Spring 2024 CS4641/CS7641 Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, March 29th, 11:59 pm EST

- No unapproved extension of the deadline is allowed. Submission past our 48-hour penalized acceptance period will lead to 0 credit.
- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be done individually.
- Plagiarism is a **serious offense**. You are responsible for completing your own work. You are not allowed to copy and paste, or paraphrase, or submit materials created or published by others, as if you created the materials. All materials submitted must be your own.
- All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures. If we observe any (even small) similarities/plagiarisms detected by Gradescope or our TAs, **WE WILL DIRECTLY REPORT ALL CASES TO OSI**, which may, unfortunately, lead to a very harsh outcome. **Consequences can be severe, e.g., academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

Instructions for the assignment

- This assignment consists of both programming and theory questions.
- Unless a theory question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.
- To switch between cell for code and for markdown, see the menu -> Cell -> Cell Type
- You can directly type Latex equations into markdown cells.
- If a question requires a picture, you could use this syntax `<img src="" style="width: 300px;"/>` to include them within your ipython notebook.
- Your write up must be submitted in PDF form. You may use either Latex, markdown, or any word processing software. We will **NOT** accept handwritten work. Make sure

that your work is formatted correctly, for example submit $\sum_{i=0} x_i$ instead of \text{sum_{i=0} x_i}

- When submitting the non-programming part of your assignment, you must correctly map pages of your PDF to each question/subquestion to reflect where they appear. **Improperly mapped questions may not be graded correctly and/or will result in point deductions for the error.**

- All assignments should be done individually, and each student must write up and submit their own answers.

- **Graduate Students**: You are required to complete any sections marked as Bonus for Undergrads

Using the autograder

- Grads will find three assignments on Gradescope that correspond to HW3: "Assignment 3 Programming", "Assignment 3 - Non-programming" and "Assignment 3 Programming - Bonus for all". Undergrads will find an additional assignment called "Assignment 3 Programming - Bonus for Undergrads".

- You will submit your code for the autograder in the Assignment 3 Programming sections. Please refer to the Deliverables and Point Distribution section 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 "Assignment 3 - Non-programming" part, you will need to submit to Gradescope a PDF copy of your Jupyter Notebook with the cell outputs.** Please refer to the **Deliverables and Point Distribution** section for an outline of the non-programming questions.

- **When submitting to Gradescope, please make sure to mark the page(s) corresponding to each problem/sub-problem. 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**. Both can be found under the **utilities** folder.
- 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 Facemask Dataset Analysis** [5 pts] *non-programming*
- **2.4 PCA Exploration** [0 pts]

Q3: Regression and Regularization [80pts: 50pts + 20pts Grad / 6% Bonus for Undergrads + 2.3% Bonus for All]

Deliverables: regression.py and Written portion

- **3.1 Regression and Regularization Implementations** [50pts: 30pts + 20pts Grad / 6% 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 Grad / 1.5% Bonus for Undergrad]
    - Linear Stochastic Gradient Descent [5pts Grad / 1.5% Bonus for Undergrad]
    - Ridge Gradient Descent [5pts Grad / 1.5% Bonus for Undergrad]
    - Ridge Stochastic Gradient Descent [5pts Grad / 1.5% 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** [7 pts] *non-programming*
- **3.5 Cross Validation** [7 pts] *non-programming*
- **3.6 Noisy Input Samples in Linear Regression** [2.3%] *non-programming* **BONUS FOR ALL**

Q4: Naive Bayes and Logistic Regression [35pts]

Deliverables: logistic_regression.py and Written portion

- **4.1 Profile Screening Problem** [7 pts] *non-programming*
  - Profile Screening Problem using Naive Bayes [5pts]
  - AI-Driven Profile Screening [2pts]
- **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 [5.6% Bonus for All]

Deliverables: feature_reduction.py and Written portion

- **6.1 Feature Reduction** [4%] - *programming*
  - forward_selection [2%]
  - backward_elimination [2%]

- **6.2 Feature Selection - Discussion** [1.6%] *non-programming*

Q7: Movie Recommendation with SVD [2.1% Bonus for All]

Deliverables: svd_recommender.py and Written portion

- **7.1 SVD Recommender**
  - recommender_svd [1.05%]
  - predict [1.05%]
- **7.2 Visualize Movie Vectors** [0pts]

Submission Instructions

Submit the following files to their respective assignments on Gradescope for the programming portions:

- **Assignment 3 Programming**
  - imgcompression.py
  - pca.py
  - regression.py
  - logistic_regression.py
- **Assignment 3 Bonus for Undergrad - Programming**
  - regression.py
- **Assignment 3 Bonus for All - Programming**
  - feature_reduction.py
  - svd_recommender.py

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

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 sys
import warnings

import matplotlib
import numpy as np
import pandas as pd
import plotly.io as pio
from matplotlib import pyplot as plt
from sklearn.datasets import load_breast_cancer, load_diabetes, load_iris
from sklearn.feature_extraction import text
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.model_selection import train_test_split

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 = 0
EO_TEXT, EO_FONT, EO_COLOR = (
    "TA VERSION",
    "Chalkduster",
    "gray",
)
EO_ALPHA, EO_SIZE, EO_ROT = 0.7, 90, 40

pio.renderers.default = "jupyterlab"
```

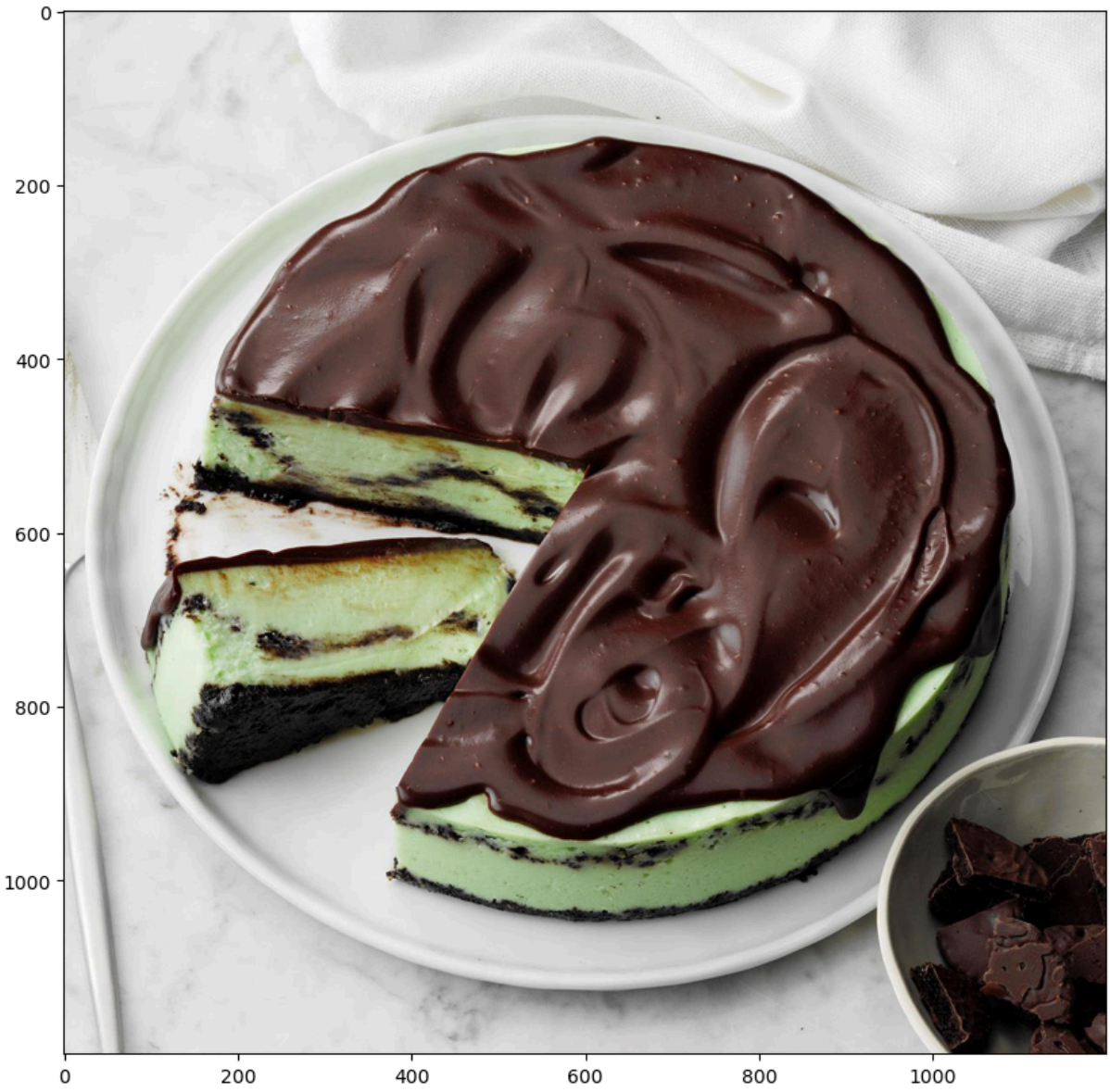
```
Version information
python: 3.11.8 (main, Feb 26 2024, 15:36:12) [Clang 14.0.6 ]
matplotlib: 3.8.0
numpy: 1.26.4
```

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.jpeg") / 255
# plot image
fig = plt.figure(figsize=(10, 10))
plt.imshow(image)
```

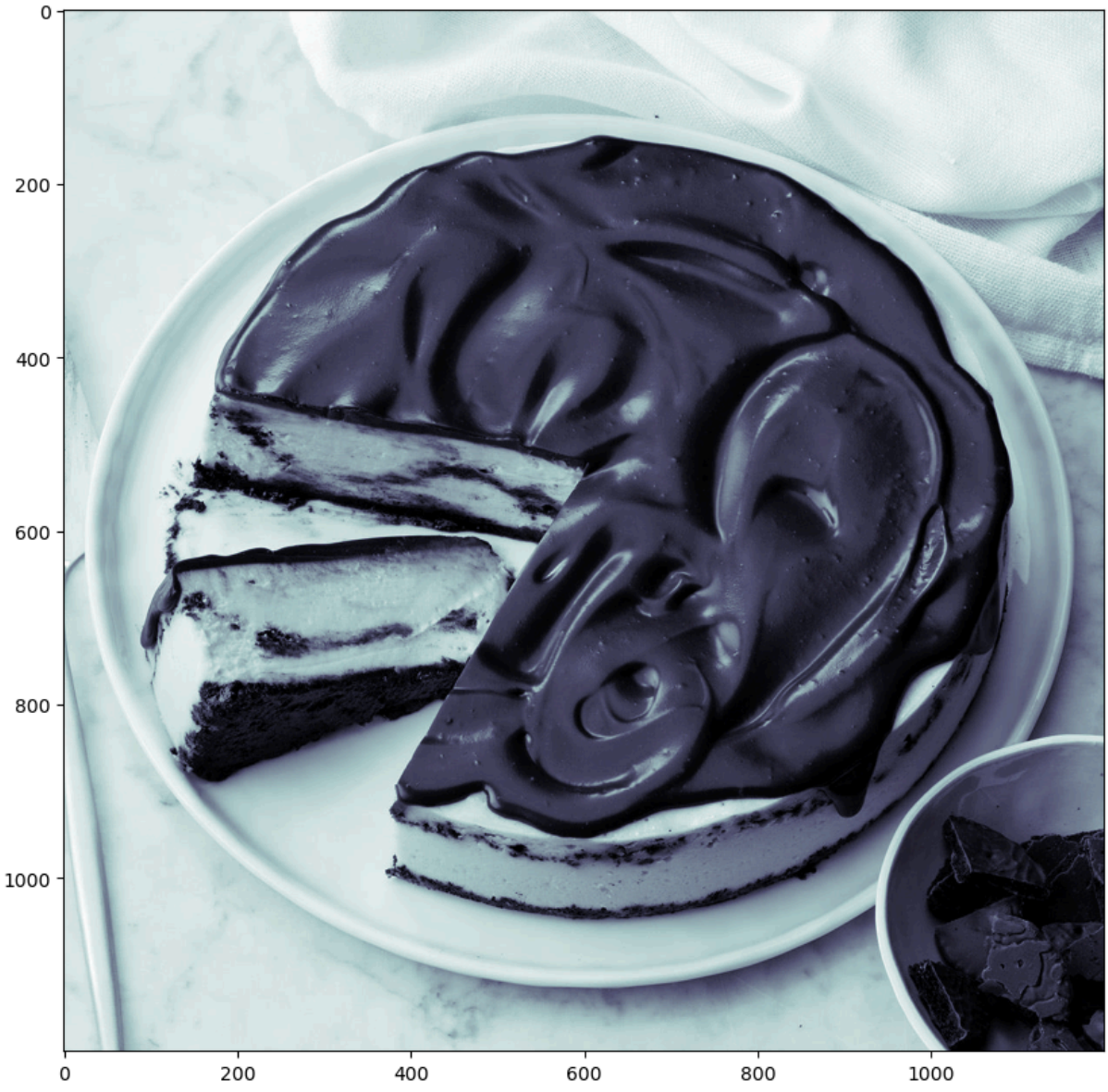
Out[ ]: <matplotlib.figure.Figure at 0x16183ac10>



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))


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

Out[ ]: <matplotlib.image.AxesImage at 0x16c33a910>



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 utilities.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 image
s"!
UnitTest passed successfully for "SVD reconstruction - black and white image
s"!
UnitTest passed successfully for "Compression ratio - black and white image
s"!
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 utilities.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 imag
es"!
```

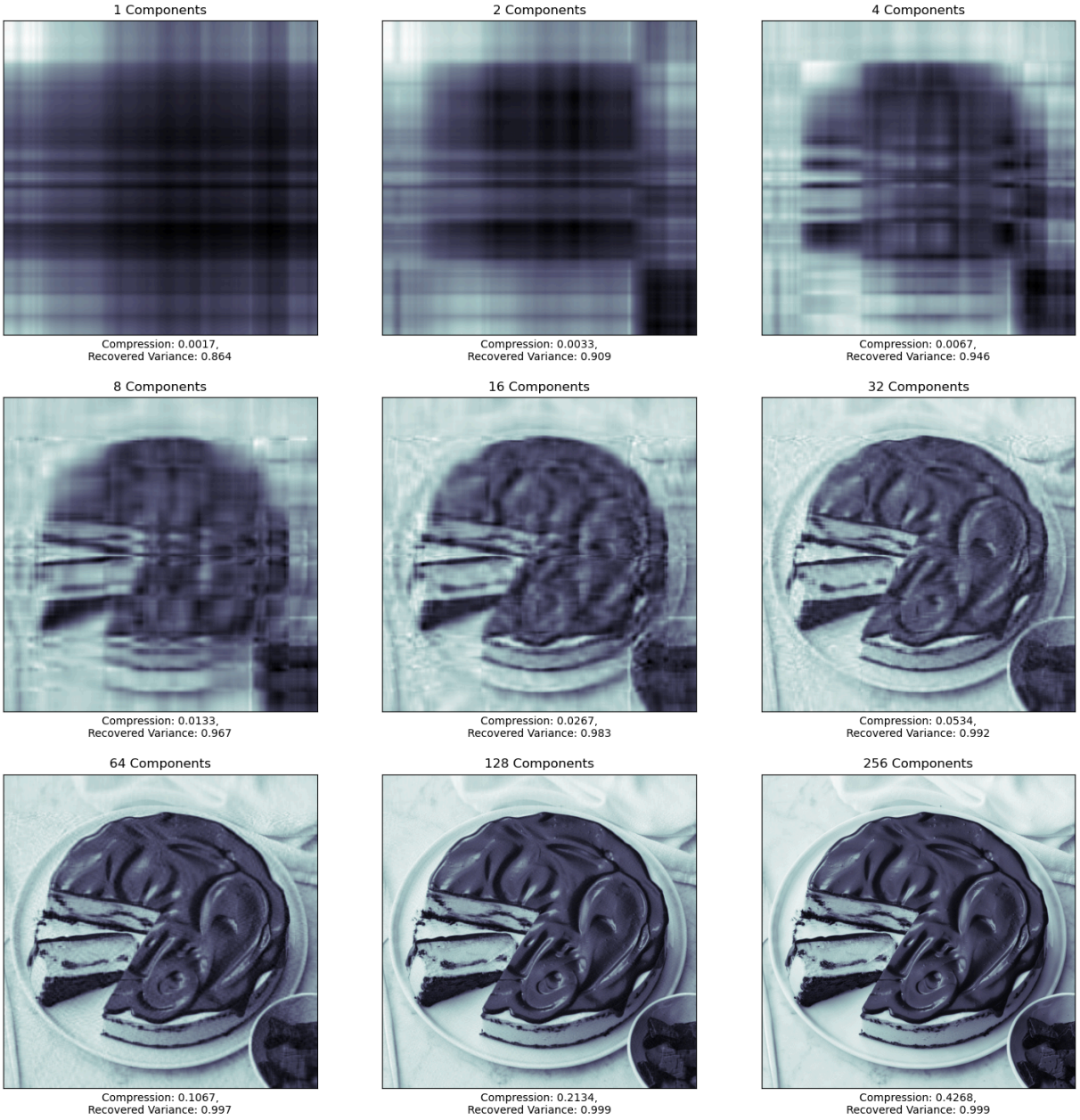
1.2.1 Black and white [5 pts] [W]

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, 4, 8, 16, 32, 64, 128, 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,
    img_rebuild = imcompression.rebuild_svd(U_compressed, S_compressed, V_cc
    c = np.around(imcompression.compression_ratio(bw_image, k), 4)
    r = np.around(imcompression.recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild, cmap=plt.cm.bone)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(f"Compression: {c},\nRecovered Variance: {r}")
    i = i + 1
```



1.2.2 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, 4, 8, 16, 32, 64, 128, 256]

# Compare memory savings for BW image
for k in component_num:
    og_bytes, comp_bytes, savings = imcompression.memory_savings(bw_image, U
    comp_ratio = og_bytes / comp_bytes
    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: {c
    )
```

```
1 components: Original Image: 10.986 MB -> Compressed Image: 18.758 KB, Savi
ngs: 10.968 MB, Compression Ratio 599.8:1
2 components: Original Image: 10.986 MB -> Compressed Image: 37.516 KB, Savi
ngs: 10.95 MB, Compression Ratio 299.9:1
4 components: Original Image: 10.986 MB -> Compressed Image: 75.031 KB, Savi
ngs: 10.913 MB, Compression Ratio 149.9:1
8 components: Original Image: 10.986 MB -> Compressed Image: 150.062 KB, Sav
ings: 10.84 MB, Compression Ratio 75.0:1
16 components: Original Image: 10.986 MB -> Compressed Image: 300.125 KB, Sa
vings: 10.693 MB, Compression Ratio 37.5:1
32 components: Original Image: 10.986 MB -> Compressed Image: 600.25 KB, Sav
ings: 10.4 MB, Compression Ratio 18.7:1
64 components: Original Image: 10.986 MB -> Compressed Image: 1.172 MB, Savi
ngs: 9.814 MB, Compression Ratio 9.4:1
128 components: Original Image: 10.986 MB -> Compressed Image: 2.345 MB, Sav
ings: 8.642 MB, Compression Ratio 4.7:1
256 components: Original Image: 10.986 MB -> Compressed Image: 4.689 MB, Sav
ings: 6.297 MB, Compression Ratio 2.3:1
```

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.

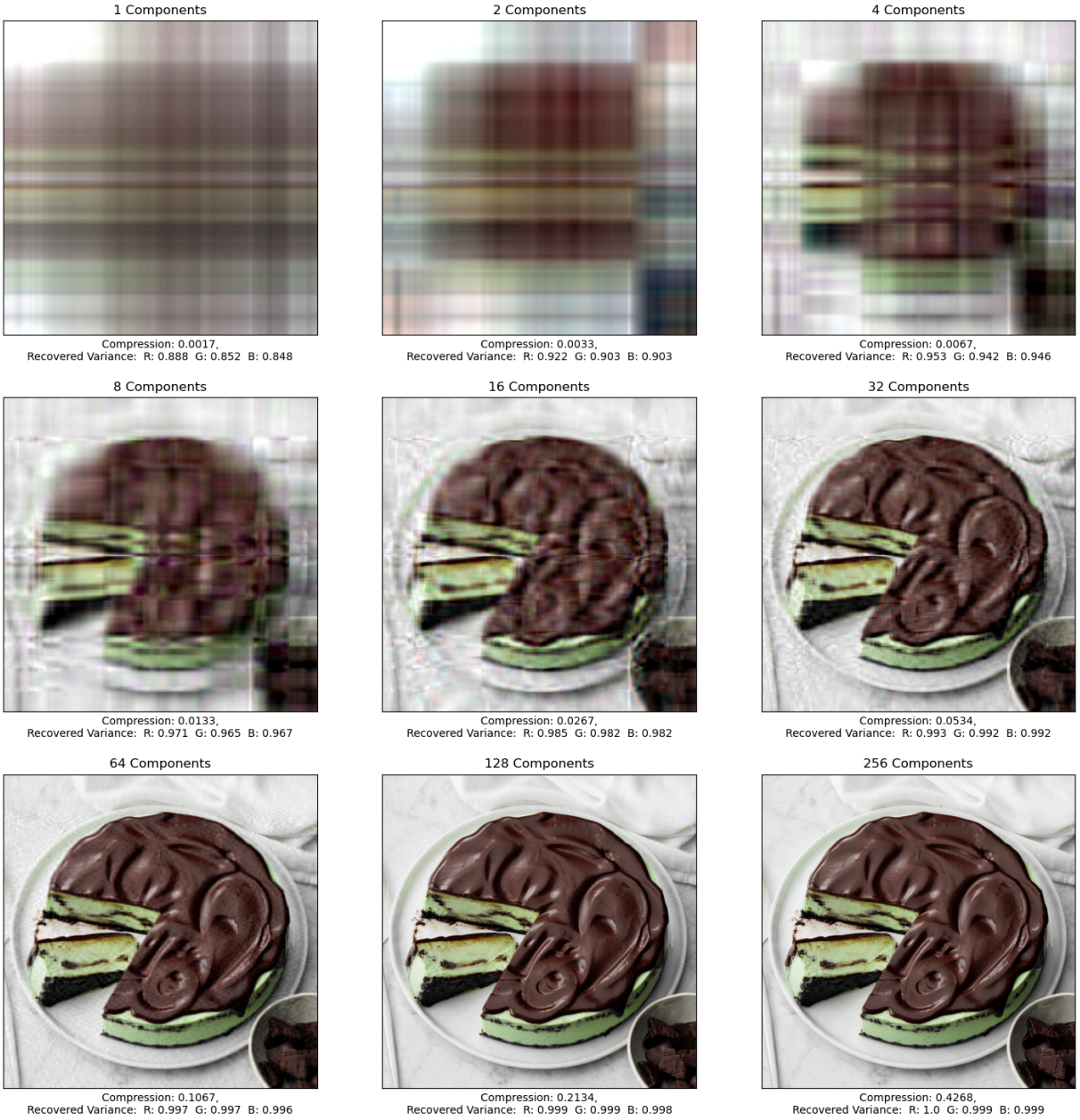
NOTE: You might get warning "Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)." This warning is acceptable since some of the pixels may go above 1.0 while rebuilding. You should see similar images to original even with such clipping.

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()
image_rolled = np.moveaxis(image, -1, 0)
U, S, V = imcompression.svd(image_rolled)

component_num = [1, 2, 4, 8, 16, 32, 64, 128, 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,
    img_rebuild = np.clip(
        imcompression.rebuild_svd(U_compressed, S_compressed, V_compressed),
    )
    img_rebuild = np.moveaxis(img_rebuild, 0, -1)
    c = np.around(imcompression.compression_ratio(image_rolled, k), 4)
    r = np.around(imcompression.recovered_variance_proportion(S, k), 3)
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(img_rebuild)
    ax.set_title(f"{k} Components")
    ax.set_xlabel(
        f"Compression: {np.around(c,4)},\nRecovered Variance:  R: {r[0]}  G:
    )
    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_rolled)

component_num = [1, 2, 4, 8, 16, 32, 64, 128, 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_rolled, U, S, V, k
    )
    comp_ratio = og_bytes / comp_bytes
    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: {c
    )
```

```
1 components: Original Image: 32.959 MB -> Compressed Image: 56.273 KB, Savi
ngs: 32.904 MB, Compression Ratio 599.8:1
2 components: Original Image: 32.959 MB -> Compressed Image: 112.547 KB, Sav
ings: 32.849 MB, Compression Ratio 299.9:1
4 components: Original Image: 32.959 MB -> Compressed Image: 225.094 KB, Sav
ings: 32.739 MB, Compression Ratio 149.9:1
8 components: Original Image: 32.959 MB -> Compressed Image: 450.188 KB, Sav
ings: 32.519 MB, Compression Ratio 75.0:1
16 components: Original Image: 32.959 MB -> Compressed Image: 900.375 KB, Sa
vings: 32.08 MB, Compression Ratio 37.5:1
32 components: Original Image: 32.959 MB -> Compressed Image: 1.759 MB, Savi
ngs: 31.2 MB, Compression Ratio 18.7:1
64 components: Original Image: 32.959 MB -> Compressed Image: 3.517 MB, Savi
ngs: 29.442 MB, Compression Ratio 9.4:1
128 components: Original Image: 32.959 MB -> Compressed Image: 7.034 MB, Sav
ings: 25.925 MB, Compression Ratio 4.7:1
256 components: Original Image: 32.959 MB -> Compressed Image: 14.068 MB, Sa
vings: 18.891 MB, Compression Ratio 2.3:1
```

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

Principal Component Analysis (PCA) is another dimensionality reduction technique that reduces dimensions or features while still preserving the maximum (or close-to) amount of information. This is useful when analyzing large datasets that contain a high number of dimensions or features that may be correlated. PCA aims to eliminate features that are highly correlated and only retain the important/uncorrelated ones that can describe most or all the variance in the data. This enables better interpretability and visualization of the multi-dimensional data. 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.

Here, we will employ Singular Value Decomposition (SVD) for PCA. In PCA, we first center the data by subtracting the mean of each feature. SVD is well suited for this task since each singular value tells us the amount of variance captured in each component for a given matrix (e.g. image). Hence, we can use SVD to extract data only in directions with high variances using either a threshold of the amount of variance or the number of bases/components. Here, we will reduce the data to a set number of components.

Recall from class that in PCA, we project the original matrix X into new components, each one corresponding to an eigenvector of the covariance matrix X^TX. We know that SVD decomposes X into three matrices U, S, and V^T. We can find the SVD decomposition of X^T*X using the decomposition for X as follows:

$$X^TX = (USV^T)^TUSV^T = (VS^TU^T)USV^T = VS^2V^T$$

This means two important things for us:

- The matrix V^T, often referred to as the *right singular vectors* of X, is equivalent to the *eigenvectors* of X^TX.
- S^2 is equivalent to the *eigenvalues* of X^TX.

So the first n-principal components are obtained by projecting X by the first n vectors from V^T. Similarly, S^2 gives a measure of the variance retained.

2.1 Implementation [10 pts] [P]

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 of each feature.

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 utilities.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 Plotly's Express make a 2d and 3d scatterplot of the data points using these features. Make sure to differentiate the data points according to their true labels using color. We recommend converting the data to a pandas dataframe before plotting. In addition, make a 2d scatterplot of the weakest two features following the same procedure and examine the difference.

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

NOTE: Here, we won't be testing for accuracy. Even with correct implementations of PCA, the accuracy can differ from the TA solution. That is fine as long as the visualizations come out similar.

Iris Dataset

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

# Use PCA for visualization of iris dataset

from pca import PCA

iris_data = load_iris(return_X_y=True)

X = iris_data[0]
```

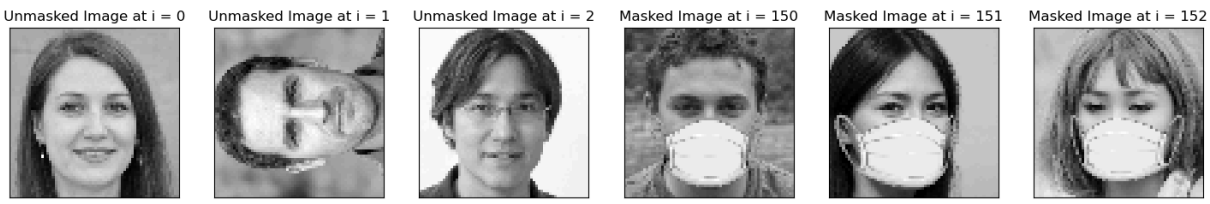
```
y = iris_data[1]

fig_title = "Iris Dataset with Dimensionality Reduction"

pca = PCA()
pca.fit(X)
pca.visualize(X, y, fig_title)
```

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

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=[])
    image = (
        np.rot90(X[idx].reshape(64, 64), k=1)
        if idx % 2 == 1 and idx < 150
        else X[idx].reshape(64, 64)
    )
    m_status = "Unmasked" if idx < 150 else "Masked"
    ax.imshow(image, cmap="gray")
    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_title = "Facemask Dataset Visualization with Dimensionality Reduction"

pca = PCA()
pca.fit(X)
pca.visualize(X, y, fig_title)


print(
    "*In this plot, the 0 points are unmasked images and the 1 points are ma
)
```

```
*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?.

1. Look at the 2-dimensional plot above. 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 in comparison to the original dataset which had 4096 pixels/features? Why? You can refer to the 2D visualization made above (One or two sentences will suffice for this question) **(3 pts)**

   Answer Based on the 2-dimensional plot, a classifier is likely to produce lower accuracy on the dataset reduced to 2 features compared to the original dataset with 4096 pixels/features. The visualization shows a significant overlap between classes, suggesting that the reduced dimensions might not capture enough variance to distinctly separate all classes effectively. In the context of image data, reducing to just 2 dimensions might oversimplify the data, losing critical information necessary for high classification accuracy.

2. What do you think is the main advantage in feeding a classifier a dataset with 2 features vs a dataset with 4096 features and what do you think is a potential disadvantage? (One/Two sentences will suffice for this question.) **(2 pts)**

   Answer The main advantage of feeding a classifier a dataset with 2 features versus a dataset with 4096 features is the significantly reduced computational complexity and faster training times. It also helps in visualizing the data, which is practically impossible with 4096 features. A potential disadvantage is the loss of information, which can lead to a significant drop in model performance since essential details that might be crucial for distinguishing between classes could be lost during the dimensionality reduction process.

2.4 PCA Exploration [No Points]

Note The accuracy can differ from the TA solution and this section is not graded.

Emotion Dataset [No Points]

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=[])
    image = (
        np.rot90(X[idx].reshape(64, 64), k=1)
        if idx % 2 == 1 and idx < 150
        else X[idx].reshape(64, 64)
    )
    m_status = "Unmasked" if idx < 150 else "Masked"
    ax.imshow(image, 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("Not Graded - Data shape before PCA ", X.shape)

pca = PCA()
pca.fit(X)

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

```
print("Not Graded - Data shape with PCA ", X_pca.shape)
```

```
Not Graded - Data shape before PCA  (600, 4096)
Not Graded - 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=0.3, 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(
    "Not Graded - Accuracy before PCA: {:.5f}".format(
        accuracy_score(y_test, preds.argmax(axis=1))
    )
)
```

```
Not Graded - Accuracy before PCA: 0.95000
```

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=0.3, 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(
    "Not Graded - Accuracy after PCA: {:.5f}".format(
        accuracy_score(y_test, preds.argmax(axis=1))
    )
)
```

```
Not Graded - 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("Not Graded - data shape with PCA ", X_pca.shape)
```

```
(442, 10) (442,)
Not Graded - 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=0.3, 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(
    "Not Graded - RMSE score using Ridge Regression before PCA: {:.5}".forma
        rmse_score
    )
)
```

```
Not Graded - 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=0.3, 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(
    "Not Graded - RMSE score using Ridge Regression after PCA: {:.5}".format
)
```

```
Not Graded - RMSE score using Ridge Regression after PCA: 53.024
```

Q3 Polynomial regression and regularization [80pts: 50pts + 20pts Grad / 1.5% Bonus for Undergrads + 2.3% Bonus for All] [P] | [W]

3.1 Regression and regularization implementations [50pts: 30 pts + 20 Grad / 1.5% Bonus for Undergrad] [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 for deriving the gradient update.

- **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**):

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

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

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

- **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**):

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

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

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

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

IMPORTANT NOTE:

- Use your RMSE function to calculate actual loss when coding GD and SGD, but use the loss listed above to derive the gradient update.
- In **ridge_fit_GD** and **ridge_fit_SGD**, you should avoid applying regularization to the bias term in the gradient update.

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 utilities.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 utilities.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 utilities.localtests import TestRegression

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

```
UnitTest passed successfully for "Closed form ridge regression"!
UnitTest passed successfully for "Ridge regression 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 [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from utilities.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 regress
ion"!
UnitTest passed successfully for "Gradient descent ridge regression"!
UnitTest passed successfully for "Stochastic gradient descent ridge regressi
on"!
```

3.2 About RMSE [3 pts] [W]

What is a good RMSE value?

If we normalize our labels such that the true labels y and the model outputs $\hat{y}$ can only be between 0 and 1, what does it mean when the RMSE = 1? Please provide an example with your explanation.

Answer: RMSE measures the standard deviation of the prediction errors or residuals, which represent the differences between predicted values and observed values in the dataset. Therefore, a lower RMSE value indicates better fit as it means the prediction errors are smaller on average. In a normalized setting where both the true labels y and the predicted labels $\hat{y}$ are between 0 and 1, the RMSE value can range from 0 to 1 as well. An RMSE of 0 would indicate perfect predictions with no errors, while an RMSE closer to 1 would suggest larger discrepancies between the predicted and actual values. When RMSE equals 1 in a normalized scenario where labels are between 0 and 1, it suggests that, on average, the model's predictions are maximally distant from the actual values. This is indicative of a model that performs poorly, effectively providing no predictive value above random guessing in the context of the problem domain.

Consider a scenario where we have true labels $y = [0, 1, 0, 1]$ and a model that makes predictions $\hat{y} = [1, 0, 1, 0]$.

- For the first data point, the error is $|0 - 1| = 1$.
- For the second data point, the error is $|1 - 0| = 1$.
- For the third data point, the error is $|0 - 1| = 1$.
- For the fourth data point, the error is $|1 - 0| = 1$.

The squared errors are all $1^2 = 1$. The mean of these squared errors is also 1, and the square root of this mean (the RMSE) is 1.

This example illustrates a situation where the model's predictions are exactly opposite to the actual labels, leading to the maximum possible average error (and thus RMSE) of 1 in a normalized [0, 1] context. In practical terms, this means the model is consistently making the worst possible predictions given the range of the data, demonstrating no predictive accuracy within the defined problem space.

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 $[1, a, a^2, a^3, \ldots, a^{\text{degree}}]$ and $[1, b, b^2, b^3, \ldots, b^{\text{degree}}]$. 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 [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

from plotter import Plotter
from regression import Regression
```

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

# Generate a sample regression dataset with polynomial features
# using the student's regression implementation.

POLY_DEGREE = 5

reg = Regression()
plotter = Plotter(regularization=reg, poly_degree=POLY_DEGREE)

x_all, y_all, p, x_all_feat = plotter.create_data()
```

```
x_all: 700 (rows/samples) 2 (columns/features)
y_all: 700 (rows/samples) 1 (columns/features)
```

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

# Visualize simulated regression data

plotter.plot_all_data(x_all, y_all, p)
```

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 red dots are for testing.

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

xtrain, ytrain, xtest, ytest, train_indices, test_indices = plotter.split_da
```

```
    x_all, y_all
)

plotter.plot_split_data(xtrain, xtest, ytrain, ytest)
```

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_indice
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
y_pred = reg.predict(x_all_feat, weight)
print("Linear (closed) RMSE: %.4f" % test_rmse)

plotter.plot_linear_closed(xtrain, xtest, ytrain, ytest, x_all, y_pred)
```

```
Linear (closed) RMSE: 1.0273
```

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 increased. 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_
)
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)
y_pred = reg.predict(x_all_feat, weight)
y_pred = np.reshape(y_pred, (y_pred.size,))

plotter.plot_linear_gd(xtrain, xtest, ytrain, ytest, x_all, y_pred)
```

```
Linear (GD) RMSE: 3.1861
```

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
    )
    test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
    print("Linear (GD) RMSE: %.4f (learning_rate=%s)" % (test_rmse, learning

plotter.plot_linear_gd_tuninglr(
    xtrain, xtest, ytrain, ytest, x_all, x_all_feat, learning_rates, weights
)
```

```
Linear (GD) RMSE: 3.1861 (learning_rate=1e-08)
Linear (GD) RMSE: 2.2901 (learning_rate=1e-06)
Linear (GD) RMSE: 1.1099 (learning_rate=0.0001)
```

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

Linear Closed 10 Samples

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
rng = np.random.RandomState(seed=3)
y_all_noisy = np.dot(x_all_feat, np.zeros((POLY_DEGREE**2 + 2, 1))) + rng.ra
    x_all_feat.shape[0], 1
)
sub_train = train_indices[10:20]
```

Due to the large RMSE values, rounding errors may result in larger than normal differences from the TA solution. Here, we will accept RMSE values ± 0.1 from the TA solution.

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)

plotter.plot_10_samples(
    x_all, y_all_noisy, sub_train, y_pred, title="Linear Regression (Closed)
)
```

```
Linear (closed) 10 Samples RMSE: 1816.3828
```

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]

3.4.1 [3pts] [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)

plotter.plot_10_samples(
    x_all, y_all_noisy, sub_train, y_pred, title="Ridge Regression (Closed)"
)
```

```
Ridge Regression (closed) RMSE: 1.1193
```

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_rat
)
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)

plotter.plot_10_samples(
    x_all, y_all_noisy, sub_train, y_pred, title="Ridge Regression (GD)"
)
```

```
Ridge Regression (GD) RMSE: 1.0413
```

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_rat
)
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)

plotter.plot_10_samples(
    x_all, y_all_noisy, sub_train, y_pred, title="Ridge Regression (SGD)"
)
```

```
Ridge Regression (SGD) RMSE: 1.0410
```

Linear vs. Ridge Regression

Analyze the difference in performance between the linear and ridge regression methods given the output RMSE **from the testing on 10 samples** and their corresponding approximation plots.

3.4.2 Why does ridge regression achieve a lower RMSE than linear regression on 10 sample points? [1pts] **[W]**

3.4.3 Describe and contrast two scenarios (real life applications): One where linear is more suitable than ridge, and one in which ridge is better choice than linear. Explain why. [1 pts] **[W]**

3.4.4 What is the impact of having some highly correlated features on the data set in terms of linear algebra? Mathematically explain (include expressions) how ridge has an advantage on this in comparison to linear regression. Include the idea of numerical stability. [2pts] **[W]**

Hint: Think about the closed form solution for the weights

3.4.1. **Answer**

Ridge regression often achieves a lower RMSE than linear regression, especially in cases with small sample sizes or when the dataset features are highly correlated. The key difference lies in how ridge regression incorporates regularization, adding a penalty term ($\lambda \sum_{j=1}^{p} \theta_j^2$) to the loss function. This penalty discourages large weights, leading to a more constrained and potentially more generalizable model. In the context of only 10 sample points, linear regression might overfit the data by relying too much on the noise of the small dataset, capturing spurious relationships that don't generalize well. Ridge regression, by introducing regularization, reduces the model's complexity, which can help in achieving lower RMSE by preventing overfitting and making the model more robust to small sample sizes.

3.4.2. **Answer**

- **Linear Regression is More Suitable**: In scenarios where the dataset is large, features are relatively uncorrelated, and there's confidence that the model is not overfitting, linear regression can be more suitable. An example is predicting house prices based on features like square footage, number of bedrooms, and age of the house, assuming these features are uncorrelated. In such cases, linear regression can fully leverage the data without the need for the complexity reduction that comes with ridge regression's regularization.
- **Ridge Regression is a Better Choice**: When dealing with datasets that have a high degree of multicollinearity among features, or when the number of features is close to or exceeds the number of observations, ridge regression is preferable. For example, in genomic data analysis where thousands of features (gene expression levels) are used to predict an outcome, and many of these features can be highly correlated, ridge regression can help in managing overfitting by penalizing large coefficients, leading to a model that generalizes better on unseen data.

3.4.3. **Answer**

In the context of linear algebra, having highly correlated features in a dataset leads to multicollinearity, which makes the feature matrix $X^T X$ close to singular or poorly conditioned. This, in turn, affects the stability and reliability of the linear regression model by making the inversion of $X^T X$ in the closed-form solution ($\theta = (X^T X)^{-1} X^T y$) numerically unstable.

3.5 Cross validation [7 pts] [W]

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

Imagine we have a dataset of 10 points [1,2,3,4,5,6,7,8,9,10] and we want to do 5-fold cross validation.

- The first iteration we would train with [3,4,5,6,7,8,9,10] and test (validate) with [1,2]
- The second iteration we would train with [1,2,5,6,7,8,9,10] and test (validate) with [3,4]
- The third iteration we would train with [1,2,3,4,7,8,9,10] and test (validate) with [5,6]
- The fourth iteration we would train with [1,2,3,4,5,6,9,10] and test (validate) with [7,8]
- The fifth iteration we would train with [1,2,3,4,5,6,7,8] and test (validate) with [9,10]

We provided a list of possible values for λ, and you will complete the `ridge_cross_validation` method to perform 5-fold cross-validation on the training data (we already use `train_indices` to get training data in the cell below). Split the training data into 5 folds, where 20 percent of the data will be used to test and 80 percent will be used to train. For each λ, you will have calculated 5 RMSE values. We provide a function `hyperparameter_search` that takes the average of the RMSE values for each λ and picks the λ with the lowest mean RMSE. (Please look at hints for more information),

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.
- To use the 5-fold method, loop over all the data 5 times, where we split a different 20% of the data at every iteration. The first iteration extracts the first 20% for testing and the remaining 80% for training. The second iteration splits the second 20% of data for testing and the (different) remaining 80% for testing. If we have the array of elements 1 - 10, the second iteration would extract the numbers "3" and "4" because that's in the second 20% of the array.
- The `hyperparameter_search` function will handle averaging the errors, so don't average the errors in `ridge_cross_validation`. We've done this so you can see your error across every fold when using the gradescope tests.

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

best_lambda, best_error, error_list = reg.hyperparameter_search(
    x_all_feat[train_indices], y_all[train_indices], lambda_list, kfold
```

```
)
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_lam
)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print("Best Test RMSE: %.4f" % test_rmse)
```

```
Lambda: 0.0001 RMSE: 0.986072
Lambda: 0.0010 RMSE: 0.987209
Lambda: 0.1000 RMSE: 0.989441
Lambda: 1.0000 RMSE: 0.987945
Lambda: 5.0000 RMSE: 0.986684
Lambda: 10.0000 RMSE: 0.986821
Lambda: 50.0000 RMSE: 0.989110
Lambda: 100.0000 RMSE: 0.994419
Lambda: 1000.0000 RMSE: 1.289583
Lambda: 10000.0000 RMSE: 2.544557
Best Lambda: 0.0001
Best Test RMSE: 1.0528
```

3.6 Noisy Input Samples in Linear Regression [2.3% 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.

Write your responses below using LaTeX in Markdown.

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:**

Given the linear model:

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

And the noisy samples (x_n' = x_n + \epsilon_n), the output with noise can be expressed as:

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

Expanding this gives:

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

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

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

2. **Answer:**

Given the sum-of-squares error function for the noise-free sample set:

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

And the error function for the noisy sample set:

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

Substituting $y(x'_n, \theta) = y(x_n, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}$ into $E_D(\theta)'$:

$$E_D(\theta)' = \frac{1}{2} \sum_{n=1}^{N} \left[(y(x_n, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}) - y_n \right]^2$$

Expanding and taking the expectation:

$$E[E_D(\theta)'] = \frac{1}{2} \sum_{n=1}^{N} E \left[(y(x_n, \theta) - y_n)^2 + 2(y(x_n, \theta) - y_n) \left(\sum_{d=1}^{D} \theta_d \epsilon_{nd} \right) + \left(\sum_{d=1}^{D} \theta_d \epsilon_{nd} \right.$$

Given $E[\epsilon_{nd}] = 0$ and $E[\epsilon_{nd}^2] = \sigma^2$, simplify the expectation:

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

This demonstrates that the expected error with noisy inputs is equivalent to the original sum-of-squares error plus a regularization term:

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

Where the Regularizer term, omitting the bias θ_0, is:

$$\text{Regularizer} = \frac{1}{2}\sigma^2 \sum_{d=1}^{D} \theta_d^2$$

This aligns with the ridge regression formulation, where the regularization term helps control the magnitude of the weights, thereby reducing overfitting and improving model generalization, particularly in the presence of noise and multicollinearity.

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 $x = [x_1, \ldots, x_d]$, which we can write as $P(y \mid x_1, \ldots, x_d)$. Bayes's theorem tells us how to express this in terms of quantities we can compute more directly:

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

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

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

Therefore, we can rewrite Bayes rule as

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

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,

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

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:

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

All we need now is to compute $P(x_1|y_i), \ldots, P(x_d \mid y_i)$ 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 Profile Screening [7pts] [W]

4.1.1 Profile Screening using Naive Bayes [5pts][W]

In the rapidly evolving job market of Techlanta, a leading tech company has devised an automated resume screening system to assist in the hiring process for three distinct roles: Machine Learning Engineer, Data Analyst, and Product Manager. This system is designed to evaluate applicants based on five key binary attributes extracted from their resumes: {coding proficiency (1 for high, 0 for low), data analysis skills (1 for strong, 0 for weak), leadership experience (1 for yes, 0 for no), product design experience (1 for yes, 0 for no), marketing skills (1 for strong, 0 for weak)}.

Aiming to ensure that the screening process is both fair and effective, the company is mindful of avoiding biases that could disadvantage any applicant. They have compiled a dataset of 12 anonymized resumes, with each position having 4 applicants, alongside feedback from previous interviews to serve as the ground truth.

Machine Learning Engineer applicants have demonstrated attributes such as: {1, 1, 0, 0, 1}, {1, 1, 1, 0, 0}, {1, 0, 1, 0, 0}, {0, 1, 0, 1, 0}

Data Analyst applicants are characterized by: {0, 1, 0, 0, 0}, {1, 0, 0, 0, 1}, {0, 1, 1, 0, 1}, {0, 1, 0, 1, 0}

Product Manager applicants display: {0, 0, 1, 1, 0}, {0, 0, 1, 0, 1}, {1, 0, 1, 1, 1}, {0, 1, 0, 1, 1}.

A new applicant's resume has been screened, and identified to have **no** coding experience but **have** data analysis skills **with** product design experience. However, **does not have** leadership or marketing skills.

Now is the time to test your method!

Using a multiclass Naive Bayes classifier, determine the most suitable job position for the new applicant.

NOTE: We expect students to show their work (prior probabilities, conditional probabilites, posterior probabilites) and not just the final answer.

4.1. **Answer:**

$P(\text{Machine Learning Engineer}) = P(\text{Data Analyst}) = P(\text{Product Manager}) = \frac{4}{12} =$

For "coding proficiency" = 0:

- $P(\text{coding proficiency} = 0|\text{MLE}) = \frac{1}{4}$
- $P(\text{coding proficiency} = 0|\text{DA}) = \frac{3}{4}$
- $P(\text{coding proficiency} = 0|\text{PM}) = \frac{3}{4}$

For "data analysis skills" = 1:

- $P(\text{data analysis skills} = 1|\text{MLE}) = \frac{2}{4}$
- $P(\text{data analysis skills} = 1|\text{DA}) = \frac{3}{4}$
- $P(\text{data analysis skills} = 1|\text{PM}) = \frac{1}{4}$

For "leadership experience" = 0:

- $P(\text{leadership experience} = 0|\text{MLE}) = \frac{2}{4}$
- $P(\text{leadership experience} = 0|\text{DA}) = \frac{3}{4}$
- $P(\text{leadership experience} = 0|\text{PM}) = \frac{1}{4}$

For "product design experience" = 1:

- $P(\text{product design experience} = 1|\text{MLE}) = \frac{1}{4}$
- $P(\text{product design experience} = 1|\text{DA}) = \frac{1}{4}$
- $P(\text{product design experience} = 1|\text{PM}) = \frac{3}{4}$

For "marketing skills" = 0:

- $P(\text{marketing skills} = 0|\text{MLE}) = \frac{3}{4}$
- $P(\text{marketing skills} = 0|\text{DA}) = \frac{3}{4}$
- $P(\text{marketing skills} = 0|\text{PM}) = \frac{1}{4}$

Since $P(x_1, \ldots, x_d)$ is constant across all roles, it can be ignored for the purpose of comparison.

For Machine Learning Engineer: *[Math Processing Error]*

For Data Analyst:

```
P(\text{DA} | \text{attributes}) \propto P(\text{coding proficiency} = 0|\text{DA}) \times
1|\text{DA}) \times P(\text{leadership experience} = 0|\text{DA}) \times P(\text{product
```

For Product Manager:

```
P(\text{PM} | \text{attributes}) \propto P(\text{coding proficiency} = 0|\text{PM}) \times
1|\text{PM}) \times P(\text{leadership experience} = 0|\text{PM}) \times P(\text{product
```

Let's calculate these unnormalized posterior probabilities:

MLE:

$$P(\text{MLE}|\text{attributes}) \propto \frac{1}{4} \times \frac{2}{4} \times \frac{2}{4} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{3}$$

DA:

$$P(\text{DA}|\text{attributes}) \propto \frac{3}{4} \times \frac{3}{4} \times \frac{3}{4} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{3}$$

PM:

$$P(\text{PM}|\text{attributes}) \propto \frac{3}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{4} \times \frac{1}{3}$$

Calculating these products:

- **MLE:** $\propto \frac{1}{4} \times \frac{1}{2} \times \frac{1}{2} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{3} = \frac{3}{256}$
- **DA:** $\propto \frac{3}{4} \times \frac{3}{4} \times \frac{3}{4} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{3} = \frac{81}{1024}$
- **PM:** $\propto \frac{3}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{3}{4} \times \frac{1}{4} \times \frac{1}{3} = \frac{9}{1024}$

Comparing these proportions, **Data Analyst** has the highest unnormalized posterior probability. Therefore, based on the attributes provided, the new applicant is most suitable for the **Data Analyst** position.

4.1.2 AI-Driven Profile Screening[2pts] [W]

Traditionally, human HR personnel review resumes to identify candidates who meet the minimum qualifications for a job. This process is subjective and can be influenced by conscious or unconscious biases. More recently, many organizations use Applicant Tracking Systems (ATS) to manage and screen resumes. ATS systems parse resumes to extract information like education, experience, skills, and other relevant details. They rank candidates based on how well their resume matches the job description and criteria set by the employer.

More advanced systems incorporate AI to not only parse and match resumes but also to predict a candidate's job performance, cultural fit, and even retention likelihood. These systems can use machine learning models trained on historical hiring data, and they often employ a combination of keyword matching, machine learning models (such as decision trees, support vector machines, and neural networks), and natural language processing (NLP) techniques.

In recent years, several high-profile cases have highlighted the challenges of bias in AI-driven resume screening processes. For example:

- In 2018, Amazon had to abandon its AI recruitment tool because it was trained on a decade's worth of resumes, predominantly from men, leading to a bias against women's resumes, such as penalizing resumes that mentioned "women's" clubs or activities.[1]
- UnitedHealth Group faced allegations of racial bias in their AI-driven hiring tool. The system reportedly favored white applicants over Black applicants. The company discontinued the tool after these concerns were raised.[2].
- HireVue's AI tool analyzes video interviews, and has been used by more than a 100 companies on over a million applicants, according to the Washington Post[3]. Critics worry that the algorithm is trained on limited data and so will be more likely to mark "traditional" applicants (white, male) as more employable.[4] As a result, applicants who deviate from the "traditional"—including people don't speak English as a native language or who are disabled—are likely to get lower scores, experts say.[5]
- LinkedIn's job-matching AI was found to exhibit gender biases in job recommendations, a consequence of the training data's inherent patterns. The algorithms ranked candidates based on their likelihood to apply for a position or respond to a recruiter. As a result, more men were referred for open roles due to their proactive approach in seeking new opportunities.[6]

Sources:

1: Guardian. 2: The Wall street Journal. 3: Washington Post. 4: MIT Technology Review. 5: Brookings. 6: MIT Technology Review.

Given the context above, which of the following approaches is most effective for detecting bias in AI-driven resume screening systems? (Choose all that apply.)

A. Increasing the size of the training dataset

B. Conducting regular audits of the AI system's decisions

C. Utilizing differential privacy techniques

D. Relying solely on keyword matching to ensure objectivity

4.1.2 **Answer:**

B. Conducting regular audits of the AI system's decisions

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.

$$p(y_i = 1 \mid x_i; \theta) \; = \; h_\theta(x_i) = \sigma(x\theta)$$

$$p(y_i = 0 \mid x_i; \theta) \; = \; 1 - h_\theta(x_i)$$

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

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

2. 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:

$$p(y * i \mid x_i; \theta) \; = \; (h * \theta(x * i))^{y_i} \, (1 - h * \theta(x_i))^{1-y_i}$$

3. 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:

$$\mathcal{L}(\theta \mid X, Y) = -\frac{1}{N} \sum_{i=1}^{N} y\_i \log h_\theta(x * i) + (1 - y\_i) \log(1 - h * \theta(x\_i))$$

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

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

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathcal{L}(\theta_t \mid X, Y)$$

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

$$\nabla_\theta \mathcal{L}(\theta \mid X, Y) = \frac{1}{N} \sum_{i=1}^{N} x_i^\top (h_\theta(x_i) - y_i)$$

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

from utilities.localtests import TestLogisticRegression

unittest_lr = TestLogisticRegression()
unittest_lr.test_sigmoid()
unittest_lr.test_bias_augment()
unittest_lr.test_loss()
unittest_lr.test_predict_probs()
unittest_lr.test_predict_labels()
unittest_lr.test_loss()
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 loss"!
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 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 as LogReg
```

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

news_data = pd.read_csv("./data/news-data.csv", encoding="cp437", header=Nor

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

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

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)

vectorizer = text.CountVectorizer(stop_words="english")

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

RANDOM_SEED = 5
BOW = vectorizer.fit_transform(X).toarray()
indices = np.arange(len(news_data))
X_train, X_test, y_train, y_test, indices_train, indices_test = train_test_s
    BOW, y, indices, 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 = LogReg()
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()
```

Check out sample evaluations from the test set.

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

num_samples = 10
for i in range(10):
    rand_index = np.random.randint(0, len(X_test))
    x_test = np.reshape(X_test[rand_index], (1, X_test.shape[1]))
    prob = model.predict_probs(model.bias_augment(x_test), theta)
    pred = model.predict_labels(prob)
    print(f"Input News: {X[indices_test[rand_index]]}\n")
    print(f"Predicted Sentiment: {label_to_class_mappings[pred[0][0]]}")
    print(f"Actual Sentiment: {label_to_class_mappings[y_test[rand_index][0]
```

Input News: Cargotec Corporation , Press Release , August 26 , 2008 at 10 a. m. Finnish time Cargotec 's MacGREGOR business area providing marine cargo h andling and offshore load handling solutions has received significant offsho re crane retrofit order .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Martela said plans to expand its recycled furniture business els ewhere in Finland , too .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Ruukki Romania , the local arm of Finnish metal producer Ruukki , increased its capital by 900,000 euro ($ 1.14 mln) through cash contribu tion , it was reported on September 19 , 2006 .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: : Lietuvos Respublikos sveikatos apsaugos ministerija has awarde d contract to UAB `` AFFECTO LIETUVA '' for financial systems software packa ge .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: In January-June 2010 , diluted loss per share stood at EUR0 .3 v ersus EUR0 .1 in the first half of 2009 .

Predicted Sentiment: positive
Actual Sentiment: negative

Input News: Componenta 's objective with this agreement is to increase busin ess on its existing production lines .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: In contrast , the company 's net loss for the third quarter of 2 009 contracted to EUR 76 million from EUR 256 million for the corresponding period a year ago .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: The Finnish supplier of BSS-OSS and VAS for telecom operators , Tecnotree , has received expansion orders worth a total US$ 7.3 mn for its c onvergent charging and next generation messaging solutions in Latin America , the company announced without specifying which operators had placed the or ders .

Predicted Sentiment: positive
Actual Sentiment: positive

```
Input News: The company said that the fall in turnover had been planned .

Predicted Sentiment: positive
Actual Sentiment: negative

Input News: According to Arokarhu , some of the purchases that had been scan
ned into the cash register computer disappeared when the total sum key was p
ressed .

Predicted Sentiment: negative
Actual Sentiment: negative
```

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 **slope.py**, 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 N x 2 matrix and use this to determine the first principal component vector of this dataset. Be careful of how you are stacking the two vectors. You can check the output by printing it which should help you debug. 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.

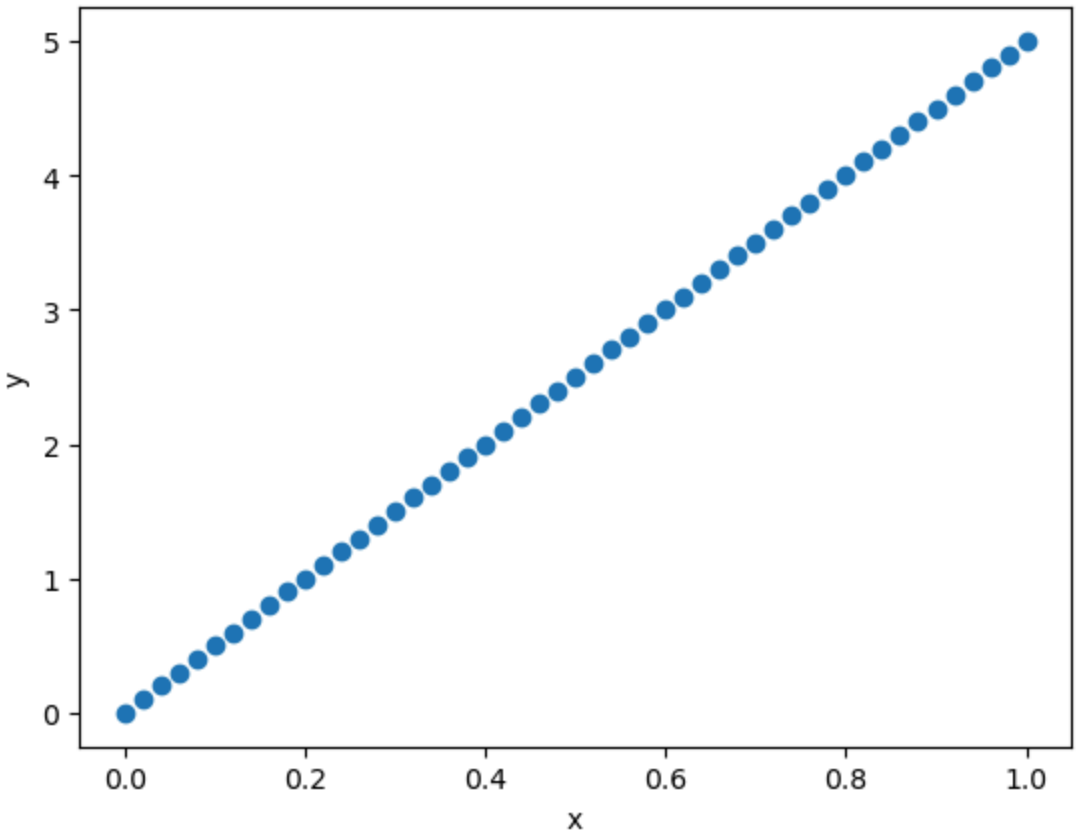
As a reminder, least squares regression minimizes the squared error of the dependent variable from its prediction. Namely, given (x_i, y_i) pairs, least squares returns the line $l(x)$ that minimizes $\sum_i (y_i - l(x_i))^2$.

In [ ]:
```
from slope import Slope
```

We will consider a simple example with two variables, x and y, where the true relationship between the variables is $y = 5x$. Our goal is to recover this relationship—namely, recover the coefficient "5". We set $X = [0, .02, .04, .06, \ldots, 1]$ and $y = 5x$. Make sure both functions return 5.

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

y = 5 * np.arange(0, 1.02, 0.02).reshape(-1, 1)

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

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

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

plt.show()
```

```
Slope of first principal component 0.9805806756909202
Slope of best linear fit [5.]
```



5.2 Analysis Setup [5 pts] [W]

In **slope.py**, 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 $\hat{y}_i \sim 5x_i + \mathcal{N}(0, c) = 5i/1000 + \mathcal{N}(0, c)$, 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 ***xnoise*** 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).

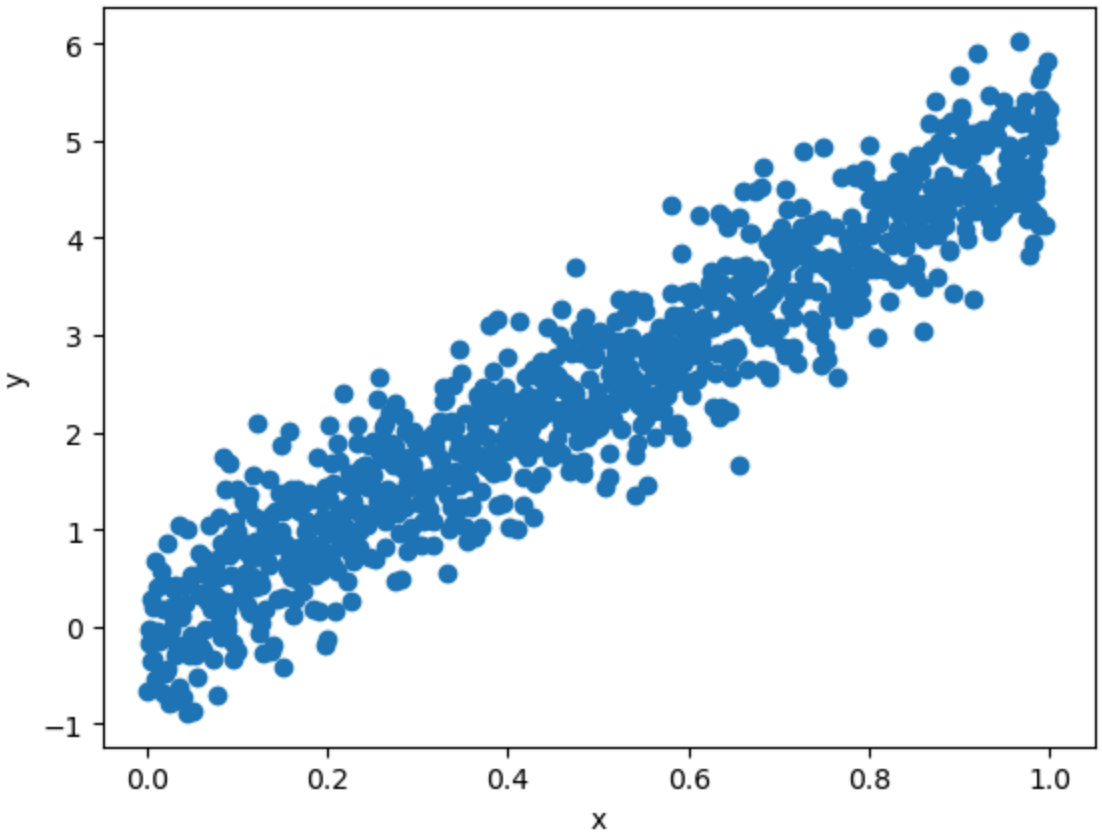
Hint 1: Refer to the example in the cell below on how to add noise to X or Y

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

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 = 5 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

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

plt.show()
```



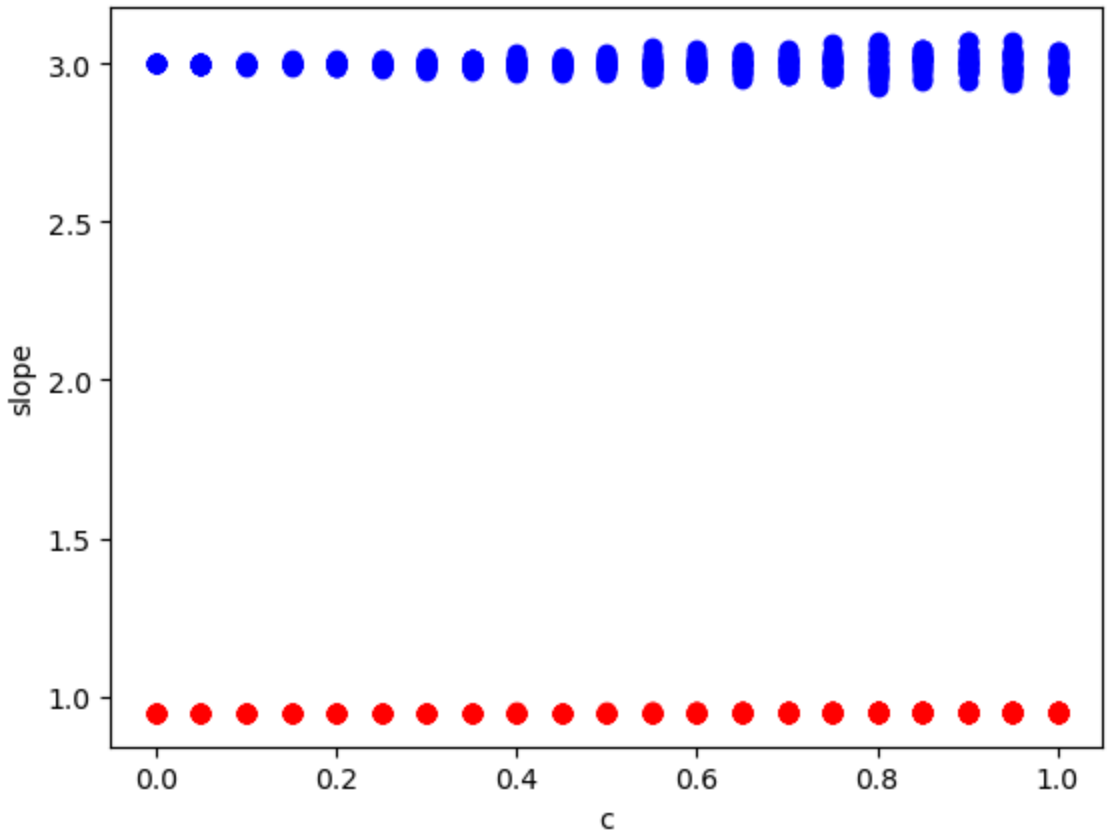
A scatter plot with c on the horizontal axis and the output of **pca_slope** and **lr_slope** on the vertical axis has already been implemented for you.

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 that the plot you get might not look exactly

like the TA version and that is fine because you might have randomized the noise slightly differently than how we did it.

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

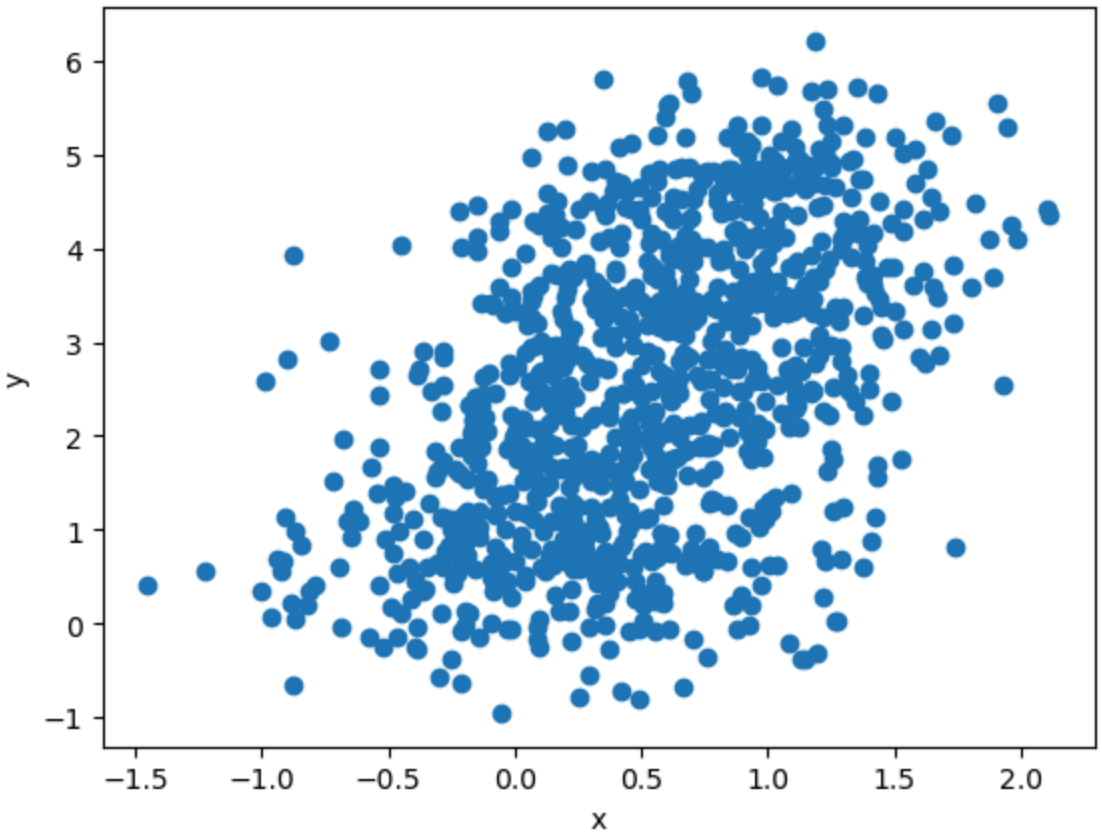
In [ ]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################
pca_slope_values = []
linreg_slope_values = []
c_values = []
s_idx = 0

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 = Slope.addNoise(c, x_noise=False, seed=s_idx)

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

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

        # Increment random seed index
        s_idx += 1

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

plt.show()
```



Error in x and y

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.

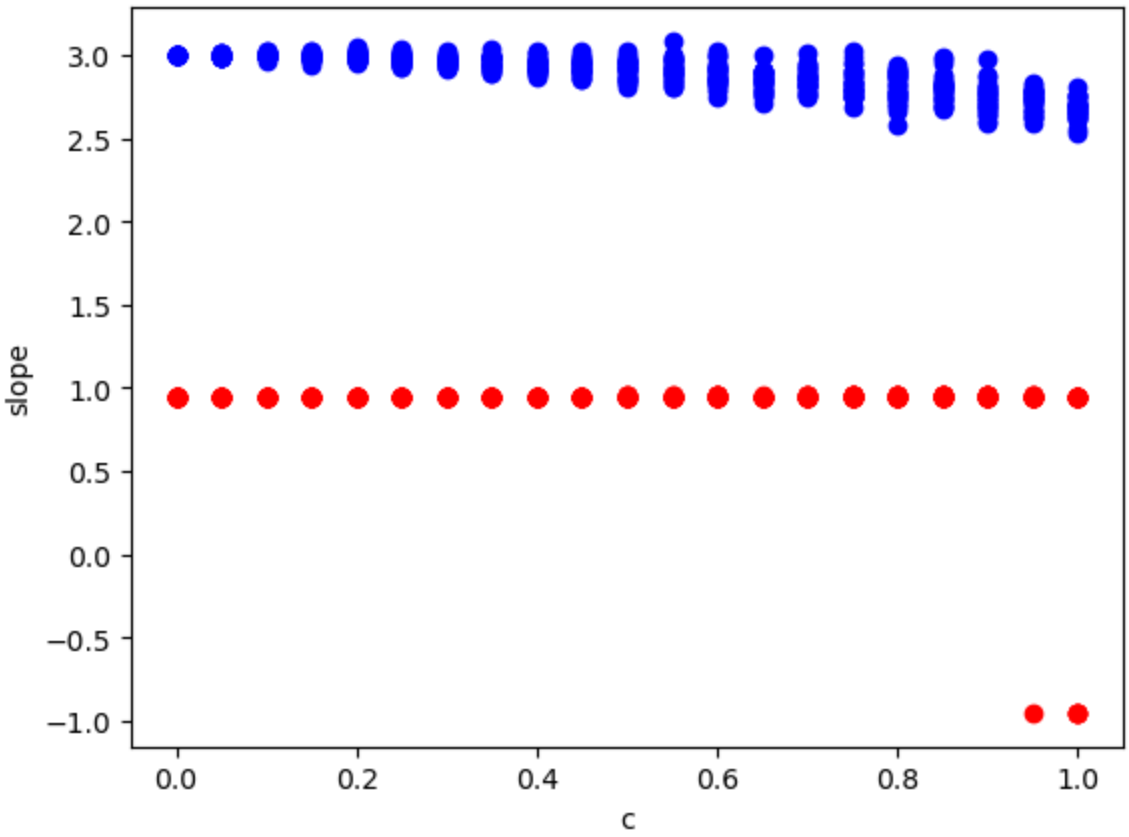
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
base = np.arange(0.001, 1, 0.001).reshape(-1, 1)
c = 0.5
X = base + np.random.normal(loc=[0], scale=c, size=base.shape)
y = 5 * base + np.random.normal(loc=[0], scale=c, size=base.shape)

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

plt.show()
```



In the below cell, we graph the predicted PCA and LR slopes on the vertical axis against the value of c on the horizontal axis. Note that the graph you get might not look exactly like the TA version and that is fine because you might have randomized the noise slightly differently than how we did it.

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
        psv, lsv = Slope.addNoise(c, x_noise=True, seed=s_idx)

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

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

        # Increment random seed index
        s_idx += 1
```

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

plt.show()
```



5.3. Analysis [5 pts] [W]

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

NOTE:

1. You don't need to provide a mathematical proof for this question.
2. Understanding how PCA and Linear Regression work should help you decipher which case was better for which algorithm. Base your answer on this understanding of how either algorithms works.

QUESTIONS:

1. Based on the obtained plots, how can you determine which technique (PCA/Linear regression) is performing better in comparison? (1 Pt)
2. In case-1 where there is error in Y which technique gave better performance and why do you think it performed better in this case? (2 Pts)

3. In case-2 where there is error in both X and Y which technique gave better performance and why do you think it performed better in this case? (2 Pts)

5.3. **Answer:**

1. To determine which technique is performing better, one would compare the consistency of the slope as noise is introduced. PCA is generally more robust against noise in the dependent variable Y, showing less variation in slope. Linear regression tends to have a constant slope when X is noise-free, but its performance decreases when X also includes noise.
2. With error present only in Y, linear regression is expected to outperform PCA because it minimizes the vertical distances from the line to the points in Y, effectively handling the noise in Y when X is without error.
3. When there is error in both X and Y, PCA is likely to give a better performance. This is due to PCA's approach of capturing the direction that accounts for the most variance in the data without making a distinction between independent and dependent variables. Consequently, PCA can dilute the noise across all variables, finding the line that best captures the inherent structure of the data.

Q6 Feature Reduction Implementation [5.6% Bonus for All] [P] | [W]

6.1 Implementation [4%] [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**

Reminder: A 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. More briefly, a p-value is a measure of how much the given feature significantly represents an observed change. A lower p-value represents higher significance. Some more information about p-values can be found here:
https://towardsdatascience.com/what-is-a-p-value-b9e6c207247f

You will be given a list of significance levels and you will perform feature reduction on the dataset for each significance level. This will display the impact of the significance levels on feature reduction. These functions should each return dictonary of significance levels and their associated features.

Forward Selection:

In forward selection, we will loop through the signifiance level list. 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 current significance level. After this terminates, we will restart this process with a new significance level.

Steps to implement it:

1. Loop through the significance levels. Start with the first significance level.
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, go to step 6.
6. Store list of features in dictonary and restart process with new significance level (go to step 2). Terminate if all significance levels have been looped through.

Backward Elimination:

In backward elimination, we will loop through the signifiance level list. We start with a full model, and then remove the insignificant feature with the highest p-value (that is greater than the current significance level). We continue to do this until we find a feature that has a p-value that is less than or equal to the current confidence level.

Steps to implement it:

1. Loop through the significance levels. Start with the first significance level.
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 go to step 6.
4. Remove the feature under consideration.
5. Fit a model without this feature. Repeat entire process from Step 3 onwards.
6. Store list of features in dictonary and restart process with new significance level (go to step 2). Terminate if all significance levels have been looped through.

HINT 1: 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

HINT 2: 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.

HINT 3: 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)
print("Dataset Features: ", bc.columns.tolist())
bc["Diagnosis"] = bc_dataset.target

X = bc.drop("Diagnosis", axis=1)
y = bc["Diagnosis"]
featureselection = FeatureReduction()
# Run the functions to make sure two dictonaries are generated, one for each
forward_selection_feature_map = FeatureReduction.forward_selection(
    X, y, [0.01, 0.1, 0.2]
)
backward_selection_feature_map = FeatureReduction.backward_elimination(
    X, y, [0.01, 0.1, 0.2]
)
```

```
Dataset Features:  ['mean radius', 'mean texture', 'mean perimeter', 'mean a
rea', 'mean smoothness', 'mean compactness', 'mean concavity', 'mean concave
points', 'mean symmetry', 'mean fractal dimension', 'radius error', 'texture
error', 'perimeter error', 'area error', 'smoothness error', 'compactness er
ror', 'concavity error', 'concave points error', 'symmetry error', 'fractal
dimension error', 'worst radius', 'worst texture', 'worst perimeter', 'worst
area', 'worst smoothness', 'worst compactness', 'worst concavity', 'worst co
ncave points', 'worst symmetry', 'worst fractal dimension']
```

In [ ]:
```
# evalute best p_value
# The evaluate features method performs linear regression on each specificed
# It calculates the rmse for each feature list and then displays the best si
```

```
FeatureReduction.evaluate_features(X, y, forward_selection_feature_map)
FeatureReduction.evaluate_features(X, y, backward_selection_feature_map)
```

```
significance level: 0.01, Removed features: {'area error', 'mean texture',
'mean radius', 'worst concavity', 'worst compactness', 'concavity error', 'm
ean perimeter', 'perimeter error', 'mean concavity', 'symmetry error', 'mean
fractal dimension', 'worst perimeter', 'worst smoothness', 'mean symmetry',
'mean area', 'texture error', 'concave points error', 'fractal dimension err
or', 'mean smoothness'}
significance level: 0.01, RMSE: 0.23191025244141578
significance level: 0.1, Removed features: {'mean fractal dimension', 'mean
symmetry', 'mean texture', 'mean perimeter', 'mean radius', 'mean area', 'wo
rst compactness', 'texture error', 'concave points error', 'perimeter erro
r', 'worst perimeter', 'worst smoothness', 'fractal dimension error', 'mean
concavity', 'symmetry error', 'mean smoothness'}
significance level: 0.1, RMSE: 0.23821726582785402
significance level: 0.2, Removed features: {'mean fractal dimension', 'mean
symmetry', 'mean texture', 'mean perimeter', 'mean radius', 'mean area', 'wo
rst compactness', 'texture error', 'perimeter error', 'worst perimeter', 'wo
rst smoothness', 'fractal dimension error', 'symmetry error', 'mean smoothne
ss'}
significance level: 0.2, RMSE: 0.2412375366022595
Best significance level: 0.01, RMSE: 0.23191025244141578

significance level: 0.01, Removed features: {'area error', 'mean texture',
'mean radius', 'worst compactness', 'mean perimeter', 'worst concave point
s', 'perimeter error', 'compactness error', 'mean concavity', 'symmetry erro
r', 'mean fractal dimension', 'worst perimeter', 'worst smoothness', 'mean s
ymmetry', 'mean area', 'texture error', 'concave points error', 'fractal dim
ension error', 'mean smoothness'}
significance level: 0.01, RMSE: 0.24042471216353772
significance level: 0.1, Removed features: {'area error', 'mean fractal dime
nsion', 'mean symmetry', 'mean texture', 'mean perimeter', 'mean area', 'wor
st compactness', 'worst concave points', 'texture error', 'perimeter error',
'worst perimeter', 'worst smoothness', 'fractal dimension error', 'compactne
ss error', 'mean concavity', 'symmetry error', 'mean smoothness'}
significance level: 0.1, RMSE: 0.2374589600443534
significance level: 0.2, Removed features: {'area error', 'mean fractal dime
nsion', 'mean symmetry', 'mean texture', 'mean perimeter', 'mean area', 'wor
st compactness', 'worst concave points', 'texture error', 'perimeter error',
'worst perimeter', 'worst smoothness', 'fractal dimension error', 'compactne
ss error', 'symmetry error', 'mean smoothness'}
significance level: 0.2, RMSE: 0.2363767160772516
Best significance level: 0.2, RMSE: 0.2363767160772516
```

6.2 Feature Selection - Discussion [1.6%] [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. Considering there are no restrictions set on the dataset,

according to you, which of these two methods is more appropriate for feature selection generally (choose one method)? Why? **(3 pts)**

6.2.1. **Answer:**

Lasso is generally more appropriate for feature selection. Lasso regression can yield sparse models, that is, models where insignificant input features have zero as their coefficients. This effectively eliminates those features from the equation, simplifying the model and highlighting the most important features. Lasso includes a penalty term that constrains the size of the coefficients. As the strength of regularization is increased, more coefficients are forced to zero, thus performing feature selection. By reducing the number of features, Lasso helps in creating more interpretable models which are easier to understand and explain because they focus only on the most significant predictors.

In contrast, Ridge regression tends to shrink the coefficients for the least important features but does not set them to zero. Therefore, it does not result in feature elimination but rather feature reduction, which is useful for addressing multicollinearity or when you want to include all features but regulate their impact on the model.

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)**

6.2.2. **Answer:**

In forward selection, we start with no features and sequentially add one feature at a time, choosing the feature that provides the best improvement to the model at each step. Since we include an intercept in the model, we only need to consider combinations of the N features.

At the first step, we have N possible models to consider, one for each single feature with the intercept. At the second step, for each feature added, we have $N-1$ additional models, and this continues such that at the k-th step we have $N-(k-1)$ models to consider. However, not all possible combinations of features are considered because we stop adding features once the improvement is not significant.

The total number of models M explored in forward selection, assuming we add all N features, would be the sum of models considered at each step:

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

But since we have a total of 2^n possible models when considering all combinations of N features, the fraction F of the total possible models explored is:

$$F = \frac{M}{2^N} = \frac{N(N+1)}{2^{N+1}}$$

This fraction indicates that, as (N) grows, forward selection explores an increasingly smaller fraction of the total possible models, since the number of possible combinations grows exponentially while the number of models explored in forward selection grows quadratically.

Q7: Netflix Movie Recommendation Problem Solved using SVD [2.1% 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:

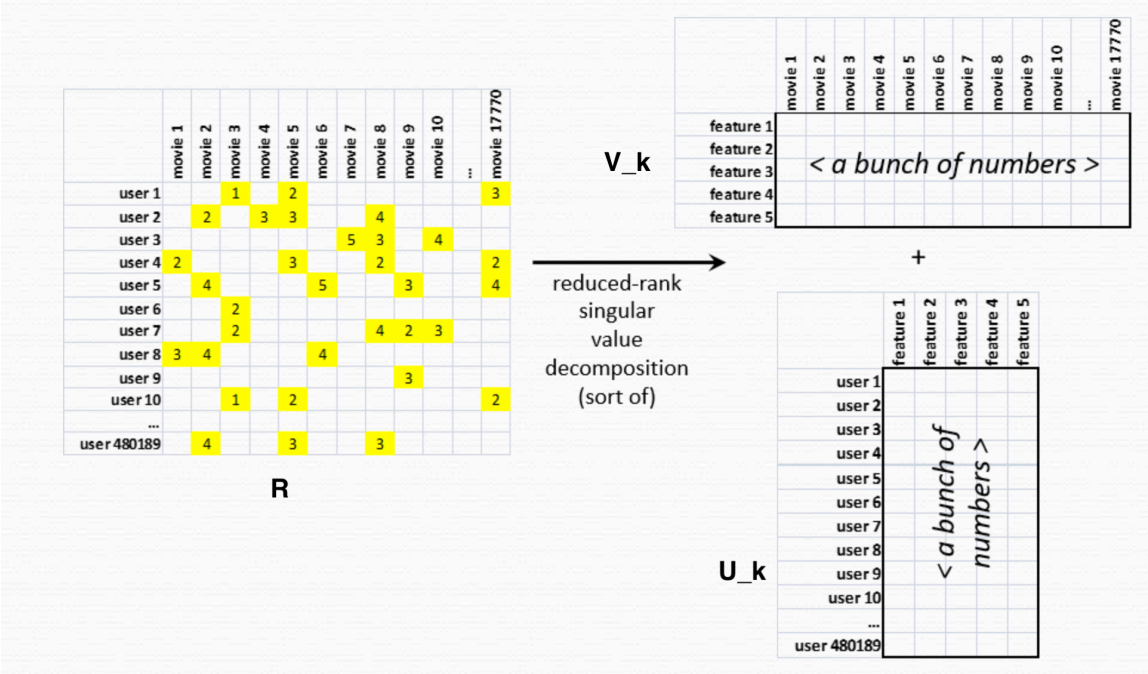
$$R = U\sqrt{\Sigma}\sqrt{\Sigma}V^T$$

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: $u_i \cdot m_j$ 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

Hint: Movie IDs are IDs assigned to the movies in the dataset and can be greater than the number of movies. This is why we have given movies_index and users_index as well that map between the movie IDs and the indices in the ratings matrix. Please make sure to use this as well.

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

from regression import Regression
from svd_recommender import SVDRecommender
```

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().s
print("Number of unique users (train): {}".format(train["movieId"].unique().
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().sha
print("Number of unique users (test): {}".format(test["movieId"].unique().sh
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
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
```

```
-----------------------------------------
Train Dataset Stats:
Shape of train dataset: (88940, 4)
Number of unique users (train): 671
Number of unique users (train): 8370
Sample of Train Dataset:
-----------------------------------------
   userId  movieId  rating   timestamp
0       1     2294     2.0  1260759108
1       1     2455     2.5  1260759113
2       1     3671     3.0  1260759117
3       1     1339     3.5  1260759125
4       1     1343     2.0  1260759131
-----------------------------------------
Test Dataset Stats:
Shape of test dataset: (10393, 4)
Number of unique users (test): 671
Number of unique users (test): 4368
Sample of Test Dataset:
-----------------------------------------
   userId  movieId  rating   timestamp
0       1     2968     1.0  1260759200
1       1     1405     1.0  1260759203
2       1     1172     4.0  1260759205
3       2       52     3.0   835356031
4       2      314     4.0   835356044
-----------------------------------------
Shape of Ratings Matrix (R): (671, 8370)
```

7.1.1 Implement the `recommender_svd` method to use SVD for Recommendation [1.05%] [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 initial U, Σ and V matrices. Then, calculate U_k and V_k based on the decomposition example above.

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 utilities.localtests import TestSVDRecommender

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

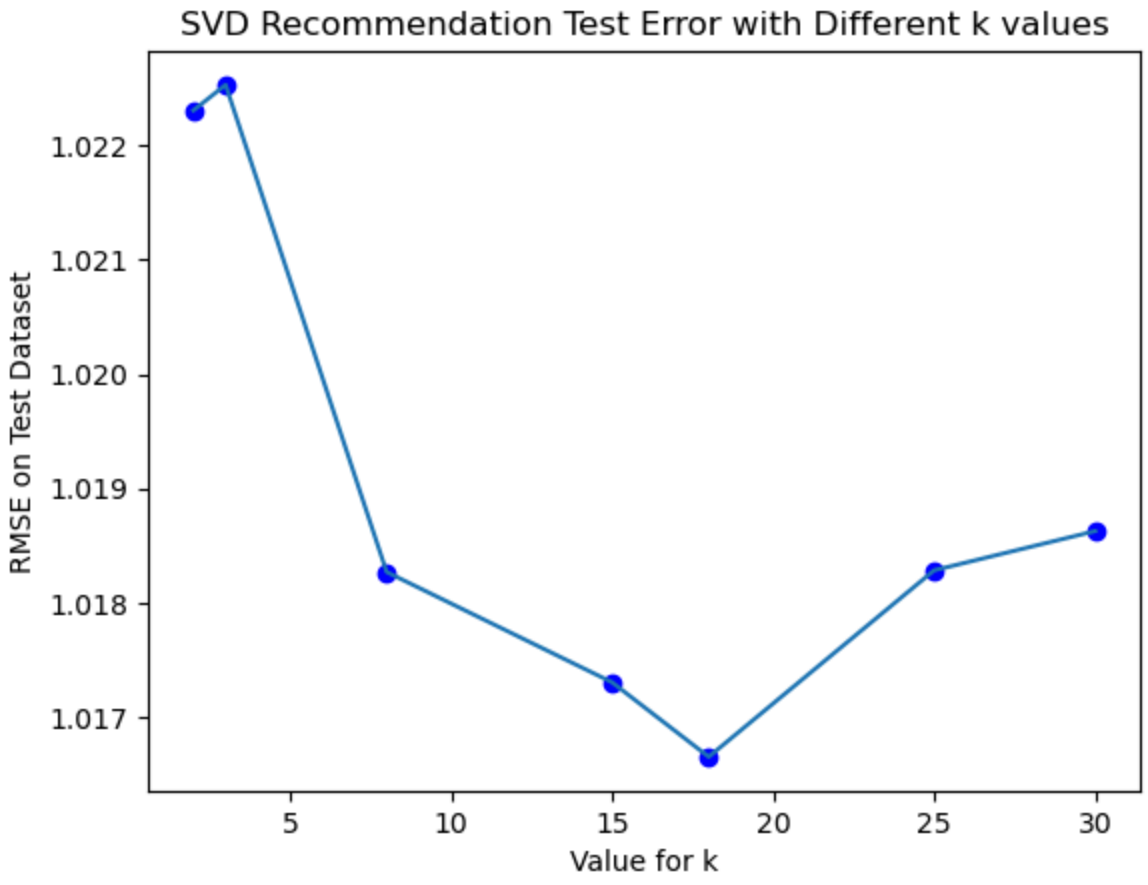
```
UnitTest passed successfully for "recommender_svd() function"!
```

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

# Implement the method `recommender_svd` and run it for the following values
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
        # Else, use an average of the users ratings
        else:
            pred_rating = np.mean(np.dot(U_k[u_index], V_k)) + r_means[m_inc
        pred.append(pred_rating)
    test_error = regression.rmse(test["rating"], pred)
    test_errors.append(test_error)
    print("RMSE for k = {} --> {}".format(k, test_error))
```

```
RMSE for k = 2 --> 1.0223035413708281
RMSE for k = 3 --> 1.022526649417955
RMSE for k = 8 --> 1.0182709203352787
RMSE for k = 15 --> 1.017307118738714
RMSE for k = 18 --> 1.0166562048687973
RMSE for k = 25 --> 1.0182856984912254
RMSE for k = 30 --> 1.0186282488126601
```

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")

plt.show()
```



7.1.2 Implement the `predict` method to find which movie a user is interested in watching next [1.05%] [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()
```

```
Top 3 Movies the User would want to watch:
Captain America: The First Avenger (2011)
Ant-Man (2015)
Avengers: Age of Ultron (2015)
------------------------------------------------------------
UnitTest passed successfully for "predict() function"!
```

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.

NOTE: There are 2 possible visualizations. Your plot could be the one that's given on the expected PDF or the one where the y-coordinates are inverted.

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_ti
]

_, 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]))
```