

Housing Price Prediction

ML Workflow — Tools & Concepts Guide

A complete reference for every tool and technique used

Your first ML model — explained step by step

Stage	What you did
1. Data Loading	Read a CSV with pandas; inspected shape, types, distributions
2. EDA	Histograms, geo-scatter, correlation bar charts
3. Train/Test Split	Stratified by income category; 80/20 split
4. Feature Engineering	Derived ratio features; log/RBF transforms
5. Preprocessing	Imputation, encoding, scaling inside sklearn Pipelines
6. Model Training	LinearRegression & DecisionTreeRegressor
7. Evaluation	MAE, RMSE, R ² , 5-fold CV, 95% CI
8. Hyperparameter Tuning	GridSearchCV over max_depth & RBF gamma
9. Persistence	joblib.dump / joblib.load

01 Data Loading & Initial Inspection

pandas — pd.read_csv()

pandas is Python's premier data manipulation library. **pd.read_csv()** reads a CSV file directly into a **DataFrame** — a 2-D table with labeled rows and columns, analogous to an Excel sheet but far more powerful.

Method	What it does
df.shape	Returns (rows, columns) — quick size check
df.info()	Column names, non-null counts, dtypes — spot missing data instantly
df.describe()	Count, mean, std, min/max/quartiles for every numeric column
df.head(n)	First n rows — sanity-check your data loaded correctly
df['col'].value_counts()	Frequency table for categorical columns

Key insight from loading

The dataset has **20,640 rows** and **10 columns**. **total_bedrooms** is the only column with missing values (~207 NaNs). This needs to be handled before training.

ocean_proximity is the sole *categorical* column — ML models require numbers, so it must be encoded before fitting.

02 Exploratory Data Analysis (EDA)

Why EDA matters

EDA is detective work: you uncover the story hidden in the numbers before writing a single model line. Skipping it leads to surprised pipelines and embarrassing bugs.

matplotlib & seaborn

matplotlib is Python's foundational plotting library. **seaborn** wraps it with higher-level statistical charts and a cleaner default style.

Chart	Tool	Purpose
df.hist()	matplotlib	Spot skew, bimodality, capped values
sns.scatterplot()	seaborn	Two-variable relationship + categorical colour/size
corr().plot.barh()	pandas/mpl	Rank features by linear association with target
plt.subplots() + twinx()	matplotlib	Overlay histogram and RBF curve on shared x-axis

Key EDA findings

- median_income is the strongest predictor of house value ($r \approx 0.69$).
- total_rooms and population are heavily right-skewed — a log transform helps.

- median_house_value is capped at \$500,000 — a data artifact to keep in mind.
- Coastal districts (NEAR OCEAN, ISLAND) command higher prices geographically.

03 Train / Test Split

Why split at all?

A model evaluated on its own training data always looks better than it really is — it has memorised the answers. Holding out a test set gives an honest estimate of performance on unseen houses.

sklearn.model_selection.train_test_split

The workhorse split function. Key parameters:

Parameter	Value used	Meaning
test_size	0.2	20% of rows reserved for testing
random_state	40	Seed for reproducibility — same split every run
stratify	income category	Preserve income distribution in both halves

Stratified Sampling — why it matters

If you randomly sample 20% of rows without stratification, you might accidentally under-represent the high-income districts that drive house prices up. **Stratification** guarantees the same income-band proportions in both sets.

You first created income bins with `pd.cut()`, then passed them as the **stratify** argument. This is a standard professional technique.

Manual vs. sklearn split

You also hand-coded two manual split methods (random permutation and CRC-32 hash-based splitting) before using sklearn. This was excellent learning: understanding what the library does under the hood prevents you from using it as a black box.

04 Feature Engineering

What is feature engineering?

Raw features are not always the most informative form for a model. Feature engineering creates new columns that better capture the underlying signal.

Ratio features

Census data reports totals per block, not per household. A block of 1,000 rooms means very different things if it has 50 households vs. 500. Ratios fix this:

New feature	Formula	Correlation gain
-------------	---------	------------------

rooms_per_house	total_rooms / households	Positive — spacious homes cost more
bedrooms_ratio	total_bedrooms / total_rooms	Negative — high bedroom ratio → cheaper
people_per_house	population / households	Negative — crowded blocks → cheaper

Log Transform (heavy tails)

Features like **population** and **total_rooms** are right-skewed: most values cluster near zero but a few are enormous. A log transform compresses the tail, making the distribution more symmetric and helping linear models.

```
FunctionTransformer(np.log, feature_names_out='one-to-one')
```

RBF Kernel Transform (housing_median_age)

A Radial Basis Function (RBF) measures how *similar* each house's age is to a reference point (35 years). It outputs a value between 0 and 1 — peaking at the reference and decaying with distance. **gamma** controls how quickly it decays: higher gamma = sharper peak.

```
rbf_kernel(housing_ages, [[35]], gamma=0.1)
```

05 Preprocessing

The problem: ML models only accept clean numbers

Raw data has missing values, categorical strings, and features on wildly different scales. Preprocessing converts everything into the clean numeric matrix a model can consume.

SimpleImputer — handling missing values

Replaces NaN values with a computed statistic. You used **strategy='median'** for numeric columns (robust to outliers) and **strategy='most_frequent'** for the categorical column.

OrdinalEncoder vs. OneHotEncoder

Encoder	Output	When to use
OrdinalEncoder	Integer 0, 1, 2... per category	Only if categories have a natural order
OneHotEncoder	One binary column per category	When categories are nominal (no order) — like ocean_proximity

ocean_proximity has no inherent order ('ISLAND' is not greater than 'INLAND'), so **OneHotEncoder** is the correct choice.

StandardScaler vs. MinMaxScaler

Scaler	Formula	Output range	Use when
StandardScaler	$(x - \text{mean}) / \text{std}$	$\sim(-3, 3)$	General purpose; handles outliers better
MinMaxScaler	$(x - \text{min}) / (\text{max} - \text{min})$	$(-1, 1)^*$	When bounded range is needed; sensitive to outliers

Pipeline & make_pipeline

A **Pipeline** chains transforms in sequence. Each step's output becomes the next step's input. The key benefit: calling **.fit()** on the pipeline fits every step in order, and **.transform()** applies them — preventing data leakage by ensuring fit statistics are computed only on training data.

```
Pipeline([('imputer', SimpleImputer()), ('scaler', StandardScaler())])
```

ColumnTransformer

Different column types need different treatment. ColumnTransformer routes each subset of columns to its own sub-pipeline and concatenates the results:

Sub-pipeline	Columns routed to it
ratio_pipeline()	Pairs of columns for ratio features
heavy_tail_pipeline	Skewed numeric columns (log-transformed)
age_rbf_transformer	housing_median_age
categorical_pipeline	ocean_proximity
default_num_pipeline (remainder)	longitude, latitude, median_income

06 Custom Transformer (Duck Typing)

Why build your own transformer?

Sometimes sklearn's built-ins don't cover your exact logic. You can write a custom class that integrates seamlessly into any Pipeline by following sklearn's **duck-typing** convention: implement **fit()** and **transform()**.

BaseEstimator + TransformerMixin

Inheriting from these two base classes gives your class free implementations of **get_params()**, **set_params()**, and **fit_transform()** with zero extra code.

```
class CustomStandardScaler(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        X = check_array(X)
        self.mean_ = X.mean(axis=0)
        self.std_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1]
        return self
    def transform(self, X):
        check_is_fitted(self)
        return (X - self.mean_) / self.std_
```

sklearn compatibility conventions

- Always return self from fit() — enables method chaining.
- Store learned values as attributes ending with _ (e.g. mean_).
- Use check_array() to validate input; check_is_fitted() to guard transform().
- n_features_in_ must be set in fit() for Pipeline compatibility.

07 Model Training

LinearRegression

Finds the straight-line relationship between features and target by minimising the sum of squared errors (OLS). It has **no hyperparameters to tune** and trains in milliseconds — always a good baseline.

Limitation: assumes the relationship is linear. House prices have clear non-linearities (price caps, geographic clusters), so LR underperforms.

DecisionTreeRegressor

Recursively partitions the feature space into rectangular regions, predicting the mean target value in each leaf. Captures non-linear interactions naturally.

Overfitting risk: an unconstrained tree will grow until every training sample has its own leaf, giving $R^2 = 1.0$ on train but terrible generalisation. **max_depth** controls this by limiting how deep the tree can grow.

Model	Train MAE	CV MAE	Verdict
LinearRegression	~\$50k	~\$51k	Consistent — low variance, moderate bias
DecisionTree (full)	~\$0	~\$65k	Overfits badly without max_depth constraint
DecisionTree (tuned)	—	~\$49k	Best after GridSearchCV

TransformedTargetRegressor

Wraps any regressor to automatically scale (and inverse-scale) the target variable. You used it to apply StandardScaler to y, removing the need to manually inverse_transform predictions.

08 Model Evaluation & Cross-Validation

Metrics

Metric	Formula	Interpretation
MAE	$\text{mean } y - \hat{y} $	Average absolute dollar error — easy to explain
MSE	$\text{mean } (y - \hat{y})^2$	Penalises large errors more heavily than MAE
RMSE	$\sqrt{\text{MSE}}$	Same unit as target; most commonly reported
R^2	$1 - \text{SS}_{\text{res}}/\text{SS}_{\text{tot}}$	Fraction of variance explained; 1.0 = perfect

cross_val_score — 5-fold Cross-Validation

Instead of a single train/validation split, CV divides the training set into **k=5 folds** and trains/evaluates k times, rotating which fold is validation. The k scores are averaged for a more stable estimate of true generalisation.

You passed **scoring='neg_mean_absolute_error'** because sklearn convention requires higher = better, so it negates the MAE. You negated it back with '-'.

Confidence Interval on RMSE

A single test-set RMSE is itself a random variable. You computed a **95% Student-t CI** on the squared errors to quantify uncertainty in that estimate — a professional finishing touch that most beginners skip.

```
se = (test_target - predictions)**2
ci = np.sqrt(stats.t.interval(0.95, df=len(se)-1, loc=np.mean(se), scale=stats.sem(se)))
```

09 Hyperparameter Tuning — GridSearchCV

What are hyperparameters?

Model hyperparameters are settings chosen *before* training — they control the learning process rather than being learned from data. Examples: **max_depth** in a Decision Tree, **gamma** in an RBF kernel.

GridSearchCV

Exhaustively tries every combination in a parameter grid, evaluating each with cross-validation, and exposes the best combination via **.best_params_**.

```
param_grid = { 'preprocessing__rbf__gamma': [ 0.1, 0.2, 0.5, 1, ], 'dtr__max_depth': [5, 10, 15], }
gs = GridSearchCV(pipeline, param_grid, scoring='neg_mean_absolute_error', cv=5)
```

You searched over **$2 \times 3 = 6$ combinations $\times 5$ folds = 30 model fits**. The double-underscore syntax (**step__param**) dives into a named Pipeline step to set its parameter — this is one of sklearn's most powerful features.

Feature Importances

After tuning, you inspected **feature_importances_** from the Decision Tree. This array sums to 1.0 and shows the fractional contribution of each feature to reducing impurity (variance). High importance = the tree relies heavily on that feature when making splits.

10 Model Persistence — joblib

Why save the model?

Training can take minutes to hours. Saving serialises the entire fitted Pipeline — including all fitted preprocessing steps — so you can load and predict instantly without re-training.

```
import joblib # Save
joblib.dump(best_model, 'housing_model.pkl') # Load anywhere
model = joblib.load('housing_model.pkl')
predictions = model.predict(new_features)
```

joblib vs. pickle

joblib is preferred over Python's built-in pickle for ML models because it uses memory-mapped files for large numpy arrays, making save/load much faster for models with big weight matrices or transformer statistics.

What gets saved?

The entire Pipeline object, including:

- All fitted ColumnTransformer sub-pipelines (imputer medians, scaler means/stds, OHE categories)

- The trained DecisionTreeRegressor (all split thresholds and leaf values)
- The GridSearchCV best_params_ configuration

11 Complete Workflow Summary

Step	Tool(s)	Key concept
Load data	pandas.read_csv	DataFrame, dtypes, missing values
EDA	matplotlib, seaborn, pandas.corr	Distributions, correlations, geography
Stratified split	train_test_split(stratify=)	Avoid sampling bias on key feature
Feature engineering	FunctionTransformer, ratios	Domain knowledge → better signal
Imputation	SimpleImputer	Median/most-frequent fill for NaNs
Encoding	OneHotEncoder	Nominal categories → binary columns
Scaling	StandardScaler, MinMaxScaler	Equal feature magnitude for models
Pipeline	Pipeline, ColumnTransformer	No leakage; reusable transforms
Training	LinearRegression, DecisionTree	Baseline comparison
Cross-validation	cross_val_score	Honest generalisation estimate
Hyperparameter tuning	GridSearchCV	Optimal depth + RBF gamma
Test evaluation	MAE, RMSE, R ² , CI	Unbiased final assessment
Persistence	joblib	Save & reload the full pipeline

What to learn next

- Random Forests & Gradient Boosting (XGBoost, LightGBM) — typically cut RMSE by 20-30%
- RandomizedSearchCV — faster than GridSearch for large hyperparameter spaces
- Permutation importance — model-agnostic feature importance
- MLflow — experiment tracking so you never lose a good run
- Cross-validated pipelines with sklearn Pipeline inside GridSearchCV (what you already did!)

Great work on your first ML model!

Every professional ML engineer started exactly where you are.