Reading this book has been a really good experience, and honestly, I've fallen in love with reading. It saves time and delivers knowledge in a much more thoughtful way. Even though I was already familiar with most of the machine learning concepts covered in the first two chapters, there's always something new to pick up. So, I made sure to take notes on anything that felt new or especially important to me.

Chapter 01

Supervised, Unsupervised, Semi-Supervised & Reinforcement Learning
These are the foundational categories in ML:

- **Supervised Learning** works with labeled data, great for tasks like classification and regression.
- **Unsupervised Learning** finds patterns in unlabeled data, such as clustering or dimensionality reduction.
- **Semi-Supervised Learning** mixes a small amount of labeled data with a large pool of unlabeled data—common when labeling is expensive.
- **Reinforcement Learning** is all about decision-making through rewards and penalties—used in games, robotics, etc.
  Familiar concepts, but it's always helpful to revisit and see examples from new angles.

Online vs Batch Learning
This helped clear some gray areas for me.

- **Batch Learning** involves training the model using the entire dataset at once. It's stable but not suitable for systems where data constantly changes.
- **Online Learning** is incremental. The model gets updated with each new data point or small batch, making it ideal for real-time environments.
  It also highlighted issues like **catastrophic forgetting** (when learning rate is too high) and slow adaptation (when it's too low). Use cases like stock prediction or live recommendation systems make online learning super relevant.

Instance-Based vs Model-Based Learning
This comparison really clicked for me.

- **Instance-Based Learning** doesn't form a general model—it just stores training examples and uses distance-based methods (like k-NN) to make predictions. It's simple but can be computationally expensive at prediction time.
- **Model-Based Learning** creates a general model based on the training data. It tries to capture the underlying patterns—like using linear regression to learn weights and intercepts.
  Big picture: instance-based learning is like memorizing, model-based learning is like understanding.

Anomaly Detection vs Novelty Detection

- **Anomaly Detection** deals with spotting unusual patterns *within* the training data (e.g., detecting fraudulent transactions).
- **Novelty Detection** is about identifying new, unseen patterns during *testing* that weren't present during training.
  The difference is subtle but crucial when designing systems for security or monitoring.

PCA / Dimensionality Reduction
PCA reduces the number of features by transforming data into a new set of components that capture the most variance. It helps in simplifying models, reducing overfitting, and visualizing high-dimensional data.
I've used PCA before, so this part just served as a solid refresher.

Challenges in ML – Bad Model or Bad Data?
The section emphasized that many ML failures are rooted in poor data rather than poor algorithms. Models trained on noisy, irrelevant, or insufficient data won't perform well regardless of their complexity.
Key lesson: focus on clean, meaningful data before blaming the model.

Sampling Bias

Sampling bias happens when the training data doesn't accurately represent the real-world population. For example, if a model is trained mostly on data from urban users, it may fail in rural scenarios.
Important reminder: the training data must mirror the deployment environment.

Data Mismatch
This was something that really stood out. Data mismatch occurs when the distribution of the training data differs from that of the test or real-world data. Even if your model performs perfectly in training and validation, it might struggle in production due to unseen shifts.
For instance, imagine training a medical diagnostic model using hospital data from Karachi, but then deploying it in rural areas with different population characteristics. The model's assumptions about features, correlations, or patterns might no longer hold true.
To mitigate this, it's essential to monitor live performance, use domain adaptation techniques, or continuously retrain the model with updated data from the actual target environment.

Exercise Questions

How would you define Machine Learning?
For me, ML is like teaching a system to **learn patterns** from data without being explicitly programmed for every single rule. It's about making computers smart enough to make predictions or decisions based on past examples. It feels less like coding instructions and more like building logic that evolves through data.

Can you name four types of problems where it shines?
Yeah definitely, ML shines when:

- The logic is too complex to hand-code (e.g., facial recognition).
- There's a lot of messy data and we need structure (e.g., customer segmentation).
- Patterns change over time (e.g., stock market predictions).
- Real-time decision-making is needed (e.g., self-driving cars reacting to roads).

What is a labeled training set?
It's basically the foundation for supervised learning. A **labeled training set** has both the input data and the correct answers (labels). For example, images of handwritten digits along with the actual digits (0–9) written on them. You feed this to the model so it can learn the connection.

What are the two most common supervised tasks?
Classification and regression.

- **Classification**: when the output is a category (e.g., spam or not).

- **Regression**: when the output is continuous (e.g., predicting house prices).

Can you name four common unsupervised tasks?
Yeah, these are the ones I see often:

- **Clustering** (grouping similar data, like customers).
- **Anomaly detection** (finding outliers or frauds).
- **Dimensionality reduction** (PCA etc. for simplifying data).
- **Association rule learning** (like market basket analysis).

What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?
That sounds like a job for **reinforcement learning**. The robot needs to learn by trial and error—like getting rewards for walking steadily and penalties for falling. It's not about labeled data, it's about learning through interaction with the environment.

What type of algorithm would you use to segment your customers into multiple groups?
Unsupervised learning, specifically **clustering algorithms** like K-Means. We don't have labeled data telling us which group each customer belongs to, so the model finds patterns and naturally occurring clusters based on behaviors.

Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?
Definitely **supervised**—we usually have a dataset of emails labeled as spam or not spam. The model learns from those examples and can then predict for new emails. If we didn't have labels, then anomaly or novelty detection might be another way.

What is an online learning system?
It's a model that **learns continuously** as new data comes in. Rather than retraining from scratch, it updates its knowledge incrementally. Great for scenarios where data is arriving in streams, like real-time social media feeds or stock prices.

What is out-of-core learning?
When your dataset is **too big to fit in memory**, you use out-of-core learning. It processes the data in mini-batches that can be loaded into memory. Libraries like `scikit-learn` offer support through things like `partial_fit()`.

What type of learning algorithm relies on a similarity measure to make predictions?
That's **instance-based learning**, like k-Nearest Neighbors (k-NN). It doesn't build a model in the traditional sense—it just compares the input with stored examples using a similarity metric (like Euclidean distance) and makes predictions based on the closest ones.

What is the difference between a model parameter and a learning algorithm's hyperparameter?

- **Model parameters** are what the algorithm learns from the data (like weights in linear regression).
- **Hyperparameters** are what *we* set before training (like learning rate, number of neighbors in k-NN). They guide the training process but aren't learned from data.

What do model-based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?

They search for the **best model parameters** that minimize some **loss function**. The most common strategy is optimization—like using **gradient descent** to minimize error. Once trained, the model uses its learned parameters to make predictions on new data by applying the same learned function.

Can you name four of the main challenges in Machine Learning?
Yep:

- **Bad data**: noisy, missing, or irrelevant features.
- **Overfitting/Underfitting**: too complex or too simple models.
- **Sampling bias**: training data isn't representative.
- **Data mismatch**: training and test distributions are different.

If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
That's classic **overfitting**. The model memorized the training data instead of learning general patterns. Solutions:

- Simplify the model (reduce complexity).
- Use regularization (like L1/L2 penalties).
- Get more training data or apply data augmentation.

What is a test set and why would you want to use it?
The **test set** is like a final exam for the model. It's unseen data that checks how well the model generalizes. It helps us understand if our model will actually work in the real world, not just on the data it was trained on.

What is the purpose of a validation set?
Validation set helps us **tune hyperparameters** and avoid overfitting to the training data. It gives us a checkpoint during model building—telling us how it's doing on unseen-but-not-final data. It's like getting feedback before the final test.

What can go wrong if you tune hyperparameters using the test set?
Then your test set becomes part of the training loop, which means you've "seen" the answers during practice. It leads to **data leakage** and gives an **overly optimistic** view of model performance. The final evaluation will be misleading.

What is repeated cross-validation and why would you prefer it to using a single validation set?

Repeated cross-validation is like doing **k-fold cross-validation multiple times** with different data splits. Each time, the data gets shuffled differently, and the model's performance is evaluated across those splits. The results are then averaged for a more stable performance estimate.

I'd prefer this over using just one validation set because **it reduces random variance**. A single validation set might give a misleading result, especially if the data split isn't great. Repeating the process multiple times helps in getting a **more reliable and consistent estimate** of how the model will perform in real-world scenarios. It also helps in spotting overfitting more easily.

Chapter 02

Pipelines

Pipelines in machine learning help streamline and organize the entire workflow, from data preprocessing to model training. The idea is to **chain together multiple steps** such as scaling, encoding, and model fitting into one unified process. By using a pipeline, you ensure that the exact same transformations applied to the training data are also applied to the test data, preventing data leakage. I find this incredibly useful because it makes the code cleaner and **saves time**. Whenever I work on a project, I know that having all steps in one pipeline means **fewer errors** and a much smoother process. Plus, it's easy to swap out different models or preprocessing steps without disturbing the whole setup.

Data Snooping Bias

Data snooping bias is when you make model decisions based on the same data you use to evaluate it, essentially "peeking" at the test data before the model is finalized. This results in overly optimistic performance metrics because the model has already seen parts of the test data or used it to tune parameters. I've learned the hard way that **sticking to proper cross-validation** is crucial here. If you over-analyze the same data too much, you start building models that look great on paper but perform terribly when you try them on unseen data. That's why I focus on **keeping test data separate** and use techniques like cross-validation to get a more **honest assessment** of how my model is likely to perform in the real world.

Stratified Sampling / pd.cut

Stratified sampling ensures that each subset of the data (such as training and test sets) has the same proportion of class labels as the original dataset. This is especially important for imbalanced datasets, where one class might dominate the others. For example, if you have a binary classification problem and 90% of the data belongs to class A, stratified sampling will ensure that both the training and test sets have a similar **90-10 ratio**. As for `pd.cut`, it's a function that helps **bin continuous data** into discrete intervals, useful for when you want to categorize continuous features like age into groups (e.g., 0-18, 19-35, etc.). I've used **stratified sampling** multiple times, and it's been key for getting models that perform well on **imbalanced datasets**. It's also handy for **grouping data into meaningful categories** using `pd.cut` when continuous variables don't need to be treated as continuous anymore.

Scikit-learn Design Principles

Scikit-learn was designed with a few guiding principles that make it incredibly easy to use. It emphasizes **consistency** in its interface, so whether you're using a classifier, regressor, or transformer, the methods are pretty much the same. It's also modular, meaning components like preprocessors, models, and feature selectors can easily be combined into pipelines. This design allows for great **flexibility and simplicity**. I find scikit-learn's principles really helpful because once you understand the general framework, it feels like you can quickly jump between different models and preprocessing techniques. Its **clean API** and **predictable behavior** make experimentation a lot more efficient, and it feels intuitive even for beginners.

Custom Transformers, BaseEstimator, TransformerMixin

Creating **custom transformers** in scikit-learn is possible by subclassing `BaseEstimator` and `TransformerMixin`. The `BaseEstimator` gives you easy access to setting and getting parameters, while `TransformerMixin` adds the `fit_transform()` method, so you don't have to write it yourself. Custom transformers are useful when you need to implement **domain-specific transformations** that aren't covered by the built-in ones. After reading about custom transformers I built a custom transformer to convert **date columns into the number of days since a particular date**. I really appreciate this feature because it lets me

integrate custom logic into my pipelines without having to manually handle the transformations. This ability to extend scikit-learn's functionality makes it a powerful tool for specialized tasks.

K-Fold Cross-Validation

K-Fold cross-validation is a method used to assess how well a model will perform on unseen data by splitting the data into K equal-sized parts. The model is trained K times, each time leaving out a different fold as the test set and using the remaining folds as training data. The performance is then averaged over all the folds, which gives a more **reliable estimate** of model performance. I find this method especially useful because it ensures that every data point gets a chance to be in both the training and test sets. It also helps prevent issues like **overfitting**, as the model is constantly tested on different subsets of the data.

GridSearchCV, RandomizedSearchCV + Checking Importance

GridSearchCV is a method where you exhaustively search through all possible combinations of hyperparameters, while **RandomizedSearchCV** randomly selects combinations from a defined hyperparameter space. The main difference is that GridSearch is more exhaustive but computationally expensive, while RandomizedSearch is faster and less prone to overfitting when the search space is large. After training, **checking feature importance** (especially in tree-based models) helps identify which features are **most responsible** for the model's predictions. I usually prefer **RandomizedSearchCV** because it's faster and still gives pretty good results, especially when I'm dealing with a **large number of hyperparameters**. Checking feature importance helps me **interpret models better**, understanding why they make certain predictions.

"Don't Fall Into the Trap of Fine-Tuning for Just Better Numbers at Training"

It's easy to get obsessed with improving training scores, but that can lead to **overfitting**. Fine-tuning hyperparameters just for better training performance might make the model perform well on the training set but **fail to generalize** to new data. Instead, I focus on tuning the model for **generalizability**, aiming to improve performance on validation sets and considering other metrics like **cross-validation** scores rather than just minimizing the training error. The key takeaway here is that **better numbers on training data don't always translate to real-world performance**, and you should be cautious not to overfit the model by pushing for perfect results on the training data at the expense of its ability to handle unseen data.

Exercise Questions

Question: Try a Support Vector Machine regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best `SVR` predictor perform?

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR

svr_parameters = [
    {
        'kernel': ['linear'],
        'C': [0.1, 1, 10, 100, 1000]  # kept smaller range for testing simplicity
    },
```

```
    {
        'kernel': ['rbf'],
        'C': [0.1, 1, 10, 100],
        'gamma': [0.001, 0.01, 0.1, 1]
    }
]

svr_model = SVR()

svr_search = GridSearchCV(
    estimator=svr_model,
    param_grid=svr_parameters,
    scoring='neg_mean_squared_error',
    cv=5,
    verbose=2
)

# Fit it on the training data
svr_search.fit(housing_prepared, housing_labels)
```

After running this grid search, the best model was selected based on **lowest negative mean squared error** (which essentially means best prediction performance).

Answer to the original question:

> The best SVR model (in my case) used the `'rbf'` kernel with specific values of `C` and `gamma`. It performed reasonably well, though not necessarily better than other simpler models like `RandomForestRegressor` or `LinearRegression` depending on the dataset. What I observed is that SVR tends to be **more sensitive to feature scaling** and might need some extra fine-tuning.

Alright Rameez, here's how I'd explain it for your project report—keeping that **humanized + slightly professional** tone:

Question 2: Switching from GridSearchCV to RandomizedSearchCV

So, the task here was to **replace `GridSearchCV` with `RandomizedSearchCV`**, which is actually a smart move when you've got a *larger hyperparameter space* and want to save some time.

While `GridSearchCV` tries **every possible combination** in your param grid (which can be super slow when the grid is big), `RandomizedSearchCV` just samples **a fixed number of random combinations** and tests those instead. You basically tell it, "*Hey, try 20 random combos from this list*" and it gets to work.

It's more efficient, and still gives pretty good results—especially when you're just trying to get a solid model fast.

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVR
from scipy.stats import uniform

svr_model = SVR()
```

```
param_distributions = {
    'kernel': ['linear', 'rbf'],
    'C': uniform(1, 1000),
    'gamma': uniform(0.001, 1)       # Only used when kernel is 'rbf'
}

random_search = RandomizedSearchCV(
    estimator=svr_model,
    param_distributions=param_distributions,
    n_iter=20,
    scoring='neg_mean_squared_error',
    cv=5,
    verbose=2,
    random_state=42
)

random_search.fit(housing_prepared, housing_labels)
```

Using `RandomizedSearchCV` was **way faster** than Grid Search, especially with more values in the hyperparameter space. While it may not *guarantee* the best combination (since it's not testing all of them), it gave me **very close results** to what GridSearchCV gave earlier.

Output:

```
[CV]  C=629.782329591372, gamma=3.010121430917521, kernel=linear, total=   4.0s
[CV] C=629.782329591372, gamma=3.010121430917521, kernel=linear ......
[CV]  C=629.782329591372, gamma=3.010121430917521, kernel=linear, total=   4.5s
[CV] C=629.782329591372, gamma=3.010121430917521, kernel=linear ......
[CV]  C=629.782329591372, gamma=3.010121430917521, kernel=linear, total=   4.5s
[CV] C=629.782329591372, gamma=3.010121430917521, kernel=linear ......
[CV]  C=629.782329591372, gamma=3.010121430917521, kernel=linear, total=   4.3s
[CV] C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf ......
[CV]  C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf, total=   8.6s
[CV] C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf ......
[CV]  C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf, total=   9.1s
[CV] C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf ......
[CV]  C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf, total=   8.8s
[CV] C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf ......
[CV]  C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf, total=   8.9s
[CV] C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf ......
[CV]  C=26290.206464300216, gamma=0.9084469696321253, kernel=rbf, total=   9.0s
[CV] C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf .....
[CV]  C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf, total=   7.0s
[CV] C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf .....
[CV]  C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf, total=   7.0s
[CV] C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf .....
[CV]  C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf, total=   6.9s
[CV] C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf .....
[CV]  C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf, total=   7.0s
[CV] C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf .....
[CV]  C=84.14107900575871, gamma=0.059838768608680676, kernel=rbf, total=   7.0s
```

```
[CV] C=432.37884813148855, gamma=0.15416196746656105, kernel=linear ..
[CV]  C=432.37884813148855, gamma=0.15416196746656105, kernel=linear, total=
4.6s
<<434 more lines>>
[CV] C=61217.04421344494, gamma=1.6279689407405564, kernel=rbf .......
[CV]  C=61217.04421344494, gamma=1.6279689407405564, kernel=rbf, total=  25.2s
[CV] C=61217.04421344494, gamma=1.6279689407405564, kernel=rbf .......
[CV]  C=61217.04421344494, gamma=1.6279689407405564, kernel=rbf, total=  23.2s
[CV] C=926.9787684096649, gamma=2.147979593060577, kernel=rbf ........
[CV]  C=926.9787684096649, gamma=2.147979593060577, kernel=rbf, total=   5.7s
[CV] C=926.9787684096649, gamma=2.147979593060577, kernel=rbf ........
[CV]  C=926.9787684096649, gamma=2.147979593060577, kernel=rbf, total=   5.7s
[CV] C=926.9787684096649, gamma=2.147979593060577, kernel=rbf ........
[CV]  C=926.9787684096649, gamma=2.147979593060577, kernel=rbf, total=   5.7s
[CV] C=926.9787684096649, gamma=2.147979593060577, kernel=rbf ........
[CV]  C=926.9787684096649, gamma=2.147979593060577, kernel=rbf, total=   5.8s
[CV] C=926.9787684096649, gamma=2.147979593060577, kernel=rbf ........
[CV]  C=926.9787684096649, gamma=2.147979593060577, kernel=rbf, total=   5.6s
[CV] C=33946.157064934, gamma=2.2642426492862313, kernel=linear ......
[CV]  C=33946.157064934, gamma=2.2642426492862313, kernel=linear, total=  10.0s
[CV] C=33946.157064934, gamma=2.2642426492862313, kernel=linear ......
[CV]  C=33946.157064934, gamma=2.2642426492862313, kernel=linear, total=   9.7s
[CV] C=33946.157064934, gamma=2.2642426492862313, kernel=linear ......
[CV]  C=33946.157064934, gamma=2.2642426492862313, kernel=linear, total=   8.9s
[CV] C=33946.157064934, gamma=2.2642426492862313, kernel=linear ......
[CV]  C=33946.157064934, gamma=2.2642426492862313, kernel=linear, total=  10.4s
[CV] C=33946.157064934, gamma=2.2642426492862313, kernel=linear ......
[CV]  C=33946.157064934, gamma=2.2642426492862313, kernel=linear, total=   9.3s
[CV] C=84789.82947739525, gamma=0.3176359085304841, kernel=linear ....
[CV]  C=84789.82947739525, gamma=0.3176359085304841, kernel=linear, total=  25.8s
[CV] C=84789.82947739525, gamma=0.3176359085304841, kernel=linear ....
[CV]  C=84789.82947739525, gamma=0.3176359085304841, kernel=linear, total=  18.5s
[CV] C=84789.82947739525, gamma=0.3176359085304841, kernel=linear ....
[CV]  C=84789.82947739525, gamma=0.3176359085304841, kernel=linear, total=  28.3s
[CV] C=84789.82947739525, gamma=0.3176359085304841, kernel=linear ....
[CV]  C=84789.82947739525, gamma=0.3176359085304841, kernel=linear, total=  20.8s
[CV] C=84789.82947739525, gamma=0.3176359085304841, kernel=linear ....
[CV]  C=84789.82947739525, gamma=0.3176359085304841, kernel=linear, total=  15.6s
```

Question: Adding a Transformer to Select Most Important Features

This helps reduce noise in the data, potentially **improving model performance** and **reducing training time**. It's especially useful when you've got lots of features and not all of them are pulling their weight.

So I created this transformer, this takes in the data and selects the top-k features based on their importance scores.

```
from sklearn.base import BaseEstimator, TransformerMixin
import numpy as np

class TopFeatureSelector(BaseEstimator, TransformerMixin):
```

```
    def __init__(self, feature_importances, k):
        self.feature_importances = feature_importances
        self.k = k

    def fit(self, X, y=None):
        self.feature_indices_ = np.sort(
            np.argsort(self.feature_importances)[-self.k:]
        )
        return self

    def transform(self, X):
        return X[:, self.feature_indices_]
```

Using It in the Pipeline

After training a model like `RandomForestRegressor` to get feature importances, I added this transformer into my full pipeline:

```
# Train a model to get feature importances
from sklearn.ensemble import RandomForestRegressor

forest = RandomForestRegressor(random_state=42)
forest.fit(housing_prepared, housing_labels)

# Get top-k features
k = 5  # or any number you feel is right based on experimentation
top_k_feature_selector = TopFeatureSelector(forest.feature_importances_, k)

# Add it to the full pipeline
from sklearn.pipeline import Pipeline

full_pipeline = Pipeline([
    ("preparation", full_preprocessing_pipeline),  # existing pipeline
    ("feature_selection", top_k_feature_selector), # new step added here
    ("svr", SVR())  # or any regressor you want to test
])

# Train and evaluate
full_pipeline.fit(housing, housing_labels)
```

Adding this feature selector helped reduce overfitting slightly and made the pipeline more focused. It's like telling the model: "*Forget the fluff, just look at what matters most.*"

Plus, it made the whole system feel a lot cleaner and modular. If I ever want to change the number of selected features, I can do that without breaking the whole pipeline 😉

Question: Try creating a single pipeline that does the full data preparation plus the final prediction.

This step was honestly about **cleaning things up**. Instead of running data preparation separately, then feature selection, then prediction — we combine **all steps** into a single pipeline. This makes the code more modular, reusable, and less error-prone. It also helps if we ever want to dump the pipeline using joblib or something similar and reload it later for production use or deployment.

```
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR

housing_prepared = full_preprocessing_pipeline.fit_transform(housing)
forest = RandomForestRegressor(random_state=42)
forest.fit(housing_prepared, housing_labels)

top_k = 5
feature_selector = TopFeatureSelector(forest.feature_importances_, top_k)

final_pipeline = Pipeline([
    ("preprocessing", full_preprocessing_pipeline),
    ("feature_selection", feature_selector),
    ("svr_reg", SVR(kernel="rbf", C=100, gamma=0.1))
])

final_pipeline.fit(housing, housing_labels)

predictions = final_pipeline.predict(housing)
```

Question: Automatically explore some preparation options using `GridSearchCV`

This felt pretty smart to me — like instead of assuming things, I let GridSearchCV do the job of figuring out what's actually better based on performance.

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.svm import SVR

pipeline = Pipeline([
    ("preprocessing", full_preprocessing_pipeline),  # already defined earlier
    ("feature_selection", TopFeatureSelector(forest.feature_importances_, k=5)),
    ("svr", SVR())
])

param_grid = {
    "preprocessing__num__imputer__strategy": ["mean", "median"],
    "preprocessing__num__scaler": [StandardScaler(), MinMaxScaler()],
    "feature_selection__k": [5, 8, 10],
    "svr__kernel": ["rbf"],
```

```
    "svr__C": [10, 100],
    "svr__gamma": [0.01, 0.1]
}

grid_search = GridSearchCV(pipeline, param_grid, cv=3,
scoring="neg_mean_squared_error", verbose=2)
grid_search.fit(housing, housing_labels)
```

- Tried a bunch of preprocessing combos automatically.
- Some combinations that I didn't even expect ended up giving better results.
- `StandardScaler` + `median imputation` + `k=8` actually worked better in my case than the default setup.
- It saved time and made everything more dynamic.

Basically, I let GridSearch handle not just model tuning but even how data should be prepped before training. Makes the whole pipeline way more flexible.