ps3

October 22, 2023

1 CS541: Applied Machine Learning, Fall 2023, Problem Set 3

Problem set 3 is due in Gradescope on Oct 24 at 11:59pm. All the questions are in this jupyter notebook file. There are five questions in this assignment, each of which could have multiple parts and consists of a mix of coding and short answer questions. This assignment is worth a total of **125 points** (**80 pts** coding, and **45 pts** short answer). Note that each individual pset contributes the same amount to the final grade regardless of the number of points it is worth.

After completing these questions you will need to covert this notebook into a .py file named **ps3.py** and a pdf file named **ps3.pdf** in order to submit it (details below).

Submission instructions: please upload your completed solution files to Gradescope by the due date. **Make sure you have run all code cells and rendered all markdown/Latex without any errors before submitting.**

Note: For coding part, **remember to return the required variable**. Simply use `print()` at the end will return `None`, which will result in failing the test case.

There will be 2 separate submission links for the assignment, one to submit **ps3.py** file for autograder on the coding part, and the other one for **ps3.PDF** for manually grading on writing part. You can use Jupyter Notebook to convert the formats: + Convert to PDF file: Go to File->Download as->PDF + Convert py file: Go to File->Download as->py (quick reference guide here)

Submission Links + PDF (ps3.pdf) submission (45 pts): https://www.gradescope.com/courses/427800/assignments/2319846 + Python file (ps3.py) submission (80 pts): https://www.gradescope.com/courses/427800/assignments/2319843

Assignment Setup

You can use Google Colab for this assignment. It has been tested on Colab, so you should be able to run it on colab without any errors.

If you would prefer to setup your code locally on your own machine, you will need Jupyter Notebook or JupyterLab installation. One way to set it up is to install "Anaconda" distribution, which has Python, several libraries including the Jupyter Notebook that we will use in class. It is available for Windows, Linux, and Mac OS X here.

If you are not familiar with Jupyter Notebook, you can follow this blog for an introduction.

[1]:
```
## import some libraries
import sklearn
```

```
from sklearn.cluster import KMeans
from sklearn import datasets
import numpy as np
from typing import Tuple, List
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.patches import Ellipse
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import mean_squared_error
from scipy.spatial.distance import cdist
```

2 Question 1. GMM: Image segmentation (*30 total points*)

In the last problem set, we encounter some cases where k-means is not flexible enough to account for non-circular datasets. It may not perform well due to lack of flexibility in cluster shape and lack of probabilistic cluster assignment.

In this section, we will work on GMM and see how it can help to solve these problems.

Recall that with GMM, we start by placing guassians randomly, then we iterate over these two following steps until it converges.

- E step: Assign probability of each data point x_i coming from each guassian based on current means and variances.
- M step: Re-estimate the guassians' mean and variance to better fit the data points.

It's interesting that GMM of Sklearn uses *K-means* and *K-means++* (a *K-means*'s variant) for its first guess (i.e., to initialize the weights, the means and the precisions).

2.1 1.1 Code: Image segmentation *(20 pts)*

Image segmentation is an important application of clustering. One breaks an image into k segments, determined by color, texture, etc. These segments are obtained by clustering image pixels by some representation of the image around the pixel (color, texture, etc.) into k clusters. Then each pixel is assigned to the segment corresponding to its cluster center.

[2]:
```
## First, let's read an image of sunset over New York city (image from␣
 ↪Shutterstock)
from skimage import io  ## import skimage to read the image

raw_img = io.imread("https://raw.githubusercontent.com/chaudatascience/
 ↪cs599_fall2022/master/ps3/img0.jpg")
print("shape of the image:", raw_img.shape)  ### width, height, and␣
 ↪num_channels (e.g., RBG) of the image
print(f"there are {np.prod(raw_img.shape[:-1])} pixels in the image.")
plt.figure(figsize=(11.5, 17.3))
plt.imshow(raw_img)
```

```
shape of the image: (469, 707, 3)
there are 331583 pixels in the image.
```

[2]:
```
<matplotlib.image.AxesImage at 0x18ed27f50>
```

[69]:
```
## print out some values of `raw_img`
raw_img[:3]  ## each pixel consists of 3 numbers: R, G, B channels, ranging␣
 ↪from 0->255
# raw_img.shape
```

[69]:
```
array([[[ 66,  58, 105],
        [ 57,  49,  96],
        [ 55,  47,  94],
        …,
        [ 52,  49,  94],
        [ 52,  49,  94],
        [ 52,  49,  94]],

       [[ 66,  59, 101],
        [ 64,  57,  99],
        [ 62,  55,  99],
        …,
        [ 51,  50,  94],
```

```
        [ 51,  50,  94],
        [ 51,  50,  94]],

       [[ 61,  55,  91],
        [ 66,  60,  98],
        [ 65,  58,  99],
        …,
        [ 51,  50,  94],
        [ 51,  50,  94],
        [ 51,  50,  94]]], dtype=uint8)
```

The image can be considered as a dataset with 331,583 samples, each has 3 features (R, G, B).

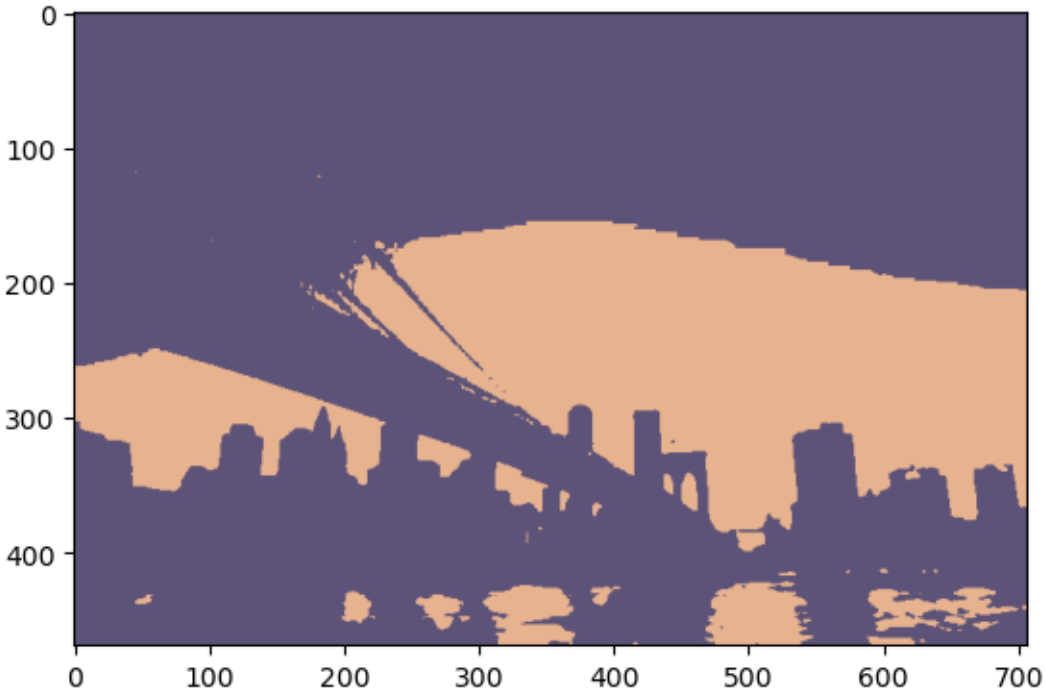
In this section, we'll cluster the pixels into 2, 3, 5, and 10 clusters, modelling the pixel values as a mixture of normal distributions and using EM. Then, we'll display the image obtained by replacing each pixel with the mean of its cluster center.

[162]:
```
from sklearn.mixture import GaussianMixture as GMM

def question_1_1(raw_img: np.ndarray, n_components: int, random_seed: int) ->␣
↪np.ndarray:
    """
        Cluster pixels into `n_components` cluster using Sklearn's GMM
        raw_img: numpy array, shape of (img_width, img_height, num_channels) ␣
↪(e.g., (469, 707, 3) )
        n_components: number of clusters for GMM
        random_seed: random state, passed to GMM when initializing.
        return the new image whose each pixel is replaced by the cluster␣
↪center, numpy array shape (img_width, img_height, num_channels)
    """
    # Write your code in this block␣
↪------------------------------------------------------------

    # shape of the raw_img:
    original_shape = raw_img.shape

    ## step 1: reshape the `raw_img` from 3d (img_width, img_height,␣
↪num_channels) to 2d (img_width*img_height, num_channels)

    reshaped_raw_img = np.reshape(raw_img, (original_shape[0] *␣
↪original_shape[1], original_shape[-1]))


    ## step 2: normalize the image from the previous step
    # We normalize each pixel's value from an int in [0, 255] to a float number␣
↪in range (0, 1) by:
    #     X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
```

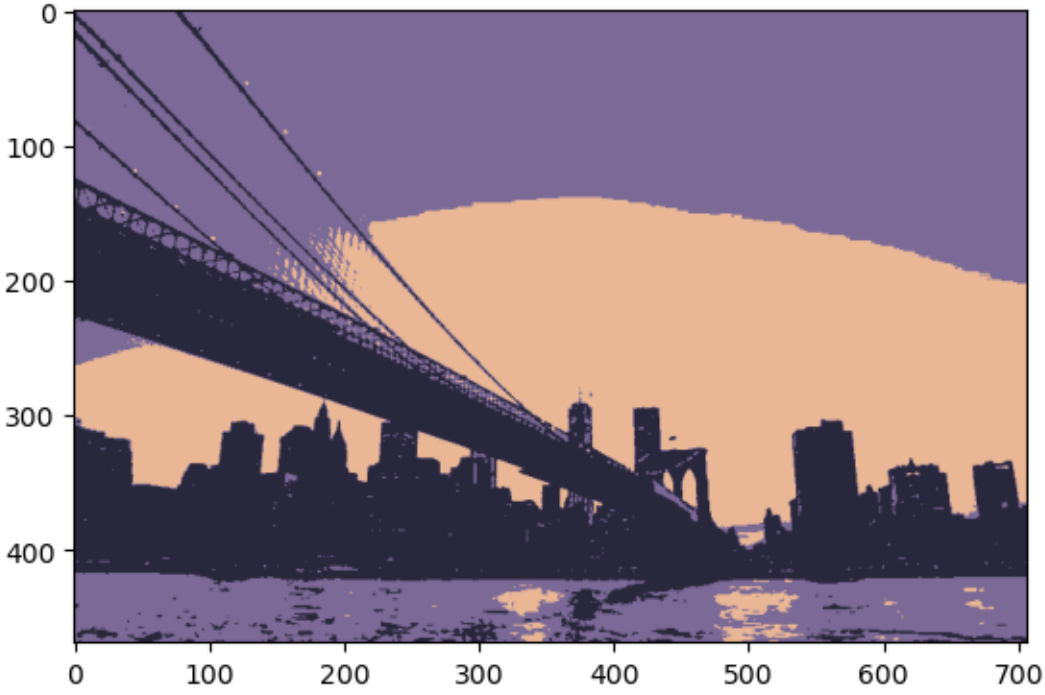
```
    #     X_scaled = X_std * (max - min) + min

    # you should use `MinMaxScaler` from sklearn for this task

    scaler = sklearn.preprocessing.MinMaxScaler()
    scaled_data = scaler.fit_transform(reshaped_raw_img)

    ## step 3: predict clusters using GMM
    # you can use `GaussianMixture` from `sklearn.mixture` to create a GMM␣
 ↪model.
    # When initalizing, set `max_iter` to 60, and `covariance_type` to "tied",
    # and `random_state` to "random_seed".
    # Then, call `fit_predict()` to get the cluster centers for each pixel of␣
 ↪the image
    # obtained from the previous step.

    gaussian_mix = sklearn.mixture.GaussianMixture(max_iter=60,␣
 ↪covariance_type="tied",n_components=n_components, random_state=random_seed)
    gaussian_mix.fit(scaled_data)
    fit_predict_result = gaussian_mix.fit_predict(scaled_data)

    ## step 4: replace each pixel by its cluster center value
    reshaped_raw_img = gaussian_mix.means_[fit_predict_result]

    ## step 5: return the image from the previous step

    return np.reshape(reshaped_raw_img, (original_shape[0], original_shape[1],␣
 ↪original_shape[-1]))

    # End of your code␣
 ↪--------------------------------------------------------------------------
```

[163]:
```
## Test your function: Plot your new images
random_seed = 2022
for k in [2, 3, 5, 10]:
    print("number of clusters:", k)
    new_img = question_1_1(raw_img, k, random_seed)
    plt.imshow(new_img)
    plt.show()
    print()
```

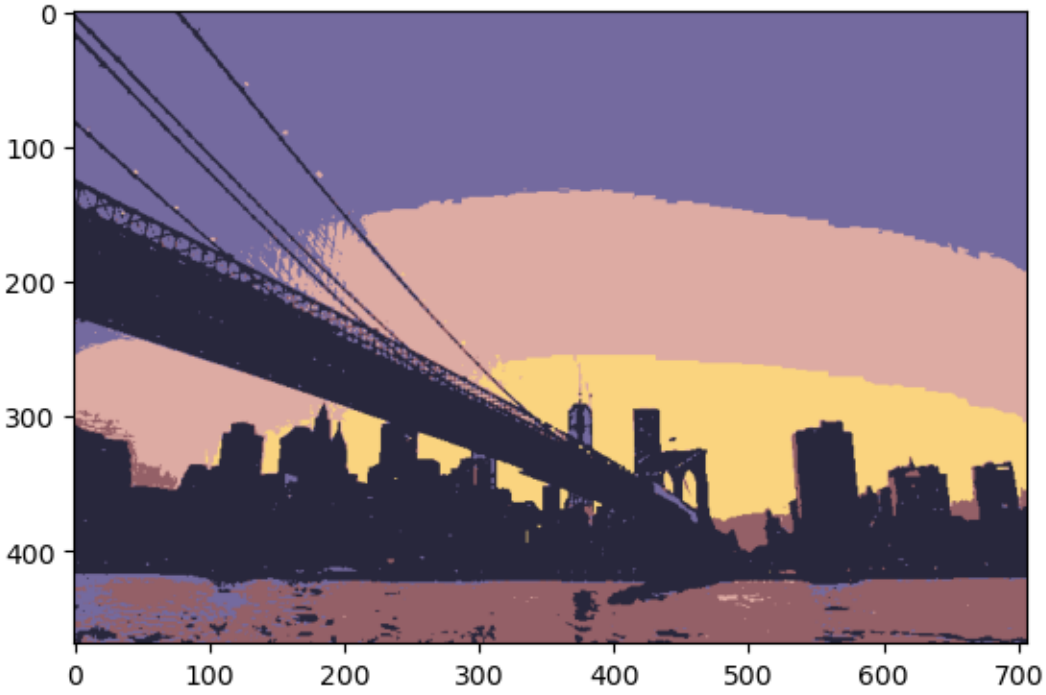
```
number of clusters: 2
```



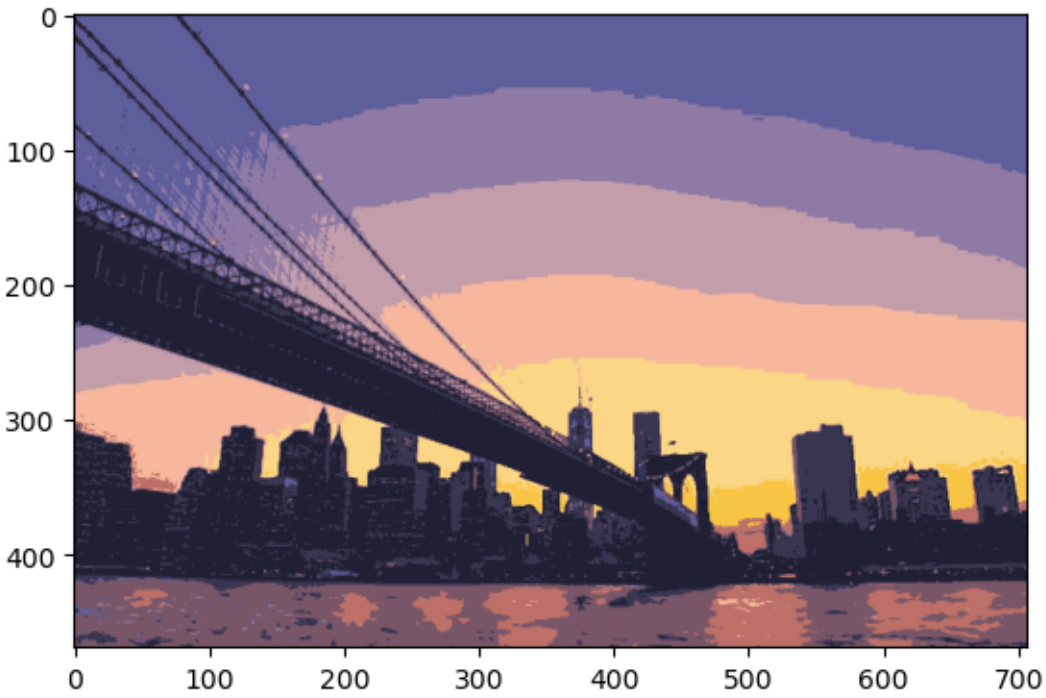

```
number of clusters: 3
```



```
number of clusters: 5
```




```
number of clusters: 10
```



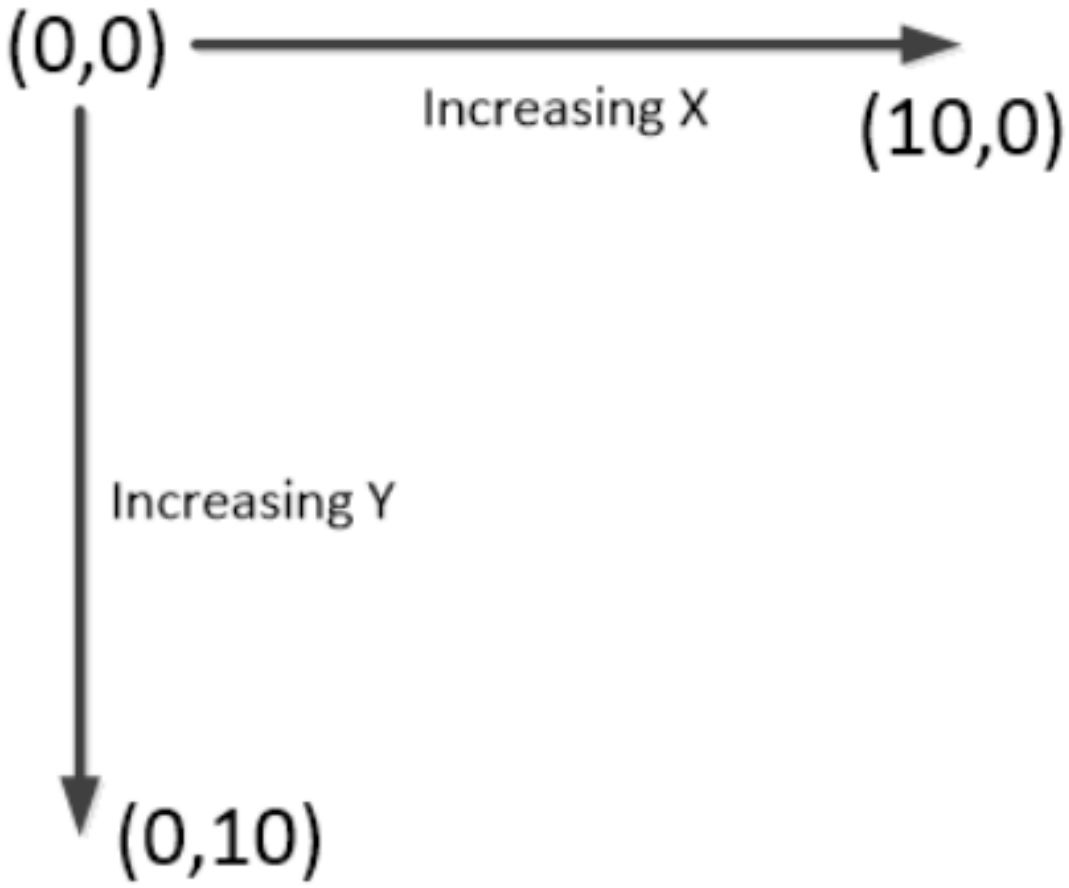
2.2 1.2 Code: GMM - Adding coordinates *(10 pts)*

You may notice that the previous section can produce image segments that have many connected components. For some applications, this is fine, but for others, we want segments that are compact clumps of pixels. One way to achieve this is to represent each pixel with 5D vector, consisting of its R, G and B values and also its x and y coordinates. You then cluster these 5D vectors.

We will add the coordinate starting by (0,0) at top left corner as the picture below.

[164]:
```
from IPython import display
display.Image("https://raw.githubusercontent.com/chaudatascience/cs599_fall2022/
  ↪master/ps3/gmm2.png")
```

[164]:



[165]:
```
def question_1_2(raw_img: np.ndarray) -> np.ndarray:
    """
        Append 2 new dimensions for each pixel: (R, G, B, x, y), where x, y is␣
↪the pixel's coordinates
        raw_img: numpy array, shape of (img_width, img_height, num_channels) ␣
↪(e.g., (1734, 2600, 3) )
        return new 3-d numpy array, shape of (img_width, img_height,␣
↪num_channels + 2)
    """
    shape = raw_img.shape

    # Write your code in this block␣
↪-----------------------------------------------------------

    print(shape)

    reshaped_raw_img = np.reshape(raw_img, (shape[0] * shape[1], 3))
```

```
    coor = np.mgrid[0:shape[0]:1, 0:shape[1]:1].reshape(2, -1).T

    x_axis = coor[:,0]
    y_axis = coor[:,1]

    result = np.concatenate((raw_img[x_axis, y_axis], coor), axis=1)

    result = np.reshape(result, (shape[0], shape[1], shape[2] + 2))

    return result

    # End of your code␣
 ↪-------------------------------------------------------------------------
```

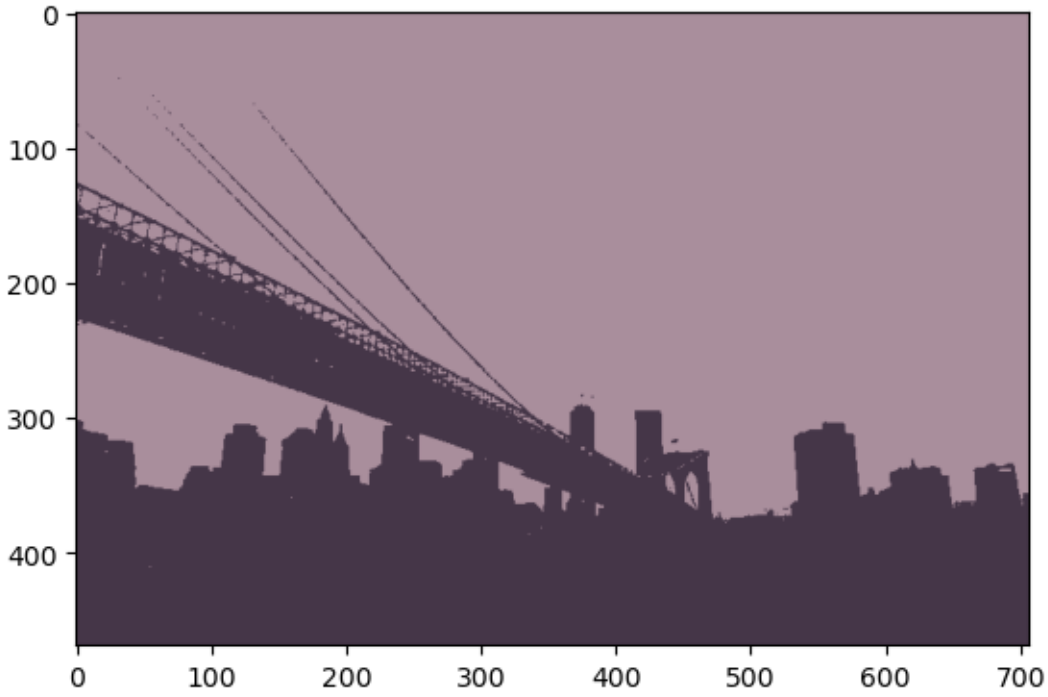
[166]:
```
## Test your function:
new_raw_img = question_1_2(raw_img)
print("new image's shape:", new_raw_img.shape)

print("\nShow the first 5 pixels on top left corner, along y-axis:\n",␣
 ↪new_raw_img[:5, 0, :])
print("\nShow the first 5 pixels on top left corner, along x-axis:\n",␣
 ↪new_raw_img[0, :5, :])


## Note: the last 2 columns are x, y coordinates, respectively
```

```
(469, 707, 3)
new image's shape: (469, 707, 5)

Show the first 5 pixels on top left corner, along y-axis:
 [[ 66  58 105   0   0]
 [ 66  59 101   1   0]
 [ 61  55  91   2   0]
 [ 45  40  70   3   0]
 [ 25  21  46   4   0]]

Show the first 5 pixels on top left corner, along x-axis:
 [[ 66  58 105   0   0]
 [ 57  49  96   0   1]
 [ 55  47  94   0   2]
 [ 61  53 100   0   3]
 [ 65  58 102   0   4]]
```

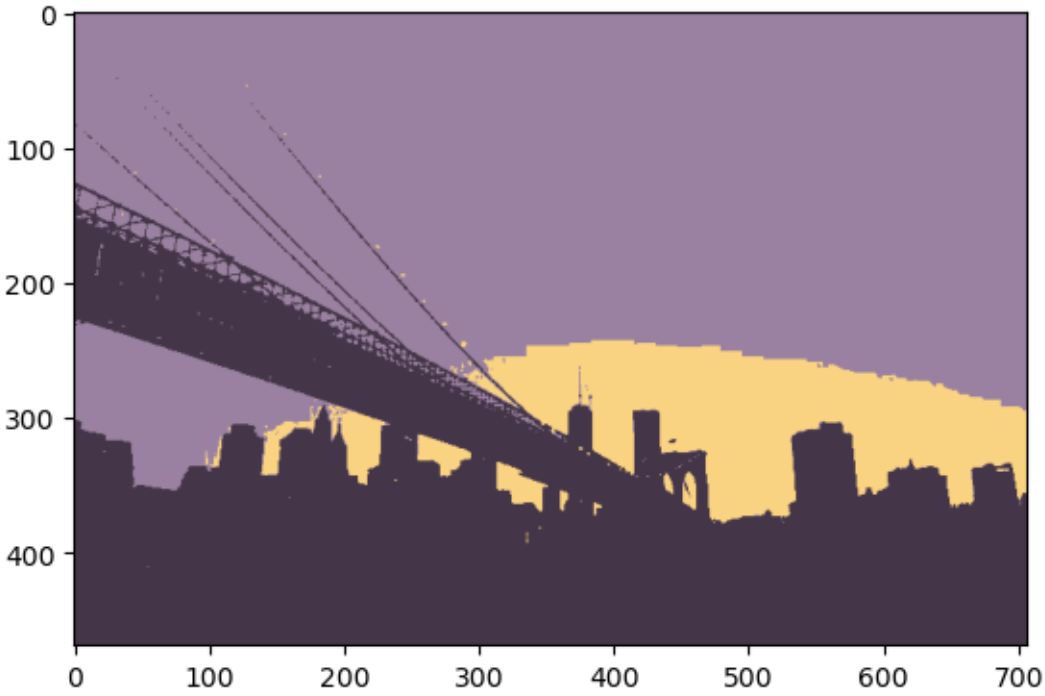
[167]:
```
## Plot the new images when fitting GMM on the 5d vectors
# We'll remove the (x,y) features from the new image before plotting
random_seed = 2022
for k in [2, 3, 5, 10]:
```

```
    print("number of clusters:", k)
    new_img = question_1_1(new_raw_img, k, random_seed)
    new_img_rgb = new_img[:,:,:3]  ## remove the last 2 dimensions (x, y) from␣
↪each pixel
    plt.imshow(new_img_rgb)
    plt.show()
    print()
```

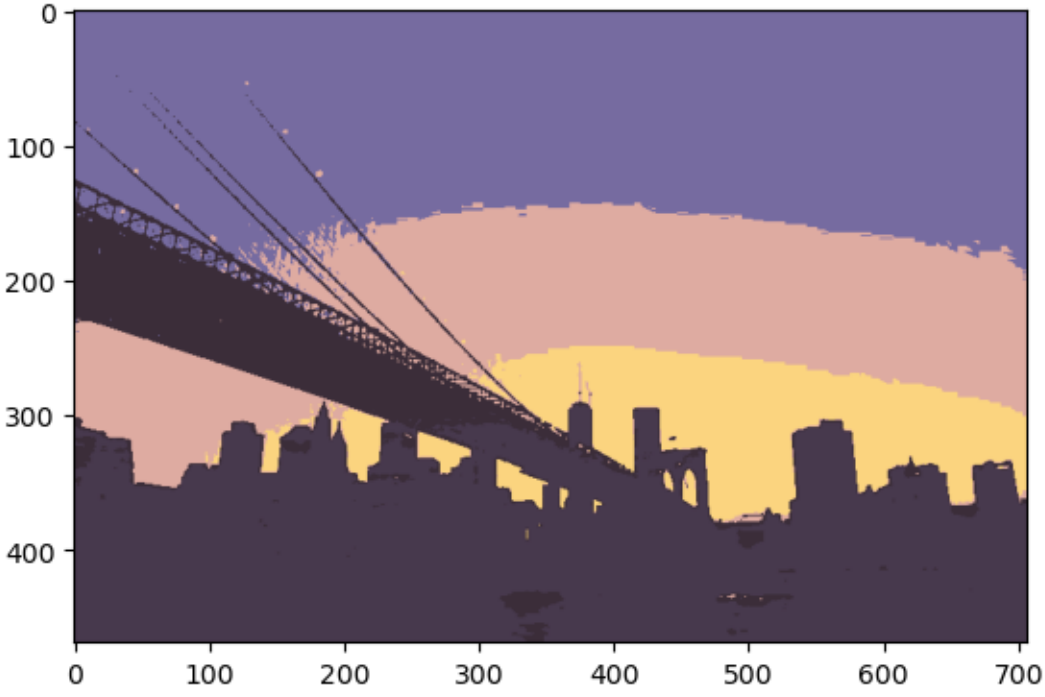
```
number of clusters: 2
```



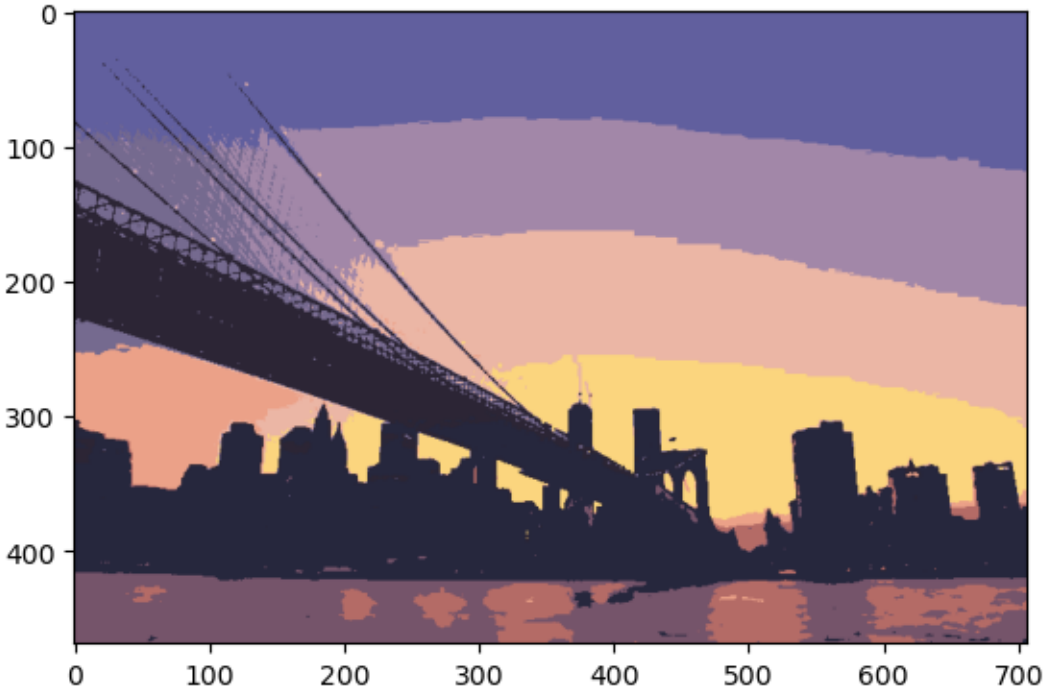

```
number of clusters: 3
```




```
number of clusters: 5
```



```
number of clusters: 10
```



We can observe that adding (x,y) into the features seems to force pixels near each other to belong the same cluster.

3 Question 2. GMM: Soft/Hard Cluster & Number of Components (*20 total points*)

3.1 2.1 Short answer: Soft clusters vs Hard clusters *(10 pts)*

Question: What are soft cluster and hard cluster? Which type of cluster GMM and K-means uses?

Write your answer in this block

Your Answer:

Soft cluster means that data points are assigned probabilities of them belonging to each cluster while hard cluster means that data points are strictly assigned one cluster. GMM uses soft clusters while K-means uses hard clusters.

3.2 2.2 Short answer: Number of components *(10 pts)*

Similar to K-means, we need to provide the number of clusters in advance for GMM to work. How should we pick an optimal value?

We can use some analytic criterion such as the Akaike information criterion (AIC) or the Bayesian information criterion (BIC).

The AIC value of the model is the following: AIC=2 − 2 ln( )

The BIC value is denoted as: BIC=−2 ln( )+ ln( )

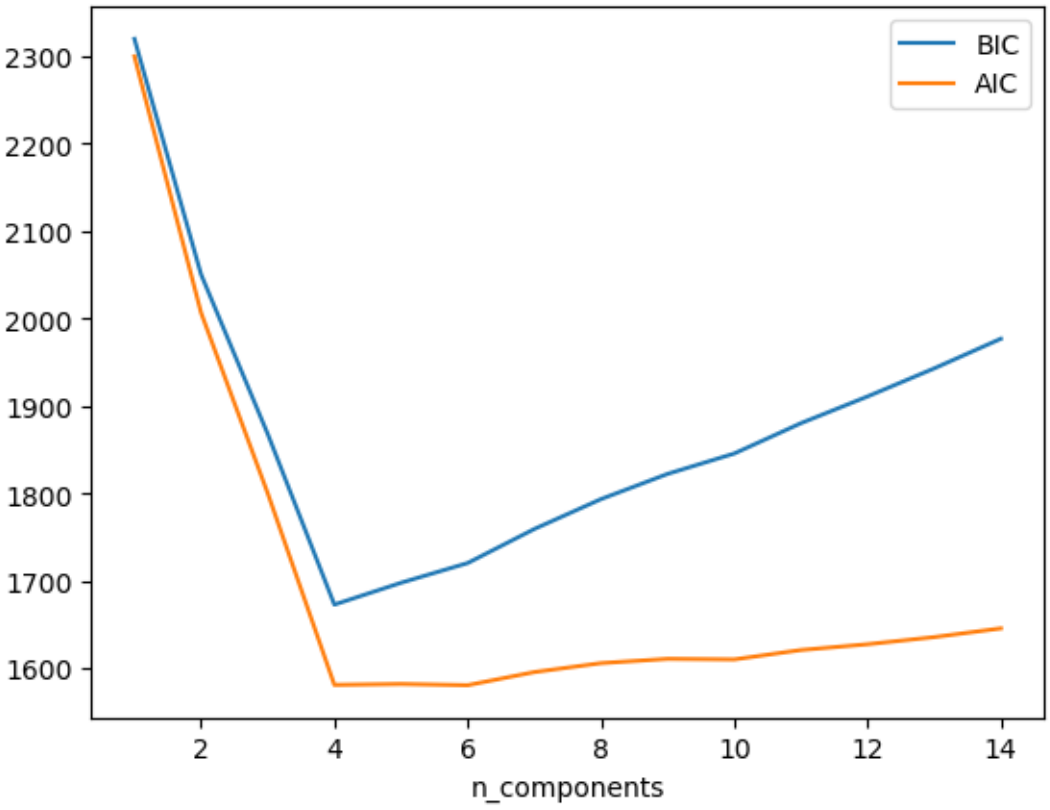
Where be the maximum value of the likelihood function for the model, be the number of estimated parameters in the model and be the total number of data points.

For both evaluation criterion, the lower the better.

It might be best to use AIC and BIC together in model selection. Alhough they usually agree on the results, BIC penalizes model complexity more heavily than AIC. In practice, we choose some numbers suggested by BIC and AIC for `num_components`, and see which one leads to a more suitable result.

[168]:
```
## Read a demo dataset
X = pd.read_csv("https://raw.githubusercontent.com/chaudatascience/
  ↪cs599_fall2022/master/ps3/gmm_data1.csv").values
print("X's shape:", X.shape)

n_components = np.arange(1, 15)
clfs = [GMM(n, random_state=2022).fit(X) for n in n_components]
bics = [clf.bic(X) for clf in clfs]
aics = [clf.aic(X) for clf in clfs]

plt.plot(n_components, bics, label = 'BIC')
plt.plot(n_components, aics, label = 'AIC')
plt.xlabel('n_components')
plt.legend()
plt.show()
```

```
X's shape: (400, 2)
```



Question: Which values should we choose for `num_components`? Justify your choice.

Write your answer in this block

Your Answer:

Any value between 4 to 6 could be chosen for `num_components`. At 4 components, both values for BIC and AIC are the lowest. The value for AIC stayed aproximately the same between 4 to 6 components, which indicates that any number of components between 4 to 6 can perform very similarly, with 4 being the most optimal number of component.

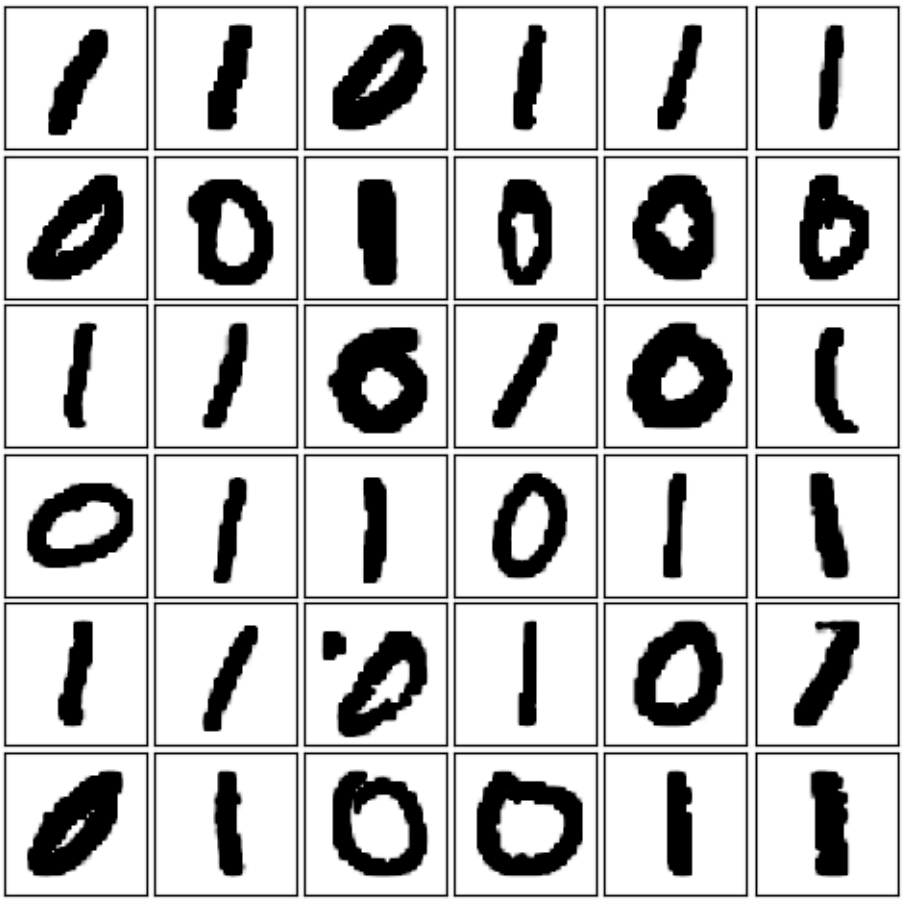
4 Question 3. GMM: Generating new samples (*20 total points*)

With GMM, we can also generate new samples from the distribution defined by the input data. In this section, we will generate new handwritten digits for digit 0 and 1. The dataset was sampled from MNIST dataset.

[169]:
```
## Read a sampling of MNIST dataset
X = pd.read_csv("https://raw.githubusercontent.com/chaudatascience/
  ↪cs599_fall2022/master/ps3/mnist_0_and_1_sampling.csv").values
X.shape  ## we have 5k pictures in gray scale of digit 0 and 1, each has a␣
  ↪resolution of 28x28
```

[169]: (5000, 784)

[170]:
```
def plot_digits(data):
    num_digit_figs = (6,6)
    fig_size = (6,6)
    digit_size = (28, 28)
    fig, ax = plt.subplots(*num_digit_figs, figsize=fig_size,␣
 ↪subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(*digit_size), cmap='binary')
        im.set_clim(0, 16)

## Let's plot some pictures to get a sense of the dataset
plot_digits(X)
```



We will use a GMM model to generate new samples similar to the ones above. You need to complete the function below.

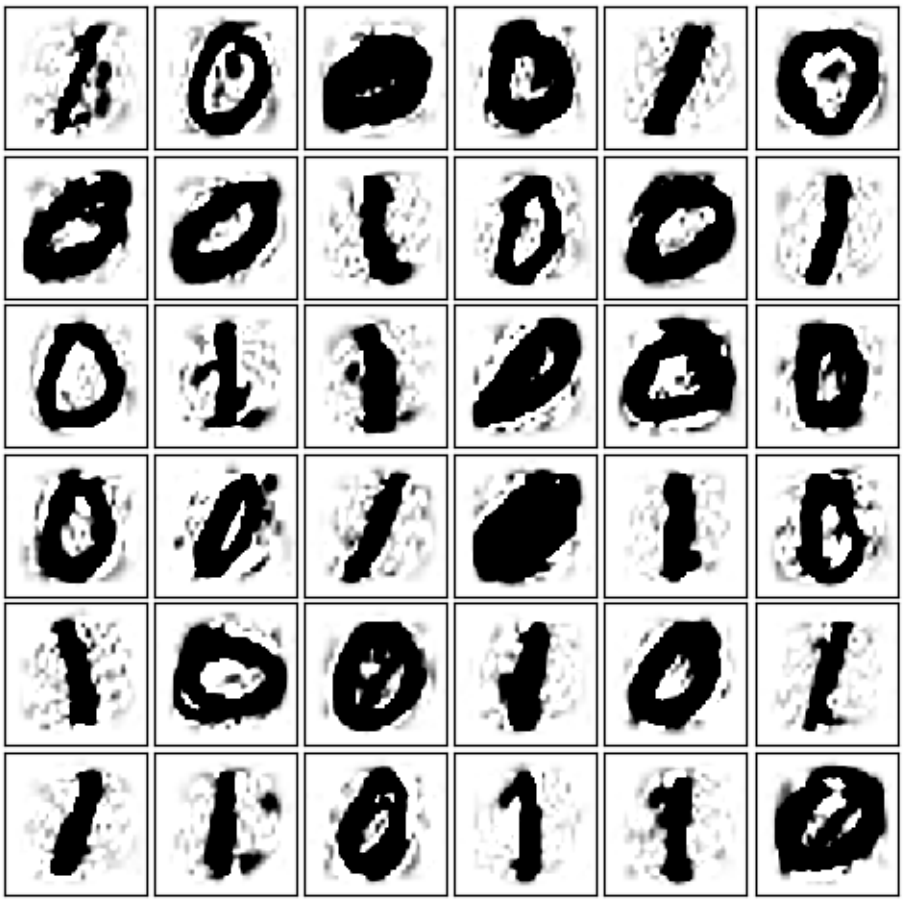
4.1 3.1 Code: Generate new samples *(20 pts)*

[181]:
```
from sklearn.decomposition import PCA

def question_3(X: np.ndarray, random_seed: int = 2022) -> np.ndarray:
    """
        X: digit inputs,  2-d numpy array shape of (num_samples, 784)
        random_seed: random seed, passed to GMM when initializing.
        return: 36 digit images,  2-d numpy array, shape of (36,)

    """
    # Write your code in this block␣
↪------------------------------------------------------------

    ## Step 1: Dimension reduction
    # Working on 28x28 = 784 dimensions requires a lot of computation.
    # It can also give GMM a difficult time to converge.
    # Thus, we will reduce the number of dimension by using PCA on the MNIST␣
↪dataset.
    # You'll need to create a Sklearn's PCA model that preserves 98% of the␣
↪variance in the recuded data.
    # Also, set random_state to `random_seed`.
    # Hint: for setting that preserves 98% variance,
    # you can take a look at attribute `n_components` when initialzing PCA␣
↪object.
    # The output of this step should have a shape of (5000, 176), which means␣
↪we keep the first 176 principle components.

    pca_model = PCA()
    pca_model.fit(X)

    variance_ratio = np.cumsum(pca_model.explained_variance_ratio_)

    n_components = np.argmax(variance_ratio >= 0.98) + 1

    actual_pca = PCA(random_state=random_seed, n_components=n_components)
    reduced_X = actual_pca.fit_transform(X)

    ## Step 2: Build a GMM model and fit it on the reduced data
    # Let's say we already used AIC and picked n_components = 140 for our GMM␣
↪model.
    # You need to create a GMM model with 140 components, and set random_state␣
↪to `random_seed`
    # for reproducing purpose, then fit the model on the reduced data.

    gmm = sklearn.mixture.GaussianMixture(n_components=140,␣
↪random_state=random_seed)
```

```
    gmm.fit(reduced_X)



    ## Step 3: from the GMM model, use method `gmm.sample()` to sample 36␣
↪images. Check out the n_samples argument.
    # Note 1: Right now, each of these new samples only has 176 dimensions.
    # In the next step, we will reconstruct the samples to have the data in 784␣
↪dimensions.
    # Note 2: `sample()` will return a tuple of both `X` and `y`, we only need␣
↪`X` for the next step

    images = gmm.sample(n_samples=36)
    image_X = images[0]


    ## Step 4: Pass `X` from the previous step into `inverse_transform()` of␣
↪the PCA model in step 1.
    # to reconstruct the new samples.

    inverse_image_X = actual_pca.inverse_transform(image_X)

    ## Step 5: Return the new samples
    # Your output should have a shape of (36, 784)

    return inverse_image_X

    # End of your code␣
↪-----------------------------------------------------------
```

[182]:
```
## Generating new digit images
digits_new = question_3(X, random_seed=2022)
print("digits_new.shape:", digits_new.shape)
plot_digits(digits_new)
```

```
digits_new.shape: (36, 784)
```



Although we only train a simple GMM on 5000 training samples, the new images look really amazing!

5 Question 4. Linear Regression (*30 total points*)

5.1 4.1 Code: Linear Regression using Sklearn *(10 pts)*

I this section, we will work with a demo dataset. The data consists of 2 columns: `hours_practice` (number of hours to practice) and `score`.

[183]:
```
## Read the dataset
df = pd.read_csv("https://raw.githubusercontent.com/chaudatascience/
  ↪cs599_fall2022/master/ps3/linear_data.csv")
print("data shape:", df.shape)
df.sample(4)
```

```
data shape: (100, 2)
```

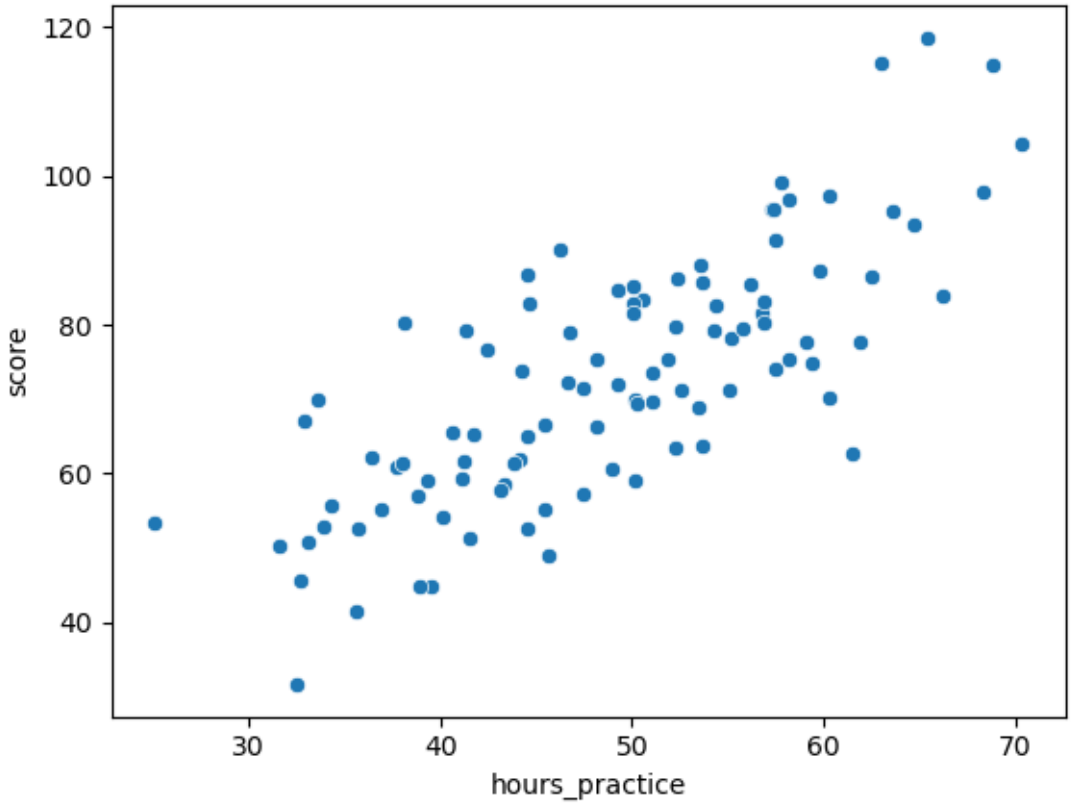
[183]:

	hours_practice	score
14	56.727208	81.436192
34	57.504448	74.084130
87	50.282836	69.510503
69	35.678094	52.721735

[184]:
```
## Extract features and labels as numpy arrays
X = df.values[:,0:1]  # features
y = df.values[:,1]  # labels
```

[185]:
```
## Plot the dataset
sns.scatterplot(data=df, x="hours_practice", y="score")
```

[185]: <Axes: xlabel='hours_practice', ylabel='score'>



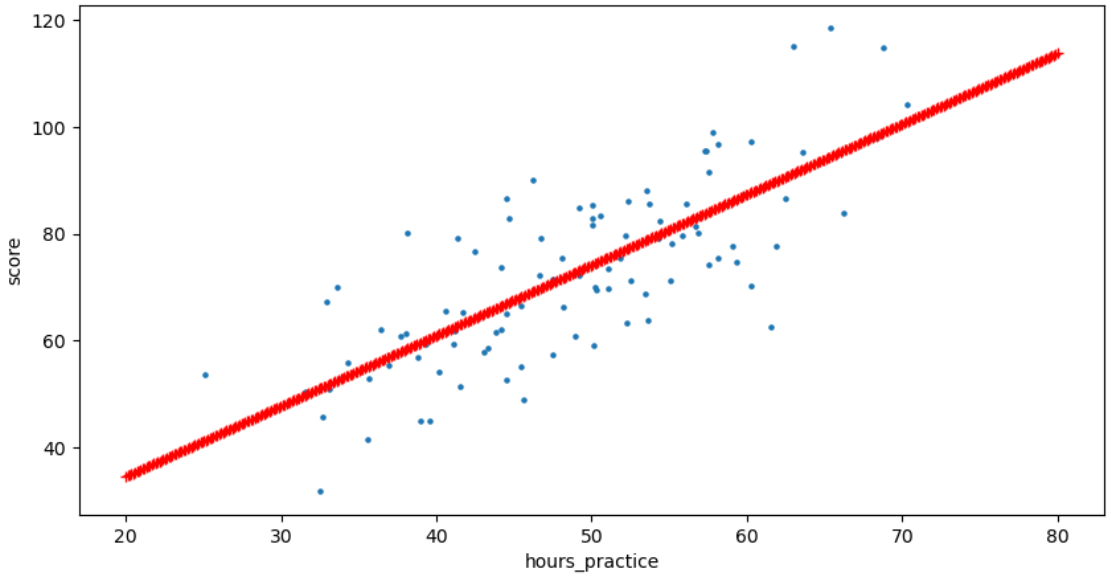
In this section, we will train a Linear Regression model on the dataset using Sklearn. You can refer to the document of Linear Regression here.

[186]:
```
from sklearn.linear_model import LinearRegression

def question_4_1(X: np.ndarray, y: np.ndarray) -> LinearRegression:
```

```
    """
    Train a Sklearn's Linear Regression model on features `X` and labels `y`.
    X: 2d numpy array, shape of (num_samples, feat_dim)
    y: numpy array, shape of (num_samples, )
    return a trained Linear Regression model

    """
    # Write your code in this block␣
↪------------------------------------------------------------
    model = LinearRegression().fit(X,y)


    # End of your code␣
↪------------------------------------------------------------

    return model
```

[187]:
```
## Test your model


def plot_model(linear_model, X, y, start, end):
    print("Model slope:    ", linear_model.coef_)
    print("Model intercept:", linear_model.intercept_)


    xfit = np.linspace(start, end, 500)[:, np.newaxis]
    if len(linear_model.coef_) == 2:
        xfit = np.concatenate([xfit, xfit**2], axis=1)
    yfit = linear_model.predict(xfit)

    plt.figure(figsize=(10,5))
    plt.scatter(X[:,0], y, s=5)
    plt.xlabel("hours_practice")
    plt.ylabel("score")
    plt.plot(xfit[:,0], yfit, 'r+')

# Get your model
linear_model = question_4_1(X, y)

# Plot: Your regression line is the red line as shown below
start, end = 20, 80  # start and end of the line
plot_model(linear_model, X, y, start, end)
```

```
Model slope:     [1.32243102]
Model intercept: 7.991020982270399
```

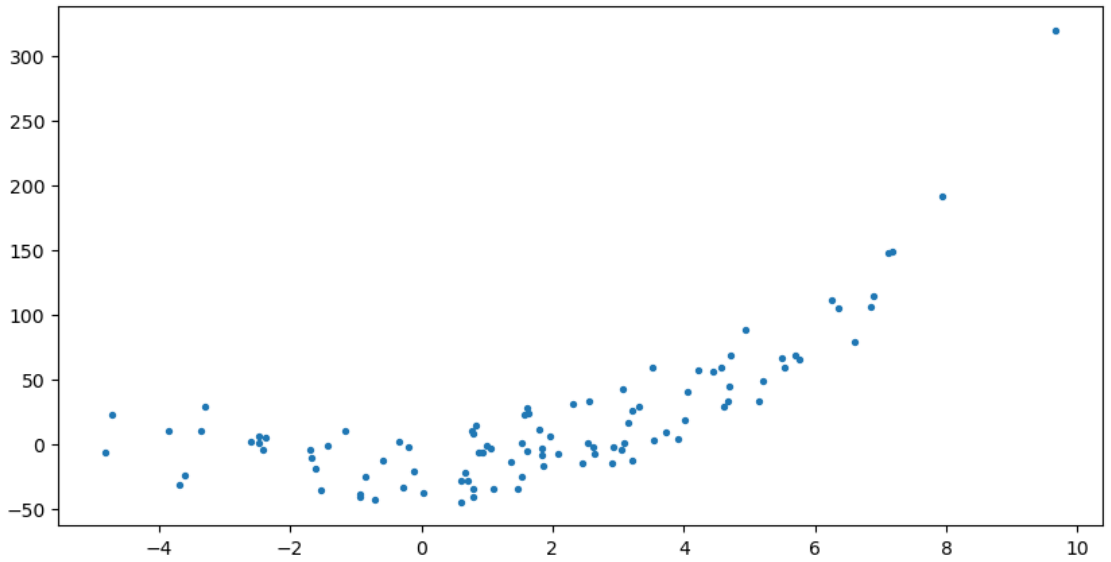


5.2 4.2 Code: Polynomial features *(10 pts)*

Let's take a look at another demo dataset

[188]:
```
data = pd.read_csv("https://raw.githubusercontent.com/chaudatascience/
↪cs599_fall2022/master/ps3/polynomial.csv").values
X2, y2 = data[:,:1], data[:,1]
plt.figure(figsize=(10,5))
plt.scatter(X2, y2, s=8)
```

[188]:
```
<matplotlib.collections.PathCollection at 0x1ec7f3f50>
```



We can see that the dataset is not linear. In other words, using a line can not capture the pattern in the data, resulting in underfitting. To solve this, we need to make our model a bit more complex.

There is a trick we can use to capture nonlinear relationships between variables: We first transform existing feature by some basic function, then use the generated data as new feature.

For example, with a linear regression for 1-d feature x:

$$Y = \theta_0 + \theta_1 x$$

We can transform the input feature x to get a new feature, such as x squared and consider it as a new feature. We now have 2 features, and the model becomes polynomial regression:

$$Y = \theta_0 + \theta_1 x + \theta_2 x^2$$

We can keep adding:

$$Y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + ...$$

to get even more complex model.

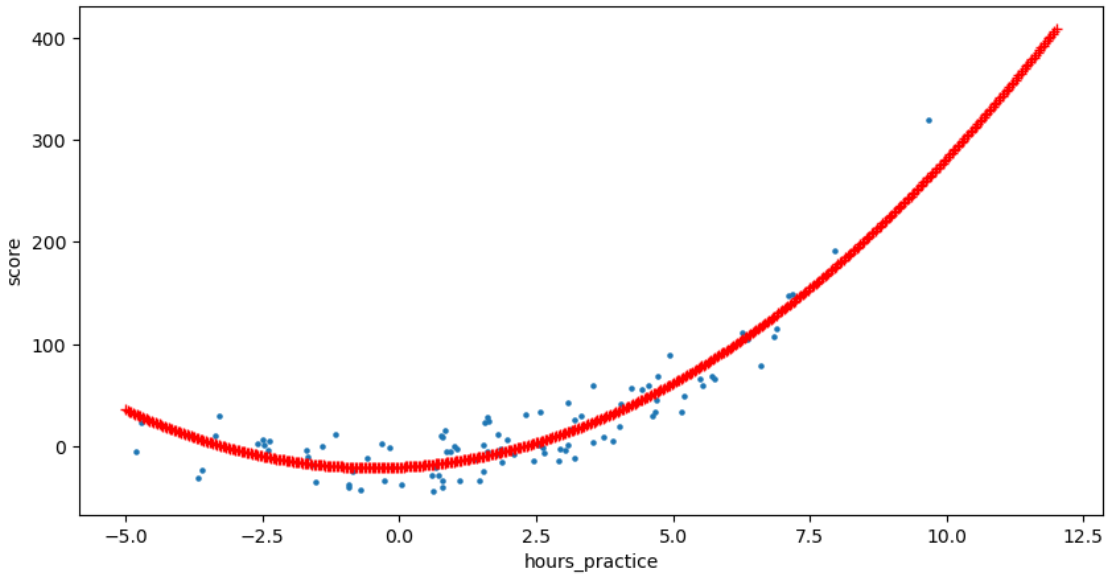
You will need to write a function to concatenate X^2 as a feature alongside X. So, now, the training data `X_new` will have X and X^2 as the features

[205]:
```
def question_4_2(X: np.ndarray) -> np.ndarray:
    """
        Given numpy array X, shape of (num_sample, 1).
        Return a numpy array, shape of (num_sample, 2) by adding a new column␣
 ↪to the right of X.
        The new column is the square of the existing column in X
    """
    # Write your code in this block␣
 ↪------------------------------------------------------------


    result = np.square(X)
    return np.concatenate((X, result), axis=1)


    # End of your code␣
 ↪------------------------------------------------------------


## Test your function
X_new = question_4_2(X2)
print("X_new.shape", X_new.shape)
print(X_new[:5]) # show the first 5 samples
```

```
X_new.shape (100, 2)
[[-3.29215704 10.83829796]
 [ 0.79952837  0.63924562]
 [-0.93621395  0.87649656]
 [-4.7226796  22.30370258]
 [-3.60267397 12.97925974]]
```

[206]:
```
## Plot the new model (in red)
new_linear_model = question_4_1(X_new, y2)
plot_model(new_linear_model, X_new, y2, start=-5, end=12)
```

```
Model slope:     [2.60168213 2.76791169]
Model intercept: -20.631851252724935
```



We can see the curve fits the data much better than a straight line.

5.3 4.3 Short answer: Linear model *(10 pts)*

Question: What is the shape of the curve (linear, or non-linear?) Is the model still considered to be a linear model?

Write your answer in this block

Your Answer: The shape of the curve is non-linear. However, the model is still considered to be a linear model.

6 Question 5. Linear Regression with Regularization *(25 total points)*

With basis functions, our model become more flexible, but it comes with a cost: The model is easier to over-fitting. One way to reduce overfitting is to penalize higher degree polynomials. This ensures that we only use the higher degree polynomials if the error is significantly reduced compared to a simpler model.

In this section, we will work on Boston Housing dataset. This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University. It consists of 13 continous features and a numerical target named *MEDV*. For more details about the dataset, you can refer this link.

Our goal is to train a linear regression model with regularization to learn the relationship between suburb characteristics and house prices.

[207]:
```
df = pd.read_csv("https://raw.githubusercontent.com/chaudatascience/
 ↪cs599_fall2022/master/ps3/boston_housing.csv")
print("df.shape", df.shape)
df.sample(5)
```

```
df.shape (506, 14)
```

[207]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	\
386	24.39380	0.0	18.10	0	0.700	4.652	100.0	1.4672	24	666.0	
79	0.08387	0.0	12.83	0	0.437	5.874	36.6	4.5026	5	398.0	
105	0.13262	0.0	8.56	0	0.520	5.851	96.7	2.1069	5	384.0	
62	0.11027	25.0	5.13	0	0.453	6.456	67.8	7.2255	8	284.0	
156	2.44668	0.0	19.58	0	0.871	5.272	94.0	1.7364	5	403.0	

	PTRATIO	B	LSTAT	MEDV
386	20.2	396.90	28.28	10.5
79	18.7	396.06	9.10	20.3
105	20.9	394.05	16.47	19.5
62	19.7	396.90	6.73	22.2
156	14.7	88.63	16.14	13.1

[208]:
```
## Extract features and labels as numpy arrays
X, y = df.iloc[:, :-1], df.iloc[:, -1]

## Split to train, test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42,␣
 ↪test_size=0.3)

## Check on the shapes
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(354, 13) (152, 13) (354,) (152,)
```

Sklearn provides a useful module named Pipeline which comes in handy when we need to perform sequence of different transformations.

An Example of using Pipeline is shown as below. We want to normalize the data, then create some new polynomial features, and finally a Linear model. Sklearn provides us PolynomialFeatures for generating polynomial and interaction features.

[211]:
```
##
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

## Steps is a list of Tuple[step_name, transformation_object]
steps = [
    ('scalar', StandardScaler()),    ## normilaze data
    ('poly', PolynomialFeatures(degree=2)),  ## add new features up to 2 degrees
    ('model', LinearRegression()) ## Linear regression model
]

linear_pipe = Pipeline(steps)
linear_pipe.fit(X_train, y_train)

## Let's see how we perform on the training and test sets
print('Training score: {}'.format(linear_pipe.score(X_train, y_train)))
print('Test score: {}'.format(linear_pipe.score(X_test, y_test)))
```

```
Training score: 0.9469794920108198
Test score: 0.66103219688773
```

On training set, the model performs very well, but the score drops significantly on test set. This suggests that our model is overfitting.

Now regulirization comes for the rescue.

Recall that there are three main techniques for regularization in linear regression, where we add a regularization term to the loss:

- Lasso Regression (L1 regularization): $\alpha \sum_{j=1}^{n} |\theta_j|$
- Ridge Regression (L2 regularization): $\alpha \sum_{j=1}^{n} |\theta_j^2|$
- Elastic Net (Combine L1 and L2 regularizations): $\alpha_1 \sum_{j=1}^{n} |\theta_j| + \alpha_2 \sum_{j=1}^{n} |\theta_j^2|$

Where n is the number of features, α is regularization parameter, which controls the degree of regularization.

In Sklearn, we can use `sklearn.linear_model.Lasso` for Linear Regression with L1 regularization. It also provides `sklearn.linear_model.Ridge` and `sklearn.linear_model.ElasticNet` for the other 2.

Similar to what we have done above, you should be able to perform a Linear Regresison with regularization.

Complete the function below for Lasso and Ridge regression by using the code example above. In the function, you should define **steps the same as we use in the example:** First, a "scalar", then "poly" followed by a "model". The only thing different here is the model (`Lasso` and `Ridge`, instead of `LinearRegression`)

6.1 5.1 Code: Lasso and Ridge *(10 pts)*

[218]:
```
from sklearn.linear_model import Ridge, Lasso

def question_5_1(regularization: str, alpha_1: float, alpha_2: float,
                 X_train, y_train, X_test, y_test) -> Tuple[float, float]:
    """
        regularization: one of ["L1", "L2"]. If "L1", use Lasso, otherwise use␣
 ↪Ridge.
        alpha_1: regularization for Lasso (if Lasso is used)
        alpha_2: regularization for Ridge (if Ridge is used)
        X_train, y_train, X_test, y_test: numpy arrays, shapes (354, 13),␣
 ↪(354,), (152, 13), (152,) respectively
        return a Tuple: (train_score, test_score) in that order,
        Note that train_score and test_score are float numbers in range [0,1]
    """
    # Write your code in this block␣
 ↪-----------------------------------------------------------
    # You should define `steps` the same as we use in the example above:
    #       first a "scalar", then "poly" followed by a "model".
    if (regularization == 'L1'):
        regression_model = Lasso(alpha=alpha_1)
    elif (regularization == 'L2'):
        regression_model = Ridge(alpha=alpha_2)

    steps = [
        ('scalar', StandardScaler()),
        ('poly', PolynomialFeatures(degree=2)),
        ('model', regression_model)
    ]

    pipe = Pipeline(steps)
    pipe.fit(X_train, y_train)

    return (pipe.score(X_train, y_train), pipe.score(X_test, y_test))



    ## Don't forget to return train and test scores!
    # End of your code␣
 ↪-----------------------------------------------------------
```

[219]:
```
## Test your model
alpha_1 = 0.1
alpha_2 = 12
for regularization in ["L1", "L2"]:
    train_score, test_score = question_5_1(regularization, alpha_1, alpha_2,
```

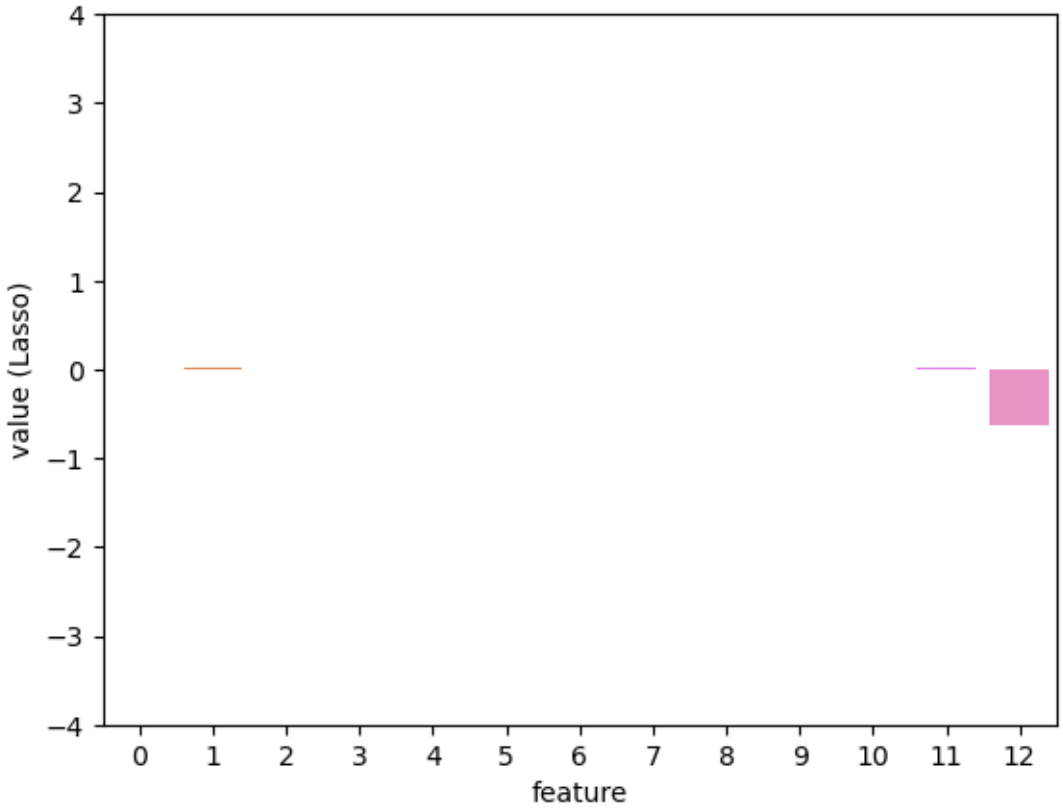
```
                                                        X_train, y_train,␣
↪X_test, y_test)
    print(f"regularization: {regularization}, train_score: {train_score},␣
↪test_score: {test_score}")
```

```
regularization: L1, train_score: 0.9070657101514069, test_score:
0.8055776105496003
regularization: L2, train_score: 0.9304830234444311, test_score:
0.8079087119535302
```

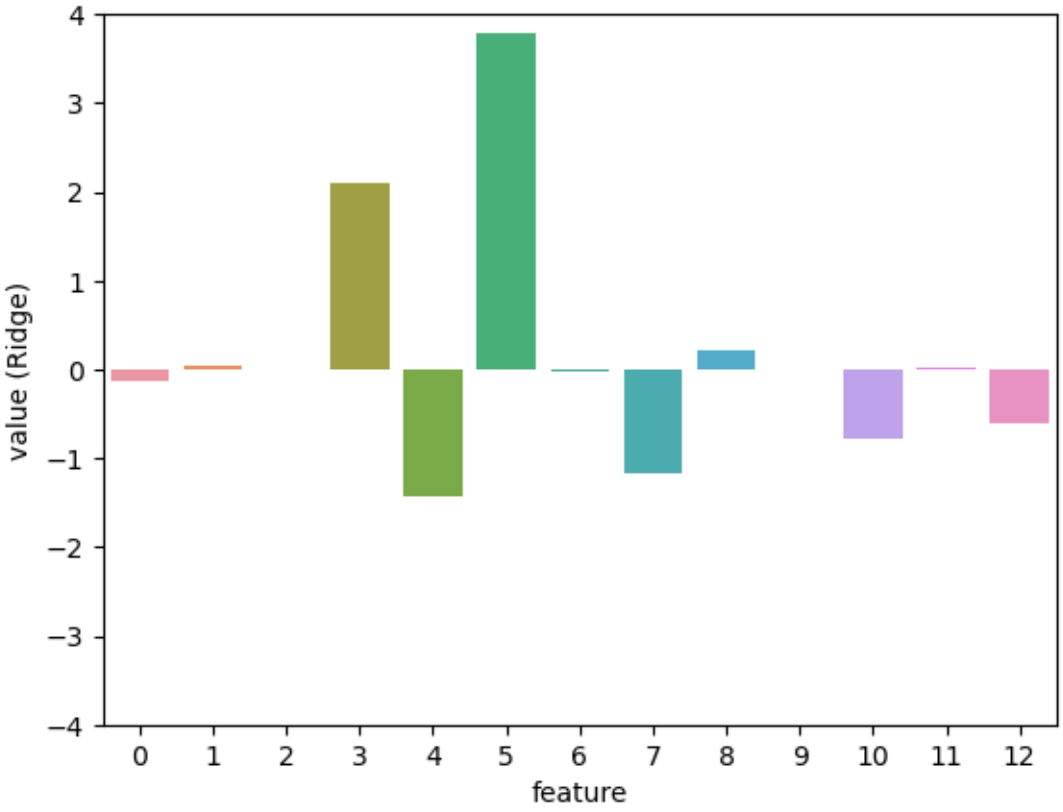
6.2 5.2 Short answer: Regularization Effects *(15 pts)*

[224]:
```
alpha = 10
lasso = Lasso(alpha).fit(X_train, y_train)
ridge = Ridge(alpha).fit(X_train, y_train)

# Here the input feature dimension is 13 (we are using X_train and not the␣
↪degree 2 polynomial features.)
# Recall that regularization will affect the coeffiecient placed on the␣
↪features while making the prediction.
# Let's see how the coefficients look like with Lasso and Ridge.

coefficients = pd.DataFrame({"feature": list(range(len(lasso.coef_))), "value␣
↪(Lasso)": lasso.coef_, "value (Ridge)": ridge.coef_})

## coefficients of Lasso
ax = sns.barplot(data=coefficients, x="feature", y="value (Lasso)")
ax.set( ylim=(-4, 4)) ## set  min, max for the y-axis
print("coefficients of Lasso")
plt.show()

## coefficients of Ridge
ax = sns.barplot(data=coefficients, x="feature", y="value (Ridge)")
ax.set( ylim=(-4, 4)) ## set  min, max for the y-axis
print("coefficients of Ridge")
plt.show()
```

```
coefficients of Lasso
```



coefficients of Ridge



Question:

1. Compare the performance of the model with and without regularization.

2. Compare the coefficients when using Lasso and Ridge.

3. How does alpha in Lasso and Ridge affect the coefficients and performance? (i.e., what happens if we use very tiny alpha, or very large alpha? You can play around with some values of alpha by changing the value `alpha=3` or in `alpha_1` and `alpha_2` above to get an intuition for this.)

Write your answer in this block

Your Answer: With regularization, the model yields about close scores on the training dataset but higher scores on the test dataset. Between Lasso and Ridge, the coefficients of Ridge change more than those of Lasso.

Congrats! You have reached to the end of Pset3