','1','Real separation','Location','northoutside')
hold off
```



We can see how this time we also obtain nice separation, and this one even finishes before maxiter is reached. In fact, we can try to see how many iterations would GD need:

```
[W,niter_GD] = train_GD(X_bias,y,0.1,1e-3,1000000)
```

```
W = 3x1
  110.2037
 -146.6376
   92.3106

niter_GD = 449262
```

Therefore, we obtain an enormous speedup:

```
speedup = niter_GD / niter_NR
```

```
speedup = 1.9791e+03
```

Almost 2000 times faster in terms of iterations...

```
function S = state(X, W)
    S = X*W;
end

function s=logistic(S)
    s = 1./(1+exp(-S));
end

function ds=grad_logistic(y, y_hat, X)
    ds = X' * (y_hat - y);
end

function H = hess_logistic(y_hat, X)
    D = diag(y_hat .* (1 - y_hat));
    H = X' * D * X;
end


function [W,niter]=train_GD(X, y, lr, threshold, maxiter)
    [~, d] = size(X);
    W = ones(d, 1);
    for niter=1:maxiter
        st = state(X,W);
        y_hat = logistic(st);
        ds = grad_logistic(y, y_hat, X);
        v = vecnorm(abs(ds));
        if v < threshold
            break
        end
        W = W - lr*ds;
    end
end

function [W, niter] = train_NR(X, y, threshold, maxiter, lambda)
```

```
    [~, d] = size(X);
    W = ones(d, 1);
    I = eye(d);
    for niter = 1:maxiter
        st = state(X, W);
        y_hat = logistic(st);
        gradient = grad_logistic(y, y_hat, X);
        v = vecnorm(abs(gradient));
        if v < threshold
            break
        end
        H = hess_logistic(y_hat, X);
        W = W - inv(H + lambda * I) * gradient; %We add a term to avoid singularities
    end
end

function pred = my_predict(X, W)
    p = logistic(state(X, W));
    pred = double(p >= 0.5);
end
```

4.4.3 Multi-class logistic regression

The multi-class case, $K > 2$, is handled by having a separator $W^{(k)}$ for each class $k = 1, ..., K$. In this case, instead of a Bernoulli distribution for the targets y_i, we use a categorical distribution. Each target y_i is represented with its one.hot encoding.
In this case, the likelihood function is

$$\mathcal{L}\left(W^{(1)}, ..., W^{(K)}\right) = \prod_{i=1}^{n} \prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}},$$

and the **cross-entropy loss** is

$$E\left(W^{(1)}, ..., W^{(K)}\right) = -\sum_{i=1}^{n} \sum_{k=1}^{k} y_{ik} \log \hat{y}_{ik}.$$

As in the binary case, we optimize this by using gradient descent or Newton's method.

4.4.4 Regularization

Finally, note that we can add regularization to the weights, in the same way we did for linear regression, by adding the $L1$ or $L2$ norms of the weights to the cross-entropy loss

$$E_{lasso}(W) = -\sum_{i=1}^{n} \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)\right] + \lambda \sum_{j=1}^{d} |W_j|,$$

and

$$E_{ridge}(W) = -\sum_{i=1}^{n} \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)\right] + \lambda \sum_{j=1}^{d} W_j^2.$$

Note that this is for the binary case, but for the general case is similar.

5 Nearest Neighbor Prediction

The **nearest neighbor predictor** uses the local neighborhood of an input point to compute a prediction. A well known family of this kind is the kNN models, which predicts the output of the input point by combining the known outputs of the k nearest training data points, for example by voting if we are classifying the data. The approach of these method is quite straightforward, and is detailed in Algorithm 7.

```
Training
  Store all training examples

Prediction
  Given an input X:
    1. Compute distance/similarity with all examples in the training set.
    2. Locate k closest points.
    3. Emit prediction by combining outputs of the k closest points.
```

Algorithm 7: kNN Pseudocode.

Of course, we have to decide:

- The distance/similarity function.
- How many neighbors to choose, k.
- How to combine the outputs of the k nearest neighbors to emit the final prediction.

5.1 Locality: similarities and distances

It is important to decide what distance or similarity between points means in our feature space, since this will determine the neighborhoods of the points.

Definition 5.1. A **distance function** is a function

$$d : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R},$$

verifying:

1. Non-negativity: $d(a,b) \geq 0, \forall a, b \in \mathbb{R}^d$.
2. Zero distance is equal: $d(a,b) = 0 \iff a = b, \forall a, b \in \mathbb{R}^d$.
3. Symmetric: $d(a,b) = d(b,a), \forall a, b \in \mathbb{R}^d$.
4. Triangle inequality: $d(a,b) \leq d(a,c) + d(c,b), \forall a, b, c \in \mathbb{R}^d$.

A **similarity function** is a function

$$s : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R},$$

verifying:

1. Ranged: $s(a,b) \in [-1,1], \forall a, b \in \mathbb{R}^d$ or $s(a,b) \in [0,1], \forall a, b \in \mathbb{R}^d$.
2. Completely similar is equal: $s(a,b) = 1 \iff a = b, \forall a, b \in \mathbb{R}^d$.
3. Symmetry: $s(a,b) = s(b,a), \forall a, b \in \mathbb{R}^d$.

Example 5.1. Some examples of distances are:

- The **Minkownski distance** family:

$$d(a,b) = \|a-b\|_p = \left(\sum_{i=1}^{d} |a_j - b_j|^p \right)^{\frac{1}{p}}.$$

Which has as special cases the Euclidean distance ($p = 2$) or the Manhattan distance ($p = 1$).

- The **Mahalanobis distance** is an interesting distance between points that takes into account the covariance matrix of the features, Σ. Its formula is

$$d(a, b) = (a - b)^T \Sigma^{-1} (a - b),$$

which might already be familiar to you, since we have used it for some probabilistic methods.

Some examples of similarities are:

- The **cosine similarity** function:

$$s(a, b) = \frac{a \cdot b}{\|a\|_2 \|b\|_2} = \cos(\angle a, b),$$

which leverages the dot product, $a \cdot b = a^T b = \|a\|_2 \|b\|_2 \cos(\angle a, b)$. Therefore, this captures 'how parallel' a and b are.

- The **Pearson correlation measure** is similar to cosine similarity, but centers data:

$$s(a, b) = \frac{(a - \mu)^T (b - \mu)}{\|a - \mu\|_2 \|b - \mu\|_2},$$

where μ is the mean of the points.

- The **Hamming distance** is computed between two sequences of bits, and it is just the proportion of common bits.

- The **Jaccard coefficient** is also computed between two sequences of bits, and it is

$$s(a, b) = \frac{\sum_j [a_j = b_j = 1]}{\sum_j [a_{j=1} = 1 | b_j = 1]}.$$

If we have $a = (10010000110000), b = (11000001100001)$, then

$$s_{Hamming}(a, b) = \frac{9}{14} = 0.643,$$

and

$$s_{Jaccard}(a, b) = \frac{2}{7} = 0.286.$$

5.2 Choosing k

Nearest neighbor methods are very sensitive to the chosen value of k. If it is too low, it is easy to overfit; if it is too large, we can underfit. Therefore, we need to choose it thoughfully. This will depend on the dataset, and the typical approach is to use cross-validation, or other resampling methods, seeing k as an hyperparamter that trades-off bias and variance of the resulting model.

5.3 How to combine outputs to make predictions

5.3.1 Classification

- Majority vote, broking ties randomly.
- Distance-weighted vote: a more advanced approach that weights the votes higher for closer points and lower for further points.

5.3.2 For regression

- Use the average of the outputs of the k nearest neighbors.
- Use the weighted average of the outputs of the k nearest neighbors. Again, the weights should be inversely proportional to distance or proportional to similarity.

When we weight the predictions, we can relax the constraint of using only k examples, and we can even use the whole training set for making predictions, since this approach lowers the chances of underfitting.

5.4 Decision boundaries for nearest neighbors classifier

In 1-nearest neighbor, the decision regions correspond the union of each example's Voronoi cell, with appropriate class. Given a set of points $\{p_i\}_{i \in I} \subset \mathbb{R}^d$, the Voronoi cell of point p_j corresponds to the set of points whose nearest neighbor is p_j. For example, here I show two different Voronoi diagrams, which show the Voronoi diagram of 6 different points:

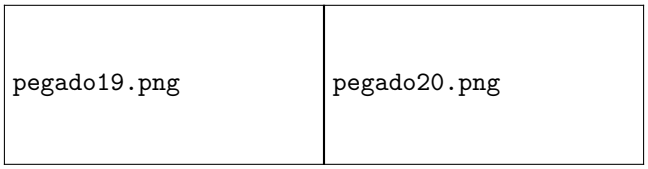


Figure 13: Voronoi diagrams. Code in MATLAB script `voronoi.mlx`.

The decision boundaries and regions are non-linear, but they get smoother as we increase the value of k. This effect can be observed in the following Figure:

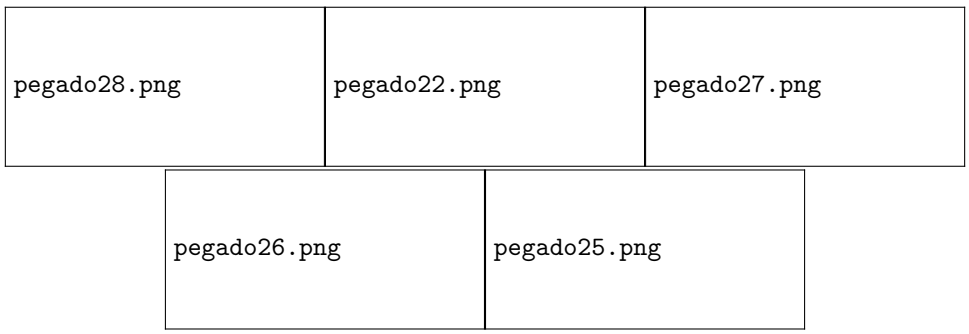


Figure 14: kNN decision regions for different values of k.

5.5 Final considerations

- Making predictions can be slow, specially if the training dataset is large. To improve the prediction speed, there are several approaches:
  - Use data structures like $kd-trees$ to speed up neighbor retrieval, although this is only good when we have a moderate amount of features. A $kd-tree$, or k-dimensional tree, is a binary search tree data structure used to organize points in k-dimensional space. In the context of kNN models, kd-trees can help speed up neighbor retrieval by partitioning the space and reducing the search area for nearest neighbors. However, the efficiency gain of kd-trees is reduced when there are many features (i.e., high-dimensional data), as the search time approaches that of a linear search.
  - Use prototypes. A protoype is a representative point or a summary of a group of points that belong to the same class. Using prototypes can reduce the size of the training dataset while still maintaining its overall structure. The idea is to replace multiple points with a single prototype, thus reducing the number of distance calculations during prediction. Examples of prototype selection methods include the nearest neighbor condensation algorithm and the edited nearest neighbor rule.

  - Use approximate neighbors, for example using locality sensitive hashing. The main idea behind LSH is to hash the input items in such a way that similar items are more likely to be hashed to the same bucket. By doing this, it reduces the search space for nearest neighbors and allows for faster retrieval. LSH trades off some accuracy for improved speed, making it suitable for large-scale applications where an approximate answer is acceptable.

- kNN is prone to overfitting. To avoid this, usual approaches include:
  - Remove noisy examples from the training data, for example, data points with neierst neighbors of different class.
  - Again, use prototypes.
  - Set the appropriate value of k.
- It suffers from the curse of dimensionality: as dimension increases, everything seems to be close.
- Suffers from the presence of irrelevant features, so feature selection is an important pre-processing step.
- Standardization of features is crucial to avoid domination of features with larger values.

6 Trees and Random Forests

An **ensemble method** is a method that combines two or more predictors, instead of using a single prediction. These are useful when we have several models that are better that just a random guess, and that are independent between them. The combination of the models can be done averaging predictions, when in regression, or by majority vote in classification.
The main types of ensemble methods are:

- Stacking: involves training a learning algorithm to combine the predictions of several other learning algorithms. First, all sub-models are trained based on the complete training set, then the meta-model is fitted based on the outputs — meta-features — of the sub-models in the ensemble. Therefore, stacking allows you to use multiple heterogeneous, possibly weak learning models and "stack" them together in a manner that allows you to use information from their predictions to make a final prediction which often has better performance.
- Bagging: is a way to decrease the variance of the prediction by generating additional data for training from dataset using combinations with repetitions to produce multi-sets of the original data. Bagging helps to avoid overfitting by averaging or voting the prediction from the multiple models. Random Forest is a classic example of bagging algorithm.
- Boosting: is a sequential technique in which the first algorithm is trained on the entire data set, and the following algorithms are built by fitting the residuals of the first algorithm, thereby giving higher weight to those observations that were poorly predicted by the previous model. It relies on creating a series of weak learners each of which might not be good at the entire problem, but might be good at recognizing some part of it, and then combining their predictions to get the final prediction. The idea is to add new models to the ensemble sequentially. At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learnt so far. Gradient Boosting, AdaBoost and XGBoost are examples of boosting algorithms.

6.1 Trees

A regression tree partitions the feature space into axis-parallel regions and predicts using the average of training points that fall into that region. We can think of a tree in terms of nodes and edges: each node in the tree specifies a test of some attribute, and each edge descending from that node corresponds to one of the possible outcomes for the test. The leaf nodes (or terminal nodes) of the tree contain an output value which is used to make a prediction. When a new data point is presented to the tree for prediction, it is routed down the tree based on the outcome of the tests in each node, starting from the root and ending at a leaf node. The value in the leaf node is then returned as the prediction.
A classification tree also partitions the feature space into axis-parallel regions, each of them representing a class. It is NP-hard to find the optimal tree of minimum size, so greedy approaches are usually used. A general approach is explained in Algorithm 8.

```
1- Feature selection
   Find the feature that best splits the data into two subsets, minimizing a impurity function.

2- Binary splitting
   Decide how to split the feature. If it is categorical, split in each possible value. If it
     is continuous, find a value that divides it in two, minimizing the impurity.

3- Recursion
   Repeat 1- and 2- until the stopping criterion is met.

4- Pruning
   To avoid overfitting.
```

Algorithm 8: Training a Tree.

We are going to use the Gini impurity metric to create trees.

Let S be a subset of example of the input data $S \subset D = \{(x_1, y_1), ..., (x_n, y_n)\}$, with $x_i \in \mathbb{R}^d$ and $y_i \in \mathcal{Y}$, with $|\mathcal{Y}| = K \geq 2$. Let also $p_k(S)$ be the proportion of examples in S that belongs to class k. Then, the Gini impurity metric is computed as

$$Gini(S) = \sum_{k=1}^{K} p_k(S)(1 - p_k(S)) = 1 - \sum_k (p_k(S))^2.$$

Therefore, to find the node to expand the tree, we need to find the pair (v^*, s^*), where v^* is a feature and $s^* \in \mathbb{R}$, such that

$$(v^*, s^*) = \arg\min_{v,s} \left[\frac{|S_{v \leq s}|}{|S|} Gini(S_{v \leq s}) + \frac{|S_{v>s}|}{|S|} Gini(S_{v>s}) \right].$$

In the case of a regression tree, the approach is similar, but instead we minimize the sum of squared errors, relative to the average target value in each induced partition. In other words, we now seek for the pair (v^*, s^*) such that

$$(v^*, s^*) = \arg\min_{v,s} [SSE(S_{v \leq s}) + SSE(S_{v>s})],$$

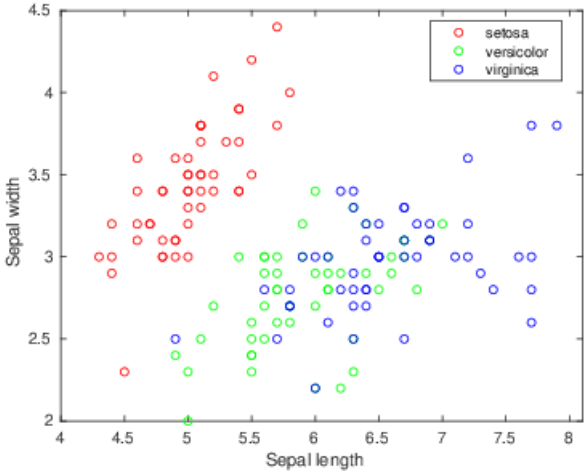
where

$$SSE(S) = \sum_{i|(x_i,y_i) \in S} (y_i - avg(S))^2.$$

Example 6.1. Classification trees in MATLAB

Classification Trees

```
load fisheriris
f = figure;
gscatter(meas(:,1), meas(:,2), species,'rgb','o',5);
xlabel('Sepal length');
ylabel('Sepal width');
```



Let's build a classification tree:

```
X = [ meas(:,1) meas(:,2)];
Y = species;

[bestFeature1, bestThreshold1, minGini1] = FindBestSplit(X,Y)
```

```
bestFeature1 = 1
bestThreshold1 = 5.4000
minGini1 = 0.4389
```

```
idx1 = X(:,bestFeature1)<=bestThreshold1;
idx2 = X(:,bestFeature1)>bestThreshold1;

X1 = X(idx1);
y1 = Y(idx1);
X2 = X(idx2);
y2 = Y(idx2);

[bestFeature11, bestThreshold11, minGini11] = FindBestSplit(X1,y1)
```

```
bestFeature11 = 1
bestThreshold11 = 4.8000
minGini11 = 0.2233
```

```
[bestFeature12, bestThreshold12, minGini12] = FindBestSplit(X2,y2)
```

```
bestFeature12 = 1
bestThreshold12 = 6.1000
minGini12 = 0.4546
```

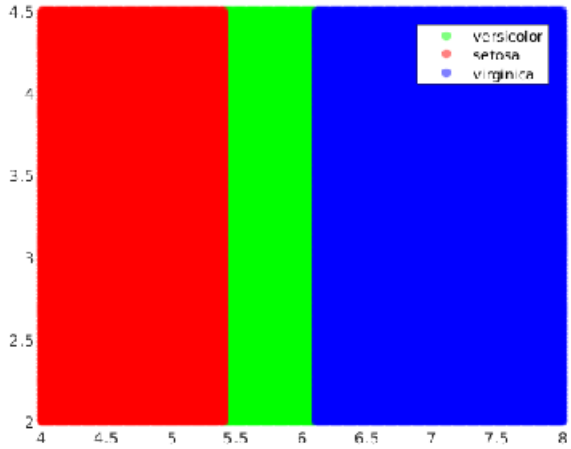
We can draw the decision regions:

```
pred = Predict([0.1,0.5], bestFeature1, bestThreshold1, bestFeature11, bestThreshold11, bestFeature12
```

```
pred =
    {'setosa'}
```

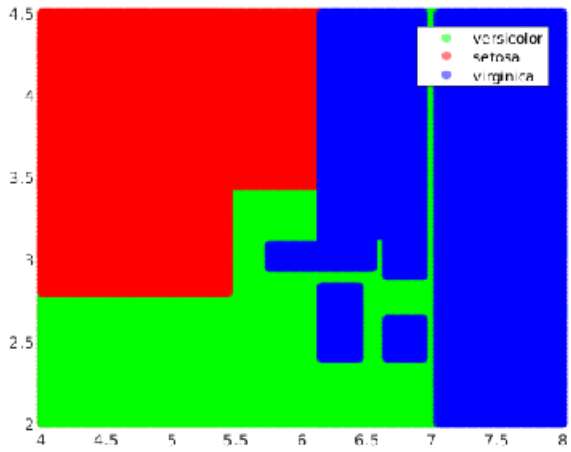
```
figure;
[x,y] = meshgrid(4:.01:8,2:.01:4.5);
x = x(:);
y = y(:);
% Predict species for each point in the grid
j = arrayfun(@(idx) Predict([x(idx), y(idx)], bestFeature1, bestThreshold1, bestFeature11, bestThresh
j = [j{:}];

colors = [0 1 0; 1 0 0; 0 0 1];
face_alpha = 0.5;
```

```
% Loop through unique species and plot points with custom colors and transparency
unique_species = {'versicolor','setosa','virginica'};
for i = 1:length(unique_species)
    scatter(x(strcmp(j, unique_species{i})), y(strcmp(j, unique_species{i})), [], colors(i,:), 'o', '
    hold on;
end
legend('versicolor','setosa','virginica')
```



As we can see, the result is not great, but this method can only divide the space in parallel hyperplanes to the axis. If we continued dividing the tree, we could improve the results. We can use the built-in trees to see what we can achieve:

```
Mdl = fitctree(X, Y, 'SplitCriterion', 'gdi'); % 'gdi' stands for Gini's diversity index
j = predict(Mdl, [x y]);

unique_species = {'versicolor','setosa','virginica'};
for i = 1:length(unique_species)
    scatter(x(strcmp(j, unique_species{i})), y(strcmp(j, unique_species{i})), [], colors(i,:), 'o', '
    hold on;
end
legend('versicolor','setosa','virginica')
```



This one seems to partition the space 'OK'.

```
function gini = GiniImpurity(y)
    K = unique(y);
    proportions = arrayfun(@(val) sum(strcmp(y,val))/numel(y), K);
    gini = 1 - sum(proportions.^2);
```

```
end

function [bestFeature, bestThreshold, minGini] = FindBestSplit(X, y)
    % Get number of instances and number of features
    [numInstances, numFeatures] = size(X);

    % Initialize bestFeature, bestThreshold, and minimum Gini impurity
    bestFeature = 0;
    bestThreshold = 0;
    minGini = Inf;

    % For each feature
    for feature = 1:numFeatures
        % For each possible threshold
        for threshold = 1:numInstances
            % Split data into two subsets
            subset1 = y(X(:, feature) <= X(threshold, feature));
            subset2 = y(X(:, feature) > X(threshold, feature));

            % Compute Gini impurities for the two subsets
            giniSubset1 = GiniImpurity(subset1);
            giniSubset2 = GiniImpurity(subset2);

            % Compute weighted Gini impurity
            gini = (numel(subset1) / numel(y)) * giniSubset1 + ...
                   (numel(subset2) / numel(y)) * giniSubset2;

            % If this split has a lower Gini impurity than the best one so far, update bestFeature, b
            if gini < minGini
                bestFeature = feature;
                bestThreshold = X(threshold, feature);
                minGini = gini;
            end
        end
    end
end

function prediction = Predict(x, bestFeature1, bestThreshold1, bestFeature11, bestThreshold11, bestFe
    [y1_numeric, group_names1] = grp2idx(y1);
    [y2_numeric, group_names2] = grp2idx(y2);

    if x(bestFeature1) <= bestThreshold1
        if x(bestFeature11) <= bestThreshold11
            prediction_idx = mode(y1_numeric(X1(:,bestFeature11)<=bestThreshold11)); % Majority vote
            prediction = group_names1(prediction_idx); % Convert back to label
        else
            prediction_idx = mode(y1_numeric(X1(:,bestFeature11)>bestThreshold11)); % Majority vote i
            prediction = group_names1(prediction_idx); % Convert back to label
        end
    else
        if x(bestFeature12) <= bestThreshold12
            prediction_idx = mode(y2_numeric(X2(:,bestFeature12)<=bestThreshold12)); % Majority vote
            prediction = group_names2(prediction_idx); % Convert back to label
        else
            prediction_idx = mode(y2_numeric(X2(:,bestFeature12)>bestThreshold12)); % Majority vote i
```

```
            prediction = group_names2(prediction_idx); % Convert back to label
        end
    end
end
```

6.2 Random Forests

Let's deepen a bit in the bagging technique, which we said it can be used to reduce the variance of our estimates by averaging the estimates obtained from independent models. Assume we have B models $\{M_b\}_{b=1}^{B}$, and an input x. Each model produces an estimate

$$Y_b = M_b(x),$$

which tries to predict the real value y. Let's suppose these estimates are unbiased, i.e. $E[Y_b] = y, \forall b$. Then, the average of the estimates is

$$E\left[\frac{1}{B}\sum_b Y_b\right] = \frac{1}{B}\sum_b E[Y_b] = \frac{1}{B}\sum_b y = y.$$

Therefore, this average is also an unbiased estimate of the value y. Let's see what happens with the variance when we perform this average. If we assume that the models are independent and that they all have the same variance, then it follows

$$\begin{aligned} Var\left[\frac{1}{B}\sum_b Y_b\right] &= \frac{1}{B^2} Var\left[\sum_b Y_b\right] \\ &= \frac{1}{B^2}\left(\sum_b Var[Y_b] + 2\cancelto{0}{\sum_{i\neq j} Cov(Y_i, Y_j)}\right) \\ &= \frac{1}{B^2}\sum_b Var[Y_b] \\ &= \frac{Var[Y_b]}{B}. \end{aligned}$$

Thus, the variance is reduced, making the predictions more consistent.
Since trees typically suffer from high variance (overfitting), so they seem like a good candidate to apply this technique. The application of bagging to trees are the **random forests**.
First, a bootstrapped dataset is created by randomly selecting samples from the original dataset with repetition. The samples that not chosen for the bootstrapped dataset are placed in a separate dataset, the out-of-bag dataset (OOB).
Then, we want to generate a diverse collection of trees that are better than a random choice, and are as independent from each other as possible. If we construct all these trees using the usual algorithm, then they will be very similar, so we have to inject stochasticity into this process. This is done by:

- The sampling introduces stochasticity, but it is usually not enough to provide independence between the trees.
- We can manipulate features or targets, by not taking some of them into account. This is done for each tree, so that different trees focus on different features, providing the independence that we seek.
- Change the learning parameters, the impurity function,...
- Performing combinations of the above methods.

The pseudocode of the random forest training algorithm is shown in Algorithm 9.
To make predictions on a test example x:

- For classification: output class probabilities or majority vote among $\{T_1(x), ..., T_B(x)\}$.

```
for b=1 to n_estimators do
  Sample D_b as a bootstrap of max_samples from D

  Create tree T_b on D_b, adding nodes by
    - Select max_features variables at random from all variables
    - Pick the best variable/split point according to Gini/SSE
    - Split the current node according to the split found
end

Output forest {T_1, ..., T_B}
```

Algorithm 9: random_forest(D,n_estimators,max_samples,max_features)

- For regression: output average prediction $\frac{1}{B}\sum_b T_b(x)$.

Then, to estimate the generalization error, we can use the OOB dataset to compute the **OOB error**. This generalization error can be used as a validation error to select appropriate values for the hyperparameters, so we don't need to perform cross-validation.

6.2.1 Interpretability of random forests

If n_estimators is too big (there are many trees in the forest), then it can be difficult to comprehend the decision process of the model: it is not interpretable. We can do variable importance plot to interpret better the results:

- Gini based variable importance: we can add gini impurity gains for variables in the splits in each tree in the forest, and sort the variables by their sum. This approach is biased towards categorical variables with many splits.
- Permutation based variable importance: for each variable, we can permmute values and compute the difference in the OOB error metrics. If a variable is important, then accuracy in the permuted copy should decrease. We then sort the variables using this difference. This approach is more reliable, but is slower.

6.2.2 Proximities

The idea of proximity of samples in a tree is that when two examples fall into the same leaf of a decision tree, this is evidence supporting the fact that these two examples are similar in some sense.
In the wider sense of forests, we can build a $n \times n$ similarity matrix, where each entry (i, j) correspond to the fraction of trees in the forest such that x_i, x_j end up in the same leaf. This matrix can be used in distance/similarity based methods, or to apply PCA to obtain numeric representation of the data.

6.2.3 Imbalanced data in classification

In random forest, there are several techniques to deal with imbalaced data:

- Use $F - score$ for model selection.
- Under-sampling majority class/over-sampling minority class when building the bootstrapped dataset.
- Use of weights when computing errors.

7 Multi-Layer Perceptron (Neural Networks)

In Section 4.4 we have seen the Perceptron, a mathematical model of a neuron that can be used in binary classification. We noted that this model is equivalent to a linear classifier, being thus a very limited model for a more general scenario. In this section, we are going to see how to extend this model to perform more complex tasks.

7.1 Multiclass classification

We saw how to use a single perceptron to perform binary classification, and there is a very natural way to extend this model to multi-class classification, using a one-hot encoding for the class.
For instance, let's work with a dataset $\mathcal{D} = \{x, y\}$ where $x \in \mathbb{R}^d$ and $y \in L = \{l_1, ..., l_K\}$. We now encode y as a one-hot vector, i.e., $y_k = \begin{cases} 0 & if\ y \neq l_k \\ 1 & if\ y = l_k \end{cases}$, for $k = 1, ..., K$. The idea is then to use K perceptrons, each of them focusing on predicting one of the classes and trained independently. The idea is depicted in Figure 15. Notice that in this case is almost compulsory to use the logistic function (or a similar function, but definitely not the step function), since this way we can interpret the values as probabilities, and therefore select the value with highest probability as our prediction. Also, remember than in x we are adding an artifical 1 to account for the bias.

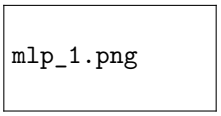


Figure 15: Multi-class classification with perceptrons.

Let's formalize this idea a little bit. We have K independent perceptrons, each of them predicting a variable, so that if we call g our activation function, we obtain, for each $k = 1, ..., K$,

$$y_k(x) = g\left(w_k^T x\right),$$

where w_k is the weight vector of perceptron k. We can integrate all this into a single formula by introducing the notation $f[\cdot]$, which applies f to each component of the input vector:

$$f[x] = \begin{pmatrix} f(x_1) \\ \vdots \\ f(x_d) \end{pmatrix}.$$

This way, we can unify the previous equations as

$$y(x) = g\left[W^T x\right],$$

where we have arranged

$$W = \begin{pmatrix} | & & | \\ w_1 & ... & w_K \\ | & & | \end{pmatrix}.$$

Now, we can define what is a layer. A **layer** is just a set of parallel neurons and a layer with M neurons outputs M variables. Usually, all neurons in a layer use the same activation function, g. Note that if this is not the case, then our previous explanations needs to be slightly adapted.
Since we have discussed that a good alternative in multiclass classification is to use the sigmoid function as activation function, we can rewrite our model as

$$y(x) = \sigma\left[W^T x\right].$$

Note, nonetheless, that here we are obtaining a bunch of different probabilities, one for each class, which could be further used to increase the information about our prediction. For instance, instead of computing the

maximum out of these independent probabilities, we can add a **softmax function** to the outputs, resulting in a normalized vector that can now be considered as a probability vector. The softmax function is

$$softmax\left(y\right)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}.$$

Also, at doing this, it is not necessary to apply the activation function, g, beforehand. This approach is shown in Figure 16.

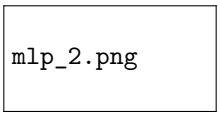


Figure 16: Multi-class classification with perceptrons and softmax.

Note that we are still in the linear classification scenario, and the next step is to improve the flexibility of the model to be able to cope with more complex relationships.

7.2 Multi-Layer Perceptron

Notice now an interesting fact: if our activation function is $g : \mathbb{R} \to I$, i.e., I is the range of g, then a layer with M perceptrons can be interpreted as a function

$$G : \mathbb{R}^n \to I^M.$$

For the sake of simplicity, let's say $G : \mathbb{R}^n \to \mathbb{R}^M$. Then, it is natural to think on stacking different layers, so that the output of a layers becomes the input of the following, effectively performing the composition of the functions defined by these layers. For example, if we have two layers, which we interpret as two functions $G^{(1)} : \mathbb{R}^n \to \mathbb{R}^{M_1}$ and $G^{(2)} : \mathbb{R}^{M_1} \to \mathbb{R}^{M_2}$, we can take the output of the first layer and use it as input for the second one, effectively obtaining

$$G^{(2)} \circ G^{(1)} : \mathbb{R}^n \to \mathbb{R}^{M_2}.$$

This is interesting mainly because a single layer, as we have seen, is only able to perform linear classification, but let's deep a bit more on what is happening on the second layer. The second layer is using an activation function $g^{(2)} : \mathbb{R} \to \mathbb{R}$, which is applied in each perceptron of the second layer as

$$y_k^{(2)}\left(x\right) = g_k^{(2)}\left(w_k^{(2)^T} \cdot y^{(1)}\left(x\right)\right) = g_k^{(2)}\left(w_k^{(2)^T} \cdot g^{(1)}\left[W^{(1)^T} \cdot x\right]\right).$$

This means that we are:

- Creating M_1 combinations of the input variables, $W^{(1)^T} \cdot x$, where $W^{(1)} \in \mathcal{M}_{M_1 \times (n+1)}$ (the $+1$ is to account for the bias).
- Applying a different transformation $g_k^{(1)}$ to each combination of variables. Note that this transformation can be arbitrary, but should be non-linear if we want to go out of the linear world, and should be differentiable if we want a smooth training process.
- For the second layer, the inputs are the transformations of the combinations of the input variables, which effectively become a base of functions (just as polynomials are), which are combined again by $w_k^{(2)^T}$.
- Finally, we apply $g_k^{(2)}$, which will be again an activation function.

The key steps for going beyond linearity are the second and third. Let's compare it to a normal linear regression. In that case, we introduced basis functions to be able to go out of the linearity world, by transforming the input variables. But then, we used still linear regression over this transformation. Therefore, it was the modification of the input what allowed us to model non-linear relationship, and not the algorithm itself, which was not changed at all. Here, the idea is similar. In fact, we could just perform the same transformations that we used to in this new case, and keep applying just one layer of perceptrons to obtain non-linear classifiers. But this is exactly the power of the multi-layer perceptron: we don't need to assume or guess the shape of the basis

function! The different weights of the different layers will adapt to our problem, effectively auto-selecting how these functions should be. Of course, this is not free, and the cost is paid by obtaining harder trainings.
After this explanation, we can first integrate the previous formula as

$$y^{(2)}(x) = g^{(2)}\left[W^{(2)^T} \cdot g^{(1)}\left[W^{(1)^T} x\right]\right].$$

We can also define the **multi-layer perceptron** as composition of layers of perceptrons, with the addition of what is called the **input layer**, which is not really a layer of perceptrons, but rather represents and fixes the size of the input, and the **output layer**, which is the final layer, outputing the predictions of the model. The rest of the layers, the real perceptron layers, are called **hidden layers**. This is depicted in Figure 17. In this case, the first layer has M_1 perceptrons, the second one has M_2 perceptrons, and the output layers has as many perceptrons as possible labels, K.

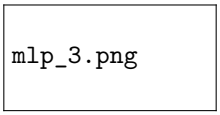


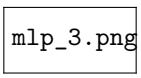
Figure 17: A Multi-Layer Perceptron.

Remark 7.1. Observe that we have connected all outputs of each layer to all perceptrons in the second layer, also, note that two neurons within the same layer do not connect. These are just decisions and is not compulsory, and many different architectures exist.

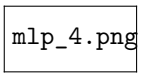
More generally, a **neural network** is a directed graph whose nodes are perceptrons. If there are no cycles within the network, then it is a **feed-forward neural network (FFNN)**, and it is called a **recurrent network (RNN)** in the case of cycles existing within the network.
MLP are thus a special case of feed-forward neural network, in which:

- Neurons are arranged in layers.
- There is at least one hidden layer.
- Every layer is fully connected to the next one.
- No connections are allowed within layers.

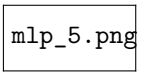
A brief visualization on different kinds of networks is depicted in Figure 18.



(a) MLP.



(b) FFNN not MLP.



(c) RNN.

Figure 18: Different kinds of NN.

7.3 Error functions

7.3.1 Regression

The error in the regression case can be the empirical mean square error, i.e.,

$$E(W) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - y_W(x_i)\right)^2,$$

where the dataset is $\{(x_i, y_i)\}_{i=1}^n$, $y_W(x_i)$ is the prediction of x_i with the weight matrix W. Note that in this formula we are assuming a single output, but it can be easily generalized as

$$E(W) = \frac{1}{2}\sum_{i=1}^{n} \|y_i - y_W(x_i)\|^2 .$$

As usual, our objective will be to minimize this error.

7.3.2 Binary classification

If the target values are $y_i \in \{0, 1\}$, then we can express

$$P(y|x) = y_W(x)^y (1 - y_W(x))^{1-y} .$$

If we assume independent and identically distributed examples, then we can define our likelihood function as

$$\mathcal{L}(W) = \prod_{i=1}^{n} y_W(x_i)^{y_i} (1 - y_W(x_i))^{1-y_i} ,$$

and the negative log-likelihood defines the **cross-entropy error**:

$$E(W) = \text{-log}\,\mathcal{L}(W) = -\sum_{i=1}^{n} y_i \log(y_W(x_i)) + (1 - y_i)\log(1 - y_W(x_i)) .$$

7.3.3 Multi-class classification

The last error formula can be generalized for multi-class classification with $K > 2$ classes, as the **generalized cross-entropy error**:

$$E(W) = -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k} \log(y_{W,k}(x_i)) .$$

7.4 Training the MLP: Backprogragation

Notice that the error functions that we have defined can be very complex, and there is no closed form solution to minimize them. Therefore, some iterative approach is necessary, as we did with the perceptron. But we encounter now a new problem: if we do gradient descent on the neural network, we will need to compute lots of gradients, and computing a gradient is not cheap. The idea to make this process more efficient is to use the chain rule and to notice that many of the gradients that we compute are actually used in many places along the network for the computation of the final gradient. Therefore, an improved version for computing gradients was developed: the **backpropagation algorithm**.

7.4.1 The chain rule

The chain rule[8] states that, given two composable differentiable functions f, g, it is

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x) .$$

In the multivariate case, when we want to compute the derivative of $f(g_1(x), ..., g_m(x))$, it is

$$\frac{d(f(g_1(x), ..., g_m(x)))}{dx} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i}(g_1(x), ..., g_m(x)) \cdot \frac{dg_i}{dx}(x) .$$

This is sometimes written by naming $z = f(g_1(x), ..., g_m(x))$ and $y_i = g_i(x)$, so that the formula becomes[9]

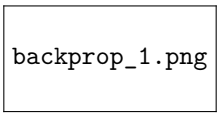
$$\frac{dz}{dx} = \sum_{i=1}^{m} \frac{dz}{dy_i}\frac{dy_i}{dx}.$$

[8] For more information you can just visit Wikipedia.
[9] Notice the abuse of notation though.

Example 7.1. Let's do an example to understand how this can be helpful in the case of neural networks. Imagine we have the functions:

$$\begin{aligned} y_1 &= x_1^2 \cdot e^{x_2} \\ y_2 &= x_2^3 \\ z &= y_1 \left(1 - e^{y_2}\right), \end{aligned}$$

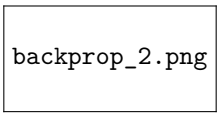
and we want to compute the gradient of z in terms of X. We can write this as a diagram:



We simply need to compute the gradient using the chain rule:

$$\nabla z = \begin{pmatrix} \frac{dz}{dx_1} \\ \frac{dz}{dx_2} \end{pmatrix} = \begin{pmatrix} \frac{dz}{dy_1}\frac{dy_1}{dx_1} + \frac{dz}{dy_2}\frac{dy_2}{dx_1} \\ \frac{dz}{dy_1}\frac{dy_1}{dx_2} + \frac{dz}{dy_2}\frac{dy_2}{dx_2} \end{pmatrix} = \begin{pmatrix} (1-e^{y_2})2x_1e^{x_2} - y_1e^{y_2} \cdot 0 \\ (1-e^{y_2})x_1^2e^{x_2} - y_1e^{y_2}3x_2^2 \end{pmatrix} = \begin{pmatrix} (1-e^{y_2})2x_1e^{x_2} \\ (1-e^{y_2})x_1^2e^{x_2} - y_1e^{y_2}3x_2^2 \end{pmatrix}.$$

Basically, the idea is to be able to reuse as many values as possible (notice the coloured expressions). To do this, we can compute **local derivatives** in each layer, and *propagate* them back to the previous layer, multiplying them. The previous computation can be done in the diagram as follows:



Note that:

1. We can avoid unnecessary computations by storing at each node:
   (a) Forward values: when we evaluate $y_1\,(x_1, x_2)$, we can keep this value stored in the node, because we will need it later. In this example we did symbolic differentiation, so this was not necessary, but in neural networks we need to first to the forward pass, and then assess the error.
   (b) Local derivatives: when we evaluate $\frac{dy_1}{dx_2}$, we can keep this value stored in the node, because we will use it several times.
   (c) Backwards gradients: local derivatives are sent to those nodes that will need them, so that each of this derivatives is computed only once. Then the gradient is constructed little by little, backwards, in a dynamic programming manner.
2. When the gradients are flowing backwards, each node has to sum all incoming gradients, according to the multivariate chain rule (if there is only one incoming gradient, we just take that one).

7.4.2 The backpropagation algorithm

The pseudocode for backpropagation is as in Algorithm 10. Let's break it down.

- `a[layer]`=$a^{(l)}$ is the input vector for layer l.
- `z[layer]`=$z^{(l)}$ is the output vector for layer l, i.e., $z^{(l)} = g_l\left[a^{(l)}\right]$, where g_l is the activation function of layer l (note in the pseudocode we are assuming the same g working on all layers).
- `W[layer]`=$W^{(l)}$ is the weight matrix of layer l.
- `w[layer]`=$\hat{W}^{(l)}$ is the weight matrix of layer l, without the weight of the bias.
- d[layer]=$\delta^{(l)}$ is the local gradient at layer l, computed as

$$\delta^{(out)} = g'_{out}\left[a^{(out)}\right] \odot \left(z^{(out)} - y\right)$$

for the output layer, *out* or `d[c+1]` in the pseudocode, and

$$\delta^{(l)} = g_l' \left[a^{(l)}\right] \odot \left(\hat{W}^{(l)}\delta^{(l+1)}\right)$$

for the rest of the layers, i.e., the hidden layers. Here, $\odot$ is the Hadamard product, i.e., the component-wise product. In the pseudocode, this is expressed by '`.*`'.

```
1. Forward pass
  - for layer in {1...c+1}
    a[layer] = W[layer]^T * z[layer-1]
    z[layer] = g[a[layer]] # apply g[.] to each component of a[layer]

2. Backward pass
  - d[c+1] = dg[a[c+1]] .* (z[c+1]-y) # .* is the component-wise product and dg is the
    derivative of g
  - grad[c+1] = z[c]*d[c+1]^T
  - for layer in {c...1}
    d[layer] = dg[a[layer]] .* (w[layer+1]*d[layer+1]) # w[k] is W[k] without the bias weigth
    grad[layer] = z[layer-1]*d[layer]^T

return grad
```

Algorithm 10: Backpropagation.

Example 7.2. Let's perform an example. Here is our neural network:

backprop_3.png

It is a fully connected two-layer network. Let's follow the algorithm, defining each of the variables at a time.

Forward pass. We define $a^{(1)} = x, a^{(2)} = z^{(1)} = g_1\left[a^{(1)}\right], a^{(out)} = z^{(2)} = g_2\left[a^2\right]$ and $z^{(out)} = z = g_{out}\left[a^{(out)}\right]$:

backprop_4.png

epBackward pass(c+1). Define $d^{(out)} = g_{out}'\left[a^{(out)}\right] \odot \left(z^{(out)} - y\right)$ and $grad^{(out)} = z^{(2)} \cdot \delta^{(out)^T}$:

backprop_5.png

Step 1. Layer 2. Define $d^{(2)} = g_2'\left[a^{(2)}\right] \odot \left(\hat{W}^{(out)}\delta^{(out)}\right)$ and $grad^{(2)} = z^{(1)}\delta^{(2)^T}$:

backprop_6.png

Step 2. Layer 1. Define $d^{(1)} = g_1'\left[a^{(1)}\right] \odot \left(\hat{W}^{(2)}\delta^{(2)}\right)$ and $grad^{(1)} = z^{(0)}\delta^{(1)^T} = a^{(1)}\delta^{(1)^T}$:

backprop_7.png

And that's it! Now we can use the gradient at each layer to train the network, updating the weights in layer l according to:

$$W^{(l)} = W^{(l)} - \gamma \cdot grad^{(l)},$$

where γ is the learning rate.

7.5 Some activation functions

7.5.1 Logistic

The logistic function is $\sigma : \mathbb{R} \to (0, 1)$, defined by

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

with derivative

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)).$$

In this case, for the backpropagation algorithm, we use

$$g'\left[a^{(l)}\right] = g\left[a^{(l)}\right] \odot \left(1 - g\left[a^{(l)}\right]\right) = z^{(l)} \odot \left(1 - z^{(l)}\right),$$

which is a very efficient formula, since we have all this values computed in the forward pass.

7.5.2 Hyperbolic tangent

The hyperbolic tangent function is $tanh : \mathbb{R} \to (-1, 1)$, defined by

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

with derivative

$$\frac{d}{dx}tanh(x) = 1 - (tanh(x))^2.$$

Thus, we end up with another very efficient formula

$$g'\left[a^{(l)}\right] = 1 - z^{(l)} \odot z^{(l)}.$$

Example 7.3. Implementing a NN for digit classification in MATLAB

Implementing Neural Networks

We are going to tackle the problem of digit classification from image. More concretely, we are going to use the MNIST digits dataset. The explanation to load the dataset is in
https://lucidar.me/en/matlab/load-mnist-database-of-handwritten-digits-in-matlab/ .

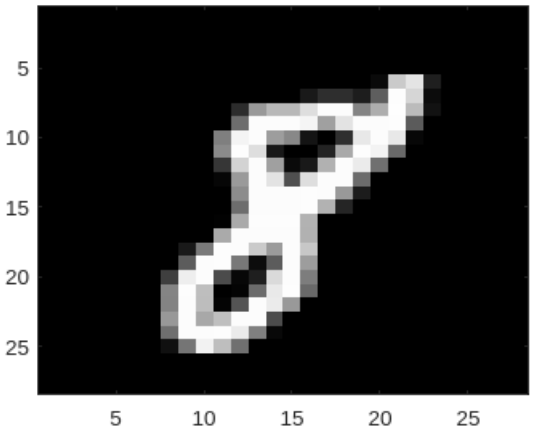
```
load ('mnist.mat')

X_train = training.images;
Y_train = training.labels;
X_test = test.images;
Y_test = test.labels;
```

```
X_train(:,:,18)
```

```
ans = 28x28
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

```
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

```
image (training.images(:,:,18)*255);
colormap("gray")
```



This images are matrices of 28x28 in grayscale. This means that our network needs a 28x28=784 sized input layer. We need to resize these matrices:

```
X_train = reshape(X_train,[784,60000])';
X_test = reshape(X_test,[784,10000])';
```

```
Y_train(1)
```

```
ans = 5
```

The labels are expressed as the digit. We want them as a one-hot encoding.

```
TT_train = Y_train + 1; % The function ind2vec needs positive values.
TTrain = full(ind2vec(TT_train'))';
TT_test = Y_test + 1;
TTest = full(ind2vec(TT_test'))';

Y_train(18)
```

```
ans = 8
```

```
TTrain(18,:)
```

```
ans = 1x10
     0     0     0     0     0     0     0     0     1     0
```

We can see how the 9th position of the one-hot encoded version is set to 1. This means the digit is 8.

Also, since there are 10 possible digits, we will tackle this problem as a 10-class classification, so the output layer will have 10 neurons.

For the hidden layers, we would need to decide what to do. We are going to go with one fully connected hidden layer of 32 neurons.

We also need to initialize the weights of our network.

And the activation function and its derivative. We are using the ReLU activation function for the hidden layer, and softmax for the output layer.

```
inputLayerSize = 784;
hiddenLayer1Size = 50;
outputLayerSize = 10;
```

We also need to initialize the weights of our network:

```
[W1,b1,Wout,bout] = generate_weights(inputLayerSize,hiddenLayer1Size,outputLayerSize,0.1);
```

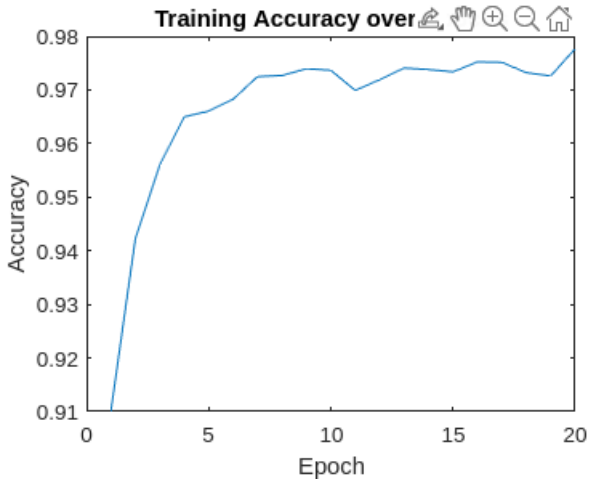
And the activation function and its derivative. We are using the sigmoid here:

```
learningRate = 0.01;
numEpochs = 20;
batchsize = 100;

accuracy_history = zeros(numEpochs, 1);

for epoch = 1:numEpochs
    for i = 1:size(X_train, 1)/batchsize
        % Use the permuted indices to get the current example and label
        X = X_train(1+batchsize*(i-1):batchsize*i,:);
        T = TTrain(1+batchsize*(i-1):batchsize*i,:);

        % Feedforward
        [A1,Z1,Aout,Zout] = forward_pass(X,W1,b1,Wout,bout);

        % Backpropagation
        [grad_W1, grad_b1, grad_Wout, grad_bout] = backward_pass(Zout, T, Z1, X, Wout, A1);

        % Update weights and biases
        [W1, b1, Wout, bout] = update_weights(learningRate,W1,b1,Wout,bout,grad_W1,grad_b1,grad_Wout,
    end
    accuracy_history(epoch) = compute_accuracy(X_train, TTrain, W1, b1, Wout, bout);
    fprintf('Epoch %d: Accuracy: %0.5f\n', epoch, accuracy_history(epoch));
end
```

```
Epoch 1: Accuracy: 0.91012
Epoch 2: Accuracy: 0.94228
Epoch 3: Accuracy: 0.95607
Epoch 4: Accuracy: 0.96497
Epoch 5: Accuracy: 0.96605
Epoch 6: Accuracy: 0.96827
```

```
Epoch 7: Accuracy: 0.97248
Epoch 8: Accuracy: 0.97268
Epoch 9: Accuracy: 0.97395
Epoch 10: Accuracy: 0.97367
Epoch 11: Accuracy: 0.96990
Epoch 12: Accuracy: 0.97192
Epoch 13: Accuracy: 0.97407
Epoch 14: Accuracy: 0.97380
Epoch 15: Accuracy: 0.97337
Epoch 16: Accuracy: 0.97522
Epoch 17: Accuracy: 0.97515
Epoch 18: Accuracy: 0.97322
Epoch 19: Accuracy: 0.97260
Epoch 20: Accuracy: 0.97768
```

```
figure;
plot(1:numEpochs, accuracy_history);
title('Training Accuracy over Epochs');
xlabel('Epoch');
ylabel('Accuracy');
```



Test:

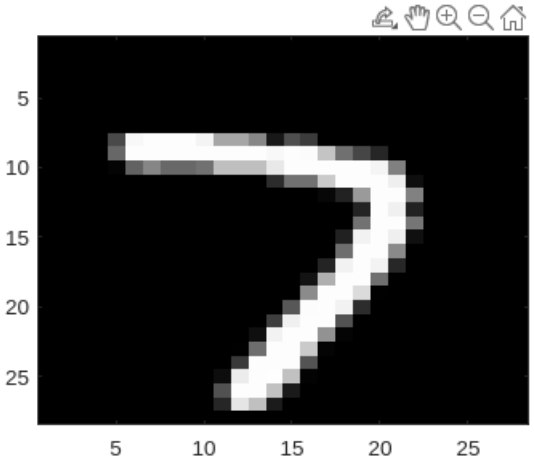
```
accuracy = compute_accuracy(X_test,TTest,W1,b1,Wout,bout);
fprintf('Accuracy: %0.5f\n', accuracy);
```

```
Accuracy: 0.95960
```

```
x = X_test(18,:);
pred = prediction(x,W1,b1,Wout,bout)
```

```
pred = 7
```

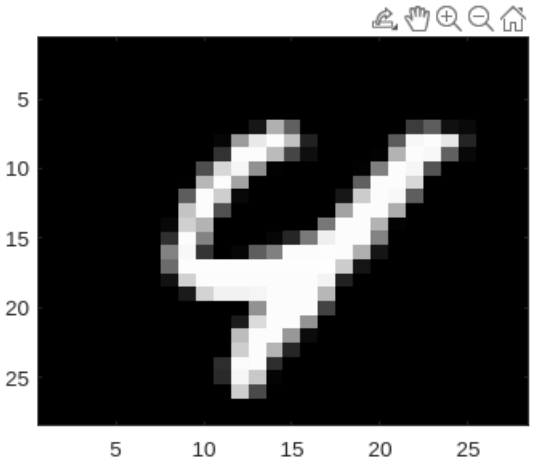
```
image (reshape(x*255,[28,28]));
colormap("gray")
```




```
x = X_test(140,:);
pred = prediction(x,W1,b1,Wout,bout)
```

```
pred = 4
```

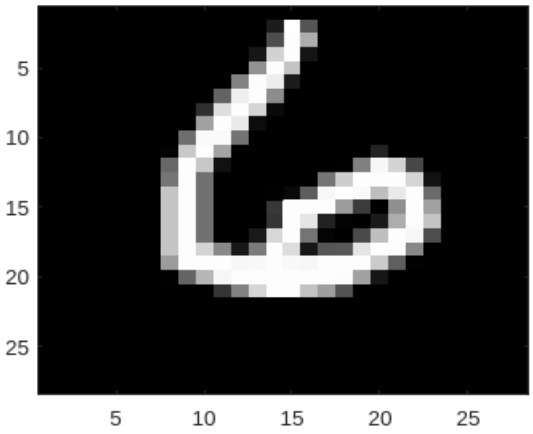
```
image (reshape(x*255,[28,28]));
colormap("gray")
```




```
x = X_test(5442,:);
pred = prediction(x,W1,b1,Wout,bout)
```

```
pred = 6
```

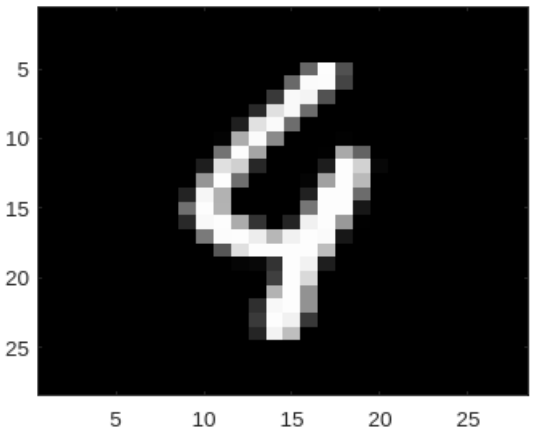
```
image (reshape(x*255,[28,28]));
colormap("gray")
```




```
pred = -1;
real = -1;
i = 82;

while pred == real
    x = X_test(i,:);
    pred = prediction(x,W1,b1,Wout,bout);

    real = Y_test(i);

    i = i+1;
end

x = X_test(i-1,:);
pred = prediction(x,W1,b1,Wout,bout)
```

```
pred = 9
```

```
image (reshape(x*255,[28,28]));
colormap("gray")
```

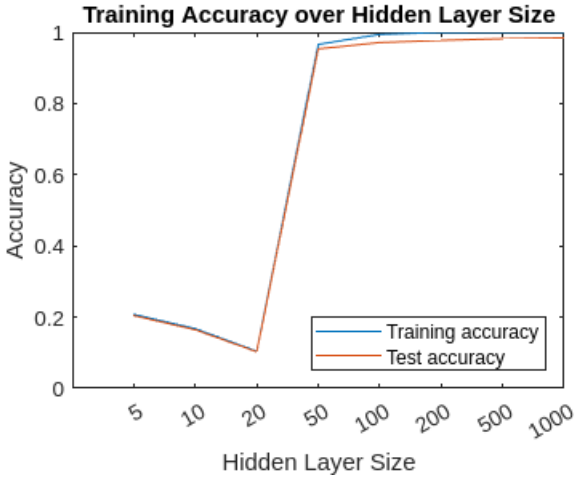


Varying parameters

```
inputLayerSize = 784;
hiddenLayer1Size = [5,10,20,50,100,200,500,1000];
outputLayerSize = 10;

accuracy_history = zeros(length(hiddenLayer1Size), 1);
accuracy_test = zeros(length(hiddenLayer1Size), 1);
for p = 1:length(hiddenLayer1Size)

    [W1,b1,Wout,bout] = generate_weights(inputLayerSize,hiddenLayer1Size(p),outputLayerSize,0.1);

    learningRate = 0.01;
    numEpochs = 20;
    batchsize = 100;

    for epoch = 1:numEpochs
        for i = 1:size(X_train, 1)/batchsize
            % Use the permuted indices to get the current example and label
            X = X_train(1+batchsize*(i-1):batchsize*i,:);
            T = TTrain(1+batchsize*(i-1):batchsize*i,:);

            % Feedforward
            [A1,Z1,Aout,Zout] = forward_pass(X,W1,b1,Wout,bout);

            % Backpropagation
            [grad_W1, grad_b1, grad_Wout, grad_bout] = backward_pass(Zout, T, Z1, X, Wout, A1);

            % Update weights and biases
            [W1, b1, Wout, bout] = update_weights(learningRate,W1,b1,Wout,bout,grad_W1,grad_b1,grad_W
        end
    end
    accuracy_history(p) = compute_accuracy(X_train, TTrain, W1, b1, Wout, bout);
    fprintf('Hidden layer of size %d: Training Accuracy: %0.5f\n', hiddenLayer1Size(p), accuracy_hist

    accuracy_test(p) = compute_accuracy(X_test, TTest, W1, b1, Wout, bout);
    fprintf('Hidden layer of size %d: Test Accuracy: %0.5f\n', hiddenLayer1Size(p), accuracy_test(p))
end
```

```
Hidden layer of size 5: Training Accuracy: 0.20782
Hidden layer of size 5: Test Accuracy: 0.20440
Hidden layer of size 10: Training Accuracy: 0.16777
Hidden layer of size 10: Test Accuracy: 0.16450
Hidden layer of size 20: Training Accuracy: 0.10442
Hidden layer of size 20: Test Accuracy: 0.10280
Hidden layer of size 50: Training Accuracy: 0.96575
Hidden layer of size 50: Test Accuracy: 0.95330
Hidden layer of size 100: Training Accuracy: 0.99427
Hidden layer of size 100: Test Accuracy: 0.97120
Hidden layer of size 200: Training Accuracy: 0.99955
Hidden layer of size 200: Test Accuracy: 0.97730
Hidden layer of size 500: Training Accuracy: 0.99995
Hidden layer of size 500: Test Accuracy: 0.98200
Hidden layer of size 1000: Training Accuracy: 1.00000
Hidden layer of size 1000: Test Accuracy: 0.98440
```

```
figure;
plot(1:length(hiddenLayer1Size), accuracy_history);
hold on
plot(1:length(hiddenLayer1Size), accuracy_test);
title('Training Accuracy over Hidden Layer Size');
legend('Training accuracy', 'Test accuracy','Location','southeast');
xlabel('Hidden Layer Size');
ylabel('Accuracy');

% Specify the locations of the x-axis ticks
xticks(1:length(hiddenLayer1Size));

% Specify the labels of the x-axis ticks
xticklabels(hiddenLayer1Size);
```

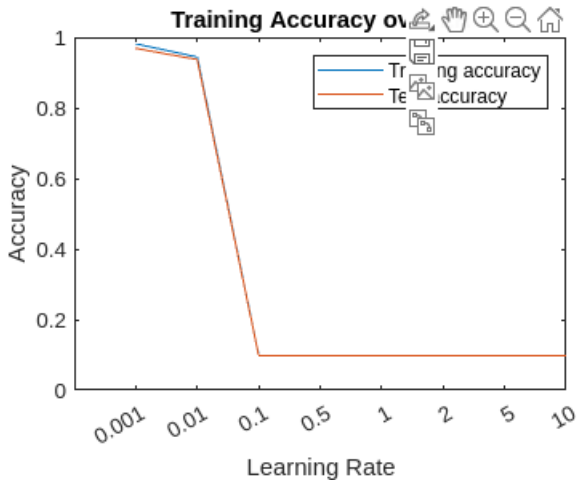



```
inputLayerSize = 784;
hiddenLayer1Size = 50;
outputLayerSize = 10;
[W1,b1,Wout,bout] = generate_weights(inputLayerSize,hiddenLayer1Size,outputLayerSize,0.1);

learningRate = [0.001,0.01,0.1,0.5,1,2,5,10];
numEpochs = 20;
batchsize = 100;

accuracy_history = zeros(length(learningRate), 1);
accuracy_test = zeros(length(learningRate), 1);

for p = 1:length(learningRate)
    lr = learningRate(p);
    for epoch = 1:numEpochs
        for i = 1:size(X_train, 1)/batchsize
            % Use the permuted indices to get the current example and label
            X = X_train(1+batchsize*(i-1):batchsize*i,:);
            T = TTrain(1+batchsize*(i-1):batchsize*i,:);

            % Feedforward
            [A1,Z1,Aout,Zout] = forward_pass(X,W1,b1,Wout,bout);
```

```
            % Backpropagation
            [grad_W1, grad_b1, grad_Wout, grad_bout] = backward_pass(Zout, T, Z1, X, Wout, A1);

            % Update weights and biases
            [W1, b1, Wout, bout] = update_weights(lr,W1,b1,Wout,bout,grad_W1,grad_b1,grad_Wout,grad_b
        end
    end
    accuracy_history(p) = compute_accuracy(X_train, TTrain, W1, b1, Wout, bout);
    fprintf('Learning Rate %d: Accuracy: %0.5f\n', lr, accuracy_history(p));

    accuracy_test(p) = compute_accuracy(X_test, TTest, W1, b1, Wout, bout);
    fprintf('Learning rate %d: Test Accuracy: %0.5f\n', lr, accuracy_test(p));
end
```

```
Learning Rate 1.000000e-03: Accuracy: 0.98153
Learning rate 1.000000e-03: Test Accuracy: 0.96870
Learning Rate 1.000000e-02: Accuracy: 0.94405
Learning rate 1.000000e-02: Test Accuracy: 0.93680
Learning Rate 1.000000e-01: Accuracy: 0.09872
Learning rate 1.000000e-01: Test Accuracy: 0.09800
Learning Rate 5.000000e-01: Accuracy: 0.09872
Learning rate 5.000000e-01: Test Accuracy: 0.09800
Learning Rate 1: Accuracy: 0.09872
Learning rate 1: Test Accuracy: 0.09800
Learning Rate 2: Accuracy: 0.09872
Learning rate 2: Test Accuracy: 0.09800
Learning Rate 5: Accuracy: 0.09872
Learning rate 5: Test Accuracy: 0.09800
Learning Rate 10: Accuracy: 0.09872
Learning rate 10: Test Accuracy: 0.09800
```

```
figure;
plot(1:length(learningRate), accuracy_history);
hold on
plot(1:length(learningRate), accuracy_test);
title('Training Accuracy over LR');
legend('Training accuracy', 'Test accuracy','Location','northeast');
xlabel('Learning Rate');
ylabel('Accuracy');

% Specify the locations of the x-axis ticks
xticks(1:length(learningRate));

% Specify the labels of the x-axis ticks
xticklabels(learningRate);
```




```
function ret = relu_func(z)
    ret = max(0,z);
end

function ret = relu_prime(z)
    ret = z>0;
end

function ret = softmax(z)
    ret = exp(z) ./ sum(exp(z),1);
end

function [W1, b1, Wout, bout] = generate_weights(input_size, layer1_size, output_size, scale)
    W1 = scale*(rand(layer1_size,input_size)-0.5);
    b1 = scale*(rand(layer1_size,1)-0.5);
    Wout = scale*(rand(output_size,layer1_size)-0.5);
    bout = scale*(rand(output_size,1)-0.5);
end

function [A1,Z1,Aout,Zout] = forward_pass(X,W1,b1,Wout,bout)
    % X has the nrows records with ncols features
    A1 = W1*X'+repmat(b1,1,size(X,1));
    Z1 = relu_func(A1);

    Aout = Wout*Z1+repmat(bout,1,size(X,1));
    Zout = softmax(Aout);
end

function [grad_W1, grad_b1, grad_Wout, grad_bout] = backward_pass(Zout, T, Z1, X, Wout, A1)
    % Error in output
    d_output = Zout - T';

    % Gradient for output weights and biases
    grad_Wout = d_output * Z1';
    grad_bout = sum(d_output, 2);

    % Error in first layer
    d_layer1 = (Wout' * d_output) .* relu_prime(A1);
```

```
    % Gradient for first layer weights and biases
    grad_W1 = d_layer1 * X;
    grad_b1 = sum(d_layer1, 2);
end

function [new_W1, new_b1, new_Wout, new_bout] = update_weights(lr,W1,b1,Wout,bout,grad_W1,grad_b1,gra
    new_W1 = W1 - lr*grad_W1;
    new_b1 = b1 - lr*grad_b1;
    new_Wout = Wout - lr*grad_Wout;
    new_bout = bout - lr*grad_bout;
end

function accuracy = compute_accuracy(X, T, W1, b1, Wout, bout)
    % Feedforward
    [~, ~, ~, Zout] = forward_pass(X, W1, b1, Wout, bout);

    % Get the indices of the max values of the predicted and target labels
    [~, predicted_labels] = max(Zout);
    [~, true_labels] = max(T');

    % Compute accuracy
    accuracy = mean(predicted_labels == true_labels);
end

function pred = prediction(x,W1,b1,Wout,bout)
    [~, ~, ~, Zout] = forward_pass(x, W1, b1, Wout, bout);
    [~, predicted_labels] = max(Zout);
    pred = predicted_labels-1;
end
```

A Notes on probability theory, Bayes theorem and Bayesian learning

These notes are adapted from [3].

A.1 Probability theory basic

Let Ω be a sample space, i.e., the set of possible outcomes of an event, and $A \subset \Omega$ an event. A probability measure is a function

$$P : \mathcal{P}(\Omega) \to \mathbb{R},$$

that assigns a real number to every event, $P(A)$. This represents how likely it is that the experiment's outcome is in A.
For (Ω, P) to be a probability space[10] we have to impose three axioms:

A1) $P(A) \geq 0, \forall A \subset \Omega$.

A2) $P(\Omega) = 1$.

A3) $P(A \cup B) = P(A) + P(B)$ if $A \cap B = \emptyset$.

From these axioms, some consequences can be derived:

- $P\left(\overline{A}\right) \stackrel{def}{:=} P(\Omega \setminus A) = 1 - P(A)$.

  Proof. We can write $\Omega = A \cup (\Omega \setminus A)$ and $A \cap (\Omega \setminus A) = \emptyset$, so we have

  $$1 \stackrel{A2}{=} P(\Omega) = P(A \cup (\Omega \setminus A)) \stackrel{A3}{=} P(A) + P(\Omega \setminus A),$$

  thus:

  $$P\left(\overline{A}\right) = P(\Omega \setminus A) = 1 - P(A).$$

  □

- $P(\emptyset) = 0$.

  Proof. $P(\emptyset) = P\left(\overline{\Omega}\right) = 1 - P(\Omega) = 0$. □

- If $A \subset B$, then $P(A) \leq P(B)$.

  Proof. In this case, we can write $B = A \cup (B \setminus A)$, so $P(B) = P(A) + P(B \setminus A)$ and we have the inequality. □

- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.

  Proof. We have $A = (A \setminus B) \cup (A \cap B)$, $B = (B \setminus A) \cup (A \cap B)$ and $A \cup B = (A \setminus B) \cup (A \cap B) \cup (B \setminus A)$, so

  $$\begin{aligned} P(A \cup B) =& P(A \setminus B) + P(A \cap B) + P(B \setminus A) \\ =& P(A) - P(A \cap B) + P(A \cap B) + P(B) - P(A \cap B) \\ =& P(A) + P(B) - P(A \cap B). \end{aligned}$$

  □

- $P(A \cup B) \leq P(A) + P(B)$.

  Proof. This is obvious from the previous result. □

[10] In fact, we need one more ingredient, $\mathcal{F}$, which is a σ-algebra of subsets of Ω. This is just a formalization, but we can abuse notation here to simplify some things.

A.1.1 Joint probability

It is usal to be interested in the probability of two events happening simultaneously. This is called the **joint probability** of events A and B:

$$P(A,B) \overset{def}{:=} P(A \cap B).$$

The join probability is useful when an event can be decomposed in simpler disjoint events, i.e., we have $A \subset B_1 \cup ... \cup B_n$, with $B_i \cap B_j = \emptyset, \forall i \neq j$. In this situation, we can use the **sum rule**:

$$P(A) = \sum_{i=1}^{n} P(A, B_i).$$

This is also known as **marginalization**: when we know the joint probability $p(x,y)$ and we want to compute $p(x)$, we marginalize out y. This basically means that if we know the probabilities of all possible pairs (x,y), we can know the probability of x by exploring all the possibilities. Here, 'exploring' is using the sum rule:

$$p(x) = \sum_{y} p(x,y)$$

or

$$p(x) = \int_{y} p(x,y)\, dy,$$

if y is continuous.

Example A.1. We have two events:

- x: earns more than 100k or earns less than 100k.
- y: is a professor, a software engineer or a data scientist.

We have some sources of information, and are able to determine that

$$p(>100, prof) = 0.05,\ p(>100, seng) = 0.1,\ p(>100, dsci) = 0.2,$$

then we can conclude that

$$p(>100) = 0.05 + 0.1 + 0.2 = 0.35.$$

A.1.2 Conditional probability

The **conditional probability** of B given A is the probability that B occurs, knowing that A has occurred. This means that we have to restrict the space of possible outcomes, from Ω to A[11]:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}.$$

If we rearrange the terms, we can obtain the **product rule**:

$$P(A,B) = P(B|A)\, P(A).$$

This formula can be generalized to an arbitrary number of events, the **general product rule**, given by[12]

$$P(A_1, ..., A_n) = \prod_{i=1}^{n} P(A_i | A_1, ..., A_{i-1}).$$

Exercise A.1. Prove that

$$P(A,B,C) = P(A)\, P(B|A)\, P(C|A,B) = P(C)\, P(B|C)\, P(A|B,C).$$

Then, prove the general product rule.

[11] Note that $P(A) > 0$ is needed.

[12] Note that here it is needed that all the intersections $A_1 \cap ... \cap A_i$ have non-zero probability, for $i = 1, ..., n-1$.

Basically, we are asked to prove the general product rule by induction.
For $n = 2$, we have already proven it.
For $n = 3$, we hav

$$\begin{aligned} P((A_1 \cap A_2) \cap A_3) =& P(A_3|A_1 \cap A_2) P(A_1 \cap A_2) \\ =& P(A_3|A_1 \cap A_2) P(A_2|A_1) P(A_1), \end{aligned}$$

which is what we wanted.
Now, assume it is true for $n-1$. Then, we have

$$\begin{aligned} P(A_1 \cap ... \cap A_n) =& P((A_1 \cap ... \cap A_{n-1}) \cap A_n) \\ =& P(A_n|A_1 \cap ... \cap A_{n-1}) P(A_1 \cap ... \cap A_{n-1}) \\ \overset{induction}{=}& P(A_n|A_1 \cap ... \cap A_{n-1}) \prod_{i=1}^{n-1} P(A_i|A_1, ..., A_{i-1}) \\ =& \prod_{i=1}^{n} P(A_i|A_1, ..., A_{i-1}). \end{aligned}$$

A.1.3 Bayes rule

Bayes theorem gives an alternative formula for the conditional probability:

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}.$$

Proof. Assuming all involved probabilities are non-zero:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \overset{prod\ rule}{=} \frac{P(B|A) P(A)}{P(B)}.$$

□

This rule is known to be useful to update the probability of an event happening, when we are able to gather new information of related events. $P(A)$ is usually called the **prior probability**, and $P(A|B)$ is the **a posteriori probability**, which means that we have observed B, and want to update the probability estimate for A.

Example A.2. Example of the Bayes rule in action
An English-speaking tourist visits a city whose language is not English. A local friend tells him that 1 in 10 natives speak English, 1 in 5 people in the streets are tourists and that half of the tourists speak English. Our visitor stops comeone in the street and finds that this person speaks English. What is the probability that this person is a tourist?
We have

$$P(EN|Tourist) = \frac{1}{2},\ P(EN|Local) = \frac{1}{10},\ P(Tourist) = \frac{1}{5}$$

We want to update our knowledge about the event of this person being a tourist. The prior probability is $\frac{1}{5}$, but since we know that this person speaks english, we have new information useful for updating the probability. First, the total probability of someone speaking english is

$$P(EN) \overset{sum\ rule+product\ rule}{=} P(EN|Tourist) P(Tourist) + P(EN|Local) P(Local) = \frac{1}{2}\frac{1}{5} + \frac{1}{10}\frac{4}{5} = \frac{9}{50}.$$

Now, the a posteriori probability of the person being a tourist, now that we know that he speaks english is

$$P(tourist|EN) = \frac{P(EN|Tourist) P(Tourist)}{P(EN)} = \frac{\frac{1}{2}\frac{1}{5}}{\frac{9}{50}} = \frac{5}{9}.$$

As we can see (and as we should expect), knowing that the person speaks english, our confidence that he is a tourist increases.

A.2 Bayes rule in the context of learning

As have been explained, Bayes rule allows us to reason about hypotheses from data:

$$P\left(hypothesis|data\right)=\frac{P\left(data|hypothesis\right)P\left(hypothesis\right)}{P\left(data\right)}.$$

In the jargon of parameters and datasets, this is: *let θ be a random variable with support Θ, and let D be the data that has been observed. Then, it is*

$$P\left(\theta|D\right)=\frac{P\left(D|\theta\right)P\left(\theta\right)}{P\left(D\right)}=\frac{P\left(D|\theta\right)P\left(\theta\right)}{\int_{\Theta}P\left(D|\theta\right)P\left(\theta\right)d\theta}.$$

Here, $P\left(\theta\right)$ is the **prior distribution of** θ. This means it is the distribution that we assume, before observing D. $P\left(D|\theta\right)$ is the **likelihood** of θ: the probability of observing D if the parameters are θ. $P\left(D\right)$ is the **evidence** or expected likelihood. $P\left(\theta|D\right)$ is the **posterior distribution of** θ, our quantity of interest, expressing what we know about θ after having observed D.
Thus, we can continue this line of thought to tackle a new way of creating a model: given some data D, find the best possible values for the unknown parameters θ, so that the posterior or its likelihood is maximized. There are, basically, two different approaches:

- **Maximum likelihood**: in this case, we want to choose θ in order to maximize its likelihood:

$$\theta_{ML}=\arg\max_{\theta}P\left(D|\theta\right).$$

- **Maximum a posteriori**: in this case, we take into account a prior distribution for θ and estimate its value maximizing its posterior distribution:

$$\theta_{MAP}=\arg\max_{\theta}P\left(D|\theta\right)P\left(\theta\right).$$

A.3 Maximum likelihood estimation

Given a sample $D=\{x_1,...,x_n\}$, where x_i are independent are identically distributed observations from a random variable X, following a distribution $p\left(X;\theta\right)$, with θ the paremeters of the distribution. Our objective will be to obtain the best values for θ, according to our data, assuming some special form for p.
For this, the likelihood can be used: since the x_i are independent, the probability of having the sample D is

$$p\left(D;\theta\right)=\prod_{i=1}^{n}p\left(x;\theta\right).$$

The **likelihood function** is thus defined as

$$L\left(\theta\right)=p\left(D;\theta\right).$$

Note that this is done with a fixed data D, so it is not a probability distribution, but a function of θ. This way, the maximum likelihood estimator for θ is given by

$$\theta_{ML}=\arg\max_{\theta}L\left(\theta\right).$$

There is a numerical issue here, though: as we are multiplying probabilities, which are values between 0 and 1, and we are likely to be multiplying many of them, we expect to obtain values very close to 0, which can lead to underflow in computations. Thus, it is convenient to use the **log-likelihood**:

$$\theta_{ML}=\arg\max_{\theta}L\left(\theta\right)=\arg\max_{\theta}\left[\log L\left(\theta\right)\right]=\arg\min_{\theta}\left[-\log L\left(\theta\right)\right].$$

Example A.3. Compute the maximum likelihood estimator (MLE) for univariate Gaussian distribution.

For this, we assume the data $D = \{x_1, ..., x_n\}$, where each $x_i \sim \mathcal{N}\left(\mu, \sigma^2\right)$. In this situation, the parameters are μ and σ^2. The likelihood function is

$$\begin{aligned} L\left(D; \mu, \sigma^2\right) =& \prod_{i=1}^{n} f\left(x; \mu, \sigma^2\right) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x_i-\mu)^2}{\sigma^2}} \\ =& \frac{1}{\left(2\pi\sigma^2\right)^{\frac{n}{2}}} e^{-\frac{1}{2\sigma^2}\sum (x_i-\mu)^2}. \end{aligned}$$

Now, the log-likelihood is

$$\begin{aligned} logL\left(D; \mu, \sigma^2\right) = \log\left(L\left(D; \mu, \sigma^2\right)\right) =& \log\left(\frac{1}{\left(2\pi\sigma^2\right)^{\frac{n}{2}}} e^{-\frac{1}{2\sigma^2}\sum (x_i-\mu)^2}\right) \\ =& \log\left(\frac{1}{\left(2\pi\sigma^2\right)^{\frac{n}{2}}}\right) + \log\left(e^{-\frac{1}{2\sigma^2}\sum (x_i-\mu)^2}\right) \\ =& \log(1) - \log\left(\left(2\pi\sigma^2\right)^{\frac{n}{2}}\right) - \frac{1}{2\sigma^2}\sum (x_i-\mu)^2 \log(e) \\ =& 0 - \frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum (x_i-\mu)^2. \end{aligned}$$

At this point, to obtain the MLE, we need to maximize this function with respect to μ and σ^2:

$$\frac{\partial}{\partial\mu} logL = -\frac{1}{\sigma^2}\sum (x_i-\mu) = 0 \iff \sum (x_i-\mu) = 0 \iff \sum x_i - n\mu = 0 \iff \mu_{MLE} = \frac{\sum x_i}{n},$$

$$\begin{aligned} \frac{\partial}{\partial\sigma^2} logL = -\frac{n}{2}\frac{1}{2\pi\sigma^2}2\pi + \frac{1}{2\sigma^4}\sum (x_i-\mu)^2 = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum (x_i-\mu)^2 = 0 \\ \iff \frac{1}{2\sigma^4}\sum (x_i-\mu)^2 = \frac{n}{2\sigma^2} \iff \sum (x_i-\mu)^2 = n\sigma^2 \iff \sigma^2 = \frac{\sum (x_i-\mu)^2}{n}. \end{aligned}$$

Now, we substitute the value obtained for μ.

$$\sigma^2_{MLE} = \frac{\sum \left(x_i - \mu_{MLE}\right)^2}{n}.$$

Example A.4. Compute the MLE for a Bernoulli distribution.
Now the observations are the results of n coin tosses. We have to compute the parameter p_{ML} of the Bernoulli random variable, whose probability function is given by

$$f(x) = p^x (1-p)^{1-x}, \ p \in (0,1), \ x \in \{0,1\}.$$

Now, we have $D = \{x_1, ..., x_n\} \subset \{0,1\}^n$. The likelihood function is

$$L(D; p) = \prod_{i=1}^{n} f(x_i) = \prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i} = p^{\sum x_i}(1-p)^{n-\sum x_i}.$$

We differentiate:

$$\begin{aligned} \frac{\partial}{\partial p} L =& \sum x_i p^{\sum x_i - 1}(1-p)^{n-\sum x_i} - p^{\sum x_i}\left[n - \sum x_i\right](1-p)^{n-\sum x_i - 1} \\ =& p^{\sum x_i - 1}(1-p)^{n-\sum x_i - 1}\left[\sum x_i (1-p) - p\left(n - \sum x_i\right)\right] \\ =& p^{\sum x_i - 1}(1-p)^{n-\sum x_i - 1}\left[\sum x_i - \cancel{p\sum x_i} - pn + \cancel{p\sum x_i}\right] \\ =& p^{\sum x_i - 1}(1-p)^{n-\sum x_i - 1}\left[\sum x_i - pn\right]. \end{aligned}$$

This derivative is zero if and only if

$$\sum x_i - pn = 0 \iff p = \frac{\sum x_i}{n} = \overline{x}.$$

A.4 Properties of estimators

Definition A.1. If we have a dataset $D = \{x_1, ..., x_n\}$ where $x_i \sim X$ and X is a random variable, we call **estimator** a function $h : \mathbb{R}^n \to \mathbb{R}^k$.

Usually, we focus on estimators that tell us something about the underlying distribution X. For example, it is usual to assume that X belongs to a certain family of random variables $X \in F(\theta)$, so that we need to estimate the parameters $\theta = (\theta_1, ..., \theta_k)$.
There are some properties that are considered desirable for an estimator to have:

- **Unbiasedness**: an estimator $\hat{\theta}$ is unbiased if in the long run it takes the value of estimated parameter. If we define the **bias** of an estimator as

$$Bias\left[\hat{\theta}\right] = E\left[\hat{\theta}\right] - \theta,$$

  then, the estimator is unbiased if

$$Bias\left[\hat{\theta}\right] = 0.$$

- **Low variance**: the variance of an estimator tells us how sensitive it is to variations in the input data D:

$$Var\left[\hat{\theta}\right] = E\left[\left(\hat{\theta} - E\left[\hat{\theta}\right]\right)^2\right] = E\left[\hat{\theta}^2\right] - E\left[\hat{\theta}\right]^2.$$

  In the case that

$$Var\left[\hat{\theta}_1\right] < Var\left[\hat{\theta}_2\right],$$

  we say that $\hat{\theta}_1$ is **more efficient** than $\hat{\theta}_2$.

  The **Cramer-Rao bound** gives a theoretical lower bound to the variance of an estimator, under some hypotheses that are usually assumed true[13]:

$$Var\left(\hat{\theta}\right) \geq \frac{\left(\frac{\partial E\left[\hat{\theta}\right]}{\partial \theta}\right)^2}{E\left[\left(\frac{\partial \ln L}{\partial \theta}\right)^2\right]}.$$

- **Efficiency**: an estimator is efficient if:
  - It is unbiased.
  - There is no other unbiased estimator with lower variance.
- **Consistency**: a sequence of estimators $\left\{\hat{\theta}_n\right\}_{n\in\mathbb{N}}$ is consistent if it converges in probability to the true value of the parameter θ:

$$\forall \varepsilon > 0, \lim_{n\to\infty} P\left(\left|\theta - \hat{\theta}\right| < \varepsilon\right) = 1.$$

  Remark A.1. If the bias and the variance of an estimator tend to 0 with n, then it is consistent.

- **Mean squeared error**: the mean squared error of an estimator is

$$MSE\left(\hat{\theta}\right) = E\left[\left(\theta - \hat{\theta}\right)^2\right],$$

  which is a value that we seek to minimize.

[13] For further information one could read my notes from a course in Statistics (in Spanish), [1].

Example A.5. Show that

$$MSE\left(\hat{\theta}\right) = Bias\left[\hat{\theta}\right]^2 + Var\left[\hat{\theta}\right].$$

Let's start from the definition:

$$\begin{aligned}
MSE\left(\hat{\theta}\right) = E\left[\left(\theta-\hat{\theta}\right)^2\right] =& E\left[\theta^2 - 2\theta\hat{\theta} + \hat{\theta}^2\right] = E\left[\theta^2\right] - 2\theta E\left[\hat{\theta}\right] + E\left[\hat{\theta}^2\right] \\
=& \theta^2 - \theta E\left[\hat{\theta}\right] - \theta E\left[\hat{\theta}\right] + E\left[\hat{\theta}^2\right] - E\left[\hat{\theta}\right]^2 + E\left[\hat{\theta}\right]^2 \\
=& \theta\left(\theta - E\left[\hat{\theta}\right]\right) - \theta E\left[\hat{\theta}\right] + E\left[\hat{\theta}^2\right] - E\left[\hat{\theta}\right]^2 + E\left[\hat{\theta}\right]^2 \\
=& Var\left[\hat{\theta}\right] - \theta \cdot Bias\left[\hat{\theta}\right] + E\left[\hat{\theta}\right]\left(E\left[\hat{\theta}\right] - \theta\right) \\
=& Var\left[\hat{\theta}\right] + Bias\left[\hat{\theta}\right]\left(E\left[\hat{\theta}\right] - \theta\right) \\
=& Var\left[\hat{\theta}\right] + Bias\left[\hat{\theta}\right]^2.
\end{aligned}$$

Example A.6. Compute the bias and the variance of the ML estimates (μ, σ^2) of an univariate Gaussian. Show that σ_{ML} is biased and that we can correct its biasedness by using a different estimator

$$\hat{\sigma}^2 = \frac{n}{n-1}\sigma^2_{ML}.$$

Compute the bias and the variance of this new estimator.
Let's start with μ:

$$\begin{aligned}
Bias\left(\mu_{MLE}\right) =& Bias\left(\frac{\sum x_i}{n}\right) = E\left[\frac{\sum x_i}{n}\right] - E\left[X\right] = \frac{1}{n}E\left[\sum x_i\right] - E\left[X\right] = \frac{1}{n}\sum E\left[x_i\right] - E\left[X\right] \\
=& \frac{1}{n}\sum E\left[X\right] - E\left[X\right] = \frac{n}{n}E\left[X\right] - E\left[X\right] = 0,
\end{aligned}$$

$$\begin{aligned}
Variance\left(\mu_{MLE}\right) =& E\left[\left(\frac{\sum x_i}{n}\right)^2\right] - E\left[\frac{\sum x_i}{n}\right]^2 \\
=& \frac{1}{n^2}E\left[\left(\sum x_i\right)^2\right] - \frac{1}{n^2}E\left[\sum x_i\right]^2 \\
=& \frac{1}{n^2}E\left[\sum x_i^2 + \sum_{i\neq j} x_i x_j\right] - \frac{1}{n^2}\left(\sum E\left[x_i\right]\right)^2 \\
=& \frac{1}{n^2}\left(\sum E\left[x_i^2\right] + \sum_{i\neq j} E\left[x_i x_j\right]\right) - \frac{1}{n^2}\left(\sum E\left[X\right]\right)^2 \\
=& \frac{1}{n^2}\left(\sum E\left[X^2\right] + \sum_{i\neq j} E\left[x_i\right]E\left[x_j\right]\right) - \frac{1}{n^2}n^2 E\left[X\right]^2 \\
=& \frac{1}{n^2}\left(nE\left[X^2\right] + \sum_{i\neq j} E\left[X\right]^2\right) - E\left[X\right]^2 \\
=& \frac{E\left[X^2\right] + n\left(n-1\right)E\left[X\right]^2 - n^2 E\left[X\right]^2}{n^2} \\
=& \frac{E\left[X^2\right] - E\left[X\right]^2}{n^2} = \frac{Var\left[X\right]}{n^2}.
\end{aligned}$$

Now σ^2_{MLE}:

$$\begin{aligned}
Bias\left(\sigma^2_{MLE}\right) =& Bias\left(\frac{\sum\left(x_i-\mu_{MLE}\right)^2}{n}\right) = E\left[\frac{\sum\left(x_i-\mu_{MLE}\right)^2}{n}\right] - Var\left[X\right] \\
=& \frac{1}{n}E\left[\sum x_i^2 - 2\mu_{MLE}\sum x_i + n\mu^2_{MLE}\right] - Var\left[X\right] \\
=& E\left[X^2\right] - \frac{2}{n}E\left[\mu_{MLE}\right]nE\left[X\right] + E\left[\mu^2_{MLE}\right] - Var\left[X\right] \\
=& E\left[X^2\right] - 2E\left[X\right]^2 + E\left[\left(\frac{\sum x_i}{n}\right)^2\right] - Var\left[X\right] \\
=& \cancel{E\left[X^2\right] - E\left[X\right]^2} - E\left[X\right]^2 + E\left[\left(\frac{\sum x_i}{n}\right)^2\right] - \cancel{Var\left[X\right]} \\
=& \frac{1}{n^2}\left(nE\left[X^2\right] + \sum_{i\neq j}E\left[X\right]^2\right) - E\left[X\right]^2 \\
=& \frac{nE\left[X^2\right] + n\left(n-1\right)E\left[X\right]^2 - n^2E\left[X\right]^2}{n^2} \\
=& \frac{nE\left[X^2\right] - nE\left[X\right]^2}{n^2} = \frac{Var\left[X\right]}{n}.
\end{aligned}$$

$$\begin{aligned}
Bias\left(\hat{\sigma}^2\right) = Bias\left(\frac{n}{n-1}\sigma^2_{MLE}\right) =& Bias\left(\frac{\sum\left(x_i-\mu_{MLE}\right)^2}{n-1}\right) = E\left[\frac{\sum\left(x_i-\mu_{MLE}\right)^2}{n-1}\right] - Var\left[X\right] \\
=& \frac{1}{n-1}E\left[\sum x_i^2 - 2\mu_{MLE}\sum x_i + n\mu^2_{MLE}\right] - Var\left[X\right] \\
=& \frac{n}{n-1}E\left[X^2\right] - \frac{2}{n-1}E\left[\mu_{MLE}\right]nE\left[X\right] + \frac{n}{n-1}E\left[\mu^2_{MLE}\right] - Var\left[X\right] \\
=& \frac{n}{n-1}E\left[X^2\right] - \frac{2n}{n-1}E\left[X\right]^2 + \frac{\cancel{n}}{n-1}\frac{1}{n^{\cancel{2}}}\left(nE\left[X^2\right] + \sum_{i\neq j}E\left[X\right]^2\right) - Var\left[X\right] \\
=& \frac{n^2E\left[X^2\right] - 2n^2E\left[X\right]^2 + nE\left[X^2\right] + n\left(n-1\right)E\left[X\right]^2 - n\left(n-1\right)Var\left[X\right]}{n\left(n-1\right)} \\
=& \frac{n^2Var\left[X\right] - n^2E\left[X\right]^2 + nE\left[X^2\right] + n^2E\left[X\right]^2 - nE\left[X\right]^2 - n^2Var\left[X\right] + nVar\left[X\right]}{n\left(n-1\right)} \\
=& 0.
\end{aligned}$$

And we see how this one is, in fact, unbiased.

A.5 Maximum a posteriori estimation

MAP (maximum a posteriori) estimation is a method of estimating the parameters of a statistical model by finding the parameter values that maximize the posterior probability distribution of the parameters, given the observed data and a prior probability distribution over the parameters. In contexts where the amount of data is limited or noisy, incorporating prior knowledge or beliefs can help to produce more stable and accurate estimates. The prior distribution serves as a regularization term, allowing us to control the degree of influence that the prior has on the estimate:

$$\hat{\theta}_{MAP} = \arg\max_\theta P\left(\theta|D\right) = \arg\max_\theta \frac{P\left(D|\theta\right)P\left(\theta\right)}{P\left(D\right)} = \arg\max_\theta P\left(D|\theta\right)P\left(\theta\right),$$

where the denominator can be ignored because it is constant for all possible θ, so it does not change the arg max.

Example A.7. Find the MAP estimate for μ of an univariate Gaussian $X \sim \mathcal{N}\left(\mu, \sigma^2\right)$ with Gaussian prior distribution for $\mu \sim \mathcal{N}\left(\mu_0, \sigma_0^2\right)$, where σ, σ_0 and μ_0 are assumed to be known.
Let $D = \{x_1, ..., x_n\}$ where $x_i \sim X$, then

$$\begin{aligned} P\left(\mu|D\right) =& \frac{P\left(D|\mu\right)P\left(\mu\right)}{P\left(D\right)} = \frac{\prod_{i=1}^{n} P\left(x_i|\mu\right)P\left(\mu\right)}{P\left(D\right)} = \frac{\prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{\left(x_i-\mu\right)^2}{\sigma^2}} \cdot \frac{1}{\sqrt{2\pi\sigma_0^2}} e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}}}{P\left(D\right)} \\ =& \frac{\frac{1}{\left(2\pi\sigma^2\right)^{\frac{n}{2}}} \frac{1}{\sqrt{2\pi\sigma_0^2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2} \cdot e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}}}{P\left(D\right)}, \end{aligned}$$

so we want to maximize

$$f\left(\mu\right) = e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2} \cdot e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}},$$

or, its log function

$$g\left(\mu\right) = \log f\left(\mu\right) = -\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2 - \frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}.$$

Thus,

$$\begin{aligned} \frac{d}{d\mu}g\left(\mu\right) =& \frac{1}{2\sigma^2}\sum_{i=1}^{n} 2\left(x_i-\mu\right) - \frac{1}{2}\frac{2\left(\mu-\mu_0\right)}{\sigma_0^2} \\ =& \frac{\sum\left(x_i-\mu\right)}{\sigma^2} - \frac{\mu-\mu_0}{\sigma_0^2} \\ =& \frac{\sum x_i - n\mu}{\sigma^2} - \frac{\mu-\mu_0}{\sigma_0^2} \\ =& \frac{\sum x_i}{\sigma^2} - \frac{n\mu\sigma_0^2+\sigma^2\mu}{\sigma^2\sigma_0^2} + \frac{\mu_0}{\sigma_0^2} \\ =& \frac{\sum x_i}{\sigma^2} - \mu\frac{n\sigma_0^2+\sigma^2}{\sigma^2\sigma_0^2} + \frac{\mu_0}{\sigma_0^2} \end{aligned}$$

and this is zero if and only if

$$\mu_{MAP} = \frac{\sigma^2\sigma_0^2}{\sigma^2+n\sigma_0^2}\left(\frac{\sum x_i}{\sigma^2} + \frac{\mu_0}{\sigma_0^2}\right) = \frac{\sigma^2\sigma_0^2}{\sigma^2+n\sigma_0^2}\left(\frac{\sigma_0^2\sum x_i + \sigma^2\mu_0}{\sigma^2\sigma_0^2}\right) = \frac{\sigma_0^2\sum x_i + \sigma^2\mu_0}{\sigma^2+n\sigma_0^2}.$$

A different way to do this is that we have

$$P\left(\mu|D\right) = \frac{\frac{1}{\left(2\pi\sigma^2\right)^{\frac{n}{2}}} \frac{1}{\sqrt{2\pi\sigma_0^2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2} \cdot e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}}}{P\left(D\right)} \propto e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2} \cdot e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}},$$

and we want (μ_n, σ_n) such that

$$P\left(\mu|D\right) \propto e^{-\frac{\left(\mu-\mu_n\right)^2}{2\sigma_n^2}}$$

and $P\left(\mu|D\right) \sim N\left(\mu_n, \sigma_n^2\right)$, so we can solve

$$e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i-\mu\right)^2} \cdot e^{-\frac{1}{2}\frac{\left(\mu-\mu_0\right)^2}{\sigma_0^2}} = e^{-\frac{\left(\mu-\mu_n\right)^2}{2\sigma_n^2}},$$

obtaining

$$\mu_n = \frac{\sigma_0^2\sum x_i + \sigma^2\mu_0}{\sigma^2+n\sigma_0^2}$$

and

$$\sigma_n = \frac{\sigma^2\sigma_0^2}{\sigma^2+n\sigma_0^2}.$$

Example A.8. The maximum likelihood estimate of p for a Bernoulli r.v. X is given by

$$p_{ML} = \frac{\sum x_i}{n}.$$

If we have $K > 2$ outcomes, then we have the categorical distribution, also known as multinoulli or generalized Bernoulli, which has support $\{1, ..., K\}$. Its parameters are $p = (p_1, ..., p_K)$, representing the probability of observing each of the possible outcomes, with $p_i \in [0, 1], \forall i$ and $\sum p_i = 1$.
It is convenient to use the *one-of-K encoding* (also called one-hot encoding) for each outcome. Thus, the pmf of this distribution becomes

$$p(x) = \prod_{i=1}^{K} p_i^{x_i}.$$

Now, given a sample $D = \{x_1, ..., x_n\}$ of possible outcomes for a multinoulli r.v. X, the maximum likelihood estimate for p is

$$\hat{p}_k = \frac{1}{n} \sum x_{ik},$$

for each $k \in \{1, ..., K\}$. We can write this compactely as

$$\hat{p} = \frac{1}{n} \sum x_i.$$

If some category k is not present in our sample, then its corresponding ML estimate is going to be 0. These 0-estimates are problematic in predictive applications, because unseen outcomes in the training data are considered to be impossible to happen, and thus can never be predicted. To avoid this, **pseudocounts** are used instead. These represent prior knowledge in the form of (imagined) counts c_k for each category k. The idea is to assume that the data is augmented with our pseudocounts, and then we estimate using maximum likelihood over the augmented data, namely

$$\hat{p} = \frac{c + \sum_i x_i}{n + \sum_k c_k},$$

so the k-th parameter is

$$\hat{p}_k = \frac{c_k + \sum_i x_{ik}}{n + \sum_k c_k}.$$

As an example, imagine that we obtain a sample from a die: $\{1, 3, 5, 4, 4, 6\}$. If the vector of pseudocounts is $c = (1, 1, 1, 1, 1, 1)$, then the estimates are

$$\hat{p}_1 = \frac{1 + 1}{6 + 6} = \frac{1}{6},$$

$$\hat{p}_2 = \frac{1}{6 + 6} = \frac{1}{12},$$

and so on. Notice that although 2 has not been observed in D, its probability estimate is not 0. This special case where all pseudocounts are 1 is known as **Laplace smoothing**.
Prove that using maximum likelihood with pseudocounts corresponds to a MAP estimate with Dirichlet prior with parameters $(c_1 + 1, ..., c_K + 1)$.
A Dirichlet distribution, $Dir(c_1 + 1, ..., c_K + 1)$ has the density function:

$$f(x) = \frac{\Gamma\left(\sum (c_i + 1)\right)}{\prod \Gamma(c_i + 1)} \prod x_i^{c_i} = \frac{\Gamma\left(\sum c_i + K\right)}{\prod c_i!} \prod x_i^{c_i} = \frac{\left(\sum c_i + K - 1\right)!}{\prod c_i!} \prod x_i^{c_i}.$$

To compute the MAP estimate, we use

$$P(D|p) P(p) = \prod_i P(x_i|p) P(p) = \prod_i \prod_k p_k^{x_{ik}} \cdot \frac{\left(\sum c_i + K - 1\right)!}{\prod c_i!} \prod_k p_k^{c_k}.$$

Thus, we want to minimize

$$f(p_1, ..., p_K) = \prod_{i,k} p_k^{x_{ik}} \prod_k p_k^{c_k},$$

or, equivalently, its log

$$g\left(p_1,...,p_K\right)=\log f\left(p_1,...,p_K\right)=\sum_{ik}x_{ik}\log\left(p_k\right)+\sum_k c_k\log\left(p_k\right).$$

We have

$$\begin{array}{ll}\min & g\left(p_1,...,p_K\right)\\ s.a. & \sum p_i=1\\ & p_i\in[0,1],\forall i\end{array}.$$

The lagrangian is

$$L=g-\lambda\left(\sum p_i-1\right),$$

so that

$$\frac{\partial}{\partial p_k}L=\frac{\partial}{\partial p_k}g-\lambda=\sum_i x_{ik}\frac{1}{p_k}+c_k\frac{1}{p_k}-\lambda=0\iff\lambda=\frac{\sum_i x_{ik}+c_k}{p_k}\iff p_k=\frac{\sum_i x_{ik}+c_k}{\lambda},$$

and then

$$1=\sum_k p_k=\sum_k\frac{\sum_i x_{ik}+c_k}{\lambda}=\frac{1}{\lambda}\left(n+\sum_k c_k\right)\iff\lambda=n+\sum_k c_k.$$

Finally, substituting back, we obtain

$$p_k=\frac{c_k+\sum_i x_{ik}}{n+\sum_k c_k},$$

as we wanted!

A.6 Bayesian Learning

In the Bayesian Learning framework, instead of working with point estimates of our unknown parameter variables θ, we work with the whole posterior distribution $P\left(\theta|D\right)$. In this case, learning is the process by which starting with some prior belief about the parameters and when facing some observations in the form of a dataset D, we update our belief about the possible values for our parameters in the form of the posterior distribution. This process is iterated over newly received data, so it can be viewed as a sequential process, in which Bayes is invoked each time we need a new posterior $P\left(\theta|D_1,D_2,...\right)$.

Example A.9. Bayesian learning in Matlab

Bayesian Learning

We are going to obtain data from a distribution $\mathcal{N}(5,2)$.

```
mu_true = 5;
sigma = 2;

num_samples = 10;

rng('default') % For reproducibility
data = mu_true + sigma_true * randn(num_samples, 1);

% Set up the x-axis for plotting
x = linspace(-5, 10, 1000);

% Plot the true distribution
figure;
hold on;
plot(x, normpdf(x, mu_true, sigma_true), 'LineWidth', 2);
title('Bayesian Learning');
xlabel('x');
```

```
ylabel('Probability Density');
xlim([-5, 10]);
legend('N(5, 2)');
```

Our prior is going to be a standard normal $\mu \sim \mathcal{N}(0, 1)$ and we are going to assume that σ is well approximated by the sample variance.

```
mu_prior = 0;
sigma_prior = 1;
```

Now, we are going to update our beliefs with the obtained datapoints, and the formulas for updating are:

$$\mu_n = \frac{\sigma_0^2 \sum x_i + \sigma^2 \mu_0}{\sigma^2 + n\sigma_0^2}$$

and

$\sigma_n = \frac{\sigma^2 \sigma_0^2}{\sigma^2 + n\sigma_0^2}$.

```
mu_posterior = (sigma_prior^2*(sum(data))+var(data)*mu_prior)/(var(data)+num_samples*sigma_prior^2)
```

```
mu_posterior = 2.7734
```

```
sigma_posterior = (var(data)*sigma_prior^2)/(var(data)+num_samples*sigma_prior^2)
```

```
sigma_posterior = 0.5561
```

```
plot(x, normpdf(x, mu_posterior, sqrt(var(data))), 'LineWidth', 2);
xlabel('x');
ylabel('Probability Density');

sigma_prior = sigma_posterior;
mu_prior = mu_posterior;

data = mu_true + sigma_true * randn(num_samples, 1);

mu_posterior = (sigma_prior^2*(sum(data))+var(data)*mu_prior)/(var(data)+num_samples*sigma_prior^2)
```

```
mu_posterior = 4.0179
```

```
sigma_posterior = (var(data)*sigma_prior^2)/(var(data)+num_samples*sigma_prior^2)
```

```
sigma_posterior = 0.2034
```

```
plot(x, normpdf(x, mu_posterior, sqrt(var(data))), 'LineWidth', 2);
xlabel('x');
ylabel('Probability Density');

sigma_prior = sigma_posterior;
mu_prior = mu_posterior;

num_samples = 500;
```

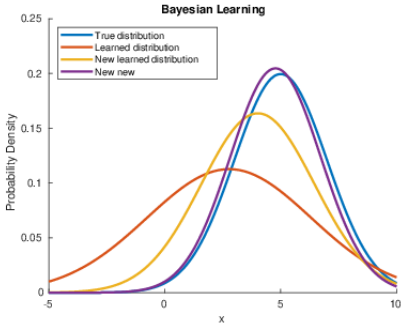
```
data = mu_true + sigma_true * randn(num_samples, 1);

mu_posterior = (sigma_prior^2*(sum(data))+var(data)*mu_prior)/(var(data)+num_samples*sigma_prior^2)
```

```
mu_posterior = 4.7770
```

```
sigma_posterior = (var(data)*sigma_prior^2)/(var(data)+num_samples*sigma_prior^2)
```

```
sigma_posterior = 0.0064
```

```
plot(x, normpdf(x, mu_posterior, sqrt(var(data))), 'LineWidth', 2);
xlabel('x');
ylabel('Probability Density');
leg = legend('True distribution', 'Learned distribution', 'New learned distribution', 'New new');
leg.Location = "northwest"
```




```
leg =
  Legend (True distribution, Learned distribution, New learned distribution, New new) with properties

         String: {'True distribution'  'Learned distribution'  'New learned distribution'  'New new'}
       Location: 'northwest'
    Orientation: 'vertical'
       FontSize: 9
       Position: [0.1483 0.7446 0.3565 0.1545]
          Units: 'normalized'

  Show all properties
```

A.6.1 Predictive posterior

When doing prediction in this framework, the whole distribution $P(\theta|D)$ is used. We view a prediction as a weighted average of all predictions each value for θ can make, weighted by its posterior probability (its expected value):

$$P(x'|D) = E_{\theta|D}[x] = \int_{\Theta} p(x'|\theta, D)\, P(\theta|D)\, d\theta.$$

Example A.10. An insightful example

Bayesian Learning: an insightful example

Suppose we have a dataset with X and Y values representing the relationship between the number of hours studied and test scores. Our true model is $y = 2X + 1$.

```
X = [1,2,3,4,4.5]';
y = [3,5,7,9,10]';
```

1) **Prior beliefs**: To start, we are searching for a linear relationship y=a*X+b. First, we need to define prior distributions for a and b. These distributions represent our initial beliefs about the values of a and b before seeing any data. A common choice is to use normal distributions with a mean of 0 and a large variance (e.g., 1000) to indicate a high level of uncertainty. For example, a~N(0,1000) and b~N(0,1000).

```
mu_prior = [0;0];
sigma_prior = 1000;
V_prior = sigma_prior^2*eye(2) %Covariance matrix for prior distributions
```

```
V_prior = 2x2
     1000000           0
           0     1000000
```

```
X_design = [X, ones(size(X))] %Design matrix [X,1]
```

```
X_design = 5x2
    1.0000    1.0000
    2.0000    1.0000
    3.0000    1.0000
    4.0000    1.0000
    4.5000    1.0000
```

2) **Likelihood function**: We need to define a likelihood function, which describes the probability of the observed data given the parameters a and b. In linear regression, we typically assume that the errors are normally distributed. Therefore, the likelihood function can be represented as $y_i \sim \mathcal{N}(aX_i + b, \sigma^2)$, where σ^2 is the variance of the error. We will assume it known, because the approaches to get it are a bit involved.

```
sigma_error = 1;
```

3) **Posterior distribution**: We update our beliefs about the coefficients a and b given the data by computing the posterior distribution. This is done by multiplying the prior distributions with the likelihood function and applying Bayes' theorem:

$P(a, b|X, y) = \frac{P(y|X,a,b)*P(a,b)}{P(Y|X)}$.

Note that the denominator is a normalizing constant that ensures the posterior distribution sums to 1.

$$V_{post} = (V_{prior}^{-1} + \frac{1}{\sigma_{error}^2} \cdot X_{design}^T * X_{design})^{-1}$$

$$\mu_{post} = V_{post} \cdot (V_{prior}^{-1} \cdot \mu_{prior} + \frac{1}{\sigma_{error}^2} \cdot X_{design}^T \cdot y)$$

```
V_posterior = inv(inv(V_prior) + (1 / sigma_error^2) * (X_design' * X_design))
```

```
V_posterior = 2x2
    0.1220   -0.3537
   -0.3537    1.2256
```

```
mu_posterior = V_posterior * (inv(V_prior) * mu_prior + (1 / sigma_error^2) * (X_design' * y))
```

```
mu_posterior = 2x1
    2.0000
    1.0000
```

4) **Making predictions**: To make predictions for new X values, we can use the posterior predictive distribution, which accounts for the uncertainty in the estimates of a and b.

```
% Number of samples
n_samples = 1000;

% Sample from the posterior distribution of a and b
ab_samples = mvnrnd(mu_posterior, V_posterior, n_samples);

% Prediction for new X value
X_new = 3.5;

% Generate predictions using the samples of a and b
Y_new_samples = zeros(n_samples, 1);
for i = 1:n_samples
    a_sample = ab_samples(i, 1);
    b_sample = ab_samples(i, 2);

    Y_new_mean_sample = [X_new, 1] * [a_sample; b_sample];
    Y_new_samples(i) = Y_new_mean_sample + sigma_error * randn;
end

% Compute summary statistics for the predictions
Y_new_mean = mean(Y_new_samples);
Y_new_std = std(Y_new_samples);

fprintf('Prediction for X = %d: Y = %.2f +/- %.2f\n', X_new, Y_new_mean, Y_new_std);
```

```
Prediction for X = 3.500000e+00: Y = 7.98 +/- 1.13
```

The true value according to the model is 8, and we have obtained 7.98 +- 1.13, which is pretty good!

References

[1] Jose A. Lorencio Abril. Apuntes de inferencia estadÃstica, 2021.

[2] Marta Arias. Machine learning. Lecture Notes.

[3] Marta Arias. Notes on probability theory, bayes theorem and bayesian learning. Lecture Notes.

[4] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.