Start coding or generate with AI.

k-NN Housing Price Prediction

k-NN Housing Price Prediction

Student: Mike Maurrasse
Date: 8/31/2025

This notebook implements a pipeline for predicting California house prices using a custom k-Nearest Neighbors model.

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

from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import fetch_california_housing

pd.set_option('display.max_columns', 100)
pd.set_option('display.width', 120)

RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)
```

Data Loading and Exploration

```
cali = fetch_california_housing(as_frame=True)
df = cali.frame.copy()
df.columns = ['MedInc','HouseAge','AveRooms','AveBedrms','Population','AveOccup','Latitude','Longitude','MedHouseVal']
target_col = 'MedHouseVal'

print("Shape:", df.shape)
df.head()
```

EDA Write-up

When I explored the target variable (median house value), I saw that the distribution was right-skewed. Most houses are in the lower-to-mid price range, and there are a few very expensive homes that show up as outliers. These can be flagged statistically, but I consider them realistic because California does have very high-value properties.

I noticed that median income, latitude, and average rooms per household are the features most strongly tied to prices. Income has a clear positive relationship with home values, and geography stood out too — coastal and metro areas like San Francisco and Los Angeles drive higher prices. Latitude also reflects the north–south gradient across the state.

Some challenges I'll need to manage include skewed feature distributions (like income, rooms, population), the presence of outliers, and multicollinearity among room-related variables. To address this, scaling and careful feature engineering will be important for keeping the model stable and fair.

```
plt.hist(df[target_col], bins=40)
plt.title("Distribution of Median House Value")
plt.xlabel("MedHouseVal")
plt.ylabel("Count")
plt.show()

corr = df.corr()
plt.imshow(corr, cmap='coolwarm', interpolation='nearest')
plt.xticks(range(len(corr.columns)), corr.columns, rotation=90)
```

```
plt.yticks(range(len(corr.columns)), corr.columns)
plt.colorbar()
plt.show()
```

Handling Outliers

Both the IQR and Z-score methods flagged expensive homes and big population counts as outliers. If I dropped them, the dataset would shrink a lot and I'd also lose valid housing scenarios, like dense city blocks. For that reason, I decided **not to remove outliers**. Instead, I'll leave them in, note their influence, and rely on scaling and robust models to manage their impact.

Outlier Example

```
q1, q3 = df[target_col].quantile([0.25, 0.75])
iqr = q3 - q1
lower, upper = q1 - 1.5*iqr, q3 + 1.5*iqr
outliers = ((df[target_col] < lower) | (df[target_col] > upper)).sum()
print("Potential outliers in target:", outliers)
```

Scaling Choice

Why I Picked StandardScaler

I decided to use **StandardScaler** for preprocessing. My features are on very different scales — population can be in the thousands, while household size is just a few digits. Without scaling, distance metrics would overweight the larger numbers.

I avoided MinMaxScaler because it compresses most values into a tiny range when outliers are present. StandardScaler centers the data and reduces the risk of one variable dominating the distance calculation.

Feature Engineering

```
df['rooms_per_household'] = df['AveRooms'] / df['AveOccup']
df['bedrooms_per_room'] = df['AveBedrms'] / df['AveRooms']
df['population_per_household'] = df['Population'] / df['AveOccup']

---
```

Custom k-NN Implementation

```
def euclidean_distance(point1, point2):
    diff = point1 - point2
    return float(np.sqrt(np.dot(diff, diff)))

def manhattan_distance(point1, point2):
    return float(np.sum(np.abs(point1 - point2)))

def minkowski_distance(point1, point2, p=2):
    return float(np.power(np.sum(np.abs(point1 - point2)**p), 1.0/p))

class CustomKNN:
    def __init__(self, k=5, distance_metric='euclidean', weights='uniform', minkowski_p=2):
        self.k = int(k)
        self.distance_metric = distance_metric
        self.weights = weights
        self.minkowski_p = minkowski_p
        self.X_train = None
        self.y_train = None

    def fit(self, X, y):
        self.X_train = np.asarray(X, dtype=float)
        self.y_train = np.asarray(y, dtype=float)
        return self
```

```
    def _calculate_distance(self, p1, p2):
        if self.distance_metric == 'euclidean':
            return euclidean_distance(p1, p2)
        if self.distance_metric == 'manhattan':
            return manhattan_distance(p1, p2)
        if self.distance_metric == 'minkowski':
            return minkowski_distance(p1, p2, p=self.minkowski_p)
        raise ValueError("Unsupported distance_metric")

    def _get_neighbors(self, test_point):
        dists = np.array([self._calculate_distance(test_point, tr) for tr in self.X_train])
        idx = np.argsort(dists)[:self.k]
        return idx, dists[idx]

    def predict_single(self, test_point):
        idx, d = self._get_neighbors(test_point)
        y_neighbors = self.y_train[idx]
        if self.weights == 'uniform':
            return float(np.mean(y_neighbors))
        if self.weights == 'distance':
            if np.any(d == 0):
                return float(y_neighbors[d == 0][0])
            w = 1.0 / d
            return float(np.sum(w * y_neighbors) / np.sum(w))
        raise ValueError("Unsupported weights")

    def predict(self, X_test):
        return np.array([self.predict_single(x) for x in X_test])

    def score(self, X_test, y_test):
        y_pred = self.predict(X_test)
        return r2_score(y_test, y_pred)
```

Implementation Decisions

- **Alternatives I thought about:** KD-Tree or Ball-Tree for faster neighbor searches, but I stuck with brute force to focus on the fundamentals.
- **Limitations:** My custom k-NN is slower on large datasets, and the feature “importance” measure I added is just an approximation.
- **Scaling choice:** StandardScaler helped keep things robust in the presence of outliers.

Personal Reflection

The hardest part for me understanding the k-NN class coding and debugging the distance-weighted prediction. Once it worked, I felt like I understood the algorithm a lot better.

If I had more time, I’d add KD-Tree optimization and try other weighting schemes beyond inverse distance.

For real-world use cases, k-NN could be applied in recommendation systems (finding similar users or movies), anomaly detection in financial data, and regression problems like property valuation.