Data Science for Mechanical Systems Notes

Machine Learning

In [35]:
```
# Week 1
```

Week 1 - hello_world.ipynb

Introduces:
- variables
- operators
- control structures
- list comprehension
- functions
- class
- using pandas

In [36]:
```
from __future__ import annotations

# test imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn
```

In [37]:
```
print(f"The version of numpy is: {np.__version__}")
print(f"The version of pandas is: {pd.__version__}")
print(f"The version of scikit-learn is: {sklearn.__version__}")
```

```
The version of numpy is: 1.24.3
The version of pandas is: 1.5.3
The version of scikit-learn is: 1.3.0
```

In [38]:
```
# Variables
```

In [39]:
```
number_1 = 1
number_2 = 2.0
print(number_1)
print(type(number_1))
print(number_2)
print(type(number_2))
```

```
1
<class 'int'>
2.0
<class 'float'>
```

In [40]:
```
string_1 = "hello"
string_2 = "hello world"

print(string_1)
print(type(string_1))
print(string_2)
print(type(string_2))
```

```
hello
<class 'str'>
hello world
<class 'str'>
```

In [41]:
```
list_1 = [1, 2, 3]

print(list_1)
print(len(list_1))
print(f"The second element in the list is: {list_1[1]}")
print(f"The last element in the list is: {list_1[-1]}")
```

```
[1, 2, 3]
3
The second element in the list is: 2
The last element in the list is: 3
```

In [42]:
```
list_2 = ["hello", "world", "1", 1]

print(list_2)
```

```
['hello', 'world', '1', 1]
```

In [43]:
```
dict_1 = {
    "class_number": "MECE2020",
    "class_capacity": 150,
}
```

In [44]:
```
print(type(dict_1))
print(dict_1["class_capacity"])
```

```
<class 'dict'>
150
```

In [45]:
```
# Operators
```

In [46]:
```
number_1 + number_2
```

Out[46]:
```
3.0
```

In [48]:
```
# this will fail
list_1 / number_1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[48], line 2
      1 # this will fail
----> 2 list_1 / number_1

TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

In [49]:
```
number_1 >= number_2
```

Out[49]:
```
False
```

In [50]:
```
len(list_2[0]) == len(list_2[1])
```

Out[50]:
```
True
```

In [51]:
```
(len(list_2[0]) == len(list_2[1])) and False
```

Out[51]:
```
False
```

In [52]:
```
# Control Structures
```

In [53]:
```
wether_today = "raining"

if wether_today == "raining":
    print("Bring an umbrella!")
elif wether_today == "sunny":
    print("Enjoy the sun!")
else:
    print("What is the wether today?")
```

```
Bring an umbrella!
```

In [54]:
```
for i in range(10):
    print("The number is:", i)
```

```
The number is: 0
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
```

In [55]:
```
i = 0
while i < 10:
    print("The number is:", i)
    i += 1
```

```
The number is: 0
The number is: 1
The number is: 2
The number is: 3
The number is: 4
The number is: 5
The number is: 6
The number is: 7
The number is: 8
The number is: 9
```

In [56]:
```
# List Comprehension
```

In [57]:
```
list_3 = []
for i in range(10):
    list_3.append(i**2)

list_3
```

Out[57]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [58]:
```
list_4 = [i**2 for i in range(10)]
print(list_4)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [59]:
```
list_5 = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
list_6 = [x if x > 0 else 0 for x in list_5]

list_6
```

Out[59]:
```
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

In [60]:
```
[x / number_1 for x in list_1]
```

Out[60]:
```
[1.0, 2.0, 3.0]
```

In [61]:
```
# Function
```

In [62]:
```
def add(number_1: float, number_2: float) -> float:
    """Adds two numbers."""
    return number_1 + number_2

add(1, 2)
```

Out[62]:
```
3
```

In [63]:
```python
def square_root(x: float) -> float:
    """Calcuates the square root of the input using Newton's method.

    Args:
        x (float): The input number, must be greater or equal to zero.

    Returns:
        (float): Sqaure root of the input.

    Raises:
        ValueError: If the input number is negative.
    """
    if x < 0:
        raise ValueError("The input number can not be negative.")

    def get_next_guess(current_guess: float) -> float:
        """Gets next guess using Newton's method."""
        return 0.5 * (current_guess + x / current_guess)

    epislon = 1e-5
    current_guess = x
    next_guess = get_next_guess(current_guess)

    while abs(current_guess - next_guess) > epislon:
        current_guess = next_guess
        next_guess = get_next_guess(current_guess)

    return next_guess
```

In [64]:
```python
square_root(3)
```

Out[64]:
```
1.7320508075688772
```

In [65]:
```python
# Class
```

In [66]:
```
class Person:
    """A simple class."""
    def __init__(self, name: str):
        self.name = name

    def say(self, words: str):
        """Says something."""
        print(f"{self.name} says: {words}")

    def pat(self, person: Person):
        """Pats another person."""
        print(f"{self.name} pats {person.name}'s shoulder.")
```

In [67]:
```
person_1 = Person("John Doe")
person_1.say("Hello!")
```

```
John Doe says: Hello!
```

In [68]:
```
person_2 = Person("Jane Doe")
person_2.say("Hello too!")
```

```
Jane Doe says: Hello too!
```

In [69]:
```
person_1.pat(person_2)
```

```
John Doe pats Jane Doe's shoulder.
```

In [70]:
```
# Using pandas
```

In [71]:
```
data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/iris.csv")
# this is the data set for the first homework
data.head()
```

Out[71]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

In [72]:
```
data.shape
```

Out[72]:
```
(150, 6)
```

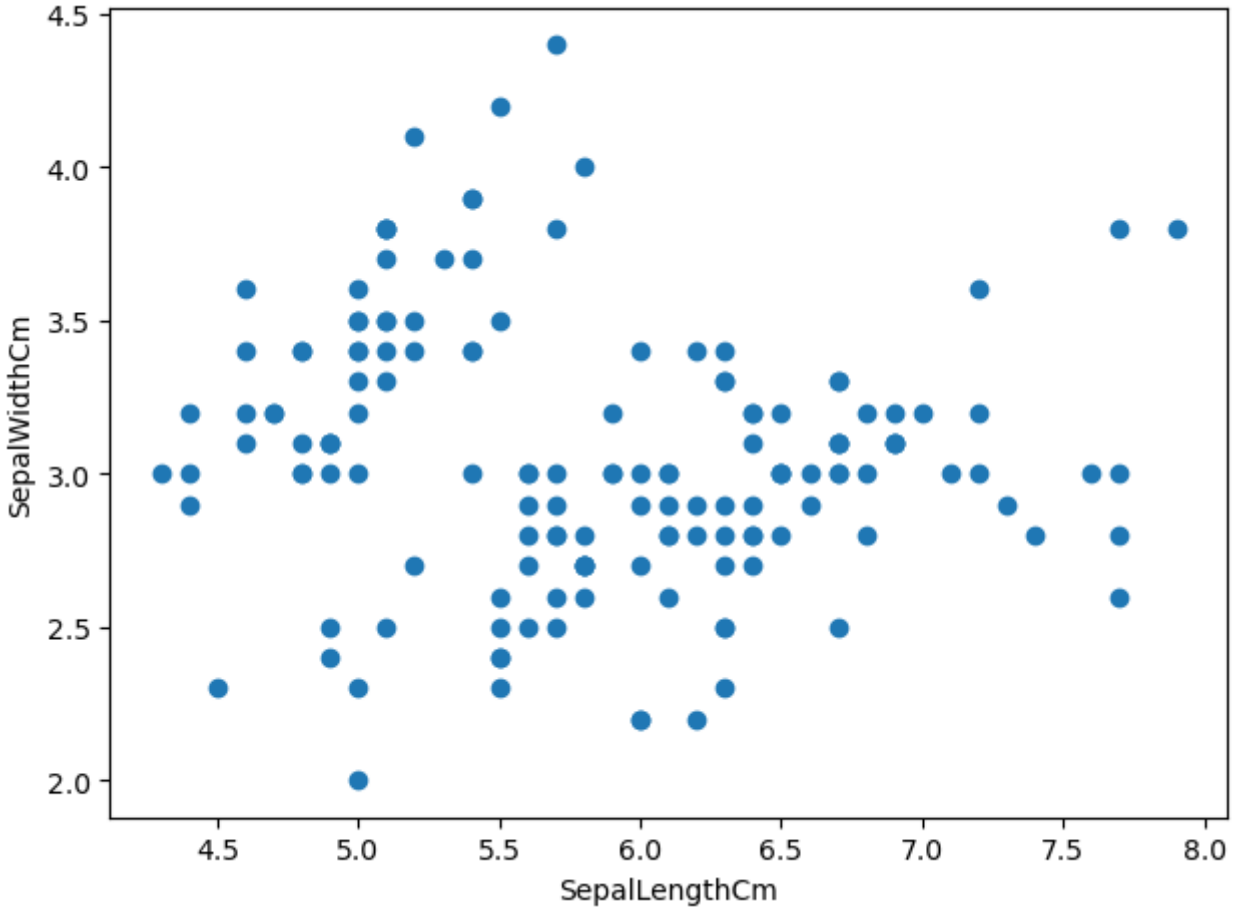
In [73]:
```
# some simple data aggregation
data["Species"].value_counts()
```

Out[73]:
```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: Species, dtype: int64
```

In [74]:
```
data.groupby("Species")["SepalLengthCm"].mean()
```

Out[74]:
```
Species
Iris-setosa        5.006
Iris-versicolor    5.936
Iris-virginica     6.588
Name: SepalLengthCm, dtype: float64
```

In [75]:
```
# Some simple visualization
# If you run this on Google Colab, change `widget` to `inline`

plt.scatter(x=data["SepalLengthCm"], y=data["SepalWidthCm"])
plt.xlabel("SepalLengthCm")
plt.ylabel("SepalWidthCm")
plt.tight_layout()
plt.show()
```



In [76]:
```
# Week 2
```

Week 2 - statistics_primer.ipynb

```
Introduces:
    - Descriptive Statistics
    - Visual Representation
        - Histogram
        - Boxplot
    - Examples of well-known distributions
        - Bernoulli
        - Binomial
        - Geometric
        - Uniform
        - Normal
```

In [77]:
```
import matplotlib.pyplot as plt
import numpy as np

# set random seed
np.random.seed(42)
```

In [78]:
```
# Descriptive Statistics
```

Descriptive statistics provides a summary of the main aspects of the dataset (sample), which we can quantify:

- Measures of Central Tendency: mean, median
- Measures of Dispersion (or Spread): variance, standard deviation
- Measures of Shape: percentile and quantile

In [79]:
```
# generate a random sample
X = np.random.normal(loc=1.0, scale=5.0, size=1000)
```

In [80]:
```
# mean
x_mean = np.mean(X)
x_mean
```

Out[80]:
```
1.0966602791116276
```

In [81]:
```
# median
x_median = np.median(X)
x_median
```

Out[81]:
```
1.126503061174441
```

In [82]:
```
# sample variance
x_var = np.var(X, ddof=1)
x_var
```

Out[82]:
```
23.97159633962756
```

In [83]:
```
# standard deviation
np.sqrt(x_var)
```

Out[83]:
```
4.896079690898378
```

In [84]:
```
np.std(X, ddof=1)
```

Out[84]:
```
4.896079690898378
```

In [85]:
```
# quantile
x_25th_quantile = np.quantile(X, q=0.25)
x_25th_quantile
```

Out[85]:
```
-2.237951527311759
```

In [86]:
```
x_50th_quantile = np.quantile(X, q=0.5)
x_50th_quantile
```

Out[86]:
```
1.126503061174441
```

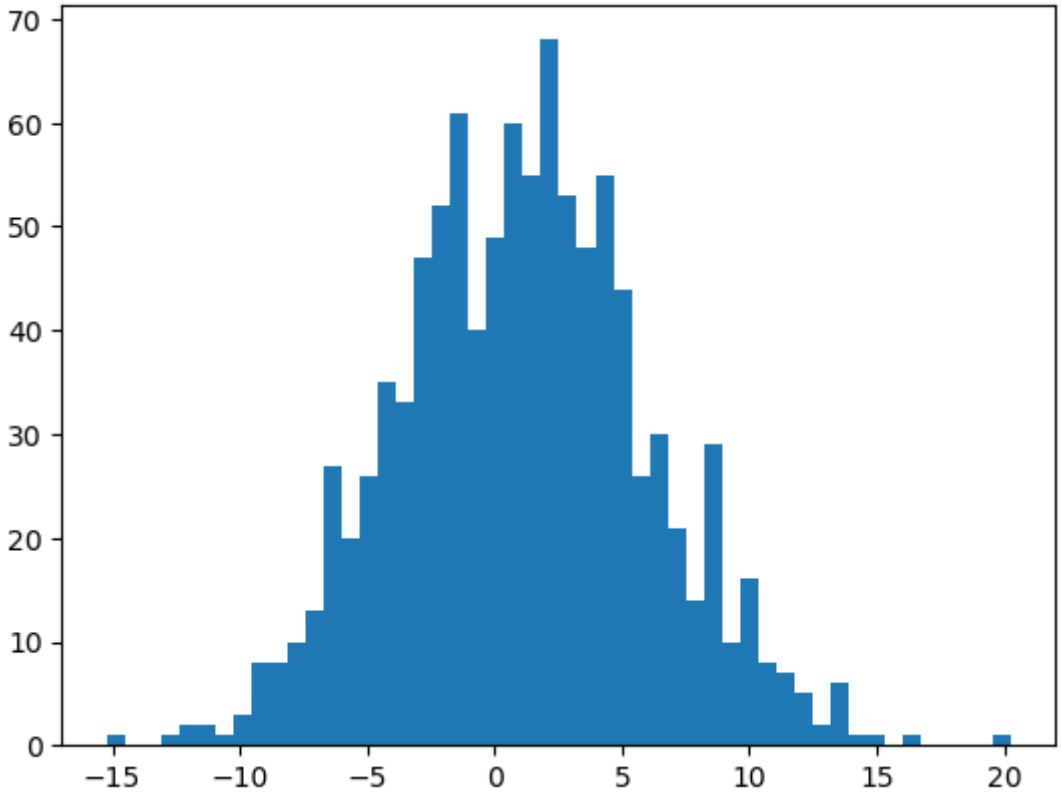
In [87]:
```
# Visual Representation
```

Histogram

A histogram is a graphical representation of the distribution of a dataset. It is an estimate of the probability distribution of a random variable.

To construct a histogram, the first step is to "bin" the range of values — that is, divide the e

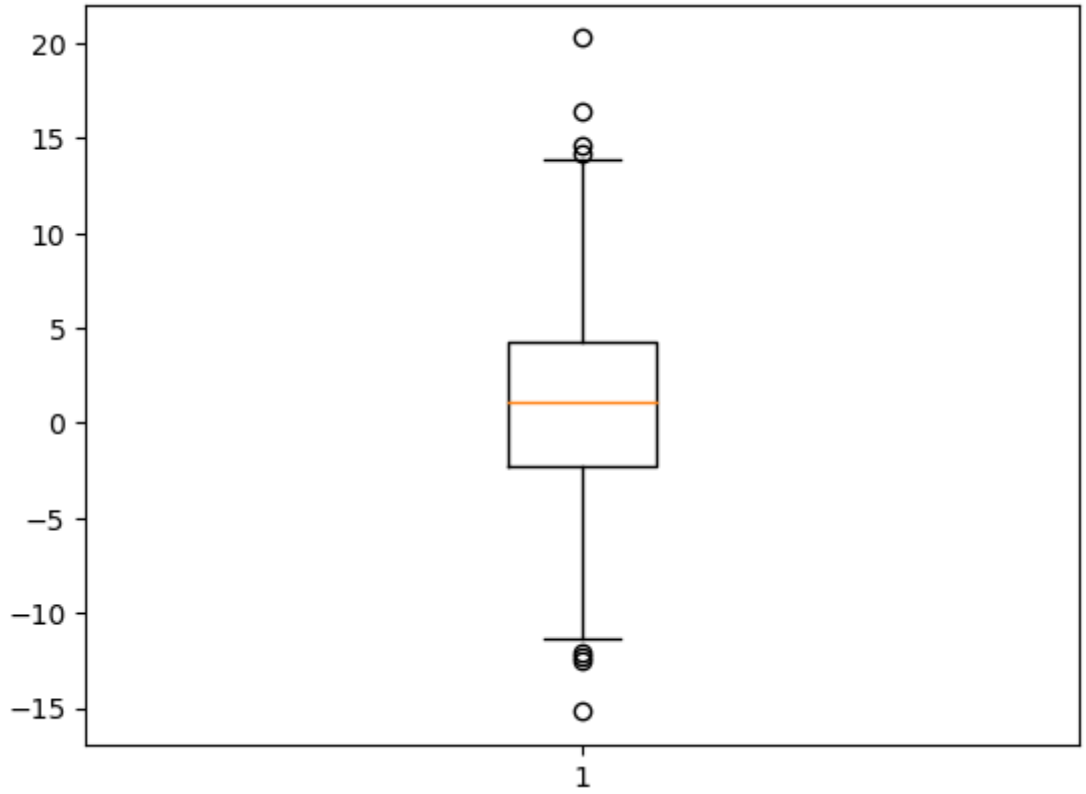
In [88]:
```
# make the histogram
plt.hist(X, bins=50)
plt.show()
```



Boxplot

A boxplot, also known as a whisker plot or box-and-whisker plot, is a standardized way of displa ying the distribution of data based on a summary, based on the minimum, the maximum, the sample median, the first and third quartiles. Other quantities include IQR (Inter Quantile Range), whic h is the difference between the first and the thrid quantiles.

In [89]:
```
# make the box-plot
plt.boxplot(X)
plt.show()
```

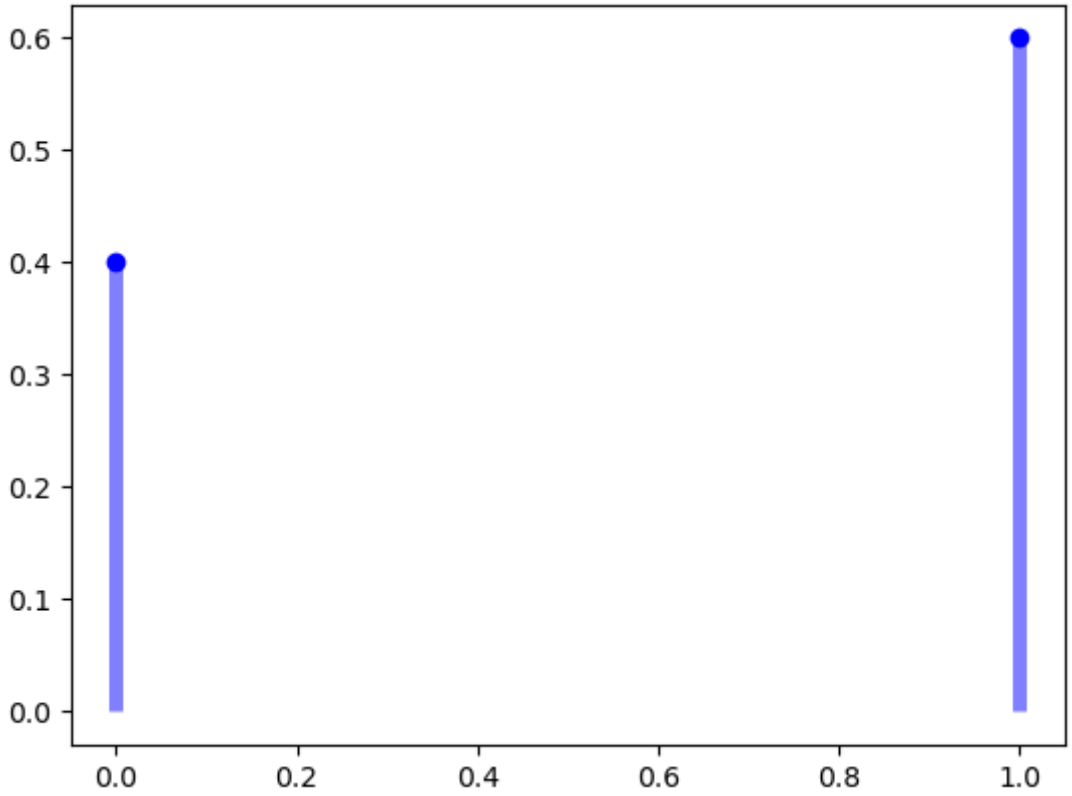


In [90]:
```
# Examples of well-known distributions
```

Bernouli

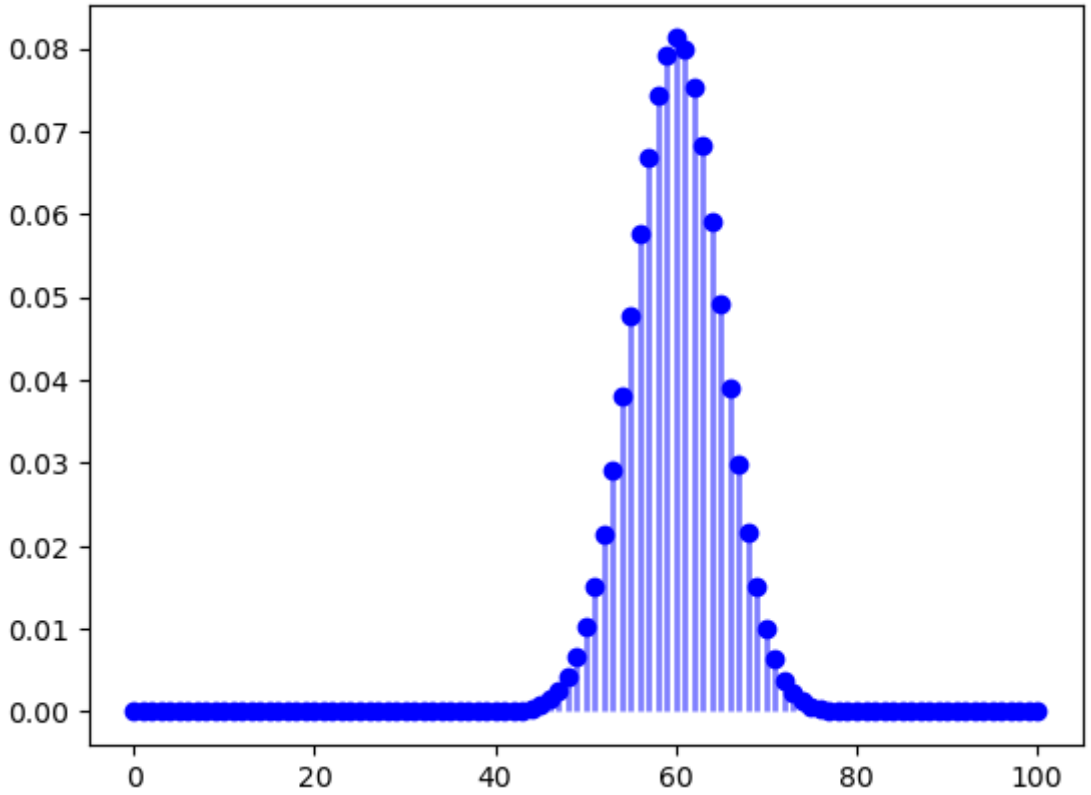
```
The Bernoulli distribution is a discrete probability distribution for a random variable which ca
n take on one of two possible outcomes, often labeled 0 and 1. It's a special case of the binomi
al distribution where a single trial is conducted.
```

In [91]:
```
from scipy.stats import bernoulli

p = 0.6
k = [0, 1]

pmf = bernoulli.pmf(k=k, p=p)

_, ax = plt.subplots(1, 1)
ax.plot(k, pmf, "bo")
_ = ax.vlines(x=k, ymin=0, ymax=pmf, colors="b", lw=5, alpha=0.5)
```



Binomial

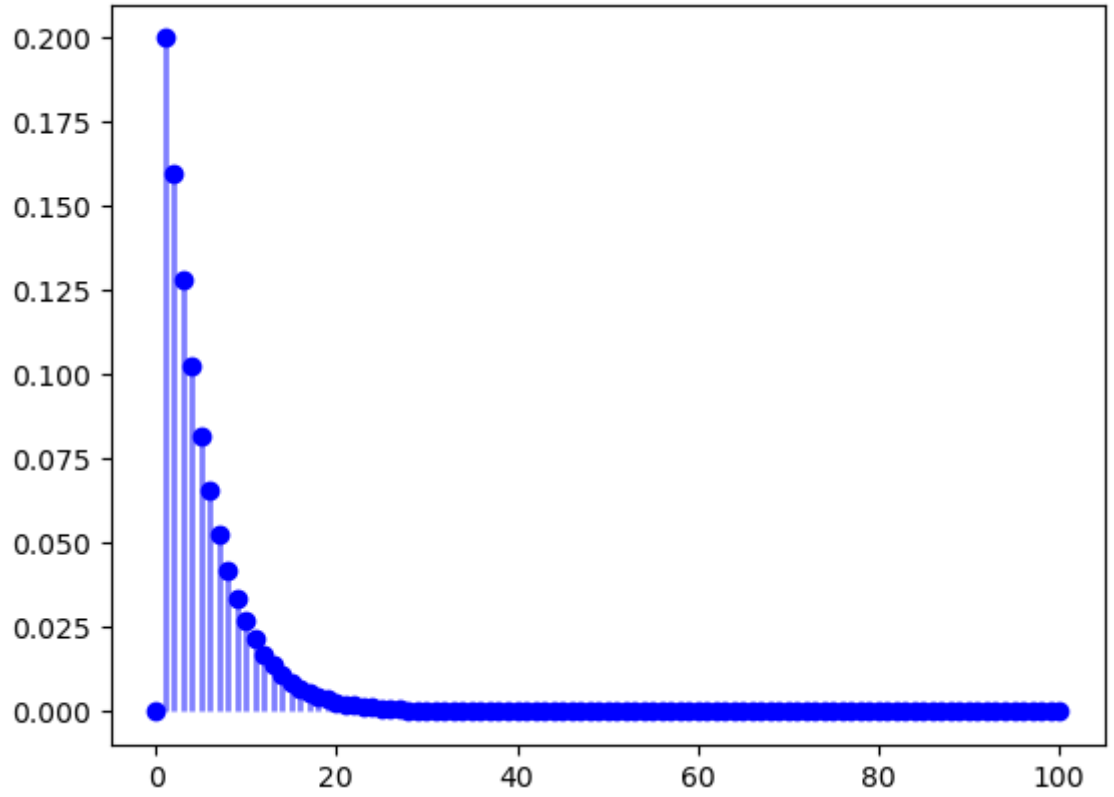
In [92]:
```
from scipy.stats import binom

p = 0.6
n = 100

k = np.arange(start=0, stop=(n + 1), step=1)
pmf = binom.pmf(k=k, n=n, p=p)

_, ax = plt.subplots(1, 1)
ax.plot(k, pmf, "bo")
_ = ax.vlines(x=k, ymin=0, ymax=pmf, colors="b", lw=2, alpha=0.5)
```



Geometric

The geometric distribution is a discrete probability distribution that describes the number of B ernoulli trials required for a success to occur for the first time. It's a model for the "waitin

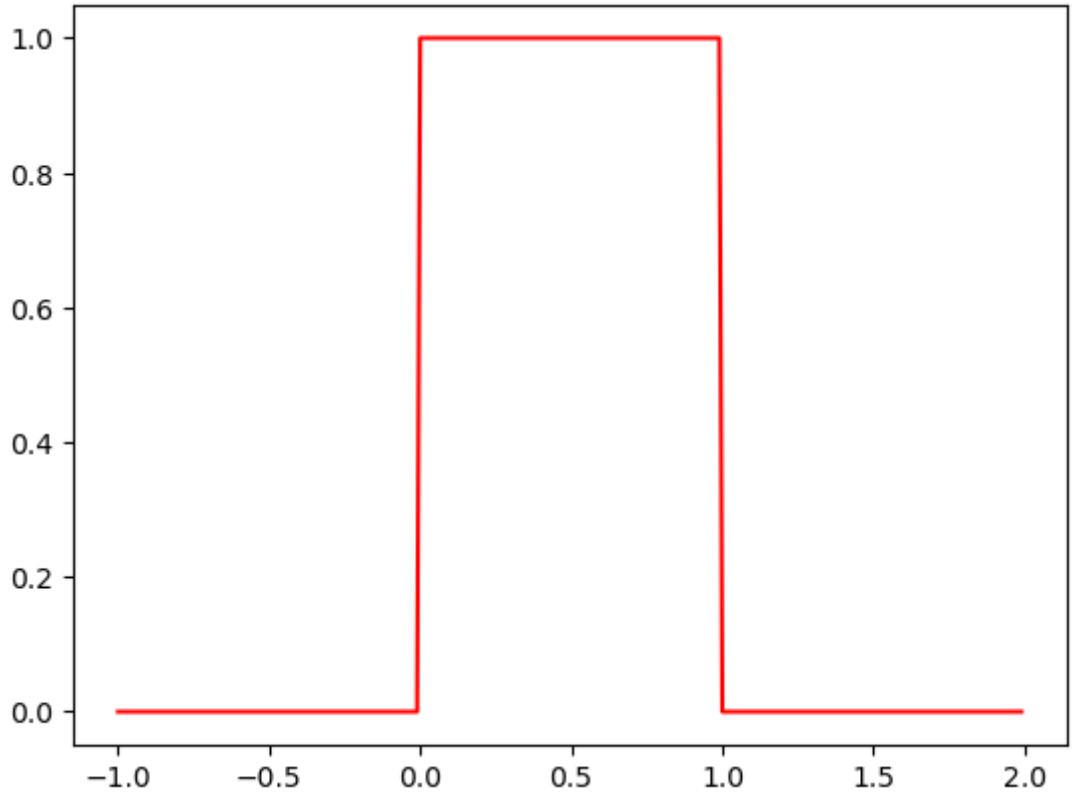
In [93]:
```
from scipy.stats import geom

p = 0.2

k = np.arange(start=0, stop=(n + 1), step=1)
pmf = geom.pmf(k=k, p=p)

_, ax = plt.subplots(1, 1)
ax.plot(k, pmf, "bo")
_ = ax.vlines(x=k, ymin=0, ymax=pmf, colors="b", lw=2, alpha=0.5)
```



Unifrom

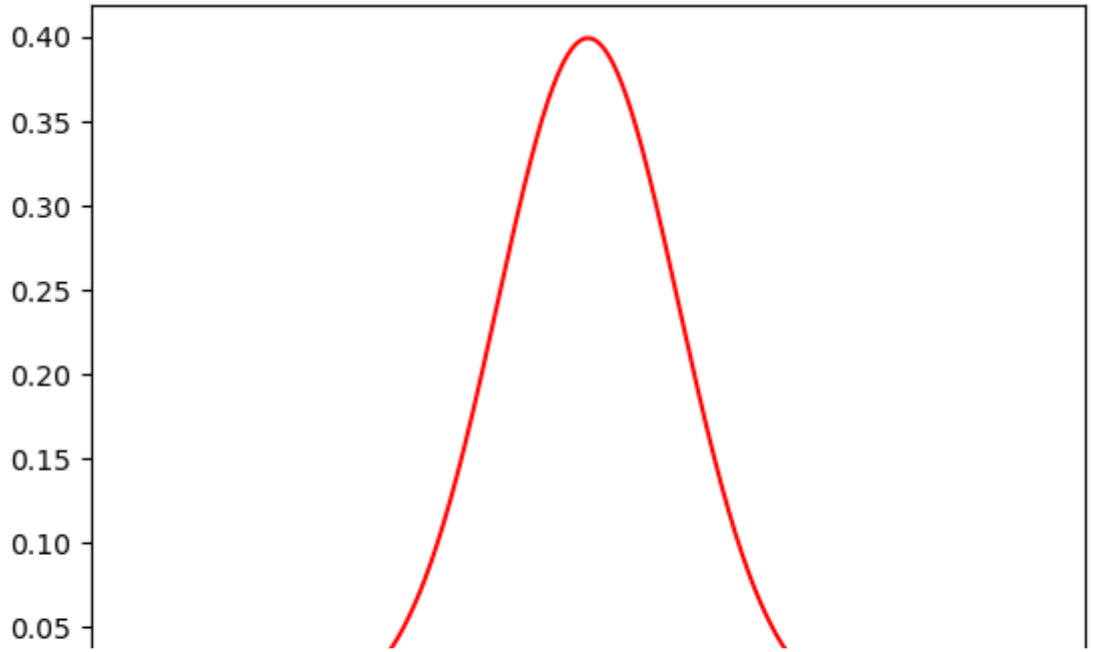
The uniform distribution is the probability distribution in which all outcomes (from within a ra

In [94]:
```
from scipy.stats import uniform

x = np.arange(start=-1, stop=2, step=0.01)
pdf = uniform.pdf(x)

_, ax = plt.subplots(1, 1)
_ = ax.plot(x, pdf, "r-")
```



Normal

In [95]:
```
from scipy.stats import norm

x = np.arange(start=-5, stop=5, step=0.01)
pdf = norm.pdf(x)

_, ax = plt.subplots(1, 1)
_ = ax.plot(x, pdf, "r-")
```



In [96]:
```
# Week 2 cont.
```

Week 2 - vector_and_matrix.ipynb

```
Introduces:
    - Vectors
        - Norm
    - Dot Product
    - Distance
        - Euclidean Distance
    - Cosine Similarity
```

In [97]:
```
from typing import List
from typing import Union

import numpy as np
from IPython.display import display
from matplotlib import pyplot as plt
from PIL import Image
from urllib.request import urlopen
from scipy.spatial import distance
```

In [98]:
```
# Vector
```

Vector is a fundamental concept that represents a quantity with both magnitude and direction. Vectors are widely used in data science for various purposes, including data representation, analysis, and machine learning.

With Python, it is common to represent a vector with the numpy.array object.

In [99]:
```
u = np.array([1, 2, 3])
print(u.shape)
```

```
(3,)
```

The norm of a vector is a mathematical operation that calculates a measure of the length or size of the vector. The norm is a scalar value that provides information about the magnitude or distance of the vector from the origin.

To define a norm, one also needs to provide the order. The choice of which norm to use depends on the specific problem and the characteristics of the data. The Euclidean norm (L2 norm) is often used when considering the overall magnitude or distance between vectors. The Taxicab norm (L1 norm) is used when you want to calculate distances in a grid-like fashion, where you can only move along the coordinate axes. The Infinity norm (L∞ norm) is useful when you want to find the maximum absolute component value of a vector.

In [100]:
```python
# Norms
def norm(
    vector: Union[List[float], np.ndarray],
    order: int = 2
) -> float:
    """Calculate the L-order norm."""
    value = (sum([abs(x)**order for x in vector]))**(1 / order)
    return value
```

You can calculate the norm of a vector by calling numpy's built-in function of linalg.norm().

In [101]:
```python
for order in range(1, 4):
    print(f"The L-{order} norm is:                    ", norm(vector=u, order=order))
    print(f"The L-{order} norm calculated by numpy is:", np.linalg.norm(x=u, ord=order))
    print()
```

```
The L-1 norm is:                    6.0
The L-1 norm calculated by numpy is: 6.0

The L-2 norm is:                    3.7416573867739413
The L-2 norm calculated by numpy is: 3.7416573867739413

The L-3 norm is:                    3.3019272488946263
The L-3 norm calculated by numpy is: 3.3019272488946263
```

In [102]:
```python
# Dot Product
```

The dot product of two vectors is a scalar value that measures the similarity or projection of one vector onto another. It's used in various machine learning algorithms and statistical calculations.

In [103]:
```python
# Dot product
def dot_product(
    u: Union[List[float], np.ndarray],
    v: Union[List[float], np.ndarray]
) -> float:
    """Calcuate the dot- (inner-) product between two vectors."""
    value = sum([x * y for x, y in zip(u, v)])
    return value
```

Similarly, you can use numpy's built-in function, or @ shorthand to perform dot product.

In [104]:
```python
v = np.array([4, 5, 6])

print(dot_product(u, v))
print(np.dot(u, v))
print(u @ v.T)  # short-hand for matrix multiplication
```

```
32
32
32
```

In [105]:
```python
# Distance
```

The distance between two vectors is a measure of how far apart they are in a vector space. There are different ways to calculate the distance between vectors, and the choice of distance metric depends on the context and the specific problem you are trying to solve. Some common distance metrics include:

Euclidean Distance

The Euclidean distance between two vectors is also known as the L2 distance or L2 norm. It calculates the straight-line distance between the two vectors in Euclidean space. It is the L2 norm of the difference (also a vector) between the two vectors.

In [106]:
```
# Distance
print(u - v)
print(norm(u - v))
print(distance.euclidean(u, v))
```

```
[-3 -3 -3]
5.196152422706632
5.196152422706632
```

In [107]:
```
# Cosine Similarity
```

Cosine similarity measures the cosine of the angle between two vectors and is often used to assess the similarity between vectors in high-dimensional spaces. It is not a distance metric in the traditional sense but a similarity measure.

In [108]:
```
def cosine_distance(
    u: Union[List[float], np.ndarray],
    v: Union[List[float], np.ndarray]
) -> float:
    """Calculate the cosine distance between two vectors."""
    value = dot_product(u, v) / (norm(u) * norm(v))
    return 1 - value

print(cosine_distance(u, v))
print(distance.cosine(u, v))
```

```
0.025368153802923787
0.0253681538029239
```

In [109]:
```
# Matrix
```

In the context of data science and mathematics, a matrix is a fundamental data structure that consists of a two-dimensional arrangement of numbers, symbols, or data elements organized in rows and columns.

In [110]:
```
A = np.array([
  [1, 2],
  [3, 4],
])

print(A.shape)
```

```
(2, 2)
```

In [111]:
```
B = np.array([
  [5, 6],
  [7, 8],
])

print(B.shape)
```

```
(2, 2)
```

In [112]:
```
A + B
```

Out[112]:
```
array([[ 6,  8],
       [10, 12]])
```

In [113]:
```
# broadcasting
A + 3
```

Out[113]:
```
array([[4, 5],
       [6, 7]])
```

In [114]:
```
A * 3
```

Out[114]:
```
array([[ 3,  6],
       [ 9, 12]])
```

In [115]:
```
# Matrix Multiplication
```

Matrix multiplication is a fundamental mathematical operation that combines two matrices to produce a new matrix.

In [116]:
```
# multiplication
A = np.array([
    [1, 2],
    [3, 4],
    [5, 6],
    [7, 8],
])
B = np.array([
    [1, 2, 3],
    [4, 5, 6],
])

print(A.shape, B.shape)
```

```
(4, 2) (2, 3)
```

In [117]:
```
np.matmul(A, B)
```

Out[117]:
```
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51],
       [39, 54, 69]])
```

In [118]:
```
A @ B
```

Out[118]:
```
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51],
       [39, 54, 69]])
```

In [119]:
```
# Basic Matrix Operations
```

In [120]:
```
# transpose
print(A)
print(A.T)
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
[[1 3 5 7]
 [2 4 6 8]]
```

In [121]:
```
# trace
print(np.trace(A @ A.T))
print(np.trace(A.T @ A))
```

```
204
204
```

In [122]:
```
# determinant
print(np.linalg.det(A @ A.T))
print(np.linalg.det(A.T @ A))
```

```
-3.34779992132379e-29
79.99999999999969
```

In [123]:
```
# identity matrix
I = np.eye(3)
print(I)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [124]:
```
# Matrix Inverse
```

The inverse of a matrix is a special matrix that, when multiplied by the original matrix, results in the identity matrix.

Not all matrices have inverses, and those that do are called "invertible" or "non-singular" matrices.

In [125]:
```
# inverse
X = A.T @ A
print(X)
np.linalg.inv(X)
```

```
[[ 84 100]
 [100 120]]
```

Out[125]:
```
array([[ 1.5 , -1.25],
       [-1.25,  1.05]])
```

In [126]:
```
Y = A @ A.T

# this will fail
np.linalg.inv(Y)
```

Out[126]:
```
array([[ 1.34189044e+14,  3.18362545e+14, -1.03929222e+15,
         5.86740633e+14],
       [ 2.86449391e+13, -8.22436575e+14,  1.55893833e+15,
        -7.65146697e+14],
       [-4.59857010e+14,  6.89785515e+14,  0.00000000e+00,
        -2.29928505e+14],
       [ 2.97023027e+14, -1.85711485e+14, -5.19646111e+14,
         4.08334569e+14]])
```

In [127]:
```
# pseudo-inverse
np.linalg.pinv(Y)
```

Out[127]:
```
array([[ 1.7225,  0.8825,  0.0425, -0.7975],
       [ 0.8825,  0.4525,  0.0225, -0.4075],
       [ 0.0425,  0.0225,  0.0025, -0.0175],
       [-0.7975, -0.4075, -0.0175,  0.3725]])
```

In [128]:
```
np.linalg.pinv(Y) @ Y
```

Out[128]:
```
array([[ 0.7,  0.4,  0.1, -0.2],
       [ 0.4,  0.3,  0.2,  0.1],
       [ 0.1,  0.2,  0.3,  0.4],
       [-0.2,  0.1,  0.4,  0.7]])
```

In [129]:
```
# Rank
```

The rank of a matrix is a fundamental concept in linear algebra and is defined as the maximum number of linearly independent rows or columns in the matrix. In other words, it measures the dimension of the vector space spanned by the rows or columns of the matrix.

In [130]:
```
# rank
print(X.shape)
print(np.linalg.matrix_rank(X))
```

```
(2, 2)
2
```

In [131]:
```
print(Y.shape)
print(np.linalg.matrix_rank(Y))
```

```
(4, 4)
2
```

In [132]:
```
# Eigenvalues and Eigenvectors
```

In linear algebra, the eigenvalues and eigenvectors of a square matrix are important concepts used to understand the behavior and transformations associated with that matrix. Let's explore what eigenvalues and eigenvectors are in more detail:

Eigenvalues:

Eigenvalues are scalars (numbers) associated with a square matrix. Each matrix can have a set of eigenvalues. The eigenvalues of a matrix A are values λ that satisfy the following equation:

$Av = \lambda v$

Here, A is the square matrix, λ is the eigenvalue, and v is the corresponding eigenvector.

In other words, when you multiply the matrix A by its eigenvector v, you get a new vector that is just a scaled version of the original eigenvector v, where the scaling factor is the eigenvalue λ.

Eigenvalues provide information about how the matrix scales or stretches space along specific directions.

Eigenvectors:

Eigenvectors are non-zero vectors associated with eigenvalues. Each eigenvalue corresponds to one or more eigenvectors.

Eigenvectors are the vectors that remain in the same direction after being transformed by the matrix A. They are only scaled (stretched or compressed) by the eigenvalue.

Eigenvectors are often normalized (scaled to have a length of 1) for convenience

In [133]:
```
# w are eigenvalues
# v are eigenvectors

w, v = np.linalg.eig(X)
```

In [134]:
```
print(f"The eigenvalues are: {w}")
print("The eigenvectors are:")
print(v)
```

```
The eigenvalues are: [  0.39291363 203.60708637]
The eigenvectors are:
[[-0.7671874  -0.64142303]
 [ 0.64142303 -0.7671874 ]]
```

In [135]:
```
# reproduce the X * v
print(X @ v[:, 0])
```

```
[-0.30143839  0.25202385]
```

In [136]:
```
# reproduce the \lambda * v
w[0] * v[:, 0]
```

Out[136]:
```
array([-0.30143839,  0.25202385])
```

In [137]:
```
# SVD (Singular Value Decomposition)
```

SVD stands for Singular Value Decomposition, and it is a fundamental matrix factorization technique used for various purposes, including dimensionality reduction, data compression, and feature engineering.

In [138]:
```
# SVD
A = np.array([
    [1, 2],
    [3, 4],
    [5, 6],
])

U, Sigma, V = np.linalg.svd(A)
V = V.T
print("U:\n", U)
print("Sigma:\n", Sigma)
print("V:\n", V)
```

```
U:
 [[-0.2298477   0.88346102  0.40824829]
 [-0.52474482  0.24078249 -0.81649658]
 [-0.81964194 -0.40189603  0.40824829]]
Sigma:
 [9.52551809 0.51430058]
V:
 [[-0.61962948 -0.78489445]
 [-0.78489445  0.61962948]]
```

In [139]:
```
# recreate the original matrix
# we need to turn the ``Sigma`` to a ndarray of the proper shape
sigma_matrix = np.insert(np.diag(Sigma), len(Sigma), [0, 0], 0)
U @ sigma_matrix @ V.T
```

Out[139]:
```
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
```

In [140]:
```
# Create the partial matrices
US = U @ sigma_matrix
US
```

Out[140]:
```
array([[-2.18941839,  0.45436451],
       [-4.99846626,  0.12383458],
       [-7.80751414, -0.20669536]])
```

In [141]:
```
# partial matrix 1
np.matrix(US[:, 0]).T @ np.matrix(V[:, 0])
```

Out[141]:
```
matrix([[1.35662819, 1.71846235],
        [3.09719707, 3.92326845],
        [4.83776596, 6.12807454]])
```

In [142]:
```
# partial matrix 2
np.matrix(US[:, 1]).T @ np.matrix(V[:, 1])
```

Out[142]:
```
matrix([[-0.35662819,  0.28153765],
        [-0.09719707,  0.07673155],
        [ 0.16223404, -0.12807454]])
```

SVD can be used for data compression, particularly in image and signal processing.

By representing an image or signal with a reduced number of singular values and corresponding components, you can achieve data compression while retaining the essential features of the data.

In [143]:
```
# Image compression
FIGURE_SIZE = (10, 10)
plt.gray()

original = Image.open(urlopen("https://github.com/changyaochen/MECE4520/raw/master/data/leena.png"))
plt.figure(figsize=FIGURE_SIZE)
plt.imshow(original, interpolation="none")
```

Out[143]:
```
<matplotlib.image.AxesImage at 0x16ba4f350>

<Figure size 640x480 with 0 Axes>
```


0
100
200
300
400

In [144]:
```
original_data = np.array(original)
print(original_data.shape)
original_data
```

```
(512, 512)
```

Out[144]:
```
array([[162, 162, 162, ..., 170, 155, 128],
       [162, 162, 162, ..., 170, 155, 128],
       [162, 162, 162, ..., 170, 155, 128],
       ...,
       [ 43,  43,  50, ..., 104, 100,  98],
       [ 44,  44,  55, ..., 104, 105, 108],
       [ 44,  44,  55, ..., 104, 105, 108]], dtype=uint8)
```

In [145]:
```
U, Sigma, V = np.linalg.svd(original_data)
V = V.T
```

In [146]:
```
k = 20  # Number of principle components to keep

U_reduced = U[:, 0:k]
V_reduced = V[:, 0:k]
Sigma_reduced = np.diag(Sigma[:k])
```

In [147]:
```
compressed = U_reduced @ Sigma_reduced @ V_reduced.T

plt.figure(figsize=FIGURE_SIZE)
plt.imshow(compressed, interpolation="none")
```

Out[147]:
```
<matplotlib.image.AxesImage at 0x16bcc7d50>
```


0
100
200
300
400

In [148]:
```
# Week 3
```

Week 3 - statistics_primer_2.ipynb

```
    Introduces:
        - Central Limit Theorem
        - Point Estimation and Confidence Interval
```

In [149]:
```
import matplotlib.pyplot as plt
import numpy as np

from scipy.stats import binom
from scipy.stats import norm
from scipy.stats import ttest_ind
from tqdm import tqdm

# set random seed
np.random.seed(42)
```

In [150]:
```
# Central Limit Theorem
```

Central Limit Theorem

The Central Limit Theorem (CLT) is a fundamental principle in probability and statistics. It states that, given certain conditions, the sum (or average) of a large number of independent, identically distributed (i.i.d.) random variables approaches a normal (or Gaussian) distribution, no matter the shape of the original distribution. This result is the reason why the normal distribution appears in many contexts and is fundamental in inferential statistics.

The "certain conditions" are referred to:

- Independence: The random variables must be independent. This means that the outcome of one observation does not influence the outcome of another observation. In many practical scenarios, this assumption is satisfied by random sampling.

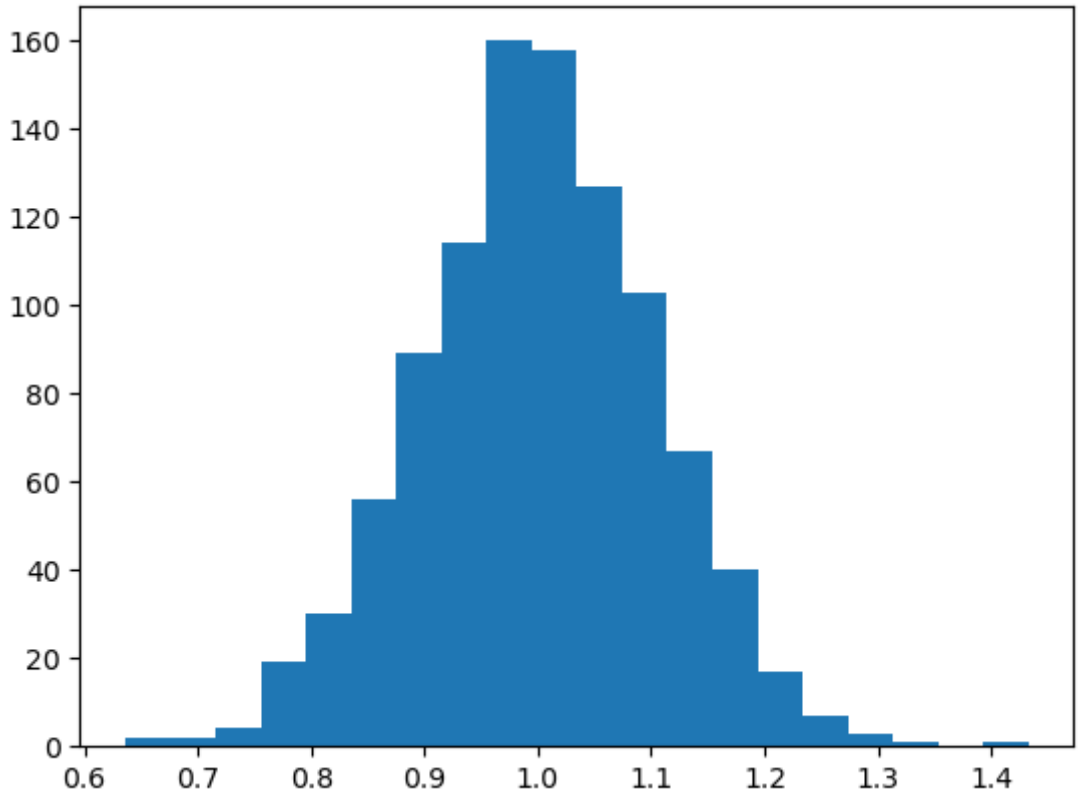
- Identically Distributed: The random variables should be identically distributed, often abbreviated as i.i.d. This means that each random variable has the same probability distribution as the others and the same parameters. In simpler terms, all variables come from the same "population".
- Finite Mean and Variance: Each of the random variables must have a finite mean

(μ) and variance (σ). This means that extremely long-tailed distributions or distributions with no defined mean or variance may not conform to the CLT.

In [151]:
```
n_trials = 1000
n_samples = 100
```

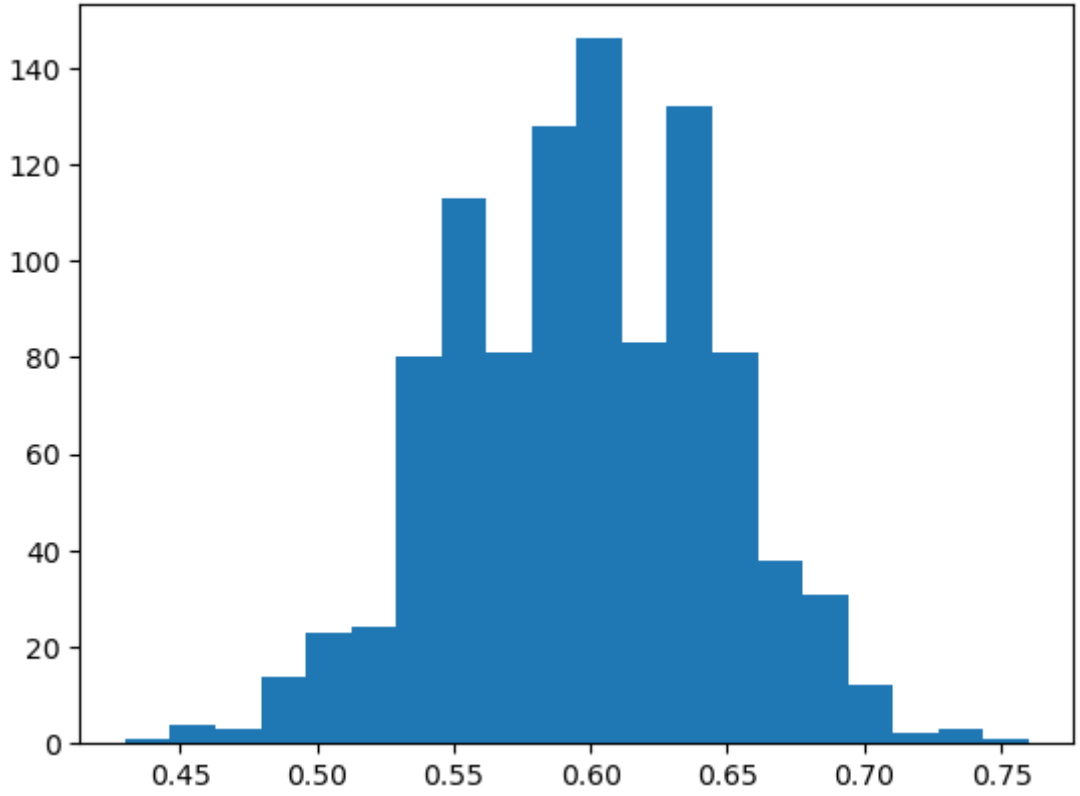
In [152]:
```
# sample means of a normal distribution
sample_means = []
for _ in range(n_trials):
    X = np.random.normal(loc=1.0, scale=1.0, size=n_samples)
    sample_means.append(np.mean(X))

print(f"The mean of sample means is: {np.mean(sample_means):5.3f}.")
print(f"The standard deviation of sample means is: {np.std(sample_means):5.3f}.")
plt.hist(sample_means, bins=20)
plt.show()
```

```
The mean of sample means is: 1.001.
The standard deviation of sample means is: 0.104.
```



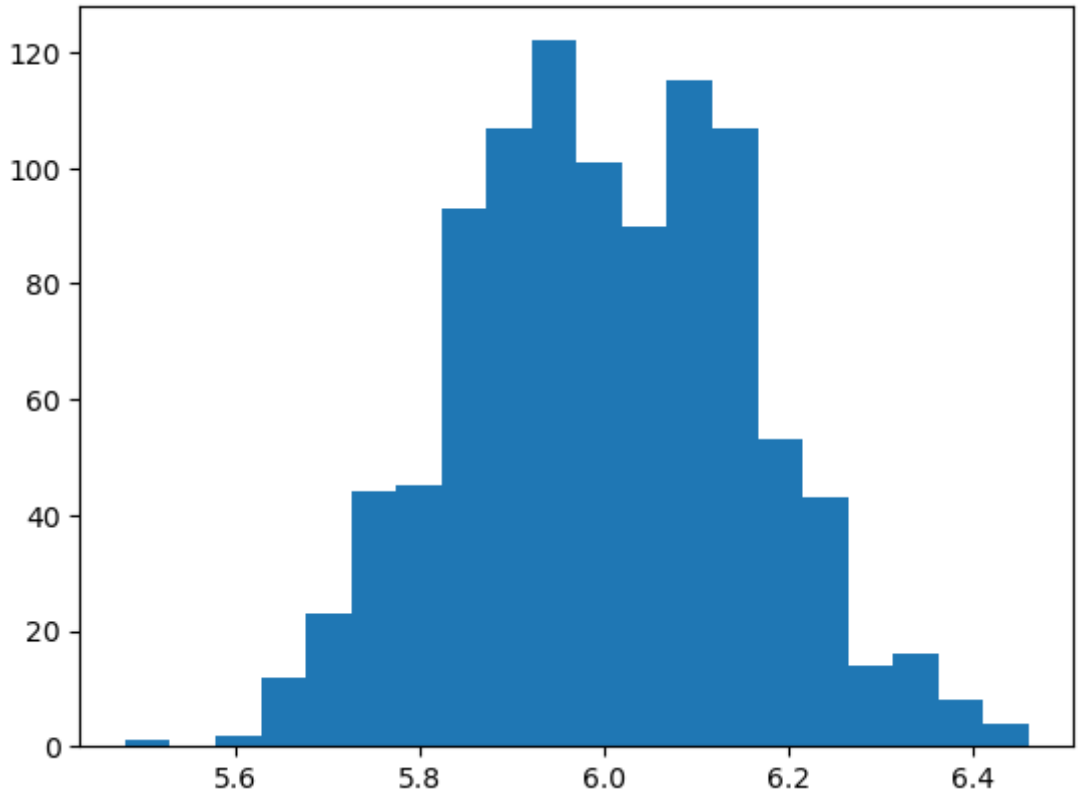
In [153]:
```
# sample means of bernoulli distribution
sample_means = []
p = 0.6
for _ in range(n_trials):
    X_uniform = np.random.random(size=n_samples)
    X = [1 if x < p else 0 for x in X_uniform]
    sample_means.append(np.mean(X))

print(f"The mean of sample means is: {np.mean(sample_means):5.3f}.")
print(f"The standard deviation of sample means is: {np.std(sample_means):5.3f}.")
plt.hist(sample_means, bins=20)
plt.show()
```

```
The mean of sample means is: 0.598.
The standard deviation of sample means is: 0.050.
```



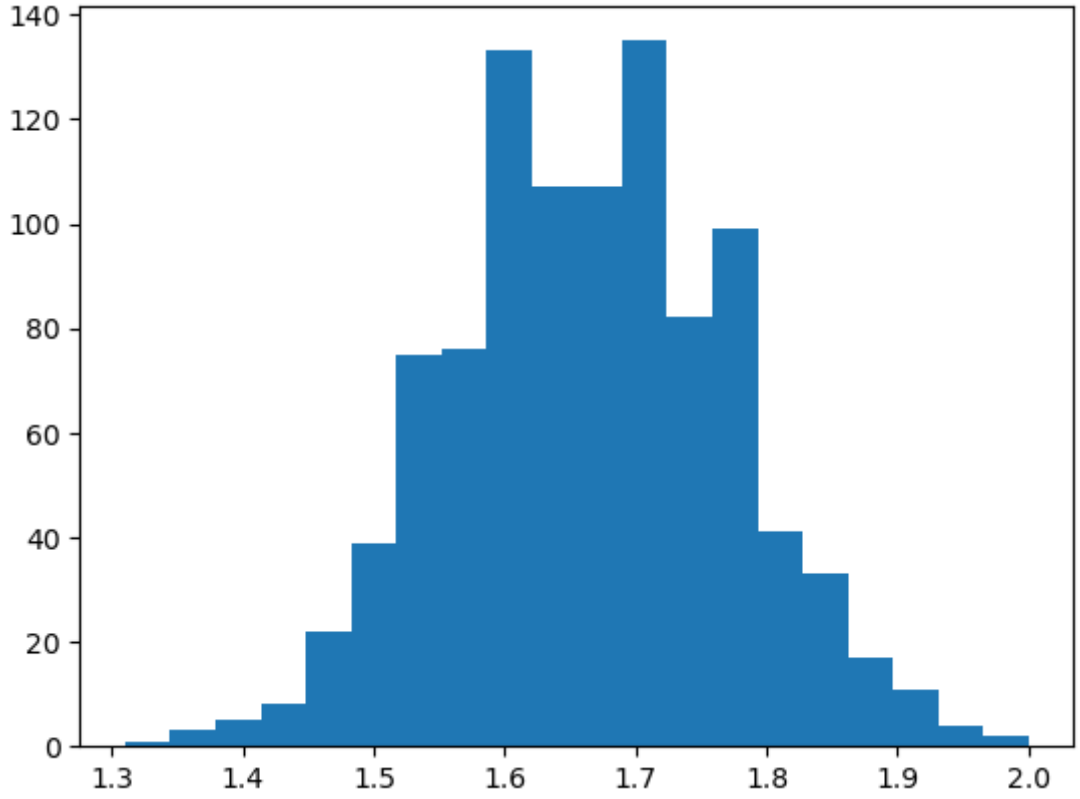
In [154]:
```
# sample means binomial distribtuion
sample_means = []
for _ in range(n_trials):
    X = np.random.binomial(n=10, p=0.6, size=n_samples)
    sample_means.append(np.mean(X))

print(f"The mean of sample means is: {np.mean(sample_means):5.3f}.")
print(f"The standard deviation of sample means is: {np.std(sample_means):5.3f}.")
plt.hist(sample_means, bins=20)
plt.show()
```

```
The mean of sample means is: 6.000.
The standard deviation of sample means is: 0.157.
```



In [155]:
```
# sample means geometric distribution
sample_means = []
for _ in range(n_trials):
    X = np.random.geometric(p=0.6, size=n_samples)
    sample_means.append(np.mean(X))

print(f"The mean of sample means is: {np.mean(sample_means):5.3f}.")
print(f"The standard deviation of sample means is: {np.std(sample_means):5.3f}.")
plt.hist(sample_means, bins=20)
plt.show()
```

```
The mean of sample means is: 1.665.
The standard deviation of sample means is: 0.107.
```



In [156]:
```
# Point Estimation and Confidence Interval
```

Point estimation refers to the process of using sample data to compute a single value, known as a statistic, which serves as the "best guess" or "best estimate" of an unknown population parameter (like the population mean or population proportion). For example, the sample mean is such a statistic, and the it is used as the point estimation (point estimate) of the population mean.

While point estimation gives a specific value as an estimate, it doesn't convey anything about the degree of certainty or possible margin of error associated with the estimate. This is where confidence intervals come into play.

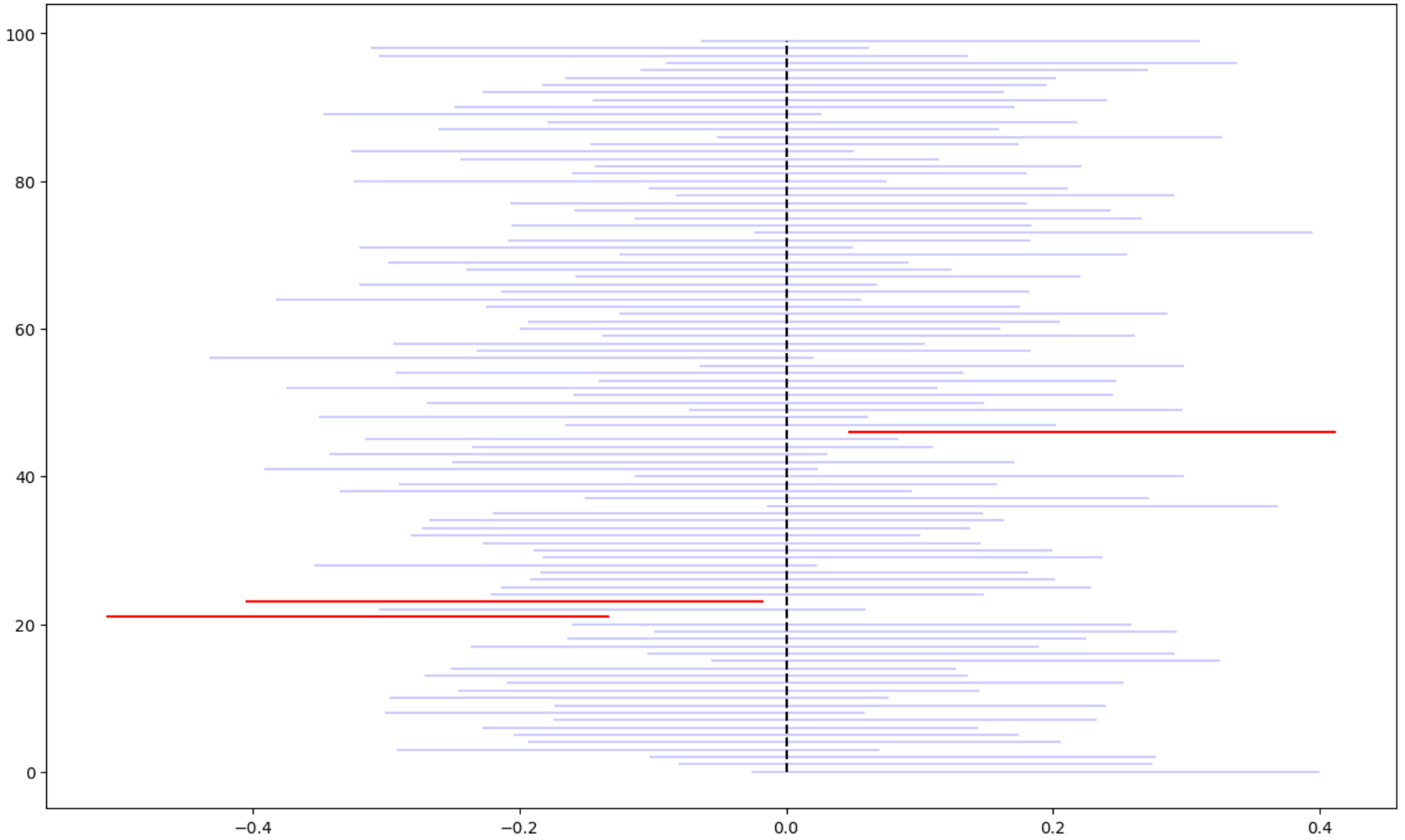
A confidence interval provides a range of values, derived from the point estimate and the associated variability of the estimate, within which the true population parameter is expected to fall with a certain probability (confidence level).

Let's run some simulation to demonstrate the concept.

Below, we already know the population parameter: the population mean. For the sake of demonstration, we will repeatly randomly sample from the population, for n_trials times. For each sampling, we draw 100 samples. Therefore, we will have 100 sample means.

For each of the sample, we can construct the point estimation, and the confidence interval (CI). We will see that, there are chances, that the CI does not include the population mean.

In [157]:
```
mu = 0  # population mean
sigma = 1

n_trials = 100
n_samples = 100
results = []
for i, _ in enumerate(range(n_trials)):
    X = np.random.normal(loc=mu, scale=sigma**0.5, size=n_samples)  # Sampling
    lower_bound = np.mean(X) - 1.96 * np.std(X, ddof=1) / np.sqrt(n_samples)
    upper_bound = np.mean(X) + 1.96 * np.std(X, ddof=1) / np.sqrt(n_samples)

    results.append((i, lower_bound, upper_bound))

plt.figure(figsize=(15, 9))
for i, lower, upper in results:
    if lower > mu or upper < mu:
        plt.plot([lower, upper], [i] * 2, color="red", alpha=1)
    else:
        plt.plot([lower, upper], [i] * 2, color="blue", alpha=0.2)
plt.plot([mu] * len(results), list(range(len(results))), "k--")
plt.show()
```



In [158]:
```
# Week 3 cont.
```

Week 3 - statistics_primer_3.ipynb

```
Introduces:
    - Hypothesis Test
    - 2-Sample Test
    - Multiple Hypothesis Testing
```

In [159]:
```
import matplotlib.pyplot as plt
import numpy as np

from scipy.stats import binom
from scipy.stats import norm
from scipy.stats import ttest_ind
from tqdm import tqdm

# set random seed
np.random.seed(42)
```

In [160]:
```
# Hypothesis Test
```

A hypothesis test (or test of significance) is a statistical method used to make decisions based on data. It involves making an initial assumption (the null hypothesis, H_o) and then assessing the evidence against that assumption to determine if a different hypothesis (the alternative hypothesis, H_a) might be more accurate.

Below, we use the coin toss example discussed in class, and generate the null distribution (the distribution of the test statistic if the null hypothesis is true).

In [161]:
```
# fair coin, binomial test
n = 10
p = 0.5
for k in range(11):
    proba = binom.pmf(k=k, n=n, p=p)
    print(f"The probability of getting {k} head(s) is: {proba:5.3f}")
```

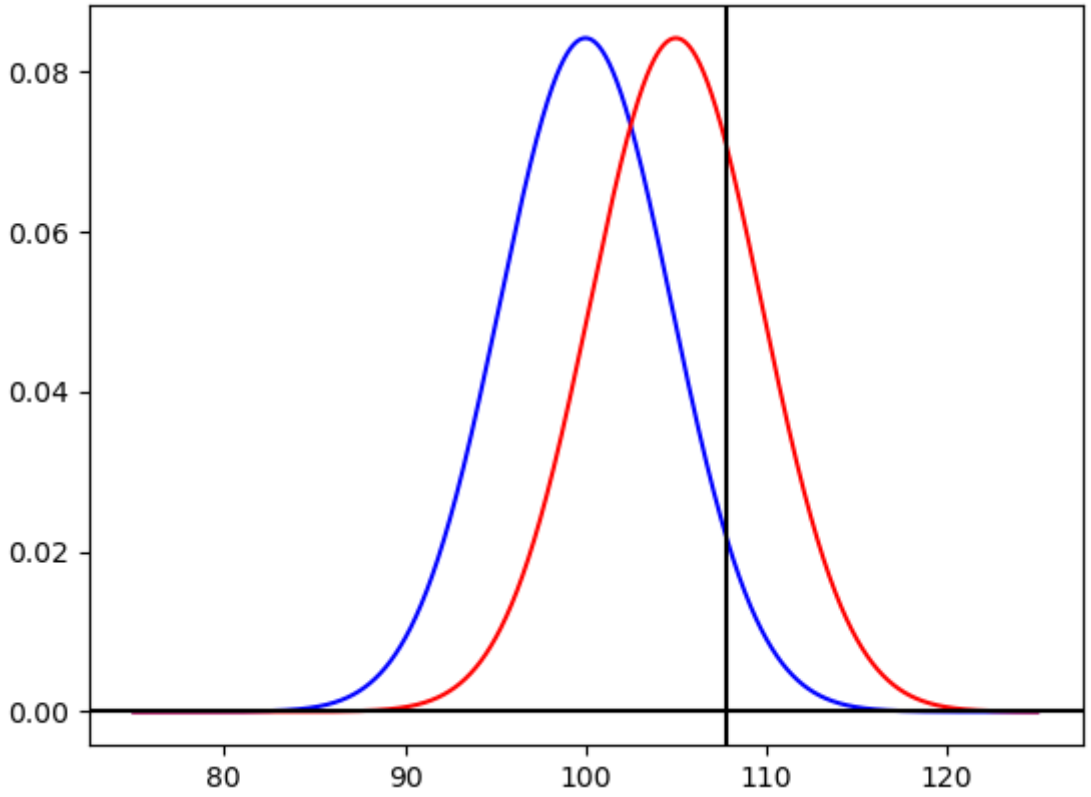
```
The probability of getting 0 head(s) is: 0.001
The probability of getting 1 head(s) is: 0.010
The probability of getting 2 head(s) is: 0.044
The probability of getting 3 head(s) is: 0.117
The probability of getting 4 head(s) is: 0.205
The probability of getting 5 head(s) is: 0.246
The probability of getting 6 head(s) is: 0.205
The probability of getting 7 head(s) is: 0.117
The probability of getting 8 head(s) is: 0.044
The probability of getting 9 head(s) is: 0.010
The probability of getting 10 head(s) is: 0.001
```

Below, we use the IQ test example discussed in the class.

Specifically, we will plot the null distribution (the IQ mean is 100), as well as the alternative distribution (the IQ mean is 105).

In [162]:
```
# IQ example
n = 10
mu_0 = 100
mu_a = 105
sigma = 15
alpha = 0.05  # Type I error rate


x_range = np.arange(75, 125, 0.001)
plt.plot(
    x_range,
    norm.pdf(x_range, loc=mu_0, scale=np.sqrt(sigma**2 / n)),
    color="blue",
    label="H_0"
)

plt.plot(
    x_range,
    norm.pdf(x_range, loc=mu_a, scale=np.sqrt(sigma**2 / n)),
    color="red",
    label="H_a"
)

# find the rejection region
x_critical = norm.ppf(q=(1 - alpha), loc=mu_0, scale=np.sqrt(sigma**2 / n))
power = norm.sf(x=x_critical, loc=mu_a, scale=np.sqrt(sigma**2 / n))

print(f"The critial value is {x_critical:5.3f}.")
print(f"The power of the test is {power:5.3f}.")
plt.axvline(x=x_critical, color="black")

plt.axhline(y=0, color="black")
plt.show()
```

```
The critial value is 107.802.
The power of the test is 0.277.
```



In [163]:
```
# Get the p-value
test_statistic = 110
norm.sf(x=test_statistic, loc=100, scale=np.sqrt(15**2 / n))
```

Out[163]:
```
0.017507490509831244
```

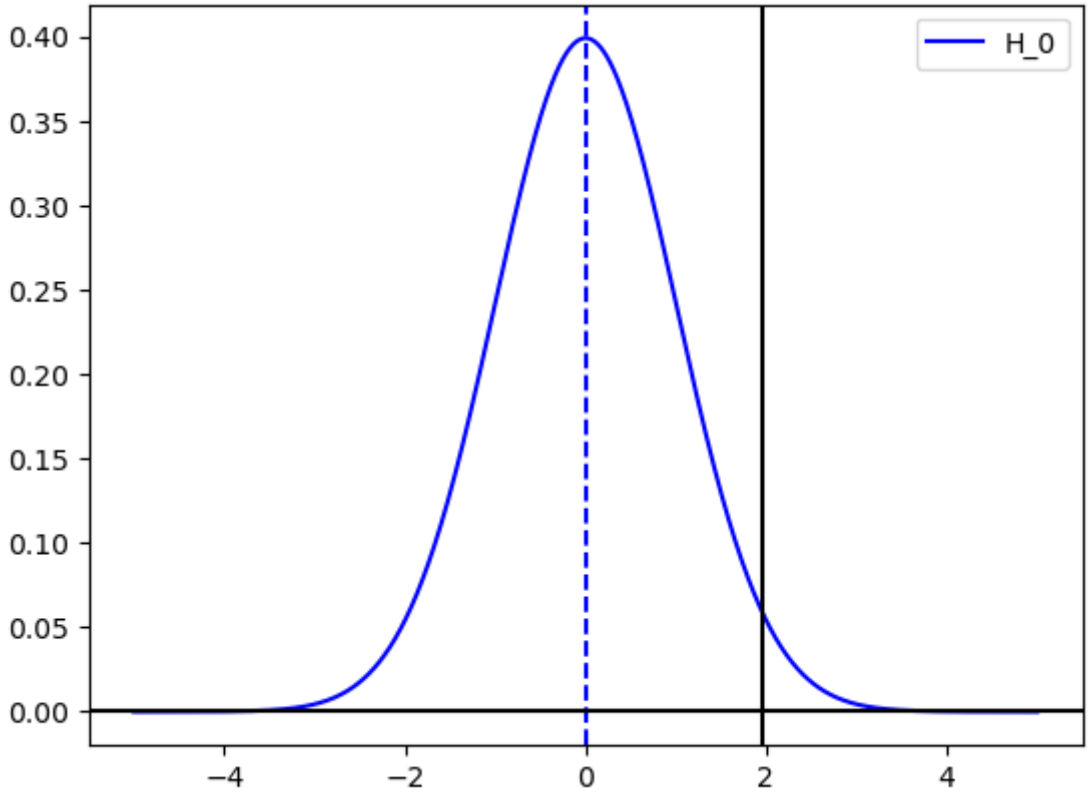
In [164]:
```
# 2-Sample Test
```

The 2-sample test, also known as 2-sample t-test, or the independent samples t-test, is used to test whether the means of two independent groups are significantly different from each other. It's commonly applied when you have two separate groups and you want to determine if their population means on some measure are different.

In the follow example, we test the difference between 150 samples from group 1, with sample mean of 100, sample standard deviation of 10, and 150 samples from group 2, with sample mean of 103, sample standard deviation of 10. We want to answer the question: is the group 1 significantly different from group 2, namely, is the population mean where group 1 is drawn from, is different from that of group 2?

In [165]:
```
# 2-sample test
x_mean = 100
y_mean = 103
s_x = 10
s_y = 10
n_x = 150
n_y = 150

delta = y_mean - x_mean
alpha = 0.05

s_pool = np.sqrt(s_x**2 / n_x + s_y**2 / n_y)
t_stat = delta / s_pool

z_range = np.arange(-5, 5, 0.001)
plt.plot(
    z_range,
    norm.pdf(z_range, loc=0, scale=1),
    color="blue",
    label="H_0"
)

# use q=(1 - alpha) for one-sided test, to find the critical value
z_critical = norm.ppf(q=(1 - alpha / 2))

print(f"The critial value is {z_critical:5.3f}.")
print(f"The test statistic is {t_stat:5.3f}.")

plt.axvline(x=z_critical, color="black")
plt.axvline(x=0, color="blue", linestyle="--")
plt.axhline(y=0, color="black")
plt.legend()
plt.show()
```

```
The critial value is 1.960.
The test statistic is 2.598.
```



In [166]:
```
# Multiple Hypothesis Testing
```

Multiple hypothesis testing refers to the situation in which several hypothesis tests are performed simultaneously on the same dataset. This can be common in fields like genomics, where thousands or even millions of tests might be conducted at once (for example, testing the association of each of many genes with a particular disease). However, conducting multiple tests introduces some challenges.

The more hypothesis tests you perform, the higher the chance of encountering a false positive. For example, if you're testing 100 independent hypotheses and you're using a significance level of $\alpha = 0.05$, you'd expect to have about 5 false positives simply by chance, even if all null hypotheses are true.

Below we will demonstrate this issue numerically. We create 2 identical population, group A and group B. Then we conduct 100 2-sample t-tests, between one sample drawn from group A, and other other drawn from group B. Since we know, statistically, group A and group B are identical. Therefore, significant result (p-value less than 0.05) from any of the 2-sample t-test is an issue (false positive).

In [167]:
```
# multiple tests
np.random.seed(42)
n_samples = 10_000

group_A = np.random.normal(size=n_samples)
group_B = np.random.normal(size=n_samples)

print(ttest_ind(a=group_A, b=group_B))
```

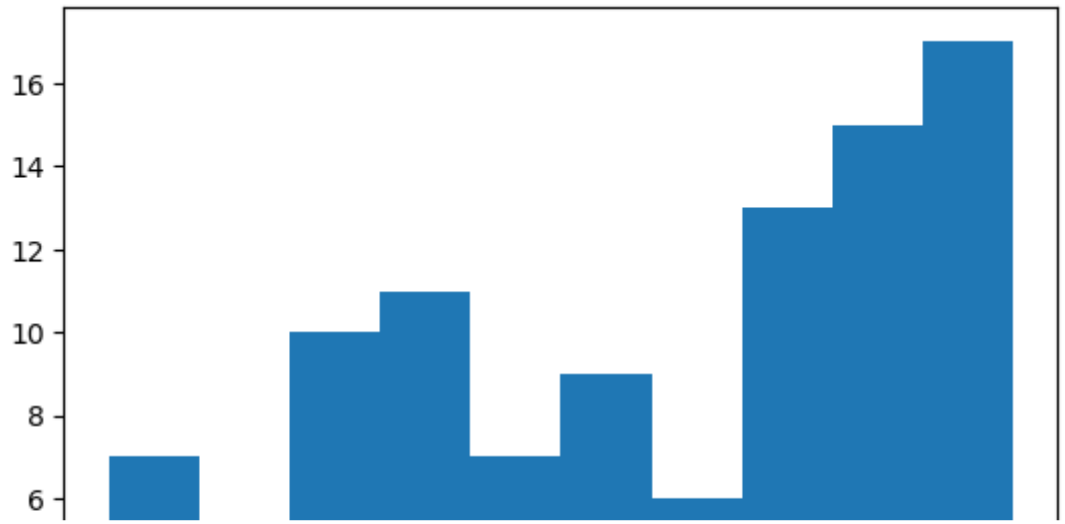
```
Ttest_indResult(statistic=-1.105565678983013, pvalue=0.2689278520062846)
```

In [168]:
```
# assign meaningless labels to each sample
n_labels = 100
group_A_labeled = [(i % n_labels, x) for i, x in enumerate(group_A)]
group_B_labeled = [(i % n_labels, x) for i, x in enumerate(group_B)]

# run t-tests by label
p_values = []
t_statistics = []
for i in tqdm(range(n_labels)):
    t_statistic, p_value =  ttest_ind(
        a=[x for label, x in group_A_labeled if label == i],
        b=[x for label, x in group_B_labeled if label == i]
    )
    t_statistics.append(t_statistic)
    p_values.append(p_value)
```

```
100%|██████████████████████████████████████| 100/100 [00:00<00:00, 2312.78it/s]
```

In [169]:
```
print("Smallest p_values are:")
print(sorted(p_values)[:5])
plt.hist(x=p_values)
plt.show()
```

```
Smallest p_values are:
[0.0016280947657787551, 0.01038592625690243, 0.011904779987561188, 0.028869509284429468, 0.04459902652
806253]
```



In [170]:
```
# Week 4
```

Week 4 - linear_regression.ipynb

```
Introduces:
    - Linear Regression
    - Exploratory Data Analysis (EDA)
    - Simple Linear Regression
    - Multi-varient Linear Regression
    - One-hot Encoding
```

In [171]:
```
from typing import List
from typing import Tuple
from typing import Union

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smf

from tqdm import tqdm

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [172]:
```
# Linear Regression
```

Linear Regression

Linear regression is a statistical method and a fundamental type of predictive analytical model used to describe the relationship between a dependent variable and one or more independent variables. In this class, we will use the "Auto MPG" dataset to learn and practice linear regression.

The data

The "Auto MPG" dataset is one of the classic datasets available in the UCI Machine Learning Repository. It contains city-cycle fuel consumption estimates for various automobiles produced in the 1970s and 1980s, making it a commonly used dataset for regression analysis tasks in machine learning.

Data Set Information:

```
The data concerns city-cycle fuel consumption in miles per gallon (MPG) to be predicted in terms
of 3 multivalued discrete and 5 continuous attributes.

mpg: continuous
cylinders: multi-valued discrete
```

In [173]:
```
data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/auto_mpg.csv")
data.head()
```

Out[173]:

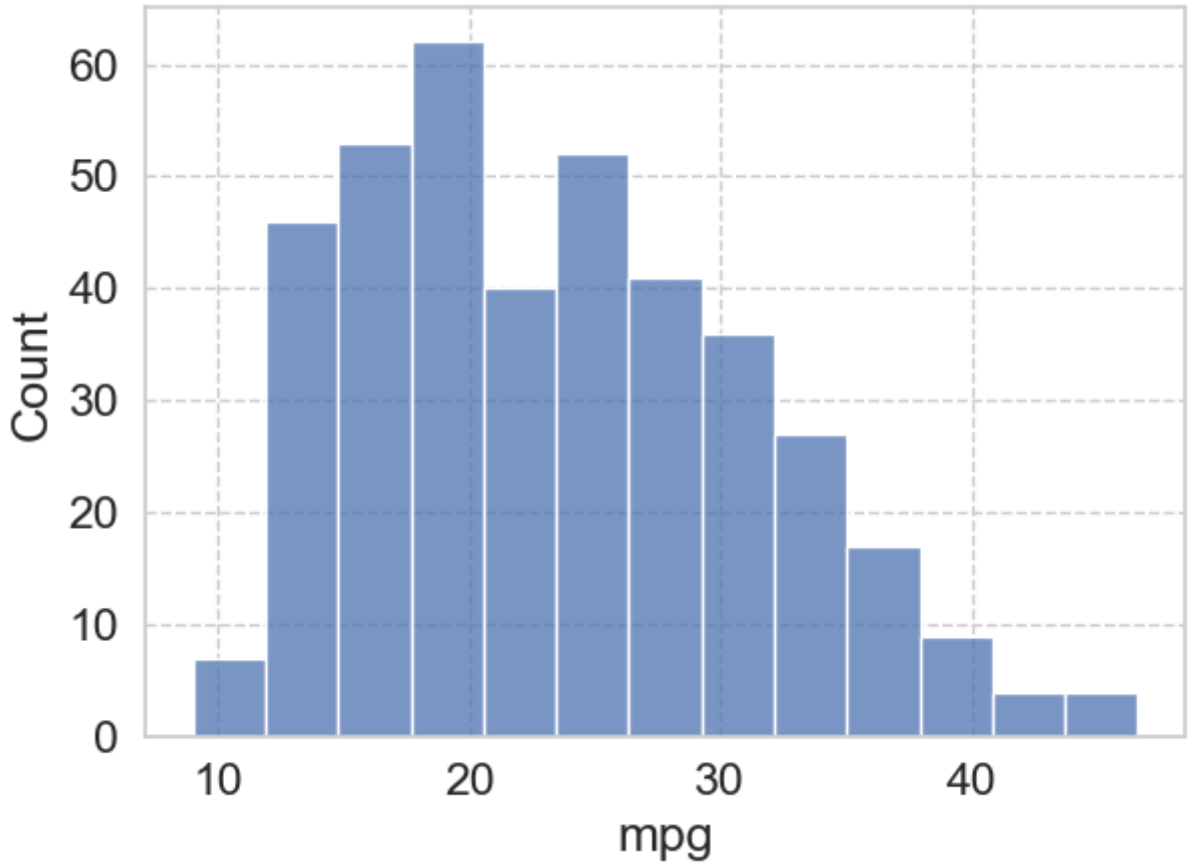
	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

In [174]:
```
# Exploratory Data Analysis (EDA)
```

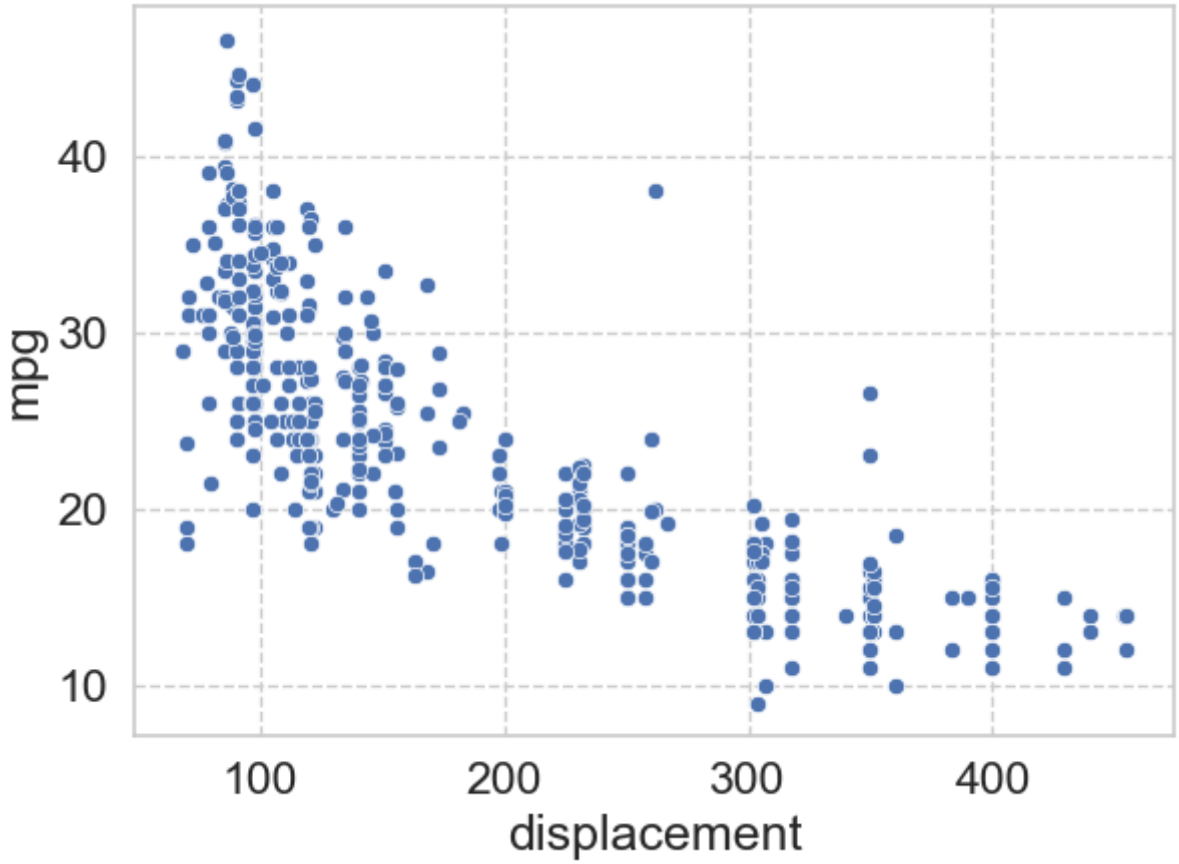
Exploratory Data Analysis (EDA) refers to the process of visually and analytically inspecting data to uncover its main characteristics, often using statistical graphics, plots, and information tables. EDA is essential in data science, and is a fundamental step before proceeding to more advanced machine learning or statistical modeling, as it helps analysts and researchers understand the nature of the data, its structure, anomalies, patterns, and relationships.

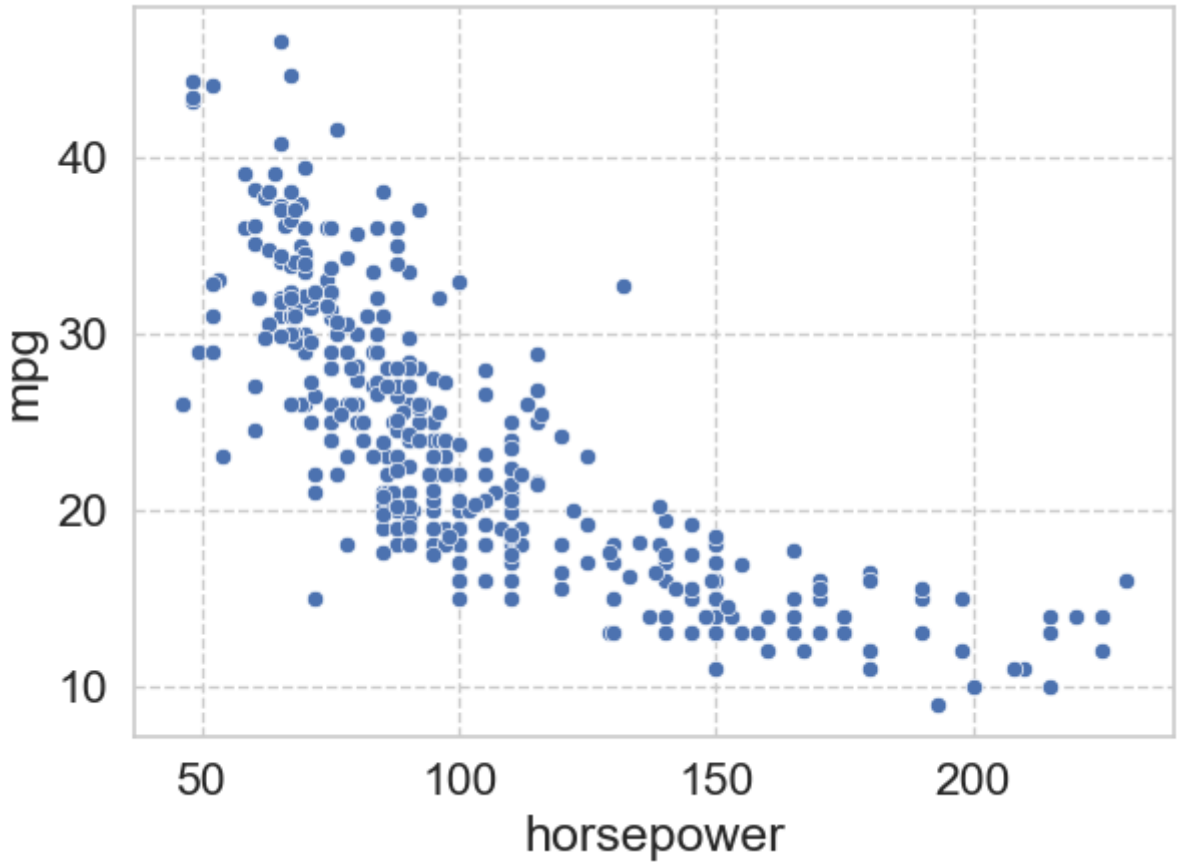
EDA is often an iterative process: as you discover one feature or pattern in your data, it might lead you to look for other related features or patterns. Below, we will use the "Auto MPG" dataset as an example.

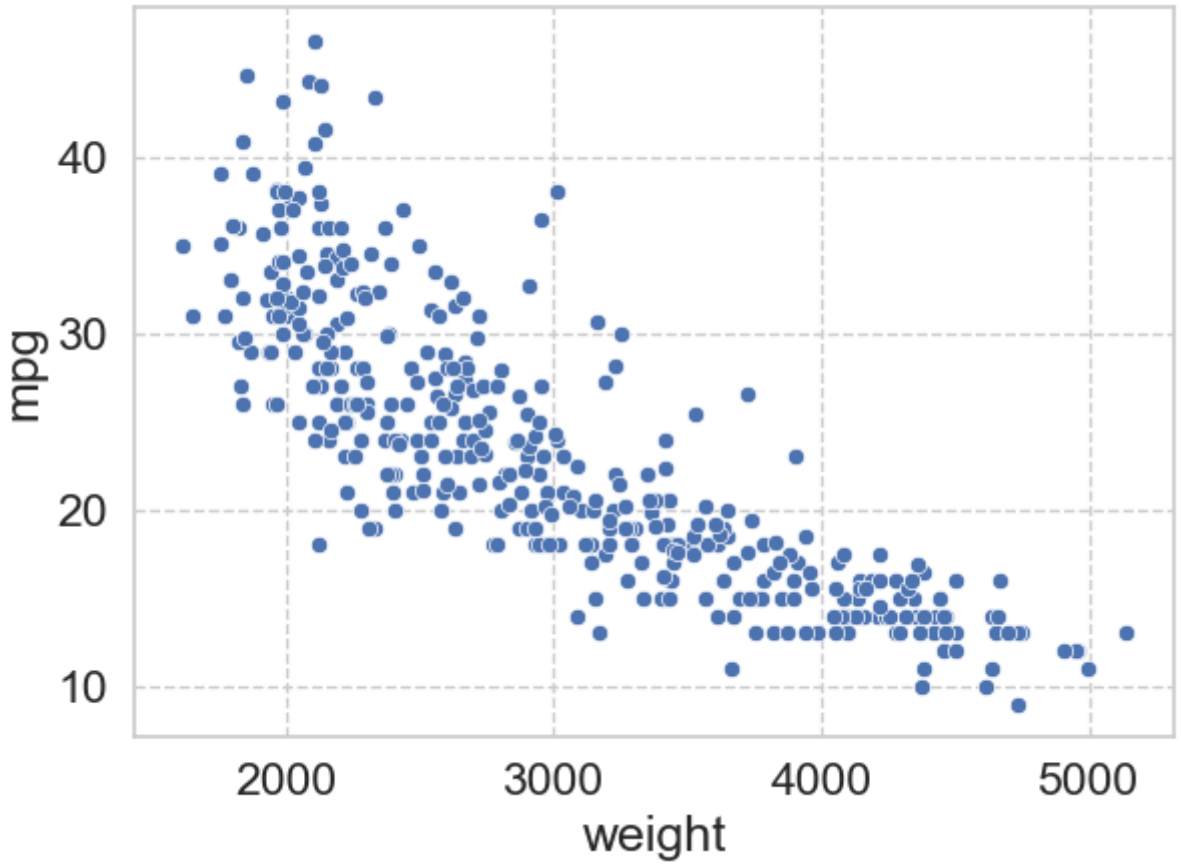
In [175]:
```
# distribution of the dependent variable
sns.histplot(x="mpg", data=data)
plt.tight_layout()
```

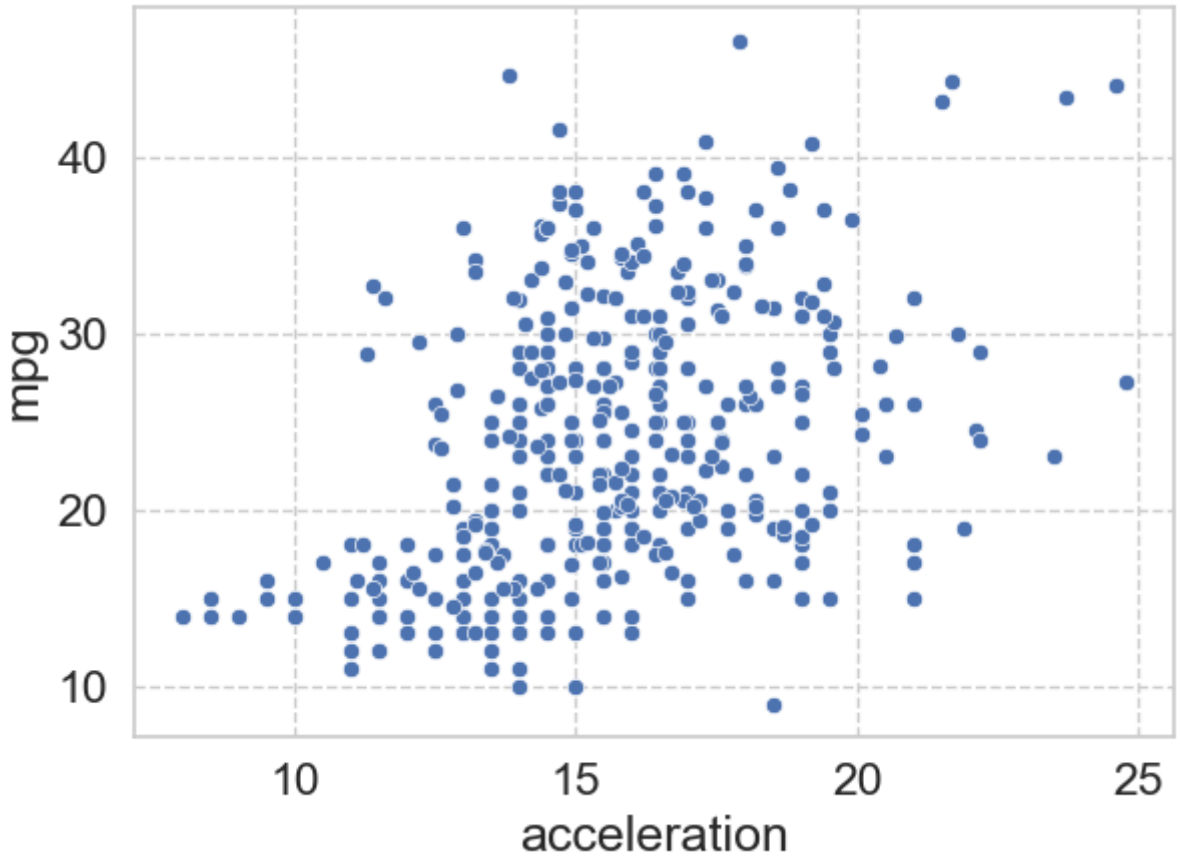


In [176]:
```
# correlations
continuous_variables = [
    "displacement",
    "horsepower",
    "weight",
    "acceleration",
]

for variable in continuous_variables:
    plt.figure()
    sns.scatterplot(x=variable, y="mpg", data=data)
    plt.tight_layout()
```




mpg
40
30
20
10
50
100
150
200
horsepower


40
30
20
10
mpg
2000
3000
4000
5000
weight



In [177]:
```
# Simple Linear Regression
```

Simple linear regression refers to the special case of linear regression where there is only a single independent variable. Namely, it takes the form of:

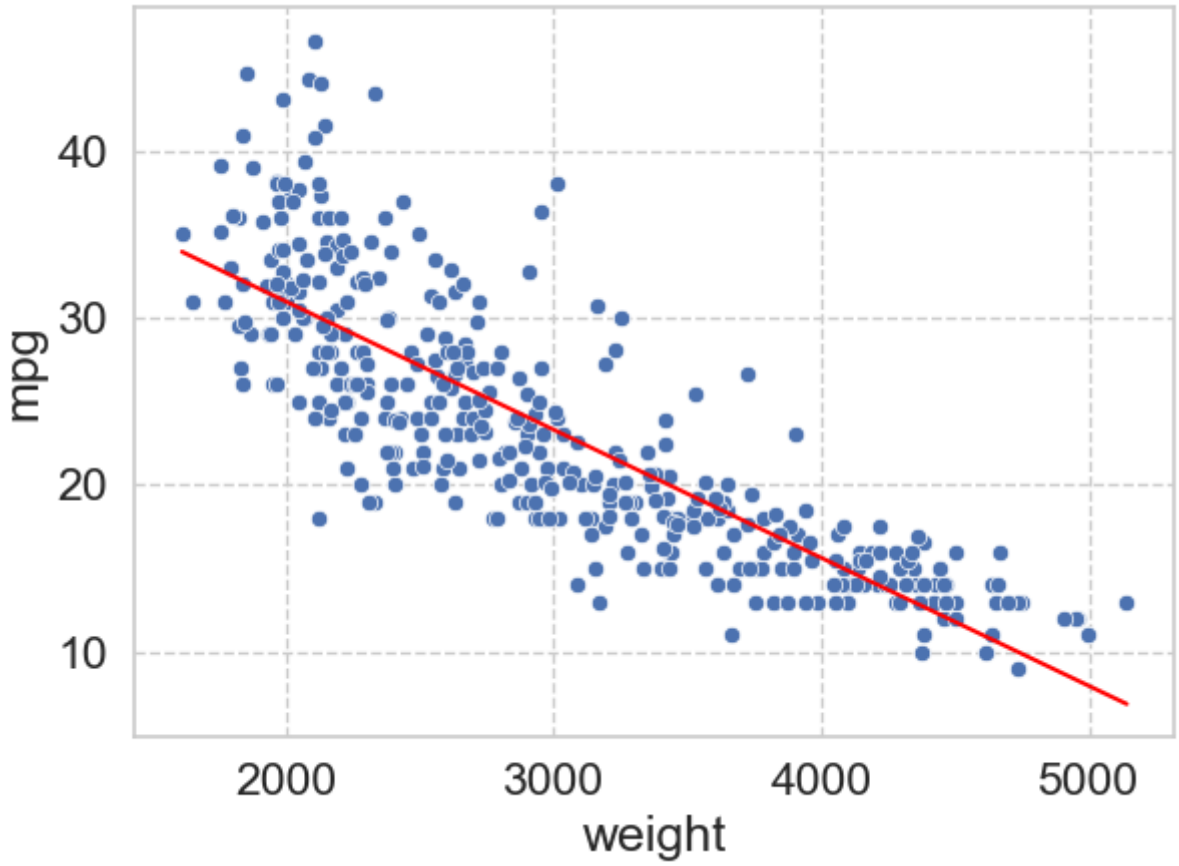
y = β_0 + β_1 * x

where y is the dependent variable, and x is the single independent variable. The goal of the linear regression is to find the best values for the intercept β_0, and the slope β_1.

Below we will use the "Auto MPG" dataset as an example.

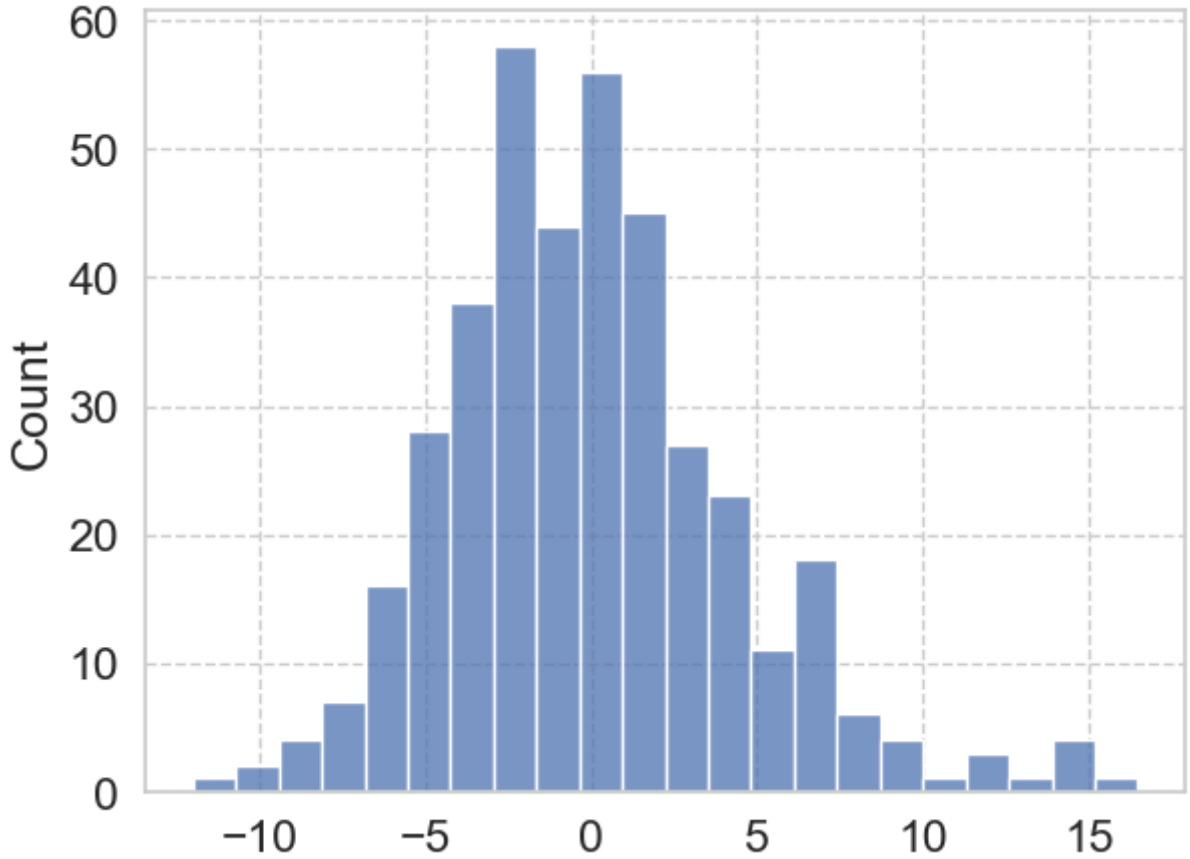
In [178]:
```
y = data["mpg"]
x = data["weight"]

def simple_linear_regression(
    x: Union[List, np.ndarray, pd.Series],
    y: Union[List, np.ndarray, pd.Series]) -> Tuple[float, float]:
    """Return the intercept and slope of a simple linear regression."""
    beta_1 = np.cov(x, y)[0][1] / np.cov(x, x)[0][1]
    beta_0 = np.mean(y) - beta_1 * np.mean(x)

    return beta_0, beta_1

beta_0, beta_1 = simple_linear_regression(x=x, y=y)

# calculate R^2
y_pred = beta_0 + beta_1 * x
SST = np.sum(np.square(y - np.mean(y)))
residual = y - y_pred
SSE = np.sum(np.square(residual))
r2 = 1 - SSE / SST

print(f"beta_0 is: {beta_0:5.4f}")
print(f"beta_1 is: {beta_1:5.4f}")
print(f"R-square is: {r2:5.4f}")

plt.figure()
x_range = np.linspace(start=np.min(x), stop=np.max(x), num=100)
sns.scatterplot(x="weight", y="mpg", data=data)
sns.lineplot(x=x_range, y=(beta_0 + beta_1 * x_range), color="red")
plt.tight_layout()
```

```
beta_0 is: 46.3174
beta_1 is: -0.0077
R-square is: 0.6918
```



Equally important to, if not more important than, the model fitting, is the model/error analysis. Here we will demonstrate a simple residual analysis, to examine if its distirubtion agrees with our (normality) assumption>

In [179]:
```
# residual analysis
plt.figure()
sns.histplot(residual)
plt.tight_layout()
plt.show()
```

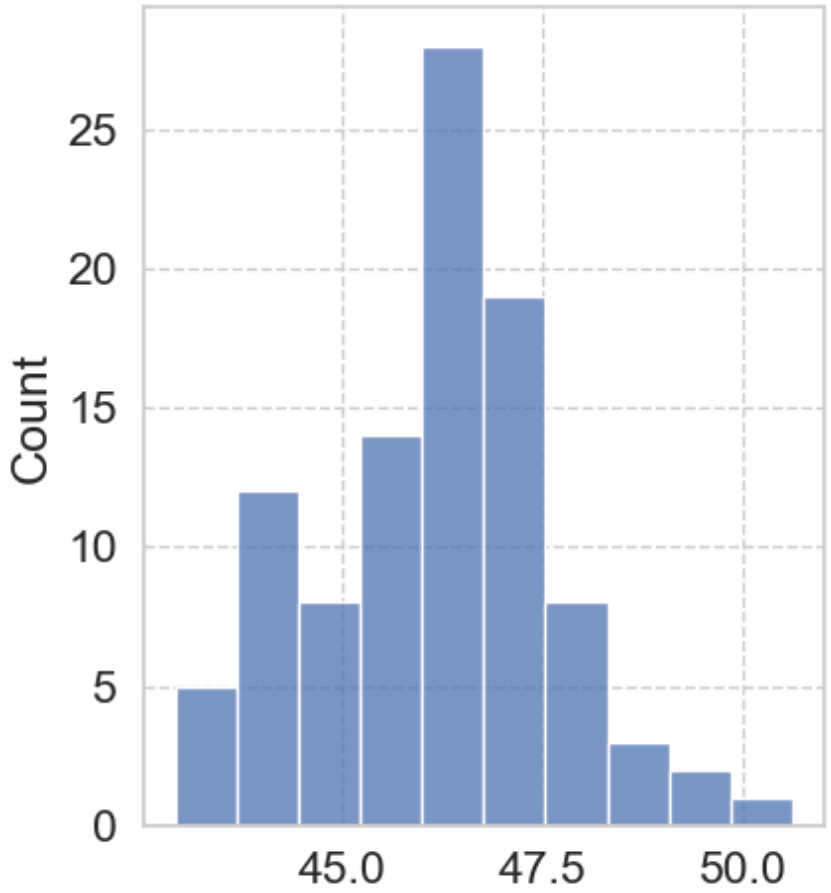


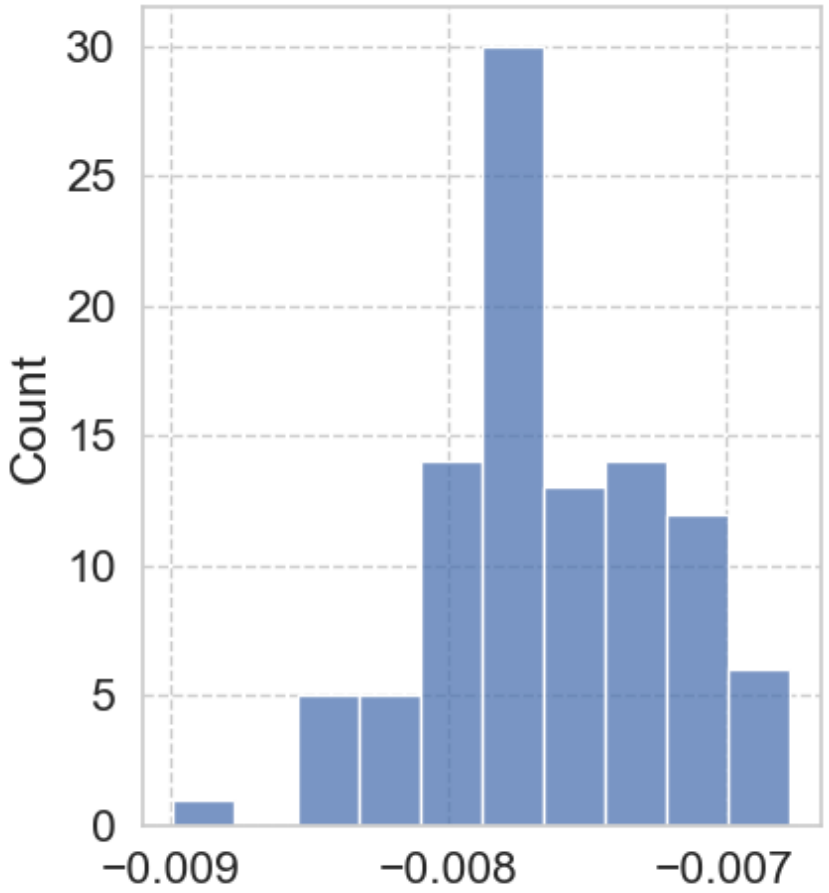
An important aspect in linear regression is that, the fitted parameters (e.g., intercept, slope), are just point estimates of the population parameter. Here the "population" refers to the underlying stochastic process that generates the observed data.

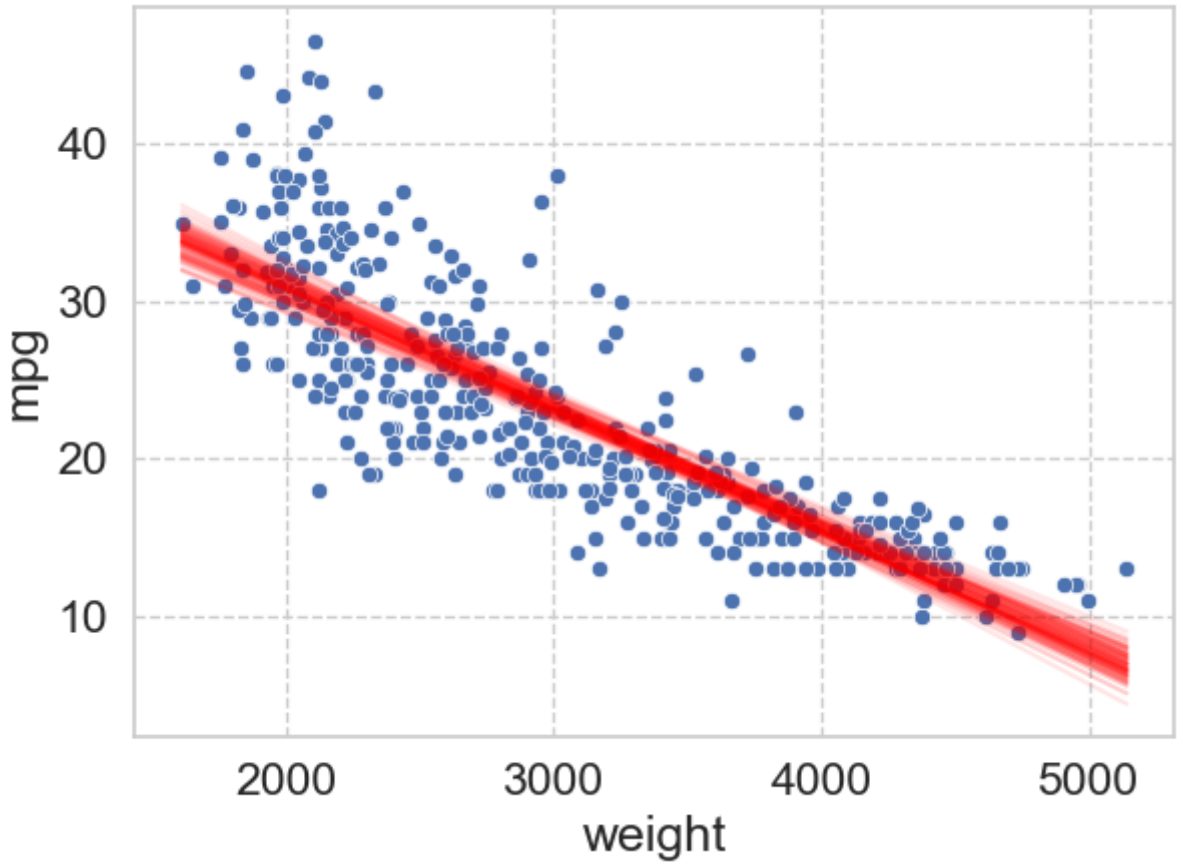
Below, we will run a simulation: in each trial, we only sample 20% from the full observation, and run the simple linear regression. We will repeat the trial 100 times. This is to show that, the fitting parameters is a random variable: it depends on the exact data that is used. Therefore, as a random variable, we need to pay attention to its bias and variance.

In [180]:
```
# accuracy of the coefficients
np.random.seed(42)
n_trials = 100

beta_0s, beta_1s = [], []
for _ in tqdm(range(n_trials)):
    sampling_proba = 0.2
    mask = np.random.choice([True, False], size=len(x), p=[sampling_proba, 1 - sampling_proba])
    x_sampled, y_sampled = x[mask], y[mask]
    beta_0, beta_1 = simple_linear_regression(x=x_sampled, y=y_sampled)
    beta_0s.append(beta_0)
    beta_1s.append(beta_1)

# plot the histograms
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 5))
sns.histplot(beta_0s, ax=axes[0])
sns.histplot(beta_1s, ax=axes[1])
plt.tight_layout()

# plot the fited lines
plt.figure()
x_range = np.linspace(start=np.min(x), stop=np.max(x), num=100)
for i in tqdm(range(len(beta_0s))):
    sns.lineplot(x=x_range, y=(beta_0s[i] + beta_1s[i] * x_range), color="red", alpha=0.1)
sns.scatterplot(x="weight", y="mpg", data=data)
plt.tight_layout()
```

```
100%|██████████| 100/100 [00:00<00:00, 3395.13it/s]
100%|██████████| 100/100 [00:00<00:00, 251.80it/s]
```


Count
25
20
15
10
5
0
45.0
47.5
50.0


Count
30
25
20
15
10
5
0
−0.009
−0.008
−0.007



In [181]:
```
# confidence intervals
SE_beta_0 = (np.var(residual, ddof=2) * (1. / len(x) + (np.mean(x))**2 / np.sum((x - np.mean(x))**2)))*
SE_beta_1 = (np.var(residual, ddof=2) / np.sum((x - np.mean(x))**2))**0.5

print(f"The standard error for beta_0 is: {SE_beta_0:5.4f}")
print(f"The standard error for beta_1 is: {SE_beta_1:5.4f}")
```

```
The standard error for beta_0 is: 0.7952
The standard error for beta_1 is: 0.0003
```

In [182]:
```
# simple linear regression with the `statsmodels` library
model_1 = smf.ols(formula='mpg ~ weight', data=data)
result_1 = model_1.fit()
print(result_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.692
Model:                            OLS   Adj. R-squared:                  0.691
Method:                 Least Squares   F-statistic:                     888.9
Date:                Mon, 11 Dec 2023   Prob (F-statistic):          2.97e-103
Time:                        13:23:46   Log-Likelihood:                -1148.4
No. Observations:                 398   AIC:                             2301.
Df Residuals:                     396   BIC:                             2309.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     46.3174      0.795     58.243      0.000      44.754      47.881
weight        -0.0077      0.000    -29.814      0.000      -0.008      -0.007
==============================================================================
Omnibus:                       40.423   Durbin-Watson:                   0.797
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               56.695
Skew:                           0.713   Prob(JB):                     4.89e-13
Kurtosis:                       4.176   Cond. No.                     1.13e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.13e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [183]:
```
# linear regress in matrix format
X = np.hstack(
    (np.ones(shape=(len(x), 1)),
     x.to_numpy().reshape(-1, 1)))

# point estimate
beta_matrix = np.linalg.inv(X.T @ X) @ X.T @ y
print("The estimates for beta are:")
print(beta_matrix)

# variance
se_matrix = np.sqrt(np.var(residual, ddof=2) * np.linalg.inv(X.T @ X))
print("\nThe standard error for beta are:")
print(se_matrix)
```

```
The estimates for beta are:
[ 4.63173644e+01 -7.67661006e-03]

The standard error for beta are:
[[7.95245230e-01            nan]
 [           nan 2.57486863e-04]]

/var/folders/3v/gxl3z6yn3fd_gv2bypr81hl80000gn/T/ipykernel_1766/1283480099.py:12: RuntimeWarning: inva
lid value encountered in sqrt
  se_matrix = np.sqrt(np.var(residual, ddof=2) * np.linalg.inv(X.T @ X))
```

In [184]:
```
# Multi-variant Linear Regression
```

Multi-variant linear regression is a linear regression that takes more than one independent variable.

In the following, we will use the Python statsmodels library to perform multi-variant linear regression.

In [185]:
```
model_2 = smf.ols(formula='mpg ~ weight + displacement + horsepower + acceleration', data=data)
result_2 = model_2.fit()
print(result_2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.707
Model:                            OLS   Adj. R-squared:                  0.704
Method:                 Least Squares   F-statistic:                     233.4
Date:                Mon, 11 Dec 2023   Prob (F-statistic):          9.63e-102
Time:                        13:23:52   Log-Likelihood:                -1120.6
No. Observations:                 392   AIC:                             2251.
Df Residuals:                     387   BIC:                             2271.
Df Model:                           4
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept       45.2511      2.456     18.424      0.000      40.422      50.080
weight          -0.0053      0.001     -6.512      0.000      -0.007      -0.004
displacement    -0.0060      0.007     -0.894      0.372      -0.019       0.007
horsepower      -0.0436      0.017     -2.631      0.009      -0.076      -0.011
acceleration    -0.0231      0.126     -0.184      0.854      -0.270       0.224
```

In [186]:
```
# correlation between the continuous variables
data[continuous_variables].corr()
```

Out[186]:

	displacement	horsepower	weight	acceleration
displacement	1.000000	0.897257	0.932824	-0.543684
horsepower	0.897257	1.000000	0.864538	-0.689196
weight	0.932824	0.864538	1.000000	-0.417457
acceleration	-0.543684	-0.689196	-0.417457	1.000000

In [187]:
```
# One-hot Encoding
```

If the independent variable is of categorical type, we can not directly use them in regression. One-hot encoding is the solution to address this issue.

At its core, one-hot encoding is about converting nominal categorical data into a format that can be used by machine learning algorithms. Categorical data refers to variables that can be divided into multiple categories but have no order or priority. These categories can be text labels, which many ML algorithms can't work with directly because they expect numerical input features.

With the statsmodels library, you simply wrap the independent variable with C(). If you are using other libraries, please refer to the corresponding APIs. For example, in the popular scikit-learn library, one can use the OneHotEncoder to achieve the same goal.

In [188]:
```
# one-hot encode the categorical variables
model_3 = smf.ols(formula='mpg ~ weight + C(origin)', data=data)
result_3 = model_3.fit()
print(result_3.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.702
Model:                            OLS   Adj. R-squared:                  0.699
Method:                 Least Squares   F-statistic:                     308.6
Date:                Mon, 11 Dec 2023   Prob (F-statistic):          4.86e-103
Time:                        13:23:55   Log-Likelihood:                -1142.0
No. Observations:                 398   AIC:                             2292.
Df Residuals:                     394   BIC:                             2308.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     43.6959      1.104     39.567      0.000      41.525      45.867
C(origin)[T.2.0]     1.2155      0.652      1.863      0.063      -0.067       2.498
C(origin)[T.3.0]     2.3554      0.662      3.558      0.000       1.054       3.657
weight        -0.0070      0.000    -22.059      0.000      -0.008      -0.006
==============================================================================
Omnibus:                       37.803   Durbin-Watson:                   0.813
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               54.615
Skew:                           0.662   Prob(JB):                     1.38e-12
Kurtosis:                       4.242   Cond. No.                     1.82e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.82e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [189]:
```
# Week 5
```

Week 5 - linear_regression_2.ipynb

```
Introduces:
    - Auto MPG dataset
    - Data representation
    - Simple linear regression
    - Confidence interval
    - Prediction interval
    - Multiple linear regression
    - Multicollinearity
```

In [190]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smf
from scipy.stats import t

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [191]:
```
# Automobile MPG dataset - used for demonstrative purposes

data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/auto_mpg.csv")
data.head()
```

Out[191]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

In [192]:
```
# Data representation
```

It is very common to represent the data in the matrix format, and with pandas, we can extract the underlying data as numpy.ndarray.

In [193]:
```
# Retrieve data as a numpy array
Y = data[["mpg"]].values
print(Y.shape)
Y[:5]
```

```
(398, 1)
```

Out[193]:
```
array([[18.],
       [15.],
       [18.],
       [16.],
       [17.]])
```

In [194]:
```
# Simple linear regression
```

In simple linear regression, there is only one independent variable. The goal of the (simple) linear regression, is to find the best values of coefficients, β, that minimize the error sum of squares.

To find β, we can simply apply calculus, via the so-call Normal equation.

In [195]:
```
X = np.append(np.ones(shape=(data.shape[0], 1)), data[["weight"]].values, axis=1)
print(X.shape)
X[:5]
```

```
(398, 2)
```

Out[195]:
```
array([[1.000e+00, 3.504e+03],
       [1.000e+00, 3.693e+03],
       [1.000e+00, 3.436e+03],
       [1.000e+00, 3.433e+03],
       [1.000e+00, 3.449e+03]])
```

In [196]:
```
# Calculates coefficients
betas = np.linalg.inv(X.T @ X) @ X.T @ Y
betas
```

Out[196]:
```
array([[ 4.63173644e+01],
       [-7.67661006e-03]])
```

In [197]:
```
# Calculates the standard error
Y_hat = X @ betas
residual = Y - Y_hat
var = np.var(residual, ddof=X.shape[1])

se = np.sqrt(var * np.linalg.inv(X.T @ X))
se
```

```
/var/folders/3v/gxl3z6yn3fd_gv2bypr81hl80000gn/T/ipykernel_1766/1414136738.py:6: RuntimeWarning: inval
id value encountered in sqrt
  se = np.sqrt(var * np.linalg.inv(X.T @ X))
```

Out[197]:
```
array([[7.95245230e-01,            nan],
       [           nan, 2.57486863e-04]])
```

In [198]:
```
# Calculates R2
r2 = np.power(Y_hat - np.mean(Y), 2).sum() / np.power(Y - np.mean(Y), 2).sum()
r2
```

Out[198]:
```
0.6917929800341555
```

In [199]:
```
# Confidence interval and prediction interval
```

Confidence interval and prediction interval

Both confidence intervals and prediction intervals are used to convey the uncertainty associated with estimates or predictions from regression models, but they serve different purposes and are interpreted differently.

Confidence Interval (CI):

A confidence interval provides a range of values for some parameter (e.g., the mean or a regression coefficient) based on the data from a sample. It captures the uncertainty about the parameter estimate in repeated sampling. If we were to collect many samples and compute a confidence interval from each of them, we expect a certain proportion (e.g., 95% for a 95% CI) of those intervals to contain the true

parameter value. For a 95% confidence interval, we'd say that we are 95% confident that the true parameter value lies within this interval.

Prediction Interval (PI):

A prediction interval provides a range within which a new individual observation is expected to fall with a certain probability, given what we know from a set of data. It captures both the uncertainty associated with the estimated regression line (or mean) and the variability of individual data points around that line. For a 95% prediction interval, we'd say that we are 95% confident that a new individual observation will fall within this interval.

In [200]:
```
# confidence interval
x_new = 4000

y_hat = (betas[0] + betas[1] * x_new)[0]
print(y_hat)

delta = np.sqrt(var) * np.sqrt(1 / X.shape[0] + (x_new - np.mean(X[:, 1]))**2 / np.sum((X[:, 1] - np.mea
multiplier = 1.96
# multiplier = t.ppf(q=0.975, df=X.shape[0] - X.shape[1])

print(f"The lower bound of the 95% CI is: {y_hat - multiplier * delta:5.3f}")
print(f"The upper bound of the 95% CI is: {y_hat + multiplier * delta:5.3f}")
```

```
15.610924164559783
The lower bound of the 95% CI is: 14.938
The upper bound of the 95% CI is: 16.283
```

In [201]:
```
delta = np.sqrt(var) * np.sqrt(1 + 1 / X.shape[0] + (x_new - np.mean(X[:, 1]))**2 / np.sum((X[:, 1] - np

print(f"The lower bound of the 95% PI is: {y_hat - multiplier * delta:5.3f}")
print(f"The upper bound of the 95% PI is: {y_hat + multiplier * delta:5.3f}")
```

```
The lower bound of the 95% PI is: 7.069
The upper bound of the 95% PI is: 24.153
```

In [202]:
```
# Multiple linear regression
```

For the general, namely, multiple linear regression, we consider more than one independent variable. All the concepts from the simple linear regression still apply.

In [203]:
```
X = data[["weight", "acceleration"]].values
X = np.append(np.ones((X.shape[0], 1)), X, axis=1)
print(X.shape)
X[:5]
```

```
(398, 3)
```

Out[203]:
```
array([[1.000e+00, 3.504e+03, 1.200e+01],
       [1.000e+00, 3.693e+03, 1.150e+01],
       [1.000e+00, 3.436e+03, 1.100e+01],
       [1.000e+00, 3.433e+03, 1.200e+01],
       [1.000e+00, 3.449e+03, 1.050e+01]])
```

In [204]:
```
# Calculates coefficients
betas = np.linalg.inv(X.T @ X) @ X.T @ Y
betas
```

Out[204]:
```
array([[ 4.13998283e+01],
       [-7.33564480e-03],
       [ 2.50815893e-01]])
```

In [205]:
```
# Calculates the standard error
Y_hat = X @ betas
residual = Y - Y_hat
var = np.var(residual, ddof=X.shape[1])

se = np.sqrt(var * np.linalg.inv(X.T @ X))
se
```

```
/var/folders/3v/gxl3z6yn3fd_gv2bypr81hl80000gn/T/ipykernel_1766/1414136738.py:6: RuntimeWarning: inval
id value encountered in sqrt
  se = np.sqrt(var * np.linalg.inv(X.T @ X))
```

Out[205]:
```
array([[1.86477532e+00,            nan,            nan],
       [           nan, 2.80724803e-04, 3.17844957e-03],
       [           nan, 3.17844957e-03, 8.62060575e-02]])
```

In [206]:
```
# Calculates R2
r2 = np.power(Y_hat - np.mean(Y), 2).sum() / np.power(Y - np.mean(Y), 2).sum()
r2
```

Out[206]:
```
0.6982595061815179
```

In [207]:
```
# Linear regression with the `statsmodels` library
model_1 = smf.ols(formula='mpg ~ weight + acceleration', data=data)
result_1 = model_1.fit()
print(result_1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.698
Model:                            OLS   Adj. R-squared:                  0.697
Method:                 Least Squares   F-statistic:                     457.0
Date:                Mon, 11 Dec 2023   Prob (F-statistic):          1.69e-103
Time:                        13:24:14   Log-Likelihood:                -1144.2
No. Observations:                 398   AIC:                             2294.
Df Residuals:                     395   BIC:                             2306.
Df Model:                           2
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept       41.3998      1.865     22.201      0.000      37.734      45.066
weight          -0.0073      0.000    -26.131      0.000      -0.008      -0.007
acceleration     0.2508      0.086      2.909      0.004       0.081       0.420
==============================================================================
Omnibus:                       30.694   Durbin-Watson:                   0.804
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               38.296
Skew:                           0.625   Prob(JB):                     4.83e-09
Kurtosis:                       3.864   Cond. No.                     2.67e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.67e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [208]:
```
# One-hot encode the categorical variables
model_2 = smf.ols(formula='mpg ~ weight + C(origin)', data=data)
result_2 = model_2.fit()
print(result_2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.702
Model:                            OLS   Adj. R-squared:                  0.699
Method:                 Least Squares   F-statistic:                     308.6
Date:                Mon, 11 Dec 2023   Prob (F-statistic):          4.86e-103
Time:                        13:24:15   Log-Likelihood:                -1142.0
No. Observations:                 398   AIC:                             2292.
Df Residuals:                     394   BIC:                             2308.
Df Model:                           3
Covariance Type:            nonrobust
===================================================================================
                      coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------
Intercept          43.6959      1.104     39.567      0.000      41.525      45.867
C(origin)[T.2.0]    1.2155      0.652      1.863      0.063      -0.067       2.498
C(origin)[T.3.0]    2.3554      0.662      3.558      0.000       1.054       3.657
weight             -0.0070      0.000    -22.059      0.000      -0.008      -0.006
==============================================================================
Omnibus:                       37.803   Durbin-Watson:                   0.813
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               54.615
Skew:                           0.662   Prob(JB):                     1.38e-12
Kurtosis:                       4.242   Cond. No.                     1.82e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.82e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [209]:
```
# Multicollinearity
```

Multicollinearity

Multicollinearity refers to a situation in multiple (linear) regression where two or more independent variables (predictors) are highly correlated with each other. This means that they have similar patterns of variance and are providing redundant information in predicting the dependent variable.

`Consequences of Multicollinearity:`

- Coefficients Become Unstable: The estimated coefficients become sensitive to small changes in the model. A slight change in the data can lead to large changes in the coefficients.
- Uncertainty in Coefficients: The standard errors of the coefficients increase, which means the coefficients might not be statistically significant even if they truly have an effect.
- Interpretation Issues: Because the variables are correlated, it becomes difficult to disentangle the individual effect of one predictor

In [210]:
```
# Multicollinearity
model_3 = smf.ols(formula='mpg ~ weight + displacement + horsepower + acceleration', data=data)
result_3 = model_3.fit()
print(result_3.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      mpg   R-squared:                       0.707
Model:                              OLS   Adj. R-squared:                  0.704
Method:                   Least Squares   F-statistic:                     233.4
Date:                  Mon, 11 Dec 2023   Prob (F-statistic):          9.63e-102
Time:                          13:24:18   Log-Likelihood:                -1120.6
No. Observations:                   392   AIC:                             2251.
Df Residuals:                       387   BIC:                             2271.
Df Model:                             4
Covariance Type:              nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept       45.2511      2.456     18.424      0.000      40.422      50.080
weight          -0.0053      0.001     -6.512      0.000      -0.007      -0.004
displacement    -0.0060      0.007     -0.894      0.372      -0.019       0.007
horsepower      -0.0436      0.017     -2.631      0.009      -0.076      -0.011
acceleration    -0.0231      0.126     -0.184      0.854      -0.270       0.224
==============================================================================
Omnibus:                       38.359   Durbin-Watson:                   0.861
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               51.333
Skew:                           0.715   Prob(JB):                     7.13e-12
Kurtosis:                       4.049   Cond. No.                     3.56e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.56e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [211]:
```
# week 5 cont.
```

Week 5 - logistic_regression.ipynb

```
Introduces:
    - Binary classification example
    - EDA
```

In [212]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

pd.set_option("display.max_columns", 500)
sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [213]:
```
# Binary classification example
```

In [214]:
```
breast_cancer_data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/br
breast_cancer_data.head()
```

Out[214]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

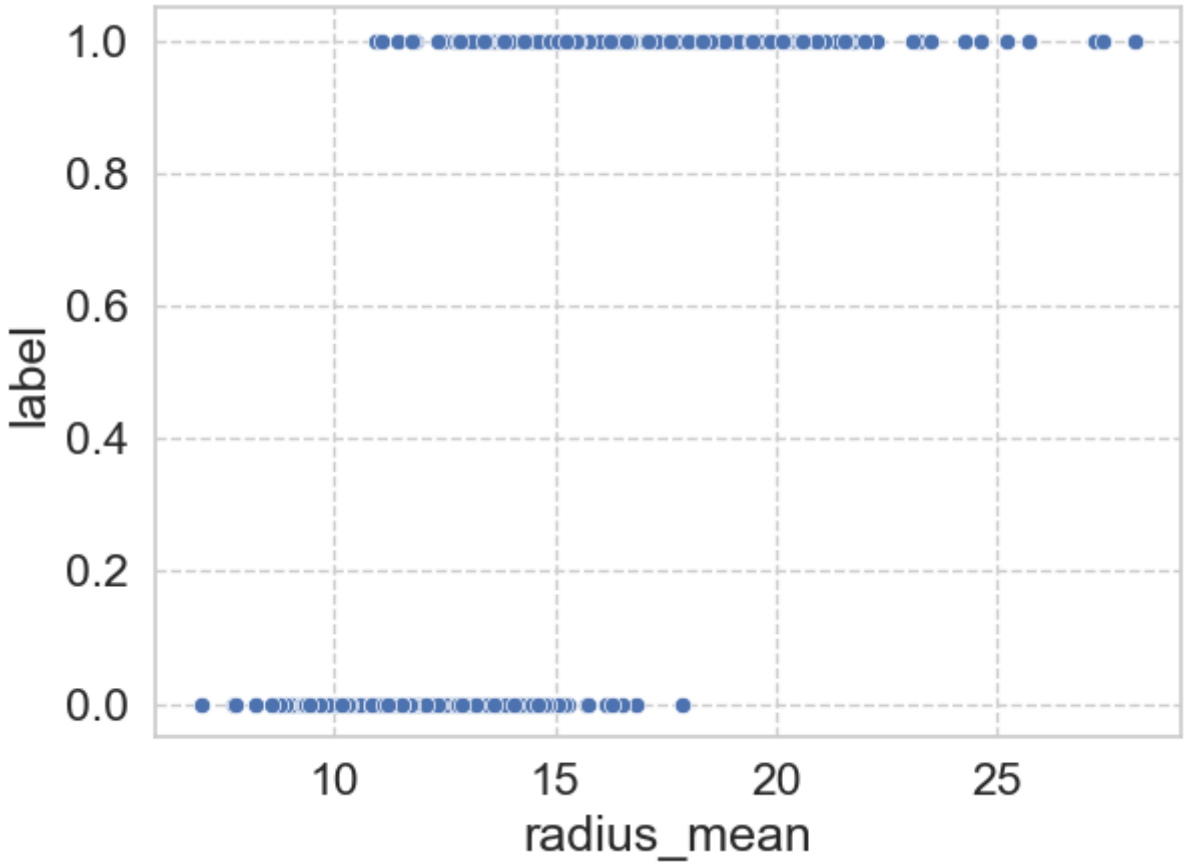
In [215]:
```
titanic_data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/titanic.
titanic_data.head()
```

Out[215]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

In [216]:
```
# EDA
```

In [217]:
```
breast_cancer_data["label"] = breast_cancer_data["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
sns.scatterplot(x="radius_mean", y="label", data=breast_cancer_data)
plt.tight_layout()
```



In [218]:
```
# Logistic regression
```

In [219]:
```
model = smf.glm(
    formula='label ~ radius_mean',
    data=breast_cancer_data,
    family=sm.families.Binomial(),
)
result = model.fit()
print(result.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                   label   No. Observations:                  569
Model:                             GLM   Df Residuals:                      567
Model Family:                Binomial   Df Model:                            1
Link Function:                   Logit   Scale:                          1.0000
Method:                           IRLS   Log-Likelihood:                -165.01
Date:                Mon, 11 Dec 2023   Deviance:                       330.01
Time:                        13:24:28   Pearson chi2:                     489.
No. Iterations:                     7   Pseudo R-squ. (CS):             0.5232
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -15.2459      1.325    -11.509      0.000     -17.842     -12.649
radius_mean     1.0336      0.093     11.100      0.000       0.851       1.216
==============================================================================
```

In [220]:
```
# Week 6
```

Week 6 - auc_roc.ipynb

```
Introduces:
    - Demonstration of AUC of ROC
```

In [221]:
```
# Demonstration of AUC of ROC
```

ROC stands for "Receiver Operating Characteristic." The ROC curve is a graphical representation that illustrates the performance of a binary classifier system as its discrimination threshold is varied. It's one of the most commonly used metrics for evaluating the performance of machine learning algorithms, particularly in problems where the classes are imbalanced.

In [222]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [223]:
```
cancer_data = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/breast_ca
cancer_data["label"] = cancer_data["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
cancer_data.head()
```

Out[223]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

In [224]:
```
# fit the logistic regression model
lr_model = LogisticRegression(penalty="none", random_state=42)
lr_model.fit(
    X=cancer_data[["radius_mean"]],
    y=cancer_data["label"],
)
```

```
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:118
2: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the p
ast behaviour, set `penalty=None`.
  warnings.warn(
```

Out[224]:
```
LogisticRegression(penalty='none', random_state=42)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [225]:
```
# make prediction on the training dataset
prediction_results = pd.DataFrame(
    data={
        "probability": lr_model.predict_proba(cancer_data[["radius_mean"]])[:, 1],
        "label": cancer_data["label"],
    }
)

# take a look some randome samples
prediction_results.sample(10, random_state=123).sort_values("probability")
```

Out[225]:

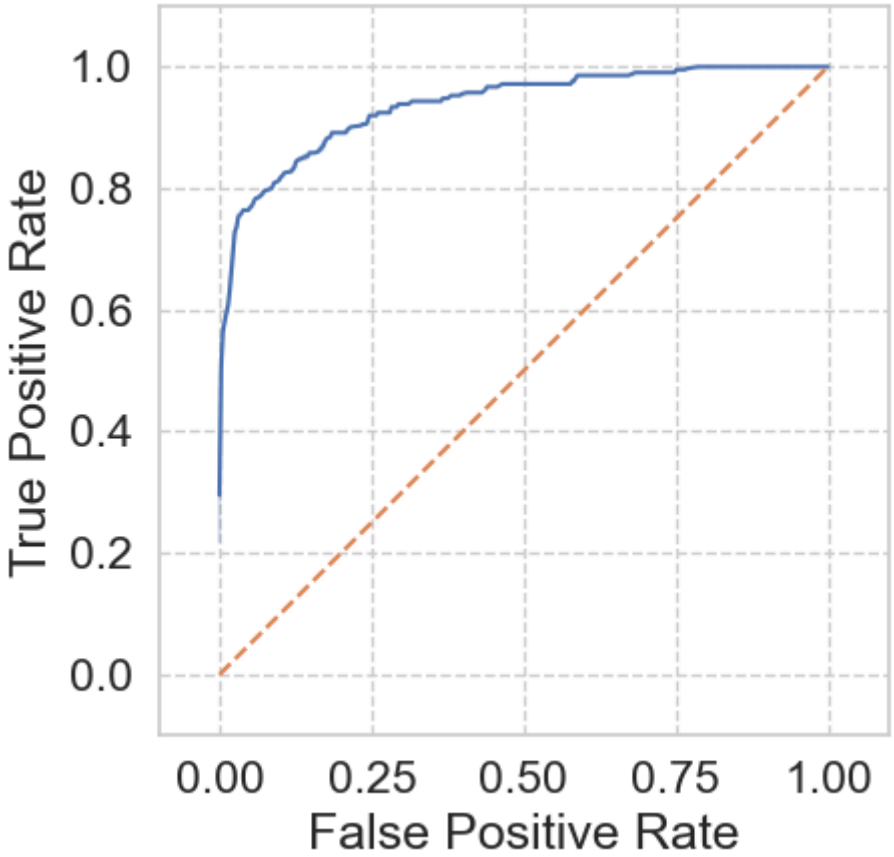
	probability	label
273	0.005616	0
333	0.026138	0
469	0.037853	0
297	0.043490	1
216	0.049435	0
178	0.141994	0
500	0.574265	0
209	0.631112	0
201	0.947007	1
85	0.978835	1

In [226]:
```
# calculate the values needed for the auc of roc
fpr, tpr, thresholds = roc_curve(
    y_true=prediction_results["label"],
    y_score=prediction_results["probability"],
)

roc_data = pd.DataFrame(
    data={
        "fpr": fpr,
        "tpr": tpr,
        "thresholds": thresholds,
    },
)

roc_data.tail()
```

Out[226]:

	fpr	tpr	thresholds
183	0.871148	1.0	0.006633
184	0.876751	1.0	0.006445
185	0.887955	1.0	0.005692
186	0.893557	1.0	0.005616
187	1.000000	1.0	0.000325

In [227]:
```
# make plot
plt.figure()
# plot the ROC
sns.lineplot(x=fpr, y=tpr)
# plot the diagonal line
sns.lineplot(x=[0, 1], y=[0, 1], linestyle="--")


plt.gca().set_aspect("equal")
plt.gca().set_xlabel("False Positive Rate")
plt.gca().set_ylabel("True Positive Rate")
plt.gca().set_xlim([-.1, 1.1])
plt.gca().set_ylim([-.1, 1.1])
plt.tight_layout()
plt.show()
```



Week 6 cont - gradient_descent.ipynb

```
Introduces:
    - Gradient descent
    - Linear regression example
```

In [228]:
```
# Gradient descent
```

Gradient descent

Gradient descent (GD) is an iterative optimization algorithm used to find the minimum of a function. It's widely used in machine learning and deep learning for training models by minimizing a loss function.

In [229]:
```
%matplotlib inline
from typing import Callable, Dict, List, Tuple, Union

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smf

from matplotlib import cm

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [230]:
```
# Linear regression example
```

Let's apply GD to the previously studied linear regression problem, where the loss function is the sum of squared error.

We will use GD to find the values of intercept and slope that minimize the loss function.

In [231]:
```
auto = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/auto_mpg.csv")
# drop the null values
auto.dropna(inplace=True)
auto.head()
```

Out[231]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

First find the "correct" answer of the intercept and slope

In [232]:
```
# simple linear regression with the `statsmodels` library
auto_model = smf.ols(formula='mpg ~ horsepower', data=auto)
auto_result = auto_model.fit()
print(auto_result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.606
Model:                            OLS   Adj. R-squared:                  0.605
Method:                 Least Squares   F-statistic:                     599.7
Date:                Mon, 11 Dec 2023   Prob (F-statistic):           7.03e-81
Time:                        13:24:44   Log-Likelihood:                -1178.7
No. Observations:                 392   AIC:                             2361.
Df Residuals:                     390   BIC:                             2369.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     39.9359      0.717     55.660      0.000      38.525      41.347
horsepower    -0.1578      0.006    -24.489      0.000      -0.171      -0.145
==============================================================================
Omnibus:                       16.432   Durbin-Watson:                   0.920
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               17.305
Skew:                           0.492   Prob(JB):                     0.000175
Kurtosis:                       3.299   Cond. No.                         322.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

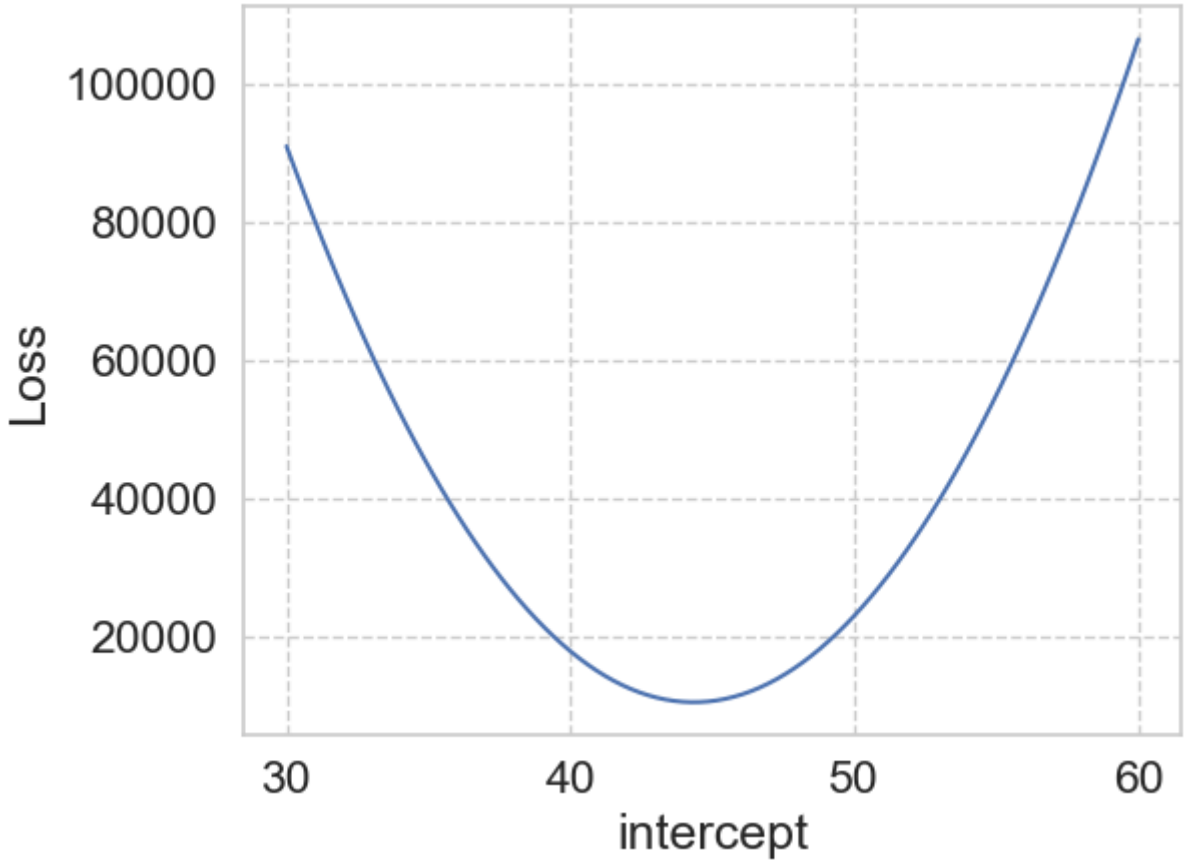
In [233]:
```python
def linear_regression_loss(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = False,
) -> float:
    """Calculates the loss of a linear regression problem.

    Args:
        X: The array (matrix) that contains the independent variables. By convention, the shape is n by
        y: The array (column vector) that contains the depedent variable. The shape is n by 1.
        betas: The array that contains the coefficients. The shape is (d+1) by 1.
        normalized: If True, the loss function is normalized by sample sizes (i.e., n). Default value: F

    Returns:
        (float): The loss function value.

    Raises:
        AssertionError: If the shape of the predicted value is different from y.
    """
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    y_pred = X @ betas
    assert y.shape == y_pred.shape, f"y has shape of {y.shape} and y_pred of {y_pred.shape}"
    loss = np.sum(np.square(y - X @ betas))

    if normalized:
        loss /= len(y)

    return loss
```

In [234]:
```
X_auto = np.vstack((np.ones(shape=len(auto)), auto["horsepower"].values.T)).T
y_auto = auto["mpg"].values

# test run
auto_betas = [50, -0.2]
linear_regression_loss(X=X_auto, y=y_auto, betas=auto_betas)
```

Out[234]:
```
22974.22
```

Next visualize the loss function by fixing the slope, but only varying the value of intercept. We should see the loss function follows a parabola shape.

In [235]:
```
# fix beta_1, i.e., slope
auto_slope = -0.2
auto_losses = []
auto_beta_0s = np.linspace(start=30, stop=60, num=100)
for b in auto_beta_0s:
    auto_betas = np.array([b, auto_slope]).reshape(-1, 1)
    loss = linear_regression_loss(X=X_auto, y=y_auto, betas=auto_betas)
    auto_losses.append(loss)

# plot the loss function
plt.figure()
sns.lineplot(x=auto_beta_0s, y=auto_losses)
plt.xlabel("intercept")
plt.ylabel("Loss")
plt.tight_layout()
```



We will define the gradient of the loss function, which also takes the dependent variables, independent variables, and the model parameters (intercpet and slope) as inputs.

In [236]:
```python
def linear_regression_loss_gradient(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = False,
) -> np.ndarray:
    """Calculates the gradient of the loss of a linear regression problem.

    Args:
        X: The array (matrix) that contains the independent variables. By convention, the shape is n by
        y: The array (column vector) that contains the depedent variable. The shape is n by 1.
        betas: The array that contains the coefficients. The shape is (d+1) by 1.
        normalized: If True, the gradient is normalized by sample sizes (i.e., n). Default value: False.

    Returns:
        (float): The value of the gradient.
    """
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    grad = -2 * (X.T @ (y - X @ betas))
    assert grad.shape == betas.shape, f"The shape of grad is {grad.shape} and betas is {betas.shape}"
    if normalized:
        grad /= len(y)

    return grad
```

Finally, let's construct the GD routine, by using the previously defined loss function, and the gradient function.

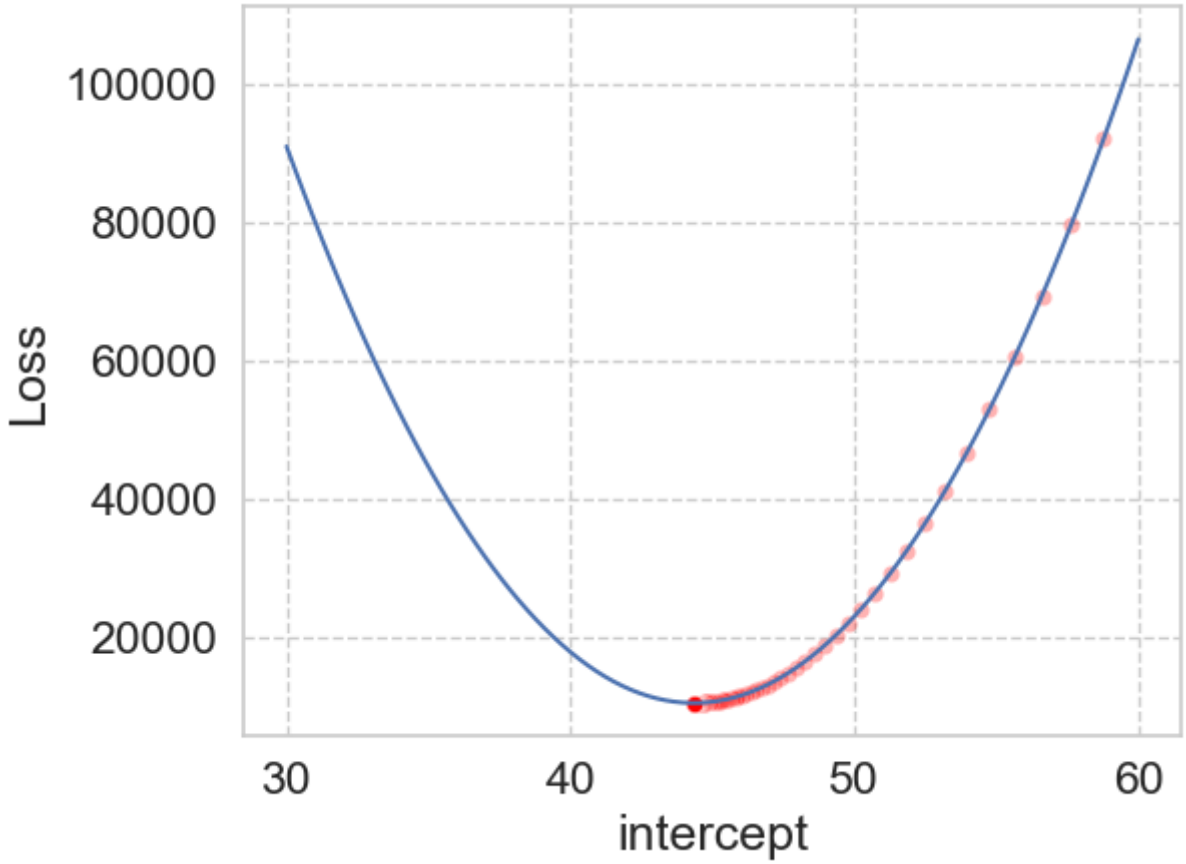
In [237]:
```
def gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    initial_guess: Union[List, np.ndarray],
    learning_rate: float,
    loss_function: Callable,
    gradient_function: Callable,
    verbose: bool = False,
    threshold: float = 1e-6,
    fix_guess: Dict = None,
) -> Tuple[List, List]:
    """Gradient descent routine.

    Args:
        X: The input data without labels.
        y: The labels.
        initial_guess: The starting point of the gradient descent.
        learning_rate: The learning rate of the gradient descent.
        loss_function: Provided loss function, that takes (X, y, parameter) as inputs.
        gradient_function: Provided gradient of the loss function, that takes (X, y, parameter) as input
        verbose: If set to True, print out intermediate results.
        threshold: Absolute value of different in loss, below which is considered converged.
        fix_guess: A dictionay to fix given dimension(s) of the parameter.

    Return:
        List: The history of parameters.
        List: The history of the losses.
    """
    guess_current = np.array(initial_guess).reshape(-1, 1)
    if fix_guess:
        for k, v in fix_guess.items():
            guess_current[k] = v

    guess_iter = [guess_current]
    losses_iter = [loss_function(X=X, y=y, betas=guess_current)]

    difference = float("inf")
    iteration_count = 0
    while abs(difference) > threshold:
        iteration_count += 1
        guess_next = guess_current - learning_rate * gradient_function(
            X=X, y=y, betas=guess_current)
        if fix_guess:
```

```
            for k, v in fix_guess.items():
                guess_next[k] = v
        guess_iter.append(guess_next)

        losses_next = loss_function(X=X, y=y, betas=guess_next)
        difference = losses_next - losses_iter[-1]
        losses_iter.append(losses_next)

        # update guess
        guess_current = guess_next
        # to print out intermediate results
        if verbose and iteration_count % 1000 == 0:
            print(guess_next, losses_next)

    # some datatype processing
    guess_iter: List[List[float]] = list(map(lambda x: list(x.flatten()), guess_iter))
    return guess_iter, losses_iter
```

In [238]:
```
auto_guess_iter, auto_losses_iter = gradient_descent(
    X=X_auto,
    y=y_auto,
    initial_guess=[60, -0.2],
    learning_rate=1e-4,
    loss_function=linear_regression_loss,
    gradient_function=linear_regression_loss_gradient,
    verbose=True,
    threshold=1e-3,
    fix_guess={1: -0.2}
)

print("The final values are:")
print(auto_guess_iter[-1])
```

```
The final values are:
[44.34358146811218, -0.2]
```

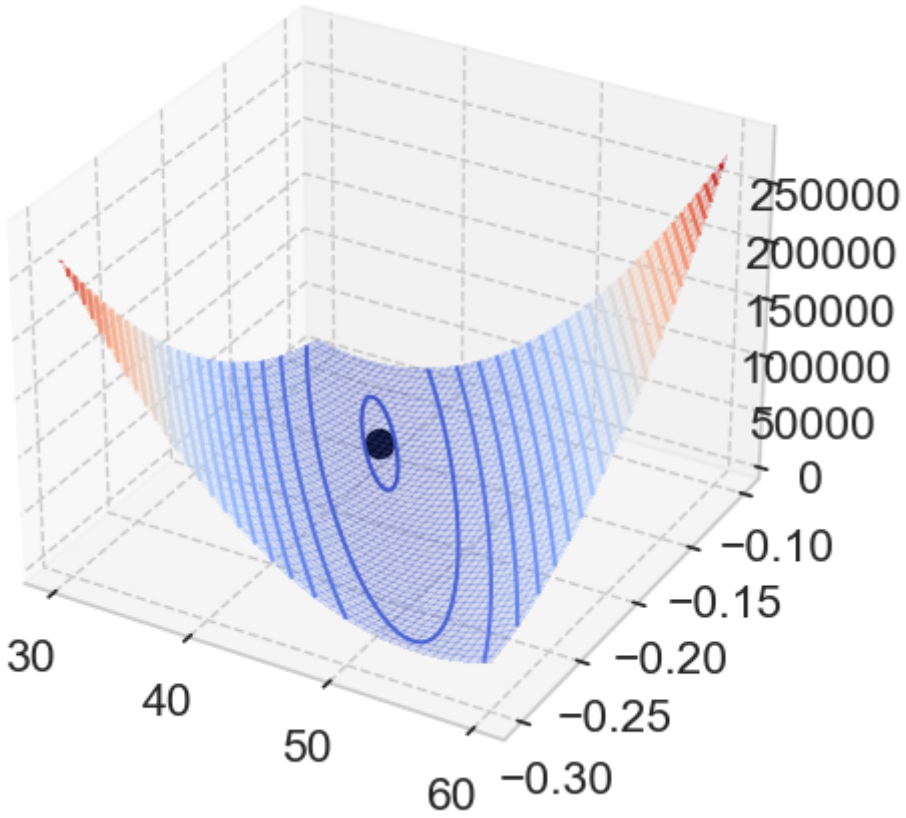
In [239]:
```
# plot the loss function
plt.figure()
sns.lineplot(x=auto_beta_0s, y=auto_losses)

# plot the iterative updates
sns.scatterplot(
    x=np.array(auto_guess_iter)[1:, 0], y=auto_losses_iter[1:], color="red", alpha=0.3)

plt.xlabel("intercept")
plt.ylabel("Loss")
plt.tight_layout()
```



In the previous example, we fixed the slope and only vary the intercept. Now let's vay both of them, as it is the "real case".

In [240]:
```
# Plot the loss function with minimal
auto_beta_1s = np.linspace(start=-0.3, stop=-0.1, num=100)
auto_X, auto_Y = np.meshgrid(auto_beta_0s, auto_beta_1s)
auto_losses_2d = np.zeros(shape=auto_X.shape)  # initialize the losses
for i, x in enumerate(auto_X):
    for j in range(len(x)):
        auto_losses_2d[i][j] = linear_regression_loss(
            X=X_auto, y=y_auto, betas=[auto_X[i][j], auto_Y[i][j]])

# make the 3d plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# plot the actual minimum
auto_betas_min = [39.9359, -0.1578]  # Assume this is known
auto_loss_min = linear_regression_loss(X=X_auto, y=y_auto, betas=auto_betas_min)
print(f"The minimum loss is {auto_loss_min:5.3f}")
ax.scatter(
    xs=[auto_betas_min[0],],
    ys=[auto_betas_min[1],],
    zs=[auto_loss_min],
    s=100,
    color="black",
)
# plot the loss function
surf = ax.plot_surface(auto_X, auto_Y, auto_losses_2d, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False, alpha=0.2)
plt.contour(auto_X, auto_Y, auto_losses_2d, levels=30, cmap=cm.coolwarm)
plt.tight_layout()
```

```
The minimum loss is 9385.926
```

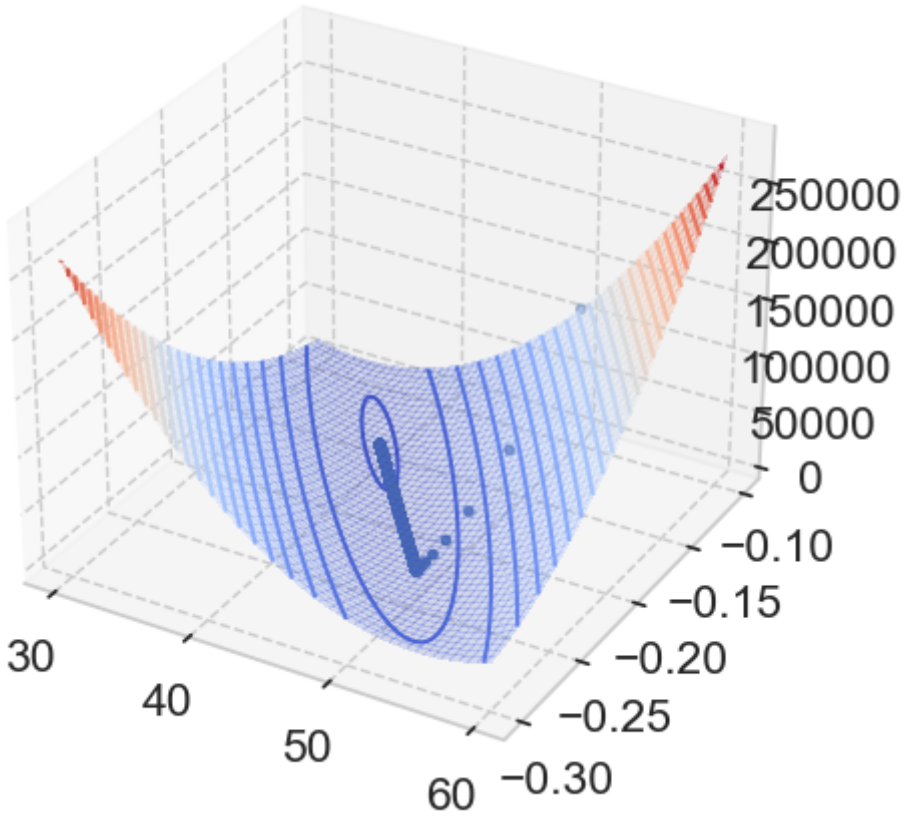

250000
200000
150000
100000
50000
0
−0.10
−0.15
−0.20
−0.25
−0.30
30
40
50
60

In [241]:
```
%%time
# gradient descent with both intercept and slope
# witht he following setting, it can take a few seconds to converge
auto_guess_iter, auto_losses_iter = gradient_descent(
    X=X_auto,
    y=y_auto,
    initial_guess=[50, -0.1],
    learning_rate=2e-7,
    loss_function=linear_regression_loss,
    gradient_function=linear_regression_loss_gradient,
    threshold=1e-5,
)

print("The final values are:")
print(auto_guess_iter[-1])
```

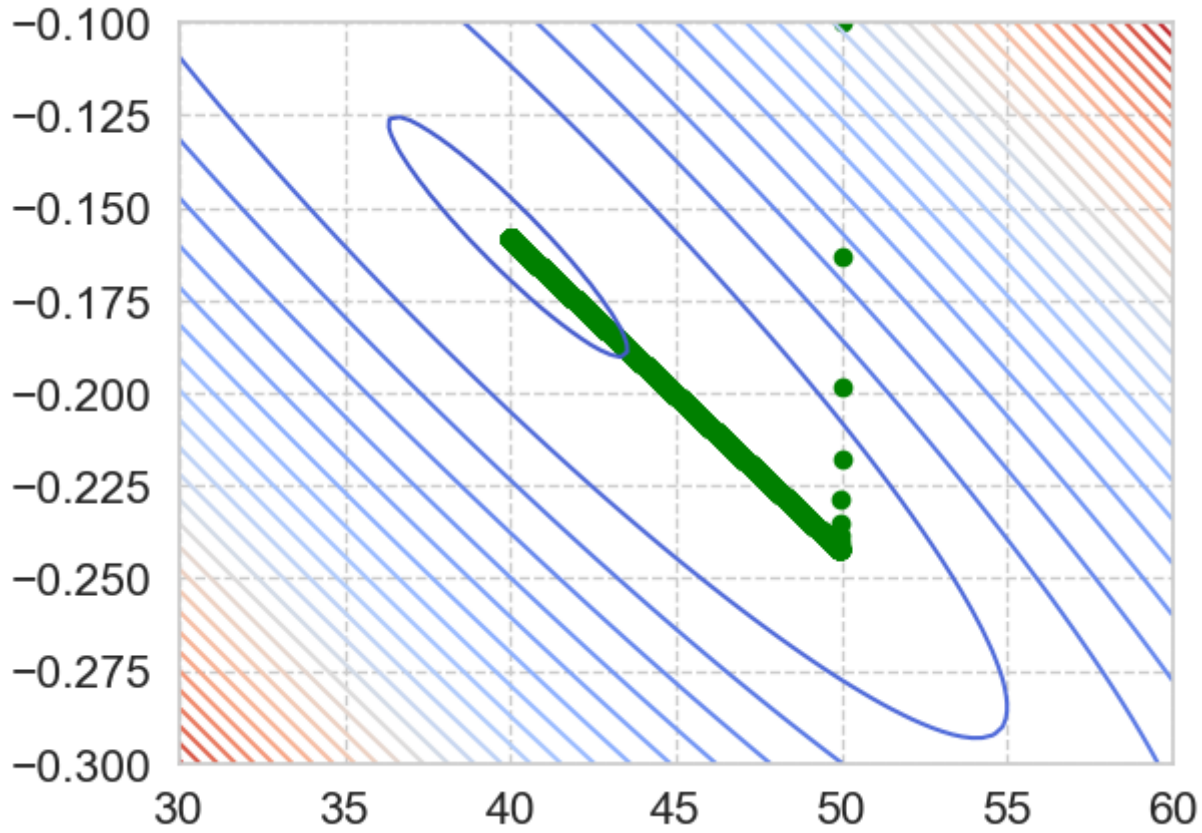
```
The final values are:
[40.01148996455557, -0.1584823393221802]
CPU times: user 2.39 s, sys: 31 ms, total: 2.42 s
Wall time: 2.42 s
```

In [242]:
```
# make the 3d plot with the gradient descent path
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# plot the loss function
surf = ax.plot_surface(auto_X, auto_Y, auto_losses_2d, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False, alpha=0.2)
plt.contour(auto_X, auto_Y, auto_losses_2d, levels=30, cmap=cm.coolwarm)

# plot the path
sample_rate = 10
ax.scatter3D(
    xs=np.array(auto_guess_iter)[:, 0][::sample_rate],
    ys=np.array(auto_guess_iter)[:, 1][::sample_rate],
    zs=auto_losses_iter[::sample_rate],
    s=10,
    cmap=cm.coolwarm,
)
plt.tight_layout()
```

```
/var/folders/3v/gxl3z6yn3fd_gv2bypr81hl80000gn/T/ipykernel_1766/2600610578.py:11: UserWarning: No data
for colormapping provided via 'c'. Parameters 'cmap' will be ignored
  ax.scatter3D(
```


250000
200000
150000
100000
50000
0
−0.10
−0.15
−0.20
−0.25
−0.30
30
40
50
60

In [243]:
```
# contour plot of the same path
plt.figure()
plt.contour(auto_X, auto_Y, auto_losses_2d, levels=30, cmap=cm.coolwarm)
plt.scatter(
    x=np.array(auto_guess_iter)[:, 0][::sample_rate],
    y=np.array(auto_guess_iter)[:, 1][::sample_rate],
    color="green",
)
plt.show()
```



In [244]:
```
# Logistic regression example
```

Logistic regression example

Here let's apply GD to the logistic regression case where the loss function is the cross-entropy, and there is no closed-form solution for the minimum.

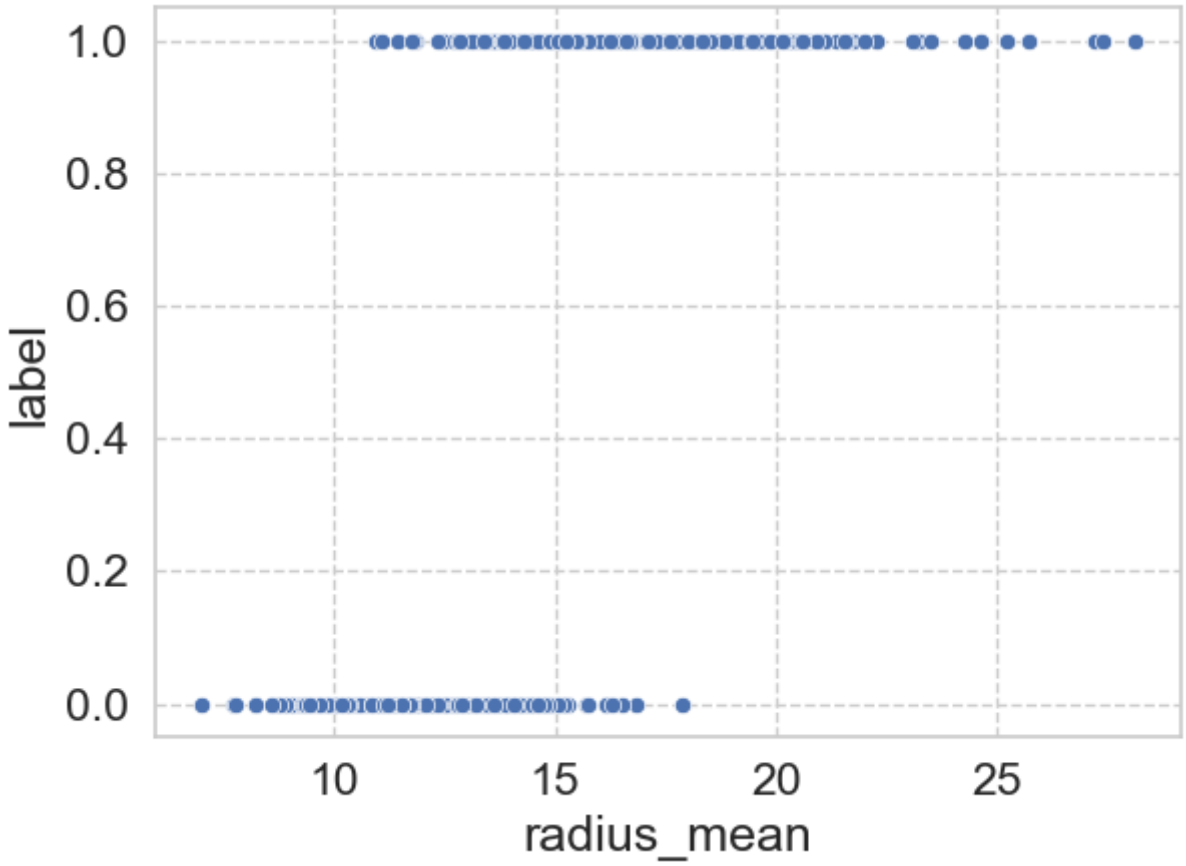
The steps will be very similar to the linear regression case: we first need to define the loss function, and the gradient, then we will apply

In [245]:
```
cancer = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/breast_cancer.
cancer_label = cancer["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
cancer.insert(2, "label", cancer_label)
cancer.head()
```

Out[245]:

	id	diagnosis	label	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_r
0	842302	M	1	17.99	10.38	122.80	1001.0	0.11840	0.27760	0
1	842517	M	1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0
2	84300903	M	1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0
3	84348301	M	1	11.42	20.38	77.58	386.1	0.14250	0.28390	0
4	84358402	M	1	20.29	14.34	135.10	1297.0	0.10030	0.13280	0

In [246]:
```
plt.figure()
sns.scatterplot(x="radius_mean", y="label", data=cancer)
plt.tight_layout()
```



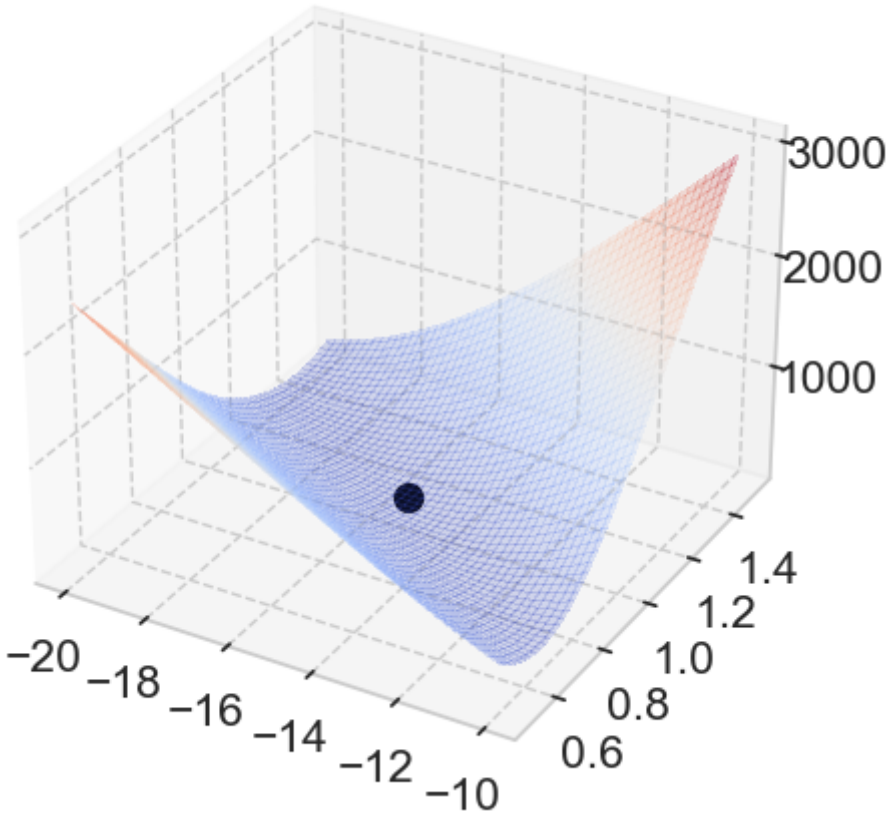
In [247]:
```
# simple logistic regression with the `statsmodels` library
cancer_model = smf.logit(formula='label ~ radius_mean', data=cancer)
cancer_result = cancer_model.fit()
print(cancer_result.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.289992
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:                  label   No. Observations:                  569
Model:                          Logit   Df Residuals:                      567
Method:                           MLE   Df Model:                            1
Date:                Mon, 11 Dec 2023   Pseudo R-squ.:                  0.5608
Time:                        13:25:04   Log-Likelihood:                -165.01
converged:                       True   LL-Null:                       -375.72
Covariance Type:            nonrobust   LLR p-value:                 1.192e-93
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -15.2459      1.325    -11.509      0.000     -17.842     -12.649
radius_mean    1.0336      0.093     11.100      0.000       0.851       1.216
==============================================================================
```

In [248]:
```
X_cancer = np.vstack((np.ones(shape=len(cancer)), cancer["radius_mean"].values.T)).T
y_cancer = cancer["label"].values
```

In [249]:
```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def logistic_regression_loss(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = False,
) -> float:
    """Calculates the loss of a logistic regression problem, i.e., cross entropy."""
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    y_pred = X @ betas
    assert y.shape == y_pred.shape, f"y has shape of {y.shape} and y_pred of {y_pred.shape}"

    # We add a small positive number inside `log()` to avoid log(0)
    loss = np.sum(y * np.log(1e-10 + sigmoid(X @ betas)) + (1 - y) * np.log(1e-10 + 1 - sigmoid(X @ beta
    if normalized:
        loss /= len(y)

    return -1. * loss
```

In [250]:
```
# Plot the loss function with minimal
cancer_beta_0s = np.linspace(start=-20, stop=-10, num=100)
cancer_beta_1s = np.linspace(start=0.5, stop=1.5, num=100)
cancer_X, cancer_Y = np.meshgrid(cancer_beta_0s, cancer_beta_1s)
cancer_losses_2d = np.zeros(shape=cancer_X.shape)  # initialize the losses
for i, x in enumerate(cancer_X):
    for j in range(len(x)):
        cancer_losses_2d[i][j] = logistic_regression_loss(
            X=X_cancer, y=y_cancer, betas=[cancer_X[i][j], cancer_Y[i][j]])

# make the 3d plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# plot the actual minimum
cancer_betas_min = [-15.2459, 1.0336]
ax.scatter(
    xs=[cancer_betas_min[0],],
    ys=[cancer_betas_min[1],],
    zs=[logistic_regression_loss(X=X_cancer, y=y_cancer, betas=cancer_betas_min)],
    s=100,
    color="black",
)
# plot the loss function
surf = ax.plot_surface(cancer_X, cancer_Y, cancer_losses_2d, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False, alpha=0.2)

plt.tight_layout()
```

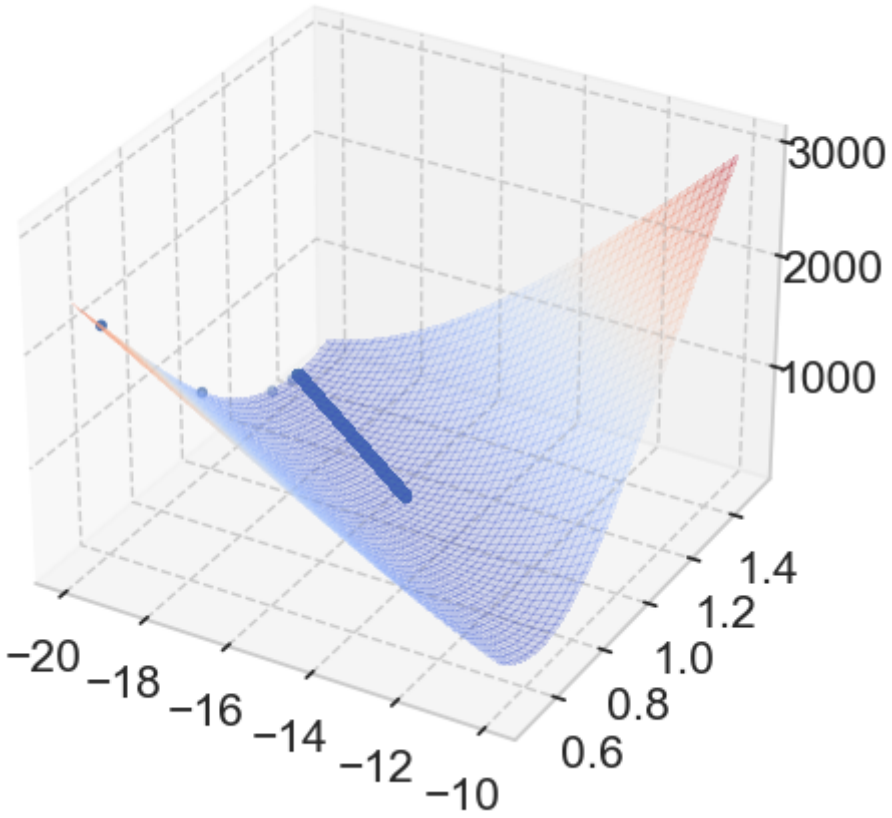

3000
2000
1000
1.4
1.2
1.0
0.8
0.6
−20
−18
−16
−14
−12
−10

In [251]:
```python
def logistic_regression_loss_gradient(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = False,
) -> float:
    """Calculate the loss of a logistic regression problem, i.e., cross entropy."""
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    grad = -1. * (X.T @ (y - sigmoid(X @ betas)))
    assert grad.shape == betas.shape, f"The shape of grad is {grad.shape} and betas is {betas.shape}"
    if normalized:
        grad /= len(y)

    return grad
```

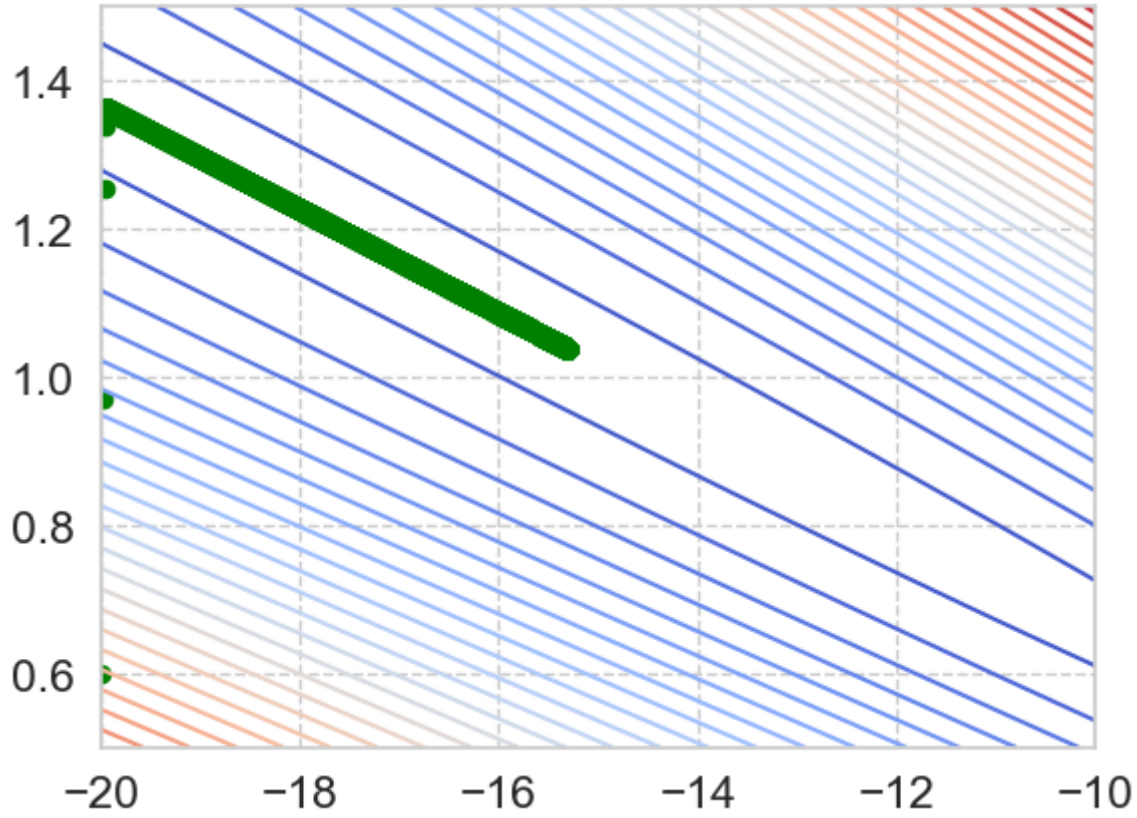
In [252]:
```python
%%time
# gradient descent for logistic regression
cancer_guess_iter, cancer_losses_iter = gradient_descent(
    X=X_cancer,
    y=y_cancer,
    initial_guess=[-20, 0.6],
    learning_rate=1e-4,
    loss_function=logistic_regression_loss,
    gradient_function=logistic_regression_loss_gradient,
    threshold=1e-7,
)

print("The final values are:")
print(cancer_guess_iter[-1])
```

```
The final values are:
[-15.301752200029057, 1.0374949359153787]
CPU times: user 2.53 s, sys: 15 ms, total: 2.54 s
Wall time: 2.54 s
```

In [253]:
```
# make the 3d plot with the gradient descent path
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

# plot the loss function
surf = ax.plot_surface(cancer_X, cancer_Y, cancer_losses_2d, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False, alpha=0.2)

# plot the path
sample_rate = 1
ax.scatter3D(
    xs=np.array(cancer_guess_iter)[:, 0][::sample_rate],
    ys=np.array(cancer_guess_iter)[:, 1][::sample_rate],
    zs=cancer_losses_iter[::sample_rate],
    s=10,
)
plt.tight_layout()
plt.show()
```

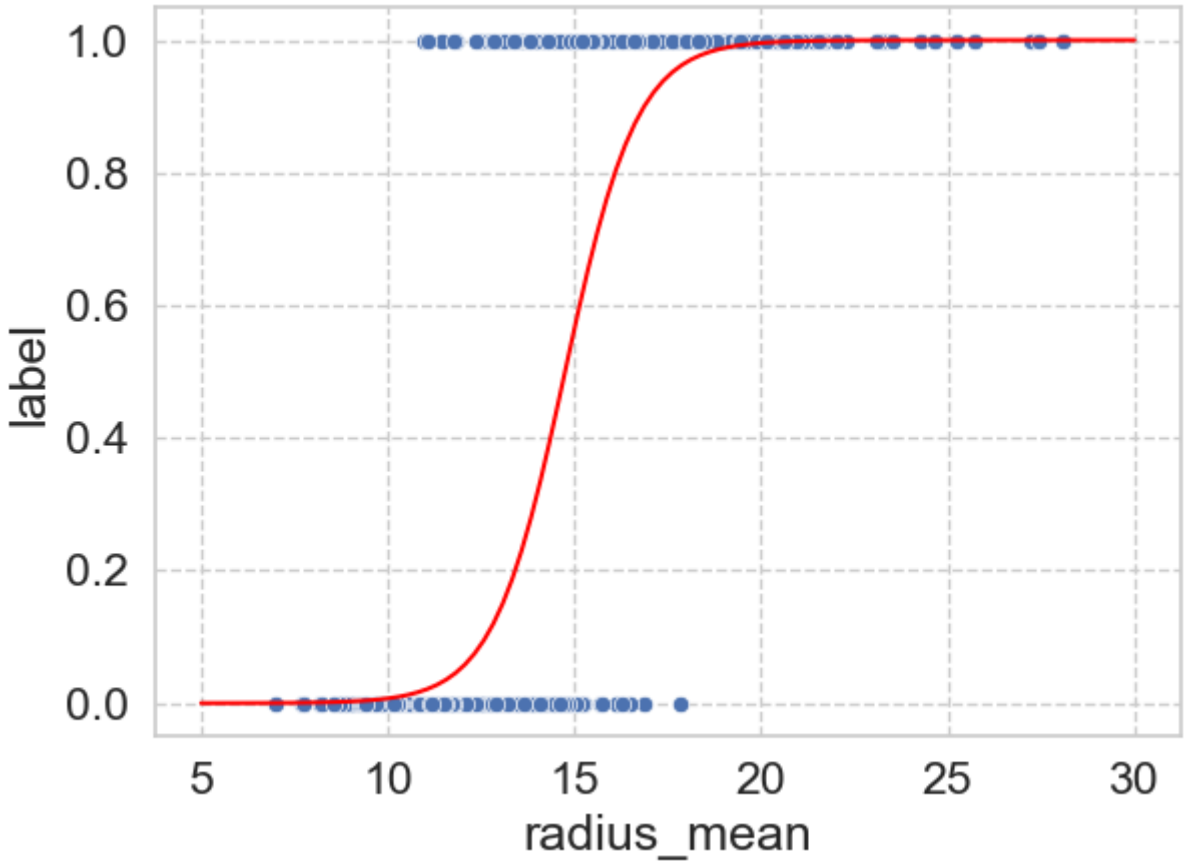

3000
2000
1000
1.4
1.2
1.0
0.8
0.6
−20
−18
−16
−14
−12
−10

In [254]:
```
# contour plot of the same path
plt.figure()
plt.contour(cancer_X, cancer_Y, cancer_losses_2d, levels=30, cmap=cm.coolwarm)
plt.scatter(
    x=np.array(cancer_guess_iter)[:, 0][::sample_rate],
    y=np.array(cancer_guess_iter)[:, 1][::sample_rate],
    color="green",
)
plt.show()
```



In [255]:
```
# plot the fitted curve
cancer_betas_fitted = np.array(cancer_guess_iter[-1])
print(cancer_betas_fitted)

plt.figure()
sns.scatterplot(x="radius_mean", y="label", data=cancer)
x = np.linspace(start=5, stop=30, num=100)
X = np.vstack((np.ones(shape=len(x)), x.T)).T
y = sigmoid(X @ cancer_betas_fitted)
sns.lineplot(x=x, y=y, color="red")
plt.tight_layout()
```

```
[-15.3017522    1.03749494]
```



In [256]:
```
# Week 7
```

Week 7 - cross_validation.ipynb, hyperparameter_tuning.ipynb, pca.ipynb

```
Introduces:
    - Cross Validation
    - Hyperparameter tuning
    - PCA
```

In [257]:
```
# Cross Validation
```

Cross validation (CV) is a statistical technique used in machine learning to assess the performance of models. It involves partitioning a dataset into multiple subsets, training the model on some of these subsets, and validating the model on the remaining subsets.

Cross-validation helps in mitigating overfitting and provides an estimate of the model's generalization performance on unseen data.

Below we will use the "Auto MPG" dataset to demonstrate this concept, whereas we fit more complex models (higher order polynomials with the same feature), and examine the corresponding RMSE (root mean squared error) evaluated on the train and validation dataset.

In [258]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from typing import Tuple
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [259]:
```
# Define the common functions
def hstack_higher_order(X: np.array, order: int = 1) -> np.array:
    """Construct a matrix with higher order terms."""
    X_raw = X.reshape(-1, 1)
    X_w_higher_orders = np.zeros(shape=(len(X_raw), order))
    for o in range(order):
        X_w_higher_orders[:, o] = np.power(X_raw, o).reshape(1, -1)
    return X_w_higher_orders

def fit_polynomial(
    X: np.array,
    y: np.array,
    order: int = 1,
) -> LinearRegression:
    """Fit a polynomial model to the underlying data."""
    X_fit = hstack_higher_order(X, order)
    # fit the model
    model = LinearRegression()
    model.fit(X_fit, y)

    return model

auto = pd.read_csv(
    "https://raw.githubusercontent.com/changyaochen/"
    "MECE4520/master/data/auto_mpg.csv")
auto.dropna(inplace=True)  # Drop the null values
print(f"The shape of the data is {auto.shape}")
auto.head()
```

```
The shape of the data is (392, 9)
```

Out[259]:

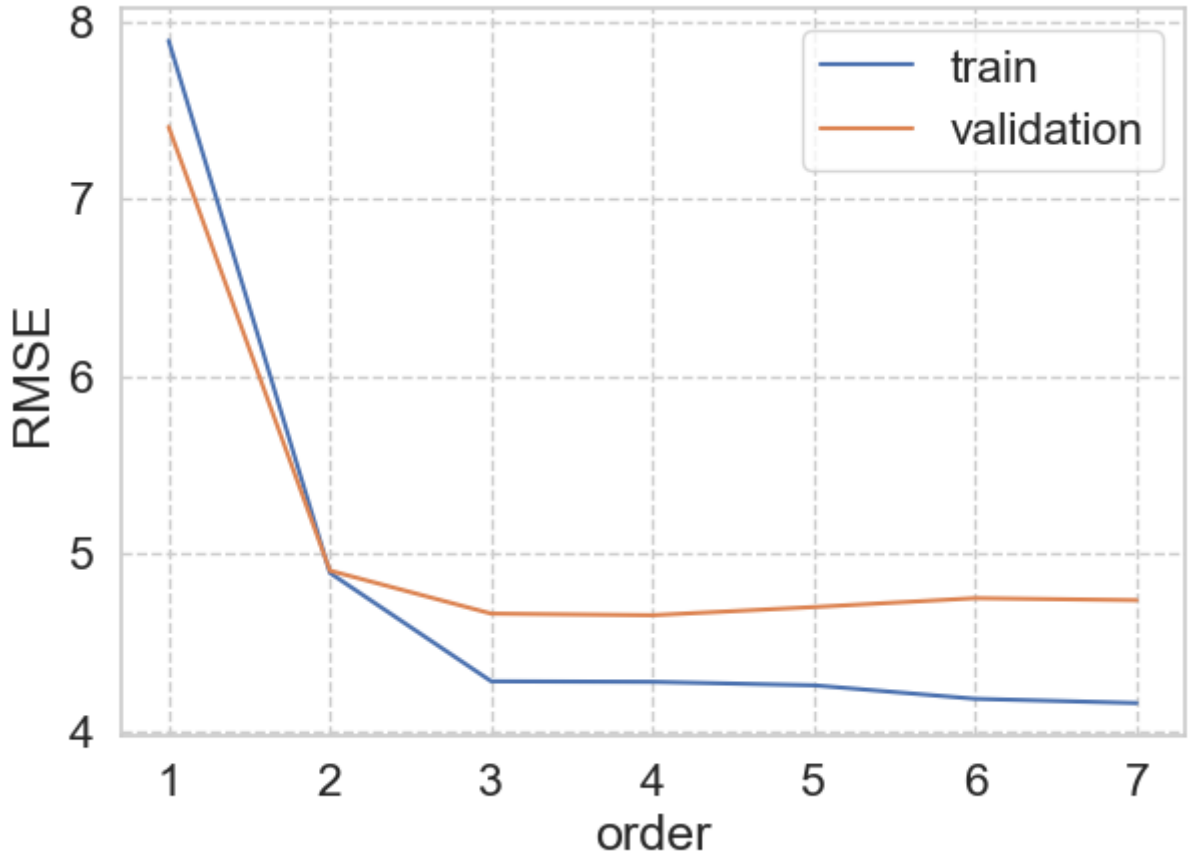
	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

In [260]:
```
# Fit models with different order polynomials
X = auto["horsepower"].values
y = auto["mpg"].values
n_trials= 10  # we will run multiple trials
max_order = 8

# CV split
# Initialize a pandas dataframe to collect the RMSEs from different trials
rmse = pd.DataFrame(columns=["trial_index", "order", "train", "validation"])
for t in range(n_trials):
    X_train, X_validation, y_train, y_validation = train_test_split(
        X, y, test_size=0.2, random_state=(103 + t)
    )

    # fit different polynomials
    for order in range(1, max_order):
        model = fit_polynomial(X=X_train, y=y_train, order=order)
        y_pred_train = model.predict(hstack_higher_order(X_train, order))
        y_pred_validation = model.predict(hstack_higher_order(X_validation, order))
        row = pd.Series({
            "trial_index": t,
            "order": order,
            "train": mean_squared_error(
                y_true=y_train, y_pred=y_pred_train, squared=False),
            "validation": mean_squared_error(
                y_true=y_validation, y_pred=y_pred_validation, squared=False),
        })
        rmse = pd.concat((rmse, row.to_frame().T), ignore_index=True)

rmse.head()
```

Out[260]:

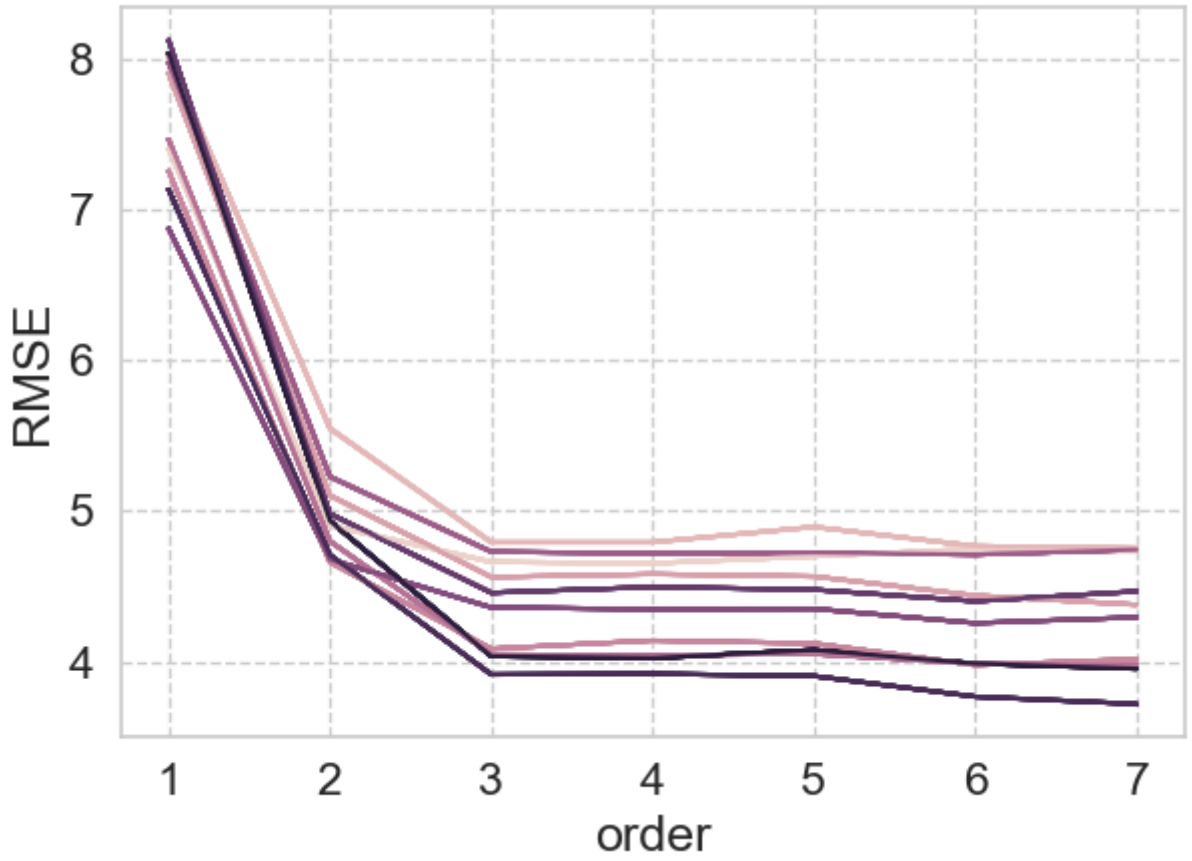
	trial_index	order	train	validation
0	0.0	1.0	7.894118	7.407799
1	0.0	2.0	4.891589	4.904329
2	0.0	3.0	4.279975	4.661947
3	0.0	4.0	4.276973	4.652892
4	0.0	5.0	4.257462	4.698976

In [261]:
```
# Plot the RMSE on train and validation dataset
trial_index = 0

plt.figure()
sns.lineplot(x="order", y="train", data=rmse[rmse["trial_index"] == trial_index], label="train")
sns.lineplot(x="order", y="validation", data=rmse[rmse["trial_index"] == trial_index], label="validatior

plt.gca().set_ylabel("RMSE")
plt.tight_layout()
plt.show()
```



The train-validation split is essentially random, therefore, the error metrics (RMSE) evaluated on them are random variables. Here we want to demonstrate the variance of the validation metrics from different train-validation splits.

In [262]:
```
plt.figure()
for t in range(n_trials):
    sns.lineplot(x="order", y="validation", data=rmse, hue="trial_index")

plt.gca().set_ylabel("RMSE")
plt.gca().get_legend().remove()
plt.tight_layout()
plt.show()
```



In [263]:
```
# Hyperparameter Tuning
```

Hyperparameter tuning

Hyperparameters are parameters whose values are set before the learning process begins. Unlike the parameters (like coefficients in linear regression, weights in neural networks) which are learned during training, hyperparameters are not learned from the data. They are external to the model and their values can be considered as settings for the learning algorithm. They directly influence the behavior of the training algorithm and have a significant impact on the performance of the model.

The purpose of hyperparameter tuning is to search for the optimal hyperparameters that yield the best performance, typically measured in

In [264]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, Normalizer
from sklearn.pipeline import make_pipeline

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In the following example, we will use the Auto MPG data to train a linear regression with, with L2 regularization. In such a case, the strength of the L2 regularization (), is the hyperparameter.

For each value of , we will train a regression model, with 4 features, to predict the MPG. For each training, we will do a simple train-validation split, and track the RMSE on both train and validation datasets.

We will then plot these RMSE for all α values used.

In [265]:
```
# use the auto data as example
auto = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/auto_mpg.csv")
auto.dropna(inplace=True)  # for the sake of simplicity
auto.head()
```

Out[265]:

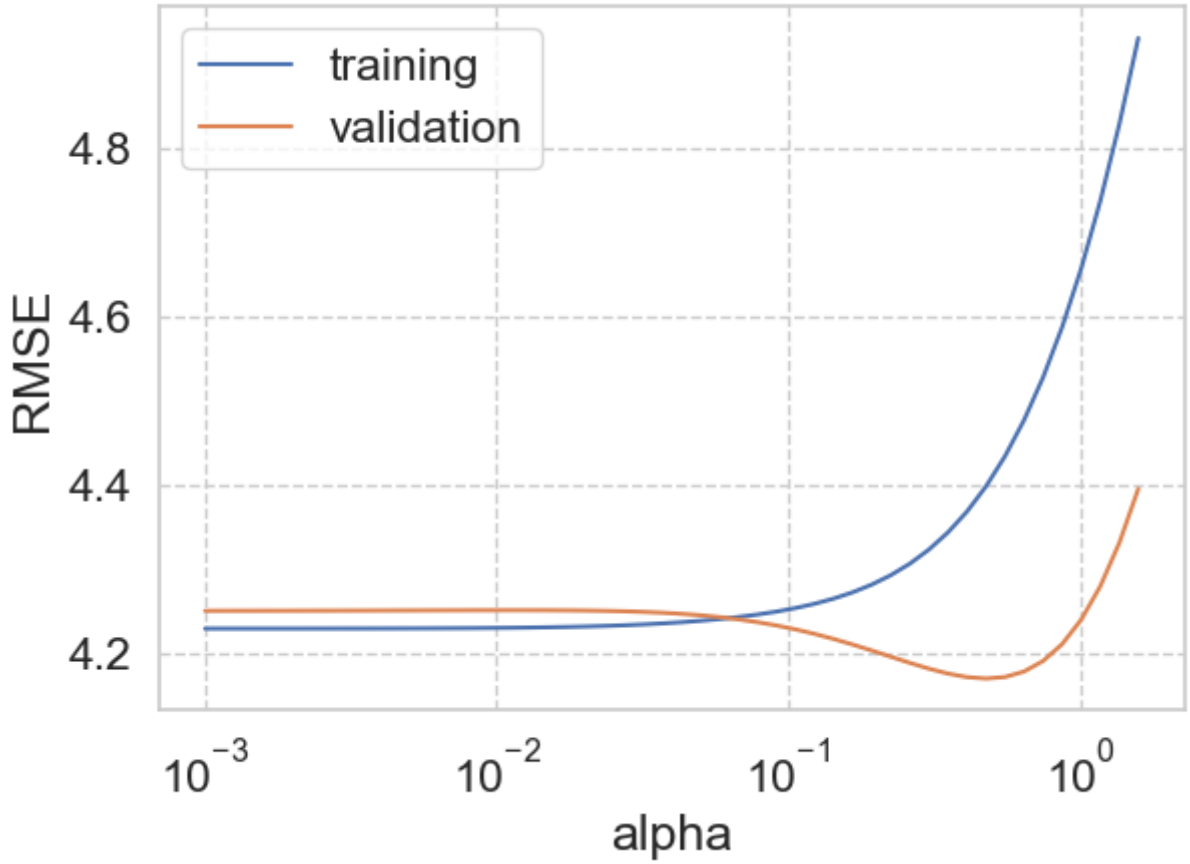
	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

In [266]:
```
numerical_features = [
    "displacement",
    "horsepower",
    "weight",
    "acceleration",
]

X_train, X_validation, y_train, y_validation = train_test_split(
    auto[numerical_features],
    auto["mpg"],
    test_size=0.2,
    random_state=42,
)

all_models = []
rmse = pd.DataFrame(columns=["alpha", "train", "validation"])

for alpha in np.logspace(start=-3, stop=.2, num=50):
    model = make_pipeline(StandardScaler(with_mean=False), Ridge(alpha=alpha * len(X_train)))
    model.fit(X=X_train, y=y_train)
    all_models.append(model)

    y_pred_train = model.predict(X_train)
    y_pred_validation = model.predict(X_validation)
    row = pd.Series({
        "alpha": alpha,
        "train": np.sqrt(mean_squared_error(y_true=y_train, y_pred=y_pred_train)),
        "validation": np.sqrt(mean_squared_error(y_true=y_validation, y_pred=y_pred_validation)),
    })
    rmse = pd.concat((rmse, row.to_frame().T), ignore_index=True)
rmse.tail()
```

Out[266]:

	alpha	train	validation
45	0.868511	4.587953	4.210900
46	1.009443	4.657502	4.239972
47	1.173242	4.737644	4.279573
48	1.363622	4.828979	4.331073
49	1.584893	4.931827	4.395570

In [267]:
```
plt.figure()
sns.lineplot(x="alpha", y="train", data=rmse, label="training")
sns.lineplot(x="alpha", y="validation", data=rmse, label="validation")

plt.gca().set_ylabel("RMSE")
plt.gca().set_xscale("log")
plt.tight_layout()
```



In [268]:
```
# PCA
```

Principal Component Analysis

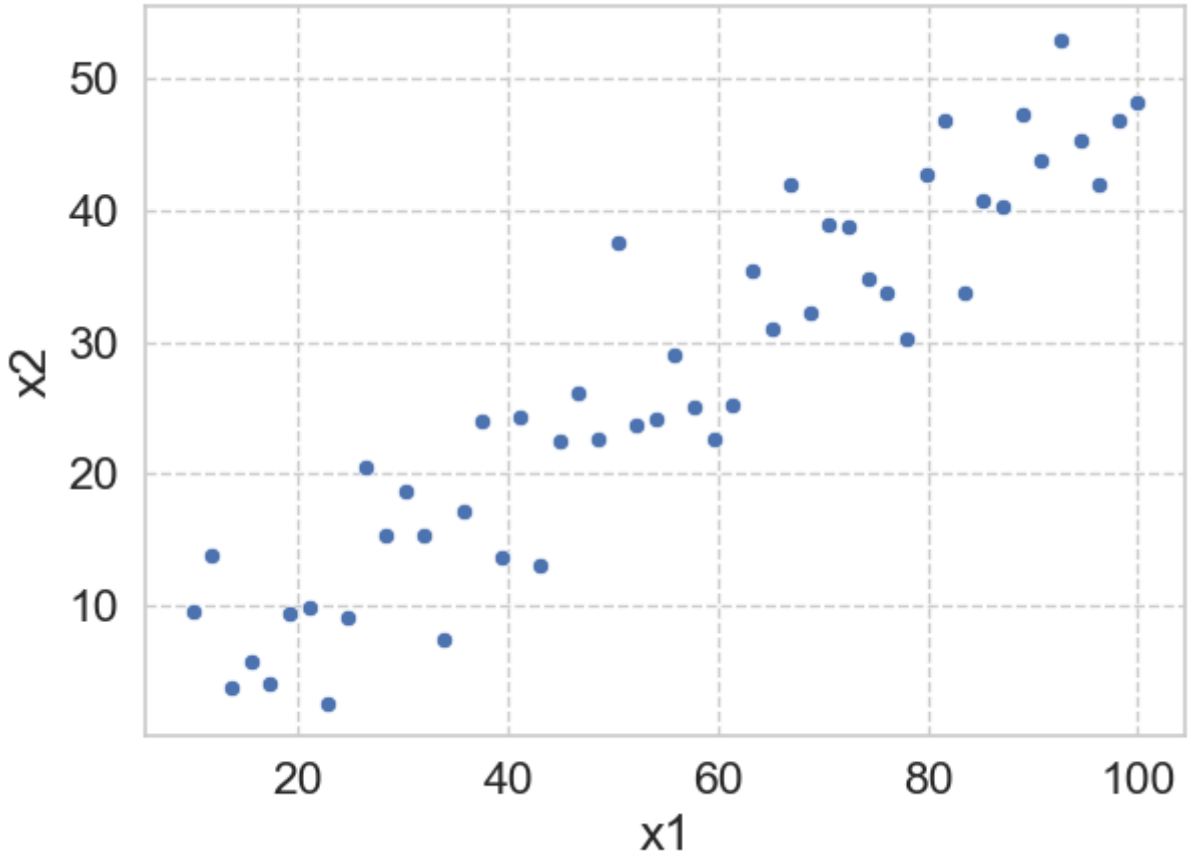
PCA, or Principal Component Analysis, is a dimensionality reduction technique used in the fields of machine learning and statistics. It's especially popular for visualization, noise filtering, feature extraction, and stock market predictions, among other applications.

The main idea behind PCA is to transform the original variables into a new set of variables, the principal components, which are orthogonal (perpendicular) to each other, and which reflect the maximum variance in the data.

In [269]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

Below we will show a 2D data, where x_1 and x_2 are highly correlated. This is a perfect use case for PCA, where we can project the 2D data onto a 1D plane (line) without losing too much information.

In [270]:
```
x1 = np.linspace(start=10, stop=100, num=50)
x2 = 0.5 * x1 + np.random.normal(scale=5, size=len(x1))

plt.figure()

sns.scatterplot(x=x1, y=x2)

plt.gca().set_xlabel("x1")
plt.gca().set_ylabel("x2")
plt.tight_layout()
```



Below we will show the process of PCA with the Auto MPG data, and compute the reconstruction error.

In [271]:
```
# use the auto data as the example
auto = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/auto_mpg.csv")
auto.dropna(inplace=True)  # for the sake of simplicity
auto.head()
```

Out[271]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model	origin	car
0	18.0	8.0	307.0	130.0	3504.0	12.0	70.0	1.0	chevrolet chevelle malibu
1	15.0	8.0	350.0	165.0	3693.0	11.5	70.0	1.0	buick skylark 320
2	18.0	8.0	318.0	150.0	3436.0	11.0	70.0	1.0	plymouth satellite
3	16.0	8.0	304.0	150.0	3433.0	12.0	70.0	1.0	amc rebel sst
4	17.0	8.0	302.0	140.0	3449.0	10.5	70.0	1.0	ford torino

In [272]:
```
# scale the input
from sklearn.preprocessing import StandardScaler

X = auto[["displacement", "horsepower", "weight", "acceleration"]].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled
```

Out[272]:
```
array([[ 1.07728956,  0.66413273,  0.62054034, -1.285258  ],
       [ 1.48873169,  1.57459447,  0.84333403, -1.46672362],
       [ 1.1825422 ,  1.18439658,  0.54038176, -1.64818924],
       ...,
       [-0.56847897, -0.53247413, -0.80463202, -1.4304305 ],
       [-0.7120053 , -0.66254009, -0.41562716,  1.11008813],
       [-0.72157372, -0.58450051, -0.30364091,  1.40043312]])
```

In [273]:
```
# perform SVD
n = len(X_scaled)
U, S, Vh = np.linalg.svd(X_scaled.T @ X_scaled / n)
```

In [274]:
```
# calculate reconstruction error
all_errors = []
for k in range(1, len(U) + 1):
    U_reduced = U[:, :k]
    Z = X_scaled @ U_reduced
    X_approx = Z @ U_reduced.T
    error = (
        np.sum(np.square(np.linalg.norm((X_scaled - X_approx), ord=2, axis=1)))
        / np.sum(np.square(np.linalg.norm(X_scaled, ord=2, axis=1)))
    )
    all_errors.append(error)

all_errors
```

Out[274]:
```
[0.19776570529397033,
 0.03425025083897679,
 0.013511965150122119,
 8.771568173220479e-32]
```

In [275]:
```
# A different way to calcuate the error
for i in range(len(S)):
    print(1 - np.sum(S[:(i + 1)])/ np.sum(S))
```

```
0.1977657052939703
0.034250250838976704
0.013511965150122096
0.0
```

In [276]:
```
# Week 8
```

Week 8

Introduces:
- Bootstrapping
- Decision Trees

In [277]:
```
# Bootstrap
```

Bootstrap is a resampling technique used to estimate the distribution of a statistic (like the mean or variance) by repeatedly resampling with replacement from the data set. It's a powerful tool used in inferential statistics and is particularly useful when the sample size is small or when the underlying distribution of the data is unknown or complex.

In [278]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.linear_model import LogisticRegression
from tqdm import tqdm
from urllib.error import URLError

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [279]:
```
# Binary classification example
```

Here we will apply the bootstrap method to estimate the uncertainty of the coefficient in a logistic regression model.

In [280]:
```
try:
    cancer = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/breast_car
except URLError:
    cancer = pd.read_csv("../lecture_4/breast_cancer_data.csv")

cancer["label"] = cancer["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
cancer.head()
```

Out[280]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

In [281]:
```
# fit a logistic regression model
model = smf.glm(
    formula='label ~ radius_mean',
    data=cancer,
    family=sm.families.Binomial(),
)
result = model.fit()
print(result.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                   label   No. Observations:                  569
Model:                             GLM   Df Residuals:                      567
Model Family:                Binomial   Df Model:                            1
Link Function:                   Logit   Scale:                          1.0000
Method:                           IRLS   Log-Likelihood:                -165.01
Date:                Mon, 11 Dec 2023   Deviance:                       330.01
Time:                        13:25:51   Pearson chi2:                     489.
No. Iterations:                     7   Pseudo R-squ. (CS):             0.5232
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -15.2459      1.325    -11.509      0.000     -17.842     -12.649
radius_mean    1.0336      0.093     11.100      0.000       0.851       1.216
==============================================================================
```

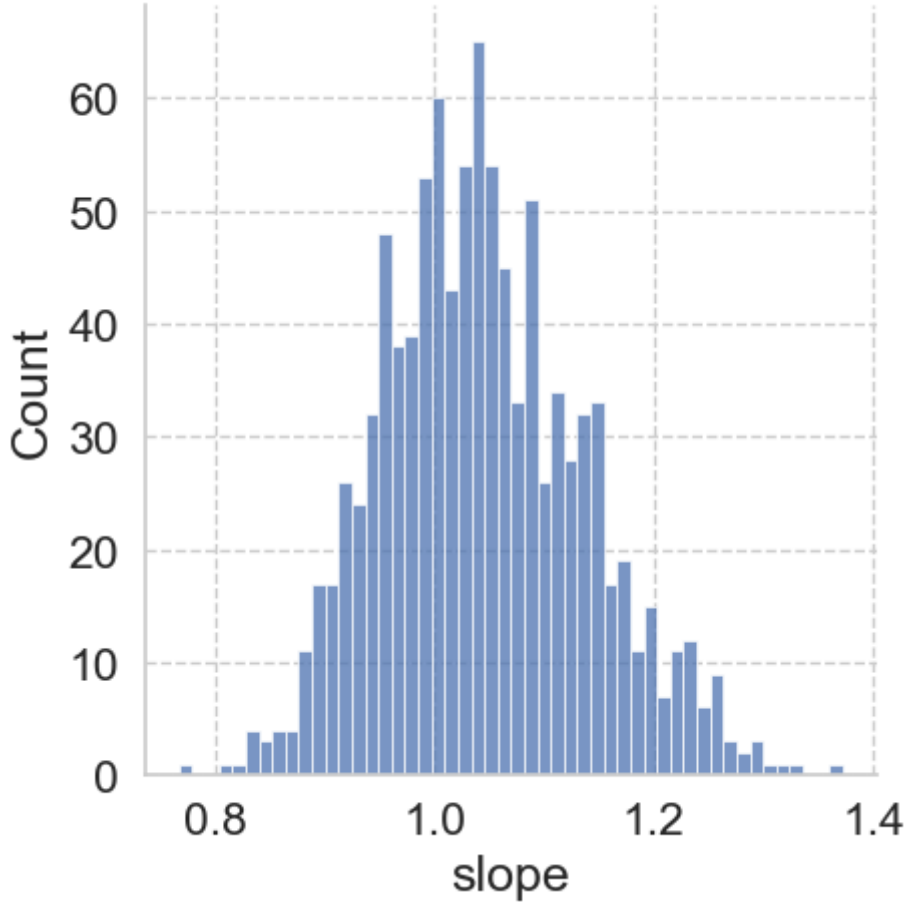
In [282]:

```
# bootstrap
np.random.seed(42)

B = 1000  # number of bootstrap
beta_1s = []
for _ in tqdm(range(B)):
    # bootstrap the indices and build the bootstrap data
    idx = np.random.randint(low=0, high=len(cancer), size=len(cancer))
    data_bootstrap = cancer.iloc[idx]

    # fit the model
    model_bootstrap = LogisticRegression(penalty="none", random_state=42)
    model_bootstrap.fit(X=data_bootstrap[["radius_mean"]], y=data_bootstrap["label"])
    beta_1s.append(model_bootstrap.coef_.flatten()[0])
```

```
  0%|                                                   | 0/1000 [00:00<?, ?it/s]/Users/nicolinoprimave
ra/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:1182: FutureWarning: `pena
lty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the past behaviour, set `pen
alty=None`.
  warnings.warn(
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:118
2: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the p
ast behaviour, set `penalty=None`.
  warnings.warn(
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:118
2: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the p
ast behaviour, set `penalty=None`.
  warnings.warn(
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:118
2: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the p
ast behaviour, set `penalty=None`.
  warnings.warn(
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/linear_model/_logistic.py:118
2: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4. To keep the p
```

In [283]:
```
print(f"The mean of beta_1 is {np.mean(beta_1s):5.3f}.")
print(f"The standard error of beta_1 is {np.std(beta_1s, ddof=1):5.3f}.")

plt.figure()
sns.displot(beta_1s, bins=50)
plt.xlabel("slope")
plt.tight_layout()
plt.show()
```

```
The mean of beta_1 is 1.044.
The standard error of beta_1 is 0.093.

<Figure size 640x480 with 0 Axes>
```



In [284]:
```
# Decision Tree
```

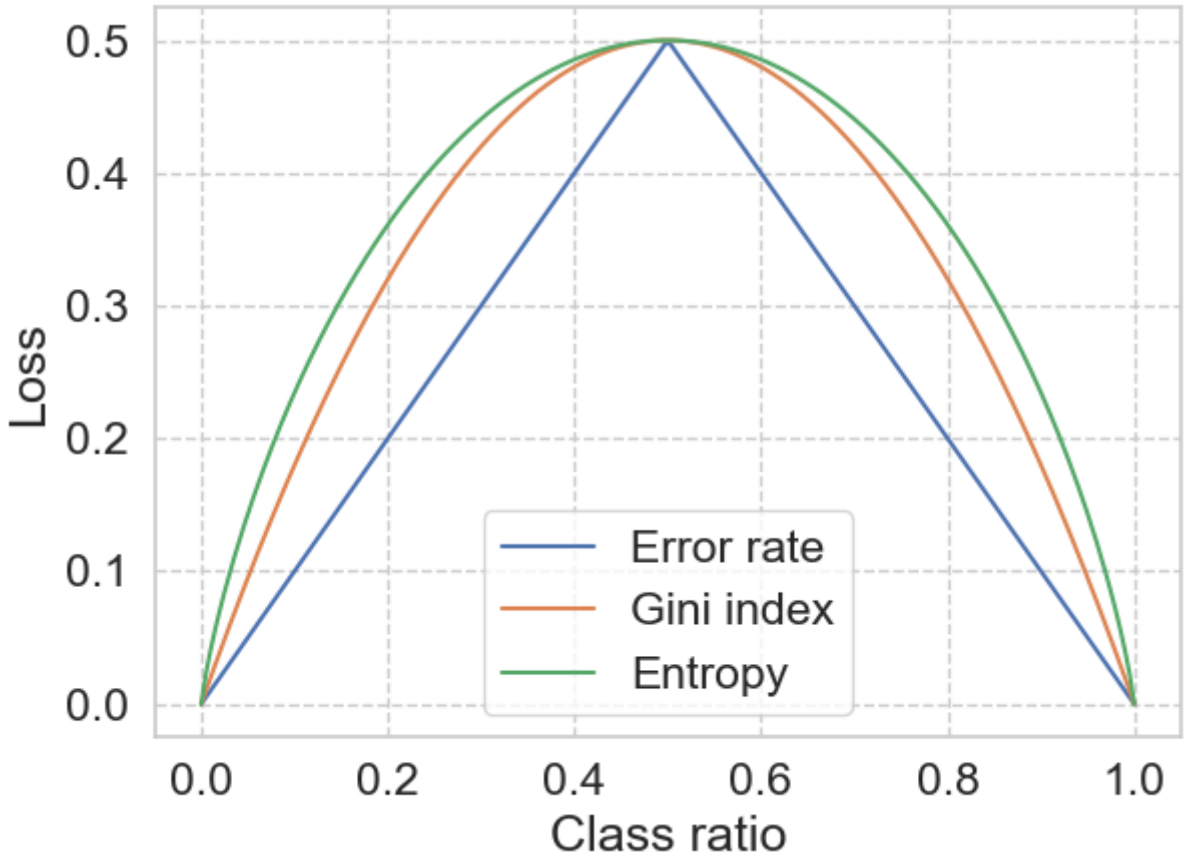
A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g., whether a car's horsepower is larger than 200), each branch represents the outcome of the test, and each leaf node represents an outcome (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

In [285]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [286]:
```
# Loss functions for classification
```

Decision tree algorithms utilize criteria or measures (loss function) to decide where to split the data.

In [287]:
```
def error_rate(p: float):
    """Classification error rate for a given binary class ratio"""
    return p if p < 0.5 else 1 - p

def gini_index(p: float):
    """Gini index for a given binary class ratio."""
    return 2 * p * (1 - p)

def entropy(p: float):
    """Entropy for a given binary class ratio, normalized by ln(2)"""
    return -(p * np.log(p + 1e-16) + (1 - p) * np.log(1 - p + 1e-16)) / (np.log(2) / 0.5)

p = np.linspace(start=0, stop=1, num=200)
plt.figure()
sns.lineplot(x=p, y=[error_rate(x) for x in p], label="Error rate")
sns.lineplot(x=p, y=[gini_index(x) for x in p], label="Gini index")
sns.lineplot(x=p, y=[entropy(x) for x in p], label="Entropy")

plt.ylabel("Loss")
plt.xlabel("Class ratio")
plt.tight_layout()
plt.show()
```



In [288]:
```
# Binary classification example
```

In the following, let's use the breast cancer example to demonstrate the appication of a decision tree.

In [289]:
```
cancer = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/breast_cancer.
cancer["label"] = cancer["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
cancer.head()
```

Out[289]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

In [290]:
```
from sklearn import tree

dt_model = tree.DecisionTreeClassifier(
    criterion="gini",
    max_depth=3,
)
features = [
    "radius_mean",
    "texture_mean",
    "perimeter_mean",
    "area_mean",
    "smoothness_mean",
    "compactness_mean",
    "concavity_mean",
    "concave_mean",
    "symmetry_mean",
    "fractal_mean",
    "radius_se",
    "texture_se",
    "perimeter_se",
    "area_se",
    "smoothness_se",
    "compactness_se",
    "concavity_se",
    "concave_se",
    "symmetry_se",
    "fractal_se",
    "radius_extreme",
    "texture_extreme",
    "perimeter_extreme",
    "area_extreme",
    "smoothness_extreme",
    "compactness_extreme",
    "concavity_extreme",
    "concave_extreme",
    "symmetry_extreme",
    "fractal_extreme",
]
label = "label"
```

```
dt_model.fit(X=cancer[features], y=cancer[label])
```

Out[290]:
```
DecisionTreeClassifier(max_depth=3)
```
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [291]:
```
import graphviz

dot_data = tree.export_graphviz(
    decision_tree=dt_model,
    out_file=None,
    feature_names=features,
    class_names=["0", "1"],
    filled=True,
    rounded=True,
    special_characters=True,
    max_depth=2,
)
graph = graphviz.Source(dot_data)
graph.render("cancer_tree")
graph
```

Out[291]:
```
<graphviz.sources.Source at 0x28ebf2650>
```

In [292]:
```
# Week 9
```

Week 9 - neural_network.ipynb

```
    Introduces:
        - Neural Networks
```

In [293]:
```
# Neural Network
```

A neural network (NN) is a computational model inspired by the structure and function of the human brain. It consists of interconnected nodes or "neurons" organized in layers: input, hidden, and output. Each neuron processes information based on its inputs and weights, passing the result to the next layer.

In [294]:
```
from typing import Tuple

import numpy as np
import pandas as pd

from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [295]:
```
data = pd.read_csv(
    "https://raw.githubusercontent.com/changyaochen/MECE4520/master/"
    "data/breast_cancer.csv")
data["label"] = data["diagnosis"].apply(lambda x: 0 if x == "B" else 1)
data.head()
```

Out[295]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

In [296]:
```
# Construction of a 2-layer NN
```

In this example, we will demonstrate a simple 2-layer NN, with the breast cancer dataset.

In [297]:
```
features = [
    "radius_mean",
    "texture_mean",
    "perimeter_mean",
    "area_mean",
    "smoothness_mean",
    "compactness_mean",
    "concavity_mean",
    "concave_mean",
    "symmetry_mean",
    "fractal_mean",
]
label = "label"

# train test split
X_raw, X_raw_test, Y, Y_test = train_test_split(
    data[features].values, data[label].values, test_size=0.2, random_state=42
)

# Standardize the input
scaler = StandardScaler()
scaler.fit(X_raw)
X = scaler.transform(X_raw)
X_test = scaler.transform(X_raw_test)

# formatting
Y = Y.reshape((-1, 1))
Y_test = Y_test.reshape((-1, 1))
```

In [298]:
```
# Forward propagation
```

Forward propagation in a NN refers to the process of passing input data through the network to generate an output.

Here we will demonstrate forward propgation in our example NN, using the matrix representation, and the notation introduced in the lecture.

In [299]:
```
# forward pass for a simple 2-layer NN, with 3 hidden units
np.random.seed(10)

def sigmoid(x):
    """Calculates sigmoid function."""
    return 1. / (1 + np.exp(-x))

# parameters for the first layer
W_1 = np.random.normal(size=(3, X.shape[1]))
print(f"Shape of W_1 is {W_1.shape}")

b_1 = np.random.normal(size=(3, 1))
print(f"Shape of b_1 is {b_1.shape}")

# parameters for the second layer
W_2 = np.random.normal(size=(1, 3))
print(f"Shape of W_2 is {W_2.shape}")

b_2 = np.random.normal(size=(1, 1))
print(f"Shape of b_1 is {b_2.shape}")

# calculate the forward propagation
Z_1 = X @ W_1.T
print(f"\nShape of Z_1 is {Z_1.shape}")
print("Samples for Z_1:")
print(Z_1[:5])

A_1 = sigmoid(Z_1 + b_1.T)
print(f"Shape of A_1 is {A_1.shape}")
print("Samples for A_1:")
print(A_1[:5])

Z_2 = A_1 @ W_2.T
print(f"\nShape of Z_2 is {Z_2.shape}")
print("Samples for Z_2:")
print(Z_1[:5])

A_2 = Y_hat = sigmoid(Z_2 + b_2.T)
print(f"Shape of A_2 is {A_2.shape}")
print("Samples for A_2:")
```

```
print(A_2[:5])
```

```
Shape of W_1 is (3, 10)
Shape of b_1 is (3, 1)
Shape of W_2 is (1, 3)
Shape of b_1 is (1, 1)

Shape of Z_1 is (455, 3)
Samples for Z_1:
[[ 0.16410112 -4.76306361  3.93309998]
 [-0.46604358  4.1992739   9.5658238 ]
 [-1.60754809 -0.23753874 -1.01727238]
 [ 1.37695245  2.28649564 -5.09016965]
 [ 0.12721277  3.49293739  0.32441791]]
Shape of A_1 is (455, 3)
Samples for A_1:
[[0.47421887 0.00490603 0.98314001]
 [0.32445766 0.97466643 0.99993863]
 [0.13297977 0.31284592 0.29223288]
 [0.75206111 0.85032936 0.00698167]
 [0.46503108 0.94996148 0.61233221]]

Shape of Z_2 is (455, 1)
Samples for Z_2:
[[ 0.16410112 -4.76306361  3.93309998]
 [-0.46604358  4.1992739   9.5658238 ]
 [-1.60754809 -0.23753874 -1.01727238]
 [ 1.37695245  2.28649564 -5.09016965]
 [ 0.12721277  3.49293739  0.32441791]]
Shape of A_2 is (455, 1)
Samples for A_2:
[[0.59207723]
 [0.84761911]
 [0.69066552]
 [0.76062638]
 [0.82363926]]
```

In [300]:
```
# Training a NN with backward propagation
```

Backward propagation, commonly known as backpropagation, is a fundamental process in training neural networks. It refers to the method of adjusting the weights and biases of a neural network by propagating the error backward from the output layer to the input layer.

Here we will demonstrate the backward propagation using the matrix notation, on the example NN.

In [301]:
```python
def forward_prop(
    X: np.array,
    W_1: np.array,
    b_1: np.array,
    W_2: np.array,
    b_2: np.array,
) -> Tuple:
    """Performs the forward propagation of the given NN."""
    # Note the NN structure is passed in from outside.
    Z_1 = X @ W_1.T
    A_1 = sigmoid(Z_1 + b_1.T)

    Z_2 = A_1 @ W_2.T
    A_2 = Y = sigmoid(Z_2 + b_2.T)

    return A_2, Z_2, A_1, Z_1

Y_hat, _, _, _ = forward_prop(X=X, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2)
```

In [302]:
```
def derivatives_by_backprop(
    X: np.array,
    Y: np.array,
    W_1: np.array,
    b_1: np.array,
    W_2: np.array,
    b_2: np.array,
) -> Tuple:
    """Calculates the derivatives of the parameters by backforward propagation.

    Here we assume it is a binary classification problem, with sigmoid activation functions.
    """
    # forward propagation
    dW_2, db_2, dW_1, db_1 = 0, 0, 0, 0
    Y_hat, Z_2, A_1, Z_1 = forward_prop(X=X, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2)
    n = len(Y_hat)

    loss = -np.mean(np.multiply(Y, np.log(Y_hat)) + np.multiply(1 - Y, np.log(1 - Y_hat)))

    dZ_2 = Y_hat - Y
    dW_2 = dZ_2.T @ A_1 / n
    db_2 = np.mean(dZ_2.T, axis=1, keepdims=True)

    dZ_1 = np.multiply(dZ_2 @ W_2, np.multiply(A_1, 1 - A_1))
    dW_1 = (dZ_1.T @ X) / n
    db_1 = np.mean(dZ_1.T, axis=1, keepdims=True)

    return dW_2, db_2, dW_1, db_1, loss

dW_2, db_2, dW_1, db_1, loss = derivatives_by_backprop(X=X, Y=Y, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2)
```

In [303]:
```
def gradient_descent(
    X: np.array,
    Y: np.array,
    W_1_init: np.array,
    b_1_init: np.array,
    W_2_init: np.array,
    b_2_init: np.array,
    learning_rate: float = 0.01,
    epsilon: float = 1e-6,
    verbose: bool = False,
) -> Tuple:
    """Runs gradient descent to fit the NN via backprop."""
    W_1 = W_1_init
    b_1 = b_1_init
    W_2 = W_2_init
    b_2 = b_2_init
    losses = [float("inf"), ]
    roc_auc_scores = [0.5, ]

    diff_in_loss = float("inf")
    iteration = 0
    while abs(diff_in_loss) > epsilon:
        iteration += 1
        dW_2, db_2, dW_1, db_1, loss = derivatives_by_backprop(
            X=X, Y=Y, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2
        )

        W_1 -= learning_rate * dW_1
        b_1 -= learning_rate * db_1
        W_2 -= learning_rate * dW_2
        b_2 -= learning_rate * db_2

        losses.append(loss)
        diff_in_loss = losses[-1] - losses[-2]

        Y_hat, _, _, _ = forward_prop(X=X, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2)
        roc_auc = roc_auc_score(y_true=Y, y_score=Y_hat)
        roc_auc_scores.append(roc_auc)

        if verbose and iteration % 10 == 0:
            print(loss, roc_auc)
```

```
    return W_1, b_1, W_2, b_2, losses
```

In [310]:
```
# parameters for the first layer
np.random.seed(42)
W_1_init = np.random.normal(size=(3, X.shape[1]))
b_1_init = np.random.normal(size=(3, 1))

# parameters for the second layer
W_2_init = np.random.normal(size=(1, 3))
b_2_init = np.random.normal(size=(1, 1))

W_1, b_1, W_2, b_2, losses = gradient_descent(
    X=X,
    Y=Y,
    W_1_init=W_1_init,
    b_1_init=b_1_init,
    W_2_init=W_2_init,
    b_2_init=b_2_init,
    learning_rate=0.1,
    epsilon=1e-3,
    verbose=True,
)
```

```
0.783355706236572 0.1278603053750983
0.6919297915722732 0.40731162328795467
0.6221831830749797 0.7705341995282824
0.5667720895357118 0.8880498200024827
0.5215585720622693 0.9298423470021102
0.48399626055251443 0.9451111019158357
0.4523753008520904 0.9553523399677246
0.4254695544919871 0.9608557123350022
0.4023620540738291 0.9643729052013075
0.3823494894203707 0.9665039102908926
0.3648838371694929 0.9681383705052344
0.34953341792280357 0.9694211114329457
0.33595538909336176 0.9708693673190715
0.32387555184301897 0.9714279803037199
0.3130730233613099 0.9721107295071791
```

In [335]:
```
# evaluate the model on the test set
Y_test_hat, _, _, _ = forward_prop(X=X_test, W_1=W_1, b_1=b_1, W_2=W_2, b_2=b_2)
roc_auc_score(y_true=Y_test, y_score=Y_test_hat)
```

Out[335]:
```
0.9905011464133638
```

In [ ]:
```
conda install -c conda-forge tensorflow
```

```
Collecting package metadata (current_repodata.json): | DEBUG:urllib3.connectionpool:Starting new HTTPS
connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): conda.anaconda.org:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): conda.anaconda.org:443
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/r/osx-arm64/current_repodata.jso
n HTTP/1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/main/noarch/current_repodata.jso
n HTTP/1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/r/noarch/current_repodata.json H
TTP/1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/main/osx-arm64/current_repodata.
json HTTP/1.1" 304 0
/ DEBUG:urllib3.connectionpool:https://conda.anaconda.org:443 "GET /conda-forge/osx-arm64/current_repo
data.json HTTP/1.1" 200 None
DEBUG:urllib3.connectionpool:https://conda.anaconda.org:443 "GET /conda-forge/noarch/current_repodata.
json HTTP/1.1" 200 None
done
Solving environment: unsuccessful initial attempt using frozen solve. Retrying with flexible solve.
Solving environment: unsuccessful attempt using repodata from current_repodata.json, retrying with nex
t repodata source.
Collecting package metadata (repodata.json): | DEBUG:urllib3.connectionpool:Starting new HTTPS connect
ion (1): conda.anaconda.org:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): conda.anaconda.org:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): repo.anaconda.com:443
DEBUG:urllib3.connectionpool:https://conda.anaconda.org:443 "GET /conda-forge/noarch/repodata.json HTT
P/1.1" 200 None
/ DEBUG:urllib3.connectionpool:https://conda.anaconda.org:443 "GET /conda-forge/osx-arm64/repodata.jso
n HTTP/1.1" 200 None
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/main/osx-arm64/repodata.json HTT
P/1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/main/noarch/repodata.json HTTP/
1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/r/osx-arm64/repodata.json HTTP/
1.1" 304 0
DEBUG:urllib3.connectionpool:https://repo.anaconda.com:443 "GET /pkgs/r/noarch/repodata.json HTTP/1.1"
304 0
- WARNING conda.models.version:get_matcher(556): Using .* with relational operator is superfluous and
```

```
deprecated and will be removed in a future version of conda. Your spec was 1.8.0.*, but conda is ignor
ing the .* and treating it as 1.8.0
WARNING conda.models.version:get_matcher(556): Using .* with relational operator is superfluous and de
precated and will be removed in a future version of conda. Your spec was 1.9.0.*, but conda is ignorin
g the .* and treating it as 1.9.0
done
Solving environment: unsuccessful initial attempt using frozen solve. Retrying with flexible solve.
Solving environment: /
Found conflicts! Looking for incompatible packages.
This can take several minutes.  Press CTRL-C to abort.
-
```

In [ ]:
```python
# train a NN with Keras
from tensorflow import keras
from tensorflow.keras import layers

def keras_model(nn_size: int, num_features: int, num_layers: int):
    """Creates a simple Keras model."""
    inputs = keras.Input(
        shape=(num_features, ), name="inputs")
    x = inputs
    for i in range(num_layers):
        x = layers.Dense(
            nn_size, activation="sigmoid", name=f"desnse_layer_{i}")(x)

    outputs = layers.Dense(
        1, activation="sigmoid", name="output")(x)

    model = keras.Model(
        inputs=inputs, outputs=outputs, name="simple_model")
    model.compile(
        optimizer="adam",
        loss="binary_crossentropy",
        metrics=["AUC"])
    return model

model = keras_model(nn_size=3, num_features=X.shape[1], num_layers=1)
history = model.fit(
    x=X,
    y=Y,
    batch_size=32,
    epochs=20,
    validation_data=(X_test, Y_test),
    verbose=1,
    shuffle=True,
)
```

In [ ]:
```python
# evaluate the model on the test set
roc_auc_score(y_true=Y_test, y_score=model.predict(X_test))
```

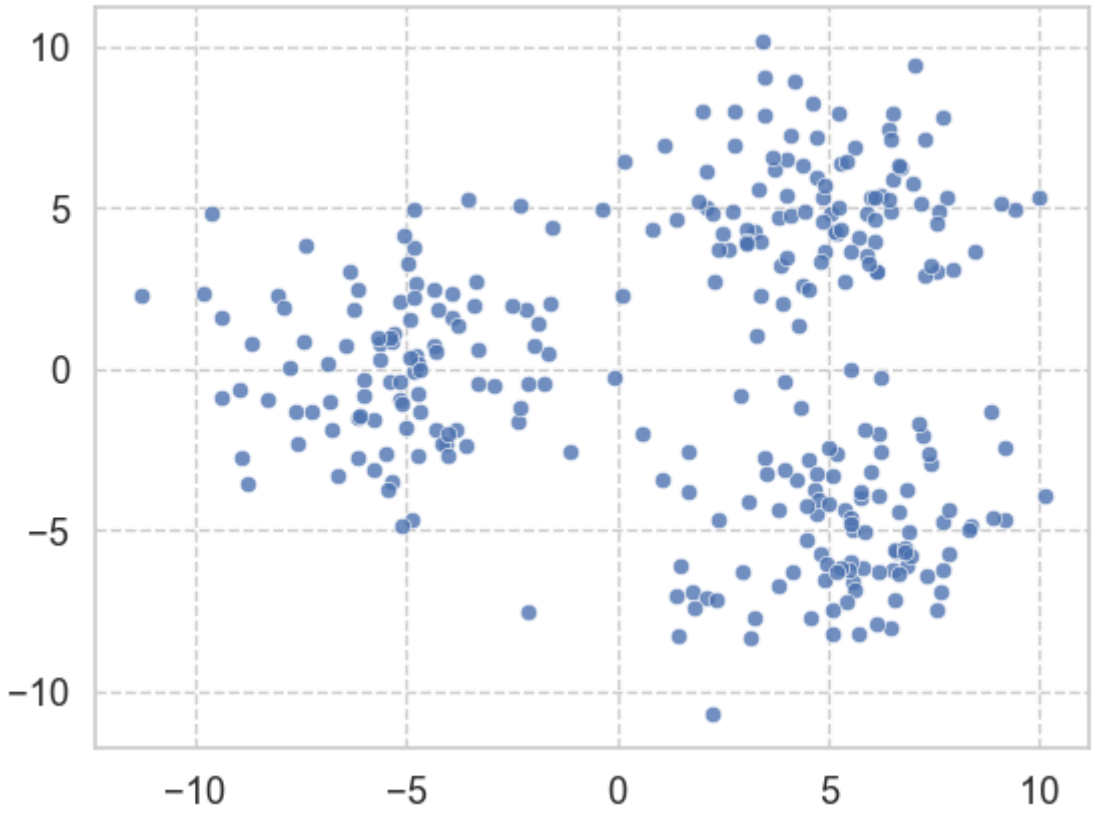
In [317]:
```
# Week 10
```

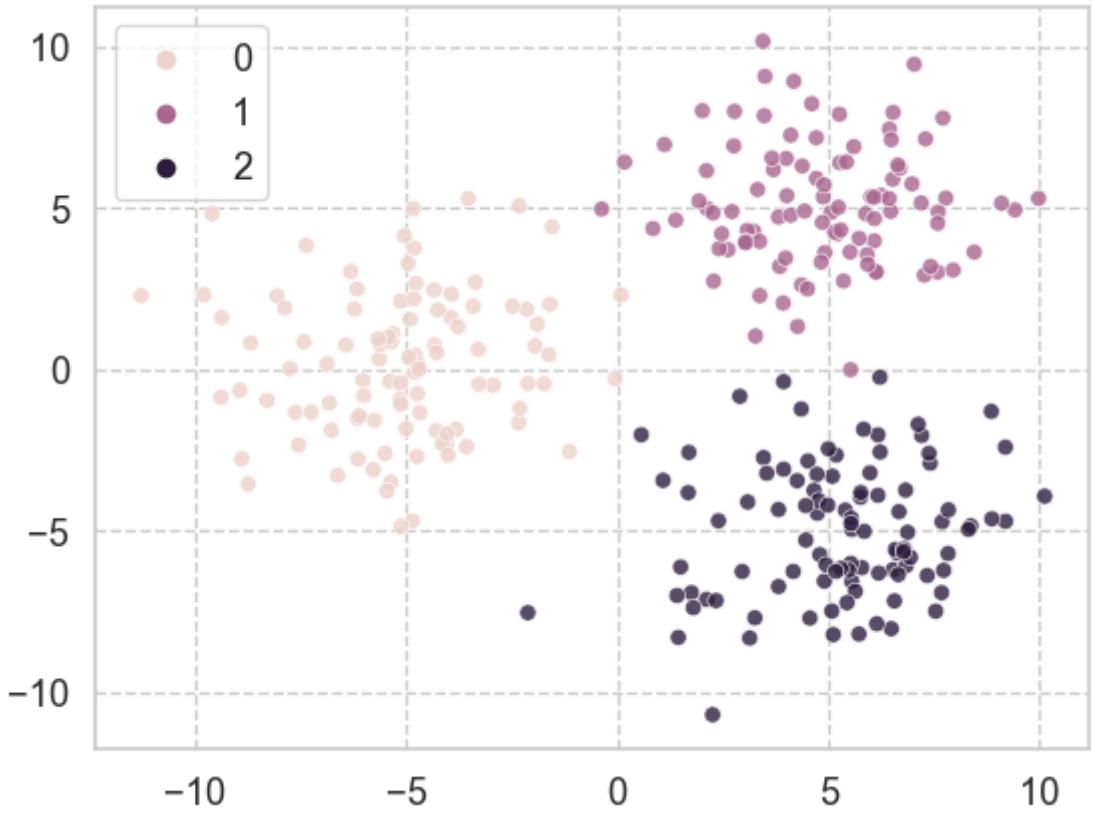
Week 10 - clustering.ipynb

```
    Introduction:
        - Clustering
        - K-means
        - K-means with improper choice of k
```

In [318]:
```
%matplotlib inline
import os
import shutil
from typing import Tuple

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from sklearn.datasets import make_blobs

sns.set(font_scale=1.2)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [319]:
```
# make blobs
X, y = make_blobs(
    n_samples=300,
    random_state=8,
    centers=[[-5, 0], [5, 5], [5, -5]],
    cluster_std=2.0,
)

# no label
plt.figure()
sns.scatterplot(x=X[:, 0], y=X[:, 1], alpha=0.8)

# with label
plt.figure()
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y, alpha=0.8)

plt.show()
```





In [320]:
```
# K-means example
```

K-means example

K-means clustering is a popular unsupervised machine learning algorithm used for partitioning a set of data points into a specified number of clusters or groups. The goal of K-means is to group similar data points together and discover patterns or structures within the data.

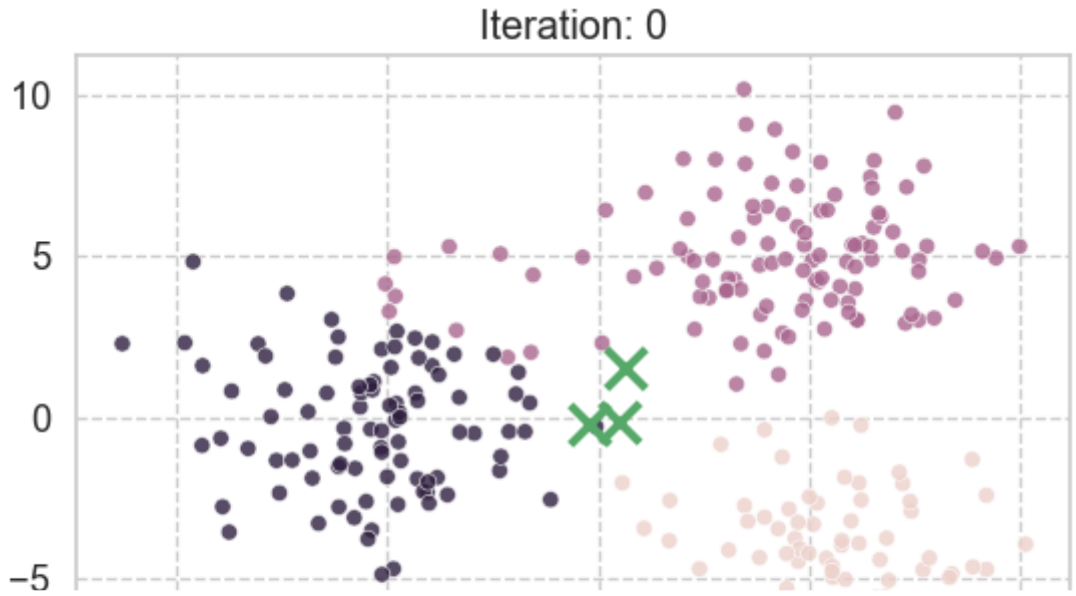
Below we will demonstrate how to use Lloyd's algorithm to perform K-means clustering.

In [321]:
```python
# Lloyd's algorithm
def find_cluster(X: np.array, centers: np.array) -> np.array:
    """
    Find the cluster assignment of each point from X, to the given centers.

    Args:
        X: The input data, that has the shape of (n, d).
        centers: The cluster centers, that have the shape of (k, d), where k is the number of clusters.
            The cluster assignment is the same as the list index.

    Returns:
        (np.array): The cluster assignment for each point. The shape is (n, 1).
    """
    all_distances = []  # it will have the shape of (n, k)
    for c in centers:
        distance = np.sum(np.square(X - c), axis=1)
        all_distances.append(distance)

    # reshape the distances to a 2D array
    all_distances = np.array(all_distances).T

    # find the cluster assignment
    assignments = np.argmin(all_distances, axis=1)
    return assignments

def find_centers(X: np.array, assignments: np.array) -> np.array:
    """Find the new center for each cluster."""
    cluster_index = sorted(np.unique(assignments))
    centers = []
    for k in cluster_index:
        center = np.mean(X[assignments == k], axis=0)
        centers.append(list(center))

    centers = np.array(centers)
    assert centers.shape == (len(cluster_index), X.shape[1]), \
        f"Expecting a shape of ({len(cluster_index)}, {X.shape[1]}), got {centers.shape}."
    return centers

def plot_assignment_and_centers(X: np.array, centers: np.array, assignments: np.array):
    """Plot the cluster assignments, and the corresponding cluster centroids."""
    plt.figure()
    sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=assignments, alpha=0.8)
    sns.scatterplot(x=centers[:, 0], y=centers[:, 1], marker="x", color="g", s=200, linewidth=3)
```

```
    plt.gca().get_legend().remove()


def kmeans(
    X: np.array,
    K: int,
    initial_centers: np.array,
    local_dir: str = "./kmeans_demo") -> Tuple:
    """Runs the kmeans using Lloyd's algorithm."""
    # initial guess of cluster assignments
    assignments_prev = np.random.choice(a=list(range(K)), size=X.shape[0])

    centers = initial_centers
    assignments_next = find_cluster(X, centers)

    centers_history = [centers]
    assignments_history = [assignments_next]
    # iterative search
    while not np.allclose(assignments_prev, assignments_next):
        centers = find_centers(X=X, assignments=assignments_next)
        assignments_prev = assignments_next.copy()
        assignments_next = find_cluster(X=X, centers=centers)
        centers_history.append(centers)
        assignments_history.append(assignments_next)

    # visualize the history
    if local_dir:
        try:
            shutil.rmtree(local_dir)
        except OSError:
            print("No directory to remove.")
        os.makedirs(local_dir)

        for i, (centers, assignments) in enumerate(zip(centers_history, assignments_history)):
            plot_assignment_and_centers(X=X, centers=centers, assignments=assignments)
            plt.title(f"Iteration: {i}")
            plt.savefig(os.path.join(local_dir, f"iteration_{i}.png"))

    return centers_history, assignments_history
```

In [322]:
```
np.random.seed(42)

K = 3  # number of clusters
# inital guess of cluster centers
centers = np.random.normal(size=(K, 2))

centers_history, assignments_history = kmeans(X=X, K=K, initial_centers=centers)
# how to make gif
# convert -background white -alpha remove -layers OptimizePlus -delay 50 *.png kmeans.gif
```

```
No directory to remove.
```

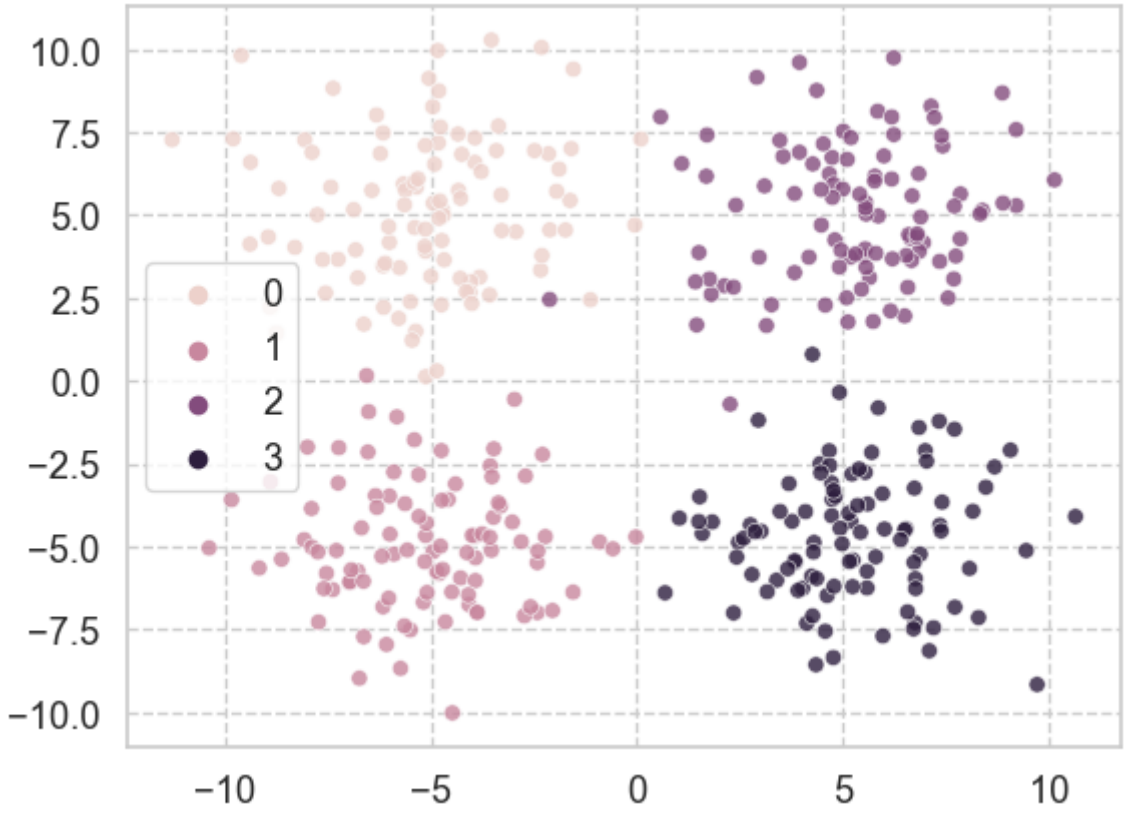


In [323]:
```
# K-means with improper choice of K example
```

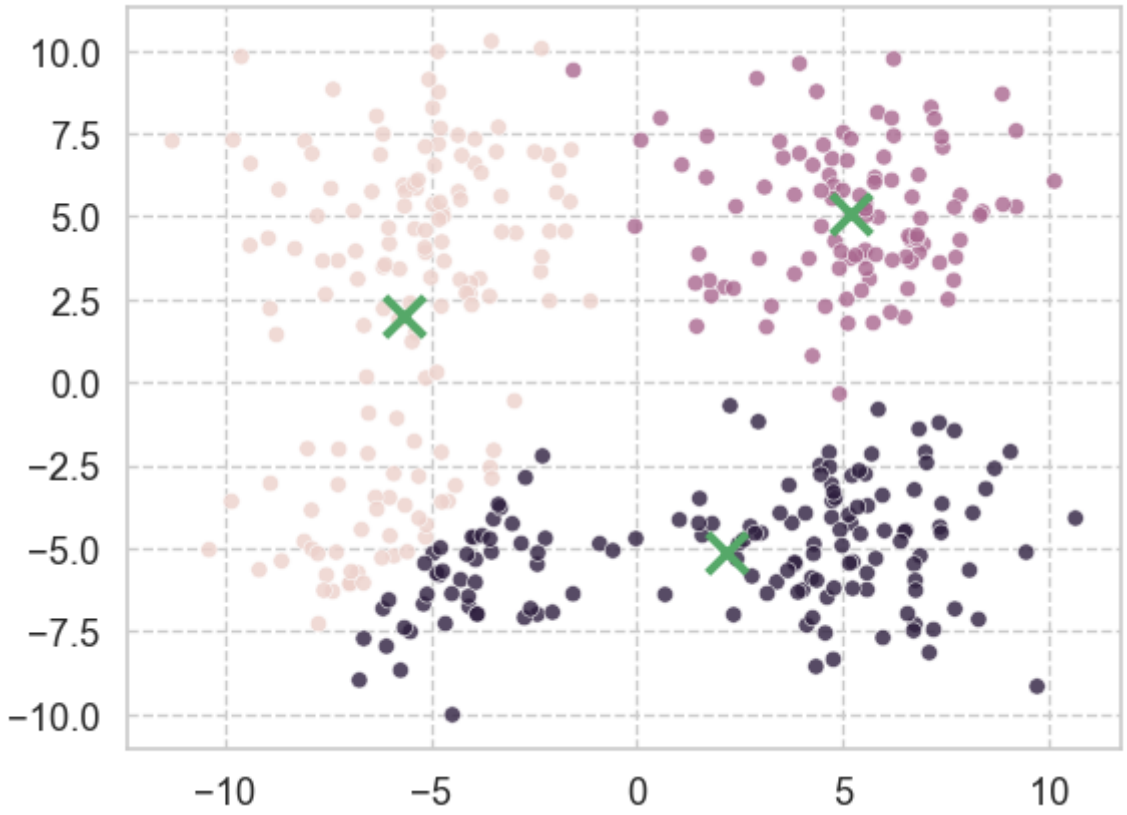
K-means with improper choice of K

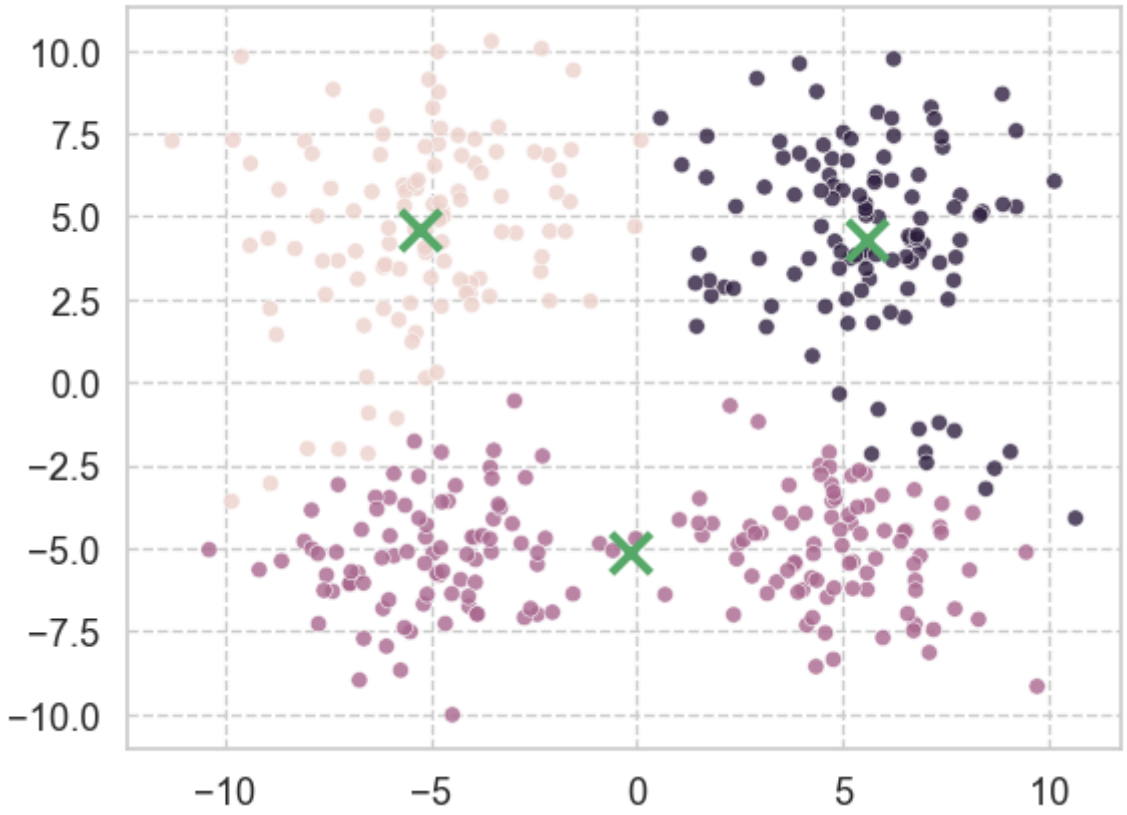
In the example below, we will show if the choice of K is poor, that is, does not reflect the nature of the underlying data, the K-means will lead to (obvious) bad results.

In [324]:
```
# different example
np.random.seed(42)

X_4_clusters, y_4_clusters = make_blobs(
    n_samples=400,
    random_state=8,
    centers=[[-5, 5], [-5, -5], [5, 5], [5, -5]],
    cluster_std=2.0,
)

plt.figure()
sns.scatterplot(x=X_4_clusters[:, 0], y=X_4_clusters[:, 1], hue=y_4_clusters, alpha=0.8)
plt.show()
```

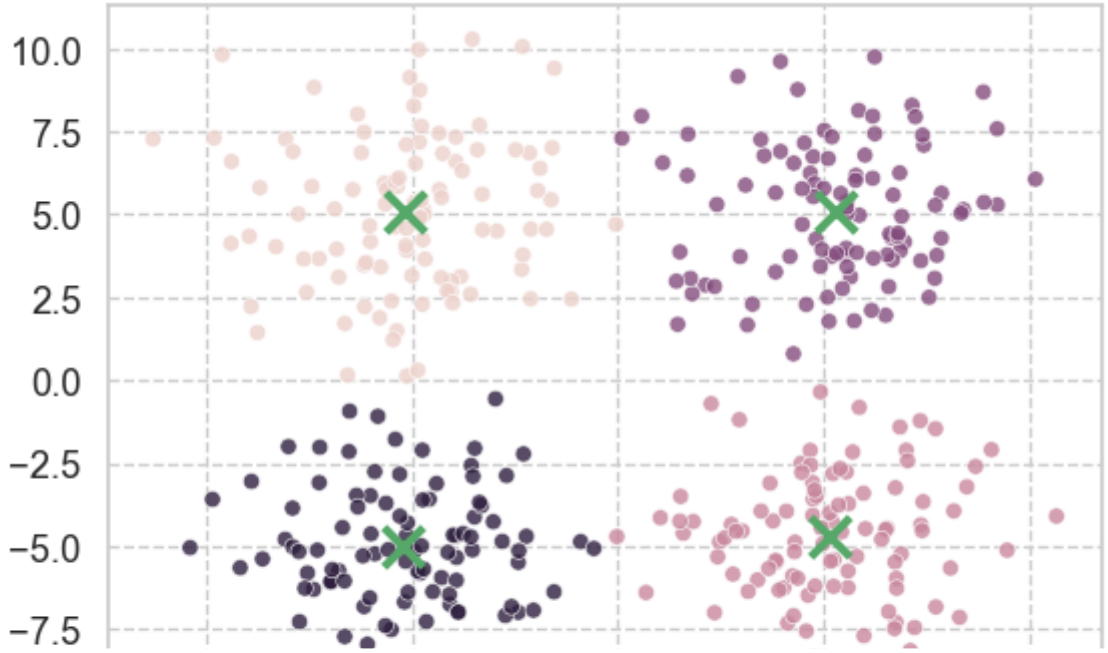


In [325]:
```
K = 3
centers_history, assignments_history = kmeans(
    X=X_4_clusters,
    K=K,
    initial_centers=[[-5, 5], [5, 5], [5, -5]],
    local_dir=None)
plot_assignment_and_centers(
    X=X_4_clusters,
    centers=centers_history[-1],
    assignments=assignments_history[-1])

centers_history, assignments_history = kmeans(
    X=X_4_clusters,
    K=K,
    initial_centers=[[-5, 5], [0, -5], [5, 5]],
    local_dir=None)
plot_assignment_and_centers(
    X=X_4_clusters,
    centers=centers_history[-1],
    assignments=assignments_history[-1])
```


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10
−5
0
5
10


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10
−5
0
5
10

In [326]:
```
np.random.seed(123)

for K in range(4, 6):
    centers_history, assignments_history = kmeans(
        X=X_4_clusters,
        K=K,
        initial_centers=np.random.normal(size=(K, 2)),
        local_dir=None)
    plot_assignment_and_centers(
        X=X_4_clusters,
        centers=centers_history[-1],
        assignments=assignments_history[-1])
```



In [327]:
```
# Week 10 cont.
```

Week 10 - special_clustering.ipynb

```
Introduction:
    - Special Clustering
```

In [328]:
```
# Special Clustering
```

Spectral clustering

Spectral clustering is a technique in machine learning for grouping objects into clusters that are similar to each other. Unlike traditional clustering methods like K-means, spectral clustering doesn't make strong assumptions about the form of the clusters.

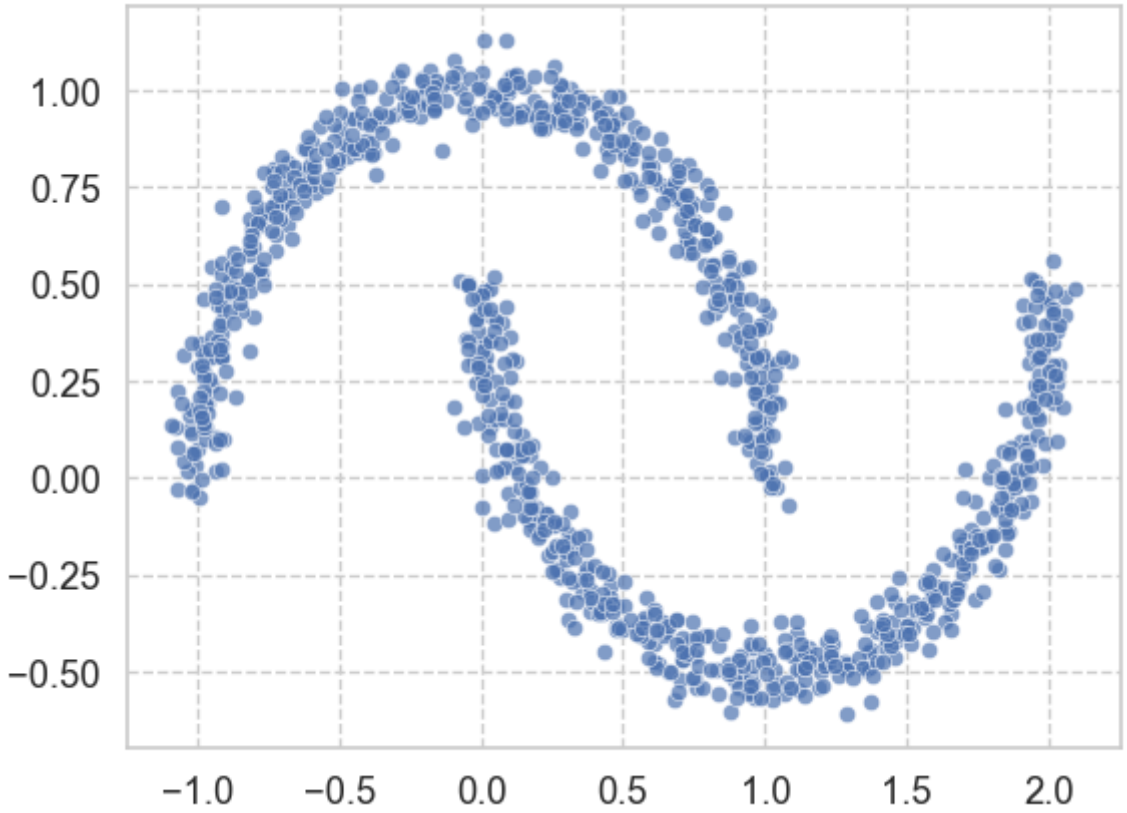
It is particularly useful when the structure of the individual clusters is highly irregular or more complex than globular shapes (like the data shown below). It's also useful for clustering graphs (like social networks). However, it can be computationally expensive for very large datasets and sensitive to the choice of similarity metric and the number of clusters specified.

Below, we will demonstrate spectral clustering with scikit-learn.

In [329]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from copy import deepcopy
from scipy.spatial.distance import euclidean
from sklearn import datasets

sns.set(font_scale=1.2)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

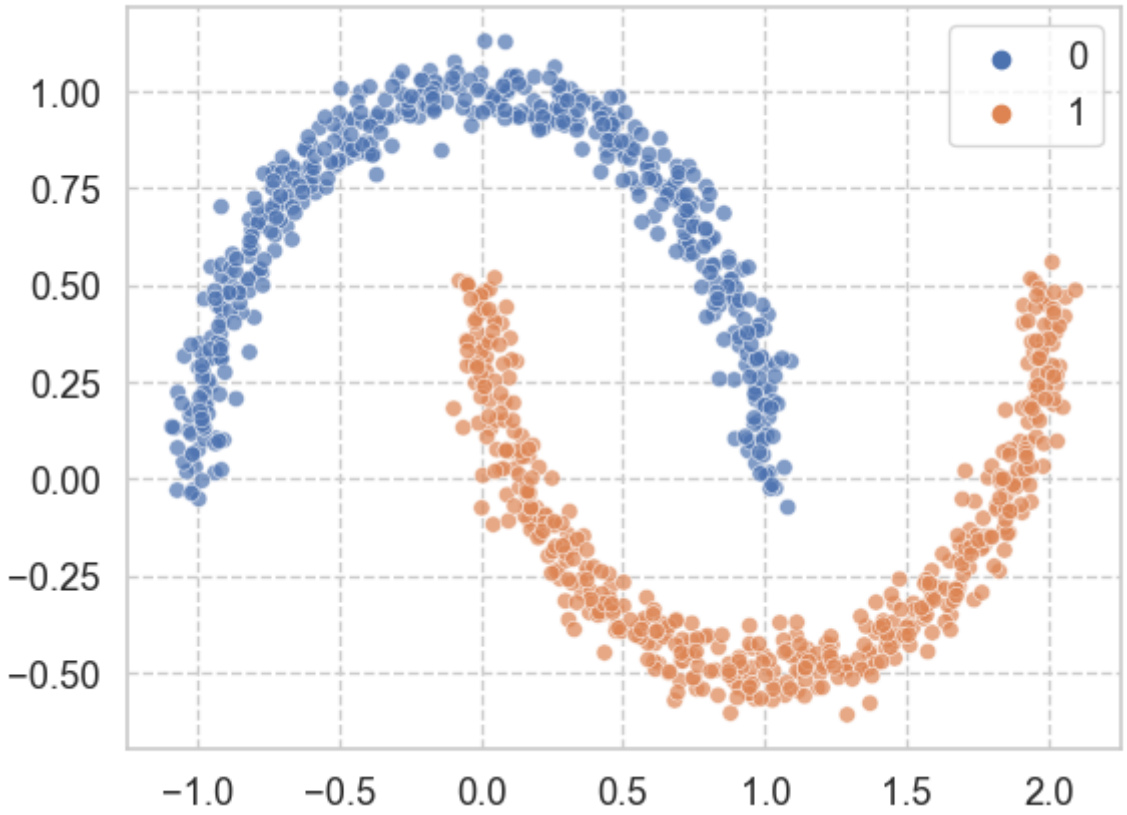
In [330]:
```
random_state = 42
np.random.seed(random_state)

n_samples = 1000
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
```

In [331]:
```
# noisy moon
plt.figure()
sns.scatterplot(x=noisy_moons[0][:, 0], y=noisy_moons[0][:, 1], alpha=0.7)
plt.show()
```

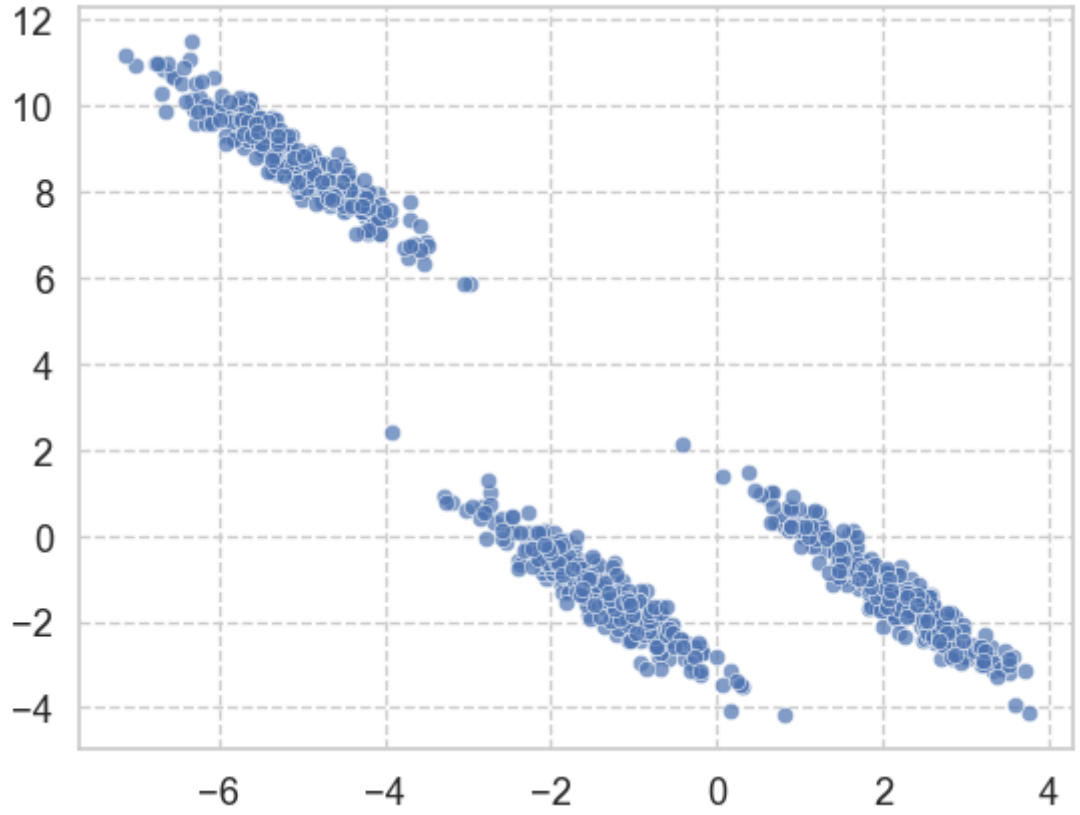


In [332]:
```
# Use scikit-learn
from sklearn.cluster import SpectralClustering

sc = SpectralClustering(
    n_clusters=2,
    affinity="nearest_neighbors",
    random_state=42
)
labels = sc.fit_predict(noisy_moons[0])

plt.figure()
sns.scatterplot(
    x=noisy_moons[0][:, 0],
    y=noisy_moons[0][:, 1],
    hue=labels,
    alpha=0.7
)
plt.show()
```

```
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/manifold/_spectral_embedding.p
y:273: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
```

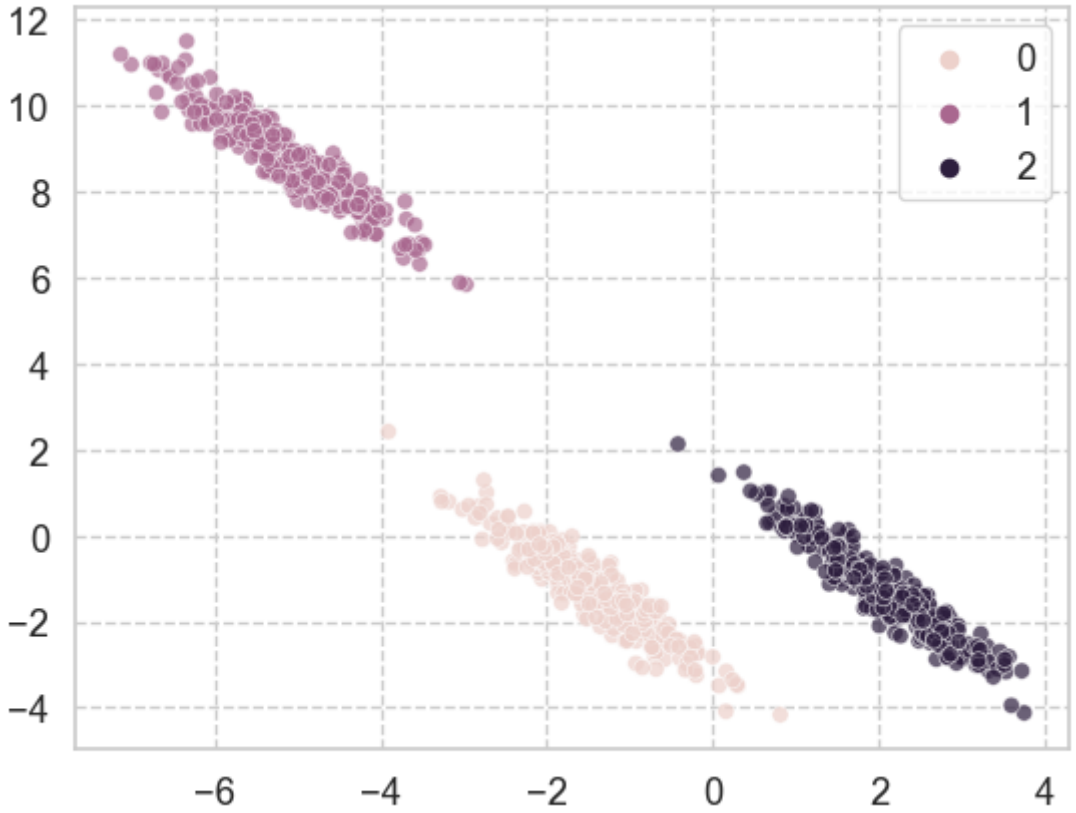

0
1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

In [333]:
```
# Anisotropicly distributed data
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

plt.figure()
sns.scatterplot(x=X_aniso[:, 0], y=X_aniso[:, 1], alpha=0.7)
plt.show()
```



In [334]:
```
sc = SpectralClustering(
    n_clusters=3,
    affinity="nearest_neighbors",
    random_state=42
)
labels = sc.fit_predict(X_aniso)

plt.figure()
sns.scatterplot(
    x=X_aniso[:, 0],
    y=X_aniso[:, 1],
    hue=labels,
    alpha=0.7
)
plt.show()
```

```
/Users/nicolinoprimavera/anaconda3/lib/python3.11/site-packages/sklearn/manifold/_spectral_embedding.p
y:273: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
```


12
10
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
0
1
2