Data and Artificial Intelligence

Cyber Shujaa Program

Week 9 Assignment

MLOPs California Housing Price Prediction

Student Name: John Brown Ouma

Student ID: CS-DA01-25030

Table of Contents

Introduction

This projects implements an end-to-end machine learning workflow to predict housing prices using the California Housing dataset. The focus is on applying key MLOps concepts including data preprocessing, model training with hyperparameter tuning, evaluation, and model serialization. The project uses scikit-learn's KNeighborsRegressor with grid search for optimal performance. The workflow includes:

- Data loading and exploration.
- Preprocessing (imputation, scaling).
- Model training (KNN Regressor with hyperparameter tuning).
- Evaluation (R_squared score, MSE, RMSE).
- Model serialization (saving the trained model).

The goal is to demonstrate MLOps best practices, including reproducible pipelines, cross validation, hyperparameter tuning, and model deployment.

Tasks Completed

1. Data Loading and Exploration

The first step is to load the California Housing dataset, which is available through scikit-learn's fetch_california_housing function. This dataset contains 20,640 samples with 8 numerical features and a target variable (median house value in $100,000s). Loading the data as a pandas DataFrame (as_frame=True) facilitates exploration and manipulation.

Import Required Libraries

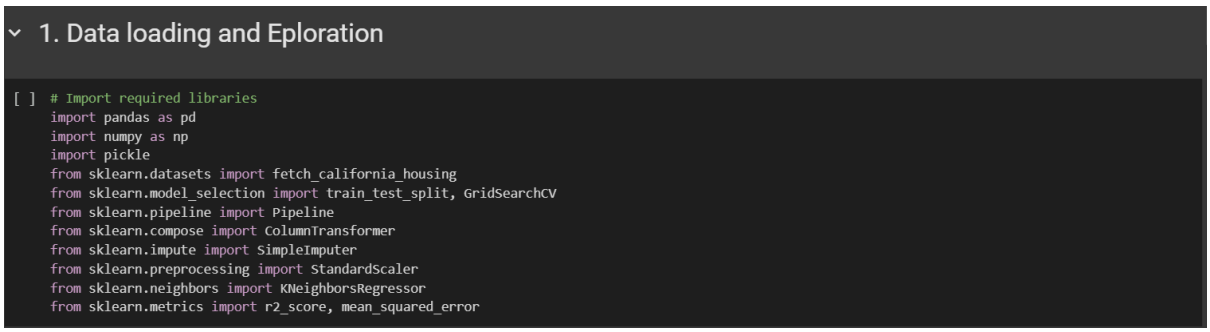

Figure 1: Showing required libraries

Pandas: Data manipulation.

Pickle: Saving/loading trained models.

Scikit-learn: Machine learning tools (datasets, preprocessing, models. Evaluation).

Load the Dataset

```
[ ] # Load Dataset
    X, y = fetch_california_housing(return_X_y=True, as_frame=True)
    print(X.head())
    print("\nTarget variable (median house value) sample:\n", y.head())

    MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude
0    -122.23
1    -122.22
2    -122.24
3    -122.25
4    -122.25

Target variable (median house value) sample:
 0    4.526
1    3.585
2    3.521
3    3.413
4    3.422
Name: MedHouseVal, dtype: float64
```

Figure 1.1: Showing Dataset Loading

Fetch_california_housing loads the dataset directly from scikit-learn.
return_X_y=True returns features **(X)** and target **(y)** separately.
As_frame=True converts them into pandas DataFrame for easier analysis.
Output: First 5 rows of features **(X)** and target **(y).**

2. Data Preprocessing and Pipeline Construction

Train-Test Split (80/20)

Purpose: Splitting the data into training (80%) and test (20%) sets ensures the model is trained on one subset and evaluated on unseen data, preventing overfitting and simulating real-world performance.

```
# Train-test split (80/20)
X_train, X_test, y_train, y_test =train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 2: Train-Test Split

Test_size=0.2: 20% of data is reserved for testing
Random_state=42: Ensures reproducibility (same split everytime) by fixing the random seed.

Significance: The 80/20 split balances sufficient training data with a robust test set for evaluation. The fixed random seed ensures consistent results across runs, which is critical for debugging and reporting.

Preprocessing Pipeline

Purpose: Preprocessing standardizes the numerical features to ensure the KNeighborsRegressor performs optimally, as KNN is sensitive to feature scales. The ColumnTransformer applies transformations (imputation and scaling) to all features.

Numerical Feature Transformation

```
#Preprocessing pipeline for numerical features
numeric_features = X.columns
numeric_transformer = Pipeline(steps=[
    ('impute', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])
```

Figure 2.1: Numerical Feature Transformation

Implementation:

- All features are numerical, so numeric_features = X.columns selects all columns.
- A Pipeline is created with:
    - **SimpleImputer(strategy='mean')** to handle potential missing values (though none exist here).
    - **StandardScaler()** to standardize features to zero mean and unit variance.
- The **ColumnTransformer** applies this pipeline to all numerical features.

Combine Preprocessing Steps

```
# ColumnTransformer for all features
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, numeric_features)
])

# Full pipeline with KNN regressor
```

Figure 2.2: ColumnTranformer

ColumnTranformer applies **numeric_transformer** to all numeric features.

Significance: Standardization is critical for KNN, as it relies on distance metrics (e.g., Euclidean or Manhattan). Imputation ensures robustness for future datasets with missing values. The ColumnTransformer allows modular preprocessing, making the pipeline adaptable to datasets with mixed feature types.

3. Hyperparameter Tuning with GrindSearchCV

Pipeline structure

Purpose: The Pipeline combines preprocessing and the KNeighborsRegressor model into a single object, streamlining training, tuning, and deployment.

Implementation:

- The pipeline is defined as Pipeline (steps=[('preprocessor', preprocessor), ('knn', KNeighborsRegressor())]).
- The preprocessor from the ColumnTransformer is paired with the KNN model.

```
# Full pipeline with KNN regressor
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('knn', KNeighborsRegressor())
])
```

Figure 3: Pipeline

- Preprocessing **(StandardScaler + Imputer)**
- Model **(KNeighborsRegressor)**

Significance: Pipelines ensure that preprocessing (e.g., scaling) is applied consistently to training and test data, preventing data leakage. They also simplify hyperparameter tuning by treating the entire workflow as a single estimator.

GridSearchCV tests all combinations of hyperparameters.

Purpose: Hyperparameter tuning optimizes the KNeighborsRegressor's performance by testing combinations of n_neighbors, weights, and p to maximize the R^2 score.

Implementation:

- The hyperparameter grid is defined as:
  - knn__n_neighbors: [3, 5, 7, 9] (number of neighbors for KNN).
  - knn__weights: ['uniform', 'distance'] (weighting scheme for neighbors).
  - knn__p: [1, 2] (Manhattan vs. Euclidean distance).
- GridSearchCV is configured with 5-fold cross-validation, R^2 scoring, and parallel processing (n_jobs=-1).
- The grid search is executed with grid_search.fit(X_train, y_train).

```
[ ]  # Define hyperparameter grid
     param_grid = {
         'knn__n_neighbors': [3, 5, 7, 9],
         'knn__weights': ['uniform', 'distance'],
         'knn__p': [1, 2]
     }

     # Grid search with 5-fold CV
     grid_search = GridSearchCV(
         estimator=pipeline,
         param_grid=param_grid,
         cv=5,
         scoring='r2',
         verbose=1,
         n_jobs=-1
     )

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

Figure 3.1: GridSearchCV tests all combinations of hyperparameters.

5-folds cross-validation ensures robust model selection.

Best model is stored in **grid_search.best_estimator_.**

Significance: Grid search exhaustively tests 4 × 2 × 2 = 16 combinations, using cross-validation to estimate generalization performance. The R^2 score measures how well the model explains variance in the target variable. Parallel processing speeds up computation.

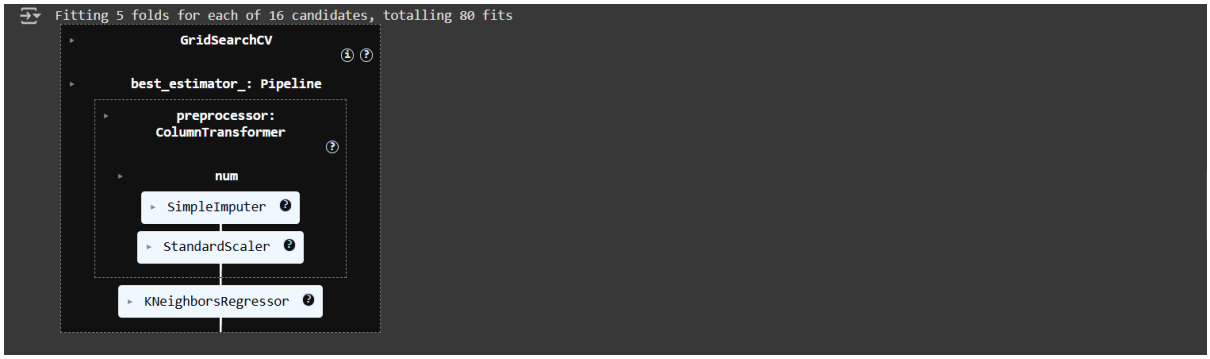

Figure 3.2 Showing Output for GridSearchCV tests all combinations of hyperparameters.

4. Model Evaluation.

Purpose: Evaluate the best model (from GridSearchCV) on the test set to assess its performance on unseen data using R^2, MSE, and RMSE.

Implementation:

- The best model is extracted with best_model = grid_search.best_estimator_.
- Predictions are made on the test set: y_pred = best_model.predict(X_test).
- Metrics are computed:
  - r2_score(y_test, y_pred) for R^2.
  - mean_squared_error(y_test, y_pred) for MSE.
  - mean_squared_error(y_test, y_pred, squared=False) for RMSE.

Significance: R^2 indicates the proportion of variance explained (higher is better). MSE and RMSE quantify prediction errors in the target's units ($100,000s), with RMSE being more interpretable due to its square root transformation. Comparing cross-validation and test R^2 scores assesses overfitting.

Interpretation:

```
[ ] # 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)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))

    # 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)

    Best Parameters: {'knn__n_neighbors': 9, 'knn__p': 1, 'knn__weights': 'distance'}
    Best CV R² Score: 0.731266870986164
    Test R² Score: 0.72210916268423
    Test MSE: 0.3641506481894662
    Test RMSE: 0.6034489607162036
```

Figure 4: Model Evaluation

R-squared = 0.7221: the model explains ~70% of the variance in housing prices.

RMSE =0.6034: Average prediction error is ~$60, 300 (since target is in $100, 000s).

5. Model Serialization.

```
[ ]  # Save the pipeline
     with open('california_knn_pipeline.pkl', 'wb') as f:
         pickle.dump(best_model, f)

     print(" Final pipeline saved to 'california_knn_pipeline.pkl'")

     Final pipeline saved to 'california_knn_pipeline.pkl'
```

Figure 5: model Serialization

Purpose: Serializing the trained pipeline to a file allows it to be reused for predictions without retraining, enabling deployment.

Implementation:

- The best model is saved using pickle.dump(best_model, f) to california_knn_pipeline.pkl.

Significance: Serialization with pickle preserves the entire pipeline (preprocessing + model), ensuring consistent preprocessing for new data. This is a key MLOps practice for model deployment.

Link to Code:

https://colab.research.google.com/drive/1E4kgMe5ctoWlAj6YO_NzSPwxOdlebqyk?usp=sharing

Conclusion

Through this assignment, I learned:

- The importance of building reproducible ML pipelines that combine preprocessing and modeling steps.
- How GridSearchCV automates the process of hyperparameter tuning with cross-validation.
- The value of model serialization for deployment and reuse.

The KNN model achieved a reasonable R-squared score of 0.722 on the test set, suggesting it captures about 70% of the variance in housing prices. For production use, we might want to experiment with more powerful algorithms like Random Forest or Gradient Boosting.