Spring 2022 CS4641/CS7641 A Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: Thursday, March 31, 11:59 pm AOE

- No unapproved extension of the deadline is allowed. Late submission will lead to 0 credit.
- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be done individually.
- Plagiarism is a **serious offense**. You are responsible for completing your own work. You are not allowed to copy and paste, or paraphrase, or submit materials created or published by others, as if you created the materials. All materials submitted must be your own.
- All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures. If we observe any (even small) similarities/plagiarisms detected by Gradescope or our TAs, **WE WILL DIRECTLY REPORT ALL CASES TO OSI**, which may, unfortunately, lead to a very harsh outcome. **Consequences can be severe, e.g., academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

Instructions for the assignment

- This assignment consists of both programming and theory questions.
- Unless a theory question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You can directly type Latex equations into markdown cells.
- If a question requires a picture, you could use this syntax `<img src="" style="width: 300px;"/>` to include them within your ipython notebook.
- Your write up must be submitted in PDF form. You may use either Latex, markdown, or any word processing software. We will **NOT** accept handwritten work. Make sure that your work is formatted correctly, for example submit $\sum_{i=0} x_i$ instead of \text{sum_{i=0} x_i}
- When submitting the non-programming part of your assignment, you must correctly map pages of your PDF to each question/subquestion to reflect where they appear. **Improperly mapped questions may not be graded correctly and/or will result in point deductions for the error.**
- All assignments should be done individually, and each student must write up and submit their own answers.
- **Graduate Students**: You are required to complete any sections marked as Bonus for Undergrads

Using the autograder

- Grads will find three assignments on Gradescope that correspond to HW3: "Assignment 3 Programming", "Assignment 3 - Non-programming" and "Assignment 3 Programming - Bonus for all". Undergrads will find an additional assignment called "Assignment 3 Programming - Bonus for Undergrads".
- You will submit your code for the autograder in the Assignment 3 Programming sections. Please refer to the **Deliverables and Point Distribution** section to find what parts are considered required, bonus for undergrads, and bonus for all.
- We provided you different .py files, and we added libraries in those files. Please DO NOT remove those lines and add your code after those lines. Note that these are the only allowed libraries that you can use for the homework.
- You are allowed to make as many submissions until the deadline as you like. Additionally, note that the autograder will test each function separately, so it can serve as a useful tool to help you debug your code if you are not sure of what part of your implementation might have an issue.
- **For the "Assignment 3 - Non-programming" part, you will download your Jupyter Notebook as HTML and submit it as a PDF on Gradescope. To download the notebook as HTML, click on "File" on the top left corner of this page and select "Download as > HTML". Then, open the HTML file and print to PDF.** Please refer to the **Deliverables and Point Distribution** section for an outline of the non-programming questions.
- **When submitting to Gradescope, please make sure to mark the page(s) corresponding to each problem/sub-problem.**

Deliverables and Points Distribution

Q1: Image Compression [30pts]

Deliverables: imgcompression.py and printed results

- **1.1 Image Compression** [20 pts] - *programming*
  - svd [5pts]
  - rebuild_svd [5pts]
  - compression_ratio [5pts]
  - recovered_variance_proportion [5pts]
- **1.2 Black and White** [5 pts] *non-programming*
- **1.3 Color Image** [5 pts] *non-programming*

Q2: Understanding PCA [20pts]

Deliverables: pca.py and written portion

- **2.1 PCA Implementation** [10 pts] - *programming*
  - fit [5pts]
  - transform [2pts]
  - transform_rv [3pts]
- **2.2 Visualize** [5 pts] *non-programming*
- **2.3 Weaknesses of PCA** [5 pts] *non-programming*

Q3: Regression and Regularization [60 + (20 bonus for undergrads) pts]

Deliverables: regression.py and Written portion

- **3.1 Regression and Regularization Implementations** [30 pts + 20 pts Bonus for Undergrad] - *programming*
  - RMSE [5pts]
  - Construct Poly Features 1D [2pts]
  - Construct Poly Features 2D [3pts]
  - Prediction [5pts]
  - Linear Fit Closed Form [5pts]
  - Ridge Fit Closed Form [5pts]
  - Cross Validation [5pts]
  - Linear Gradient Descent [5pts] *Bonus for Undergrad*
  - Linear Stochastic Gradient Descent [5pts] *Bonus for Undergrad*
  - Ridge Gradient Descent [5pts] *Bonus for Undergrad*
  - Ridge Stochastic Gradient Descent [5pts] *Bonus for Undergrad*
- **3.2 About RMSE** [3 pts] *non-programming*
- **3.3 Testing: General Functions and Linear Regression** [5 pts] *non-programming*
- **3.4 Testing: Ridge Regression** [5 pts] *non-programming*
- **3.5 Cross Validation** [7 pts] *non-programming*
- **3.6 Noisy Input Samples in Linear Regression** [10 pts] *non-programming*

Q4: Naive Bayes Classification [25pts]

Deliverables: nb.py and Written portion

- **4.1 Naive Bayes in Marketing** [5 pts] *non-programming*
- **4.2 Amazon Product Ratings from Product Reviews** [15 pts] - *programming*
  - priors_prob [6pts]
  - likelihood_ratio [6pts]
  - analyze_star_rating [3pts]
- **4.3 Accuracy result analysis** [5 pts] *non-programming*

Q5: Noise in PCA and Linear Regression [15pts]

Deliverables: Written portion

- **5.1 Slope Functions** [5 pts] *non-programming*
- **5.2 Error in Y and Error in X and Y** [5 pts] *non-programming*
- **5.3 Analysis** [5 pts] *non-programming*

Q6: Feature Reduction.py [25pts Bonus for All]

Deliverables: feature_reduction.py and Written portion

- **6.1 Feature Reduction** [18 pts] - *programming*
  - forward_selection [9pts]
  - backward_elimination [9pts]
- **6.2 Feature Selection - Discussion** [7 pts] *non-programming*

0 Set up

This notebook is tested under python 3.. (https://www.python.org/downloads/release/python-368/), and the corresponding packages can be downloaded from miniconda (https://docs.conda.io/en/latest/miniconda.html). You may also want to get yourself familiar with several packages:

- jupyter notebook (https://jupyter-notebook.readthedocs.io/en/stable/)
- numpy (https://docs.scipy.org/doc/numpy-1.15.1/user/quickstart.html)
- matplotlib (https://matplotlib.org/users/pyplot_tutorial.html)

- sklearn (https://matplotlib.org/users/pyplot_tutorial.html)
- Axes3D (https://matplotlib.org/users/pyplot_tutorial.html)

There is also a VS Code and Anaconda Setup Tutorial (https://edstem.org/us/courses/16925/discussion/995686) on Ed under the "Links" category

Please implement the functions that have `raise NotImplementedError` , and after you finish the coding, please delete or comment out `raise NotImplementedError` .

Library imports

In [1]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
# This is cell which sets up some of the modules you might need
# Please do not change the cell or import any additional packages.

import numpy as np
import pandas as pd
import json
import math
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.feature_extraction import text
from sklearn.datasets import load_boston, load_diabetes, load_digits, load_breast_cancer, load_iris, load_wine
from sklearn.linear_model import Ridge, LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, accuracy_score
import warnings

import re
import gzip
from tqdm.notebook import tqdm

warnings.filterwarnings('ignore')

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

Q1: Image Compression [30 pts]

Load images data and plot

In [2]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# load Image
image = plt.imread("./data/HW3_image_compression_sp22.jpg")/255
#plot image
fig = plt.figure(figsize=(10,10))
fig = plt.figure(figsize=(10, 10))
plt.imshow(image)
```

Out[2]: <matplotlib.image.AxesImage at 0x7fd14c97a550>

<Figure size 720x720 with 0 Axes>

In [3]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
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)
```

Out[3]: <matplotlib.image.AxesImage at 0x7fd1488447f0>

1.1 Image compression [20pts] [P]

SVD is a dimensionality reduction technique that allows us to compress images by throwing away the least important information.

Higher singular values capture greater variance and, thus, capture greater information from the corresponding singular vector. To perform image compression, apply SVD on each matrix and get rid of the small singular values to compress the image. The loss of information through this process is negligible, and the difference between the images can be hardly spotted.

For example, the proportion of variance captured by the first component is

$$\frac{\sigma_1^2}{\sum_{i=1}^{n} \sigma_i^2}$$

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

In the **imgcompression.py** file, complete the following functions:

- **svd**: You may use `np.linalg.svd` in this function, and although the function defaults this parameter to true, you may explicitly set `full_matrices=True` using the optional `full_matrices` parameter. Hint 2 may be useful.
- **rebuild_svd**
- **compression_ratio**: Hint 1 may be useful
- **recovered_variance_proportion**: Hint 1 may be useful

Hint 1: http://timbaumann.info/svd-image-compression-demo/ (http://timbaumann.info/svd-image-compression-demo/) is a useful article on image compression and compression ratio. You may find this article useful for implementing the functions compression_ratio and recovered_variance_proportion

Hint 2: If you have never used `np.linalg.svd`, it might be helpful to read Numpy's SVD documentation (https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html) and note the particularities of the V matrix and that it is returned already transposed.

Hint 3: The shape of S resulting from SVD may change depending on if N>D, N<D, or N=D. Therefore, when checking the shape of S, note that min(N,D) means the value should be equal to whichever is lower between N and D.

1.2 Black and white [5 pts] [W]

This question will use your implementation of the functions from Q1.1 to generate a set of images compressed to different degrees. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

Make sure these images are displayed when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment.

In [4]:

```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
from imgcompression import ImgCompression

imcompression = ImgCompression()
bw_image = rgb2gray(image)
U, S, V = imcompression.svd(bw_image)
component_num = [1,2,5,10,20,40,80,160,256]

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

# plot several images
i=0
for k in component_num:
    img_rebuild = imcompression.rebuild_svd(U, S, V, k)
    c = np.around(imcompression.compression_ratio(bw_image, k), 4)
    r = np.around(imcompression.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
```

1.3 Color image [5 pts] [W]

This section will use your implementation of the functions from Q1.1 to generate a set of images compressed to different degrees. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

Make sure these images are displayed when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment.

Note: You might get warning "Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)." This warning is acceptable since some of the pixels may go above 1.0 while rebuilding. You should see similar images to original even with such clipping.

Hint 1: Make sure your implementation of `recovered_variance_proportion` returns an array of 3 floats for a color image.
Hint 2: Try performing SVD on the individual color channels and then stack the individual channel U, S, V matrices.
Hint 3: You may need separate implementations for a color or grayscale image in the same function.

In [5]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
from imgcompression import ImgCompression

imcompression = ImgCompression()
U, S, V = imcompression.svd(image)

# component_num = [1,2,5,10,20,40,80,160,256]
component_num = [1,2,5,10,20,40,80,160,256]

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

# plot several images
i=0
for k in component_num:
    img_rebuild = np.clip(imcompression.rebuild_svd(U, S, V, k),0,1)
    c = np.around(imcompression.compression_ratio(image, k), 4)
    r = np.around(imcompression.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
```

Q2: Understanding PCA [20 pts]

2.1 Implementation [10 pts] [P]

Principal Component Analysis (https://en.wikipedia.org/wiki/Principal_component_analysis) (PCA) is another dimensionality reduction technique that reduces dimensions by eliminating 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. 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.

Implement PCA. In the **pca.py** file, complete the following functions:

- **fit**: You may use `np.linalg.svd`. Set `full_matrices=False`. Hint 1 may be useful.
- **transform**
- **transform_rv**: You may find `np.cumsum` helpful for this function.

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. However, it is possible that many of these dimensions contain redundant information. Each feature explains part of the variance in our dataset, and some features may explain more variance than others.

In the pca.py file, complete the PCA class by completing functions fit, transform, and transform_rv.

Hint 1: Make sure you remember to first center your data by subtracting the mean.

2.2 Visualize [5 pts] [W]

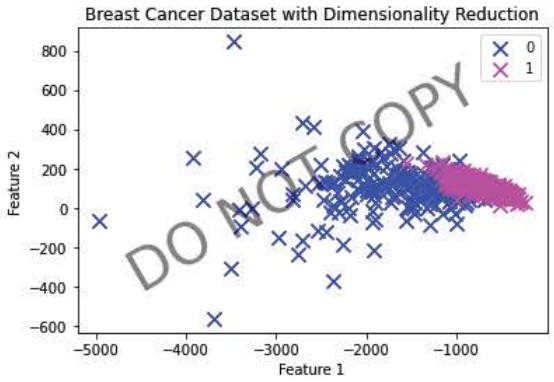
PCA is used to transform multivariate data tables into smaller sets so as to observe the hidden trends and variations in the data. Here you will visualize two datasets (iris and wine) using PCA. Use the above implementation of PCA and reduce the datasets such that they contain only two features. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q2.1.

Make 2-D scatter plots of the data points using these features. Make sure to differentiate the data points according to their true labels. The datasets have already been loaded for you. In addition, return the retained variance obtained from the reduced features.

In [6]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Use PCA for visualization of breast cancer data
from pca import PCA
bc_data = load_breast_cancer(return_X_y=True)

X = bc_data[0]
y = bc_data[1]

fig = plt.figure()
plt.title('Breast Cancer Dataset with Dimensionality Reduction')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
PCA().visualize(X,y,fig)
print('*In this plot, the 0 points are malignant and the 1 points are benign.')

```

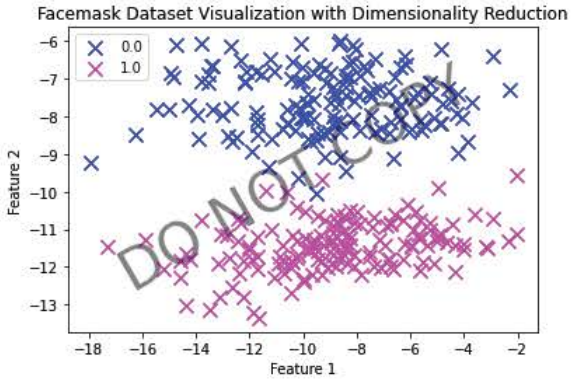
```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are igno
red when legend() is called with no argument.

data before PCA (569, 30)
data shape with PCA  (569, 2)
```




```
*In this plot, the 0 points are malignant and the 1 points are benign.
```

In [7]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Use PCA for visualization of masked and unmasked images

X = np.load('./data/smallflat.npy')
y = np.load('./data/masked_labels.npy')

fig = plt.figure()
plt.title('Facemask Dataset Visualization with Dimensionality Reduction')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
PCA().visualize(X,y, fig)
print('*In this plot, the 0 points are unmasked images and the 1 points are masked images.')

```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are igno
red when legend() is called with no argument.

data before PCA (300, 1024)
data shape with PCA  (300, 2)
```




```
*In this plot, the 0 points are unmasked images and the 1 points are masked images.
```

Notice the distinct separation between the data points with different labels in both plots above.

Now you will use PCA on an actual real-world dataset. We will use your implementation of PCA function to reduce the dataset with 99% retained variance and use it to obtain the reduced features. On the reduced dataset, we will use logistic and linear regression to compare results between PCA and non-PCA datasets. Run the following cells to see how PCA works on regression and classification tasks.

In [8]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
#load the dataset
digits = load_digits()

X = digits.data
y = digits.target

print("data shape before PCA ",X.shape)

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X)

print("data shape with PCA ",X_pca.shape)
```

```
data shape before PCA  (1797, 64)
data shape with PCA  (1797, 41)
```

In [9]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3,
                                                    stratify=y,
                                                    random_state=42)

# Use logistic regression to predict classes for test set
clf = LogisticRegression()
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy before PCA: {:.5f}'.format(accuracy_score(y_test,
                                                  preds.argmax(axis=1))))
```

```
Accuracy before PCA: 0.95741
```

In [10]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=.3,
                                                    stratify=y,
                                                    random_state=42)

# Use logistic regression to predict classes for test set
clf = LogisticRegression()
clf.fit(X_train, y_train)
preds = clf.predict_proba(X_test)
print('Accuracy after PCA: {:.5f}'.format(accuracy_score(y_test,
                                                  preds.argmax(axis=1))))
```

```
Accuracy after PCA: 0.95926
```

In [11]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
def apply_regression(X_train, y_train, X_test):
    ridge = Ridge()
    weight = ridge.fit(X_train, y_train)
    y_pred = ridge.predict(X_test)

    return y_pred
```

In [12]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
#load the dataset
diabetes = load_diabetes()
X = diabetes.data
y = diabetes.target

print(X.shape, y.shape)

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X, retained_variance = 0.9)
print("data shape with PCA ",X_pca.shape)
```

```
(442, 10) (442,)
data shape with PCA  (442, 7)
```

In [13]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# Train, test splits
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=42)

#Ridge regression without PCA
y_pred = apply_regression(X_train, y_train, X_test)

# calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print('RMSE score using Ridge Regression before PCA: {:.5}'.format(rmse_score))
```

```
RMSE score using Ridge Regression before PCA: 55.794
```

In [14]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
#Ridge regression with PCA
X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=.3, random_state=42)

#use Ridge Regression for getting predicted labels
y_pred = apply_regression(X_train,y_train,X_test)

#calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print('RMSE score using Ridge Regression after PCA: {:.5}'.format(rmse_score))
```

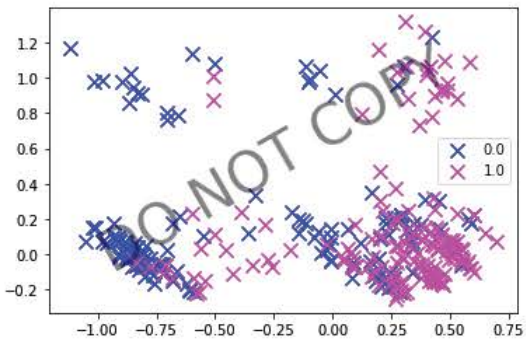
```
RMSE score using Ridge Regression after PCA: 55.725
```

2.3 Weakness of PCA [5 pts] [W]

Sometimes, PCA does not improve performance. Let's run PCA on a different dataset. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q2.1.

In [15]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
X = np.load('./data/heart_disease_features.npy')
y = np.load('./data/heart_disease_labels.npy')

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X, retained_variance = 0.9)
print("data shape with PCA ",X_pca.shape)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=42)

#Ridge regression without PCA
y_pred = apply_regression(X_train, y_train, X_test)

# calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print('RMSE score using Ridge Regression before PCA: {:.5}'.format(rmse_score))

X_train, X_test, y_train, y_test = train_test_split(X_pca, y, test_size=.3, random_state=42)

#use Ridge Regression for getting predicted labels
y_pred = apply_regression(X_train,y_train,X_test)

#calculate RMSE
rmse_score = np.sqrt(mean_squared_error(y_pred, y_test))
print('RMSE score using Ridge Regression after PCA: {:.5}'.format(rmse_score))

PCA().visualize(X,y)
```

```
data shape with PCA  (303, 8)
RMSE score using Ridge Regression before PCA: 0.39976
RMSE score using Ridge Regression after PCA: 0.40407

No artists with labels found to put in legend.  Note that artists whose label start with an underscore are igno
red when legend() is called with no argument.

data before PCA (303, 12)
data shape with PCA  (303, 2)
```



Provide one reason as to why PCA does not improve performance for this example

ANSWER:

3 Polynomial regression and regularization [60 pts + 20 pts bonus for CS 4641] [P] | [W]

3.1 Regression and regularization implementations [30 pts + 20 pts bonus for CS 4641] [P]

We have three methods to fit linear and ridge regression models: 1) closed form solution; 2) gradient descent (GD); 3) stochastic gradient descent (SGD). For undergraduate students, you are required to implement the closed form for linear regression and for ridge regression, but the others 4 methods are bonus parts. For graduate students, you are required to implement all of them. We use the term weight in the following code. Weights and parameters (θ) have the same meaning here. We used parameters (θ) in the lecture slides.

In the **regression.py** file, complete the Regression class by implementing the functions:

- **rmse**
- **construct_polynomial_feats**
- **predict**
- **linear_fit_closed**: You should use `np.linalg.pinv` in this function
- **linear_fit_GD** (bonus for undergrad)
- **linear_fit_SGD** (bonus for undergrad)
- **ridge_fit_closed**: You should adjust your `I` matrix to handle the bias term differently than the rest of the terms
- **ridge_fit_GD** (bonus for undergrad)
- **ridge_fit_SGD** (bonus for undergrad)

For graduate students, you are required to implement all of them. The points for each function is in **regression.py**.

3.2 About RMSE [3 pts] [W]

What is a good RMSE value? If we normalize our labels such that the outputs of our regression model can only be between 0 and 1, what does it mean when normalized RMSE = 1? Please provide an example with your explanation.

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.

YOUR SOLUTION HERE

3.3 Testing: general functions and linear regression [5 pts] [W]

In this section. we will test the performance of the linear regression. As long as your test RMSE score is close to the TA's answer (TA's answer ± 0.5), you can get full points. Let's first construct a dataset for polynomial regression.

In this case, we construct the polynomial features up to degree 5. Each data sample consists of two features $[a, b]$. We compute the polynomial features of both a and b in order to yield the vectors $[1, a, a^2, a^3, \dots, a^{degree}]$ and $[1, b, b^2, b^3, \dots, b^{degree}]$. We train our model with the cartesian product of these polynomial features. The cartesian product generates a new feature vector consisting of all polynomial combinations of the features with degree less than or equal to the specified degree.

For example, if degree = 2, we will have the polynomial features $[1, a, a^2]$ and $[1, b, b^2]$ for the datapoint $[a, b]$. The cartesian product of these two vectors will be $[1, a, b, ab, a^2, b^2]$. We do not generate a^3 and b^3 since their degree is greater than 2 (specified degree).

In [16]:
```
1 from regression import Regression
```

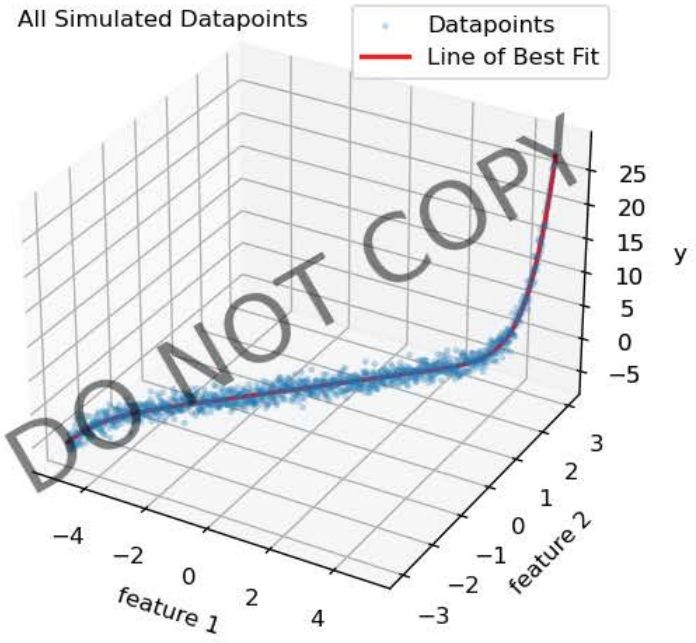
In [17]:

```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
POLY_DEGREE = 7
N_SAMPLES = 1200

rng = np.random.RandomState(seed=10)

# Simulating a regression dataset with polynomial features.
true_weight = rng.rand(POLY_DEGREE ** 2 + 2, 1)
x_feature1 = np.linspace(-5, 5, N_SAMPLES)
x_feature2 = np.linspace(-3, 3, N_SAMPLES)
x_all = np.stack((x_feature1, x_feature2), axis=1)

reg = Regression()
x_all_feat = reg.construct_polynomial_feats(x_all, POLY_DEGREE)
x_cart_flat = []
for i in range(x_all_feat.shape[0]):
    point = x_all_feat[i]
    x1 = point[:,0]
    x2 = point[:,1]
    x1_end = x1[-1]
    x2_end = x2[-1]
    x1 = x1[:-1]
    x2 = x2[:-1]
    x3 = np.asarray([[m*n for m in x1] for n in x2])

    x3_flat = list(np.reshape(x3, (x3.shape[0] ** 2)))
    x3_flat.append(x1_end)
    x3_flat.append(x2_end)
    x3_flat = np.asarray(x3_flat)
    x_cart_flat.append(x3_flat)

x_cart_flat = np.asarray(x_cart_flat)
x_cart_flat = (x_cart_flat - np.mean(x_cart_flat)) / np.std(x_cart_flat)  # Normalize
x_all_feat = np.copy(x_cart_flat)

# We must add noise to data, else the data will look unrealistically perfect.
y_noise = rng.randn(x_all_feat.shape[0], 1)
y_all = np.dot(x_cart_flat, true_weight) + y_noise
print("x_all: ", x_all.shape[0], " (rows/samples) ", x_all.shape[1], " (columns/features)", sep="")
print("y_all: ", y_all.shape[0], " (rows/samples) ", y_all.shape[1], " (columns/features)", sep="")
```

```
x_all: 1200 (rows/samples) 2 (columns/features)
y_all: 1200 (rows/samples) 1 (columns/features)
```

In [18]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

p = np.reshape(np.dot(x_cart_flat, true_weight), (N_SAMPLES,))
ax.scatter(x_all[:,0], x_all[:,1], y_all, label='Datapoints', s=4, alpha=0.2)
ax.plot(x_all[:,0], x_all[:,1], p, label='Line of Best Fit', c="red", linewidth=2)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.legend()
ax.text2D(0.05, 0.95, "All Simulated Datapoints", transform=ax.transAxes)
plt.show()
```

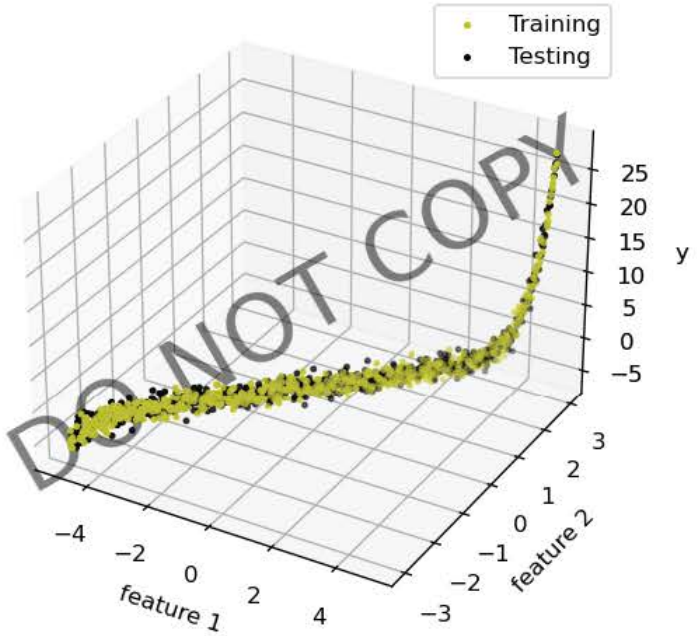


In the figure above, the red curve is the true fuction we want to learn, while the blue dots are the noisy data points. The data points are generated by $\boldsymbol{Y = X\theta + \epsilon}$, where $\boldsymbol{\epsilon_i \sim N(0,1)}$ are i.i.d. generated noise.

Now let's split the data into two parts, the training set and testing set. The yellow dots are for training, while the black dots are for testing.

In [19]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
PERCENT_TRAIN = 0.5

all_indices = rng.permutation(N_SAMPLES)  # Random indicies
train_indices = all_indices[:round(N_SAMPLES * PERCENT_TRAIN)]  # 80% Training
test_indices = all_indices[round(N_SAMPLES * PERCENT_TRAIN):]  # 20% Testing

xtrain = x_all[train_indices]
ytrain = y_all[train_indices]
xtest = x_all[test_indices]
ytest = y_all[test_indices]

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

ax.scatter(xtrain[:,0], xtrain[:,1], ytrain, label='Training', c='y',s=4)
ax.scatter(xtest[:,0], xtest[:,1], ytest, label='Testing', c='black',s=4)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.legend(loc = 'upper right')
plt.show()
```

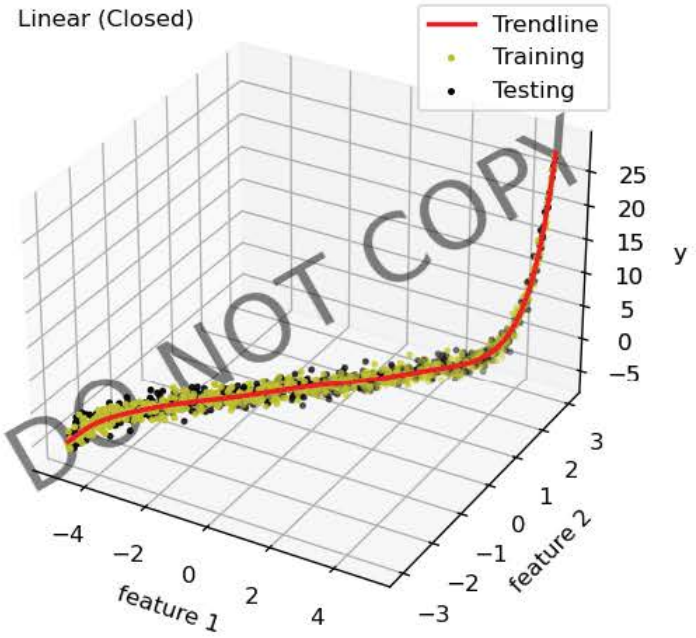


Now let us train our model using the training set and see how our model performs on the testing set. Observe the red line, which is our model's learned function.

In [20]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
weight = reg.linear_fit_closed(x_all_feat[train_indices], y_all[train_indices])
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('Linear (closed) RMSE: %.4f' % test_rmse)

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

y_pred = reg.predict(x_all_feat, weight)
y_pred = np.reshape(y_pred, (y_pred.size,))
ax.plot(x_all[:,0], x_all[:,1], y_pred, label='Trendline', color='r', lw=2, zorder=5)

ax.scatter(xtrain[:,0], xtrain[:,1], ytrain, label='Training', c='y',s=4)
ax.scatter(xtest[:,0], xtest[:,1], ytest, label='Testing', c='black',s=4)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.text2D(0.05, 0.95, "Linear (Closed)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (closed) RMSE: 1.0088
```

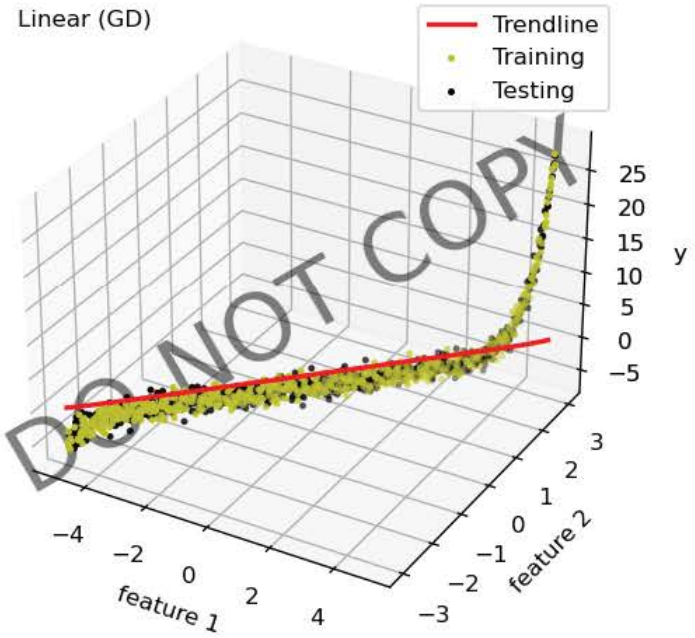


Now let's use our linear gradient descent function with the same setup. Observe that the trendline is now less optimal, and our RMSE decreased. Do not be alarmed.

In [21]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
#This cell may take more than 1 minute
weight, _ = reg.linear_fit_GD(x_all_feat[train_indices],
                              y_all[train_indices],
                              epochs=50000,
                              learning_rate=1e-8)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print('Linear (GD) RMSE: %.4f' % test_rmse)

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

y_pred = reg.predict(x_all_feat, weight)
y_pred = np.reshape(y_pred, (y_pred.size,))
ax.plot(x_all[:,0], x_all[:,1], y_pred, label='Trendline', color='r', lw=2, zorder=5)

ax.scatter(xtrain[:,0], xtrain[:,1], ytrain, label='Training', c='y',s=4)
ax.scatter(xtest[:,0], xtest[:,1], ytest, label='Testing', c='black',s=4)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.text2D(0.05, 0.95, "Linear (GD)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (GD) RMSE: 4.3573
```



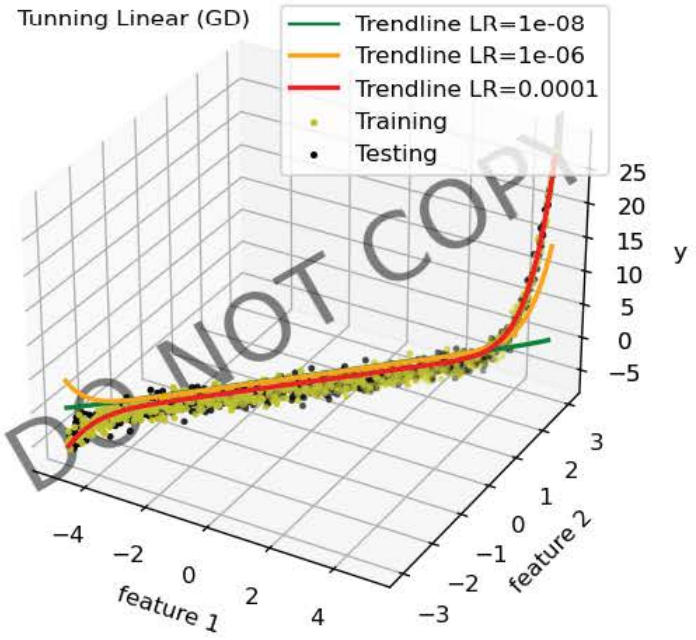
We must tune our epochs and learning_rate. As we tune these parameters our trendline will approach the trendline generated by the linear closed form solution. Observe how we slowly tune (increase) the epochs and learning_rate below to create a better model.

Note that the closed form solution will always give the most optimal/overfit results. We cannot outperform the closed form solution with GD. We can only approach closed forms level of optimality/overfitness. We leave the reasoning behind this as an exercise to the reader.

In [22]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
#This cell may take more than 1 minute
learning_rates = [1e-8, 1e-6, 1e-4]
weights = np.zeros((3, POLY_DEGREE ** 2 + 2))

for ii in range(len(learning_rates)):
    weights[ii,:] = reg.linear_fit_GD(x_all_feat[train_indices],
                                      y_all[train_indices],
                                      epochs=50000,
                                      learning_rate=learning_rates[ii])[0].ravel()
    y_test_pred = reg.predict(x_all_feat[test_indices],
                              weights[ii, :].reshape((POLY_DEGREE ** 2 + 2, 1)))
    test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
    print('Linear (GD) RMSE: %.4f (learning_rate=%s)' % (test_rmse, learning_rates[ii]))

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

colors = ['g', 'orange', 'r']
for ii in range(len(learning_rates)):
    y_pred = reg.predict(x_all_feat, weights[ii])
    y_pred = np.reshape(y_pred, (y_pred.size,))
    ax.plot(x_all[:,0], x_all[:,1], y_pred,
            label='Trendline LR=' + str(learning_rates[ii]),
            color=colors[ii], lw=2, zorder=5)

ax.scatter(xtrain[:,0], xtrain[:,1], ytrain, label='Training', c='y',s=4)
ax.scatter(xtest[:,0], xtest[:,1], ytest, label='Testing', c='black',s=4)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.text2D(0.05, 0.95, "Tunning Linear (GD)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (GD) RMSE: 4.3573 (learning_rate=1e-08)
Linear (GD) RMSE: 2.9430 (learning_rate=1e-06)
Linear (GD) RMSE: 1.1425 (learning_rate=0.0001)
```



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

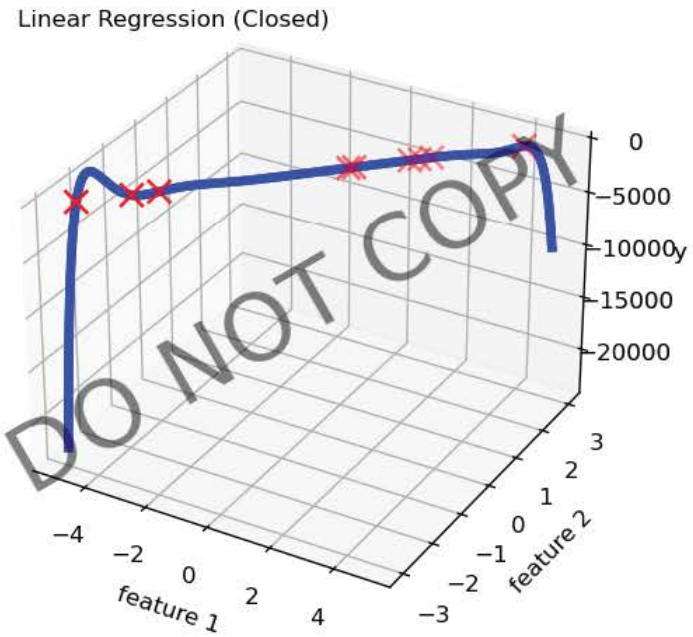
In [23]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
rng = np.random.RandomState(seed=5)
y_all_noisy = np.dot(x_cart_flat, np.zeros((POLY_DEGREE ** 2 + 2, 1))) + rng.randn(x_all_feat.shape[0], 1)
sub_train = train_indices[10:20]
```

In [24]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

weight = reg.linear_fit_closed(x_all_feat[sub_train], y_all_noisy[sub_train])
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print('Linear (closed) 10 Samples RMSE: %.4f' % test_rmse)

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,1]
y_pred = np.reshape(y_pred, (N_SAMPLES,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all_noisy[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.set_zlim([None, 8])
ax.text2D(0.05, 0.95, "Linear Regression (Closed)", transform=ax.transAxes)
```

```
Linear (closed) 10 Samples RMSE: 2176.2678
```

Out[24]: Text(0.05, 0.95, 'Linear Regression (Closed)')



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

3.4 Testing: Testing ridge regression [5 pts] [W]

Now let's try ridge regression. Like before, undergraduate students need to implement the closed form, and graduate students need to implement all the three methods. We will call the prediction function from linear regression part. As long as your test RMSE score is close to the TA's answer (TA's answer ± 0.5), you can get full points.

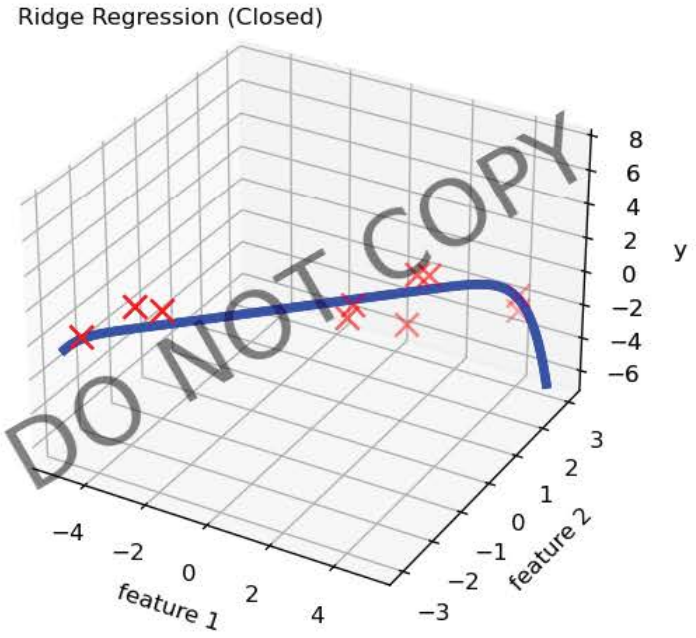
Again, let's see what we have learned. You only need to run the cell corresponding to your specific implementation.

In [25]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
weight = reg.ridge_fit_closed(x_all_feat[sub_train],
                              y_all_noisy[sub_train],
                              c_lambda=10)
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print('Ridge Regression (closed) RMSE: %.4f' % test_rmse)

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,1]
y_pred = np.reshape(y_pred, (N_SAMPLES,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all_noisy[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.set_zlim([None, 8])
ax.text2D(0.05, 0.95, "Ridge Regression (Closed)", transform=ax.transAxes)
```

```
Ridge Regression (closed) RMSE: 1.3642
```

Out[25]: Text(0.05, 0.95, 'Ridge Regression (Closed)')

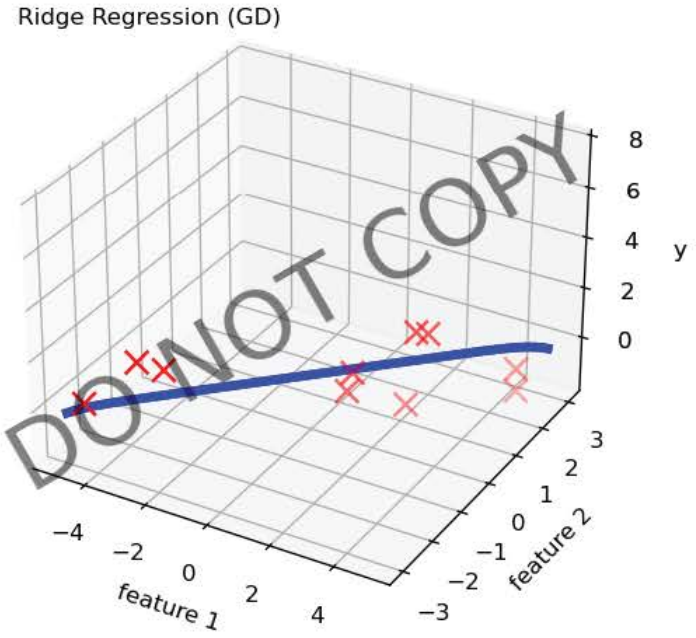


In [26]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
weight, _ = reg.ridge_fit_GD(x_all_feat[sub_train],
                             y_all_noisy[sub_train],
                             c_lambda=10, learning_rate=1e-5)
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print('Ridge Regression (GD) RMSE: %.4f' % test_rmse)

# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,1]
y_pred = np.reshape(y_pred, (N_SAMPLES,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all_noisy[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.set_zlim([None, 8])
ax.text2D(0.05, 0.95, "Ridge Regression (GD)", transform=ax.transAxes)
```

```
Ridge Regression (GD) RMSE: 0.9847
```

Out[26]: Text(0.05, 0.95, 'Ridge Regression (GD)')

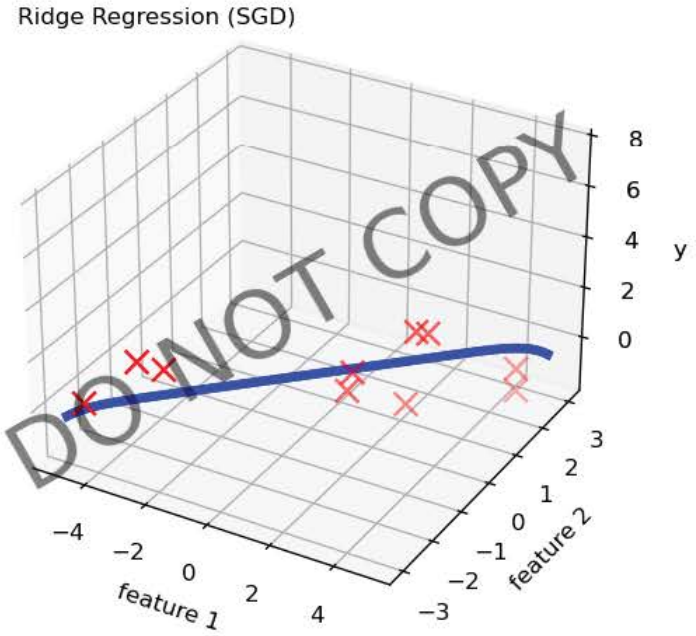


In [27]:

```
################################
### DO NOT CHANGE THIS CELL ###
################################
weight, _ = reg.ridge_fit_SGD(x_all_feat[sub_train],
                              y_all_noisy[sub_train],
                              c_lambda=10,
                              learning_rate=1e-5)
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print('Ridge Regression (SGD) RMSE: %.4f' % test_rmse)


# -- Plotting Code --
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

x1 = x_all[:,0]
x2 = x_all[:,1]
y_pred = np.reshape(y_pred, (N_SAMPLES,))
ax.plot(x1, x2, y_pred, color='b', lw=4)

x3 = x_all[sub_train,0]
x4 = x_all[sub_train,1]
ax.scatter(x3, x4, y_all_noisy[sub_train], s=100, c='r', marker='x')

y_test_pred = reg.predict(x_all_feat[test_indices], weight)
ax.set_xlabel("feature 1")
ax.set_ylabel("feature 2")
ax.set_zlabel("y")
ax.set_zlim([None, 8])
ax.text2D(0.05, 0.95, "Ridge Regression (SGD)", transform=ax.transAxes)
```

```
Ridge Regression (SGD) RMSE: 0.9867
```

Out[27]: Text(0.05, 0.95, 'Ridge Regression (SGD)')



3.5 Cross validation [7 pts] [W]

Let's use Cross Validation to find the best value for `c_lambda` in ridge regression.

In [28]:

```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
# 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
# For the training data, split them in 10 folds which means that use 10 percent of training data for test and
# 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.
best_lambda = None
best_error = None
kfold = 10
lambda_list = [0.0001, 0.001, 0.1, 1, 5, 10, 50, 100, 1000, 10000]

for lm in lambda_list:
    err = reg.ridge_cross_validation(x_all_feat[train_indices], y_all[train_indices], kfold, lm)
    print('Lambda: %.4f' % lm, 'RMSE: %.6f'% err)
    if best_error is None or err < best_error:
        best_error = err
        best_lambda = lm

print('Best Lambda: %.4f' % best_lambda)
weight = reg.ridge_fit_closed(x_all_feat[train_indices], y_all_noisy[train_indices], c_lambda=10)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print('Best Test RMSE: %.4f' % test_rmse)
```

```
Lambda: 0.0001 RMSE: 0.979903
Lambda: 0.0010 RMSE: 0.978994
Lambda: 0.1000 RMSE: 0.978234
Lambda: 1.0000 RMSE: 0.977545
Lambda: 5.0000 RMSE: 0.977622
Lambda: 10.0000 RMSE: 0.978004
Lambda: 50.0000 RMSE: 0.979567
Lambda: 100.0000 RMSE: 0.982586
Lambda: 1000.0000 RMSE: 1.225502
Lambda: 10000.0000 RMSE: 2.812906
Best Lambda: 1.0000
Best Test RMSE: 0.9964
```

3.6 Noisy Input Samples in Linear Regression [10 pts] [W]

Consider a linear model of the form:

$$y(x_n, \theta) = \theta_0 + \sum_{d=1}^{D} \theta_d x_{nd}$$

where $x_n = (x_{n1}, \ldots, x_{nD}) \in \mathbb{R}^D$ and weights $\theta = (\theta_0, \ldots, \theta_D) \in \mathbb{R}^D$. Given the the D-dimension input sample set $x = \{x_1, \ldots, x_n\}$ with corresponding target value $y = \{y_1, \ldots, y_n\}$, the sum-of-squares error function is:

$$E_D(\theta) = \frac{1}{2} \sum_{n=1}^{N} [y(x_n, \theta) - y_n]^2$$

Now, suppose that Gaussian noise $\epsilon_n \in \mathbb{R}^D$ is added independently to each of the input sample x_n to generate a new sample set $x' = \{x_1 + \epsilon_1, \ldots, x_n + \epsilon_n\}$. Here, ϵ_{ni} (an entry of ϵ_n) has zero mean and variance σ^2. For each sample x_n, let $x'_n = (x_{n1} + \epsilon_{n1}, \ldots, x_{nD} + \epsilon_{nd})$, where n and d is independent across both n and d indices.

1. (3pts) Show that $y(x'_n, \theta) = y(x_n, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}$
2. (7pts) Assume the sum-of-squares error function of the noise sample set $x' = \{x_1 + \epsilon_1, \ldots, x_n + \epsilon_n\}$ is $E_D(\theta)'$. Prove the expectation of $E_D(\theta)'$ is equivalent to the sum-of-squares error $E_D(\theta)$ for noise-free input samples with the addition of a weight-decay regularization term (e.g. L_2 norm) , in which the bias parameter θ_0 is omitted from the regularizer. In other words, show that

$$E[E_D(\theta)'] = E_D(\theta) + \textbf{Regularizer.}$$

N.B. You should be incorporating your solution from the first part of this problem into the given sum of squares equation for the second part.

Hint:

- During the class, we have discussed how to solve for the weight θ for ridge regression, the function looks like this:

$$E(\theta) = \frac{1}{N} \sum_{i=1}^{N} [y(x_i, \theta) - y_i]^2 + \frac{\lambda}{N} \sum_{i=1}^{d} \|\theta_i\|^2$$

  where the first term is the sum-of-squares error and the second term is the regularization term. N is the number of samples. In this question, we use another form of the ridge regression, which is:

$$E(\theta) = \frac{1}{2} \sum_{i=1}^{N} [y(x_i, \theta) - y_i]^2 + \frac{\lambda}{2} \sum_{i=1}^{d} \|\theta_i\|^2$$

- For the Gaussian noise ϵ_n, we have $E[\epsilon_n] = 0$

- Assume the noise $\epsilon = (\epsilon_1, \ldots, \epsilon_n)$ are **independent** to each other, we have

$$E[\epsilon_n \epsilon_m] = \begin{cases} \sigma^2 & m = n \\ 0 & m \neq n \end{cases}$$

YOUR SOLUTION HERE

Q4: Naive Bayes Classification [25pts] [P] | [W]

4.1 Bayes in Advertisements [5pts] [W]

A doctor wants to evaluate her patients' health conditions and their relations to lifestyle. She sampled 12 patients randomly and conducted a survey to learn about their lifestyles. The table below shows the current health risk a patient is at and his/her lifestyle.

Health Risk	Smoker?	Exercise Frequency (days/wk)	Average Sleeping hours per day
High	Yes	0-3	<6
Medium	Yes	0-3	6-9
Low	Yes	>3	6-9
Medium	No	>3	6-9
Low	No	0-3	6-9
Low	No	>3	<6
High	Yes	>3	<6
Medium	No	>3	<6
High	No	0-3	6-9
Low	No	0-3	6-9
Medium	Yes	0-3	6-9
Low	Yes	>3	6-9

Given that a smoker who exercises >3 days/wk and sleeps 6-9 hours on a daily average, assess the health condition this person is most likely to be in using Naive Bayes.

Note: You can assume that each habit of a person is independent from other habits i.e. A person who exercises regularly does not tell any information about his sleeping pattern or whether he is a smoker.

Note: Please write down the formula that you'll be using for this and mention what each term represents.

Type *Markdown* and LaTeX: α^2

4.2 Determining Amazon Product Ratings from Product Reviews [15pts] [P]

Budd was recently hired by Amazon to analyze product reviews from select Luxury Beauty Products to determine political product rating (ex: 1-star, 2-star, 3-star, etc.). Budd, a skilled CS4641/7641 alumnus himself, decides to use a Naive Bayes approach to classify Amazon product reviews.

The original dataset has 5 classes: 1-star (class label = 1), 2-star (class label = 2), 3-star (class label = 3), 4-star (class label = 4), 5-star (class label = 5). However, Brian is interested to see how Naive Bayes would perform when he groups these original labels together. He decides to make a 2-label, 3-label, 4-label, and 5-label (which is the original dataset) Naive Bayes models. There are over 200,000 product review. However, to save computational time as well as memory resources, the dataset has been reduced to 40,000 unique product reviews. These product reviews have also been cleaned to remove extra spaces, punctuation, emojis, etc. The dataset (which remains the same except for the number of labels) is then split into a training and testing dataset that has a 8:2 ratio.

The code which is provided loads the product reviews and builds a "bag of words" representation (https://en.wikipedia.org/wiki/Bag-of-words_model) of each product review. Your task is to complete the missing portions of the code and to determine what the star-rating was given for that product review.

In the **nb.py** file, complete the following functions (explanations below are assuming the 5-label model):

- **priors_prob**: calculates the ratio of class probabilities of 1-star, 2-star, 3-star, 4-star, and 5-star. We do this based on word counts rather than document counts.
- **likelihood_ratio**: calculates the ratio of word probablities given the label of whether the star rating was 1-star, 2-star, 3-star, 4-star, and 5-star
- **analyze_star_rating**: takes in the likelihood ratio, priors probabilities for each class and a number of test product reviews represented in Bag-of-Words representation, and analyzes the star-rating

For example, if we have a matrix like: (the first column denotes the class label, the entries in the remaining columns denote the number of occurrances for each word). We have two more columns for words. The first word is "machine" and the second word is "learning"

For this example, we will be assuming the 2-label model. Here, rating <= 2 is assigned to label '0' and rating >= 3 is assigned to label '1'

$$\begin{bmatrix} \text{label} & \text{machine} & \text{learning} \\ 0 & 1 & 4 \\ 0 & 0 & 6 \\ 1 & 3 & 2 \\ 0 & 3 & 1 \\ 1 & 4 & 0 \end{bmatrix}$$

Then we have

$$\textit{prior}(\text{rating} \leq 2) = \frac{1+4+0+6+3+1}{1+4+0+6+3+2+3+1+4+0} = \frac{15}{24}$$
$$\textit{prior}(\text{rating} \geq 3) = \frac{3+2+4+0}{1+4+0+6+3+2+3+1+4+0} = \frac{9}{24}$$

Note 1: In likelihood_ratio(), add one to each word count so as to avoid issues with zero word count. This is known as Add-1 smoothing (https://en.wikipedia.org/wiki/Additive_smoothing). It is a type of additive smoothing. For the numerator, we just add 1 at the end. For the denominator, we add 1 for each feature (in this example, for each word).

$$\textit{likelihood}(\text{rating} \leq 2) = \left[\frac{1+0+3+\mathbf{1}}{1+0+3+\mathbf{1}+4+6+1+\mathbf{1}} \quad \frac{4+6+1+\mathbf{1}}{1+0+3+\mathbf{1}+4+6+1+\mathbf{1}} \right] = \left[\frac{5}{17} \quad \frac{12}{17} \right]$$
$$\textit{likelihood}(\text{rating} \geq 3) = \left[\frac{3+4+\mathbf{1}}{3+4+\mathbf{1}+2+0+\mathbf{1}} \quad \frac{2+0+\mathbf{1}}{3+4+\mathbf{1}+2+0+\mathbf{1}} \right] = \left[\frac{8}{11} \quad \frac{3}{11} \right].$$

Note 2: In `analyze_affiliation()`, we can calculate the posterior probability given the count for each word

	Machine	Learning
Count	3	4

$$P(\text{rating} \leq 2) = \left(\frac{5}{17}\right)^3 * \left(\frac{12}{17}\right)^4 * \frac{15}{24}$$
$$P(\text{rating} \geq 3) = \left(\frac{8}{11}\right)^3 * \left(\frac{3}{11}\right)^4 * \frac{9}{24}$$

The prediction will then be the label with the highest probability

In [29]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from nb import NaiveBayes
import preprocessor as p  #if this command errors, please pip install tweet-preprocessor
```

In [30]:

```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

def clean_tweet(tweet):
    tweet = p.clean(tweet)
    tweet = re.sub(r'[^\w\s]', '', tweet)
    return tweet

def assign_labels(train_dataset, class_to_label_mappings, vectorizer):
    new_train = train_dataset.copy()
    new_train["overall"] = new_train["overall"].map(class_to_label_mappings)
    X = new_train['summary'].values
    y = new_train['overall'].values
    BOW = vectorizer.fit_transform(X).toarray()
    X_train, X_test, y_train, y_test = train_test_split(BOW, y, test_size=0.2, random_state=RANDOM_SEED)
    return X_train, y_train, X_test, y_test

def build_and_test_model(X_train, y_train, X_test, y_test):
    list_of_labels = [X_train[y_train == label] for label in np.unique(y_train)]
    likelihood_ratio = NB.likelihood_ratio(list_of_labels)
    priors_prob = NB.priors_prob(list_of_labels)
    resolved = NB.analyze_star_rating(likelihood_ratio, priors_prob, X_test)
    return np.sum(resolved == y_test) / len(resolved) * 1.

RANDOM_SEED = 5

# Source: https://nijianmo.github.io/amazon/index.html
print("Opening Dataset...")
data = []
with gzip.open('./data/Luxury_Beauty_5.json.gz') as f:
    for l in f:
        data.append(json.loads(l.strip()))

print("Preprocessing Dataset...")
train = pd.DataFrame.from_dict(data)
train = train.fillna('')

train = train.drop(columns=[column for column in train.columns if column != 'overall' and column != 'summary'
train['overall'] = train['overall'].astype('int8')

pbar = tqdm(total=train.shape[0])
for _, row in train.iterrows():
    row['summary'] = clean_tweet(row['summary'])
    pbar.update(1)
train.drop_duplicates(inplace=True)

class_to_label_2 = {
    1: 0,
    2: 0,
    3: 1,
    4: 1,
    5: 1
}

class_to_label_3 = {
    1: 0,
    2: 0,
    3: 1,
    4: 2,
    5: 2
}

class_to_label_4 = {
    1: 0,
    2: 0,
    3: 1,
    4: 2,
    5: 3
}

class_to_label_5 = {
    1: 1,
    2: 2,
    3: 3,
    4: 4,
    5: 5
}

stop_words = text.ENGLISH_STOP_WORDS
vectorizer = text.CountVectorizer(stop_words=stop_words)
```

```
82
83 print("Assigning labels...")
84 X_train_2, y_train_2, X_test_2, y_test_2 = assign_labels(train, class_to_label_2, vectorizer)
85 X_train_3, y_train_3, X_test_3, y_test_3 = assign_labels(train, class_to_label_3, vectorizer)
86 X_train_4, y_train_4, X_test_4, y_test_4 = assign_labels(train, class_to_label_4, vectorizer)
87 X_train_5, y_train_5, X_test_5, y_test_5 = assign_labels(train, class_to_label_5, vectorizer)
88
89 print("Building and testing models...")
90 NB = NaiveBayes()
91 accuracy_2 = build_and_test_model(X_train_2, y_train_2, X_test_2, y_test_2)
92 accuracy_3 = build_and_test_model(X_train_3, y_train_3, X_test_3, y_test_3)
93 accuracy_4 = build_and_test_model(X_train_4, y_train_4, X_test_4, y_test_4)
94 accuracy_5 = build_and_test_model(X_train_5, y_train_5, X_test_5, y_test_5)
```

```
Opening Dataset...
Preprocessing Dataset...

  0%|          | 0/34278 [00:00<?, ?it/s]

Assigning labels...
Building and testing models...
```

In [31]:

```
1  ##############################
2  ### DO NOT CHANGE THIS CELL ###
3  ##############################
4
5  # Should be 91%
6  print(round(accuracy_2 * 100, 3))
7
8  # Should be 78%
9  print(round(accuracy_3 * 100, 3))
10
11 # Should be 54%
12 print(round(accuracy_4 * 100, 3))
13
14 # Should be 22%
15 print(round(accuracy_5 * 100, 3))
16
```

```
91.266
78.409
54.013
22.688
```

4.3 Accuracy result analysis [5pts] [W]

What is the trend between accuracy and number of labels? Why do you think this is the case? What assumptions did your model make that limits the accuracy? (This is an open question, any reasonable assumptions will be acceptable).

Q5: Noise in PCA and Linear Regression [15pts] [W]

Both PCA and least squares regression can be viewed as algorithms for inferring (linear) relationships among data variables. In this part of the assignment, you will develop some intuition for the differences between these two approaches and develop an understanding of the settings that are better suited to using PCA or better suited to using the least squares fit.

The high level bit is that PCA is useful when there is a set of latent (hidden/underlying) variables, and all the coordinates of your data are linear combinations (plus noise) of those variables. The least squares fit is useful when you have direct access to the independent variables, so any noisy coordinates are linear combinations (plus noise) of known variables.

5.1 Slope Functions (5 Pts) [W]

In the **following cell**, complete the following:

1. **pca_slope**: For this function, assume that X is the first feature and Y is the second feature for the data. Write a function, that takes in the first feature vector X and the second feature vector Y. Stack these two feature vectors into a single Nx2 matrix and use this to determine the first principal component vector of this dataset. Finally, return the slope of this first component. You should use the PCA implementation from Q2.

2. **lr_slope**: Write a function that takes X and y and returns the slope of the least squares fit. You should use the Linear Regression implementation from Q3 but do not use any kind of regularization. Think about how weight could relate to slope.

In later subparts, we consider the case where our data consists of noisy measurements of x and y. For each part, we will evaluate the quality of the relationship recovered by PCA, and that recovered by standard least squares regression.

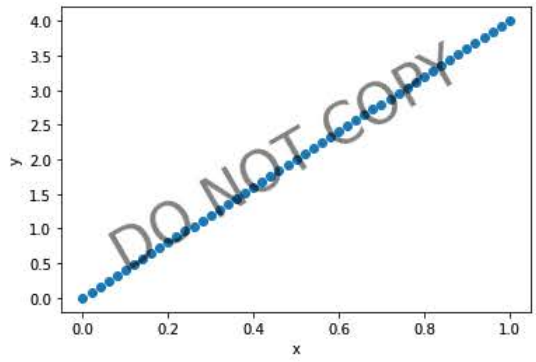
As a reminder, least squares regression minimizes the squared error of the dependent variable from its prediction. Namely, given (x_i, y_i) pairs, least squares returns the line $l(x)$ that minimizes $\sum_i (y_i - l(x_i))^2$.

In [32]:
```
import numpy as np
from pca import PCA
from regression import Regression

def pca_slope(X, y):
    """
    Calculates the slope of the first principal component given by PCA

    Args:
        x: N x 1 array of feature x
        y: N x 1 array of feature y
    Return:
        slope: (float) scalar slope of the first principal component
    """

    raise NotImplementedError


def lr_slope(X, y):
    """
    Calculates the slope of the best fit returned by linear_fit_closed()

    For this function don't use any regularization

    Args:
        X: N x 1 array corresponding to a dataset
        y: N x 1 array of labels y
    Return:
        slope: (float) slope of the best fit
    """

    raise NotImplementedError
```

We will consider a simple example with two variables, x and y, where the true relationship between the variables is $y = 4x$. Our goal is to recover this relationship—namely, recover the coefficient "4". We set $X = [0, .02, .04, .06, \ldots, 1]$ and $y = 4x$. Make sure both functions return 4.

In [33]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
x = np.arange(0, 1.02, 0.02).reshape(-1, 1)

y = 4 * np.arange(0, 1.02, 0.02).reshape(-1, 1)

print("Slope of first principal component", pca_slope(x, y))

print("Slope of best linear fit", lr_slope(x, y))

fig = plt.figure()
plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

```
Slope of first principal component 4.0
Slope of best linear fit 4.0
```

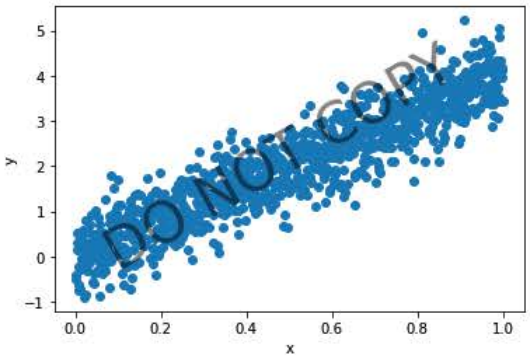


5.2 Analysis Setup (5 Pts) [W]

Error in y

In this subpart, we consider the setting where our data consists of the actual values of x, and noisy estimates of y. Run the following cell to see how the data looks when there is error in y.

In [34]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
base = np.arange(0.001, 1.001, 0.001).reshape(-1, 1)
c = 0.5
X = base
y = 4 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

fig = plt.figure()
plt.scatter(X, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



In **following cell**, you will implement the **addNoise** function:

1. Create a vector $\boldsymbol{X}$ where $\boldsymbol{X} = [x_1, x_2, \ldots, x_{1000}] = [.001, .002, .003, \ldots, 1]$.
2. For a given noise level $\boldsymbol{c}$, set $\hat{y}_i \sim 4x_i + \mathcal{N}(0, c) = 4i/1000 + \mathcal{N}(0, c)$, and $\hat{\boldsymbol{Y}} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{1000}]$. You can use the np.random.normal function, where scale is equal to noise level, to add noise to your points.
3. Notice the parameter **x_noise** in the **addNoise** function. When this parameter is set to ***True***, you will have to add noise to $\boldsymbol{X}$. For a given noise level c, let $\hat{x}_i \sim x_i + \mathcal{N}(0, c) = i/1000 + \mathcal{N}(0, c)$, and $\hat{\boldsymbol{X}} = [\hat{x}_1, \hat{x}_2, \ldots . \hat{x}_{1000}]$
4. Return the **pca_slope** and **lr_slope** values of this $\boldsymbol{X}$ and $\hat{\boldsymbol{Y}}$ dataset you have created where $\hat{\boldsymbol{Y}}$ has noise ($\boldsymbol{X} = \boldsymbol{X}$ or $\hat{\boldsymbol{X}}$ depending on the problem).

In [35]:
```
def addNoise(c, x_noise = False, seed = 1):
    """
    Creates a dataset with noise and calculates the slope of the dataset
    using the pca_slope and lr_slope functions implemented in this class.

    Args:
        c: (float) scalar, a given noise level to be used on Y and/or X
        x_noise: (Boolean) When set to False, X should not have noise added
                 When set to True, X should have noise.
                 Note that the noise added to X should be different from the
                 noise added to Y. You should NOT use the same noise you add
                 to Y here.
        seed: (int) Random seed
    Return:
        pca_slope_value: (float) slope value of dataset created using pca_slope
        lr_slope_value: (float) slope value of dataset created using lr_slope

    """
    np.random.seed(seed) #### DO NOT CHANGE THIS ####
    ############## START YOUR CODE BELOW ##############

    raise NotImplementedError
```

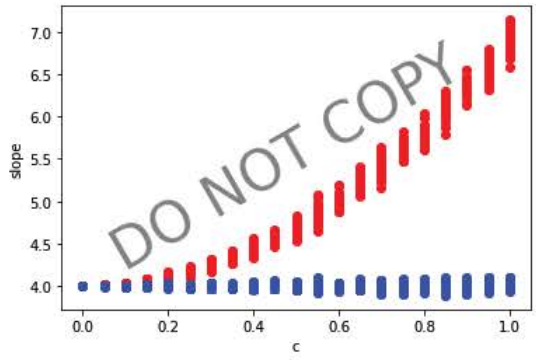
A scatter plot with $\boldsymbol{c}$ on the horizontal axis and the output of **pca_slope** and **lr_slope** on the vertical axis has already been implemented for you.

A sample $\hat{\boldsymbol{Y}}$ has been taken for each $\boldsymbol{c}$ in $[0, 0.05, 0.1, \ldots, 0.95, 1.0]$. The output of **pca_slope** is plotted as a red dot, and the output of **lr_slope** as a blue dot. This has been repeated 30 times, you can see that we end up with a plot of 1260 dots, in 21 columns of 60, half red and half blue.

Note: Here, `x_noise = False` since we only want Y to have noise.

In [36]:

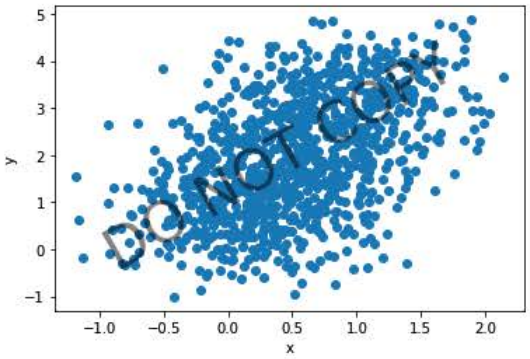
```
################################
### DO NOT CHANGE THIS CELL ###
################################
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

for i in range(30):
    for c in np.arange(0, 1.05, 0.05):

        # Calculate pca_slope_value (psv) and lr_slope_value (lsv)
        psv, lsv = addNoise(c, seed = s_idx)

        # Append pca and lr slope values to list for plot function
        pca_slope_values.append(psv)
        linreg_slope_values.append(lsv)

        # Append c value to list for plot function
        c_values.append(c)

        # Increment random seed index
        s_idx += 1

fig = plt.figure()
plt.scatter(c_values, pca_slope_values, c='r')
plt.scatter(c_values, linreg_slope_values, c='b')
plt.xlabel("c")
plt.ylabel("slope")
plt.show()
```



Error in x and y[W]

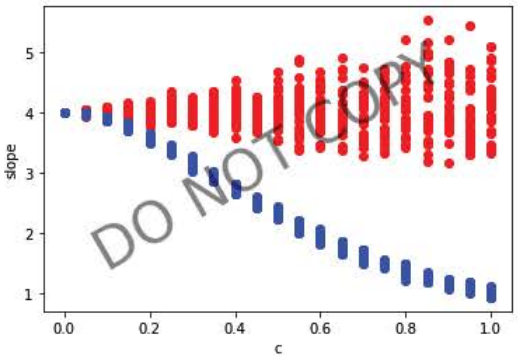
We will now examine the case where our data consists of noisy estimates of **both x** and **y**. Run the following cell to see how the data looks when there is error in both.

In [37]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
base = np.arange(0.001, 1, 0.001).reshape(-1, 1)
c = 0.5
X = base + np.random.normal(loc=[0], scale=c, size=base.shape)
y = 4 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

fig = plt.figure()
plt.scatter(X, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



In the below cell, we graph the predicted PCA and LR slopes on the vertical axis against the value of c on the horizontal axis.

In [38]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

for i in range(30):
    for c in np.arange(0, 1.05, 0.05):

        # Calculate pca_slope_value (psv) and lr_slope_value (lsv), notice x_noise = True
        psv, lsv = addNoise(c, x_noise = True, seed = s_idx)

        # Append pca and lr slope values to list for plot function
        pca_slope_values.append(psv)
        linreg_slope_values.append(lsv)

        # Append c value to list for plot function
        c_values.append(c)

        # Increment random seed index
        s_idx += 1

fig = plt.figure()
plt.scatter(c_values, pca_slope_values, c='r')
plt.scatter(c_values, linreg_slope_values, c='b')
plt.xlabel("c")
plt.ylabel("slope")
plt.show()
```



5.3. Analysis (5 Pts) [W]

Based on your observations from previous subsections answer the following questions about the two cases (error in Y and error in both X and Y) in 2-3 lines.

Note:

1. The closer the value of slope to actual slope ("4" here) the better the algorithm is performing.
2. You don't need to provide a mathematical proof for this question.

Questions:

1. Which case does PCA perform worse in? Why does PCA perform worse in this case? (2 Pts)
2. Why does PCA perform better in the other case? (1 Pt)
3. Which case does Linear Regression perform well? Why does Linear Regression perform well in this case? (2 Pts)

6 Feature Reduction Implementation [25 Points Bonus for All] [P + W]

6.1 Implementation [18 Points] [P]

Feature selection is an integral aspect of machine learning. It is the process of selecting a subset of relevant features that are to be used as the input for the machine learning task. Feature selection may lead to simpler models for easier interpretation, shorter training times, avoidance of the curse of dimensionality, and better generalization by reducing overfitting.

In the **feature_reduction.py** file, complete the following functions:

- **forward_selection**
- **backward_elimination**

These functions should each output a list of features.

Forward Selection:

In forward selection, we start with a null model, start fitting the model with one individual feature at a time, and select the feature with the minimum p-value. We continue to do this until we have a set of features where one feature's p-value is less than the confidence level.

Steps to implement it:

1. Choose a significance level (given to you).
2. Fit all possible simple regression models by considering one feature at a time.
3. Select the feature with the lowest p-value.
4. Fit all possible models with one extra feature added to the previously selected feature(s).
5. Select the feature with the minimum p-value again. if p_value < significance, go to Step 4. Otherwise, terminate.

Backward Elimination:

In backward elimination, we start with a full model, and then remove the insignificant feature with the highest p-value (that is greater than the significance level). We continue to do this until we have a final set of significant features.

Steps to implement it:

1. Choose a significance level (given to you).
2. Fit a full model including all the features.
3. Select the feature with the highest p-value. If (p-value > significance level), go to Step 4, otherwise terminate.
4. Remove the feature under consideration.
5. Fit a model without this feature. Repeat entire process from Step 3 onwards.

TIP 1: The p-value is known as the observed significance value for a null hypothesis. In our case, the p-value of a feature is associated with the hypothesis $H_0: \beta_j = 0$. If $\beta_j = 0$, then this feature contributes no predictive power to our model and should be dropped. We reject the null hypothesis if the p-value is smaller than our significance level. Some more information about p-values can be found here: https://towardsdatascience.com/what-is-a-p-value-b9e6c207247f (https://towardsdatascience.com/what-is-a-p-value-b9e6c207247f)

TIP 2: For this function, you will have to install statsmodels if not installed already. To do this, run `pip install statsmodels` in command line/terminal. In the case that you are using an Anaconda environment, run `conda install -c conda-forge statsmodels` in the command line/terminal. For more information about installation, refer to https://www.statsmodels.org/stable/install.html (https://www.statsmodels.org/stable/install.html). The statsmodels library is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration. You will have to use this library to choose a regression model to fit your data against. Some more information about this module can be found here: https://www.statsmodels.org/stable/index.html (https://www.statsmodels.org/stable/index.html)

TIP 3: For step 2 in each of the forward and backward selection functions, you can use the `sm.OLS` function as your regression model. Also, do not forget to add a bias to your regression model. A function that may help you is the `sm.add_constants` function.

TIP 4: You should be able to implement these function using only the libraries provided in the cell below.

In [39]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

from feature_reduction import FeatureReduction
```

In [40]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
boston = load_boston()
bos = pd.DataFrame(boston.data, columns = boston.feature_names)
bos['Price'] = boston.target
X = bos.drop("Price", 1)       # feature matrix
y = bos['Price']               # target feature
featureselection = FeatureReduction()
#Run the functions to make sure two lists are generated, one for each method
print("Features selected by forward selection:", FeatureReduction.forward_selection(X, y))
print("Features selected by backward elimination:", FeatureReduction.backward_elimination(X, y))
```

```
Features selected by forward selection: ['LSTAT', 'RM', 'PTRATIO', 'DIS', 'NOX', 'CHAS', 'B', 'ZN', 'CRIM', 'RA
D', 'TAX']
Features selected by backward elimination: ['CRIM', 'ZN', 'CHAS', 'NOX', 'RM', 'DIS', 'RAD', 'TAX', 'PTRATIO',
'B', 'LSTAT']
```

6.2 Feature Selection - Discussion [7pts] [W]

Question 6.2.1:

We have seen two regression methods namely Lasso and Ridge regression earlier in this assignment. Another extremely important and common use-case of these methods is to perform feature selection. According to you, which of these two methods are more appropriate for feature selection? Why? **(3 pts)**

Question 6.2.2:

We have seen that we use different subsets of features to get different regression models. These models depend on the relevant features that we have selected. Using forward selection, what fraction of the total possible models can we explore? Assume that the total number of features that we have at our disposal is N. Remember that in stepwise feature selection (like forward selection and backward elimination), we always include an intercept in our model, so you only need to consider the N features. **(4 pts)**