# Module import

In [ ]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import random
import csv
```

In [ ]:
```
# You can import other modules if needed
```

# 1. Visualize a mixture of Gaussians [15 pts]

Consider a mixture of two 2D Gaussian distrbutions as follows
$0.4 \cdot \mathcal{N}\left(\begin{bmatrix}10\\2\end{bmatrix}, \begin{bmatrix}1,0\\0,1\end{bmatrix}\right) + 0.6 \cdot \mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}8.4, 2.0\\2.0, 1.7\end{bmatrix}\right)$

a. Compute the marginal distributions for each dimension.

b. Compute the mean for each marginal distribution.

c. Compute the mean for the two-dimensional distribution.

d. Generate both joint and marginal density of this distribution by taking at least 10 thousand samples. Please refer to Lab4 notebook for the visualization approach.

Type the answers to first three questions here:

(a) For the first dimension the marginal distribution of two gaussians are $x_1$ = $N(10,1)$ and $x_1$ = $N(0,8.4)$. the mixture of two 2D Gaussiansn has a probability ratio of 0.4 and 0.6. So

$$p(x_1) = 0.4 \times N(10,1) + 0.6 \times N(0,8.4)$$

Similarly, for $p(x_2)$

$$p(x_2) = 0.4 \times N(2,1) + 0.6 \times N(0,1.7)$$

(b) I specify the marginal distribution for each dimension as normal distributions. For the first dimension,

The mean for $x_1$,

$$\mu x_1 = 0.4 \times 10 + 0.6 \times 0 = 4$$

and for $x_2$,

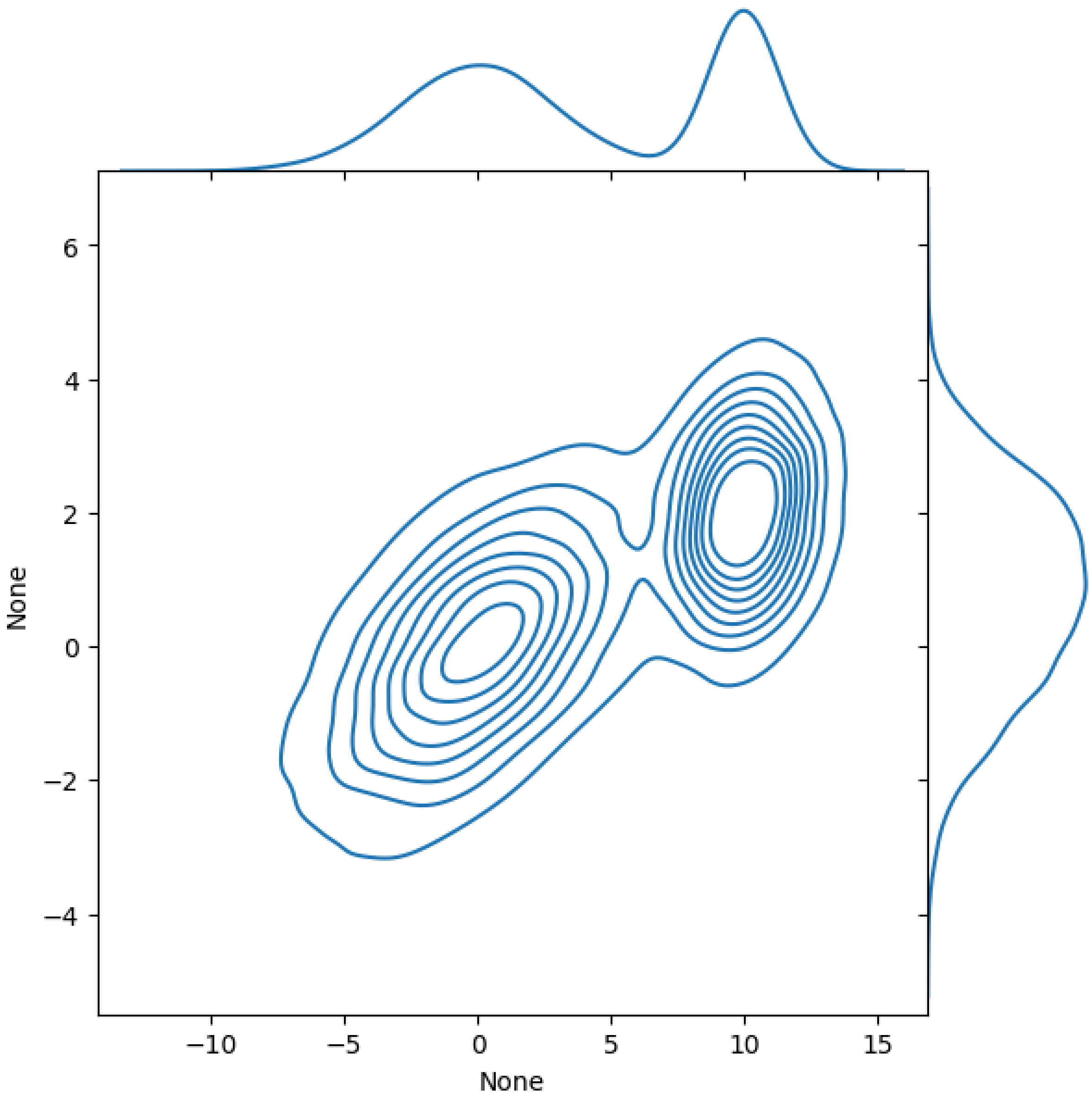
$$\mu x_2 = 0.4 \times 2 + 0.6 \times 0 = 0.8$$

(c) The mean for the two-dimensional distribution is for $x_1$ = $0.4 * 10 + 0.6 * 0 = 4$, and for $x_2$ = $0.4 * 2 + 0.6 * 0 = 1.2$. So the mean for the two-dimensional distribution is

$$\mu = 0.4 \times \begin{bmatrix} 10 \\ 2 \end{bmatrix} + 0.6 \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 0.8 \end{bmatrix}$$

```
In [ ]: # Your code for question d here
        d = 2                                                   #2D gaussian has 2 c


        def generate_two_d_gaussian(m, cov, d, samples):
            epsilon = 0.0001
            K = cov + epsilon*np.identity(d)
            L = np.linalg.cholesky(K)
            n = samples
            u = np.random.normal(loc=0, scale=1, size=d*n).reshape(d, n)

            x = m + np.dot(L, u)
            return x

        first_gaussian_mean = np.array([10, 2]).reshape(2, 1)
        second_gaussian_mean = np.array([0, 0]).reshape(2, 1)

        first_covariance_matrix = np.array([[1, 0], [0, 1]])
        second_covariance_matrix = np.array([[8.4, 2.0], [2.0, 1.7]])

        x_samples = 4000                                    #4000 samples from first
        y_samples = 6000                                    #6000 samples from secor

        x = generate_two_d_gaussian(first_gaussian_mean, first_covariance_matrix, d,
        y = generate_two_d_gaussian(second_gaussian_mean, second_covariance_matrix,

        z = np.hstack((x, y))
        sns.jointplot(x=z[0], y=z[1], kind="kde", space=0)
```

```
Out[ ]: <seaborn.axisgrid.JointGrid at 0x105e760f0>
```



## 2. Simulation: Drop needles [15 pts]

Suppose we have a floor made of parallel strips of wood, each the same width $t$, and we drop a needle with length $l = t$ onto the floor. What is the probability that the needle will lie across a line between two strips?

Below is an example of two needles dropped. Needle a falls across a line, while needle b does not.

Example

In this coding homework, we will simulate such experiments and connect them with the estimation of $\pi$.

## Functions

The first thing to write is a function *drop_needle*. It simulates dropping a needle onto the floor we described and returns whether the needle lies across a line between two strips.

Now the question is how to describe the position of a needle using random variables. The figure below visualizes a needle sampled, with $t = l = 1$ (see figure above). Remember that the needle should have an equal probability of landing in any position. In fact, we can uniformly sample the position of the needle's mass center and then uniformly sample the angle formed by the needle and the x-axis. Specifically, we only focus on the mass center's position with respect to (w.r.t.) the x-axis since we can assume the strip is long enough.

Besides, we do not need to sample the x-value of the center from $-\inf$ to $\inf$. Instead, we can uniformly sample it from $0$ to $2t$. Why is this the case?

needleExmple2

In [ ]:
```
def drop_needle(strip_length, needle_length):
    """
    Simulate dropping a needle on to the floor made of parallel strips of wo
    Return whether the needle lie across a line between two strips.

    :return: An Integer that equals to 1 if the needle lie across a line, ar
    """

    # write your code here
    """ middle point of the needle is in the range of 0 - 2 x needle_length
    middle_point = random.uniform(0, 2 * needle_length)
    """ theta is in the range of 0 - 90 degrees """
    theta = random.uniform(0, math.pi/2)
    half_length = math.cos(theta) * needle_length / 2
    """ x is in the range of 0 - strip_length """
    """Determining the distance between the middle point of the needle and t
    if middle_point < 1 * strip_length:
        if middle_point - half_length < 0 or middle_point + half_length > 1
            return 1
        else:
            return 0
    elif middle_point < 2 * strip_length:
        if middle_point - half_length < 1 * strip_length or middle_point + h
            return 1
        else:
            return 0
```

Next, write a function run_simulation that calls drop_needle repetitively for n times. The function should return the probability that a dropped needle lies across a line based on the n trials.

In [ ]:
```
def run_simulation(n, strip_length, needle_length):
    """
    Repeat drop_needle experiment for n times. Return the probability that t

    :return: float, the probability that the needle will lie across a line a
    """
    # Write your code here
    i, count = 0, 0
    while i < n:
        i += 1
        if drop_needle(strip_length, needle_length) == 1:
            count += 1
    probability = count / n
    return probability
```

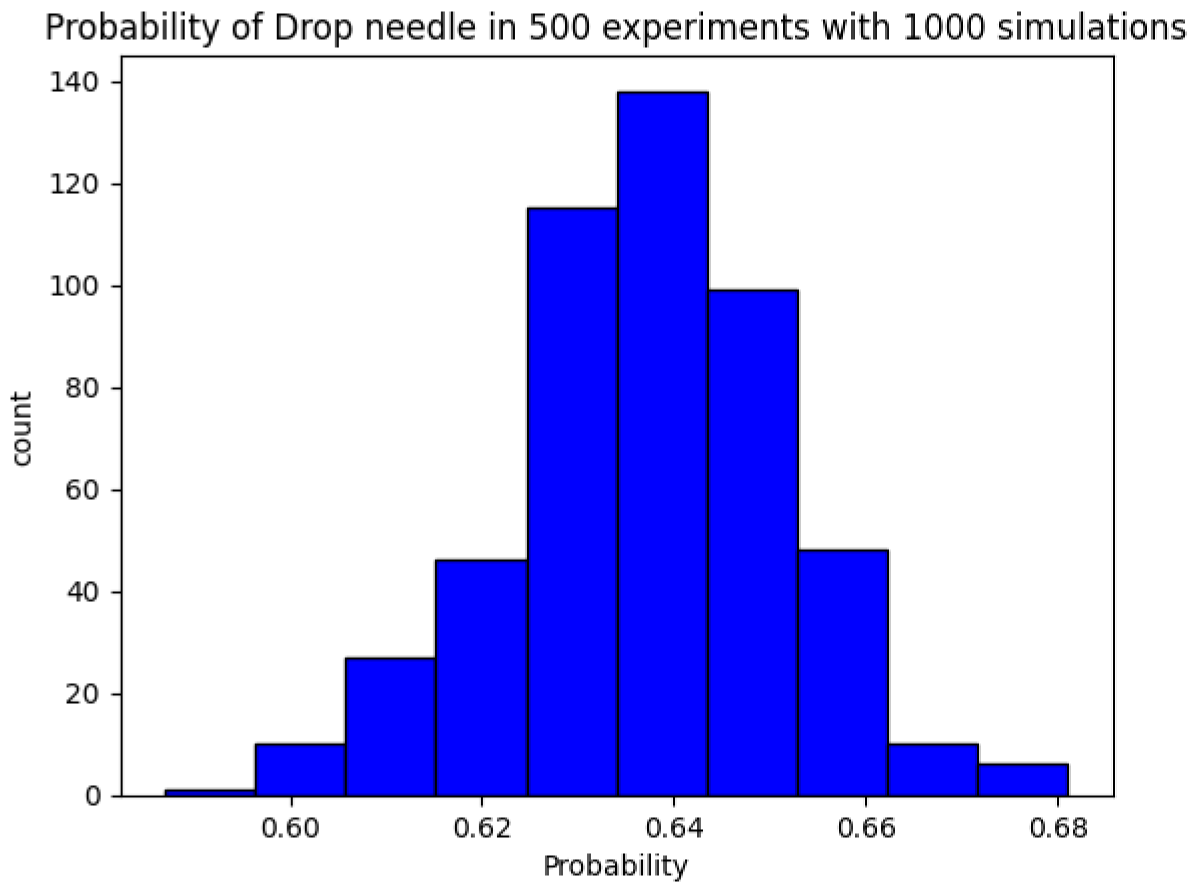
## Run the simulation

Run the *run_simulation* function 500 times with parameters n=1000, strip_length=1, and needle_length=1. Each time the function is going to return a probability of the needle lying across the line. Plot a histogram of those 500 probabilities.

In [ ]:
```
# Write your code here
""" Run the simulation 1000 times with strip_length = 1 and needle_length =
""" parameter: experiment, strip_length, needle_length """
experiment = 500
n = 1000
strip_length = 1
needle_length = 1

probabilities = []
""" Run the simulator 500 times, calculate the probability that the needle w
for i in range(experiment):
    probability = run_simulation(n, strip_length, needle_length)
    probabilities.append(probability)
```

In [ ]:
```
""" Plot the result"""
plt.hist(probabilities, color = 'blue', edgecolor = 'black')
plt.title('Probability of Drop needle in 500 experiments with 1000 simulatic
plt.xlabel('Probability')
plt.ylabel('count')
plt.show
```

Out[ ]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



## 3. Email Spam Naive Bayes [35 pts]

### Naive Bayes

For a classification task, we aim to predict a binary label $Y \in \{0, 1\}$ of an example given its $d$ dimensional features $X_1, X_2, \ldots, X_d$. Bayes Theorem states the (posterior) porbability of have a label $Y$ given the feature observations as follows:

$P(Y|X_1, X_2, \ldots, X_d) = \frac{P(Y)*P(X_1, X_2, \ldots, X_d|Y)}{P(X_1, X_2, \ldots, X_d)}$

Note $P(Y)$ is called priori, i.e., probability of seeing label $Y$ without observing the features. To determine the binary label $Y$, we need to compare $P(Y = 0|X_1, X_2, \ldots, X_d)$ with $P(Y = 1|X_1, X_2, \ldots, X_d)$. In Bayes Theorem, the fractions representing both conditional probabilities have the same denominator $P(X_1, X_2, \ldots, X_d)$. Thus we can ignore the denominator and only focusing on

$$P(Y|X_1, X_2, \ldots, X_d) \propto P(Y) * P(X_1, X_2, \ldots, X_d|Y)$$

Next, we make a strong assumption: no pair of features in the dataset are dependent (this is the reason why we call it *Naive* Bayes). Under such assumption,

$P(X_1, X_2, \ldots, X_d|Y) = P(X_1|Y) * P(X_2|Y) * \ldots * P(X_d|Y)$. Therefore, we conclude

$$P(Y|X_1, X_2, \ldots, X_n) \propto P(Y) * P(X_1|Y) * P(X_2|Y) * \ldots * P(X_d|Y)$$

The classifier will decide what class each input belongs to based on highest probability from the equation above.

How do we know the priori $P(Y)$ and the conditional probabilities $P(X_i|Y)$? In supervised classification tasks we have labeled data available (we call it training set). We can thus estimate these probabilities based on the training data.

## Overview/Task

The goal of this programming assignment is to build a naive bayes classifier from scratch that can determine whether email text should be labled spam or not spam based on its contents

## Reminders

Please remember that the classifier must be written from scratch; do NOT use any libraries that implement the classifier for you, such as but not limited to sklearn.

You CAN, however, use SKlearn to split up the dataset between testing and training.

Feel free to look up any tasks you are not familiar with, e.g. the function call to read a csv

## Task list/Recommended Order

In order to provide some guidance, I am giving the following order/checklist to solve this task:

1. Compute the "prior": P(Y) for Y = 0 (not spam) and Y = 1 (spam)
2. Compute the "likelihood": $P(X_n|Y)$. In this task, we recommend you to start with setting up a dictionary of words appeared in the training set $W = \{w_1, w_2 \ldots\}$, and constructing feature set $\{X_1, X_2 \ldots\}$ where $X_i$ is a binary indicator of whether the word $w_i$ exists in the text or not. Feel free to add preprocessing and modify the features if that can increase your accuracy!
3. Write code that uses the two items above to make a decision on whether or not an email is spam or ham (aka not spam)
4. Write code to evaluate your model. Test model on training data to debug
5. Test model on testing data to debug

## Function template

In [ ]:
```
def prior(df):
    ham_prior = 0
    spam_prior =  0
    '''YOUR CODE HERE'''
    num_rows = len(df.label)                                        #number of r
    for row in range(num_rows):                                     #iterate thr
        if df.label[row] == "spam":                                 #if the labe
            spam_prior += 1
        else:                                                       #if the labe
            ham_prior += 1
    ''' Compute the probability of each class'''
    ham_prior = ham_prior/num_rows                                  #divide the
    spam_prior = spam_prior/num_rows                                #divide the
    '''END'''
    return ham_prior, spam_prior

def likelihood(df):
    ham_like_dict = {}
    spam_like_dict = {}
    '''YOUR CODE HERE'''
    ''' Preprocess the text column of the dataframe to get the likelihood of
    # spam_total = 0
    # ham_total = 0
    num_rows = len(df.text)                                         #number of r
    punctuation = '''!()-[]{};:'",<>./?@#$%^&*_~'''               #define the pu
    # word_appear = []
    ''' Add each words to the dictionary and count the number of times it ap
    for row in range(num_rows):                                     #iterate thr
        for word in df.text[row].split():                           #iterate thr
            word = word.replace('Subject:', '')                     #remove the
            word = word.replace('\n', ' ')                          #remove new
            word = word.lower()                                     #convert the
            if word not in punctuation:
                if df.label[row] == "spam":
                    if word in spam_like_dict:
                        spam_like_dict[word] += 1
                    else:
                        spam_like_dict[word] = 1
                else:
                    if word in ham_like_dict:
                        ham_like_dict[word] += 1
                    else:
                        ham_like_dict[word] = 1
    '''END'''
    return ham_like_dict, spam_like_dict

def predict(ham_prior, spam_prior, ham_like_dict, spam_like_dict, text):
    '''
    prediction function that uses prior and likelihood structure to compute
    '''
    #ham_spam_decision = 1 if classified as spam, 0 if classified as normal/
    ham_spam_decision = None
    '''YOUR CODE HERE'''
    punctuation = '''!()-[]{};:'",<>./?@#$%^&*_~'''        #define the punct
    text = text.replace('Subject:', '')                     #remove the word
    text = text.replace('\n', ' ')                          #remove new line
    text = text.lower()                                     #convert the wor

    #ham_posterior = posterior probability that the email is normal/ham
    ham_posterior = ham_prior
    ham_count = 0

    """ Count the total numbers of word appearances in the ham_like_dict"""
    for ham_word in ham_like_dict:
        ham_count += ham_like_dict[ham_word]

    for word in text.split():                                   #iterate thr
        word = word.replace('Subject:', '')                     #remove the
        word = word.replace('\n', ' ')                          #remove new
        word = word.lower()                                     #convert the
        if word not in punctuation:
            if word in ham_like_dict:
                ham_posterior *= ham_like_dict[word] / ham_count
            else:
                ham_posterior *= 0.1 / ham_count

    #spam_posterior = posterior probability that the email is spam
    spam_posterior = spam_prior
    spam_count = 0

    """ Count the total numbers of word appearances in the ham_like_dict"""
    for spam_word in spam_like_dict:
        spam_count += spam_like_dict[spam_word]

    for word in text.split():                                   #iterate thr
        if word not in punctuation:
            if word in spam_like_dict:
                spam_posterior *= spam_like_dict[word] / spam_count
            else:
                spam_posterior *= 0.1 / spam_count

    ''' Multiply the posterior probability by the prior probability to get t
    if spam_posterior  > ham_posterior :
        ham_spam_decision = 1
    else:
        ham_spam_decision = 0

    '''END'''
    return ham_spam_decision


def metrics(ham_prior, spam_prior, ham_dict, spam_dict, df):
    '''
    Calls "predict" function and report accuracy, precision, and recall of y
    '''

    '''YOUR CODE HERE'''
    ''' define the variables for the metrics'''
    true_positive = 0
    false_positive = 0
    true_negative = 0
    false_negative = 0

    ''' iterate through the rows of the dataframe and calculate the metrics
    for row in range(len(df.text)):
        prediction = predict(ham_prior, spam_prior, ham_dict, spam_dict, df.
        if prediction == 1 and df.label[row] == "spam":
            true_positive += 1
        elif prediction == 1 and df.label[row] == "ham":
            false_positive += 1
        elif prediction == 0 and df.label[row] == "ham":
            true_negative += 1
        elif prediction == 0 and df.label[row] == "spam":
            false_negative += 1

    ''' calculate the accuracy, precision, and recall'''
    acc = (true_positive + true_negative) / (true_positive + true_negative +
    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negative)
    '''END'''
    return acc, precision, recall
```

## Generate answers with your functions

In [ ]:
```
#loading in the training data
train_df = pd.read_csv("./TRAIN_balanced_ham_spam.csv")
test_df = pd.read_csv("./TEST_balanced_ham_spam.csv")
df = train_df
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2398 entries, 0 to 2397
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   Unnamed: 0.1  2398 non-null   int64 
 1   Unnamed: 0    2398 non-null   int64 
 2   label         2398 non-null   object
 3   text          2398 non-null   object
 4   label_num     2398 non-null   int64 
dtypes: int64(3), object(2)
memory usage: 93.8+ KB
```

In [ ]:
```
#compute the prior
ham_prior, spam_prior = prior(df)

print(ham_prior, spam_prior)
```

```
0.5 0.5
```

```
In [ ]: # compute likelihood
        ham_like_dict, spam_like_dict = likelihood(df)
```

```
In [ ]: # Test your predict function with some example TEXT
        some_text_example = ""
        spam_text_example = " Congratulations! You've been selected as a winner of c
        not_spam_text_example = "I hope this email finds you well. I'm writing to ir
        print("Not Spam text ", predict(ham_prior, spam_prior, ham_like_dict, spam_l
        print("Spam text ", predict(ham_prior, spam_prior, ham_like_dict, spam_like_
```

```
Not Spam text  0
Spam text  1
```

```
In [ ]: # Predict on test_df and compute metrics
        df = test_df
        acc, precision, recall = metrics(ham_prior, spam_prior, ham_like_dict, spam_
        print(acc, precision, recall)
```

```
0.745 0.9932885906040269 0.49333333333333335
```

```
In [ ]:
```

```
In [ ]:
```

## 4. Gradient Decent [35 pts]

### Exact Gradient Computation

Given a function f, sometimes we can compute its exact gradient at any x if f's derivative is easy to compute. For example, let $f(x) = 2x^2 - 3x + \ln x$, where $x > 0$. Please compute the derivative of f and report its gradient at $x = 2$.

Your answer: 5.5

### Numerical Gradient Computation

Instead of computing the derivative of a function, we can also estimate the gradient numerically with various methods. These methods are essential, especially when a callable function is not exposed due to privacy reasons, or it is hard to differentiate analytically.

To numerically compute the gradient, the simple way is to follow Newton's difference quotient: $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$. Another two-point formula is to compute the slope through the points $(x-h, f(x-h))$ and $(x+h, f(x+h))$. Let us reuse the example function $f(x) = 2x^2 - 3x + \ln x$ and test the precision of these two approaches. Define the function in the next cell, and try to compute its gradient via both methods at $x = 2$. Range h value in [0.1,0.01,0.001,0.0001] and report all gradients calculated. Which method is more accurate, and why does it work better?

```
In [ ]: def f(x):
            # Your code here
            return 2 * x**2 - 3 * x + math.log(x)
```

```
In [ ]: # Compute gradient using the first method (Newton's difference quotient)
        h = [0.1, 0.01, 0.001, 0.0001]
        gradient_result =[]
        for i in h:
            result = (f(2 + i) - f(2)) / i
            gradient_result.append(result)

        print(gradient_result)
```

```
[5.687901641694317, 5.518754151103744, 5.50187504165045, 5.5001875004201395]
```

```
In [ ]: # Compute gradient using the second method
        h = [0.1, 0.01, 0.001, 0.0001]

        gradient_result =[]
        for i in h:
            result = (f(2 + i) - f(2 - i)) / (2 * i)
            gradient_result.append(result)

        print(gradient_result)
```

```
[5.500417292784909, 5.500004166729067, 5.500000041665842, 5.500000000417948]
```

### Remark

You may find the gradient more accurate when using a smaller h value. However, this is not always the case. Due to the finite precision of the floating-point, rounding errors always exist and can dominate the computation when the h value is too small. Run the following two cells to observe such scenarios.

```
In [ ]: eps = 1e-15
        print((f(2+eps)-f(2-eps))/2/eps)
```

```
5.551115123125783
```

```
In [ ]: eps = 1e-20
        print((f(2+eps)-f(2-eps))/2/eps)
```

```
0.0
```

```
In [ ]:
```

## Logistic regression

Logistic regression is a classification tool that models the probability of an event taking place by having the log odds for the event be a linear combination of one or more independent variables. Specifically, let $\vec{x} =< x_1, \ldots, x_m >$ be an m dimensional vector of independent variables (features), and $y$ be the corresponding binary dependent variable (label). The probability of having $y = 1$ is modeled as

$$P_y = \frac{1}{1 + e^{-(b_0 + b_1 \cdot x_1 + \cdots + b_m \cdot x_m)}} = \frac{1}{1 + e^{-(b_0 + \vec{b}_{1:m} \cdot \vec{x})}}$$

Given a set of data points $< \vec{x}_k, y_k >$ with $k \in [1, n]$, how can we fit the model with these data, i.e., how to choose the best $\vec{b} = b_0, b_1 \cdots, b_m$?

One way is to write out the likelihood

$$\prod_{k:y_k=1} P_{y_k} \prod_{k:y_k=0} (1 - P_{y_k})$$

and find $b_0, b_1 \cdots, b_m$ that maximize its logarithm,

$$l = \sum_{k:y_k=1} \ln(P_{y_k}) + \sum_{k:y_k=0} \ln(1 - P_{y_k})$$

In contrast to computing the closed form gradient of a Least-squares loss in a linear model (chapter 5 of MML book), doing the same for logistic regression is not possible. Gradient descent can be used to optimize such function $l$, and we will implement it step-by-step. First, write a function log_likelihood in the next cell that computes the log-likelihood given data points and $\vec{b}$. [5 pts]

In [ ]:
```python
import numpy as np
import sklearn
```

In [ ]:
```python
def log_likelihood(X,y,b):
    """
    X: n*m numpy data array.
    y: one dimension numpy data array of length n
    b: one dimension numpy data array of length m+1

    Return the log likelihood.
    """

    """ Add a column of ones to the X matrix."""
    ones_column = np.ones((X.shape[0], 1))
    new_X = np.append(ones_column, X, axis=1)

    """ Calculate the dot product of the new_X matrix and the b vector."""
    dot_product = np.dot(new_X, b)

    """ Calculate the log likelihood of the logistic regression model."""
    p_1 = 0
    p_0 = 0
    for i in range(len(y)):
        p_y = 1 / (1 + math.exp(-dot_product[i]))
        if y[i] == 1:
            p_1 += math.log(p_y)
        else:
            p_0 += math.log(1 - p_y)
    log_likelihood = p_1 + p_0
    return log_likelihood
```

## Test your log_likelihood function with a small example below.

In [ ]:
```
X=np.array([[0.1],[0.5],[1.]])
y=np.array([0,0,1])
b=np.array([0.,1.])

# Your answer should be around -2.03
log_likelihood(X,y,b)
```

Out[ ]:
```
-2.031735331771901
```

In [ ]:
```
```

Now that we have a function to maximize, the next step is to compute the gradient of the log-likelihood with respect to parameter $\vec{b}$. Use the method with Newton's difference quotient, and set $h = 0.0001$. Implement the function compute_gradient in the next cell. [7 pts]

In [ ]:
```
def compute_gradient(X,y,b):
# The inputs are the same as the ones of log_likelihood
    h = 0.0001
    gradient = np.zeros(b.shape)
    for i in range(len(b)):
        new_b = b.copy()
        new_b[i] += h
        each_gradient = (log_likelihood(X, y, new_b) - log_likelihood(X,y,b)
        gradient[i] = each_gradient

    return gradient
```

In [ ]:
```
# Test your function here, preserve the output
X=np.array([[0.1],[0.5],[1.]])
y=np.array([0,0,1])
b=np.array([0.,1.])
compute_gradient(X,y,b)
```

Out[ ]:
```
array([-0.87853115, -0.09479906])
```

Once we know how to compute the gradients, we can optimize the objective, which is log-likelihood in our case, using gradient descent. It iteratively changes the parameters in a small "step" towards the gradient direction, i.e., the direction where the objective increases at the fastest pace. Formally, denote the calculated gradients as $\Delta(\vec{b})$, we can update our parameters via $\vec{b} = \vec{b} + \gamma \cdot \Delta(\vec{b})$, where $\gamma$ is the size of the "step". Repeat this process until the objective stop improving or a pre-set max number of iterations is reached. **Note in practice, the value of gradient changes over iterations and can be very large/small, so you should normalize the gradient vector every iteration, i.e., scale it to $\frac{\Delta(\vec{b})}{||\Delta(\vec{b})||_2}$, before using it to compute the new $\vec{b}$. Therefore, the update rule for parameters becomes $\vec{b} = \vec{b} + \gamma \cdot \frac{\Delta(\vec{b})}{||\Delta(\vec{b})||_2}$.**

Implement the gradient_descent function below. [7 pts]

```
In [ ]: def gradient_descent(X, y, initial_b, step_size, max_iteration):
            """
            X: n*m numpy data array.
            y: one dimension numpy data array of length n
            initial_b: one dimension numpy data array of length m+1
            step_size: scalar, the size of one step update
            max_iteration: scalar, the max number of iterations
            Return the updated coefficient vector b.
            """
            b = initial_b
            for i in range(max_iteration):
                gradient = compute_gradient(X, y, b)
                squared_gradient = np.square(gradient)
                add_squared_gradient = 0
                for i in range(squared_gradient.shape[0]):
                    add_squared_gradient += squared_gradient[i]
                l2_norm = np.sqrt(add_squared_gradient)
                scale = np.divide(gradient, l2_norm)
                b = b + step_size * scale

            result_b = b

            return result_b
```

Test the function with the previous example again. Print for each sample from X, based on your model, the probability of having label=1.

```
In [ ]: optimized_b = gradient_descent(X, y, b, 0.1, 1000)

        # compute and print the probability for each row in X below using optimized_
        print(optimized_b)

        """ Add a column of ones to the X matrix."""
        probabilities = []
        ones_column = np.ones((X.shape[0], 1))
        new_X = np.append(ones_column, X, axis=1)

        """ Calculate the dot product of the new_X matrix and the b vector."""
        dot_product = np.dot(new_X, b)

        """ Calculate the log likelihood of the logistic regression model."""
        for i in range(len(X)):
```

```
    p_y = 1 / (1 + math.exp(-dot_product[i]))
    probabilities.append(p_y)

print(probabilities)
```

```
[-60.22501209  80.46040454]
[0.52497918747894, 0.6224593312018546, 0.7310585786300049]
```

Next, we apply the implemented logistic regression model to a real dataset. The dataset is a trimmed breast-cancer-Wisconsin dataset from UCI machine learning Repository. Only 100 data points are offered in the training set to make sure the computation can be finished swiftly, no matter how you implement the optimizer. The training dataset is loaded in the next cell, and the vector $\vec{b}$ is also randomly initialized.

Fit three models with the training set using different step size ranging in [0.01,0.05,0.1] and set the max number of iterations as 10000. How do the final log-likelihood value and the number of iterations change with different step sizes?

In [ ]:
```
f = open("breast-cancer-wisconsin.data","r")
X_train = []
y_train = []
for line in f:
    tmp = []
    for part in line.strip().split(",")[1:-1]:
        tmp.append(float(part))
    y_train.append((0 if line.strip().split(",")[-1]=="2" else 1))
    X_train.append(tmp)
X_train = np.array(X_train)
y_train = np.array(y_train)
random_b = np.random.uniform(0,1,size=(10))
```

In [ ]:
```
# Fit three models with different step size, report the final log-likelihooc
# number of iterations and the final coefficent vector b.
""" Step size h = [0.01, 0.05, 0.1] """
h = [0.01, 0.05, 0.1]
iteration = 10000
optimized_b_array = []

for i in range(len(h)):
    optimized_b = gradient_descent(X_train, y_train, random_b, h[i], iterati
    optimized_b_array.append(optimized_b)
    print("Step size: ", h[i])
    print("Final log-likelihood: ", log_likelihood(X_train, y_train, optimiz
    print("Number of iterations: ", iteration)
    print("Final coefficient of 0.01 vector b: ", optimized_b)
    # print("optimized_b arrary: ", optimized_b_array)
    print("\n")

print(optimized_b_array)
```

```
Step size:  0.01
Final log-likelihood:  -7.188013503615361
Number of iterations:  10000
Final coefficient of 0.01 vector b:  [-17.19374863   1.11091861  -1.83273667
1.18301251   1.08353328
   0.60951482  -0.06676765   1.04661244   1.27995542   1.75562097]


Step size:  0.05
Final log-likelihood:  -7.273867083649284
Number of iterations:  10000
Final coefficient of 0.01 vector b:  [-17.49186509   1.14053887  -1.86847903
1.21944908   1.09989347
   0.63481186  -0.06482084   1.07345198   1.31088433   1.79536519]


Step size:  0.1
Final log-likelihood:  -7.62805476018813
Number of iterations:  10000
Final coefficient of 0.01 vector b:  [-17.96776344   1.18431986  -1.91050999
1.26753129   1.12603921
   0.67308974  -0.05984781   1.10993465   1.35379181   1.85275011]


[array([-17.19374863,   1.11091861,  -1.83273667,   1.18301251,
         1.08353328,   0.60951482,  -0.06676765,   1.04661244,
         1.27995542,   1.75562097]), array([-17.49186509,   1.14053887,  -1.
86847903,   1.21944908,
         1.09989347,   0.63481186,  -0.06482084,   1.07345198,
         1.31088433,   1.79536519]), array([-17.96776344,   1.18431986,  -1.
91050999,   1.26753129,
         1.12603921,   0.67308974,  -0.05984781,   1.10993465,
         1.35379181,   1.85275011])]
```

Finally, load the test dataset, and predict for each sample in the test set what labels it should have using the model obtained. Compare your results with the ground truth labels, and report the accuracy rate.

```
In [ ]: f = open("test_data.txt","r")
        X_test = []
        y_test = []
        for line in f:
            tmp = []
            for part in line.strip().split(",")[1:-1]:
                tmp.append(float(part))
            y_test.append((0 if line.strip().split(",")[-1]=="2" else 1))
            X_test.append(tmp)
```

```
In [ ]: # Predict based on your models and report the accuracy

        """ Predict the test data using the optimized_b from the three models"""

        ''' --------------------------------------------------------------------'''
        ''' Calculate the accuracy of the model'''
        def accuracy(y_true, y_predict):
```

```
    length = len(y_true)
    result = 0

    for i in range(len(y_true)):
        if y_true[i] == y_predict[i]:
            result += 1
    return result / length

""" ---------------------------------------------------------------------"""
ones_column = np.ones((len(X_test), 1))                                 # Mak
new_X = np.append(ones_column, X_test, axis=1)                           # Ac

""" Calculate the dot product of the new_X matrix and the b vector."""


''' ---------------------------------------------------------------------'''
for i in range(len(optimized_b_array)):
    probabilities = []
    dot_product = np.dot(new_X, optimized_b_array[i])                    # Ca

    for j in range(len(X_test)):
        p_y = 1 / (1 + math.exp(-dot_product[j]))
        if p_y > 0.5:
            probabilities.append(1)
        else:
            probabilities.append(0)

    print("Step size: ", h[i])
    print("%d Optimized b : " %(i+1) , optimized_b_array[i])
    print(y_test)
    print("probabilities list : ", probabilities)
    print("Accuracy: ", accuracy(y_test, probabilities))
    print("\n")
```

```
Step size:  0.01
1 Optimized b :  [-17.19374863   1.11091861  -1.83273667   1.18301251   1.08
353328
   0.60951482  -0.06676765   1.04661244   1.27995542   1.75562097]
[0, 1, 1, 1, 1, 1, 0, 1, 0, 1]
probabilities list :  [0, 1, 1, 1, 1, 1, 0, 1, 0, 0]
Accuracy:  0.9


Step size:  0.05
2 Optimized b :  [-17.49186509   1.14053887  -1.86847903   1.21944908   1.09
989347
   0.63481186  -0.06482084   1.07345198   1.31088433   1.79536519]
[0, 1, 1, 1, 1, 1, 0, 1, 0, 1]
probabilities list :  [0, 1, 1, 1, 1, 1, 0, 1, 0, 0]
Accuracy:  0.9


Step size:  0.1
3 Optimized b :  [-17.96776344   1.18431986  -1.91050999   1.26753129   1.12
603921
   0.67308974  -0.05984781   1.10993465   1.35379181   1.85275011]
[0, 1, 1, 1, 1, 1, 0, 1, 0, 1]
probabilities list :  [0, 1, 1, 1, 1, 1, 0, 1, 0, 0]
Accuracy:  0.9
```

```
In [ ]:
```