Chapter 2

End-to-End Machine Learning Project

Chapter 2 walks through a complete end-to-end Machine Learning workflow using the California Housing Prices dataset. The goal is to behave like a data scientist designing a real estate price prediction system, illustrating each step from problem framing to deployment. The chapter emphasizes real-world practices: using genuine datasets, preparing robust pipelines, evaluating models correctly, and ensuring long-term maintainability.

Look at the Big Picture

The project's business objective is to predict median housing prices for California districts using census attributes such as income, population, and housing age. Predictions will feed a downstream ML system (illustrated in **Figure 2-2**), which determines real estate investment priorities. This makes accurate regression outputs critical. The problem is thus framed as **supervised**, **multiple**, and **univariate regression**, and batch learning is sufficient since the dataset is small and static.

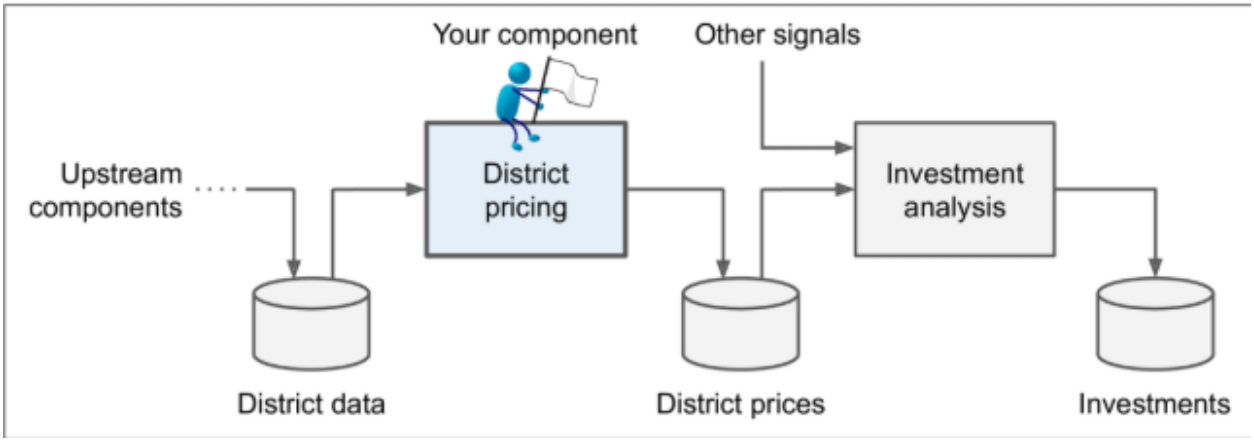

Figure 2-2. A Machine Learning pipeline for real estate investments

Selecting a performance metric is also essential. The chapter introduces **RMSE** (Equation 2-1) and **MAE** (Equation 2-2), explaining vector norms and their sensitivity to outliers. Before proceeding, assumptions must be checked—for instance, whether downstream systems require precise values or just categories.

Get the Data

The chapter stresses using real datasets and introduces the California Housing dataset (shown in **Figure 2-1**).

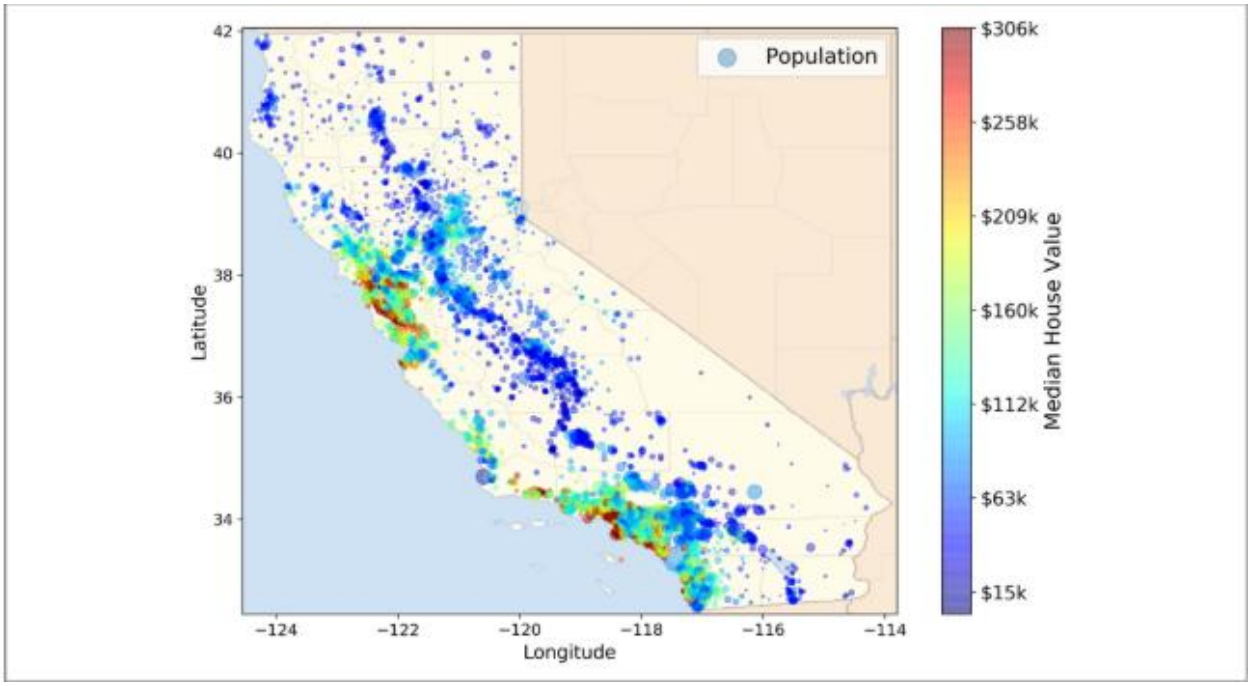

Figure 2-1. California housing prices

It demonstrates how to organize a workspace, install dependencies, set up isolated environments, and load data via pandas. The dataset contains 20,640 districts and both numerical and categorical attributes.

Initial inspection using head(), info(), and describe() reveals missing values (notably in total_bedrooms) and one categorical feature (ocean_proximity). Histograms (**Figure 2-8**) show capped values and heavy-tailed distributions, suggesting potential transformations.

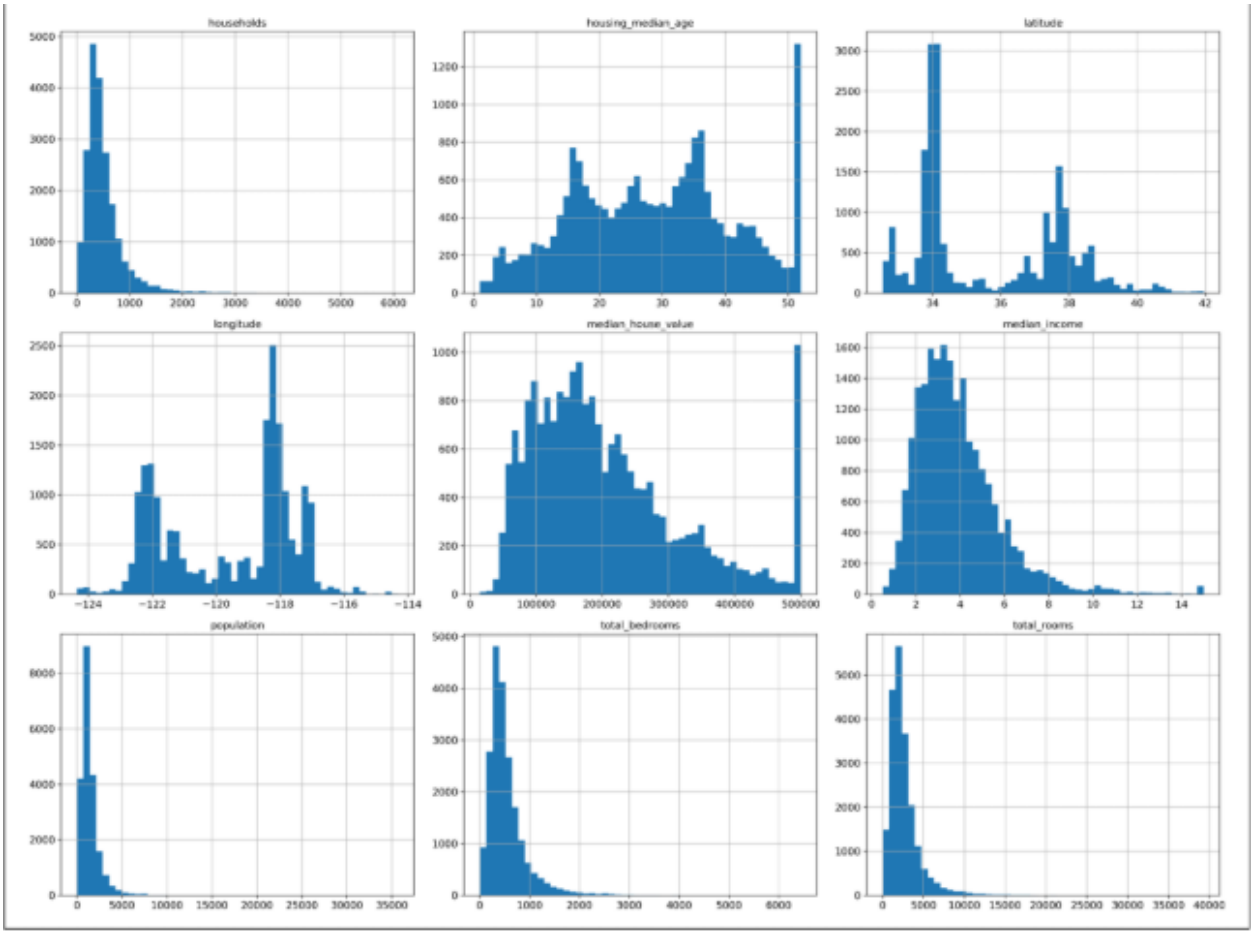

Figure 2-8. A histogram for each numerical attribute

Before exploring further, the chapter emphasizes creating a **test set** early to avoid data snooping bias. Methods include random splitting, stable splitting via hashed identifiers, and Scikit-Learn's train_test_split(). To reduce sampling bias, **stratified sampling** based on income categories is introduced using StratifiedShuffleSplit, shown in contrast with random sampling in **Figure 2-10**.

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

Discover and Visualize the Data to Gain Insights

The chapter recommends analyzing only the training set. Visualization begins with geographical scatterplots (**Figure 2-11** and **2-12**), which highlight dense population clusters and coastal price patterns.

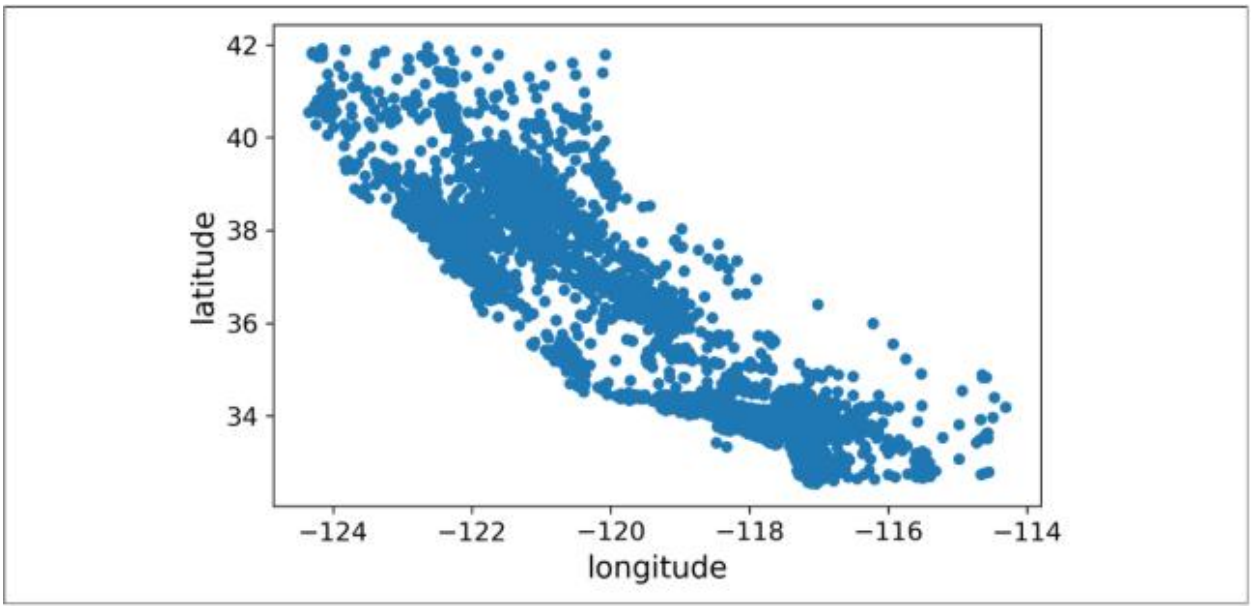

Figure 2-11. A geographical scatterplot of the data

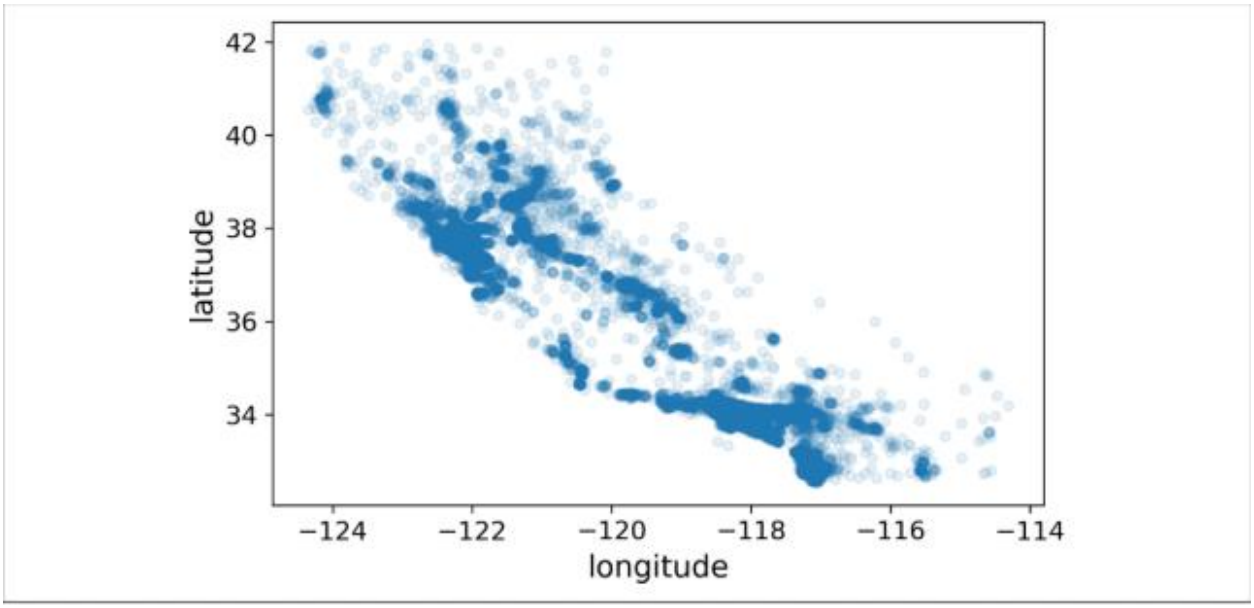

Figure 2-12. A better visualization that highlights high-density areas

A richer visualization (**Figure 2-13**) uses color to indicate housing values and circle size for population.

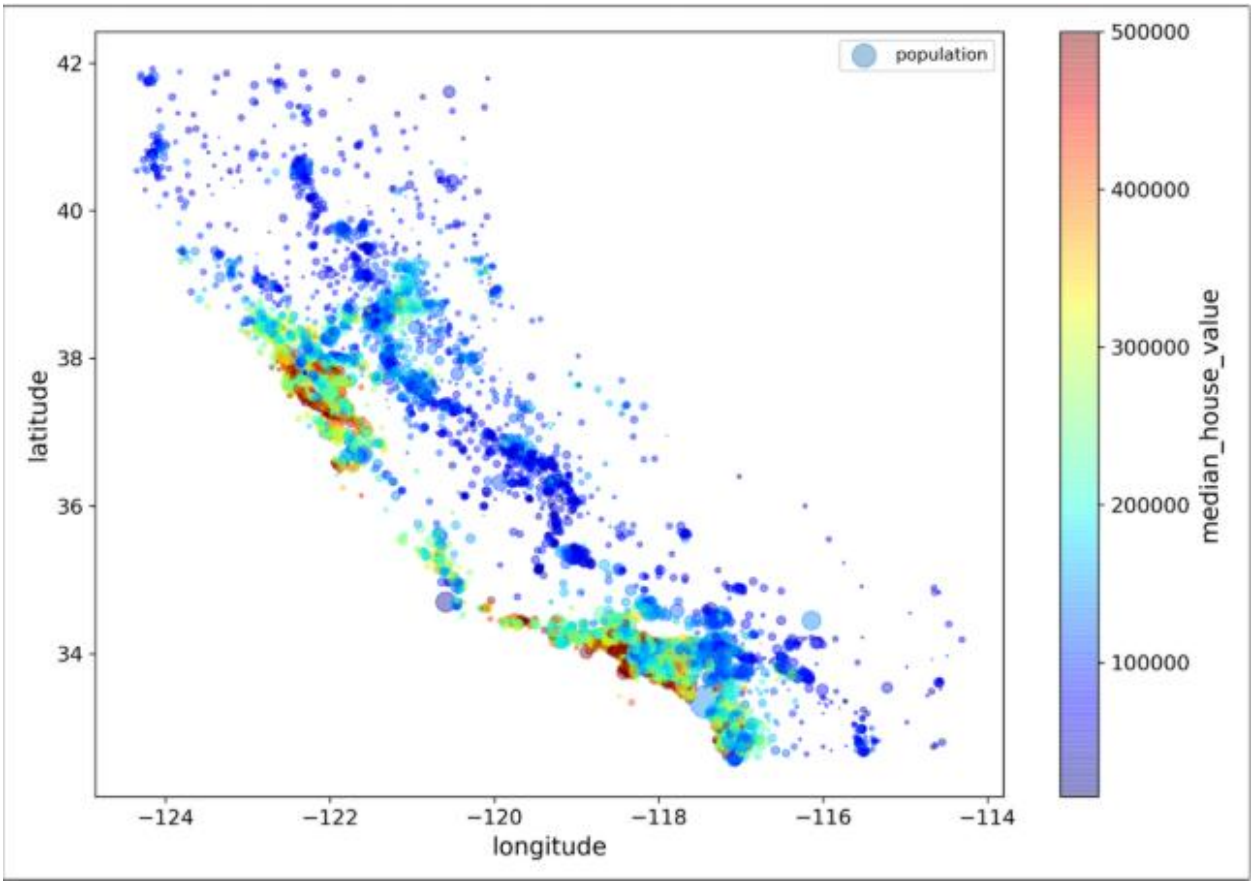

Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indi-cate areas with a larger population

Correlation analysis using corr() identifies **median income** as the strongest predictor, while scatter matrices (**Figure 2-15**) and focused plots (**Figure 2-16**) reveal linear trends and price caps. The chapter also demonstrates how engineered attributes—such as rooms_per_household and bedrooms_per_room—can significantly improve correlations, with the latter showing strong predictive value.

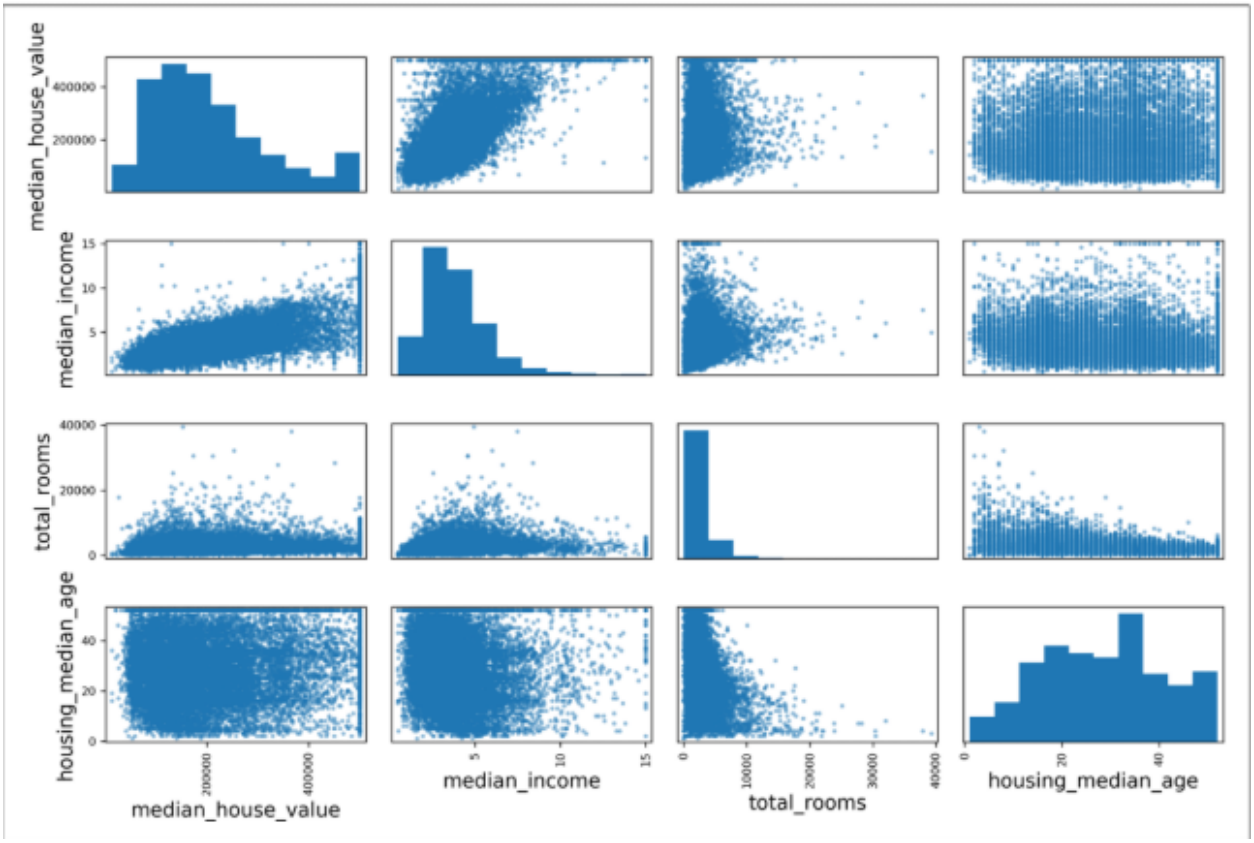

Figure 2-15. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

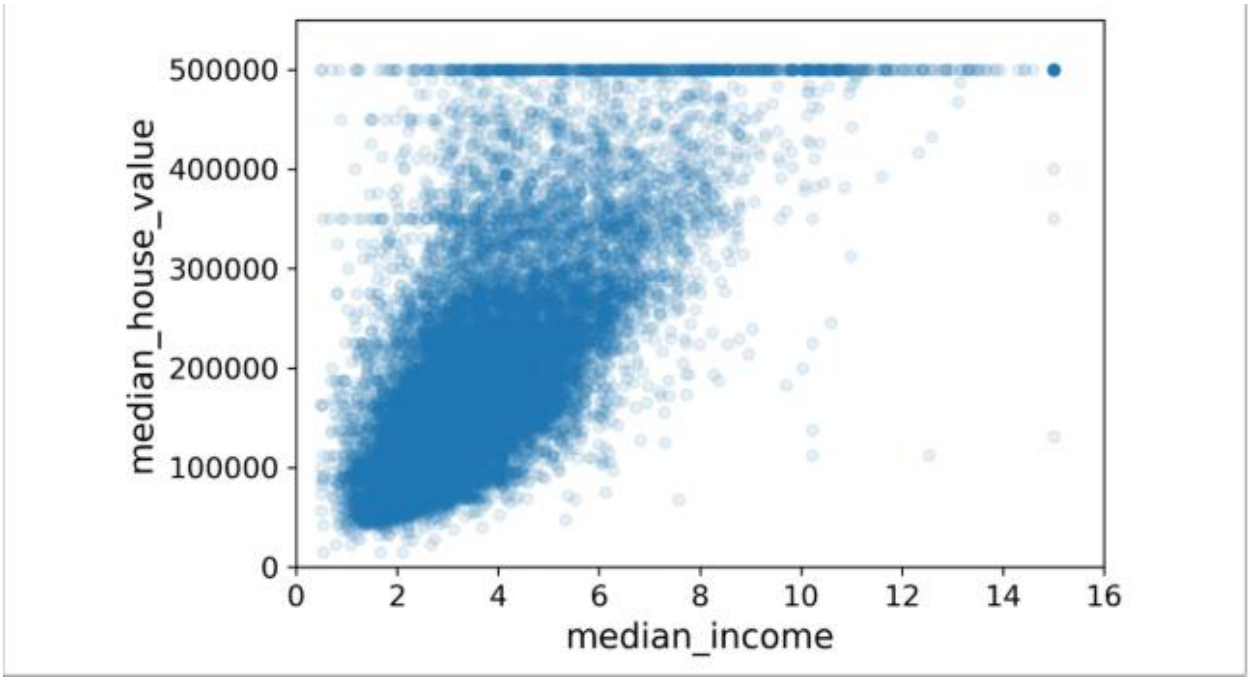

Figure 2-16. Median income versus median house value

Prepare the Data for Machine Learning

Data preparation is formalized into reusable transformation functions. Missing values can be handled by dropping rows, dropping attributes, or imputing values. Scikit-Learn's SimpleImputer is used to apply median imputation safely to numerical fields.

Handling categorical data involves converting text labels using OrdinalEncoder or OneHotEncoder, with sparse matrices used for memory efficiency. The chapter notes that high-cardinality categorical features often require alternative strategies or embeddings.

Custom transformers are introduced via Scikit-Learn's API design principles. An example CombinedAttributesAdder transformer creates engineered ratios automatically.

Feature scaling is crucial; the chapter explains **min-max scaling** and **standardization**, recommending fitting scalers only on the training set to avoid data leakage.

Finally, a unified **ColumnTransformer** with a **Pipeline** is built to automate all preparation steps, producing the fully processed dataset (housing_prepared) ready for modeling.

Select and Train a Model

With the preprocessing pipeline ready, the chapter trains a **Linear Regression** model. Initial predictions show large errors, and RMSE indicates underfitting. A **Decision Tree Regressor** fits the training set perfectly—RMSE = 0—indicating severe overfitting.

To evaluate models properly, **k-fold cross-validation** is used. Cross-validated RMSE reveals that the Decision Tree performs worse than Linear Regression despite memorizing training data. A **Random Forest Regressor** performs substantially better, reducing RMSE significantly, though some overfitting remains.

The chapter stresses evaluating multiple model families before tuning.

Fine-Tune Your Model

Hyperparameter tuning is performed with **GridSearchCV**, which exhaustively evaluates combinations of hyperparameters, using cross-validation to measure performance. The best Random Forest configuration improves RMSE further. The chapter notes that data preparation steps can themselves be treated as hyperparameters.

When search spaces grow large, **RandomizedSearchCV** becomes preferable, offering efficient coverage and fixed computation budgets.

Ensemble methods are also introduced as a strategy to improve performance by combining diverse models.

Model interpretability is touched upon by examining feature importances. In this dataset, **median income** dominates, while certain ocean_proximity categories contribute little.

Evaluate Your System on the Test Set

Once the best model is finalized, it is evaluated on the untouched test set. The pipeline transforms the data using only previously learned parameters, and the final RMSE is computed. A **95% confidence interval** for the generalization error estimates uncertainty in performance.

The chapter warns against tweaking hyperparameters after seeing test results, as this invalidates the test set.

Launch, Monitor, and Maintain Your System

The final step is deployment. The trained pipeline can be serialized using joblib and loaded into production. The model may be embedded directly in an application or served via a **REST API** (**Figure 2-17**). Monitoring is critical: data drift, stale pipelines, or failed components in an ML workflow can degrade performance over time.

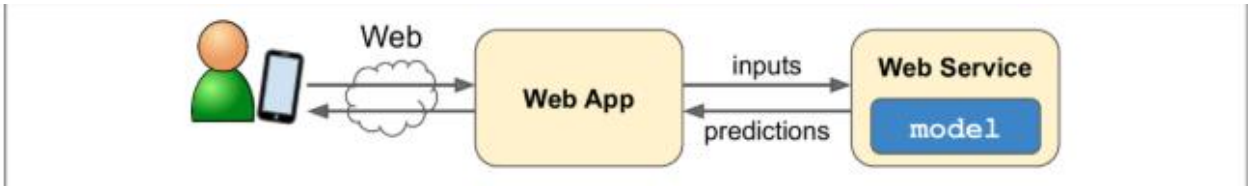

Figure 2-17. A model deployed as a web service and used by a web application

Deployment is not the end but the start of a continuous maintenance cycle involving retraining, monitoring logs, evaluating model decay, and updating the pipeline as data evolves.