

Enter Your Student ID:

Your Name:

Deadline:

Submit: Write your answers in this file, and submit a single Jupyter Notebook file (.ipynb) on Wattle. Rename this file with your student number as 'uXXXXXXX.ipynb'. Note: you don't need to submit the .png or .npy files.

Enter Discussion Partner IDs Below: You could add more IDs with the same markdown format above.

Programming Section:

- Task 1.1 - 1.3: 30%
- Task 1.4: 40%
- Task 2: 30%

In []:

```
import time
import sys
! pip install numpy
import numpy as np
! pip install matplotlib
import matplotlib.pyplot as plt
import math
import os
from matplotlib.pyplot import imread
! pip install patchify
from patchify import patchify
np.random.seed(1)
```

Task1: Clustering and Bag of Visual Words

These programming exercises will focus on K-means clustering.

If you're unsure of how k-means works, read this very helpful and freely available online breakdown from Stanford's CS221 course; <https://stanford.edu/~cpiech/cs221/handouts/kmeans.html>

This assignment requires you to loosely interpret how k-means is a specific case of a more general algorithm named Expectation Maximisation. This is explained toward the end of the above article.

First, lets loading the dataset.

In []:

K-means is a special, simple case of the Expectation Maximisation (EM) algorithm.

This simplified EM (k-means), is divided into two steps.

The **E-Step**, where for every sample in your dataset you find which "centroid" that datapoint is closest to that sample, and record that information.

The **M-Step**, where you move each "centroid" to the center of the samples which were found to be closest to it in the **E-Step**.

Each centroid is simply an estimated mean of a cluster. If you have 1 centroid, then this centroid will become the mean of all your data.

Centroids are initially random values, and the k-means algorithm attempts to modify them so that each one represents the center of a cluster.

We have implemented a centroids initialization function.

In []:

```
def initialise_parameters(m, X):
    C = X[np.random.choice(X.shape[0], m)]
    return C
C = initialise_parameters(4, X)
print(C)
```

Now let's implement K-Means algorithm.

TASK 1.1: Create a function $E_step(C, X) = L$, where L is a matrix of the same dimension of the dataset X .

This function is the **E-Step** (or "assignment step") mentioned earlier.

HINT:

- <https://stanford.edu/~cpiech/cs221/handouts/kmeans.html>
- https://en.wikipedia.org/wiki/k-means_clustering#Standard_algorithm
- Each row of L is a centroid taken from C .

In []:

```
def E_step(C, X):
    # YOUR CODE HERE
    pass
L = E_step(C, X)
plt.scatter(L[:, 0], L[:, 1])
plt.show()
```

TASK 1.2: Create a function $M_step(C, X, L) = C'$ which returns C' modified so that each centroid in C' is placed in the middle of the samples assigned to it. This is the **M-Step**.

In other words, make each centroid in C' the average of all the samples which were found to be closest to it during the **E-step**. This is also called the "update step" for K-means.

HINT: https://docs.scipy.org/doc/numpy/reference/generated/numpy.array_equal.html

In []:

```
def M_step(C, X, L):
    # YOUR CODE HERE
    pass
print('Before:')
print(C)
print('\nAfter:')
new_C = M_step(C, X, L)
print(new_C)
```

TASK 1.3: Implement $kmeans(X, m, threshold) = C, L$ which takes a dataset X (of any dimension) and a scalar value m and a scalar $threshold$ as input. This function uses the 3 functions you wrote previously to:

- generate m centroids.
- iterate between the E and M steps until the difference of loss values between two iterations is less than the threshold to classify the m clusters.

...and then returns:

- C , the centers of the m clusters after convergence.
- L , the labels (centroid vectors) assigned to each sample in the dataset after convergence.

HINT: Using initialise_parameters to initial centroid


In []:

```
def kmeans(X, m, threshold):
    # YOUR CODE HERE
    pass

#CODE TO DISPLAY YOUR RESULTS. DO NOT MODIFY.
C_final, L_final = kmeans(X, 4, 1e-6)
print('Initial Parameters:')
print(C)
print('\nFinal Parameters:')
print(C_final)

def allocator(X, L, c):
    cluster = []
    for i in range(L.shape[0]):
        if np.array_equal(L[i, :], c):
            cluster.append(X[i, :])
    return np.array(cluster)

colours = ['r', 'g', 'b', 'y']
for i in range(4):
    cluster = allocator(X, L_final, C_final[i, :])
    plt.scatter(cluster[:, 0], cluster[:, 1], c=colours[i])
plt.show()
```

Your answer should like this, maybe with different colors: 

TASK 1.4: Implement Bag of Visual Words (BOVW) to perform pedestrian retrieval. See more information at:

https://en.wikipedia.org/wiki/Bag-of-words_model

https://en.wikipedia.org/wiki/Bag-of-words_model#computer_vision

First, let's understand the settings of datasets.

We provide you with 3 pedestrian image folders, 'train', 'gallery', and 'val_query'. There are 99 images in 'train' which are used to create a vocabulary through clustering. 'Gallery' contains 90 images which belong to 15 different pedestrians. If two images' file name have same first four digits, then these two images belong to same pedestrian. When we randomly select a query image from 'val_query', we aim to find the images from the 'gallery' that contain the same person as the query. Let's load the images in 'train' and visualise an example.

In []:

```
train_images = []
for file in os.listdir("./train"):
    if file.endswith(".jpg"):
        im = imread("./train/" + file)
        train_images.append(im)
assert im.shape == (128, 64, 3)
plt.imshow(train_images[0])
plt.show()
```

To generate the vocabulary, the first step is computing local image features. For simplicity, patches of size 8x8 are densely sampled, and we use these patches for local feature extraction. The sampling step is 8, so there is no overlapping between patches.

In []:

```
def patchify_images(image):
    return patchify(image, (8, 8, 3), step=8).reshape((-1, 8, 8, 3))

patches = patchify_images(train_images[0])
print(f'Before splitting, the image size is {train_images[0].shape}')
print(f'After splitting, the patches are {patches.shape}')
print(f'A patch is like:')
plt.imshow(patches[50])
plt.show()
```

Then you need to implement your feature extractor to compute the feature of an image patch. Complete the function

$compute_patch_feature(patch) = patch_feature$

Requirements:

- You are not allowed to import any other packages except Numpy and Scipy. (You are allowed to import packages in Numpy or Scipy, such as scipy.signal).
- You can find inspiration from internet. However, you have to code by yourself. Hint: color feature might be a useful option.
- Your implementation for feature extraction of a patch should be of reasonable speed, e.g., within a second. You will have 0 if your algorithm takes too much time to run.

In [5]:

```
def compute_patch_feature(patch):
    #YOUR CODE HERE
    pass

start = time.time()
print(f'The shape of the feature of a patch is {(compute_patch_feature(patches[50])).shape} in my implementation.')
end = time.time()
print(f'It takes {end-start} seconds to compute the feature for one image patch.')
```

Now, you can create the vocabulary from patch features in the 'train' folder. Complete the function

$create_vocabulary(train_images) = vocabulary$

HINT:

- Remember to call the functions you defined in Task 1.1, 1.2 and 1.3.
- You are free to decide the size of vocabulary as long as it can be generated within 2 minutes.
- It is NOT allowed to use 'gallery' or 'val_query' images for vocabulary training. You will have 0 for Task 1.4 if you use them.
- It is NOT allowed to use external images (e.g., those from the web), either.

In []:

```
def create_vocabulary(train_images):
    #YOUR CODE HERE
    pass

start = time.time()
vocabulary = create_vocabulary(train_images)
end = time.time()
print(f'The shape of my vocabulary is {vocabulary.shape}.')
print(f'It takes {end-start} seconds to generate the vocabulary.')
```

You have built the vocabulary successfully. You must know how to compute the feature representation of an image. Now, let's do a simple pedestrian retrieval task where we are going to pick a query image from 'val_query' and try to search for images in 'gallery' which contain the same person.

In []:

```
gallery_images = []
gallery_filenames = []
for file in os.listdir("./gallery"):
    if file.endswith(".jpg"):
        im = imread("./gallery/" + file)
        gallery_images.append(im)
        gallery_filenames.append(file)

query_image = imread("./val_query/0001_c5_0022.jpg")
# show a query image
plt.imshow(query_image)
plt.show()
```

Complete the function

$image_similarity_ranking(gallery_images, query_image, vocabulary, gallery_filenames) = list_of_name_of_the_gallery_images$

gallery_images is the collection of all 90 images with dimension $90 \times 128 \times 64 \times 3$.

query_image is one image with dimension $128 \times 64 \times 3$.

The return value should be a list of image file names. Each name indicates a gallery image, ranked according to their similarities with the query. That is, the first file name corresponds to the image with the highest similarity to the query. The second file name in the list is image with second highest similarity to the query, etc. A file name should be '0001_c5_0022.jpg'. The length of the result should be 90, same as the number of gallery images. **For Task 1.4, you will be marked based on the retrieval accuracy.**

Requirements:

- You can't use the filename to do the task trivially. In other words, you must use a machine learning solution.
- You are free to improve the distance metric (an ℓ_2 distance is a basic option).
- You need to extract the feature of an image by the BOVW method. Other options (e.g., deep learning) are not allowed.
- Your implementation (e.g., BOVW feature extraction for an image, nearest neighbor search) should be of reasonable speed, e.g., within 20 seconds. You will receive 0 if your algorithm takes too much time to run (e.g., more than 2 minutes).

Marking criteria: Our evaluation process will calculate a matching score, a weighted sum of your top-1, 2, 3, 4, 5 accuracy using some test queries (not provided to students). These test queries will be used to probe your gallery data (90 images). Top-k accuracy measures the percentage of queries for which you could find the true match within the top-k position of the rank list. Your mark will be given based on your accuracy. For example, if your accuracy is within top 10% in the class, you will receive 40 marks; if your accuracy is 50% in the class, you will get 20 marks. If your program contains errors/bugs, you will receive 0.

In []:

```
def image_similarity_ranking(gallery_images, query_image, vocabulary, gallery_filenames):
    #YOUR CODE HERE
    pass

# Visualise your query image and its best match in gallery. Ideally, they should be the same person.
start = time.time()
name_list = image_similarity_ranking(gallery_images, imread("./val_query/0001_c5_0022.jpg"), vocabulary, gallery_filenames)
end = time.time()
print(f'It takes {end-start} seconds to get the matching results of a query')
print('Your query image is:')
plt.imshow(imread("./val_query/0001_c5_0022.jpg"))
plt.show()
print('The best matching is:')
plt.imshow(gallery_images[gallery_filenames.index(name_list[0])])
plt.show()
# We have 3 query images, you can try other two queries to see whether your algorithm performs well.
```

Please use the following code to calculate the matching score of 'val_query' dataset. Make sure the score is reasonable to you. We will be evaluating your implementation using some test queries that are not provided to you.

In []:

```
def match_score(name, name_list):
    def reid(idx):
        return name_list[idx][1:4]

    base = 0.0
    code = name[1:4]
    if reid(0) == code or reid(1) == code or reid(2) == code:
        base += 0.4
    if reid(0) == code:
        base += 0.3
    elif reid(1) == code:
        base += 0.2
    elif reid(2) == code:
        base += 0.1
    if reid(0) == code and reid(1) == code or reid(0) == code and reid(2) == code or reid(1) == code and reid(2) == code:
        base += 0.2
    if reid(0) == code and reid(1) == code and reid(2) == code:
        base += 0.1
    else:
        if reid(3) == code:
            base += 0.4
        elif reid(4) == code:
            base += 0.2
    return base

def total_score():
    score = 0
    for file in os.listdir("./val_query"):
        name_list = image_similarity_ranking(gallery_images, imread("./val_query/" + file), vocabulary, gallery_filenames)
        score += match_score(file, name_list)
    return score

print(total_score())
```

Task 2: Linear Regression and Gradient Descent

For exercise 2, we're going to implement multiple target batch linear regression with mean squared loss,

$$\mathcal{L} = \frac{1}{2m} \sum_{i=0}^m ||x_i \theta - y_i||^2$$

For the following questions:

- $x \in \mathbb{R}^m$ is the vector directly representing input features from the provided dataset. Every element of it is a single training example.
- $X \in \mathbb{R}^{m \times n}$ is the constructed feature matrix (e.g. polynomial features) used for learning. Each row of X is a single training example.
- θ is our parameters.
- $y \in \mathbb{R}^m$ is a matrix of the target values we're trying to estimate for each row of X . Each row i of X corresponds to row i of Y .
- m is the number of training examples.
- n is the dimensionality of one training example.

Typically when people think of linear regression, they think of a mapping from $\mathbb{R}^n \rightarrow \mathbb{R}$, where they're trying to predict a single scalar value.

First, we load the data.

In []:

```
x_train, _, y_train, _ = np.load("./data_regression.npy")
plt.plot(x_train, y_train, 'o')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Training data')
plt.ylim((-1,3))
plt.show()
```

It is obvious that it is not a good idea to perform linear regression directly on the input feature X . We need to add polynomial features. Lets construct an appropriate feature vector.

Task 2.1: Complete the `get_polynomial_features` function with the following specifications.

- Input1: an array X of shape $(m, 1)$.
- Input2: `degree` of the polynomial (integer greater than or equal to one).
- Output: matrix of shape $(m, degree + 1)$ consisting of horizontally concatenated polynomial terms.
- Output: the first column of output matrix should be all ones.

In []:

```
def get_polynomial_features(x, degree=5):
    # YOUR CODE HERE
    pass

# get polynomial features
X_train = get_polynomial_features(x_train, degree=2)
```

Let us implement gradient descent to find the optimal θ .

TASK 2.2: a function $initialise_parameters(n) = \theta$, where θ is the parameters we will use for linear regression $X\theta = Y$ for $X \in \mathbb{R}^{m \times n}$, $Y \in \mathbb{R}^m$.

The values of θ should be randomly generated. You will be judged on whether the matrix θ is correctly constructed for this problem.

HINT: θ should be an array of length n .

In []:

```
def initialise_parameters(n):
    # YOUR CODE HERE
    pass

# initialize theta
theta = initialise_parameters(X_train.shape[1])
print(theta)
```

TASK 2.3: Implement a function $ms_error(X, \theta, y) = err$, which gives the **mean squared error** over all m training examples.

In []:

```
def ms_error(X, theta, y):
    # YOUR CODE HERE
    pass

print(ms_error(X_train, theta, y_train))
```

TASK 2.4: Implement $grad(X, \theta, Y) = g$, a function that returns the average gradient $(\partial \mathcal{L} / \partial \theta)$ across all the training examples $x_i \in \mathbb{R}^{1 \times n}$.

HINT:

- The gradient should be an array with same length as θ .
- <https://www.sharpsightlabs.com/blog/numpy-sum/>
- <https://docs.scipy.org/doc/numpy/reference/generated/numpy.multiply.html>
- <https://docs.scipy.org/doc/numpy/reference/generated/numpy.tile.html>

In []:

```
def grad(X, theta, Y):
    # YOUR CODE HERE
    pass

print(grad(X_train, theta, y_train))
```

TASK 2.5: Implement $batch_descent(X, Y, iterations, learning_rate) = \theta, L$, a function which implements batch gradient descent returning θ (parameters which estimate Y from X), and L .

iterations is the number of gradient descent iterations to be performed.

learning_rate is, of course, the learning rate.

L is a matrix recording the mean squared error after every iteration of gradient descent. It will be an array of length *iterations*.

You should use the functions you completed earlier to complete this.

HINT:

- Remember, the point of gradient descent is to minimise the loss function.
- It does this by taking "steps". The gradient always points uphill, so by stepping in the opposite direction of the gradient we move toward the value of θ that minimises the loss function.

In []:

```
def batch_descent(X, Y, iterations, learning_rate):
    # YOUR CODE HERE
    pass

#REPORTING CODE. YOU MAY NEED TO MODIFY THE LEARNING RATE OR NUMBER OF ITERATIONS
new_theta, L = batch_descent(X_train, y_train, 5000, 0.5)
plt.plot(L)
plt.title('Mean Squared Error vs Iterations')
plt.show()
print('New theta: \n', new_theta)
print('\nFinal Mean Squared Error: \n', ms_error(X_train, new_theta, y_train))

def get_prediction(X, theta):
    pred = X@theta
    return pred

x_fit = np.linspace(-0.7, 0.8, 1000)
x_fit = get_polynomial_features(x_fit, degree=2)
pred_y_train = get_prediction(x_fit, new_theta)

# plot results
plt.plot(x_train, y_train, 'o', label='data point')
plt.plot(x_fit, pred_y_train, label='fitting result')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('show fitting result')
plt.ylim((-1,3))
plt.show()
```