Fall 2022 CS4641/CS7641 Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, November 11, 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 Programming sections. Please refer to the Deliverables and Point Distribution section for 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 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 Written Portion, you will need to submit your Jupyter Notebook as a PDF on the Gradescope Non-Programming section.** If you need help converting the Jupter Notebook to a PDF, please see this EdStem post. Please refer to the **Deliverables and Point Distribution** section for an outline of the non-programming written portion questions.
- **When submitting to Gradescope, please make sure to mark the page(s) corresponding to each problem/sub-problem. The pages in the PDF should be of size 8.5" x 11", otherwise there may be a deduction in points for extra long sheets.**

Using the local tests

- For some of the programming questions we have included a local test using a small toy dataset to aid in debugging. The local test sample data and outputs are stored in .py files in the **local_tests_folder**. The actual local tests are stored in **localtests.py**.
- There are no points associated with passing or failing the local tests, you must still pass the autograder to get points.
- **It is possible to fail the local test and pass the autograder** since the autograder has a certain allowed error tolerance while the local test allowed error may be smaller. Likewise, passing the local tests does not guarantee passing the autograder.
- **You do not need to pass both local and autograder tests to get points, passing the Gradescope autograder is sufficient for credit.**
- It might be helpful to comment out the tests for functions that have not been completed yet.
- It is recommended to test the functions as it gets completed instead of completing the whole class and then testing. This may help in isolating errors. Do not solely rely on the local tests, continue to test on the autograder regularly as well.

Deliverables and Points Distribution

Q1: Image Compression [30pts]

Deliverables: imgcompression.py and printed results

- **1.1 Image Compression** [20 pts] - *programming*
  - svd [4pts]
  - compress [4pts]
  - rebuild_svd [4pts]
  - compression_ratio [4pts]
  - recovered_variance_proportion [4pts]
- **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] *programming and non-programming*
- **2.3 PCA Reduced Emotion Dataset Analysis** [5 pts] *non-programming*

Q3: Regression and Regularization [80pts: 50pts + 20pts Bonus for Undergrads + 10pts Bonus for All]

Deliverables: regression.py and Written portion

- **3.1 Regression and Regularization Implementations** [50pts: 30pts + 20pts 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* **BONUS FOR ALL**

Q4: Naive Bayes and Logistic Regression [35pts]

Deliverables: logistic_regression.py and Written portion

- **4.1 Llama Breed Problem using Naive Bayes** [5 pts] *non-programming*
- **4.2 News Data Sentiment Classification Using Logistic Regression** [30 pts] - *programming*
    - sigmoid [2 pts]
    - bias_augment [3 pts]
    - predict_probs [5 pts]
    - predict_labels [2 pts]
    - loss [3 pts]
    - gradient [3 pts]
    - accuracy [2 pts]
    - evaluate [5 pts]
    - fit [5 pts]

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*

Q7: Movie Recommendation with SVD [10pts Bonus for All]

Deliverables: svd_recommender.py and Written portion

- **7.1 SVD Recommender**
    - recommender_svd [5pts]
    - predict [5pts]
- **7.2 Visualize Movie Vectors** [0pts]

0 Set up

This notebook is tested under python 3. . , and the corresponding packages can be downloaded from miniconda. You may also want to get yourself familiar with several packages:

- jupyter notebook
- numpy
- matplotlib
- sklearn
- Axes3D

There is also a VS Code and Anaconda Setup Tutorial 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 [ ]:
```
##############################
### 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
import matplotlib
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 sys
import re
import gzip
from tqdm.notebook import tqdm

print('Version information')

print('python: {}'.format(sys.version))
print('matplotlib: {}'.format(matplotlib.__version__))
print('numpy: {}'.format(np.__version__))

warnings.filterwarnings('ignore')

%matplotlib inline
%load_ext autoreload
%autoreload 2

STUDENT_VERSION = 1
EO_TEXT, EO_FONT, EO_COLOR = 'TA VERSION', 'Arial', 'gray',
EO_ALPHA, EO_SIZE, EO_ROT = 0.5, 90, 40
```

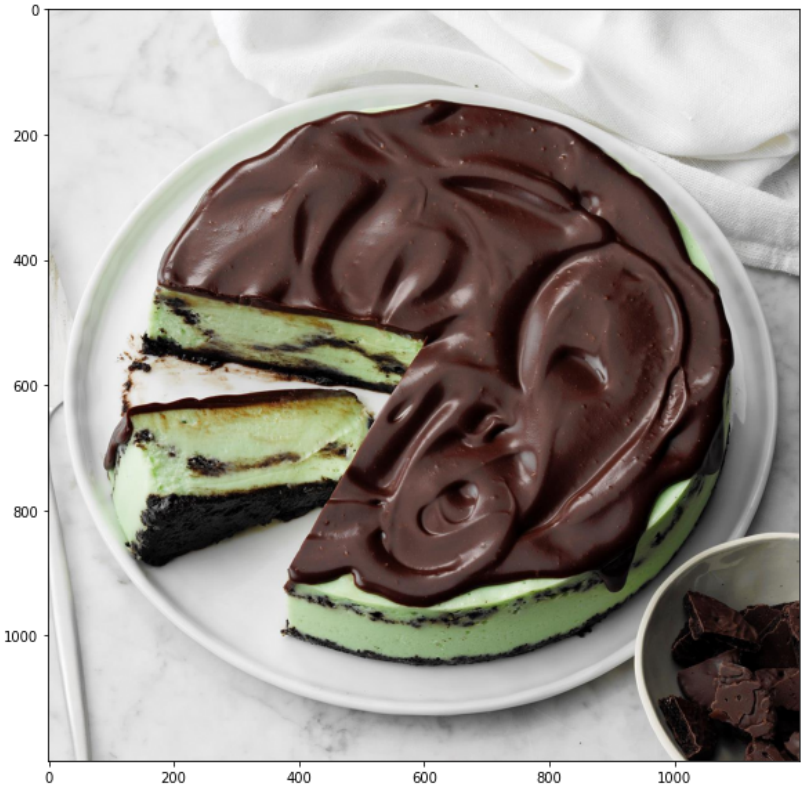
```
Version information
python: 3.10.6 (main, Nov  2 2022, 18:53:38) [GCC 11.3.0]
matplotlib: 3.5.2
numpy: 1.22.3
```

Q1: Image Compression [30 pts] **[P]** | **[W]**

Load images data and plot

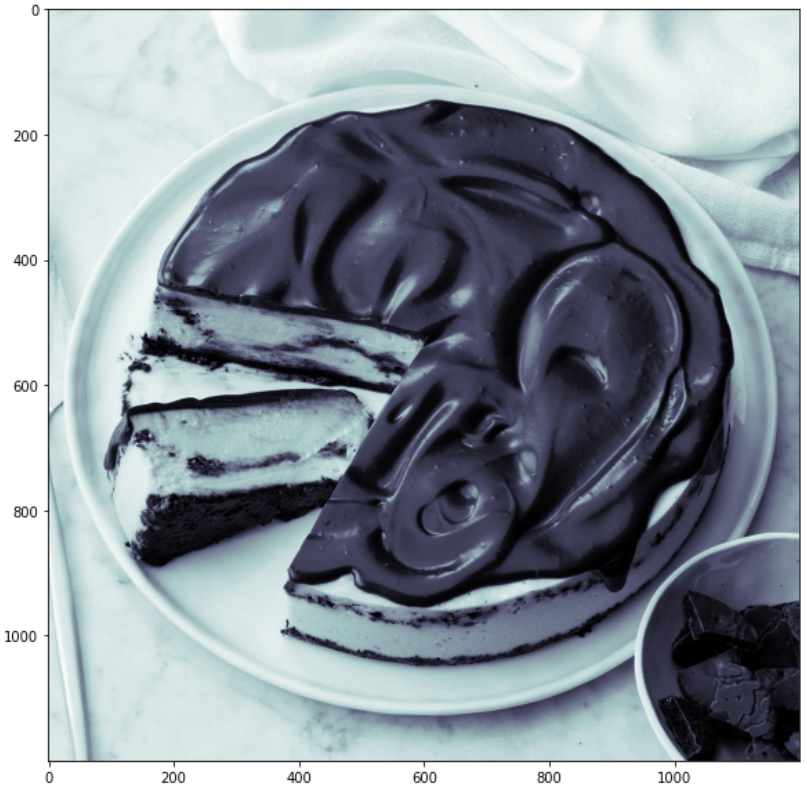
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
# load Image
image = plt.imread("./data/hw3_image_compression.jpg")/255
#plot image
fig = plt.figure(figsize=(10,10))
if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)
plt.imshow(image)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7feb985164d0>
```



In [ ]:
```
##############################
### 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))

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)
# plot several images
plt.imshow(rgb2gray(image), cmap=plt.cm.bone)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7feb98392bf0>
```



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.
- **compress**
- **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/ 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 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.

1.1.1 Local Tests for Image Compression Black and White Case [No Points]

You may test your implementation of the functions contained in **imgcompression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

```
from localtests import TestImgCompression

unittest_ic = TestImgCompression()
unittest_ic.test_svd_bw()
unittest_ic.test_compress_bw()
unittest_ic.test_rebuild_svd_bw()
unittest_ic.test_compression_ratio_bw()
unittest_ic.test_recovered_variance_proportion_bw()
```

```
UnitTest passed successfully for "SVD calculation - black and white images"!
UnitTest passed successfully for "Image compression - black and white images"!
UnitTest passed successfully for "SVD reconstruction - black and white images"!
UnitTest passed successfully for "Compression ratio - black and white images"!
UnitTest passed successfully for "Recovered variance proportion - black and white images"!
```

1.1.2 Local Tests for Image Compression Color Case [No Points]

You may test your implementation of the functions contained in **imgcompression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

from localtests import TestImgCompression

unittest_ic = TestImgCompression()

unittest_ic.test_svd_color()
unittest_ic.test_compress_color()
unittest_ic.test_rebuild_svd_color()
unittest_ic.test_compression_ratio_color()
unittest_ic.test_recovered_variance_proportion_color()
```

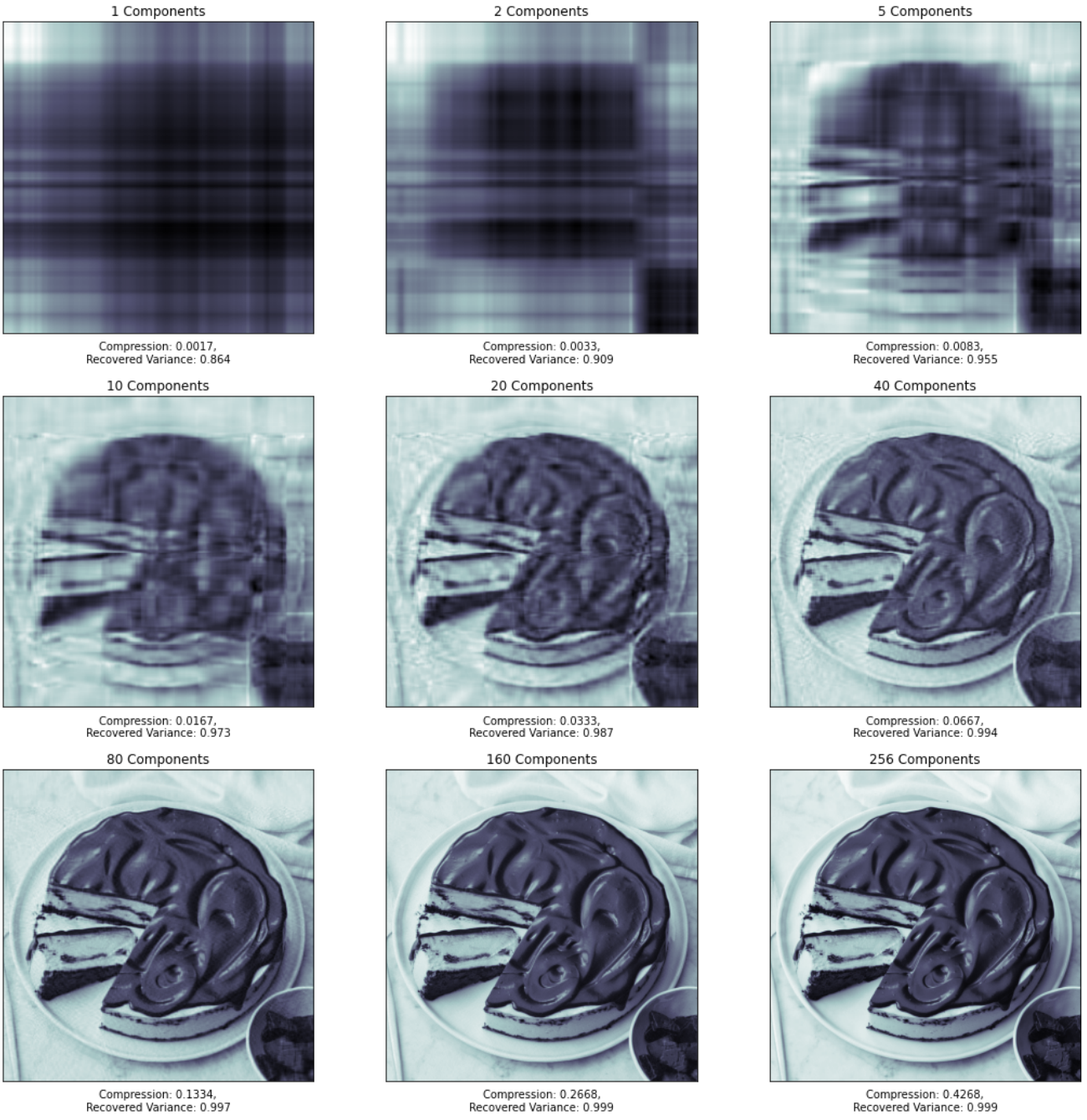
```
UnitTest passed successfully for "SVD calculation - color images"!
UnitTest passed successfully for "Image compression - color images"!
UnitTest passed successfully for "SVD reconstruction - color images"!
UnitTest passed successfully for "Compression ratio - color images"!
UnitTest passed successfully for "Recovered variance proportion - color images"!
```

1.2.1 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 [ ]:
```
##############################
### 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:
    U_compressed, S_compressed, V_compressed = imcompression.compress(U, S, V, k)
    img_rebuild = imcompression.rebuild_svd(U_compressed, S_compressed, V_compressed)
    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")
    if not STUDENT_VERSION:
        ax.text(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
            fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
            ha='center', va='center', rotation=EO_ROT)
    ax.set_xlabel(f"Compression: {c},\nRecovered Variance: {r}")
    i = i+1
```



1.2.2 Black and White Compression Savings [No Points]

This question will use your implementation of the functions from Q1.1 to compare the number of bytes required to represent the SVD decomposition for the original image to the compressed image using different degrees of compression. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

Running this cell is primarily for your own understanding of the compression process.

In [ ]:
```
##############################
### 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]

# Compare memory savings for BW image
for k in component_num:
    og_bytes, comp_bytes, savings = imcompression.memory_savings(bw_image, U, S, V, k)
    og_bytes = imcompression.nbytes_to_string(og_bytes)
```

```
    comp_bytes = imcompression.nbytes_to_string(comp_bytes)
    savings = imcompression.nbytes_to_string(savings)
    print(f"{k} components: Original Image: {og_bytes} -> Compressed Image: {comp_bytes}, Savings: {savings}")
```

```
1 components: Original Image: 10.986 MB -> Compressed Image: 18.758 KB, Savings: 10.968 MB
2 components: Original Image: 10.986 MB -> Compressed Image: 37.516 KB, Savings: 10.95 MB
5 components: Original Image: 10.986 MB -> Compressed Image: 93.789 KB, Savings: 10.895 MB
10 components: Original Image: 10.986 MB -> Compressed Image: 187.578 KB, Savings: 10.803 MB
20 components: Original Image: 10.986 MB -> Compressed Image: 375.156 KB, Savings: 10.62 MB
40 components: Original Image: 10.986 MB -> Compressed Image: 750.312 KB, Savings: 10.254 MB
80 components: Original Image: 10.986 MB -> Compressed Image: 1.465 MB, Savings: 9.521 MB
160 components: Original Image: 10.986 MB -> Compressed Image: 2.931 MB, Savings: 8.055 MB
256 components: Original Image: 10.986 MB -> Compressed Image: 4.689 MB, Savings: 6.297 MB
```

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

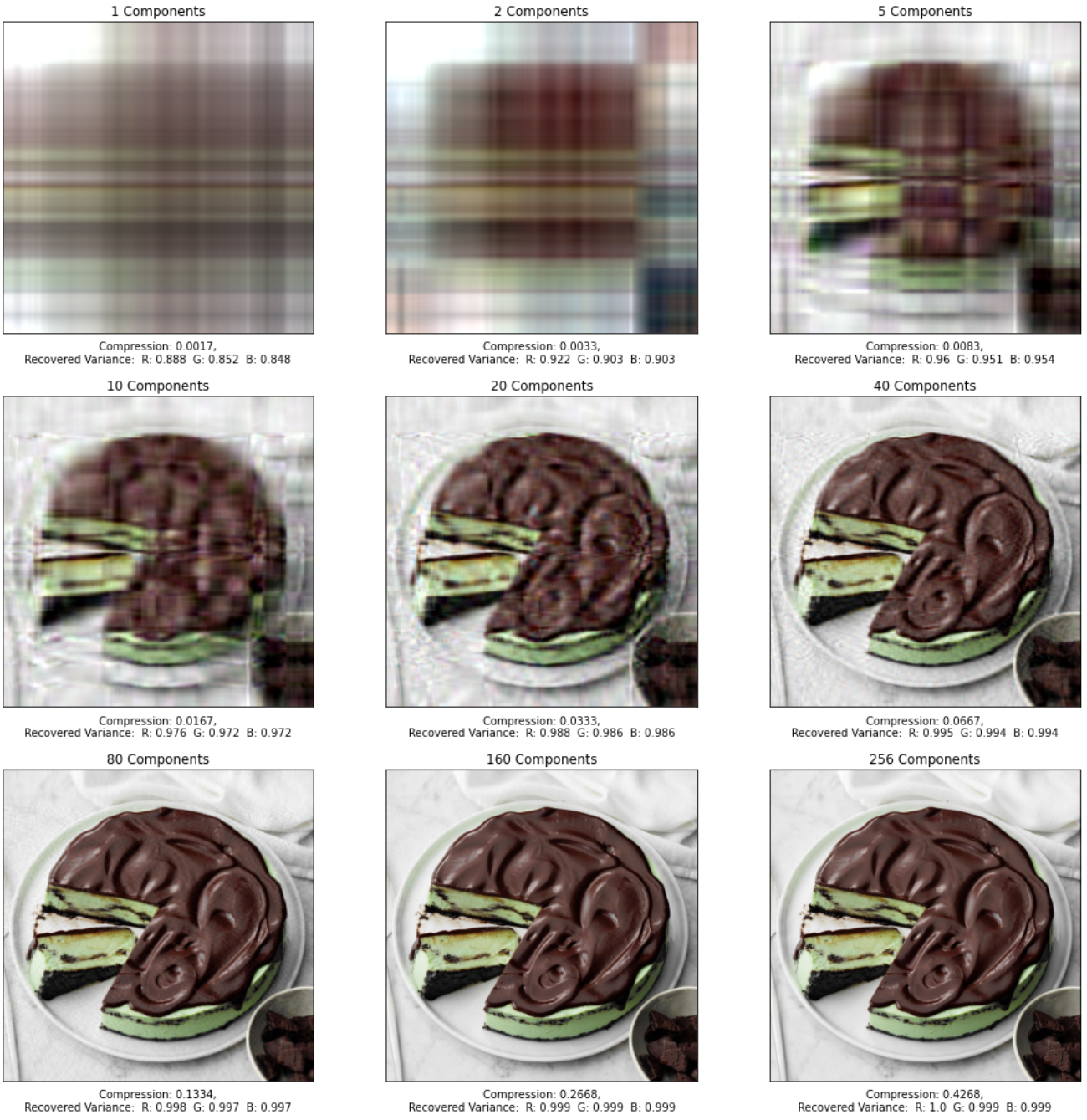
This section will use your implementation of the functions from Q1.1 to generate a set of images compressed to different degrees. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

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 [ ]:
```
##############################
### 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:
    U_compressed, S_compressed, V_compressed = imcompression.compress(U, S, V, k)
    img_rebuild = np.clip(imcompression.rebuild_svd(U_compressed, S_compressed, V_compressed),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")
    if not STUDENT_VERSION:
        ax.text(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
            fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
            ha='center', va='center', rotation=EO_ROT)
    ax.set_xlabel(f"Compression: {np.around(c,4)},\nRecovered Variance:  R: {r[0]}  G: {r[1]}  B: {r[2]}")
    i = i+1
```



1.3.2 Color Compression Savings [No Points]

This question will use your implementation of the functions from Q1.1 to compare the number of bytes required to represent the SVD decomposition for the original image to the compressed image using different degrees of compression. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

Running this cell is primarily for your own understanding of the compression process.

In [ ]:
```
##############################
### 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]

# Compare the memory savings of the color image
i=0
for k in component_num:
    og_bytes, comp_bytes, savings = imcompression.memory_savings(image, U, S, V, k)
    og_bytes = imcompression.nbytes_to_string(og_bytes)
```

```
    comp_bytes = imcompression.nbytes_to_string(comp_bytes)
    savings = imcompression.nbytes_to_string(savings)
    print(f"{k} components: Original Image: {og_bytes} -> Compressed Image: {comp_bytes}, Savings: {savings}")
```

```
1 components: Original Image: 32.959 MB -> Compressed Image: 56.273 KB, Savings: 32.904 MB
2 components: Original Image: 32.959 MB -> Compressed Image: 112.547 KB, Savings: 32.849 MB
5 components: Original Image: 32.959 MB -> Compressed Image: 281.367 KB, Savings: 32.684 MB
10 components: Original Image: 32.959 MB -> Compressed Image: 562.734 KB, Savings: 32.409 MB
20 components: Original Image: 32.959 MB -> Compressed Image: 1.099 MB, Savings: 31.86 MB
40 components: Original Image: 32.959 MB -> Compressed Image: 2.198 MB, Savings: 30.761 MB
80 components: Original Image: 32.959 MB -> Compressed Image: 4.396 MB, Savings: 28.563 MB
160 components: Original Image: 32.959 MB -> Compressed Image: 8.793 MB, Savings: 24.166 MB
256 components: Original Image: 32.959 MB -> Compressed Image: 14.068 MB, Savings: 18.891 MB
```

Q2: Understanding PCA [20 pts] **[P]** | **[W]**

2.1 Implementation [10 pts] **[P]**

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.

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

2.1.1 Local Tests for PCA [No Points]

You may test your implementation of the functions contained in **pca.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

from localtests import TestPCA

unittest_pca = TestPCA()
unittest_pca.test_pca()
unittest_pca.test_transform()
unittest_pca.test_transform_rv()
```

```
UnitTest passed successfully for "PCA fit"!
UnitTest passed successfully for "PCA transform"!
UnitTest passed successfully for "PCA transform with recovered variance"!
```

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 **pca.py**, complete the following function:

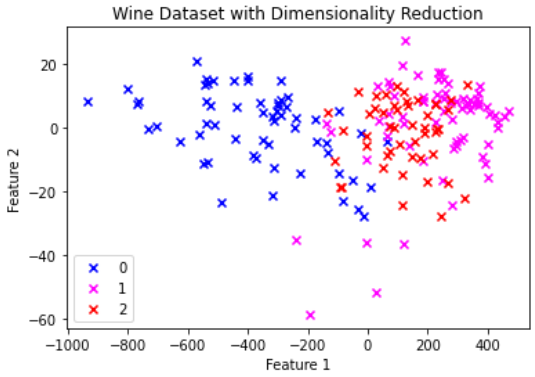
- **visualize**: Use your implementation of PCA and reduce the datasets such that they contain only two features. Using Matplotlib's Pyplot, create 2-D scatter plots of the data points using these features. Make sure to differentiate the data points according to their true labels using color.

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

Wine Dataset

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

# Use PCA for visualization of iris dataset
```

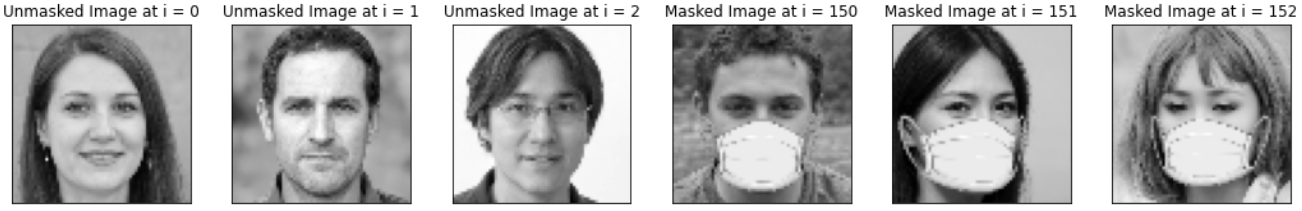
```
from pca import PCA

wine_data = load_wine(return_X_y=True)

X = wine_data[0]
y = wine_data[1]

fig = plt.figure()
plt.title('Wine Dataset with Dimensionality Reduction')
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
PCA().visualize(X,y,fig)
```



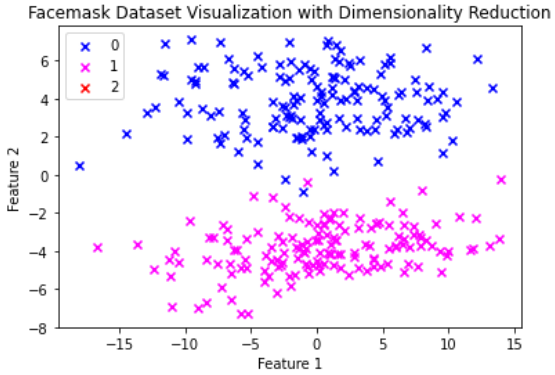
Facemask Dataset

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

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(6, 6, 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 i = {idx}")
    i += 1
```



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

X = np.load('./data/smallflat_64.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.')
```



*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.

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.

The first dataset we will use is an emotion dataset made up of grayscale images of human faces faces that are visibly happy and visibly sad. Note how Accuracy increases after reducing the number of features used.

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

X = np.load('./data/emotion_features.npy')
y = np.load('./data/emotion_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(6, 6, i + 1, xticks=[], yticks=[])
    ax.imshow(X[idx].reshape(64, 64), cmap = 'gray')
    m_status = 'Sad' if idx < 150 else 'Happy'
    ax.set_title(f"{m_status} Image at i = {idx}")
    i += 1
```

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

X = np.load('./data/emotion_features.npy')
y = np.load('./data/emotion_labels.npy').astype('int')

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

pca = PCA()
pca.fit(X)

X_pca = pca.transform_rv(X, retained_variance = 0.99)

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

```
Data shape before PCA  (600, 4096)
Data shape with PCA  (600, 150)
```

In [ ]:
```
##############################
### 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: 0.94444
```

In [ ]:
```
###############################
### 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: 0.95556
```

Now we will explore sklearn's Diabetes dataset using PCA dimensionality reduction and regression. Notice the RMSE score reduction after we apply PCA.

In [ ]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
from sklearn.linear_model import RidgeCV
def apply_regression(X_train, y_train, X_test):
    ridge =  RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1])
    clf = ridge.fit(X_train, y_train)
    y_pred = ridge.predict(X_test)

    return y_pred
```

In [ ]:
```
###############################
### 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 [ ]:
```
###############################
### 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: 53.101
```

In [ ]:
```
###############################
### 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: 53.024
```

2.3 PCA Reduced Facemask Dataset Analysis [5 pts] **[W]**

1. If the facemask 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 ...
   If the facemask dataset that has been reduced to 2 features was fed into a classifier, it would have produced low accuracy because face structure has a complicated topological features and reducing these features to only 2 would result in eliminating many important features associated with the image.

1. 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 behind this lies in the computation efficiency. Given an equal rate of accuracy using 4096 features seems like an overkill and a waste of resource (time/power etc.)

3 Polynomial regression and regularization [80pts: 50pts + 20pts Bonus for Undergrads + 10pts Bonus for All] **[P]** | **[W]**

3.1 Regression and regularization implementations [50pts: 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). Some of the functions are bonus, see the below function list on what is required to be implemented for graduate and undergraduate students. 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 listed functions below. We have provided the Loss function, L, associated with the GD and SGD function for Linear and Ridge Regression

- **rmse**
- **construct_polynomial_feats**
- **predict**
- **linear_fit_closed**: You should use `np.linalg.pinv` in this function
- **linear_fit_GD** (bonus for undergrad, **required for grad**):

$$L_{linear,GD}(\theta) = \frac{1}{2N}\sum_{i=0}^{N}[y_i - \hat{y}_i(\theta)]^2 \qquad y_i = label,\ \hat{y}_i(\theta) = prediction$$

- **linear_fit_SGD** (bonus for undergrad, **required for grad**):

$$L_{linear,SGD}(\theta) = \frac{1}{2}[y_i - \hat{y}_i(\theta)]^2 \qquad y_i = label,\ \hat{y}_i(\theta) = prediction$$

- **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, **required for grad**):

$$L_{ridge,GD}(\theta) = L_{linear,GD}(\theta) + \frac{c_\lambda}{2}\theta^T\theta$$

- **ridge_fit_SGD** (bonus for undergrad, **required for grad**):

$$L_{ridge,SGD}(\theta) = L_{linear,SGD}(\theta) + \frac{c_\lambda}{2N}\theta^T\theta$$

- **ridge_cross_validation**: Use `ridge_fit_closed` for this function

The points for each function is in the **Deliverables and Points Distribution** section.

3.1.1 Local Tests for Helper Regression Functions [No Points]

You may test your implementation of the functions contained in **regression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

from localtests import TestRegression

unittest_reg = TestRegression()
unittest_reg.test_rmse()
unittest_reg.test_construct_polynomial_feats()
unittest_reg.test_predict()
```

```
UnitTest passed successfully for "RMSE"!
UnitTest passed successfully for "Polynomial feature construction"!
UnitTest passed successfully for "Linear regression prediction"!
```

3.1.2 Local Tests for Linear Regression Functions [No Points]

You may test your implementation of the functions contained in **regression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

from localtests import TestRegression

unittest_reg = TestRegression()
unittest_reg.test_linear_fit_closed()
```

```
UnitTest passed successfully for "Closed form linear regression"!
```

3.1.3 Local Tests for Ridge Regression Functions [No Points]

You may test your implementation of the functions contained in **regression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

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

from localtests import TestRegression

unittest_reg = TestRegression()
unittest_reg.test_ridge_fit_closed()
unittest_reg.test_ridge_cross_validation()
```

3.1.4 Local Tests for Gradient Descent and SGD (Bonus for Undergrad Tests) [No Points]

You may test your implementation of the functions contained in **regression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

In [ ]:
```
from localtests import TestRegression

unittest_reg = TestRegression()
unittest_reg.test_linear_fit_GD()
unittest_reg.test_linear_fit_SGD()
unittest_reg.test_ridge_fit_GD()
unittest_reg.test_ridge_fit_SGD()
```

```
UnitTest passed successfully for "Gradient descent linear regression"!
UnitTest passed successfully for "Stochastic gradient descent linear regression"!
UnitTest passed successfully for "Gradient descent ridge regression"!
UnitTest passed successfully for "Stochastic gradient descent ridge regression"!
```

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.

Answer ... A good RMSE value depends on whether the dataset is normalized or not. RMSE is given by

$$RMSE = \sqrt{\frac{\sum (y_{actual} - y_{predict})^2}{N}}$$

Here, when RMSE = 1, we get:

$$\sum_{i=1}^{N} (y_{actual} - y_{predict})^2 = N$$

If the output can only be between 0 and 1; the above expression tells that our model completely misclassifies every instance of data.

Therefore, normalized RMSE = 1 means that the model is predicting completely opposite labels.

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 ± 0.05), 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.

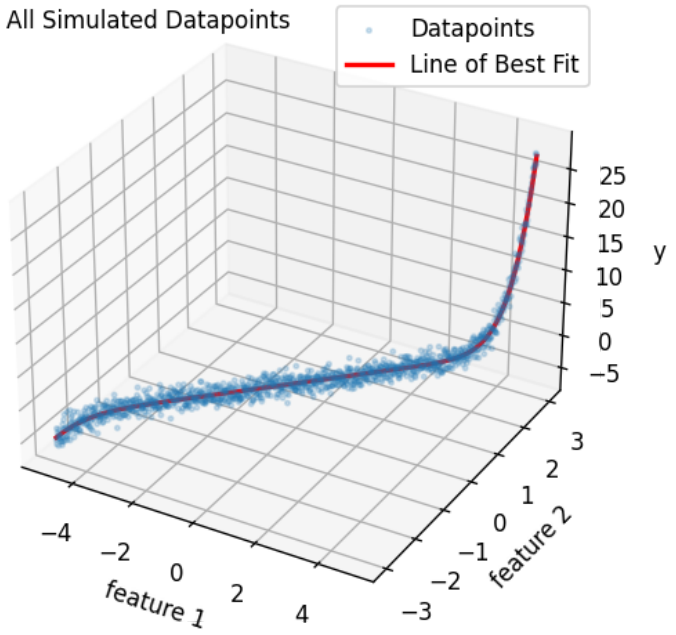
For example, if degree = 2, we will have the polynomial features $[1, a, a^2]$ and $[1, b, b^2]$ for the datapoint $[a, b]$. The cartesian product of these two vectors will be $[1, a, b, ab, a^2, b^2]$. We do not generate a^3 and b^3 since their degree is greater than 2 (specified degree).

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

In [ ]:
```
##############################
### 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)
```

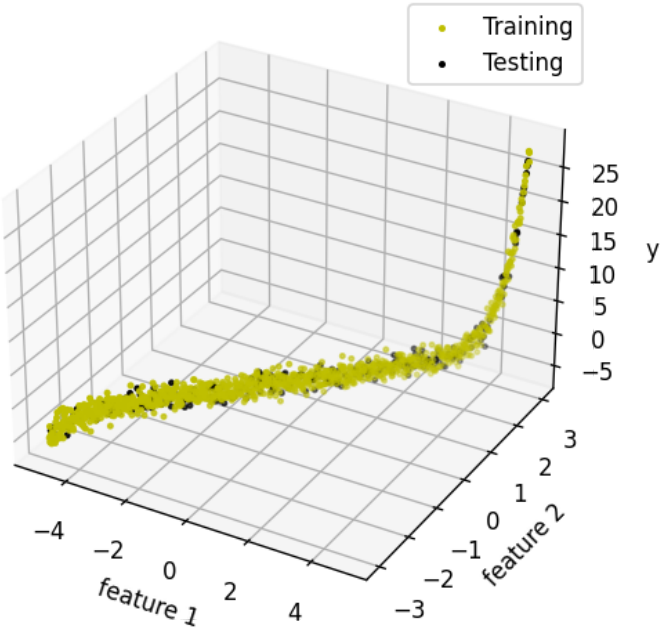
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
fig = plt.figure(figsize=(8,5), dpi=120)
ax = fig.add_subplot(111, projection='3d')

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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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 $Y = X\theta + \epsilon$, where $\epsilon_i \sim N(0,1)$ 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.

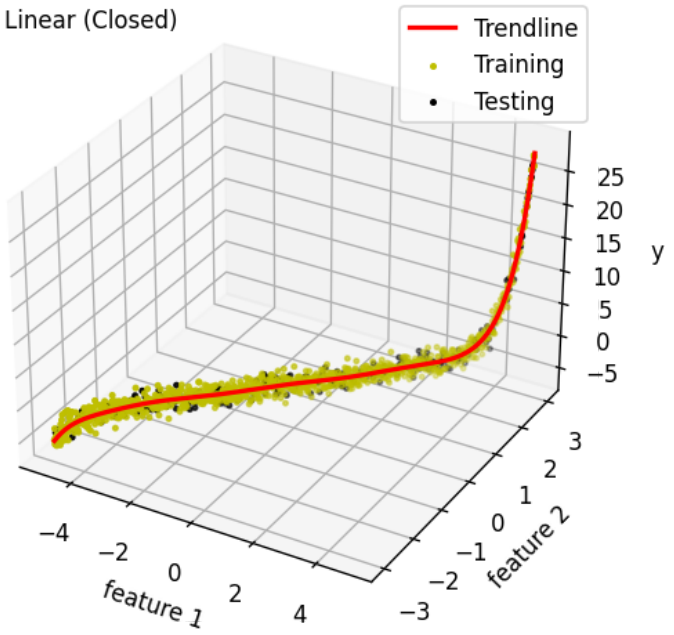
In [ ]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
PERCENT_TRAIN = 0.8

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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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

# Required for both Grad and Undergrad

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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Linear (Closed)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (closed) RMSE: 0.9097
```

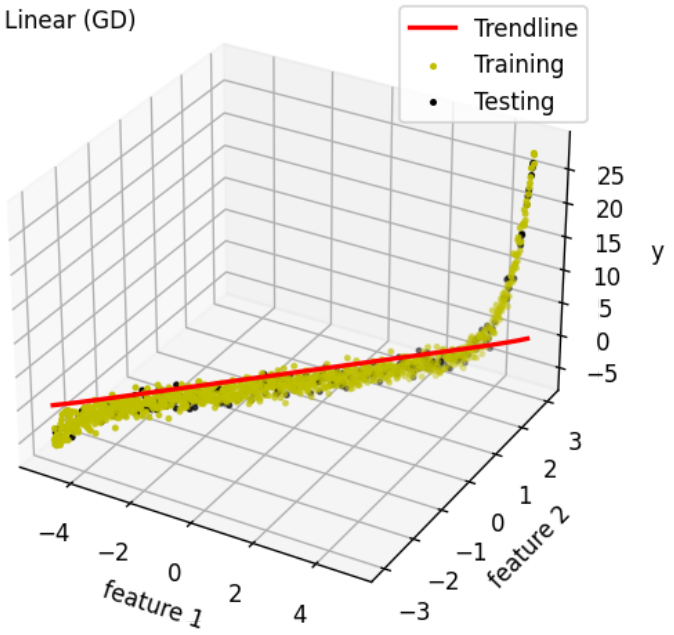


HINT: If your RMSE is off, make sure to follow the instruction given for `linear_fit_closed` in the list of functions to implement above.

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

# Required for Grad Only
# 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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Linear (GD)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (GD) RMSE: 5.1484
```

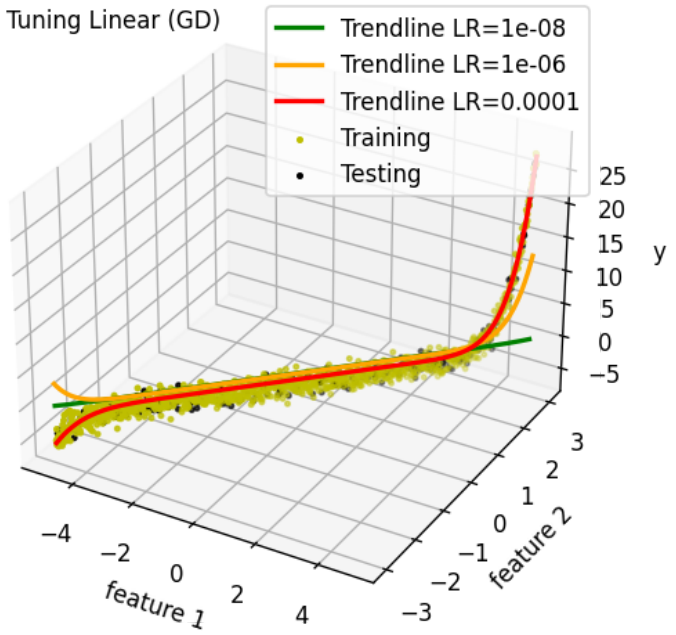


We must tune our epochs and learning_rate. As we tune these parameters our trendline will approach the trendline generated by the linear closed form solution. Observe how we slowly tune (increase) the epochs and learning_rate below to create a better model.

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

# Required for Grad Only
# 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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Tuning Linear (GD)", transform=ax.transAxes)
ax.legend(loc = 'upper right')
plt.show()
```

```
Linear (GD) RMSE: 5.1484 (learning_rate=1e-08)
Linear (GD) RMSE: 3.3447 (learning_rate=1e-06)
Linear (GD) RMSE: 1.1079 (learning_rate=0.0001)
```

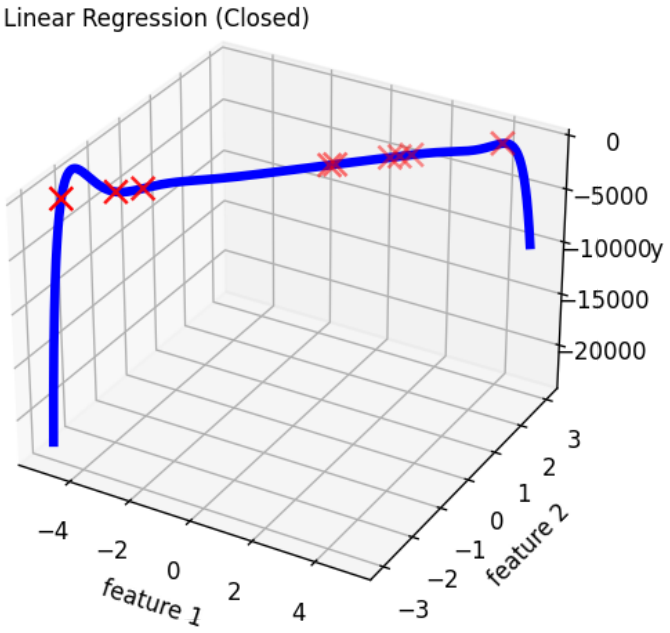


And what if we just use the first 10 data points to train?

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

# Required for both Grad and Undergrad

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])

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Linear Regression (Closed)", transform=ax.transAxes)
```

Out[ ]:
```
Linear (closed) 10 Samples RMSE: 2457.9318
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.

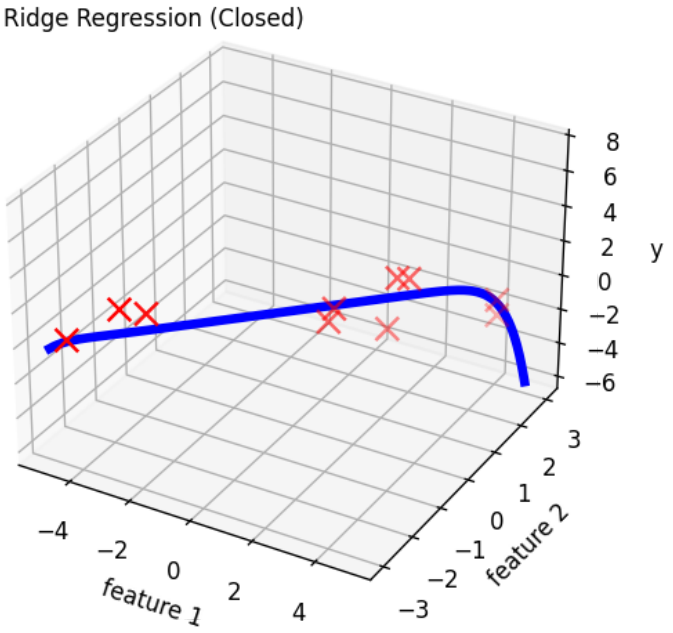
3.4 Testing: Testing ridge regression [5 pts] **[W]**

Now let's try ridge regression. Like before, undergraduate students need to implement the closed form, and graduate students need to implement all 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 ± 0.05), 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 [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

# Required for both Grad and Undergrad

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")

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)
ax.set_zlim([None, 8])
ax.text2D(0.05, 0.95, "Ridge Regression (Closed)", transform=ax.transAxes)
```

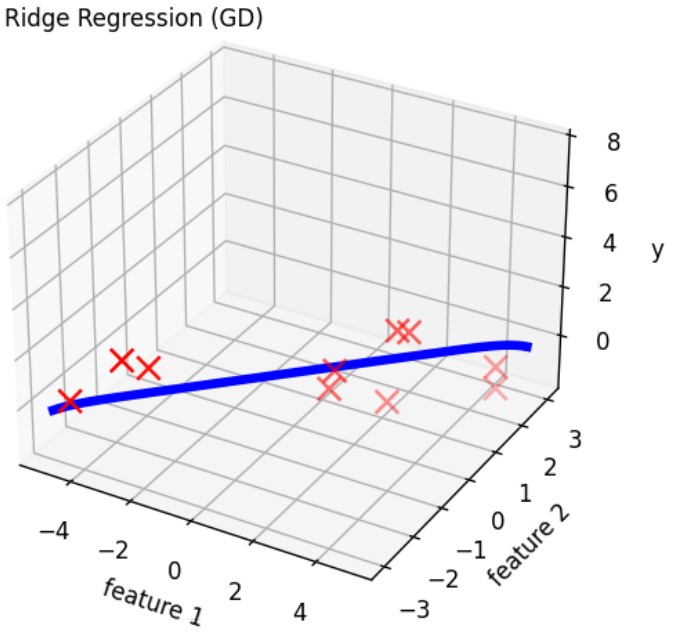
Out[ ]:
```
Ridge Regression (closed) RMSE: 1.4212
Text(0.05, 0.95, 'Ridge Regression (Closed)')
```



HINT: Make sure to follow the instruction given for `ridge_fit_closed` in the list of functions to implement above.

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

# Required for Grad Only

weight, _ = reg.ridge_fit_GD(x_all_feat[sub_train],
                          y_all_noisy[sub_train],
                          c_lambda=20, 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])

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Ridge Regression (GD)", transform=ax.transAxes)
```

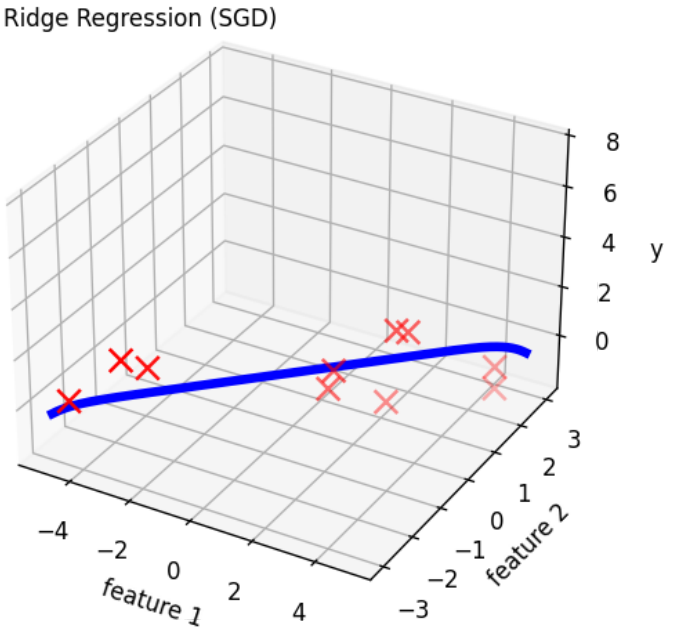
```
Ridge Regression (GD) RMSE: 0.9422
```
Out[ ]:
```
Text(0.05, 0.95, 'Ridge Regression (GD)')
```



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

# Required for Grad Only

weight, _ = reg.ridge_fit_SGD(x_all_feat[sub_train],
                              y_all_noisy[sub_train],
                              c_lambda=20,
                              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])

if not STUDENT_VERSION:
    ax.text2D(0.5, 0.5, EO_TEXT, transform=ax.transAxes,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.4, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

ax.text2D(0.05, 0.95, "Ridge Regression (SGD)", transform=ax.transAxes)
```

```
Ridge Regression (SGD) RMSE: 0.9453
```

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



3.5 Cross validation [7 pts] **[W]**

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

We provided a list of possible values for lambda, and you will use them in cross validation. For cross validation, use 10-fold method and only use it for your training data (you already have the train_indices to get training data). For the training data, split them in 10 folds which means that use 10 percent of training data for test and 90 percent for training At the end for each lambda, you will have calculated 10 rmse values, and get the mean value of that. Then pick the lambda with the lowest mean value of rmse.

Hints:

- np.concatenate is your friend
- Make sure to follow the instruction given for ridge_fit_closed in the list of functions to implement above.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
lambda_list = [0.0001, 0.001, 0.1, 1, 5, 10, 50, 100, 1000, 10000]
kfold = 10

best_lambda, best_error, error_list = reg.hyperparameter_search(x_all_feat[train_indices], y_all[train_indices], lambda_l
for lm, err in zip(lambda_list, error_list):
    print('Lambda: %.4f' % lm, 'RMSE: %.6f'% err)

print('Best Lambda: %.4f' % best_lambda)
weight = reg.ridge_fit_closed(x_all_feat[train_indices], y_all_noisy[train_indices], c_lambda=best_lambda)
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: 1.031590
Lambda: 0.0010 RMSE: 1.031130
Lambda: 0.1000 RMSE: 1.029535
Lambda: 1.0000 RMSE: 1.028789
Lambda: 5.0000 RMSE: 1.029808
Lambda: 10.0000 RMSE: 1.034498
Lambda: 50.0000 RMSE: 1.135883
Lambda: 100.0000 RMSE: 1.307186
Lambda: 1000.0000 RMSE: 2.577907
Lambda: 10000.0000 RMSE: 7.398101
Best Lambda: 1.0000
Best Test RMSE: 0.9451
```

3.6 Noisy Input Samples in Linear Regression [10 pts Bonus for All] **[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

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

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

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

1. **Answer:**

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

If $x'_n = x_n + \epsilon_n$ where $\epsilon_n = \mathcal{N}(0, \sigma^2)$.

$$\begin{aligned} y(x'_n, \theta) &= \theta_0 + \sum_{d=1}^{D} \theta_d x'_{nd} \\ &= \theta_0 + \sum_{d=1}^{D} \theta_d (x_{nd} + \epsilon_{nd}) \\ &= \theta_0 + \sum_{d=1}^{D} \theta_d x_{nd} + \sum_{d=1}^{D} \theta_d \epsilon_{nd} \\ \therefore \quad y(x'_n, \theta) &= y(x_{nd}, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd} \end{aligned}$$

1. **Answer:**

We have; $E_D(\theta) = \frac{1}{2}\sum_{n=1}^{N}\big(y(x_n,\theta) - y_n\big)^2$
Sum of error function of the noise samples is given by;

$$\begin{aligned}
E_D(\theta') &= \frac{1}{2}\sum_{n=1}^{N}\big\{y(x'_n,\theta) - y_n\big\}^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\big\{y(x_n,\theta) - y_n + \sum_{d=1}^{D}\theta_d\epsilon_n d\big\}^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\big\{(y(x_n,\theta)-y_n)^2 + 2(y(x_n,\theta)-y_n)\sum_{d=1}^{D}\theta_d\epsilon_{nd} + (\sum_{d=1}^{D}\theta_d\epsilon_{nd})^2\big\} \\
&= \frac{1}{2}\sum_{n=1}^{N}(y(x_n,\theta)-y_n)^2 + \sum_{n=1}^{N}(y(x_n,\theta)-y_n)\sum_{d=1}^{D}\theta_d\epsilon_{nd} + \frac{1}{2}\sum_{n=1}^{N}(\sum_{d=1}^{D}\theta_d\epsilon_{nd})^2 \\
&= E_D(\theta) + \sum_{n=1}^{N}(y(x_n,\theta)-y_n)\sum_{d=1}^{D}\theta_d\epsilon_{nd} + \frac{1}{2}\sum_{n=1}^{N}(\sum_{d=1}^{D}\theta_d\epsilon_{nd})^2
\end{aligned}$$

Taking expectation on $E_D(\theta')$ we get;

$$\begin{aligned}
\mathbb{E}[E_D(\theta')] &= \mathbb{E}[E_D(\theta)] + \mathbb{E}\Big[\sum_{n=1}^{N}(y(x_n,\theta)-y_n)\sum_{d=1}^{D}\theta_d\epsilon_{nd}\Big] + \frac{1}{2}\mathbb{E}\Big[\sum_{n=1}^{N}(\sum_{d=1}^{D}\theta_d\epsilon_{nd})^2\Big] \\
&= E_D(\theta) + \sum_{n=1}^{N}(y(x_n,\theta)-y_n)\sum_{d=1}^{D}\theta_d\mathbb{E}[\epsilon_{nd}] + \frac{1}{2}\sum_{n=1}^{N}\mathbb{E}\Big[\sum_{d=1}^{D}\sum_{d'=1}^{D}\theta_d\theta_{d'}\epsilon_{nd'}\epsilon_{nd}\Big] \\
&= E_D(\theta) + 0 + \frac{1}{2}\sum_{n=1}^{N}\Big[\sum_{d=1}^{D}\sum_{d'=1}^{D}\theta_d\theta_{d'}\mathbb{E}[\epsilon_{nd'}\epsilon_{nd}]\Big] \\
&= E_D(\theta) + \frac{N}{2}\sum_{d=1}^{D}\theta_d^2\sigma^2 \qquad\qquad \because\ \mathbb{E}[\epsilon_n\epsilon_m] = \sigma^2 \text{ if } m = n \text{ and } 0 \text{ otherwise} \\
&= E_D(\theta) + \frac{N\sigma^2}{2}\sum_{d=1}^{D}\theta_d^2 \\
&= E_D(\theta) + \frac{\lambda}{2}\|\theta\|_2^2 \qquad\qquad \text{where } \lambda = N\sigma^2
\end{aligned}$$

Q4: Naive Bayes and Logistic Regression [35pts] **[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

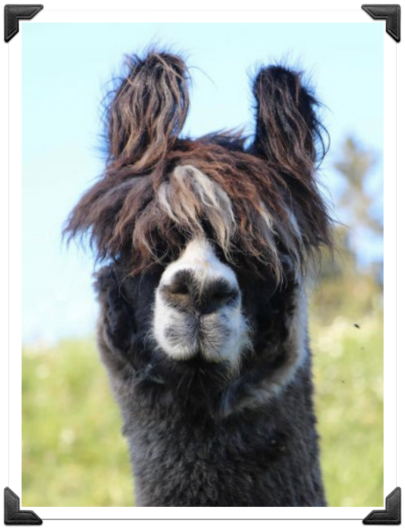
During the testing phase, we try to estimate the probability of a label given an observed feature vector. We combine the probabilities computed from training data to estimate the probability of a given label. For example, if we are trying to decide between two labels y_1 and y_2, then we compute the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

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 vector {1,1,0,1} would have curly hair, a tail over 14 inches, be **less** than 400 pounds, and be extremely shy.

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

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

Now is the time to test your method!

You see a new llama you are interested in that does **not** have curly hair, has a tail over 14 inches, is **less than** 400 pounds, and is **not** shy.

Using Naive Bayes, **is this new llama a Suri or Wooly Llama?**

Answer ...

Curly Hair	Over 14" Tail	400 lb +	Extremely Shy	Label
1	0	0	0	Suri
1	1	0	1	Suri
1	1	1	1	Suri
0	0	0	1	Suri
0	0	1	0	Wooly
1	1	0	0	Wooly

Let y_1 = Suri Llama and y_2 = Wooly Llama.
Here; $\mathcal{P}(y_1) = 2/3$ and $\mathcal{P}(y_2) = 1/3$
According to Bayes's theorem;

$$\mathcal{P}(x_1, x_2, x_3, x_4|y_i) = \frac{\mathcal{P}(x_1|y_i)\mathcal{P}(x_2|y_i)...\mathcal{P}(x_4|y_i)\mathcal{P}(y_i)}{\mathcal{P}(x_1, x_2, x_3, x_4)}$$

We have; $(x_1, x_2, x_3, x_4) = (0, 1, 0, 0)$. Therefore, we can do the following computation;

$$\mathcal{P}(x_1|y_1) = \mathcal{P}(0|y_1) = \frac{1}{4}; \qquad \mathcal{P}(x_1|y_2) = \mathcal{P}(0|y_2) = \frac{1}{2}$$
$$\mathcal{P}(x_2|y_1) = \mathcal{P}(1|y_1) = \frac{1}{2}; \qquad \mathcal{P}(x_2|y_2) = \mathcal{P}(1|y_2) = \frac{1}{2}$$
$$\mathcal{P}(x_3|y_1) = \mathcal{P}(0|y_1) = \frac{3}{4}; \qquad \mathcal{P}(x_3|y_2) = \mathcal{P}(0|y_2) = \frac{1}{2}$$
$$\mathcal{P}(x_4|y_1) = \mathcal{P}(0|y_1) = \frac{1}{4}; \qquad \mathcal{P}(x_4|y_2) = \mathcal{P}(0|y_2) = \frac{1}{1}$$

And,

$$\begin{aligned}\mathcal{P}(x_1, x_2, x_3, x_4) &= \mathcal{P}(x_1)\mathcal{P}(x_2)\mathcal{P}(x_3)\mathcal{P}(x_4) \qquad \because \text{independent features} \\ &= \frac{2}{6} \times \frac{3}{6} \times \frac{4}{6} \times \frac{3}{6} \\ &= \frac{1}{18}\end{aligned}$$

$\mathcal{P}(y_1|0100) = \frac{3}{128} \times \frac{2}{3} \times \frac{18}{1} = \frac{9}{32}$ and $\mathcal{P}(y_2|0100) = \frac{1}{8} \times \frac{1}{3} \times \frac{18}{1} = \frac{3}{4}$

Here; $\mathcal{P}(y_1|0100) < \mathcal{P}(y_2|0100)$. Therefore, the new llama must be a Wooly Llama.

4.2 News Data Sentiment Classification via Logistic Regression [30pts] **[P]**

This dataset contains the sentiments for financial news headlines from the perspective of a retail investor. The sentiment of news has 3 classes, negative, positive and neutral. In this problem, we only use the negative(class label = 0) and positive(class label = 1) classes for binary logistic regression. For data preprocessing, we remove the duplicate headlines and remove the neutral class to get 1967 unique news headlines. Then we randomly split the 1967 headlines into training set and evaluation set with 8:2 ratio. We use the training set to fit a binary logistic regression model.

The code which is provided loads the documents, preprocess the data, builds a "bag of words" representation of each document. Your task is to complete the missing portions of the code in **logisticRegression.py** to determine whether a news headline is negative or positive.

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

- **sigmoid**: transform $s = x\theta$ to probability of being positive using sigmoid function, which is $\frac{1}{1+e^{-s}}$.
- **bias_augment**: augment x with 1's to account for bias term in θ
- **predict_probs**: predicts the probability of positive label $P(y = 1|x)$
- **predict_labels**: predicts labels
- **loss**: calculates binary cross-entropy loss
- **gradient**: calculate the gradient of the loss function with respect to the parameters θ.
- **accuracy**: calculate the accuracy of predictions
- **evaluate**: gives loss and accuracy for a given set of points
- **fit**: fit the logistic regression model on the training data.

Logistic Regression Overview:

1. In logistic regression, we model the conditional probability using parameters θ, which includes a bias term b.

$$p(y_i = 1 \,|\, x_i; \theta) \;=\; h_\theta(x_i) = \sigma(x\theta)$$
$$p(y_i = 0 \,|\, x_i; \theta) \;=\; 1 - h_\theta(x_i)$$

where $\sigma(\cdot)$ is the sigmoid function as follows:

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

1. The conditional probabilities of the positive class $(y = 1)$ and the negative class $(y = 0)$ of the sample x_i attributes are combined into one equation as follows:

$$p(y_i \mid x_i; \theta) \; = \; (h_\theta(x_i))^{y_i} \, (1 - h_\theta(x_i))^{1-y_i}$$

1. Assuming that the samples are independent of each other, the likelihood of the entire dataset is the product of the probabilities of all samples. We use maximum likehood estimation to estimate the model parameters θ. The negative log likelihood (scaled by the dataset size N) is given by:

$$\mathcal{L}(\theta) \; = -\frac{1}{N} \sum_{i=1}^{N} y_i log \, h_\theta(x_i) + \; (1 - y_i) log(1 - h_\theta(x_i))$$

where:

$N =$ number of training samples
$x_i =$ *bag of words* features of the i-th training sample
$y_i =$ label of the i-th training sample

Note that this will be our model's loss function

1. Then calculate the gradient $\nabla_\theta \mathcal{L}$ and use gradient descent to optimize the loss function:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathcal{L}$$

where η is the learning rate and the gradient $\nabla_\theta \mathcal{L}$ is given by:

$$\nabla_\theta \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} x_i^\top \left(h_\theta(x_i) - y_i\right)$$

In [ ]:
```
true = np.array([1,0,1,0,1])
pred = 0.2345

a = [true, pred]

if np.mod(101,100) == 0:
    print ('Yass!')
```

4.2.1 Local Tests for Logistic Regression [No Points]

You may test your implementation of the functions contained in **logistic_regression.py** in the cell below. Feel free to comment out tests for functions that have not been completed yet. See Using the Local Tests for more details.

In [ ]:
```
from localtests import TestLogisticRegression

unittest_lr = TestLogisticRegression()
unittest_lr.test_sigmoid()
unittest_lr.test_bias_augment()
unittest_lr.test_predict_probs()
unittest_lr.test_predict_labels()
unittest_lr.test_loss()
unittest_lr.test_gradient()
unittest_lr.test_accuracy()
unittest_lr.test_evaluate()
unittest_lr.test_fit()
```

```
UnitTest passed successfully for "Logistic Regression sigmoid"!
UnitTest passed successfully for "Logistic Regression bias_augment"!
UnitTest passed successfully for "Logistic Regression predict_probs"!
UnitTest passed successfully for "Logistic Regression predict_labels"!
UnitTest passed successfully for "Logistic Regression loss"!
UnitTest passed successfully for "Logistic Regression gradient"!
UnitTest passed successfully for "Logistic Regression accuracy"!
UnitTest passed successfully for "Logistic Regression evaluate"!
Epoch 0:
        train loss: 0.675       train acc: 0.7
        val loss:   0.675       val acc:   0.7
UnitTest passed successfully for "Logistic Regression fit"!
```

4.2.2 Logistic Regression Model Training [No Points]

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

from logistic_regression import LogisticRegression
```

In [ ]:
```
##############################
```

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

news_data = pd.read_csv("./data/news-data.csv",
                    encoding='cp437', header=None)

class_to_label_mappings = {
    "negative": 0,
    "positive": 1
}

news_data.columns = ["Sentiment", "News"]
news_data.drop_duplicates(inplace=True)

news_data = news_data[news_data.Sentiment != "neutral"]

news_data["Sentiment"] = news_data["Sentiment"].map(
    class_to_label_mappings)


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

X = news_data['News'].values
y = news_data['Sentiment'].values.reshape(-1, 1)

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)
```

Fit the model to the training data Try different learning rates `lr` and number of `epochs` to achieve >80% test accuracy.

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

model = LogisticRegression()
lr = 0.05
epochs = 10000
theta = model.fit(X_train, y_train, X_test, y_test, lr, epochs)
```

```
Epoch 0:
        train loss: 0.69        train acc: 0.7
        val loss:   0.691       val acc:   0.665
Epoch 1000:
        train loss: 0.436       train acc: 0.794
        val loss:   0.532       val acc:   0.701
Epoch 2000:
        train loss: 0.364       train acc: 0.846
        val loss:   0.484       val acc:   0.746
Epoch 3000:
        train loss: 0.318       train acc: 0.873
        val loss:   0.456       val acc:   0.761
Epoch 4000:
        train loss: 0.286       train acc: 0.896
        val loss:   0.438       val acc:   0.772
Epoch 5000:
        train loss: 0.262       train acc: 0.914
        val loss:   0.425       val acc:   0.782
Epoch 6000:
        train loss: 0.242       train acc: 0.926
        val loss:   0.416       val acc:   0.789
Epoch 7000:
        train loss: 0.226       train acc: 0.933
        val loss:   0.409       val acc:   0.797
Epoch 8000:
        train loss: 0.212       train acc: 0.943
        val loss:   0.404       val acc:   0.802
Epoch 9000:
        train loss: 0.2 train acc: 0.95
        val loss:   0.4 val acc:   0.799
```

4.2.3 Logistic Regression Model Evaluation [No Points]

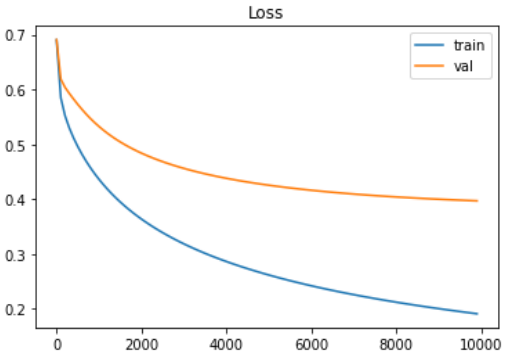
Evaluate the model on the test dataset

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

test_loss, test_acc = model.evaluate(X_test, y_test, theta)
print(f"Test Dataset Accuracy: {round(test_acc, 3)}")
```

```
Test Dataset Accuracy: 0.807
```

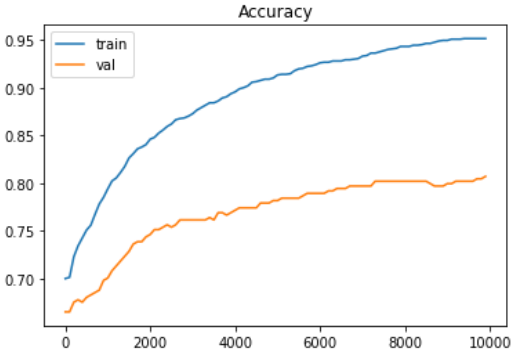
Plotting the loss function on the training data and the test data for every 100th epoch

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

model.plot_loss()
```



Plotting the accuracy function on the training data and the test data for each epoch

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

model.plot_accuracy()
```



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.

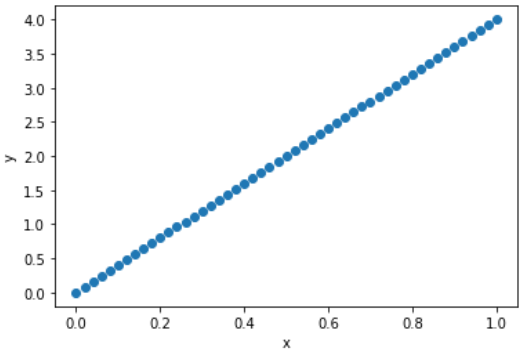
As a reminder, least squares regression minimizes the squared error of the dependent variable from its prediction. Namely, given (x_i, y_i) pairs, least squares returns the line $l(x)$ that minimizes $\sum_i (y_i - l(x_i))^2$.

In [ ]:
```
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
    stack = np.hstack([X,y])
    construct = PCA()
    construct.transform(stack)
    first_component = construct.get_V()[0]
    slope = first_component[1]/first_component[0]

    return slope



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
    reg = Regression()
    weight = reg.linear_fit_closed(X, y)
    return weight[0][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 $X = [0, .02, .04, .06, \ldots, 1]$ and $y = 4x$. Make sure both functions return 4.

In [ ]:
```
################################
### 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")

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT*0.8)

plt.show()
```

```
Slope of first principal component 4.0
Slope of best linear fit 3.9999999999999996
```

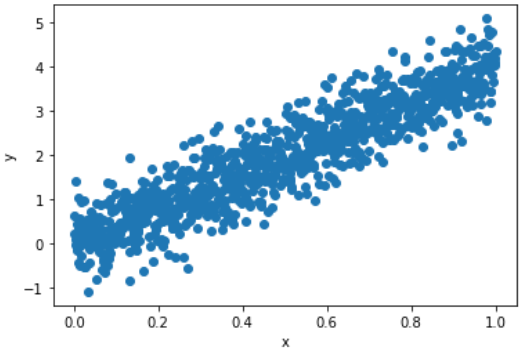


5.2 Analysis Setup [5 pts] **[W]**

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 [ ]:
```
##############################
### 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")

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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. 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 $\hat{X} = [\hat{x}_1, \hat{x}_2, \ldots \hat{x}_{1000}]$

4. Return the **pca_slope** and **lr_slope** values of this X and $\hat{Y}$ dataset you have created where $\hat{Y}$ has noise ($X = X$ or $\hat{X}$ depending on the problem).

Hint 1: Refer to the above example on how to add noise to X or Y

Hint 2: Be careful not to add double noise to X or Y

In [ ]:
```
def addNoise(c, x_noise = False, seed = 1):
```

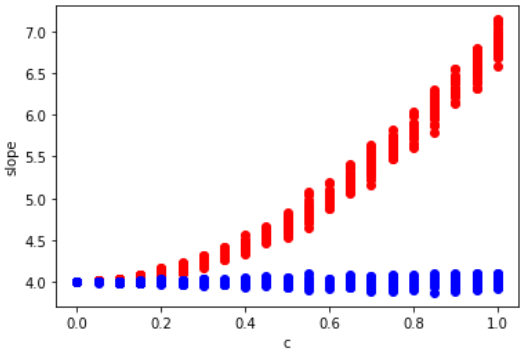
```
    """
    Creates a dataset with noise and calculates the slope of the dataset
    using the pca_slope and lr_slope functions implemented in this class.

    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 #############

    # TODO: Finish the rest of this function
    x = np.linspace(0.001,1,1000).reshape(-1,1)

    if x_noise:
        X = x + np.random.normal(loc = 0.0, scale = c, size = x.shape)
    else:
        X = x

    y_noise = np.random.normal(loc = 0.0, scale = c, size = x.shape)
    y = 4*x + y_noise

    pca_slope_value = pca_slope(X, y)
    lr_slope_value = lr_slope(X, y)

    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 $[0, 0.05, 0.1, \ldots, 0.95, 1.0]$. 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.

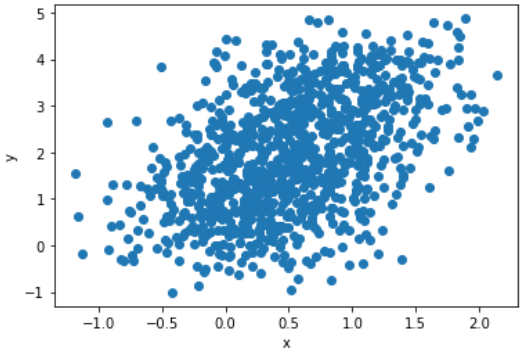
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

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")

if not STUDENT_VERSION:
    fig.text(0.6, 0.4, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.5, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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.

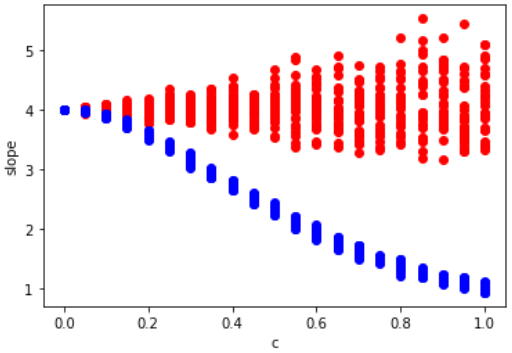
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
base = np.arange(0.001, 1, 0.001).reshape(-1, 1)
c = 0.5
X = base + np.random.normal(loc=[0], scale=c, size=base.shape)
y = 4 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

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

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.8, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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 [ ]:
```
##############################
### 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")

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.5, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

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)

Answer ...

1. Judging from the two figures above it can be seen that PCA performs worse when there is noise only in 'y'. This can be attributed to the fact that PCA assumes full variance of the variables. When only one of the variables is subjected to noise then PCA fails to precisely minimize the distance between these variables.

2. On the other hand, when noise is introduced in 'x' as well; the variances are almost evenly distributed among the two variables and PCA performs reasonably well.

3. Here, linear regression works better if there is noise in 'y' only. It assumes X's are independent variables and y's are depended variables that we're trying to predict using squared error. It is because of this 'implicit bias' in the problem formulation for linear regression, we obtain more accurate result than with PCA.

6 Feature Reduction Implementation [25pts 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

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. 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

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

from feature_reduction import FeatureReduction
```

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
bc_dataset = load_breast_cancer()
bc = pd.DataFrame(bc_dataset.data, columns = bc_dataset.feature_names)
bc['Diagnosis'] = bc_dataset.target
X = bc.drop('Diagnosis', 1)
y = bc['Diagnosis']
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: ['worst concave points', 'worst radius', 'worst texture', 'worst area', 'smoothn
ess error', 'worst symmetry', 'compactness error', 'radius error', 'worst fractal dimension', 'mean compactness', 'mean
concave points', 'worst concavity', 'concavity error', 'area error']
Features selected by backward elimination: ['mean radius', 'mean compactness', 'mean concave points', 'radius error', 's
moothness error', 'concavity error', 'concave points error', 'worst radius', 'worst texture', 'worst area', 'worst conca
vity', 'worst symmetry', 'worst fractal dimension']
```

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)**

Answer ...
We know that Lasso regression involves L_1 regularization of the weights. Therefore, it introduces sparsity in its solution i.e. only the weights that are important are non-zero and majority of the weights are forced to be zero. We use cross validation to select a suitable value for penalty coefficient. On the other hand Ridge regression penalizes the sum of squares of all the weights which is not very helpful in feature selection.

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)**

Answer ...
We know that for a set of N features (predictor variables), there are 2^N possible models. However, due to multicollinearity of null model there are actually 2^{N-1} possible models. Furthermore, the possible features that we can select in the process is:

$$1 + 2 + \ldots . + (N-2) + (N-1) + N = \frac{N(N+1)}{2}$$

Now, fraction explored = $\frac{1}{2^N}$

Not sure if the answer is $\frac{1}{2^N - 1}$ instead or perhaps something different

Q7: Netflix Movie Recommendation Problem Solved using SVD [10pts Bonus for All] **[P]**

Let us try to tackle the famous problem of movie recommendation using just our SVD functions that we have implemented. We are given a table of reviews that 600+ users have provided for close to 10,000 different movies. Our challenge is to predict how much a user would rate a movie that they have not seen (or rated) yet. Once we have these ratings, we would then be able to predict which movies to recommend to that user.

Understanding How SVD Helps in Movie Recommendation

We are given a dataset of user-movie ratings (R) that looks like the following:

UserID/MovieID	1	2	3	4	...	...	8370
1	nan	nan	2	1	...	...	3
2	nan	nan	nan	nan	...	...	nan
3	nan	3	4	nan	...	...	nan
4	1	nan	nan	nan	...	...	5
....	...	...	...	...	...	...	...
....	...	...	...	...	...	...	...
....	...	...	...	...	...	...	...
671	4	nan	nan	nan	...	...	nan

Ratings in the matrix range from 1-5. In addition, the matrix contains `nan` wherever there is no rating provided by the user for the corresponding movie. One simple way to utilize this matrix to predict movie ratings for a given user-movie pair would be to fill in each row / column with the average rating for that row / column. For example: For each movie, if any rating is missing, we could just fill in the average value of all available ratings and expect this to be around the actual / expected rating.

While this may sound like a good approximation, it turns out that by just using SVD we can improve the accuracy of the predicted rating.

How does SVD fit into this picture?

Recall how we previously used SVD to compress images by throwing out less important information. We could apply the same idea to our above matrix (R) to generate another matrix (R_) which will provide the same information, i.e ratings for any user-movie pairs but by combining only the most important features.

Let's look at this with an example:

Assume that decomposition of matrix R looks like:

$$R = U \Sigma V^T$$

We can re-write this decomposition as follows:

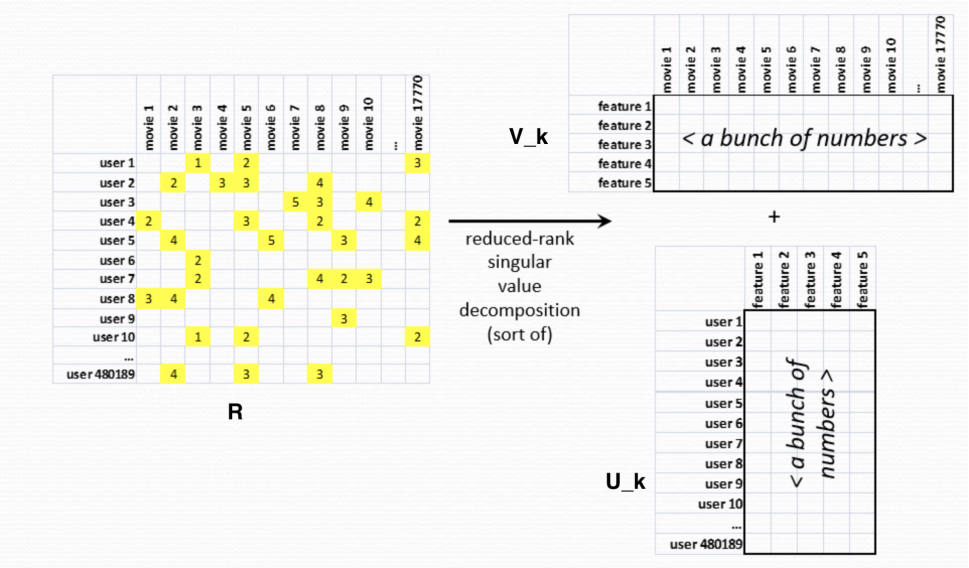
$$R = U\sqrt{\Sigma}\sqrt{\Sigma}V^T$$

If we were to take only the top K singular values from this matrix, we could again write this as:

$$R\_ = U\sqrt{\Sigma}_k\sqrt{\Sigma}_k V^T$$

Thus we have now effectively separated our ratings matrix R into two matrices given by: $U_k = U_{[:k]}\sqrt{\Sigma}_k$ and $V_k = \sqrt{\Sigma}_k V^T_{[:k]}$

There are many ways to visualize the importance of U and V matrices but with respect to our context of movie ratings, we can visualize these matrices as follows:



We can imagine each row of U_k to be holding some information how much each user likes a particular feature (feature1, feature2, feature 3...feature k). On the contrary, we can imagine each column of V_k^T to be holding some information about how much each movie relates to the given features (feature 1, feature 2, feature 3 ... feature k).

Lets denote the row of U_k by u_i and the column of V_k^T by m_j. Then the dot-product: $u_i * m_j$ can provide us with information on how much a user *i* likes movie *j*.

What have we achieved by doing this?

Starting with a matrix R containing very few ratings, we have been able to summarize the sparse matrix of ratings into matrices U_k and V_k which each contain feature vectors about the Users and the Movies. Since these feature vectors are summarized from only the most important K features (by our SVD), we can predict any User-Movie rating that is closer to the actual value than just taking any average rating of a row / column (recall our brute force solution discussed above).

Now this method in practice is still not close to the state-of-the-art but for a naive and simple method we have used, we can still build some powerful visualizations as we will see in part 3.

We have divided the task into 3 parts:

1) Implement `recommender_svd` to return matrices U_k and V_k

2) Implement `predict` to predict top 3 movies a given user would watch

3) *(Ungraded)* Feel free to run the final cell labeled to see some visualizations of the feature vectors you have generated

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

from svd_recommender import SVDRecommender
from regression import Regression
```

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

recommender = SVDRecommender()
recommender.load_movie_data()
regression = Regression()
# Read the data into the respective train and test dataframes
train, test = recommender.load_ratings_datasets()
print("-----------------------------------------------")
```

```
print("Train Dataset Stats:")
print("Shape of train dataset: {}".format(train.shape))
print("Number of unique users (train): {}".format(train['userId'].unique().shape[0]))
print("Number of unique users (train): {}".format(train['movieId'].unique().shape[0]))
print("Sample of Train Dataset:")
print("----------------------------------------")
print(train.head())
print("----------------------------------------")
print("Test Dataset Stats:")
print("Shape of test dataset: {}".format(test.shape))
print("Number of unique users (test): {}".format(test['userId'].unique().shape[0]))
print("Number of unique users (test): {}".format(test['movieId'].unique().shape[0]))
print("Sample of Test Dataset:")
print("----------------------------------------")
print(test.head())
print("----------------------------------------")

# We will first convert our dataframe into a matrix of Ratings: R
# R[i][j] will indicate rating for movie:(j) provided by user:(i)
# users_index, movies_index will store the mapping between array indices and actual userId / movieId
R, users_index, movies_index = recommender.create_ratings_matrix(train)
print("Shape of Ratings Matrix (R): {}".format(R.shape))

# Replacing `nan` with average rating given for the movie by all users
# Additionally, zero-centering the array to perform SVD
mask = np.isnan(R)
masked_array = np.ma.masked_array(R, mask)
r_means = np.array(np.mean(masked_array, axis=0))
R_filled = masked_array.filled(r_means)
R_filled = R_filled - r_means
```

7.1.1 Implement the `recommender_svd` method to use SVD for Recommendation [5pts] **[P]**

In **svd_recommender.py** file, complete the following function:

- **recommender_svd**: Use the above equations to output U_k and V_k. You can utilize the `svd` and `compress` methods from `imgcompression.py` to retrieve your U, Σ and V matrices.

Local Test for `recommender_svd` Function [No Points]

You may test your implementation of the function in the cell below. See Using the Local Tests for more details.

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

from localtests import TestSVDRecommender

unittest_svd_rec = TestSVDRecommender()
unittest_svd_rec.test_recommender_svd()
```

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

# Implement the method `recommender_svd` and run it for the following values of features
no_of_features = [2, 3, 8, 15, 18, 25, 30]
test_errors = []

for k in no_of_features:
    U_k, V_k = recommender.recommender_svd(R_filled, k)
    pred = [] # to store the predicted ratings
    for _, row in test.iterrows():
        user = row['userId']
        movie = row['movieId']
        u_index = users_index[user]
        # If we have a prediction for this movie, use that
        if movie in movies_index:
            m_index = movies_index[movie]
            pred_rating = np.dot(U_k[u_index, :], V_k[:,m_index]) + r_means[m_index]
        # Else, use an average of the users ratings
        else:
            pred_rating = np.mean(np.dot(U_k[u_index], V_k)) + r_means[m_index]
        pred.append(pred_rating)
    test_error = regression.rmse(test['rating'], pred)
    test_errors.append(test_error)
    print("RMSE for k = {} --> {}".format(k, test_error))
```

Plot the Test Error over the different values of `k`

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
fig = plt.figure()
plt.plot(no_of_features, test_errors, 'bo')
plt.plot(no_of_features, test_errors)
plt.xlabel("Value for k")
plt.ylabel("RMSE on Test Dataset")
plt.title("SVD Recommendation Test Error with Different k values")

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.5, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)

plt.show()
```

7.1.2 Implement the `predict` method to find which movie a user is interested in watching next [5pts] **[P]**

Our goal here is to predict movies that a user would be interested in watching next. Since our dataset contains a large list of movies and our model is very naive, filtering among this huge set for top 3 movies can produce results that we may not correlate immediately. Therefore, we'll restrict this prediction to only movies among a subset as given by movies_pool.

Let us consider a user (ID: 660) who has already watched and rated well (>3) on the following movies:

- Iron Man (2008)
- Thor: The Dark World (2013)
- Avengers, The (2012)

The following cell tries to predict which among the movies given by the list below, the user would be most interested in watching next: `movies_pool` :

- Ant-Man (2015)
- Iron Man 2 (2010)
- Avengers: Age of Ultron (2015)
- Thor (2011)
- Captain America: The First Avenger (2011)
- Man of Steel (2013)
- Star Wars: Episode IV - A New Hope (1977)
- Ladybird Ladybird (1994)
- Man of the House (1995)
- Jungle Book, The (1994)

In **svd_recommender.py** file, complete the following function:

- **predict**: Predict the next 3 movies that the user would be most interested in watching among the ones above.

HINT: You can use the method `get_movie_id_by_name` to convert movie names into movie IDs and vice-versa.

NOTE: The user may have already watched and rated some of the movies in `movies_pool`. Remember to filter these out before returning the output. The original Ratings Matrix, R might come in handy here along with np.isnan

Local Test for `predict` Functions [No Points]

You may test your implementation of the function in the cell below. See Using the Local Tests for more details.

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

unittest_svd_rec.test_predict()
```

7.2 Visualize Movie Vectors [No Points]

Our model is still a very naive model, but it can still be used for some powerful analysis such as clustering similar movies together based on user's ratings.

We have said that our matrix V_k that we have generated above contains information about movies. That is, each column in V_k contains (feature 1, feature 2, ... feature k) for each movie. We can also say this in other terms that V_k gives us a feature vector (of length k) for each movie that we can visualize in a k-dimensional space. For example, using this feature vector, we can find out which movies are similar or vary.

While we would love to visualize a k-dimensional space, the constraints of our 2D screen wouldn't really allow us to do so. Instead let us set K=2 and try to plot the feature vectors for just a couple of these movies.

As a fun activity run the following cell to visualize how our model separates the two sets of movies given below.

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

marvel_movies = ['Thor: The Dark World (2013)',
                 'Avengers: Age of Ultron (2015)',
                 'Ant-Man (2015)',
                 'Iron Man 2 (2010)',
                 'Avengers, The (2012)',
                 'Thor (2011)',
                 'Captain America: The First Avenger (2011)']
marvel_labels = ['Blue'] * len(marvel_movies)
star_wars_movies = [
                 'Star Wars: Episode IV - A New Hope (1977)',
                 'Star Wars: Episode V - The Empire Strikes Back (1980)',
                 'Star Wars: Episode VI - Return of the Jedi (1983)',
                 'Star Wars: Episode I - The Phantom Menace (1999)',
                 'Star Wars: Episode II - Attack of the Clones (2002)',
                 'Star Wars: Episode III - Revenge of the Sith (2005)',
]
star_wars_labels = ['Green'] * len(star_wars_movies)


movie_titles = star_wars_movies + marvel_movies
genre_labels = star_wars_labels + marvel_labels

movie_indices = [movies_index[recommender.get_movie_id_by_name(str(x))] for x in movie_titles]

_, V_k = recommender.recommender_svd(R_filled, k=2)
x, y = V_k[0, movie_indices], V_k[1, movie_indices]
fig = plt.figure()
plt.scatter(x, y, c=genre_labels)
for i, movie_name in enumerate(movie_titles):
    plt.annotate(movie_name, (x[i], y[i]))

if not STUDENT_VERSION:
    fig.text(0.5, 0.5, EO_TEXT, transform=fig.transFigure,
        fontsize=EO_SIZE/2, color=EO_COLOR, alpha=EO_ALPHA*0.5, fontname=EO_FONT,
        ha='center', va='center', rotation=EO_ROT)
```