0.0.1 Problem 1.4: Plotting Features with Matplotlib

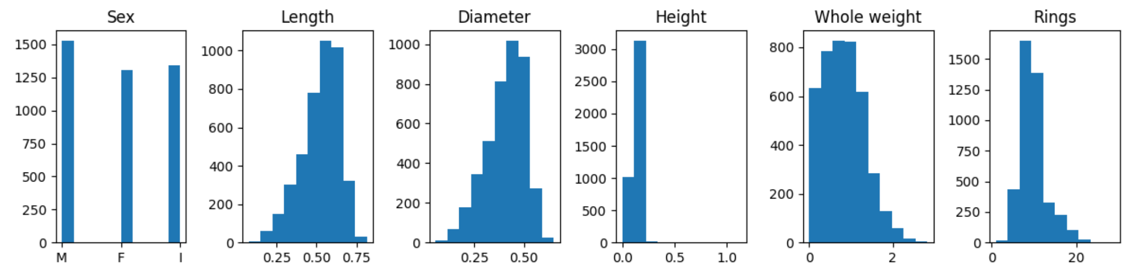
0.0.2 Problem 1.4.a (2 Points): Plotting Feature Histograms with matplotlib

Now, you will visualize the distribution of each feature with histograms. Use `matplotlib.pyplot` to do this, which is already imported for you as `plt`. Do not use any other plotting libraries, such as `seaborn`.

- For every feature in `abalone_clean`, plot a histogram of the values of the feature. Your plot should consist of a grid of subplots with 1 row and 6 columns.
- Include a title above each subplot to indicate which feature we are plotting. For example, if the first feature is "Rings" you can call the first feature "Rings", the second feature "Whole weight", etc.

Some starter code is provided for you below. (Hint: `axes[0].hist(...)` will create a histogram in the first subplot.)

```
In [81]: # Create a figure with 1 row, and 6 columns
         figure, axes = plt.subplots(1, 6, figsize=(12, 3))

         # Select physical features of the abalone (Sex, Length, Diameter, Height, Whole weight, Rings)
         abalone_phys_attr = ["Sex", "Length", "Diameter", "Height", "Whole weight", "Rings"]

         # Plot histogram for each feature
         # Include a title on each subplot
         ### YOUR CODE STARTS HERE ###
         for i, feature in enumerate(abalone_phys_attr):
             axes[i].set_title(feature)
             axes[i].hist(abalone_clean[feature])
         ### YOUR CODE ENDS HERE ###

         plt.tight_layout()
```



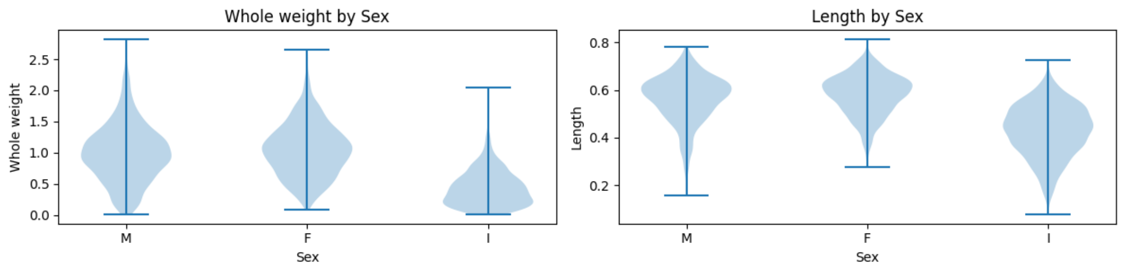
0.0.3 Problem 1.4.b (2 Points): Plotting Distributions per Category with matplotlib

Now, you will visualize the distribution of “Whole weight” and “Length” with Violin plots. Use `matplotlib.pyplot` to do this, which is already imported for you as `plt`. Do not use any other plotting libraries, such as `seaborn`.

- For every class in `abalone_clean["Sex"]`, plot a Violin plot of the values of “Whole weight” and “Length”. Your plot should consist of a grid of subplots with 1 row and 2 columns.
- Include a title above each subplot to indicate which feature we are plotting.

Some starter code is provided for you below. (Hint: `axes[0].violinplot(...)` will create a violinplot in the first subplot.)

```
In [82]: figure, axes = plt.subplots(1, 2, figsize=(12, 3))

         ### YOUR CODE STARTS HERE ###
         # Plot a scatter for each feature pair.
         # Make sure to color the points by their class label.
         # Include an x-label and a y-label for each subplot.
         features = ['Whole weight', 'Length']
         sex_categories = abalone_clean["Sex"].unique()

         for i, feature in enumerate(features):
             data = [abalone_clean[abalone_clean["Sex"] == sex][feature].values for sex in sex_categori
             axes[i].violinplot(data)
             axes[i].set_title(f"{feature} by Sex")
             axes[i].set_xlabel("Sex")
             axes[i].set_ylabel(feature)
             axes[i].set_xticks(range(1, len(sex_categories) + 1))
             axes[i].set_xticklabels(sex_categories)

         plt.tight_layout()
```



0.0.4 Problem 1.5 (10 points): Feature Scatter Plots

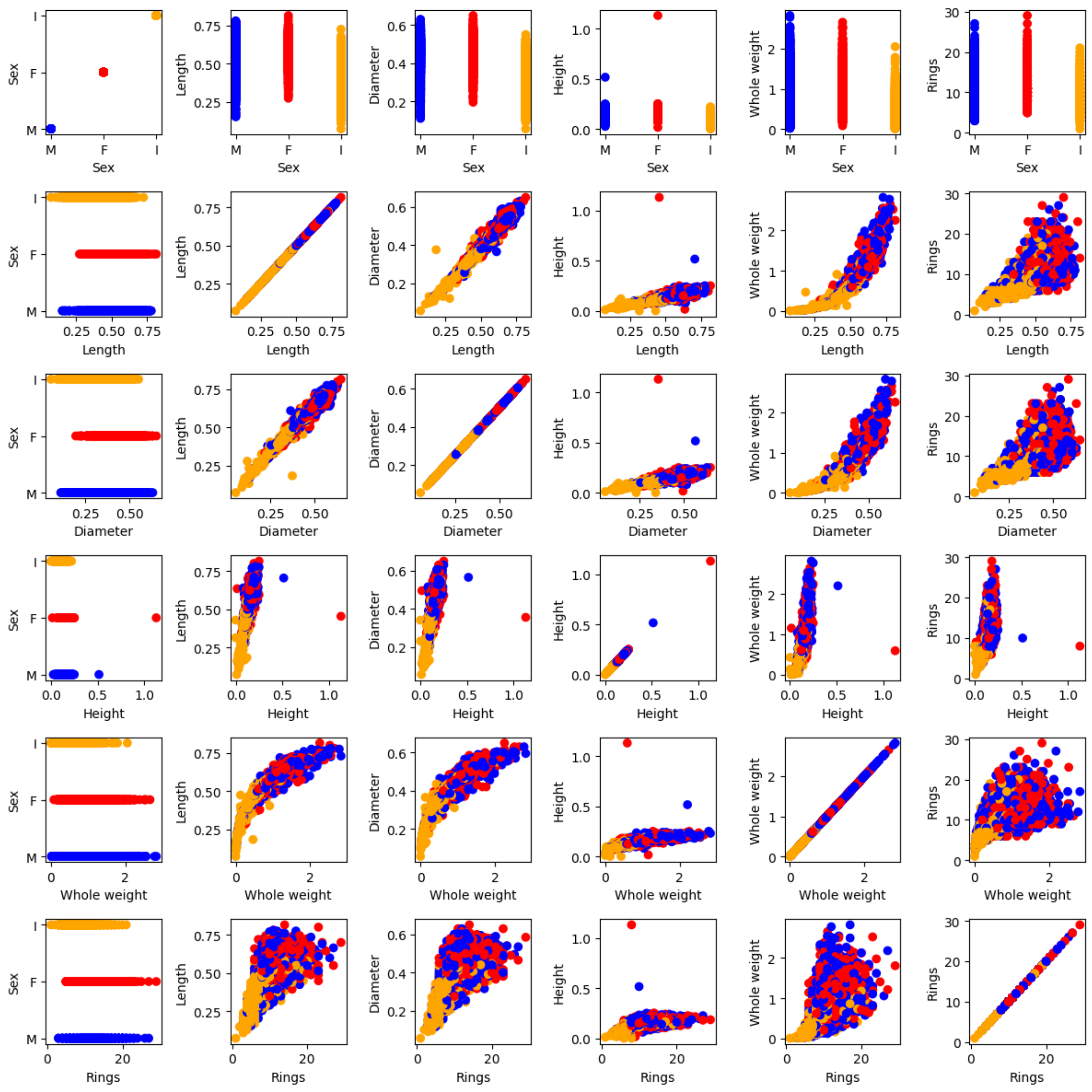
To help further visualize the abalone datset, you will now create several scatter plots of the features. Use `matplotlib.pyplot` to do this, which is already imported for you as `plt`. Do not use any other plotting libraries, such as `seaborn`.

- For every pair of features in `abalone_X`, plot a scatter plot of the feature values, colored according to their labels. For example, plot all data points with `F` to red, `M` to blue, `I` to orange.
- Include an x-label and a y-label on each subplot to indicate which features we are plotting.

Use the feature as the label for the x-axis and y-axis.

Some starter code is provided for you below.

```
In [83]: # create a figure  with 6 rows and 6 columns for features in abalone_clean
         figure, axes = plt.subplots(6, 6, figsize=(12, 12))

         # Define the features to plot
         abalone_X = abalone_clean[abalone_phys_attr]

         # Create a dictionary mapping sex to unique colors
         sex_to_color = {'F': 'red', 'M': 'blue', 'I': 'orange'}

         # Replace sex letter (M,F,I) with integers in the DataFrame
         abalone_color = abalone_clean['Sex'].map(sex_to_color)

         ### YOUR CODE STARTS HERE ###
         # Plot a scatter for each feature pair.
         # Make sure to color the points by their class label.
         # Include an x-label and a y-label for each subplot.
         for i, feature in enumerate(abalone_phys_attr):
             for j, feature in enumerate(abalone_phys_attr):
                 axes[i,j].scatter(abalone_X.iloc[:,i], abalone_X.iloc[:,j], c = abalone_color)
                 axes[i,j].set_xlabel(abalone_phys_attr[i])
                 axes[i,j].set_ylabel(abalone_phys_attr[j])



         plt.tight_layout()
```

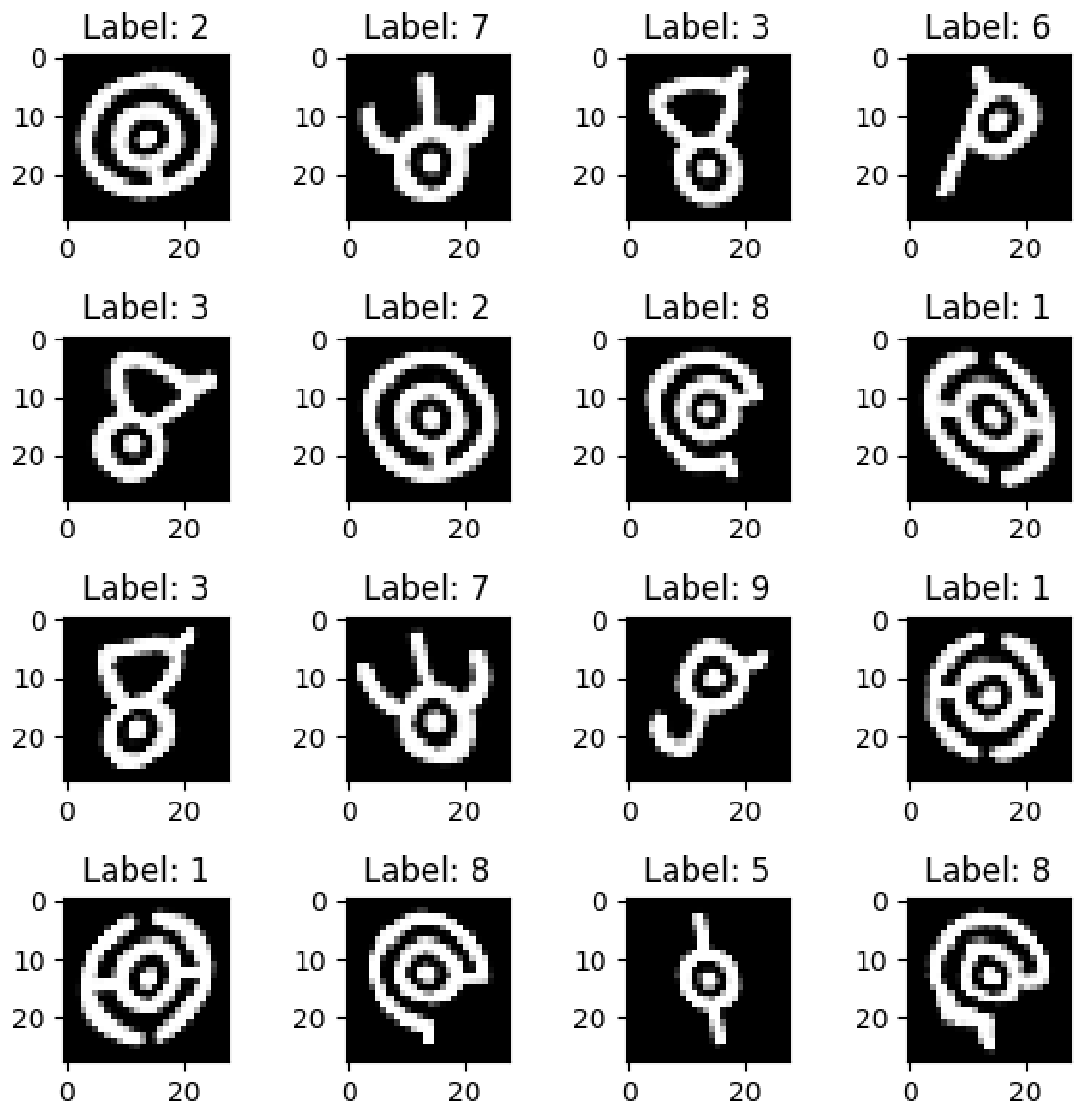

Sex
Length
Diameter
Height
Whole weight
Rings
M
F
I

0.0.5 Problem 2.1 (5 points): Unown-MNIST Visualization

Let's begin by visualizing a few of the images in the Unown-MNIST dataset. Use `matplotlib.pyplot` to do this, which is already imported for you as `plt`. Do not use any other plotting libraries, such as `seaborn`.

- Plot the first 16 images in `unown_X_tr` in a 4x4 grid. (Hint 1 Hint 2)
- Include a title for each subplot indicating the label of the image.

```
In [86]: # Some default settings for our plots
         plt.rcParams['image.interpolation'] = 'nearest'
         plt.rcParams['image.cmap'] = 'gray'

         # Create a figure with 4 rows and 4 columns
         figure, axes = plt.subplots(4, 4, figsize=(6, 6))

         ### YOUR CODE STARTS HERE ###
         # Plot the first 16 images in our dataset.
         # Include a title on each subplot to indicate the corresponding label.
         # ( 5 lines of code)
         for i, ax in enumerate(axes.flat[:16]):
             ax.imshow(unown_X_tr[i].reshape(28, 28))
             ax.set_title(f"Label: {unown_y_tr[i]}")



         ### YOUR CODE ENDS HERE  ###

         plt.tight_layout()
```


Label: 2
Label: 7
Label: 3
Label: 6
Label: 3
Label: 2
Label: 8
Label: 1
Label: 3
Label: 7
Label: 9
Label: 1
Label: 1
Label: 8
Label: 5
Label: 8

0.0.6 Problem 2.2 (20 points): Implementing a Nearest Centroid Classifier

In the code given below, we define the class `NearestCentroidClassifier` which has an unfinished implementation of a nearest centroid classifier. For this problem, you will complete this implementation. Your nearest centroid classifier will use the Euclidean distance, which is defined for two feature vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ as

$$d_E(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{j=1}^{d} (x_{1j} - x_{2j})^2}.$$

- Implement the method `fit`, which takes in an array of features `X` and an array of labels `y` and trains our classifier. You should store your computed centroids in the list `self.centroids`.
- Test your implementation of `fit` by training a `NearestCentroidClassifier` on the Unown-MNIST training set, and using the provided method `plot_centroids` to visualize the centroids. If your implementation is correct, the centroids should resemble the corresponding class label in the plot.
- Implement the method `predict`, which takes in an (array of) feature vectors `X` and predicts their class labels.
- Print the predicted labels (using your `predict` function) and the true labels for the first ten images in the Unown-MNIST testing set. Make sure to indicate which are the predicted labels and which are the true labels.

You are allowed to modify the given code as necessary to complete the problem, e.g. you may create helper functions. Leave the type hints and function signatures as is.

```
In [87]: class NearestCentroidClassifier:
             def __init__(self):
                 # A list containing the centroids; to be filled in with the fit method.
                 self.centroids = []

             def plot_centroids(self):
                 # Some default settings for our plots
                 plt.rcParams['image.interpolation'] = 'nearest'
                 plt.rcParams['image.cmap'] = 'gray'

                 # Create a figure with 2 rows and 5 columns
                 figure, axes = plt.subplots(2, 5, figsize=(12, 4))

                 # Plot the centroids
                 for i in range(10):
                     axes[i//5, i%5].imshow(self.centroids[i].reshape(28, 28))
                     axes[i//5, i%5].set_title(f'Label: {i}')

                 plt.tight_layout()
                 plt.show()

             def fit(self, X: np.ndarray, y: np.ndarray) -> None:
                 """
```

```
        Fits the nearest centroid classifier with training features X and training labels y.

        Parameters:
        X (np.ndarray): array of training features; shape (n, d), where n is the number of dat
        y (np.ndarray): array training labels; shape (n, ), where n is the number of datapoint

        Returns:
        None
        """
        ### YOUR CODE STARTS HERE ###
        # Hint: you should append to self.centroids with the corresponding centroids.
        for label in np.unique(y):
            curr = X[y == label]
            centroid = np.mean(curr, axis=0)
            self.centroids.append(centroid)




        ### YOUR CODE ENDS HERE

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Makes predictions with the nearest centroid classifier on the features in X.

        Parameters:
        X (np.ndarray): array of features; shape (n, d), where n is the number of datapoints,

        Returns:
        y_pred (np.ndarray): a numpy array of predicted labels; shape (n, ), where n is the nu
        """
        ### YOUR CODE STARTS HERE ###
        y_pred = []
        for item in X:
            distance = {}
            for label, centroid in enumerate(self.centroids):
                distance[label] = np.linalg.norm(item - centroid)
            predicted_label = min(distance, key=distance.get)
            y_pred.append(predicted_label)
        ### YOUR CODE ENDS HERE ###
        return y_pred
```

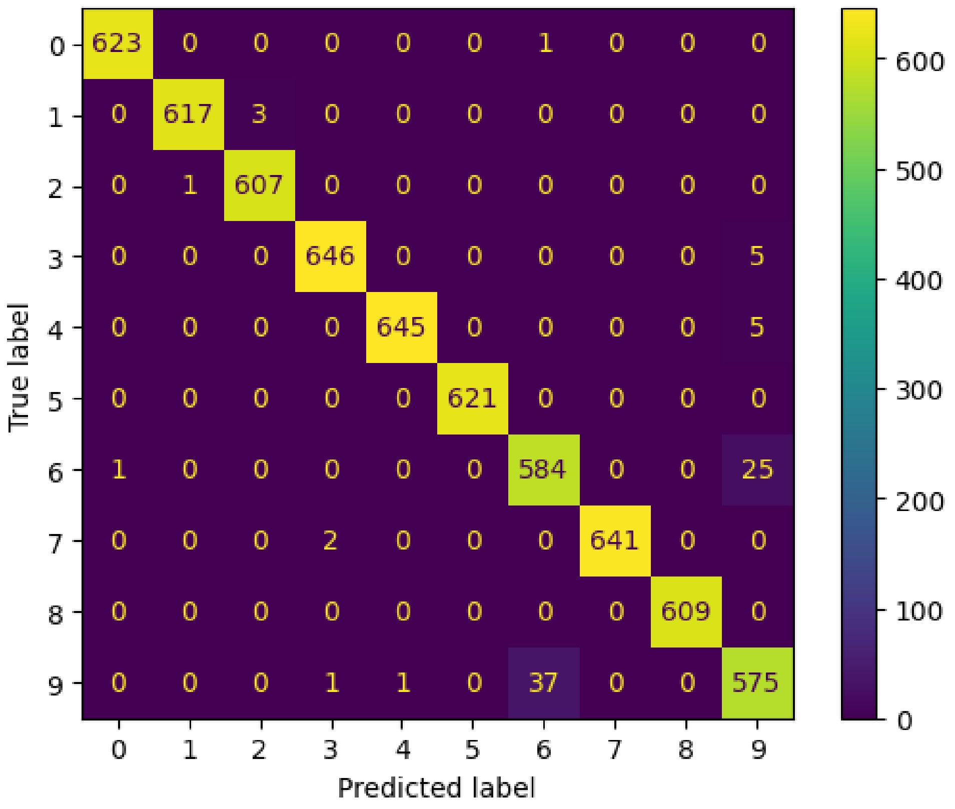
```
In [92]: def compute_accuracy(y, y_pred):
             ### YOUR CODE STARTS HERE ###

             accuracy = np.mean(y_pred == y)

             ### YOUR CODE ENDS HERE ###

             return accuracy
```

```
In [93]: def compute_confusion_matrix(y, y_pred):

             ### YOUR CODE STARTS HERE ###
             n = len(np.unique(y))
             conf_matrix = []
             for i in range(n):
                 row = []
                 for j in range(n):
                     row.append(0)
                 conf_matrix.append(row)

             for true_label, pred_label in zip(y, y_pred):
                 conf_matrix[true_label][pred_label] += 1
             ### YOUR CODE ENDS HERE ###
             return np.array(conf_matrix)
```

```
In [94]: ###############################################
         ### Results with the sklearn implementation ###
         ###############################################

         def eval_sklearn_implementation(X_tr, y_tr, X_te, y_te):
             # Nearest centroid classifier implemented in sklearn
             sklearn_nearest_centroid = NearestCentroid()

             # Fit on training dataset
             sklearn_nearest_centroid.fit(X_tr, y_tr)

             # Make predictions on training and testing data
             sklearn_y_pred_tr = sklearn_nearest_centroid.predict(X_tr)
             sklearn_y_pred_te = sklearn_nearest_centroid.predict(X_te)

             # Evaluate accuracies using the sklearn function accuracy_score
             sklearn_acc_tr = accuracy_score(y_tr, sklearn_y_pred_tr)
             sklearn_acc_te = accuracy_score(y_te, sklearn_y_pred_te)

             print(f'Sklearn Results:')
             print(f'--- Accuracy (train): {sklearn_acc_tr}')
             print(f'--- Accuracy (test): {sklearn_acc_te}')

             # Evaluate confusion matrix using the sklearn function confusion_matrix
             sklearn_cm = confusion_matrix(y_te, sklearn_y_pred_te)
             sklearn_disp = ConfusionMatrixDisplay(confusion_matrix = sklearn_cm)
```

```
    sklearn_disp.plot()

# Call the function
unown_X_tr_flattened = unown_X_tr.reshape((18750, 784))
unown_X_te_flattened = unown_X_te.reshape((6250, 784))

non_zero_var = []

for i in range(784):
    unique = np.unique(unown_X_tr_flattened[:, i])
    if len(unique) != 1:
        non_zero_var.append(i)

print(len(non_zero_var))

eval_sklearn_implementation(unown_X_tr_flattened[:, non_zero_var], unown_y_tr, unown_X_te_flat
```

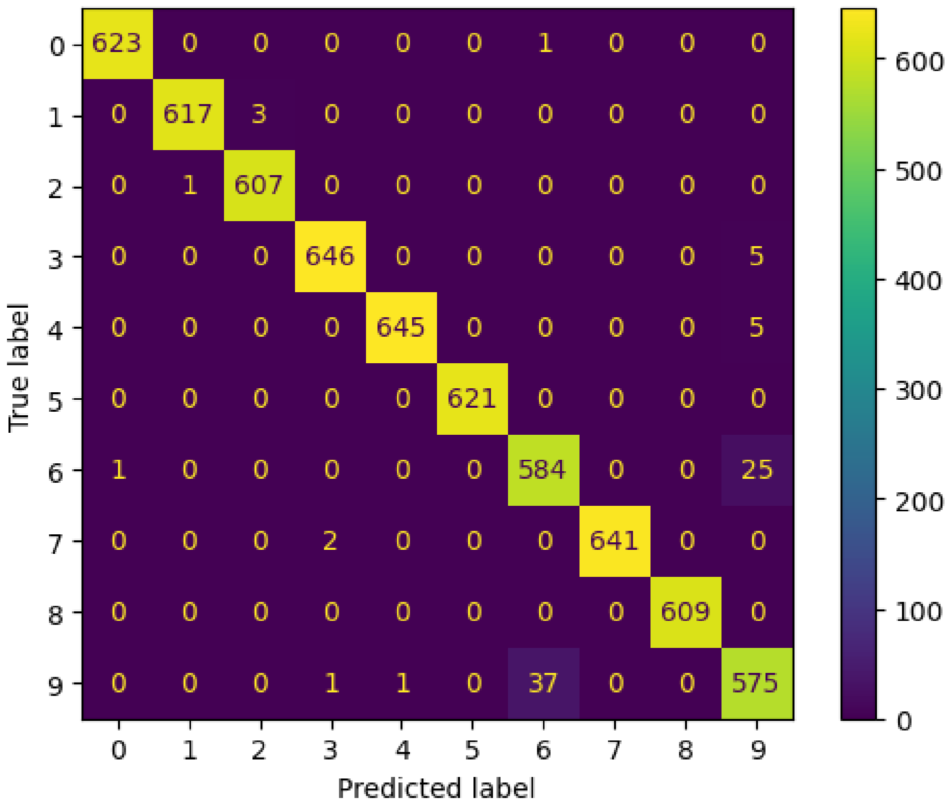
```
643
Sklearn Results:
--- Accuracy (train): 0.9859733333333334
--- Accuracy (test): 0.98688
```



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	623	0	0	0	0	0	1	0	0	0
1	0	617	3	0	0	0	0	0	0	0
2	0	1	607	0	0	0	0	0	0	0
3	0	0	0	646	0	0	0	0	0	5
4	0	0	0	0	645	0	0	0	0	5
5	0	0	0	0	0	621	0	0	0	0
6	1	0	0	0	0	0	584	0	0	25
7	0	0	0	2	0	0	0	641	0	0
8	0	0	0	0	0	0	0	0	609	0
9	0	0	0	1	1	0	37	0	0	575

```
In [95]: ##########################################
         ### Results with your implementation ###
         ##########################################

         def eval_my_implementation(X_tr, y_tr, X_te, y_te):
             # Now test your implementation of NearestCentroidClassifier
             nearest_centroid = NearestCentroidClassifier()

             # Fit on training dataset
             nearest_centroid.fit(X_tr, y_tr)

             # Make predictions on training and testing data
             y_pred_tr = nearest_centroid.predict(X_tr)
             y_pred_te = nearest_centroid.predict(X_te)

             # Evaluate accuracies using your function compute_accuracy
             acc_tr = compute_accuracy(y_tr, y_pred_tr)
             acc_te = compute_accuracy(y_te, y_pred_te)

             print(f'Your Results:')
             print(f'--- Accuracy (train): {acc_tr}')
             print(f'--- Accuracy (test): {acc_te}')

             # Evaluate confusion matrix using your function compute_confusion_matrix
             cm = compute_confusion_matrix(y_te, y_pred_te)
             disp = ConfusionMatrixDisplay(confusion_matrix = cm)
             disp.plot();

         # Call the function
         eval_my_implementation(unown_X_tr, unown_y_tr, unown_X_te, unown_y_te)
```

```
Your Results:
--- Accuracy (train): 0.9859733333333334
--- Accuracy (test): 0.98688
```



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	623	0	0	0	0	0	1	0	0	0
1	0	617	3	0	0	0	0	0	0	0
2	0	1	607	0	0	0	0	0	0	0
3	0	0	0	646	0	0	0	0	0	5
4	0	0	0	0	645	0	0	0	0	5
5	0	0	0	0	0	621	0	0	0	0
6	1	0	0	0	0	0	584	0	0	25
7	0	0	0	2	0	0	0	641	0	0
8	0	0	0	0	0	0	0	0	609	0
9	0	0	0	1	1	0	37	0	0	575

0.0.7 Problem 3.1: Decision Boundaries

Problem 3.1.a (4 points): Train/Test Split

- Using the code in Problem 1 and Problem 2, to create a train/test split of the Abalone dataset containing two features `['Whole weight','Length']`. Use 75% of the data for training, and 25% of the data for testing. Set `shuffle=True` and be sure to use `random_state=hw1_seed`.
- Fit a kNN classifier on this new training set, and plot the resulting decision boundary for values of `k = [1, 5, 10, 50]`.
- Write a short description of what you see happen as you increase the value of `k`.

Here are a few tips to help you get started. - In `sklearn`, you can create a kNN classifier with `k` neighbors via `knn = KNeighborsClassifier(n_neighbors=k)`. - You can then use `knn.fit(...)` and `knn.predict(...)` to fit the classifier and make predictions with it. - See here for the corresponding documentation.

```
In [96]: # First define the abalone_y values for the abalone dataset
         abalone_y = abalone_clean['Sex']
```

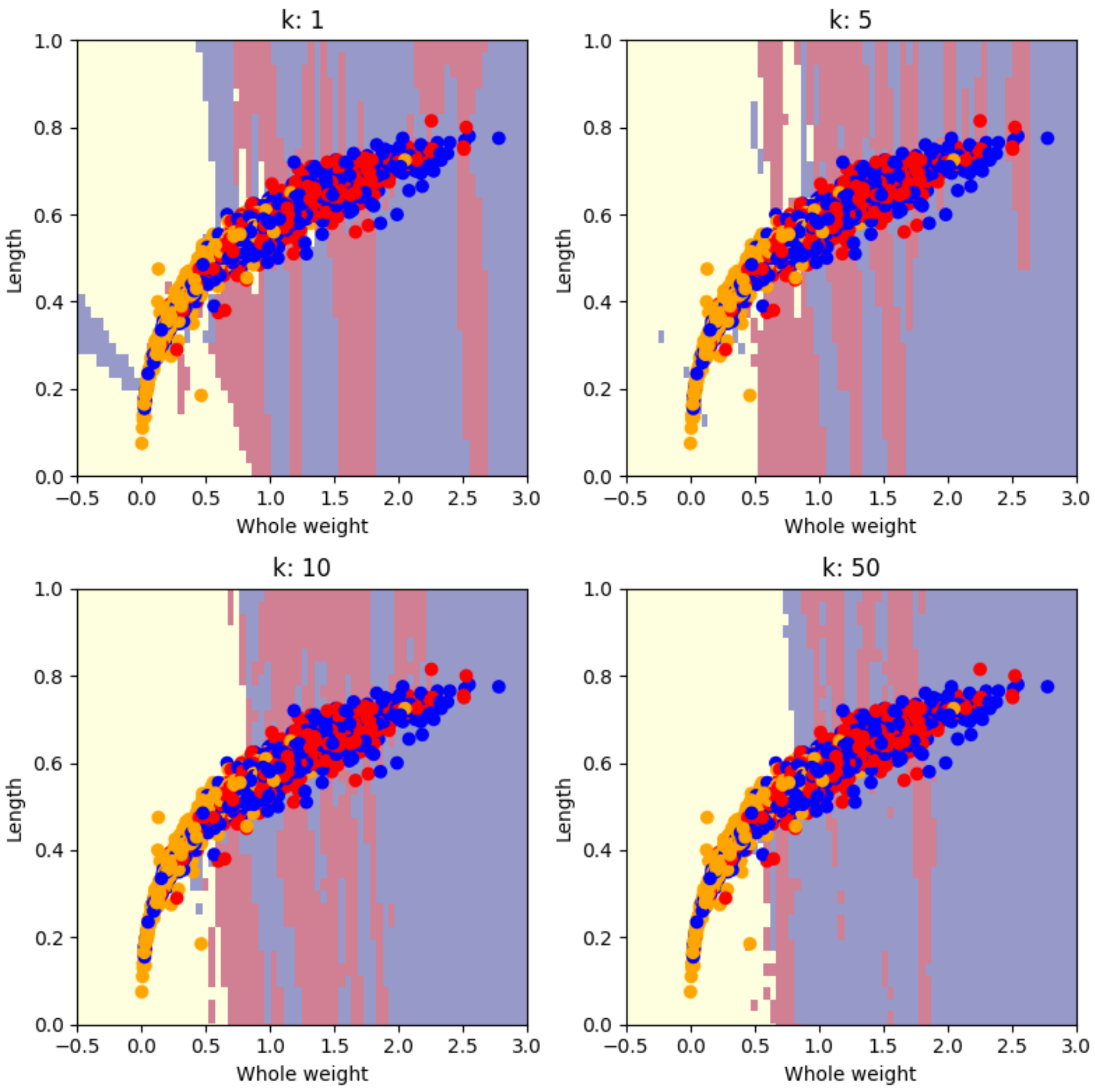
```
In [97]: # Create a 75%/25% train/test split using 'Whole weight' and 'Length' features
         ### YOUR CODE STARTS HERE ###
         abalone_X_tr2, abalone_X_te2, abalone_y_tr2, abalone_y_te2 = train_test_split(abalone_clean[['
         ###  YOUR CODE ENDS HERE  ###
         abalone_X_tr2
         abalone_X_te2
         abalone_y_tr2
         abalone_y_te2
```

```
Out[97]: 306     M
         272     M
         247     I
         1578    I
         3196    I
                ..
         1039    F
         13      F
         2330    F
         3127    F
         127     M
         Name: Sex, Length: 1045, dtype: category
         Categories (3, object): ['F', 'I', 'M']
```

Problem 3.1.b (10 points): KNN Decision Boundary Plot

- Fit a kNN classifier on this new training set, and plot the resulting decision boundary for values of `k = [1, 5, 10, 50]`.
- Write a short description of what you see happen as you increase the value of `k`.
- Hint: this function might be useful for the plot.

```
In [98]: # Plot the decision boundaries for the kNN classifiers for various values of k

         # Some keyword arguments for making nice looking plots.
         # Feel free to change grid_resolution to a higher number -- this results in better looking plo
         # but may result in your code running more slowly.
         plot_kwargs = {'cmap': 'RdYlBu',
                        'response_method': 'predict',
                        'plot_method': 'pcolormesh',
                        'shading': 'auto',
                        'alpha': 0.5,
                        'grid_resolution': 100}



         # Create a figure with 2 rows and 2 columns
         figure, axes = plt.subplots(2, 2, figsize=(8, 8))

         abalone_color = abalone_y_tr2.map(sex_to_color)

         k_vals = [1, 5, 10, 50]
         for i, k in enumerate(k_vals):
             knn = KNeighborsClassifier(n_neighbors = k)
             knn.fit(abalone_X_tr2[['Whole weight', 'Length']], abalone_y_tr2)
             DecisionBoundaryDisplay.from_estimator(knn, abalone_X_tr2[['Whole weight', 'Length']], ax=

             axes[i//2, i%2].scatter(abalone_X_tr2['Whole weight'], abalone_X_tr2['Length'], c= abalone
             axes[i//2, i%2].set_xlim((-0.5, 3))
             axes[i//2, i%2].set_ylim((0, 1))
             axes[i//2, i%2].set_title(f"k: {k}")
             axes[i // 2][i % 2].set_xlabel("Whole weight")
             axes[i // 2][i % 2].set_ylabel("Length")
         plt.tight_layout()
         plt.show()
```


k: 1
k: 5
k: 10
k: 50
Length
Whole weight

0.0.8 Problem 3.2 (15 points): Error Rates vs k

Now, we will vary the value of k and see what effect this has on our predictions.

- Again, using only the given two features in the Abalone dataset, compute the error rate on both the training and testing data as a function of `k`. Do this for all values of `k = [1, 2, 5, 10, 50, 100, 110]`. You may use your own implementation of the accuracy from Problem 2.3, or the scikit-learn function `sklearn.metrics.accuracy_score`.
- Plot the resulting error rate functions using a semi-log plot (i.e. the x-axis is on a logarithmic scale), with the training error in red and the validation error in green. This can be done using `axes[0].semlilogx(...)`. Use `matplotlib.pyplot` to do this, which is already imported for you as `plt`. Do not use any other plotting libraries, such as `pandas` or `seaborn`.
- What value of `k` would you recommend, and why?

```
In [ ]: # Create a figure with only one subplot
        figure, axes = plt.subplots(1, figsize=(6, 6))
        ### YOUR CODE STARTS HERE ###
        k_vals = [1, 2, 5, 10, 50, 100, 110]
        errorsTr = []
        errorsTe = []
        for k in k_vals:
            knn = KNeighborsClassifier(n_neighbors = k)
            knn.fit(abalone_X_tr2[['Whole weight', 'Length']], abalone_y_tr2)
            sklearn_y_pred_tr = knn.predict(abalone_X_tr2[['Whole weight', 'Length']])
            sklearn_y_pred_te = knn.predict(abalone_X_te2[['Whole weight', 'Length']])
            sklearn_acc_tr = accuracy_score(abalone_y_tr2, sklearn_y_pred_tr)
            sklearn_acc_te = accuracy_score(abalone_y_te2, sklearn_y_pred_te)
            print(f'Your Results:')
            print(f'--- Accuracy (train): {sklearn_acc_tr}')
            print(f'--- Accuracy (test): {sklearn_acc_te}')
            errorsTr.append(1 - sklearn_acc_tr)
            errorsTe.append(1 - sklearn_acc_te)


        axes.semilogx(k_vals, errorsTr, linestyle = '-', color = 'red', label = 'Train')
        axes.semilogx(k_vals, errorsTe, linestyle = '-', color = 'green', label = 'Test')
        axes.set_ylabel('Error Rate')
        axes.set_ybound(0,.6)
        axes.set_xlabel('k')
        plt.legend(loc = "upper right")
        plt.tight_layout()
        plt.show()

        ###  YOUR CODE ENDS HERE  ###
```

```
Your Results:
--- Accuracy (train): 0.9833971902937421
--- Accuracy (test): 0.4421052631578947
Your Results:
```

```
--- Accuracy (train): 0.7385057471264368
--- Accuracy (test): 0.46507177033492825
Your Results:
--- Accuracy (train): 0.6516602809706258
--- Accuracy (test): 0.4842105263157895
Your Results:
--- Accuracy (train): 0.6273946360153256
--- Accuracy (test): 0.49473684210526314
Your Results:
--- Accuracy (train): 0.5740740740740741
--- Accuracy (test): 0.5291866028708134
Your Results:
--- Accuracy (train): 0.5622605363984674
--- Accuracy (test): 0.5473684210526316
```

0.0.9 Statement of Collaboration (4 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using EdD) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to EdD, etc.). Especially after you have started working on the assignment, try to restrict the discussion to EdD as much as possible, so that there is no doubt as to the extent of your collaboration.