final-exam

August 26, 2024

1 Problem Set 1

Due: AUGUST 27, before class (before 4:00 PM).

How to submit

You need to send the `.ipynb` file with your answers plus an `.html` file, which will serve as a backup for us in case the `.ipynb` file cannot be opened on my or the TA's computer. In addition, you may also export the notebook as PDF and attach it to the email as well.

Please use the following subject header for sending in your homework, so that we can make sure that nothing gets lost:

Your id mentioned on offer letter: Abdullah Jahangir
.

Note No help will be provided bny instructor you have to do it on your own.

you will get certificates on basis its result

[81]:
```
from watermark import watermark
print(watermark(author="Abdullah Jahangir",␣
  ↪packages="numpy,scipy,matplotlib,sklearn"))
print(watermark())
```

```
Author: Abdullah Jahangir

numpy     : 1.26.4
scipy     : 1.14.0
matplotlib: 3.9.1
sklearn   : 1.5.1

Last updated: 2024-08-26T18:22:33.096563+05:00

Python implementation: CPython
Python version       : 3.11.7
IPython version      : 8.26.0

Compiler    : MSC v.1937 64 bit (AMD64)
OS          : Windows
Release     : 10
```

```
Machine     : AMD64
Processor   : Intel64 Family 6 Model 154 Stepping 4, GenuineIntel
CPU cores   : 12
Architecture: 64bit
```

The watermark package that is being used in the next code cell provides a helper function of the same name, `%watermark` for showing information about your computational environment. This is usefult to keep track of what software versions are/were being used. If you should encounter issues with the code, please make sure that your software package have the same version as the the ones shown in the pre-executed watermark cell.

Before you execute the watermark cell, you need to install watermark first. If you have not done this yet. To install the watermark package, simply run

```
!pip install watermark
```

or

```
!conda install watermark -c conda-forge
```

in the a new code cell. Alternatively, you can run either of the two commands (the latter only if you have installed Anaconda or Miniconda) in your command line terminal (e.g., a Linux shell, the Terminal app on macOS, or Cygwin, Putty, etc. on Windows).

For more information installing Python, please refer to the previous lectures and ask the TA for help.

1.1 E 1)

Pick 3 machine learning application examples from the first lecture (see section 1.2 in the lecture notes, shared in group and answer the following questions:

- What is the overall goal?
- How would an appropriate dataset look like?
- Which general machine learning category (supervised, unsupervised, reinforcement learning) does this problem fit in?
- How would you evaluate the performance of your model (in very general, non technical terms)

Example – Email Spam classification:

- **Goal.** A potential goal would be to learn how to classify emails as spam or non-spam.
- **Dataset.** The dataset is a set consisting of emails as text data and their spam and non-spam labels.
- **Category.** Since we are working with class labels (spam, non-spam), this is a supervised learning problem.
- **Measure Performance.** Predict class labels in the test dataset and count the number of correct predictions to asses the prediction accuracy.

1.2 Answer:

1. Stock Prediction:

Goal: The goal is to forecast the future prices of stocks based on historical data. This can help investors make informed decisions about buying, holding, or selling stocks.

Dataset: The dataset would typically include historical stock prices, trading volumes, and other financial indicators. It might also incorporate additional features such as economic indicators, company financial reports, and news sentiment. Each data point usually includes attributes like date, open price, close price, high price, low price, volume, etc.

Category: This is a supervised learning problem if you're predicting specific future stock prices or price movements based on labeled historical data. If you're clustering stocks into categories based on similar price movements or patterns, it could be unsupervised learning. Reinforcement learning might be used if the model is part of a trading strategy that learns to make trading decisions based on simulated rewards.

Measure Performance: Performance can be evaluated using metrics like mean squared error (MSE) or mean absolute error (MAE) for continuous predictions. For classification tasks (e.g., predicting whether the stock will go up or down), accuracy, precision, recall, and F1-score can be used. Additionally, financial metrics such as return on investment (ROI) and Sharpe ratio might be relevant to assess the effectiveness of trading strategies derived from the predictions.

2. Drug Design:

Goal: The goal is to predict the efficacy and safety of new drug compounds by modeling their interactions with biological targets. This can help in identifying promising candidates for further testing and reducing the number of unsuccessful drugs that reach clinical trials.

Dataset: The dataset would consist of chemical properties of drug compounds, biological activity data (e.g., binding affinity, toxicity), and possibly information about the biological targets (e.g., protein structures). Each data point would include features representing the compound's structure or properties and labels indicating its activity or efficacy.

Category: This is a supervised learning problem because it involves predicting outcomes (such as activity or toxicity) based on labeled data. Alternatively, it can also involve unsupervised learning if you're clustering compounds into groups based on similarity without predefined labels.

Measure Performance Evaluate the model's performance by comparing its predictions against known data. For classification tasks, you might use metrics like accuracy, precision, recall, or F1 score. For regression tasks, you could use mean squared error (MSE) or mean absolute error (MAE) to assess how close the predictions are to actual values.

3. Credit Card Fraud:

Goal: The goal is to identify fraudulent transactions among credit card transactions to prevent financial loss and protect cardholders.

Dataset: The dataset would include transaction records with features such as transaction amount, transaction time, merchant details, and cardholder information. Each record would be labeled as fraudulent or non-fraudulent. The dataset might be imbalanced, with a much smaller proportion of fraudulent transactions compared to non-fraudulent ones.

Category: This is a supervised learning problem since it involves predicting a categorical outcome (fraudulent or non-fraudulent) based on labeled data.

Measure Performance: Evaluate the model by examining how well it identifies fraudulent transactions. Metrics like precision, recall, F1 score, and the area under the ROC curve (AUC-ROC) are useful because they handle class imbalance better than simple accuracy. Precision measures the proportion of correctly identified frauds among all identified frauds, while recall measures the proportion of actual frauds that were identified.

1.3 E 2)

If you think about the task of spam classification more thoroughly, do you think that the classification accuracy or misclassification error is a good error metric of how good an email classifier is? What are potential pitfalls? (Hint: think about false positives [non-spam email classified as spam] and false negatives [spam email classified as non-spam]).

1.4 Answer:

Using classification accuracy or misclassification error as the sole metric may not fully capture the performance of the classifier due to potential pitfalls:

Class Imbalance: If the dataset has a significant imbalance between spam and non-spam emails (e.g., 90% non-spam and 10% spam), a classifier could achieve high accuracy by simply predicting all emails as non-spam. This would result in a high accuracy rate but fail to identify any spam emails, which is a critical issue.

False Positives: A false positive occurs when a non-spam email is incorrectly classified as spam. This could lead to important emails being wrongly placed in the spam folder, causing inconvenience to users who might miss important messages.

False Negatives: A false negative occurs when a spam email is incorrectly classified as non-spam. This is problematic as it allows unwanted spam to reach the user's inbox, potentially leading to security risks and reduced user satisfaction.

Better Metrics to Consider:

Precision: Measures the proportion of true spam emails among those classified as spam. High precision means that most emails classified as spam are indeed spam, which is important to minimize false positives.

Recall: Measures the proportion of actual spam emails that are correctly classified. High recall means that most spam emails are identified, which is important to minimize false negatives.

F1 Score: The harmonic mean of precision and recall, providing a balance between the two metrics. It is particularly useful when dealing with imbalanced datasets.

Area Under the ROC Curve (AUC-ROC): Evaluates the classifier's ability to distinguish between spam and non-spam across various threshold settings, providing a comprehensive view of performance.

In summary, while accuracy and misclassification error provide a general sense of performance, precision, recall, and the F1 score offer more insights into the classifier's effectiveness, especially in handling class imbalances and minimizing both false positives and false negatives.

1.5 E 3)

In the exercise example of E 1), email spam classification was listed as an example of a supervised machine learning problem. List 2 examples of unsupervised learning tasks that would fall into the category of clustering. In one or more sentences, explain why you would describe these examples as clustering tasks and not supervised learning tasks. Select examples that are not already that are in the "Lecture note list" from E 1).

1.6 Answer:

1. Customer Segmentation:

Description: Customer segmentation involves grouping customers based on their purchasing behavior, demographic information, or other attributes. The goal is to identify distinct groups of customers with similar characteristics or behaviors.

Reason for Clustering: This task is unsupervised because we do not have predefined labels or categories for the customer groups. Instead, we use clustering algorithms to discover inherent groupings within the data based on similarity measures. The objective is to find meaningful patterns and group customers into clusters without any prior knowledge of the number of segments or their characteristics.

2. Document Clustering:

Description: Document clustering involves grouping a set of documents into clusters where each cluster contains documents that are similar to each other. This can be useful for organizing large collections of text documents into thematic groups.

Reason for Clustering: This task is unsupervised because there are no pre-assigned categories or labels for the documents. Clustering algorithms analyze the content of the documents to find patterns and group similar documents together based on their textual similarity. The clusters are formed based on the similarity of the content rather than predefined categories.

In both cases, the goal is to explore and discover natural groupings within the data without relying on predefined labels, which is why these tasks are considered clustering problems rather than supervised learning problems.

1.7 E 4)

In the k-nearest neighbor (k-NN) algorithm, what computation happens at training and what computation happens at test time? Explain your answer in 1-2 sentences.

1.8 Answer:

In the k-nearest neighbor (k-NN) algorithm:

Training Time: The algorithm stores the training data and associated labels, which involves simply memorizing the dataset without any computation for model parameters.

Test Time: For each test sample, the algorithm computes the distances between the test sample and all training samples to identify the k nearest neighbors, and then classifies the test sample based on the majority label among these neighbors.

This approach results in minimal computation during training but can be computationally intensive during testing due to the distance calculations required.

1.9 E 5)

Does (k-NN) work better or worse if we add more information by adding more feature variables (assuming the number of training examples is fixed)? Explain your reasoning.

1.10 Answer:

In the k-nearest neighbor (k-NN) algorithm, adding more feature variables can have mixed effects:

Better Performance: If the additional features are relevant and provide useful information about the data, they can help k-NN make more accurate classifications by offering a richer representation of the data.

Worse Performance: If the additional features are noisy or irrelevant, they can lead to poorer performance. This is because k-NN relies on distance metrics in the feature space, and irrelevant features can distort these distances, making it harder to identify true nearest neighbors. Additionally, more features can increase the risk of the "curse of dimensionality," where the distance between points becomes less meaningful as the number of dimensions increases.

Overall, the impact of adding more features depends on their relevance and the ability of the model to handle high-dimensional spaces.

1.11 E 6)

If your dataset contains several noisy examples (or outliers), is it better to increase or decrease k? Explain your reasoning.

1.12 Answer:

It is generally better to increase k:

Increasing k: By increasing the value of k, the k-nearest neighbor algorithm considers more neighbors when making a prediction. This can help to smooth out the impact of noisy examples or outliers, as their influence is diluted by the majority of more representative neighbors. A larger k can lead to more robust and stable predictions by averaging out the noise and reducing the sensitivity to individual noisy data points.

Decreasing k: A smaller k makes the algorithm more sensitive to noise and outliers because it relies more on the immediate, potentially noisy neighbors. With a smaller k, outliers have a greater influence on the prediction, which can lead to less accurate results.

In summary, increasing k can mitigate the effects of noisy examples and outliers by making predictions based on a broader set of neighbors.

1.13 E 7)

Implement the Kronecker Delta function in Python,

$$\delta(i,j) = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

The `assert` statements are here to help you: They will raise an `AssertionError` if your function returns unexpected results based on the test cases.

[12]:
```
# This is an example implementing a Dirac Delta Function

def dirac_delta(x):
    if x > 0.5:
        return 1
    else:
        return 0

assert dirac_delta(1) == 1
assert dirac_delta(2) == 1
assert dirac_delta(-1) == 0
assert dirac_delta(0.5) == 0
```

[13]:
```
def kronecker_delta(i, j):
    return 1 if i == j else 0



# DO NOT EDIT THE LINES BELOW
assert kronecker_delta(1, 0) == 0
assert kronecker_delta(2, 2) == 1
assert kronecker_delta(-1, 1) == 0
assert kronecker_delta(0.5, 0.1) == 0
```

1.14 E 8)

Suppose `y_true` is a list that contains true class labels, and `y_pred` is an array with predicted class labels from some machine learning task. Calculate the prediction accuracy in percent (without using any external libraries).

[15]:
```
y_true = [1, 2, 0, 1, 1, 2, 3, 1, 2, 1]
y_pred = [1, 2, 1, 1, 1, 0, 3, 1, 2, 1]



# Calculate the number of correct predictions
correct_predictions = sum([1 for true, pred in zip(y_true, y_pred) if true ==␣
 ↪pred])

# Calculate accuracy
accuracy = (correct_predictions / len(y_true)) * 100
```

```
print('Accuracy: %.2f%%' % accuracy)
```

```
Accuracy: 80.00%
```

1.15 E 9)

Import the NumPy library to create a 3x3 matrix with values ranging 0-8. The expected output should look as follows:

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

[22]:
```
import numpy as np

matrix = np.arange(9).reshape(3, 3)

matrix
```

[22]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

1.16 E 10)

Use create a 2x2 NumPy array with random values drawn from a standard normal distribution using the random seed `123`:

If you are using the 123 random seed, the expected result should be:

```
array([[-1.0856306 ,  0.99734545],
       [ 0.2829785 , -1.50629471]])
```

[21]:
```
import numpy as np

# Set the random seed
np.random.seed(123)

# Create a 2x2 array with random values from a standard normal distribution
random_array = np.random.randn(2, 2)

random_array
```

[21]:
```
array([[-1.0856306 ,  0.99734545],
       [ 0.2829785 , -1.50629471]])
```

1.17 E 11)

Given an array `A`,

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

use the NumPy slicing syntax to only select the 2x2 center of this matrix, i.e., the subarray

```
array([[ 6,  7],
       [10, 11]]).
```

[20]:
```
import numpy as np
A = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
    [13, 14, 15, 16]])


# Select the 2x2 center of the matrix
center_subarray = A[1:3, 1:3]

center_subarray
```

[20]:
```
array([[ 6,  7],
       [10, 11]])
```

1.18 E 12)

Given the array `A` below, find the most frequent integer in that array:

[23]:
```
import numpy as np
rng = np.random.RandomState(123)
A = rng.randint(0, 10, 200)

# Find the most frequent integer in the array
most_frequent = np.bincount(A).argmax()

print("Most frequent integer:", most_frequent)
```

```
Most frequent integer: 3
```

1.19 E 13)

Complete the line of code below to read in the `'train_data.txt'` dataset, which consists of 3 columns: 2 feature columns and 1 class label column. The columns are separated via white spaces. If your implementation is correct, the last line should show a data array in below the code cell that has the following contents:

```
       x1     x2     y
0   -3.84   -4.40   0
```

```
1    16.36  6.54    1
2   -2.73   -5.13   0
3    4.83    7.22   1
4    3.66   -5.34   0
```

[25]:
```
import pandas as pd

# Read the dataset
df_train = pd.read_csv('train_data.txt', sep='\s+', header=None, names=['x1',␣
 ↪'x2', 'y'])

# Display the first few rows of the data to confirm
df_train.head()
```

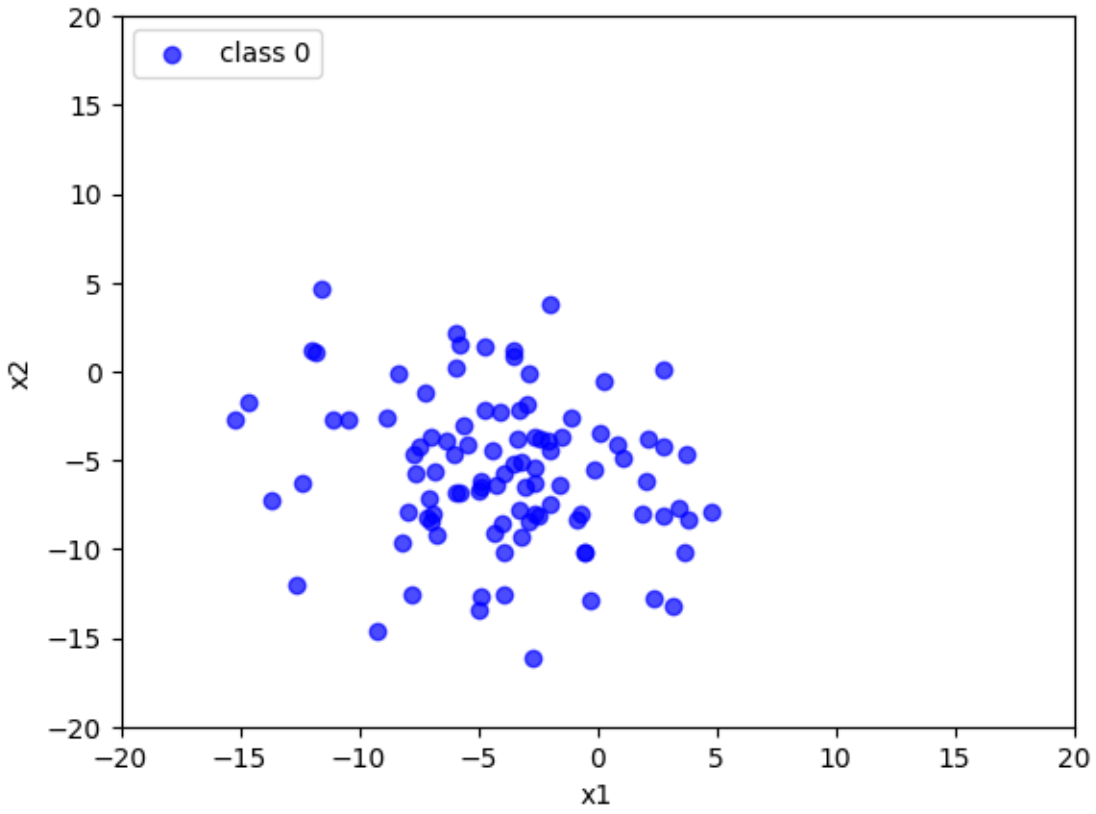
[25]:
```
      x1     x2  y
0     x1     x2  y
1  -3.84  -4.40  0
2  16.36   6.54  1
3  -2.73  -5.13  0
4   4.83   7.22  1
```

1.20 E 14)

Consider the following code below, which plots one the samples from class 0 in a 2D scatterplot using matplotlib:

[29]:
```
X_train = df_train[['x1', 'x2']].values
y_train = df_train['y'].values
```

[44]:
```
import matplotlib.pyplot as plt

X_train_class_0 = X_train[y_train == 0]
plt.scatter(X_train_class_0[:, 0],
            X_train_class_0[:, 1],
            label='class 0',
            color='blue',  # Optional: color for class 0
            alpha=0.7)     # Optional: transparency for better visualization


plt.xlabel('x1')
plt.ylabel('x2')
plt.xlim([-20, 20])
plt.ylim([-20, 20])
plt.legend(loc='upper left')
plt.show()
```



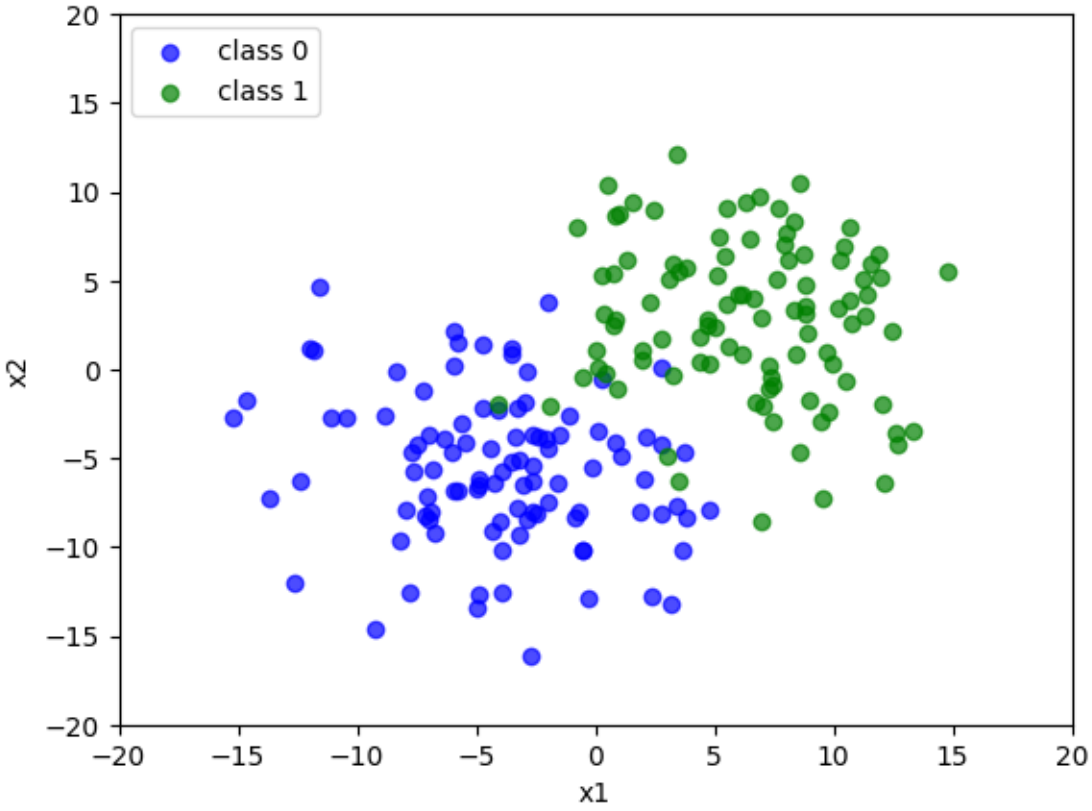
Now, the following code below is identical to the code in the previous code cell but contains partial code to also include the examples from the second class. Complete the second `plt.scatter` function to also plot the trainign examples from `class 1`.

[45]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load data from test_data.txt with different delimiters
# Adjust delimiter based on the actual format of your file
df_train = pd.read_csv('test_data.txt', delimiter=r'\s+', engine='python')

# Display the first few rows of the DataFrame
print("First few rows of df_train:")
print(df_train.head())

# Check column names and data types
print("Column names:", df_train.columns)
print("Data types:", df_train.dtypes)

# Ensure column names are correct and strip any extra spaces
```

```
df_train.columns = df_train.columns.str.strip()

# Verify column names again after stripping spaces
print("Cleaned column names:", df_train.columns)

# Extract features and labels
X_train = df_train[['x1', 'x2']].values
y_train = df_train['y'].values

# Check unique values in y_train and their counts
unique_classes, class_counts = np.unique(y_train, return_counts=True)
print("Class distribution:", dict(zip(unique_classes, class_counts)))

# Filter samples for class 0 and class 1
X_train_class_0 = X_train[y_train == 0]
X_train_class_1 = X_train[y_train == 1]

# Print the number of samples for each class
print("Number of samples for class 0:", len(X_train_class_0))
print("Number of samples for class 1:", len(X_train_class_1))

# Plot examples from class 0
plt.scatter(X_train_class_0[:, 0],
            X_train_class_0[:, 1],
            label='class 0',
            color='blue',  # Optional: color for class 0
            alpha=0.7)     # Optional: transparency for better visualization

# Plot examples from class 1
plt.scatter(X_train_class_1[:, 0],
            X_train_class_1[:, 1],
            label='class 1',
            color='green',  # Optional: color for class 1
            alpha=0.7)     # Optional: transparency for better visualization

plt.xlabel('x1')
plt.ylabel('x2')
plt.xlim([-20, 20])
plt.ylim([-20, 20])
plt.legend(loc='upper left')
plt.show()
```

```
First few rows of df_train:
     x1    x2  y
0 -5.75 -6.83  0
1  5.51  3.67  1
2  5.11  5.32  1
```

```
3  0.85 -4.11  0
4 -0.50 -0.45  1
Column names: Index(['x1', 'x2', 'y'], dtype='object')
Data types: x1    float64
x2    float64
y       int64
dtype: object
Cleaned column names: Index(['x1', 'x2', 'y'], dtype='object')
Class distribution: {0: 100, 1: 100}
Number of samples for class 0: 100
Number of samples for class 1: 100
```



1.21 E 15)

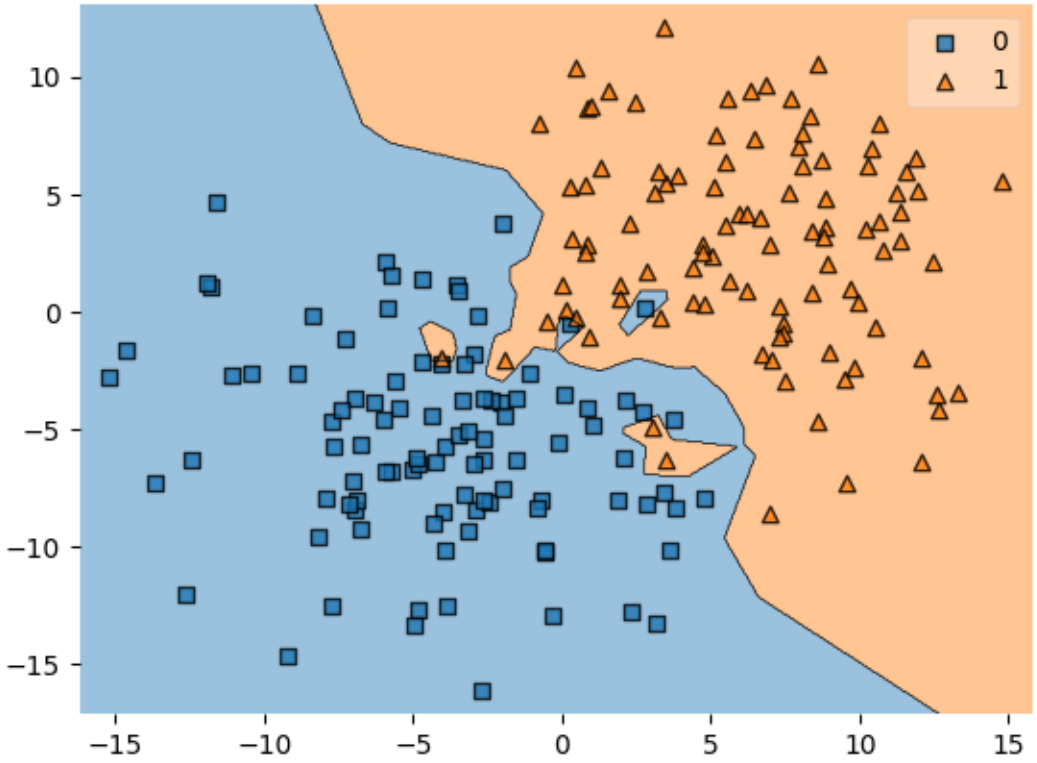
Consider the we trained a 1-nearest neighbor classifier using scikit-learn on the previous training dataset:

[46]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

[46]:
```
KNeighborsClassifier(n_neighbors=1)
```

[48]:
```
from mlxtend.plotting import plot_decision_regions

plot_decision_regions(X_train, y_train, knn)
```

[48]:
```
<Axes: >
```



Compute the misclassification error of the 1-NN classifier on the training set:

[50]:
```
from sklearn.metrics import accuracy_score
# Predict the labels on the training set
y_pred = knn.predict(X_train)
accuracy = accuracy_score(y_train, y_pred)
misclassification_error = 1 - accuracy

print(f"Misclassification Error: {misclassification_error:.4f}")
```

```
Misclassification Error: 0.0000
```

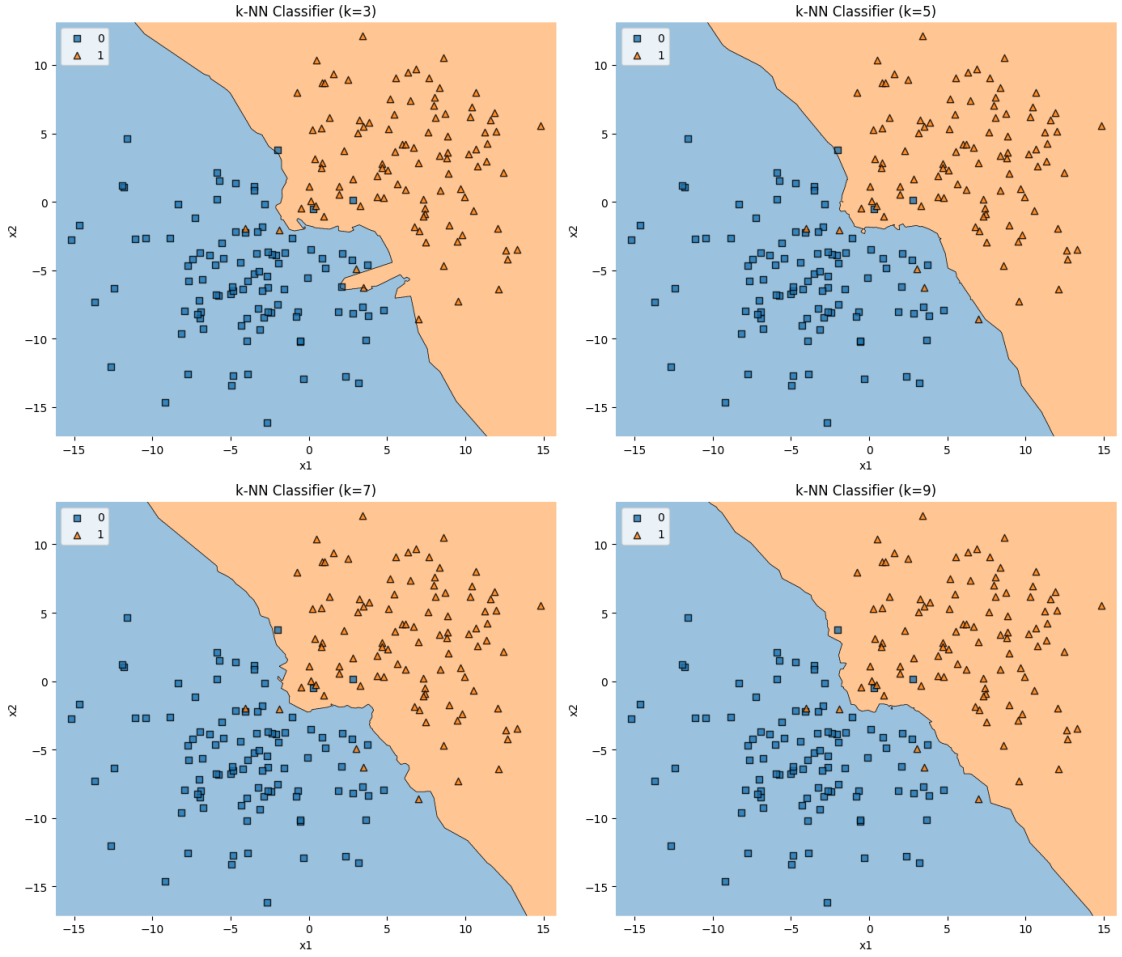
1.22 E 16)

Use the code from E 15) to

- also visualize the decision boundaries of k-nearest neighbor classifiers with k=3, k=5, k=7, k=9
- compute the prediction error on the training set for the k-nearest neighbor classifiers with k=3, k=5, k=7, k=9

[51]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from mlxtend.plotting import plot_decision_regions

# Define the k values to be used
k_values = [3, 5, 7, 9]

# Create a figure to hold the plots
fig, axes = plt.subplots(2, 2, figsize=(14, 12))

# Flatten the axes array for easy iteration
axes = axes.ravel()

# Store errors for each k value
errors = {}

for i, k in enumerate(k_values):
    # Train the k-NN classifier
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Plot decision boundaries
    plot_decision_regions(X_train, y_train, clf=knn, ax=axes[i])
    axes[i].set_title(f'k-NN Classifier (k={k})')
    axes[i].set_xlabel('x1')
    axes[i].set_ylabel('x2')
    axes[i].legend(loc='upper left')

    # Compute prediction error
    y_pred = knn.predict(X_train)
    accuracy = accuracy_score(y_train, y_pred)
    misclassification_error = 1 - accuracy
    errors[k] = misclassification_error

# Show all plots
plt.tight_layout()
plt.show()

# Print prediction errors for each k
for k, error in errors.items():
```

```
    print(f"Misclassification Error for k={k}: {error:.4f}")
```




```
Misclassification Error for k=3: 0.0400
Misclassification Error for k=5: 0.0400
Misclassification Error for k=7: 0.0400
Misclassification Error for k=9: 0.0400
```

1.23 E 17)

Using the same approach you used in E 13), now load the `test_data.txt` file into a pandas array.

[52]:
```
import pandas as pd

# Read the dataset
df_test = pd.read_csv('test_data.txt', sep='\s+', header=None, names=['x1',␣
 ↪'x2', 'y'])

# Display the first few rows of the data to confirm
```

```
df_test.head()
```

[52]:

	x1	x2	y
0	x1	x2	y
1	-5.75	-6.83	0
2	5.51	3.67	1
3	5.11	5.32	1
4	0.85	-4.11	0

Assign the features to `X_test` and the class labels to `y_test` (similar to E 13):

[53]:
```
X_test = df_test[['x1', 'x2']].values
y_test = df_test['y'].values
```

1.24 E 18)

Use the `train_test_split` function from scikit-learn to divide the training dataset further into a training subset and a validation set. The validation set should be 30% of the training dataset size, and the training subset should be 70% of the training dataset size.

For you reference, the `train_test_split` function is documented at http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[55]:
```
from sklearn.model_selection import train_test_split

# Split the dataset into training subset and validation set
X_train_sub, X_val, y_train_sub, y_val = train_test_split(
    X_train,  # Features for training
    y_train,  # Labels for training
    test_size=0.30,  # 30% for validation
    random_state=123,  # For reproducibility
    stratify=y_train  # Maintain class distribution
)
```

1.25 E 19)

Write a for loop to evaluate different k nn models with k=1 to k=14. In particular, fit the `KNeighborsClassifier` on the training subset, then evaluate it on the training subset, validation subset, and test subset. Report the respective classification error or accuracy.

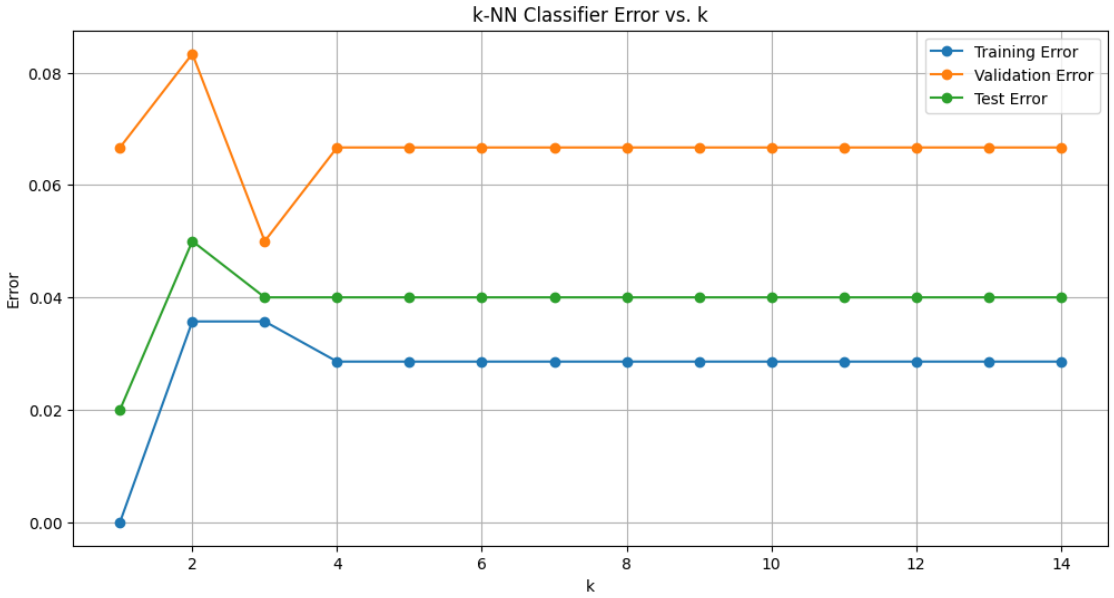
[60]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Read the dataset
df_test = pd.read_csv('test_data.txt', sep='\s+', header=None, names=['x1',␣
 ↪'x2', 'y'])
```

```
# Ensure the columns are of correct type
df_test['x1'] = pd.to_numeric(df_test['x1'], errors='coerce')
df_test['x2'] = pd.to_numeric(df_test['x2'], errors='coerce')
df_test['y'] = pd.to_numeric(df_test['y'], errors='coerce')

# Drop rows with any NaN values
df_test = df_test.dropna()

# Define X and y
X_test = df_test[['x1', 'x2']].values
y_test = df_test['y'].values

# Example of splitting training data
X_train_sub, X_val, y_train_sub, y_val = train_test_split(X_train, y_train,␣
 ↪test_size=0.3, random_state=123, stratify=y_train)

# Initialize dictionaries to store errors and accuracies
train_errors = {}
val_errors = {}
test_errors = {}

# Evaluate k-NN classifiers for k in range 1 to 14
for k in range(1, 15):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_sub, y_train_sub)

    # Predict and compute accuracy for the training subset
    y_train_pred = knn.predict(X_train_sub)
    train_accuracy = accuracy_score(y_train_sub, y_train_pred)
    train_error = 1 - train_accuracy
    train_errors[k] = train_error

    # Predict and compute accuracy for the validation subset
    y_val_pred = knn.predict(X_val)
    val_accuracy = accuracy_score(y_val, y_val_pred)
    val_error = 1 - val_accuracy
    val_errors[k] = val_error

    # Predict and compute accuracy for the test subset
    y_test_pred = knn.predict(X_test)
    test_accuracy = accuracy_score(y_test, y_test_pred)
    test_error = 1 - test_accuracy
    test_errors[k] = test_error

    # Print the results for the current value of k
    print(f'k={k}')
```

```
    print(f'  Training error: {train_error:.4f}')
    print(f'  Validation error: {val_error:.4f}')
    print(f'  Test error: {test_error:.4f}')
    print('---')

# Optionally, you can also plot the errors for a visual representation
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(list(train_errors.keys()), list(train_errors.values()), marker='o',␣
 ↪label='Training Error')
plt.plot(list(val_errors.keys()), list(val_errors.values()), marker='o',␣
 ↪label='Validation Error')
plt.plot(list(test_errors.keys()), list(test_errors.values()), marker='o',␣
 ↪label='Test Error')
plt.xlabel('k')
plt.ylabel('Error')
plt.title('k-NN Classifier Error vs. k')
plt.legend()
plt.grid(True)
plt.show()
```

```
k=1
  Training error: 0.0000
  Validation error: 0.0667
  Test error: 0.0200
---
k=2
  Training error: 0.0357
  Validation error: 0.0833
  Test error: 0.0500
---
k=3
  Training error: 0.0357
  Validation error: 0.0500
  Test error: 0.0400
---
k=4
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=5
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
```

```
k=6
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=7
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=8
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=9
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=10
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=11
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=12
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=13
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
k=14
  Training error: 0.0286
  Validation error: 0.0667
  Test error: 0.0400
---
```



1.26 E 20)

Consider the following code cell, where I implemented k-nearest neighbor classification algorithm following the the scikit-learn API

[62]:
```
import numpy as np


class KNNClassifier(object):
    def __init__(self, k, dist_fn=None):
        self.k = k
        if dist_fn is None:
            self.dist_fn = self._euclidean_dist

    def _euclidean_dist(self, a, b):
        dist = 0.
        for ele_i, ele_j in zip(a, b):
            dist += ((ele_i - ele_j)**2)
        dist = dist**0.5
        return dist

    def _find_nearest(self, x):
        dist_idx_pairs = []
        for j in range(self.dataset_.shape[0]):
            d = self.dist_fn(x, self.dataset_[j])
            dist_idx_pairs.append((d, j))
```

```
        sorted_dist_idx_pairs = sorted(dist_idx_pairs)

        return sorted_dist_idx_pairs

    def fit(self, X, y):
        self.dataset_ = X.copy()
        self.labels_ = y.copy()
        self.possible_labels_ = np.unique(y)

    def predict(self, X):
        predictions = np.zeros(X.shape[0], dtype=int)
        for i in range(X.shape[0]):
            k_nearest = self._find_nearest(X[i])[:self.k]
            indices = [entry[1] for entry in k_nearest]
            k_labels = self.labels_[indices]
            counts = np.bincount(k_labels,
                                 minlength=self.possible_labels_.shape[0])
            pred_label = np.argmax(counts)
            predictions[i] = pred_label
        return predictions
```

[63]:
```
five_test_inputs = X_train[:5]
five_test_labels = y_train[:5]

knn = KNNClassifier(k=1)
knn.fit(five_test_inputs, five_test_labels)
print('True labels:', five_test_labels)
print('Pred labels:', knn.predict(five_test_inputs))
```

```
True labels: [0 1 1 0 1]
Pred labels: [0 1 1 0 1]
```

Since this is a very simple implementation of kNN, it is relatively slow – very slow compared to the scikit-learn implementation which uses data structures such as Ball-tree and KD-tree to find the nearest neighbors more efficiently, as discussed in the lecture.

While we won't implement advanced data structures in this class, there is already an obvious opportunity for improving the computational efficiency by replacing for-loops with vectorized NumPy code (as discussed in the lecture). In particular, consider the `_euclidean_dist` method in the `KNNClassifier` class above. Below, I have written is as a function (as opposed to a method), for simplicity:

[64]:
```
def euclidean_dist(a, b):
    dist = 0.
    for ele_i, ele_j in zip(a, b):
        dist += ((ele_i - ele_j)**2)
    dist = dist**0.5
    return dist
```

Your task is now to benchmark this function using the `%timeit` magic command that we talked about in class using two random vectors, `a` and `b` as function inputs:

[65]:
```
rng = np.random.RandomState(123)

a = rng.rand(100)
b = rng.rand(100)
```

[67]:
```
# Time the Euclidean distance function using timeit
import timeit

# Define a wrapper function for timeit
def wrapper():
    return euclidean_dist(a, b)

# Measure the time
time_taken = timeit.timeit(wrapper, number=1000)
print(f"Time taken for 1000 runs: {time_taken:.4f} seconds")

# Time the Euclidean distance function
%timeit euclidean_dist(a, b)
```

```
Time taken for 1000 runs: 0.0532 seconds
35.7  s ± 3.58  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

1.27 E 21)

Now, rewrite the Euclidean distance function from E 20) in NumPy using - either using the `np.sqrt` and `np.sum` function - or using the `np.linalg.norm` function

and benchmark it again using the `%timeit` magic command. Then, compare results with the results you got in E 22). Did you make the function faster? Yes or No? Explain why, in 1-2 sentences.

[68]:
```
# using np.sqrt and np.sum function
import numpy as np

def euclidean_dist_numpy(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

# Benchmark the function using %timeit in a Jupyter Notebook
import timeit

# Define a wrapper function for timeit
def wrapper_numpy():
    return euclidean_dist_numpy(a, b)

# Measure the time
time_taken_numpy = timeit.timeit(wrapper_numpy, number=1000)
```

```
print(f"Time taken for 1000 runs (NumPy version): {time_taken_numpy:.4f}␣
 ↪seconds")
```

```
Time taken for 1000 runs (NumPy version): 0.0172 seconds
```

[69]:
```
#using np.linalg.norm
import numpy as np

def euclidean_dist_norm(a, b):
    return np.linalg.norm(a - b)

# Benchmark the function using %timeit in a Jupyter Notebook
import timeit

# Define a wrapper function for timeit
def wrapper_norm():
    return euclidean_dist_norm(a, b)

# Measure the time
time_taken_norm = timeit.timeit(wrapper_norm, number=1000)
print(f"Time taken for 1000 runs (Norm version): {time_taken_norm:.4f} seconds")
```

```
Time taken for 1000 runs (Norm version): 0.0055 seconds
```

1.28 E 22)

Another inefficient aspect of the `KNNClassifier` implementation is that it uses the sorted function to sort all values in the distance value array. Since we are only interested in the k nearest neighbors, sorting *all* neighbors is quite unnecessary.

Consider the array `c`:

[70]:
```
rng = np.random.RandomState(123)
c = rng.rand(10000)
```

Call the sorted function to select the 3 smallest values in that array, we can do the following:

[73]:
```
# Define a function to benchmark
def benchmark_sorted():
    return sorted(c)[:3]
```

In the code cell below, use the `%timeit` magic command to benchmark the sorted command above:

[74]:
```
# Use %timeit to measure the execution time
import timeit

# Measure the time for 1000 runs
time_taken_sorted = timeit.timeit(benchmark_sorted, number=1000)
print(f"Time taken for 1000 runs (sorted): {time_taken_sorted:.4f} seconds")
```

```
Time taken for 1000 runs (sorted): 3.7847 seconds
```

A more efficient way to select the k smallest values from an array is to use a priority queue, for example, implemented using a heap data structure. A convenient `nsmallest` function that does exactly that is available from Python's standard library:

[75]:
```
from heapq import nsmallest

nsmallest(3, c)
```

[75]:
```
[6.783831227508141e-05, 8.188761366767494e-05, 0.0001201014889748997]
```

In the code cell below, use the `%timeit` magic command to benchmark the `nsmallest` function:

[76]:
```
# Define a function to benchmark
def benchmark_nsmallest():
    return nsmallest(3, c)

# Use %timeit to measure the execution time
import timeit

# Measure the time for 1000 runs
time_taken_nsmallest = timeit.timeit(benchmark_nsmallest, number=1000)
print(f"Time taken for 1000 runs (nsmallest): {time_taken_nsmallest:.4f}␣
 ↪seconds")
```

```
Time taken for 1000 runs (nsmallest): 0.8140 seconds
```

Summarize your findings in 1-3 sentences.

1.29 Summary:

The nsmallest function is typically more efficient than sorting the entire array and then selecting the smallest k values, especially for large datasets. This is because nsmallest uses a heap data structure, which has better time complexity for finding a small number of smallest elements compared to full sorting. As a result, nsmallest will generally be faster and more suitable for this task than using sorted for large arrays.