HW3

December 12, 2019

1 Fall 2019 CX4641/CS7641 Homework 3

Grade: 120/120

1.1 Programming (100pts)

1.2 1 Principal Component Analysis [20pts]

1.2.1 Environment Setup

[1]:
```
import os
from scipy import ndimage, misc
from matplotlib import pyplot as plt
import numpy as np
import imageio
from sklearn.datasets import load_boston, load_diabetes, load_digits,␣
↪load_breast_cancer
%matplotlib inline
```

1.3 Load images data and plot

We load the dataset we will deal with, and show the image

[2]:
```
image = plt.imread("hw_3_img.jpg")/255.
fig = plt.figure(figsize=(10,10))
plt.imshow(image)
```

[2]:
```
<matplotlib.image.AxesImage at 0x1027e6f28>
```

We can transfer the image to grayscale as follows:

[3]:
```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

fig = plt.figure(figsize=(10, 10))
# plot several images
plt.imshow(rgb2gray(image), cmap=plt.cm.bone)
```

[3]:
```
<matplotlib.image.AxesImage at 0x1028672b0>
```

1.3.1 Image compression

The SVD allows us to compress an image by throwing away the least important information. The greater the singular values -> the great the variance -> most information from the corresponding singular vector.

SVD each matrix and get rid of the small singular values to compress the image. The loss of inforamtion is negligible as the difference is very difficult to be spotted.

Principal Component Analysis(PCA) follows the same process to elimate the small variance eigenvalues and their vectors. With PCA, we center the data first by subtracting the mean.

Each singular value tells us how much of the variance of a matrix (e.g. image) is captured in each component. For example, the variance captured by the first component is

$$\frac{\sigma_1}{\sum_{i=1}^{n} \sigma_i}$$

where σ_i is the i^{th} singular value.

You need to finish the following functions to do SVD and then reconstruct the image by components.

[4]:
```
def svd(X):
    """
    Do SVD. You could use numpy SVD.
    Your function should be able to handle black and white
    images (N*D arrays) as well as color images (N*D*3 arrays)

    In the image compression, we assume that each coloum of the image is a␣
 ↪feature. Image is the matrix X.

    Args:
        X: N * D array corresponding to an image (N*D*3 if color image)
    Return:
        U: N*N (*3 for color images)
        S: min(N, D)*1 (*3 for color images)
        V: D*D (*3 for color images)
    """

    X_row = X.shape[0]
    X_col = X.shape[1]

    # for gray images
    if len(X.shape) == 2:
        U, S, V = np.linalg.svd(X)
        S = S.reshape(min(X_row, X_col), 1)

    # for color images
    elif len(X.shape) == 3:
        U = np.zeros((X_row, X_row, 3))
        S = np.zeros((min(X_row, X_col), 1, 3))
        V = np.zeros((X_col, X_col, 3))
        for i in range(0, 3):
            U_i, S_i, V_i = np.linalg.svd(X[:, :, i])
            S_i = S_i.reshape(min(X_row, X_col), 1)
            U[:, :, i] = U_i
            S[:, :, i] = S_i
            V[:, :, i] = V_i
    else:
        print('Error, please try again...')

    return U, S, V


def rebuildsvd(U, S, V, k):
    """
    Rebuild SVD by k componments.
    Args:
        U: N*N (*3 for color images)
```

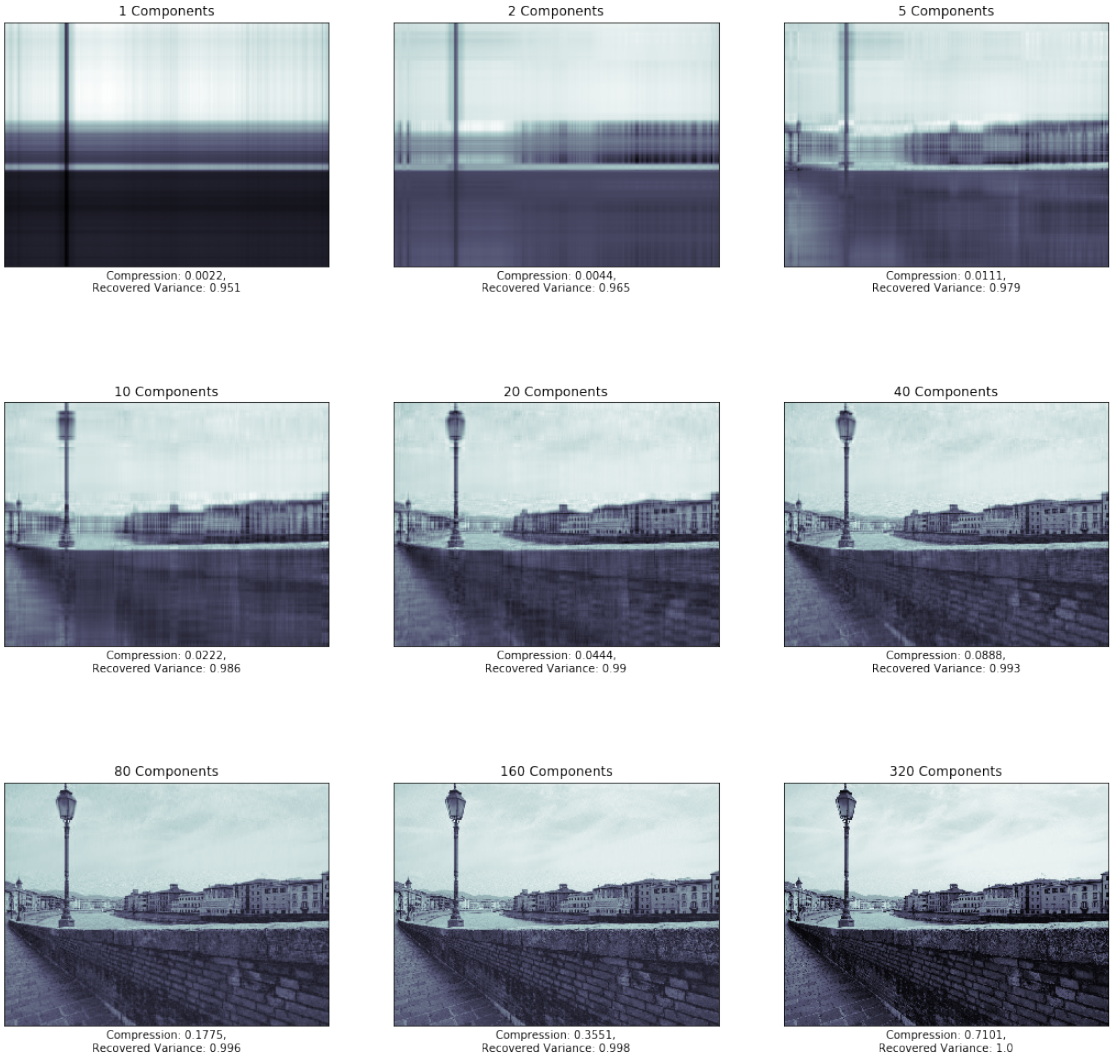
```
        S: min(N, D)*1 (*3 for color images)
        V: D*D (*3 for color images)
        k: int corresponding to number of components
    Return:
        Xrebuild: N*D array of reconstructed image (N*D*3 if color image)

    Hint: numpy.matmul may be helpful for reconstructing color images
    """

    if len(U.shape) == 2:
        S = S.flatten()
        Xrebuild = np.matrix(U[:, :k]) * np.diag(S[:k]) * np.matrix(V[:k, :])

    elif len(U.shape) == 3:
        Xrebuild = np.zeros((U.shape[0], V.shape[0], 3))
        for i in range(0, 3):
            S_new = S[:, :, i].flatten()
            temp_mat = np.matmul(np.diag(S_new[:k]), V[:, :, i][:k, :])
            Xrebuild[:, :, i] = np.matmul(U[:, :, i][:, :k], temp_mat)

    else:
        print('Dimension error, please try again with gray or color images')

    return Xrebuild


def compression_ratio(X, k):
    """
    Compute compression of an image: (num stored values in original)/(num␣
 ↪stored values in compressed)
    Args:
        X: N * D array corresponding to an image (N * D * 3 if color image)
        k: int corresponding to number of components
    Return:
        compression_ratio: float of proportion of storage used by compressed␣
 ↪image
    """

    X_row = X.shape[0]
    X_col = X.shape[1]

    if len(X.shape) == 2:
        compressed_size = X_row * k + k + k * X_col
        original_size = X_row * X_col

    elif len(X.shape) == 3:
        compressed_size = X_row * k * 3 + k * 3 + k * X_col * 3
```

```
        original_size = X_row * X_col * 3

    else:
        print('Error, please try again...')

    # did this because in the comment it's original/stored but in sample answer␣
↪it's stored/original
    compression_ratio = original_size / compressed_size
    compression_ratio = 1 / compression_ratio

    return compression_ratio

def recovered_variance_proportion(S, k):
    """
    Compute the proportion of the variance in the original matrix recovered by␣
↪a rank-k approximation

    Args:
       S: min(N, D)*1 (*3 for color images) of singular values for the image
       k: int, rank of approximation
    Return:
       recovered_var: int (array of 3 ints for color image) corresponding to␣
↪proportion of recovered variance
    """

    if len(S.shape) == 2:
        recovered_var = (S[:k] ** 2).sum() / (S ** 2).sum()

    elif len(S.shape) == 3:
        recovered_var = np.zeros((3,))
        for i in range(0, 3):
            recovered_var[i] = (S[:, :, i][:k] ** 2).sum() / (S[:, :, i] ** 2).
↪sum()

    else:
        print('Error, please try again...')

    return recovered_var
```

Now let see the results for image reconstruction.
1) Black and White

[5]:
```
#helper do not need to change

bw_image = rgb2gray(image)
U, S, V = svd(bw_image)
component_num = [1,2,5,10,20,40,80,160,320]
```

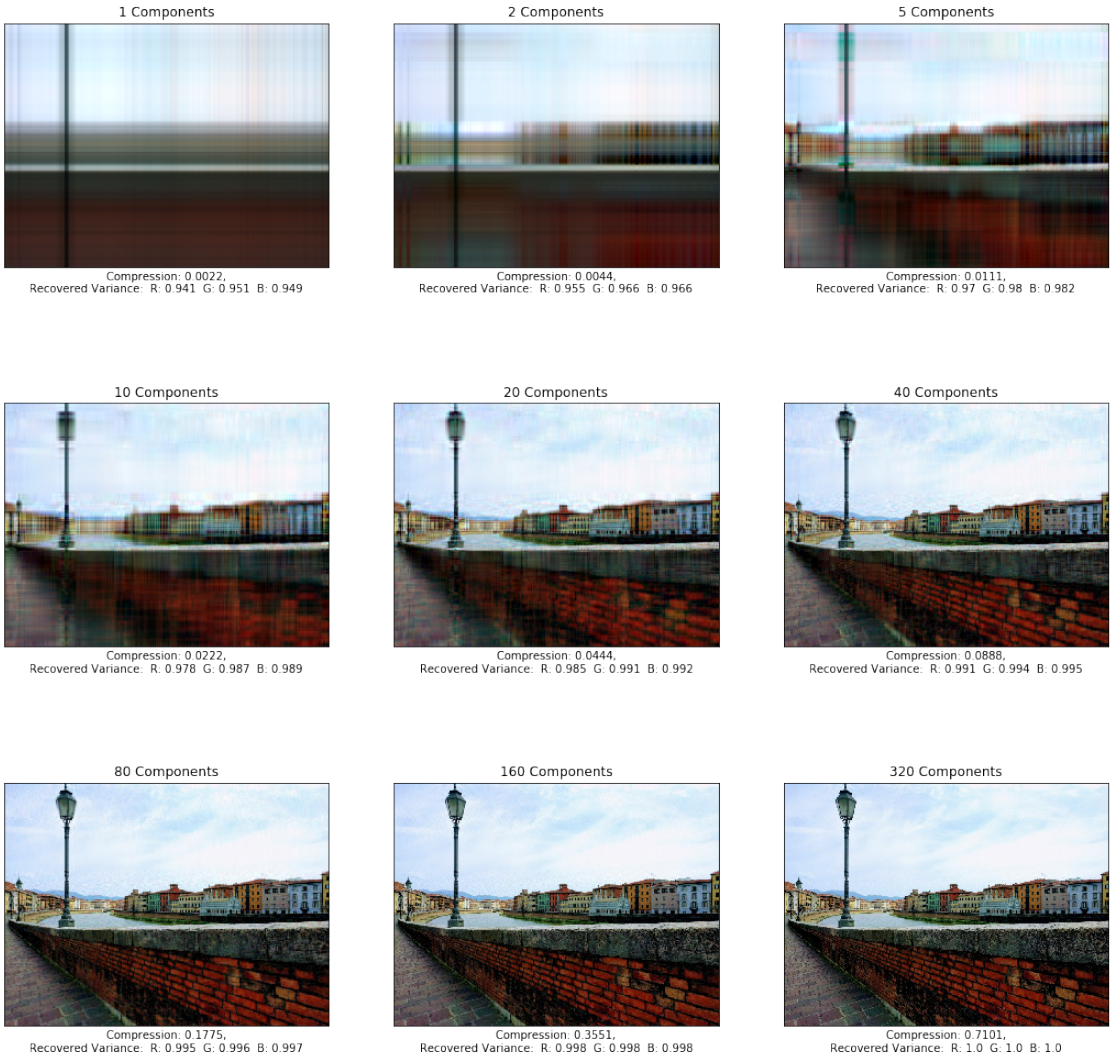
```
fig = plt.figure(figsize=(18, 18))

# plot several images
i=0
for k in component_num:
    img_rebuild = rebuildsvd(U, S, V, k)
    c = np.around(compression_ratio(image, k), 4)
    r = np.around(recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild, cmap=plt.cm.bone)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(f"Compression: {c},\nRecovered Variance: {r}")
    i = i + 1
```



2) Color

[6]:
```
#helper do not need to change

U, S, V = svd(image)
component_num = [1,2,5,10,20,40,80,160,320]

fig = plt.figure(figsize=(18, 18))

# plot several images
i=0
for k in component_num:
    img_rebuild = rebuildsvd(U, S, V, k)
    c = np.around(compression_ratio(image, k), 4)
    r = np.around(recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(f"Compression: {np.around(c,4)},\nRecovered Variance:  R:␣
↪{r[0]}  G: {r[1]}  B: {r[2]}")
    i = i + 1
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



1.4 2 Polynomial Regression with Regualrization [35pts + 20pts]

2.1 Helper functions [10pts]

1) RMSE [5pts]

[7]:
```
def rmse(pred, label):
    '''
    This is the root mean square error.
    Args:
        pred: numpy array of length N * 1, the prediction of labels
        label: numpy array of length N * 1, the ground truth of labels
    Return:
        a float value
```

```
    '''
    mse = (np.power((label - pred), 2)).mean()
    rmse = np.sqrt(mse)

    return rmse

# Hint: get the sqaure root of theta hat in slide 14 (https://mahdi-roozbahani.
 ↪github.io/cse4240-spring2019.github.io/course/14-linear-regression.pdf)
prediction = np.array([1, 2, 3])
label = np.array([1.5, 2.5, 3.5])
print('rmse =', rmse(prediction, label))
```

```
rmse = 0.5
```

Do you know whether this RMSE is good or not? If you don't know, we could normalize our labels between 0 and 1. After normalization, what does it mean when RMSE = 1?

Hint: think of the way that you can enforce your RMSE = 1. Note that you can not change the actual labels to make RMSE = 1.

Answer: Generally speaking, in a not normalized dataset, the smaller the RMSE is relative to original data, the better. It means our predicted values are very close to the real data. If we normalize our labels between 0 and 1, the labels in the problem become $[0, 0.5, 1]$, but then of course our model changes to fitting $[0, 0.5, 1]$. If RMSE = 1 after normalization, $\rightarrow$ MSE = 1, then potentially every $y_i - \hat{y}_i = 1$. This means our model is pretty bad and fails to capture the real "trend" of the original data, because every prediction is off the label by the largest difference (maximum label - minimum label which is normalized to 1).

2) Construct polynomial features [5pts]

[8]:
```
def construct_polynomial_feats(x, degree):
    """
    Args:
        x: numpy array of length N, the 1-D observations
        degree: the max polynomial degree
    Return:
        feat: numpy array of shape Nx(degree+1), remember to include
        the bias term. feat is in the format of:
        [[1.0, x1, x1^2, x1^3, ....,],
         [1.0, x2, x2^2, x2^3, ....,],
         ......
        ]
    """

    x_feat = np.zeros((x.shape[0], degree + 1))
    for i in range(0, x.shape[0]):
        x_feat[i] = [x[i] ** j for j in range(0, degree + 1)]

    return x_feat
```

```
# Here there are two data points. One data point is 0.5 and other one is 0.2.
# We try to represent these two data points in higher dimensions.

x = np.array([0.5, 0.2])
x_feat = construct_polynomial_feats(x, 4)
print(x_feat)
```

```
[[1.     0.5    0.25   0.125  0.0625]
 [1.     0.2    0.04   0.008  0.0016]]
```

[9]:
```
#helper do not need to change
def plot_curve(x, y, curve_type='.', color='b', lw=2):
    plt.plot(x, y, curve_type, color=color, linewidth=lw)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid(True)
```

2.2 Linear Regression [10pts + 10pts] We have three methods to fit linear gression model: 1) close form; 2)Gradient descent (GD); 3) Stochastic gradient descent (SGD). For undergraduate students, you are required to implement 1), and the other two are bonus parts; for graduate students, you are required to implement all of them. Each method has 5 points.

After fitting the model, we need to implement the prediction function. [5pts]

[10]:
```
class LinearReg(object):
    @staticmethod
    # static method means that you can use this method or function for any␣
 ↪other classes, it is not specific to LinearReg
    def fit_closed(xtrain, ytrain):
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytarin: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
        product = np.matmul(xtrain.T, xtrain)
        inv = np.linalg.inv(product)
        weight = np.matrix(inv) * np.matrix(xtrain.T) * np.matrix(ytrain)

        return weight

    @staticmethod
    def fit_GD(xtrain, ytrain, epochs=100, learning_rate=0.001):
```

```
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """

        theta = np.array([[0.0],[0.0],[0.0],[0.0],[0.0],[0.0]])
        rmse_per_epoch = []

        for i in range(0, epochs, 1):
            y_pred = np.dot(xtrain, theta)

            # N is 500.
            d_theta = (-2/500) * np.matrix(xtrain.T) * np.matrix(ytrain -␣
↪y_pred)
            # d_theta = (-2) * (np.multiply(xtrain, (ytrain - y_pred)).
↪mean(axis=0))
            theta = theta - (learning_rate * d_theta).reshape(6, 1)

            rmse_per_epoch.append(rmse(y_pred, ytrain))

        plt.figure()
        plt.plot(rmse_per_epoch)
        plt.xlabel('# Epochs')
        plt.ylabel('RMSE per Epoch')
        plt.show()

        weight = theta

        return weight

    @staticmethod
    def fit_SGD(xtrain, ytrain, epochs=100, learning_rate=0.001):
        """
        Args:
            xtrain: NxD numpy array, where N is number
                    of instances and D is the dimensionality of each
                    instance
            ytrain: Nx1 numpy array, the true labels
        Return:
            weight: Dx1 numpy array, the weights of linear regression model
        """
```

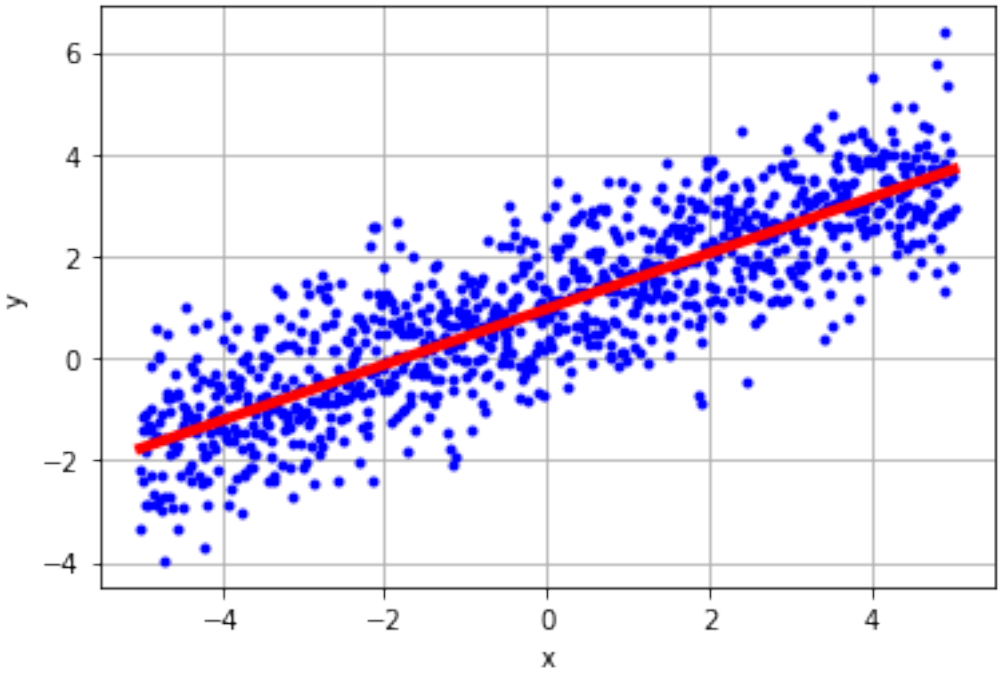
```
        theta = np.array([[0.0],[0.0],[0.0],[0.0],[0.0],[0.0]])
        rmse_per_epoch = []

        for i in range(0, epochs, 1):
            loss = 0.0
            for r in range(0, 500, 1):
#                 r = np.random.randint(0, 500)
                y_pred_r = np.dot(xtrain[r], theta)
                d_theta = (-2) * (xtrain[r] * (ytrain[r] - y_pred_r))
                theta = theta - (learning_rate * d_theta).reshape(6, 1)
                loss +=  rmse(y_pred_r, ytrain[r]) / 500
            rmse_per_epoch.append(loss)

        plt.figure()
        plt.plot(rmse_per_epoch)
        plt.xlabel('# Epochs')
        plt.ylabel('RMSE per Epoch')
        plt.show()

        weight = theta

        return weight

    @staticmethod
    def predict(xtest, weight):
        """
        Args:
            xtest: NxD numpy array, where N is number
                   of instances and D is the dimensionality of each
                   instance
            weight: Dx1 numpy array, the weights of linear regression model
        Return:
            prediction: Nx1 numpy array, the predicted labels
        """
        predict = np.matmul(xtest, weight)

        return predict
```

Let's first construct a dataset for polynomial regression.

In this case, we construct the polynomial features up to degree 5, where the groundtruth function is just a linear function (i.e., only require polynomial features up to degree 1).

[11]:
```
#helper, do not need to change

POLY_DEGREE = 5
NUM_OBS = 1000
```

```
rng = np.random.RandomState(seed=4)

true_weight = rng.rand(POLY_DEGREE + 1, 1)
true_weight[2:, :] = 0
x_all = np.linspace(-5, 5, NUM_OBS)
x_all_feat = construct_polynomial_feats(x_all, POLY_DEGREE)
y_all = np.dot(x_all_feat, true_weight) + rng.randn(x_all_feat.shape[0], 1) #␣
 ↪in the second term, we add noise to data
# Note that here we try to produce y_all as our training data
plot_curve(x_all, y_all) # Data with noise that we are going to predict
plot_curve(x_all, np.dot(x_all_feat, true_weight), curve_type='-', color='r',␣
 ↪lw=4) # the groundtruth information

indices = rng.permutation(NUM_OBS)
```

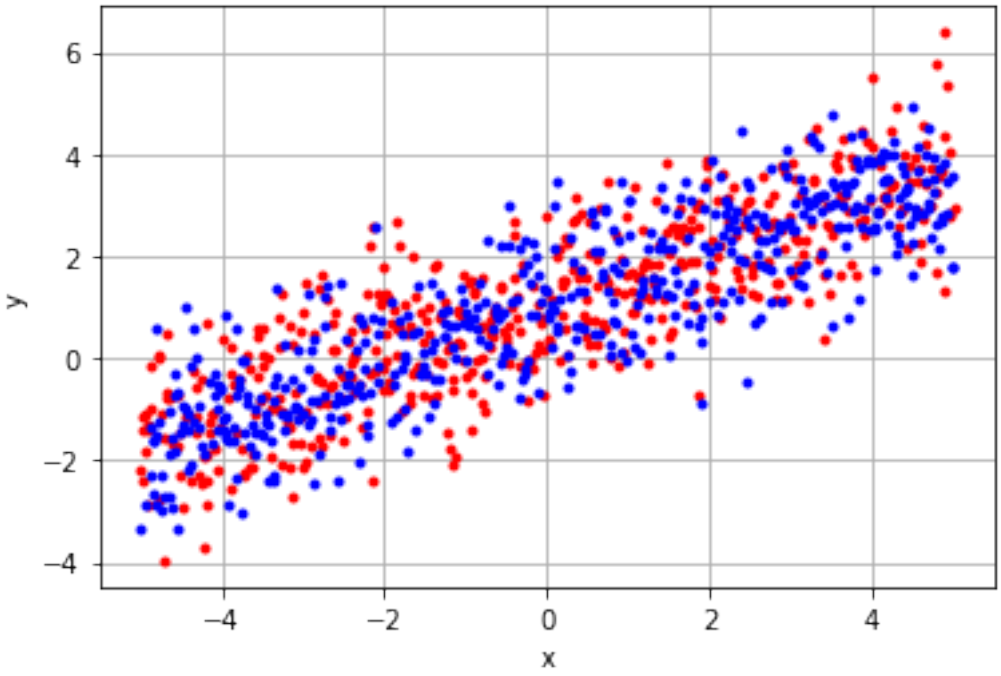


In the figure above, the red curve is the true fuction we want to learn, while the blue dots are the noisy observations. The observations are generated by $Y = XW + \sigma$, where $\sigma \sim \mathcal{N}(0, 1)$ are i.i.d. generated noise.

Now let's split the data into two parts, namely the training set and test set. The red dots are for training, while the blue dots are for testing.

[12]:
```
#helper, do not need to change
```

```
train_indices = indices[:NUM_OBS//2]
test_indices = indices[NUM_OBS//2:]

plot_curve(x_all[train_indices], y_all[train_indices], color='r')
plot_curve(x_all[test_indices], y_all[test_indices], color='b')
```

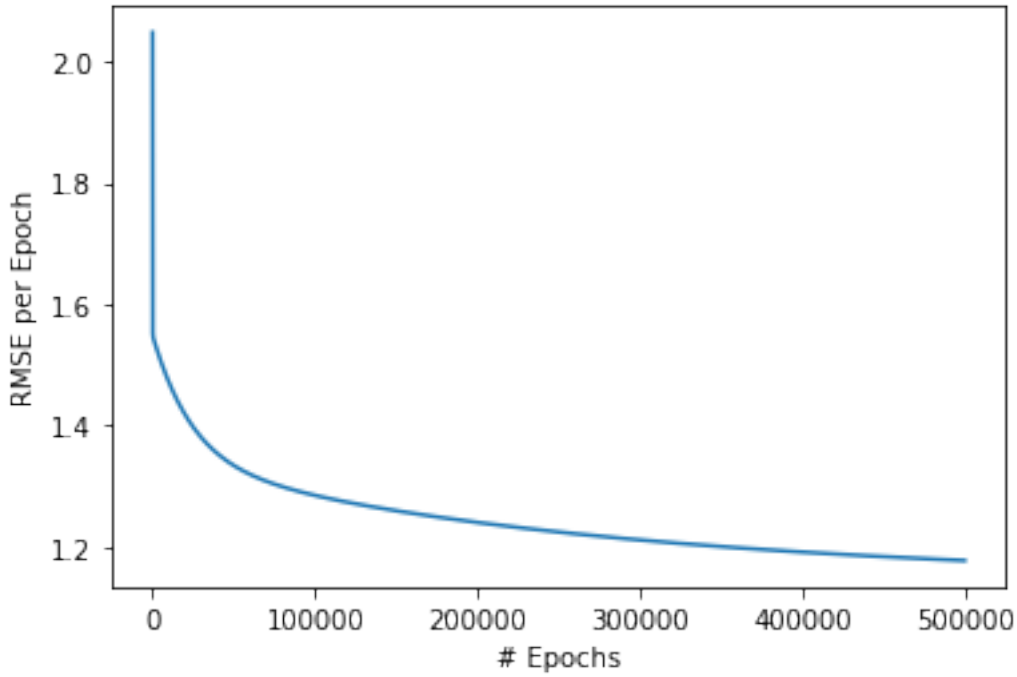


Now let's first train using the entire training set, and see how we performs on the test set and how the learned function look like.

[13]:
```
#helper, do not need to change

weight = LinearReg.fit_closed(x_all_feat[train_indices], y_all[train_indices])
y_test_pred = LinearReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

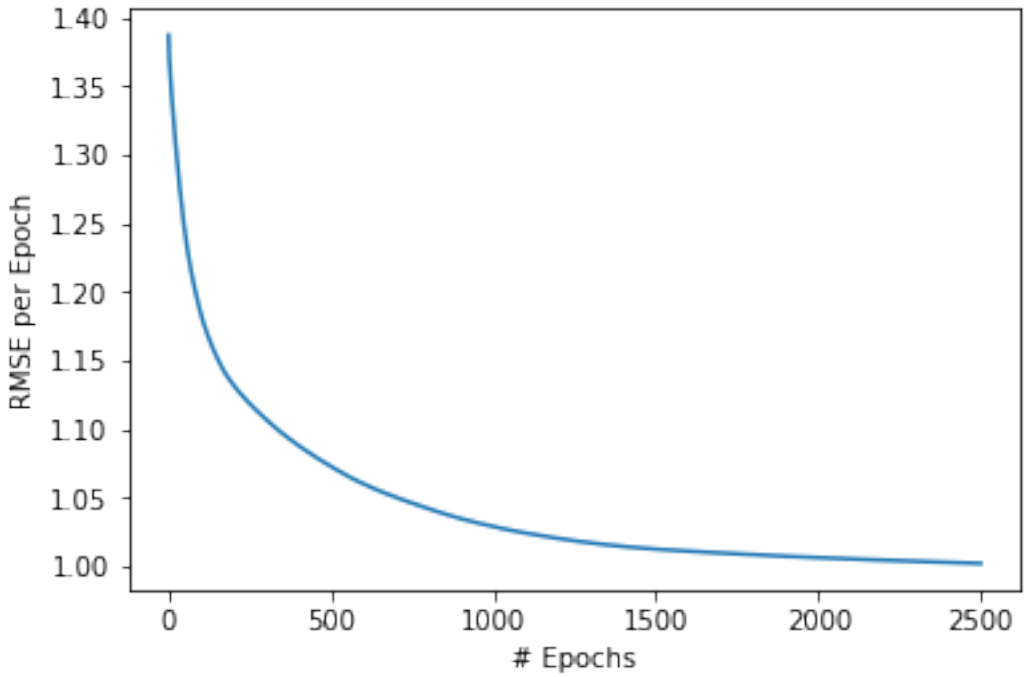
```
test rmse: 0.9222
```

[14]:
```
weight = LinearReg.fit_GD(x_all_feat[train_indices], y_all[train_indices],␣
 ↪epochs=500000, learning_rate=1e-7)
y_test_pred = LinearReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```




```
test rmse: 1.1233
```

[15]:
```
weight = LinearReg.fit_SGD(x_all_feat[train_indices], y_all[train_indices],␣
 ↪epochs=2500, learning_rate=1e-7)
y_test_pred = LinearReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```




```
test rmse: 1.3712
```

And what if we just use the first 10 observations to train?

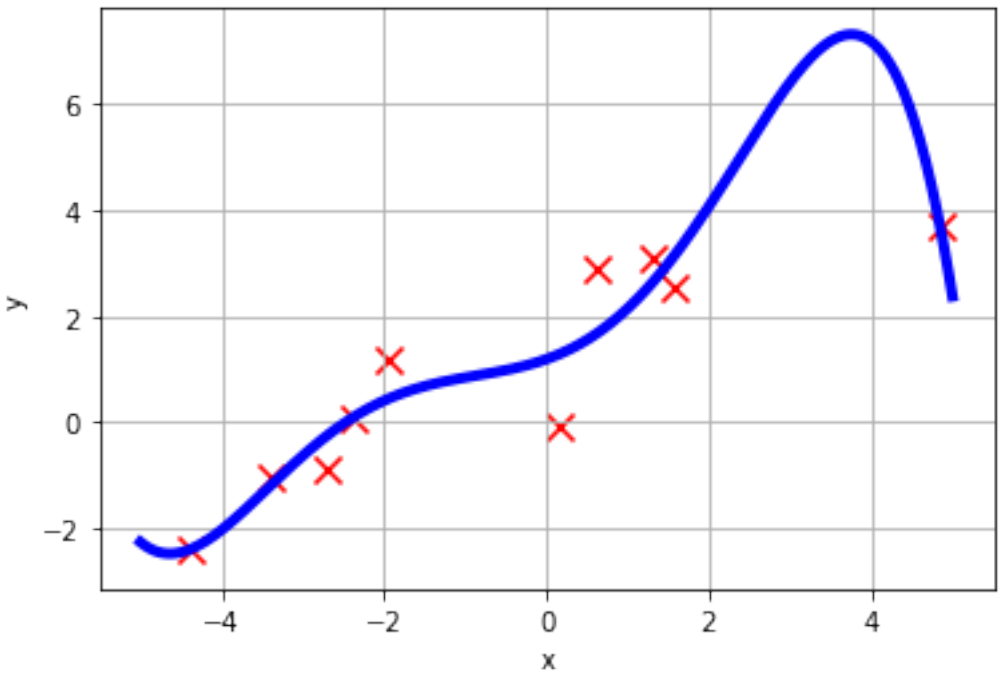
[16]:
```
sub_train = train_indices[:10]
weight = LinearReg.fit_closed(x_all_feat[sub_train], y_all[sub_train])
y_test_pred = LinearReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 2.1910
```

Did you see a worse performance? Let's take a closer look at what we have learned.

Answer: Yes, using only first 10 observations to train is worse than using all training data, because rmse calcualted is larger than using all training data. Visualizing the 10 data points we see that our fitted model does try to capture the "trend" of the 10 data points, but with less data points, it doesn't know how the data really looks like, so the fitting line has a big "hill" in it, which does not exist in the original datset.

[17]:
```
#helper, do not need to change

y_pred = LinearReg.predict(x_all_feat, weight)
plot_curve(x_all, y_pred, curve_type='-', color='b', lw=4)
plt.scatter(x_all[sub_train], y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = LinearReg.predict(x_all_feat[test_indices], weight)
```



2.3 Ridge Regression [5pts + 10pts] Now let's try ridge regression. Similarly, undergraduate students need to implement the close form, and graduate students need to implement all the three methods. We will call the prediction function from linear regression part.

[18]:
```
class RidgeReg(LinearReg):

    @staticmethod
    def fit_closed(xtrain, ytrain, c_lambda):
        D = xtrain.shape[1]
        lambda_mat = np.zeros((D, D))
        np.fill_diagonal(lambda_mat, c_lambda)
        lambda_mat[:, 0] = 0 # have to set first column to zero because bias␣
↪need not to be penalized
        product = np.matmul(xtrain.T, xtrain) + lambda_mat
        inv = np.linalg.inv(product)
        weight = np.matrix(inv) * np.matrix(xtrain.T) * np.matrix(ytrain)

        return weight

    @staticmethod
    def fit_GD(xtrain, ytrain, c_lambda, epochs=100, learning_rate=0.001):
        """
        Args:
            xtrain: NxD numpy array, where N is number
```

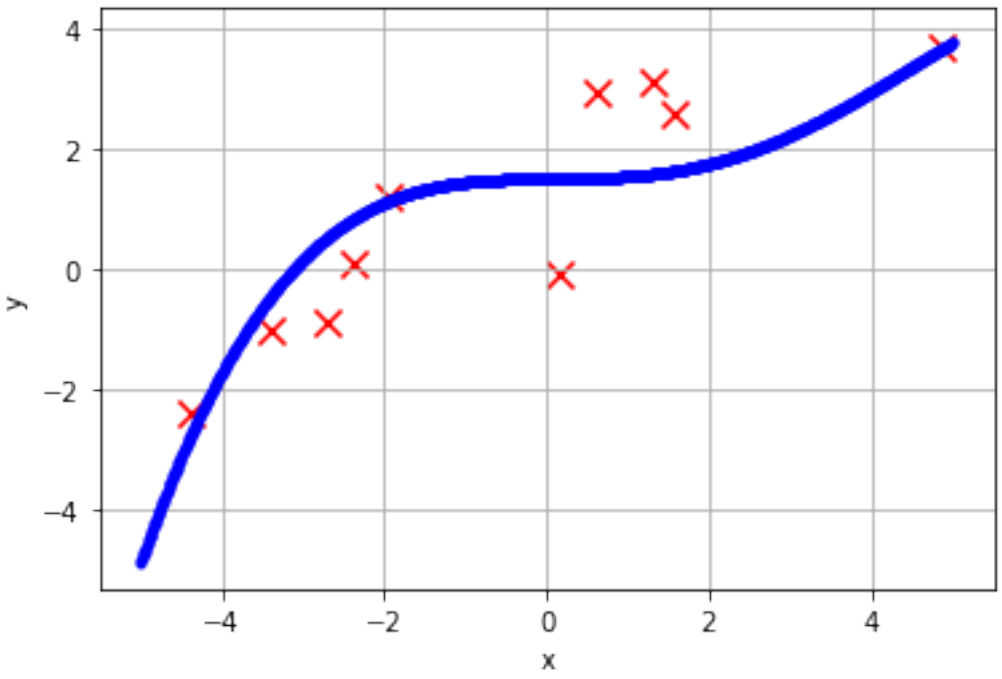
```
                    of instances and D is the dimensionality of each
                    instance
                ytrain: Nx1 numpy array, the true labels
            Return:
                weight: Dx1 numpy array, the weights of linear regression model
            """
            # strictly following
            # https://mahdi-roozbahani.github.io/CS46417641-fall2019/course/
 ↪15-regularized-regression.pdf page 20

            theta = np.array([[0.0],[0.0],[0.0],[0.0],[0.0],[0.0]])

            # lambda doesn't need to be multiplied by identity matrix according to␣
 ↪formula
#             D = xtrain.shape[1]
#             lambda_mat = np.zeros((D, D))
#             np.fill_diagonal(lambda_mat, c_lambda)
#             lambda_mat[:, 0] = 0

            for i in range(0, epochs, 1):
                y_pred = np.dot(xtrain, theta)
#                 d_theta = (-1) * np.matrix(xtrain.T) * np.matrix(ytrain - y_pred)␣
 ↪+ np.matmul(lambda_mat.T, theta)
                d_theta = (-1) * np.matrix(xtrain.T) * np.matrix(ytrain - y_pred) +␣
 ↪c_lambda * theta
                theta = theta - (learning_rate * d_theta).reshape(6, 1)
            weight = theta

            return weight


        @staticmethod
        def fit_SGD(xtrain, ytrain, c_lambda, epochs=100, learning_rate=0.001):
            """
            Args:
                xtrain: NxD numpy array, where N is number
                        of instances and D is the dimensionality of each
                        instance
                ytrain: Nx1 numpy array, the true labels
            Return:
                weight: Dx1 numpy array, the weights of linear regression model
            """
            N = xtrain.shape[0]

#             D = xtrain.shape[1]
#             lambda_mat = np.zeros((D, D))
#             np.fill_diagonal(lambda_mat, c_lambda)
```

```
#         lambda_mat[:, 0] = 0

        theta = np.array([[0.0],[0.0],[0.0],[0.0],[0.0],[0.0]])
        for i in range(0, epochs, 1):
            for r in range(0, N, 1):
                y_pred_r = np.dot(xtrain[r], theta)
                d_theta = (-1) * (xtrain[r].T * (ytrain[r] - y_pred_r)).
 ↪reshape(6, 1) + c_lambda * theta
#                 d_theta = (-1) * ((xtrain[r].T) * (ytrain[r] - y_pred_r)).
 ↪reshape(6, 1) + np.matmul(lambda_mat.T, theta)
                theta = theta - (learning_rate * d_theta).reshape(6, 1)

        weight = theta
        return weight
```

Again, let's see what we have learned. You only need to run the cell corresponding to your specific implementation.

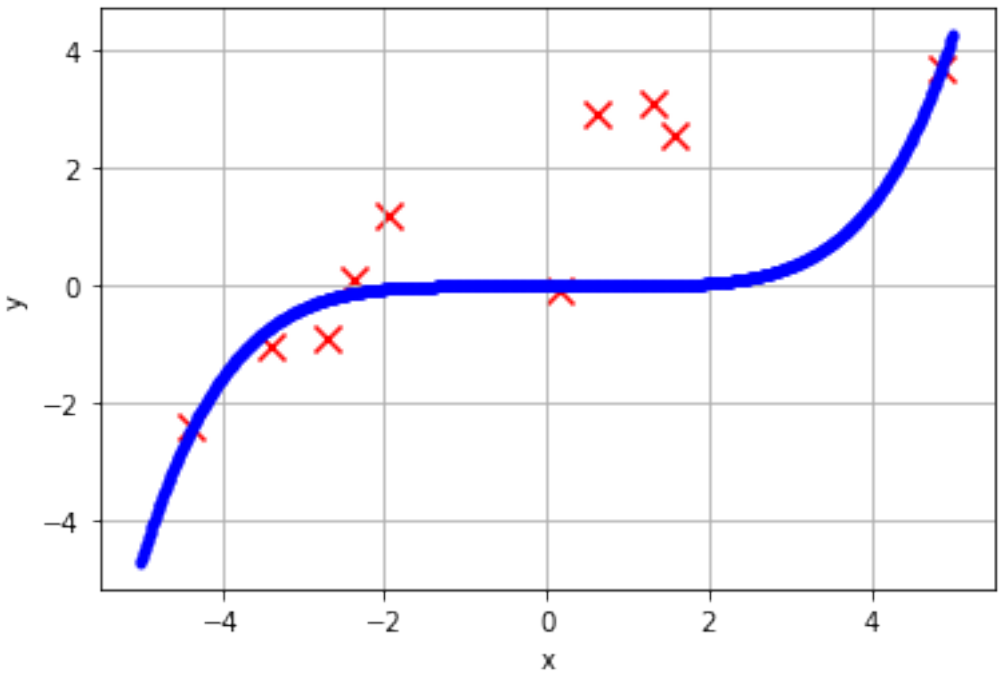
[19]:
```
sub_train = train_indices[:10]
weight = RidgeReg.fit_closed(x_all_feat[sub_train], y_all[sub_train],␣
 ↪c_lambda=1000)

y_pred = RidgeReg.predict(x_all_feat, weight)
plot_curve(x_all, y_pred)
plt.scatter(x_all[sub_train], y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = RidgeReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 1.2115
```



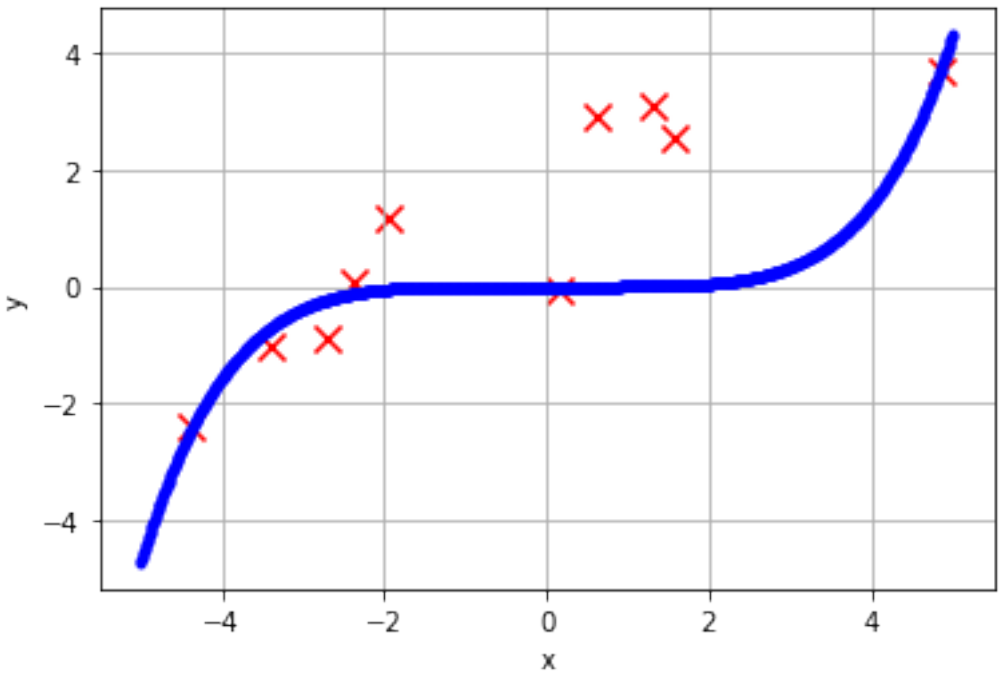
[20]:
```
sub_train = train_indices[:10]
weight = RidgeReg.fit_SGD(x_all_feat[sub_train], y_all[sub_train],␣
 ↪c_lambda=1000, learning_rate=1e-7)

y_pred = RidgeReg.predict(x_all_feat, weight)
plot_curve(x_all, y_pred)
plt.scatter(x_all[sub_train], y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = RidgeReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 1.6666
```



[21]:
```
sub_train = train_indices[:10]
weight = RidgeReg.fit_GD(x_all_feat[sub_train], y_all[sub_train],␣
 ↪c_lambda=1000, learning_rate=1e-7)

y_pred = RidgeReg.predict(x_all_feat, weight)
plot_curve(x_all, y_pred)
plt.scatter(x_all[sub_train], y_all[sub_train], s=100, c='r', marker='x')

y_test_pred = RidgeReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
test rmse: 1.6660
```



2.4 Cross Validation [10pts] Let's use Cross Validation to find the best value for c_lambda.

[22]:
```
# We provided 6 possible values for lambda, and you will use them in cross␣
 ↪validation.
# For cross validation, use 10-fold method and only use it for your training␣
 ↪data (you already have the train_indices to get training data).
# For the training data, split them in 10 folds which means that use 10 percent␣
 ↪of training data for test and 90 percent for training.
# At the end for each lambda, you have caluclated 10 rmse and get the mean␣
 ↪value of that.
# That's it. Pick up the lambda with the lowest mean value of rmse.
# Hint: np.concatenate is your friend.

def cross_validation(X, y, kfold, c_lambda):
    N = X.shape[0]
    interval = N // kfold
    rmse_arr = np.zeros((kfold, 1))

    for i in range(0, kfold, 1):
        X_copy = X
        X_test = X_copy[i * interval : i * interval + interval, :]
        X_train = np.delete(X_copy, slice(i * interval, i * interval +␣
 ↪interval), axis=0)
```

```
        y_copy = y
        y_test = y_copy[i * interval : i * interval + interval, :]
        y_train = np.delete(y_copy, slice(i * interval, i * interval +␣
 ↪interval), axis=0)

        weight = RidgeReg.fit_closed(X_train, y_train, c_lambda)
        y_test_pred = RidgeReg.predict(X_test, weight)
        rmse_arr[i] = rmse(y_test_pred, y_test)
        error = rmse_arr.mean()

    return error

best_lambda = None
best_error = None
kfold = 10
lambda_list = [0, 0.1, 1, 5, 10, 100, 1000]
for lm in lambda_list:
    err = cross_validation(x_all_feat[train_indices], y_all[train_indices],␣
 ↪kfold, lm)
    print('lambda: %.2f' % lm, 'error: %.6f'% err)
    if best_error is None or err < best_error:
        best_error = err
        best_lambda = lm

print('best_lambda: %.2f' % best_lambda)

weight = RidgeReg.fit_closed(x_all_feat[train_indices], y_all[train_indices],␣
 ↪c_lambda=10)
y_test_pred = RidgeReg.predict(x_all_feat[test_indices], weight)
test_rmse = rmse(y_test_pred, y_all[test_indices])
print('test rmse: %.4f' % test_rmse)
```

```
lambda: 0.00 error: 0.999078
lambda: 0.10 error: 0.999076
lambda: 1.00 error: 0.999057
lambda: 5.00 error: 0.998997
lambda: 10.00 error: 0.998973
lambda: 100.00 error: 1.003054
lambda: 1000.00 error: 1.029322
best_lambda: 10.00
test rmse: 0.9227
```

1.5 3 PCA analysis [25 pts]

In this problem, we will investigate how PCA can be used to improve features for regression and classification tasks and how the data itself affects the behavior of PCA.

Intrinsic Dimensionality [15pts]
Assume a dataset is composed of N datapoints, each of which has D features with D < N. The *dimension* of our data would be D. It is possible, however, that many of these dimensions contain redundant information. The *intrinsic dimensionality* is the number of dimensions we need to reconstruct our data with high fidelity. For our purposes, we will define the intrinsic dimension as the number of principal components needed to reconstruct 99% of the variation within our data.

We define a set of features as linearly independent if we cannot construct one out of a linear combination of the others. The number of linearly independent features is the number of nonzero principal components (where we define 0 is anything less than 10^{-11} due to floating point error). Zero principal components mean that we can not find any weights to linearly combine features in order to create an indenpendent feature. Thus, our algorithm will assign 0 to these weights.

[23]:
```
def pca(X):
    """
    Decompose dataset into principal components.
    You may use your SVD function from the previous part in your implementation.

    Args:
        X: N*D array corresponding to a dataset
    Return:
        U: N*N
        S: min(N, D)*1
        V: D*D
    """
    # center x
#     X_centered = X - X.mean(axis = 0) this gives different result than sample␣
 ↪answer

    U, S, V = svd(X)

    return U, S, V

def intrinsic_dimension(S, recovered_variance=.99):
    """
    Find the number of principal components necessary to recover given␣
 ↪proportion of variance

    Args:
        S: 1-d array corresponding to the singular values of a dataset

        recovered_varaiance: float in [0,1].  Minimum amount of variance
            to recover from given principal components
```

```
    Return:
        dim: int, the number of principal components necessary to recover
            the given proportion of the variance
    """

    dim = 0

    for i in range(0, S.shape[0]):

        recovered_var_i = (S[:i] ** 2).sum() / (S ** 2).sum()

        if recovered_var_i >= recovered_variance:
            dim = i
            break

    return dim

def num_linearly_ind_features(S, eps=1e-11):
    """
    Find the number of linearly independent featues in dataset

    Args:
        S: 1-d array corresponding to the singular values of a dataset
    Return:
        dim: int, the number of linearly independent dimensions in our data
    """
    dim = 0

    for s in S:
        if s >= eps:
            dim += 1

    return dim
```

Use your above functions to find the intrinsic dimensionality and number of linearly independent components in the following datasets: * Digits (handwritten digits) * Breast Cancer * Boston Housing * Diabetes

[53]:
```
#helper, don't need to change
digits = load_digits()
cancer = load_breast_cancer()
boston = load_boston()
diabetes = load_diabetes()
```

[25]:
```
print("Total Features")
print("\tDigits", digits.data.shape[1])
print("\tBreast Cancer", cancer.data.shape[1])
```

```
print('\tBoston', boston.data.shape[1])
print('\tDiabetes', diabetes.data.shape[1])
```

```
Total Features
        Digits 64
        Breast Cancer 30
        Boston 13
        Diabetes 10
```

[41]:
```
print("Linearly Independent Features:")
print("\tDigits", num_linearly_ind_features(pca(digits.data)[1]))
print("\tBreast Cancer", num_linearly_ind_features(pca(cancer.data)[1]))
print("\tBoston", num_linearly_ind_features(pca(boston.data)[1]))
print("\tDiabetes", num_linearly_ind_features(pca(diabetes.data)[1]))
```

```
Linearly Independent Features:
        Digits 61
        Breast Cancer 30
        Boston 13
        Diabetes 10
```

[42]:
```
print("Intrinsic Dimensionality:")
print("\tDigits", intrinsic_dimension(pca(digits.data)[1]))
print("\tBreast Cancer", intrinsic_dimension(pca(cancer.data)[1]))
print("\tBoston", intrinsic_dimension(pca(boston.data)[1]))
print("\tDiabetes", intrinsic_dimension(pca(diabetes.data)[1]))
```

```
Intrinsic Dimensionality:
        Digits 33
        Breast Cancer 1
        Boston 2
        Diabetes 8
```

Feature Scaling [10pts]

Principal component analysis is not agnostic to the scale of your features. Measuring a feature with different units can change your principal components.

For this problem, randomly choose one of your a column in each of the above datasets and multiply it by 1000. For each of the datasets, answer the following: 1. How does this change the distribution of variance among the first 10 components?
2. How does this change the first principal component of the data? 3. How does this affect the number of linearly independent components and intrinsic dimensionality? Why?

It may be helpful to plot the variance captured by each component in a scree plot (see function below) and to make a bar plot of the absolute value of each feature in the first principal component.

[43]:
```
def randomly_perturb_data(data, multiplier=1000):
    """
    Multiply a random column in data by multiplier
```

```
    Inputs:
        data: N*D numpy array of features
        multiplier: multiplier by which to perturb a random column in data

    Returns:
        perturbed_data: Data with random column multiplied by multiplier
    """

    D = data.shape[1]
    rand_col = np.random.randint(0, D)
    data[:, rand_col] = data[:, rand_col] * multiplier
    perturbed_data = data

    return perturbed_data


def scree_plot(S, n_components=10, ax=None):
    """
    Plot proportion of variance contained in each individual component
    """
    variance_list = np.zeros((n_components))

    for i in range(0, n_components):
        variance = (S[i] ** 2) / (S ** 2).sum() * 100
        variance_list[i] = variance

    components_list = [j for j in range(1, n_components + 1)]

    ax.set_xticks(np.arange(1, n_components + 1, step=1))
    ax.set_xlabel('n components')
    ax.set_ylabel('proportion (%) of variance in each component')
    return ax.plot(components_list, variance_list)


def plot_component_vector(V,ax=None):

    features_list = [i for i in range(1, V.shape[1] + 1)]
    ax.set_xlabel('features')
    ax.set_ylabel('absolute value')
    return ax.bar(features_list, np.absolute(V[0]))
```

Please write your analysis here:

Answer:

digits.data:

From the plots and new linearly independent features and intrinsic dimensionality for digits, we

see that: The distribution of variance changed from ~70% for the 1st component and ~5% for the 2nd component changed to almost 100% for the 1st component, and accordingly the intrinsic dimensionality decreases from 33 to 1.

The absolute value of each feature of the 1st component was more equally distributed for the original data, compared to the perturbed data, where 1 feature almost took up all of the absolute value ~1 (because we multiplied one of the columns of the data by 1000, this makes other columns negligible.)

The linearly independent components stays the same.

cancer.data:

The distribution of variance didn't change too much, as we can't really tell visually from the graph.

The absolute value of each feature of the first component was slightly changed, as some of the absolute values increase and some decrease, but the generally speaking it didn't change too much.

The number of linearly independent components and intrinsic dimensionality didn't change.

THe results for this dataset is different from the conclusion we get from other three datasets. By examining the data I found out there are some features with large value (4000+). That's why in original svd, 1 (or 2) of the components are already prominent enough, and multiplying one smaller column by 1000 didn't affect this component.

boston.data

This one is similar to digits.data. The distribution of variance changed a lot, the 1st component almost took up all variances ~100%, and accordingly, intrinsic dimensionality drops from 2 to 1.

The absolute value of each feature of the 1st component also changed. 1 particular feature became extremely large.

Number of linearly independent components didn't change.

diabetes.data

diabetes.data is also similar the digits.data. The distribution of variance was more equally distributed between each components, but once we multiply one column by 1000, the 1st components took up to almost 100% of total variance.

The absolute value of feature no. 10 became extremely large.

Number of linearly independent components didn't change. The intrinsic dimensionality drops from 8 to 1.

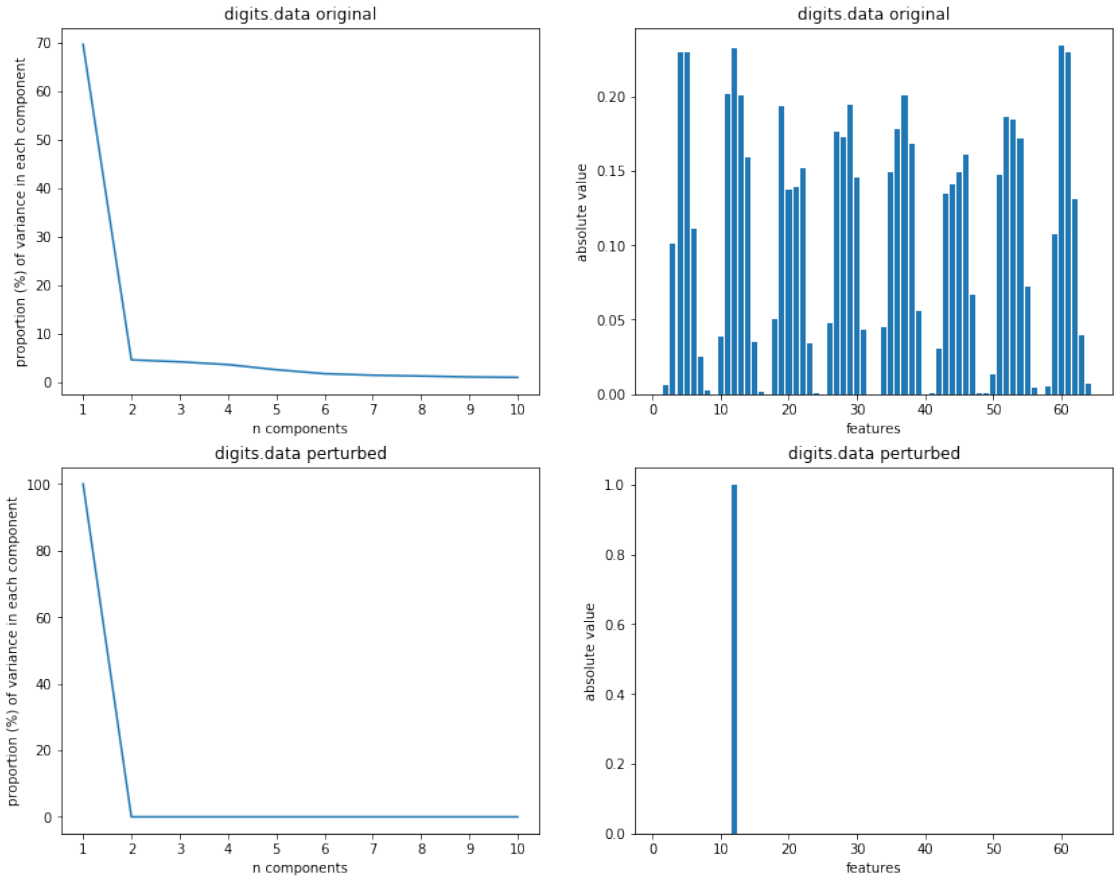
The general pattern is pretty clear in this case. PCA is subjective to scaling. Because we scale up a column by 1000, almost all of the recovered variance are contained in the 1st principal component. Mathematically speaking, when we scale up our feature, the covariance matrix changes, and since $Cov = V\Sigma^2 V^T$, V and Σ will change, too.

[44]:
```
# For digits.data

_, S, V = pca(digits.data)
```

```
digits_pertubred = randomly_perturb_data(digits.data)
_, S_perturbed, V_perturbed = pca(digits_pertubred)

print("Linearly Independent Features for Digits:")
print("\tBefore Perturbation", num_linearly_ind_features(S))
print("\tAfter Perturbation", num_linearly_ind_features(S_perturbed))

print("Intrinsic Dimensionality for Digits:")
print("\tBefore Perturbation", intrinsic_dimension(S))
print("\tAfter Perturbation", intrinsic_dimension(S_perturbed))

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, sharex=False,␣
 ↪sharey=False, figsize=(14,11))
scree_plot(S, n_components=10, ax=ax1)
ax1.set_title('digits.data original')

plot_component_vector(V, ax=ax2)
ax2.set_title('digits.data original')

scree_plot(S_perturbed, n_components=10, ax=ax3)
ax3.set_title('digits.data perturbed')

plot_component_vector(V_perturbed, ax=ax4)
ax4.set_title('digits.data perturbed')
```

```
Linearly Independent Features for Digits:
        Before Perturbation 61
        After Perturbation 64
Intrinsic Dimensionality for Digits:
        Before Perturbation 33
        After Perturbation 1
```

[44]: Text(0.5, 1.0, 'digits.data perturbed')

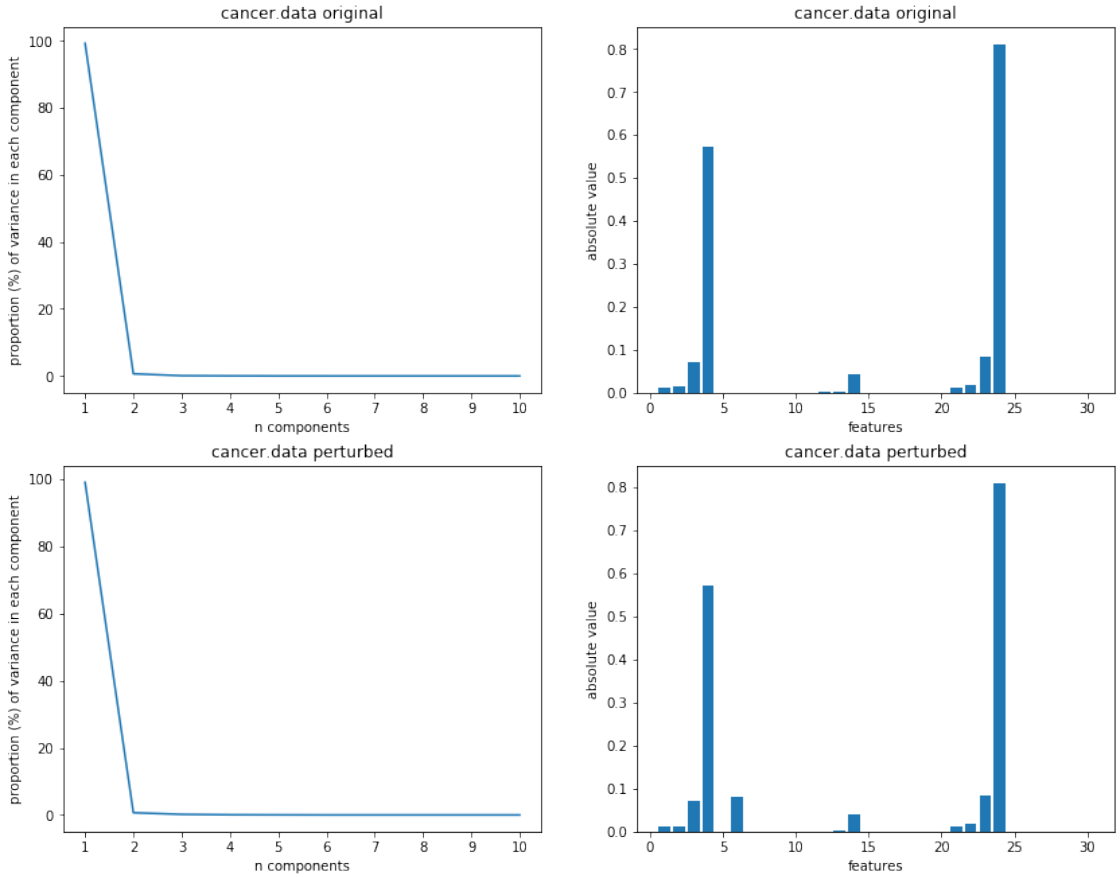


[45]:
```
# For cancer.data

_, S, V = pca(cancer.data)

cancer_pertubred = randomly_perturb_data(cancer.data)
_, S_perturbed, V_perturbed = pca(cancer_pertubred)

print("Linearly Independent Features for Breast Cancer:")
print("\tBefore Perturbation", num_linearly_ind_features(S))
print("\tAfter Perturbation", num_linearly_ind_features(S_perturbed))

print("Intrinsic Dimensionality for Breast Cancer:")
print("\tBefore Perturbation", intrinsic_dimension(S))
print("\tAfter Perturbation", intrinsic_dimension(S_perturbed))

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, sharex=False,␣
 ↪sharey=False, figsize=(14,11))
scree_plot(S, n_components=10, ax=ax1)
ax1.set_title('cancer.data original')
```

```
plot_component_vector(V, ax=ax2)
ax2.set_title('cancer.data original')

scree_plot(S_perturbed, n_components=10, ax=ax3)
ax3.set_title('cancer.data perturbed')

plot_component_vector(V_perturbed, ax=ax4)
ax4.set_title('cancer.data perturbed')
```

```
Linearly Independent Features for Breast Cancer:
        Before Perturbation 30
        After Perturbation 30
Intrinsic Dimensionality for Breast Cancer:
        Before Perturbation 1
        After Perturbation 1
```

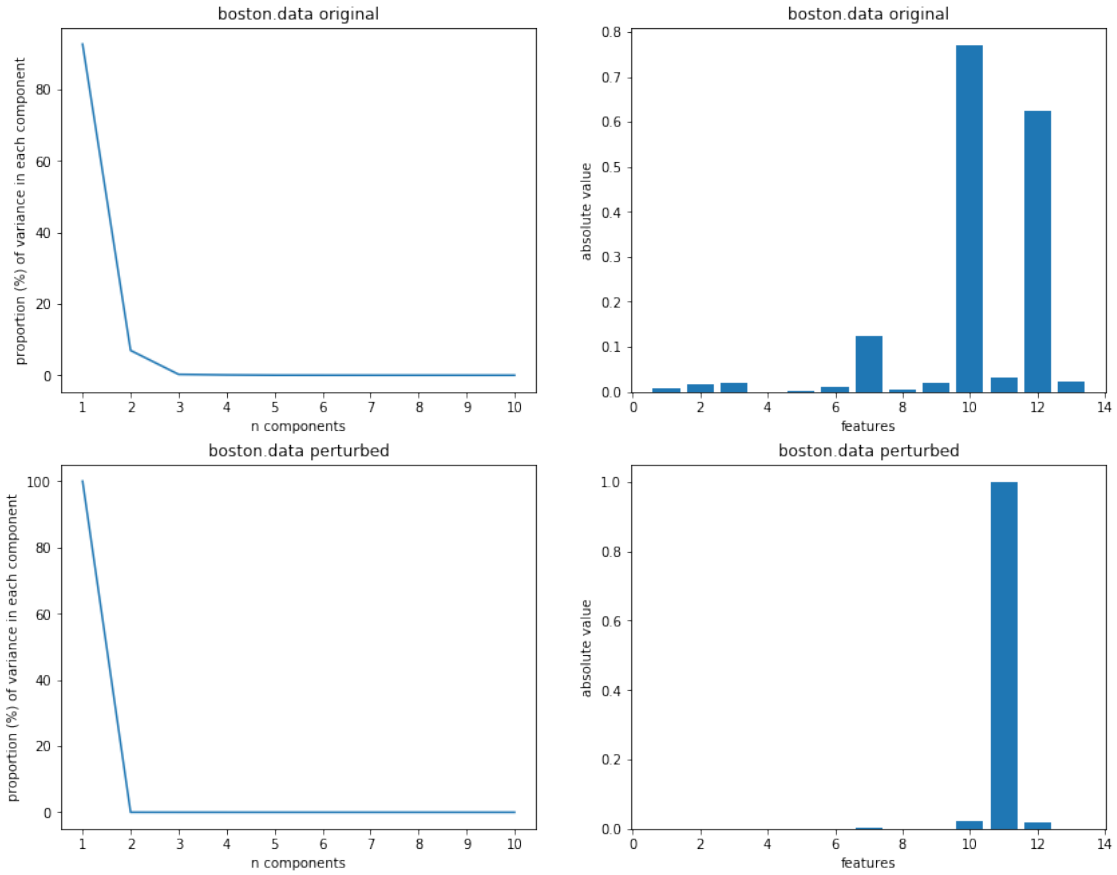
[45]: `Text(0.5, 1.0, 'cancer.data perturbed')`



[54]:
```
# for boston.data

_, S, V = pca(boston.data)

boston_pertubred = randomly_perturb_data(boston.data)
_, S_perturbed, V_perturbed = pca(boston_pertubred)

print("Linearly Independent Features for Boston:")
print("\tBefore Perturbation", num_linearly_ind_features(S))
print("\tAfter Perturbation", num_linearly_ind_features(S_perturbed))

print("Intrinsic Dimensionality for Boston:")
print("\tBefore Perturbation", intrinsic_dimension(S))
print("\tAfter Perturbation", intrinsic_dimension(S_perturbed))

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, sharex=False,␣
 ↪sharey=False, figsize=(14,11))
scree_plot(S, n_components=10, ax=ax1)
ax1.set_title('boston.data original')

plot_component_vector(V, ax=ax2)
ax2.set_title('boston.data original')

scree_plot(S_perturbed, n_components=10, ax=ax3)
ax3.set_title('boston.data perturbed')

plot_component_vector(V_perturbed, ax=ax4)
ax4.set_title('boston.data perturbed')
```

```
Linearly Independent Features for Boston:
        Before Perturbation 13
        After Perturbation 13
Intrinsic Dimensionality for Boston:
        Before Perturbation 2
        After Perturbation 1
```

[54]:
```
Text(0.5, 1.0, 'boston.data perturbed')
```

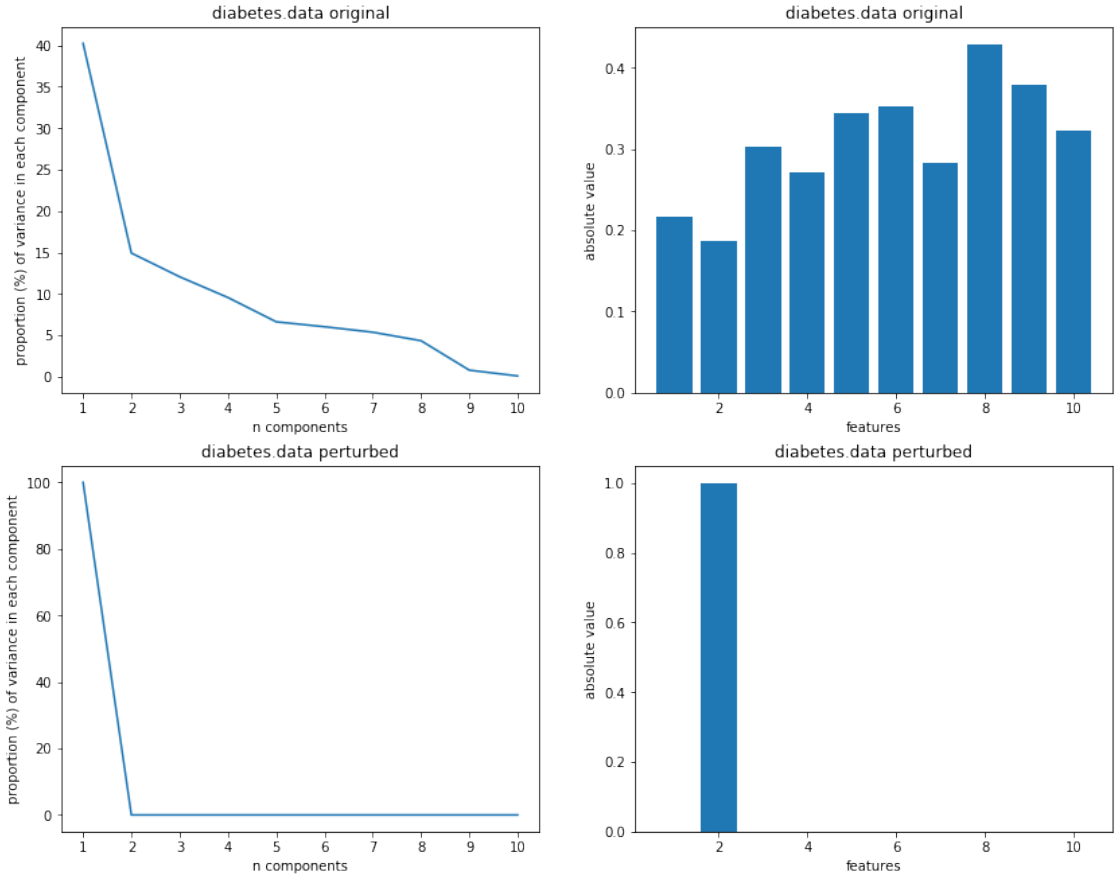


[48]:
```
# for diabetes.data

_, S, V = pca(diabetes.data)

diabetes_pertubred = randomly_perturb_data(diabetes.data)
_, S_perturbed, V_perturbed = pca(diabetes_pertubred)

print("Linearly Independent Features for Diabetes:")
print("\tBefore Perturbation", num_linearly_ind_features(S))
print("\tAfter Perturbation", num_linearly_ind_features(S_perturbed))

print("Intrinsic Dimensionality for Diabetes:")
print("\tBefore Perturbation", intrinsic_dimension(S))
print("\tAfter Perturbation", intrinsic_dimension(S_perturbed))

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2, sharex=False,␣
 ↪sharey=False, figsize=(14,11))
scree_plot(S, n_components=10, ax=ax1)
ax1.set_title('diabetes.data original')
```

```
plot_component_vector(V, ax=ax2)
ax2.set_title('diabetes.data original')

scree_plot(S_perturbed, n_components=10, ax=ax3)
ax3.set_title('diabetes.data perturbed')

plot_component_vector(V_perturbed, ax=ax4)
ax4.set_title('diabetes.data perturbed')
```

```
Linearly Independent Features for Diabetes:
        Before Perturbation 10
        After Perturbation 10
Intrinsic Dimensionality for Diabetes:
        Before Perturbation 8
        After Perturbation 1
```

[48]: `Text(0.5, 1.0, 'diabetes.data perturbed')`



1.6 4 Comparison of PCA and LDA [20pts]

We will now look at the results obtained using PCA and compare it to those obtained using **Linear Discriminant Analysis** (LDA). PCA is an unsupervised dimensionality reduction method, whereas LDA is used to find a linear decision boundary for multi-class classification problems. It achieves this by projecting the data onto a lower dimensional subspace to maximize class separability. Due to this, it can also be used for dimensionality reduction. For more details about LDA, refer to section 4.3 of the book Elements of Statistical Learning (https://web.stanford.edu/~hastie/Papers/ESLII.pdf).

For this question, you will be allowed to use the PCA and LDA functions from the **scikit-learn library**. You will be working with the wine dataset. Your goal is to use PCA and LDA to project the data into a two-dimensional subspace and visualize your results using a scatter plot. You should also output the explained variance ratio of the two components that you find using PCA.

[49]:
```
# helper function, do not need to change
def load_wine_data():
    from sklearn.datasets import load_wine
    wine = load_wine()
    X, y = wine.data, wine.target
    return X, y #X, y: Feature data and targets of wine dataset

X, y = load_wine_data()
```

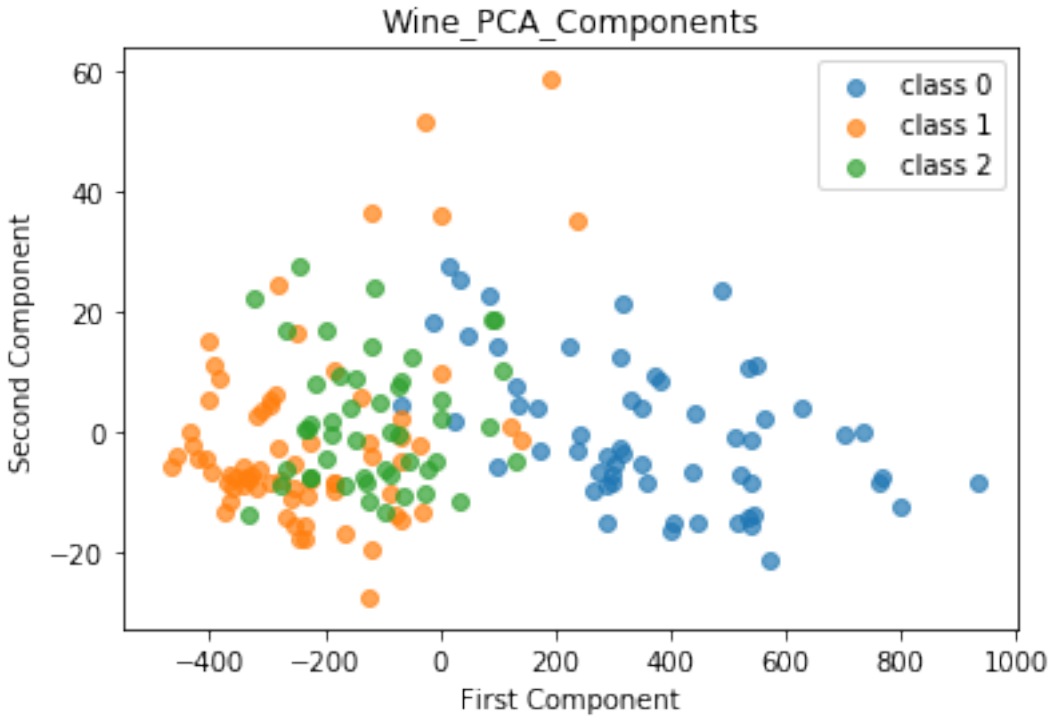
[50]:
```
def PCA(X, y):
    '''
    Use this function to obtain two PCA components which are later used for␣
 ↪plotting
    Returns:
        wine_pca: The PCA components
        explained_variance_ratio: Explained variance ratio of the two PCA␣
 ↪components
        (Hint:use sklearn function)
    '''

    from sklearn.decomposition import PCA

    pca = PCA(n_components=2)
    pca.fit(X)

    explained_variance_ratio = pca.explained_variance_ratio_
    wine_pca = pca.transform(X)

    return wine_pca, explained_variance_ratio

wine_pca, explained_variance_ratio = PCA(X, y)
print('Explained variance ratio of the first two components: %s'
      % str(explained_variance_ratio))
```

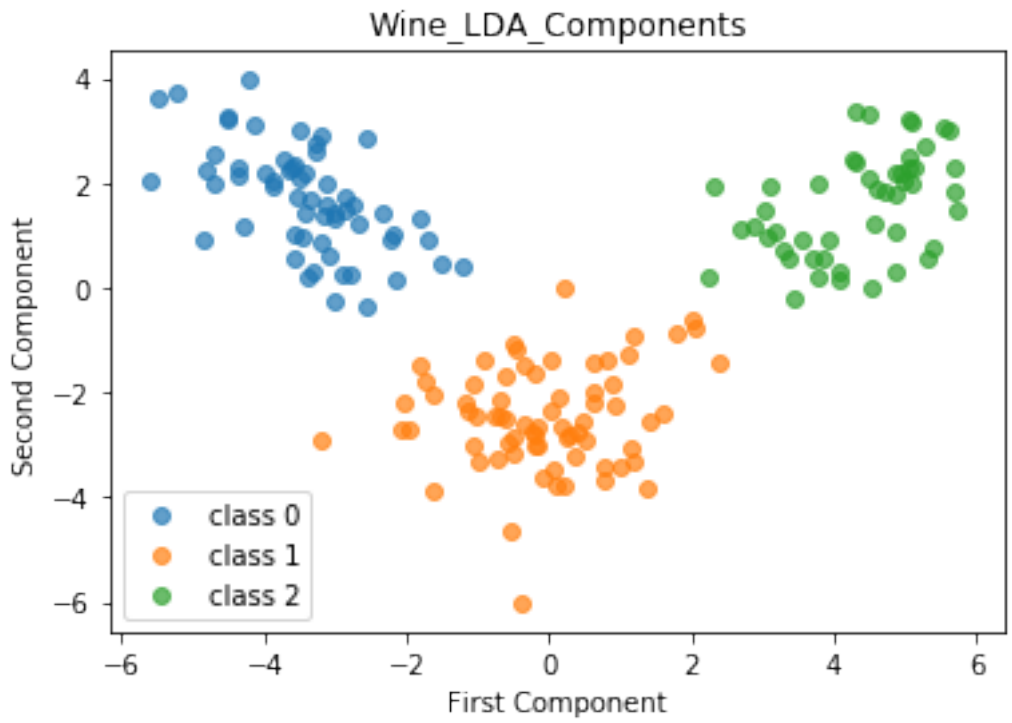
```
Explained variance ratio of the first two components: [0.99809123 0.00173592]
```

[51]:
```python
def LDA(X, y):
    '''
    Use this function to obtain 2 (two) LDA components which are later used for␣
 ↪plotting
    Returns:
        wine_lda: The LDA components
    '''

    from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
    lda_model = LinearDiscriminantAnalysis(n_components=2)
    lda_model.fit(X, y)

    wine_lda = lda_model.transform(X)

    return wine_lda

wine_lda = LDA(X, y)
```

[52]:
```python
def plot_PCA_LDA(wine_pca, wine_lda):
    """
    Use a scatter plot to plot PCA and LDA components obtained using the␣
 ↪functions above.
    Use a different color for each class.
    """

    # PCA
    # split the data into three subsets by class, it's easier this way for us␣
 ↪to show legends.
    wine_pca_class0 = wine_pca[y == 0]
    wine_pca_class1 = wine_pca[y == 1]
    wine_pca_class2 = wine_pca[y == 2]

    plt.scatter(wine_pca_class0[:,0], wine_pca_class0[:,1], alpha=0.7,␣
 ↪cmap='viridis', label = 'class 0')
    plt.scatter(wine_pca_class1[:,0], wine_pca_class1[:,1], alpha=0.7,␣
 ↪cmap='viridis', label = 'class 1')
    plt.scatter(wine_pca_class2[:,0], wine_pca_class2[:,1], alpha=0.7,␣
 ↪cmap='viridis', label = 'class 2')

    plt.xlabel('First Component')
    plt.ylabel('Second Component')
    plt.title('Wine_PCA_Components')
    plt.legend()
    plt.show()
```

```
    # LDA
    wine_lda_class0 = wine_lda[y == 0]
    wine_lda_class1 = wine_lda[y == 1]
    wine_lda_class2 = wine_lda[y == 2]

    plt.scatter(wine_lda_class0[:,0], wine_lda_class0[:,1], alpha=0.7,␣
↪cmap='viridis', label = 'class 0')
    plt.scatter(wine_lda_class1[:,0], wine_lda_class1[:,1], alpha=0.7,␣
↪cmap='viridis', label = 'class 1')
    plt.scatter(wine_lda_class2[:,0], wine_lda_class2[:,1], alpha=0.7,␣
↪cmap='viridis', label = 'class 2')

    plt.xlabel('First Component')
    plt.ylabel('Second Component')
    plt.title('Wine_LDA_Components')
    plt.legend()
    plt.show()


plot_PCA_LDA(wine_pca, wine_lda)
```





Which dimensionality reduction method is better? Why [5pts]

Answer: From the two plots shown above, the LDA method is better. After dimension reduction, wine data using LDA are clearly separated into three clusters, each of which are distinctively separated without outliers (class 0 point in class 1 cluster, etc). There are also rooms of margin between them. LDA reduction makes it easier to differentiate data and classes if we have a new data point to test. However, in PDA plot, data points are overlapped with each other, we can't really tell them apart, i.e, there's no obvious x1, x2 thresholds to classify data. Although class 0 seems to be a little bit easier to see, it still has some points overlapped with other 2 classes.

End