1.In Lecture 4, we discussed feature selection and demonstrated how to use the selectKBest function from sklearn with f_regression and mutual_info_regression as scoring functions to select the 4 most important features from the "monthly_meteo_streamflow.csv" dataset.

a) Write a function to select the 4 most important features from the dataset using Pearson, Spearman, and Partial correlation. Note: For partial correlation between feature x1 and label y, condition on all other remaining features.

In [139…
```
# import libraries
import os
import sys
import numpy as np
import pandas as pd
import pingouin as pg
import matplotlib.pyplot as plt
from scipy.stats import pearsonr, spearmanr
from sklearn.feature_selection import mutual_info_regression
```

In [140…
```
# Load the dataset
```

In [141…
```
df = pd.read_csv('monthly_meteo_streamflow.csv')
df.head()
```

Out[141…

	year	month	P	E	Tmean	Tmax	Tmin	Tdmean	vpd
0	1980	1	336.051600	19.272449	4.62255	8.74525	0.50080	-3.86055	6.4(
1	1980	2	418.161899	29.872766	5.51400	10.94060	0.08835	-5.08145	8.78
2	1980	3	154.332850	58.054099	4.03085	9.76090	-1.69825	-5.70805	8.08
3	1980	4	19.936850	115.392255	8.77280	16.27545	1.27150	-5.28575	15.1(
4	1980	5	26.090900	116.888575	8.76150	15.23100	2.29315	0.08110	11.65

In [142…
```
x1 = df.iloc[:,2:-1]
x1
```

Out[142…

	P	E	Tmean	Tmax	Tmin	Tdmean	vpdmax
0	336.051600	19.272449	4.622550	8.745250	0.500800	-3.860550	6.400100
1	418.161899	29.872766	5.514000	10.940600	0.088350	-5.081450	8.782650
2	154.332850	58.054099	4.030850	9.760900	-1.698250	-5.708050	8.087550
3	19.936850	115.392255	8.772800	16.275450	1.271500	-5.285750	15.164100
4	26.090900	116.888575	8.761500	15.231000	2.293150	0.081100	11.650450
...	...	...	...	...	...	...	...
499	3.187385	152.323303	21.948775	27.692679	16.205010	4.492985	29.707269
500	0.694000	112.327974	19.826580	25.139330	14.513980	1.894015	26.081899
501	34.804679	80.789519	10.903875	17.622540	4.185345	-2.362750	13.043130
502	0.000000	50.394858	11.795860	16.642759	6.949110	-6.681535	15.416895
503	232.752304	23.204832	4.356815	8.752690	-0.038955	-6.133355	8.025990

504 rows × 8 columns

In [153…

```
# Convert to a pandas Series
y = pd.Series(df.iloc[:,-1])
y
```

Out[153…

```
0       423.612903
1      1685.965517
2       609.161290
3       236.533333
4       174.290323
           ...
499       0.425161
500       0.536333
501       1.584839
502       2.848667
503     151.384194
Name: S, Length: 504, dtype: float64
```

b) Identify the 4 most important features for each of Pearson, Spearman, and Partial correlation. Are these features different across the methods? If so, why?

In [154…

```
# Pearson correlation
pearson_corr = x1.corrwith(y, method='pearson')

# Spearman correlation
```

```
spearman_corr = x1.corrwith(y, method='spearman')

# Select the top 4 most important features for each correlation method
pearson_4features = pearson_corr.abs().nlargest(4).index
spearman_4features = spearman_corr.abs().nlargest(4).index
```

In [155…
```
pearson_4features
```

Out[155…
```
Index(['P', 'vpdmax', 'Tmax', 'vpdmin'], dtype='object')
```

In [156…
```
spearman_4features
```

Out[156…
```
Index(['vpdmin', 'vpdmax', 'Tmin', 'Tmean'], dtype='object')
```

In [157…
```
print("Top 4 most important features for Pearson features:", pearson_4featur
print("Top 4 most important features for Spearman features:", spearman_4feat
```

```
Top 4 most important features for Pearson features: ['P', 'vpdmax', 'Tmax',
'vpdmin']
Top 4 most important features for Spearman features: ['vpdmin', 'vpdmax', 'T
min', 'Tmean']
```

In [158…
```
def partial_corr(x1, y):
    df_corr = x1.copy()
    df_corr['S'] = y
    partial_corrs = {}

    # compute Partial Correlation
    for feature in x1.columns:
        pcorr = pg.partial_corr(data = df_corr, x = feature, y='S', covar =
        partial_corrs[feature] = pcorr['r'].values[0]

    return pd.Series(partial_corrs)

partial_corr_scores = partial_corr(x1, y)

# Select the top 4 most important features for each correlation method
pearson_4features = pearson_corr.abs().nlargest(4).index
spearman_4features = spearman_corr.abs().nlargest(4).index
partial_corr_4features = partial_corr_scores.abs().nlargest(4).index

# Print the results
print("Top 4 most important features based on Pearson correlation:", pearsor
print("Top 4 most important features based on Spearman correlation:", spearm
print("Top 4 most important features based on Partial correlation:", partial
```

```
Top 4 most important features based on Pearson correlation: ['P', 'vpdmax',
'Tmax', 'vpdmin']
Top 4 most important features based on Spearman correlation: ['vpdmin', 'vpd
max', 'Tmin', 'Tmean']
Top 4 most important features based on Partial correlation: ['P', 'vpdmax',
'E', 'Tdmean']
```

All of methords selected different features. (1)Pearson correlation is used to measures the linear relationship between two variables and very sensitive to outliers and assumes the data is normally distributed. (2)Spearman correlation is used to measures the rank-order relationship between two variables and less sensitive to outliers than Pearson and can measure not just linear relationships. (3)Partial correlation is used to measures the relationship between two variables while controlling for other variables. Therefore, that is why we get different 4 features for each methords.

c) Consider whether cross-correlation and feature redundancy are issues that need to be addressed. If they are, which scoring function you implemented (in addition to those available in sklearn, such as f_regression and mutual_info_regression) is more suitable for handling feature redundancy? Explain why.

In [159…
```
# Calculate mutual information
mutual_scores = mutual_info_regression(x1, y)

# Create a series to get top 4 features
mutual_scores_series = pd.Series(mutual_scores, index = x1.columns)
mutual_4features = mutual_scores_series.nlargest(4).index

print("Top 4 most important for Mutual Information features:", mutual_4featu
```

```
Top 4 most important for Mutual Information features: ['vpdmin', 'Tmean', 'T
min', 'vpdmax']
```

If features are highly correlated, potentially causing redundancy. It's essential to check if features are highly correlated with each other. I thought the mutual information os more suitable for handing feature redundancy. This is because it can handle non-linear relationship and avoid redundancy.

2. Following the steps discussed in Lecture 5, implement the K-means algorithm from scratch.

a) Attach a copy of your K-means implementation code.

In [180…
```
# Import libraries
```

```
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from scipy.integrate import odeint
from IPython.display import Image
from statistics import mode
from scipy.stats import pearsonr, spearmanr
from sklearn.feature_selection import mutual_info_regression
import statsmodels.api as sm
import pingouin as pg
from sklearn.datasets import make_classification

# Suppress warnings
warnings.filterwarnings('ignore')

# Set number of decimals for np print options
np.set_printoptions(precision=3)
```

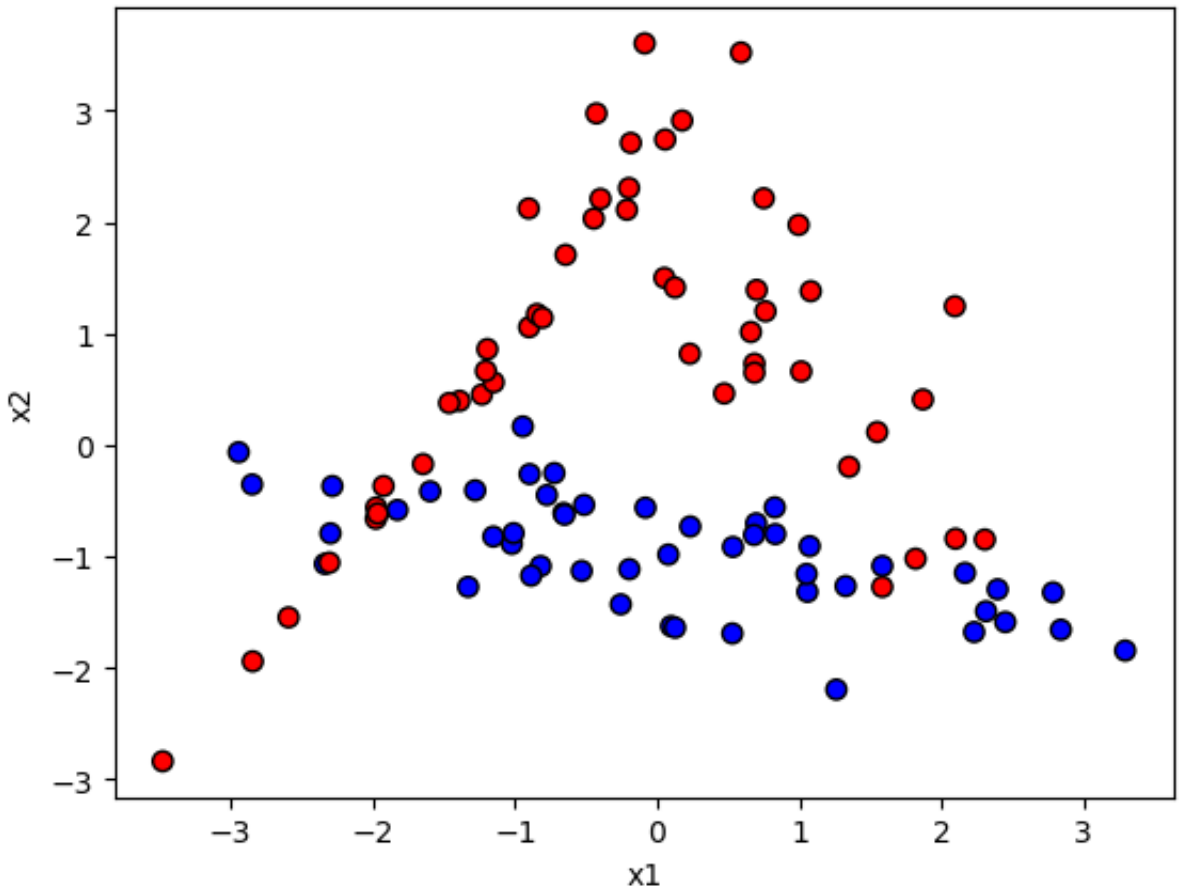
In [181…

```
X, y = make_classification(n_features=2, n_redundant=0, n_informative=2)
print(X.shape)
print(y.shape)
```

```
(100, 2)
(100,)
```

In [182…

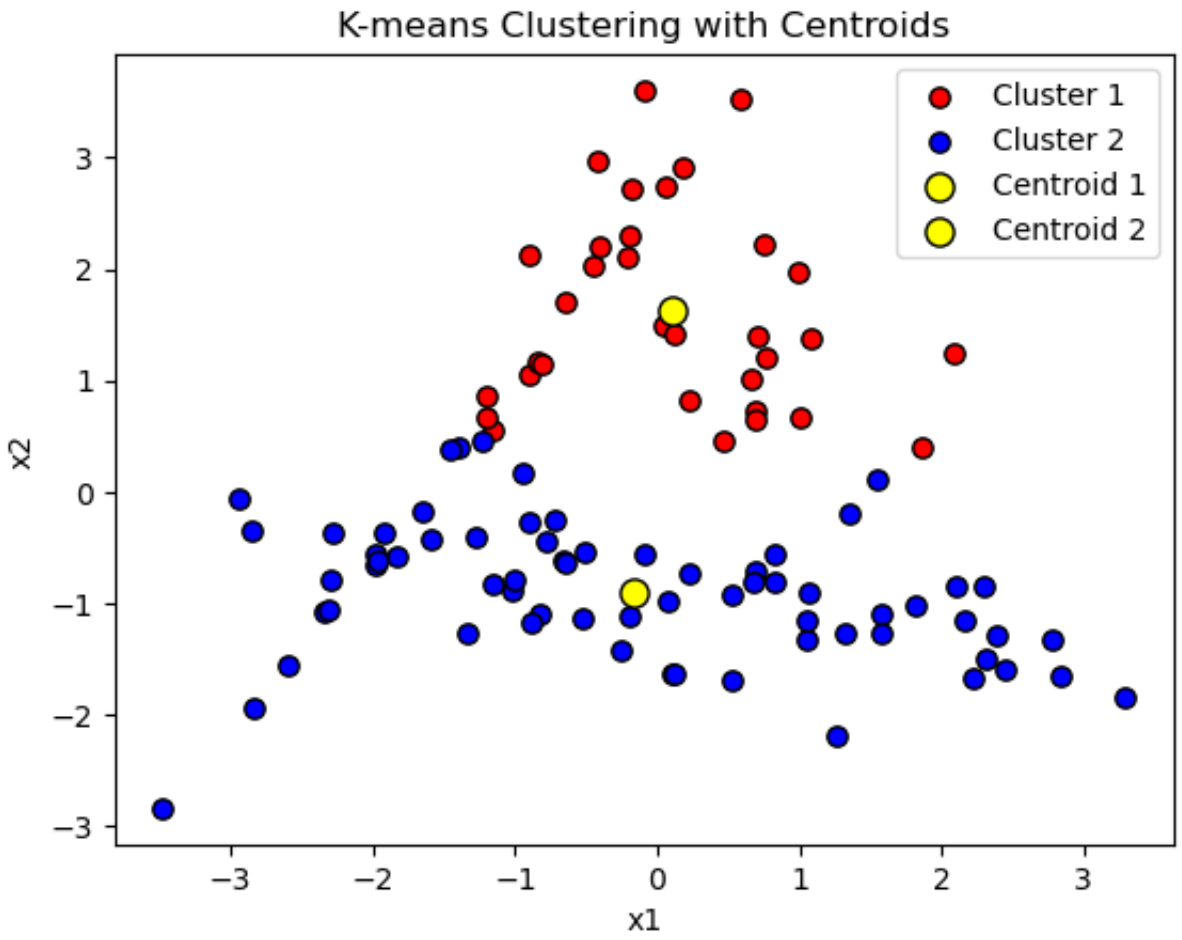
```
df = pd.DataFrame(np.column_stack((X,y)), columns = list(['x1', 'x2', 'y']))
# plot our df
color_map = {0: 'blue', 1: 'red'}
df['color'] = df['y'].map(color_map)
plt.scatter(df['x1'], df['x2'], marker = "o", c = df['color'], s = 45, edgec
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```



In [183…

```
# set the K value
k = 2
# randomly select K centroids
k1_x1 = np.random.uniform(low=np.min(df['x1']), high=np.max(df['x1']), size=
k1_x2 = np.random.uniform(low=np.min(df['x2']), high=np.max(df['x2']), size=

k2_x1 = np.random.uniform(low=np.min(df['x1']), high=np.max(df['x1']), size=
k2_x2 = np.random.uniform(low=np.min(df['x2']), high=np.max(df['x2']), size=
# calculate the distance
k1 = np.concatenate([k1_x1, k1_x2])
k2 = np.concatenate([k2_x1, k2_x2])
# copy dateframe
df_k_means = df.copy()

# create function to calculate Euclidean distance
def euclidean_distance(sample_point, centroid):
    return np.sqrt(np.sum((sample_point - centroid) ** 2))

# K-means algorithm
def kmeans_update(X, k1, k2, max_iters=100):
    for _ in range(max_iters):
        # Assign points to the nearest centroid
        cluster_1 = []
```

```
        cluster_2 = []

        for i in range(len(X)):
            dist_k1 = euclidean_distance(X[i], k1)
            dist_k2 = euclidean_distance(X[i], k2)

            if dist_k1 < dist_k2:
                cluster_1.append(X[i])
            else:
                cluster_2.append(X[i])

        # Convert to numpy arrays
        cluster_1 = np.array(cluster_1)
        cluster_2 = np.array(cluster_2)

        # Update centroids by calculating the mean of each cluster
        renew_k1 = np.mean(cluster_1, axis=0)
        renew_k2 = np.mean(cluster_2, axis=0)

        # Check if centroids do not change
        if np.all(k1 == renew_k1) and np.all(k2 == renew_k2):
            break

        k1, k2 = renew_k1, renew_k2

    return k1, k2, cluster_1, cluster_2

# the data points
X = df[['x1', 'x2']].values

# Run the K-means algorithm
last_k1, last_k2, cluster_1, cluster_2 = kmeans_update(X, k1, k2)

# Plot the final clusters and centroids
plt.scatter(cluster_1[:, 0], cluster_1[:, 1], c = 'red', s=45, edgecolor = "
plt.scatter(cluster_2[:, 0], cluster_2[:, 1], c = 'blue', s=45, edgecolor="k
plt.scatter(last_k1[0], last_k1[1], c = 'yellow', s = 90, edgecolor = "k", m
plt.scatter(last_k2[0], last_k2[1], c = 'yellow', s = 90, edgecolor = "k", m
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('K-means Clustering with Centroids')
plt.show()
```



b) Use your K-means implementation to cluster the synthetic data provided in clustering_example.csv. This dataset has two features (x1 and x2) and one label (y), with two possible classes for the labels: {0, 1}.

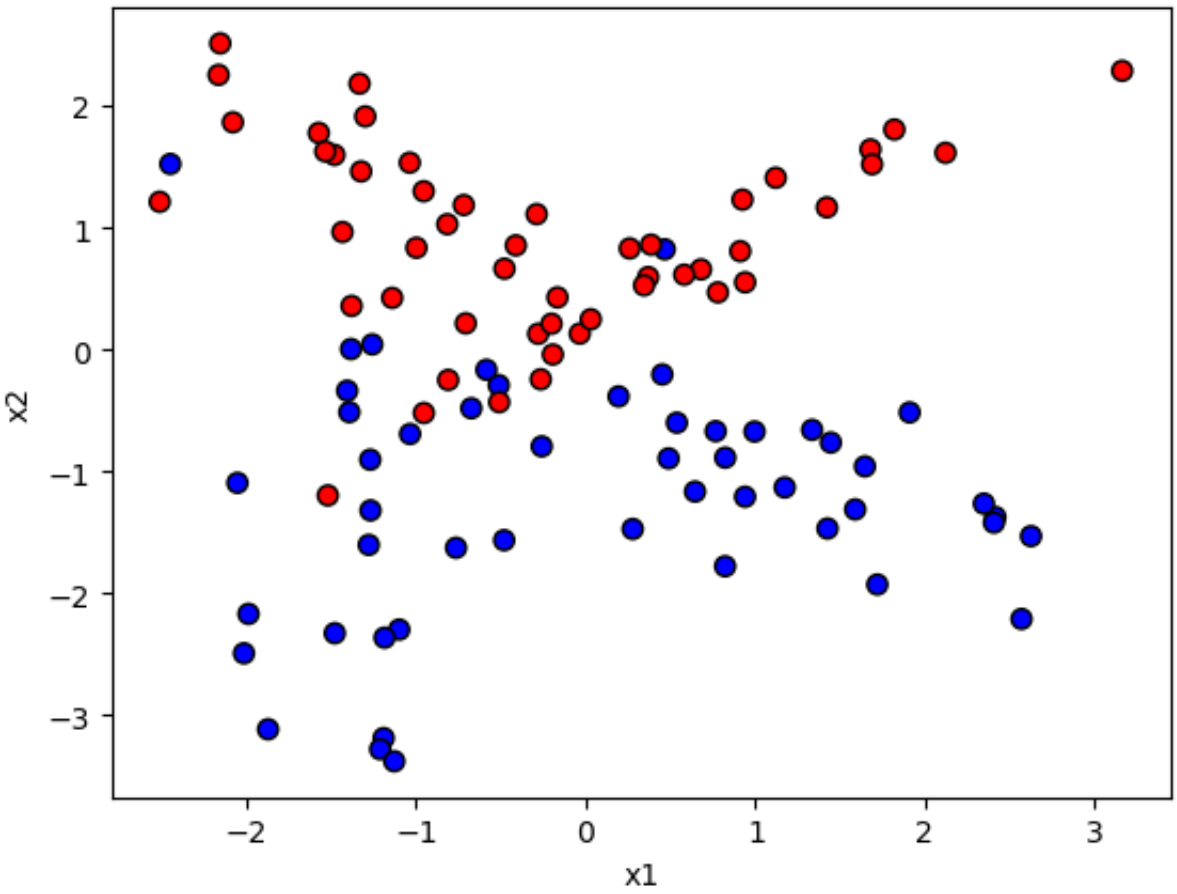
In [184…
```
# Load the dataset
dt = pd.read_csv('clustering_example.csv')
dt.head()
```

Out[184…

	x1	x2	y
0	-0.512817	-0.293085	0
1	2.414166	-1.375149	0
2	1.718505	-1.929213	0
3	-2.154521	2.511907	1
4	0.679351	0.657054	1

In [185…
```
# Select features from dataframe
X = dt[['x1', 'x2']].values
```

```
# Plot our dataframe
color_map = {0: 'blue', 1: 'red'}
dt['color'] = dt['y'].map(color_map)
plt.scatter(dt['x1'], dt['x2'], marker = "o", c = dt['color'], s = 45, edgec
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

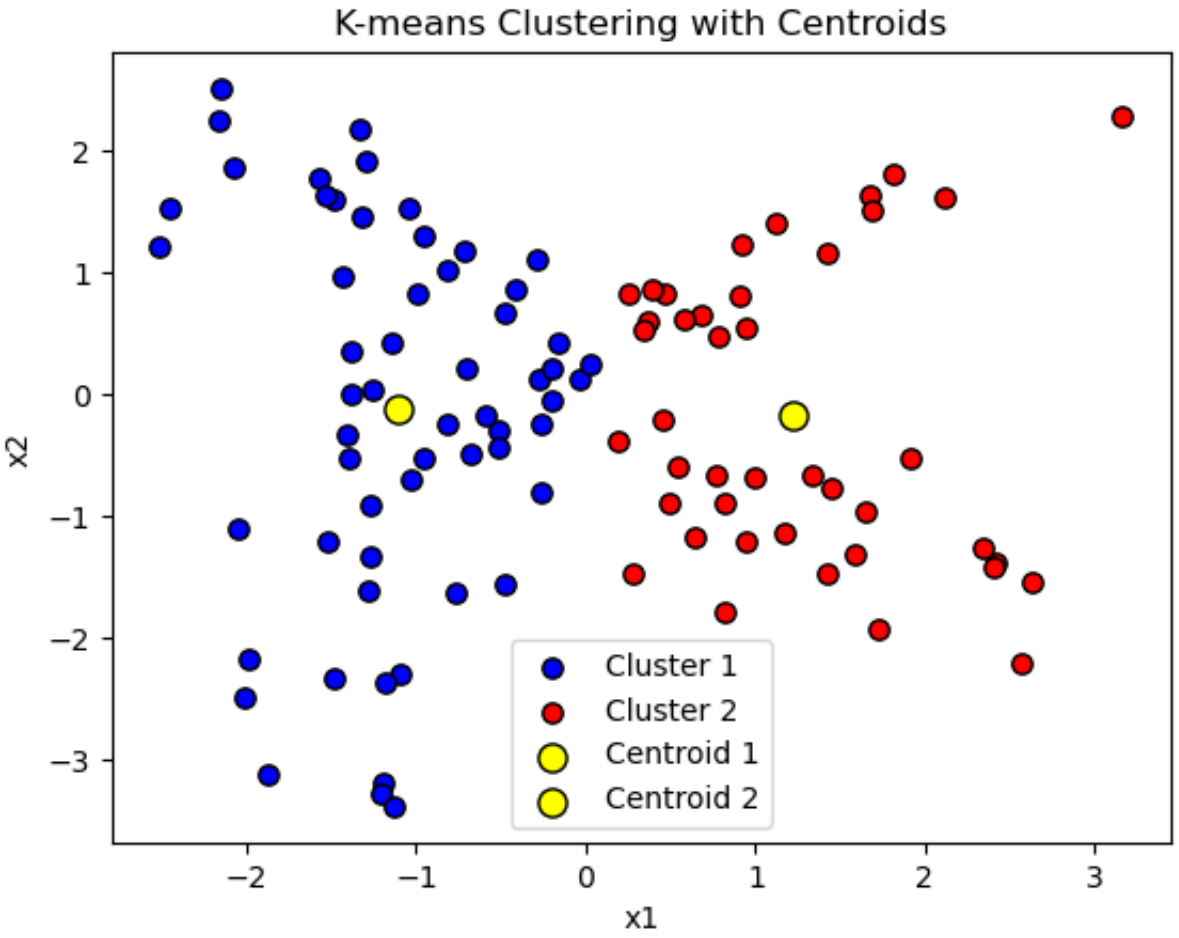


In [186…

```
# set the K value
k = 2
# randomly select K centroids
k1_x1 = np.random.uniform(low=np.min(dt['x1']), high=np.max(dt['x1']), size=
k1_x2 = np.random.uniform(low=np.min(dt['x2']), high=np.max(dt['x2']), size=

k2_x1 = np.random.uniform(low=np.min(dt['x1']), high=np.max(dt['x1']), size=
k2_x2 = np.random.uniform(low=np.min(dt['x2']), high=np.max(dt['x2']), size=
# calculate the distance
k1 = np.concatenate([k1_x1, k1_x2])
k2 = np.concatenate([k2_x1, k2_x2])
# copy dateframe
dt_k_means = dt.copy()

# create function to calculate Euclidean distance
def euclidean_distance(sample_point, centroid):
```

```
    return np.sqrt(np.sum((sample_point - centroid) ** 2))

# K-means algorithm
def kmeans_update(X, k1, k2, max_iters=100):
    for _ in range(max_iters):
        # Assign points to the nearest centroid
        cluster_1 = []
        cluster_2 = []

        for i in range(len(X)):
            dist_k1 = euclidean_distance(X[i], k1)
            dist_k2 = euclidean_distance(X[i], k2)

            if dist_k1 < dist_k2:
                cluster_1.append(X[i])
            else:
                cluster_2.append(X[i])

        # Convert to numpy arrays
        cluster_1 = np.array(cluster_1)
        cluster_2 = np.array(cluster_2)

        # Update centroids by calculating the mean of each cluster
        renew_k1 = np.mean(cluster_1, axis=0)
        renew_k2 = np.mean(cluster_2, axis=0)

        # Check if centroids do not change
        if np.all(k1 == renew_k1) and np.all(k2 == renew_k2):
            break

        k1, k2 = renew_k1, renew_k2

    return k1, k2, cluster_1, cluster_2

# the data points
X = dt[['x1', 'x2']].values

# Run the K-means algorithm
last_k1, last_k2, cluster_1, cluster_2 = kmeans_update(X, k1, k2)

# Plot the final clusters and centroids
plt.scatter(cluster_1[:, 0], cluster_1[:, 1], c = 'blue', s=45, edgecolor =
plt.scatter(cluster_2[:, 0], cluster_2[:, 1], c = 'red', s=45, edgecolor = "
plt.scatter(last_k1[0], last_k1[1], c = 'yellow', s = 90, edgecolor = "k", m
plt.scatter(last_k2[0], last_k2[1], c = 'yellow', s = 90, edgecolor = "k", m
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('K-means Clustering with Centroids')
plt.show()
```

```
#plt.scatter(df['x1'], df['x2'], marker = "o", c = 'white', s = 45, edgecolo
#plt.scatter(k1[0], k1[1], marker = "o", c ='yellow', s = 90, edgecolor = "k
#plt.scatter(k2[0], k2[1], marker = "o", c = 'yellow', s = 90, edgecolor = "
#plt.xlabel('x1')
#plt.ylabel('x2')
#plt.show()
```



c) Visually compare the true clusters with those obtained from your K-means implementation. What are your main observations?

In [187…

```
dt
```

Out[187…

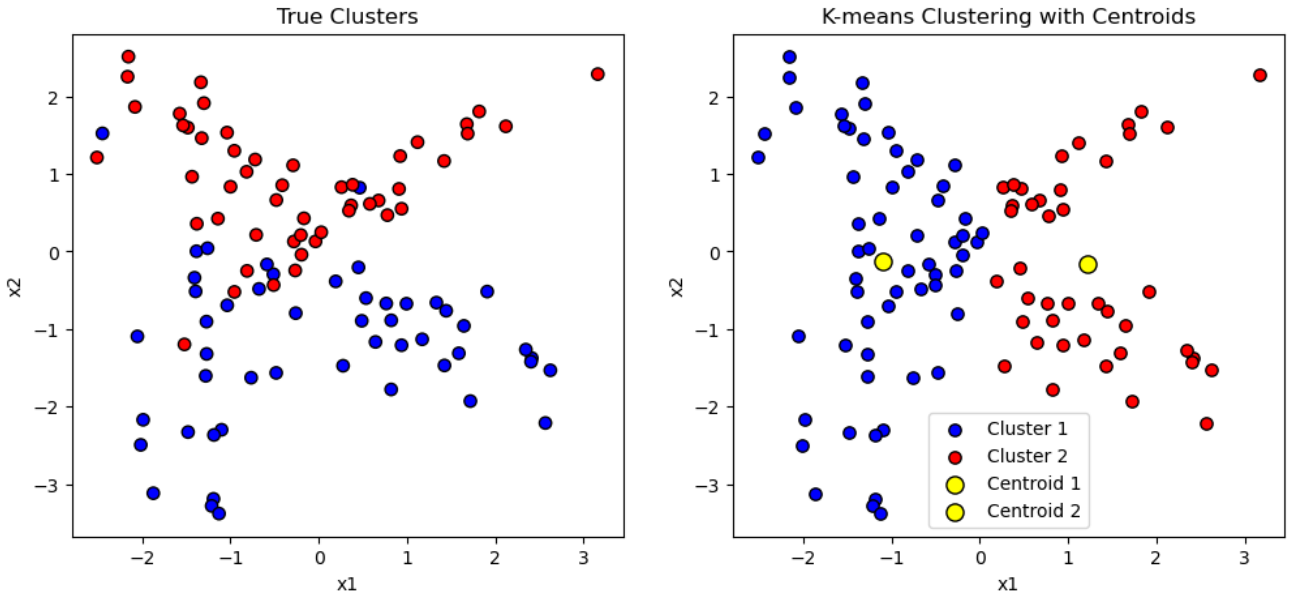
	x1	x2	y	color
0	-0.512817	-0.293085	0	blue
1	2.414166	-1.375149	0	blue
2	1.718505	-1.929213	0	blue
3	-2.154521	2.511907	1	red
4	0.679351	0.657054	1	red
...	...	...	...	...
95	-1.128670	-3.379703	0	blue
96	-0.674691	-0.482545	0	blue
97	-1.258426	0.041474	0	blue
98	2.120491	1.613728	1	red
99	-2.164145	2.253380	1	red

100 rows × 4 columns

In [188…

```
# set the plot size
plt.figure(figsize = (12, 5))

# Plot the true clusters
plt.subplot(1, 2, 1)
color_map = {0: 'blue', 1: 'red'}
dt['color'] = dt['y'].map(color_map)
plt.scatter(dt['x1'], dt['x2'], c = dt['color'], s=45,
            edgecolor = "k", label="True Clusters")
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('True Clusters')

# Plot K-means clusters
plt.subplot(1, 2, 2)
plt.scatter(cluster_1[:, 0], cluster_1[:, 1], c = 'blue', s=45,
            edgecolor = "k", label = "Cluster 1")
plt.scatter(cluster_2[:, 0], cluster_2[:, 1], c = 'red', s=45,
            edgecolor = "k", label = "Cluster 2")
plt.scatter(last_k1[0], last_k1[1], c = 'yellow', s = 90,
            edgecolor = "k", marker = "o", label = "Centroid 1")
plt.scatter(last_k2[0], last_k2[1], c = 'yellow', s = 90,
            edgecolor = "k", marker = "o", label = "Centroid 2")
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
```

```
plt.title('K-means Clustering with Centroids')
plt.show()
```



From my result, I thought K-Means assigns most points to correct cluster rather than true clusters. As the plot showes, the Truw clusters assgins a few points to wrong cluster. For this data, the K-means is more suitable, however, if the data is not approximately spherical, then the k-means classification is not very good. On the contrary, if the data is compact, true clusters may perform better.

d) Use the built-in K-means function in your preferred programming language to cluster the data from (b). For Python users, use sklearn.cluster.KMeans. Do the clusters obtained match the original clusters?

In [174…
```
# import library
from sklearn.cluster import KMeans
```

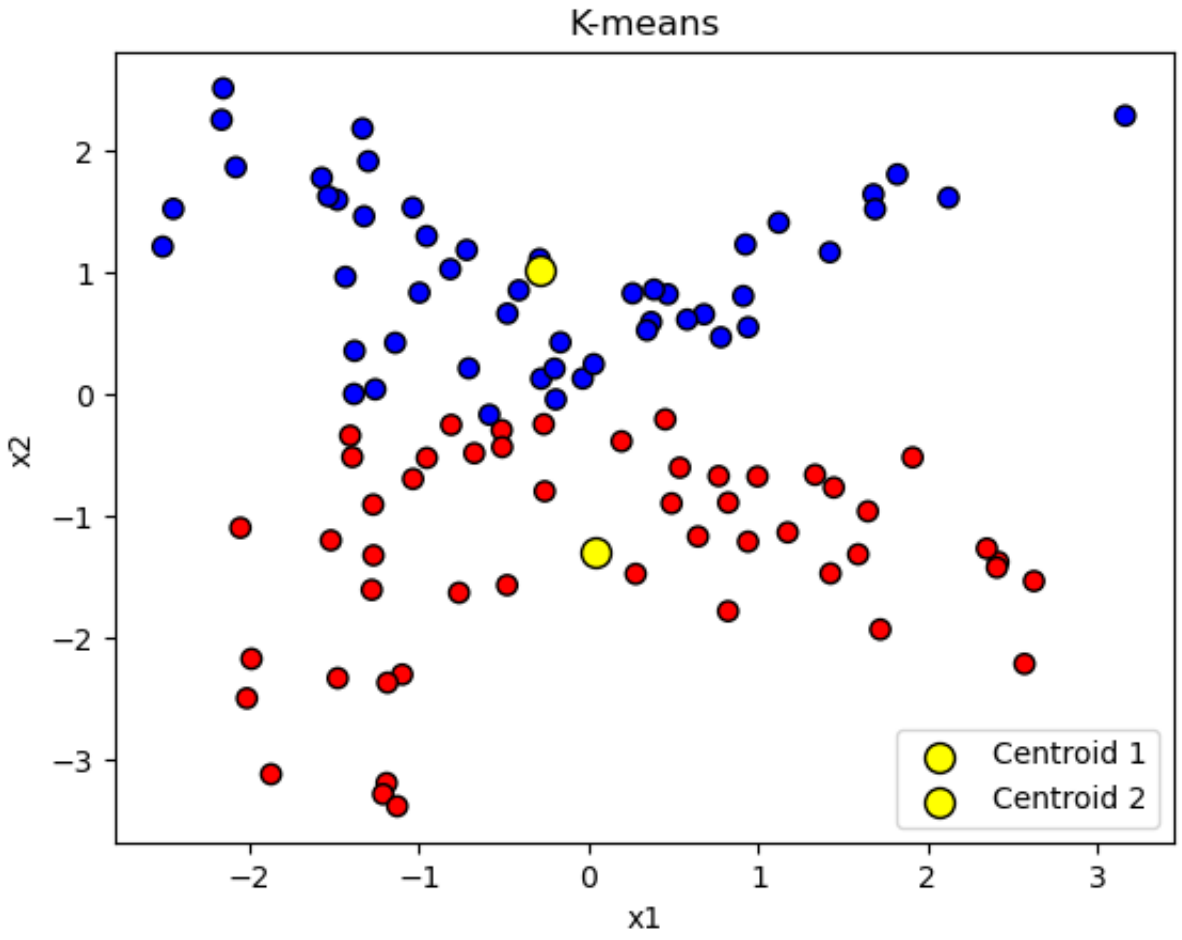
In [175…
```
# set the number of clusters
kmeans = KMeans(n_clusters= 2)

# Fit the model
kmeans.fit(X)

# Get cluster centers
centers = kmeans.cluster_centers_

# Get labels
labels = kmeans.labels_
```

In [176…
```
centers
```

Out[176…
```
array([[-0.294,  1.015],
       [ 0.038, -1.3  ]])
```

In [177…
```
df = pd.DataFrame(np.column_stack((X,labels)), columns= ['x1', 'x2', 'y'])

color_map = {0: 'blue', 1: 'red'}
df['colors'] = df['y'].map(color_map)

plt.scatter(df['x1'], df['x2'], marker="o", c=df['colors'], s=45, edgecolor=
plt.scatter(centers[0,0], centers[0,1], marker="o", c='yellow', s=100, edgec
plt.scatter(centers[1,0], centers[1,1], marker="o", c='yellow', s=100, edgec
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('K-means')
plt.show
print("Centroids from K-means:", centers)
```

```
Centroids from K-means: [[-0.294  1.015]
 [ 0.038 -1.3  ]]
```



The clustering results obtained using built-in k-means are a little different from those obtained by writing my own k-means. The main reason is that the location of the

centroid is shifted, which leads to different categories of several points on the boundary of the cluster.

3. The Köppen–Geiger (KG) climate classification system categorizes global land regions

based on temperature and wetness. This system is widely used in climate change studies and assessments of its impacts on human health and ecology. The KG classification includes five main groups: A (tropical), B (arid), C (temperate), D (continental), and E (polar).

a) The files prec.mat and temp.mat contain average monthly gridded data for precipitation and temperature across the contiguous United States (CONUS). Each file has dimensions of 444 (latitudes) × 922 (longitudes) × 12 (months). These .mat files can be read in Python, R, or MATLAB. Use sklearn.cluster.KMeans or any K-means function to cluster the grid cells into 5 clusters (k = 5), similar to the KG classification. Hint: you can read the .mat files in Python using this code snippet:

In [112…

```
# import library
from scipy.io import loadmat
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm
import cartopy.crs as ccrs
import cartopy.feature as cfeature
# load the data
temp_data = loadmat('temp')['temp']
prec_data = loadmat('prec')['prec']
temp_data
prec_data
```

Out[112…

```
array([[[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]],

       [[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]],

       [[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]],

       ...,

       [[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]],

       [[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]],

       [[nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        ...,
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan],
        [nan, nan, nan, ..., nan, nan, nan]]], dtype=float32)
```

In [113…
```
# get the number of NaN
print("NaNs in temperature data is", np.isnan(temp_data).sum())
print("NaNs in precipitation data is", np.isnan(prec_data).sum())
```

```
NaNs in temperature data is 2372172
NaNs in precipitation data is 2372172
```

In [114…
```
# get dimensions
lat, lon, months = prec_data.shape
# reshape the data
prec_reshaped = prec_data.reshape(lat * lon, months)
temp_reshaped = temp_data.reshape(lat * lon, months)

# combine the prec and temp data
X = np.concatenate([prec_reshaped, temp_reshaped], axis=1)

# remove any NaN values
X_clean = X[~np.isnan(X).any(axis=1)]
```

In [130…
```
X_clean.shape
```

Out[130… (211687, 24)

In [131…
```
prec_data.shape
```

Out[131… (444, 922, 12)

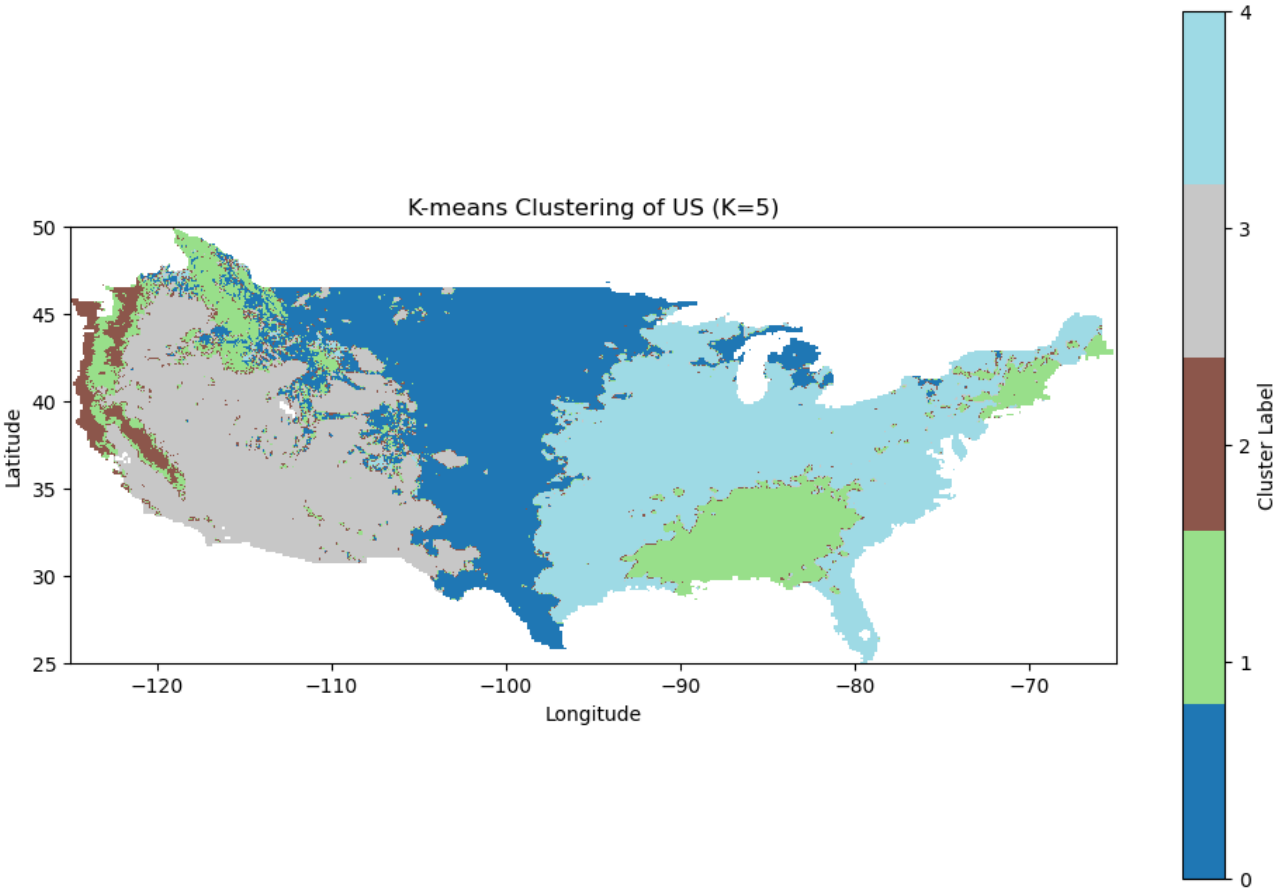
In [132…
```
444*922
```

Out[132… 409368

b) Plot a map of the final five clusters resulting from K-means, with each cluster represented in a different color. You can use matplotlib.pyplot.imshow or the cartopy library for plotting. What are your main observations? Does the map exhibit coherent spatial patterns?

In [161…
```
# use KMeans clustering
kmeans = KMeans(n_clusters=5, random_state=0)
# fit the model
clusters_clean = kmeans.fit_predict(X_clean)

# get grid clusters
clusters_grid = np.full((lat, lon), np.nan)

# get the indices of valid grid cells without NaN
valid_indices = np.where(~np.isnan(prec_reshaped).any(axis=1))[0]

# Fill the cluster grid with predicted cluster labels
```

```
clusters_grid.flat[valid_indices] = clusters_clean

# Reshape the flat cluster grid back into a 2D grid
clusters_grid_2d = clusters_grid.reshape(lat, lon)

# Fix the longitude and latitude
lon_range = np.linspace(-125, -65, lon)
lat_range = np.linspace(25, 50, lat)

# Set up the color mapping for clusters
cmap = plt.get_cmap('tab20', 5)  # 5 clusters
norm = BoundaryNorm(np.arange(-0.5, 5, 1), cmap.N)

# Plot the clusters
plt.subplots(figsize=(12, 8))
cax = plt.imshow(clusters_grid_2d, cmap=cmap, extent=[-125, -65, 25, 50], or

# Add colorbar
cbar = plt.colorbar(cax, ticks=np.arange(5))
cbar.set_label('Cluster Label')

# Set labels and title
plt.title('K-means Clustering of US (K=5)')
plt.xlabel('Longitude')
plt.ylabel('Latitude')

#ax = plt.axes(projection=ccrs.PlateCarree())
# Add features for coastline, borders, and land
#ax.add_feature(cfeature.COASTLINE)
#ax.add_feature(cfeature.BORDERS)
#ax.add_feature(cfeature.STATES)
# Plot the clusters
#cmap = plt.get_cmap('tab20', 5)  # Using a colormap with 5 distinct colors
#plt.contourf(lon_grid, lat_grid, clusters_grid, cmap=cmap, transform=ccrs.P
#plt.colorbar(ax=ax, orientation='vertical', label='Cluster Label')
#plt.title('K-means Climate Clusters (K=5) in US')
#plt.xlabel('Longitude')
#plt.ylabel('Latitude')

plt.show()
```



As the result shows, here is a K-means clustering of US, when k=5. There is a difference in temperature and rainfall between the eastern and western parts in US. The map shows a consistent spatial pattern. But I think if the season or other factors are considered it could be better to show K-Means and link to geographic location.

c) Compare the map from (b) with Köppen–Geiger classification maps available online for the contiguous United States. Summarize the similarities and differences.

Similarities:They both used clustering algorithms to analyze climate classification.

Differences: Comparing with Köppen–Geiger Climate Classification map, I can find there are different boundaries. Maybe this is because the K-means clusters just capture temperature and precipitation, While the Köppen–Geiger consider more complex ecological and seasonal factors.