Summer 2021 CS 4641\7641 A: Machine Learning Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: July 14th, Wednesday, 11:59 AOE

- Late submission will receive 0 credit.
- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be completed individually.

Instructions for the assignment

- In this assignment, we have programming and writing questions.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type

  ***** Typing with LaTeX is required for all the written questions, and can be done in markdown cell types. *****
  ***** HANDWRITTEN ANSWERS WILL NOT BE ACCEPTED. *****

- If a question requires a picture, you could use this syntax "$<imgsrc=""style="width:300px;" />$" to include them within your ipython notebook.
- Questions marked with **[P]** are programming only and should be submitted to the autograder. Questions marked with **[W]** may require that you code a small function or generate plots, but should **NOT** be submitted to the autograder. It should be submitted on the written portion of the assignment on gradescope
- The outline of the assignment is as follows:
  - Q1 [30 pts] > Image compression with SVD **[W]** 1.2 and 1.3 | **[P]** items 1.1
  - Q2 [20 pts] > Understanding PCA **[W]** items 2.2, 2.3 | **[P]** 2.1
  - Q3 [60+(20 bonus for undergrads)]> Regression and regularization **[W]** items 3.2, 3.3, 3.4 ,3.5 and 3.6 | **[P]** items 3.1
  - Q4 [25 pts] > Naive Bayes classification. **[W]** items 4.1 | **[P]** items 4.2
  - Q5 [15 pts] > Noise in PCA and Linear Regression. **[W]** All parts
  - Q6 [Bonus for all][25 pts] > Feature Selection. **[W]** items 6.2 | **[P]** items 6.1

Bonus for undergrads in Q3: For undergraduate students, you are required to implement the closed form for linear regression and for ridge regression, the other 4 methods are bonus questions. **For graduate students, you are required to implement all of them.**

Using the autograder

- Undergrad students will find four assignments on Gradescope that correspond to HW3: "HW3 - Programming", "HW3 - Programming (Bonus)", "HW3 - Programming (Bonus for all)", and "HW3 - Non-programming".
- Graduate students will find three assignments on Gradescope that correspond to HW3: "HW3 - Programming", "HW3 - Programming (Bonus for all)", and "HW3 - Non-programming".
- You will submit your code for the autograder on "HW3 - Programming" in the following format:
  - imgcompression.py
  - pca.py
  - regression.py
  - nb.py

    - feature_selection.py
- All you will have to do is implement the classes "ImgCompression", "PCA", "Regression", "NaiveBayes", "FeatureSelection" in the respective files. We have provided you different .py files and 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 tests each function separately, therefore 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 "HW3 - Non-programming" part, you will download your jupyter notebook as HTML, print it as a PDF from your browser and submit it on Gradescope. To download the notebook as html, click on "File" on the top left corner of this page and select "Download as > HTML". The non-programming part corresponds to Q1.2 - 1.3, Q2.2, Q3.2 - 3.6, Q4.1, Q5.3 and Q6.2. For questions that include images include both your response and the generated images in your submission**

In [36]:
```
# HELPER CELL, DO NOT MODIFY
# 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 json
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

warnings.filterwarnings('ignore')

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

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1. Image compression with SVD [30 pts] **[P]** **[W]**

Load images data and plot

In [37]:
```
# HELPER CELL, DO NOT MODIFY
# load Image
image = plt.imread("./data/hw3_image_owl.jpg")/255
#plot image
fig = plt.figure(figsize=(7,7))
plt.xticks([])
plt.yticks([])
plt.imshow(image)
```

Out[37]: <matplotlib.image.AxesImage at 0x7f96ea5d5580>

In [38]:
```
# HELPER CELL, DO NOT MODIFY
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

fig = plt.figure(figsize=(7, 7))
plt.xticks([])
plt.yticks([])
plt.imshow(rgb2gray(image), cmap=plt.cm.bone)
```

Out[38]: <matplotlib.image.AxesImage at 0x7f96ea8c4190>

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 hardly be spotted.

For example, the variance captured by the first component

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

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_matrix` parameter. It may be helpful to use the `full_matrix` parameter if you would like to reuse this function in question 2.
- **rebuild_svd**
- **compression_ratio**
- **recovered_variance_proportion**

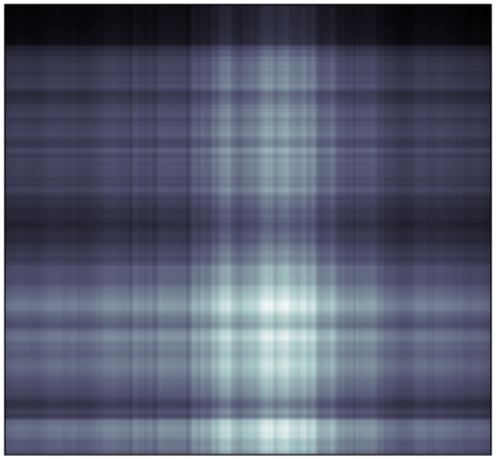
Hint 1: http://timbaumann.info/svd-image-compression-demo/ (http://timbaumann.info/svd-image-compression-demo/) is a useful article on image compression and compression ratio.
Hint 2: If you have never used `np.linalg.svd` it might be helpful to read Numpy's SVD documentation (https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html) and note the particularities of the V matrix and that it is returned already transposed.

1.2 Black and white [5 pts] **[W]**

Use your implementation to generate a set of images compressed to different degrees.
Include the images in your non-programming submission of the assignment.

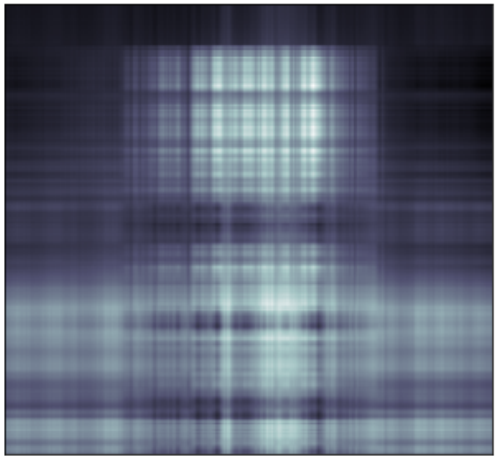
In [39]:
```
# HELPER CELL, DO NOT MODIFY
from imgcompression import ImgCompression

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 Components



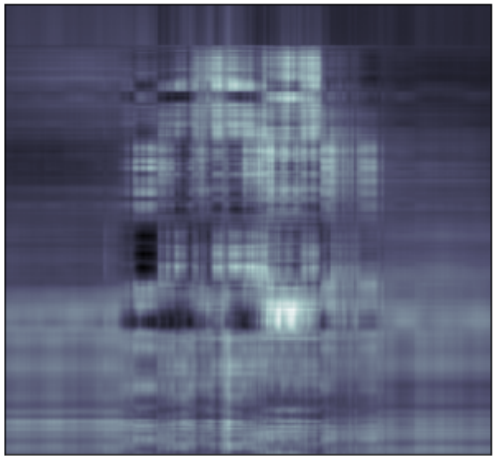
Compression: 0.0033,
Recovered Variance: [0.858]

2 Components



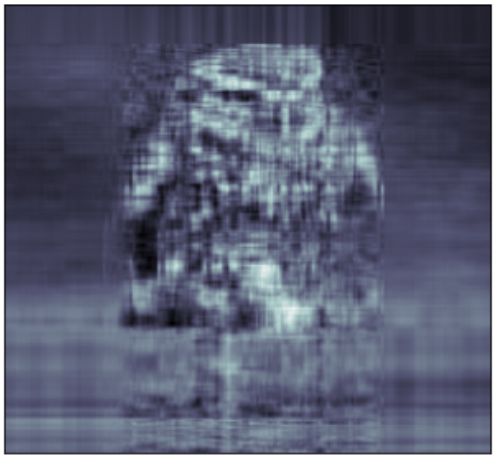
Compression: 0.0065,
Recovered Variance: [0.879]

5 Components



Compression: 0.0163,
Recovered Variance: [0.909]

10 Components



Compression: 0.0326,
Recovered Variance: [0.934]

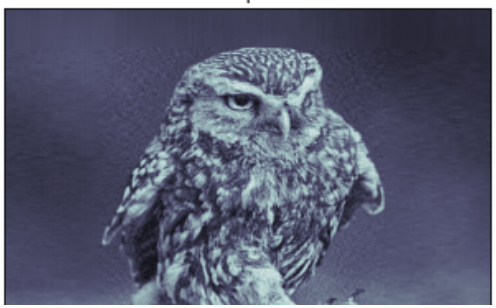
20 Components

Compression: 0.0652,
Recovered Variance: [0.955]

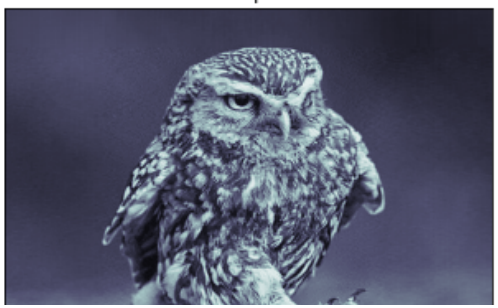
40 Components

Compression: 0.1304,
Recovered Variance: [0.974]

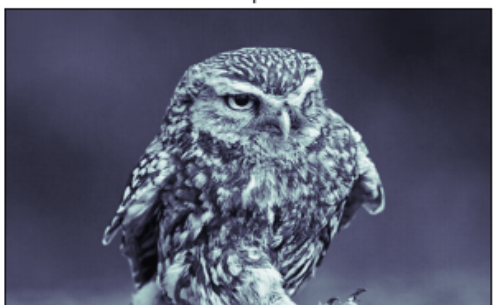
80 Components



160 Components



256 Components



Compression: 0.2608,
Recovered Variance: [0.989]

Compression: 0.5216,
Recovered Variance: [0.998]

Compression: 0.8346,
Recovered Variance: [1.]

1.3 Color image [5 pts] **[W]**

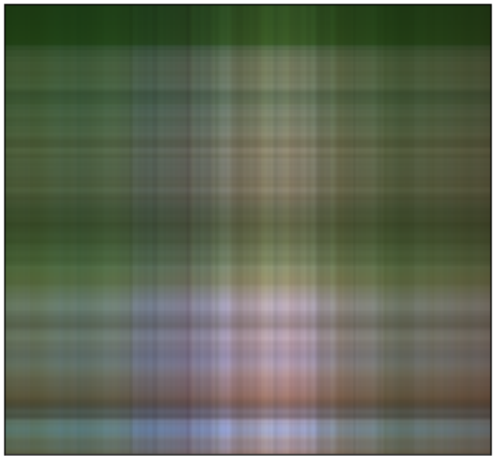
Use your implementation to generate a set of images compressed to different degrees.
Include the images in your non-programming submission of the assignment.

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 while rebuilding some of the pixels may go above 1.0. You should see similar image 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.

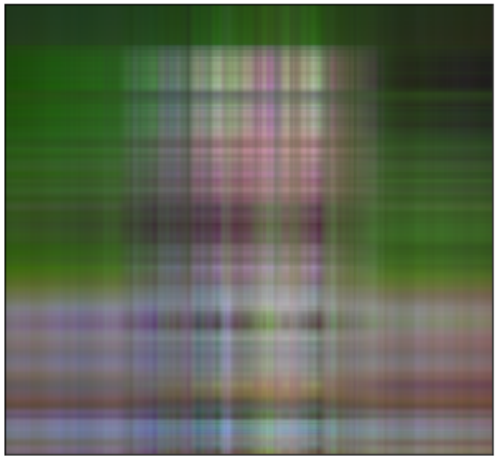
In [40]:
```
# HELPER CELL, DO NOT MODIFY

from imgcompression import ImgCompression

imcompression = ImgCompression()

U, S, V = imcompression.svd(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 = 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
```

1 Components



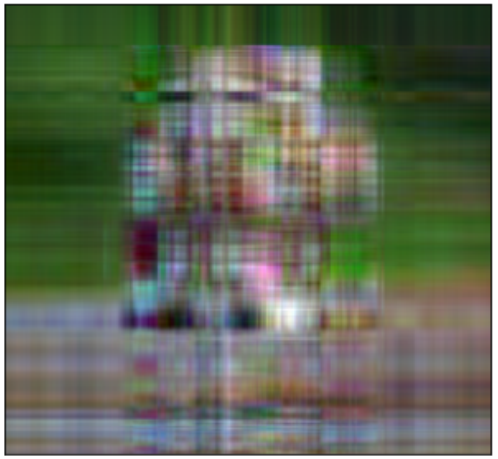
Compression: 0.0033,
Recovered Variance: R: 0.844 G: 0.869 B: 0.784

2 Components



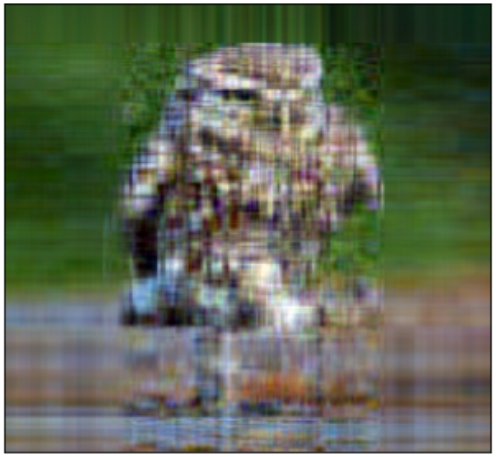
Compression: 0.0065,
Recovered Variance: R: 0.88 G: 0.885 B: 0.829

5 Components



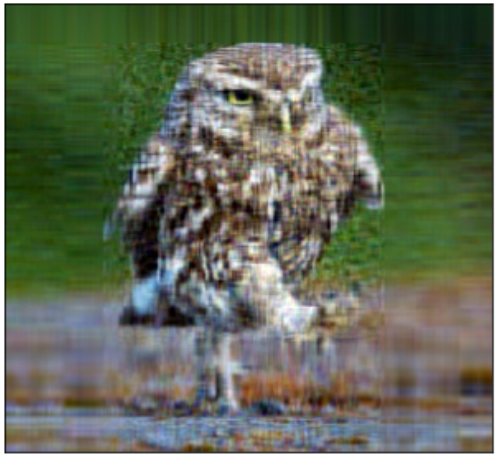
Compression: 0.0163,
Recovered Variance: R: 0.912 G: 0.913 B: 0.877

10 Components



Compression: 0.0326,
Recovered Variance: R: 0.936 G: 0.936 B: 0.91

20 Components



Compression: 0.0652,
Recovered Variance: R: 0.956 G: 0.957 B: 0.941

40 Components

Compression: 0.1304,
Recovered Variance: R: 0.974 G: 0.975 B: 0.968

80 Components

160 Components

256 Components


Compression: 0.2608,
Recovered Variance: R: 0.989 G: 0.989 B: 0.987


Compression: 0.5216,
Recovered Variance: R: 0.998 G: 0.998 B: 0.997


Compression: 0.8346,
Recovered Variance: R: 1.0 G: 1.0 B: 1.0

2 Understanding PCA [20 pts] **[P]** | **[W]**

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` or your implementation of svd from ImgCompression. Set `full_matrices=False`
- **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. It is possible, however, that many of these dimensions contain redundant information. Each feature explains part of the variance in our dataset. Some features may explain more variance than others.

In [41]:
```
from pca import PCA
```

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. It can also be used as a feature extractor for images. Here you will visualize two datasets using PCA, first is the iris dataset and then a dataset of masked and unmasked images.

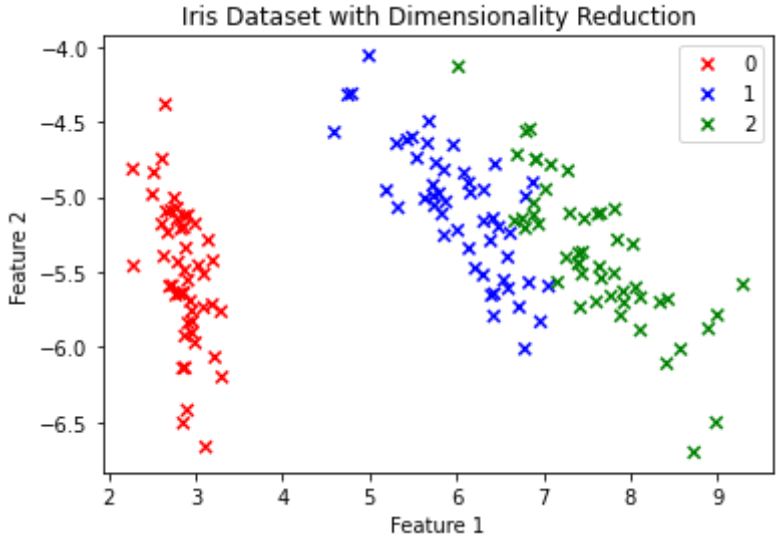
In the **following cell**, complete the following:

1. **visualize**: Use the above implementation of PCA and reduce the datasets such that they contain only two features. Using Matplotlib's Pyplot (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html), replicate the 2-D scatter plots shown below of the data points using these features. Make sure to differentiate the data points according to their true labels using color.

The datasets have already been loaded for you in the subsequent cells.

In [42]:
```
from matplotlib.colors import ListedColormap
def visualize(X,y): # 5 pts
    """
    Reduce the dimensionality of a datset to two features using your implementation of PCA.

    Visualize the dataset in 2-dimensional space using matplotlib's pyplot.

    Include a legend and differentiate the points according to their true labels using
    color. You are free to choose your colors as long as they differ for each label.

    Args:
        xtrain: NxD numpy array, where N is number of instances and D is the dimensionality of each i
nstance
        ytrain: numpy array (N,), the true labels

    Return: None
    """

    # TODO: implement the visualize function



    mypca = PCA()
    mypca.fit(X)
    pcapcaTransformX = mypca.transform(X,2)
    keys = ['0','1','2']
    values = ['0', '0', '1','2','2','2']
    colors = ListedColormap(['r', 'b', 'g'])
    scatter = plt.scatter(pcapcaTransformX[:,0], pcapcaTransformX[:,1],c=y, cmap = colors, marker =
'x')
    plt.legend(handles = scatter.legend_elements()[0], labels = keys)
```

In [43]:
```
# HELPER CELL, DO NOT MODIFY
# Use PCA for visualization of iris dataset
iris_data = load_iris(return_X_y=True)

X = iris_data[0]
y = iris_data[1]

plt.title('Iris Dataset with Dimensionality Reduction')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
visualize(X,y)
```



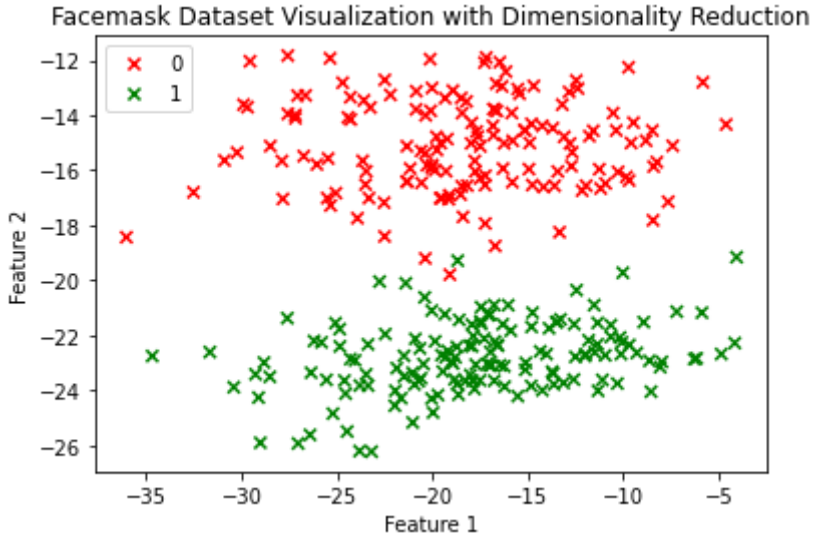
The masked and unmasked dataset is made up of grayscale images of human faces facing forward. Half of these images are faces that are completely unmasked, and the remaining images show half of the face covered with an artificially generated face mask. The images have already been preprocessed, they are also reduced to a small size of 64x64 pixels and then reshaped into a feature vector of 4096 pixels. Below is a sample of some of the images in the dataset.

In [44]:
```
X = np.load('./data/smallflat_64.npy')
y = np.load('./data/masked_labels.npy').astype('int')
i = 0
fig = plt.figure(figsize=(18, 18))
for idx in [0,1,2,150,151,152]:
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(X[idx].reshape(64, 64), cmap = 'gray')
    m_status = 'Unmasked' if idx < 150 else 'Masked'
    ax.set_title(f"{m_status} Image at index {idx}")
    i += 1
```


Unmasked Image at index 0
Unmasked Image at index 1
Unmasked Image at index 2
Masked Image at index 150
Masked Image at index 151
Masked Image at index 152

In [45]:
```
# HELPER CELL, DO NOT MODIFY
# Use PCA for visualization of masked and unmasked images

X = np.load('./data/smallflat_64.npy')
y = np.load('./data/masked_labels.npy').astype('int')


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

```
data shape before PCA (300, 4096)
*In this plot, the 0 points are unmasked images and the 1 points are masked images.
```



What do you think of this 2 dimensional plot, knowing that the original dataset was originally a set of flattened image vectors that had 4096 pixels/features? No written answer necessary.

2.3 Reduced Masked Dataset Analysis [5 pts] **[W]**

1. If the face mask dataset that has been reduced to 2 features was fed into a classifier, do you think the classifier would produce high accuracy or low accuracy? Why? (One or two sentences will suffice for this question.) **(3 pts)**

   Answer ...
   It seems that the data are lineaerly disjoint. So, a classifier should be able to classify them with high accuracy!
2. Assuming an equal rate of accuracy, what do you think is the main advantage in feeding a classifier a dataset with 2 features vs a dataset with 4096 features? (One sentence will suffice for this question.) **(2 pts)**

   Answer ... The main advantage is that computation with less features would be easier and faster.

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) close form; 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, 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 completing functions rmse, construct_polynomial_feates, predict first. Then, construct linear_fit_closed, linear_fit_GD, linear_fit_SGD for linear regression and ridge_fit_closed, ridge_fit_GD, and ridge_fit_SGD for ridge regression. For undergraduate students, you are required to implement the closed form for linear regression and for ridge regression, the other 4 methods are bonus questions. **For graduate students, you are required to implement all of them.** The points for each function is in regression.py

In [46]:
```
from regression import Regression
```

3.2 About RMSE [3 pts] **[W]**

What is a good RMSE value? If we normalize our labels 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.

Answer:

Given the fact that our labels vary in the range of [-4,4] and our RMSE is around 1, this means that our model's accuracy is not good, since this indicates that on average each predicted vs. ground truth has difference=1 which is around 12\% of [-4,4] range. Note that if we normalize lables to be within [0,1] and get RMSE = 1, this indicates that our model accuracy is even worse since in this case, RMSE=1 is 100\% of label range.

By definition, RMSE is

$$RMSE = \sqrt{(\frac{1}{N}\sum_{n=1}^{\infty}(ypredicted_n - ytrue_n)^2)}$$

The labels can be normalized between 0 and 1 by dividing all the labels by the largest label. The predicted values are normalized in the same manner. When normalized between 0 and 1, the largest value the RMSE can take is 1 Hence, RMSE of 1 after normalization corresponds to the highest mismatch between the predicted and actual values.

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 $[1, a, a^2, a^3, \ldots a^{degree}]$ and $[1, b, b^2, b^3, \ldots, b^{degree}]$. 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, for 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).

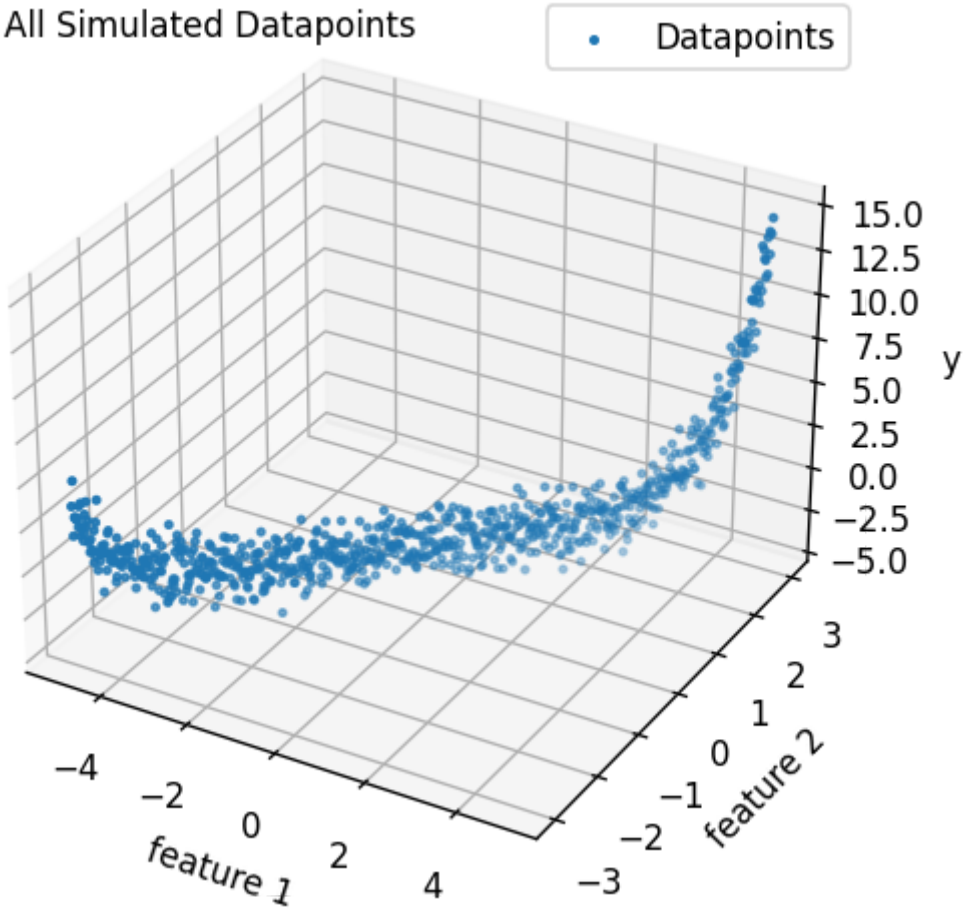
In [47]:
```
# HELPER CELL, DO NOT MODIFY
POLY_DEGREE = 5
N_SAMPLES = 800

rng = np.random.RandomState(seed=5)

# 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: 800 (rows/samples) 2 (columns/features)
y_all: 800 (rows/samples) 1 (columns/features)
```

In [48]:
```
x_all.shape
```

Out[48]:
```
(800, 2)
```

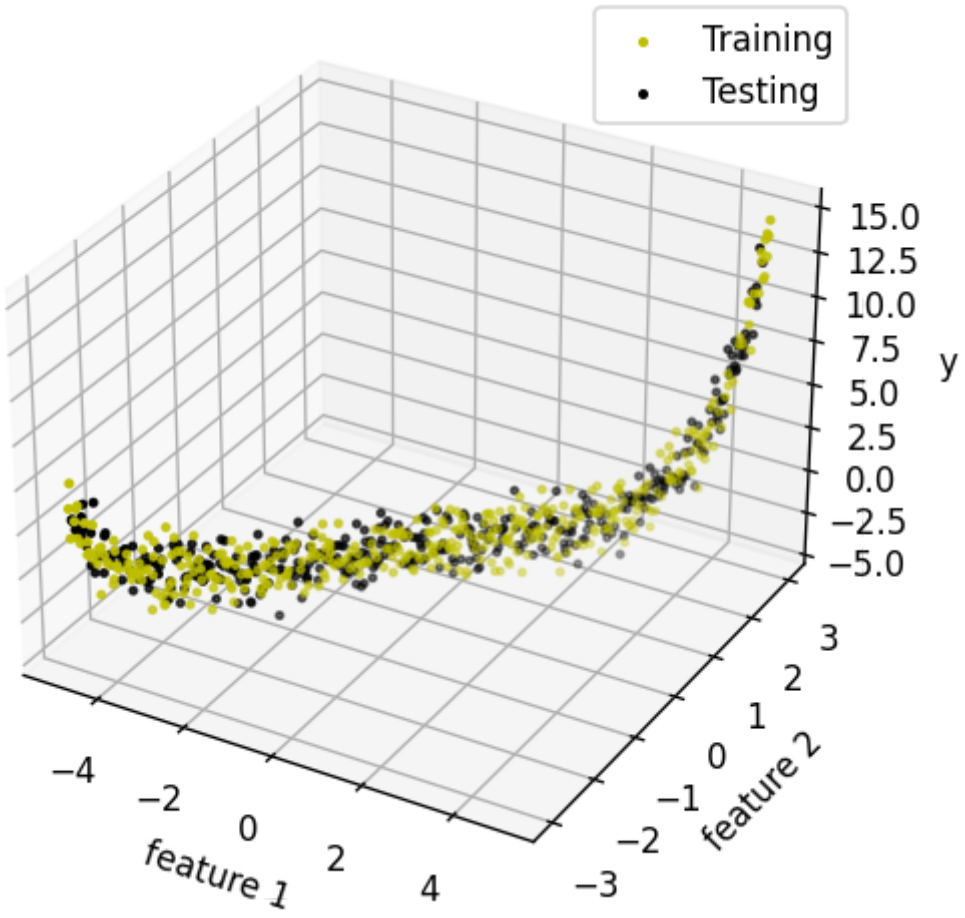
In [49]:
```
# HELPER CELL, DO NOT MODIFY
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

p = np.reshape(np.dot(x_cart_flat, true_weight), (N_SAMPLES,))
#ax.plot(x_all[:,0], x_all[:,1], p, label='Line of Best Fit', c="red", linewidth=2)
ax.scatter(x_all[:,0], x_all[:,1], y_all, label='Datapoints', s=4)
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 $Y = X\theta + \sigma$, where σ~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.

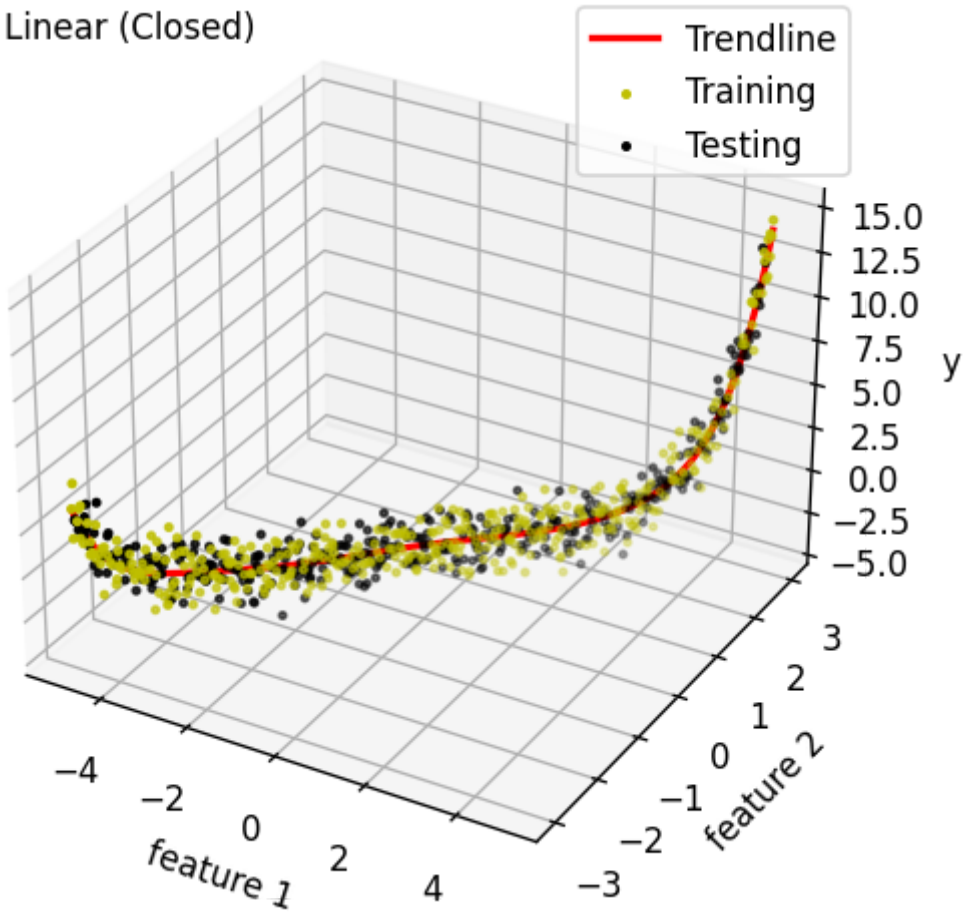
In [50]:
```
# HELPER CELL, DO NOT MODIFY
PERCENT_TRAIN = 0.5

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 models learn function.

In [51]:
```
# HELPER CELL, DO NOT MODIFY
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)

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()
```

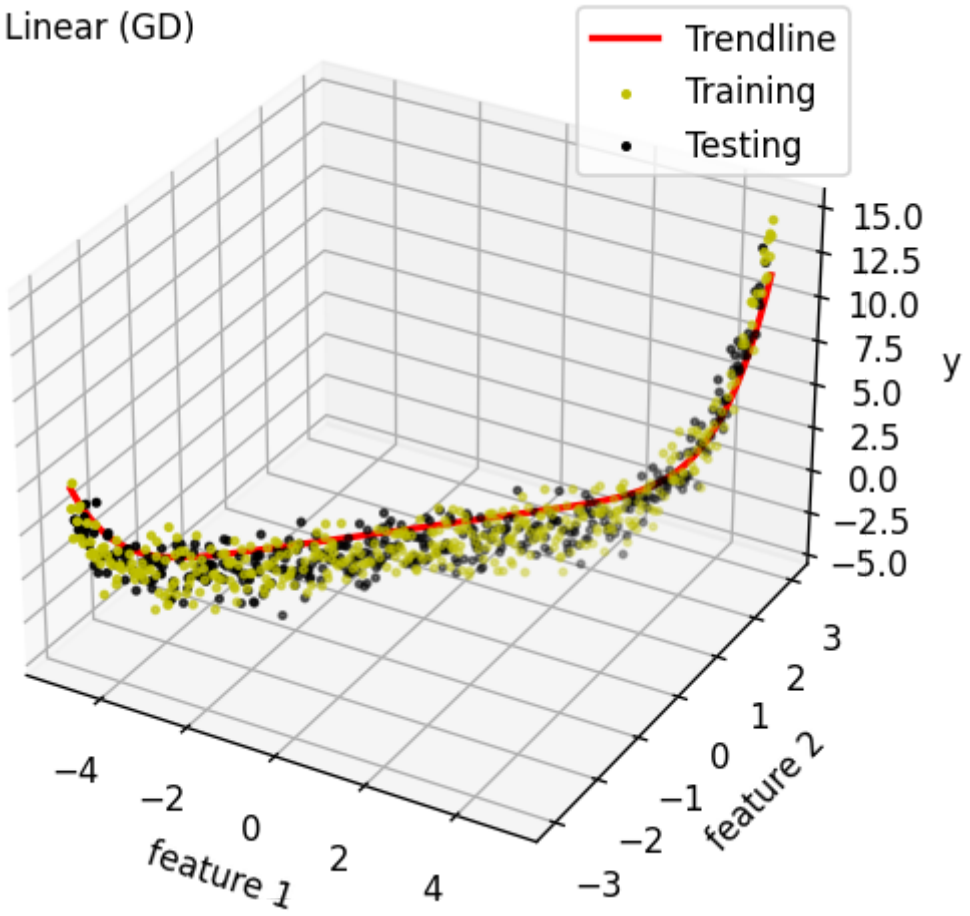
```
Linear (closed) RMSE: 0.9418
```



Now let us use our linear gradient descent function with the same setup. Observe that the trendline is now a bit unoptimal and our RMSE decreased. Do not be alarmed.

In [52]:
```
# HELPER CELL, DO NOT MODIFY
#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)

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: 1.3902
```

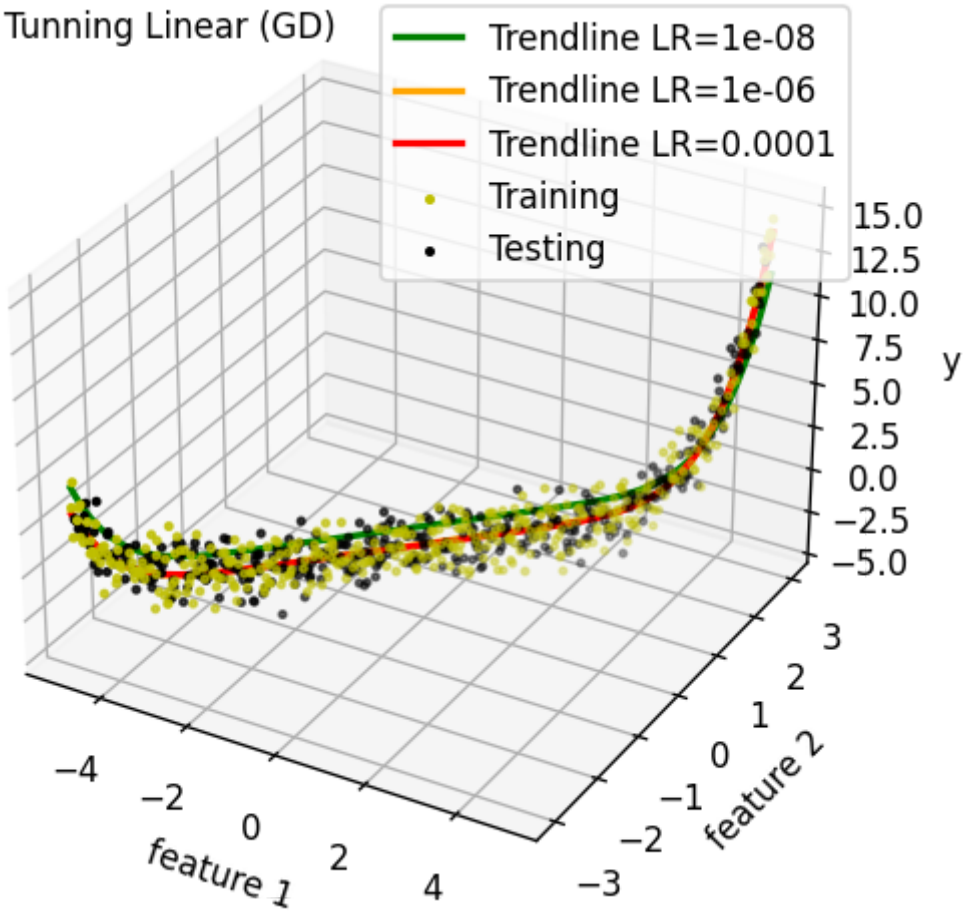


We must tune our epochs and learning_rate. As we tune these parameters our trendline will approach the trendline generated by the linear closed form solution. Observe how we slowly tune (increase) the epochs and learning_rate below to create a better model.

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 excersize to the reader.

In [53]:
```
# HELPER CELL, DO NOT MODIFY
#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]).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)

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: 1.3902 (learning_rate=1e-08)
Linear (GD) RMSE: 0.9487 (learning_rate=1e-06)
Linear (GD) RMSE: 0.9439 (learning_rate=0.0001)
```



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

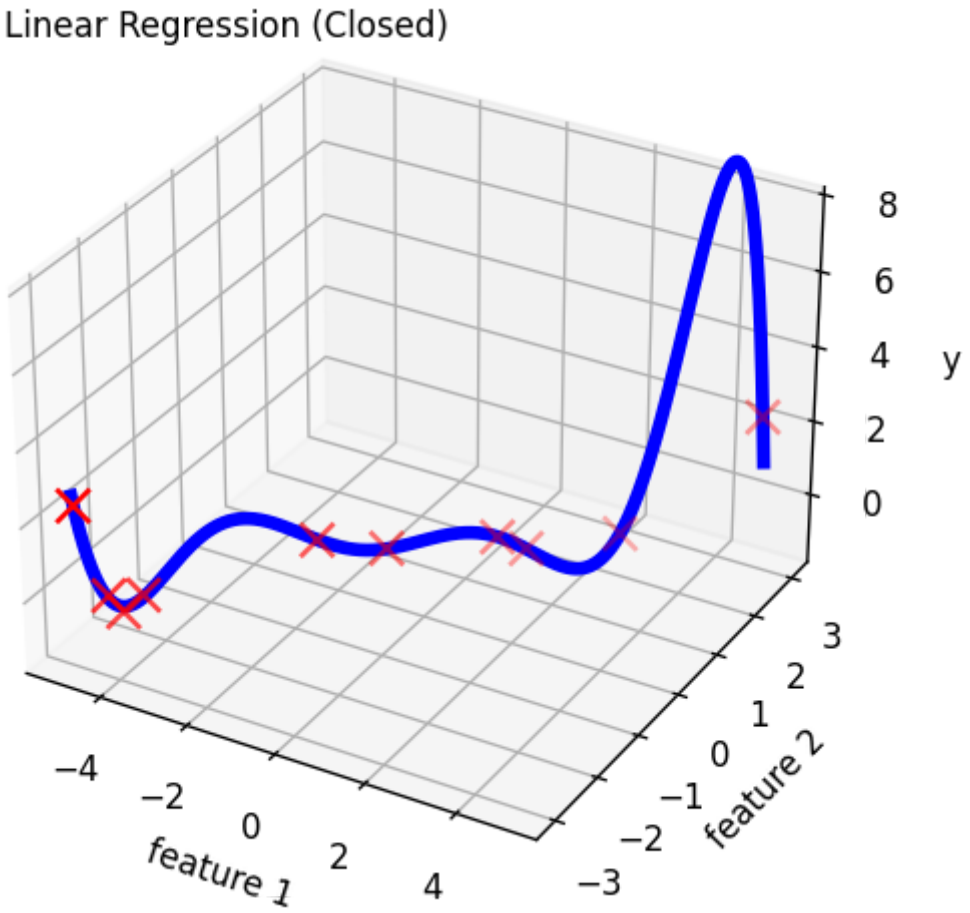
In [54]:
```
# HELPER CELL, DO NOT MODIFY
rng = np.random.RandomState(seed=5)
y_all_noisy = np.dot(x_cart_flat, np.zeros((POLY_DEGREE ** 2 + 2, 1))) + rng.randn(x_all_feat.shape[0
], 1)
sub_train = train_indices[10:20]
```

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

In [55]:
```
# HELPER CELL, DO NOT MODIFY

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: 3.0833
```

Out[55]: Text(0.05, 0.95, 'Linear Regression (Closed)')



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

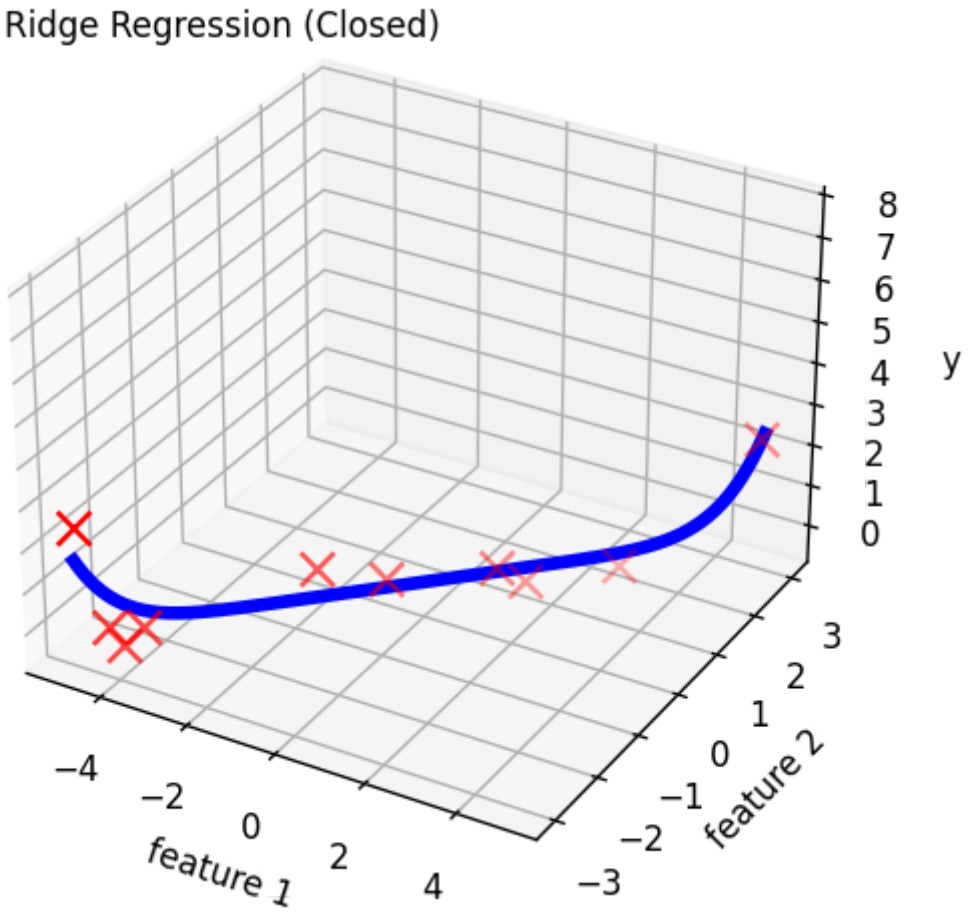
Now let's try ridge regression. Similarly, undergraduate students need to implement the closed form, and graduate students need to implement all the three methods. We will call the prediction function from linear regression part. As long as your test rmse score is close to the TA's answer (TA's answer ± 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 [56]:
```
# HELPER CELL, DO NOT MODIFY
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: 1.0586
```

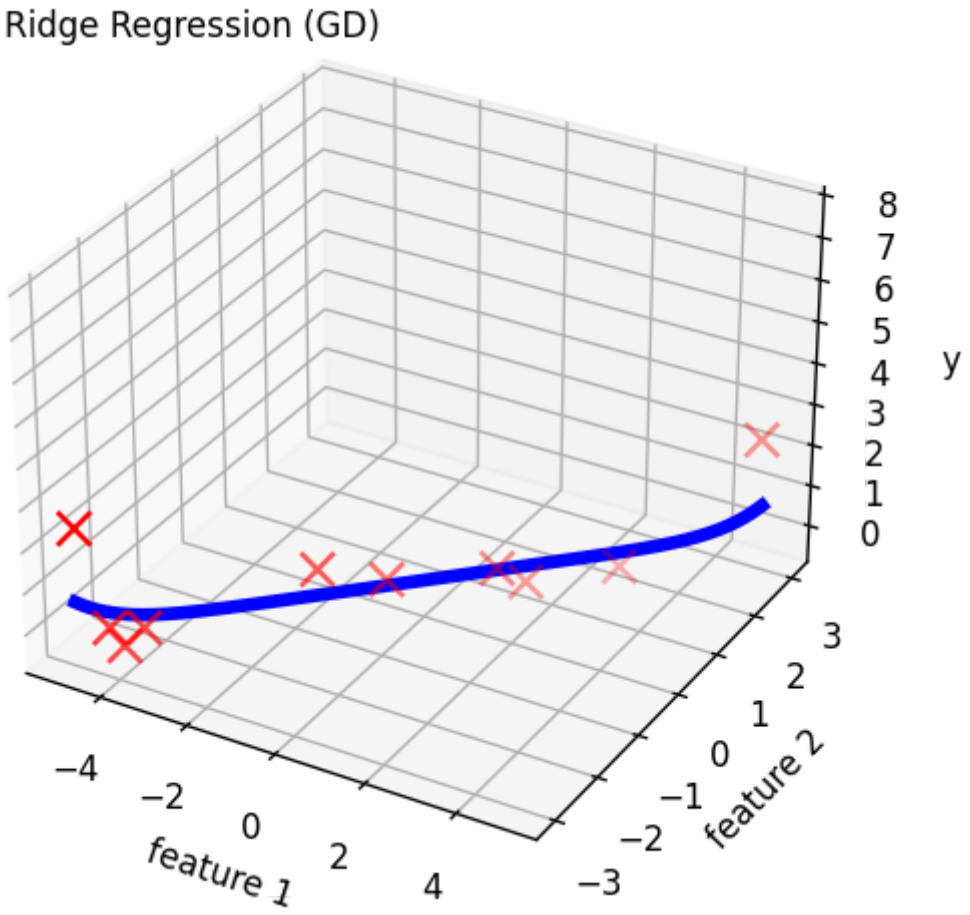
Out[56]: Text(0.05, 0.95, 'Ridge Regression (Closed)')



In [57]:
```
# HELPER CELL, DO NOT MODIFY
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: 0.9638
```

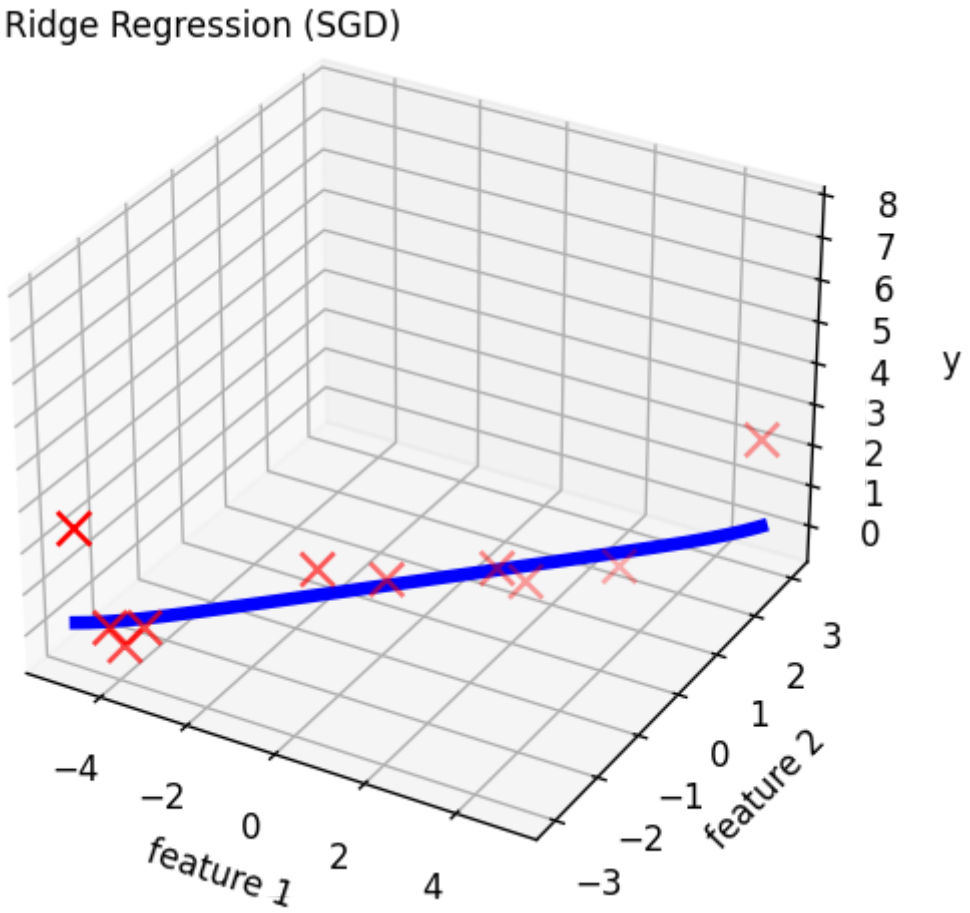
Out[57]: Text(0.05, 0.95, 'Ridge Regression (GD)')



In [58]:
```
# HELPER CELL, DO NOT MODIFY
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: 0.9553
```

Out[58]: 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 [59]:
```
# HELPER CELL, DO NOT MODIFY
# 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 t
he 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.
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: 1.037397
Lambda: 0.0010 RMSE: 1.036370
Lambda: 0.1000 RMSE: 1.036088
Lambda: 1.0000 RMSE: 1.034583
Lambda: 5.0000 RMSE: 1.034768
Lambda: 10.0000 RMSE: 1.039187
Lambda: 50.0000 RMSE: 1.101013
Lambda: 100.0000 RMSE: 1.164980
Lambda: 1000.0000 RMSE: 1.481197
Lambda: 10000.0000 RMSE: 2.206495
Best Lambda: 1.0000
Best Test RMSE: 0.9602
```

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 $x_n = (x_{n1}, \ldots, x_{nD})$ and weights $\theta = (\theta_0, \ldots, \theta_D)$. 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 ϵ_n with zero mean and variance σ^2 is added independently to each of the input sample x_n to generate a new sample set $x' = \{x_1 + \epsilon_1, \ldots, x_n + \epsilon_n\}$. For each sample x_n, $x'_n = (x_{n1} + \epsilon_{n1}, \ldots, x_{nD} + \epsilon_{nd})$, where n and d is independent across both n and d indices.

1. (3pts) Show that $y(x'_n, \theta) = y(x_n, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}$ **Answer** ...

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

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

$$E[E_D(\theta)'] = E_D(\theta) + regularizer$$

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

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

  Answer ...

$$E_D(\theta') = \frac{1}{2}\sum_{n=1}^{N}\{y(x'_n,\theta) - y_n\}^2 = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n+\epsilon_n,\theta) - y_n\}^2$$

We know $(a+b)^2 = a^2 + b^2 + 2ab$. So, if apply E on the both sides of equation we have:

$$E(E_D(\theta')) = E(\frac{1}{2}\sum_{n=1}^{N}\{y(x'_n,\theta) - y_n\}^2) = E(\frac{1}{2}\sum_{n=1}^{N}\{y(x_n+\epsilon_n,\theta) - y_n\}^2))$$

$$= E(\frac{1}{2}\sum_{n=1}^{N}[\{y(x_i,\theta) - y_i\}^2 + (\sum_{d=1}^{D}\theta_d\epsilon_{id})^2 + 2\sum_{n=1}^{N}\{y(x_i,\theta) - y_i\} \times \sum_{d=1}^{D}\theta_d\epsilon_{id}])$$

E can go inside summation and E[mean] = mean.

$$E(E_D(\theta')) = \frac{1}{2}\sum_{n=1}^{N}[\{y(x_i,\theta) - y_i\}^2 + E(\sum_{d=1}^{D}\theta_d\epsilon_{id})^2 + 2 \times E(\sum_{n=1}^{N}\{y(x_i,\theta) - y_i\} \times \sum_{d=1}^{D}\theta_d\epsilon_{id})] = E_D(\theta) + \frac{1}{2}$$

$$\sum_{n=1}^{N}[E(\sum_{d=1}^{D}\theta_d\epsilon_{id})^2 + 2 \times \sum_{n=1}^{N}\{y(x_i,\theta) - y_i\} \times \sum_{d=1}^{D}\theta_d E(\epsilon_{id})]$$

So,by applying zero we have:

$$E(E_D(\theta')) = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}[E(\sum_{d=1}^{D}\theta_d\epsilon_{id})^2]$$

A^2 can be represened as A.A

$$E(E_D(\theta')) = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}[E(\sum_{d=1}^{D}\theta_d\epsilon_{id}\sum_{d=1}^{D}\theta_d\epsilon_{id})] = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}[\sum_{i=1}^{D}\sum_{j=1}^{D}\theta_d\theta_d E(\epsilon_{id}\epsilon_{jd})]$$

Now, we can use above formula:

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

$$E(E_D(\theta')) = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}[E(\sum_{d=1}^{D}\theta_d\epsilon_{id}\sum_{d=1}^{D}\theta_d\epsilon_{id})] = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}[\sum_{i=1}^{D}\sum_{j=1}^{D}\theta_d\theta_d E(\epsilon_{id}\epsilon_{jd})] = E_D(\theta) + \frac{1}{2}\sum_{n=1}^{N}\sigma^2[\sum_{i=1}^{D}\sum_{j=1}^{D}\theta_d\theta_d]$$

$$E(E_D(\theta')) = E_D(\theta) + \frac{\sigma^2}{2}\sum_{n=1}^{N}[\sum_{i=1}^{D}\sum_{j=1}^{D}\theta_d\theta_d] = E_D(\theta) + \frac{\sigma^2}{2}\sum_{n=1}^{N}[\sum_{k=1}^{D}|\theta_k|^2] = E_D(\theta) + regularizer$$

4. Naive Bayes Classification [25pts]**[P]** | **[W]**

In Bayesian classification, we're interested in finding the probability of a label given some observed feature vector $x = [x_1, .., x_d]$, which we can write as $P(y \mid x_1, .., x_d)$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(y \mid x_1, .., x_d) = \frac{P(x_1, .., x_d \mid y)P(y)}{P(x_1, .., x_d)}$$

The main assumption in Naive Bayes is that, given the label, the observed features are conditionally independent i.e.

$$P(x_1, .., x_d \mid y) = P(x_1 \mid y) \times .. \times P(x_d \mid y)$$

Therefore, we can rewrite Bayes rule as

$$P(y \mid x_1, .., x_d) = \frac{P(x_1 \mid y) \times .. \times P(x_d \mid y)P(y)}{P(x_1, .., x_d)}$$

Training Naive Bayes

One way to train a Naive Bayes classifier is done using frequentist approach to calculate probability, which is simply going over the training data and calculating the frequency of different observations in the training set given different labels. For example,

$$P(x_1 = i \mid y = j) = \frac{P(x_1 = i, y = j)}{P(y = j)} = \frac{\text{Number of times in training data } x_1 = i \text{ and } y = j}{\text{Total number of times in training data } y = j}$$

Testing Naive Bayes

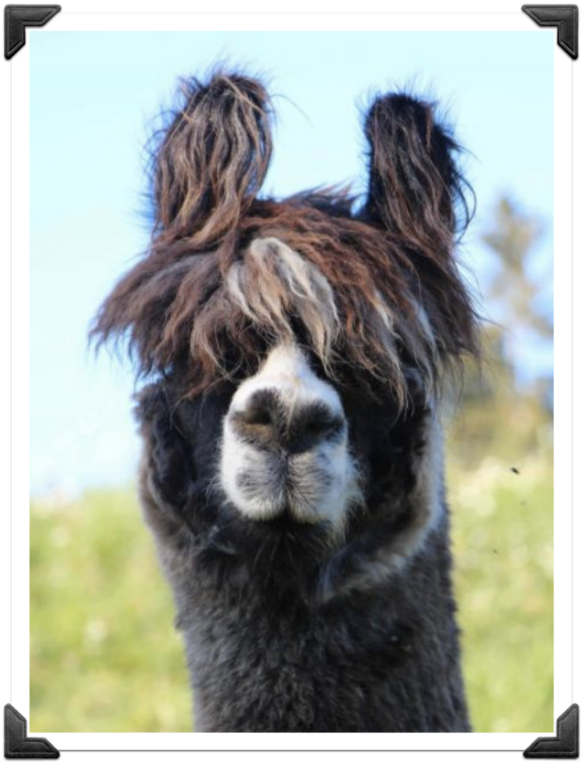
During the testing phase, we try to estimate the probability of a label given an observed feature vector. We combine the probabilities computed from training data to estimate the probability of a given label. For example, if we are trying to decide between two labels y_1 and y_2, then we compute the ratio of the posterior probabilities for each label:

$$\frac{P(y_1 \mid x_1, .., x_d)}{P(y_2 \mid x_1, .., x_d)} = \frac{P(x_1, .., x_d \mid y_1)}{P(x_1, .., x_d \mid y_2)} \frac{P(y_1)}{P(y_2)} = \frac{P(x_1 \mid y_1) \times .. \times P(x_d \mid y_1)P(y_1)}{P(x_1 \mid y_2) \times .. \times P(x_d \mid y_2)P(y_2)}$$

All we need now is to compute $P(x_1|y_i), .., P(x_d \mid y_i)$ and $P(y_i)$ for each label by plugging in the numbers we got during training. The label with the higher posterior probabilities is the one that is selected.

4.1 Llama Breed Problem [5pts] **[W]**

Suri Llama



Wooly Llama

Above are images of two different breeds of llamas – the suri llama and the wooly llama. The difference between these two breeds is subtle, as these two breeds are often mixed up. However the Suri Llama is vastly more valuable than the wooly llama. You devise a way to determine with some confidence, which llama is which – without the need for expensive genetic testing. You look at four key features of the llama: {curly hair, over 14 inch tail, over 400 pounds, extremely shy/timed}. You only have 6 randomly chosen llamas to work with, and their breed as the ground truth. You record the data as vectors with the entry 1 if true and 0 if false.

For example a llama with {1,1,0,1} would have curly hair, a tail over 14 inches, be **less** than 400 pounds, and be extreamly shy/timed.

The **Suri Llamas** yield the following data: {1,0,0,0}, {1,0,0,1}, {1,1,1,1}, {0,0,0,1}

The **Wooly Llamas** yield the following data: {0,1,1,0}, {1,1,1,0}

Now is time for the test of your method. You see a new llama you are interested in that does **not** have curly hair, has a tail over 14 inches, is over 400 pounds, and is **not** shy/timid. Using naive Bayes, **Is this new llama a Suri Llama?**

Answer ...

Based on the formula we have:

$p(y_1) = \frac{2}{3} : p(x_1|y_1) = \frac{3}{4}, p(x_2|y_1) = \frac{1}{4}, p(x_3|y_1) = \frac{1}{4}, p(x_4|y_1) = \frac{3}{4}$

$p(y_2) = \frac{1}{3} : p(x_1|y_2) = \frac{1}{2}, p(x_2|y_2) = 1, p(x_3|y_2) = 1, p(x_4|y_2) = 0$

$$\frac{P(y_1 \mid x_1, .., x_d)}{P(y_2 \mid x_1, .., x_d)} = \frac{P(x_1, .., x_d \mid y_1)}{P(x_1, .., x_d \mid y_2)} \frac{P(y_1)}{P(y_2)} = \frac{P(x_1 \mid y_1) \times .. \times P(x_d \mid y_1) P(y_1)}{P(x_1 \mid y_2) \times .. \times P(x_d \mid y_2) P(y_2)}$$

$$\frac{P(y_1 \mid 0,1,1,0)}{P(y_2 \mid 0,1,1,0)} = \frac{P(x_1, .., x_d \mid y_1)}{P(x_1, .., x_d \mid y_2)} \frac{P(y_1)}{P(y_2)} = \frac{P(x_1 \mid y_1) \times .. \times P(x_d \mid y_1) P(y_1)}{P(x_1 \mid y_2) \times .. \times P(x_d \mid y_2) P(y_2)} = \frac{\frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{2}{3}}{1 \times 1 \times 1 \times \frac{1}{2} \times \frac{2}{3}} = \frac{1}{64}$$

So, it is wooly

4.2 Determining Political Party Affiliation from Twitter Data [15pts] **[P]**

Brian was recently hired by the Washington Post to analyze tweets from select Twitter Accounts to determine political party affiliation (ex: republican, democrat, etc.). Brian, a skilled CS4641/7641 alumnus himself, decides to use a Naive Bayes approach to classify tweets as republican or democratic.

This dataset has 2 classes, republican (class label = 0) and democratic (class label = 1). There are over 80,000 tweets. However, to save computational time as well as memory resources, the dataset has been reduced to 29,115 unique tweets after removing duplicates. These tweets have also been cleaned to remove various artifacts (such as emojis, etc.) as well as extra punctuation. The dataset is then split into a training and testing dataset that has a 8:2 ratio.

The code which is provided loads the tweets and builds a "bag of words" representation (https://en.wikipedia.org/wiki/Bag-of-words_model) of each tweet. Your task is to complete the missing portions of the code and to determine whether the author of a tweet is republican or democratic. (Hint: Label 0 denotes the tweet is republican, label 1 denotes the tweet is democratic. Our job here is to determine whether a tweet is republican or democratic using Naive Bayes).

priors_prob function calculates the ratio of class probabilities of negative, neutral or positive. We do this based on word counts rather than document counts.

likelihood_ratio function calculates the ratio of word probablities given the label of whether the tweet is republican or democratic.

analyze_affiliation function takes in the likelihood ratio, priors probabilities for each class and a number of test news represented in Bag-of-Words representation, and analyzes the party affiliation for each tweet.

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"

$$\begin{bmatrix} label & machine & learning \\ 0(republican) & 1 & 4 \\ 0 & 0 & 6 \\ 1(democratic) & 3 & 2 \\ 0 & 3 & 1 \\ 1 & 4 & 0 \end{bmatrix}$$

Then we have

$$prior(republican) = \frac{1+4+0+6+3+1}{1+4+0+6+3+2+3+1+4+0} = \frac{15}{24}$$
$$prior(democratic) = \frac{3+2+4+0}{1+4+0+6+3+2+3+1+4+0} = \frac{9}{24}$$

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. 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).

$$likelihood(republican) = [\frac{1+0+3+\mathbf{1}}{1+0+3+\mathbf{1}+4+6+1+\mathbf{1}} \ \frac{4+6+1+\mathbf{1}}{1+0+3+\mathbf{1}+4+6+1+\mathbf{1}}] = [\frac{5}{17} \ \frac{12}{17}]$$
$$likelihood(democratic) = [\frac{3+4+\mathbf{1}}{3+4+\mathbf{1}+2+0+\mathbf{1}} \ \frac{2+0+\mathbf{1}}{3+4+\mathbf{1}+2+0+\mathbf{1}}] = [\frac{8}{11} \ \frac{3}{11}]$$

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

	Machine	**Learning**
Count	3	4

$$P(\text{republican}) = (\frac{5}{17})^3 * (\frac{12}{17})^4 * \frac{15}{24}$$
$$P(\text{democratic}) = (\frac{8}{11})^3 * (\frac{3}{11})^4 * \frac{9}{24}$$

The prediction will then be the label with the highest probability

In [60]:
```
from nb import NaiveBayes
import pandas as pd
import preprocessor as p  #if you do not have this module, install via 'pip install tweet-preprocesso
r'
import re
```

In [61]:
```
# HELPER CELL, DO NOT MODIFY
'''
Read data and convert to BOW matrix
DO NOT modify this function
'''

train = pd.read_csv("./data/ExtractedTweets.csv", usecols=["Party", "Tweet", "Handle"], dtype={"Part
y":'string', "Tweet":'string', "Handle":"string"})

class_to_label_mappings = {
    "Republican": 0,
    "Democratic": 1,
}

print("Reducing Dataset...")
new_train = pd.DataFrame(columns=["Party", "Handle", "Tweet"])
for handle in train["Handle"].unique():
    new_train = new_train.append(train[train["Handle"]==handle][:10], ignore_index=True)
train = new_train.drop(columns="Handle")

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

print("Cleaning Dataset...")
for _, row in train.iterrows():
    row['Tweet'] = clean_tweet(row['Tweet'])

train.columns = ["Party", "Tweet"]
train.drop_duplicates(inplace=True)

train["Party"] = train["Party"].map(
    class_to_label_mappings)

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

X = train['Tweet'].values
y = train['Party'].values

print("Preparing Dataset...")
```

```
RANDOM_SEED = 5
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)

X_republican = X_train[y_train == 0]
X_democratic = X_train[y_train == 1]
```

```
Reducing Dataset...
Cleaning Dataset...
Preparing Dataset...
```

In [62]:
```
# HELPER CELL, DO NOT MODIFY
NB = NaiveBayes()
likelihood_ratio = NB.likelihood_ratio(X_republican, X_democratic)
priors_prob = NB.priors_prob(X_republican, X_democratic)
resolved = NB.analyze_affiliation(likelihood_ratio, priors_prob, X_test)

# You should be getting around 51% of accuracy
print("test accuracy: ", np.sum(resolved == y_test) / resolved.shape[1])
```

```
test accuracy:  0.5168269230769231
```

4.3 Accuracy result analysis [5pts] **[W]**

Do you think this is a good accuracy? What assumptions can you make that limit the accuracy? (This is an open question, any reasonable assumptions will be acceptable).

Answer ...

I would say no. 51\% is more like random guessing. The assumption that features (words) are independet of each other has made the accuracy low. In general, they are not complemetly of each other. For example, after "the", "a", "an" we have always a nou and so on. Considering those can increase the accuracy.

5 Noise in PCA and Linear Regression (15 Pts) **[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 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.

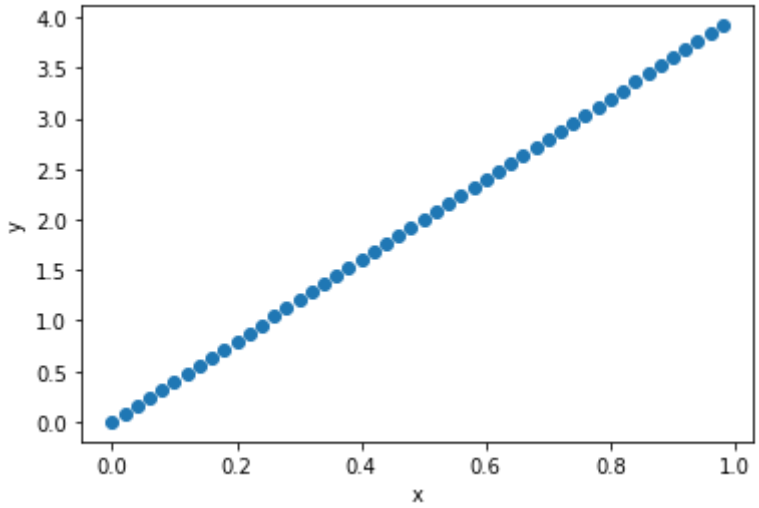
As a reminder, least squares regression minimizes the squared error of the dependent variable from its prediction. Namely, given (x_i, y_i) pairs, least squares returns the line $l(x)$ that minimizes $\sum_i (y_i - l(x_i))^2$.

In [63]:
```
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,) vector of feature x
        y: (N,) vector of feature y
    Return:
        slope: slope of the first principal component
    """
    concat_vector = np.concatenate((np.expand_dims(x,1), np.expand_dims(y,1)),axis=1)
    mypca = PCA()
    mypca.fit(concat_vector)
    mypcaV = mypca.get_V()[0,:]
    return mypcaV[1]/mypcaV[0]

    #return pcapcaTransform.shape(1)
    # TODO: implement this function


def lr_slope(X, y):
    """
    Calculates the slope of the best fit as given by Lineear Regression

    For this function don't use any regularization

    Args:
        X: N*1 array corresponding to a dataset
        y: N*1 array of labels y
    Return:
        slope: slope of the best fit
    """

    # TODO: implement this function
    myregression = Regression()
    return myregression.linear_fit_closed(X,y)[0]
```

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, . . . , 1] and y = 4x. Make sure both functions return 4.

In [64]:
```
# HELPER CELL, DO NOT MODIFY
x = np.arange(0, 1, 0.02)
y = 4 * np.arange(0, 1, 0.02)

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

print("Slope of best linear fit", lr_slope(x[:, None], y))

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
```

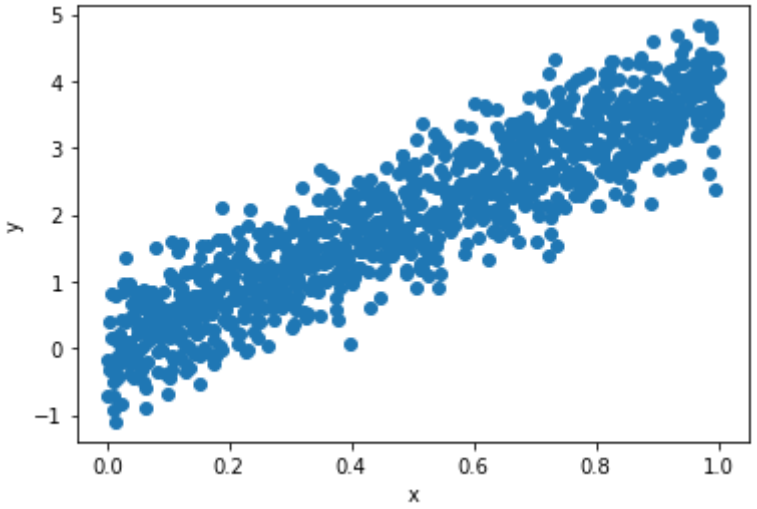


5.2 Analysis Setup (5 Pts) **[W]**

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 [65]:
```
# HELPER CELL, DO NOT MODIFY
base = np.arange(0.001, 1.001, 0.001)
c = 0.5
X = base
y = 4 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

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 X where $X = [x_1, x_2, \ldots, x_{1000}] = [.001, .002, .003, \ldots, 1]$.

2. For a given noise level c, set $\hat{y}_i \sim 4x_i + \mathcal{N}(0, c) = 4i/1000 + \mathcal{N}(0, c)$, and $\hat{Y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{1000}]$. You can use the np.random.normal function, where scale is equal to noise level, to add noise to your points.

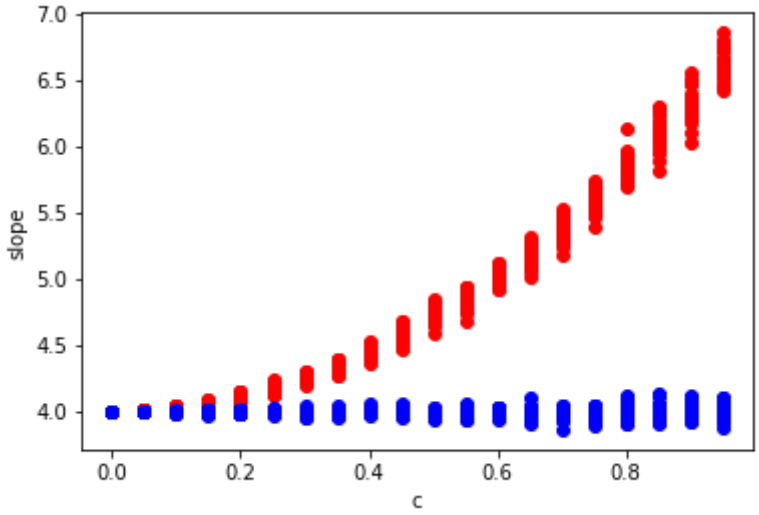
3. Return the **pca_slope** and **lr_slope** values of this X and $\hat{Y}$ dataset you have created where only $\hat{Y}$ has noise.

In [66]:
```
from pca import PCA
from regression import Regression

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: 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
        seed: Random seed
    Return:
        pca_slope_value: slope value of dataset created using pca_slope
        lr_slope_value: slope value of dataset created using lr_slope

    """

    # TODO: implement this function
    firstVar = np.arange(0, 1.001, 0.001)
    var = 4*firstVar + np.random.normal(loc=[0], scale=c, size=firstVar.shape)
    if x_noise:
        firstVar = firstVar + np.random.normal(loc=[0], scale=c, size=firstVar.shape)
    pca_slope_value = pca_slope(firstVar, var)
    lr_slope_value = lr_slope(firstVar.reshape([-1,1]),var)

    return pca_slope_value, lr_slope_value
```

A scatter plot with 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 $\hat{Y}$ has been taken for each c in $[0, 0.05, 0.1, \ldots, .95, 1.0]$. 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.

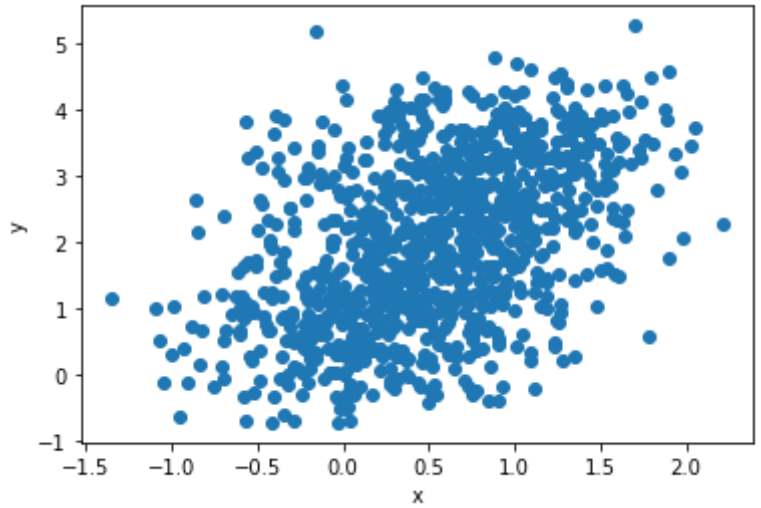
In [67]:
```
# HELPER CELL, DO NOT MODIFY
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

for i in range(30):
    for c in np.arange(0, 1, 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

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

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 [68]:
```
# HELPER CELL, DO NOT MODIFY
base = np.arange(0.001, 1.001, 0.001)
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)

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



In **addNoise** function you created in the previous step, you will modify the following:

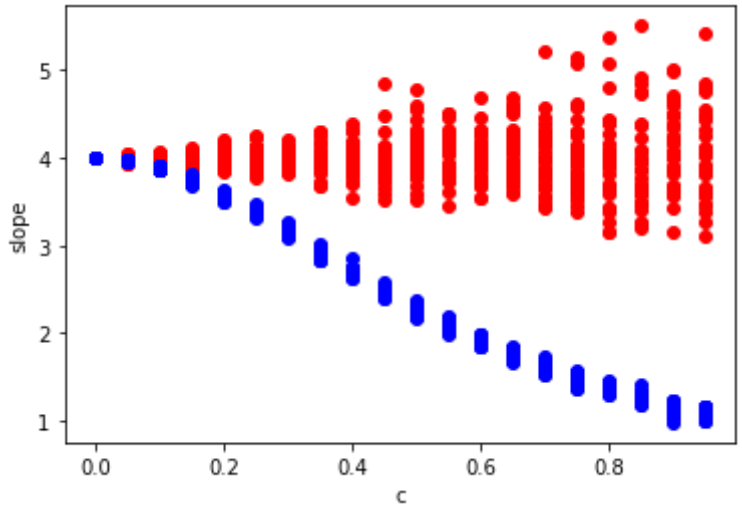
1. Notice the parameter **x_noise** in the **addNoise** function. When this parameter is set to $True$, you will have to add noise to X. For a given noise level c, let $\hat{x}_i \sim x_i + \mathcal{N}(0, c) = i/1000 + \mathcal{N}(0, c)$, and $\hat{X} = [\hat{x}_1, \hat{x}_2, \ldots . \hat{x}_{1000}]$

2. For the same noise level c, set $\hat{y}_i \sim 4x_i + \mathcal{N}(0, c) = 4i/1000 + \mathcal{N}(0, c)$, and $\hat{Y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{1000}]$. Again, you can use *np.random.normal function to add noise to your points.

3. Return the **pca_slope** and **lr_slope** values of this $\hat{X}$ and $\hat{Y}$ dataset you have created where both $\hat{X}$ and $\hat{Y}$ have noise.

A scatter plot with 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 $\hat{X}$ and $\hat{Y}$ has been taken for each c in $[0, 0.05, 0.1, \ldots, .95, 1.0]$. 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.

In [69]:
```
# HELPER CELL, DO NOT MODIFY
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

for i in range(30):
    for c in np.arange(0, 1, 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

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()
```


slope
c
5
4
3
2
1
0.0
0.2
0.4
0.6
0.8

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)

Answer ...

In the case that we have noise in one side (i.e., y), PCA perfroms worse as it makes the data bias toward one side.

1. Why does PCA perform better in the other case? (1 Pt)

Answer ...

In the case that we have noise in both sides (i.e., x, y), PCA perfroms better as it makes the data unbias.

1. Which case does Linear Regression perform well? Why does Linear Regression perform well in this case? (2 Pts)

Answer ...

Since in linear regression we have access to the independet variable (i.e, x), it acts better in case we have noise in one side (i.e., y)

6 Feature Selection [Bonus for everyone] [25 Points] **[P]** | **[W]**

6.1 Implementation [15 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.

Implement a method to find the final list of significant features due to forward selection and backward elimination.

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 test hypothesis. It tests all the assumptions about how the data was generated in the model, not just the target hypothesis it was supposed to test. 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. 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 [35]:
```
# HELPER CELL, DO NOT MODIFY
import pandas as pd
import statsmodels.api as sm
from sklearn import linear_model
from feature_selection import FeatureSelection

data_feature_names = np.array(load_diabetes().feature_names)

data = pd.DataFrame(load_diabetes().data,
                    columns = data_feature_names)

print("All Features:\n", data_feature_names)

data['age'] = load_diabetes().target
X = data.drop("age", 1)  # feature matrix
y = data['age']  # target feature

# Reducing the number of samples to make dataset harder to work with.
X = X[:100]
y = y[:100]

featureselection = FeatureSelection()
#Run the functions to make sure two lists are generated, one for each method
print("\nFeatures selected by forward selection:\n",
      sorted(featureselection.forward_selection(X, y)))
print("Features selected by backward elimination:\n",
      sorted(featureselection.backward_elimination(X, y)[0]))

clf = linear_model.LassoCV()
clf.fit(X, y)
data_feature_names_X = np.delete(data_feature_names, np.where(data_feature_names == 'age'))
print("Features selected with other advanced methods:\n",
      sorted(data_feature_names_X[np.where(clf.coef_ != 0)]))
```

```
All Features:
 ['age' 'sex' 'bmi' 'bp' 's1' 's2' 's3' 's4' 's5' 's6']

---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-35-cc404a899a86> in <module>
     23 #Run the functions to make sure two lists are generated, one for each method
     24 print("\nFeatures selected by forward selection:\n",
---> 25       sorted(featureselection.forward_selection(X, y)))
     26 print("Features selected by backward elimination:\n",
     27       sorted(featureselection.backward_elimination(X, y)[0]))

~/Dropbox/G/PhD/Gatech/ML/HWs/HW3/feature_selection.py in forward_selection(data, target, significan
ce_level)
     21
     22         '''
---> 23         raise NotImplementedError
     24
     25     @staticmethod

NotImplementedError:
```

6.2 Feature Selection - Discussion [5pts] **[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? **(2 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 and backward selection, what fraction of the total possible models are we exploring? Assume that the total number of features that we have at our disposal is N. **(3 pts)**

6.3 Implementing Closed Form Solution for Lasso [Bonus for everyone] [5 pts] **[W]**

You have two options for this bonus problem. **Only one of the two options is correct.** You can either: (1) implement a closed solution for Lasso below in Python, or (2) you can explain in detail why one cannot implement a closed solution for Lasso.

Notes: We are not asking you to implement gradient descent lasso or stochastic gradient descent lasso. We are asking you to implement the close form solution for lasso in Python. This implementation must be able to handle n dimensions. For option (2) you must give reasonable proof that a closed form solution for Lasso cannot be implemented.

Option 1 | Implementation

In [ ]:
```
""" No need to add this function to regression.py. """
def lasso_fit_closed(xtrain, ytrain, c_lambda):
    """
    Args:
        xtrain: N x D numpy array, where N is number of instances and D is the dimensionality of each
instance
        ytrain: N x 1 numpy array, the true labels
        c_lambda: floating number
    Return:
        weight: Dx1 numpy array, the weights of lasso regression model
    """
    raise NotImplementedError
```

Option 2 | Explanation

Type your answer here...

In [ ]:
```
```