STOR 235 — Lab 7

Instructions

There are three parts to this lab:

1. Introduction to logistic regression;
2. Logistic regression for handwriting recognition;
3. Gradient descent.

Each part will be composed of a number of tasks for you to complete. The tasks are labeled in bold as **Problem 1**, **Problem 2**, and so on. Please make sure that you do not forget to complete any of the problems. Each problem will be worth 4 points.

There is a single code block under this introductory material that imports various things you will use later. Please run it.

Important Information About Submitting Your Assignment

Please upload your assignment as a PDF file, and make sure to select to all relevant pages for each part. Make sure that your PDF file contains all of your work, and that nothing has been cut off at the end. **You will not receive credit for solutions that are not in the PDF.**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.datasets import fetch_openml
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
```

Important Information About This Lab

I am releasing this lab before discussing the relevant concepts in class, due to scheduling constraints. (It will be due after they are discussed.) In this notebook, I am including some minimal exposition, which should suffice to complete the coding problems. Fuller explanations will be given in class. So, if certain things do not yet make sense, please don't worry. Please ask your TA in recitation about any immediate questions you might have.

Part 1: Logistic Regression

We previously discussed linear regression models. For one independent variable, these take the form `y = beta_0 + beta_1*x`.

We now consider a variation on linear regression tailored to classification problems. For simplicity, we will consider binary classification problems, with the two classes coded as 0 and 1. Our model will return a number `p` between 0 and 1, which you can think of as the "confidence" or "certainty" the model has that the given data point is a 1. You may have seen this described in other statistics classes as the "probability" that the given data point is 1. Our classifier will classify something as a 1 if `p>.5`, and as 0 otherwise.

When we previously did classification with support vector machines and k-nearest neighbors, we had no ability to tell which points the model was confident or not confident about, so this is a new twist.

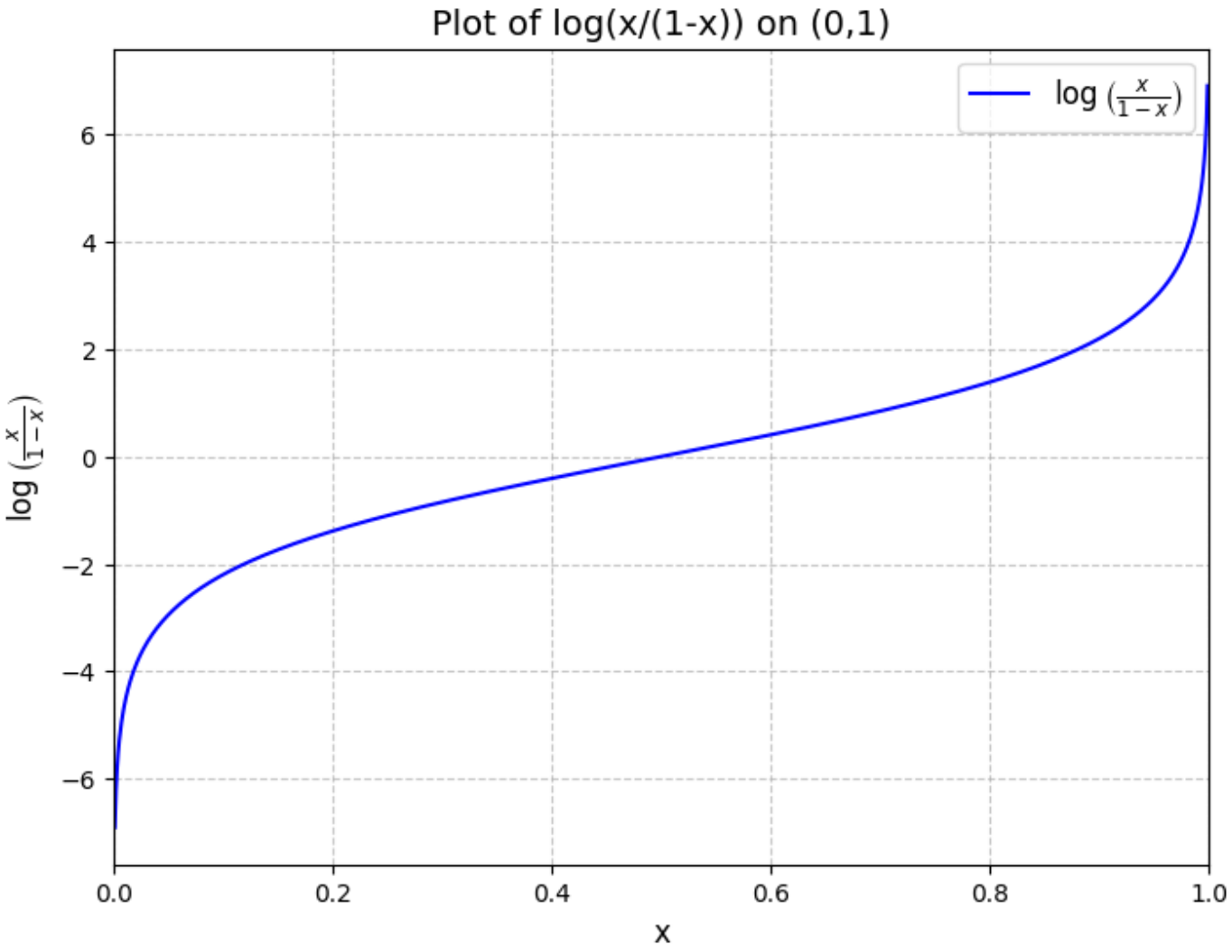
You might consider trying to estimate `p` using linear regression. However, this will lead to values of `p` outside the interval [0,1], which is not what we want. Instead, we will use linear regression but take the y-value to be the so-called "log odds," or `log(p/(1-p))`. This function maps the interval [0,1] to the entire real line, so it is a more suitable target for linear regression. The, once we've fit the model, we can solve for the estimated value of `p` given any data point (by reversing the log-odds transformatinon). This model results in a characteristic "logistic curve" for the predicted probabilies, which we will see in a plot below.

Much more will be said in class about how this model is set up, and how it is fit to data. For now, I will just say that the coefficients are chosen by minimizing a certain error measure on the training data using gradient descent.

Let's first plot the log-odds function, so you can see that the claim about mapping [0,1] to the real line is true.

```
def log_odds(x):
    return np.log(x / (1 - x))

# Generate x values in the interval (0,1)
x = np.linspace(0.001, 0.999, 1000)  # Avoid 0 and 1 to prevent division by zero
```

```
# Compute y values
y = log_odds(x)

# Create the plot
plt.figure(figsize=(8, 6))
plt.plot(x, y, label=r'$\log\left(\frac{x}{1-x}\right)$', color='blue')

# Add labels and title
plt.xlabel('x', fontsize=12)
plt.ylabel(r'$\log\left(\frac{x}{1-x}\right)$', fontsize=12)
plt.title('Plot of log(x/(1-x)) on (0,1)', fontsize=14)

# Add grid and legend
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12)

# Set x-axis limits
plt.xlim(0, 1)

# Show the plot
plt.show()
```



Next, we revisit the wine dataset from a previous lab. (You may wish to review that lab to remember the structure of the data.) We begin by loading the data, using only the first feature (alcohol content) to start with. We will run a logistic regression with this as the independent variable.

```
data = load_wine()
X = data.data[:, 0].reshape(-1, 1)  # Use only the first feature (alcohol content
y = data.target                     # Original classes (0, 1, 2)
```

The wine dataset contains three class (the three regions where the wines were grown). Since we are demonstrating binary classification, we will drop the third class. You may wish to print each of the following variables (or the previous ones) to see how exactly this code works.

Also, it is fairly straightforward to adapt logistic regression to multi-class classification (essentially by following the method we used to transform support vector machines from a method for binary classifcation to multi-class classification, as described on the SVM slides). But for brevity, we'll stick to the two-class case in this lab.

```
mask = (y == 0) | (y == 1)
X = X[mask]
y = y[mask]


#split into test and train sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

We now use the logistic regression function we imported from sklearn above, and apply it to the training data.

```
# Train logistic regression
model = LogisticRegression()
model.fit(X_train, y_train)
```

```
▾ LogisticRegression
LogisticRegression()
```

We now print the accuracy, along with the coefficients and intercept. The accuracy is reasonable for using only one feature.

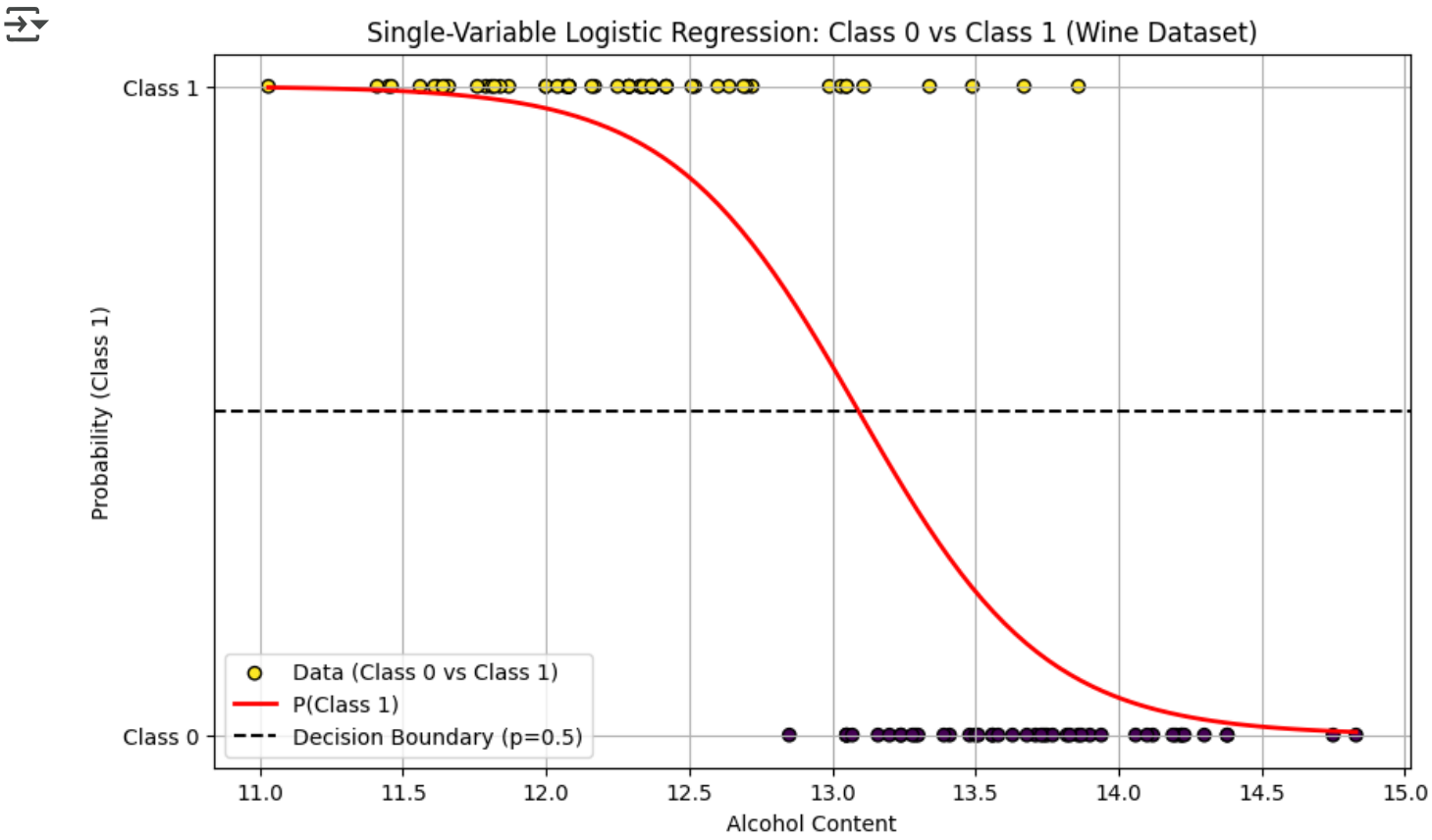
```
# Print accuracy and coefficients
print(f"Test Accuracy: {model.score(X_test, y_test):.2f}")
print(f"Coefficient (slope): {model.coef_[0][0]:.2f}")
print(f"Intercept: {model.intercept_[0]:.2f}")
```

```
Test Accuracy: 0.92
Coefficient (slope): -3.07
Intercept: 40.22
```

The rest of this code is just for creating a nice plot of the decision boundary, and you can ignore the details.

```
X_range = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
probabilities = model.predict_proba(X_range)[:, 1]  # P(class=1)
```

```
# Plot the data and decision boundary
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, c=y_train, cmap='viridis', edgecolors='k', label='D
plt.plot(X_range, probabilities, color='red', linewidth=2, label='P(Class 1)')
plt.axhline(0.5, color='black', linestyle='--', label='Decision Boundary (p=0.5)'

plt.xlabel('Alcohol Content')
plt.ylabel('Probability (Class 1)')
plt.title('Single-Variable Logistic Regression: Class 0 vs Class 1 (Wine Dataset)
plt.yticks([0, 1], ['Class 0', 'Class 1'])  # Label y-axis with class names
plt.legend()
plt.grid(True)
plt.show()
```



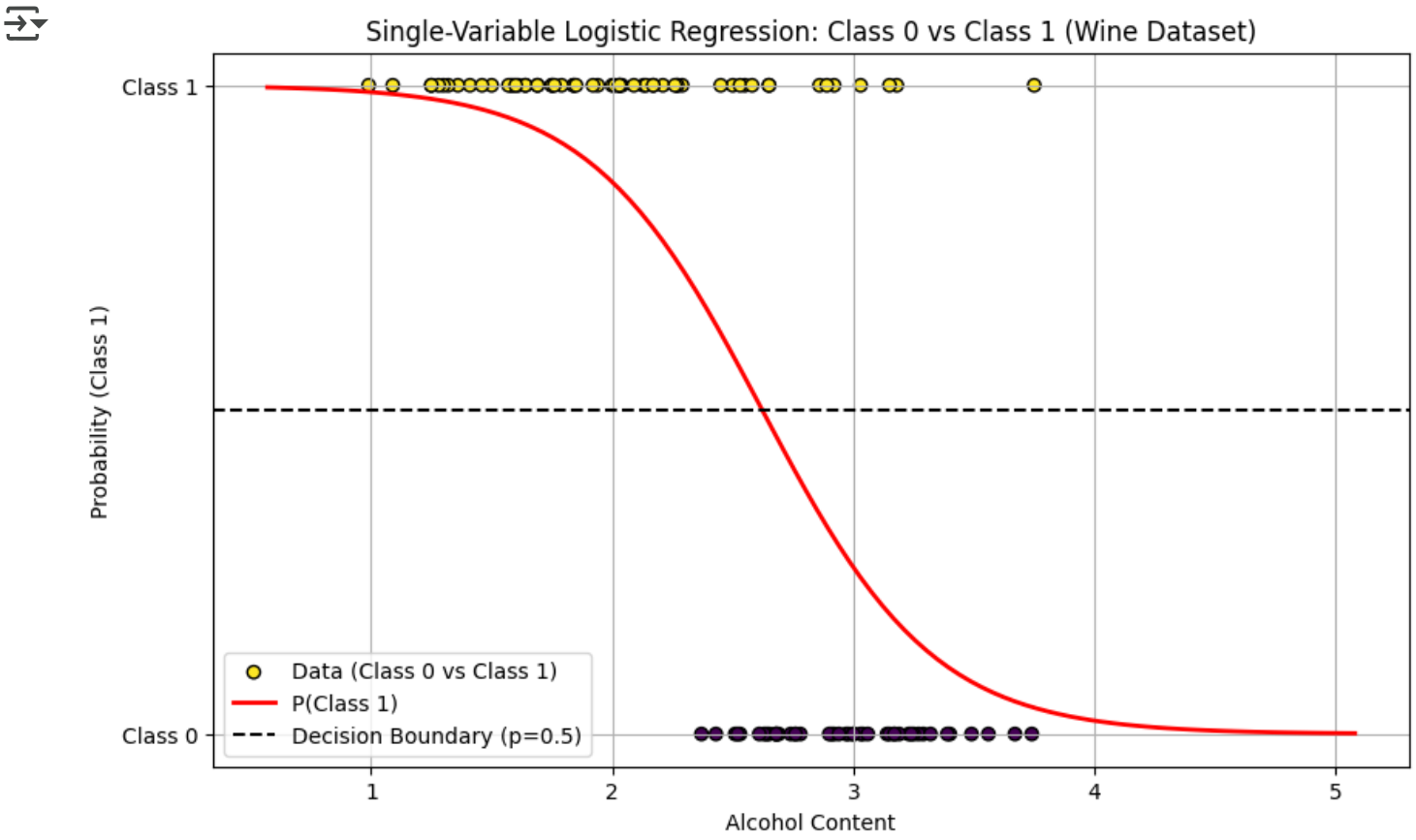
Problem 1. Pick a different feature (independent variable) from the wine dataset. (You may wish to look back at the previous lab that used this dataset to see the list of features.) Using this dataset, and dropping the third region (as above), run a single-variable logistic regression using your chosen independent variable to build a predictor that predicts which of the first two regions the wine came from. Print the accuracy on a 20/80 test-train split, and provide a plot of the predicted probabilities as a function of the independent variable.

You don't need to do anything fancy here; just copy the previous code as modify it as necessary.

```
data = load_wine()
X = data.data[:, 6].reshape(-1, 1)  # Use only the first feature (alcohol content
y = data.target

mask = (y == 0) | (y == 1)
X = X[mask]
y = y[mask]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = LogisticRegression()
model.fit(X_train, y_train)

print(f"Test Accuracy: {model.score(X_test, y_test):.2f}")
print(f"Coefficient (slope): {model.coef_[0][0]:.2f}")
print(f"Intercept: {model.intercept_[0]:.2f}")

X_range = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
probabilities = model.predict_proba(X_range)[:, 1]  # P(class=1)
```

```
Test Accuracy: 0.65
Coefficient (slope): -2.80
Intercept: 7.35
```

```
plt.figure(figsize=(10, 6))
plt.scatter(X_train, y_train, c=y_train, cmap='viridis', edgecolors='k', label='D
plt.plot(X_range, probabilities, color='red', linewidth=2, label='P(Class 1)')
plt.axhline(0.5, color='black', linestyle='--', label='Decision Boundary (p=0.5)'

plt.xlabel('Alcohol Content')
plt.ylabel('Probability (Class 1)')
plt.title('Single-Variable Logistic Regression: Class 0 vs Class 1 (Wine Dataset)
plt.yticks([0, 1], ['Class 0', 'Class 1'])  # Label y-axis with class names
plt.legend()
plt.grid(True)
plt.show()
```



Now we repeat the entire process with two features. Again, you can ignore the plotting code, which is somewhat complicated and unrelated to the mathematics.

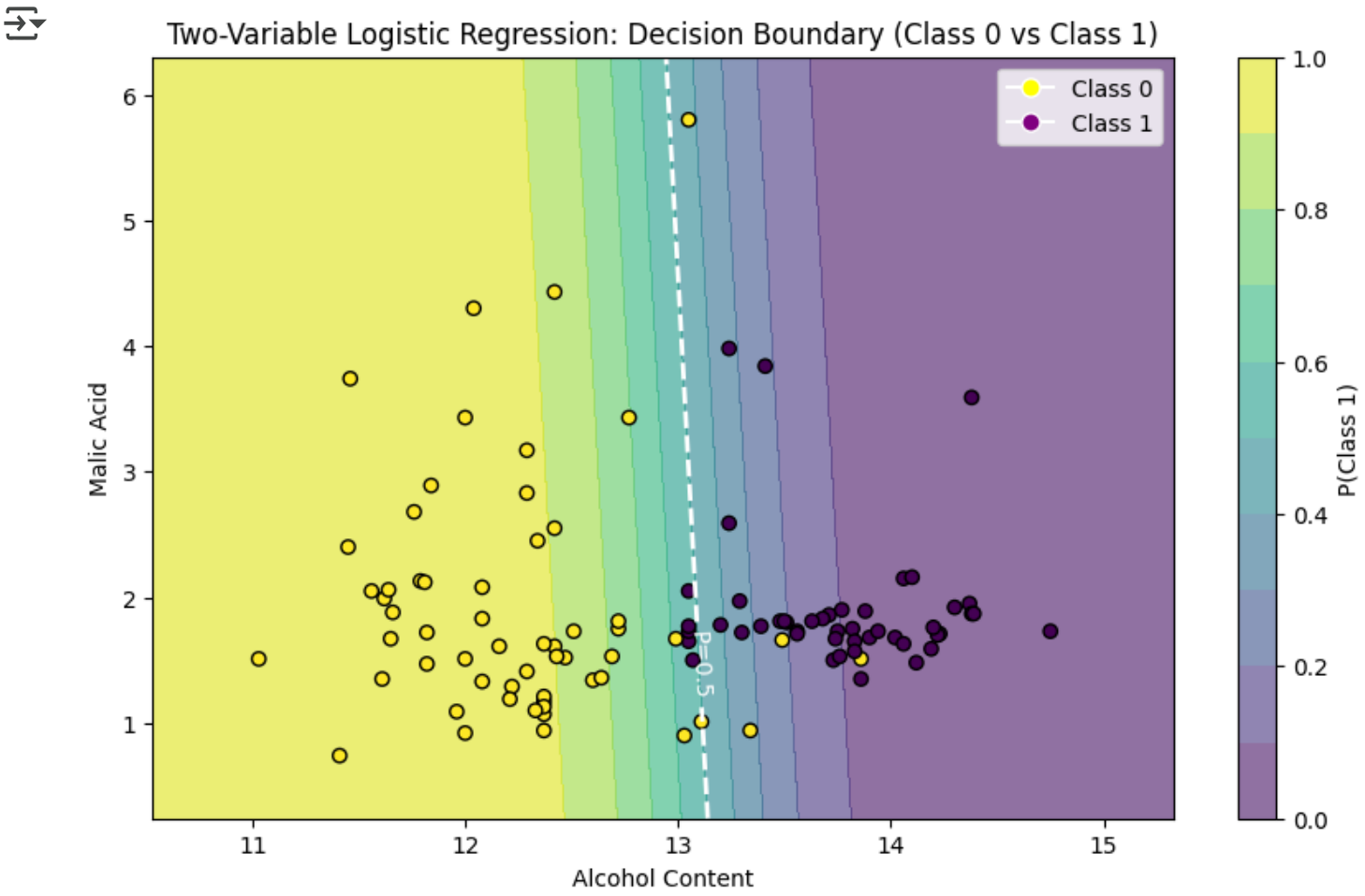
Of course, one could use even more features for increased accuracy (but then it would become hard to visualize).

```
# Select two features (e.g., alcohol + malic acid)
X2 = data.data[:, :2][mask]  # Features 0 (alcohol) and 1 (malic acid)
y2 = y  # Same binary target (Class 0 vs Class 1)

# Split into train/test (new variable names)
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2, test_size=0.2)

# Train logistic regression
model2 = LogisticRegression()
model2.fit(X2_train, y2_train)

print(f"Test Accuracy (2 variables): {model2.score(X2_test, y2_test):.2f}")
print(f"Coefficients: {model2.coef_[0]}")
print(f"Intercept: {model2.intercept_[0]}")
```

```
Test Accuracy (2 variables): 0.85
Coefficients: [-3.26155674 -0.10535611]
Intercept: 42.88133664936839
```

```
# Create a grid to plot decision boundary
x_min, x_max = X2[:, 0].min() - 0.5, X2[:, 0].max() + 0.5
y_min, y_max = X2[:, 1].min() - 0.5, X2[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 10

# Predict probabilities for the grid
Z = model2.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape)

# Plot decision boundary and data
plt.figure(figsize=(10, 6))

# Make the colormap
contour = plt.contourf(xx, yy, Z, levels=np.linspace(0, 1, 11), cmap='viridis', a
plt.colorbar(contour, label='P(Class 1)')
```

```
# Highlight the decision boundary (P=0.5)
decision_boundary = plt.contour(xx, yy, Z, levels=[0.5], colors='white', linewidt
plt.clabel(decision_boundary, inline=True, fmt='P=0.5', fontsize=10)

# Scatter plot of data points
scatter = plt.scatter(X2_train[:, 0], X2_train[:, 1], c=y2_train, cmap='viridis',
plt.xlabel('Alcohol Content')
plt.ylabel('Malic Acid')
plt.title('Two-Variable Logistic Regression: Decision Boundary (Class 0 vs Class

# Manually add legend for class labels
legend_labels = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='yel
                 plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='pur
plt.legend(handles=legend_labels)

plt.show()
```



Problem 2. Pick a different set of two features from the wine dataset. Using this dataset, and dropping the third region (as above), run a single-variable logistic regression using your chosen independent variable to build a predictor that predicts which of the first two regions the wine came from. Print the accuracy on a 20/80 test-train split, and provide a plot of the decision boundary.

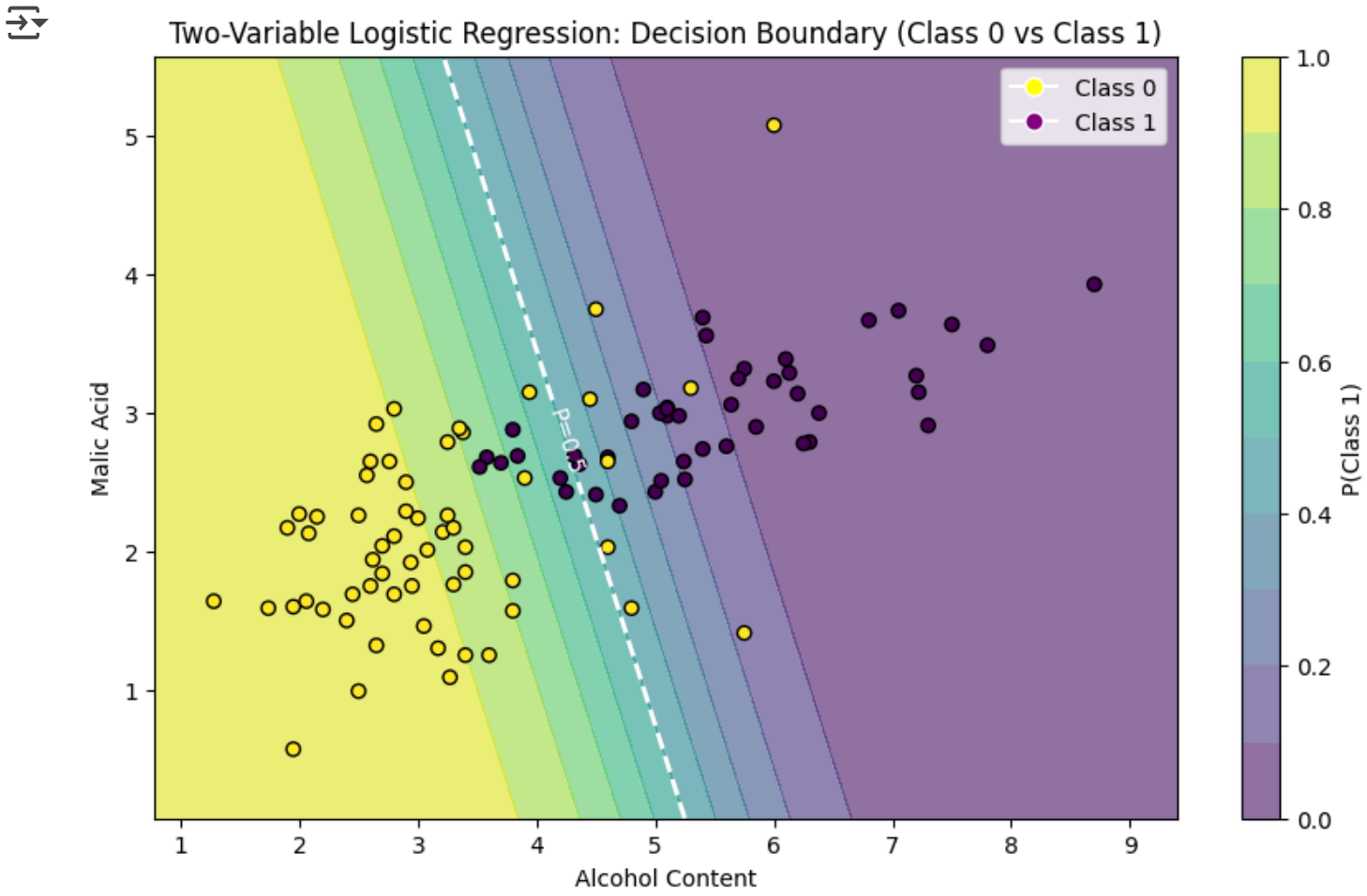
You don't need to do anything fancy here; just copy the previous code as modify it as necessary.

```
X2 = data.data[:, [9, 6]][mask]
y2 = y  # Same binary target (Class 0 vs Class 1)

# Split into train/test (new variable names)
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2, test_size=0.2)

# Train logistic regression
model2 = LogisticRegression()
model2.fit(X2_train, y2_train)

print(f"Test Accuracy (2 variables): {model2.score(X2_test, y2_test):.2f}")
print(f"Coefficients: {model2.coef_[0]}")
print(f"Intercept: {model2.intercept_[0]}")
```

```
Test Accuracy (2 variables): 0.85
Coefficients: [-1.56504443 -0.5776737 ]
Intercept: 8.254937435915116
```

```
# Create a grid to plot decision boundary
x_min, x_max = X2[:, 0].min() - 0.5, X2[:, 0].max() + 0.5
y_min, y_max = X2[:, 1].min() - 0.5, X2[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 10

# Predict probabilities for the grid
Z = model2.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape)

# Plot decision boundary and data
plt.figure(figsize=(10, 6))

# Make the colormap
contour = plt.contourf(xx, yy, Z, levels=np.linspace(0, 1, 11), cmap='viridis', a
plt.colorbar(contour, label='P(Class 1)')

# Highlight the decision boundary (P=0.5)
decision_boundary = plt.contour(xx, yy, Z, levels=[0.5], colors='white', linewidt
plt.clabel(decision_boundary, inline=True, fmt='P=0.5', fontsize=10)

# Scatter plot of data points
scatter = plt.scatter(X2_train[:, 0], X2_train[:, 1], c=y2_train, cmap='viridis',
plt.xlabel('Alcohol Content')
plt.ylabel('Malic Acid')
plt.title('Two-Variable Logistic Regression: Decision Boundary (Class 0 vs Class
```

```
# Manually add legend for class labels
legend_labels = [plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='yel
                 plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='pur
plt.legend(handles=legend_labels)

plt.show()
```



Part 2: Digit Recognition

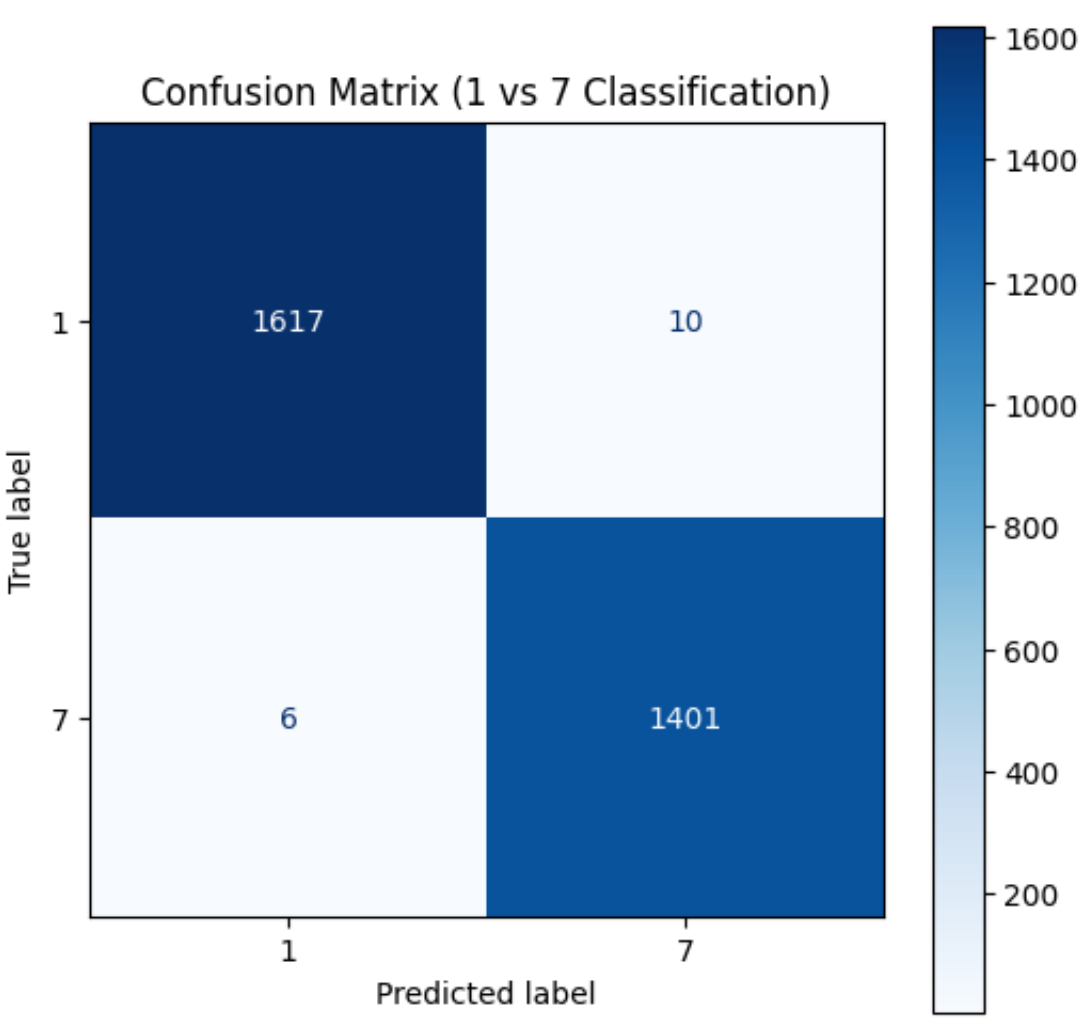
Now we consider an example of logistic regression with many features. We will consider the binary classification problem of telling handwritten `1`'s from `7`'s, using the MNIST dataset from a previous lab. (You may wish to revisit the classification lab to reminder yourself how this dataset is structured.) The features are the numerical values of each pixel (representing how dark it is). We'll write out the exact equations in class.

Note that in the plot of misclassified examples below, we've labeled `1` as `0` and `7` as `1`, so be careful interpreting it. Note also that most of these are a bit ambiguous, due to bad handwriting.

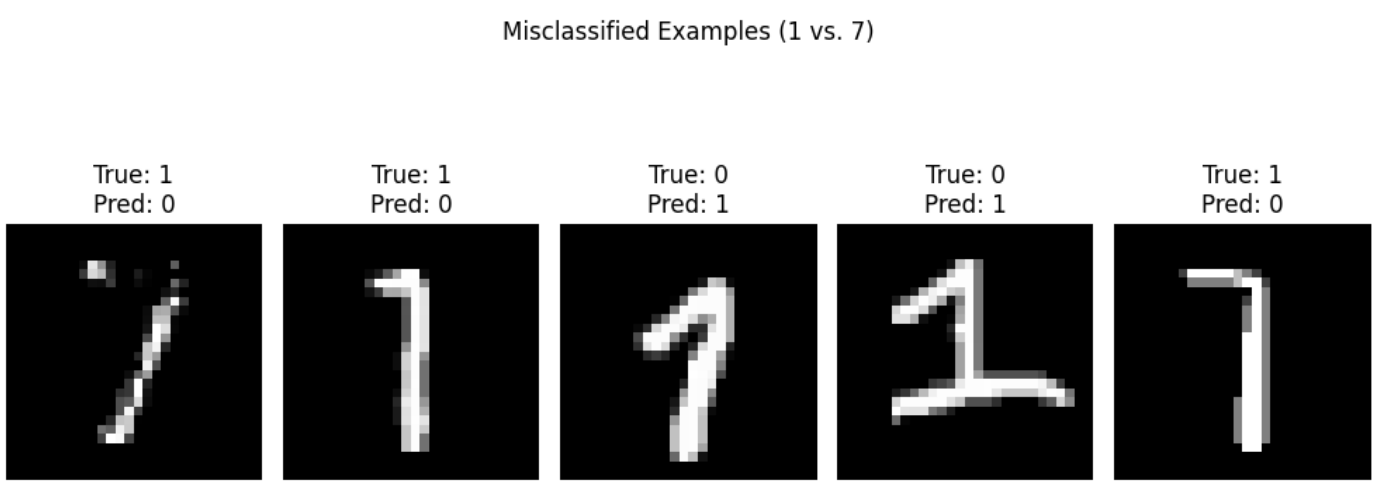
```
# Load MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X_mnist, y_mnist = mnist.data, mnist.target

# Filter for only digits 1 and 7
mask = (y_mnist == '1') | (y_mnist == '7')
X_binary = X_mnist[mask]
y_binary = y_mnist[mask]

# Convert labels to binary (0 for '1', 1 for '7')
y_binary = (y_binary == '7').astype(int)

# Split into train/test sets (80/20 split)
X_train_mnist, X_test_mnist, y_train_mnist, y_test_mnist = train_test_split(
    X_binary, y_binary, test_size=0.2, random_state=42
)

# Train logistic regression (linear model for classification)
mnist_model = LogisticRegression(max_iter=1000)
mnist_model.fit(X_train_mnist, y_train_mnist)

# Evaluate on test set
y_pred = mnist_model.predict(X_test_mnist)
accuracy = accuracy_score(y_test_mnist, y_pred)
conf_matrix = confusion_matrix(y_test_mnist, y_pred)

print(f"Test Accuracy (1 vs. 7): {accuracy:.4f}")
```

```
Test Accuracy (1 vs. 7): 0.9947
```

```
# Create a labeled display
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=['1', '7'])  # Match your binary enco

# Plot with labels
fig, ax = plt.subplots(figsize=(6,6))
disp.plot(cmap='Blues', ax=ax, values_format='d')  # 'd' for integer formatting
plt.title("Confusion Matrix (1 vs 7 Classification)")
plt.grid(False)  # Remove grid lines for cleaner look
plt.show()
```



```
# Plot some misclassified examples
misclassified = np.where(y_pred != y_test_mnist)[0]
plt.figure(figsize=(10, 4))
for i, idx in enumerate(misclassified[:5]):
    plt.subplot(1, 5, i+1)
    plt.imshow(X_test_mnist[idx].reshape(28, 28), cmap='gray')
    plt.title(f"True: {y_test_mnist[idx]}\nPred: {y_pred[idx]}")
    plt.axis('off')
plt.suptitle("Misclassified Examples (1 vs. 7)", y=1.05)
plt.tight_layout()
plt.show()
```



Problem 1. Use logistic regression on the MNIST dataset to build a classifier that distinguishes `3`'s from `8`'s. Evaluate it on a 20/80 test-train split. Print the test set accuracy and a confusion matrix.

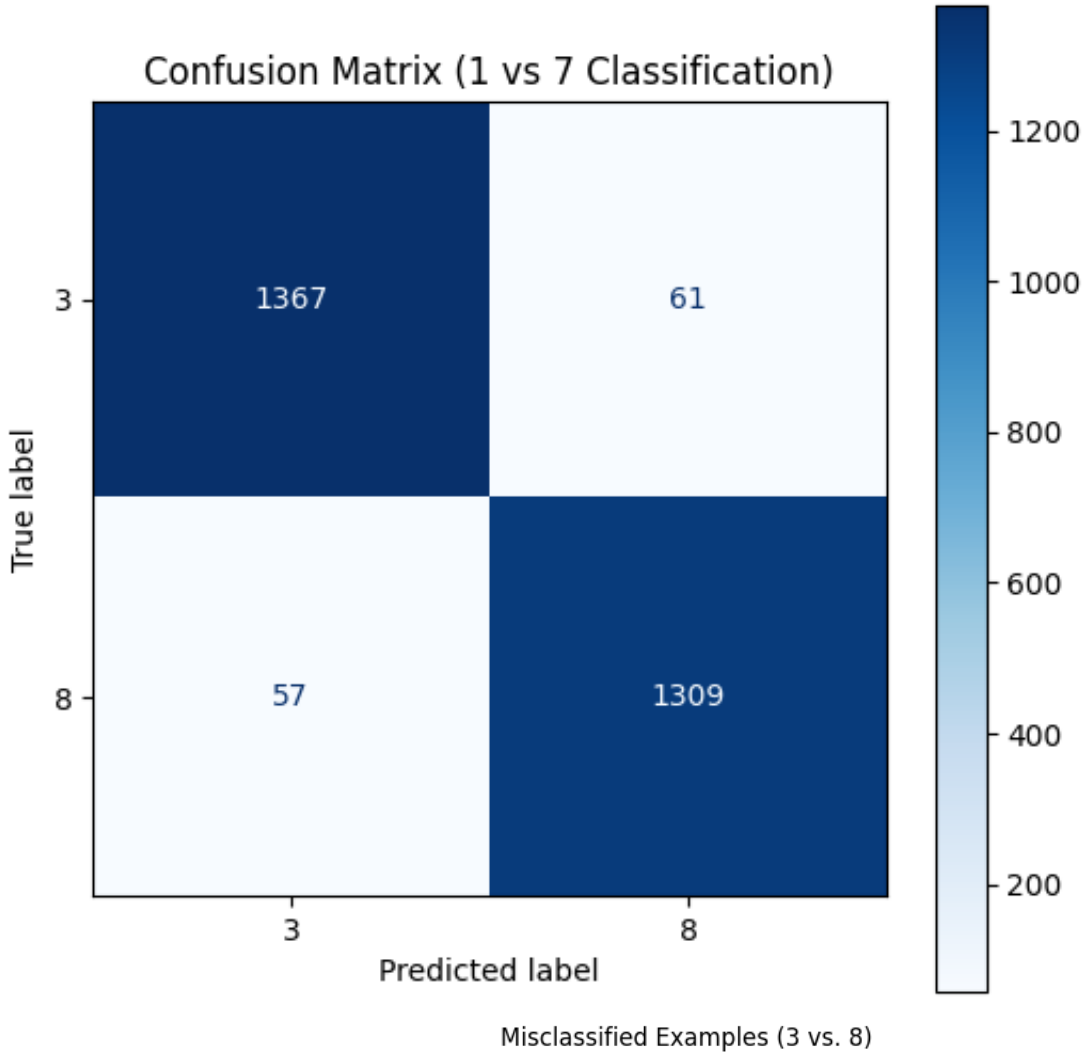
Again, you can do this rather straightfowardly by adapting the previous code. Provide your code below.

```
# Load MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X_mnist, y_mnist = mnist.data, mnist.target
```

```
mask = (y_mnist == '3') | (y_mnist == '8')
X_binary = X_mnist[mask]
y_binary = y_mnist[mask]


y_binary = (y_binary == '8').astype(int)

# Split into train/test sets (80/20 split)
X_train_mnist, X_test_mnist, y_train_mnist, y_test_mnist = train_test_split(
    X_binary, y_binary, test_size=0.2, random_state=42
)

# Train logistic regression (linear model for classification)
mnist_model = LogisticRegression(max_iter=1000)
mnist_model.fit(X_train_mnist, y_train_mnist)

# Evaluate on test set
y_pred = mnist_model.predict(X_test_mnist)
accuracy = accuracy_score(y_test_mnist, y_pred)
conf_matrix = confusion_matrix(y_test_mnist, y_pred)

print(f"Test Accuracy (3 vs. 8): {accuracy:.4f}")


# Create a labeled display
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                              display_labels=['3', '8'])  # Match your binary enco

# Plot with labels
fig, ax = plt.subplots(figsize=(6,6))
disp.plot(cmap='Blues', ax=ax, values_format='d')  # 'd' for integer formatting
plt.title("Confusion Matrix (1 vs 7 Classification)")
plt.grid(False)  # Remove grid lines for cleaner look
plt.show()

# Plot some misclassified examples
misclassified = np.where(y_pred != y_test_mnist)[0]
plt.figure(figsize=(10, 4))
for i, idx in enumerate(misclassified[:5]):
    plt.subplot(1, 5, i+1)
    plt.imshow(X_test_mnist[idx].reshape(28, 28), cmap='gray')
    plt.title(f"True: {y_test_mnist[idx]}\nPred: {y_pred[idx]}")
    plt.axis('off')
plt.suptitle("Misclassified Examples (3 vs. 8)", y=1.05)
plt.tight_layout()
```

```
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:465:
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regress
  n_iter_i = _check_optimize_result(
Test Accuracy (3 vs. 8): 0.9578
```



Part 3: Gradient Descent in Multiple Variables

We will discuss how to perform gradient descent for functions of multiple variables in class. The algorithm is almost identical to the single-variable case. A short description can be found here: https://introml.mit.edu/notes/gradient_descent.html.

Problem 1. A polynomial function in two variables is provided in the text box below. Write a function with three arguments: a learning rate parameter, an initial guess, and a number of iterations. This function should perform gradient descent on the given polynomial with the given parameters. Write your function from scratch; that is, write out the gradient descent iteration logic yourself (and do not just call a function from an optimization library).

Using this function, find the point (x,y) that minimizes the given function, and write it after the bold answer prompt below. Your answer should be supported by the output of your function calls.

Hint. The function is convex, so it has a unique global minimum. You will want to try a few different learning rates and starting points to be confident in your answer. Try decreasing the learning rate (e.g., to 1/100 or even less) and increasing the number of iterations if you have trouble. The coordinates of the minimizing point are integers.

Answer:

The min for the function is (1,2)

```
def gradient_descent(learning_rate, initial_guess, iterations):
    x, y = initial_guess
    for _ in range(iterations):
        dfdx = 4*x**3 + 2*x*y**2 - 12*x**2 - 8*x*y - 2*y**2 + 22*x + 8*y - 14
        dfdy = 4*y**3 + 2*y*x**2 - 24*y**2 - 4*x**2 - 4*x*y+ 52*y + 8*x - 40
        x -= learning_rate * dfdx
        y -= learning_rate * dfdy
    return x, y
gradient_descent(learning_rate=0.01, initial_guess=(0, 0), iterations=10000)
```

```
(0.9999999999999968, 1.9999999999999944)
```

Function for Problem 1. f(x,y) = x^4 + y^4 + x^2y^2 - 4x^3 - 8y^3 - 4x^2y - 2xy^2 + 11x^2 + 26y^2 + 8xy - 14x - 40y + 26

Problem 2. A polynomial function in three variables is provided in the text box below. Write a function with three arguments: a learning rate parameter, an initial guess, and a number of iterations. This function should perform gradient descent on the given polynomial with the given parameters. Write your function from scratch; that is, write out the gradient descent iteration logic yourself (and do not just call a function from an optimization library).

Using this function, find the point (x,y,z) that minimizes the given function, and write it after the bold answer prompt below. Your answer should be supported by the output of your function calls.

Hint. The function is convex, so it has a unique global minimum. You will want to try a few different learning rates and starting points to be confident in your answer. Try decreasing the learning rate (e.g., to 1/100 or even less) and increasing the number of iterations if you have trouble. The coordinates of the minimizing point are integers.

Answer:

The min of this function is (2, 3, 5)

```
def gradient_descent2(learning_rate, initial_guess, iterations):
    x, y, z = initial_guess
    for _ in range(iterations):
        dfdx = 2*x + y - 7
        dfdy = 2*y + x + z - 13
        dfdz = 2*z + y - 13
        x -= learning_rate * dfdx
        y -= learning_rate * dfdy
        z -= learning_rate * dfdz
    return x, y, z
gradient_descent2(learning_rate=0.01, initial_guess=(0, 0, 0), iterations=10000)
```

```
(1.9999999999999698, 3.0000000000000493, 4.999999999999954)
```

Function for Problem 2. f(x,y,z) = x^2 + y^2 + z^2 + xy + yz - 7x - 13y - 13z + 59