Contents

1 Getting started with Artificial Intelligence.

Objectives:

Introduction to AI: Brief overview of artificial intelligence, including its definition, applications, and importance in today's world.
Hands-On Programming Experience: AI programming environment or framework where they can write basic code to solve a problem or implement a simple AI algorithm.
Data Exploration and Preprocessing: How to explore a dataset, perform basic data preprocessing tasks such as cleaning and normalization, and understand the importance of data quality in AI.
Model Training and Evaluation: Guide participants through the process of training a machine learning model using a preprocessed dataset, evaluating the model's performance, and interpreting the results.

Deliverables:

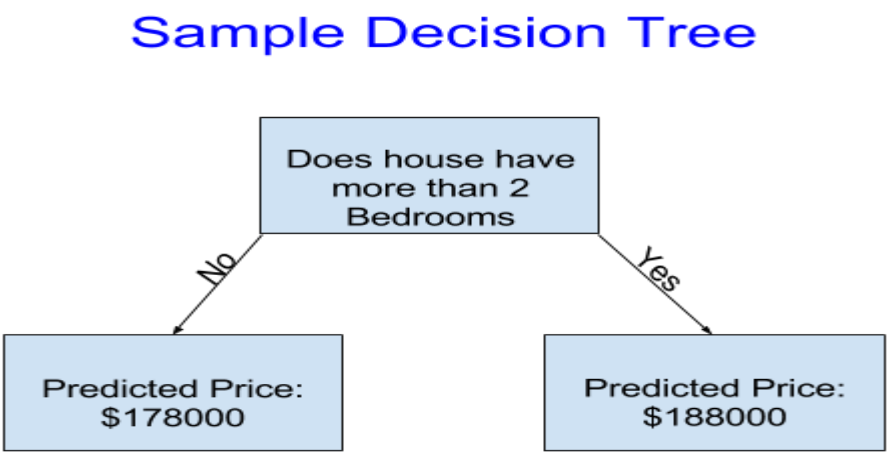
Code Implementation: Write and execute basic code in the chosen AI programming environment or framework, demonstrating their understanding of fundamental programming concepts in the context of AI.
Preprocessed Dataset: Perform data exploration and preprocessing tasks.
Trained Model: Train machine learning model with acceptable performance metrics on real-world datasets.
Evaluation Metrics: Interpret the evaluation metrics of the trained model and understand its performance in solving the given problem.

1.1 How Models Work

The very first step is to understand how models works in AI.

1.1.1 Introduction

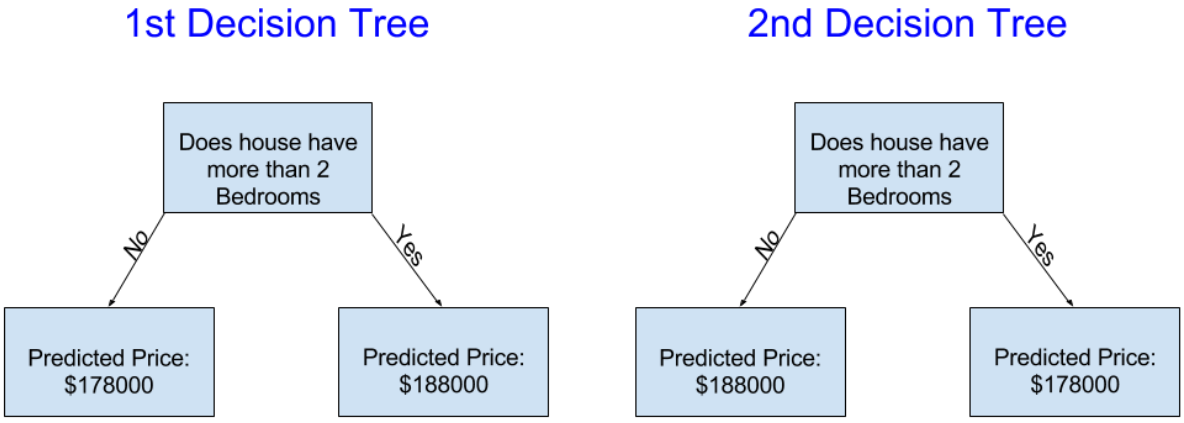
We'll start with an overview of how machine learning models work and how they are used. This may feel basic if you've done statistical modeling or machine learning before. Don't worry, we will progress to building powerful models soon.
This course will have you build models as you go through following scenario:
Your cousin has made millions of dollars speculating on real estate. He's offered to become business partners with you because of your interest in data science. He'll supply the money, and you'll supply models that predict how much various houses are worth.
You ask your cousin how he's predicted real estate values in the past, and he says it is just intuition. But more questioning reveals that he's identified price patterns from houses he has seen in the past, and he uses those patterns to make predictions for new houses he is considering.
Machine learning works the same way. We'll start with a model called the Decision Tree. There are fancier models that give more accurate predictions. But decision trees are easy to understand, and they are the basic building block for some of the best models in data science.
For simplicity, we'll start with the simplest possible decision tree.



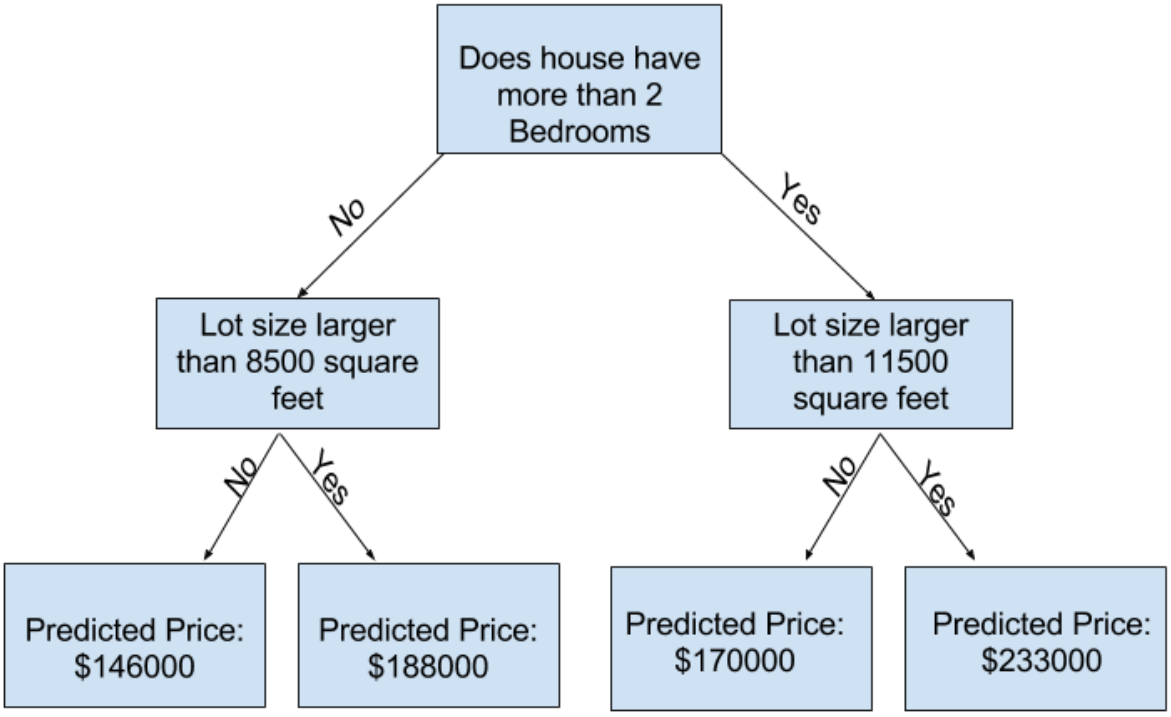
It divides houses into only two categories. The predicted price for any house under consideration is the historical average price of houses in the same category.
We use data to decide how to break the houses into two groups, and then again to determine the predicted price in each group. This step of capturing patterns from data is called **fitting** or **training** the model. The data used to **fit** the model is called the **training data**.
The details of how the model is fit (e.g. how to split up the data) is complex enough that we will save it for later. After the model has been fit, you can apply it to new data to **predict** prices of additional homes.

1.1.2 Improving the Decision Tree

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



The decision tree on the left (Decision Tree 1) probably makes more sense, because it captures the reality that houses with more bedrooms tend to sell at higher prices than houses with fewer bedrooms. The biggest shortcoming of this model is that it doesn't capture most factors affecting home price, like number of bathrooms, lot size, location, etc.
You can capture more factors using a tree that has more "splits." These are called "deeper" trees. A decision tree that also considers the total size of each house's lot might look like this:



You predict the price of any house by tracing through the decision tree, always picking the path corresponding to that house's characteristics. The predicted price for the house is at the bottom of the tree. The point at the bottom where we make a prediction is called a **leaf.**
The splits and values at the leaves will be determined by the data, so it's time for you to check out the data you will be working with.

1.2 Basic Data Exploration

The second step is to Load and understand your data.

1.2.1 Using Pandas to get familiar with your data

The first step in any machine learning project is familiarize yourself with the data. You'll use the Pandas library for this. Pandas is the primary tool data scientists use for exploring and manipulating data. Most people abbreviate pandas in their code as pd. We do this with the command.

```
import pandas as pd
```

The most important part of the Pandas library is the DataFrame. A DataFrame holds the type of data you might think of as a table. This is similar to a sheet in Excel, or a table in a SQL database.
Pandas has powerful methods for most things you'll want to do with this type of data.
As an example, we'll look at data about home prices in Melbourne, Australia. In the hands-on exercises, you will apply the same processes to a new dataset, which has home prices in Iowa.
The example (Melbourne) data is at the file path **../input/melbourne-housing-snapshot/melb_data.csv**.
We load and explore the data with the following commands:

```
# 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)
# print a summary of the data in Melbourne data
melbourne_data.describe()
```

	Rooms	Price	Distance	Postcode	Bedroom2	Bathroom
count	13580.000000	1.358000e+04	13580.000000	13580.000000	13580.000000	13580.000000
mean	2.937997	1.075684e+06	10.137776	3105.301915	2.914728	1.534242
std	0.955748	6.393107e+05	5.868725	90.676964	0.965921	0.691712
min	1.000000	8.500000e+04	0.000000	3000.000000	0.000000	0.000000
25%	2.000000	6.500000e+05	6.100000	3044.000000	2.000000	1.000000
50%	3.000000	9.030000e+05	9.200000	3084.000000	3.000000	1.000000
75%	3.000000	1.330000e+06	13.000000	3148.000000	3.000000	2.000000
max	10.000000	9.000000e+06	48.100000	3977.000000	20.000000	8.000000

1.2.2 Interpreting Data Description

The results show 8 numbers for each column in your original dataset. The first number, the **count**, shows how many rows have non-missing values.

Missing values arise for many reasons. For example, the size of the 2nd bedroom wouldn't be collected when surveying a 1-bedroom house. We'll come back to the topic of missing data.

The second value is the **mean**, which is the average. Under that, **std** is the standard deviation, which measures how numerically spread out the values are.

To interpret the **min**, **25%**, **50%**, **75%** and **max** values, imagine sorting each column from lowest to highest value. The first (smallest) value is the min. If you go a quarter way through the list, you'll find a number that is bigger than 25% of the values and smaller than 75% of the values. That is the **25%** value (pronounced "25th percentile"). The 50th and 75th percentiles are defined analogously, and the **max** is the largest number.

1.3 Your First Machine Learning Model

Building your first model. Hurray!

1.3.1 Selecting Data for Modeling

Your dataset had too many variables to wrap your head around, or even to print out nicely. How can you pare down this overwhelming amount of data to something you can understand?

We'll start by picking a few variables using our intuition. Later courses will show you statistical techniques to automatically prioritize variables.

To choose variables/columns, we'll need to see a list of all columns in the dataset. That is done with the **columns** property of the DataFrame (the bottom line of code below).

```
import pandas as pd
melbourne_file_path = '../input/melbourne-housing-
snapshot/melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path)
melbourne_data.columns
```

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

```
# The Melbourne data has some missing values (some houses for
which some variables weren't recorded.)
# We'll learn to handle missing values in a later tutorial.
# Your Iowa data doesn't have missing values in the columns you
use.
# So we will take the simplest option for now, and drop houses
from our data.
# Don't worry about this much for now, though the code is:

# dropna drops missing values (think of na as "not available")
melbourne_data = melbourne_data.dropna(axis=0)
```

1.3.2 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
```

1.3.3 Choosing "Features"

The columns that are inputted into our model (and later used to make predictions) are called "features." In our case, those would be the columns used to determine the home price. Sometimes, you will use all columns except the target as features. Other times you'll be better off with fewer features.
For now, we'll build a model with only a few features. Later on you'll see how to iterate and compare models built with different features.
We select multiple features by providing a list of column names inside brackets. Each item in that list should be a string (with quotes).
Here is an example:

```
melbourne_features = ['Rooms', 'Bathroom', 'Landsize',
'Lattitude', 'Longtitude']
```

By convention, this data is called **X**.

```
X = melbourne_data[melbourne_features]
```

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

```
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 checking your data with these commands is an important part of a data scientist's job. You'll frequently find surprises in the dataset that deserve further inspection.

1.3.4 Building Your Model

You will use the **scikit-learn** library to create your models. When coding, this library is written as **sklearn**, as you will see in the sample code. 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.
Here is an example of defining a decision tree model with scikit-learn and fitting it with the features and target variable.

```
from sklearn.tree import DecisionTreeRegressor
# Define model. Specify a number for random_state to ensure same
results each run
melbourne_model = DecisionTreeRegressor(random_state=1)

# Fit model
melbourne_model.fit(X, y)
```

```
Output: DecisionTreeRegressor(random_state=1)
```

Many machine learning models allow some randomness in model training. Specifying a number for random_state ensures you get the same results in each run. This is considered a good practice. You use any number, and model quality won't depend meaningfully on exactly what value you choose.
We now have a fitted model that we can use to make predictions.
In practice, you'll want to make predictions for new houses coming on the market rather than the houses we already have prices for. But we'll make predictions for the first few rows of the training data to see how the predict function works.

```
print("Making predictions for the following 5 houses:")
print(X.head())
print("The predictions are")
print(melbourne_model.predict(X.head()))
```

Output:

```
Making predictions for the following 5 houses:
   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
The predictions are
[1035000. 1465000. 1600000. 1876000. 1636000.]
```

1.4 Model Validation

Measure the performance of your model, so you can test and compare alternatives.
You've built a model. But how good is it?
In this lesson, you will learn to use model validation to measure the quality of your model. Measuring model quality is the key to iteratively improving your models.

1.4.1 What is Model Validation?

You'll want to evaluate almost every model you ever build. In most (though not all) applications, the relevant measure of model quality is predictive accuracy. In other words, will the model's predictions be close to what actually happens.
Many people make a huge mistake when measuring predictive accuracy. They make predictions with their *training data* and compare those predictions to the target values in the *training data*. You'll see the problem with this approach and how to solve it in a moment, but let's think about how we'd do this first.
You'd first need to summarize the model quality into an understandable way. If you compare predicted and actual home values for 10,000 houses, you'll likely find mix of good and bad predictions. Looking through a list of 10,000 predicted and actual values would be pointless. We need to summarize this into a single metric.
There are many metrics for summarizing model quality, but we'll start with one

called **Mean Absolute Error** (also called **MAE**). Let's break down this metric starting with the last word, error.
The prediction error for each house is:

```
error=actual-predicted
```

So, if a house cost $150,000 and you predicted it would cost $100,000 the error is $50,000.
With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors. This is our measure of model quality. In plain English, it can be said as

On average, our predictions are off by about X.

To calculate MAE, we first need a model.

```
# Data Loading Code Hidden Here
import pandas as pd

# Load data
melbourne_file_path = '../input/melbourne-housing-
snapshot/melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path)
# Filter rows with missing price values
filtered_melbourne_data = melbourne_data.dropna(axis=0)
# Choose target and features
y = filtered_melbourne_data.Price
melbourne_features = ['Rooms', 'Bathroom', 'Landsize',
'BuildingArea',
                      'YearBuilt', 'Lattitude', 'Longtitude']
X = filtered_melbourne_data[melbourne_features]

from sklearn.tree import DecisionTreeRegressor
# Define model
melbourne_model = DecisionTreeRegressor()
# Fit model
melbourne_model.fit(X, y)
```

Once we have a model, here is how we calculate the mean absolute error:

```
from sklearn.metrics import mean_absolute_error
predicted_home_prices = melbourne_model.predict(X)
mean_absolute_error(y, predicted_home_prices)
```

```
Output: 434.71594577146544
```

1.4.2 The Problem with "In-Sample" Scores

The measure we just computed can be called an "in-sample" score. We used a single "sample" of houses for both building the model and evaluating it. Here's why this is bad.
Imagine that, in the large real estate market, door color is unrelated to home price. However, in the sample of data you used to build the model, all homes with green doors were very expensive. The model's job is to find patterns that predict home prices, so it will see this pattern, and it will always predict high prices for homes with

green doors.
Since this pattern was derived from the training data, the model will appear accurate in the training data.
But if this pattern doesn't hold when the model sees new data, the model would be very inaccurate when used in practice.
Since models' practical value come from making predictions on new data, we measure performance on data that wasn't used to build the model. The most straightforward way to do this is to exclude some data from the model-building process, and then use those to test the model's accuracy on data it hasn't seen before. This data is called **validation data**.

1.4.3 Coding It

The scikit-learn library has a function train_test_split to break up the data into two pieces. We'll use some of that data as training data to fit the model, and we'll use the other data as validation data to calculate mean_absolute_error.
Here is the code:

```
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))
```

```
Output: 265806.91478373145
```

Wow!
Your mean absolute error for the in-sample data was about 500 dollars. Out-of-sample it is more than 250,000 dollars.
This is the difference between a model that is almost exactly right, and one that is unusable for most practical purposes. As a point of reference, the average home value in the validation data is 1.1 million dollars. So, the error in new data is about a quarter of the average home value.
There are many ways to improve this model, such as experimenting to find better features or different model types.

1.5 Underfitting and Overfitting

Fine-tune your model for better performance.

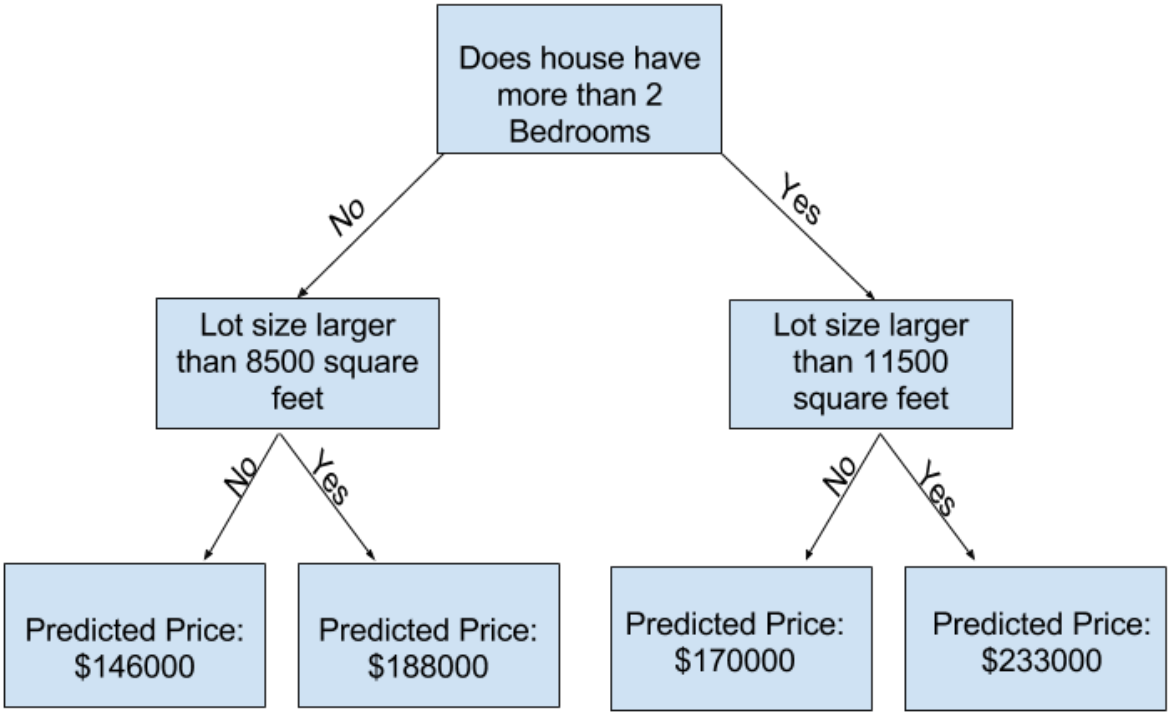
At the end of this step, you will understand the concepts of underfitting and overfitting, and you will be able to apply these ideas to make your models more accurate.

1.5.1 Experimenting With Different Models

Now that you have a reliable way to measure model accuracy, you can experiment with alternative models and see which gives the best predictions. But what alternatives do you have for models?

The most important options determine the tree's depth.

This is a relatively shallow tree.

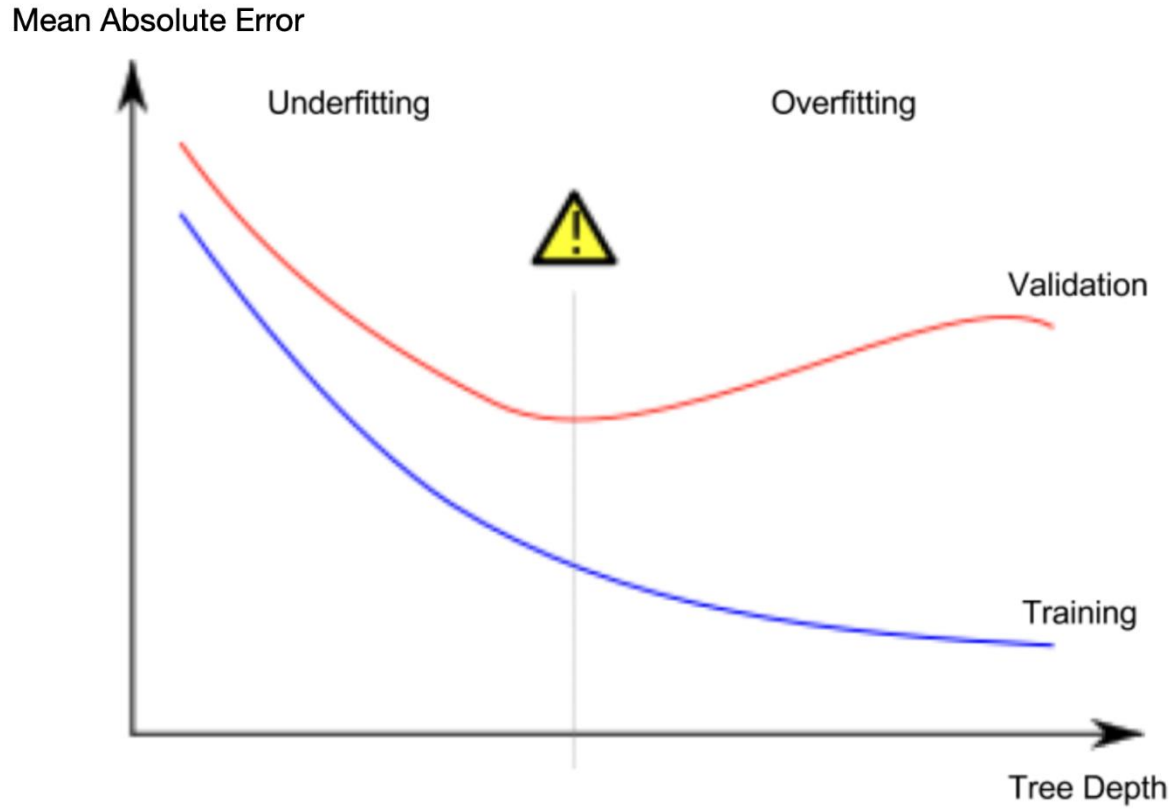


In practice, it's not uncommon for a tree to have 10 splits between the top level (all houses) and a leaf. As the tree gets deeper, the dataset gets sliced up into leaves with fewer houses. If a tree only had 1 split, it divides the data into 2 groups. If each group is split again, we would get 4 groups of houses. Splitting each of those again would create 8 groups. If we keep doubling the number of groups by adding more splits at each level, we'll have 210210 groups of houses by the time we get to the 10th level. That's 1024 leaves.

When we divide the houses amongst many leaves, we also have fewer houses in each leaf. Leaves with very few houses will make predictions that are quite close to those homes' actual values, but they may make very unreliable predictions for new data (because each prediction is based on only a few houses).

This is a phenomenon called **overfitting**, where a model matches the training data almost perfectly, but does poorly in validation and other new data. On the flip side, if we make our tree very shallow, it doesn't divide up the houses into very distinct

groups.
At an extreme, if a tree divides houses into only 2 or 4, each group still has a wide variety of houses. Resulting predictions may be far off for most houses, even in the training data (and it will be bad in validation too for the same reason). When a model fails to capture important distinctions and patterns in the data, it performs poorly even in training data, that is called **underfitting**.
Since we care about accuracy on new data, which we estimate from our validation data, we want to find the sweet spot between underfitting and overfitting. Visually, we want the low point of the (red) validation curve in the figure below.



Example
There are a few alternatives for controlling the tree depth, and many allow for some routes through the tree to have greater depth than other routes. But the *max_leaf_nodes* argument provides a very sensible way to control overfitting vs underfitting. The more leaves we allow the model to make, the more we move from the underfitting area in the above graph to the overfitting area.
We can use a utility function to help compare MAE scores from different values for *max_leaf_nodes*:

```
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 you've

already seen (and which you've already written).

```
# Data Loading Code Runs At This Point
import pandas as pd

# Load data
melbourne_file_path = '../input/melbourne-housing-
snapshot/melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path)
# Filter rows with missing values
filtered_melbourne_data = melbourne_data.dropna(axis=0)
# Choose target and features
y = filtered_melbourne_data.Price
melbourne_features = ['Rooms', 'Bathroom', 'Landsize',
'BuildingArea',
                      'YearBuilt', 'Lattitude', 'Longtitude']
X = filtered_melbourne_data[melbourne_features]

from sklearn.model_selection import train_test_split

# split data into training and validation data, for both features
and target
train_X, val_X, train_y, val_y = train_test_split(X,
y,random_state = 0)
```

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))
```

Output:

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

Of the options listed, 500 is the optimal number of leaves.

1.5.2 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.

1.6 Random Forests

Using a more sophisticated machine learning algorithm.

1.6.1 Introduction

Decision trees leave you with a difficult decision. A deep tree with lots of leaves will overfit because each prediction is coming from historical data from only the few houses at its leaf. But a shallow tree with few leaves will perform poorly because it fails to capture as many distinctions in the raw data.

Even today's most sophisticated modeling techniques face this tension between underfitting and overfitting. But, many models have clever ideas that can lead to better performance. We'll look at the **random forest** as an example.

The random forest uses many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better predictive accuracy than a single decision tree and it works well with default parameters. If you keep modeling, you can learn more models with even better performance, but many of those are sensitive to getting the right parameters.

Example

You've already seen the code to load the data a few times. At the end of data-loading, we have the following variables:

train_X
val_X
train_y
val_y

```
import pandas as pd

# Load data
melbourne_file_path = '../input/melbourne-housing-
snapshot/melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path)
# Filter rows with missing values
melbourne_data = melbourne_data.dropna(axis=0)
# Choose target and features
y = melbourne_data.Price
melbourne_features = ['Rooms', 'Bathroom', 'Landsize',
'BuildingArea',
                      'YearBuilt', 'Lattitude', 'Longtitude']
X = melbourne_data[melbourne_features]

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)
```

We build a random forest model similarly to how we built a 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))
```

```
Output: 191669.7536453626
```

1.6.2 Conclusion

There is likely room for further improvement, but this is a big improvement over the best decision tree error of 250,000. There are parameters which allow you to change the performance of the Random Forest much as we changed the maximum depth of the single decision tree. But one of the best features of Random Forest models is that they generally work reasonably even without this tuning.