Data and Artificial Intelligence

Cyber Shujaa Program

Week 9 Assignment
MLOPS

Student Name: Deborah Kwamboka Omae

Student ID: CS-DA02-25075

Table of Contents

Table of Figures

Introduction

In this assignment, I built an end-to-end machine learning workflow to predict housing prices using the California Housing dataset. The focus was on applying key concepts covered in class: preprocessing, cross-validation, hyperparameter tuning, pipeline construction, and model deployment.

The California Housing dataset is a well-known regression dataset that contains information collected from the 1990 California census. It is available directly through scikit-learn.

Tasks Completed

Step 1: Loading the dataset

In this step I imported the necessary libraries and loaded the California Housing dataset using the fetch_california_housing function. I did this to obtain both the feature variables (X) and the target variable (y) in a structured DataFrame format

Code:

```
#importing the necessary libraries

import pandas as pd

import pickle

from sklearn.datasets import fetch_california_housing

from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import StandardScaler
```

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import r2_score, mean_squared_error

 # Loading the dataset into features and target as pandas data frame

X, y = fetch_california_housing(return_X_y=True, as_frame=True)

screenshot:

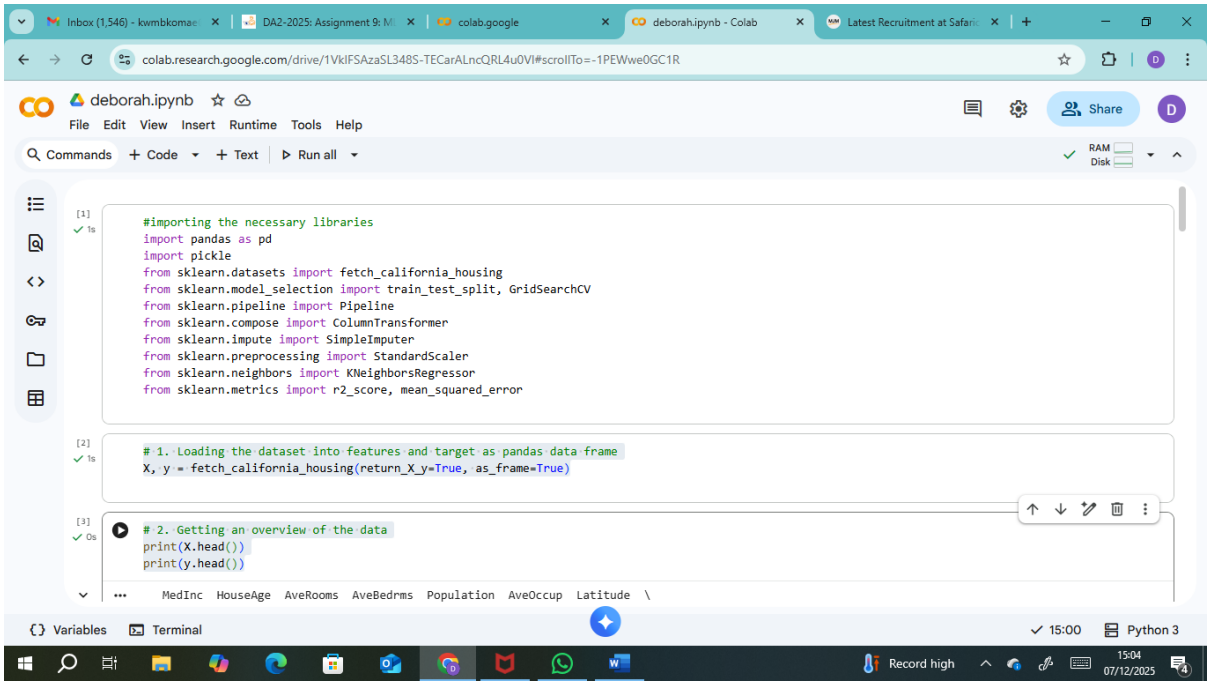

Figure 1: screenshot showing code for loading the dataset

Step 2: Getting an overview of the data

Code:

print(X.head())

print(y.head())

screenshot:

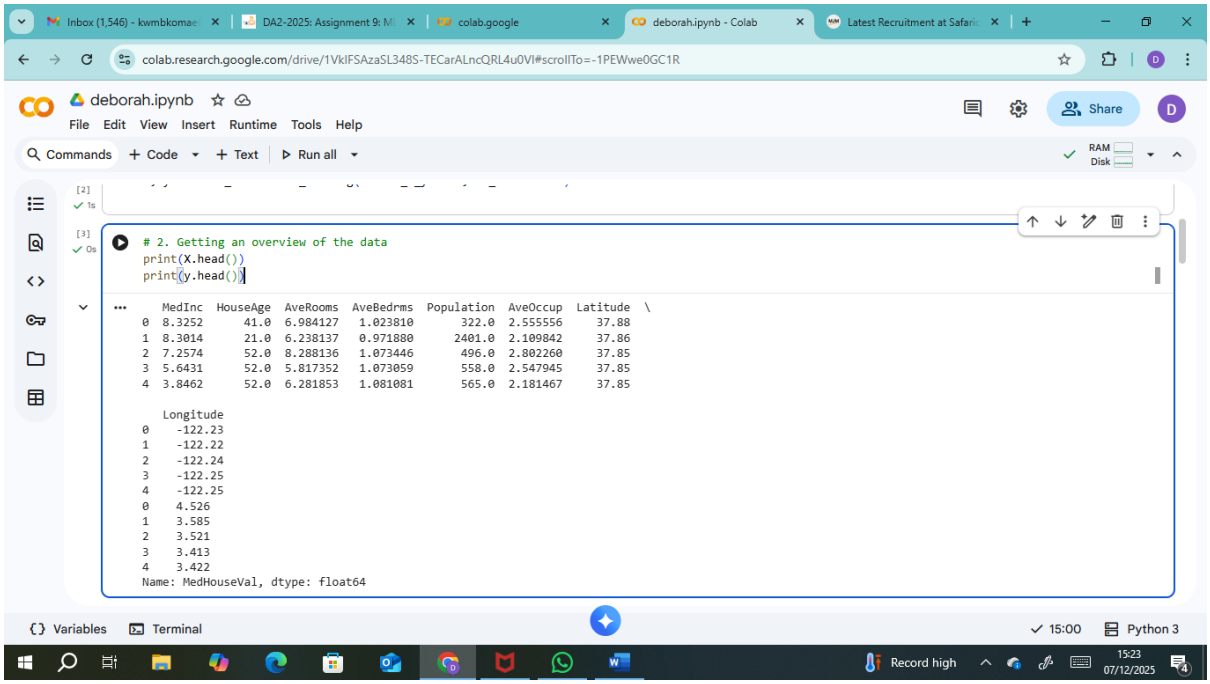

Figure 2: screenshot showing an overview of the data

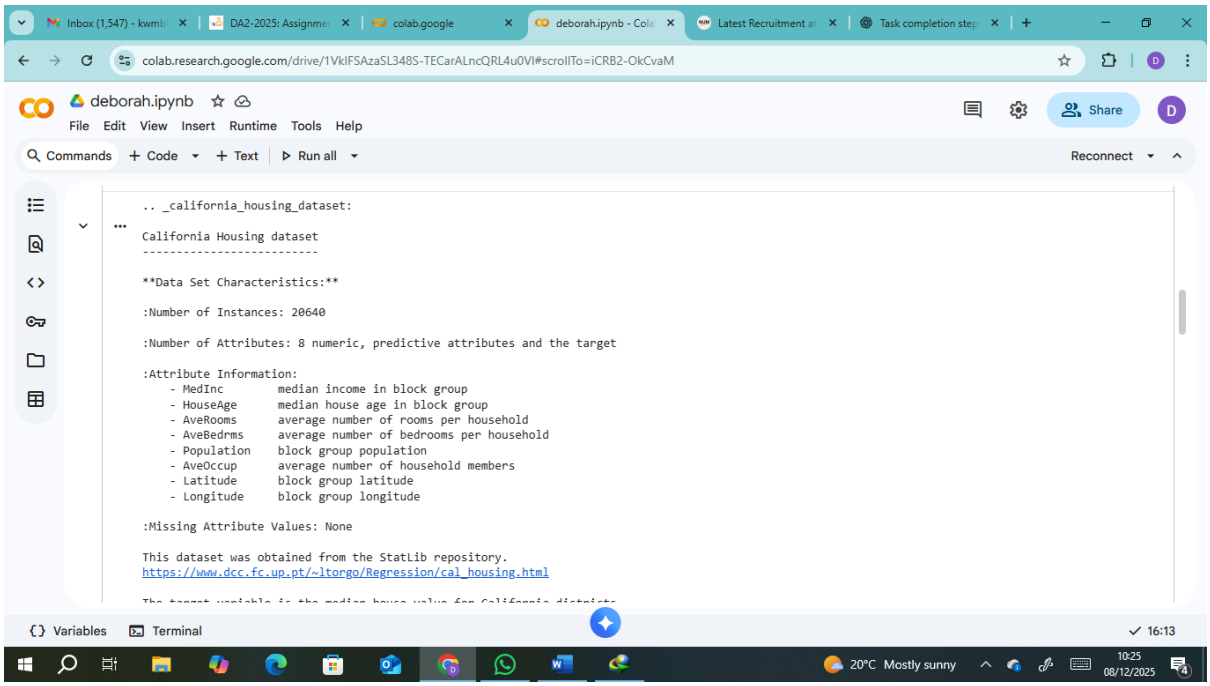

Figure 3: Screenshot showing the attributes of the dataset

Step 3: splitting the dataset into training and testing set

In this step, I performed a train-test split using train_test_split. I did this to separate the dataset into two parts: one for training the model and the other for evaluating its

performance. The test size was set to 20% to ensure enough data remained for proper learning while still having sufficient samples to test the model's generalization ability.

Code:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Step 4: Setting Up the Preprocessing Pipeline

In this step, I created a preprocessing pipeline for handling numerical features. I used a SimpleImputer with a mean-imputation strategy and StandardScaler for normalization. I did this because the K-Nearest Neighbors algorithm is sensitive to missing data and differences in feature scales. Imputation ensures there are no gaps in the dataset, and scaling ensures fair distance calculations during KNN predictions.

Code:

```
#  Preprocessing: Imputation + Scaling for numerical features
numeric_features = X.columns  # all are numerical
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])
```

Step 5: Applying ColumnTransformer

In this step, I wrapped the numerical preprocessing steps inside a ColumnTransformer. I did this to ensure that the defined transformations are applied specifically to the numerical columns. Since the dataset contains only numerical features, this made it easy to preprocess all columns uniformly while keeping the workflow clean and modular.

Code:

```
#  Combine preprocessing using ColumnTransformer
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features)
])
```

Step 6: Building the End-to-End Pipeline

In this step, I constructed a full machine-learning pipeline that included preprocessing followed by the KNN regressor. I did this to streamline the entire modelling process, ensuring that every time the model is trained or evaluated, all preprocessing steps are automatically applied. This prevents data leakage and makes the model reproducible.

Code:

```
#  Build pipeline: preprocessing + KNN
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('knn', KNeighborsRegressor())
])
```

Step 7: Defining the Hyperparameter Grid

In this step, I created a hyperparameter grid for tuning the KNN model. I varied n_neighbors, weights, and the distance metric p. I did this because selecting the correct combination of hyperparameters significantly improves the model's accuracy. GridSearch requires this grid to explore different configurations during cross-validation.

Code:

```
# 6. Define hyperparameter grid
param_grid = {
    'knn__n_neighbors': [3, 5, 7, 9],
```

```
    'knn__weights': ['uniform', 'distance'],
    'knn__p': [1, 2]
}
```

Step 8: Running GridSearchCV with 5-Fold Cross-Validation

In this step, I applied GridSearchCV using the pipeline and the parameter grid. I used 5-fold cross-validation to ensure the model was evaluated fairly across different subsets of the training data. I did this to identify the best hyperparameters while minimizing overfitting, and to maximize the R^2 performance metric.

Code:

```
#  Apply GridSearchCV with 5-fold cross-validation
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    cv=5,
    scoring='r2',
    verbose=1,
    n_jobs=-1
)
```

Step 9: Fitting the Model on the Training Data

In this step, I trained the model by calling grid_search.fit(). I did this to allow GridSearch to try every combination of hyperparameters and select the one that performs best. The output of this step is the fully trained and optimized KNN model.

Code:

```
#  Fit the model
grid_search.fit(X_train, y_train)
```

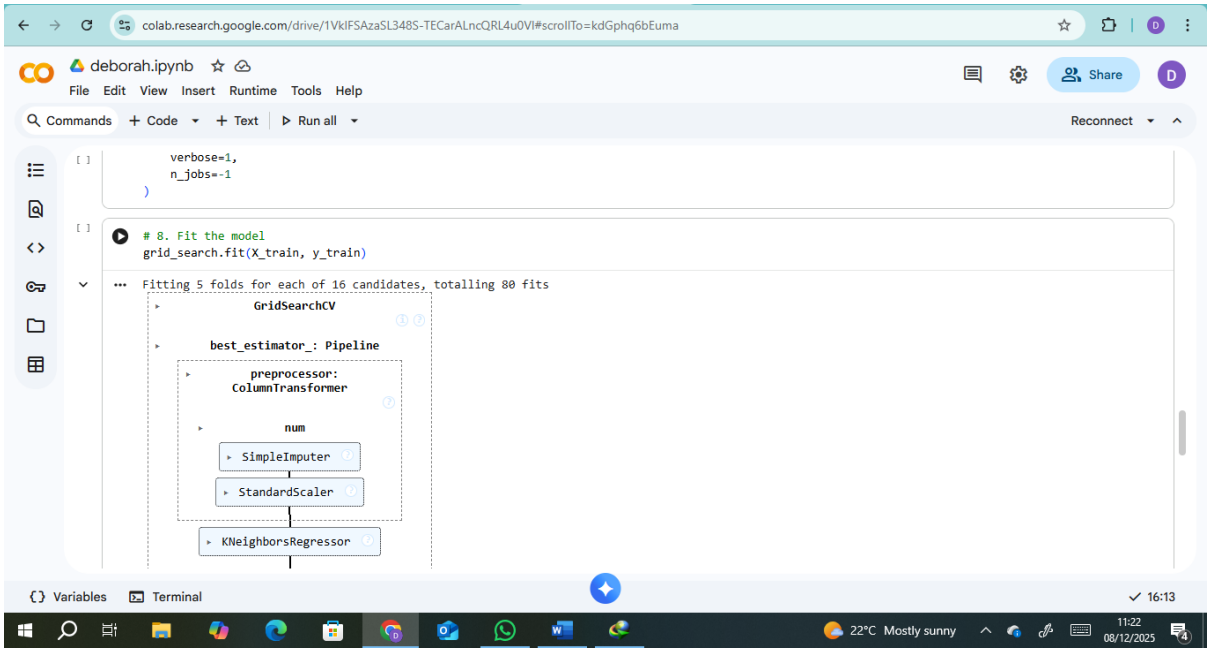

Figure 4: screenshot showing the fitted model

Step 10: Evaluating the Model on the Test Set

In this step, I used the best estimator to make predictions on the test set. Then I calculated performance metrics including R^2 score, MSE, and RMSE. I did this to assess how well the model generalizes to unseen data. These metrics helped me understand whether the selected hyperparameters improved the model's overall accuracy.

Code:

```
#  Evaluate on test set

best_model = grid_search.best_estimator_

y_pred = best_model.predict(X_test)

r2 = r2_score(y_test, y_pred)

mse = mean_squared_error(y_test, y_pred)

rmse = mean_squared_error(y_test, y_pred, squared=False)
```

Step 10: Displaying the Results

In this step, I printed the best hyperparameters, the best cross-validation R^2 score, and all test-set performance results.

Code:

```
#  Print results

print("Best Parameters:", grid_search.best_params_)

print("Best CV R² Score:", grid_search.best_score_)

print("Test R² Score:", r2)

print("Test MSE:", mse)

print("Test RMSE:", rmse)
```

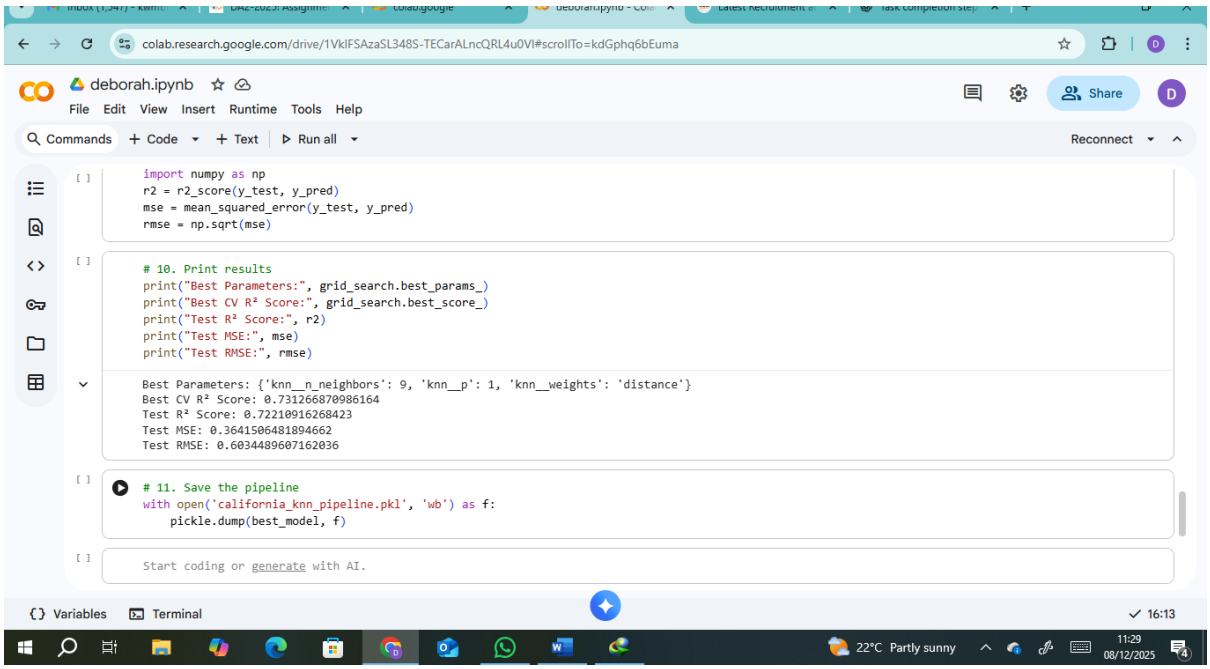

Figure 5: screenshot showing the performance results

Step 11: Saving the Final Pipeline

In this step, I saved the optimized pipeline using the pickle library. I did this so that the trained model can be reused later without retraining.

Code:

```
# Save the pipeline
```

with open('california_knn_pipeline.pkl', 'wb') as f:

pickle.dump(best_model, f)

Link to Code

Link to Code: https://colab.research.google.com/drive/1VkIFSAzaSL348S-TECarALncQRL4u0VI?usp=sharing

Conclusion

In this assignment, I successfully built an end-to-end machine learning workflow using the California Housing dataset. I applied essential steps including data loading, preprocessing, model training, hyperparameter tuning, and performance evaluation. The project improved my understanding of working with real-world datasets, handling missing values, scaling numerical features, and evaluating regression models using metrics such as R^2, MSE, and RMSE. Overall, the assignment strengthened my skills in building clean, reproducible, and well-structured machine learning solutions.