Random Forest Regression: A Comprehensive Beginner's Guide

Type	@datasciencebrain

Random Forest Regression Introduction

Random Forest is one of the most popular and powerful machine learning algorithms . It can be used for both classification and regression tasks, but this guide will focus on its use in **regression** (predicting continuous numeric outcomes). In a random forest, multiple decision trees are built and combined to produce a more robust model than a single decision tree. By the end of this guide, you will understand what random forests are, how they work for regression, and how to build, evaluate, tune, and deploy a Random Forest regression model in Python using scikit-learn.

What is a Random Forest?

A **Random Forest** is an **ensemble learning** method that constructs a multitude of decision trees and **averages their predictions** to improve accuracy and control over-fitting . In essence, it is a "forest" of decision trees, each slightly different due to randomness in the training process. When used for regression, each decision tree in the forest produces a numerical prediction, and the random forest's final prediction is the **average** of all the individual tree predictions. This ensemble approach often yields more accurate and generalizable results than any single decision tree, because the errors of individual trees can cancel out when averaged .

Why Random Forests? For regression problems, random forests provide high predictive accuracy, handle many input features, and reduce the risk of overfitting. They achieve this by combining many trees that are **uncorrelated** (thanks to

injected randomness) so that no individual tree dominates the prediction . The trade-off is that random forests can be slower and more memory-intensive than simpler models, and the results are less interpretable than, say, a single decision tree or a linear regression model.

Key Components of Random Forests

Random forests are built on three key concepts: (1) Decision Trees, (2) Bagging (Bootstrap Aggregation), and (3) Ensemble Learning. Understanding these components will clarify how random forests work.

1. Decision Trees for Regression

A **decision tree** is a tree-structured model used for making predictions. It splits the dataset into smaller subsets based on feature values, forming a flowchart-like structure of nodes (decision points) and leaves (outcomes). Each internal node asks a yes/no question about one of the features, and the data is split accordingly. In regression trees, each leaf node ends up with a predicted numeric value, often the **mean** of the target values of training samples in that leaf. The tree learns to ask questions (choose splits) that partition the data in a way that minimizes prediction error (typically measured by variance or mean squared error within the leaves).

For example, a decision tree might predict house prices by first asking “Is the house smaller than 2000 sqft?”, then branching to ask “Is the location urban or rural?”, and so on, until reaching a final predicted price. Decision trees are intuitive and **interpretable**, but a single tree can easily overfit the training data by growing too complex. This is where random forests improve upon trees by **using many trees** and averaging their results.

2. Bootstrap Aggregation (Bagging)

Bagging, short for *bootstrap aggregation*, is an ensemble technique that helps reduce overfitting and variance. The idea is to train each individual tree on a different random subset of the training data: each tree gets a **bootstrap sample**, which is a sample of the training set **chosen randomly with replacement**. Replacement means the same data point can appear multiple times in one tree’s dataset, and some points may be left out. On average, each tree’s bootstrap

sample is about 63% of the full dataset (with some repeats), leaving about 37% of the training instances as “out-of-bag” data for that tree (this can be used later for validation if desired).

By training each tree on a different bootstrapped dataset, we ensure the trees are diverse. Each tree will see a slightly different set of examples, so their predictions will vary. After training, when it’s time to predict, the random forest **aggregates the results** of all trees (by averaging in regression). This **bootstrap aggregation** process greatly reduces the variance of the model: although individual trees might overfit, their **average** is much more stable.

3. Ensemble Learning in Random Forests

Random Forest is an **ensemble learning** method, meaning it combines multiple models (decision trees) to produce a final result. The principle of ensemble learning is that a group of weak learners can come together to form a strong learner. In a random forest, the **ensemble’s prediction is the averaged prediction of all the trees** . This averaging smooths out the noise and idiosyncrasies of individual trees.

In addition to bagging, random forests add an extra layer of randomness: when growing each tree, the algorithm does not consider all features for splitting at a node, but instead picks a **random subset of features** and finds the best split among those . This is sometimes called the “random subspace method” or *feature bagging*. By randomizing feature choice, random forests **decorrelate** the trees (each tree is less likely to consistently pick the same dominant features), further reducing the variance of the ensemble’s prediction .

To summarize: **each tree** in a random forest is trained on a random subset of the data *and* at each split considers a random subset of features. This makes each tree unique. Finally, all trees’ outputs are combined (averaged) to produce the final prediction. The result is a model that generally achieves **higher accuracy and robustness** than an individual decision tree while **avoiding overfitting** in most cases .

Building a Random Forest Regression Model: Step-by-Step

Next, we will walk through the process of building a Random Forest regression model using Python's **scikit-learn** library. We will cover data preparation, model training, prediction, and evaluation. For illustration purposes, we'll use a built-in dataset. In practice, you would replace this with your own dataset (e.g., from a CSV file).

Step 1: Import Libraries and Load the Dataset

Begin by importing the necessary libraries and loading your data. We'll use the **diabetes dataset** from scikit-learn as an example (a small regression dataset with 10 features). In a real scenario, you might load data using pandas (e.g., `pd.read_csv('mydata.csv')`).

```
# Import required libraries
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_sc
ore

# Load a sample dataset (Diabetes dataset for regression)
data = load_diabetes()
X = data.data       # Features (e.g., medical predictor variables)
y = data.target     # Target variable (e.g., disease progression measure)
print(X.shape, y.shape)  # (442, 10) meaning 442 samples, 10 features
```

In the code above, we import:

- `RandomForestRegressor` for the model,
- `train_test_split` for splitting data, and
- some metrics (MAE, MSE, R^2) for evaluation.

The dataset `X` is a 2D array of shape *(n_samples, n_features)* and `y` is a 1D array of target values. Always make sure to examine your dataset (e.g., check shape, look for missing values, etc.) before modeling. In this case, the diabetes dataset has 442 samples and 10 features, and it's already preprocessed with no missing values, so we can proceed directly.

Step 2: Split Data into Training and Testing Sets

It's important to evaluate your model on data it hasn't seen during training to gauge performance on new, unseen data. We split the dataset into a **training set** (to train the model) and a **test set** (to evaluate the model's performance on unseen data). A common split is 80% training and 20% testing.

```
# Split data into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
print("Training samples:", X_train.shape[0])
print("Testing samples:", X_test.shape[0])
```

Here, `test_size=0.2` means 20% of the data (about 88 samples in this case) will be set aside as the test set. We also set `random_state=42` to ensure the split is reproducible (using a fixed random seed). After this step, `X_train, y_train` will be used to train the Random Forest, and we'll later use `X_test, y_test` for evaluating how well the model generalizes.

Step 3: Train the Random Forest Regressor

Now we create a `RandomForestRegressor` model and train (fit) it on the training data. We can start with default hyperparameters, or specify some. One important parameter is `n_estimators`, which is the number of decision trees in the forest. The default is often 100, which is a good starting point. We'll also set `random_state` for reproducibility.

```
# Initialize the Random Forest regressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

# Train the model on the training data
rf_model.fit(X_train, y_train)
```

This code initializes a random forest with 100 trees and fits it to the training data (`.fit(X_train, y_train)`). Under the hood, the algorithm will:

- Build 100 decision trees, each on a bootstrap sample of `X_train`.

- For each tree, at each split, choose a random subset of features and find the best split among those.
- Grow each tree to its maximum depth (unless limited by default settings or parameters), and use mean squared error (by default) to decide the best splits for regression.

Training may take some time if the dataset is large or `n_estimators` is very high. With 100 trees on a moderate dataset, it should finish quickly. After this step, `rf_model` has learned from the data.

Step 4: Make Predictions on Test Data

After training the model, we use it to predict on the test set features `X_test` (which the model has never seen during training).

```
# Use the trained model to make predictions on the test set
y_pred = rf_model.predict(X_test)
```

The result `y_pred` is an array of predicted values for each sample in `X_test`. Because this is a regression problem, these predictions are continuous values. Each prediction is essentially the **average output of all the trees** in the forest for that sample. Now we'll compare these predictions to the true values `y_test` to see how well our model performed.

Step 5: Evaluate Model Performance

To evaluate the regression model, we will compute common regression metrics: **Mean Absolute Error (MAE)**, **Root Mean Squared Error (RMSE)**, and **R^2 (R-squared)**. Scikit-learn provides functions to compute these. We'll also compute Mean Squared Error (MSE) internally to get RMSE (since RMSE is just the square root of MSE).

```
# Evaluate the model using common regression metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)  # directly get RM
SE
```

```
r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R² Score: {r2:.2f}")
```

These metrics tell us:

- **MAE**: The average absolute difference between predicted and actual values. For example, an MAE of 3.5 means that on average, our predictions are off by 3.5 (in whatever units the target variable is) in either direction.
- **RMSE**: The square root of the average squared difference between predictions and actual values. It gives an error value on the same scale as the original target. RMSE penalizes larger errors more than MAE (because of the squaring). For instance, an RMSE of 4.5 would mean the typical prediction error is about 4.5 units.
- **R^2 Score**: The coefficient of determination, which ranges from 0 to 1, indicating how much variance in the target is explained by the model. An R^2 of 0.90 means 90% of the variance in the target variable is explained by the model (which is very good), whereas an R^2 of 0.0 means the model is no better than predicting the mean.

We will explain these metrics in more detail in the next section. For now, by looking at MAE and RMSE, you can judge how far off the model's predictions are on average. Lower values indicate better performance. The R^2 score closer to 1 indicates a better fit. Keep in mind that performance on the test set reflects how well the model generalizes. If there's a large gap between training and testing performance (e.g., much lower error on training data than on test data), the model might be overfitting.

Model Evaluation Metrics for Regression

When assessing a regression model like Random Forest, several metrics can be used. Here are three key metrics and what they mean:

- **Mean Absolute Error (MAE):** MAE measures the average magnitude of errors in predictions, without considering their direction. It is the average of the absolute differences between predicted values and true values . For example, an MAE of 5 means that on average, the predictions are 5 units off from the actual values. MAE is easy to interpret (same units as the target) and **less sensitive to outliers** than metrics that square the errors, because it doesn't punish large errors as aggressively.
- **Root Mean Squared Error (RMSE):** RMSE is the square root of the average of squared errors (Mean Squared Error). It brings the error to the same unit as the target variable, making it more interpretable . RMSE penalizes large errors more strongly than MAE (due to squaring). For instance, if RMSE = 10, it suggests that the typical deviation of predictions from actual values is about 10 units. RMSE is often used when large errors are particularly undesirable, since it will magnify those in the metric.
- **R^2 (R-squared, Coefficient of Determination):** R^2 indicates the proportion of variance in the target variable that is explained by the model. It is a number between 0 and 1. An R^2 of 1.0 means the model perfectly explains the data (all predictions fit exactly), while R^2 of 0 means the model explains none of the variance (it's no better than always predicting the mean). For example, R^2 = 0.85 means 85% of the variability in the target is accounted for by the model. Higher R^2 is better, but be cautious: a very high R^2 on training data and a much lower R^2 on test data signals overfitting.

In practice, you should look at all these metrics (and others like **MSE** or **MAPE** if relevant) to get a full picture of model performance. MAE and RMSE tell you in absolute terms how wrong your predictions typically are, and R^2 tells you how well the model is explaining the target variability.

Hyperparameter Tuning for Random Forests

Random Forests have several **hyperparameters** that can be tuned to improve performance. Tuning means finding the best combination of parameter values for your specific dataset. It's often done via techniques like grid search or random search with cross-validation. Below, we discuss important hyperparameters and how they affect the model, and then we'll outline how to systematically tune them.

Important Random Forest Hyperparameters

- **n_estimators** (number of trees): The number of decision trees in the forest. More trees can improve performance (the predictions average out more noise) but also increase computation time. In practice, you might start with 100 and increase to 200, 500 or even more until you see diminishing returns. Too few trees can cause underfitting, while a very large number of trees can slightly overfit and will slow down training and prediction.
- **max_depth** (maximum depth of each tree): The depth of a tree is the number of splits from the root to a leaf. By default, trees grow until all leaves are pure or until other stopping criteria are met. Setting `max_depth` to a smaller number (e.g., 5, 10) will constrain the tree from growing too deep. This can prevent overfitting because it limits model complexity. If `max_depth=None`, trees grow until they can't split further. Tuning this is important: a very large depth can overfit, a very shallow depth can underfit.
- **min_samples_split** (minimum samples required to split an internal node): This controls the minimum number of data samples that must be present in a node for it to be split. By default it's 2 (a node will split even if it has only 2 samples). Increasing this value (e.g., to 5 or 10) means nodes must have more samples to consider splitting, which tends to make the tree less deep and reduce overfitting . This is a form of **regularization** for the tree.
- **min_samples_leaf** (minimum samples required to be at a leaf node): Similar to above, but for leaf nodes. It specifies the minimum number of samples that a leaf (terminal node) must have. A higher value forces the tree to have leaves with more samples, reducing how pure (and potentially overfitted) they can be. For example, `min_samples_leaf=5` means a leaf must have at least 5 training samples in it. This also tends to reduce overfitting.
- **max_features** (number of features to consider at each split): This is the number (or fraction) of features that are randomly chosen to evaluate at each split in a tree. In regression, the default is usually all features (`max_features=1.0`, meaning 100% of features), whereas in classification the default may be `sqrt(n_features)`. If you set `max_features` to a smaller fraction (like 0.5 or "sqrt"), each split will only consider a subset of features. Lower `max_features` can reduce overfitting by further randomizing trees (making them more independent), but if set too low it might deprive trees of useful information. Common options to

try are “sqrt” (square root of total features) or a fixed number. The optimal value often depends on the dataset.

- **bootstrap** (True/False): This flag controls whether bootstrap sampling is used. By default, `bootstrap=True` (meaning each tree is trained on a bootstrap sample of the data). If set to False, each tree will be trained on the entire dataset (which essentially turns the Random Forest into an ensemble of bagged trees without sampling variance). Generally, you should leave this True for the classic Random Forest algorithm.
- **oob_score** (True/False): Stands for “out-of-bag score”. If True, the random forest will use the out-of-bag samples (the data not used in a tree’s bootstrap sample) to estimate the performance of the model as it is being trained. This gives a useful cross-validation like performance measure without needing a separate validation set. It’s particularly handy when you don’t have a lot of data to spare for validation. You can enable this and later access `rf_model.oob_score_` after training.
- **n_jobs** (number of CPU cores to use): For training large forests, setting `n_jobs=-1` can be helpful as it tells scikit-learn to use all available CPU cores to train trees in parallel (since each tree can be grown independently). This won’t affect the model’s accuracy, but it can significantly speed up training on multi-core machines.

Other hyperparameters include things like `max_leaf_nodes`, `min_impurity_decrease`, or post-pruning related parameters (`ccp_alpha` for cost-complexity pruning), but the ones above are the primary ones to focus on initially.

Hyperparameter Tuning Strategies

To tune a Random Forest, a common approach is to use **Grid Search** or **Random Search** with cross-validation:

- **Grid Search CV:** You define a grid (dictionary) of possible values for some hyperparameters and exhaustively try all combinations, evaluating each using cross-validation on the training set. For example, you might try `n_estimators` = {100, 200, 500}, `max_depth` = {None, 10, 20}, `min_samples_split` = {2, 5, 10}. The grid search will train a model for each combination and evaluate via CV, then report

the best combination. This ensures you find the optimal values from the grid but can be slow if the grid is large.

- **Randomized Search CV:** Instead of trying all combinations, you specify a distribution of values for each hyperparameter and randomly sample a fixed number of combinations to try. This can explore the space more efficiently if you have many parameters or very wide ranges.

Let's see a brief example of using `GridSearchCV` for our random forest (this is just to illustrate; in practice you might use more folds or more combinations):

```
from sklearn.model_selection import GridSearchCV

# Define a grid of hyperparameter values to test
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5]
}

# Set up the grid search with 3-fold cross-validation
grid_search = GridSearchCV(
    RandomForestRegressor(random_state=42),
    param_grid,
    cv=3,              # 3-fold cross-validation
    scoring='neg_mean_squared_error',  # use MSE (negative because higher = worse)
    n_jobs=-1          # parallelize across cores
)
grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
```

In the code above, we test a few values for `n_estimators`, `max_depth`, and `min_samples_split`. The scoring metric is negative MSE (since scikit-learn's convention is that higher score is better, it takes negative MSE so lower MSE yields higher (less negative) score). After running, `grid_search.best_params_` will show the best

combination found. You could then set your RandomForestRegressor with those parameters and retrain on the full training set.

Tips for tuning:

- Try tuning one or two parameters at a time (for instance, fix `n_estimators` at a reasonable number like 100 while you tune `max_depth` and `min_samples_split`).
- Often, **n_estimators** can be increased until the benefit plateaus. It usually doesn't overfit with more trees, it just gives diminishing returns and longer training.
- Use **validation curves** or **learning curves** to understand how increasing `n_estimators` or depth affects performance.
- If the model is overfitting (train score much better than test), consider reducing `max_depth`, increasing `min_samples_split` or `min_samples_leaf`, or reducing `max_features`. If underfitting, consider the opposite (allow deeper trees, more features, etc.).
- Keep `random_state` fixed during tuning for reproducibility.
- Be mindful of training time. Tuning can be expensive, so sometimes a RandomizedSearch with a budget (n_iter) is faster than a full GridSearch.

Pros and Cons of Random Forests in Regression

Like any algorithm, Random Forest has its advantages and disadvantages. Here are some to consider:

Advantages

- **High Accuracy and Robustness:** Random forests often achieve very competitive accuracy out-of-the-box. By averaging many trees, they reduce the risk of overfitting significantly . Even if individual trees overfit, their averaged result generalizes well (the variance is lower). This makes the model robust on many types of data.
- **Works Well Without Extensive Tuning:** Although you *can* tune hyperparameters for optimal performance, random forests often perform

reasonably well with default settings. You typically get a solid model without heavy parameter tweaking, which is great for beginners.

- **Handles Non-linear Relationships and Interactions:** The model can capture complex interactions between features. Decision trees naturally handle non-linear data patterns, and an ensemble of them can model very intricate relationships that would be hard for a linear model to capture.
- **Insensitive to Feature Scaling:** Features do not need normalization or standardization for tree-based models. A random forest split on a feature is based on thresholds (e.g., is `feature_i <= 5.3?`), so whether the feature is in meters or centimeters doesn't change the splits – it's scale-invariant in that sense.
- **Handles a Large Number of Features:** Random forests can handle high-dimensional data (many features) well. By selecting random subsets of features for splits, they can even be efficient in situations where p (features) >> n (samples).
- **Resistant to Outliers and Missing Data:** Individual decision trees can be somewhat sensitive to outliers, but the averaging effect in a forest makes it more robust to outliers in the target. Also, the random forest algorithm can maintain accuracy even when portions of the data are missing, especially if using the built-in ability to estimate missing values or using surrogate splits. (However, it's still best to handle missing data in preprocessing when possible.)
- **Feature Importance:** Random Forests provide a way to measure feature importance. After training, most implementations can give an importance score for each feature (for example, based on how much each feature split reduces error across all trees). This can be useful for understanding the data and reducing dimensionality.

Disadvantages

- **Less Interpretable:** While a single decision tree is relatively easy to interpret (you can visualize the tree and follow the decision path), a random forest with hundreds of trees is essentially a black box. It's more challenging to explain why the model made a particular prediction, which can be an issue in domains where interpretability is important.

- **Computational Cost:** Training hundreds of deep trees can be computationally intensive, both in terms of CPU and memory. Random forests can be **time-consuming** to train on very large datasets. Prediction is also slower than simple models because it must aggregate results from many trees. However, using parallelism (`n_jobs`) and limiting tree depth can help.
- **Model Size:** A random forest can consume a lot of memory, especially if you have many trees and deep trees. Storing the model (trees and node values) might be an issue if model size is a constraint (for example, deploying to environment with limited memory).
- **Not Great for Extrapolation:** Like all tree-based methods, random forests cannot extrapolate beyond the range of the training data. If your model hasn't seen a certain range of feature values, it will not predict values outside the training range well, since trees make piecewise constant predictions within the range of observed data.
- **Potential to Overfit on Noisy Data:** While ensemble averaging generally protects against overfitting, if there is a lot of noise in the data, the model might still overfit some and not perform optimally. Also, if `n_estimators` is extremely large and trees are allowed to be very complex, the model might eventually start overfitting (though usually the effect is minor compared to a single deep tree).
- **Need for Careful Feature Engineering Still Stands:** Random forests handle many features, but if those features include irrelevant ones, they can still hurt performance (though random forests are somewhat robust due to feature randomness). It's still beneficial to do feature selection or removal of highly correlated features in some cases. They also can't handle categorical variables natively (you need to encode them as dummy variables or use an implementation that supports categorical splits), which could lead to many one-hot features.

In summary, Random Forests are a strong go-to model for regression if you need a balance of ease-of-use, accuracy, and robustness. They shine in many scenarios, but be aware of their limitations, especially regarding interpretability and computational cost.

Saving and Loading the Model (Deployment Basics)

After you've trained a good Random Forest model, you'll likely want to **save** it so that you can reuse it later or deploy it in an application without retraining from scratch. Scikit-learn models can be saved using Python's `pickle` or, more recommended, using the `joblib` library which is optimized for storing large numpy arrays (which are present in Random Forest models).

Here's how to save a trained scikit-learn model using joblib and then load it back:

```
import joblib

# Save the trained model to a file
joblib.dump(rf_model, "random_forest_model.joblib")

# ... Later or in another script/environment ...

# Load the model from the file
loaded_model = joblib.load("random_forest_model.joblib")

# Use the loaded model to make predictions (for example, on new data or the t
est set)
new_predictions = loaded_model.predict(X_test)
```

After running `joblib.dump`, the model is saved in the file `random_forest_model.joblib`. You can then transfer this file to any environment where you want to use the model (as long as Python and scikit-learn are available there). Loading it back with `joblib.load` returns the same `RandomForestRegressor` object with all the trained trees, ready to predict.

Basic Deployment Advice: Once you have a saved model, deploying it means integrating it into a system that provides predictions for new data. This could be a backend server that loads the model and receives data via an API, then returns predictions. For example, you could create a Flask or FastAPI web service that on each request loads or (better) holds in memory the random forest model and calls

`model.predict()` on incoming data. Another scenario is batch prediction, where you load the model in a script and run it on new data periodically.

A few tips for deployment:

- **Environment:** Ensure the Python environment where you deploy has the same version of scikit-learn (and dependencies) that was used to train the model. Models pickled with one version might not load with a very different version of scikit-learn.
- **Preprocessing Consistency:** If your data requires preprocessing (scaling, encoding, etc.), you must apply the **exact same preprocessing** to new data before feeding it to the model. It's common to save preprocessing objects (like a fitted scaler or encoder) along with the model. In our example, we didn't need special preprocessing, but if you did (say, one-hot encoding), your deployment must replicate that.
- **Performance:** For large-scale or real-time deployment, consider the prediction latency of the model. Random Forests are not the fastest at predicting (since hundreds of trees must be evaluated). If latency is a concern, you might need to optimize (e.g., by reducing number of trees, or using techniques like model distillation, or switching to a different model like Gradient Boosting which can sometimes achieve similar accuracy with fewer trees).
- **Monitoring:** Once deployed, monitor your model's predictions. Ensure the input data stays in the same distribution as the training data. If you see a degradation in performance over time (due to data drift), you may need to retrain the model with fresh data.

California Housing Price Prediction using Random Forest Regression

Libraries Required

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

```
import seaborn as sns

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
```

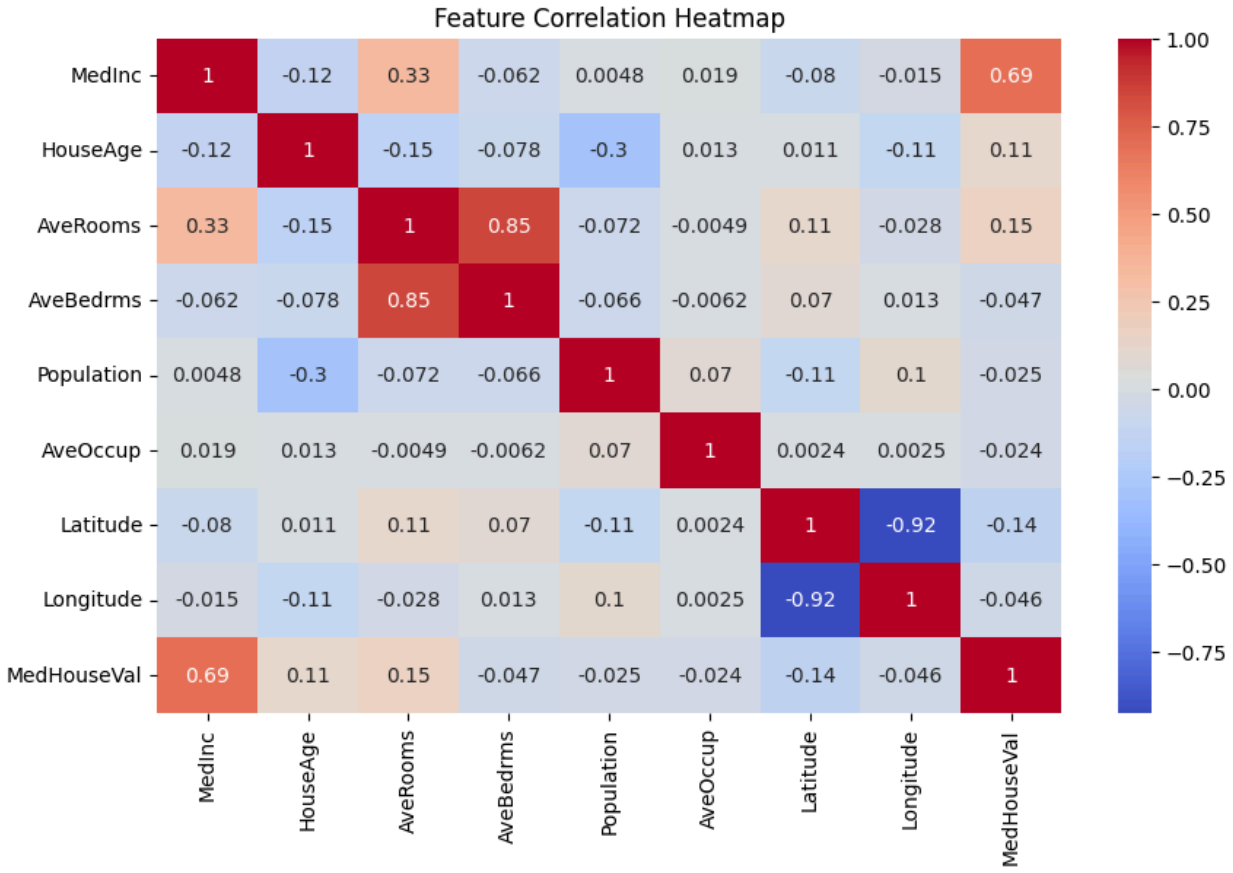
Step 1: Load the Dataset

```
housing = fetch_california_housing(as_frame=True)
df = housing.frame
df.head()
```

Step 2: Exploratory Data Analysis (EDA)

```
df.info()
df.describe()
df.isnull().sum()

# Correlation heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
plt.title("Feature Correlation Heatmap")
plt.show()
```



Step 3: Data Cleaning and Feature Engineering

```
# Check for missing values
df.isnull().sum()

# Add some new features (feature engineering)
df['Rooms_per_Person'] = df['AveRooms'] / df['AveOccup']
df['Bedrooms_per_Room'] = df['AveBedrms'] / df['AveRooms']
```

Step 4: Train-Test Split

```
X = df.drop("MedHouseVal", axis=1)
y = df["MedHouseVal"]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
e=42)
```

Step 5: Preprocessing Pipeline

```
# Identify numerical features
numerical_features = X.columns

# Define transformer
numerical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])

# Combine into ColumnTransformer
preprocessor = ColumnTransformer(transformers=[
    ("num", numerical_transformer, numerical_features)
])
```

Step 6: Define and Train Random Forest Model Pipeline

```
model = Pipeline(steps=[
    ("preprocessor", preprocessor),
    ("regressor", RandomForestRegressor(random_state=42))
])

model.fit(X_train, y_train)
```

Step 7: Evaluate Initial Model

```
y_pred = model.predict(X_test)

print("Initial Model Performance:")
```

```
print("MAE:", mean_absolute_error(y_test, y_pred))
print("MSE:", mean_squared_error(y_test, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred)))
print("R² Score:", r2_score(y_test, y_pred))
```

Step 8: Cross-Validation

```
cv_scores = cross_val_score(model, X, y, cv=5, scoring='r2')
print("Cross-validation R² scores:", cv_scores)
print("Average CV R² score:", np.mean(cv_scores))
```

Step 9: Hyperparameter Tuning with GridSearchCV

```
param_grid = {
    "regressor__n_estimators": [50, 100],
    "regressor__max_depth": [10, 20, None],
    "regressor__min_samples_split": [2, 5],
    "regressor__min_samples_leaf": [1, 2]
}

grid_search = GridSearchCV(model, param_grid, cv=3, scoring='r2', n_jobs=-
1, verbose=2)
grid_search.fit(X_train, y_train)

print("Best parameters found:\n", grid_search.best_params_)
```

Step 10: Evaluate the Tuned Model

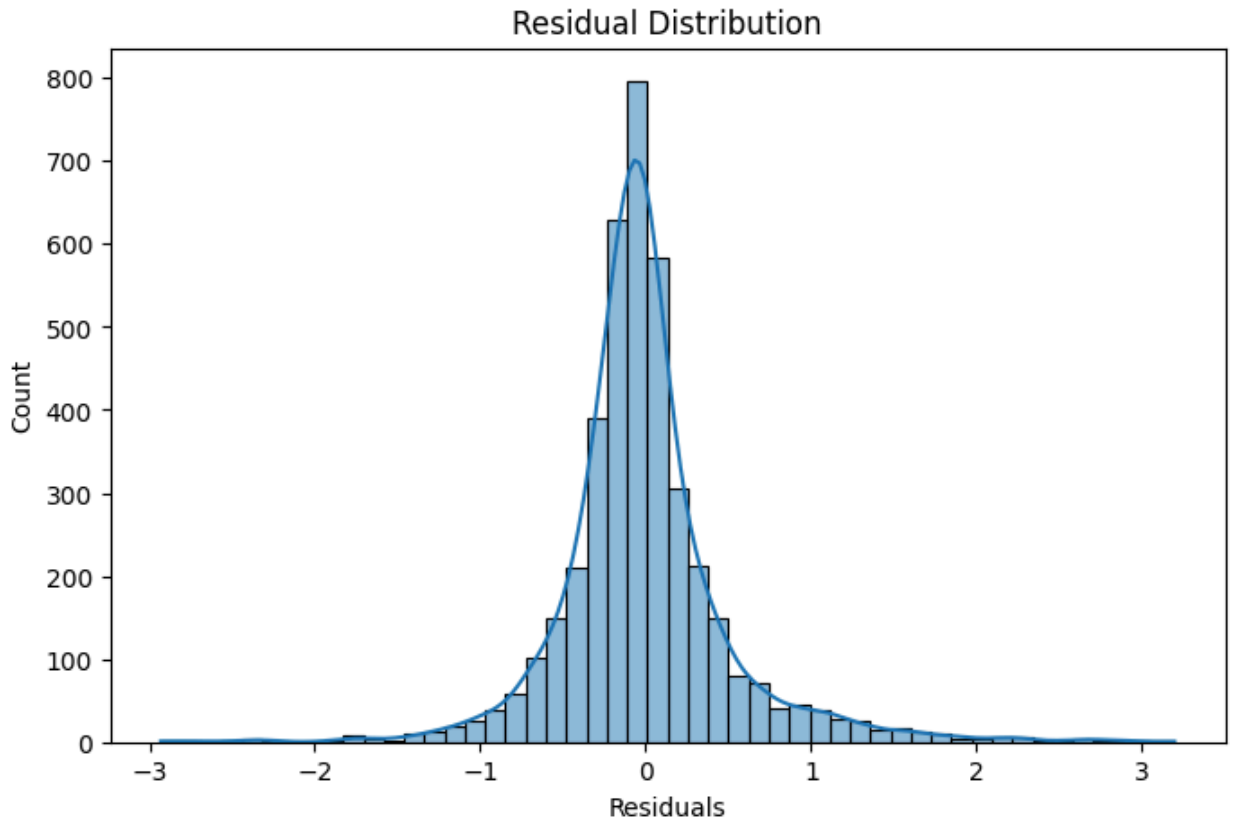
```
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)

print("Tuned Model Performance:")
print("MAE:", mean_absolute_error(y_test, y_pred_best))
```

```
print("MSE:", mean_squared_error(y_test, y_pred_best))
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred_best)))
print("R² Score:", r2_score(y_test, y_pred_best))
```

Step 11: Residual Plot

```
residuals = y_test - y_pred_best
plt.figure(figsize=(8, 5))
sns.histplot(residuals, bins=50, kde=True)
plt.title("Residual Distribution")
plt.xlabel("Residuals")
plt.show()
```



```
import shap
feature_names = X.columns
# Create SHAP explainer
explainer = shap.TreeExplainer(best_model.named_steps["regressor"])
shap_values = explainer.shap_values(preprocessor.transform(X_test))

# Plot summary
shap.summary_plot(shap_values, preprocessor.transform(X_test), feature_names
```