Lecture 1: Basics of Machine Learning

DATA 602 @ UMBC

Masoud Soroush

August 28, 2022

1 Basics of Machine Learning

In today's lecture[1], we explain what machine learning is about, and what you should expect from this introductory course on machine learning. Moreover, we will review some basic mathematical and statistical concepts that will be used throughout the course. Specifically, a good understanding of the following areas in math and statistics will facilitate your understanding of machine learning algorithms:

- **Linear Algebra**
- **Calculus**
- **Basic Statistic**

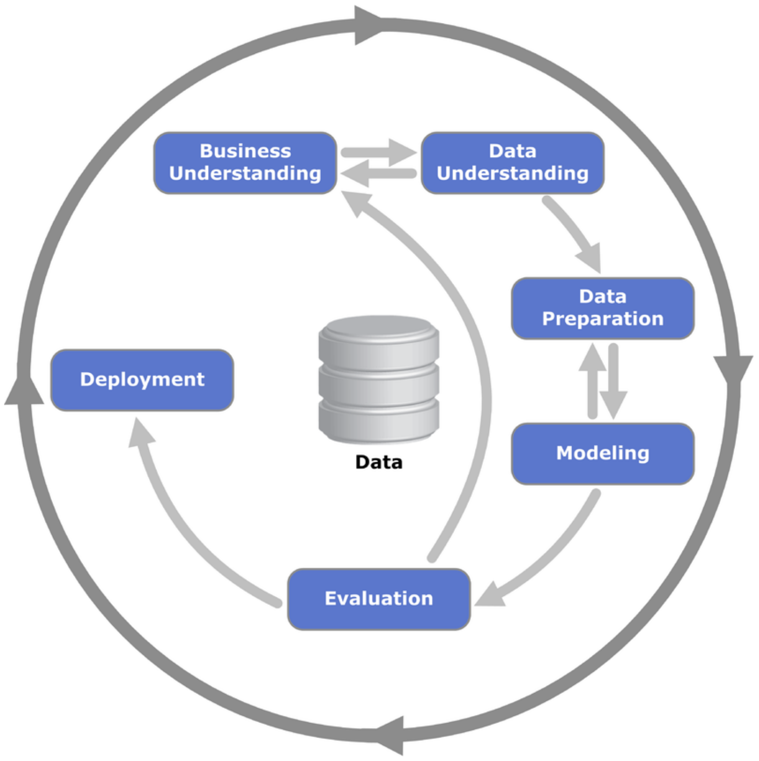
Today, we will review certain facts from each of the above areas that will be needed for our machine learning class. Specifically, we will review *eigenvalue problems* in linear algebra, *optimizations of functions* in calculus, and *hypothesis testing* in statistics.

1.1 What Is Machine Learning?

Before we explain what machine learning is, let us see where machine learning stands in the big scheme of things in data science. Recall from your DATA 601 the life cycle of data science projects. The following picture depicts different steps of the life cycle of data science projects. A data science project starts with a set of research questions to define goals and objectives to achieve. In the next step, you need to acquire a business understanding (domain knowledge) of your project in order to define your objectives from a business point of view. Note that your business objectives must be **SMART** (*i.e.* Specific, Measurable, Achievable, Relevant, and Time-bound). Once you have defined your objectives, you will then look for relevant data. The data must be relevant to your research questions and objectives. After identifying your data sources and acquiring the necessary understanding of your data, the next step is data preparation. This step includes the data wrangling process that you learned in DATA 601. The focus of this course is exactly on the

[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

next two steps of the life cycle, namely **Modeling** and **Evaluation**. Machine learning concerns these two steps.



OK, we now know where machine learning enters in the life cycle of data science projects, but we have not yet offered a definition of machine learning. Here we offer a definition of machine learning:

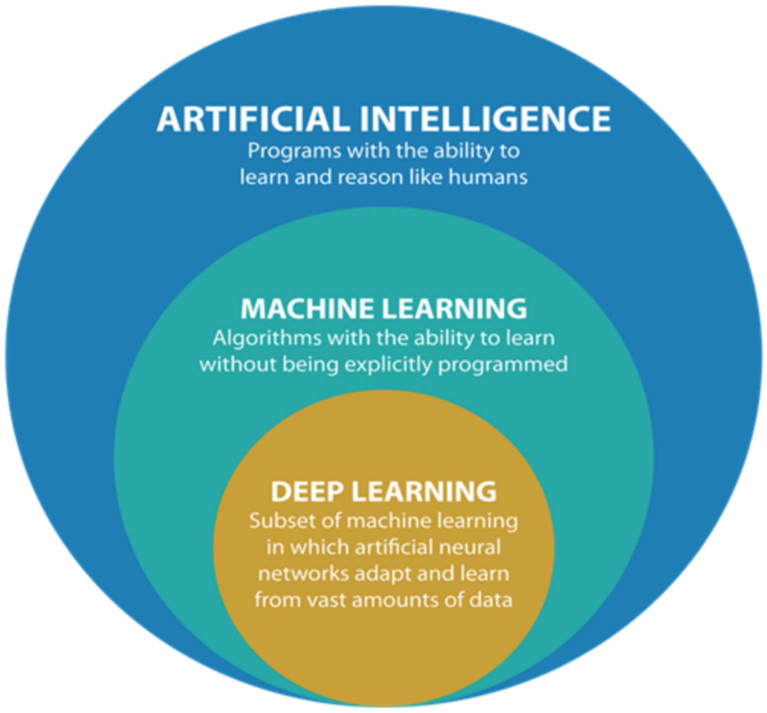
Machine Learning is a subcategory of artificial intelligence, that refers to the process by which computers develop pattern recognition, or the ability to continuously learn from and make predictions based on data, then make adjustments without being specifically programmed to do so.

Machine learning is innately data driven, and data is at the core of machine learning. The goal of machine learning is to design general-purpose methodologies to *extract valuable patterns from data*, ideally without much domain-specific expertise. To achieve this goal, we *design models* that are typically related to the process that generates data, similar to the dataset we are given. To adjust the parameters of the model to their optimal values, the *model has to learn from data*. A model is said *to learn from data* if its performance on a given task improves after the data is taken into account. The goal is to find good models that generalize well to yet unseen data(that we may care about in the future).

Learning can be introduced as a method to automatically find patterns and structure in data by optimizing the parameters of the model.

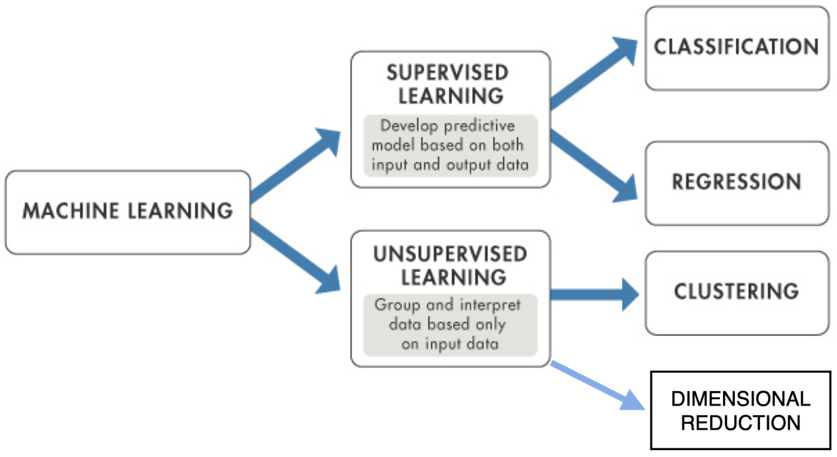
The following Venn diagram illustrates the relative situation of artificial intelligence, machine learning, and deep learning.

In statistical and machine learning, we deal with two major types of learning: **Supervised learning** and **Unsupervised learning**. In a supervised learning, we have a response variable (or **target variable**), whereas in an unsupervised learning, there is no response/target variable. Depending

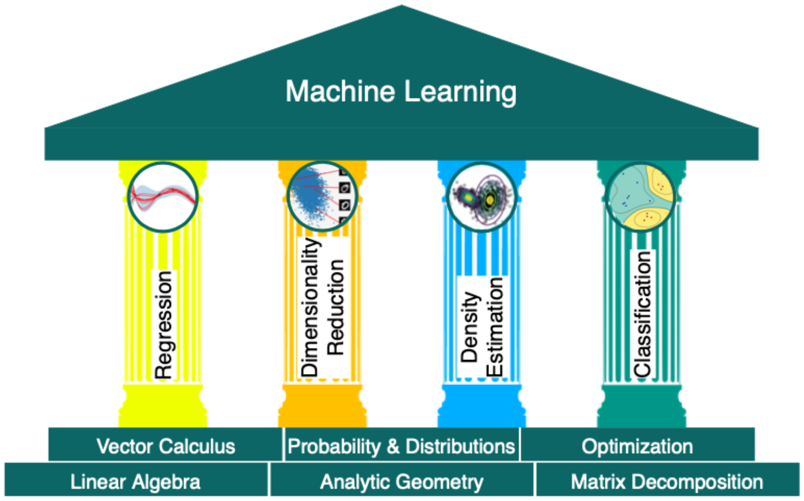

ARTIFICIAL INTELLIGENCE
Programs with the ability to learn and reason like humans
MACHINE LEARNING
Algorithms with the ability to learn without being explicitly programmed
DEEP LEARNING
Subset of machine learning in which artificial neural networks adapt and learn from vast amounts of data

on the nature of the target variable in a supervised learning, we have two types of machine learning algorithms. If the target variable is a **quantitative variable**, we'll have regression analysis, and if the target variable is **qualitative/categorical**, we will have a classification task. Roughly speaking, regression is the *process of predicting a value for a continuous target variable*, and classification is the *process of predicting a value for a categorical target variable*.

In unspuervised learning, there exists no target variable (or the target variable is dismissed). The goal is to analyze the *unlabeled data* and discover patterns in the dataset. Two of the common and popular algorithms used in unspuervised learning are **Clustering** and **Dimensional Reduction**. The following chart summarizes the two major types of machine learning.



In order to develop machine learning algorithms and obtain a solid understanding from their underlying mechanisms, one must be equipped with certain mathematical and statistical tools. In a nutshell, one should be familiar with basics of *calculus*, *linear algebra*, and *basic statistics*. In calculus, we specifically need to know basics of function optimization (We will go through it in this lecture). In linear algebra, we need to feel comfortable with matrix algebra and eigenvalue problems (We will briefly review basic facts in this lecture). Finally, it is important, as a data science student, to be fairly comfortable with the tools of basic statistics (In this lecture, we will review the hypothesis testing). The following figure[2] indicates some of the common mathematical and statistical tools used in machine learning.



Side Question: While the necessary mathematical and statistical prerequisites of machine learning algorithms were known for a long time, why did machine learning receive an apt attention only in the past few decades?

There are several important characteristics that one should consider when a machine learning algorithm is adopted:

- **Power and Expressibility:** Machine learning algorithms differ in the richness and complexity of the models they support. Greater expressive power provides the possibility of more accurate models, as well as the dangers of overfitting.
- **Interpretability:** Powerful algorithms sometimes produce models that are very accurate, but no human-readable explanation of why they work the way they do. Interpretability is an important property of a model, and some data scientists are happier to take a lesser-performing model they understand over a slightly more accurate one that they don't.
- **Ease of Use:** Certain machine learning methods feature relatively few parameters or decisions, meaning they easy to work with. In contrast, some machine learning algorithms provide much greater scope to optimize the performance with the proper settings.
- **Training Speed:** Machine learning algorithms differ greatly in how fast they fit the necessary parameters of the model, which determines how much training data you can afford to use

[2] Source: M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for Machine Learning*, Cambridge University Press (2020).

in practice. Don't forget that your computational power is limited!

- **Prediction Speed:** Machine learning algorithms differ in how fast they make classification decisions on a new queries.

The above characteristics will not be equally important in every data science problem you intend to solve. In each problem, you have to decide where to put your emphasis based on the goals of your project and your available resources.

1.2 Eigenvalue Problems in Linear Algebra

First, let us quickly review some basic facts about matrix algebra. Two matrices A and B can be added or subtracted only if they have the same number of columns and the same number of rows. Then, addition/subtraction is performed component-wise.

1.2.1 Matrix Multiplication

Matrix A can be multiplied by another matrix B to form AB if and only if the number of columns of A is the same as the number of rows of B. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then the product AB is an $m \times p$ matrix. We can calculate matrix multiplication through numpy and sympy libraries.

Example 1: Let $A = \begin{pmatrix} 1 & 3 & 4 \\ 2 & -1 & -2 \end{pmatrix}$ and $B = \begin{pmatrix} -2 & -1 & 4 & 5 \\ -3 & 4 & -1 & 5 \\ 6 & -4 & 7 & -2 \end{pmatrix}$. Find AB.

This calculation can be easily conducted through numpy:

```
[1]: # Import 'numpy' library

     import numpy as np

     A = np.array([[1, 3, 4], [2, -1, -2]])        # Define matrix 'A' as a numpy array
     B = np.array([[-2, -1, 4, 5], [-3, 4, -1, 5], [6, -4, 7, -2]])      # Define␣
      ↪matrix 'B' as a numpy array

     np.matmul(A, B)        # Calculate matrix 'AB' through 'matmul'
```

```
[1]: array([[ 13,  -5,  29,  12],
            [-13,   2,  -5,   9]])
```

Alternatively, we can perform the calculation through sympy:

```
[2]: # Import 'sympy' library

     from sympy import *

     mat_A = Matrix(A)     # Define matrix 'A'
```

```
mat_B = Matrix(B)     # Define matrix 'B'

display(mat_A, mat_B)  # Display matrices 'A' and 'B'
```

$$\begin{bmatrix}1 & 3 & 4\\ 2 & -1 & -2\end{bmatrix}$$

$$\begin{bmatrix}-2 & -1 & 4 & 5\\ -3 & 4 & -1 & 5\\ 6 & -4 & 7 & -2\end{bmatrix}$$

[3]:
```
mat_AB = mat_A*mat_B    # Matrix multiplication in sympy is done by '*'

display(mat_AB)         # Display the result of multiplication
```

$$\begin{bmatrix}13 & -5 & 29 & 12\\ -13 & 2 & -5 & 9\end{bmatrix}$$

1.2.2 Determinants and Inverse Matrix

Square matrices are particularly important in linear algebra, there are certain operations that can solely performed on square matrices. One of the key quantities that we would be interested in the context of square matrices is **determinant**. Let us see how we can calculate determinants through numpy and sympy libraries.

Example 2: Let $A = \begin{pmatrix}3 & 2 & -1 & 4\\ -2 & 3 & 5 & -3\\ 1 & 2 & -3 & 5\\ 4 & -3 & 1 & -2\end{pmatrix}$. Find $\mathrm{Det}(A)$.

Just like the previous example, we can conduct the calculation in either numpy or sympy. Let's first perform the calculation through numpy:

[4]:
```
A = np.array([[3, 2, -1, 4], [-2, 3, 5, -3], [1, 2, -3, 5], [4, -3, 1, -2]])  #␣
 ↪Define matrix 'A' as a numpy array

np.linalg.det(A)   # To calculate determinant, you need to call the 'linalg'␣
 ↪module of numpy
```

[4]:
```
12.000000000000048
```

Note that since numpy does the calculations numerically, you may sometimes see some annoying very small decimals attached to your actual result. In above the exact answer for determinant is 12. Now, let's see how to calculate the determinant through sympy:

[5]:
```
mat_A = Matrix(A)    # Define matrix 'A'

mat_A.det()          # calculate the determinant simply by joining '.det()' to␣
 ↪the name of matrix
```

[5]: 12

Note that not all square matrices are invertible. In order to see whether a given matrix B is invertible, we proceed as follows:

$$\boxed{\text{Matrix } B \text{ is invertible if and only if } \mathrm{Det}(B) \neq 0}$$

When $\mathrm{Det}(B) \neq 0$, the inverse matrix exists, and is denoted by B^{-1}. The inverse matrix satisfies the following property

$$BB^{-1} = B^{-1}B = \mathbb{1}\,, \tag{1}$$

where $\mathbb{1}$ is the identity matrix. If an invertible matrix is *not too large in size*, we can easily find the inverse of the matrix through numpy and sympy libraries.

Example 3: Let $B = \begin{pmatrix} 1 & -2 & 2 & -1 \\ 2 & -2 & 2 & 1 \\ 2 & -1 & 2 & -1 \\ 1 & -1 & 1 & 2 \end{pmatrix}$. Find the inverse of B.

Finding the inverse through numpy first:

[6]:
```
B = np.array([[1, -2, 2, -1], [2, -2, 2, 1], [2, -1, 2, -1], [1, -1, 1, 2]]) #␣
↪Define matrix 'B' as a numpy array

np.linalg.det(B)    # Calculate determinant of 'B'
```

[6]:
```
2.9999999999999996
```

Now that $\mathrm{Det}(B) \neq 0$, we know that matrix B is indeed invertible, and B^{-1} exists. To calculate its inverse, we take the advantage of linalg module of numpy:

[7]:
```
np.linalg.inv(B)    # Calculate inverse of 'B' through '.inv(matrix name)'
```

[7]:
```
array([[-1.        ,  1.66666667,  0.        , -1.33333333],
       [-0.        , -1.66666667,  1.        ,  1.33333333],
       [ 1.        , -2.66666667,  1.        ,  2.33333333],
       [ 0.        , -0.33333333,  0.        ,  0.66666667]])
```

Performing the calculation through sympy:

[8]:
```
mat_B = Matrix(B)  # Define matrix 'B'

mat_B.det()        # Calculate determinant of 'B'
```

[8]: 3

[9]:
```
Binv = mat_B.inv()   # Calculate inverse of 'B' through 'matrix_name.inv()'

display(Binv)        # Display inverse of 'B'
```

$$\begin{bmatrix} -1 & \frac{5}{3} & 0 & -\frac{4}{3} \\ 0 & -\frac{5}{3} & 1 & \frac{4}{3} \\ 1 & -\frac{8}{3} & 1 & \frac{7}{3} \\ 0 & -\frac{1}{3} & 0 & \frac{2}{3} \end{bmatrix}$$

Now, let's check and make sure that indeed $BB^{-1} = B^{-1}B = \mathbb{1}$.

```
[10]: display(B*Binv, Binv*B)  # Multiply 'B' with 'B^{-1}' in both orders and display␣
      ↪the result
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As expected, both BB^{-1} and $B^{-1}B$ result in the 4×4 identity matrix!

1.2.3 Eigenvalues and Eigenvectors

Let A be an $n \times n$ matrix. Consider the equation

$$A\,\vec{v} = \lambda\,\vec{v}\,, \tag{2}$$

where $\vec{v}$ is an n dimensional vector (*i.e.* $\vec{v} \in \mathbb{R}^n$) and λ is an scalar (*i.e.* real number). Any vector $\vec{v}$ that satisfies (2) is said to be an **eigenvector** of matrix A, and λ is the **eigenvalue** associated with eigenvector $\vec{v}$. Eigenvalues and eigenvectors of square matrices are extremely important characteristics that carry vital information about matrix A. Moreover, eigenvalues and eigenvectors are widely used for practical calculations in the context of linear algebra.

How do we find eigenvalues? In order to find eigenvalues of the $n \times n$ matrix A, one has to solve the following polynomial equation (known as the **characteristic equation** for matrix A) with variable λ

$$\mathrm{Det}(A - \lambda\mathbb{1}) = 0\,, \tag{3}$$

where $\mathbb{1}$ is the $n \times n$ identity matrix. To find the eigenvalues and eigenvectors, *we do not need to carry out any calculation by hand*! We can easily find eigenvalues and eigenvectors of a square matrix through numpy or sympy libraries.

Example 4: Find all eigenvalues and eigenvectors of $A = \begin{pmatrix} 1 & 0 & 2 \\ -2 & 2 & 2 \\ 1 & 0 & 3 \end{pmatrix}$. Find eigenvalues and eigenvectors of A.

As in previous examples, the necessary calculations for finding eigenvalues and eigenvectors can be performed in both numpy and sympy.

```
[11]: A = np.array([[1, 0, 2], [-2, 2, 2], [1, 0, 3]])   # Define matrix 'A'

      # 'linalg.eig(matrix_name)' calculates both eigenvalues and eigenvectors:

      print('Eigenvalues of A:', np.linalg.eig(A)[0], '\n')     # 'linalg.eig(A)[0]'␣
       ↪gives eigenvalues
      print('Eigenvectors of A:\n', np.linalg.eig(A)[1])        # 'linalg.eig(A)[1]'␣
       ↪gives eigenvectors
```

```
Eigenvalues of A: [2.         0.26794919 3.73205081]

Eigenvectors of A:
 [[ 0.         -0.52544259 -0.57310042]
 [ 1.         -0.82880701 -0.24222074]
 [ 0.          0.19232534 -0.78286973]]
```

```
[12]: # You can also use '.eigenvals(matrix_name)' in numpy to calculate eigenvalues

      np.linalg.eigvals(A)
```

```
[12]: array([2.        , 0.26794919, 3.73205081])
```

```
[13]: # Through 'sympy'

      mat_A = Matrix(A)   # Define matrix A

      mat_A.eigenvals()   # Calculating eigenvalues through 'sympy'
```

```
[13]: {2: 1, 2 - sqrt(3): 1, sqrt(3) + 2: 1}
```

```
[14]: mat_A.eigenvects()   # Calculating eigenvalues and eigenvectors simultaneously in␣
       ↪'sympy'
```

```
[14]: [(2,
        1,
        [Matrix([
         [0],
         [1],
         [0]])]),
       (2 - sqrt(3),
        1,
        [Matrix([
         [                -2/(-1 + sqrt(3))],
         [-(-2*sqrt(3) - 2)/(-3 + sqrt(3))],
         [                                1]])]),
       (sqrt(3) + 2,
        1,
        [Matrix([
         [             -2/(-sqrt(3) - 1)],
         [-(2 - 2*sqrt(3))/(sqrt(3) + 3)],
         [                             1]])])]
```

1.3 Function Optimization in Calculus

We briefly review the basic facts in optimizing both single variable and multivariable functions. In reality, you rarely deal with a single variable function, as you often have a set of features for your model. Nonetheless, to set the stage, we start by reviewing some basic facts about single variable functions.

1.3.1 Single Variable Functions

Extremizing functions is an important task done in calculus. We review function extremization very briefly in here. Suppose you have a single-variable function $f(x)$. To extremize the function we set the derivative of the function to zero:

$$f'(x) = \frac{df(x)}{dx} = 0\,. \tag{4}$$

The solutions to equation (4) are said to be the extreme points of function f. Suppose x_0 is an extreme point of f. To decide whether x_0 is a local maximum, or local minimum, we have to apply the *second derivative test*:

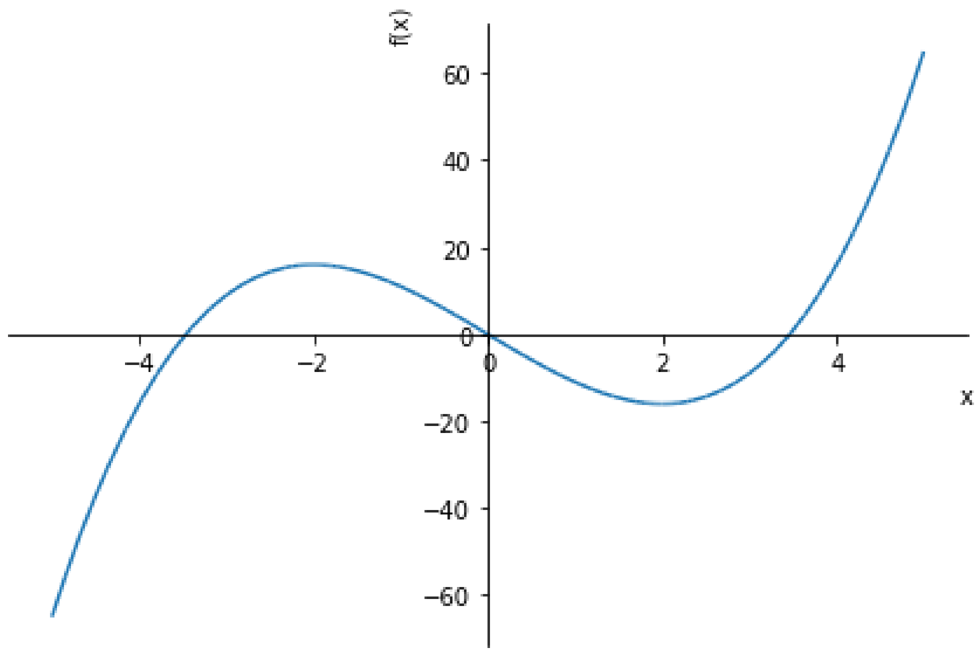
$$\left.\frac{d^2f}{dx^2}\right|_{x=x_0} = \begin{cases} f''(x_0) > 0, & x_0 \text{ is a local minimum,} \\ f''(x_0) < 0, & x_0 \text{ is a local maximum,} \\ f''(x_0) = 0, & \text{inconclusive.} \end{cases} \tag{5}$$

Example 5: Suppose $f(x) = x^3 - 12x$. Find local extrema of f and determine their nature.

[15]:
```
# Import sympy library

from sympy import *

x = symbols('x')            # Define 'x' as a symbol
f = Function('f')('x')      # Define function 'f' with variable 'x'

f = x**3 - 12*x             # Replacing the example's expression for 'f'

f                           # Display 'f'
```

[15]: $x^3 - 12x$

[16]:
```
p1 = plot(f, (x, -5, 5), show=True)  # Plot graph of function 'f'
```



As observed above, $f(x)$ attains one minimum at $x = 2$ and one maximum at $x = -2$.

[17]:
```
fp = f.diff(x)    # Take derivative of 'f' by 'function_name.diff(variable_name)'
fp                # Display the first derivative of 'f'
```

[17]: $3x^2 - 12$

[18]:
```
eq1 = Eq(fp, 0)      # Define an equation by 'Eq(left_side, right_side)'
sol1 = solve(eq1)    # Solve an equation by 'solve(equation_name)'

sol1                 # Display the solution set
```

[18]:
```
[-2, 2]
```

[19]:
```
fpp = f.diff(x, 2)     # Find the second derivative of 'f'

display(fpp.subs(x, sol1[0]))  # Evaluate the second derivative of 'f' at first␣
 ↪critical point
display(fpp.subs(x, sol1[1]))  # Evaluate the second derivative of 'f' at second␣
 ↪critical point
```

-12

12

This indicates that the first critical point of f (*i.e.* $x = -2$) is a local maximum, and the second critical point of f (*i.e.* $x = 2$) is a local minimum.

[20]:
```
# Define a function to decide about the nature of a given critical point of a␣
↪given function

def critical_point_type(cr_pt, f, var):
    cr_pt = {var: cr_pt}
    fxx = diff(f, var, 2).subs(cr_pt)    # Evaluating second derivative at a␣
↪critical point
    if fxx > 0:                          # Applying the second derivative test
        string = 'local minimum'
    elif fxx < 0:
        string = 'local maximum'
    else:
        string = 'inconclusive!'
    return string
```

[21]:
```
# Displaying the nature of the two critical points

print('Critical point x=%s is a %s.\n' %(sol1[0], critical_point_type(sol1[0],␣
↪f, x)))
print('Critical point x=%s is a %s.' %(sol1[1], critical_point_type(sol1[1], f,␣
↪x)))
```

```
Critical point x=-2 is a local maximum.

Critical point x=2 is a local minimum.
```

1.3.2 Multivariable Functions

Suppose f is a function of n variables $x_1, x_2, \cdots, x_n$. To extremize f, we need to set the gradient of f to 0

$$\vec{\nabla} f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right\rangle = \vec{0}\,. \tag{6}$$

To find out the nature of the extreme points of $f(x_1, \cdots, x_n)$, we have to find the Hessian of f. The Hessian of f is defined by

$$\text{Hess}(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}. \tag{7}$$

The second derivative test generalizes as follows:

- If eigenvalues of Hess(f) are all positive at an extremum of f, then that extremum is a local minimum of f.
- If eigenvalues of Hess(f) are all negative at an extremum of f, then that extremum is a local maximum of f.
- If some eigenvalues of Hess(f) are positive and some are negative (*but no zero eigenvalue among the set of eigenvalues*) at an extreme point of f, then that point is a saddle point of f.
- If Hess(f) possesses a zero eigenvalue at an extremum of f, the test is inconclusive.

Example 6: Suppose $g(x) = 3x^2y^2 + x^2 - 12xy + 2x + 15$. Find local extrema of g and determine their nature.

[22]:
```
x, y = symbols('x y')              # Define 'x' and 'y' as symbols
g = Function('g')(x, y)            # Define function 'g' with variables 'x' and 'y'

g = 3*x**2*y**2 + x**2 - 12*x*y + 2*x + 15   # Specifying the defining expresion␣
 ↪for 'g'

display(g)                         # Display the function
```

$3x^2y^2 + x^2 - 12xy + 2x + 15$

[23]:
```
gx = diff(g, x)   # Differentiate 'g' with respect to 'x'
gy = diff(g, y)   # Differentiate 'g' with respect to 'x'

display(gx, gy)   # Display partial derivatives of 'g'
```

$6xy^2 + 2x - 12y + 2$

$6x^2y - 12x$

[24]:
```
eqs = [Eq(gx, 0), Eq(gy, 0)]    # Define two equations for the two partial␣
 ↪derivatives
sol = solve(eqs)                # Solve the partial derivative equations to find␣
 ↪critical points of 'g'

display(sol)                    # Display the solution set (i.e. critical points)

print('Number of critical points:', len(sol))    # Find number of critical points
```

```
[{x: -1, y: -2}, {x: 0, y: 1/6}]
```

```
Number of critical points: 2
```

[25]:
```
# Define a function to calculate the Hessian matrix

def hessian(f, var):
    return Matrix([[simplify(diff(f, var[i], var[j])) for j in range(len(var))]␣
  ↪for i in range(len(var))])
```

[26]:
```
var = [x, y]            # Set of variables

hess = hessian(g, var)  # Calculate Hessian of 'g'

display(hess)           # Display Hessian matrix
```

$$\begin{bmatrix} 6y^2+2 & 12xy-12 \\ 12xy-12 & 6x^2 \end{bmatrix}$$

[27]:
```
hess0 = hess.subs(sol[0])  # Evalute Hessian at 1st critical point
hess1 = hess.subs(sol[1])  # Evalute Hessian at 2nd critical point

display(hess0, hess1)      # Display Hessian at the two critical points
```

$$\begin{bmatrix} 26 & 12 \\ 12 & 6 \end{bmatrix}$$

$$\begin{bmatrix} \frac{13}{6} & -12 \\ -12 & 0 \end{bmatrix}$$

[28]:
```
l0 = list(hess0.eigenvals().keys())    # Calculate the eigenvalues of Hessian at␣
  ↪1st critical point
l1 = list(hess1.eigenvals().keys())    # Calculate the eigenvalues of Hessian at␣
  ↪2nd critical point

print(l0, l1, '\n')                    # Show the exact eigenvalues

l0 = [l0[i].evalf(3) for i in range(len(l0))]    # Convert eigenvalues to float
l1 = [l1[i].evalf(3) for i in range(len(l1))]    # Convert eigenvalues to float

print('Eigenvalues of Hessian at critical point %s are %s' %(sol[0], l0))    #␣
  ↪Show the results
print('Eigenvalues of Hessian at critical point %s are %s' %(sol[1], l1))
```

```
[16 - 2*sqrt(61), 2*sqrt(61) + 16] [13/12 - sqrt(20905)/12, 13/12 +
sqrt(20905)/12]

Eigenvalues of Hessian at critical point {x: -1, y: -2} are [0.380, 31.6]
Eigenvalues of Hessian at critical point {x: 0, y: 1/6} are [-11.0, 13.1]
```

[29]:
```
# Define a function to decide about the nature of a given critical point of a␣
  ↪given function

def critical_point_type(cr_pt, f, var):
    hess = hessian(f, var).subs(cr_pt)    # Evaluating Hessian at the critical␣
↪point
    l = list(hess.eigenvals().keys())     # Finding eigenvalues of the Hessian
    count = 0                             # Setting a counter
    for k in l:
        if k>0:
            count += 1                    # Counting the number of positive␣
↪eigenvalues
    if 0 in l:                            # Applying the second derivative test␣
↪(multivariable case)
        string = 'The second derivative test is inconclusive'
    else:
        if count == 0:
            string = 'local maximum'
        elif count == len(l):
            string = 'local minimum'
        else:
            string = 'saddle point'
    return string
```

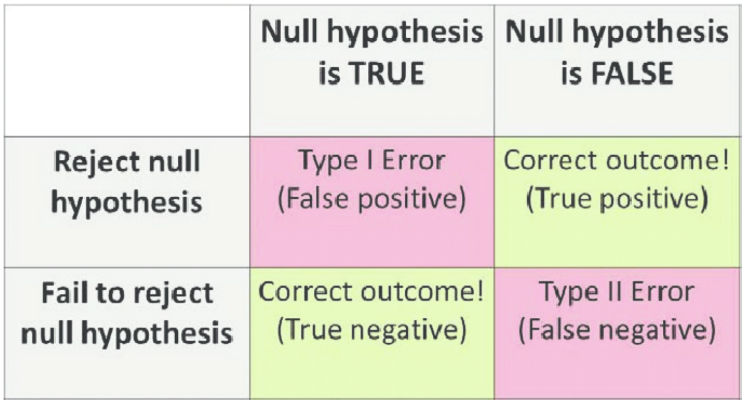
[30]:
```
# Displaying the nature of the two critical points

print('Critical point %s is a %s.\n' %(sol[0], critical_point_type(sol[0], g,␣
↪var)))
print('Critical point %s is a %s.' %(sol[1], critical_point_type(sol[1], g,␣
↪var)))
```

```
Critical point {x: -1, y: -2} is a local minimum.

Critical point {x: 0, y: 1/6} is a saddle point.
```

1.4 Hypothesis Testing in Statistics

Hypothesis testing which is widely used in inferential statistics enables one to decide whether the data at hand sufficiently support a particular hypothesis. Let us briefly review the elements of hypothesis testing. Hypothesis testing is formulated in terms of two hypotheses, namely the **null hypothesis**, denoted by H_0, and the **alternative hypothesis**, denoted by H_1. Hypothesis testing uses data from a *sample* to draw conclusions about a *population parameter* or a population probability distribution. First, a tentative assumption, H_0, is made about the parameter or distribution. An alternative hypothesis, H_1, which is the opposite of what is stated in the null hypothesis, is then defined. The testing procedure involves using sample data to determine whether or not H_0 can be rejected. If H_0 is rejected, the statistical conclusion is that the alternative hypothesis H_1 is accepted. Once H_0 and H_1 are formulated, there four possibilities to take place:

The probability of type-I error (rejecting the null hypothesis H_0 when it is actually true) is denoted by α. In using the hypothesis testing to determine if the null hypothesis should be rejected, the person conducting the hypothesis test specifies the *maximum allowable probability of making a type I*



	Null hypothesis is TRUE	Null hypothesis is FALSE
Reject null hypothesis	Type I Error (False positive)	Correct outcome! (True positive)
Fail to reject null hypothesis	Correct outcome! (True negative)	Type II Error (False negative)

error, called the **level of significance** for the test. Common choices for the level of significance are $\alpha = 0.05$ and $\alpha = 0.01$. A concept known as the p-value provides a convenient basis for drawing conclusions in a hypothesis testing. A p-value is the *probability of obtaining extreme test results*, under the assumption that the null hypothesis is correct.

In order to decide whether H_0 or H_1 should be rejected, we proceed as follows:

- If p-value is less than the level of significance α (*i.e.* $p < \alpha$), we reject H_0 and accept H_1.
- If p-value is greater than the level of significance α (*i.e.* $p > \alpha$), we reject H_1, and continue to accept H_0 as the valid hypothesis.

Example 7: Suppose you are given a coin and you would like to decide whether the given coin is fair coin, using the hypothesis testing. You flip the coin 400 times, and you realize that you get 223 heads.

We first form the null and alternative hypotheses:

$$\begin{aligned} H_0 &: \text{ The coin is fair and the probability of getting a head is } p = \frac{1}{2} \\ H_1 &: \text{ The coin is biased, and the probability of finding head is } p \neq \frac{1}{2} \end{aligned} \tag{8}$$

The observed probability for getting a head in your sample is $\hat{p} = \frac{223}{400} = 0.5575$. Note that the coin follows a binomial distribution. However, since the number of observations in your sample is large, you can approximate the distribution by a normal distribution. The standard deviation of the sample is then calculated by

$$\sigma_p = \sqrt{\frac{p(1-p)}{n}} = \sqrt{\frac{0.5(1-0.5)}{400}} = \frac{1}{40} = 0.025\ . \tag{9}$$

We now calculate the z-score:

$$z = \frac{\hat{p} - p}{\sigma_p} = \frac{0.5575 - 0.5}{0.025} = 2.3 \tag{10}$$

To calculate the p-value, you can simply use the normal cdf (cumulative distribution function). The cdf function exists in the scipy library. Note that we are conducting a two-tailed hypothesis testing.

[31]:
```
# Import 'norm' from scipy

from scipy.stats import norm

pvalue = 2*(1-norm.cdf(2.3))    # p-value = 2(1-cdf(z_score))

print('p-value =', pvalue)
```

```
p-value = 0.021448220043351673
```

Now, let us consider two levels of significance: $\alpha = 0.05$ and $\alpha = 0.01$.

- $\alpha = 0.05$: In this case, p-value is less than the level of significance. Hence, with 95% confidence, we can reject the null hypothesis H_0, and accept the alternative hypothesis H_0. This implies that with 95% confidence, we can claim that the coin is not fair and is biased.
- $\alpha = 0.01$: In this case, p-value is greater than the level of significance. Hence, we cannot reject the null hypothesis H_0. In other words, with 99% confidence, we cannot claim that the coin is biased.

[ ]:
```

```