PROGRAMMING FOR AI

Lecture 7

Machine Learning Fundamentals

Instructor: Zafar Iqbal

Agenda

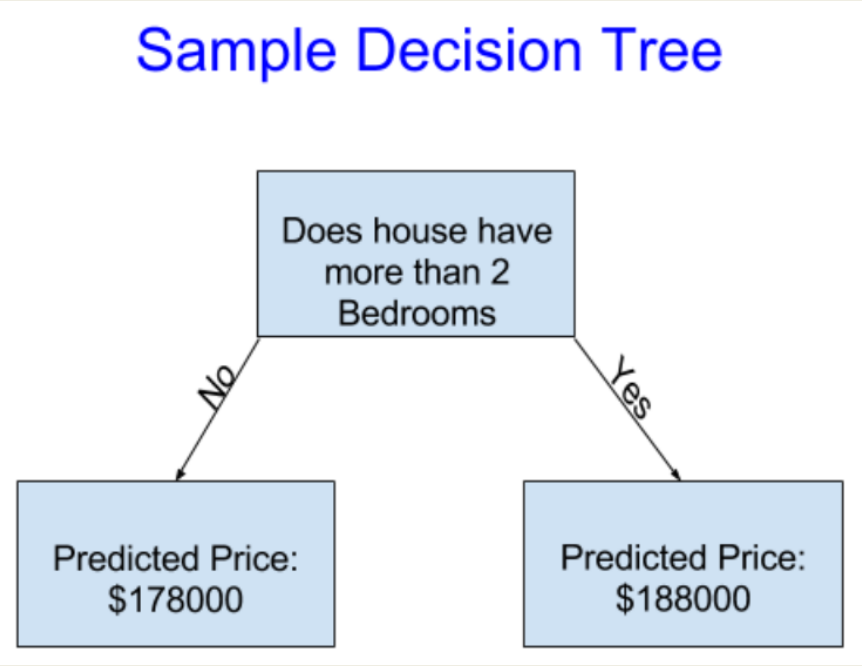
- Introduction
- How Models Work
- Basic Data Exploration
- Your First Machine Learning Model
- Model Validation
- Underfitting and Overfitting
- Random Forests

Introduction

- A branch of AI where computers learn patterns from data to make predictions.
- **Example**: Predicting house prices based on size and location.
- **Types**: Supervised (labeled data), Unsupervised (unlabeled data), Reinforcement (learning through rewards).

Starting Model: Decision Tree

- Initial model to learn: Decision Tree.
- Simple to understand, serves as a foundation for advanced models.
- Building block for some of the best data science models despite fancier options.

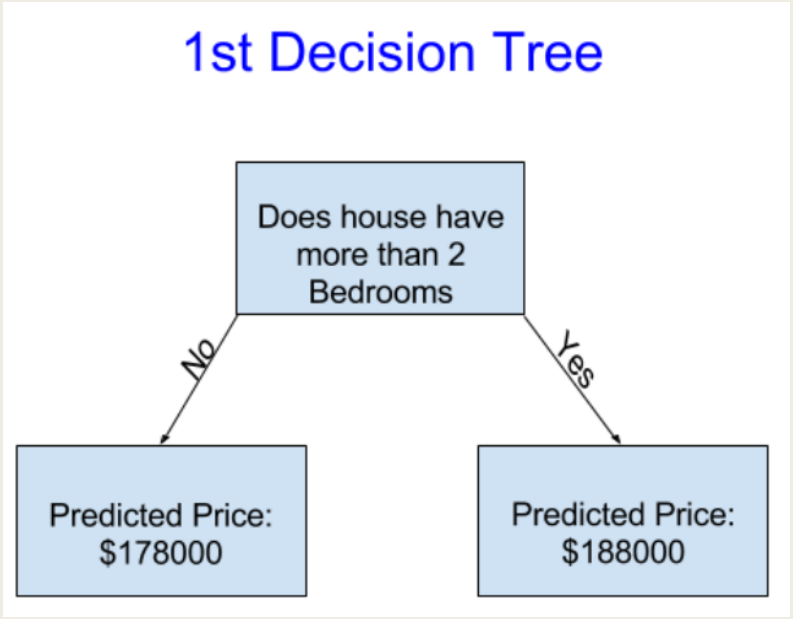


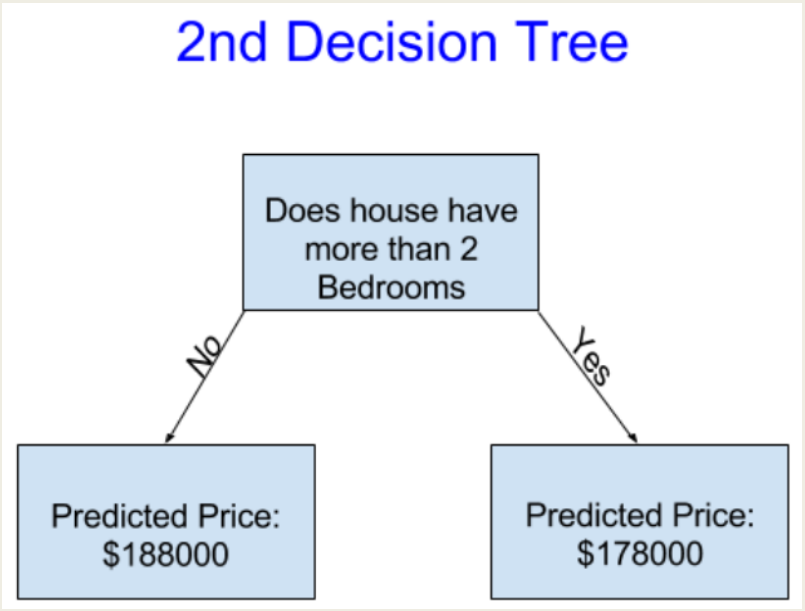
Starting Model: Decision Tree

- Houses are divided into only two categories.
- The predicted price for a house is the historical average price of houses in the same category.
- Data is used to:
  - *Decide how to divide houses into two groups.*
  - *Determine the predicted price for each group.*
- This process of identifying patterns in the data is called **fitting or training the model**.
- The data used for training the model is referred to as training data.
- The specific method for fitting the model (e.g., how the data is split) is complex and will be discussed later.

Improving the Decision Tree

- Which of the following two decision trees is more likely to result from fitting the real estate training data?

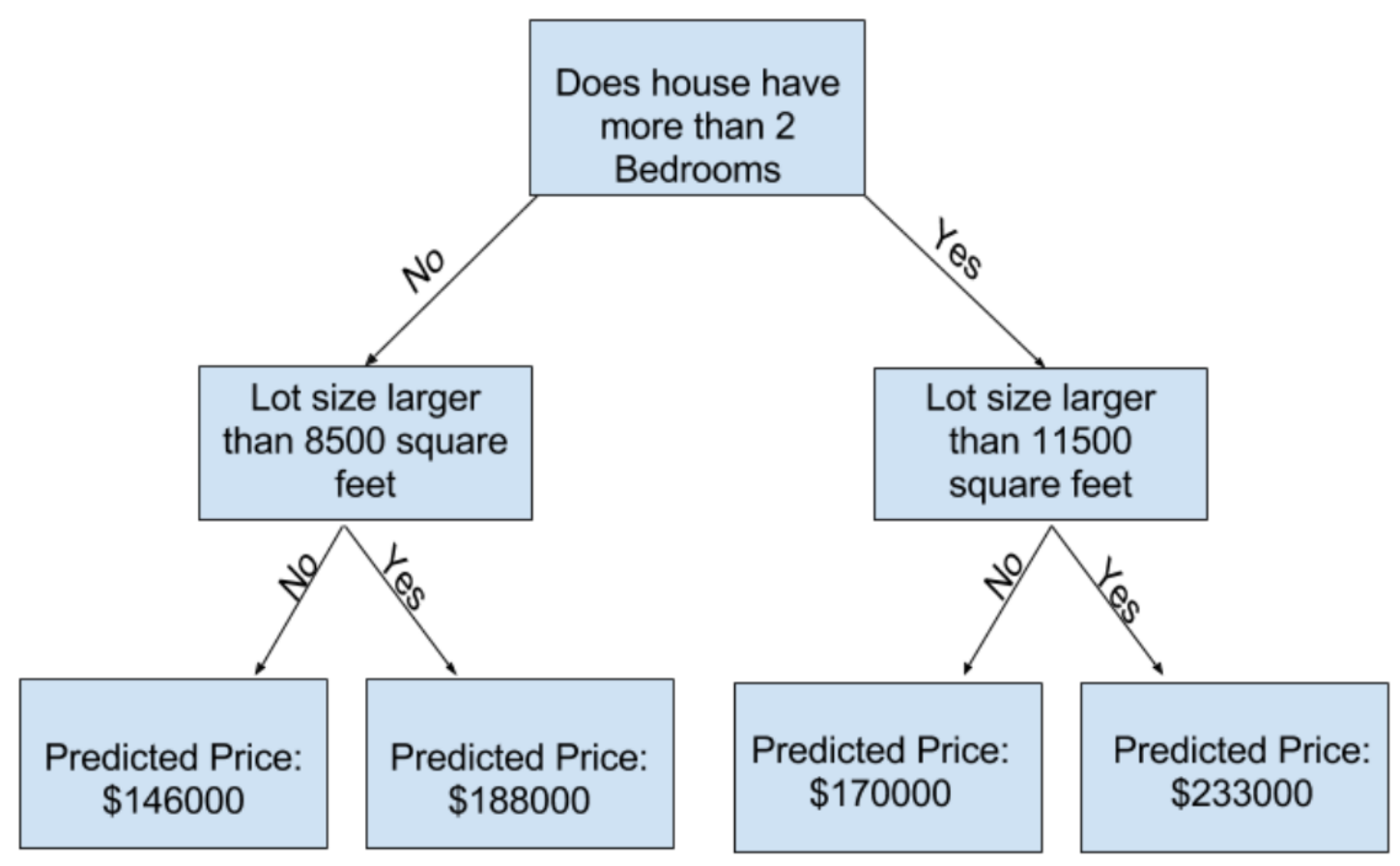




Improving the Decision Tree

- **Decision Tree 1** likely makes more sense because:
- It reflects the general trend that houses with more bedrooms usually sell for higher prices.
- **Limitation of this model:**
- It doesn't account for other important factors like:
  - *Number of bathrooms*
  - *Lot size*
  - *Location*
  - *Etc.*
- **Improving the model:**
- Use a **deeper tree** with more "splits" to capture additional factors.
- Example of improvement:
- A deeper decision tree could also consider the **lot size** of each house.

Improving the Decision Tree



Improving the Decision Tree

- To **predict a house's price**, follow the decision tree path that matches the house's characteristics.
- The **predicted price** is found at the **bottom** of the tree.
- The point at the bottom where a prediction is made is called a **leaf**.
- The **splits** in the tree and the **values at the leaves** are determined using the data.

Data Exploration

```
# save filepath to variable for easier access
melbourne_file_path= '../input/melbourne-housing-snapshot/melb_data.csv'
# read the data and store data in DataFrame titled melbourne_data
melbourne_data = pd.read_csv(melbourne_file_path)
melbourne_data.columns
# print a summary of the data in Melbourne data
melbourne_data.describe()
```

Index
(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG', 'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car', 'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude', 'Longtitude', 'Regionname', 'Propertycount'], dtype='object')

Subset of data

- There are many ways to select a subset of your data.
  - Dot notation, which we use to select the "**prediction target**"
  - Selecting with a column list, which we use to select the "**features**"

Selecting The Prediction Target

- You can pull out a variable with dot-notation. This single column is stored in a Series, which is broadly like a DataFrame with only a single column of data.
- We'll use the dot notation to select the column we want to predict, which is called the prediction target. By convention, the prediction target is called y. So the code we need to save the house prices in the Melbourne data is

- `y = melbourne_data.Price`

Choosing "Features"

- **Features** are the columns used as inputs to a model to make predictions.
  - In a housing model, features are the columns used to predict home price (the target).
- **Feature selection:**
  - You can use all columns except the target as features.
  - In some cases, using fewer features leads to better model performance.
- **Initial modeling approach:**
  - Start by building a model with a few selected features.
  - Later, iterate and compare models with different sets of features.
- Here is an example:
- `melbourne_features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']`
- By convention, this data is called **X**.
- `X = melbourne_data[melbourne_features]`

Data describe()

Let's quickly review the data we'll be using to predict house prices using the describe method and the head method, which shows the top few rows.

```
X.describe()
```

	Rooms	Bathroom	Landsize	Lattitude	Longtitude
count	6196.000000	6196.000000	6196.000000	6196.000000	6196.000000
mean	2.931407	1.576340	471.006940	-37.807904	144.990201
std	0.971079	0.711362	897.449881	0.075850	0.099165
min	1.000000	1.000000	0.000000	-38.164920	144.542370
25%	2.000000	1.000000	152.000000	-37.855438	144.926198
50%	3.000000	1.000000	373.000000	-37.802250	144.995800
75%	4.000000	2.000000	628.000000	-37.758200	145.052700
max	8.000000	8.000000	37000.000000	-37.457090	145.526350

Interpreting Data Description

- The results display **8 statistics** for each column in the dataset:
- Count: Number of rows with **non-missing** values.
- Missing values can occur for many reasons (e.g., no 2nd bedroom in a 1-bedroom house).
- Mean: The average value of the column.
- Std (Standard Deviation): Measures how spread out the values are.
- The following values are **percentiles** based on sorted data:
  - ***Min**: The smallest value.*
  - ***25% (25th percentile)**: Value greater than 25% of the data, smaller than 75%.*
  - ***50% (Median or 50th percentile)**: Middle value.*
  - ***75% (75th percentile)**: Value greater than 75% of the data.*
  - ***Max**: The largest value.*

X.head()

	Rooms	Bathroom	Landsize	Lattitude	Longtitude
1	2	1.0	156.0	-37.8079	144.9934
2	3	2.0	134.0	-37.8093	144.9944
4	4	1.0	120.0	-37.8072	144.9941
6	3	2.0	245.0	-37.8024	144.9993
7	2	1.0	256.0	-37.8060	144.9954

- Visually inspecting your data is a critical step in a data scientist's workflow.
- Use basic commands (like .head(), .describe(), .info(), or visual plots) to examine the dataset.

Building Your Model

- You will use the **scikit-learn** library to create your models.
- Scikit-learn is easily the most popular library for modeling the types of data typically stored in DataFrames.
- The steps to building and using a model are:
  - **Define**: What type of model will it be? A decision tree? Some other type of model? Some other parameters of the model type are specified too.
  - **Fit**: Capture patterns from provided data. This is the heart of modeling.
  - **Predict**: Just what it sounds like
  - **Evaluate**: Determine how accurate the model's predictions are.

Building Your Model

- Import the Model
  - `from sklearn.tree import DecisionTreeRegressor`
  - Decision Tree Regressor is a supervised learning model used for regression tasks—like predicting house prices.

- Define the Model
  - `melbourne_model = DecisionTreeRegressor(random_state=1)`
  - This creates an instance of the model.
  - random_state=1 ensures reproducibility, meaning you'll get the same results each time you run the code.

Decision Tree Model

- Fit the Model
  - **`melbourne_model.fit(X, y)`**
  - X: Your features (input columns).
  - y: Your target (what you're trying to predict — e.g., house price).
  - The model learns the relationships between the features and the target.
- Make Predictions
  - **`melbourne_model.predict(X.head())`**
  - This predicts the prices for the first five houses in your dataset using the features in X.head().
  - The predictions are
  - [1035000. 1465000. 1600000. 1876000. 1636000.]
  - These are the model's predicted prices for the corresponding feature inputs.

Model Validation

- Model validation is the process of evaluating a model to assess its quality.
- The primary focus is often on predictive accuracy—how close the model's predictions are to actual outcomes.
- A common mistake is evaluating model performance using the same training data used to build the model.
- This approach is flawed because it does not accurately represent how the model performs on unseen data.
- To evaluate model quality, predictions must be compared to actual results in a summarized, interpretable way.
- Reviewing thousands of individual predictions (e.g., 10,000 house price predictions) is impractical.

Model Validation

- Instead, model performance is often summarized using a single metric.
- One commonly used metric is Mean Absolute Error (MAE).
- MAE measures the average magnitude of errors in predictions, without considering their direction.
- The concept of “error” in MAE refers to the difference between the predicted and actual values.
- The prediction error for each house is:
- **`error=actual−predicted`**
- Example: If a house costs $150,000 and the predicted price is $100,000, the error is $50,000.
- MAE uses the absolute value of each error to ensure all errors are positive.
- These absolute errors are then averaged to produce a single performance metric.

Model Validation

```
from sklearn.metrics import mean_absolute_error

predicted_home_prices = melbourne_model.predict(X)
mean_absolute_error(y, predicted_home_prices)
```

- 434.71594577146544

The Problem with "In-Sample" Scores

- The score calculated using training data is called an in-sample score.
- Using the same data for training and evaluation leads to misleading results.
- Example: If green doors appear expensive in training data, the model may wrongly learn that door color affects price.
- The model seems accurate on training data but fails on new, real-world data.
- To evaluate real performance, we must test the model on unseen data.
- This is done by excluding part of the data during training.
- The excluded data used for testing is called validation data.

Coding it

- scikit-learn provides a function called train_test_split to divide data.
- It splits the dataset into training data and validation data.
- The training data is used to fit the model.
- The validation data is used to evaluate the model's accuracy.
- We use mean_absolute_error on the validation set to measure model performance.

Coding it

```
from sklearn.model_selection import train_test_split


# split data into training and validation data, for both features and target.
The split is based on a random number generator. Supplying a numeric value to
the random_state argument guarantees we get the same split every time we run
this script.
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)
# Define model
melbourne_model = DecisionTreeRegressor()
# Fit model
melbourne_model.fit(train_X, train_y)


# get predicted prices on validation data
val_predictions = melbourne_model.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))
```

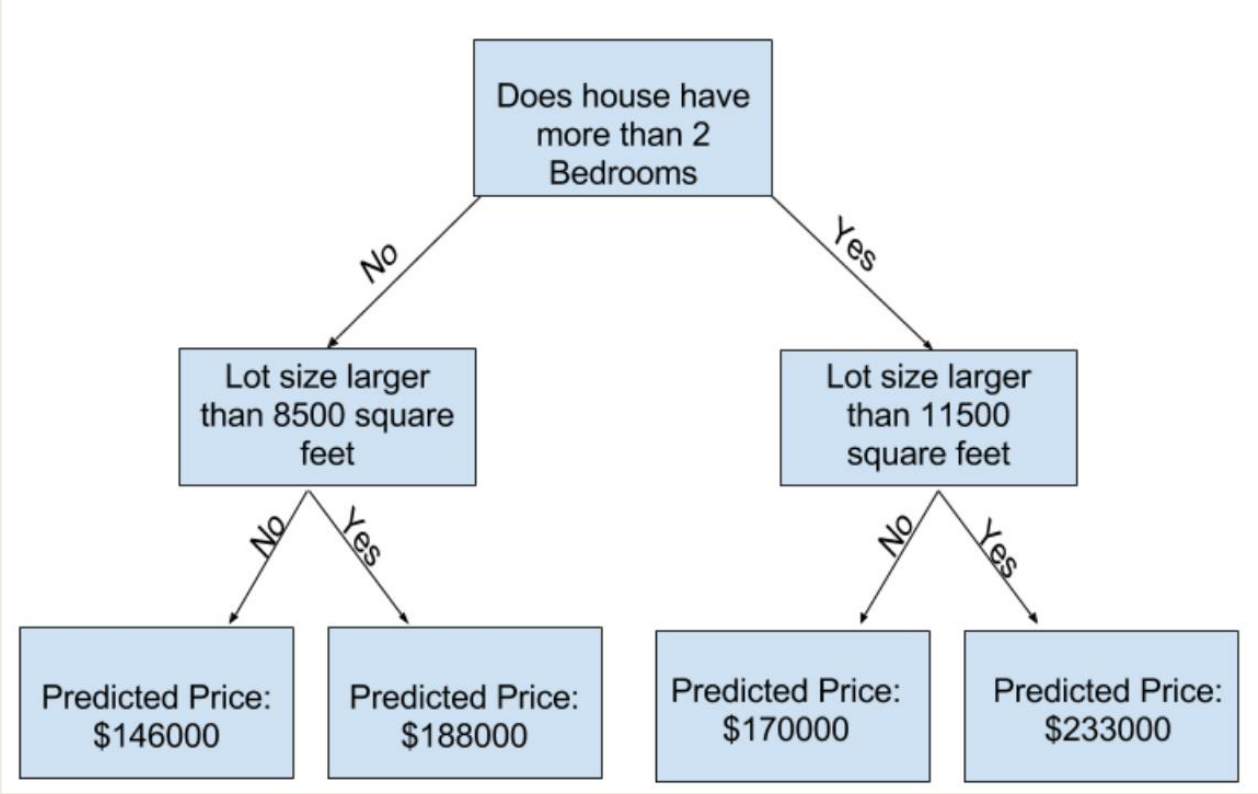
265806.91478373145

Coding it

- **In-sample MAE** was about **$500**, suggesting high accuracy on training data.
- **Out-of-sample MAE** was over **$250,000**, revealing poor performance on new data.
- This highlights the risk of relying only on training data: the model may appear accurate but fail in practice.
- The error is roughly **25% of the average home value** ($1.1 million), making the model **unusable** for real-world predictions.
- To improve the model, consider:
  - Using **better features**
  - Trying **different model types**
  - Refining the **training/validation process**

Underfitting and Overfitting

- There are reliable way to measure model accuracy,
- Experiment with alternative models and see which gives the best predictions.



Experimenting With Different Models

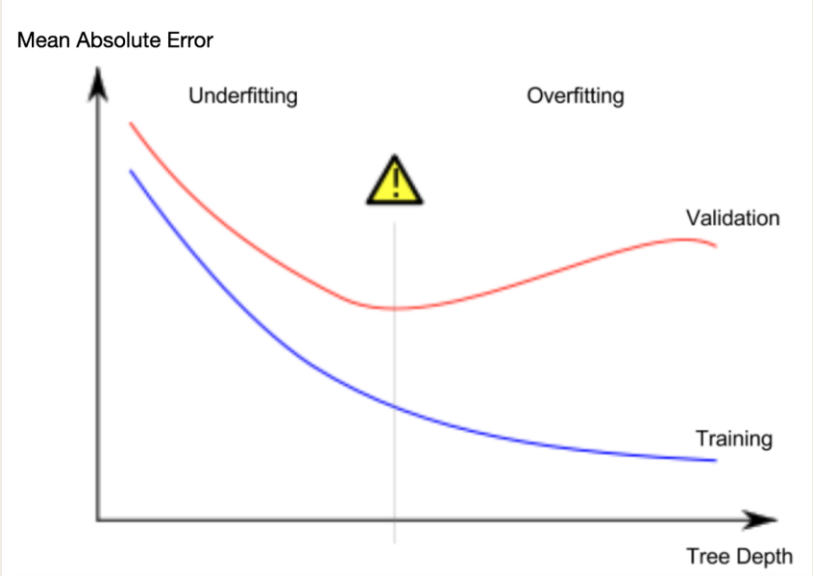
- It's common for a decision tree to have around 10 splits from the top level (all houses) to a leaf.
- As the tree becomes deeper:
  - The dataset is divided into more leaves.
- Each leaf contains fewer houses.
  - Example of splitting:
  - 1 split ⟶ 2 groups
  - 2 splits ⟶ 4 groups
  - 3 splits ⟶ 8 groups
  - …
  - 10 splits ⟶ 210=10242^{10} = 1024210=1024 groups (leaves)
- More leaves = fewer houses per leaf
- Predictions based on these small groups are very close to the training data.
- But predictions on new data may be unreliable.

Underfitting and Overfitting

- This is called **overfitting**:
  - The model **fits training data almost perfectly**.
  - It performs poorly on validation or unseen data due to lack of generalization.
- On the other hand, a very **shallow tree**:
- Has fewer splits (e.g., 2 or 4 groups total).
- Each group still contains a wide variety of houses.
- Predictions are inaccurate, even on the training data.
- This is known as **underfitting**:
  - The model **fails to capture important patterns**.
  - Results in poor performance on both training and validation data.

Underfitting and Overfitting

- Our goal is to achieve high accuracy on new (unseen) data.
- We estimate this accuracy using validation data.
- The key is to find the balance between underfitting and overfitting.
- Visually, this balance appears as the lowest point on the validation error curve (often shown in red).



Example

- There are multiple ways to control the depth of a decision tree.
- Some methods allow varying depths for different branches of the tree.
- The **max_leaf_nodes** parameter offers a practical and intuitive way to manage:
- The balance between underfitting and overfitting.
- Fewer leaf nodes:
  - Simpler model.
  - Risk of underfitting.
- More leaf nodes:
  - More complex model.
  - Risk of overfitting.
- Using max_leaf_nodes, you can gradually increase model complexity to find the optimal point.
- A utility function can be used to:
  - Test different max_leaf_nodes values.
  - Compare MAE (Mean Absolute Error) for each configuration.
  - Select the model with the lowest validation error.

Example

```
from sklearn.metrics import mean_absolute_error
from sklearn.tree import DecisionTreeRegressor


def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)
```

The data is loaded into **train_X**, **val_X**, **train_y** and **val_y** using the code

Max leaf nodes

- We can use a for-loop to compare the accuracy of models built with different values for *max_leaf_nodes.*

```
# compare MAE with differing values of max_leaf_nodes
for max_leaf_nodes in [5, 50, 500, 5000]:
    my_mae = get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y)
    print("Max leaf nodes: %d  \t\t Mean Absolute Error:  %d" %(max_leaf_nodes, my_mae))
```

  - Max leaf nodes: 5 Mean Absolute Error: 347380
  - Max leaf nodes: 50 Mean Absolute Error: 258171
  - Max leaf nodes: 500 Mean Absolute Error: 243495
  - Max leaf nodes: 5000 Mean Absolute Error: 254983

Conclusion

- Here's the takeaway: Models can suffer from either:
  - **Overfitting**: capturing spurious patterns that won't recur in the future, leading to less accurate predictions, or
  - **Underfitting**: failing to capture relevant patterns, again leading to less accurate predictions.
- We use **validation** data, which isn't used in model training, to measure a candidate model's accuracy. This lets us try many candidate models and keep the best one.

Random Forests

- Decision trees present a trade-off:
  - A **deep tree** with many leaves may **overfit**, as predictions are based on very few houses per leaf.
  - A **shallow tree** with few leaves may **underfit**, failing to capture important patterns in the data.
- This tension between underfitting and overfitting is common in most machine learning models.
- Random forest is a model designed to address this:
  - It uses **many decision trees.**
  - It makes predictions by **averaging the predictions** of each individual tree.
  - This approach **improves predictive accuracy** compared to a single tree.

Advantages of random forests

- Generally performs well **out of the box** with default parameters.
- Less prone to overfitting compared to a single decision tree.

- More advanced models may offer better performance:
  - However, they are often more sensitive to hyperparameter tuning.
  - Requires more careful modeling and validation.

Example

- At the end of data-loading, we have the following variables:
  - train_X
  - val_X
  - train_y
  - val_y

Random Forests

- Random forest model is similar to decision tree in scikit-learn
- This time using the RandomForestRegressor class instead of DecisionTreeRegressor.

- `from sklearn.ensemble import RandomForestRegressor`
- `from sklearn.metrics import mean_absolute_error`

- `forest_model = RandomForestRegressor(random_state=1)`
- `forest_model.fit(train_X, train_y)`
- `melb_preds = forest_model.predict(val_X)`
- `print(mean_absolute_error(val_y, melb_preds))`

- 191669.7536453626