Fall 2024 CS4641/CS7641 Homework 3

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, November 8th, 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 format. Please ensure all questions are answered within the Jupyter Notebook using either Markdown or LaTeX. 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

- **1.1 Image Compression** [30 pts] - *programming*
  - svd [6pts]
  - compress [6pts]
  - rebuild_svd [6pts]
  - compression_ratio [6pts]
  - recovered_variance_proportion [6pts]
- **1.2 Black and White Compression** [0 pts]
- **1.3 Color Compression** [0 pts]

Q2: Understanding PCA [20pts]

Deliverables: pca.py and Written portion

- **2.1 PCA Implementation** [10 pts] - *programming*
  - fit [5pts]
  - transform [2pts]
  - transform_rv [3pts]
- **2.2 Visualize** [5 pts] *non-programming*
- **2.3 PCA Reduced Facemask Dataset Analysis** [5 pts] *non-programming*
- **2.4 PCA Exploration** [0 pts]

Q3: Regression and Regularization [72pts: 52pts + 20pts Grad / 6% Bonus for Undergrads + 2.3% Bonus for All]

Deliverables: regression.py and Written portion

- **3.1 About RMSE** [3 pts] *non-programming*
- **3.2 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]
- **3.3 Testing: General Functions and Linear Regression** [5 pts] *programming*
- **3.4 Testing: Ridge Regression** [5 pts] *programming*

- **3.5 Linear vs. Ridge Regression Analysis** [4 pts] *non-programming*
- **3.6 Cross Validation Hyperparameter Search** [5 pts] *programming*
- **3.7 Noisy Input Samples in Linear Regression** [2.3% **BONUS FOR ALL**] *non-programming*

Q4: Naive Bayes and Logistic Regression [39pts]

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]
- **4.3 Logistic Regression Threshold Experiments** [2 pts] *non-programming*

Q5: Feature Selection [30pts]

Deliverables: feature_reduction.py

- **5.1 Feature Reduction** [30pts] - *programming*
  - forward_selection [15pts]
  - backward_elimination [15pts]

Q6: Imbalanced Classes in Classification Tasks [5.6% Bonus for All]

Deliverables: smote.py

- **6.1 A More Comprehensive Measure** [1.75%] *programming*
  - generate_confusion_matrix [0.75%]
  - f1_scores [1%]
- **6.2 SMOTE** [3.85%] *programming*
  - interpolate [1%]
  - k_nearest_neighbors [0.75%]
  - smote [2.1%]

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

Deliverables: svd_recommender.py and Written portion (also include imgcompression.py in the submission as svd_recommender.py imports imgcompression)

- **7.1 SVD Recommender** [2.1%] *programming*
  - recommender_svd [1.05%]
  - predict [1.05%]
- **7.2 Visualize Movie Vectors** [0pts]

Points Totals:

- Total Base: 191 pts for grads / 171 pts for undergrads
- Total Undergrad Bonus: 6%
- Total Bonus for All: 10%

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
    - feature_reduction.py
- **Assignment 3 Bonus for Undergrad - Programming**
    - regression.py
- **Assignment 3 Bonus for All - Programming**
    - smote.py
    - imgcompression.py (used as an import in svd_recommender.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,
    make_classification,
)
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
from sklearn.svm import SVC

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 = "notebook+vscode+png"
```

```
Version information
python: 3.11.10 (main, Oct  3 2024, 02:26:51) [Clang 14.0.6 ]
matplotlib: 3.9.2
numpy: 1.26.4
```

Q1: Image Compression [30 pts] [P]

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)
plt.show()
```

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))

plt.imshow(rgb2gray(image), cmap=plt.cm.bone)
plt.show()
```

1.1 Image compression [30pts] [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.

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.

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: To improve image processing functions, you may need to implement separate methods for handling color and grayscale images within the same function.

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()
```

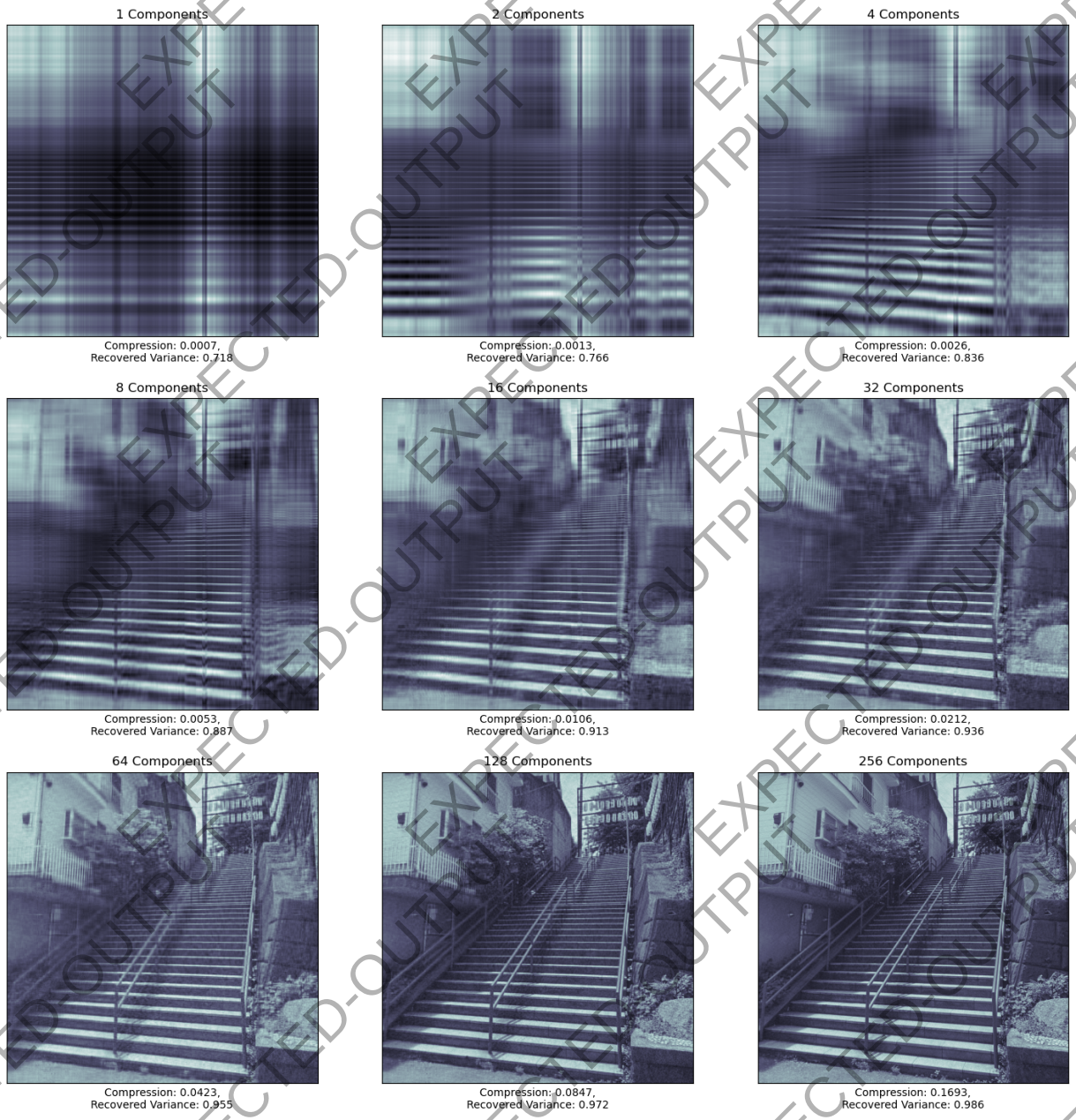
```
UnitTest passed successfully for "SVD calculation - color images"!
UnitTest passed successfully for "Image compression - color images"!
UnitTest passed successfully for "SVD reconstruction - color images"!
UnitTest passed successfully for "Compression ratio - color images"!
UnitTest passed successfully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

1.2 Black and White Compression [No Points]

This question will utilize your implementation of the functions from Q1.1 to compare the byte size needed for representing the SVD decomposition of the original image versus the compressed image at various compression levels. You can directly execute the following cell without modifying it, provided that you have already 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]

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_co
    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

plt.show()
```



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: 69.768 MB -> Compressed Image: 47.258 KB, Savi
ngs: 69.721 MB, Compression Ratio 1511.8:1
2 components: Original Image: 69.768 MB -> Compressed Image: 94.516 KB, Savi
ngs: 69.675 MB, Compression Ratio 755.9:1
4 components: Original Image: 69.768 MB -> Compressed Image: 189.031 KB, Sav
ings: 69.583 MB, Compression Ratio 377.9:1
8 components: Original Image: 69.768 MB -> Compressed Image: 378.062 KB, Sav
ings: 69.398 MB, Compression Ratio 189.0:1
16 components: Original Image: 69.768 MB -> Compressed Image: 756.125 KB, Sa
vings: 69.029 MB, Compression Ratio 94.5:1
32 components: Original Image: 69.768 MB -> Compressed Image: 1.477 MB, Savi
ngs: 68.291 MB, Compression Ratio 47.2:1
64 components: Original Image: 69.768 MB -> Compressed Image: 2.954 MB, Savi
ngs: 66.814 MB, Compression Ratio 23.6:1
128 components: Original Image: 69.768 MB -> Compressed Image: 5.907 MB, Sav
ings: 63.86 MB, Compression Ratio 11.8:1
256 components: Original Image: 69.768 MB -> Compressed Image: 11.814 MB, Sa
vings: 57.953 MB, Compression Ratio 5.9:1
```

1.3 Color Compression [No Points]

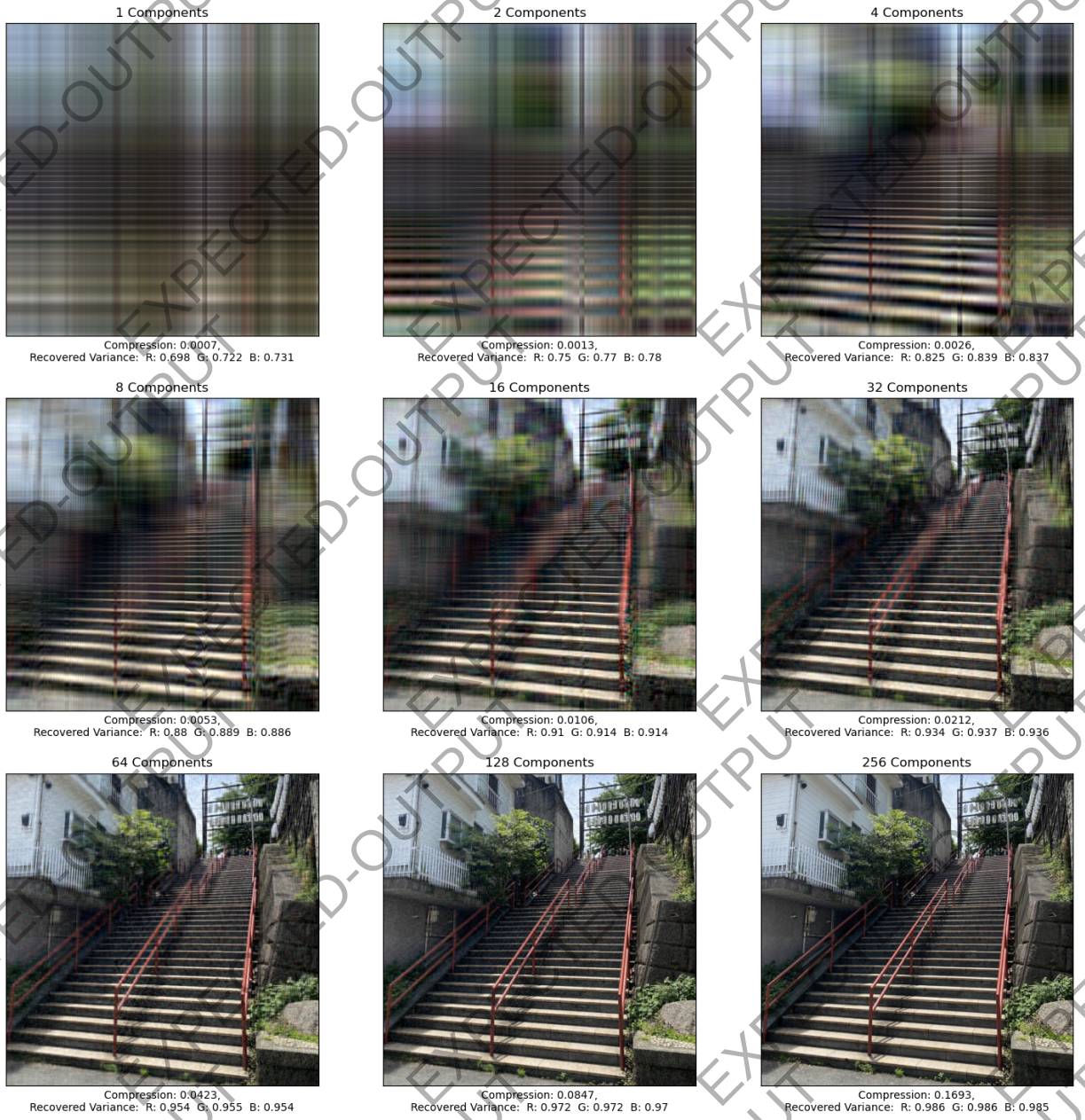
This section will use your implementation of the functions from Q1.1 to generate a set of images compressed to different degrees. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

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

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.

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

plt.show()
```



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: 209.303 MB -> Compressed Image: 141.773 KB, Sa
vings: 209.164 MB, Compression Ratio 1511.8:1
2 components: Original Image: 209.303 MB -> Compressed Image: 283.547 KB, Sa
vings: 209.026 MB, Compression Ratio 755.9:1
4 components: Original Image: 209.303 MB -> Compressed Image: 567.094 KB, Sa
vings: 208.749 MB, Compression Ratio 377.9:1
8 components: Original Image: 209.303 MB -> Compressed Image: 1.108 MB, Savi
ngs: 208.195 MB, Compression Ratio 189.0:1
16 components: Original Image: 209.303 MB -> Compressed Image: 2.215 MB, Sav
ings: 207.088 MB, Compression Ratio 94.5:1
32 components: Original Image: 209.303 MB -> Compressed Image: 4.43 MB, Savi
ngs: 204.872 MB, Compression Ratio 47.2:1
64 components: Original Image: 209.303 MB -> Compressed Image: 8.861 MB, Sav
ings: 200.442 MB, Compression Ratio 23.6:1
128 components: Original Image: 209.303 MB -> Compressed Image: 17.722 MB, S
avings: 191.581 MB, Compression Ratio 11.8:1
256 components: Original Image: 209.303 MB -> Compressed Image: 35.443 MB, S
avings: 173.859 MB, Compression Ratio 5.9:1
```

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, including Iris Dataset and Facemask Dataset.

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 two random features following the same procedure and examine the difference.

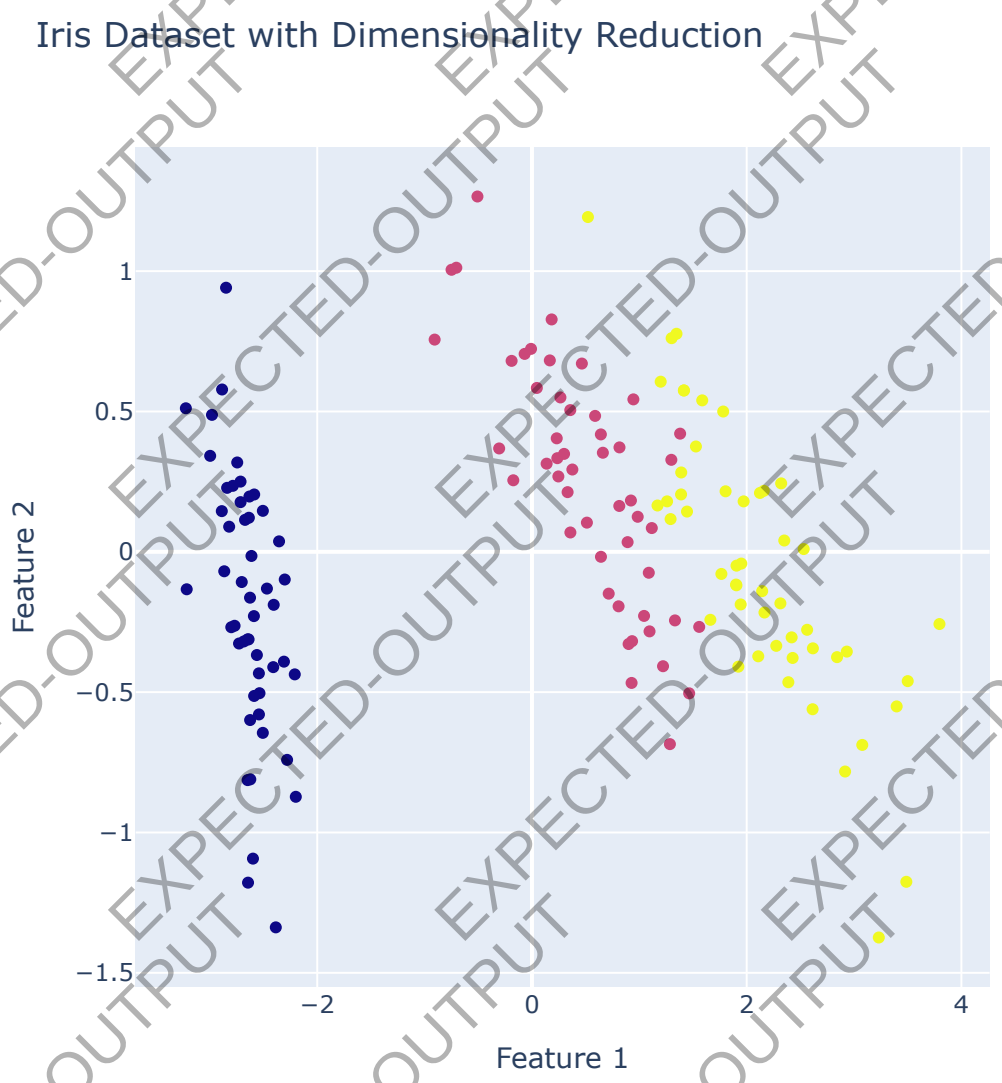
The datasets have already been loaded for you in the subsequent cells.

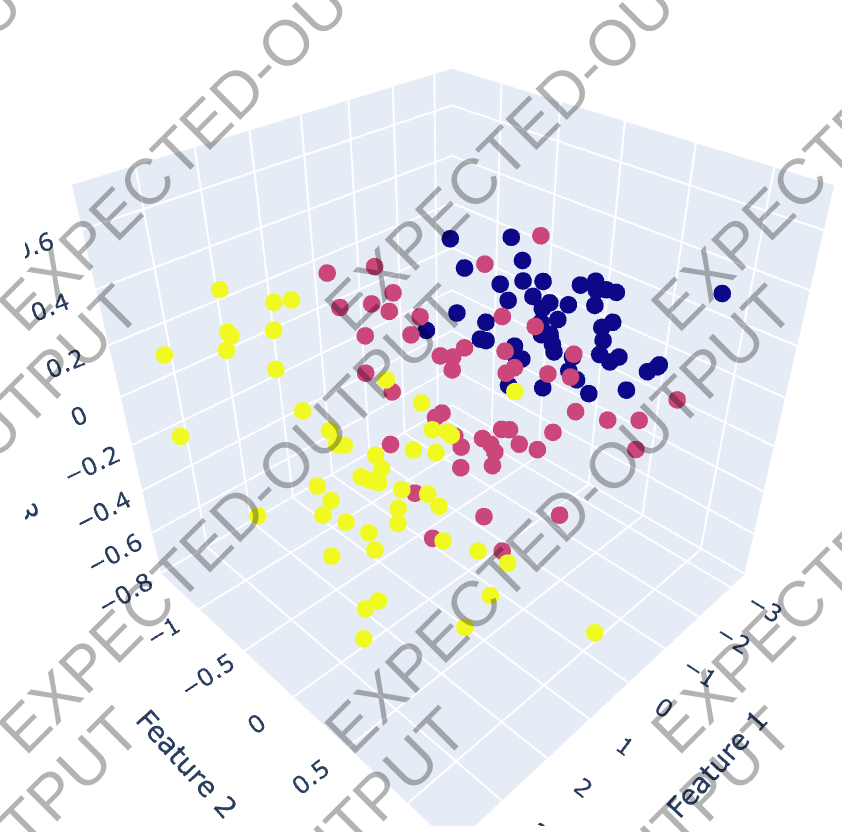
NOTE: Here, we won't be testing for accuracy. Even with correct implementations of PCA, the accuracy can differ from the TA solution. That is fine as long as the visualizations come out similar.

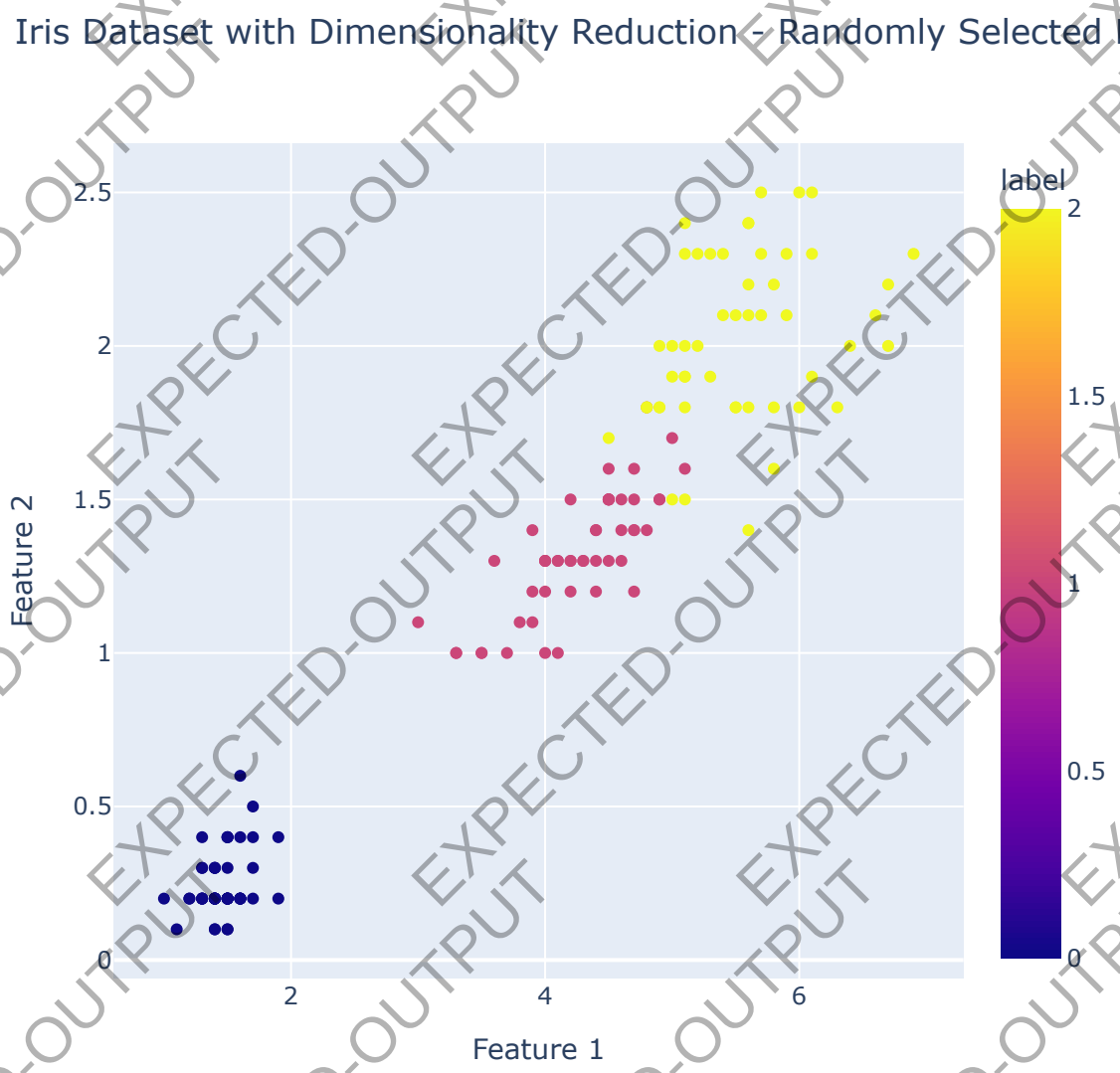
Iris Dataset

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







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

Facemask Dataset

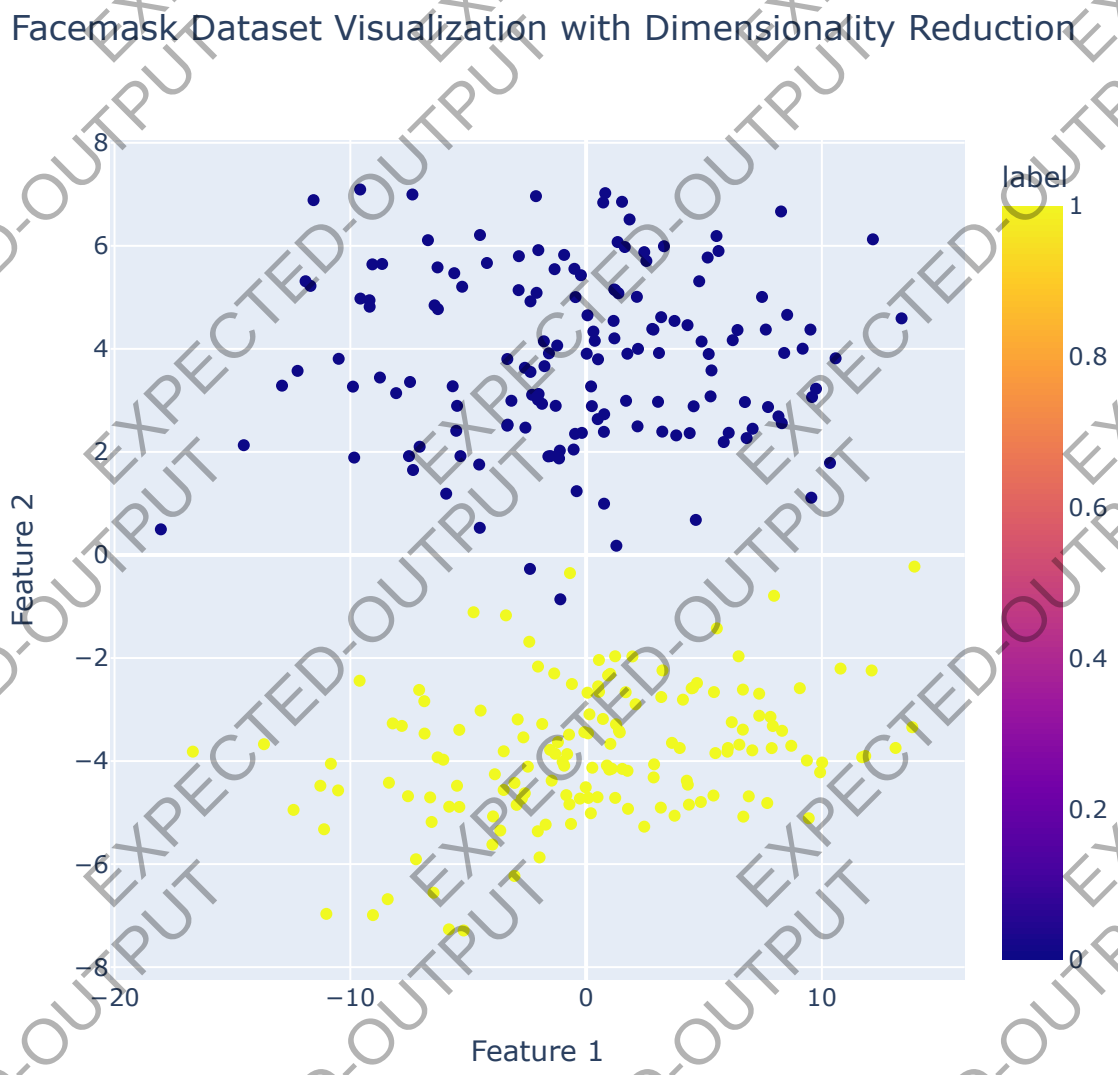
The masked and unmasked dataset is made up of grayscale images of human faces facing forward. Half of these images are faces that are completely unmasked, and the remaining images show half of the face covered with an artificially generated face mask. The images have already been preprocessed, they are 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.

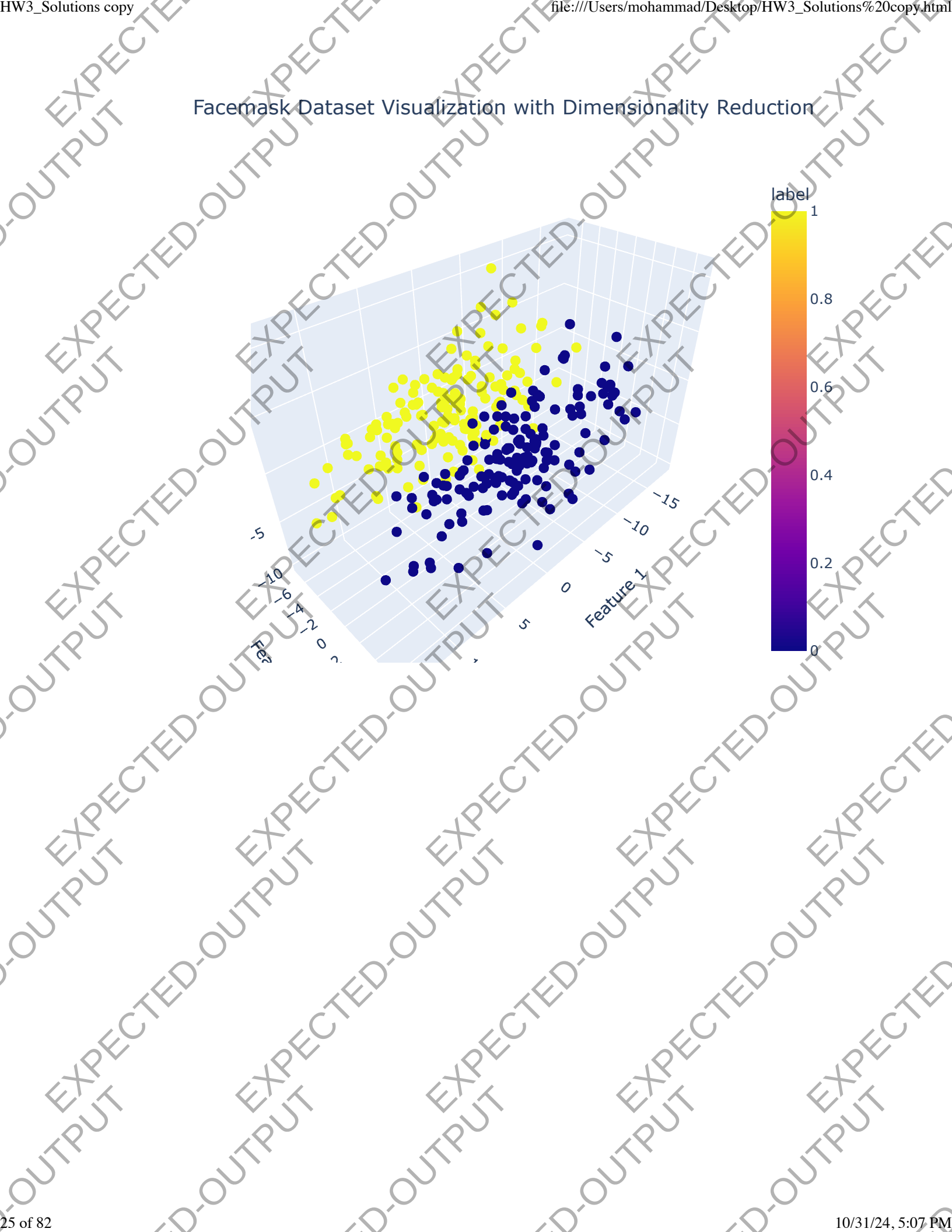
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

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

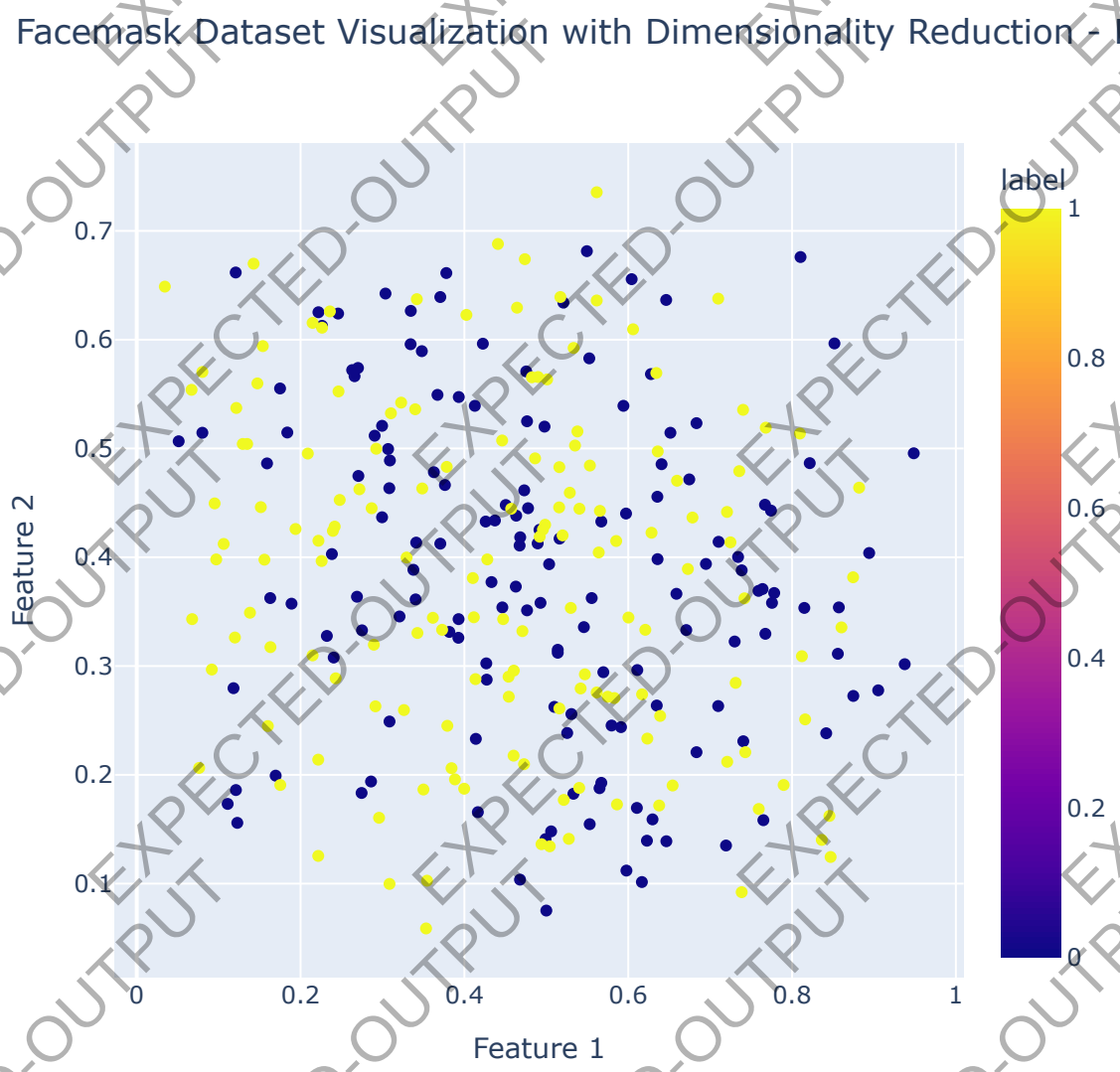
```
i = 0
fig = plt.figure(figsize=(18, 18))
for idx in [0, 1, 2, 150, 151, 152]:
    ax = fig.add_subplot(6, 6, i + 1, xticks=[], yticks=[])
    image = (
        np.rot90(X[idx].reshape(64, 64), k=1)
        if idx % 2 == 1 and idx < 150
        else X[idx].reshape(64, 64)
    )
    m_status = "Unmasked" 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().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. Examine the 2-dimensional plot of the *facemask* dataset reduced to 2 principal components. How might PCA help in dealing with noise, overfitting, and feature correlation in a high-dimensional dataset? Discuss scenarios where using PCA-reduced data could outperform a classifier trained on the original high-dimensional data. **(3 pts)**

   Answer ...

2. When applying PCA to reduce a high-dimensional dataset (e.g., 4096 features) to just 2 components, how might this impact the classifier's ability to capture non-linear patterns in the data? Discuss how the trade-off between maximizing variance

(PCA's goal) and preserving class separability could affect classification performance. **(2 pts)**

Answer ...

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 [72pts: 52pts + 20pts Grad / 6% Bonus for Undergrads + 2.3% Bonus for All] [P] | [W]

3.1 About RMSE (Root Mean Square Error) [3 pts] [W]

Mean Squared Error (MSE) is used to provide an indication of how well a model is performing in terms of prediction accuracy.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

where:

- n is the number of data points
- y_i is the actual target value
- $\hat{y_i}$ is the predicted value by the model

However, Root Mean Square Error (RMSE) is preferred over MSE because it brings the error back to the same units as the target variable, providing easier comparison to the actual values.

This helps in providing better practical interpretation of how well the model is performing on both small and large errors.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2}$$

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 a toy example with your explanation.

Answer:

3.2 Regression and regularization implementations [50pts: 30pts + 20pts Grad / 6% 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.

3.2.1 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"!
```

3.2.2 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"!
```

3.2.3 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"!
```

3.2.4 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"!
```

3.3 Testing: General Functions and Linear Regression [5 pts] [P]

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.

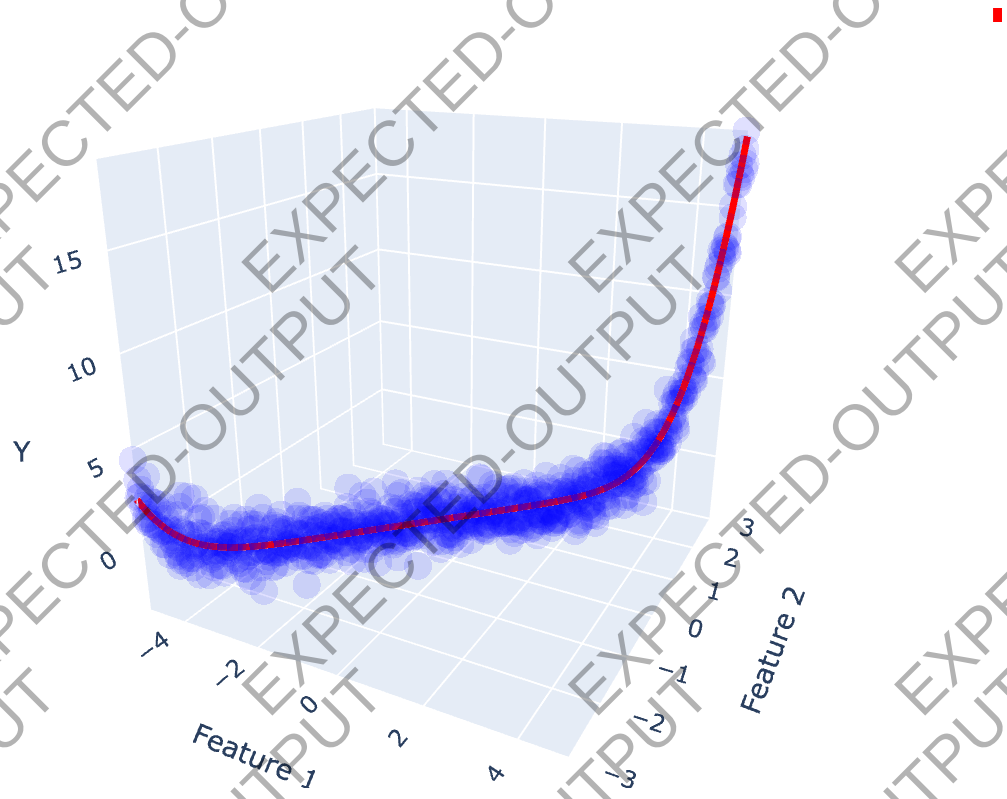
For example, if degree = 2, we will have the polynomial features $[1, a, a^2]$ and $[1, b, b^2]$ for the datapoint $[a, b]$. The cartesian product of these two vectors will be $[1, a, b, ab, a^2, b^2]$. We do not generate a^3 and b^3 since their degree is greater than 2 (specified degree).

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)
```

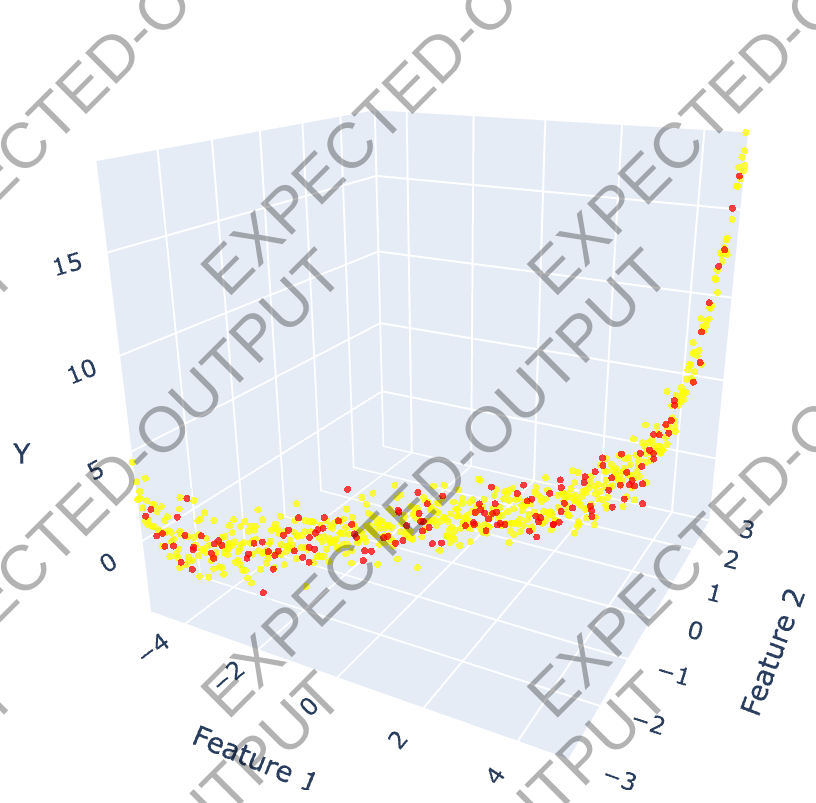
In [ ]:
```
###############################
### DO NOT CHANGE THIS CELL ###
###############################

# 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.

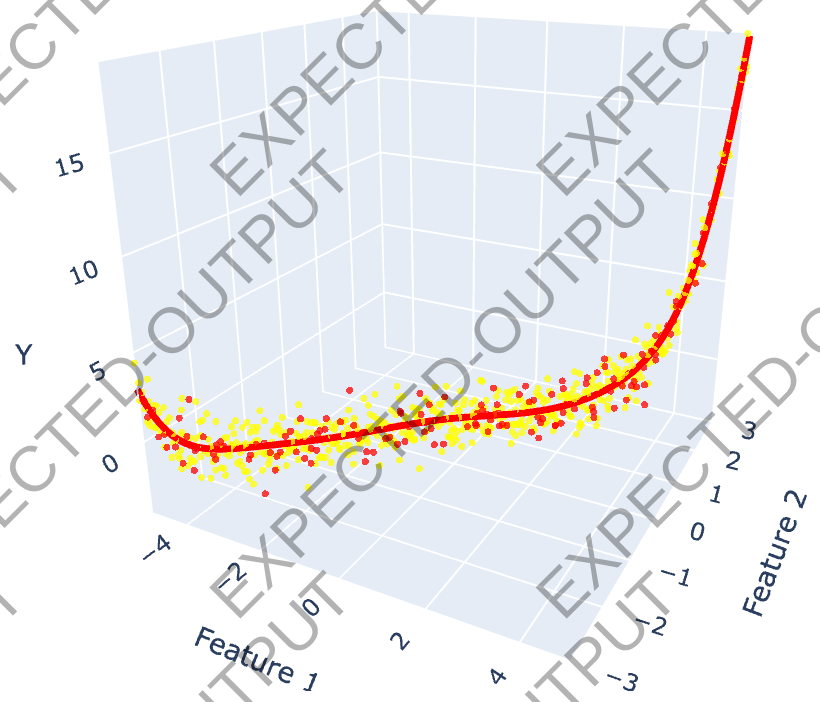
In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

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
```



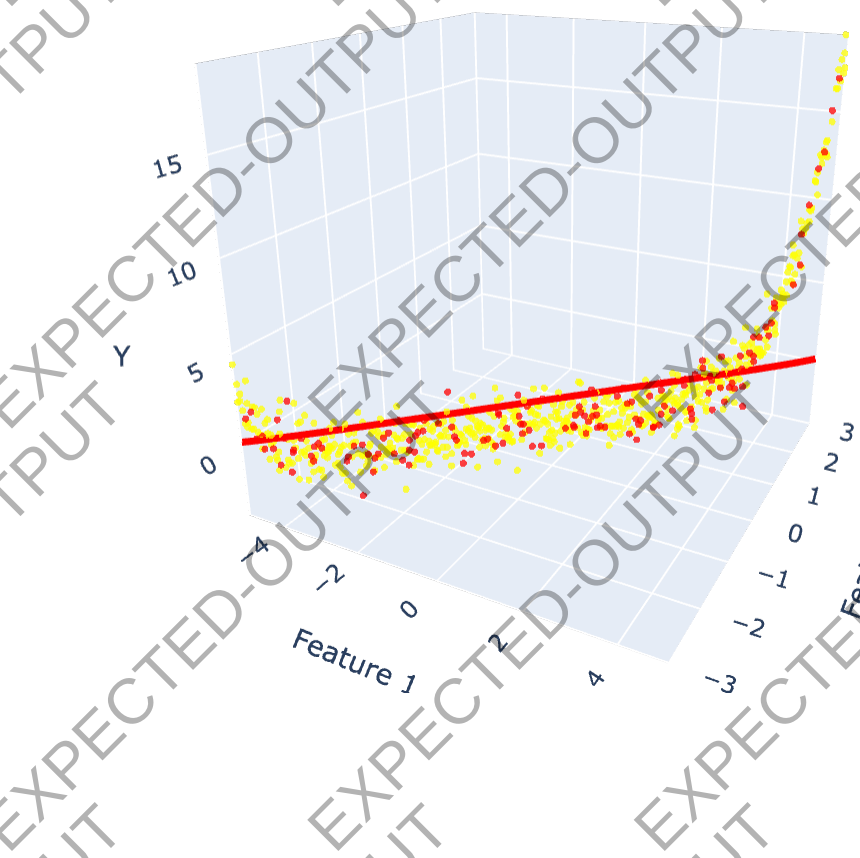
HINT: If your RMSE is off, make sure to follow the instruction given for `linear_fit_closed` in the list of functions to implement above.

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
```



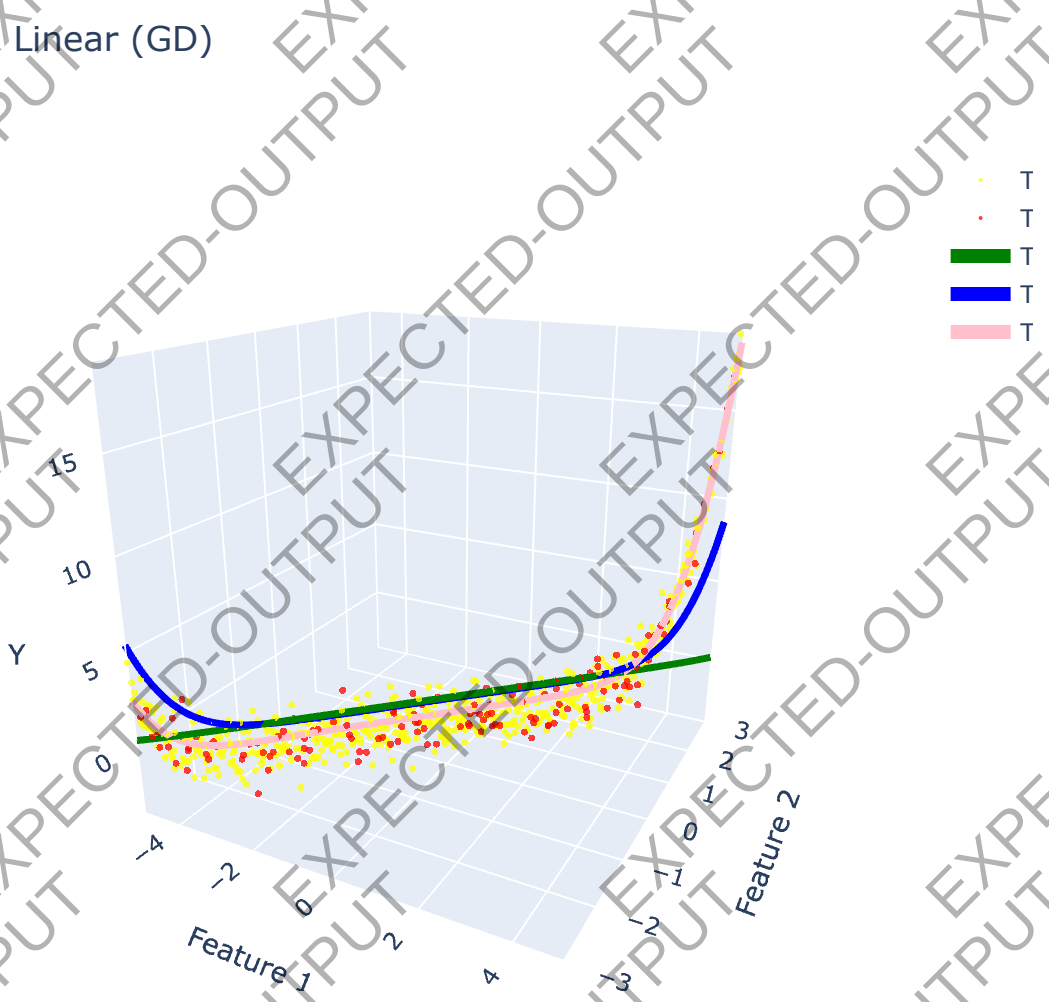
We must tune our epochs and learning_rate. As we tune these parameters our trendline will approach the trendline generated by the linear closed form solution. Observe how we slowly tune (increase) the epochs and learning_rate below to create a better model.

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?

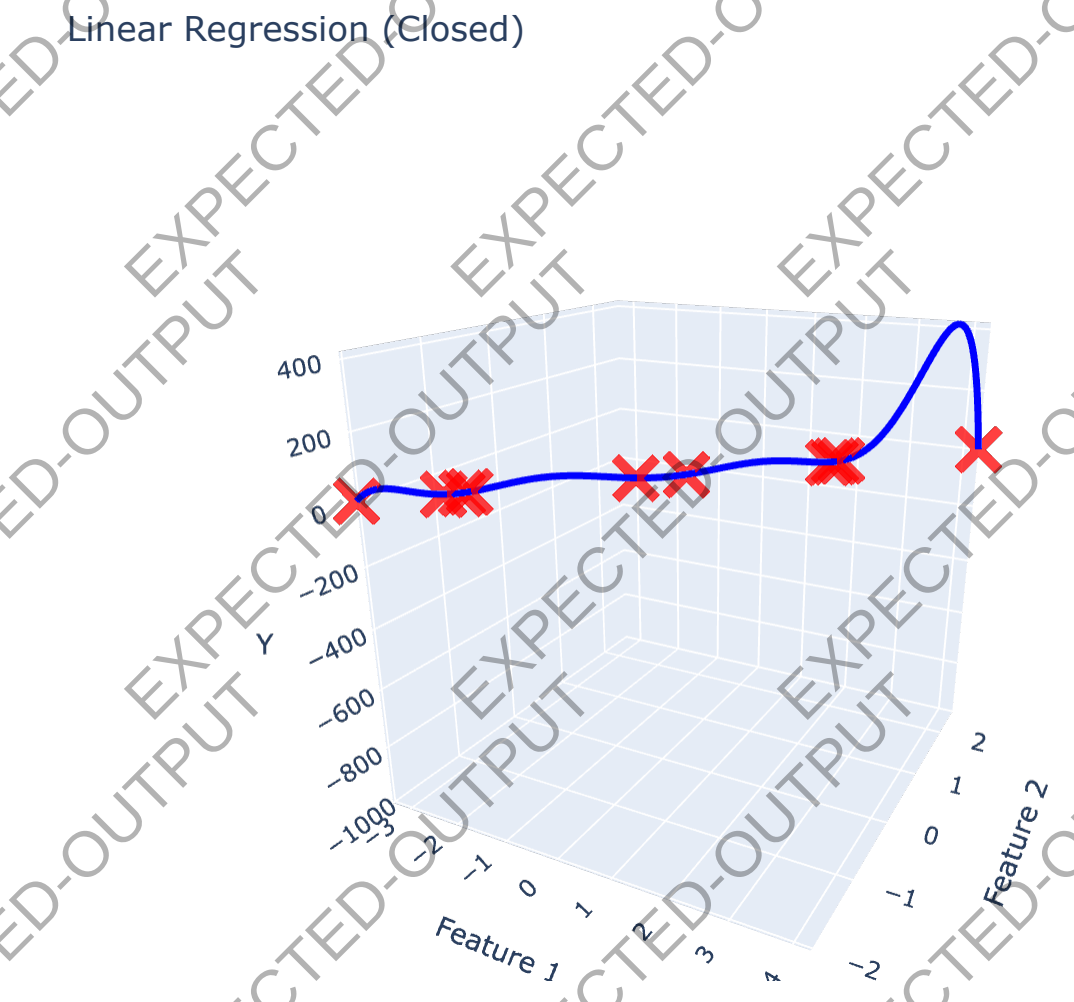
Linear Closed 10 Samples

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
z_min = -1000, z_max = 421.3040762825178
```



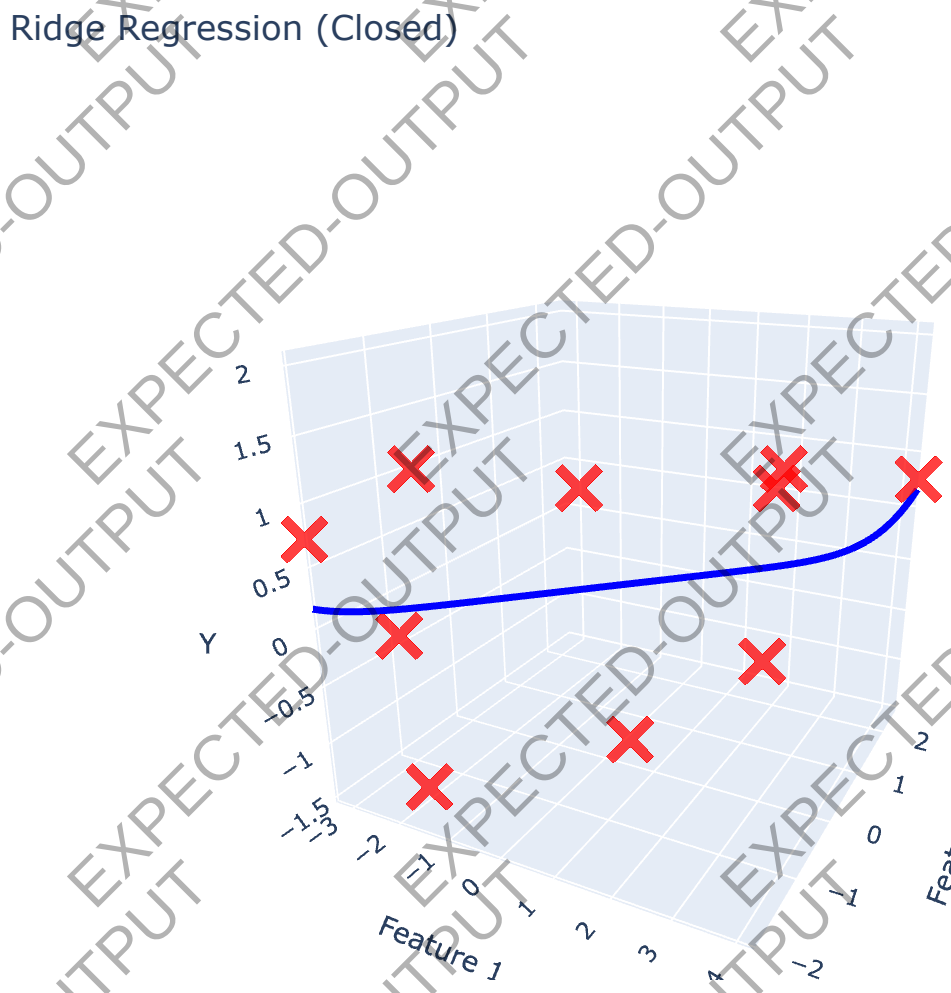
Did you see a worse performance? Let's take a closer look at what we have learned.

3.4 Testing: Ridge Regression [5 pts] [P]

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
z_min = -1.5465558228110778, z_max = 2.100013144934235
```

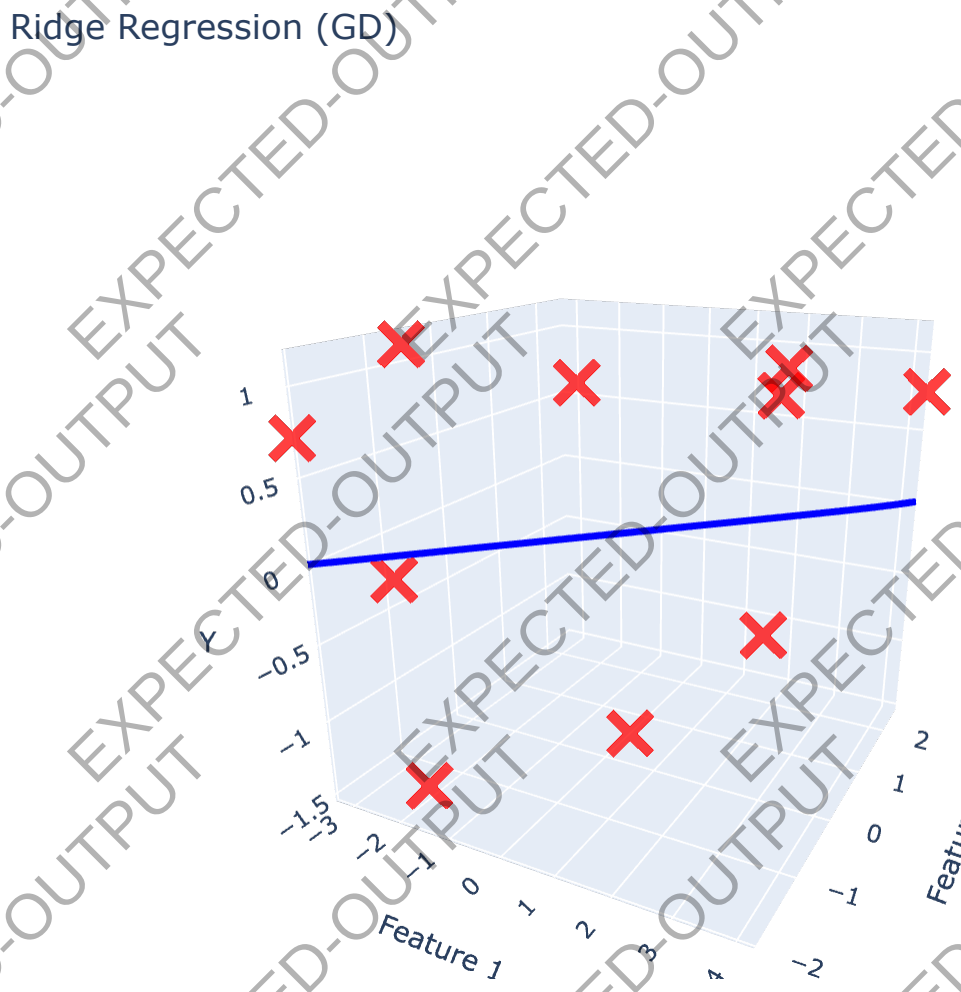


HINT: Make sure to follow the instruction given for `ridge_fit_closed` in the list of functions to implement above.

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
z_min = -1.5465558228110778, z_max = 1.1950583359712759
```

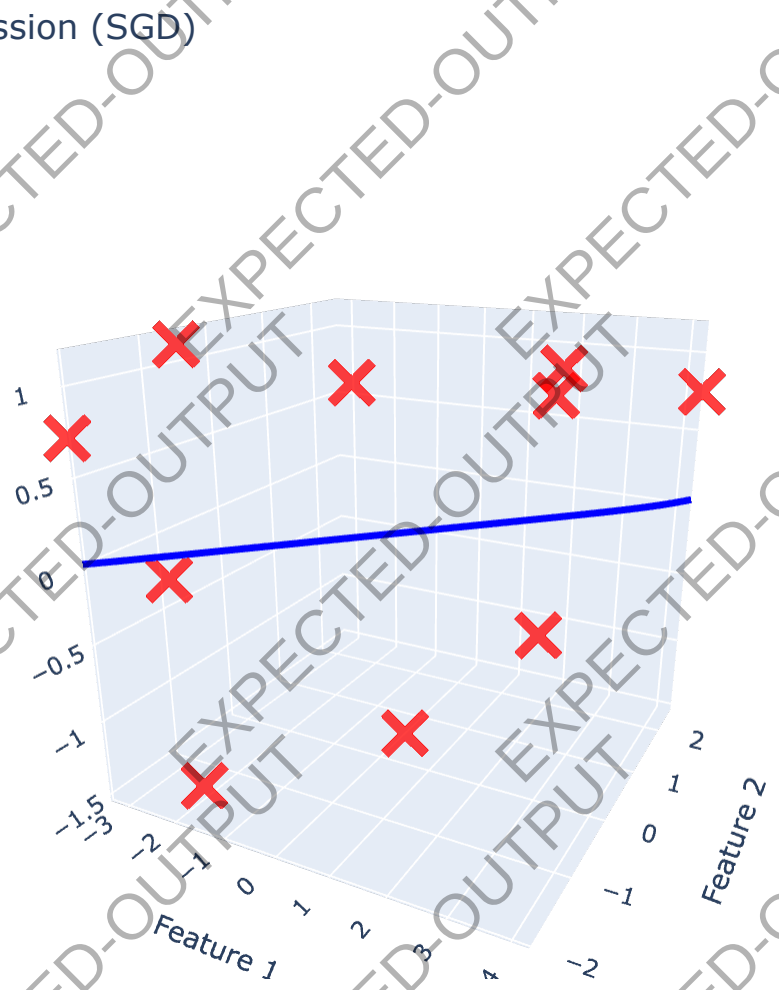


In [ ]:

```
################################
### 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
z_min = -1.5465558228110778, z_max = 1.1950583359712759
```



3.5 Linear vs. Ridge Regression Analysis [4pts] [W]

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.

1. Why does ridge regression achieve a lower RMSE than linear regression on 10 sample points? **(1pts)**

2. 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)**

3. 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)**

Hint: Think about the closed form solution for the weights

1. **Answer** ...

2. **Answer** ...

3. **Answer** ...

3.6 Cross Validation Hyperparameter Search [5 pts] [P]

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
```

3.7 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 $x_n = (x_{n1}, \ldots, x_{nD}) \in \mathrm{R}^D$ and weights $\theta = (\theta_0, \ldots, \theta_D) \in \mathrm{R}^D$. 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:

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

Now, suppose that Gaussian noise $\epsilon_n \in \mathrm{R}^D$ 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. (0.7% Bonus) Show that $y(x_n^{'}, \theta) = y(x_n, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}$

2. (1.6% Bonus) 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:

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

  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:

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

- 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:**

2. **Answer:**

Q4: Naive Bayes and Logistic Regression [39pts] [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.

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

Now, let's apply this to a real-life scenario.

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, 1, 0}, {0, 1, 0, 1, 0}

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

Product Manager applicants display: {0, 1, 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** leadership, data analysis skills **with** product design experience. However, **does not have** coding experience 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:**

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]. Notice that the algorithm can learn historical patterns in the data (e.g., gender, race, socioeconomic status) and be more likely to mark "traditional" applicants (white, male, able-bodied) 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.[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 mitigating 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 (introducing "noise" or subtle alterations to the data in a way that protects individual privacy while allowing the aggregate patterns to remain intact)

D. Relying solely on keyword matching to ensure objectivity

4.1.2 **Answer:**

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:

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

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 $\nabla_\theta \mathrm{L}$ and use gradient descent to optimize the loss function:

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

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

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

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=Non

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
threshold = 0.5
theta = model.fit(X_train, y_train, X_test, y_test, lr, epochs, threshold)
```

```
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]

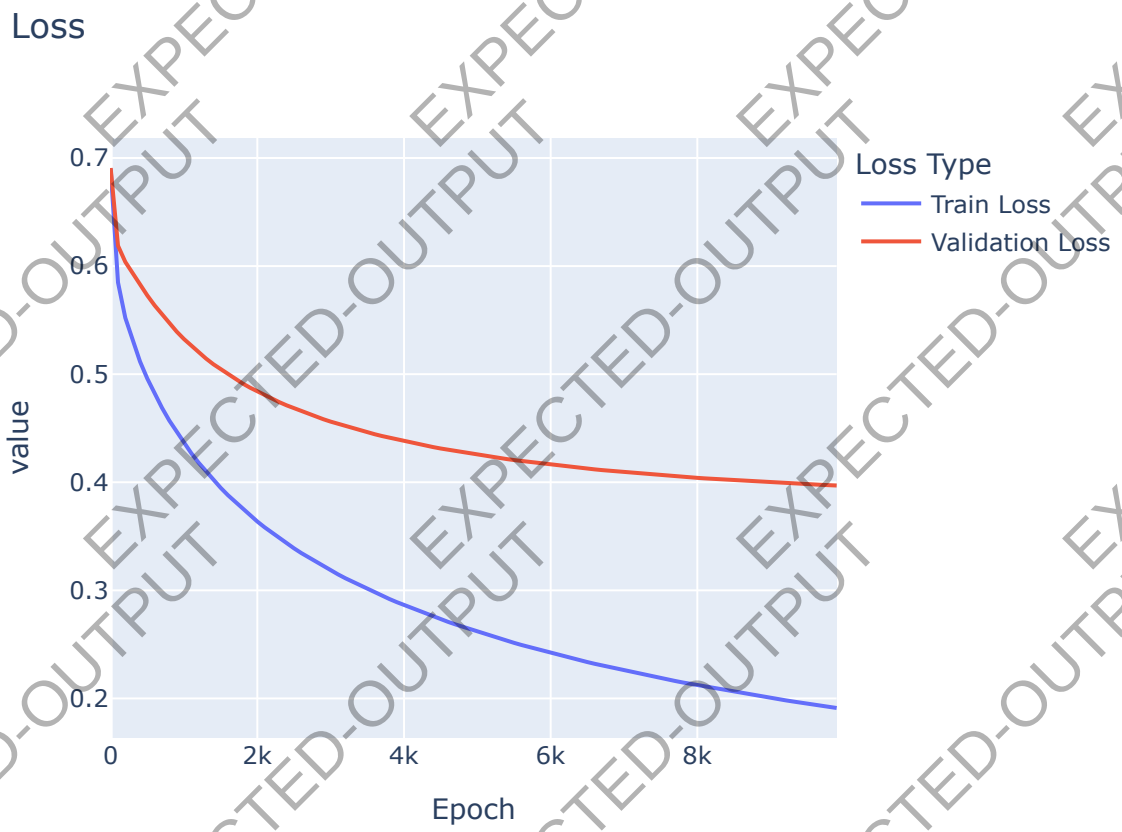
Evaluate the model on the test dataset

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

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

```
Test Dataset Accuracy: 0.807
```

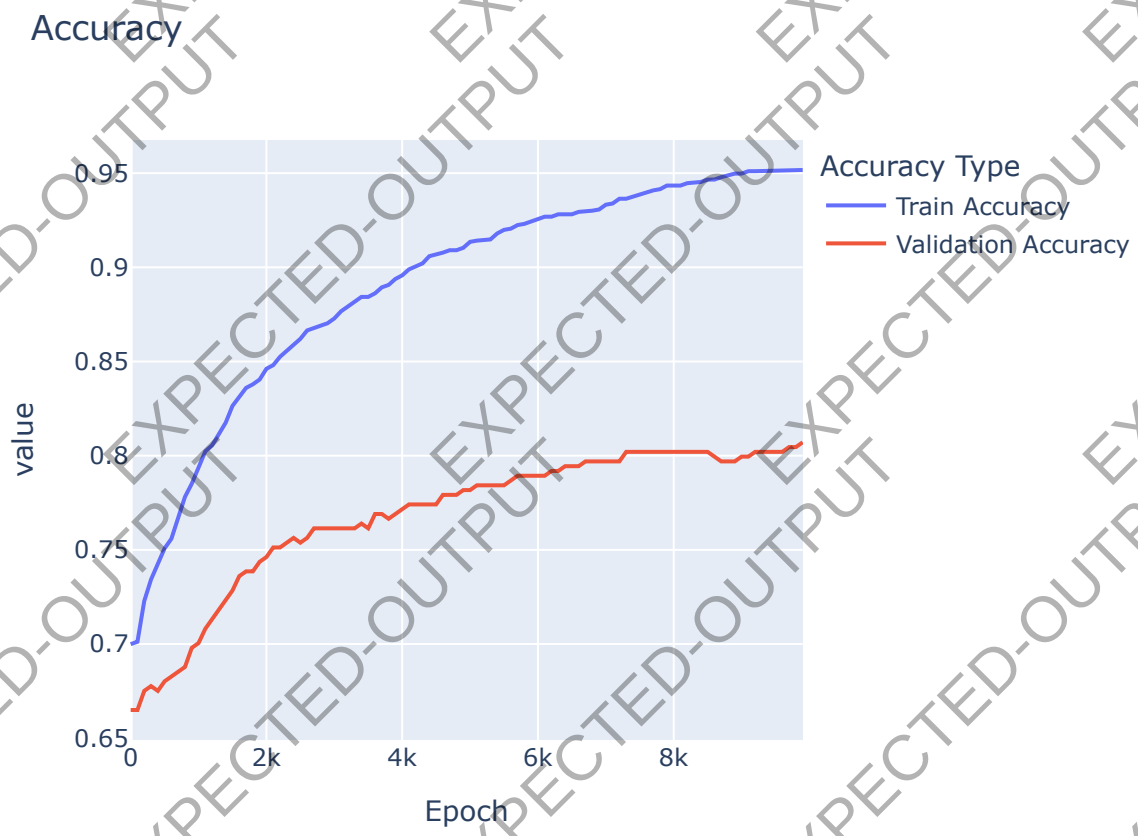
Plotting the loss function on the training data and the test data for every 100th epoch

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, threshold)
    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: Metso said it has won an order worth around 40 mln eur to supply
a kraftliner board machine to China 's Lee & Man Paper Co. .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Xerox and Stora Enso have teamed up to tailor the iGen3 to the s
hort-run , on-demand packaging market .

Predicted Sentiment: negative
Actual Sentiment: positive

Input News: In addition to Russia , we now seek additional growth in Ukraine
.

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: `` Stonesoft sees great promise in the future of IPv6 .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: These financing arrangements will enable the company to ensure ,
in line with its treasury policy , that it has sufficient financial instrume
nts at its disposal for its potential capital requirements .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Via the move , the company aims annual savings of some EUR3m , t
he main part of which are expected to be realized this year .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Ragutis , which is based in Lithuania 's second-largest city Kau
nas , boosted its sales last year 22.3 per cent to 36.4 million liters .

Predicted Sentiment: positive
Actual Sentiment: positive

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

Input News: `` Stonesoft sees great promise in the future of IPv6 .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: However , the suspect stole his burgundy Nissan Altima .
```

```
Predicted Sentiment: positive
Actual Sentiment: negative
```

4.3 Logistic Regression Model Threshold Experiments [2pts] [W]

Recall that the sigmoid function in a logistic regression model outputs a decimal value between 0 and 1. For a classification problem, we need a threshold to determine which outputs are considered as positive and which are considered as negatives.

Instead of the default threshold of 0.5, let's train 4 different logistic regression models each with a different threshold as defined below.

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

thresholds = [0.1, 0.3, 0.7, 0.9]
```

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

# This cell may take about 5 minutes to run

thresholds.sort()
model = LogReg()
lr = 0.05
epochs = 10000
outputs = []
for t in thresholds:
    theta = model.fit(X_train, y_train, X_test, y_test, lr, epochs, t)
    test_loss, test_acc = model.evaluate(X_test, y_test, theta, t)
    outputs.append(
        f"Test Dataset Accuracy: {round(test_acc, 3)} with threshold of {rou
    )
for o in outputs:
    print(o)
```

```
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: 0.7
        val loss:   0.532       val acc:   0.665
Epoch 2000:
        train loss: 0.364       train acc: 0.701
        val loss:   0.484       val acc:   0.668
Epoch 3000:
        train loss: 0.318       train acc: 0.709
        val loss:   0.456       val acc:   0.673
Epoch 4000:
        train loss: 0.286       train acc: 0.715
        val loss:   0.438       val acc:   0.675
Epoch 5000:
        train loss: 0.262       train acc: 0.726
        val loss:   0.425       val acc:   0.678
Epoch 6000:
        train loss: 0.242       train acc: 0.737
        val loss:   0.416       val acc:   0.683
Epoch 7000:
        train loss: 0.226       train acc: 0.746
        val loss:   0.409       val acc:   0.685
Epoch 8000:
        train loss: 0.212       train acc: 0.755
        val loss:   0.404       val acc:   0.693
Epoch 9000:
        train loss: 0.2 train acc: 0.764
        val loss:   0.4 val acc:   0.693
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: 0.716
        val loss:   0.532       val acc:   0.673
Epoch 2000:
        train loss: 0.364       train acc: 0.762
        val loss:   0.484       val acc:   0.69
Epoch 3000:
        train loss: 0.318       train acc: 0.791
        val loss:   0.456       val acc:   0.708
Epoch 4000:
        train loss: 0.286       train acc: 0.815
        val loss:   0.438       val acc:   0.726
Epoch 5000:
        train loss: 0.262       train acc: 0.83
        val loss:   0.425       val acc:   0.728
Epoch 6000:
        train loss: 0.242       train acc: 0.844
        val loss:   0.416       val acc:   0.734
Epoch 7000:
        train loss: 0.226       train acc: 0.855
        val loss:   0.409       val acc:   0.741
Epoch 8000:
        train loss: 0.212       train acc: 0.869
```

```
        val loss:   0.404       val acc:   0.754
Epoch 9000:
        train loss: 0.2 train acc: 0.878
        val loss:   0.4 val acc:   0.759
Epoch 0:
        train loss: 0.69        train acc: 0.3
        val loss:   0.691       val acc:   0.335
Epoch 1000:
        train loss: 0.436       train acc: 0.858
        val loss:   0.532       val acc:   0.764
Epoch 2000:
        train loss: 0.364       train acc: 0.89
        val loss:   0.484       val acc:   0.784
Epoch 3000:
        train loss: 0.318       train acc: 0.906
        val loss:   0.456       val acc:   0.805
Epoch 4000:
        train loss: 0.286       train acc: 0.919
        val loss:   0.438       val acc:   0.81
Epoch 5000:
        train loss: 0.262       train acc: 0.925
        val loss:   0.425       val acc:   0.81
Epoch 6000:
        train loss: 0.242       train acc: 0.932
        val loss:   0.416       val acc:   0.815
Epoch 7000:
        train loss: 0.226       train acc: 0.936
        val loss:   0.409       val acc:   0.817
Epoch 8000:
        train loss: 0.212       train acc: 0.942
        val loss:   0.404       val acc:   0.817
Epoch 9000:
        train loss: 0.2 train acc: 0.945
        val loss:   0.4 val acc:   0.82
Epoch 0:
        train loss: 0.69        train acc: 0.3
        val loss:   0.691       val acc:   0.335
Epoch 1000:
        train loss: 0.436       train acc: 0.318
        val loss:   0.532       val acc:   0.348
Epoch 2000:
        train loss: 0.364       train acc: 0.395
        val loss:   0.484       val acc:   0.419
Epoch 3000:
        train loss: 0.318       train acc: 0.465
        val loss:   0.456       val acc:   0.482
Epoch 4000:
        train loss: 0.286       train acc: 0.537
        val loss:   0.438       val acc:   0.518
Epoch 5000:
        train loss: 0.262       train acc: 0.589
        val loss:   0.425       val acc:   0.561
Epoch 6000:
        train loss: 0.242       train acc: 0.625
        val loss:   0.416       val acc:   0.594
Epoch 7000:
```

```
        train loss: 0.226       train acc: 0.664
        val loss:   0.409       val acc:   0.604
Epoch 8000:
        train loss: 0.212       train acc: 0.689
        val loss:   0.404       val acc:   0.632
Epoch 9000:
        train loss: 0.2 train acc: 0.711
        val loss:   0.4 val acc:   0.655
Test Dataset Accuracy: 0.698 with threshold of 0.1
Test Dataset Accuracy: 0.764 with threshold of 0.3
Test Dataset Accuracy: 0.817 with threshold of 0.7
Test Dataset Accuracy: 0.665 with threshold of 0.9
```

Take a look at the final accuracies of the model with different thresholds in the cell output from above. Which threshold would you pick for the model and why?

4.3 **Answer:**

Q5 Feature Selection Implementation [30 pts] [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.

5.1 Feature Reduction [30pts] [P]

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. In short, 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://www.youtube.com/watch?v=vemZtEM63GY

Forward Selection:

In forward selection, we start with a null model and fit the model with one individual feature at a time. We then select the most significant feature with the lowest p-value.

We continue to do this until we try to select a feature with a p-value >= significance level. This implies that all remaining features are insignificant with p-values < significance level and that no more features should be added.

Steps to implement it:

1. Choose a significance level (provided to you)
2. Start with an empty list of selected features
3. For each feature NOT yet included in the selected features:
    - Fit a simple regression model using the the selected features AND the feature under consideration
    - Record the p-value of the feature under consideration
4. Find the feature with the minimum p-value.
    - If the feature's p-value < significance level, ADD the feature to the selected features and repeat from Step 2.
    - Otherwise, stop and return the selected features

Backward Elimination:

In backward elimination, we start with a full model and then remove the most insignificant feature with the highest p-value. We continue to do this untilwe try to remove a feature with p-value < significance level. This implies that all of the remaining features are significant with pvalues < significance level, and should therefore be kept.

Steps to implement it:

1. Choose a significance level (provided to you)
2. Start with a full list of ALL features as selected features.
3. Fit a simple regression model using the selected features
4. Find the feature with the maximum p-value.
    - If the feature's p-value >= significance level, REMOVE the feature from the selected features and repeat from Step 3.
    - Otherwise, stop and return the selected features.

HINT: Use `sm.OLS` as your regression model (documentation here). Be sure to add bias to your regression model by augmenting your data using the `sm.add_constants` function

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

from utilities.localtests import TestFeatureReduction

unittest_feature_reduction = TestFeatureReduction()
unittest_feature_reduction.test_forward_selection()
unittest_feature_reduction.test_backward_elimination()
```

```
UnitTest passed successfully for "Forward Selection"!
UnitTest passed successfully for "Backward Elimination"!
```

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from feature_reduction import FeatureReduction

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 feature lists are generated, one for ea
forward_selection_feature_list = FeatureReduction.forward_selection(X, y, 0.
backward_selection_feature_list = FeatureReduction.backward_elimination(X, y
```

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

# View selected features and perform linear regression using the selected fe

print("Forward Feature Selection")
FeatureReduction.evaluate_features(X, y, forward_selection_feature_list)
print("Backward Feature Elimination")
FeatureReduction.evaluate_features(X, y, backward_selection_feature_list)
```

```
Forward Feature Selection
Significant Features: ['worst concave points', 'worst radius', 'worst textur
e', 'worst area', 'smoothness error', 'worst symmetry', 'compactness error',
'radius error', 'worst fractal dimension', 'mean compactness', 'mean concave
points', 'worst concavity', 'concavity error', 'area error']
RMSE: 0.23821726582785402

Backward Feature Elimination
Significant Features: ['mean radius', 'mean compactness', 'mean concave poin
ts', 'radius error', 'smoothness error', 'concavity error', 'concave points
error', 'worst radius', 'worst texture', 'worst area', 'worst concavity', 'w
orst symmetry', 'worst fractal dimension']
RMSE: 0.2374589600443534
```

Q6: Imbalanced Classes in Classification Tasks [5.6%

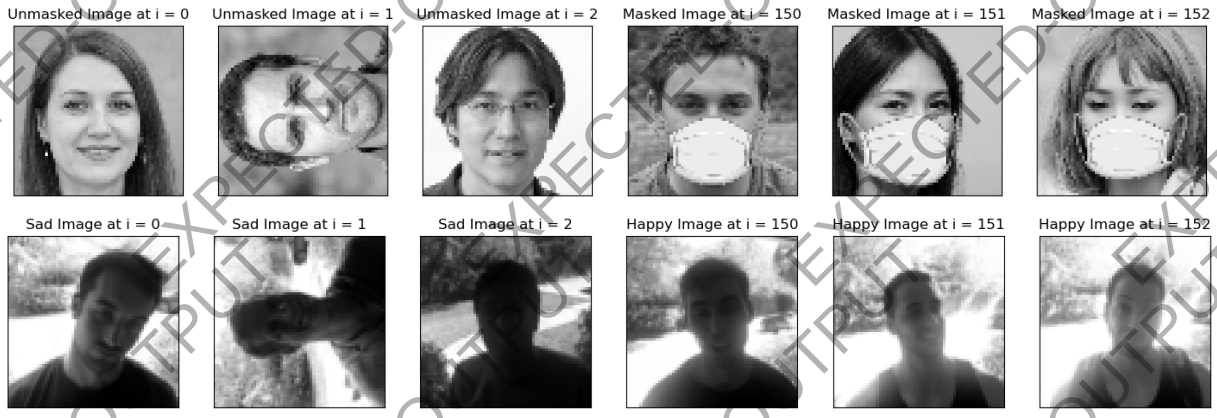
Bonus For All] [P]

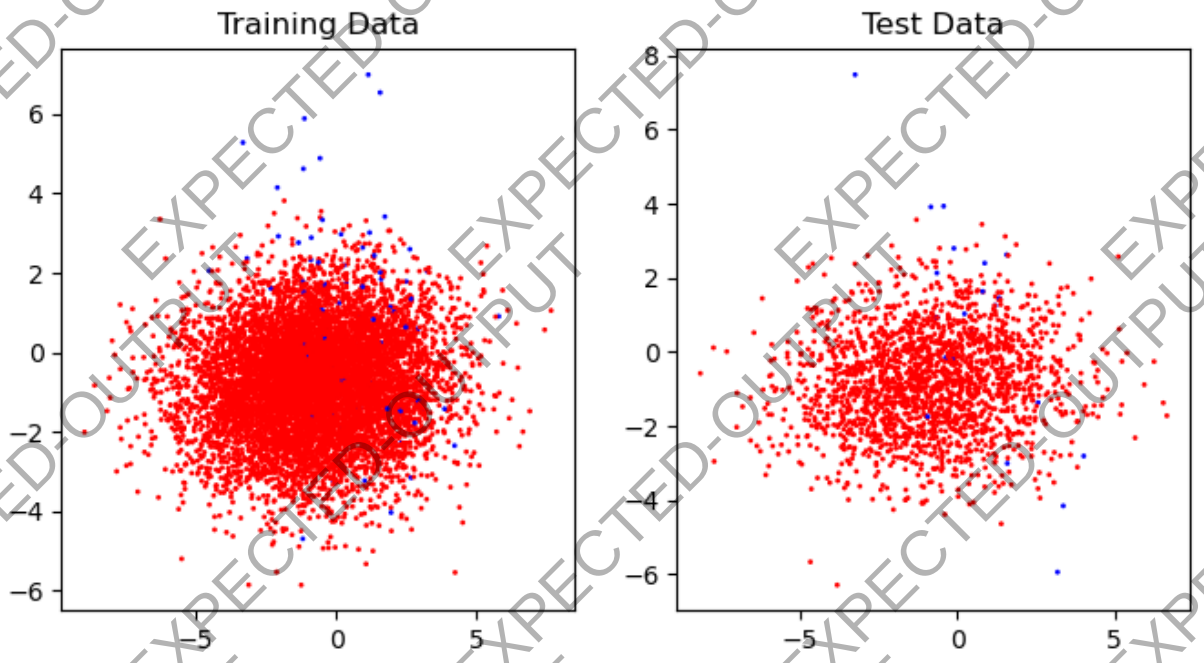
In many datasets, the representation of classes in the training data is unequal. For example, most transactions in a banking system are legitimate, most images of manufactured parts show no visual defect, most email attachments are entirely benign. However, when you have highly imbalanced data, your model may converge to a highly biased estimator, classifying most inputs to the majority class. This is a valid solution, after all, the model is simplying converging based on the a priori distribution of classes in the training data, but we still want our model to have accurate prediction on the minority class.

To illustrate this point, take a look at the following 2D artificial dataset with a high class imbalance, and the results of training a classifier on it.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
SEED = 1
imb = 0.99

# Generate an artifical 10D classification problem with a high degree of cla
X, y = make_classification(
    n_samples=10000,
    weights=[imb],
    n_features=10,
    n_informative=10,
    n_redundant=0,
    n_repeated=0,
    n_classes=2,
    n_clusters_per_class=1,
    flip_y=0,
    class_sep=0.8,
    random_state=SEED,
)
# Splitting the data into training and testing.
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=SEED
)

# Plot both the training and testing data.
a, b = 0, 5
fig, axs = plt.subplots(1, 2, figsize=(8, 4))
axs[0].scatter(
    X_train[:, a],
    X_train[:, b],
    c=y_train,
    cmap=matplotlib.colors.ListedColormap(["red", "blue"]),
    s=1,
)
axs[0].title.set_text(f"Training Data")
axs[1].scatter(
    X_test[:, a],
```

```
    X_test[:, b],
    c=y_test,
    cmap=matplotlib.colors.ListedColormap(["red", "blue"]),
    s=1,
)
axs[1].title.set_text(f"Test Data")
plt.show()
```





This data is higher than 2 dimensional, but we can visualize any dim-2 subspace, just for understanding purposes. If you want, you can tinker with `a` and `b` to choose different subspaces, though most won't be particularly informative. Note that while the data doesn't look even vaguely separable in 2D, distances stack up as you add dimensions, so in 10D, this data is separable.

Let's run a basic classifier on the data!

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

classifier = SVC(random_state=SEED)  # sklearn's Support Vector Classifier
```

```
classifier.fit(X_train, y_train)
y_predicted = classifier.predict(X_test)

# Display the accuracy of this prediction.
print(f"Accuracy: {100*accuracy_score(y_test, y_predicted)}%")
```

```
Accuracy: 99.5%
```

This classifier achieved very good accuracy, but as we'll see, accuracy is not always a perfect measure for determining the goodness of a model, since it can hide a bias.

6.1 A More Comprehensive Measure [1.75% Bonus] [P]

To get a better view of our model's accuracy, we need to generate a confusion matrix. Each data point in the test set has a true class value and a predicted class value. A confusion matrix counts the instances of every possible combination of test and prediction values.

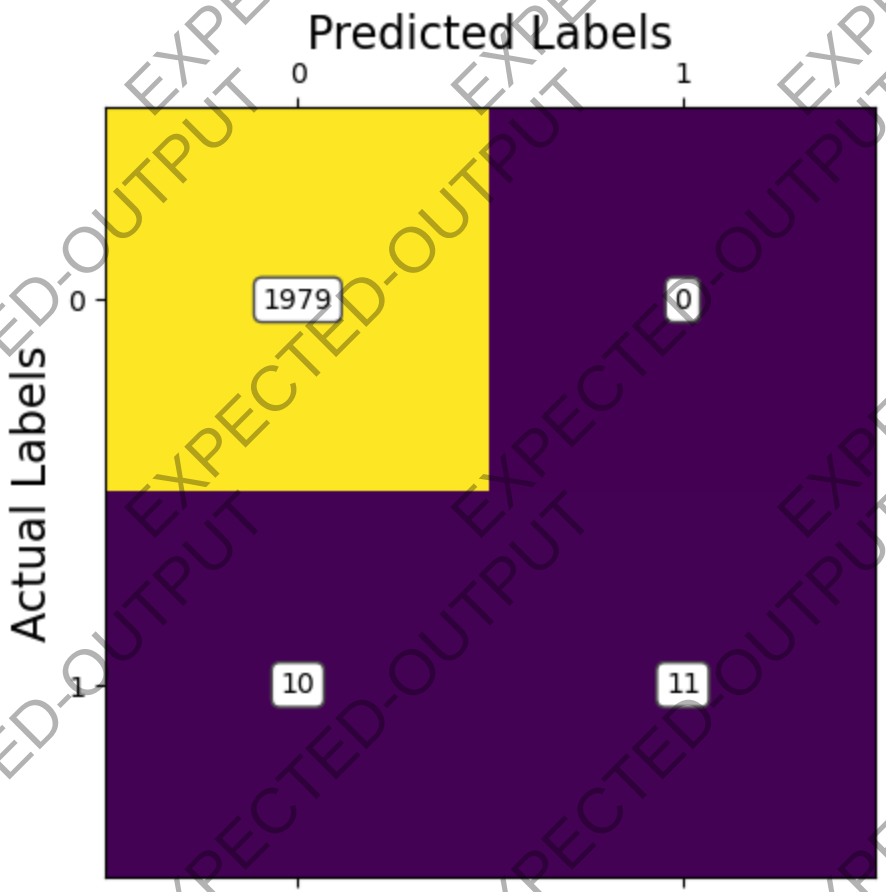
In the **smote.py** file, complete the following:

1. **generate_confusion_matrix**: Given the true test labels and the predicted labels from a model, generate a confusion matrix. C[i, j] should denote the number of instances where a sample from class i was predicted to be in class j. Even though our example is binary classification, your code should work for an arbitrary number of classes.

In [ ]:
```
from smote import SMOTE
from utilities.localtests import TestSMOTE

unittest_sm = TestSMOTE()

sm = SMOTE()

unittest_sm.test_simple_confusion_matrix()
unittest_sm.test_complex_confusion_matrix()
```

```
UnitTest passed successfully for "simple confusion matrix"!
UnitTest passed successfully for "multiclass confusion matrix"!
```

In [ ]:
```
# Display the confusion matrix of the prediction we made.
from smote import confusion_matrix_vis

confusion_matrix_vis(sm.generate_confusion_matrix(y_test, y_predicted))
```



Just from a first look, this relatively high accuracy is clearly not a good measure of the model's performance, as the model is highly biased. This (albeit naive) model is very accurate when given a point from class 0. However, when given a point from class 1, it's much less inaccurate.

Depending on the application, e.g., cancer screening or facial recognition, you may want to prioritize minimizing false negatives or false positives depending on your application's objectives, in which case you might have to accept a low degree of accuracy for some classes, but in this exploration, we just want a balanced performance across our labels. We need a measure of test performance that not only conveys the accuracy of the model, but is robust against bias.

F1 score (or F-Measure) is one metric that seeks to measure the performance of a model across all classes. F1 score can be calculated on any class, and here, we'll calculate it every class.

In the **smote.py** file, complete the following:

2. **f1_scores**: Given the confusion matrix from a classification, calculate the F1 scores for each class. To calculate the F1 score, take the harmonic mean of the precision and recall for that class's predictions. Even though our example is binary classification, your code should work for an arbitrary number of classes.

In [ ]:
```
unittest_sm.test_simple_f1s()
unittest_sm.test_complex_f1s()
```

```
UnitTest passed successfully for "F1 scores for perfect performance"!
UnitTest passed successfully for "F1 scores in general"!
```

Let's take a look at our new metric.

In [ ]:
```
conf = sm.generate_confusion_matrix(y_test, y_predicted)
print(f"Accuracy: {100*accuracy_score(y_test, y_predicted)}%")
print(f"F1 Scores: {np.round(sm.f1_scores(conf), 3)}")
```

```
Accuracy: 99.5%
F1 Scores: [0.997 0.688]
```

F1 takes the range $[0, 1]$, with 1 meaning perfect precision and recall, so we obviously have some room for improvement now, since the F1 score for the majority class is significantly higher than that of the minority class.

Note that the work you did corresponded to 1 value per class. In literature, you may see some variations on F1 score. First of all, the 1 in F1 is actually a parameter. There are an infinite number of F_β scores, though F_1 is most common. Second, you may see a single F1 score on binary tasks where the performance on only 1 class is important for the downstream system. And finally, you may see these values averaged across the classes. Macro-F1 refers to the arithmetic mean of all F1 scores. Weighted-F1 refers to the arithmetic mean weighted by class size.

For our example, we'll keep both F1 values so we can see the relative change in performance between the classes.

To improve the performance bias, many models employ a technique called class weighting. Essentially, when the model fit or iterative training is performed, the loss/gain/effect (varies from model to model) caused by each point in the training set can be weighted by the inverse of the proportion of that point's class. Applied to what you just implemented (logistic regression), that might look like this:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{N}\sum_{i=1}^{N} x_i^\top \left(h_\theta(x_i) - y_i\right)$$

$$\downarrow$$

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{N}\sum_{i=1}^{N} x_i^\top \left(h_\theta(x_i) - y_i\right) \cdot \frac{\mathbf{N}}{|\{\mathbf{y} \in \mathbf{Y} | \mathbf{y} = \mathbf{y_i}\}|} \quad \left(\frac{\text{total \# of points}}{\text{\# of points in } y_i\text{'s class}}\right)$$

Class weighting is generally applicable, and is often the preferred solution, but on some models it's inaccessible, unwieldy, or produces undesirable results. There is an alternative solution: instead of changing the model, eliminate the problem. If our training data had balanced classes, we could just run the model. This is the idea behind

oversampling. Oversampling refers to the practice of populating the input space with points from the input space itself.

Though, if you just sampled minority classes with replacement, you would get duplicates of the same point in the training data. This can genuinely help (since it approximates the class weight technique), but there is a more sophisticated solution: the Synthetic Minority Oversampling TEchnique (SMOTE). You can read the original paper on the arxiv for your own edification, but we'll be implementing a slightly different version, so the details in this Notebook and the function docstrings are sufficient to complete this HW.

6.2 SMOTE [3.85% Bonus] [P]

Instead of directly sampling the points from the minority class to bring it up to size, SMOTE samples a training point from the minority, samples another training point from the minority class that is "within a neighborhood around the first point," then randomly linearly interpolates between them to generate a new "synthetic" point lying on the line segment drawn between those two points. In this section, we're going to focus in on our binary classification problem, so we will only be oversampling the points from the one minority class. Additionally, our algorithm will only oversample to equality (the point at which the two classes are of equal size). Though, in practice, the amount you oversample becomes a hyperparameter to your model, which you can sweep with cross-validation.

In the **smote.py** file, complete the following:

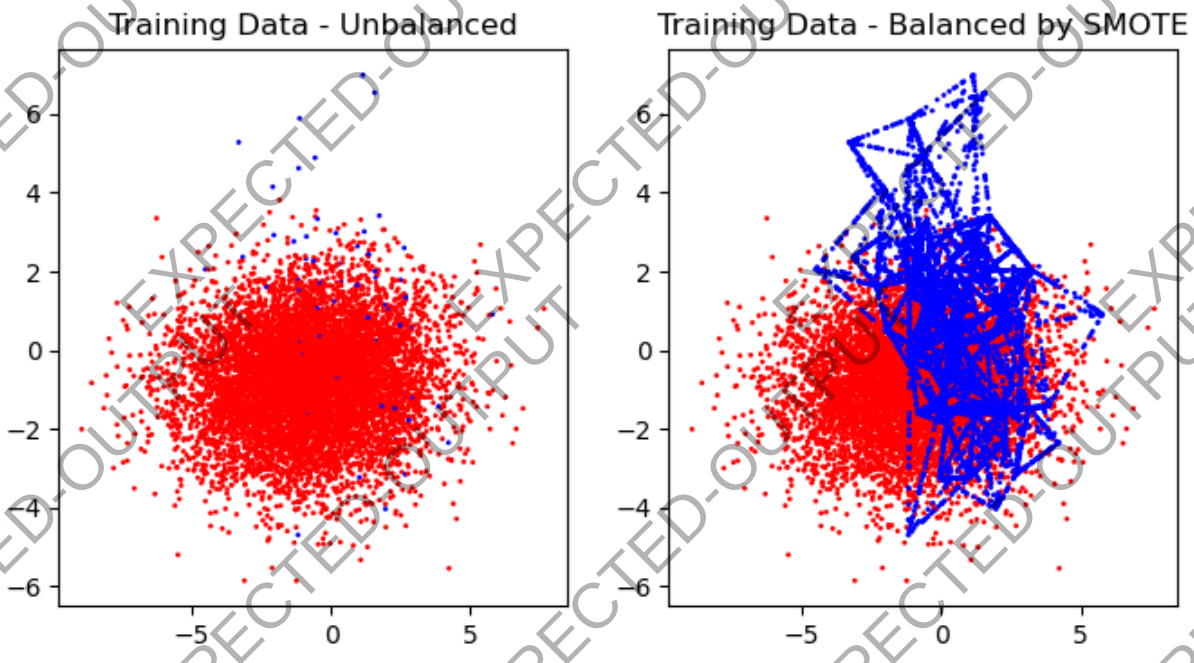
1. **interpolate**: Given a start point, an end point, and an interpolation coefficient, return a linearly interpolated point.
2. **k_nearest_neighbors**: Given some set of points (N, D) and a parameter k, generate an (N, k) array of indices such that `output[i]` contains the k indices corresponding to the k nearest neighbors of point i.
3. **smote**: Given some data `X` (|maj|+|min|, D) and their binary labels `y` (|maj|+|min|,), generate |maj|-|min| new synthetic points from the minority class and return only those new synthetic points.

In [ ]:
```
unittest_sm.test_interpolate()
unittest_sm.test_knn()
unittest_sm.test_smote()
```

```
UnitTest passed successfully for "interpolation"!
UnitTest passed successfully for "k nearest neighbors"!
UnitTest passed successfully for "SMOTE"!
```

Let's apply SMOTE to our dataset and see the downstream effect on our classification task!

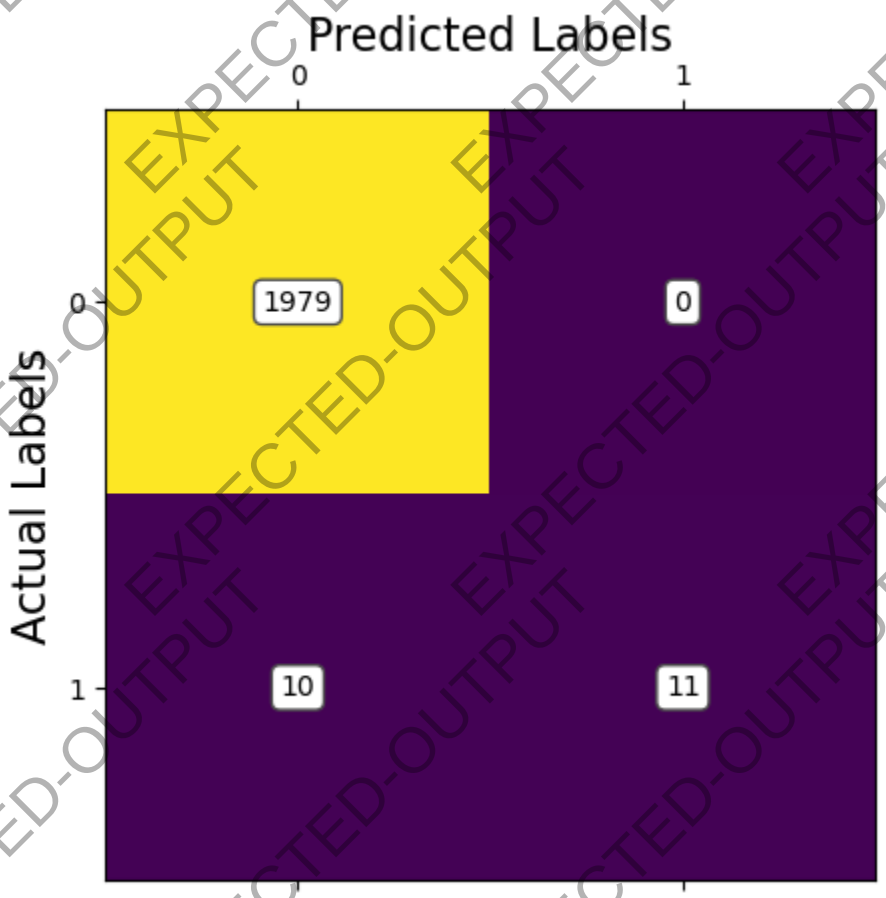
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# Run SMOTE!
X_train_synth, y_train_synth = sm.smote(X_train, y_train, k=5, inter_coeff_r
# Combine synthetic data with original data.
X_train_balanced = np.vstack((X_train, X_train_synth))
y_train_balanced = np.hstack((y_train, y_train_synth))

# Visualize
a, b = 0, 5
fig, axs = plt.subplots(1, 2, figsize=(8, 4))
axs[0].scatter(
    X_train[:, a],
    X_train[:, b],
    c=y_train,
    cmap=matplotlib.colors.ListedColormap(["red", "blue"]),
    s=1,
)
axs[0].title.set_text("Training Data - Unbalanced")
axs[1].scatter(
    X_train_balanced[:, a],
    X_train_balanced[:, b],
    c=y_train_balanced,
    cmap=matplotlib.colors.ListedColormap(["red", "blue"]),
    s=1,
)
axs[1].title.set_text("Training Data - Balanced by SMOTE")
plt.show()
```



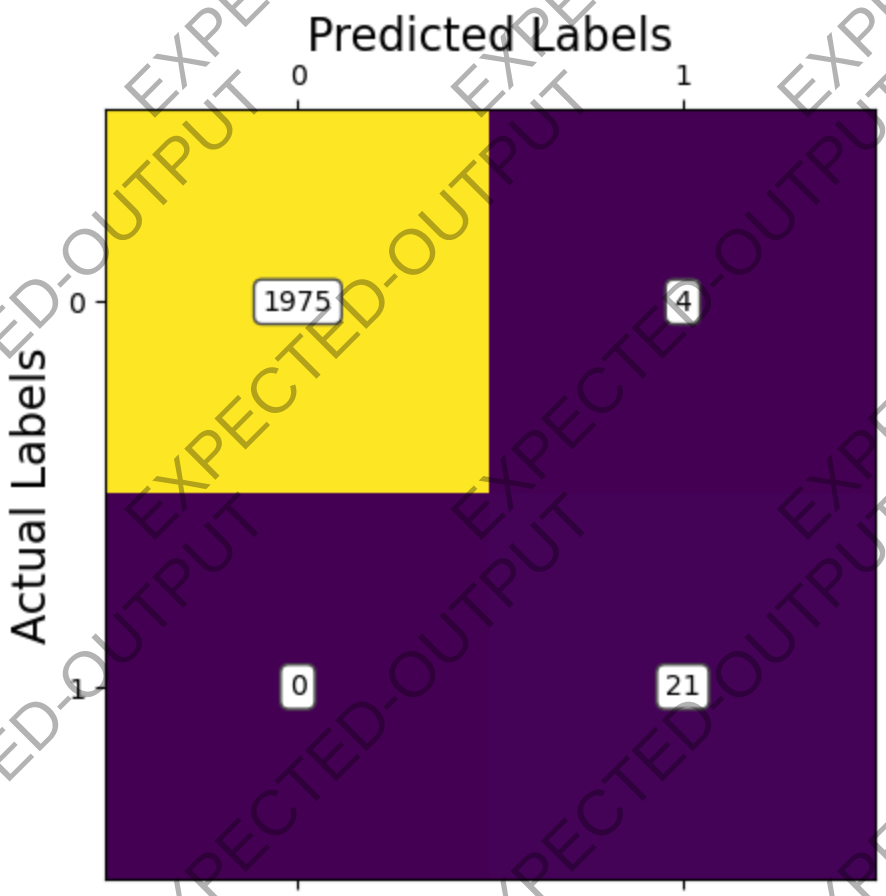
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
```

```
# Original
classifier.fit(X_train, y_train)
y_predicted1 = classifier.predict(X_test)

conf1 = sm.generate_confusion_matrix(y_test, y_predicted1)
print("Original performance:")
confusion_matrix_vis(conf1)
print(f"Accuracy: {100*accuracy_score(y_test, y_predicted1)}%")
print(f"F1 Scores: {sm.f1_scores(conf1)}")

# Balanced
classifier.fit(X_train_balanced, y_train_balanced)
y_predicted2 = classifier.predict(X_test)

print("\n\n\nPerformance after SMOTE:")
conf2 = sm.generate_confusion_matrix(y_test, y_predicted2)
confusion_matrix_vis(conf2)
print(f"SMOTEd Accuracy: {100*accuracy_score(y_test, y_predicted2)}%")
print(f"SMOTEd F1 Scores: {sm.f1_scores(conf2)}")

# If you're getting errors running this cell, check your implementation, the
```

```
Original performance:
```




```
Accuracy: 99.5%
F1 Scores: [0.99747984 0.6875   ]


Performance after SMOTE:
```




```
SMOTEd Accuracy: 99.8%
SMOTEd F1 Scores: [0.99898837 0.91304348]
```

With our new synthetic data (if done correctly), the model is now significantly more performant on the test points for the minority class! The F1 scores aren't equal, but you could probably continue upping the number of synthetic minority points created until that equality is reached. Again, the amount of points we choose to generate with SMOTE is a hyperparameter.

Of course, SMOTE has limitations.

- Since SMOTE only works by interpolating on a line segment, SMOTE generates points within the convex hull of the input data. For classes with non-convex or multi-modal spatial distributions, SMOTE can actually ruin the quality of the class's representation in the input space for certain choices of k.
- For other types of data, like images, linear interpolation in general will not produce any meaningful data. You may have to use some semantic or generative combination instead.
- Additionally, SMOTE is very susceptible to outliers. Without undergoing some filtering, SMOTE will generate unreasonable points when interpolating with an outlier.
- Often, improving performance on the minority class can decrease performance on the majority class. Thus, if you want to reduce bias, sometimes you have to accept

lower accuracy as well.

In general, just consider this another tool in your toolbox. It won't work on everything, but it will work exceptionally well on a few.

The class imbalance problem shows up all across ML and when creating human-facing ML models, the class imbalance problem has an additional ethical element tied on; since a lot of data collection is heavily biased to people of higher socio-economic status in a society, the resulting models trained from it may have higher performance quality for high SES individuals. Moreover, this isn't the only kind of data collection bias that currently exists. In general, a model that is performant to only a particular class of people will create an inherently unjust system that has the potential to relatively worsen the lives of those who are less represented in the training data, especially since some users will use your model as though it is foolproof.

This class imbalance problem will also probably show up in your project (if you're doing classification)! Remember that accuracy isn't everything, and analysis of your system's bias is also important.

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:

No description has been provided for this image

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 SVD Recommender [2.1% Bonus for All] [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.
- **predict**: Predict the next 3 movies (sorted by high to low rating) that the user would be most interested in watching among the ones above.

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)

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`

7.1.1 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_ind
        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 --> 1.0225266494179552
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 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]))
```