INTRO TO MACHINE LEARNING

⇒ The 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**.

⇒ After the model has been fit, you can apply it to new data to **predict** new 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.

```
import pandas as pd
```

⇒ The most important part of the Pandas library is the DataFrame.

```
file_path = '../input/melbourne-housing-snapshot/melb_data.csv'
melbourne_data = pd.read_csv(melbourne_file_path)
melbourne_data.describe()
```

Selecting The Prediction Target¶

⇒ We need to select the column we want to predict, which is called the **prediction target**. By convention, the prediction target is called **y**.

```
y = data.Price
```

Choosing "Features"¶

⇒ The columns that are inputted into our model are called "features. Sometimes, you will use all columns except the target as features.

```
features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', ‘Longtitude']
```

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

```
X = data[features]
```

Building Your Model¶

You will use the **scikit-learn** library to create your models.

The steps to building and using a model are:

- **Define:** What type of model will it be?
- **Fit:** Capture patterns from provided data. This is the heart of modeling.
- **Predict:**
- **Evaluate**: Determine how accurate the model's predictions are.

```
from sklearn.tree import DecisionTreeRegressor

# Define model.
melbourne_model = DecisionTreeRegressor(random_state=1)

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

We now have a fitted model that we can use to make predictions. In practice, you'll want to make predictions for new.

Measuring model quality is the key to iteratively improving your models.

There are many metrics for summarizing model quality, but we'll start with one called **Mean Absolute Error** (also called **MAE**).

The prediction error for each house is:

Error = actual − predicted

With the MAE metric, we take the absolute value of each error. Then we take the average of those absolute errors. This is our model quality.

```
from sklearn.metrics import mean_absolute_error

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

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.

Since this pattern was derived from the training data, the model will appear accurate in the training data.

Problem -> make predictions with their training data and compare those predictions to the target values in the training data is a big mistake.

But if this pattern doesn't hold when the model sees new data, the model would be very inaccurate when used in practice.

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**.

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.

```
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)

melbourne_model = DecisionTreeRegressor()

melbourne_model.fit(train_X, train_y)

val_predictions = melbourne_model.predict(val_X)

print(mean_absolute_error(val_y, val_predictions))
```

Overfitting, where a model matches the training data almost perfectly, but does poorly in validation and other new data.

When a model fails to capture important distinctions and patterns in the data, so 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.

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)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)
```

We can use a for-loop to compare the accuracy of models built with different values for 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 : %d Mean Absolute Error:%d"  % (max_leaf_nodes, my_mae))
```

We use **validation** data, which isn't used in model training, to measure a candidate model's accuracy.

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**.

The random forest uses many trees. It generally has much better predictive accuracy than a single decision tree, using the RandomForestRegressor.

Step 1: Gather the data.

Step 2: Prepare the data - Deal with missing values and categorical data.

Step 4: Train the model - Fit decision trees and random forests to patterns in training data.

Step 5: Evaluate the model - Use a validation set to assess how well a trained model performs on unseen data.

Step 6: Tune parameters - Tune parameters to get better performance from XGBoost models.

Step 7: Get predictions - Generate predictions with a trained model and submit your results to a Kaggle competition.

Automated machine learning (AutoML) -> **Google Cloud AutoML Tables** to automate the machine learning process.