Introduction

Nous commencerons avec un modèle appelé l'Arbre de Décision. C'est un modèle simple à comprendre et c'est la base de certains des meilleurs modèles en science des données.

Pour commencer, nous utiliserons l'arbre de décision le plus simple. Il divise les maisons en deux catégories, et le prix prédit est la moyenne historique des prix des maisons de chaque catégorie. Ce processus s'appelle l'entraînement du modèle, et les données utilisées sont les données d'entraînement. Une fois le modèle entraîné, il peut prédire les prix de nouvelles maisons.

Cependant, ce modèle a des limites puisqu'il ne prend pas en compte d'autres facteurs comme la taille du terrain, le nombre de salles de bain, etc.

Un arbre plus complexe, avec plus de divisions, peut prendre en compte d'autres facteurs, comme la taille du terrain. Vous prédisez le prix d'une maison en suivant le chemin correspondant à ses caractéristiques jusqu'à une feuille, où le prix est déterminé.

Il est maintenant temps de consulter les données avec lesquelles vous allez travailler.

Example

In [1]:
```
import pandas as pd
```

Le DataFrame est l'élément central de Pandas, similaire à un tableau dans Excel ou une base de données SQL. Pandas propose des méthodes puissantes pour manipuler ce type de données. L'exemple donné concerne les prix des maisons, que l'on charge et explore via des commandes spécifiques.

In [2]:
```
# save filepath to variable for easier access
me_file_path = './immodata.csv'
# read the data and store data in DataFrame titled me_data
me_data = pd.read_csv(me_file_path)
# print a summary of the data in me data
me_data.describe()
```

Out[2]:

	Rooms	Price	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	BuildingArea	YearB
count	13580.000000	1.358000e+04	13580.000000	13580.000000	13580.000000	13580.000000	13518.000000	13580.000000	7130.000000	8205.000
mean	2.937997	1.075684e+06	10.137776	3105.301915	2.914728	1.534242	1.610075	558.416127	151.967650	1964.684
std	0.955748	6.393107e+05	5.868725	90.676964	0.965921	0.691712	0.962634	3990.669241	541.014538	37.273
min	1.000000	8.500000e+04	0.000000	3000.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1196.000
25%	2.000000	6.500000e+05	6.100000	3044.000000	2.000000	1.000000	1.000000	177.000000	93.000000	1940.000
50%	3.000000	9.030000e+05	9.200000	3084.000000	3.000000	1.000000	2.000000	440.000000	126.000000	1970.000
75%	3.000000	1.330000e+06	13.000000	3148.000000	3.000000	2.000000	2.000000	651.000000	174.000000	1999.000
max	10.000000	9.000000e+06	48.100000	3977.000000	20.000000	8.000000	10.000000	433014.000000	44515.000000	2018.000

Les 8 chiffres affichés pour chaque colonne incluent des informations comme le nombre de valeurs non manquantes (count) et la moyenne (mean). L'écart-type (std) montre la dispersion des valeurs. Les valeurs min, 25%, 50%, 75% et max sont des points de repère obtenus en triant les données par ordre croissant, représentant respectivement le minimum, les percentiles (25e, 50e, 75e) et le maximum.

Lorsque votre jeu de données contient trop de variables, il est difficile de tout comprendre. On peut commencer par sélectionner certaines variables avec intuition. Plus tard, des techniques statistiques permettront de prioriser les variables de manière automatique. La propriété columns d'un DataFrame permet de lister toutes les colonnes du jeu de données.

In [3]:
```
import pandas as pd

me_file_path = './immodata.csv'
me_data = pd.read_csv(me_file_path)
me_data.columns
```

Out[3]:
```
Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',
       'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',
       'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude',
       'Longtitude', 'Regionname', 'Propertycount'],
      dtype='object')
```

In [4]:
```
# dropna drops missing values (think of na as "not available")
me_data = me_data.dropna(axis=0)
```

Il existe plusieurs façons de sélectionner un sous-ensemble de données dans Pandas. Pour l'instant, on se concentre sur deux méthodes : la notation par point pour la cible de prédiction et la sélection par liste de colonnes pour les caractéristiques. La cible de prédiction, souvent appelée y, peut être extraite via la notation par point et est stockée dans une Série (un équivalent à une colonne de DataFrame).

In [5]:
```
y = me_data.Price
```

Les colonnes utilisées dans un modèle pour faire des prédictions sont appelées "caractéristiques". Vous pouvez utiliser toutes les colonnes sauf la cible, ou n'en choisir que quelques-unes. Pour commencer, un modèle avec peu de caractéristiques sera construit. Vous pourrez plus tard comparer les modèles en fonction des différentes caractéristiques utilisées. La sélection se fait en listant les noms des colonnes entre crochets.

In [6]:
```
me_features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']
```

In [7]:
```
X = me_data[me_features]
```

In [8]:
```
X.describe()
```

Out[8]:

	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

In [9]:
```
X.head()
```

Out[9]:

	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

Pour construire un modèle, vous utiliserez la bibliothèque scikit-learn (sklearn). Les étapes principales sont : définir le type de modèle, l'entraîner avec les données, faire des prédictions, et évaluer la précision des prédictions. Un exemple montre comment définir et entraîner un modèle d'arbre de décision.

In [10]:
```
from sklearn.tree import DecisionTreeRegressor

# Define model. Specify a number for random_state to ensure same results each run
me_model = DecisionTreeRegressor(random_state=1)

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

Out[10]:
```
DecisionTreeRegressor(random_state=1)
```

Certains modèles de machine learning incluent une part de hasard. En définissant random_state, on garantit des résultats reproductibles, une bonne pratique. Après l'entraînement, le modèle est prêt à faire des prédictions, généralement pour de nouvelles données. Pour l'instant, nous allons tester la fonction de prédiction sur les premières lignes des données d'entraînement.

In [11]:
```
print("Making predictions for the following 5 houses:")
print(X.head())
print("The predictions are")
print(me_model.predict(X.head()))
```

```
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.]
```

La qualité d'un modèle se mesure souvent par sa précision prédictive. Un bon indicateur pour résumer la qualité du modèle est l'Erreur Absolue Moyenne (MAE), qui correspond à la moyenne des écarts absolus entre les valeurs réelles et prédites.

In [12]:
```
# Filter rows with missing price values
filtered_me_data = me_data.dropna(axis=0)
# Choose target and features
y = filtered_me_data.Price
fme_features = ['Rooms', 'Bathroom', 'Landsize', 'BuildingArea',
                        'YearBuilt', 'Lattitude', 'Longtitude']
X = filtered_me_data[fme_features]

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

Out[12]:
```
▾ DecisionTreeRegressor

DecisionTreeRegressor()
```

In [13]:
```
from sklearn.metrics import mean_absolute_error

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

Out[13]:
```
434.71594577146544
```

On utilise des données de validation qui n'ont pas servi à la construction du modèle. La fonction train_test_split de scikit-learn permet de diviser les données en deux parties : une pour l'entraînement, et l'autre pour valider l'exactitude du modèle.

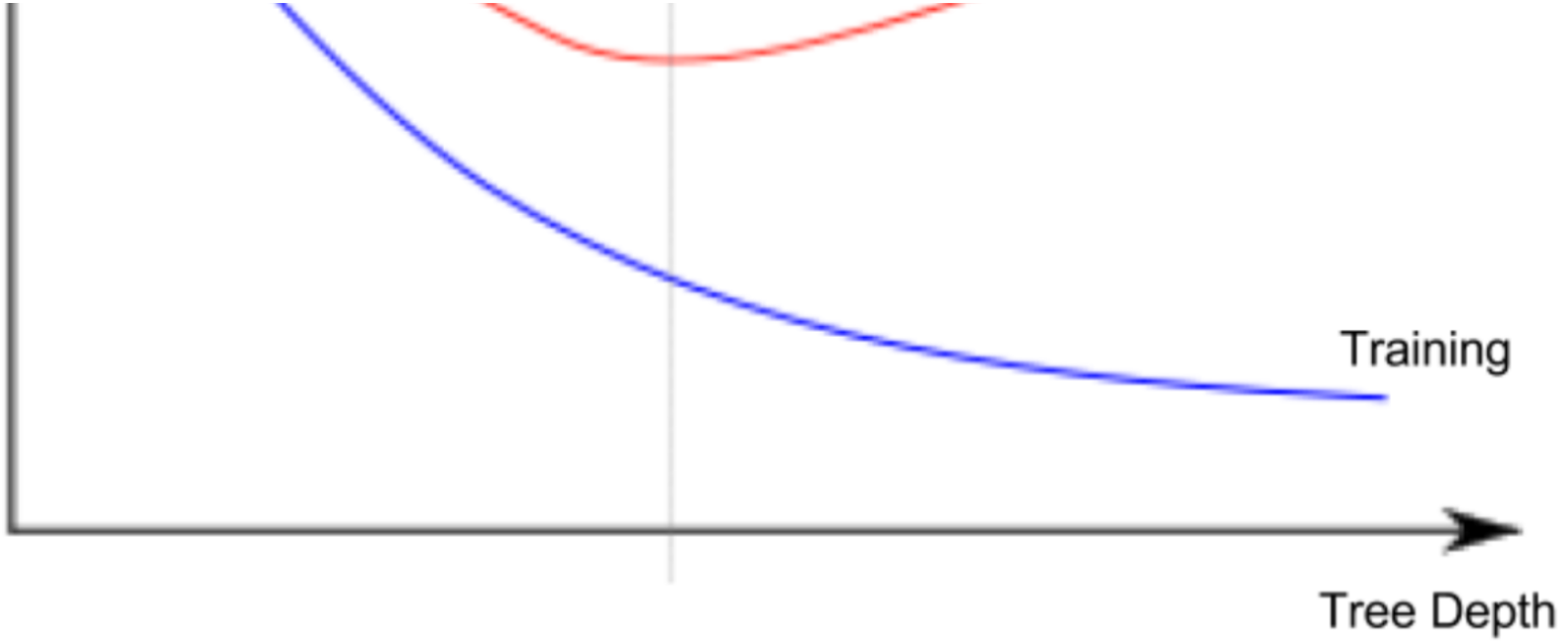
In [14]:
```
from sklearn.model_selection import train_test_split

# 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
me_model = DecisionTreeRegressor()
# Fit model
me_model.fit(train_X, train_y)

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

```
266321.08521626855
```

Le sur-ajustement se produit lorsqu'un modèle correspond presque parfaitement aux données d'entraînement mais est imprécis pour de nouvelles données. Le sous-ajustement se produit lorsqu'un modèle ne capture pas les distinctions importantes, même dans les données d'entraînement. En ajustant la profondeur de l'arbre, avec l'argument max_leaf_nodes par exemple, on peut trouver un équilibre entre ces deux extrêmes et améliorer la précision du modèle.



In [15]:
```
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)
```

In [16]:
```
# Data Loading Code Runs At This Point
import pandas as pd

# Load data
me_file_path = './immodata.csv'
me_data = pd.read_csv(me_file_path)
# Filter rows with missing values
filtered_me_data = me_data.dropna(axis=0)
# Choose target and features
y = filtered_me_data.Price
fme_features = ['Rooms', 'Bathroom', 'Landsize', 'BuildingArea',
                        'YearBuilt', 'Lattitude', 'Longtitude']
X = filtered_me_data[fme_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)
```

In [17]:
```
# compare MAE with differing values of max_leaf_nodes
for max_leaf_nodes in [5, 50, 150, 250, 500, 1500, 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: 150             Mean Absolute Error:  253766
Max leaf nodes: 250             Mean Absolute Error:  247206
Max leaf nodes: 500             Mean Absolute Error:  243495
Max leaf nodes: 1500            Mean Absolute Error:  252130
Max leaf nodes: 5000            Mean Absolute Error:  255575
```

Les arbres de décision présentent un dilemme entre sur-ajustement (trop de feuilles, basées sur peu de données) et sous-ajustement (trop peu de feuilles, ne capturant pas les distinctions dans les données). La forêt aléatoire résout ce problème en utilisant plusieurs arbres et en moyennant leurs prédictions, offrant ainsi une meilleure précision. Elle fonctionne bien avec des paramètres par défaut, et vous pouvez explorer des modèles encore plus performants en ajustant les bons paramètres.

In [18]:
```
import pandas as pd

# Load data
me_file_path = './immodata.csv'
me_data = pd.read_csv(me_file_path)
# Filter rows with missing values
me_data = me_data.dropna(axis=0)
# Choose target and features
y = me_data.Price
fme_features = ['Rooms', 'Bathroom', 'Landsize', 'BuildingArea',
                        'YearBuilt', 'Lattitude', 'Longtitude']
X = me_data[fme_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

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

In [19]:
```
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
```