Fall 2025 CS4641/CS7641 Homework 3

Instructor: Dr. Mahdi Roozbahani, Dr. Nimisha Roy

Deadline: Friday, November 14th, 11:59 pm ET

- No unapproved extension of the deadline is allowed. For late submissions, please refer to the course website.
- Discussion is encouraged on Ed as part of the Q/A. We encourage whiteboard-level discussions about the homework. **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, and you must not collaborate with anyone or share your HW content except the ML instructional team.
- Working with a generative-AI platform *may constitute plagiarism*. In line with the Joyner Heuristic being used by many classes at GT, you should treat collaboration with generative-AI as collaboration with a knowledgeable peer. **Sharing the question or your work verbatim with an AI agent so as to generate an answer to the question is considered academic dishonesty and will be treated the same as any other incident of academic dishonesty.** If you find yourself turning to generative-AI for help answering a question, we suggest that you instead make use of Ed or TA office hours.
- Even using generative-AI for formatting your answers is *not permitted*. **While we understand that LaTeX has a learning curve, you may not use any generative-AI platform to improve your writing or LaTeX formatting.** These tools inherently produce output that may include a partial or complete solution to the question, including corrections to work you give it, even if purely prompted for syntactic use. Additionally, you have no custody over the data you give these platforms, which may leak or be used to inform subsequent training. Thus, while you are more than welcome to ask it about LaTeX commands and formatting tips, sharing the question or your answer to a question verbatim with an AI agent, even purely for syntactic use, is considered academic dishonesty and will be treated the same as any other incident of academic dishonesty. If you find yourself turning to generative-AI for help rewording your work to improve the language therein, remember that many of these questions will not be graded on language, but the content of your work. If you wish to improve it nonetheless, you can make use of the Georgia Tech Communications Lab or TA office hours. If you find yourself turning to generative-AI for help reformatting your LaTeX, we suggest that you instead use the resources in the instructions below or TA office hours. Ed is also an appropriate place to ask about LaTeX formatting, so long as your post doesn't reveal answers to a question (or make a private post if your question necessitates revealing answers).
- *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 the case to OSI, which may, unfortunately, lead to a very harsh outcome, pending review. Consequences can be severe, including 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. We will be using Gradescope for submission and grading of assignments.
- **Unless a question explicitly states that no work is required to be shown, you must provide an explanation, justification, or calculation for your answer.** Basic arithmetic can be combined (it does not need to each have its own step); your work should be at a level of detail that a TA can follow it.
- **For the "Assignment 3 - Non-programming" turn-in of this assignment, you will need to submit to Gradescope a PDF copy of your Jupyter Notebook with the cells ran.** Please refer to the Deliverables and Point Distribution section for an outline of the non-programming questions.
- 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 sum_{i=0} x_i. You can directly type LaTeX equations into markdown cells using $...$ for inline and $$...$$ for equations. Using LaTeX to format your answers is highly encouraged.
- When submitting your assignment on Gradescope, **you are required to correctly map pages of your PDF to each question/subquestion to reflect where your solutions appear in your PDF. Improperly mapped questions will receive a 0.** You are permitted to submit a regrade request in this event; however, the review of such a request is at the

sole discretion of the instructional staff and is not likely to be accepted.

- **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, up to and including full credit deducted.** Again, you are permitted to submit a regrade request in this event; however, the review of such a request is at the sole discretion of the instructional staff and is not likely to be accepted.
- 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.

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: Audio Compression (with SVD) [25pts]

Deliverables: audiocompression.py and Written portion

- **1.1 Audio Compression** [20pts] - **[Programming]**
  - `svd` [4pts]
  - `compress` [4pts]
  - `rebuild_svd` [4pts]
  - `compression_ratio` [4pts]
  - `recovered_variance_proportion` [4pts]
- **1.2 Single Channel Audio** [0pts]
- **1.3 Stereo Audio** [5pts] - **[Written]**

Q2: Eigenfaces (with PCA) [20pts]

Deliverables: eigenfaces.py

- **2.1 Eigenfaces** [20pts] - **[Programming]**

    - `svd` [10pts]
    - `compute_eigenfaces` [5pts]
    - `project` [5pts]
- **2.2 Eigenface Demonstration** [0pts]

Q3: SVD Recommender [2.1% Bonus for All]

Deliverables: svd_recommender.py, and Written portion

- **3.1 SVD Recommender for Movies** [1.4%] - **[Programming]**
    - `recommender_svd` [0.70%]
    - `predict` [0.70%]
- **3.2 Comparison: SVD vs. PCA** [0.70%] - **[Written]**

Q4: Polynomial Regression and Regularization [65pts + 30pts Grad/6% Bonus for Undergrad + 2.3% Bonus for All]

Deliverables: regression.py and Written portion

- **4.1 About RMSE** [5pts] - **[Written]**
- **4.2 Regression Implementation** [30pts + 30pts Grad/6% Bonus for Undergrad] - **[Programming]**
    - `rmse` [5pts]
    - `construct_polynomial_feats` (1D and 2D) [5pts]
    - `predict` [5pts]
    - Linear Fit
        - `linear_fit_closed` [5pts]
        - `linear_fit_GD` [5pts Grad/1.0% Bonus for Undergrad]
        - `linear_fit_SGD` [5pts Grad/1.0% Bonus for Undergrad]
        - `linear_fit_MBGD` [5pts Grad/1.0% Bonus for Undergrad]
    - Ridge Fit
        - `ridge_fit_closed` [5pts]
        - `ridge_fit_GD` [5pts Grad/1.0% Bonus for Undergrad]
        - `ridge_fit_SGD` [5pts Grad/1.0% Bonus for Undergrad]
        - `ridge_fit_MBGD` [5pts Grad/1.0% Bonus for Undergrad]
    - Cross Validation (`ridge_cross_validation`) [5pts]
- **4.3 Testing: General Functions and Linear Regression** [10pts] - **[Programming]** **[Written]**
    - Performance on data [5pts] **[P]**
    - Analysis of closed-form vs. GD [5pts] **[W]**
- **4.4 Testing: Ridge Regression** [5pts] - **[Programming]**
- **4.5 The Power of Ridge Regression** [10pts] - **[Written]**
- **4.6 Cross Validation Hyperparameter Search** [5pts] - **[Programming]**
- **4.7 Noisy Input Samples in Linear Regression** [2.3% Bonus for All] - **[Written]**

Q5: Naive Bayes and Logistic Regression [40pts]

Deliverables: logistic_regression.py and Written portion

- **5.1 Profile Screening using Naive Bayes** [7pts] - **[Written]**
- **5.2 News Data Sentiment Classification Using Logistic Regression** [30pts] - **[Programming]**
    - `sigmoid` [2pts]
    - `bias_augment` [3pts]
    - `predict_probs` [5pts]
    - `predict_labels` [2pts]
    - `loss` [3pts]
    - `gradient` [3pts]
    - `accuracy` [2pts]
    - `evaluate` [5pts]
    - `fit` [5pts]
- **5.3 Setting the Threshold** [3pts] - **[Programming]**

Q6: Feature Selection [20pts]

Deliverables: feature_reduction.py

- **Selection Methods** [20pts] - **[Programming]**
  - `forward_selection` [10pts]
  - `backward_elimination` [10pts]

Q7: Imbalanced Classes in Classification Tasks [5.6% Bonus for All]

Deliverables: smote.py

- **7.1 A More Comprehensive Measure** [1.75% Bonus for All] - **[Programming]**
  - `generate_confusion_matrix` [0.75%]
  - ROC-AUC
    - `threshold_eval` [0.25%]
    - `generate_roc` [0.50%]
    - `integrate_auc` [0.25%]
- **7.2 SMOTE** [3.85% Bonus for All] - **[Programming]**
  - `interpolate` [1%]
  - `k_nearest_neighbors` [0.75%]
  - `smote` [2.1%]

Point Totals:

- Total Base: 200pts for grads / 170pts for undergrads
  - Programming: 168pts for grads / 138pts for undergrads
  - Written: 32pts for both
- Total Undergrad Bonus: 6%
  - Programming: 6%
  - Written: 0%
- Total Bonus for All: 10%
  - Programming: 7%
  - Written: 3%

Submission Instructions

Do not add or remove imports from any files, as this will cause the autograder to fail to parse your solution.

For actual submission to Gradescope, upload the following files:

- **Assignment 3 Programming**
  - audiocompression.py
  - eigenfaces.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
  - audiocompression.py (used as an import in svd_recommender.py; you need it to be resolvable as a module even if you don't use the import)
  - svd_recommender.py
- **Assignment 3 Non-Programming**
  - Use some tool to convert this notebook into a pdf.
  - It is your responsibility to make sure that all LaTeX renders, all generated matplotlib figures render, none of your answers are clipped, the font is a reasonable size, the pdf is a reasonable resolution, the pdf is the correct size

(8.5"x11"). Failure to do so may result in heavy penalties.
- It is also your responsibility to correctly assign pages to the relevant subquestions. If you are unsure what is being graded, to be safe, you may submit any page containing content for the entire subquestion.

0 Set up

This notebook is tested under python 3.11._, 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 a VS Code and Anaconda Setup Tutorial on Ed under the "Links" category. You can also find brief setup instructions in the environment folder.

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 os
import sys
import time
import warnings

import matplotlib
import numpy as np
import pandas as pd
from IPython.display import Audio, display
from matplotlib import colors as mcolors
from matplotlib import pyplot as plt
from scipy import signal
from scipy.io import wavfile
from sklearn.datasets import load_breast_cancer, make_classification, make_regression
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
```

```
Version information
python: 3.14.0 | packaged by conda-forge | (main, Oct 22 2025, 23:45:45) [Clang 20.1.8 ]
matplotlib: 3.10.7
numpy: 2.3.4
```

Q1: Audio Compression (with SVD) [25 pts] [P] | [W]

In the below cells, we load in audio from a .wav file and plot it as a spectrogram. An audio file is merely a recording of pressure as a function of time. If we perform a Fourier transform, we can decompose that function into pure frequency signals at varying amplitudes. A spectrogram is a visual way to graph this result. The x-axis is the time elapsed in the audio, the y-axis denotes the frequency of the pure signals in question, and the amplitude of a given frequency will have higher intensity colors, corresponding to the color bar on the plot.

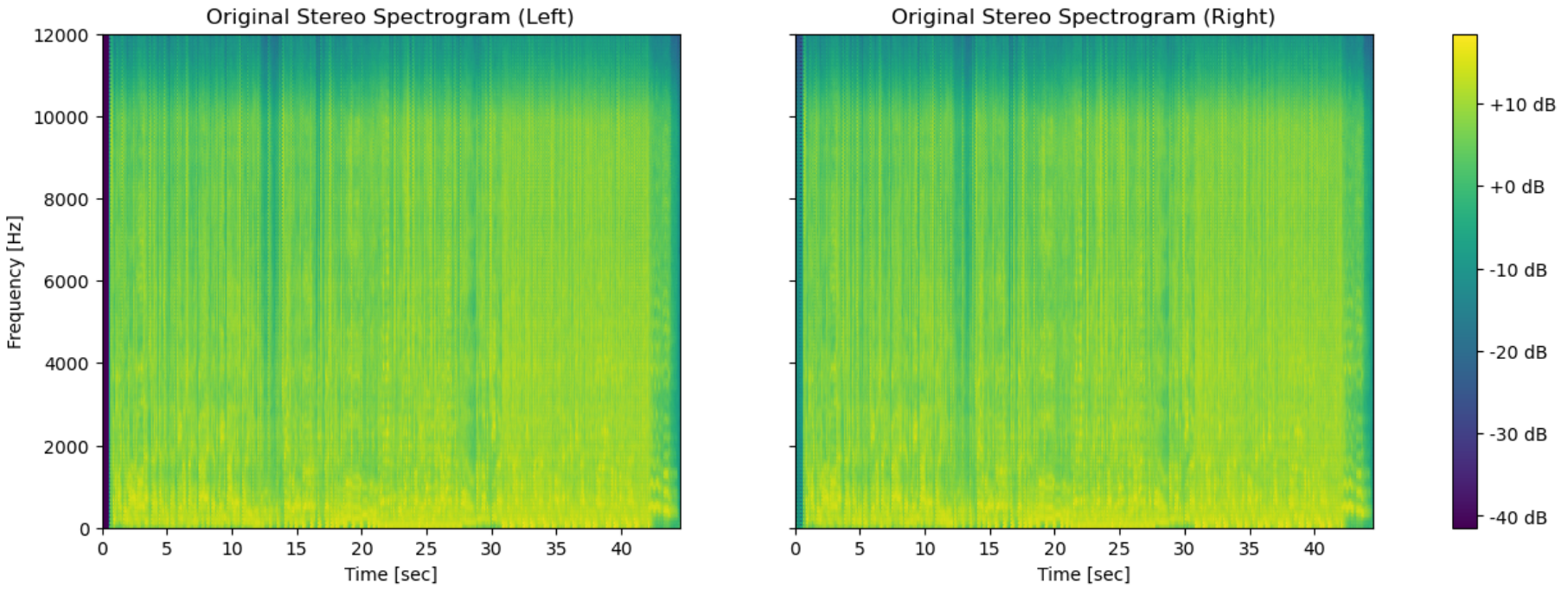
Spectrograms are quite useful in machine learning, as, even though the spectrogram represents the same information as the pressure function, there's a lot of information present in the frequency space that is simply easier to access from a spectrogram. If you were to design a model that only uses the raw pressure signal, the Nyquist-Shannon theorem would require you to sample at a timestep at least double the frequency of the signal you want to capture, and far more frequently if you want fidelity, and you still wouldn't have any guarantee that your model would capture the important information. Though there are merits in a pure approach, spectrograms offer a great deal of information, presented as an image, for which there are a whole host of established techniques.

In this question, we will be compressing and reconstructing these spectrograms to demonstrate the power of certain compression techniques.

Load audio data and plot spectrogram

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
# load audio file
sample_rate, stereo_audio_data = wavfile.read("data/InLoveWithMachineLearning.wav")

# Decimate each channel by a factor of 2 to reduce data size
stereo_audio_data = np.apply_along_axis(
    lambda x: signal.decimate(x, 2), 0, stereo_audio_data
)
stereo_audio_data = stereo_audio_data.transpose()
# Update the sample rate to match the new data
sample_rate = sample_rate // 2

print("Stereo Audio")
display(Audio(data=stereo_audio_data, rate=sample_rate))

# compute spectrogram
stereo_frequencies, stereo_times, stereo_Zxx = signal.stft(
    stereo_audio_data, fs=sample_rate
)
stereo_spectrogram_data = 2 * np.log(np.abs(stereo_Zxx) + 1e-9)
stereo_phase = np.angle(stereo_Zxx)

# plot spectrogram
fig, axs = plt.subplots(1, 2, figsize=(16, 5), sharey=True)
im_left = axs[0].pcolormesh(
    stereo_times, stereo_frequencies, stereo_spectrogram_data[0], shading="gouraud"
)
axs[0].set_title("Original Stereo Spectrogram (Left)")
axs[0].set_ylabel("Frequency [Hz]")
axs[0].set_xlabel("Time [sec]")
im_right = axs[1].pcolormesh(
    stereo_times, stereo_frequencies, stereo_spectrogram_data[1], shading="gouraud"
)
axs[1].set_title("Original Stereo Spectrogram (Right)")
axs[1].set_xlabel("Time [sec]")

fig.colorbar(im_left, ax=axs, format="%+2.0f dB")
plt.show()
```

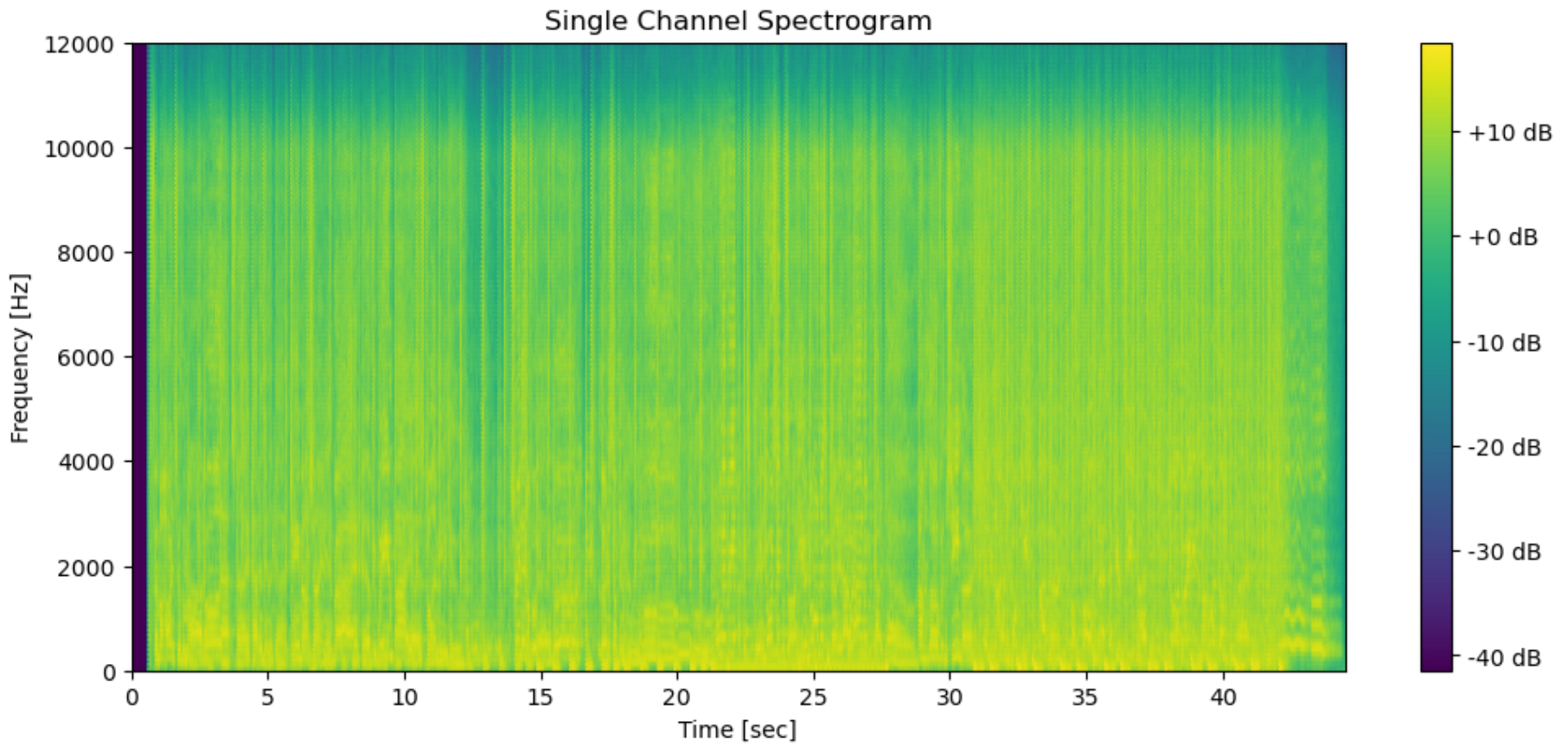
```
Stereo Audio
```



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


# take only the first audio channel
sc_audio_data = stereo_audio_data[[0]]

print("Single Channel Audio")
display(Audio(data=sc_audio_data, rate=sample_rate))

# compute spectrogram
sc_frequencies, sc_times, sc_Zxx = signal.stft(sc_audio_data, fs=sample_rate)
single_channel_spectrogram_data = 2 * np.log(np.abs(sc_Zxx) + 1e-9)
sc_phase = np.angle(sc_Zxx)

# plot spectrogram
fig, axs = plt.subplots(1, 1, figsize=(12, 5))
im1 = axs.pcolormesh(
    sc_times, sc_frequencies, single_channel_spectrogram_data[0], shading="gouraud"
)
axs.set_title("Single Channel Spectrogram")
axs.set_ylabel("Frequency [Hz]")
axs.set_xlabel("Time [sec]")
fig.colorbar(im1, ax=axs, format="%+2.0f dB")
plt.show()
```

Single Channel Audio

0:00 -0:44



1.1 Audio Compression [20pts] [P]

Singular Value Decomposition (SVD) is a dimensionality reduction technique that allows us to compress audio 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 audio compression, apply SVD on each matrix and get rid of the small singular values to compress the audio. The loss of information through this process is negligible, and the difference between the initial and compressed audio can hardly be noticed.

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 **audiocompression.py** file, complete the following functions:

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

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

HINT 2: If you have never used `np.linalg.svd`, it might be helpful to read Numpy's SVD documentation and note the particularities of the V matrix and that it is returned already transposed.

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

1.1.1 Local Tests for Audio Compression Single Channel Case [No Points]

You may test your implementation of the functions contained in **audiocompression.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 TestAudioCompression

unittest_ic = TestAudioCompression()
unittest_ic.test_svd_mono()
unittest_ic.test_compress_mono()
unittest_ic.test_rebuild_svd_mono()
unittest_ic.test_compression_ratio_mono()
unittest_ic.test_recovered_variance_proportion_mono()
```

```
UnitTest passed successfully for "SVD calculation - single channel audio"!
UnitTest passed successfully for "Audio compression - single channel audio"!
UnitTest passed successfully for "SVD reconstruction - single channel audio"!
UnitTest passed successfully for "Compression ratio - single channel audio"!
UnitTest passed successfully for "Recovered variance proportion - single channel audio"!
```

1.1.2 Local Tests for Audio Compression Stereo Case [No Points]

You may test your implementation of the functions contained in **audiocompression.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 TestAudioCompression

unittest_ic = TestAudioCompression()

unittest_ic.test_svd_stereo()
unittest_ic.test_compress_stereo()
unittest_ic.test_rebuild_svd_stereo()
unittest_ic.test_compression_ratio_stereo()
unittest_ic.test_recovered_variance_proportion_stereo()
```

```
UnitTest passed successfully for "SVD calculation – stereo audio"!
UnitTest passed successfully for "Audio compression – stereo audio"!
UnitTest passed successfully for "SVD reconstruction – stereo audio"!
UnitTest passed successfully for "Compression ratio – stereo audio"!
UnitTest passed successfully for "Recovered variance proportion – stereo audio"!
```

1.2 Single Channel Audio [No Points]

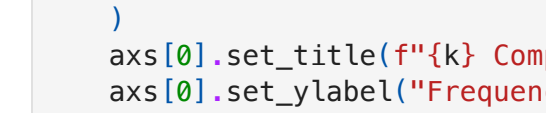
As a test of your implementation of the functions from Q1.1, try out compression on the single channel audio. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

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 expected and acceptable since some of the pixels may go above 1.0 while rebuilding. There's no need to change your code to account for this. You should see reasonable spectrograms to original even with such clipping.

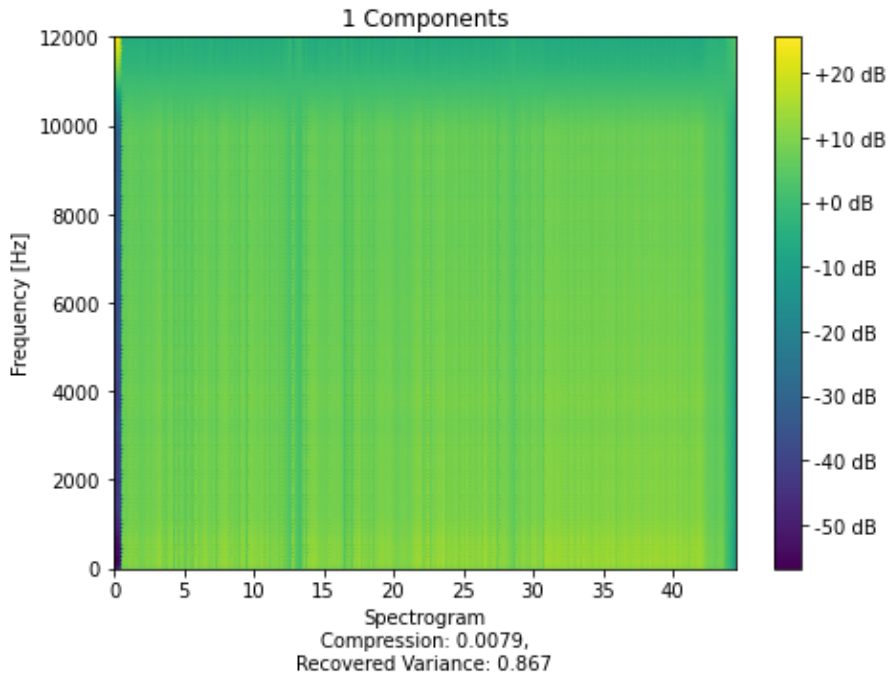
This part of the notebook is not being graded. You may consider section 1.2 to be a warmup.

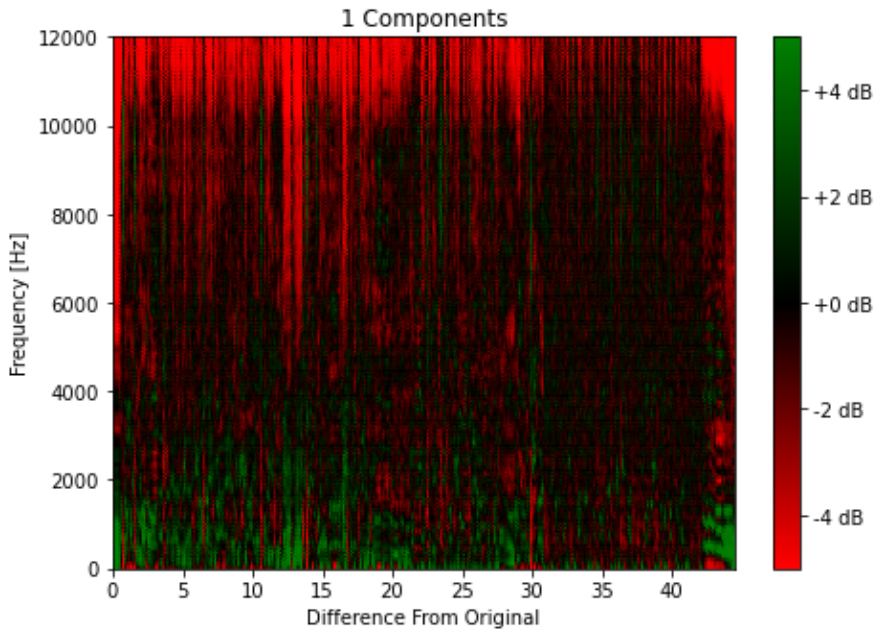
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from audiocompression import AudioCompression

# Color mapping
residue_map = mcolors.LinearSegmentedColormap.from_list(
    "red_green", [(0, "red"), (0.5, "black"), (1, "green")]
)

audiocompression = AudioCompression()
U, S, V = audiocompression.svd(single_channel_spectrogram_data)
component_num = [1, 2, 4, 8, 16, 32]

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

# plot several reconstructions
i = 0
for k in component_num:
    U_compressed, S_compressed, V_compressed = audiocompression.compress(U, S, V, k)
    spectrogram_data_reconstructed = audiocompression.rebuild_svd(
        U_compressed, S_compressed, V_compressed
    )
    sc_Zxx_reconstructed = (np.exp(spectrogram_data_reconstructed / 2) - 1e-9) * np.exp(
        1j * sc_phase
    )
    _, sc_audio_data_reconstructed = signal.istft(sc_Zxx_reconstructed, fs=sample_rate)
    print(f"Reconstructed Audio with {k} components")
    display(Audio(data=sc_audio_data_reconstructed, rate=sample_rate))
    c = np.around(
        audiocompression.compression_ratio(spectrogram_data_reconstructed, k), 4
    )
    r = np.around(audiocompression.recovered_variance_proportion(S, k), 3)
    fig, axs = plt.subplots(1, 2, figsize=(16, 5), dpi=72)
    im1 = axs[0].pcolormesh(
        sc_times, sc_frequencies, spectrogram_data_reconstructed[0], shading="gouraud"
    )
    axs[0].set_title(f"{k} Components")
    axs[0].set_ylabel("Frequency [Hz]")
    axs[0].set_xlabel(f"Spectrogram\nCompression: {c},\nRecovered Variance: {r}")
    residue = axs[1].pcolormesh(
        sc_times,
        sc_frequencies,
        single_channel_spectrogram_data[0] - spectrogram_data_reconstructed[0],
        shading="gouraud",
        cmap=residue_map,
        vmin=-5,
        vmax=5,
    )
    axs[1].set_title(f"{k} Components")
    axs[1].set_ylabel("Frequency [Hz]")
    axs[1].set_xlabel(f"Difference From Original")
    fig.colorbar(im1, ax=axs[0], format="%+2.0f dB")
    fig.colorbar(residue, ax=axs[1], format="%+2.0f dB")
    plt.show()
    i = i + 1
```

```
Reconstructed Audio with 1 components
```



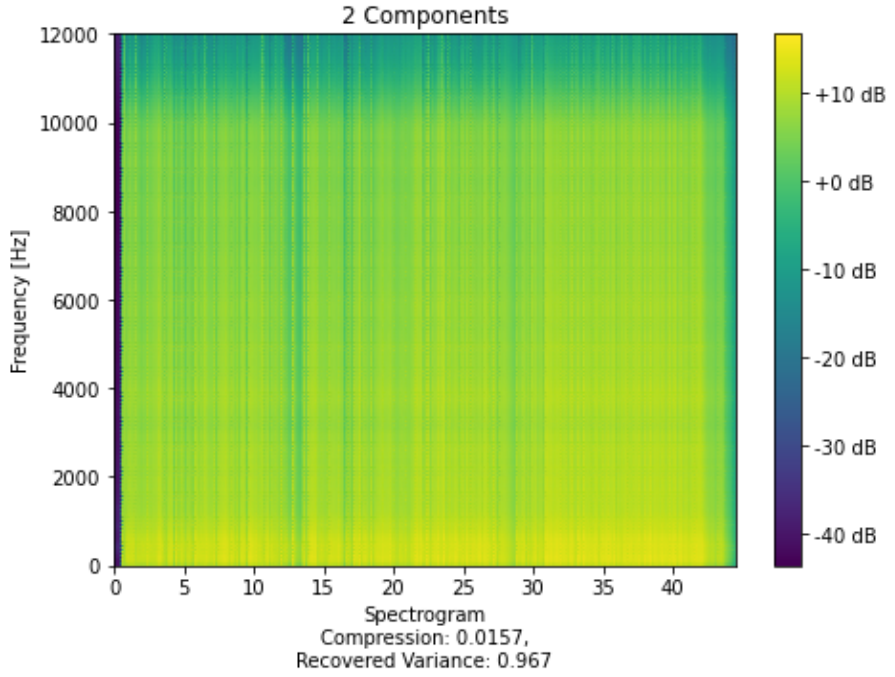

```
<Figure size 1800x1800 with 0 Axes>
```

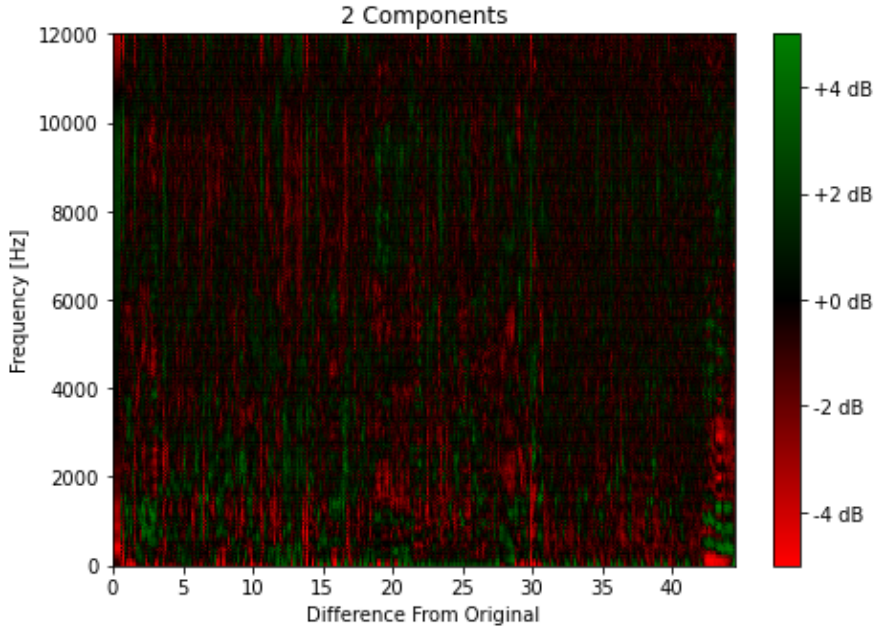




Reconstructed Audio with 2 components

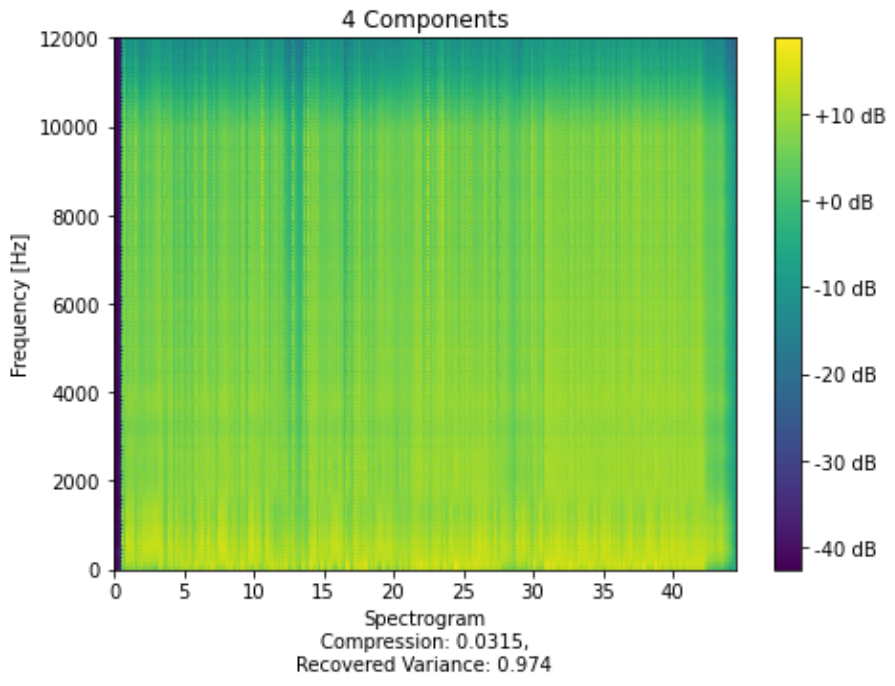
0:00 -0:44

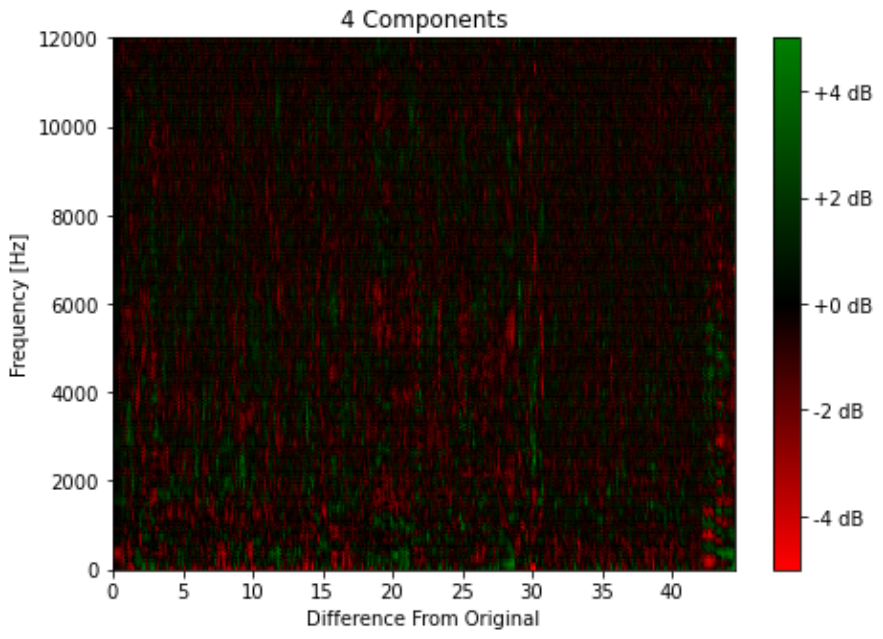




Reconstructed Audio with 4 components

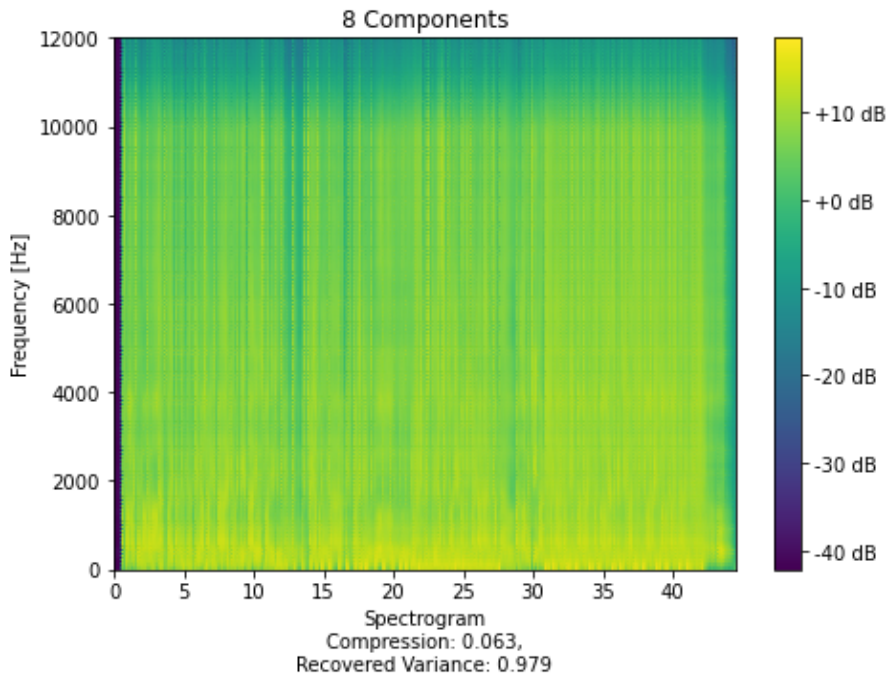
0:00 -0:44

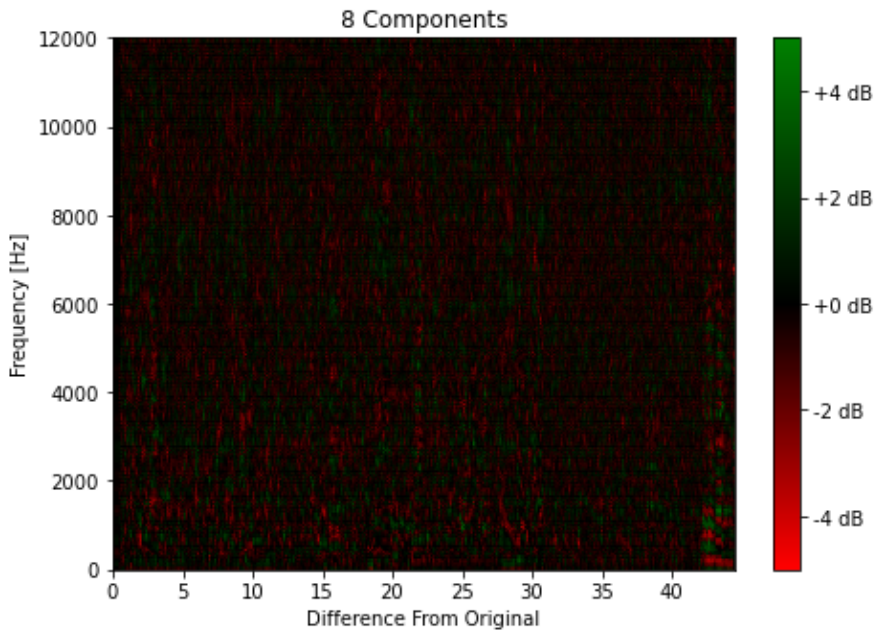




Reconstructed Audio with 8 components

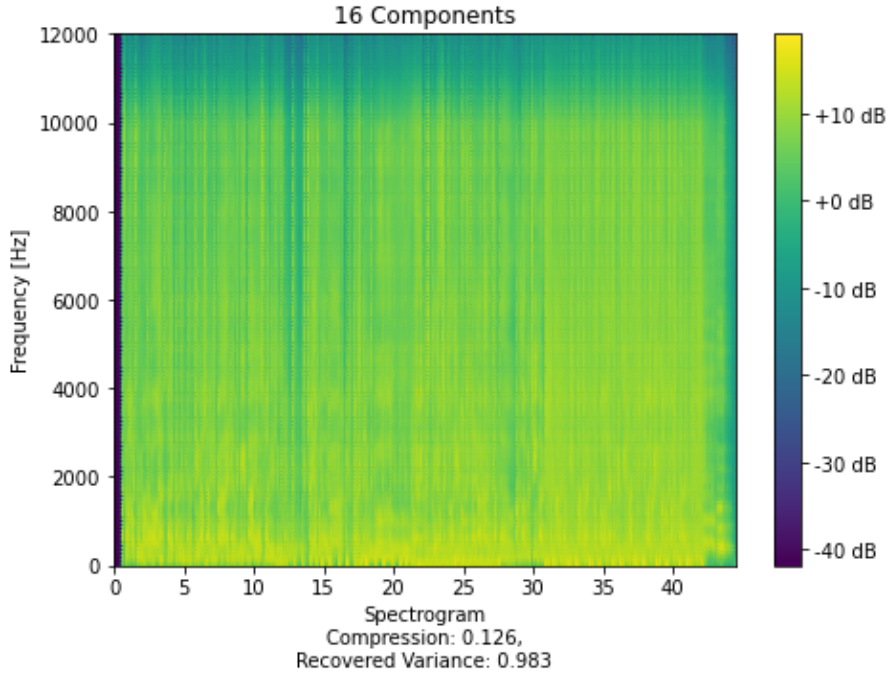
0:00 -0:44

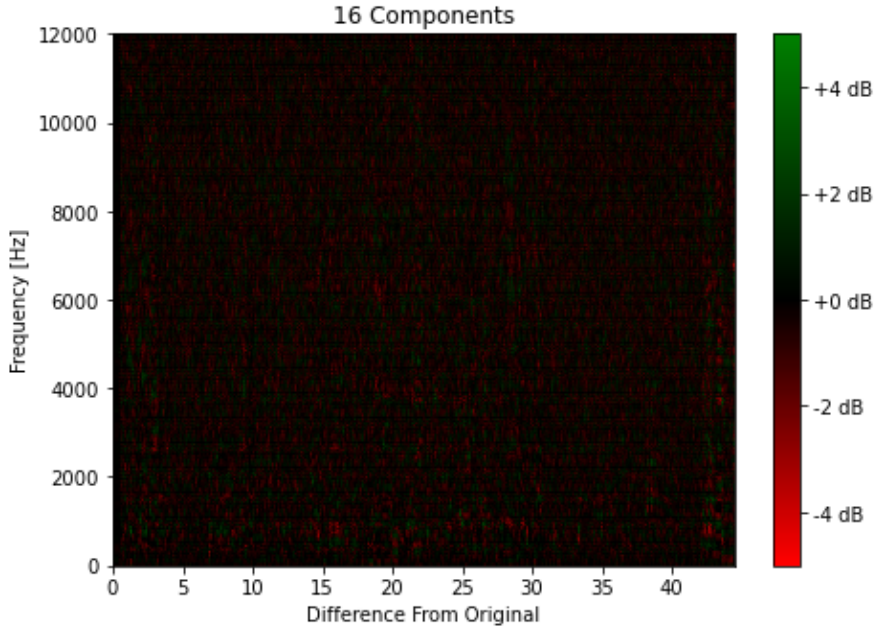




Reconstructed Audio with 16 components

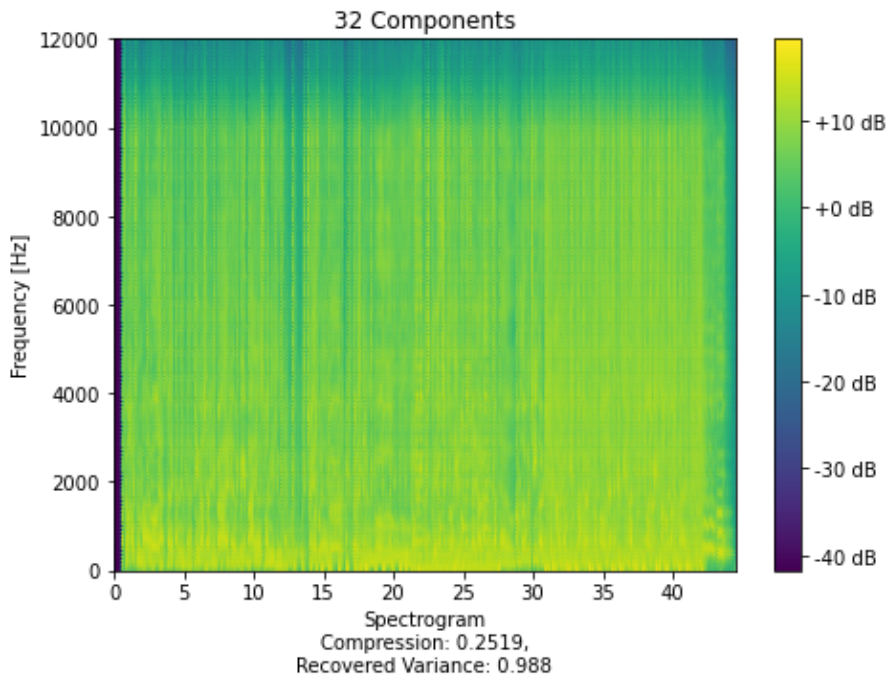
0:00 -0:44

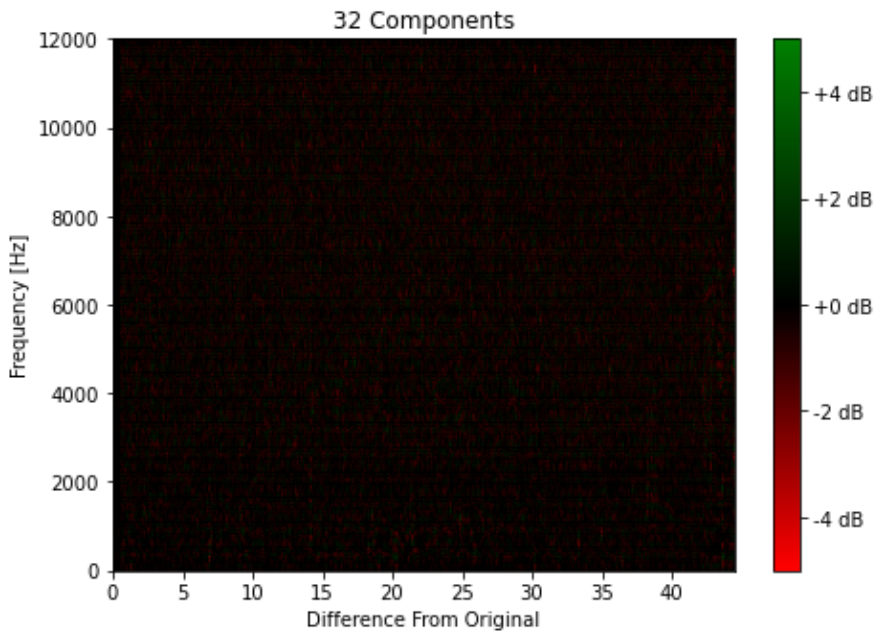




Reconstructed Audio with 32 components

0:00 -0:44





In the output of this cell, you should see your reconstruction and a "residue plot." This contains your reconstruction minus the original, so you can explicitly see the errors caused by compression. The highest fidelity reconstruction with 32 components should have a residue plot that's almost entirely blank, with only a bit of fine detail missing from the reconstruction.

If you're curious, you should investigate what times or frequency bands have higher error than others and what might cause that!

Now, you'll see the actual memory savings that result from your compression. This will use your implementation of the functions from Q1.1. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

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

```
from audiocompression import AudioCompression

audiocompression = AudioCompression()
U, S, V = audiocompression.svd(single_channel_spectrogram_data)
component_num = [1, 2, 4, 8, 16, 32]

# Compare memory savings for single channel audio
for k in component_num:
    og_bytes, comp_bytes, savings = audiocompression.memory_savings(
        single_channel_spectrogram_data, U, S, V, k
    )
    comp_ratio = og_bytes / comp_bytes
    og_bytes = audiocompression.nbytes_to_string(og_bytes)
    comp_bytes = audiocompression.nbytes_to_string(comp_bytes)
    savings = audiocompression.nbytes_to_string(savings)
    print(
        f"{k} components: Original Audio: {og_bytes} -> Compressed Audio: {comp_bytes}, Savings: {savings},
    )
```

```
1 components: Original Audio: 8.209 MB -> Compressed Audio: 66.18 KB, Savings: 8.145 MB, Compression Ratio
127.0:1
2 components: Original Audio: 8.209 MB -> Compressed Audio: 132.359 KB, Savings: 8.08 MB, Compression Ratio
63.5:1
4 components: Original Audio: 8.209 MB -> Compressed Audio: 264.719 KB, Savings: 7.951 MB, Compression Rati
o 31.8:1
8 components: Original Audio: 8.209 MB -> Compressed Audio: 529.438 KB, Savings: 7.692 MB, Compression Rati
o 15.9:1
16 components: Original Audio: 8.209 MB -> Compressed Audio: 1.034 MB, Savings: 7.175 MB, Compression Ratio
7.9:1
32 components: Original Audio: 8.209 MB -> Compressed Audio: 2.068 MB, Savings: 6.141 MB, Compression Ratio
4.0:1
```

1.3 Stereo Audio [5 pts] [W]

This section will run the same processes as 1.2, but on the stereo (2-channel) audio.

[W]: Make sure the following cells and their outputs are displayed and properly assigned to Q1.3 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. We will leniently grade your reconstructions and their data usage to ensure reasonable output.

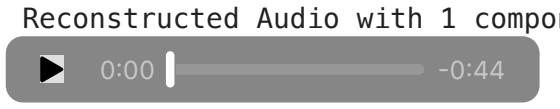
HINT 1: Make sure your implementation of `recovered_variance_proportion` returns an array of 2 floats when given stereo audio.

HINT 2: Try performing SVD on the individual audio channels and then stack the individual channel U, S, V matrices.

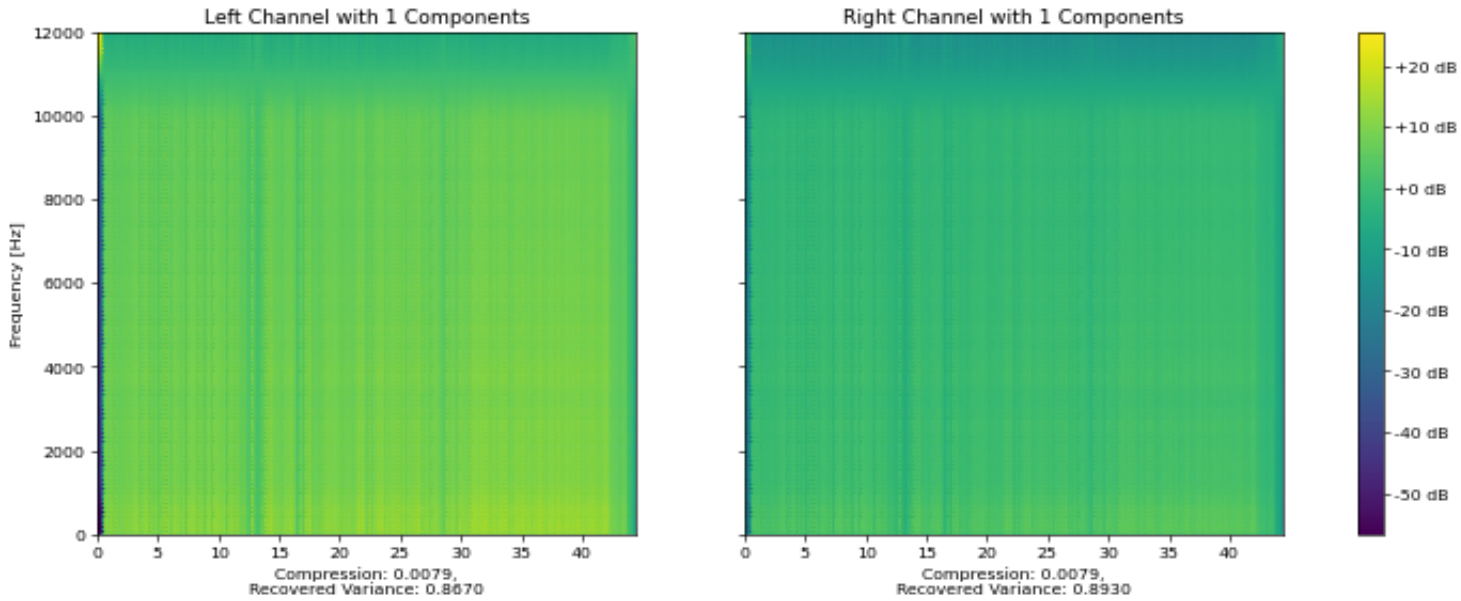
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from audiocompression import AudioCompression

residue_map = mcolors.LinearSegmentedColormap.from_list(
    "red_green", [(0, "red"), (0.5, "black"), (1, "green")]
)

audiocompression = AudioCompression()
U, S, V = audiocompression.svd(stereo_spectrogram_data)
component_num = [1, 2, 4, 8, 16, 32]

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

# plot several images
i = 0
for k in component_num:
    U_compressed, S_compressed, V_compressed = audiocompression.compress(U, S, V, k)
    spectrogram_data_reconstructed = audiocompression.rebuild_svd(
        U_compressed, S_compressed, V_compressed
    )
    stereo_Zxx_reconstructed = (
        np.exp(spectrogram_data_reconstructed / 2) - 1e-9
    ) * np.exp(1j * stereo_phase)
    _, stereo_audio_data_reconstructed = signal.istft(
        stereo_Zxx_reconstructed, fs=sample_rate
    )
    print(f"Reconstructed Audio with {k} components")
    display(Audio(data=stereo_audio_data_reconstructed, rate=sample_rate))
    c = np.around(audiocompression.compression_ratio(stereo_spectrogram_data, k), 4)
    r = np.around(audiocompression.recovered_variance_proportion(S, k), 3)
    fig, axs = plt.subplots(2, 2, figsize=(16, 12), sharey=True, dpi=56)
    im_left = axs[0][0].pcolormesh(
        stereo_times,
```

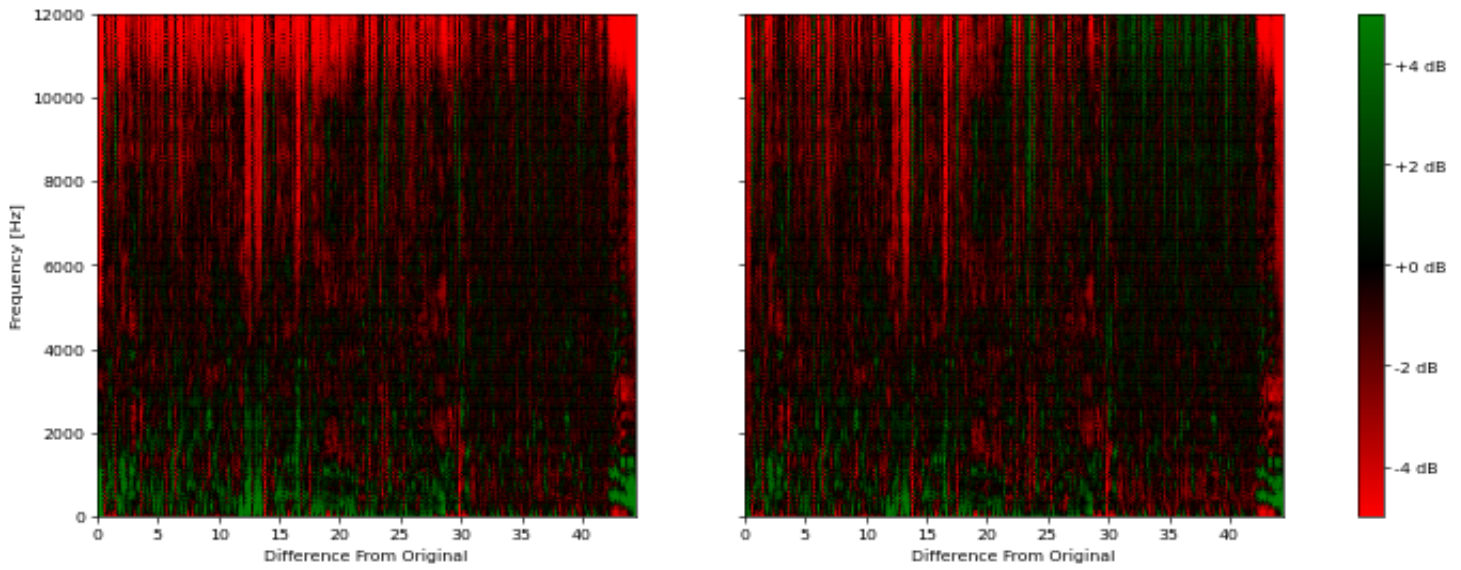
```
        stereo_frequencies,
        spectrogram_data_reconstructed[0],
        shading="gouraud",
    )
    axs[0][0].set_title(f"Left Channel with {k} Components")
    axs[0][0].set_ylabel("Frequency [Hz]")
    axs[0][0].set_xlabel(
        f"Compression: {np.around(c,4)},\nRecovered Variance: {r[0]:.4f}"
    )
    im_right = axs[0][1].pcolormesh(
        stereo_times,
        stereo_frequencies,
        spectrogram_data_reconstructed[1],
        shading="gouraud",
    )
    axs[0][1].set_title(f"Right Channel with {k} Components")
    axs[0][1].set_xlabel(
        f"Compression: {np.around(c,4)},\nRecovered Variance: {r[1]:.4f}"
    )
    residue_left = axs[1][0].pcolormesh(
        stereo_times,
        stereo_frequencies,
        stereo_spectrogram_data[0] - spectrogram_data_reconstructed[0],
        shading="gouraud",
        cmap=residue_map,
        vmin=-5,
        vmax=5,
    )
    axs[1][0].set_ylabel("Frequency [Hz]")
    axs[1][0].set_xlabel(f"Difference From Original")
    residue_right = axs[1][1].pcolormesh(
        stereo_times,
        stereo_frequencies,
        stereo_spectrogram_data[1] - spectrogram_data_reconstructed[1],
        shading="gouraud",
        cmap=residue_map,
        vmin=-5,
        vmax=5,
    )
    axs[1][1].set_xlabel(f"Difference From Original")

    fig.colorbar(im_left, ax=axs[0], format="%+2.0f dB")
    fig.colorbar(residue_left, ax=axs[1], format="%+2.0f dB")
    plt.show()
    i = i + 1
```

```
Reconstructed Audio with 1 components
```



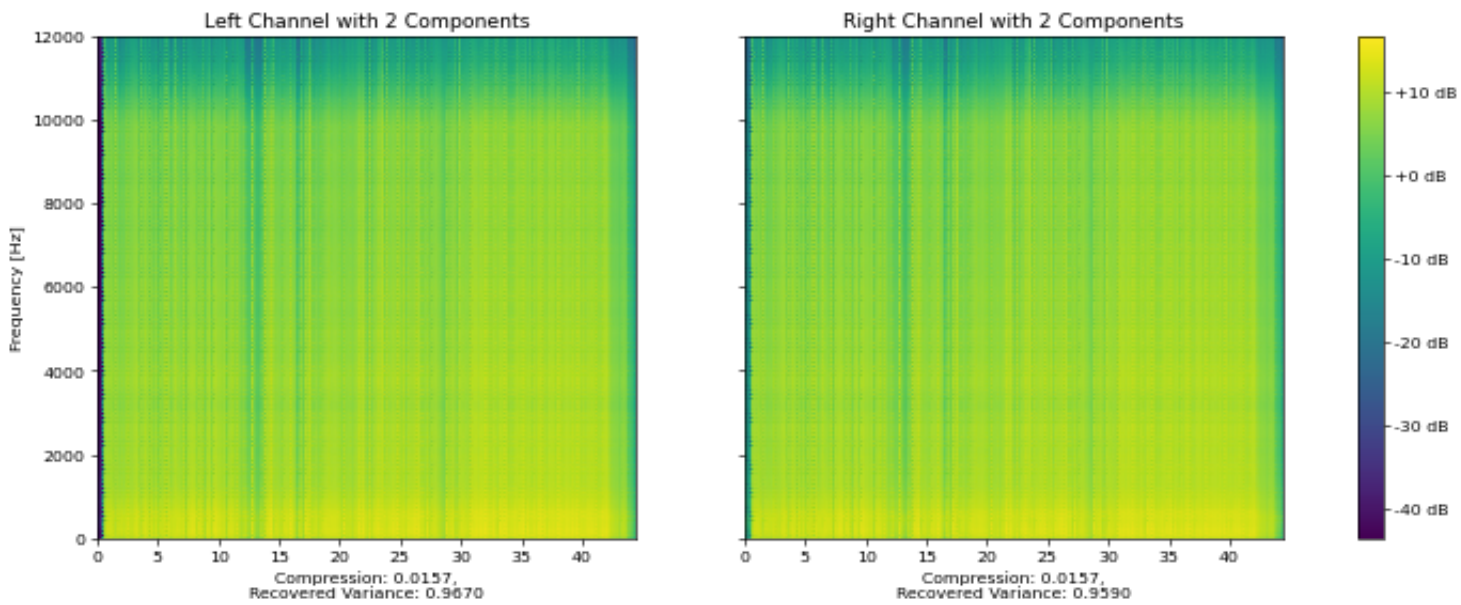

```
<Figure size 1800x1800 with 0 Axes>
```

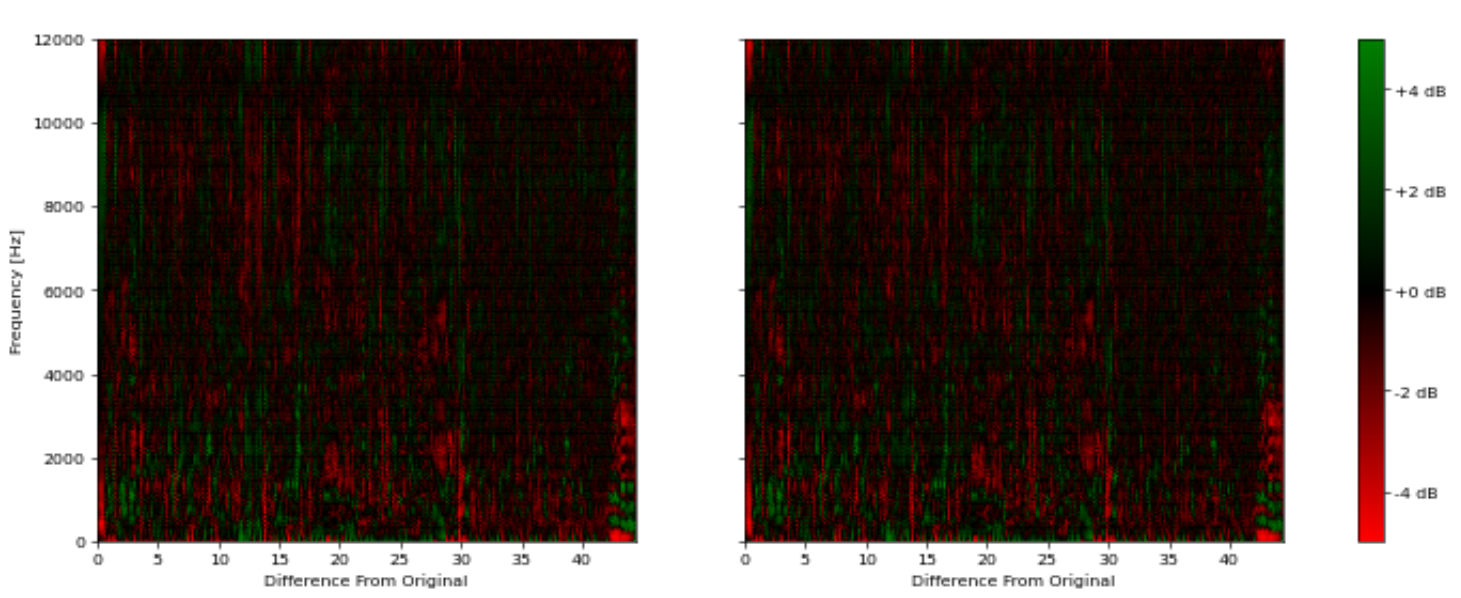




Reconstructed Audio with 2 components

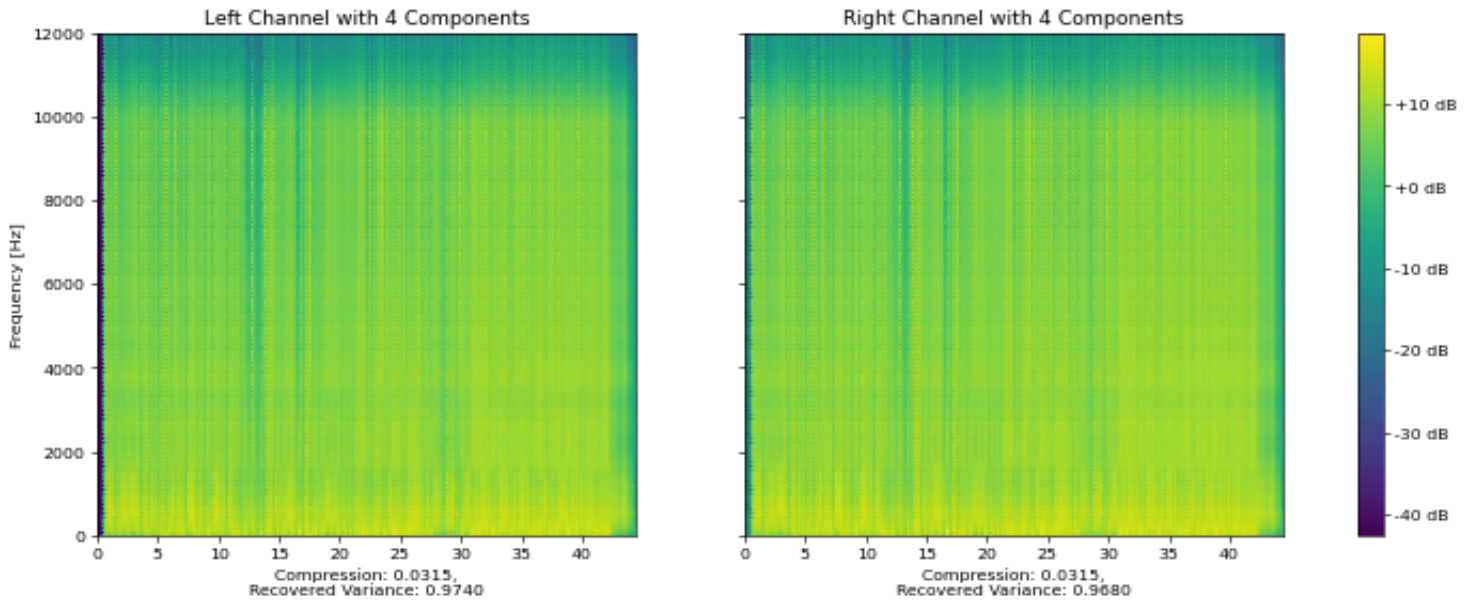
0:00 -0:44

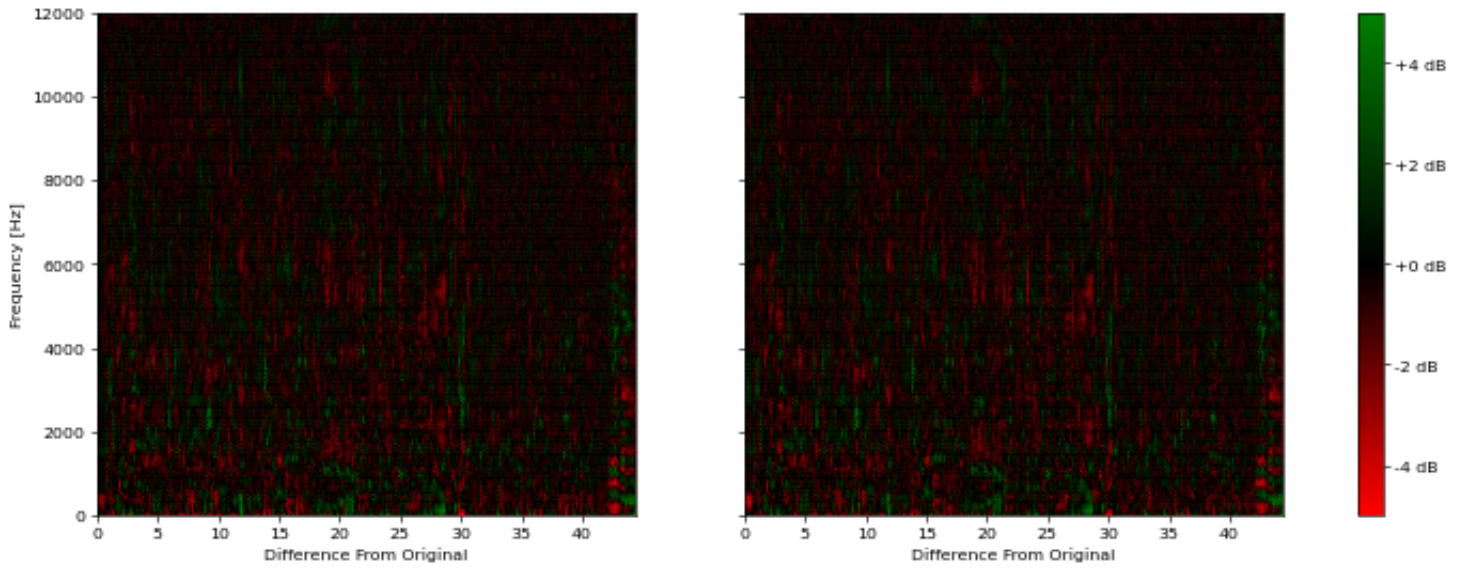




Reconstructed Audio with 4 components

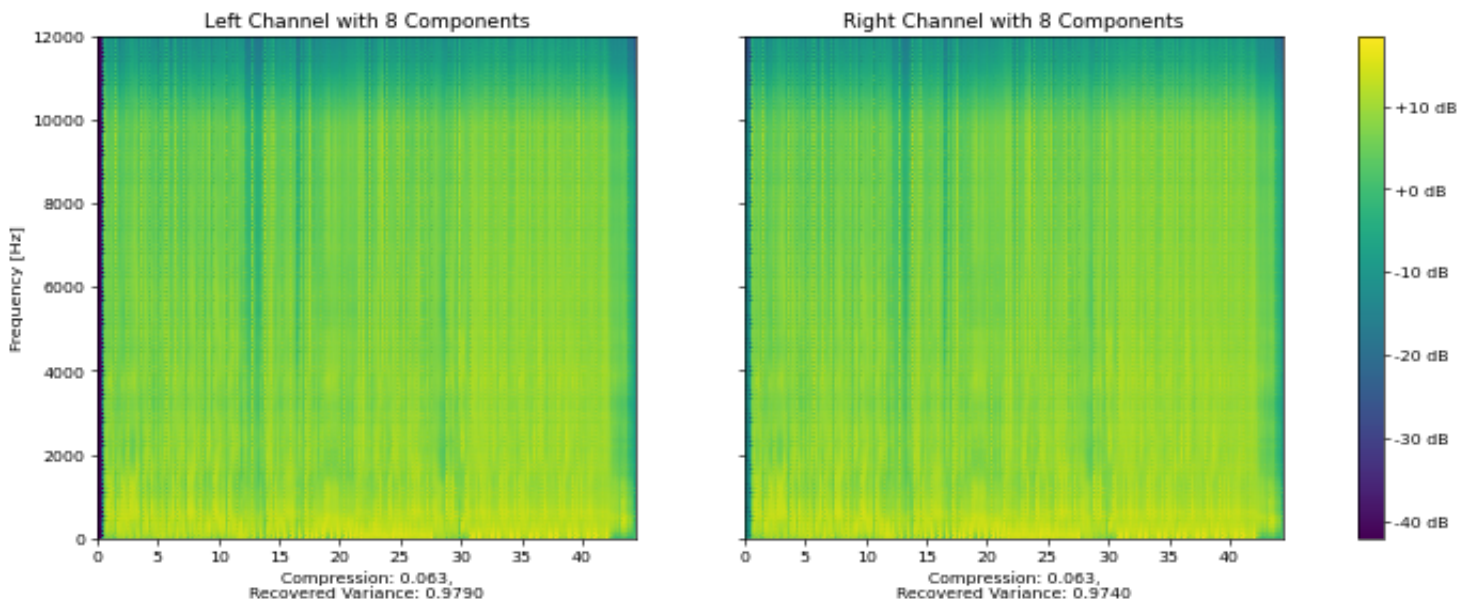
0:00 -0:44

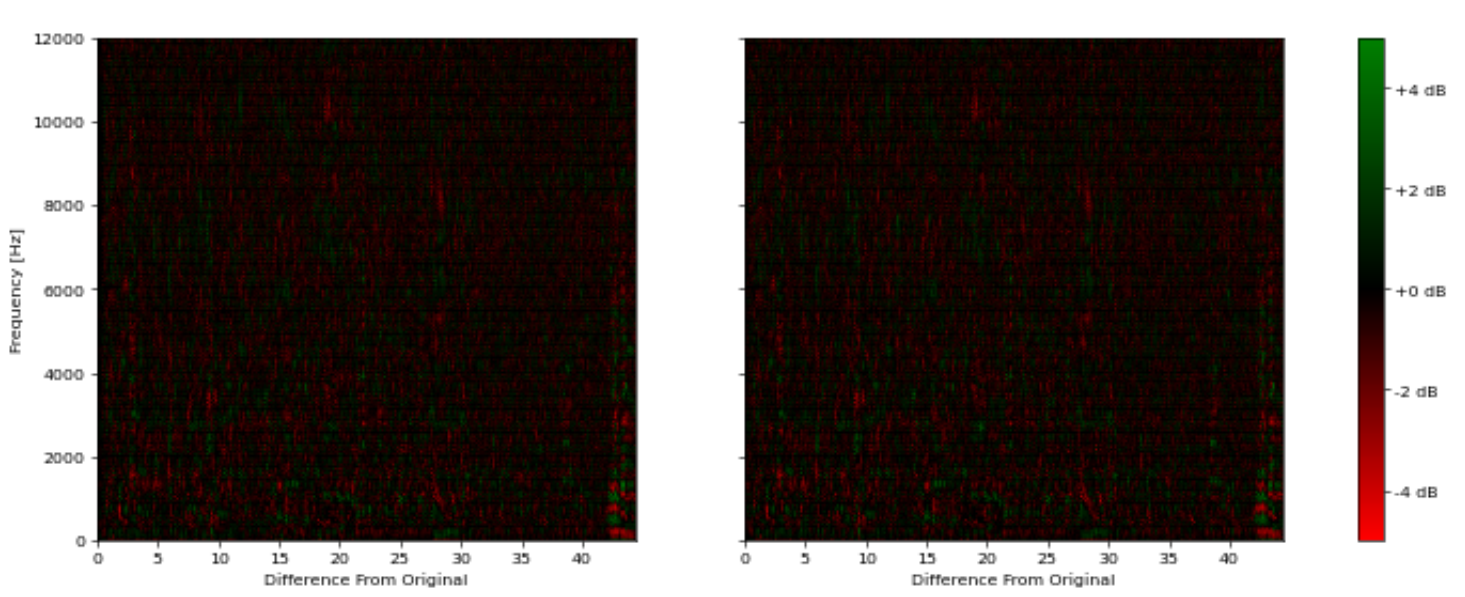




Reconstructed Audio with 8 components

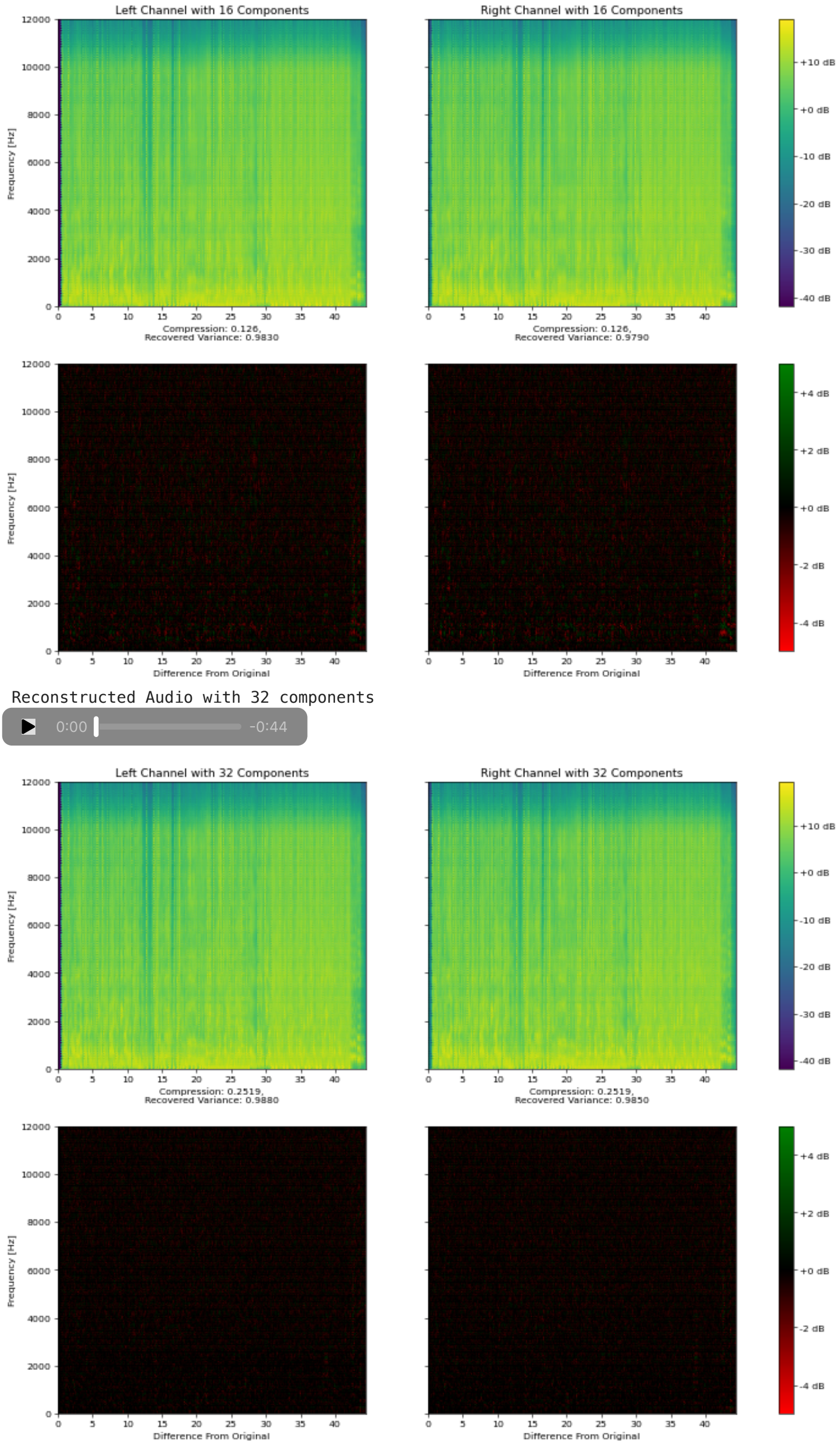
0:00 -0:44





Reconstructed Audio with 16 components

0:00 -0:44



Reconstructed Audio with 32 components

0:00 -0:44

Now, you'll see the actual memory savings that result from your compression. This will use your implementation of the functions from Q1.1. You can simply run the below cell without making any changes to it, assuming you have implemented the functions in Q1.1.

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
from audiocompression import AudioCompression
```

```
audiocompression = AudioCompression()
U, S, V = audiocompression.svd(stereo_spectrogram_data)

component_num = [1, 2, 4, 8, 16, 32]

# Compare the memory savings of the stereo audio
i = 0
for k in component_num:
    og_bytes, comp_bytes, savings = audiocompression.memory_savings(
        stereo_spectrogram_data, U, S, V, k
    )
    comp_ratio = og_bytes / comp_bytes
    og_bytes = audiocompression.nbytes_to_string(og_bytes)
    comp_bytes = audiocompression.nbytes_to_string(comp_bytes)
    savings = audiocompression.nbytes_to_string(savings)
    print(
        f"{k} components: Original Audio: {og_bytes} -> Compressed Audio: {comp_bytes}, Savings: {savings},
    )
```

```
1 components: Original Audio: 16.418 MB -> Compressed Audio: 132.359 KB, Savings: 16.289 MB, Compression Ra
tio 127.0:1
2 components: Original Audio: 16.418 MB -> Compressed Audio: 264.719 KB, Savings: 16.16 MB, Compression Rat
io 63.5:1
4 components: Original Audio: 16.418 MB -> Compressed Audio: 529.438 KB, Savings: 15.901 MB, Compression Ra
tio 31.8:1
8 components: Original Audio: 16.418 MB -> Compressed Audio: 1.034 MB, Savings: 15.384 MB, Compression Rati
o 15.9:1
16 components: Original Audio: 16.418 MB -> Compressed Audio: 2.068 MB, Savings: 14.35 MB, Compression Rati
o 7.9:1
32 components: Original Audio: 16.418 MB -> Compressed Audio: 4.136 MB, Savings: 12.282 MB, Compression Rat
io 4.0:1
```

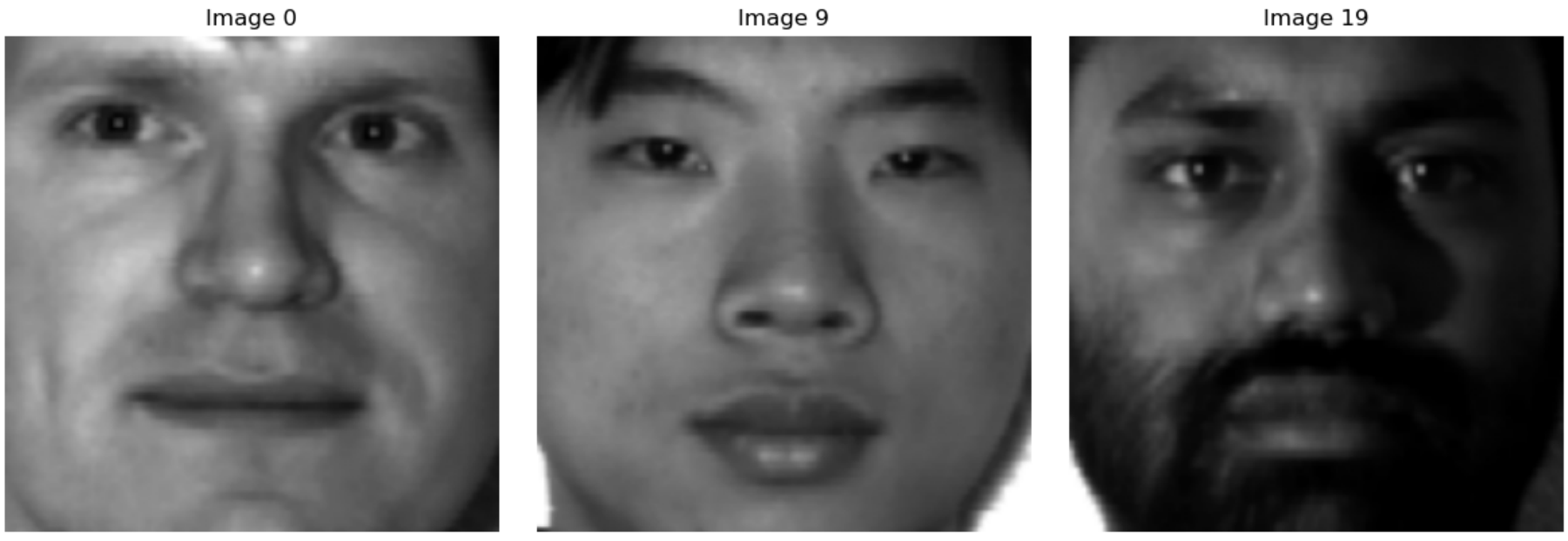
Q2: Eigenfaces (with PCA) [20 pts] [P]

The Yale Face Database is a set of pose-normalized images of subjects with varying appearance and lighting. For the purposes of this question, you'll be passed the data already preprocessed. The data is grayscaled and flattened to (75625,). Below is a sample of some of the images in the dataset.

We'll investigate one technique that one might use to analyze regularized images.

Load images data and plot

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

imgNum = 0
images = []
selected_indices = [0, 9, 19]
selected_images = []

# load
for filename in sorted(os.listdir("./data/faces")):
    img_path = os.path.join("./data/faces", filename)
    image = plt.imread(img_path)
    image = np.dot(image[..., :3], [0.299, 0.587, 0.114])  # RGB → grayscale
    if imgNum in selected_indices:
        selected_images.append(image)
    images.append(image.flatten())  # flatten for later use
    imgNum += 1
faces = np.array(images)


# display
fig, axes = plt.subplots(1, len(selected_indices), figsize=(12, 4))
for i, ax in enumerate(axes):
    ax.imshow(selected_images[i], cmap="gray")
    ax.axis("off")
    ax.set_title(f"Image {selected_indices[i]}")
plt.tight_layout()
plt.show()
plt.close()
```



2.1 Eigenfaces [20pts] [P]

Singular Value Decomposition (SVD) is a matrix decomposition technique. For a real-valued matrix M,

$$M_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n} = \begin{bmatrix} | & | & & | \\ \vec{u_1} & \vec{u_2} & \cdots & \vec{u_m} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_{\min(m,n)} \\ & & & \end{bmatrix} \begin{bmatrix} - & \vec{v_1}^T & - \\ - & \vec{v_2}^T & - \\ & \vdots & \\ - & \vec{v_n}^T & - \end{bmatrix}$$

This product $U\Sigma V^T$ can be simplified to a sum of outer vector products between the left singular vectors u and the right singular vectors v.

$$M = \sum_{i=1} \sigma_i \cdot \vec{u_i}\vec{v_i}^T$$

Just like in the previous question, if we interpret the image as a matrix, then apply SVD, the outer product $\vec{u_i}\vec{v_i}^T$ creates a facsimile of the image with even the proper shape: $m \times n$. This is then weighted by the singular value σ_i. Higher singular values therefore denote that the corresponding singular vectors have captured more useful information. Thus, we can discard the singular values that are small (and their corresponding singular vectors), while still retaining much of the variance present in the original image.

This is the standard usage of SVD. For any matrix, you can use that formula $M = \sum_{i=1} \sigma_i \cdot \vec{u_i}\vec{v_i}^T$, but you can terminate the sum early, allowing you to control how much information you want to retain.

The proportion of variance captured by the i^{th} product is $\frac{\sigma_i^2}{\sum_{j=1}\sigma_j^2}$ and we can sum this over the singular values that we choose to retain to calculate how much variance we capture.

In the previous problem, we were interested in finding a matrix a lower-rank matrix approximation of X. Here, we're moreso interested in finding a lower-dimensional subspace that preserves as much of the variance of the dataset as possible.

This is potentially useful, perhaps in the compression of images, but we'll use SVD a little bit differently. In facial recognition, we want to be able to analyze a whole set of images, and recognize similarities. One pretty basic idea is to construct a set of basis faces. If you have a bunch of characteristic faces, perhaps with different skin tones and textures, face proportions and shapes, instead of analyzing the pixels of a new image, you can analyze how similar it is the the set of basis faces. This is a really easy way to achieve dimensionality reduction. However, we need to be able to generate those basis faces in a way that's divorced selection biases and maximizes the separability of the data.

One such technique is dubbed "eigenfaces", and is a close variant of a technique called principal component analysis. To calculate the eigenfaces, we need a way to represent our data of faces. We'll achieve this by flattening each image, so the image becomes a row vector, then we can stack our N images, getting a $(N, D) = (N, H \times W)$ dataset. With that, we apply SVD on the centered matrix of all of those images, and isolate the right singular vectors (the rows of V^T). When we project our dataset of images onto this collection of eigenfaces, we retain the highest amount of variance. We can then simply choose the first k eigenfaces to get however much dimensionality reduction we require for our facial recognition method.

Principal Component Analysis or PCA provides the framework that justifies why eigenfaces work. PCA finds the principal components that maximize variance in the data. When we apply SVD to the matrix of face images, the singular vectors

correspond to these principal components. Simply, PCA tells us which directions in the high-dimensional space capture the most variation across different people's faces. By projecting a new face onto the top k eigenfaces, we get a compact representation that preserves the k most important features while discarding noise.

This technique is also the namesake for "eigenfaces." You may recall from lecture that you can also calculate the principal components of a matrix by calculating the eigenvectors of the covariance matrix of the centered data, hence "eigen-"faces.

In the `eigenfaces.py` file, complete the following functions:

- `svd`
- `compute_eigenfaces`
- `project`

HINT 1: 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 (that it is returned already transposed!). Additionally, note how `full_matrices=False` affects output shape.

2.1.1 Local Tests for Eigenfaces [No Points]

You may test your implementation of the functions contained in **eigenfaces.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 TestEigenfaces

unittest_ef = TestEigenfaces()

unittest_ef.test_svd()
unittest_ef.test_compute_eigenfaces()
unittest_ef.test_project()
```

```
UnitTest passed successfully for "SVD calculation"!
UnitTest passed successfully for "Eigenfaces"!
UnitTest passed successfully for "Projection"!
```

2.2 Eigenface Demonstration [No Points]

This question will utilize your implementation of the functions from Q2.1 to display the eigenfaces calculated for various images as well as the utility of PCA.

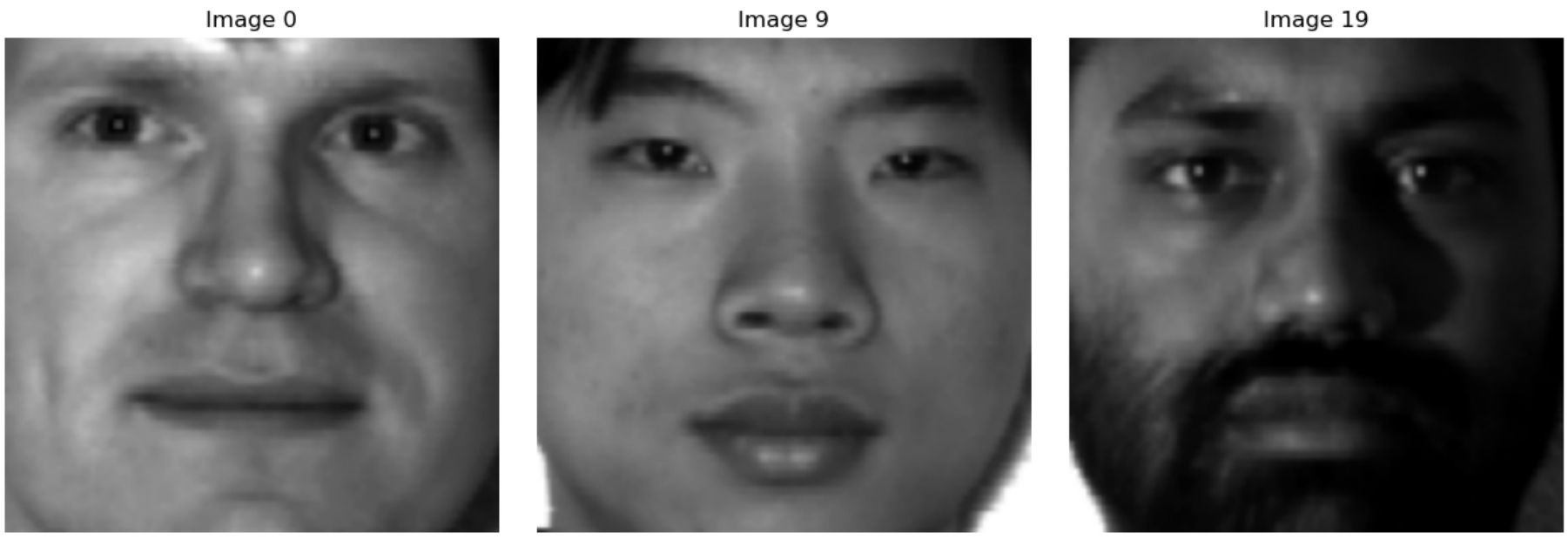
Running these cells is primarily for your own understanding of the compression process.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
from eigenfaces import Eigenfaces

ef = Eigenfaces()
num_eigenfaces = 5
eigenfaces = ef.compute_eigenfaces(faces, num_eigenfaces)
visualize = (eigenfaces - eigenfaces.min()) / (
    eigenfaces.max() - eigenfaces.min()
)  # normalize
fig, axes = plt.subplots(1, num_eigenfaces, figsize=(20, 7))

for i in range(num_eigenfaces):
    ax = axes[i]
    ax.imshow(visualize[i].reshape((275, 275)), cmap="gray")
    ax.axis("off")
    ax.set_title(f"Eigenface {i+1}")

plt.show()
```

We can then use these eigenfaces to get a dense representation of some images. By projecting each image onto the basis, we'll get a (k,) vector denoting similarity to each eigenface.

In [ ]:
```
# display
print("Our sample images:")
fig, axes = plt.subplots(1, len(selected_indices), figsize=(12, 4))
for i, ax in enumerate(axes):
    ax.imshow(selected_images[i], cmap="gray")
    ax.axis("off")
    ax.set_title(f"Image {selected_indices[i]}")
plt.tight_layout()
plt.show()

ef = Eigenfaces()
sample_data = faces[selected_indices]
eigenfaces = ef.compute_eigenfaces(faces, k=50)
feature_vectors = ef.project(sample_data, eigenfaces)
for i in range(len(selected_indices)):
    print(f"Image {selected_indices[i]} can be represented as {feature_vectors[i]}")
```

```
Our sample images:
```




```
Image 0 can be represented as [-2.86089449e+01 -1.76640677e+01 -1.04503647e+01  1.37568058e+00
 -2.12689260e+01  4.27039571e+00 -3.59353518e+00  4.00706345e+00
 -8.60613954e+00  6.08713460e+00 -1.22792431e+01  4.42094758e+00
  9.84974060e-01 -2.62042292e+00 -2.65759064e+00 -1.65555559e+00
 -5.07078164e-02 -1.37957001e+00  1.65442882e+00  1.49553373e+00
  7.05849393e-01  2.49904651e-01  5.52700308e-01  3.49944869e-01
 -2.22293017e-01 -7.85479861e-01 -2.42941925e-01 -4.14994052e+00
  4.60445235e+00  4.74420850e-14]
Image 9 can be represented as [-9.52459277e+00 -7.19217267e+00 -1.12319424e+01  3.60402176e+00
  4.15404129e+00  1.28204602e+01  6.08330883e+00  7.86538864e+00
  1.33000725e+01  1.11053084e+00 -2.11407515e+00 -3.99186859e+00
 -5.63881710e+00 -3.97143252e+00  2.78538750e-01 -1.70613353e+00
 -1.02297378e-01 -6.94918561e-01 -7.44641568e+00  1.13948840e+00
  5.29327128e+00 -2.35076224e+00  2.49263229e+00 -4.35325840e+00
  1.97673847e+00 -1.83017160e+00  1.73244201e+00  1.19234432e+00
  8.04966855e-01  3.13247692e-14]
Image 19 can be represented as [ 5.16959713e+00  5.77164657e+01 -1.55595038e+01 -6.73797942e+00
  6.50802219e+00 -6.50549874e+00 -4.59124016e+00 -4.23835914e+00
 -1.12608750e+00 -2.41650793e+00 -4.01740070e-01 -1.03355461e+00
  2.52417729e+00  3.17697635e+00  8.57675946e+00  1.14521533e+00
  2.51544426e+00 -2.83448978e+00 -1.08195992e+00  6.32103776e+00
  4.04528077e+00  3.99931571e+00 -3.50661921e-01 -2.37731352e+00
 -3.14518009e+00  2.81029085e+00  8.02481149e-01 -1.79999688e+00
  3.11563950e-01 -6.36851683e-14]
```

Q3: SVD Recommender [2.1% Bonus for All] [P] | [W]

In Q1 you used SVD to compress matrices and in Q2 you've seen the extension of SVD to PCA, which you used to build a dense and small feature space for images by finding a set of highly representational and uncorrelated features by maximizing variance.

In this section, we aim to introduce one more application of SVD. If we can manage to build a matrix decomposition when the matrix has missing values, then we can use the decomposition to just reconstruct the original matrix, hopefully getting some meaningful results where there used to be missingness. Let's 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.

How Can SVD Help?

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

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

Ratings in the matrix range from 1-5. In addition, the matrix contains `nan` wherever there is no rating provided by the user for the corresponding movie. The simplest way to predict ratings for a new viewer is simply to take the average rating of everyone who has rated the movie. If we know nothing about the new viewer, that's probably a good MLE, but we do have their other ratings. Supervised techniques can give us better results, but SVD can help right away.

Recall how we previously used SVD to compress images by throwing out less important information. While we can't apply SVD to a matrix with `nan`s in it, we can make a best guess for those `nan`s, then apply SVD, and remove the singular components that explain relatively little variance, then reconstruct. The signals that would be constructing the set of values that we set to the mean would explain little variance, so the low-rank approximation formed by our reconstruction may actually improve the guesses for new viewers.

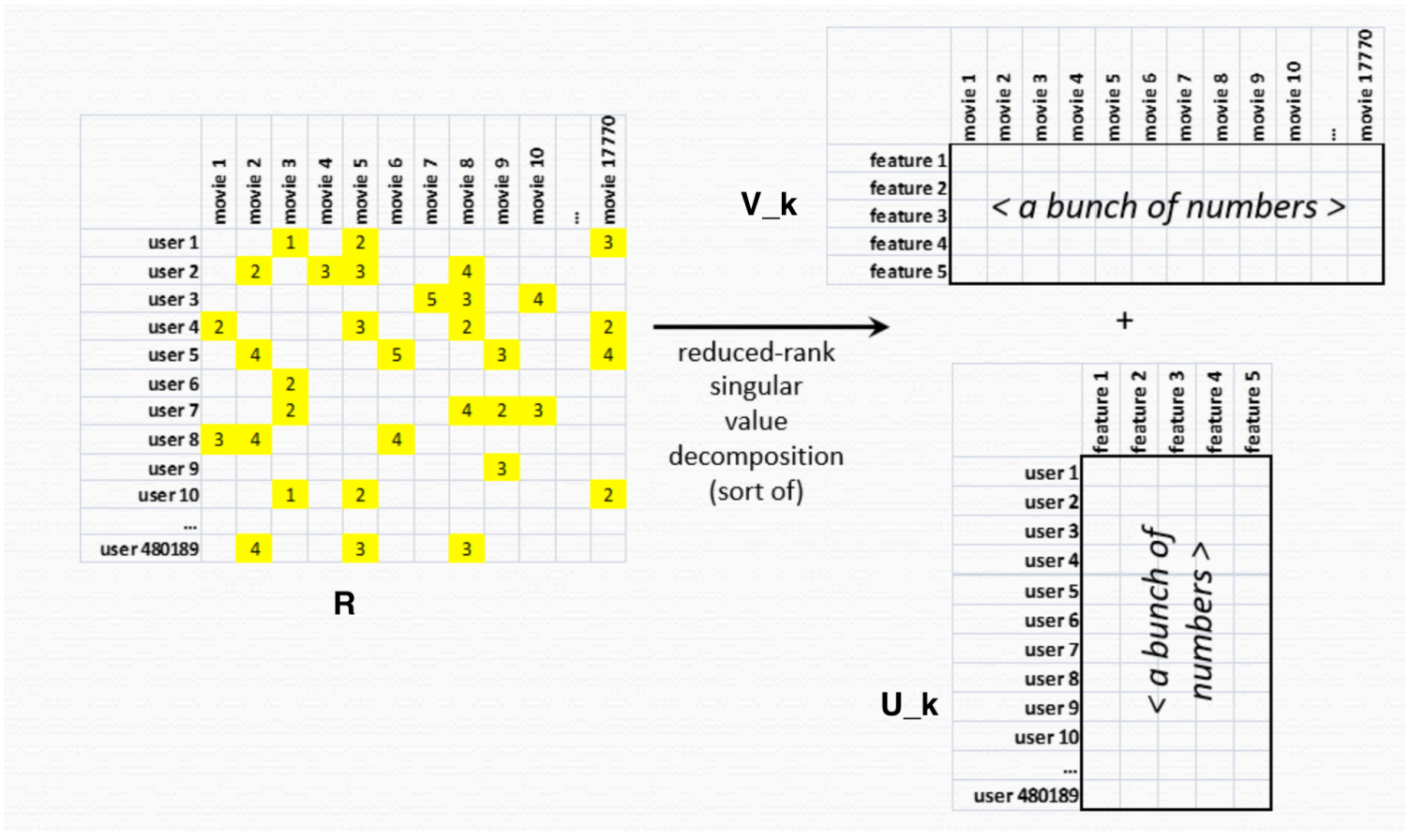
There's also some math based intuition here. We can start with the SVD decomposition.

$$
\begin{aligned}
R &= U\Sigma V^\top \\
R &= U\sqrt{\Sigma}\cdot\sqrt{\Sigma}V^\top \\
R &\approx U_{[:k]}\sqrt{\Sigma_{[:k]}}\cdot\sqrt{\Sigma_{[:k]}}V^\top_{[:k]}
\end{aligned}
$$

With this manipulation, we have 2 key terms:

- $U_k = U_{[:k]}\sqrt{\Sigma_{[:k]}}$
- $V_k = \sqrt{\Sigma_{[:k]}}V^\top_{[:k]}$

Consider the shapes of these matrices. R was (N, D). U_k is (N, k) and V_k is (k, D). We can visualize this as follows:



This technique is useful beyond just imputation, because it has also exposed our user space and the movie space to an arbitrary dimension feature space, which can itself be used in conjunction with other techniques. We can imagine each row of U_k to be holding some information how much each user likes a particular feature (feature 1, feature 2, ..., feature k). Similarly, we can imagine each column of V_k to be holding some information about how much each movie relates to the given features (feature 1, feature 2, ..., feature k).

Moving back to our purposes, though, we have simply that $U_k \cdot V_k \approx R$. To take a closer look, let u_i be the i^{th} row of U_k and

let v_j be the j^{th} column of V_k. Then the dot-product: $u_i \cdot v_j$ can provide us with information on how much a user *i* likes movie *j*.

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.

For a more detailed explanation and practical examples, you can check out this resource: Singular Value Decomposition (SVD) - GeeksforGeeks

Load data and print samples

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

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

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

```
------------------------------------------
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)
```

3.1 SVD Recommender for Movies [1.4% Bonus for All] [P]

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

- **recommender_svd**: Use the following equations to output U_k and V_k. You can utilize the `svd` method from Numpy and `compress` method from `audiocompression.py` to retrieve your initial U, Σ and V matrices. Then, calculate U_k and V_k based on the decomposition example above.
  - $U_k = U_{[:k]}\sqrt{\Sigma_{[:k]}}$
  - $V_k = \sqrt{\Sigma_{[:k]}}V^\top_{[:k]}$
- **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` **

3.1.1 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()
```

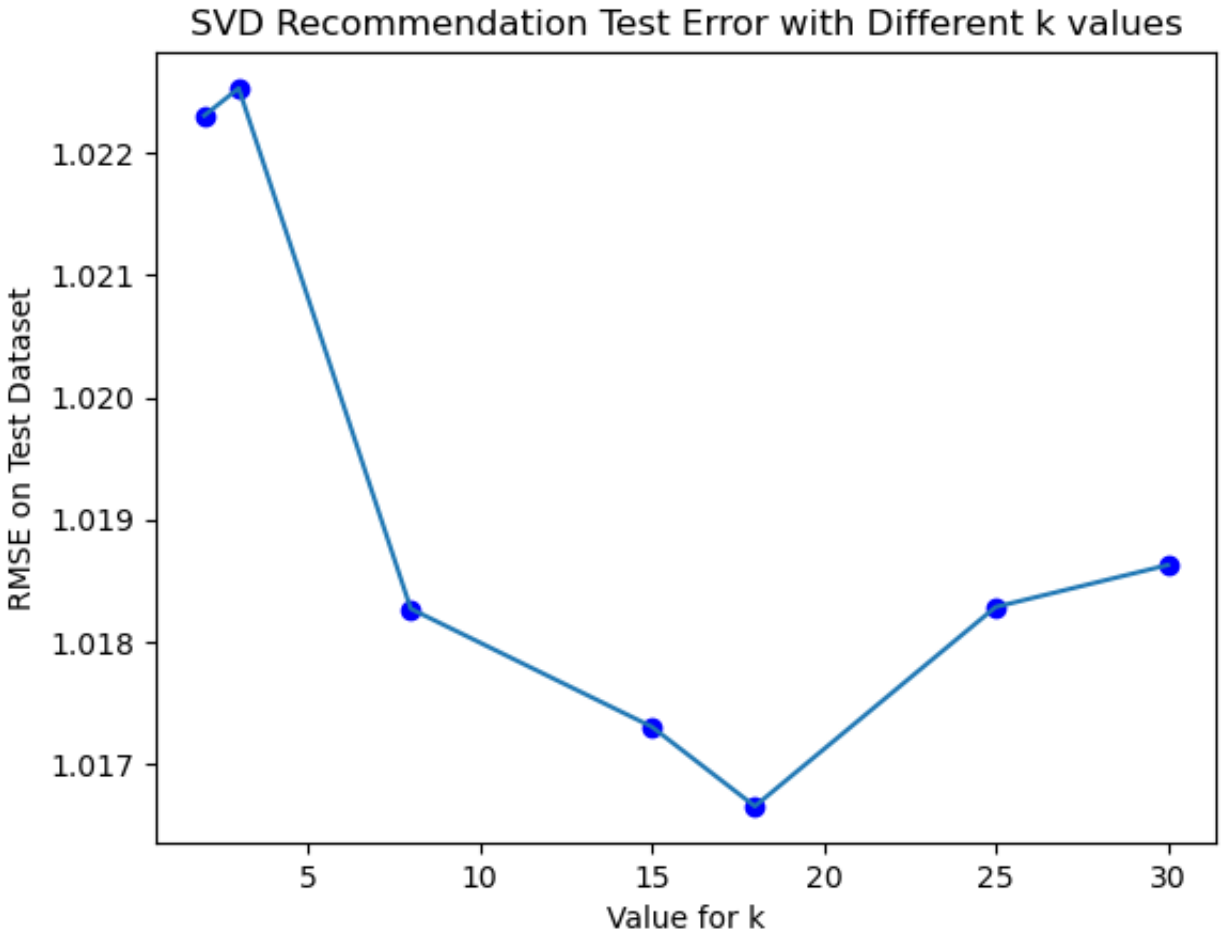
```
UnitTest passed successfully for "recommender_svd() function"!
```

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

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

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

```
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 --> 1.0166562048687975
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()
```



3.1.2 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()
```

3.2 Comparison: SVD vs. PCA [0.70% Bonus for All] [W]

[W]: Make sure the markdown cell containing your answer is displayed and properly assigned to Q3.2 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment.

Both PCA and SVD use the same mathematical backbone, but are used for different tasks.

Consider:

- When compressing audio spectrograms by eliminating singular components with SVD, we *did not* center the matrix.
- When calculating the eigenfaces of the face image dataset, we *did* center the matrix.
- When rebuilding the amputed movie rating matrix with SVD, we *did not* center the matrix.

Why does PCA involve centering the data but vanilla SVD does not?

Answer

The key difference lies in the goal. PCA is a statistical method that must center the data by subtracting the mean, as its purpose is to find the principal components that capture the maximum variance around that mean. Without centering, the first component just points to the average of the data, not its variance. Vanilla SVD, when used for compression, is a tool where the goal is to reconstruct the original matrix as accurately as possible; centering isn't necessary and would actually be harmful, as we want to rebuild the original absolute values (like audio amplitudes), not their deviation from an average.

Q4: Polynomial Regression and Regularization [65pts + 30pts Grad/6% Bonus for Undergrad + 2.3% Bonus for All] [P] | [W]

4.1 About RMSE [5pts] [W]

In a regression task, we attempt to predict target values. In this context, accuracy or entropy loss measures can't apply, so we use error functions to measure how far off the predicted value is from the target.

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

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

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.

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

[W]: Make sure the markdown cell containing your answer is displayed and properly assigned to Q4.1 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. Please format your work with LaTeX.

Suppose we have some data with the following target values:

$$y = [0.1, 0.3, 0.7, 0.8]$$

Now, for a regression model that is only capable of producing numbers in the range $[0.0, 1.0]$, **what is the highest possible RMSE it could achieve?**

Then, **what is the lowest possible RMSE it could achieve?**

Show your work and round your RMSE's to 3 decimal places where necessary. [5pts]

Answer

Highest Possible RMSE

To find the highest possible RMSE, we must find the prediction $\hat{y}_i$ in the allowed range $[0.0, 1.0]$ that is **farthest** from each true value y_i.

- For $y_1 = 0.1$, the farthest prediction in $[0.0, 1.0]$ is $\hat{y}_1 = 1.0$.
- For $y_2 = 0.3$, the farthest prediction in $[0.0, 1.0]$ is $\hat{y}_2 = 1.0$.
- For $y_3 = 0.7$, the farthest prediction in $[0.0, 1.0]$ is $\hat{y}_3 = 0.0$.
- For $y_4 = 0.8$, the farthest prediction in $[0.0, 1.0]$ is $\hat{y}_4 = 0.0$.

The vector of worst-case predictions is $\hat{y}_{worst} = [1.0, 1.0, 0.0, 0.0]$.

$$\text{MSE}_{max} = \frac{1}{4}\left[(0.1 - 1.0)^2 + (0.3 - 1.0)^2 + (0.7 - 0.0)^2 + (0.8 - 0.0)^2\right]$$

$$\text{MSE}_{max} = \frac{1}{4}\left[(-0.9)^2 + (-0.7)^2 + (0.7)^2 + (0.8)^2\right]$$

$$\text{MSE}_{max} = \frac{1}{4}[0.81 + 0.49 + 0.49 + 0.64]$$

$$\text{MSE}_{max} = \frac{1}{4}[2.43] = 0.6075$$

$$\text{RMSE}_{max} = \sqrt{0.6075} \approx 0.77942...$$

The highest possible RMSE is **0.779**.

Lowest Possible RMSE

To find the lowest possible RMSE, we must find the prediction $\hat{y}_i$ in the allowed range $[0.0, 1.0]$ that is **closest** to each true value y_i.

Since every target value y_i in the set $y = [0.1, 0.3, 0.7, 0.8]$ is *already* within the model's allowed output range of $[0.0, 1.0]$, the model can achieve a perfect prediction.

The vector of best-case predictions is $\hat{y}_{best} = [0.1, 0.3, 0.7, 0.8]$.

$$\text{MSE}_{min} = \frac{1}{4}\left[(0.1-0.1)^2 + (0.3-0.3)^2 + (0.7-0.7)^2 + (0.8-0.8)^2\right]$$

$$\text{MSE}_{min} = \frac{1}{4}\left[0^2 + 0^2 + 0^2 + 0^2\right]$$

$$\text{MSE}_{min} = 0$$

$$\text{RMSE}_{min} = \sqrt{0} = 0$$

The lowest possible RMSE is **0.000**.

4.2 Regression Implementation [30pts + 30pts Grad/6% Bonus for Undergrad] [P]

We have four methods to fit linear and ridge regression models:

1. closed form solution
2. gradient descent (GD)
3. stochastic gradient descent (SGD)
4. mini-batch gradient descent (MBGD)

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, SGD, and MBGD 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**):

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

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

$$L_{\text{linear, MBGD}}(\theta) = \frac{1}{2B}\sum_{j}^{j+B}\left[y_j - \hat{y}_j(\theta)\right]^2 \qquad y_j = \text{label},\ \hat{y}_j(\theta) = \text{prediction}$$

  where B is the batch size and j is the starting index of the current batch.

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

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

- **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`, `ridge_fit_SGD`, and `ridge_fit_MBGD`, 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.

4.2.1 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"!
```

4.2.2 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"!
```

4.2.3 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"!
```

4.2.4 Local Tests for Gradient Descent (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_linear_fit_MBGD()
unittest_reg.test_ridge_fit_GD()
```

```
unittest_reg.test_ridge_fit_SGD()
unittest_reg.test_ridge_fit_MBGD()
```

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

4.3 Testing: General Functions and Linear Regression [10pts] [P] | [W]

In this section we will test the RMSE performance of the linear regression. Your test RMSE score should be close to the TA's answer, but the Gradescope test (5pts) for this should have some leniency, since there may be some slight differences in implementation. The actual cells here in the notebook will not be used, but the test cases will have the same setup, so this is a good place to debug.

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.

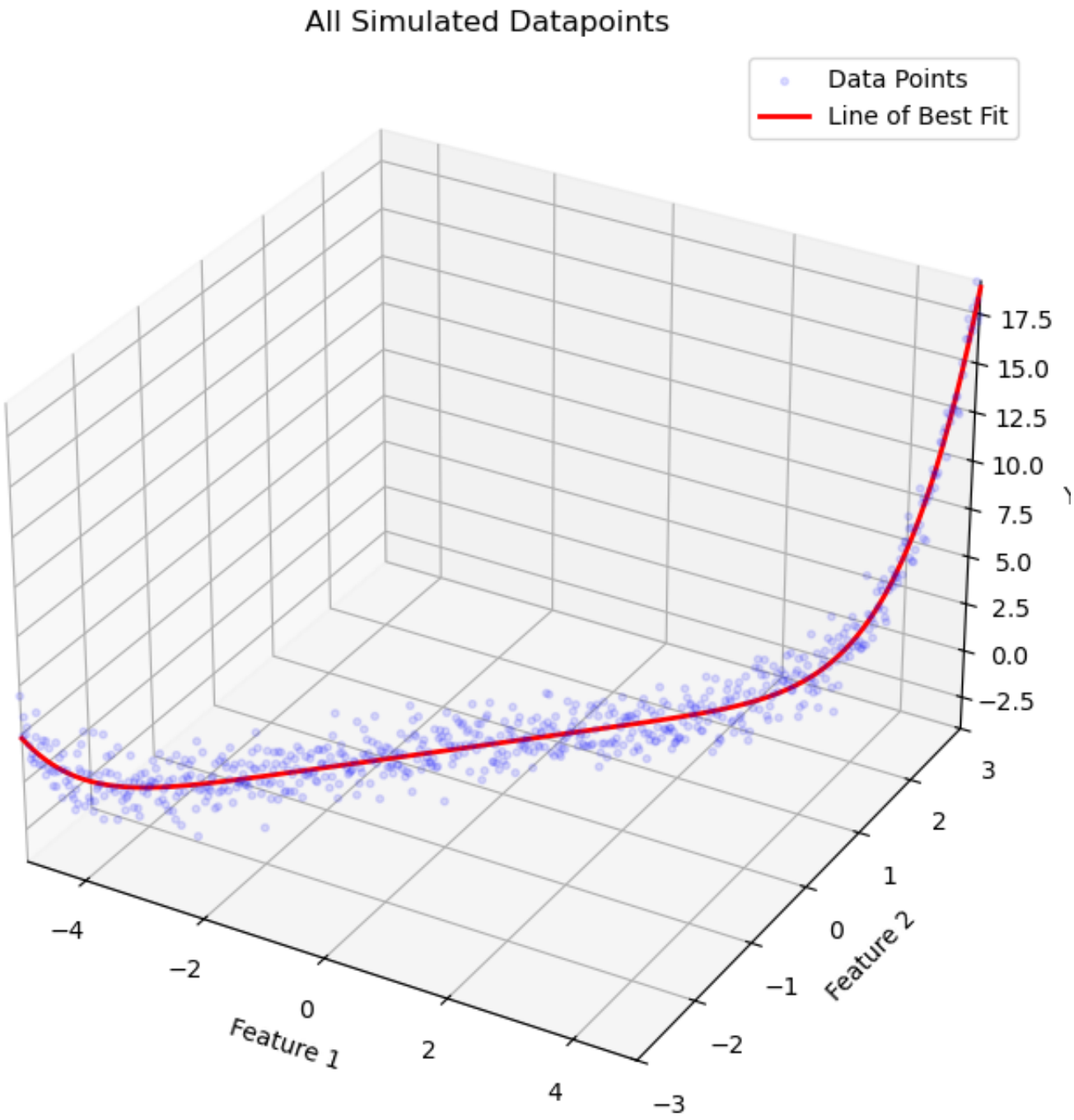
[W]: We recommend including the following code cells and their outputs in your submission for Q4.3 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. While these code cells are not graded, they may be used to give you partial credit in the case of an incorrect written answer.

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)
```

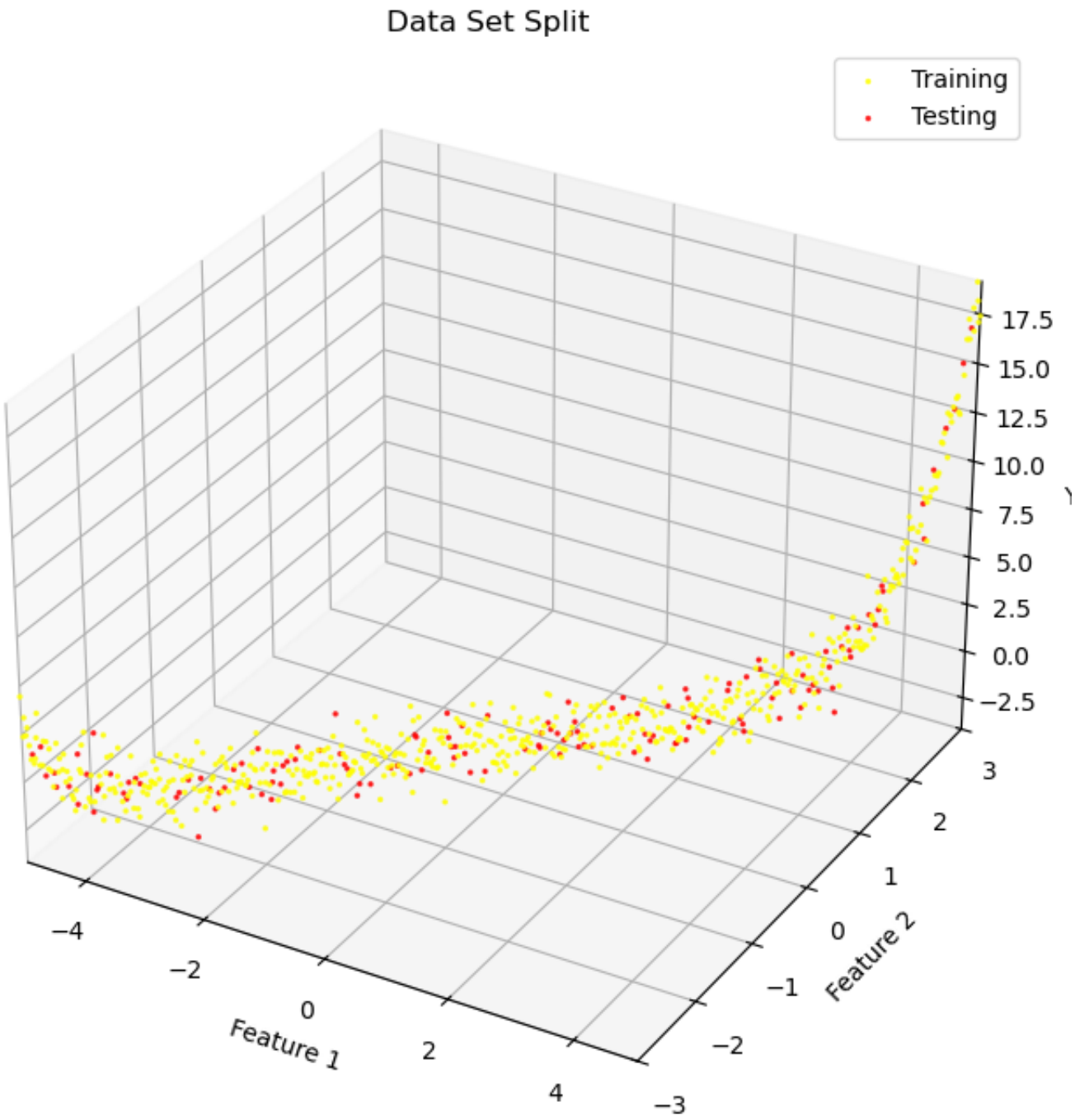
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# 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.

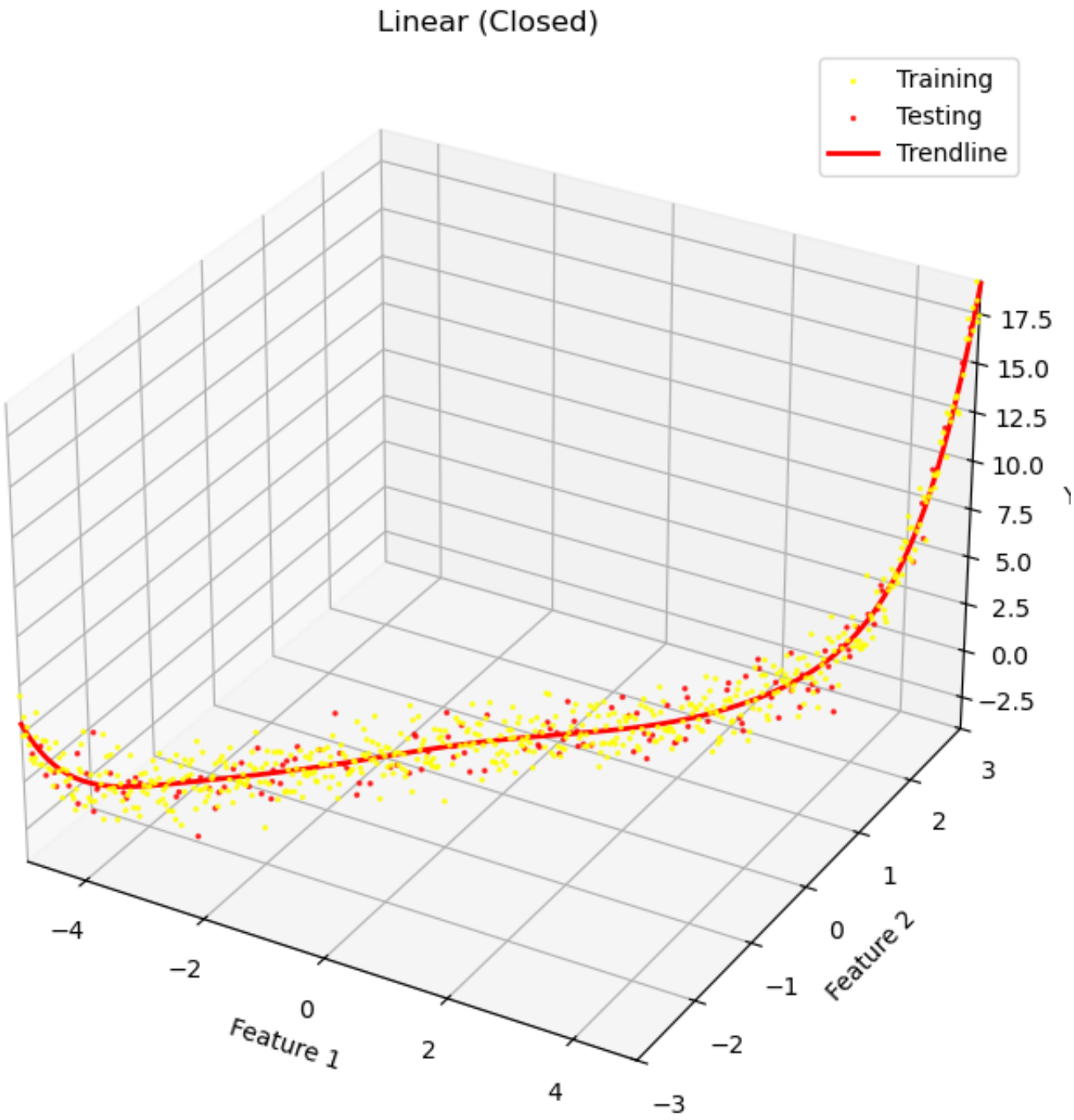
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

xtrain, ytrain, xtest, ytest, train_indices, test_indices = plotter.split_data(
    x_all, y_all
)

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



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

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

# Required for both Grad and Undergrad

weight = reg.linear_fit_closed(x_all_feat[train_indices], y_all[train_indices])
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
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
```

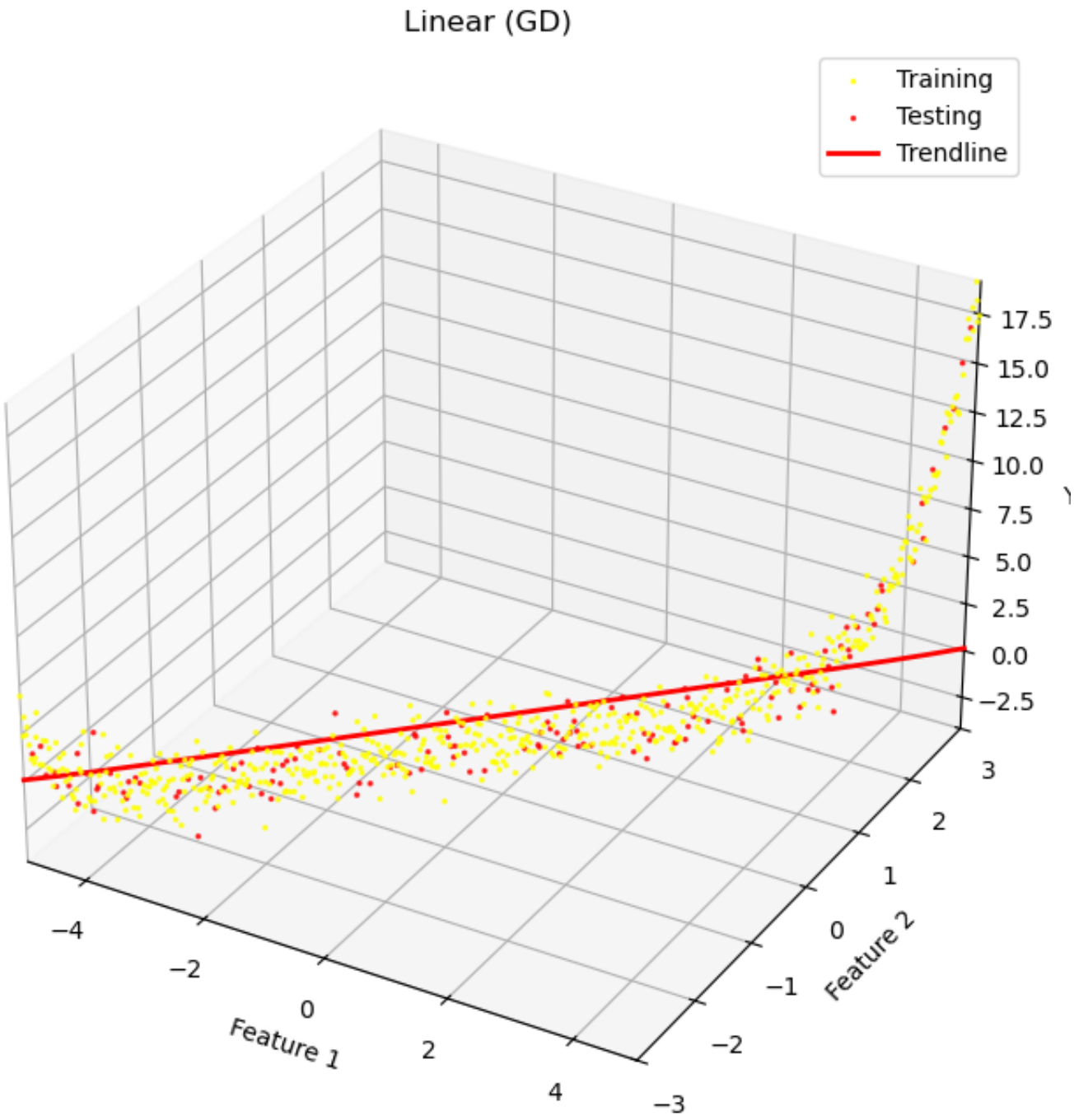


HINT: If your RMSE is off, make sure to follow the instruction given for `linear_fit_closed` in the list of functions to implement above. (Note: this content is autograded separately on Gradescope, but you may also compare to the expected outputs)

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.

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

# Required for Grad Only
# This cell may take more than 1 minute
weight, _ = reg.linear_fit_GD(
    x_all_feat[train_indices], y_all[train_indices], epochs=50000, learning_rate=1e-8
)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all[test_indices])
print("Linear (GD) RMSE: %.4f" % test_rmse)
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
```

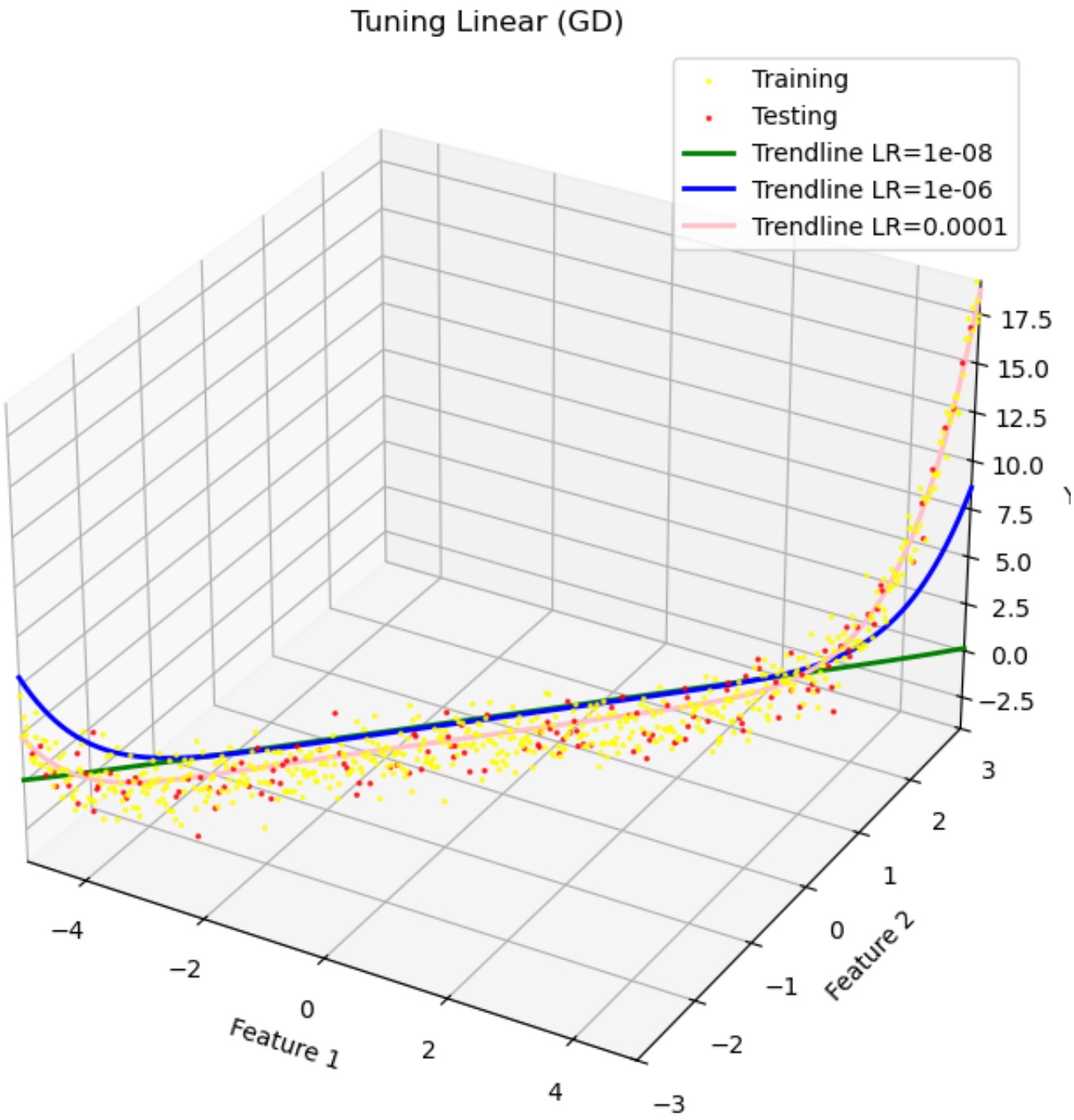


Don't be alarmed. We still have to tune our epochs and learning_rate. As we tune these parameters our trendline will approach the trendline generated by the linear closed form solution. Observe how we slowly tune (increase) the epochs and learning_rate below to create a better model.

Note that the closed form solution will always give the most optimal/overfit results. We cannot outperform the closed form solution on the training set, since the closed form is an optimal algorithm. We leave the reasoning behind this as an exercise to the reader.

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

# Required for Grad Only
# This cell may take more than 1 minute

learning_rates = [1e-8, 1e-6, 1e-4]
weights = np.zeros((3, POLY_DEGREE**2 + 2))

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

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)
```



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

# Required for Grad Only
# This cell may take more than 1 minute

n_samples = 5000
n_features = 5000
train_frac = 0.8

X, y = make_regression(
    n_samples=n_samples, n_features=n_features, noise=0.1, random_state=42
)
y = y.reshape(-1, 1)
X_bias = np.hstack([np.ones((n_samples, 1)), X])


# Train/test split
n_train = int(train_frac * n_samples)
indices = np.random.permutation(n_samples)
train_indices2, test_indices2 = indices[:n_train], indices[n_train:]
X_train, X_test = X_bias[train_indices2], X_bias[test_indices2]
y_train, y_test = y[train_indices2], y[test_indices2]

# --- Closed form ---
start = time.time()
w_closed = reg.linear_fit_closed(X_train, y_train)
end = time.time()
train_pred_closed = reg.predict(X_train, w_closed)
test_pred_closed = reg.predict(X_test, w_closed)
train_rmse_closed = reg.rmse(train_pred_closed, y_train)
test_rmse_closed = reg.rmse(test_pred_closed, y_test)
print(f"Closed Form training time: {end - start:.6f}s")
print(f"Closed Form RMSE (train): {train_rmse_closed:.6f}")
print(f"Closed Form RMSE (test):  {test_rmse_closed:.6f}")

# --- Gradient descent ---
start = time.time()
w_gd, _ = reg.linear_fit_GD(X_train, y_train, epochs=100, learning_rate=0.03)
end = time.time()
train_pred_gd = reg.predict(X_train, w_gd)
test_pred_gd = reg.predict(X_test, w_gd)
```

```
train_rmse_gd = reg.rmse(train_pred_gd, y_train)
test_rmse_gd = reg.rmse(test_pred_gd, y_test)
print(f"GD training time: {end - start:.6f}s")
print(f"GD RMSE (train): {train_rmse_gd:.6f}")
print(f"GD RMSE (test):  {test_rmse_gd:.6f}")
```

```
Closed Form training time: 29.472929s
Closed Form RMSE (train): 0.000000
Closed Form RMSE (test):  71.856565
GD training time: 1.166028s
GD RMSE (train): 18.999037
GD RMSE (test):  90.670957
```

Analysis [5pts]

[W]: Make sure the markdown cells containing your answers are displayed and properly assigned to Q4.3 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment.

Note: although this question requires you to comment on the behavior of gradient descent, which is optional for undergrads, this written question is still required for all. You should be able to make claims about the behavior of gradient descent even without having implemented it. If you'd like to see the output of the gradient descent functions and their behavior without implementing them, you can consult the expected output for the content of the above cells.

Though gradient descent has hyperparameters and may result it poorer fit fidelity, it's still used quite frequently. There are two potential reasons for you to evaluate here.

1. Which form of linear regression (closed or GD) performed faster? Why might that be the case? **(3 pts)**
2. Which form of linear regression (closed or GD) would take up more memory? Why might that be the case? **(2 pts)**

1. **Answer** ...

Based on the output, Gradient Descent (GD) performed significantly faster (1.17 seconds) than the closed-form solution (29.47 seconds). This is due to the difference in computational complexities of both the approaches. The closed-form solution's runtime is dominated by the matrix inversion, which has a high polynomial complexity relative to the number of features (D≈5001), roughly O(D^3). In contrast, GD's complexity is O(kND), scaling linearly with the number of epochs (k), samples (N), and features (D). In general high-dimensional scenarios, the iterative O(kND) calculation is far more efficient than the one-time O(D^3) computation.

2. **Answer** ...

The closed-form solution would take up significantly more memory. To solve for the weights, this method must first compute and store the intermediate X^TX matrix. Since the data has D=5001 features (including bias), this requires allocating memory for a massive (5001×5001) matrix. Gradient Descent, on the other hand, is much more memory-efficient. It only needs to store the original (N×D) data and the (D×1) weight vector, performing its calculations using matrix-vector products without ever creating the enormous (D×D) matrix, and inverting it.

Motivation

What if we were data limited? Suppose we just had the first 10 data points to train?

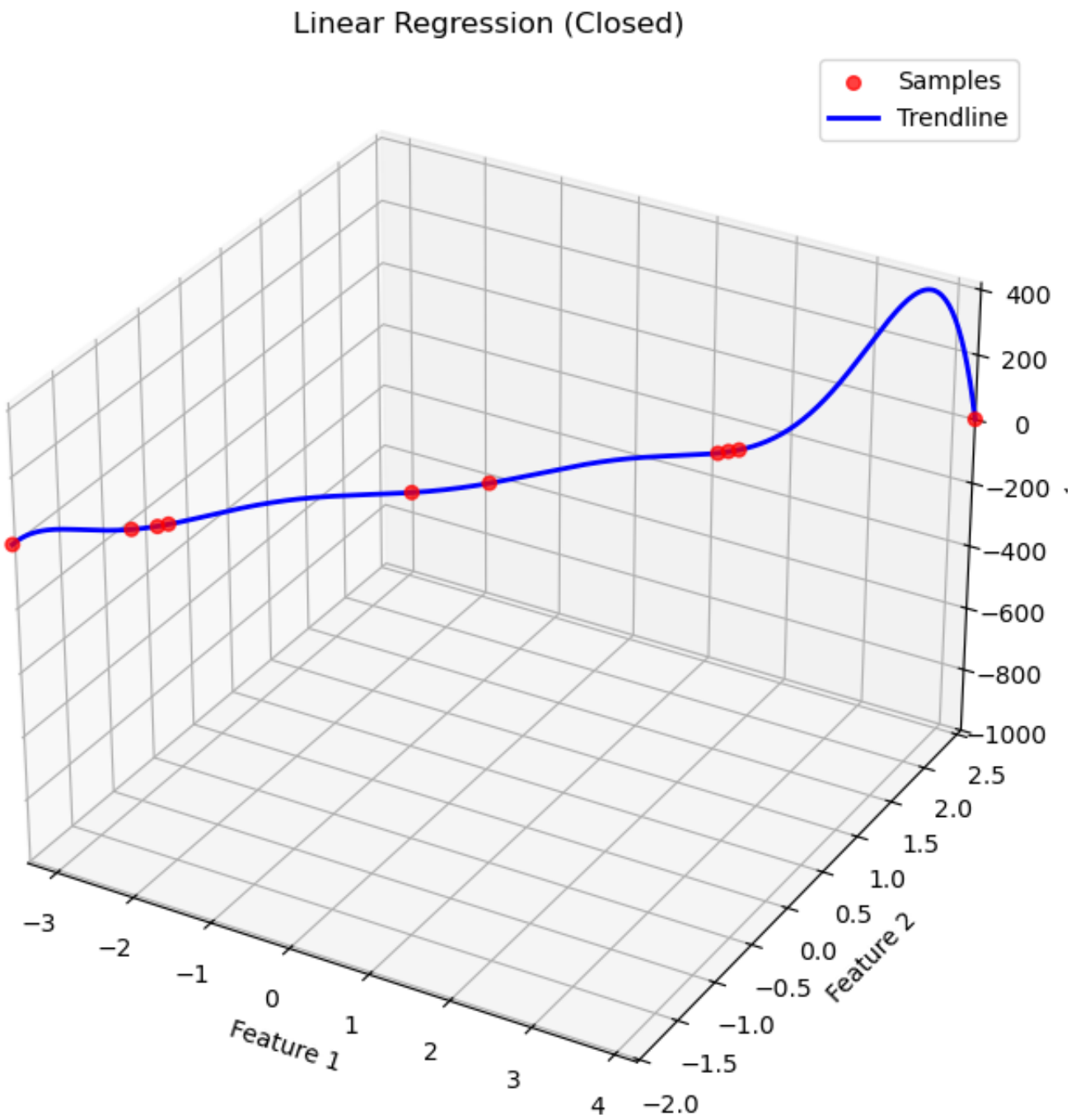
Note: Due to the large RMSE values, rounding errors may result in larger than normal differences from the TA solution. Thus, the autograder has a wider acceptance bound commensurate to that wider variance.

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.randn(
    x_all_feat.shape[0], 1
)
sub_train = train_indices[10:20]
```

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

# Required for both Grad and Undergrad

weight = reg.linear_fit_closed(x_all_feat[sub_train], y_all_noisy[sub_train])
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
```

```
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print("Linear (closed) 10 Samples RMSE: %.4f" % test_rmse)

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

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



You should see a perfect train fit (due to a lack of model bias), but horrible test accuracy (due to very high model variance). This is overfitting.

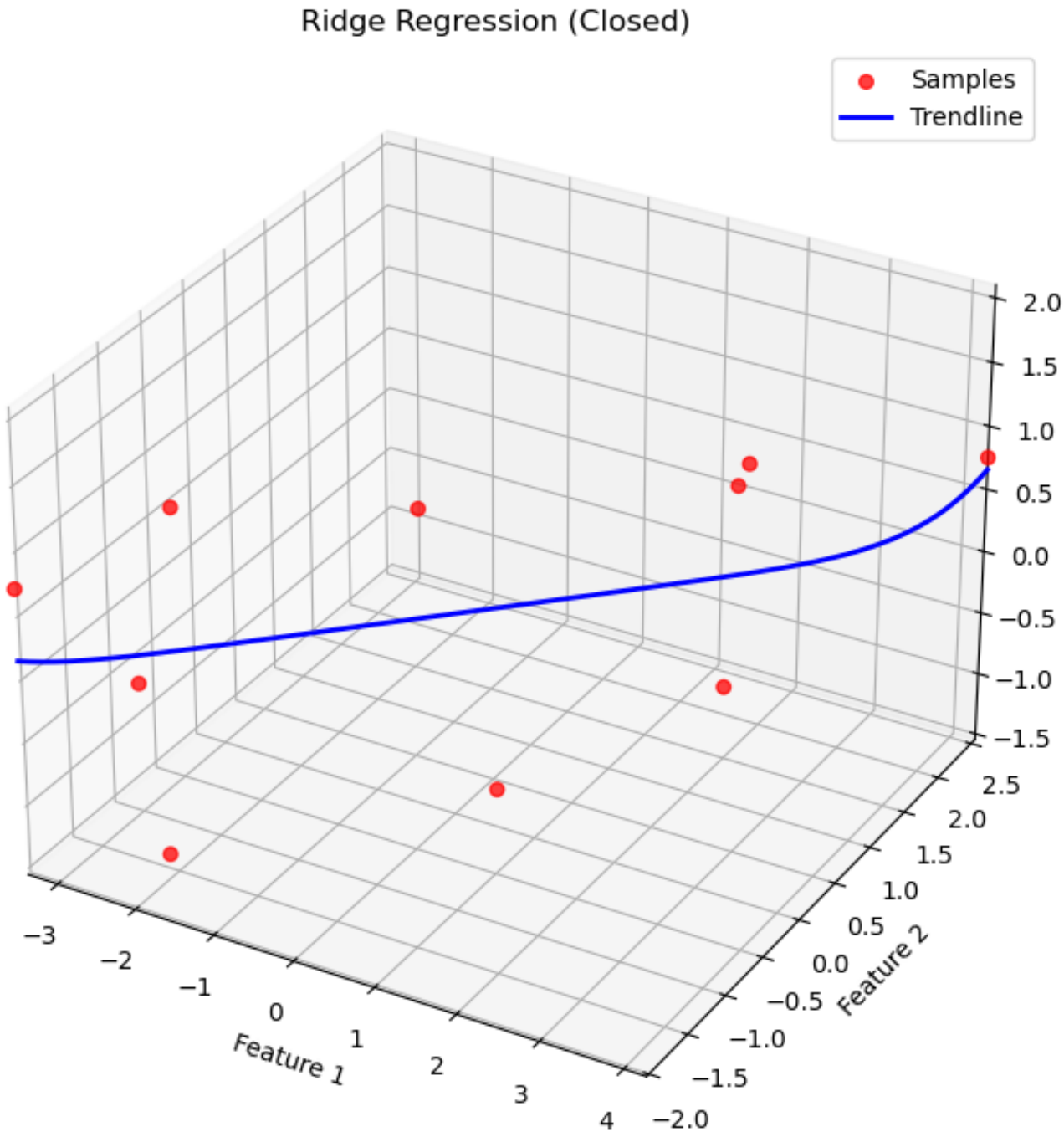
4.4 Testing: Ridge Regression [5pts] [P]

Regularization is a very common way to improve the fit of an overfitting model. Ridge regression is one such technique. By placing a penalty on large coefficients, we can reign in model variance without impacting expressibility.

Note: Again, similar to 4.3's programming part, the content covered in these cells are going to be autograded separately. The figure outputs of these cells should match the expected outputs, but the figures themselves will not be graded. This section acts as a local test, and your grade will be determined by the test cases in the autograder.

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

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

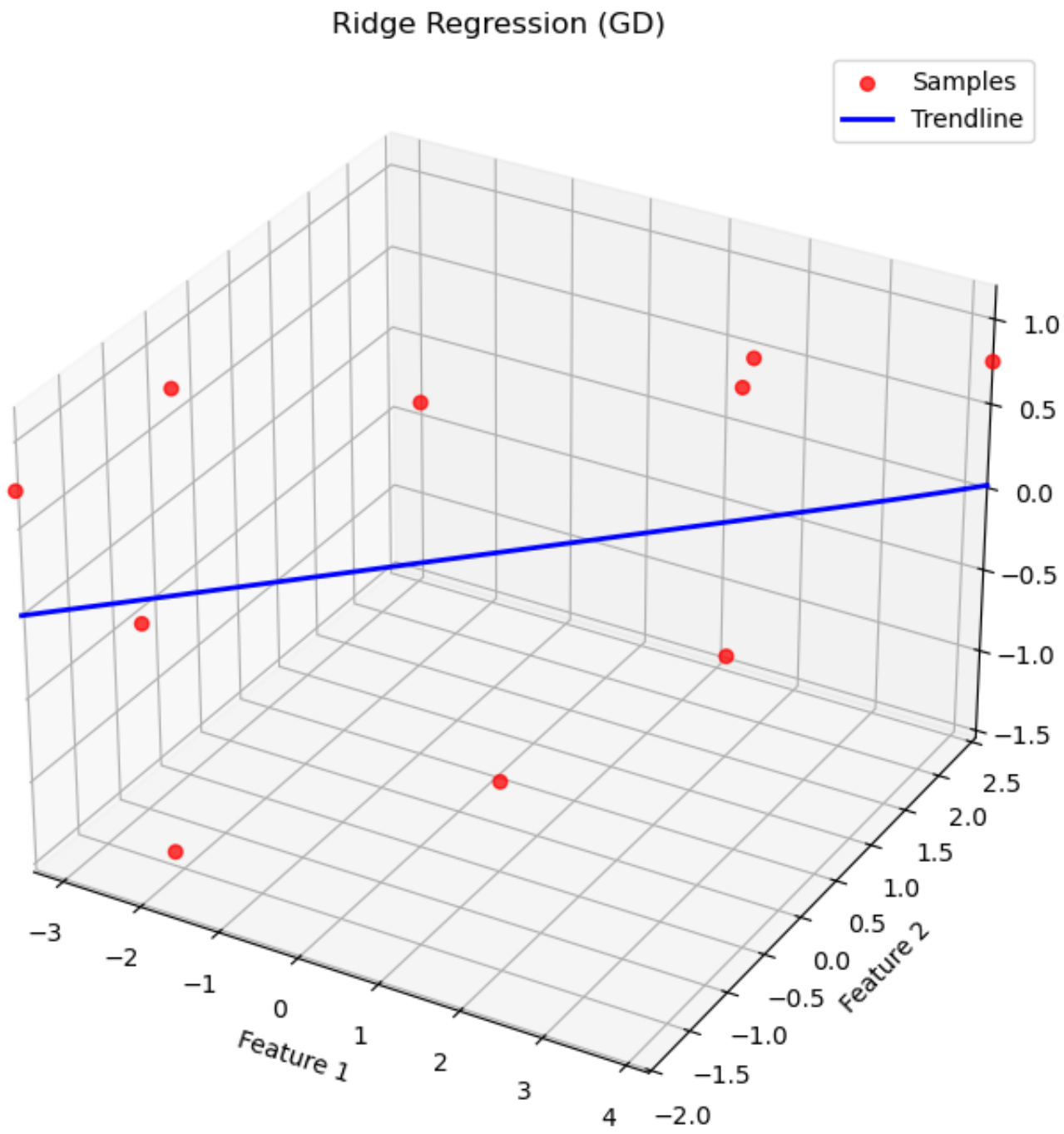
```
Ridge Regression (closed) RMSE: 1.1193
```



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

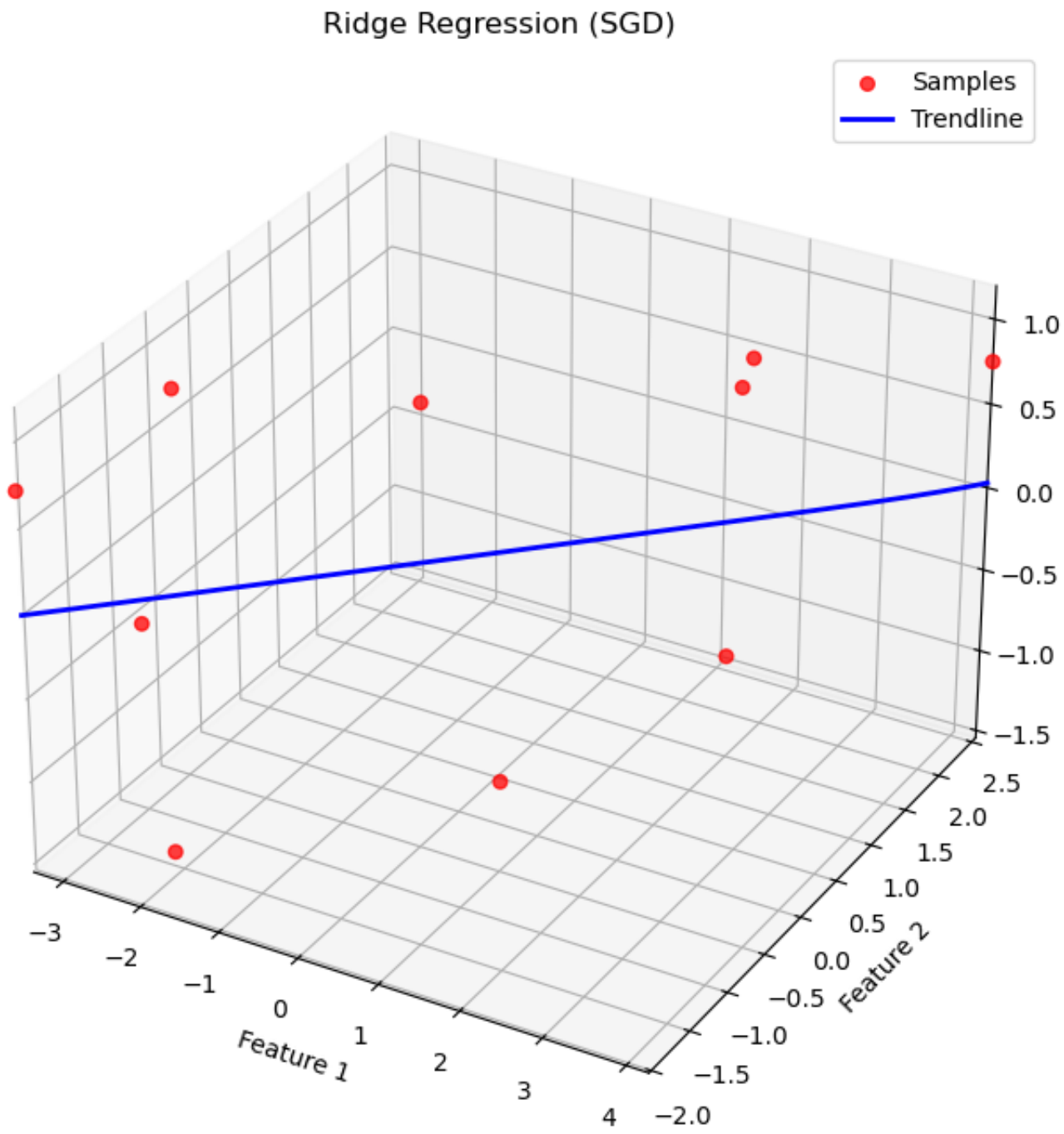
In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################

# Required for Grad Only

weight, _ = reg.ridge_fit_GD(
    x_all_feat[sub_train], y_all_noisy[sub_train], c_lambda=20, learning_rate=1e-5
)
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print("Ridge Regression (GD) RMSE: %.4f" % test_rmse)

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

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



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

# Required for Grad Only

weight, _ = reg.ridge_fit_SGD(
    x_all_feat[sub_train], y_all_noisy[sub_train], c_lambda=20, learning_rate=1e-5
)
y_pred = reg.predict(x_all_feat, weight)
y_test_pred = reg.predict(x_all_feat[test_indices], weight)
test_rmse = reg.rmse(y_test_pred, y_all_noisy[test_indices])
print("Ridge Regression (SGD) RMSE: %.4f" % test_rmse)

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

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



4.5 The Power of Ridge Regression [10pts] [W]

[W]: Make sure the markdown cells containing your answers are displayed and properly assigned to Q4.5 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. Please format your work with LaTeX.

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? **(2 pts)**
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. Describe specific properties of the methods that explain one is better than the other in each case. **(3 pts)**
3. One weakness of linear regression is that it requires the pseudo-inverse of X, namely, $\theta = (X^T X)^{-1} X^T y$, which is problematic since highly correlated features may cause the inverse of $X^T X$ to not exist or be numerically instable. Ridge regression avoids this my adding a term to the solution: $\theta = (X^T X + \lambda I)^{-1} X^T y$ for some $\lambda > 0$. *Prove* that, for any X, $X^T X + \lambda I$ is invertible (with $\lambda > 0$). **(5 pts)**

Hint: If and only if $v^T A v > 0$ for all vectors $v \neq \mathbf{0}$, then A is considered to be "positive definite."

Hint: Every positive definite matrix is invertible.

Note: 4.5.3 asks you to prove your result. Recall that this requires you to present your work formally.

1. **Answer** ...

Ridge regression's RMSE is dramatically lower (1.1193 vs. 1816.38) because it mitigates the issue of severe overfitting seen in the standard linear model. With only 10 samples and a high-degree polynomial feature set, the linear model has far more features than data points, allowing it to fit the random noise in the training samples. This causes the model's weights to explode to massive values, resulting in an extremely high-variance weight function that fails to generalize to the test data. Ridge regression adds a regularization penalty (λθ^Tθ) that punishes large weights, forcing the model to find a "simpler," smoother, and more stable solution that performs far better on unseen data.

2. **Answer** ...

Scenario where Linear Regression is more suitable: Standard Linear regression is preferable when the number of samples (N) is significantly larger than the number of features (D) and the features are not highly correlated. A good application would be predicting house prices (N=1,000,000 sales) from a few core, independent features (D=5) like square-footage, number-of-bedrooms, and location. In this N≫D case, overfitting is not a major risk, and the X^TX matrix is stable and invertible. Adding a ridge penalty here would be unnecessary and would only introduce a small amount of bias into the weights, slightly hurting the model's accuracy.

Scenario where Ridge Regression is more suitable: Ridge regression is the better choice when the number of features (D) is quite large, relative to the number of samples (N), or when features are highly correlated. A classic example is an example like predicting a patient's disease risk (N=200 patients) based on the expression levels of thousands of different genes (D=10,000 genes). Here, D>>N, which makes standard linear regression a bad choice since it guarantees it will overfit. The λ penalty in ridge regression makes the X^TX + λI matrix invertible and, more importantly, shrinks the model's weights to prevent this overfitting, allowing for a stable and predictive model.

3. **Answer** ...

To prove that the matrix $A = X^T X + \lambda I$ is invertible, we can show that A is **positive definite**, given that $\lambda > 0$. A matrix A is positive definite if $v^T A v > 0$ for all non-zero vectors v.

Let v be any non-zero vector (i.e., $v \neq \mathbf{0}$).

$$v^T A v = v^T (X^T X + \lambda I) v$$

$$v^T A v = v^T (X^T X) v + v^T (\lambda I) v$$

Using the property $(Xv)^T = v^T X^T$, we can rewrite this as:

$$v^T X^T X v = (Xv)^T (Xv)$$

This is the dot product of the vector (Xv) with itself, which is its squared L2-norm: $||Xv||_2^2$. The squared norm of any real vector is always non-negative, so $||Xv||_2^2 \geq 0$. This term is *positive semi-definite*.

Since λ is a scalar and I is the identity matrix, the second term, $v^T (\lambda I) v$, simplifies to:

$$v^T (\lambda I) v = \lambda v^T (Iv) = \lambda v^T v$$

Now, $v^T v$ is the squared L2-norm of v: $||v||_2^2$.

Substituting these back into the expression:

$$v^T A v = ||Xv||_2^2 + \lambda ||v||_2^2$$

We know $||Xv||_2^2 \geq 0$. We are given that $\lambda > 0$. Since we assumed $v \neq \mathbf{0}$, its squared norm must be strictly positive: $||v||_2^2 > 0$. Therefore, the second term $\lambda ||v||_2^2$ is the product of two strictly positive numbers, making it *strictly positive* ($\lambda ||v||_2^2 > 0$).

The full expression $v^T A v$ is the sum of a non-negative number (≥ 0) and a strictly positive number (> 0). This sum must be strictly positive.

$$v^T A v > 0$$

Since $v^T A v > 0$ for all $v \neq \mathbf{0}$, the matrix $A = X^T X + \lambda I$ is, by definition, *positive definite*. Since every positive definite matrix is invertible, we prove our hypothesis.

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

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

Imagine we have a training 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 "validate" with [1,2]
- The second iteration we would train with [1,2,5,6,7,8,9,10] and "validate" with [3,4]
- The third iteration we would train with [1,2,3,4,7,8,9,10] and "validate" with [5,6]
- The fourth iteration we would train with [1,2,3,4,5,6,9,10] and "validate" with [7,8]
- The fifth iteration we would train with [1,2,3,4,5,6,7,8] and "validate" with [9,10]

Then, we can consider the performance across all folds and (usually) average them to make decisions about our choice of hyperparameters.

Note, here, that we do not use the testing set. Using the testing set to pick the best hyperparameters ruins the generalizability of your result. You've simply shown that those hyperparameters are a good choice for your test data. You must cut off some chunk of your training data to do hyperparameter cross-validation.

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 = [0.01, 0.1, 1, 5, 10, 50, 100]
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, "Average Validation RMSE: %.6f" % err)

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

```
Lambda: 0.0100 Average Validation RMSE: 0.987995
Lambda: 0.1000 Average Validation RMSE: 0.989441
Lambda: 1.0000 Average Validation RMSE: 0.987945
Lambda: 5.0000 Average Validation RMSE: 0.986684
Lambda: 10.0000 Average Validation RMSE: 0.986821
Lambda: 50.0000 Average Validation RMSE: 0.989110
Lambda: 100.0000 Average Validation RMSE: 0.994419
Best Lambda: 5.0000
Best Test RMSE: 1.0508
```

4.7 Noisy Input Samples in Linear Regression [2.3% Bonus for All] **[W]**

[W]: Make sure the markdown cells containing your answers are displayed and properly assigned to Q4.7 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. Please format your work with LaTeX.

Consider a linear model of the form:

$$\hat{y}(\vec{x}_i, \theta) = \theta_0 + \sum_{d=1}^{D} \theta_d x_{id}$$

where $\vec{x}_i = (x_{i1}, \ldots, x_{iD}) \in \mathbb{R}^D$ and weights $\theta = \{\theta_0, \ldots, \theta_D\} \in \mathbb{R}^{D+1}$. Given the the D-dimensional input data set $X = \{\vec{x_1}, \ldots, \vec{x_N}\}$ with corresponding target values $y = \{y_1, \ldots, y_N\}$, the sum-of-squares error function for the model is defined as:

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

Now, suppose that Gaussian noise $\vec{\epsilon_n} \in \mathbb{R}^D$ is added independently to each of the input samples $\vec{x_n}$ to generate a new data set $X' = \{\vec{x_1}', \ldots, \vec{x_N}'\} = \{\vec{x_1} + \vec{\epsilon_1}, \ldots, \vec{x_N} + \vec{\epsilon_N}\}$. To be specific, $\vec{x_k}' = (x_{k1} + \epsilon_{k1}, \ldots, x_{kD} + \epsilon_{kD})$. We define the

Gaussian noise to be zero mean (i.e., $E[\epsilon_{ij}] = 0$), with variance σ^2 (i.e., $\mathsf{Var}(\epsilon_{ij}) = \sigma^2$), and each individual noise term is independent and identically distributed (i.i.d.).

1. (0.7% Bonus) Show that $\hat{y}(\vec{x_n}', \theta) = \hat{y}(\vec{x_n}, \theta) + \sum_{d=1}^{D} \theta_d \epsilon_{nd}$

2. (1.6% Bonus) Denote the sum-of-squares error function of the noised data set X' as $E_D(\theta)'$. Prove that the expectation of $E_D(\theta)'$ is equivalent to the sum-of-squares error for noise-free input data, $E_D(\theta)$, with the addition of some term which resembles the ℓ_2 norm of θ, in which the bias parameter θ_0 is omitted, that is, a "weight-decay regularizer." In other terms, prove that

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

Write your responses below using LaTeX in Markdown cells.

HINTS:

- You should use your answer from part 1 in your answer for part 2.
- During the class, we have discussed how to solve for the weight θ for ridge regression, the function looks like this:

$$E(\theta) = \frac{1}{N} \sum_{n=1}^{N} [y(\vec{x_n}, \theta) - y_i]^2 + \frac{\lambda}{N} \sum_{i=1}^{D} \theta_i^2$$

  where the first term is the sum-of-squares error and the second term is the regularization term. N is the number of samples. In this question, we are looking for a similar form of ridge regression, which is:

$$E(\theta) = \frac{1}{2} \sum_{n=1}^{N} [y(\vec{x_n}, \theta) - y_i]^2 + \frac{\lambda}{2} \sum_{i=1}^{D} \theta_i^2$$

- Since the noise has $\mathsf{Var}(\epsilon_{ij}) = \sigma^2$ and zero mean, we can see:

$$\begin{aligned} \mathsf{Var}(\epsilon_{ij}) &= \mathbb{E}[\epsilon_{ij}^2] - (\mathbb{E}[\epsilon_{ij}])^2 \\ \sigma^2 &= \mathbb{E}[\epsilon_{ij}^2] - 0^2 \\ \mathbb{E}[\epsilon_{ij}\epsilon_{ij}] &= \sigma^2 \end{aligned}$$

- Since the noise terms are i.i.d., we can also say something about the product of different noise terms.

$$\begin{aligned} &\mathbb{E}[\epsilon_{ij}\epsilon_{mn}] = \mathbb{E}[\epsilon_{ij}]\mathbb{E}[\epsilon_{mn}] \quad \text{by the definition of independence, iff } i \neq m \vee j \neq n \\ &\mathbb{E}[\epsilon_{ij}\epsilon_{mn}] = 0 \cdot 0 = 0 \end{aligned}$$

- Altogether, the expectation of the product of two noise terms is piecewise, depending on whether they are the same noise term.

$$\mathbb{E}[\epsilon_\phi \epsilon_\psi] = \begin{cases} \sigma^2 & \phi = \psi \\ 0 & \phi \neq \psi \end{cases}$$

- LaTeX tips! None of these are necessary, but your grader will thank you! If you use \mathbb{E}, it renders the expectation symbol: $\mathbb{E}[\circ]$, which will visually differentiate from the error function $E_D(\circ)$. Also, if you use \vec{...}, it will render the vector symbol: $\vec{\circ}$. Finally, if you use \left(... \right), it will render the height of brackets dynamically w.r.t. whatever you put between them:

$$(\circ) \quad \left[\sum_{i}^{I} \circ \right]$$

1. **Answer:**

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

We are given that $x'_{nd} = x_{nd} + \epsilon_{nd}$. Substituting that:

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

By distributing the θ_d term, we can split the summation into two parts:

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

Grouping the first two terms, which are exactly the definition of the original, noise-free prediction $\hat{y}(\vec{x_n}, \theta)$:

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

Therefore we arrive at our final result:

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

2. **Answer:**

$$\mathbb{E}[E_D(\theta)'] = \mathbb{E}\left[\frac{1}{2} \sum_{n=1}^{N} \left[\hat{y}(\vec{x_n}', \theta) - y_n \right]^2 \right]$$

Using Linearity of Expectation:

$$\mathbb{E}[E_D(\theta)'] = \frac{1}{2} \sum_{n=1}^{N} \mathbb{E}\left[\left(\hat{y}(\vec{x_n}', \theta) - y_n \right)^2 \right]$$

Now, we can substitute the result from Part 1. Let $A = (\hat{y}(\vec{x_n}, \theta) - y_n)$ and $B = \sum_{d=1}^{D} \theta_d \epsilon_{nd}$. The term inside the expectation is $(A+B)^2 = A^2 + 2AB + B^2$.

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

Analysing the expectation of each of the terms in the sum of expectations:

1. $\mathbb{E}\left[(\hat{y}(\vec{x_n}, \theta) - y_n)^2 \right] = (\hat{y}(\vec{x_n}, \theta) - y_n)^2$. (This term is constant with respect to the noise ϵ).
2. $\mathbb{E}\left[2(\hat{y}(\vec{x_n}, \theta) - y_n) \left(\sum_{d=1}^{D} \theta_d \epsilon_{nd} \right) \right]$. Since the first part is constant, this is $2(\hat{y}(\vec{x_n}, \theta) - y_n) \sum_{d=1}^{D} \theta_d \mathbb{E}[\epsilon_{nd}]$. Since $\mathbb{E}[\epsilon_{nd}] = 0$, this term also equals *0*.
3. $\mathbb{E}\left[\left(\sum_{d=1}^{D} \theta_d \epsilon_{nd} \right)^2 \right] = \mathbb{E}\left[\sum_{j=1}^{D} \sum_{k=1}^{D} \theta_j \theta_k \epsilon_{nj} \epsilon_{nk} \right] = \sum_{j=1}^{D} \sum_{k=1}^{D} \theta_j \theta_k \mathbb{E}[\epsilon_{nj} \epsilon_{nk}]$.
   - Since the noise is i.i.d., $\mathbb{E}[\epsilon_{nj} \epsilon_{nk}] = 0$ when $j \neq k$.
   - When $j = k$, $\mathbb{E}[\epsilon_{nj} \epsilon_{nj}] = \mathbb{E}[\epsilon_{nj}^2] = \sigma^2$. (Since $\text{Var}(\epsilon) = \mathbb{E}[\epsilon^2] - (\mathbb{E}[\epsilon])^2 \implies \sigma^2 = \mathbb{E}[\epsilon^2] - 0^2$).
   - Therefore, the sum simplifies to: $\sum_{d=1}^{D} \theta_d^2 \sigma^2 = \sigma^2 \sum_{d=1}^{D} \theta_d^2$.

Substituting these, we get:

$$\mathbb{E}[E_D(\theta)'] = \frac{1}{2} \sum_{n=1}^{N} \left[(\hat{y}(\vec{x_n}, \theta) - y_n)^2 + 0 + \sigma^2 \sum_{d=1}^{D} \theta_d^2 \right]$$

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

The first term is the original, noise-free error $E_D(\theta)$. The second term sums a constant N times:

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

This proves the identity, where the $\text{Regularizer} = \left(\frac{N\sigma^2}{2} \right) \sum_{d=1}^{D} \theta_d^2$. This is an L_2 regularisation on all weights except the bias term θ_0, with a regularization constant $\lambda_{\text{eff}} = N\sigma^2$.

Q5: Naive Bayes and Logistic Regression [40pts] [P] | [W]

In Bayesian classification, we're interested in finding the probability distribution of the label space 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.

5.1 Profile Screening using Naive Bayes [7pts] [W]

[W]: Make sure the markdown cell containing your answer is displayed and properly assigned to Q5.1 when submitting the PDF version of the Juypter notebook as part of the non-programming submission of this assignment. Please format your work with LaTeX.

A tech company has devised an automated resume screening system 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)}.

They have compiled a dataset of 12 anonymized resumes, with each position having 4 applicants.

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

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

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

A new applicant's resume has been screened, and identified to **not have** leadership, **have** data analysis skills **without** product design experience but **does have** coding experience and 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, likelihood, and resulting posterior probabilites) and not just the final answer.

Don't divide your answer by the marginal probability P(x). Since this is not a function of the label, it will be the same for every label, thus this additional division is not necessary when using Naive Bayes classification and should be skipped. Doing so will result in a deduction during grading.

Answer

We need to find the class y that maximizes the posterior probability $P(y|X)$, where X is the new applicant's feature vector.

$$P(y|X) \propto P(y) \times P(X|y) = P(y) \times \prod_{i=1} P(X_i|y)$$

The new applicant's feature vector X is:

- X_1 (Coding)
- X_2 (Data Analysis)
- X_3 (Leadership)
- X_4 (Product Design)
- X_5 (Marketing)

---

1. Prior Probabilities, $P(y)$

There are $N = 12$ total samples.

- $N_{\text{MLE}} = 4$
- $N_{\text{DA}} = 4$
- $N_{\text{PM}} = 4$

The prior probabilities for each class are:

$$P(y = \text{MLE}) = 4/12 = 1/3$$

$$P(y = \text{DA}) = 4/12 = 1/3$$

$$P(y = \text{PM}) = 4/12 = 1/3$$

---

2. Likelihoods, $P(X_i|y)$

For y = Machine Learning Engineer (MLE): (Data: {1,1,0,0,1}, {1,1,0,0,0}, {1,0,1,0,0}, {0,1,0,1,0})

- $P(X_1 = 1|\text{MLE}) = 3/4$
- $P(X_2 = 1|\text{MLE}) = 3/4$
- $P(X_3 = 0|\text{MLE}) = 3/4$
- $P(X_4 = 0|\text{MLE}) = 3/4$
- $P(X_5 = 1|\text{MLE}) = 1/4$

For y = Data Analyst (DA): (Data: {0,1,0,0,0}, {1,1,0,0,1}, {0,1,1,0,1}, {0,1,0,1,0})

- $P(X_1 = 1|\text{DA}) = 1/4$
- $P(X_2 = 1|\text{DA}) = 4/4 = 1$
- $P(X_3 = 0|\text{DA}) = 3/4$
- $P(X_4 = 0|\text{DA}) = 3/4$
- $P(X_5 = 1|\text{DA}) = 2/4 = 1/2$

For y = Product Manager (PM): (Data: {0,0,1,1,0}, {0,0,1,0,1}, {1,0,1,1,1}, {0,1,1,1,1})

- $P(X_1 = 1|\text{PM}) = 1/4$
- $P(X_2 = 1|\text{PM}) = 1/4$
- $P(X_3 = 0|\text{PM}) = 0/4 = 0$
- $P(X_4 = 0|\text{PM}) = 1/4$
- $P(X_5 = 1|\text{PM}) = 3/4$

---

3. Posterior Probabilities

We multiply the prior by the likelihoods for each class.

Machine Learning Engineer:

$$P(\text{MLE}|X) \propto P(\text{MLE}) \times P(X|\text{MLE})$$

$$P(\text{MLE}|X) \propto (1/3) \times (3/4 \times 3/4 \times 3/4 \times 3/4 \times 1/4)$$

$$P(\text{MLE}|X) \propto (1/3) \times (81/1024) = 81/3072$$

Data Analyst:

$$P(\text{DA}|X) \propto P(\text{DA}) \times P(X|\text{DA})$$

$$P(\text{DA}|X) \propto (1/3) \times (1/4 \times 1 \times 3/4 \times 3/4 \times 2/4)$$

$$P(\text{DA}|X) \propto (1/3) \times (18/1024) = 18/3072$$

Product Manager:

$$P(\text{PM}|X) \propto P(\text{PM}) \times P(X|\text{PM})$$

$$P(\text{PM}|X) \propto (1/3) \times (1/4 \times 1/4 \times 0 \times 1/4 \times 3/4)$$

$$P(\text{PM}|X) \propto (1/3) \times 0 = 0$$

---

4. Conclusion

Comparing the results:

$$(81/3072) > (18/3072) > 0$$

Therefore, the *MLE* class has the highest posterior probability

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

This dataset contains the sentiments for financial news headlines from the perspective of a retail investor. The sentiment of news has 3 classes, negative, positive and neutral. In this problem, we only use the negative (class label = 0) and positive (class label = 1) classes for binary logistic regression. For data preprocessing, we remove the duplicate headlines and remove

the neutral class to get 1967 unique news headlines. Then we randomly split the 1967 headlines into training set and evaluation set with 8:2 ratio. We use the training set to fit a binary logistic regression model.

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

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

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

Logistic Regression Overview

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

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

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

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

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

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

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

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

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

where:

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

Note that this will be our model's loss function

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

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

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

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

5.2.1 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"!
UnitTest passed successfully for "Logistic Regression fit"!
```

5.2.2 Logistic Regression Model Training [No Points]

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

from logistic_regression import LogisticRegression as LogReg
from logistic_regression import hyperparameter_tuning
```

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

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

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

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_split(
    BOW, y, indices, test_size=0.2, random_state=RANDOM_SEED
)
```

Fit the model to the training data. If you'd like, try different `lr` and `epochs` to achieve >80% test accuracy.

In [ ]:
```
model = LogReg()
lr = 0.01
epochs = 5000
threshold = 0.5
theta = model.fit(X_train, y_train, X_test, y_test, lr, epochs, threshold)
```

```
Epoch 1000:
    train loss: 0.552  train acc: 0.723
    val loss:   0.604    val acc:   0.675
Epoch 2000:
    train loss: 0.512  train acc: 0.743
    val loss:   0.583    val acc:   0.675
Epoch 3000:
    train loss: 0.482  train acc: 0.756
    val loss:   0.564    val acc:   0.683
Epoch 4000:
    train loss: 0.457  train acc: 0.778
    val loss:   0.547    val acc:   0.688
Epoch 5000:
    train loss: 0.436  train acc: 0.794
    val loss:   0.533    val acc:   0.701
```

5.2.3 Logistic Regression Model Evaluation [No Points]

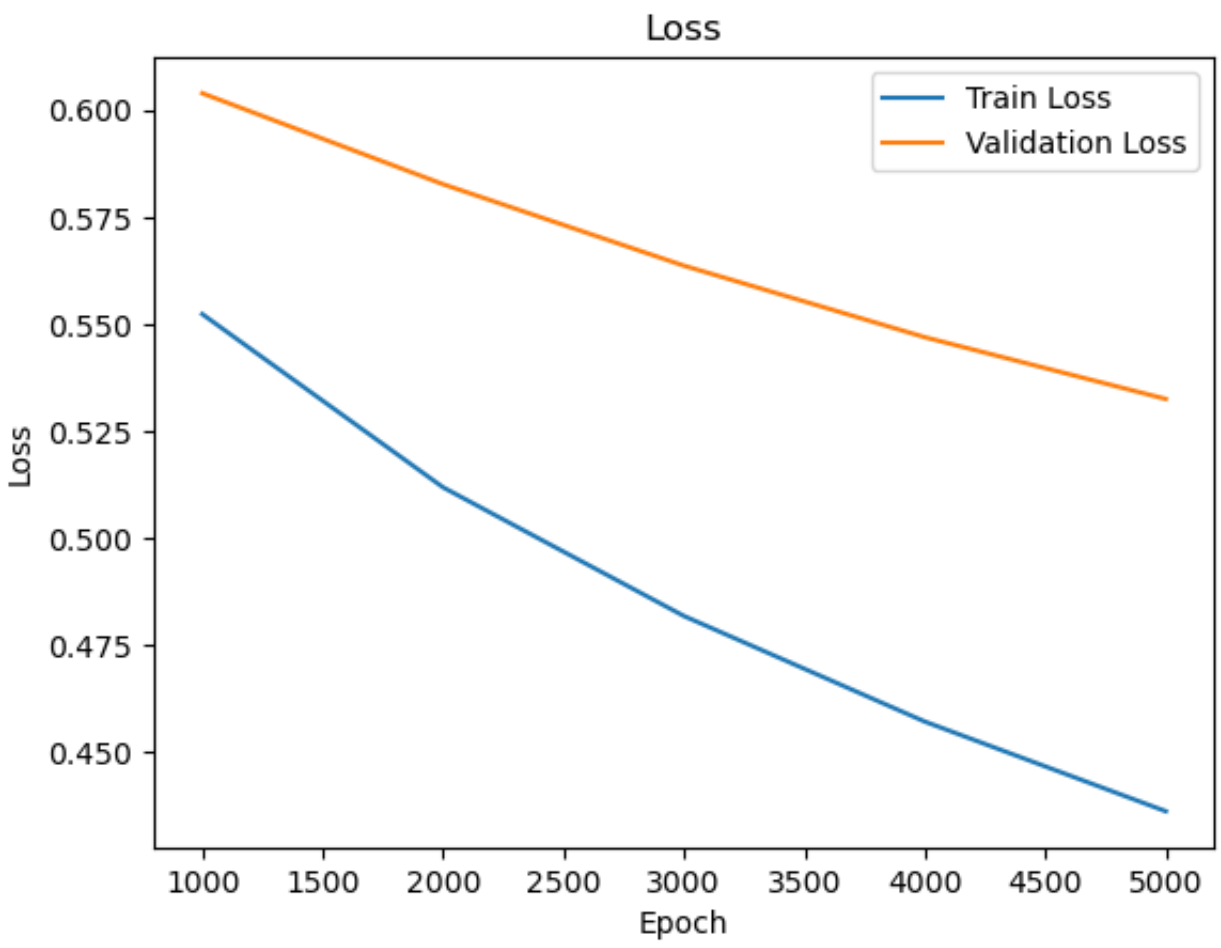
Evaluate the model on the test dataset

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: 0.701
```

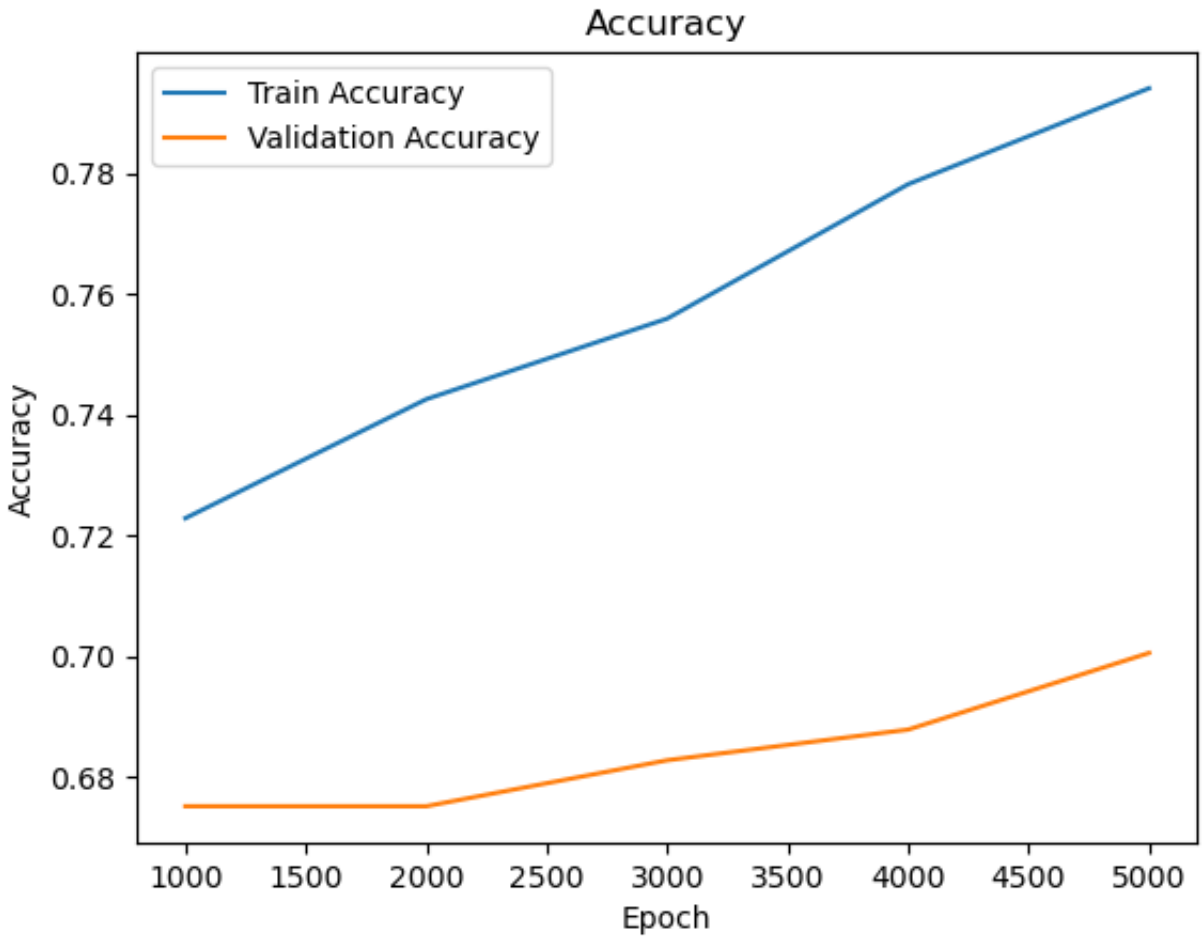
Plotting the loss function on the training data and the test data for every 100th epoch

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]]}\n")
```

```
Input News: Fiskars , a 360-year-old global business best known for its orange-handled scissors , expects t
o derive synergies of EUR5 million a year by consolidating certain parts of the housewares division where t
he two compete .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: One of the installed elevators will be a double-deck elevator , which enables more efficient us
e of the building 's core space .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: In addition the deal includes a call option which will enable Maritim Food AS to increase its h
olding in Sandanger AS to 51 % in the future .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: The payment of 2.779 million litas in interest on a long-term loan provided by Ragutis ' majori
ty shareholder , Estonia 's A. Le Coq , also added to the losses .

Predicted Sentiment: positive
Actual Sentiment: negative

Input News: Operating profit decreased to EUR 16mn from EUR 21.1 mn in 2008 .

Predicted Sentiment: negative
Actual Sentiment: negative

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

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: In addition , Cramo and Peab have signed exclusive five-year rental agreements in Finland and h
ave extended their existing rental agreements in the Swedish market for another five years .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: In December alone , the members of the Lithuanian Brewers ' Association sold a total of 20.3 mi
llion liters of beer , an increase of 1.9 percent from the sales of 19.92 million liters in December 2004 .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: This new representation extends Comptel 's global presence to a total of 18 countries , serving
over 250 customers in over 80 countries worldwide .

Predicted Sentiment: positive
Actual Sentiment: positive

Input News: Kemira Coatings is the leading supplier of paints in Northern and Eastern Europe , providing co
nsumers and professionals with branded products in approx .

Predicted Sentiment: positive
Actual Sentiment: positive
```

5.3 Setting the Threshold [3pts] [P]

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.

Please implement the hyperparameter_tuning method in logistic_regression.py to perform hyperparameter tuning on the thresholds.

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

unittest_lr.test_thresholding()
```

```
UnitTest passed successfully for "Hyperparameter Tuning"!
```

Q6: Feature Selection [20 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.

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 until we 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 each method
forward_selection_feature_list = FeatureReduction.forward_selection(X, y, 0.1)
backward_selection_feature_list = FeatureReduction.backward_elimination(X, y, 0.1)
```

```
Dataset Features:  ['mean radius', 'mean texture', 'mean perimeter', 'mean area', 'mean smoothness', 'mean
compactness', 'mean concavity', 'mean concave points', 'mean symmetry', 'mean fractal dimension', 'radius e
rror', 'texture error', 'perimeter error', 'area error', 'smoothness error', 'compactness error', 'concavit
y error', 'concave points error', 'symmetry error', 'fractal dimension error', 'worst radius', 'worst textu
re', '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 features

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 texture', 'worst area', 'smoothness e
rror', 'worst symmetry', 'compactness error', 'radius error', 'worst fractal dimension', 'mean compactnes
s', 'mean concave points', 'worst concavity', 'concavity error', 'area error']
RMSE: 0.23821726582785294

Backward Feature Elimination
Significant Features: ['mean radius', 'mean compactness', 'mean concave points', 'radius error', 'smoothnes
s error', 'concavity error', 'concave points error', 'worst radius', 'worst texture', 'worst area', 'worst
concavity', 'worst symmetry', 'worst fractal dimension']
RMSE: 0.2374589600443535
```

Q7: 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.

Data Setup

In [ ]:
```
################################
### DO NOT CHANGE THIS CELL ###
################################
SEED = 42
imb = 0.99

# Generate an artifical 10D classification problem with a high degree of class imbalance.
X, y = make_classification(
    n_samples=10000,
    weights=[imb],
    n_features=10,
    n_informative=5,
    n_redundant=5,
    n_repeated=0,
    n_classes=2,
    n_clusters_per_class=2,
    flip_y=0,
    class_sep=0.4,
    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(
    C=0.1, probability=True, random_state=SEED
)  # sklearn's Support Vector Classifier

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

# y_pred has the probability of the positive class
y_pred = classifier.predict_proba(X_test)[:, 1]

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

```
Accuracy: 99.15%
```

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.

7.1 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.

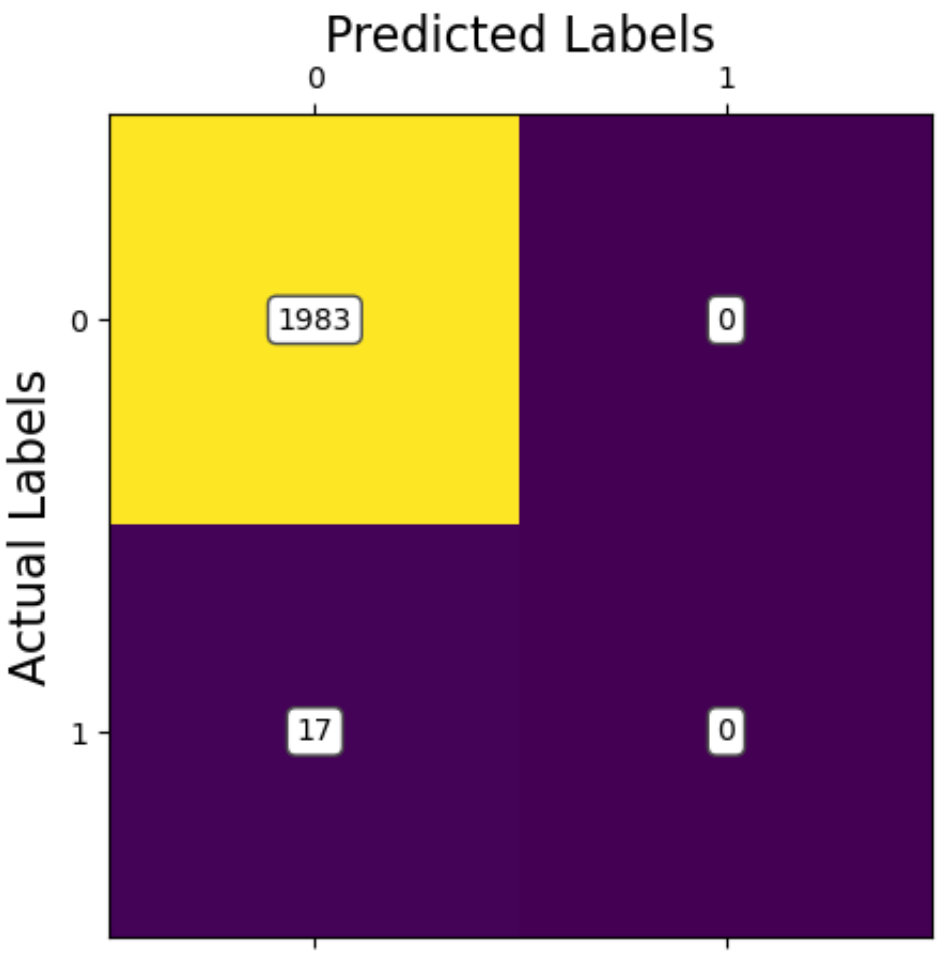
In the **smote.py** file, complete the following:

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 useless. In fact, this model only predicts 0, so it is actually doing nothing. This is a useless classifier.

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 some kind of balanced performance across our labels.

To analyze this, we need a measure of test performance that not only conveys the accuracy of the model, but is robust against bias.

Receiver Operation Characteristic Area Under the Curve (ROC AUC)

The Receiver Operating Characteristic (ROC) curve is a tool which is used to evaluate the performance of a classification model. Classification models often give a probability distribution across the labels. In a binary setting, like with our SVC, this can be represented with a single number, which, WLOG, we choose to refer to the probability of predicting the positive class. Then, we can adjust the classification threshold, which is the probability cutoff for deciding between classes, allowing us to generate several different predictions from the model.

Each such binary prediction from the cutoff can be evaluated with normal metrics. In ROC-AUC, we want to look at False Positive Rate (FPR) and True Positive Rate (TPR).Mathematically, the True Positive Rate (TPR) and False Positive Rate (FPR) are defined as:

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

where:

- **TP** (True Positives) are correctly predicted positive cases,
- **FN** (False Negatives) are actual positive cases wrongly predicted as negative,
- **FP** (False Positives) are actual negative cases wrongly predicted as positive,
- **TN** (True Negatives) are correctly predicted negative cases.

For every prediction we can generate a (FPR, TPR) pair, and plot that on a graph of TPR (y-axis) vs. FPR (x-axis). The curve connecting these pairs forms the Receiver Operating Characteristic. Here is a reference to ROC AUC.

Here is an example walking through that calculation.

We consider 10 samples where y_{true} contains the actual class labels, and y_{pred} contains the predicted probabilities from our classifier.

$$y_{true} = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]$$

$$y_{pred} = [0.1, 0.2, 0.3, 0.5, 0.6, 0.4, 0.5, 0.7, 0.8, 0.9]$$

First, we threshold y_{pred} to make it into a binary vector. Let us consider the calculation at threshold value 0.5, where $\hat{y}_{pred} = y_{pred} \geq 0.5$.

$$\hat{y}_{pred} = [0, 0, 0, 1, 1, 0, 1, 1, 1, 1]$$

Now we can compute the True Positive (TP), False Negative (FN), False Positive (FP) and True Negative (TN):

$$TP = 4$$

$$FN = 1$$

$$FP = 2$$

$$TN = 3$$

Based on the above values, we can compute the FPR and TPR:

$$FPR = \frac{4}{4 + 1} = \frac{4}{5} = 0.8$$

$$TPR = \frac{2}{2 + 3} = \frac{2}{5} = 0.4$$

This gives us the point $(0.8, 0.4)$. But now, we must do this for several thresholds.

We must consider the thresholds at every unique value in y_{pred}, as well as 0 and 1. Thus, we consider the thresholds:

$$threshold = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.0, 1.01]$$

1.01 is consider, since it is greated that the maximum possible value of y_{pred}, which is 1.0, and thus we will get FPR and TPR of 0.0 (starting point of the ROC curve). Similarly, 0.0 is considered, which will get the maximum possible FPR and TPR of 1.0 (ending point of the ROC curve).

After doing this for all the thresholds, we get FPR and TPR values as follows:

$$FPR = [1.0, 0.8, 0.6, 0.4, 0.4, 0.2, 0.0, 0.0, 0.0, 1.0, 0.0]$$

$$TPR = [1.0, 1.0, 1.0, 1.0, 0.8, 0.6, 0.6, 0.4, 0.2, 1.0, 0.0]$$

We need to sort the values such that FPR is in ascending order. However, when multiple points have the same FPR, we must sort them by TPR in ascending order to ensure that the ROC curve correctly moves upwards and does not jag back down. Sorting properly is important for accurately calculating AUC. The indices of the sorted values must also be correctly aligned so that each TPR corresponds to its respective FPR.

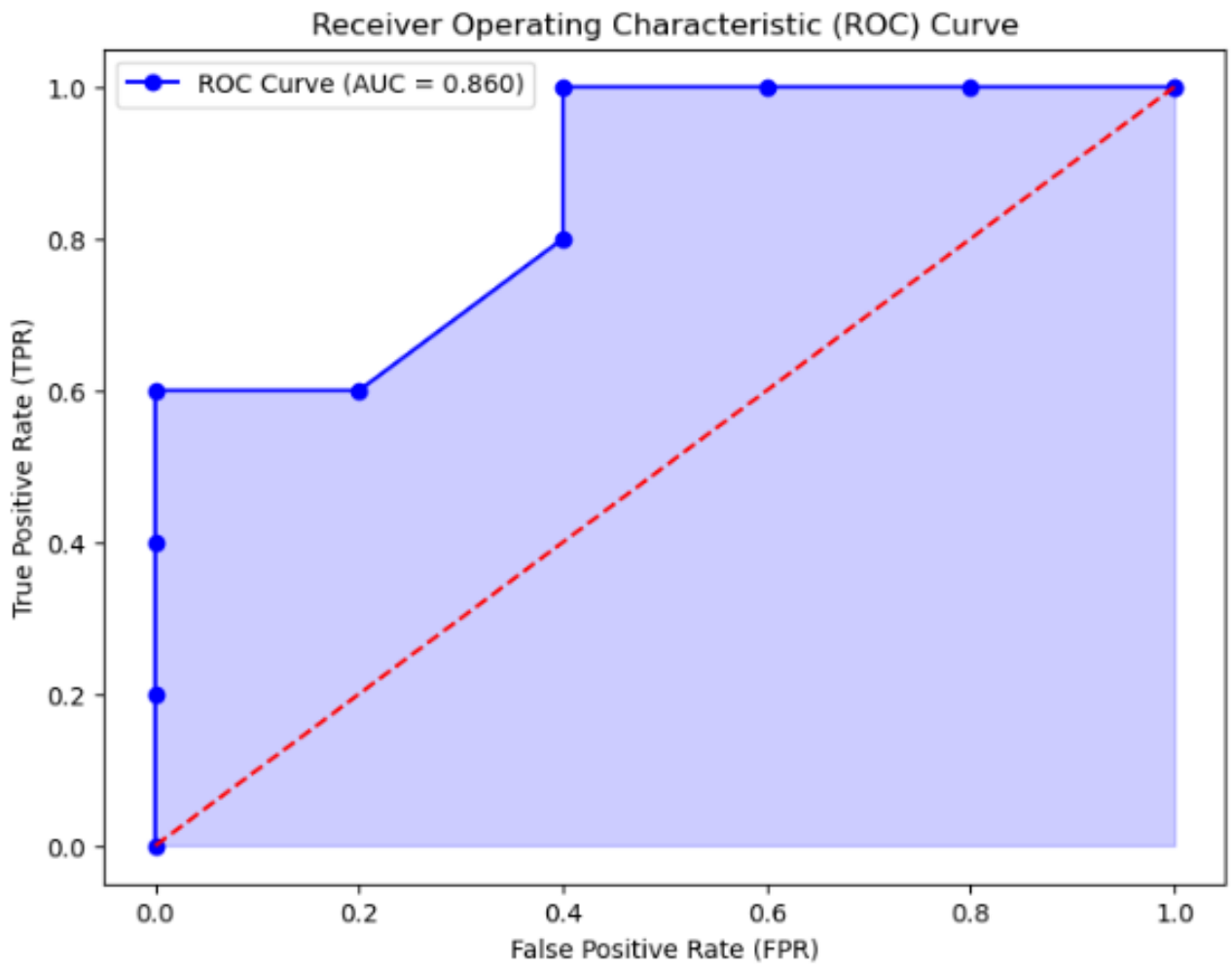
After sorting, we get:

$$FPR = [0.0, 0.0, 0.0, 0.0, 0.2, 0.4, 0.4, 0.6, 0.8, 1.0, 1.0]$$

$$TPR = [0.0, 0.2, 0.4, 0.6, 0.6, 0.8, 1.0, 1.0, 1.0, 1.0, 1.0]$$

This alignment preserves the correct order of points for plotting the ROC curve and ensures the correct integration for computing AUC.

After that, we get the ROC plot as follows:



Based on the above ROC curve, the area under that curve (AUC) will tell us how well our model has differentiated between the two classes.

We calculate the AUC as the integral of TPR with respect to FPR:

$$AUC = \int_0^1 TPR(FPR)\, d(FPR)$$

Since the ROC is a plot of TPR versus FPR, the area under the curve represents how much the model's TPR improves as we allow more FPR.

- If AUC = 1, the model is classifying perfectly, and there exists a threshold at which the model perfectly discriminates the classes
- If AUC = 0.5, the model is doing no better than randomly guessing

Higher value of AUC represents better classification performance.

In this example, the value of AUC is 1 which means that the classifier model is perfect in distinguishing between positive and negative classes.

Local Tests for ROC AUC [No points]

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

1. **threshold_eval**: Input y_true and y_pred calculated using predict_proba() which contains the probability of class 1 (true class). You will need to calculate the True Positive Rate (TPR) and False positive Rate (FPR) by for the particular threshold in the function. The output should contain the value of FPR, TPR as a tuple. Please be careful with the order of the output. Ensure that the FPR is first, and then have TPR.

2. **generate_roc**: Input y_true and y_pred calculated using predict_proba() which contains the probability of class 1 (true class). For the thresholds, you will need to consider every unique value in y_pred, and also consider 0 and 1 as thresholds. Also use the threshold_eval function to calculate the FPR and TPR for each threshold. The output should contain the list of tuple which has FPR, TPR values for each threshold. Sort the list based on the FPR values in ascending order. In case of multiple points having the same FPR, sort them by TPR in ascending order.

3. **integrate_auc**: Input the list of tuples containing FPR and TPR values (the output of generate_roc function). You will need to calculate the area of the curve produced by the ROC points by integrating TPR with respect to FPR. The output should be the value of AUC.

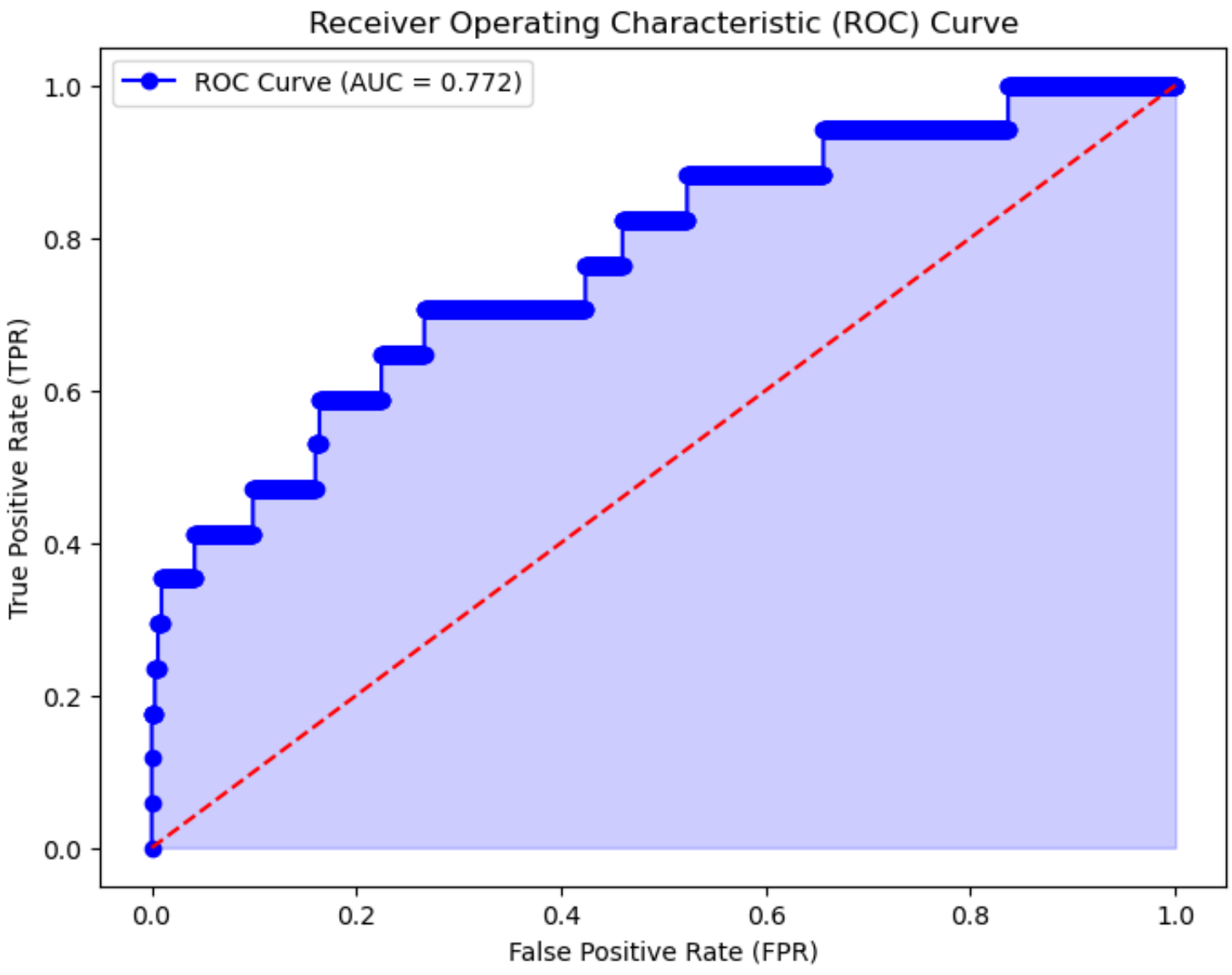
In [ ]:
```
unittest_sm.test_threshold_eval()
unittest_sm.test_generate_roc()
unittest_sm.test_integrate_curve()
```

```
UnitTest passed successfully for "Threshold evaluation"!
UnitTest passed successfully for "Generate ROC Points"!
UnitTest passed successfully for "Integrating ROC Curve to get AUC"!
```

Visualize the ROC of the Data

Let us take a look at the ROC curve of our model.

In [ ]:
```
fpr_tpr_actual = sm.generate_roc(y_test, y_pred)
auc_actual = sm.integrate_curve(fpr_tpr_actual)

sm.plot_roc_auc(fpr_tpr_actual, auc_actual)
print(f"Area under the curve: {auc_actual}")
```




```
Area under the curve: 0.7721811871495952
```

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:

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

$$\downarrow$$

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

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.

7.2 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:

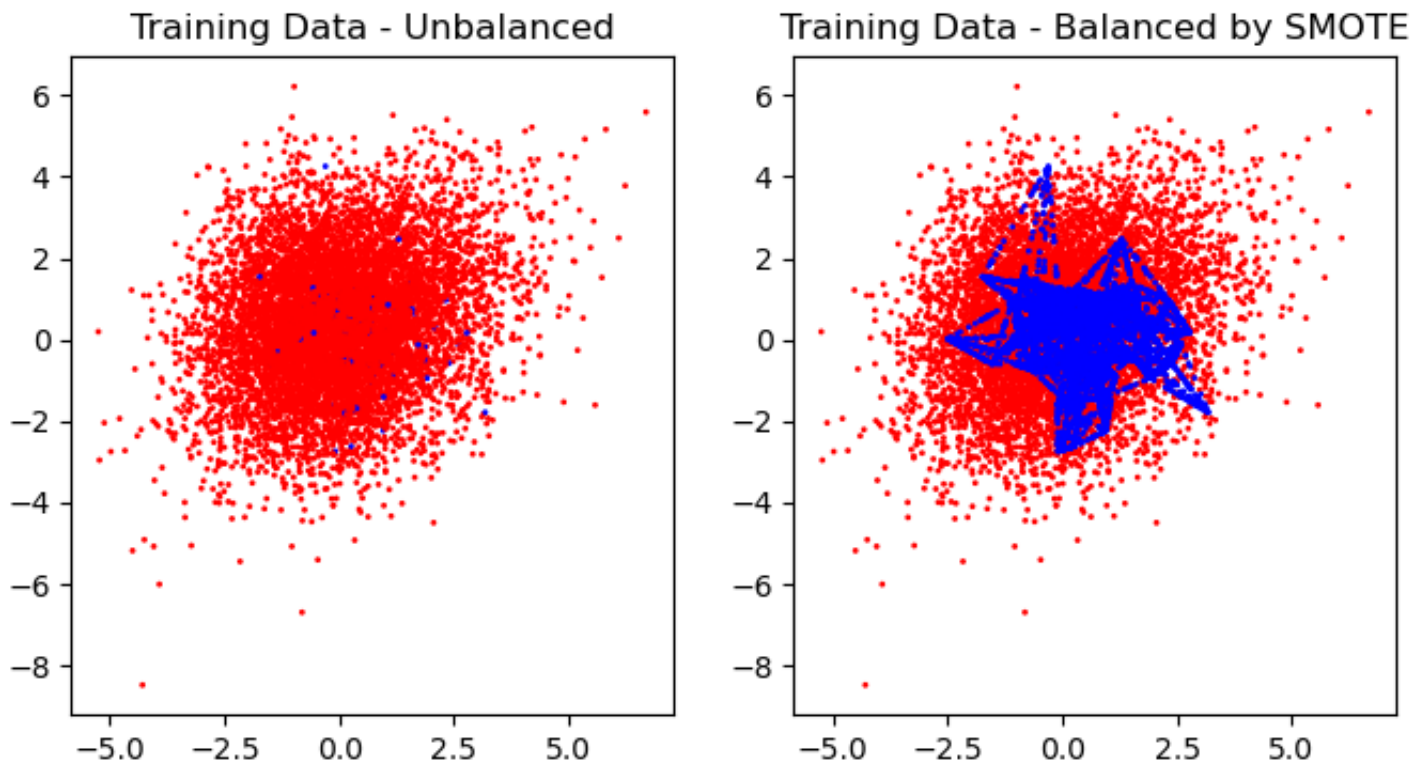
1. **interpolate**: Given a start point, an end point, and an interpolation coefficient, return a linearly interpolated point.
2. **k_nearest_neighbors**: Given some set of points (N, D) and a parameter k, generate an (N, k) array of indices such that `output[i]` contains the k indices corresponding 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 (|maj|-|min|, D).

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"!
```

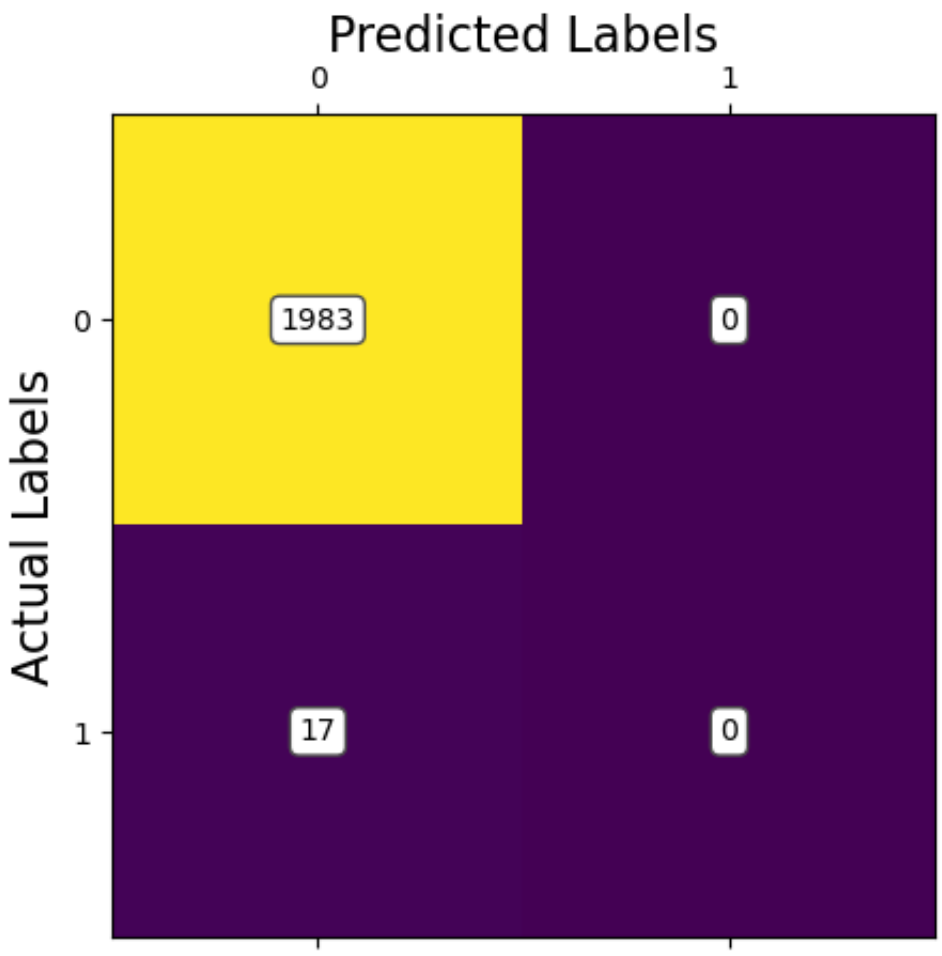
Let's apply SMOTE to our dataset and see the downstream effect on our classification task!

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

# Run SMOTE!
X_train_synth, y_train_synth = sm.smote(X_train, y_train, k=5, inter_coeff_range=(0, 1))
# 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()
```

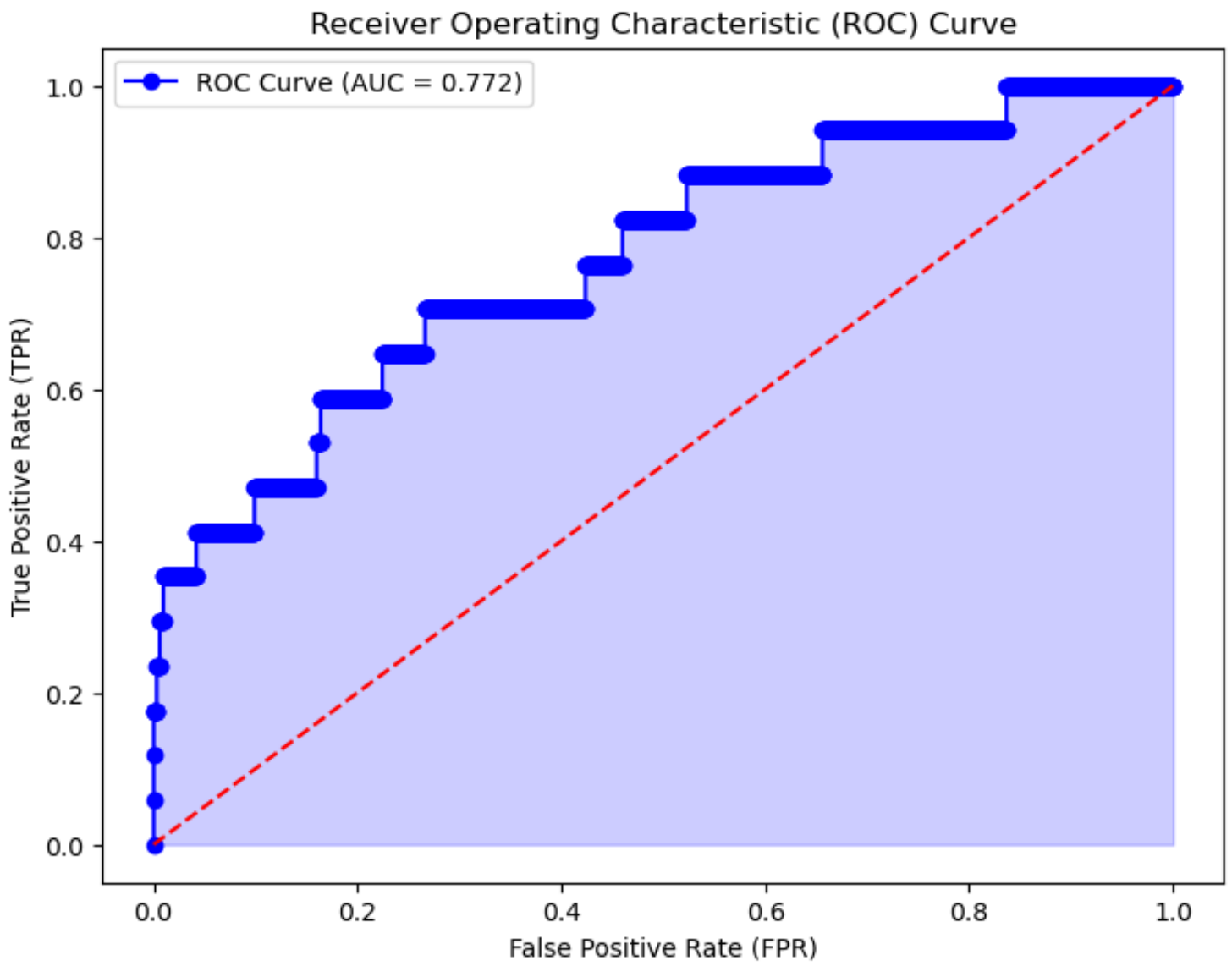


In [ ]:
```
##################################
### DO NOT CHANGE THIS CELL ###
##################################

# Original
classifier.fit(X_train, y_train)
y_predicted1 = classifier.predict(X_test)
y_pred1 = classifier.predict_proba(X_test)[:, 1]

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)}%")
fpr_tpr_1 = sm.generate_roc(y_test, y_pred1)
auc1 = sm.integrate_curve(fpr_tpr_1)
print(f"AUC: {auc1}")
sm.plot_roc_auc(fpr_tpr_1, auc1)


# Balanced
classifier.fit(X_train_balanced, y_train_balanced)
y_predicted2 = classifier.predict(X_test)
y_pred2 = classifier.predict_proba(X_test)[:, 1]

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)}%")

fpr_tpr_2 = sm.generate_roc(y_test, y_pred2)
auc2 = sm.integrate_curve(fpr_tpr_2)
print(f"SMOTEd AUC: {auc2}")
sm.plot_roc_auc(fpr_tpr_2, auc2)


# If you're getting errors running this cell, check your implementation, then regenerate the SMOTE data by
```

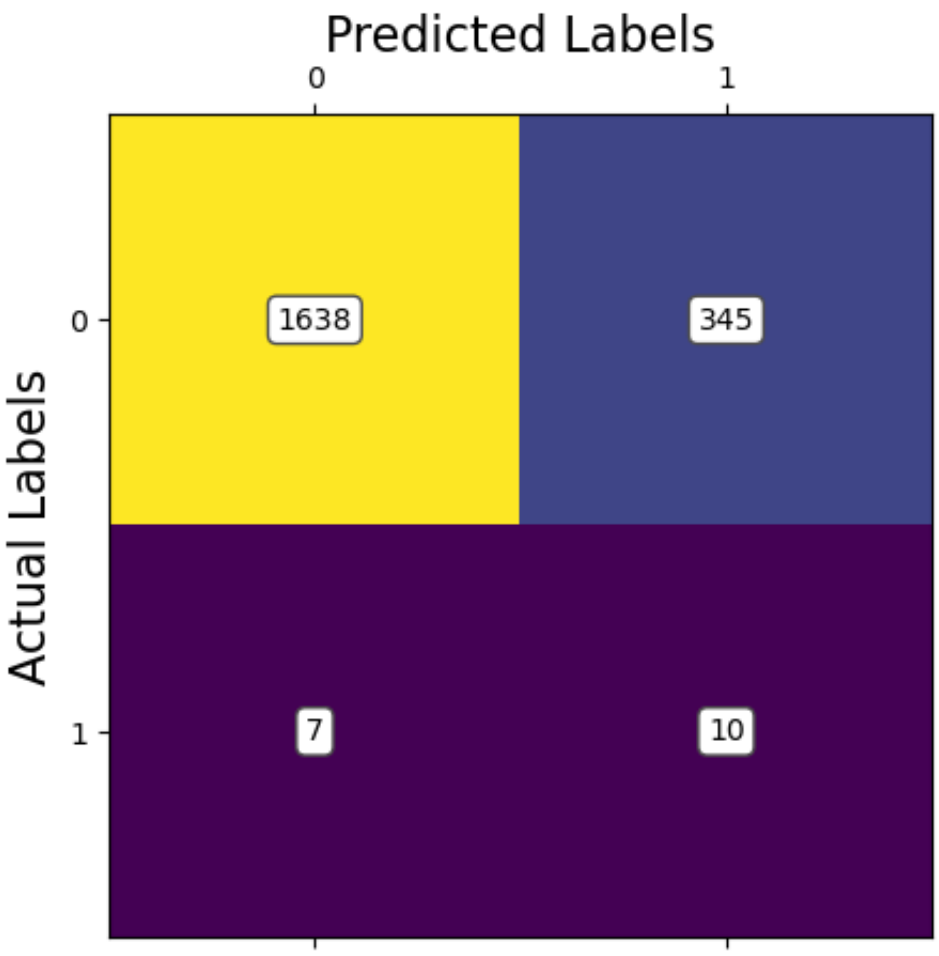
```
Original performance:
```



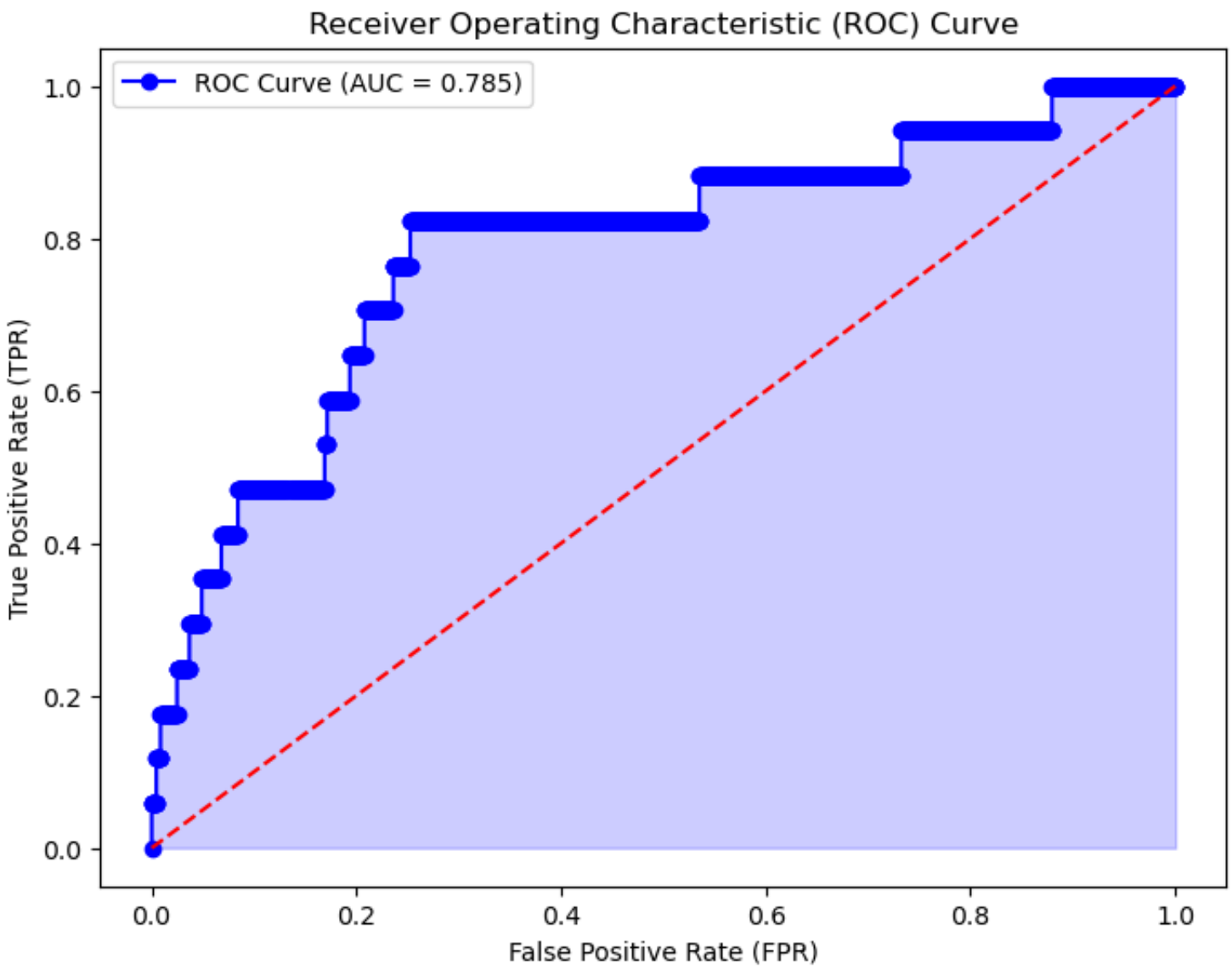

```
Accuracy: 99.15%
AUC: 0.7721811871495952
```




```
Performance after SMOTE:
```




```
SMOTEd Accuracy: 82.39999999999999%
SMOTEd AUC: 0.7851146510041234
```



With our new synthetic data (if done correctly), the model is now significantly more performant on the test points for the minority class! We can observe that the accuracy has reduced however, if the model is doing a better job at distinguishing the classes after applying SMOTE. 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 improve the usefulness of your model, 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! Remember that accuracy isn't everything, and analysis of your system's bias is also important.