

# Assignment 1

- **Weight:** 4%

## What you'll practice

- Implementing a **DataTable** class to load and query CSV data.
- Designing a tiny OOP stack with an abstract Predictor and concrete subclasses.
- Raising a custom exception for out-of-range inputs.
- Producing predictions and comparing them to ground truth.

## Problem overview

A common Data Science task is to use historical data to make predictions about new cases. You will use the Wisconsin Breast Cancer Dataset (WBCD) to build a simple class that can predict whether a tumor is benign (0) or malignant (1) using 9 numeric features that describe cell nuclei.

More details about the WBCD dataset can be found here:

<https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original>

You are given three CSVs:

- `data/cancer_historical.csv` — 599 rows; columns:  
`Feature1, Feature2, ..., Feature9, TumorType`  
Each row of the table represents one tumor that was examined by doctors. The first 9 columns contain measurements of cell nuclei in the tumor. The final `TumorType` column shows whether the tumor was benign (0) or malignant (1).
- `data/cancer_new_cases.csv` — 100 rows; columns:  
`Feature1, Feature2, ..., Feature9`  
(No `TumorType`; you will predict this.)  
This file contains data on 100 additional tumors, but doesn't include the true diagnosis of malignant/benign.
- `data/cancer_new_cases_labels.csv` — 100 rows; a single column:  
`TumorType`

True outcomes for the 100 cases in `cancer_new_cases.csv`, in the same row order.

You will use `cancer_historical.csv` to build a class capable of predicting whether a new case is malignant/benign. You will use an object of this class to make predictions for the 100 cases in `cancer_new_cases.csv`, and then compare the predictions to the true outcomes in `cancer_new_cases_labels.csv` to evaluate their accuracy.

All feature values are numeric and already scaled to `[0, 1]`.

## Task:

### 1) `DataTable` class (`data_representation.py`)

This class represents a table of data.

- Store table data in any representation you like (e.g., list of dicts).
- Convert feature values (`Feature1...Feature9`) to **float**
- Convert `TumorType` to **int**

Provide at least the following methods:

- `__init__(csv_path)`: load the CSV into memory.
- `get(row_idx, col_name)` → value
- `get_column(col_name)` → list
- `get_row(row_idx)` → dict {col: value}
- `get_column_names()` → list of str
- `get_column_average(col_name)` → float
- `get_column_min(col_name), get_column_max(col_name)` → values

### 2) Predictors (`predictors.py`)

Create an abstract base class and two concrete predictors. These two concrete predictor classes will make malignant/benign predictions based on the historical data in `cancer_historical.csv`. But they will do so using two different techniques.

- abstract class `Predictor`:
  - `__init__(reference_data: DataTable)` — store historical data
  - `predict(new_case: dict) -> int` — abstract method (return 0 or 1)
- class `MajorityClassPredictor(Predictor)`:

- Always predicts the **most frequent** `TumorType` in the historical data.
  - Return **0** or **1**.
- `class NearestNeighborPredictor(Predictor):`
  - For each new case, compute Euclidean distance over the **9 features** to **every** historical row:
  - Predict the `TumorType` of the **closest** historical row (copy its label).
  - Do **not** use the label in the distance.

$$\text{Euclidean distance } (x, h) = \sqrt{\sum_{i=1}^9 (x_i - h_i)^2}$$

- $x = (x_1, x_2, \dots, x_9)$  are the **features of the new case**.
- $h = (h_1, h_2, \dots, h_9)$  are the **features of a historical case**.
- The summation goes from  $i = 1$  to  $9$ , since you have **9 features**.

### 3) Custom exception (`exceptions.py`)

- Define a custom exception class:

```
class OutOfSampleError(Exception):
    pass
```

- Before making a prediction, check each feature of the new case against the historical dataset.
  - For every feature (`Feature1...Feature9`), compute the `[min, max]` range from the **historical data only**.
  - If a new case has any feature value **less than the min** or **greater than the max**, immediately raise `OutOfSampleError`.

This ensures your program does not attempt to make predictions on cases that are outside the range of values observed in the training data.

#### Example

Suppose in the historical data:

- `Feature4` ranges from `0.1` to `0.9`

If a new case has `Feature4 = 1.0`, then raise `OutOfSampleError`

A meaningful error message should be displayed in the terminal. Eg:

```
OutOfSampleError: Feature4 with value 1.0 is outside historical range [0.1, 0.9]
```

#### 4) Main program (`main.py`)

- Load `cancer_historical.csv` and `cancer_new_cases.csv` using `DataTable`.
- Instantiate `MajorityClassPredictor` and `NearestNeighborPredictor` objects.
- For each new case:
  - Check ranges; if out-of-range, raise `OutOfSampleError` (program exits with an error).
  - Otherwise, print both predictions (format below).
- **Evaluation step:** load `cancer_new_cases_labels.csv` and compute accuracy for each predictor. Because there are 100 new cases:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{100}$$

#### Suggested print format

```
Case 0: Majority=0, NN=1
Case 1: Majority=0, NN=0
...
Majority: accuracy=0.72
NearestNeighbor: accuracy=0.86
```

## Code requirements

- **No external libraries** (no `pandas`, `numpy`, `scikit-learn`, etc.).
- Allowed: `math.sqrt` and your own modules.
- Use the **9 features only** for distances (never the label).
- Use OOP: keep logic in classes/methods; keep `main.py` small.
- Implement and enforce `OutOfSampleError` exactly as specified.

#### Starter code

- `data_representation.py` - write the `DataTable` class in this file
- `predictors.py` - write the `MajorityClassPredictor`, and `NearestNeighborPredictor` classes in this file
- `exceptions.py` - write the `OutOfSampleError` class in this file

- `main.py` - write your main program in this file. Note that most of the logic for this assignment will be encapsulated in your classes, making your main program quite short.

## Hints

This is a large program, so take advantage of the modularity that classes provide: tackle the coding one class, one method at a time, testing as you go. Then use those classes to build your main program.

Write the `DataTable` class first, then the `Predictor` abstract class, then the others.

## Grading

Item	Points
Classes written & used appropriately	50
Functionality (requirements met)	40
Code style and readability	10
<b>Total</b>	<b>100</b>

## Submission

- Zip your entire project folder.
- Use the naming format: `lastname-firstname-Assignment-1.zip`
- Upload the zip file to the **Assignment 1** folder in D2L.