1. Design Pattern & Architecture

This project employs a **Modular, Multi-Layered Architecture** centered around a **Centralized Configuration** pattern. This design promotes a strong separation of concerns, making the code clean, reusable, and easy to maintain.

The architecture is divided into four distinct layers:

1. **Interface Layer (`knn_runner.py`):** The single entry point for the user. Its only job is to parse command-line arguments and delegate tasks to the layer below.
2. **Orchestration Layer (`knn_implement/`):** The "control center" of the project. It manages the entire experimental workflow, from preprocessing data to running tests and generating reports. It knows *what* to do but relies on the Core Logic layer to know *how* to do it.
3. **Core Logic / Algorithm Layer (`knn_scratch/`):** Contains the fundamental, reusable, from-scratch implementations of the algorithms (KNN, distance metrics, cross-validation). This layer has no knowledge of the specific datasets or the overall experiment.
4. **Configuration Layer (`config.py`):** Acts as the "brain" or "settings file" for the project. It provides all the necessary parameters and directs the Orchestration Layer, telling it which functions and paths to use for each dataset.

2. Directory Structure

The project is organized into the following key directories:

- **`Assignment 1 Python Workspace/`**: The root directory of the project.
    - **`knn_scratch/`**: Contains all the from-scratch algorithm implementations. These are generic and could be reused in other projects.
    - **`knn_implement/`**: Contains the scripts that control the specific workflow for this assignment (preprocessing, running experiments).
    - **`Datasets/`**: Stores the raw, unmodified dataset files.

    - **`Data Preprocessing/`**: Stores the output of the `prep` command—the cleaned and numerically encoded CSV files.
    - **`cross_validation_report/`**: Stores the first output of the `run` command—the `.txt` reports comparing fold-by-fold accuracies.
    - **`evaluation_test_report/`**: Stores the second output of the `run` command—the `.txt` reports with the hypothesis test results.

3. Module & Component Breakdown

This project is primarily function-based, not class-based. Below is a description of each module and its key functions.

`knn_scratch` Module (Core Logic)

- **`distance_metrics.py`**
    - **Purpose**: Provides standalone functions for calculating the distance between two data points.
    - **Functions**:
        - `euclidean_distance(row1, row2)`: Calculates the straight-line distance, ideal for continuous numerical data.
        - `manhattan_distance(row1, row2)`: Calculates the "city block" distance, useful for ordinal data.
        - `hamming_distance(row1, row2)`: Counts the number of differing features, ideal for nominal categorical data.
- **`knn.py`**
    - **Purpose**: Contains the core from-scratch implementation of the k-Nearest Neighbors algorithm.
    - **Functions**:
        - `get_neighbors(train_dataset, test_row, k, distance_metric)`: Takes a training set and a single test row, and uses the provided `distance_metric` function to find and return the `k` closest training rows.

        - `predict_classification(train_dataset, test_row, k, distance_metric)`: Orchestrates the prediction for a single test row by first calling `get_neighbors`, then performing a majority vote on the resulting neighbors' class labels to determine the final prediction.

- **`cross_validation.py`**
    - **Purpose**: Provides from-scratch functions to perform and evaluate k-fold cross-validation.
    - **Functions**:
        - `cross_validation_split(dataset, n_folds)`: Randomly splits a dataset into a specified number of folds.
        - `accuracy_metric(actual, predicted)`: Calculates the percentage of correct predictions.
        - `evaluate_algorithm(...)`: A standalone function (used for testing this module) that orchestrates a full cross-validation run for a given algorithm.

`knn_implement` Module (Orchestration)

- **`config.py`**
    - **Purpose**: Acts as a central, static configuration hub for the entire project.
    - **Attributes**:
        - `CONFIG` (dict): A dictionary where each key is a dataset name. Each value is another dictionary containing all parameters for that dataset, including display names, file paths, the chosen `k` value, a direct reference to the appropriate distance metric function, and a direct reference to the appropriate preprocessing function.
- **`dataset_preprocessor.py`**
    - **Purpose**: Contains specialized functions to clean, encode, and transform each of the three raw datasets into a format suitable for KNN.
    - **Functions**:
        - `preprocess_hayes_roth(...)`: Imputes missing values and removes the ID column.

        - `preprocess_car_evaluation(...)`: Applies ordinal encoding to all features.
        - `preprocess_breast_cancer(...)`: Imputes missing values and applies one-hot encoding to all features.
- **`knn_controller.py`**
    - **Purpose**: The primary engine for the experiment. It contains the logic to execute the full 10-fold cross-validation comparison.
    - **Functions**:
        - `run_experiment(config, ...)`: This is the main orchestrator. It loads data, calls `cross_validation_split`, loops through the folds to train and test both the from-scratch KNN and the Scikit-learn KNN, performs a paired t-test on the results, and writes the final report files.

Interface Layer

- **`knn_runner.py`**
    - **Purpose**: The command-line interface (CLI) for the project. It is the only script the user needs to interact with directly.
    - **Functions**:
        - `main()`: Uses Python's `argparse` library to define and parse the `prep` and `run` commands. Based on the user's input, it loads the appropriate configuration from `config.py` and calls the relevant function (either a preprocessor function or `run_experiment`).

4. Program Workflow & Interaction

The modules interact in a clear, top-down flow initiated by the user.

Workflow for the `prep` command:

1. The user executes `python knn_runner.py prep <dataset_name>` in the terminal.
2. `knn_runner.py` parses the arguments and loads the specific `config` dictionary for that dataset from `CONFIG` in `config.py`.

3. `knn_runner.py` retrieves the preprocessing function reference (e.g., `preprocess_car_evaluation`) from the loaded `config` object.
4. It then calls that function, passing it the required input and output paths from the `config` object.
5. The specified function inside `dataset_preprocessor.py` runs, processing the raw data and saving the output CSV files.

Workflow for the `run` command:

1. The user executes `python knn_runner.py run <dataset_name>` in the terminal.
2. `knn_runner.py` parses the arguments and loads the appropriate `config` from `config.py`.
3. `knn_runner.py` calls the `run_experiment` function from `knn_controller.py`, passing the loaded `config` object to it.
4. `run_experiment` then takes over and orchestrates the entire experiment: a. It loads the preprocessed data specified in the `config`. b. It calls `cross_validation_split` from `cross_validation.py` to get the 10 data folds. c. It iterates through each fold, treating one as the test set and the rest as the training set. d. Inside the loop, for the from-scratch model, it calls `predict_classification` from `knn.py` for each row in the test set. e. `predict_classification` in turn calls the specific `distance_metric` function (e.g., `manhattan_distance`) referenced in the `config` and retrieved from `distance_metrics.py`. f. It runs the Scikit-learn model on the same folds for a fair comparison. g. After the loop, it performs a paired t-test on the two lists of accuracy scores. h. Finally, it formats and saves the results into the two `.txt` report files.