knn-iris

September 11, 2024

1 k-Nearest Neighbors (kNN) algorithm

$ $

1.1 Name: ***Write your name here***

1.2 Instructions

In this assignment, you will begin practicing some basic linear algebra concepts through explicit exercises and inherently by implementing the kNN algorithm for classification. Additionally, you will practice loading, visualizing, and splitting data.

Your job is to read through the assignment and fill in any code segments that are marked by `# YOUR CODE HERE` comments. Some segments will have a `todo_check()` function which will give you a rough estimate of whether your code is functioning as excepted.

At any point, if you feel lost concerning how to program a specific part, take some time and visit the official documentation for each library and read about the methods that you need to use.

1.3 Submission

1. Save the notebook.
2. Enter your name in the appropriate markdown cell provided at the top of the notebook.
3. Select `Kernel` -> `Restart Kernel and Run All Cells`. This will restart the kernel and run all cells. Make sure everything runs without errors and double-check the outputs are as you desire!
4. Submit the `.ipynb` notebook on Canvas.

1.4 Utilities

1.4.1 Necessary imports

[1]:
```
from typing import List, Dict, Tuple, Callable
import os
import gc
import traceback
import warnings
from pdb import set_trace

import sklearn
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

1.4.2 Function for auto-grading of submissions

[2]:
```
class TodoCheckFailed(Exception):
    pass

def todo_check(asserts, mute=False, **kwargs):
    locals().update(kwargs)
    failed_err = "You passed {}/{} and FAILED the following code checks:\n{}"
    failed = ""
    n_failed = 0
    for check, (condi, err) in enumerate(asserts):
        exc_failed = False
        if isinstance(condi, str):
            try:
                passed = eval(condi)
            except Exception:
                exc_failed = True
                n_failed += 1
                failed += f"\nCheck [{check+1}]: Failed to execute check␣
↪[{check+1}] due to the following error...\n{traceback.format_exc()}"
        elif isinstance(condi, bool):
            passed = condi
        else:
            raise ValueError("asserts must be a list of strings or bools")

        if not exc_failed and not passed:
            n_failed += 1
            failed += f"\nCheck [{check+1}]: Failed\n\tTip: {err}\n"

    if len(failed) != 0:
        passed = len(asserts) - n_failed
        err = failed_err.format(passed, len(asserts), failed)
        raise TodoCheckFailed(err.format(failed))
    if not mute: print("Your code PASSED all the code checks!")
```

2 NumPy and Linear Algebra Exercises

Below will be several NumPy exercises for testing your NumPy and linear algebra understandings. Refer to the notes for assitance!

Matrix operations (10 points): Write code in NumPy that creates the following three 1D vectors:

- **x** = [1, 2, 3]
- **y** = [-0.5, 1, +0.5]
- **z** = 2 **x** - **y**

Write code that creates a 3x3 matrix A with rows **x**, **y**, and **z** (in this order).

Write code that calculates the rank of A.

```
[ ]: x = # YOUR CODE HERE

print(x)

y = # YOUR CODE HERE

print(y)

z = # YOUR CODE HERE

print(z)

A = # YOUR CODE HERE

print(A)

rank = # YOUR CODE HERE

print(rank)
```

Systems of Linear Equations (10 points) It is time to practice the matrix product and inverse. To do so, solve the linear system of equations $A \cdot x = b$ by solving for x such that $x = A^{-1} \cdot b$ where $\cdot$ denotes the matrix or dot product.

$$4x_1 + 3x_2 - 5x_3 = 2 \quad (1)$$

$$-2x_1 - 4x_2 + 5x_3 = 5 \quad (2)$$

$$8x_1 + 8x_2 + 1x_3 = -3 \quad (3)$$

1. Convert the left-hand side of the linear system of equations in a matrix. Store the matrix into the variable `A`.
2. Convert the right-hand side of the linear system of equations into a 1D vector array. Store the matrix into the variable `b`
3. Solve for x by converting the equation $x = A^{-1} \cdot b$ into code. Store the output into the variable `x`.

```
[ ]: A = # YOUR CODE HERE

print(A)

b = # YOUR CODE HERE
```

```
print(b)

x = # YOUR CODE HERE

print(f"x: {x}")
```

[ ]:
```
todo_check([
    ("A.sum() == 18", "A potentially has incorrect values."),
    ("b.sum() == 4", "b potentially has incorrect values."),
    ("isinstance(x, np.ndarray)", "x should be a NumPy array (type np.ndarray).
 ↪"),
    ("np.isclose(x.flatten(), np.array([2.215, -2.569, -0.169]), rtol=0.01).
 ↪all()", "x has potentially incorrect values."),
])
```

3 K-Nearest Neighbors(kNN)

3.1 Iris Dataset

This assignment will have you tackle the Iris flower classification problem, where the goal is to classify three different species of Iris flowers. The Iris flower dataset is a frequently used dataset from the UCI Machine Learning Repository. Kaggle's description of the dataset is as follows:

> The Iris flower data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper. The use of multiple measurements in taxonomic problems. It is sometimes called Anderson's Iris dataset because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of 3 related species. The dataset consists of 50 samples from each of three species of Iris (Iris Setosa, Iris virginica, and Iris versicolor). 4 features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

Thus, the Iris flower classification problem is a 3-way multi-classification problem where you must classify each data sample either as an *Iris setosa*, *Iris virginica*, or *Iris versicolor*.

3.1.1 Data Loading with *sklearn*

Instead of downloading the data directly, we will have you try using sklearn's datasets which can be directly downloaded and accessed via `sklearn`'s API. You can see the relevant load API documentation here.

[3]:
```
# Import relevant load function.
from sklearn.datasets import load_iris

# Load the dataset and indicate that data should be returned as a Pandas␣
 ↪DataFrame.
iris = load_iris(as_frame = True)
```

3.1.2 Inspecting the data types, features, and labels

Below, we will walk you through the `sklearn.utils.Bunch` data type used by `sklearn` to represent the dataset, as you will need to understand it if you want to properly access the Iris data.

[10]:
```
type(iris)
```

[10]:
```
sklearn.utils._bunch.Bunch
```

The first thing you should always do when you do not know what a variable's type represents is to check the documentation and printout the variable itself.

After looking at the docs, you should have read that a `Bunch` is a "container object exposing keys as attributes." Looking at the example code in the `Bunch` docs, you should see that it acts similar to a Python dictionary but with some added functionality.

Below, you can see what happens when we printout the `iris` variable. It outputs a dictionary with keys and values where some of the keys seem to correspond to the Iris data while others seem to correspond to meta-data (information about our data).

[ ]:
```
iris
```

So far, it looks like the `iris` variable is an instance of the Sklearn `Bunch` class and within said class instance is some data and meta-data relating to the Iris dataset. But how do can you access the actual data?

After reading the `Bunch` docs and the `load_iris()` function docs, it looks like each key seen in the above output actually corresponds to a class attributes. This means you can access the data by accessing the class attribute inside the `iris` instance.

Before you do that, we first printout all the keys by calling `iris.keys()` method.

[12]:
```
iris.keys()
```

[12]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename', 'data_module'])
```

Okay, now we can clearly see the keys. It is time to try to access the data and meta-data now. As mentioned, we can access the 'data' key of `iris` by treating it as a class attribute.

[13]:
```
iris.data
```

[13]:
```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 …                 …                  …                 …
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
```

```
148               6.2               3.4               5.4               2.3
149               5.9               3.0               5.1               1.8

[150 rows x 4 columns]
```

You can now access the Iris data! Recall, you should have passed a parameter to the load function that returns the data as a Pandas DataFrame. You can check this is true by checking the type of `iris.data`.

[14]:
```
type(iris.data)
```

[14]:
```
pandas.core.frame.DataFrame
```

Next, you can check the shape of the data which will tell you the number of data samples (rows) and number of features (columns).

[15]:
```
iris.data.shape
```

[15]:
```
(150, 4)
```

You should see that there are 150 data samples (rows) and 4 features (columns)

Looking at the `iris.data` DataFrame, you should see 4 features: sepal length, sepal width, petal length, and petal width that are all in centimeters (cm). What do these features even refer to?

Well, we can see exactly what each feature corresponds to by looking at the below image! Notice, each iris has a sepal and a petal. Thus, these features correspond to the length and width of the sepal and petal.

Next, let's look at the labels for each class. We can do so by accessing the `target` class attribute.

[16]:
```
iris.target
```

[16]:
```
0      0
1      0
2      0
3      0
4      0
      ..
145    2
146    2
147    2
148    2
149    2
Name: target, Length: 150, dtype: int64
```

Once again, the type of the targets should be a Pandas Series object. This is confirmed by checking the type.

[17]:
```
type(iris.target)
```

[17]: pandas.core.series.Series

By using NumPy's `np.unique` function and passing `return_counts = True`, we can see that there are 3 class labels (as expected) and each class has 50 data samples each ($3 \times 50 = 150$). This means the data is perfectly balanced.

[18]:
```
np.unique(iris.target, return_counts = True)
```

[18]: (array([0, 1, 2]), array([50, 50, 50]))

The last thing to do is check which numerical label (index) corresponds to which Iris type (string). This can be done by looking at `target_names`.

[19]:
```
iris.target_names
```

[19]: array(['setosa', 'versicolor', 'virginica'], dtype='<U10')

Here we can see that label 0 corresponds to 'setosa', label 1 corresponds to 'versicolor', and label 2 corresponds to 'virginica' as the class labels in the array `iris.targets` correspond to the indexes of the array `iris.target_names`.

As such, this dataset rerpesents each sample using 4 features 'sepal length', 'sepal width', 'petal length', and 'petal width', and maps it into one of the three classes 'setosa', 'versicolor', 'virginica'.

3.1.3 Visualization

In this section we visualize the data using a scatter matrix across features that will plot each feature values against another feature values, coloer coded by class. This will also plot the distribution of each feature per class along the main diagonal. We will use the Seaborn module to create the scatter matrix.

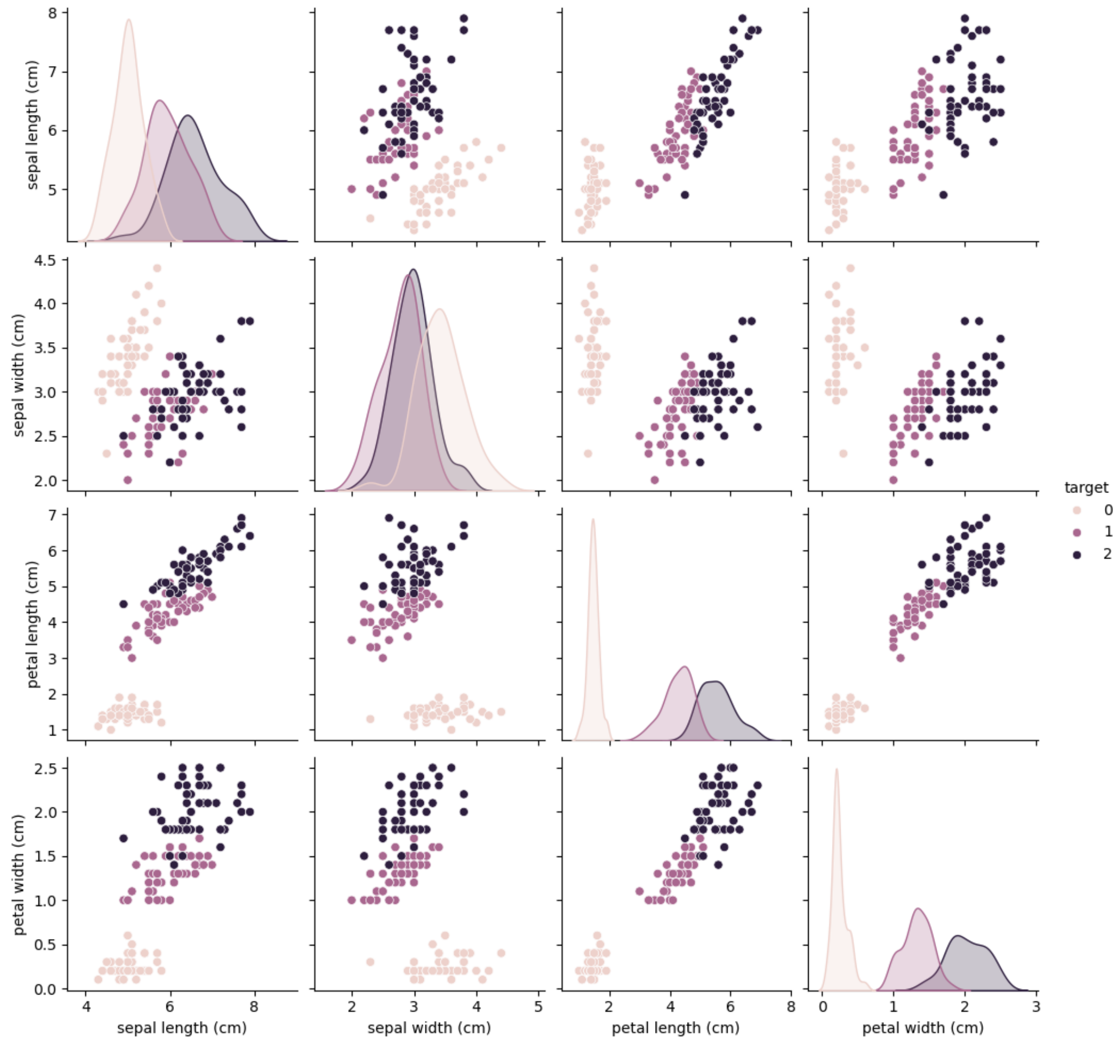
Plot Scatter Matrix We will create the scatter matrix using the followign sequence of steps:

1. Import the `pairplot` function from seaborn. This function will create a plot similar to a scatter matrix. Refer to the documentation for more details.
2. Concatenate the iris data and iris targets into one DataFrame. Store the output into `iris_df`. For this, we use the `pd.concat()` function and and stack horizontally (`axis = 1`).
3. Call the `pairplot()` function and pass the `iris_df` and `hue = target`. Setting the hue to be the target column will color code the data samples based on their class label.

[20]:
```
from seaborn import pairplot

iris_df = pd.concat([iris.data, iris.target], axis = 1)

pairplot(iris_df, hue = 'target');
```



Analysis of scatter plot (10 points) In the textbox below, describe the insights that you get from looking at the scatter plots. Which features are more discriminative? What features appear to be more correlated with each other?

`Answer:`

3.1.4 Splitting data into train and test subsets (10 points)

Before you can train and evalaute any ML algorithm, you first need to split your data into, at the very least, a train and test set. In the future, you will be tasked with splitting data into a train, validation, and test set. For now, we will keep things simple and only use a train and test set, even if it might not be best practice.

Complete the `train_test_split()` function by following steps below. We have provided you with code for randomizing the data using `rng.permutation` to generate randomly shuffled indices in `random_perm`. We then index `X` and `y` both by `random_perm` to shuffle the data while keeping each

label correctly matched with its corresponding input features in `X`.

1. Compute the index at which the data should be split at by multiplying the number of data samples in `X` by the proportion of the training dataset size given by `train_size`. Cast the output as an `int` and then store it in the variable `split_idx`.
2. Set the training data `X_trn` to contain all the data in `X` **up to** `split_idx`.
3. Set the training labels `y_trn` to contain all the labels in `y` **up to** `split_idx`.
4. Set the test data `X_tst` to contain all the data in `X` **from** `split_idx` **to the end**.
5. Set the testing labels `y_tst` to contain all the labels in `y` **from** `split_idx` **to the end**.

[21]:
```
def train_test_split(
    X: np.ndarray,
    y: np.ndarray,
    train_size: float,
    random_seed: int):
    """ Randomizes and then splits the data into train and test sets.

        Args:
            X: Data given as a 2D matrix

            y: Labels given as a vector

            train_size: A number between 0.0 and 1.0 and represent the
                proportion of the dataset to include in the train split.

            random_seed: The seed which controls the shuffling applied
                to the data before being split.
    """

    # Shuffle data using random state seed.
    rng = np.random.RandomState(random_seed)
    random_perm = rng.permutation(len(X))
    X = X[random_perm]
    y = y[random_perm]

    split_idx = # YOUR CODE HERE

    X_trn, X_tst, y_trn, y_tst = # YOUR CODE HERE

    return X_trn, X_tst, y_trn, y_tst
```

[ ]:
```
X_trn, X_tst, y_trn, y_tst =
    train_test_split(iris.data.values, iris.target.values,
                     train_size = 0.8, random_seed = 42)

print(f"X_trn shape: {X_trn.shape}")
print(f"X_tst shape: {X_tst.shape}")
print(f"y_trn shape: {y_trn.shape}")
```

```
print(f"y_tst shape: {y_tst.shape}")

todo_check([
    ("np.isclose(X_trn[10], np.array([6.5, 3.2, 5.1, 2. ])).all()", "X_trn␣
↪value was not correct, check implementation and random_state."),
    ("np.isclose(y_trn[10], np.array(2)).all()", "y_trn value was not correct,␣
↪check implementation and random_state."),
    ("np.isclose(X_tst[15], np.array([6.9, 3.1, 4.9, 1.5])).all()", "X_tst␣
↪value was not correct, check implementation and random_state."),
    ("np.isclose(y_tst[15], np.array(1)).all()", "y_tst value was not correct,␣
↪check implementation and random_state."),
])
```

3.2 Euclidean distance (10 points)

Complete the `euclidean_distance()` function below, which takes as input a vector $\mathbf{x}$ and a matrix Y and returns a vector containing the Euclidean the distance between $\mathbf{x}$ and each of the row vectors in Y.

Remeber that the Euclidean distance between two vectors $\mathbf{x}$ and $\mathbf{y}$ is the Euclidean norm (L2 norm) of their difference as shown below, where D is the number of elements in each vector:

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{(\mathbf{x}_1 - \mathbf{y}_1)^2 + (\mathbf{x}_2 - \mathbf{y}_2)^2 + ... + (\mathbf{x}_D - \mathbf{y}_D)^2}$$

Pay attention to the inline documentation regarding the acceptable types of arrays for `x` and `Y`. The function should be implemented without using loops by harnessing NumPys broadcasting abilities to subtract a vector from a matrix.

[23]:
```
def euclidean_distance(x: np.ndarray, Y: np.ndarray) -> np.ndarray:
    """ Compute the euclidean distance between a vector and a matrix.
        Args:
            x: The 1st NumPy array given as a 1D vector or a 2D row vector.

            Y: The 2nd NumPy array given as a 2D matrix.

        Return:
            A 1D vector of floats representing the distance between x and each␣
↪of the rows in Y.
    """
    assert len(Y.shape) == 2, f"y is a 1D vector, expected 2D row vector or␣
↪matrix"

    # YOUR CODE HERE
```

```
    return None
```

```
[ ]: def TEST_euclidean_distance():
    x = np.array([[1, 2]])
    Y = np.array([[7, 8], [9, 10]])
    print(f"x: {x}\nx shape: {x.shape}\nY: {Y}\nY shape: {Y.shape}")
    dist1 = euclidean_distance(x, Y)
    print(f"dist1 for X1 and X2: {dist1}")
    print("-"*50)
    x = np.array([[7, 8]])
    Y = np.array([[9, 10]])
    print(f"x: {x}\nx shape: {x.shape}\nY: {Y}\nY shape: {Y.shape}")
    dist2 = euclidean_distance(x, Y)
    print(f"dist2 for X1 and X2: {dist2}")
    todo_check([
        ("np.isclose(dist1, np.array([ 8.485, 11.313 ]), rtol=0.01).all()",␣
 ↪"dist1 euclidean_distance values are potentially incorrect."),
        ("np.isclose(dist2, 2.828, rtol=0.01).all()", "dist2 euclidean_distance␣
 ↪values are potentially incorrect.")
    ], **locals())
TEST_euclidean_distance()
```

3.3 Accuracy metric (10 points)

Finally, we will need a metric for evaluating the performance of the kNN model. One simple and easy metric is accuracy, defined as the percentage of samples that are correctly classified. That is, when there are 1000 labels to classify, if 950 are correctly predicted, then the model has achieved 95% accuracy. Below is the equation for computing accuracy.

$$\text{Accuracy} = \frac{\text{The number of correctly classified samples}}{\text{The total number of samples}}$$

Complete the `accuracy()` function below, which takes as input the vector $\mathbf{y}$ of ground truth labels and the vector $\mathbf{y}_hat$ of predicted labels.

Hint: See what `y == y_hat` calculates, and how to use that to compute accuracy in one line of code.

```
[ ]: def accuracy(y: np.ndarray, y_hat: np.ndarray) -> float:
    """ Computes the accuracy between two 1D vectors

        Args:
            y: Ground truth labels given as a 1D vector

            y_hat: Predicted labels given as a 1D vector

        Return:
            A float corresponding to the accuracy
```

```
    """
    y =  y.reshape(-1,) # Reshape to ensure 1D vector.
    y_hat = y_hat.reshape(-1,) # Reshape to ensure 1D vector.

    acc = # YOUR CODE HERE

    print(f"Accuracy is: {acc}.")

    return acc
```

```
[ ]: def TEST_accuracy():
         dummy_y = np.ones([100, 1])
         dummy_y_hat = np.ones([100, 1])
         dummy_y_hat[90:] = -1

         dummy_acc = accuracy(dummy_y_hat, dummy_y)
         print(f"Accuracy is: {dummy_acc}")

         todo_check([
             ("dummy_acc == .9", "Incorrect accuracy for dummy_acc"),

         ], **locals())

     TEST_accuracy()
```

3.4 Implementing the kNN algorithm (20 points)

Complete the class `KNearestNeighbors` by writing code for hte methods below.

`fit(X, y)`

Memorize the training data by storing it into class attributes as follows:

- Store the passed input features `X` into the class variable `self.X`.
- Store the passed input targets/labels `y` into the class variable `self.y`

`predict(X)`

Follow the following pseudocode to implement the `predict()` method. If done efficiently, it should only take 7-9 lines of code.

- Loop over the test samples stored in the rows of input data `X` and compute the nearest neighbor for each sample.
  - For each test sample, compute the distance to ALL saved training samples using the distance function given by `self.distance_measure`.
  - Sort the distances and get the *indices* of the training sample corresponding to the top `self.k` smallest distances. The `np.argsort()` function can be used here.
  - Get the labels of the top k nearest neighbors obtained above. For each label, count the number of times it appears among these `k` labels. The `np.bincount()` function can be used here.

  - Use the label with the highest count as the prediction for this test sample, e.g. append this label to the `y_hats` list. Theuse `np.argmax()` function can be used here.
- Once the loop is finished, convert the list `y_hat` into 1D vector array and return it.

[27]:
```
class KNearestNeighbors():
    """
        Attributes:
            k: Number of nearest neighbors.

            distance_measure: A python function reference which will compute a␣
 ↪valid distance measure.

            X: The training samples (their feature vectors).

            y: The training targets/labels.
    """
    def __init__(self, k: int, distance_measure: Callable):
        """
            Args:
                k: Number of nearest neighbors.

                distance_measure: A python function that computes a distance␣
 ↪measure
        """
        self.k = k
        self.distance_measure = distance_measure
        self.X: np.ndarray = None
        self.y: np.ndarray = None

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """ Memorizes the training examples.

            Args:
                X: Training samples (feature vectors) given as a 2D array.

                y: Training labels given as a 1D array.
        """
        # YOUR CODE HERE




    def predict(self, X: np.ndarray) -> np.ndarray:
        """ Performs kNN classification using stored training examples.

            Args:
                X: Testing samples (feature vectors) given as a 2D array.
```

```
        Return:
            Returns a 1D array of predictions for the test samples in X.
    """
    y_hats: list = []

    # YOUR CODE HERE









    return y_hats
```

3.4.1 Evaluate the kNN implementation on the Iris dataset

We will use the train vs. test split created above, and the Eucildean distance defined earlier.

```
[ ]: # Create kNN object.
knn = KNearestNeighbors(k = 3, distance_measure = euclidean_distance)

# Train kNN, which simply stores the training examples.
knn.fit(X_trn, y_trn)

# Compute predictions on test examples.
y_hat = knn.predict(X_tst)

print(f"Predictions: {y_hat}")

# Compute test accuracy
test_acc = accuracy(y_tst, y_hat)
print(f"Test accuracy: {test_acc:.2f}")
```

Note, the below TODO check will ONLY work if `random_state` for the `train_test_split()` function was set to 42, `k` is set to 3 and the distance measure is set to `euclidean_distance`!

```
[ ]: todo_check([
    ("np.isclose(knn.X[10], np.array([6.5, 3.2, 5.1, 2. ])).all()", "knn.X␣
 ↪value was not correct, check implementation and seed."),
    ("np.isclose(knn.y[10], np.array(2)).all()", "knn.y value was not correct,␣
 ↪check implementation and seed."),
    ("isinstance(y_hat, np.ndarray)", "y_hat must be an np.ndarray."),
```

```
    ("y_hat.shape == (30,)", "y_hat must have the shape (30,). Make sure you␣
↪stacked y_hat correctly."),
    ("np.isclose(test_acc, 0.969, rtol=0.1)", "test_acc was not correct, check␣
↪implementation and seed.")
])
```

kNN hyper-parameters (20 points)

1. Copy the above code for running the KNN and evaluating the test accuracy to the below code cell, but this time experiment with different values for `k` (e.g., 1, 25, 50, 100).
2. Write your analysis about what happens to the accuracy as you increase the value of `k`. Why do you think this happens? Create a graph where you plot the kNN accuracy on tranining data and the kNN accuracy on test data (vertical axs) vs. different values of `k` (horizontal axis).

[ ]:
```
# YOUR CODE HERE
```

`Analysis:`

3.5 Bonus points

Any nontrivial task / analysis / viosualization that is relevant to this assignment on kNNs is a candidate for bonus points.

[ ]:
```

```