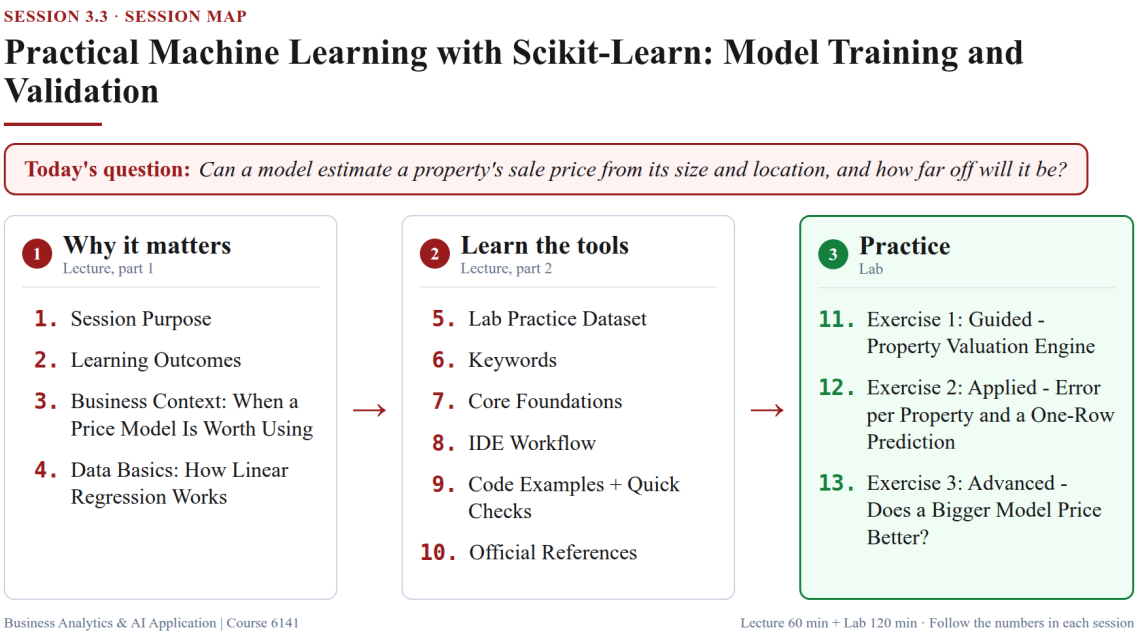
Session 3.3: Practical Machine Learning with Scikit-Learn: Model Training and Validation

Course 6141: Business Analytics & AI Application | Reading for Week 11



Schedule: Week 11 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- A commercial property valuation firm prices office buildings by hand, which takes an expert several days per property
- It has the sale records of 14 recent deals: size, number of rooms, distance to the city center, tax rate, and price
- The valuation manager asks: can a model estimate a property's sale price from its size and location, and how far off will it be? Today, you train your first machine learning model and measure its error honestly

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "Can a model learn prices from past deals?": train a linear regression model with Scikit-Learn's `.fit()` and read its coefficients (how much the price changes for one more unit of each feature)

2. "How far off are the predictions?": measure the error with RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error), and check stability with cross-validation (testing on several different splits)
3. "What will this new building sell for?": predict the price of one new property, and fix the `ValueError` a one-row prediction often raises

3. Business Context: When a Price Model Is Worth Using

- An estimate is useful if its typical error is small compared with the price. Our properties sell for about $513,000 on average, so a typical error of $10,000 (about 2%) is useful for a first quote, while a typical error of $100,000 would not be.
- A model learns from past deals only. It should not be trusted far outside them, for example for a 10,000 square-foot tower when the largest past deal was 3,400 square feet.
- Regression vs classification:
  - **Regression** predicts a number, such as a price. Today's model, linear regression, is one example.
  - **Classification** predicts a category, such as "fraud" or "not fraud". Logistic regression, used in Session 3.4, is one example.

4. Data Basics: How Linear Regression Works

- Linear regression finds the best straight-line formula: `price = intercept + coefficient_1 x feature_1 + coefficient_2 x feature_2 + ...`. "Best" means the smallest total squared error on the training rows.
- Reading a coefficient: the change in price for one more unit of that feature, **while the other features stay the same**. Example from Micro-Example 1: `+180.1` per square foot.
- Coefficients can surprise you. On its own, distance rises with price (correlation `+0.98`), because the big buildings in our data are far from the center. Once size is in the model, each extra mile **lowers** the price by about $1,900: compared at the same size, a farther building is cheaper.
- Three error measures, in plain words:

Measure	Meaning	Lower or higher is better?
MAE (Mean Absolute Error)	the average size of the errors, in dollars	lower
RMSE (Root Mean Squared Error)	like MAE, but large errors count extra	lower
R-squared	the share of the price differences the model explains, from 0 to 1	higher

5. Lab Practice Dataset

- File Link: `data/session_03_03_dataset.csv`
- Audited 14-row commercial property sales dataset (fictional teaching data), one row per sale:
  - `Property_ID`: Sale code (e.g., `PROP-401`).

- `Square_Feet_Units`: Usable floor area in square feet.
- `Bedrooms_Count`: Number of separate rooms or office suites.
- `Distance_To_CBD_Miles`: Distance to the CBD (Central Business District, the city's main business center), in miles.
- `Local_Tax_Rate_Pct`: Local property tax rate, in percent.
- `Sale_Price_USD`: The price the property sold for, in U.S. Dollars. This is the target.

6. Keywords (Core Programming Concepts)

- `Estimator` (`.fit()` / `.predict()`):
  - What: any Scikit-Learn model; `.fit(X, y)` learns from examples, `.predict(X)` makes predictions
  - Why: every model in the library uses the same two methods
- `Linear Regression`:
  - What: a model that predicts a number with a straight-line formula
  - Why: simple, fast, and its coefficients are easy to explain to a manager
- `Logistic Regression`:
  - What: a model that predicts the probability of a yes/no outcome
  - Why: the classification partner of linear regression (Session 3.4)
- `RMSE` and `MAE`:
  - What: the typical prediction error, in the target's own unit (here, dollars)
  - Why: tells the manager how far off an estimate can be
- `Cross-Validation`:
  - What: splitting the data into K parts and testing K times, each time on a different part
  - Why: one lucky split can make a model look better than it is
- `Overfitting`:
  - What: a model that memorizes its training rows and does worse on new data
  - Why: the main risk when data is small; testing on unseen rows reveals it

7. Core Programming & Computational Foundations

- **7.1 The estimator pattern: create, fit, predict**
  - *The idea:* Every model in Scikit-Learn, from the simplest to the most advanced, is used in the same three steps. Learn them once and you can use hundreds of models:
    1. **Create** an empty model: `model = LinearRegression()`.
    2. **Fit** it to examples with known answers: `model.fit(X_train, y_train)`. This is where the model learns.
    3. **Predict** answers for rows it has not seen: `model.predict(X_test)`.

  - *See it on real data:*

Step	Code	What happens
Create	`model = LinearRegression()`	an empty model, nothing learned yet
Fit	`model.fit(X_train, y_train)`	learns the price formula from 10 sales
Predict	`model.predict(X_test)`	4 estimated prices for the 4 test sales
Score	`model.score(X_test, y_test)`	`0.996`, the share of price differences explained

  - *Common mistake:* calling `.predict()` before `.fit()`. The model has nothing learned yet and stops with `NotFittedError`.
  - *More examples:*

Code	Result	What it tells you
`model.intercept_`	`123568.1`	the formula's starting value
`model.coef_`	`[180.1, 14703.6, -1915.8]`	one coefficient per feature, in column order
`model.score(X_train, y_train)`	`0.999`	on the training rows: always looks better
`model.score(X_test, y_test)`	`0.996`	on unseen rows: the honest measure

  - *Remember:* create, fit, predict; learned values end with `_`.

- **7.2 Reading the price formula**
  - *The idea:* Linear regression learns a formula of the form price = intercept + coefficient x feature + … Each coefficient is the change in price for one more unit of that feature **while the other features stay the same**.
  - *See it on real data:* our formula is `123568.1 + 180.1 x square feet + 14703.6 x rooms - 1915.8 x miles`. For a building of 1,500 square feet, 3 rooms, 5 miles from the center:

Part	Calculation	Amount
intercept		`123568.1`
size	`180.1 x 1500`	`270150.0`
rooms	`14703.6 x 3`	`44110.8`
distance	`-1915.8 x 5`	`-9579.0`
price	sum	about `428,250`

    `model.predict()` gives `428,269`: the small difference comes from rounding the coefficients to one decimal in the table.

  - *More examples:* change one feature at a time, starting from that building.

Change	Predicted price	Difference
none (1,500 sq ft, 3 rooms, 5 miles)	`428,269`	
1,000 more square feet	`608,382`	`+180,113`: 1,000 x `180.1`
one more room	`442,973`	`+14,704`
5 more miles from the center	`418,690`	`-9,579`: 5 x `-1915.8`

  - *Common mistake:* reading a coefficient alone. On its own, distance rises with price (correlation `+0.98`), and a model with distance as its only feature even gives it a coefficient of `+30,026.8` per mile, because the big buildings in our data are far from the center. Only when size is in the model does the true effect of distance at the same size appear: `-1,915.8` per mile.
  - *Remember:* a coefficient is the effect of one feature with the others held fixed.

- **7.3 Measuring error: MAE, RMSE, and R-squared**
  - *The idea:* An error is predicted price minus actual price. Three summaries turn the errors into one number each.

Measure	How it is computed	Our test set
MAE (Mean Absolute Error)	average of the error sizes	`$9,668`
RMSE (Root Mean Squared Error)	square each error, average, take the square root	`$11,539`
R-squared	share of the price differences the model explains	`0.996`

  - *See it on real data:* the 4 test errors are `-4,732`, `+15,735`, `+16,064`, and `-2,141` dollars. MAE simply averages their sizes. RMSE squares them first, so the two large errors weigh more, and RMSE comes out higher.
  - *A useful comparison:* a "model" that always guesses the average training price would have a test RMSE of `$186,691`. Ours, at `$11,539`, is about 16 times better.
  - *Common mistake:* trusting R-squared on the training rows (`0.999`). It measures how well the model fits rows it has already seen.
  - *More examples:*

Errors	MAE	RMSE	What it tells you
errors `+10` and `-10`	`10`	`10`	equal errors: MAE and RMSE agree
errors `0` and `20`	`10`	`14.14`	one large error: RMSE grows, MAE does not
our 4 test errors	`9,668`	`11,539`	two larger errors raise RMSE
always guessing the average price		`186,691`	the benchmark any model must beat

- *Remember:* MAE is the plain average error; RMSE punishes large errors; test on unseen rows.

- **7.4 One split is not enough: cross-validation**
  - *The idea:* With only 14 sales, the result depends on which 4 land in the test set. K-fold cross-validation cuts the data into K parts (folds), trains K times, and tests each time on a different fold, so every sale is tested once.
  - *See it on real data:* `cv=5` cuts our 14 sales into 5 folds, in file order.

Fold	Test sales	R-squared
1	`PROP-400`, `PROP-401`, `PROP-402`	`0.994`
2	`PROP-403`, `PROP-404`, `PROP-405`	`0.992`
3	`PROP-406`, `PROP-407`, `PROP-408`	`0.999`
4	`PROP-409`, `PROP-410`, `PROP-411`	`0.996`
5	`PROP-412`, `PROP-413`	`0.999`

  - *More examples:*

Setting	R-squared per fold	What it tells you
`cv=3`	`[0.994, 0.986, 0.998]`	`0.993`
`cv=5` (ours)	`[0.994, 0.992, 0.999, 0.996, 0.999]`	`0.996`
`cv=7`	`[0.982, 0.986, 0.996, 1.0, 0.959, 0.996, 0.999]`	`0.988`: smaller folds, more variation
`cv=KFold(5, shuffle=True, random_state=1)`	`[0.981, 0.999, 0.998, 0.983, 0.969]`	shuffled folds: different sales in each test

  - *Remember:* report the average over the folds, and look at the worst fold too.

- **7.5 One row is still a table: the 2D rule**
  - *The idea:* `.predict()` always expects a table: rows and columns, called 2D (two-dimensional). A single property is a table with one row. A plain list like `[1500, 3, 5.0]` is 1D (one dimension) and is refused.

Input	Shape	Result
`[1500, 3, 5.0]`	1D	`ValueError: Expected 2D array, got 1D array instead`
`[[1500, 3, 5.0]]`	2D, no column names	a price, plus a warning about missing column names
`pd.DataFrame([[1500, 3, 5.0]], columns=features)`	2D, with names	`428,269`, no warning

  - *Common mistake:* predicting far outside the training data. A 10,000 square-foot building gets a "price" of about `$1,959,228`, but our largest sale was 3,400 square feet: the model has never seen such a building, and the number should not be trusted.

– *More examples:*

Code	Result	What it tells you
`np.array([1500, 3, 5.0]).shape`	`(3,)`	1D: three values, no rows
`np.array([1500, 3, 5.0]).reshape(1, -1).shape`	`(1, 3)`	2D: one row, three columns
`model.predict(pd.DataFrame([[1500, 3, 5.0], [2500, 4, 10.0]], columns=features))`	`[428269, 613506]`	two buildings in one call
`model.predict(df[features].iloc[[0]])`	`[361064]`	one existing row; double brackets keep it a table (actual price: `345000`)

– *Remember:* one row = a one-row DataFrame with the same column names.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S33_demo_1.py`), and check the dataset is in `data/`.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors. First time only: `pip install scikit-learn`.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input**, **2. Process**, and **3. Output**. Name each file with your own name and the session code, for example `Reacher_S33_demo_1.py` (pattern: `YourName_S33_demo_1.py`).
- *Micro-Example 1 (Train a Price Model):* `YourName_S33_demo_1.py`

```
# 1. Input: load the sales and split them
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
df = pd.read_csv("data/session_03_03_dataset.csv")
features = ["Square_Feet_Units", "Bedrooms_Count", "Distance_To_CBD_Miles"]
X = df[features]
y = df["Sale_Price_USD"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=42
)
```

```
# 2. Process: learn the price formula from the training rows
model = LinearRegression()
model.fit(X_train, y_train)

# 3. Output: the learned formula
print("Intercept:", round(model.intercept_, 1))
for name, coef in zip(features, model.coef_):
    print(f"{name}: {coef:,.1f}")
```

  - *How it works, line by line:*
    - `features = [...]` lists the 3 input columns; `X = df[features]` and `y = df["Sale_Price_USD"]` separate inputs and answer (Session 3.2).
    - `train_test_split(..., test_size=0.25, random_state=42)` keeps 4 of the 14 sales aside for testing.
    - `model = LinearRegression()` creates an empty model; `model.fit(X_train, y_train)` learns the formula from the 10 training sales (section 7.1).
    - `zip(features, model.coef_)` walks through the feature names and their coefficients side by side, so each line prints a name with its number.
    - `{coef:,.1f}` prints the number with thousands separators and 1 decimal.
  - *Read the output:*

```
Intercept: 123568.1
Square_Feet_Units: 180.1
Bedrooms_Count: 14,703.6
Distance_To_CBD_Miles: -1,915.8
```

    - Each extra square foot adds about `$180`, each extra room about `$14,700`, and each extra mile from the center subtracts about `$1,900`, with the other features held fixed (section 7.2).
    - The intercept is the formula's starting point. On its own it has no business meaning: no building has 0 square feet.

- *Quick Check 1 (`.score()`):*
  - New tool: `model.score(X_test, y_test)` returns R-squared on the test rows in one call. Why: a quick first check of how well the model explains prices it has not seen.
  - Task: add one line to part **3. Output**: `print("Test R-squared:", round(model._____(X_test, y_test), 3))`
  - Check: the Terminal shows `Test R-squared: 0.996`.

- *Micro-Example 2 (Predict the Test Sales and Measure the Error):* `YourName_S33_demo_2.py`

```
# 1. Input: load, split, and train (same as Micro-Example 1)
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

```
df = pd.read_csv("data/session_03_03_dataset.csv")
features = ["Square_Feet_Units", "Bedrooms_Count", "Distance_To_CBD_Miles"]
X_train, X_test, y_train, y_test = train_test_split(
    df[features], df["Sale_Price_USD"], test_size=0.25, random_state=42
)
model = LinearRegression().fit(X_train, y_train)

# 2. Process: predict the 4 test sales and compute the RMSE
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
comparison = pd.DataFrame({
    "Property_ID": df.loc[X_test.index, "Property_ID"],
    "Actual_USD": y_test,
    "Predicted_USD": y_pred.round(0),
})

# 3. Output: actual vs predicted, then the typical error
print(comparison)
print(f"Test RMSE: ${rmse:,.0f}")
```

  - *How it works, line by line:*
    - Part 1 loads, splits, and trains, as in Micro-Example 1. `LinearRegression().fit(...)` creates and fits in one line.
    - `model.predict(X_test)` returns 4 estimated prices.
    - `mean_squared_error()` averages the squared errors; `np.sqrt()` (square root, from NumPy) brings the result back to dollars: the RMSE (section 7.3).
    - `pd.DataFrame({...})` builds a comparison table. `df.loc[X_test.index, "Property_ID"]` finds the IDs of the test rows, because the split keeps the original row labels.
  - *Read the output:*

```
   Property_ID  Actual_USD  Predicted_USD
9     PROP-409    465000.0       460268.0
11    PROP-411    330000.0       345735.0
0     PROP-400    345000.0       361064.0
12    PROP-412    780000.0       777859.0
Test RMSE: $11,539
```

    - The model is close on the large building (`PROP-412`: off by `$2,141`) and on `PROP-409`.
    - It overprices the two small buildings, `PROP-411` and `PROP-400`, by about `$16,000` each: small buildings deserve a manual check.
    - A typical error of about `$11,500` on prices averaging `$513,000` is about 2%: good enough for a first quote.

- *Quick Check 2 (`mean_absolute_error()`):*
  - New tool: `mean_absolute_error(y_test, y_pred)` returns the average error size in dollars. Why: the easiest error number to explain: "on average, we are off by this much".

  - Task: change the import line to `from sklearn.metrics import mean_squared_error, mean_absolute_error`, then add one line to part **3. Output**: `print(f"Test MAE: ${_____(y_test, y_pred):,.0f}")`
  - Check: the Terminal shows `Test MAE: $9,668`, lower than the RMSE because RMSE weighs the two larger errors (about $16,000 each) more heavily.

- *Micro-Example 3 (Five Splits Instead of One):* `YourName_S33_demo_3.py`

```
# 1. Input: load the sales
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
df = pd.read_csv("data/session_03_03_dataset.csv")
features = ["Square_Feet_Units", "Bedrooms_Count", "Distance_To_CBD_Miles"]

# 2. Process: train and test 5 times, each time on a different part
scores = cross_val_score(LinearRegression(), df[features],
                         df["Sale_Price_USD"], cv=5, scoring="r2")

# 3. Output: the 5 scores and their average
print("R-squared per fold:", scores.round(3).tolist())
print(f"Average R-squared: {scores.mean():.3f}")
```

  - *How it works, line by line:*
    - `cross_val_score(LinearRegression(), X, y, cv=5, scoring="r2")` does all the work of section 7.4: it cuts the 14 sales into 5 folds, trains 5 fresh models, and tests each one on its own fold.
    - It returns 5 scores, one per fold. `.round(3).tolist()` prints them as a plain list.
    - `scores.mean()` averages them.
  - *Read the output:*

```
R-squared per fold: [0.994, 0.992, 0.999, 0.996, 0.999]
Average R-squared: 0.996
```

    - Every fold scores above `0.99`: the model does well no matter which sales are held out, not just on one lucky split.
    - The average, `0.996`, equals the single split's score from Quick Check 1, which confirms that result.

- *Quick Check 3 (RMSE per fold):*
  - New tool: `scoring="neg_root_mean_squared_error"` makes `cross_val_score()` return the RMSE of each fold as a negative number (Scikit-Learn treats higher scores as better), so put a minus sign in front to read it. Why: shows the error in dollars for every fold.
  - Task: add two lines to part **3. Output**: `rmse = -cross_val_score(LinearRegression(), df[features], df["Sale_Price_USD"], cv=5, scoring="_____")` and `print(rmse.round(0).tolist())`
  - Check: `[7862.0, 15224.0, 4004.0, 8201.0, 3374.0]`. The error ranges from about $3,400 to $15,200 depending on which properties are tested.

10. Official Python Documentation & References

- Scikit-Learn: Linear Models (LinearRegression Documentation).
- Scikit-Learn: Cross-Validation Metrics and Scoring.
- Scikit-Learn: Regression Metrics (MAE, RMSE, R-squared).
- Gareth James et al.: An Introduction to Statistical Learning (ISLR with Applications in Python).

11. Exercise 1: Guided Practice - Property Valuation Engine

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S33_regression.py`, replacing `YourName` with your own name (for example, `Reacher_S33_regression.py`).
- *Step 2: Train, Predict, and Evaluate:* Type the following code:

```
# 1. Input: load the sales and split them
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
df = pd.read_csv("data/session_03_03_dataset.csv")
features = ["Square_Feet_Units", "Bedrooms_Count", "Distance_To_CBD_Miles"]
X_train, X_test, y_train, y_test = train_test_split(
    df[features], df["Sale_Price_USD"], test_size=0.25, random_state=42
)

# 2. Process: train, then predict the test sales
reg = LinearRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

# 3. Output: the formula and the test error
print("Coefficients:", dict(zip(features, reg.coef_.round(1).tolist())))
print(f"Test RMSE: ${rmse:,.0f} | R-squared: {r2:.3f}")
```

- *Step 3: Interpret for the Manager:* In three comment lines, explain each coefficient in plain words, using "while the other features stay the same".
- *Step 4: Execute in Terminal:* Run `python YourName_S33_regression.py` in Panel `[4]` and confirm return code `0`.
- *Success Check:* coefficients `180.1` (square feet), `14703.6` (rooms), and `-1915.8` (distance); `Test RMSE: $11,539 | R-squared: 0.996`. Each extra mile from the center lowers the price when size and rooms stay the same.

12. Exercise 2: Applied Practice - Error per Property and a One-Row Prediction

- *Coding Challenge:* In `YourName_S33_practice.py`, extend Exercise 1 to show the error of each test property in percent, then price one new building.
- *Task 1 (Percentage Error per Property):* Build a table with `Property_ID`, `Actual_USD`, `Predicted_USD` (rounded to whole dollars), and `Error_Pct` = absolute error / actual price x 100, rounded to 2 decimals. Use `abs(y_test - y_pred)` for the absolute error. Then print the average `Error_Pct`, called MAPE (Mean Absolute Percentage Error).
- *Task 2 (Read the ValueError & AI Diagnosis):* Predict a new 1,500 square-foot property with 3 rooms, 5.0 miles from the center, by writing `reg.predict([1500, 3, 5.0])`. The script stops with `ValueError: Expected 2D array, got 1D array instead`. Capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and ask what "2D" means.
- *Task 3 (Fix the Shape):* Wrap the new property in a one-row DataFrame with the same column names: `new_property = pd.DataFrame([[1500, 3, 5.0]], columns=features)`, then predict and print the price with thousands separators.
- *Success Check:*
  - Error per property: `PROP-409` `1.02%`, `PROP-411` `4.77%`, `PROP-400` `4.66%`, `PROP-412` `0.27%`; MAPE `2.68%`.
  - New property: about `$428,269`. The two smaller properties have the largest percentage errors, so the firm should check small-building quotes by hand.

13. Exercise 3: Advanced Practice - Does a Bigger Model Price Better?

- *Goal:* The firm could collect more features for every quote, but collecting data costs money. Find out, with 5-fold cross-validation, whether extra features really reduce the pricing error.
- *What you need:*
  - File: `data/session_03_03_dataset.csv` (described in **5. Lab Practice Dataset**).
  - Script name: `YourName_S33_challenge.py`.
  - Tools you already know: `LinearRegression`, `cross_val_score()` with `scoring="neg_root_mean_squared_error"` (Quick Check 3), dictionaries, `for` loops, and `pd.DataFrame()`.
- *Requirements (follow them in order):*
  1. Define four candidate feature lists in a dictionary:
     - `"A: size only": ["Square_Feet_Units"]`
     - `"B: size + distance": ["Square_Feet_Units", "Distance_To_CBD_Miles"]`
     - `"C: size + rooms + distance": ["Square_Feet_Units", "Bedrooms_Count", "Distance_To_CBD_Miles"]`
     - `"D: all four"`: the three above plus `"Local_Tax_Rate_Pct"`
  2. For each candidate, run 5-fold cross-validation with `cv=5` and RMSE scoring, and record the average RMSE and the largest fold RMSE, both rounded to whole dollars, plus the number of features.
  3. Collect the results in one DataFrame with the columns `Model`, `Features`, `Avg_RMSE_USD`, and `Worst_Fold_RMSE_USD`, sorted by `Avg_RMSE_USD` from lowest to highest.

4. Apply the firm's rule: choose the model with the lowest average RMSE, **unless** a model with fewer features is within `$500` of it; then choose the model with the fewest features among those within `$500`.
5. Print the table and the chosen model with one sentence that explains the choice.

- *New tools for this exercise:*
  - Looping over a dictionary: `for name, cols in candidates.items():` gives each key (`name`) and its value (`cols`) in turn.
  - Collecting rows: start with `rows = []`, append one dictionary per model, such as `rows.append( {"Model": name, "Features": len(cols), ...})`, and finish with `pd.DataFrame(rows)`.
  - `.max()` on the fold results gives the worst fold.
- *Hints:*
  - First print the 5 fold values of each model before you average them.
  - For the rule, first keep the rows with `Avg_RMSE_USD <= best + 500`, then sort them by `Features`.
- *Self-Check (expected results):*
  - Average RMSE: C `$7,733`, B `$7,837`, A `$7,957`, D `$9,296`. Worst folds: C `$15,224`, B `$10,415`, A `$10,279`, D `$15,463`.
  - Within `$500` of the best (`$7,733`): C, B, and A. Chosen: **A: size only**.
  - Business meaning: in these 14 deals, adding rooms and distance cut the average error by only about $224, while adding the tax rate made it worse by about $1,563. The firm can quote from floor area alone and spend its data budget on collecting more deals instead.