<https://www.kaggle.com/code/dansbecker/your-first-machine-learning-model/tutorial>

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.

import pandas as pd

*# 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()

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.

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).

melbourne\_data.columns

*# dropna drops missing values (think of na as "not available")*

melbourne\_data = melbourne\_data.dropna(axis=0)

## **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"

The columns that are inputted into our model (and later used to make predictions) are called "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:

In [4]:

linkcode

melbourne\_features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']

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

In [5]:

X = melbourne\_data[melbourne\_features]

X.describe()

# 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? 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)

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.

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()))

# What is Model Validation

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. That is built in a hidden cell below, which you can review by clicking the code button.

from sklearn.metrics import mean\_absolute\_error

predicted\_home\_prices = melbourne\_model.predict(X)

mean\_absolute\_error(y, predicted\_home\_prices)

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**.

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.

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))

Docs:

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>

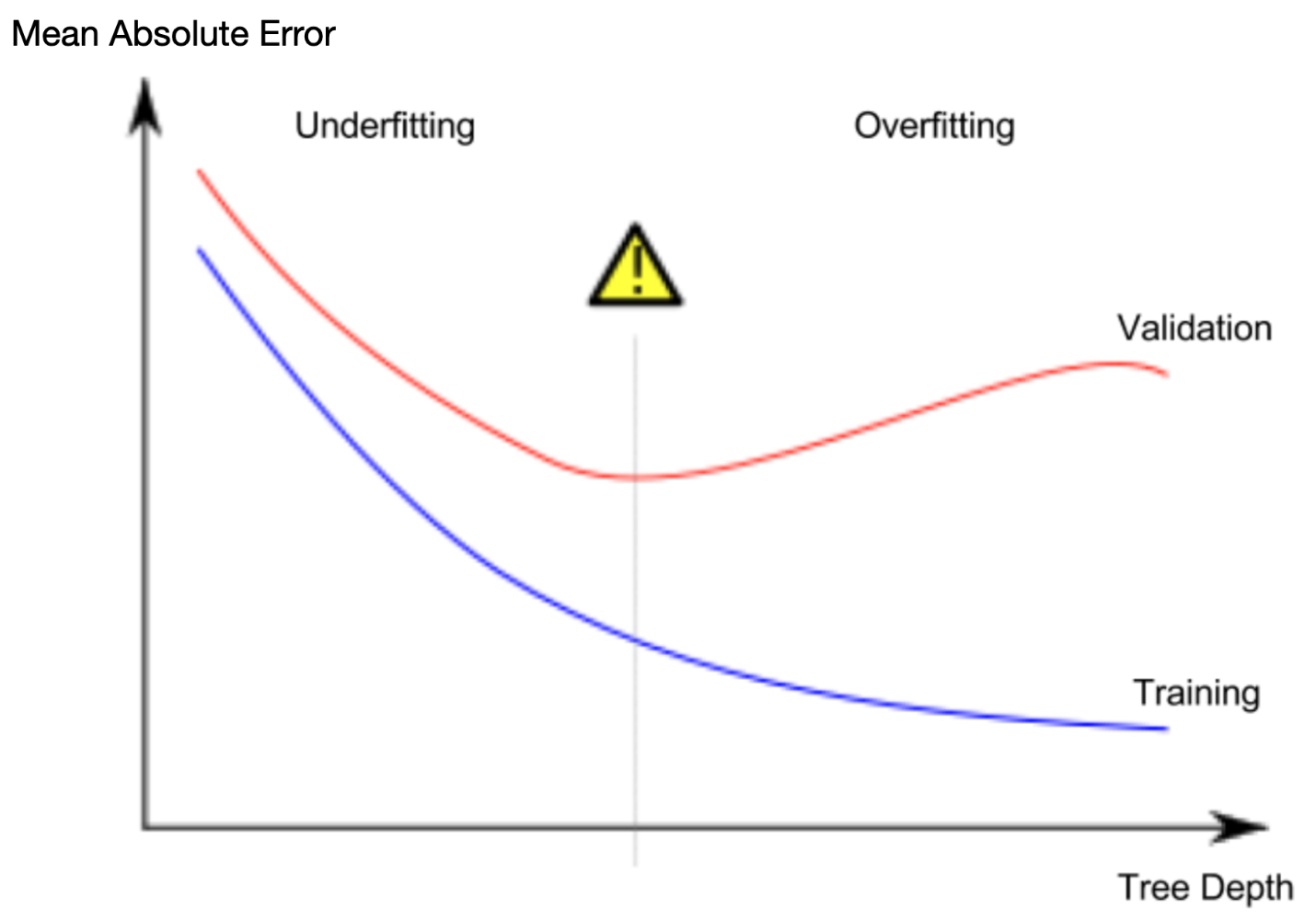
As the tree gets deeper, the dataset gets sliced up into leaves with fewer houses. If a tree only had 1 split, it divides the data into 2 groups. If each group is split again, we would get 4 groups of houses. Splitting each of those again would create 8 groups. If we keep doubling the number of groups 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, 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**. Visually, we want the low point of the (red) validation curve in the figure below.



But the max\_leaf\_nodes argument provides a very sensible way to control overfitting vs underfitting

We can use a utility function to help compare MAE scores from different values for *max\_leaf\_nodes*:

In [1]:

linkcode

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)

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))

The **random forest** uses many trees, and it makes a prediction by averaging the predictions of each component tree.

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))

There are many ways to improve your model, and **experimenting is a great way to learn at this point.**

The best way to improve your model is to add features. To add more **features** to the data, revisit the first code cell, and change this line of code to include more column names:

# Intermediate ML course:

## Test train split:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

# Read the data

X\_full = pd.read\_csv('../input/train.csv', index\_col='Id')

X\_test\_full = pd.read\_csv('../input/test.csv', index\_col='Id')

# Obtain target and predictors

y = X\_full.SalePrice

features = ['LotArea', 'YearBuilt', '1stFlrSF', '2ndFlrSF', 'FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd']

X = X\_full[features].copy()

X\_test = X\_test\_full[features].copy()

# Break off validation set from training data

X\_train, X\_valid, y\_train, y\_valid = train\_test\_split(X, y, train\_size=0.8, test\_size=0.2,

random\_state=0)

from sklearn.ensemble import RandomForestRegressor

## various ways for random forest model init:

<https://www.kaggle.com/code/dansbecker/random-forests>

# Define the models

model\_1 = RandomForestRegressor(n\_estimators=50, random\_state=0)

model\_2 = RandomForestRegressor(n\_estimators=100, random\_state=0)

model\_3 = RandomForestRegressor(n\_estimators=100, criterion='absolute\_error', random\_state=0)

model\_4 = RandomForestRegressor(n\_estimators=200, min\_samples\_split=20, random\_state=0)

model\_5 = RandomForestRegressor(n\_estimators=100, max\_depth=7, random\_state=0)

models = [model\_1, model\_2, model\_3, model\_4, model\_5]

## find MAE to get the best model:

from sklearn.metrics import mean\_absolute\_error

# Function for comparing different models

def score\_model(model, X\_t=X\_train, X\_v=X\_valid, y\_t=y\_train, y\_v=y\_valid):

model.fit(X\_t, y\_t)

preds = model.predict(X\_v)

return mean\_absolute\_error(y\_v, preds)

for i in range(0, len(models)):

mae = score\_model(models[i])

print("Model %d MAE: %d" % (i+1, mae))

## use the best model to use for submission:

# Fit the model to the training data

my\_model.fit(X, y)

# Generate test predictions

preds\_test = my\_model.predict(X\_test)

# Save predictions in format used for competition scoring

output = pd.DataFrame({'Id': X\_test.index,

'SalePrice': preds\_test})

output.to\_csv('submission.csv', index=False)

# Missing Values

### 1. **Deletion**

* **Listwise Deletion:** Remove entire rows where any data is missing.
* **Pairwise Deletion:** Use available data while ignoring any instances where data is missing, typically used in statistical analyses.
* **Drop Columns:** Remove entire columns that have a high percentage of missing values.

### 2. **Imputation**

* **Mean/Median/Mode Imputation:** Replace missing values with the mean, median, or mode of the column. Mean is typically used for continuous data, while median or mode is used for skewed continuous data or categorical data, respectively.
* **Constant Value:** Replace missing values with a constant. This can be a default value, like 0, a particular token, or a value like -999 to denote missingness.
* **Last Observation Carried Forward (LOCF) or Next Observation Carried Backward (NOCB):** Useful in time-series data where the previous or next data point is used as an imputation.

### 3. **Predictive Models**

* **Regression Imputation:** Use a regression model to predict and fill in missing values based on other available variables in the dataset.
* **K-Nearest Neighbors (KNN):** The missing values are imputed using the mean value from the nearest neighbors found in the dataset.
* **Decision Tree Imputation:** Trees or ensemble methods like random forests and gradient boosting can predict missing values based on the non-missing data.

### 4. **Iterative Imputation**

* **Multivariate Imputation by Chained Equations (MICE):** Performs multiple imputations using chained equations. Each missing variable is imputed using a regression model with other variables as predictors, iterating over each variable.
* **Deep Learning Models:** Neural networks, especially those using autoencoders, can be trained to predict missing values effectively.

### 5. **Using Algorithms Robust to Missing Values**

* Some algorithms can handle missing values inherently, such as XGBoost and LightGBM, which can treat missing values as a separate category or use surrogate splits.

### 6. **Probabilistic Imputation**

* **Stochastic Imputation:** Instead of filling missing entries with the same value, each missing value is filled with a value drawn from an estimated distribution.
* **Expectation-Maximization (EM) Algorithm:** A probabilistic model to estimate the likelihood of missing data points.

### 7. **Imputation Using Auxiliary Datasets**

* If similar datasets are available, these can sometimes be used to estimate missing values in your data.

### 8. **Row Mean/Column Mean Imputation**

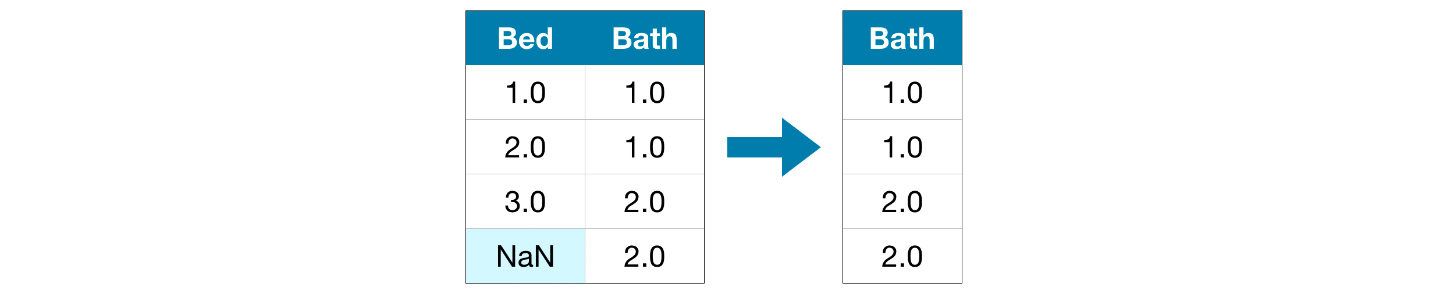
* Replace missing data in a row or column with the average of the rest of the data in that row or column, often used in specific types of data like survey data.

### Considerations

* **Data Type and Distribution:** The choice of method often depends on the nature of the data (numerical, categorical) and its distribution.
* **Missing Data Mechanism:** Understanding why data is missing (missing completely at random, missing at random, or missing not at random) can influence the choice of method.
* **Impact on Analysis:** Consider how each method might bias your analyses or affect the results.

## Three Approaches:

### A Simple Option: Drop Columns with Missing Values



cols\_with\_missing = [col for col **in** X\_train.columns

if X\_train[col].isnull().any()]

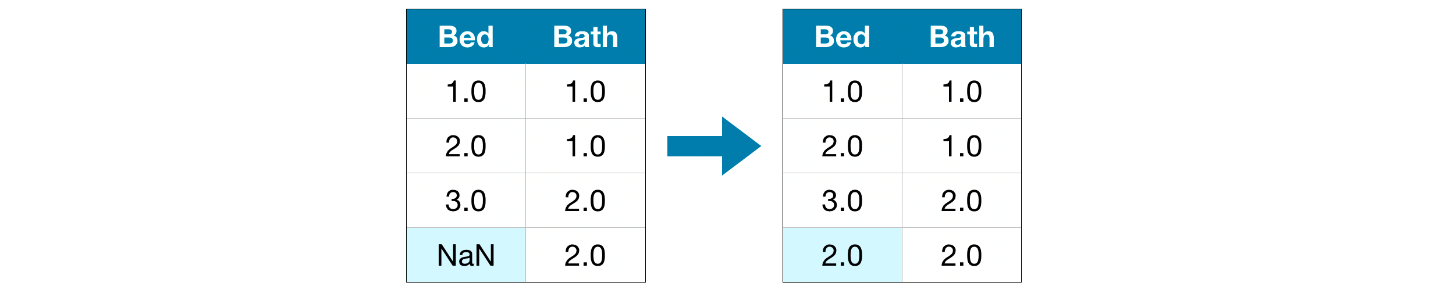
*# Drop columns in training and validation data*

reduced\_X\_train = X\_train.drop(cols\_with\_missing, axis=1)

reduced\_X\_valid = X\_valid.drop(cols\_with\_missing, axis=1)

### 2) A Better Option: Imputation

Imputation fills in the missing values with some number. For instance, we can fill in the mean value along each column.



we use SimpleImputer to replace missing values with the mean value along each column.

regression imputation, is also a way to go but it results in not much improvement in sophisticated ML models

from sklearn.impute import SimpleImputer

*# Imputation*

my\_imputer = SimpleImputer()

#si = SimpleImputer(strategy='median')

imputed\_X\_train = pd.DataFrame(my\_imputer.fit\_transform(X\_train))

imputed\_X\_valid = pd.DataFrame(my\_imputer.transform(X\_valid))

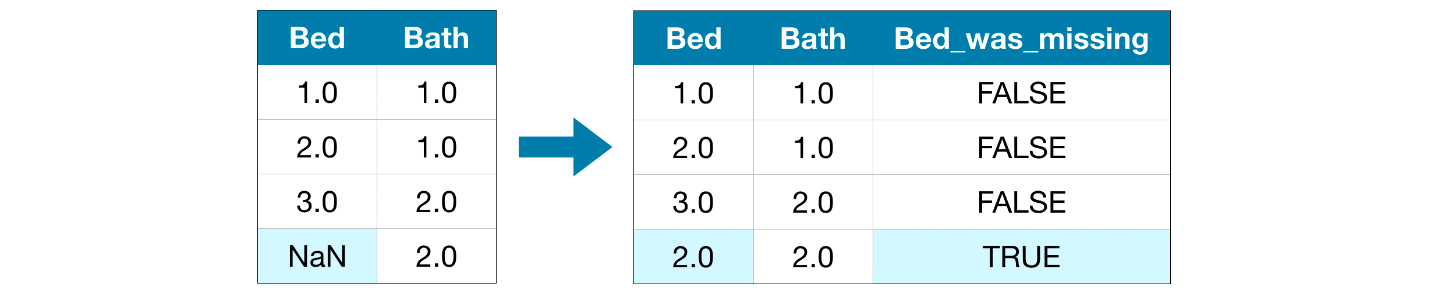
*# Imputation removed column names; put them back*

imputed\_X\_train.columns = X\_train.columns

imputed\_X\_valid.columns = X\_valid.columns

### 3) An Extension To Imputation

imputed values may be systematically above or below their actual values (which weren't collected in the dataset). Or rows with missing values may be unique in some other way.



In this approach, we impute the missing values, as before. And, additionally, for each column with missing entries in the original dataset, we add a new column that shows the location of the imputed entries.

*# Make copy to avoid changing original data (when imputing)*

X\_train\_plus = X\_train.copy()

X\_valid\_plus = X\_valid.copy()

*# Make new columns indicating what will be imputed*

for col **in** cols\_with\_missing:

X\_train\_plus[col + '\_was\_missing'] = X\_train\_plus[col].isnull()

X\_valid\_plus[col + '\_was\_missing'] = X\_valid\_plus[col].isnull()

*# Imputation*

my\_imputer = SimpleImputer()

imputed\_X\_train\_plus = pd.DataFrame(my\_imputer.fit\_transform(X\_train\_plus))

imputed\_X\_valid\_plus = pd.DataFrame(my\_imputer.transform(X\_valid\_plus))

*# Imputation removed column names; put them back*

imputed\_X\_train\_plus.columns = X\_train\_plus.columns

imputed\_X\_valid\_plus.columns = X\_valid\_plus.columns

## Metadata extraction:

# Shape of training data (num\_rows, num\_columns)

print(X\_train.shape)

# Number of missing values in each column of training data

missing\_val\_count\_by\_column = (X\_train.isnull().sum())

print(missing\_val\_count\_by\_column[missing\_val\_count\_by\_column > 0])

# Categorical Variables

## Three Approaches

### 1) Drop Categorical Variables

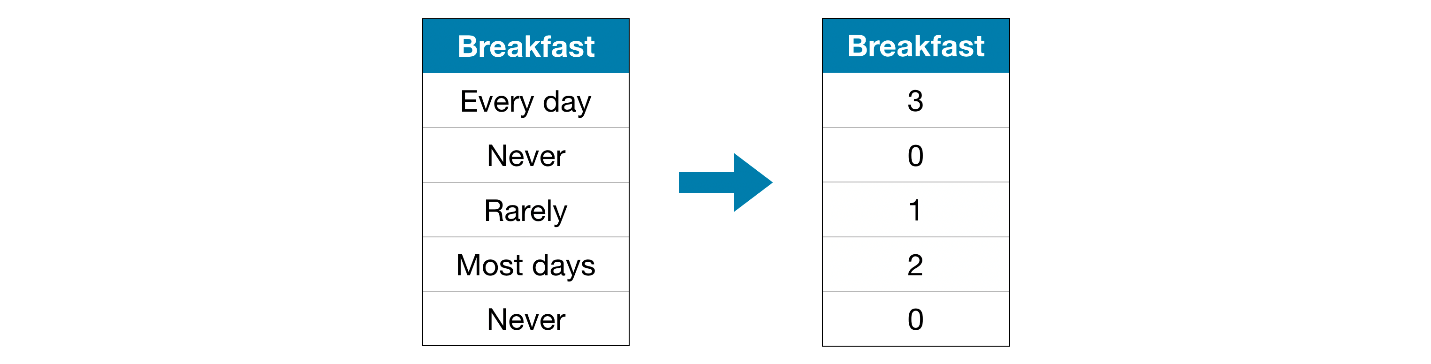
The easiest approach to dealing with categorical variables is to simply remove them from the dataset. This approach will only work well if the columns did not contain useful information.

drop\_X\_train = X\_train.select\_dtypes(exclude=['object'])

drop\_X\_valid = X\_valid.select\_dtypes(exclude=['object'])

### 2) Ordinal Encoding

Ordinal encoding assigns each unique value to a different integer.



This approach assumes an ordering of the categories: "Never" (0) < "Rarely" (1) < "Most days" (2) < "Every day" (3).

from sklearn.preprocessing import OrdinalEncoder

*# Make copy to avoid changing original data*

label\_X\_train = X\_train.copy()

label\_X\_valid = X\_valid.copy()

*# Apply ordinal encoder to each column with categorical data*

ordinal\_encoder = OrdinalEncoder()

label\_X\_train[object\_cols] = ordinal\_encoder.fit\_transform(X\_train[object\_cols])

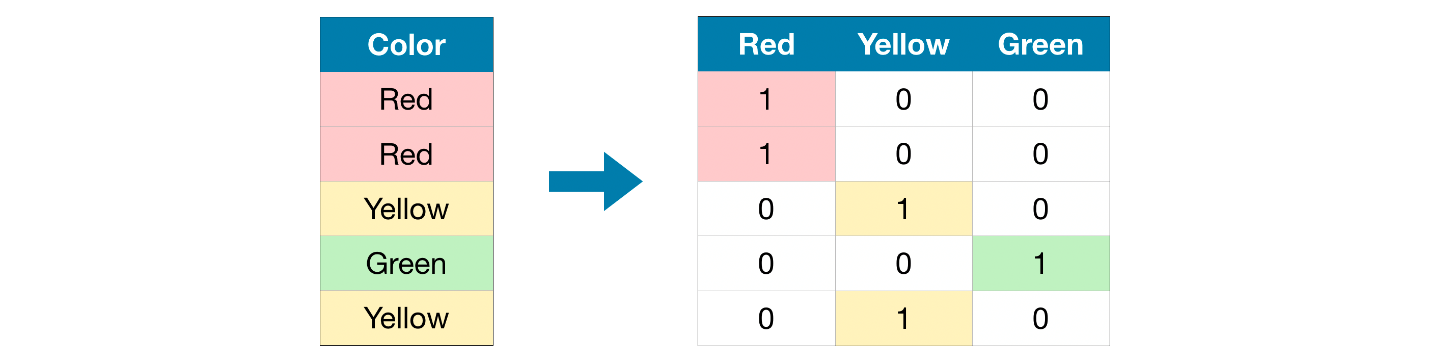
label\_X\_valid[object\_cols] = ordinal\_encoder.transform(X\_valid[object\_cols])

### 3) One-Hot Encoding

One-hot encoding creates new columns indicating the presence (or absence) of each possible value in the original data. e.g., "Red" is neither more nor less than "Yellow")

We refer to categorical variables without an intrinsic ranking as nominal variables.

One-hot encoding generally does not perform well if the categorical variable takes on a large number of values



We set handle\_unknown='ignore' to avoid errors when the validation data contains classes that aren't represented in the training data, and

setting sparse=False ensures that the encoded columns are returned as a numpy array (instead of a sparse matrix).

from sklearn.preprocessing import OneHotEncoder

*# Apply one-hot encoder to each column with categorical data*

OH\_encoder = OneHotEncoder(handle\_unknown='ignore', sparse=False)

OH\_cols\_train = pd.DataFrame(OH\_encoder.fit\_transform(X\_train[object\_cols]))

OH\_cols\_valid = pd.DataFrame(OH\_encoder.transform(X\_valid[object\_cols]))

*# One-hot encoding removed index; put it back*

OH\_cols\_train.index = X\_train.index

OH\_cols\_valid.index = X\_valid.index

*# Remove categorical columns (will replace with one-hot encoding)*

num\_X\_train = X\_train.drop(object\_cols, axis=1)

num\_X\_valid = X\_valid.drop(object\_cols, axis=1)

*# Add one-hot encoded columns to numerical features*

OH\_X\_train = pd.concat([num\_X\_train, OH\_cols\_train], axis=1)

OH\_X\_valid = pd.concat([num\_X\_valid, OH\_cols\_valid], axis=1)

*# Ensure all columns have string type*

OH\_X\_train.columns = OH\_X\_train.columns.astype(str)

OH\_X\_valid.columns = OH\_X\_valid.columns.astype(str)

# **Drop** columns with missing values (simplest approach)

cols\_with\_missing = [col for col in X\_train\_full.columns if X\_train\_full[col].isnull().any()]

X\_train\_full.drop(cols\_with\_missing, axis=1, inplace=True)

# "**Cardinality**" means the number of unique values in a column

# Select categorical columns with relatively low cardinality (convenient but arbitrary)

low\_cardinality\_cols = [cname for cname in X\_train\_full.columns if X\_train\_full[cname].nunique() < 10 and

X\_train\_full[cname].dtype == "object"]

# Select numerical columns

numerical\_cols = [cname for cname in X\_train\_full.columns if X\_train\_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only

my\_cols = low\_cardinality\_cols + numerical\_cols

X\_train = X\_train\_full[my\_cols].copy()

X\_valid = X\_valid\_full[my\_cols].copy()

# Categorical columns in the training data

object\_cols = [col for col in X\_train.columns if X\_train[col].dtype == "object"]

### Finding the bad and good columns:

# Columns that can be safely ordinal encoded

good\_label\_cols = [col for col in object\_cols if

set(X\_valid[col]).issubset(set(X\_train[col]))]

# Problematic columns that will be dropped from the dataset

bad\_label\_cols = list(set(object\_cols)-set(good\_label\_cols))

print('Categorical columns that will be ordinal encoded:', good\_label\_cols)

print('\nCategorical columns that will be dropped from the dataset:', bad\_label\_cols)

#### # Get number of unique entries in each column with categorical data

object\_nunique = list(map(lambda col: X\_train[col].nunique(), object\_cols))

d = dict(zip(object\_cols, object\_nunique))

# Print number of unique entries by column, in ascending order

sorted(d.items(), key=lambda x: x[1])

The output above shows, for each column with categorical data, the number of unique values in the column

# Pipeline:

Pipelines are a simple way to keep your data preprocessing and modeling code organized. Specifically, a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step.

* Cleaner Code: Accounting for data at each step of preprocessing can get messy. With a pipeline, you won't need to manually keep track of your training and validation data at each step.
* Fewer Bugs: There are fewer opportunities to misapply a step or forget a preprocessing step.
* Easier to Productionize: It can be surprisingly hard to transition a model from a prototype to something deployable at scale. We won't go into the many related concerns here, but pipelines can help.
* More Options for Model Validation: You will see an example in the next tutorial, which covers cross-validation.

We construct the full pipeline in three steps:

## Step 1: Define Preprocessing Steps

we use the ColumnTransformer class to bundle together different preprocessing steps.

* imputes missing values in numerical data, and
* imputes missing values and applies a one-hot encoding to categorical data.

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import OneHotEncoder

*# Preprocessing for numerical data*

numerical\_transformer = SimpleImputer(strategy='constant')

*# Preprocessing for categorical data*

categorical\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='most\_frequent')),

('onehot', OneHotEncoder(handle\_unknown='ignore'))

])

*# Bundle preprocessing for numerical and categorical data*

preprocessor = ColumnTransformer(

transformers=[

('num', numerical\_transformer, numerical\_cols),

('cat', categorical\_transformer, categorical\_cols)

])

## Step 2: Define the Model

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n\_estimators=100, random\_state=0)

## Step 3: Create and Evaluate the Pipeline

* With the pipeline, we preprocess the training data and fit the model in a single line of code. (In contrast, without a pipeline, we have to do imputation, one-hot encoding, and model training in separate steps. This becomes especially messy if we have to deal with both numerical and categorical variables!)
* With the pipeline, we supply the unprocessed features in X\_valid to the predict() command, and the pipeline automatically preprocesses the features before generating predictions. (However, without a pipeline, we have to remember to preprocess the validation data before making predictions.)

from sklearn.metrics import mean\_absolute\_error

*# Bundle preprocessing and modeling code in a pipeline*

my\_pipeline = Pipeline(steps=[('preprocessor', preprocessor),

('model', model)

])

*# Preprocessing of training data, fit model*

my\_pipeline.fit(X\_train, y\_train)

*# Preprocessing of validation data, get predictions*

preds = my\_pipeline.predict(X\_valid)

*# Evaluate the model*

score = mean\_absolute\_error(y\_valid, preds)

print('MAE:', score)

# Cross-Validation

A better way to test your models.

What is cross-validation?

In cross-validation, we run our modeling process on different subsets of the data to get multiple measures of model quality.

* For small datasets, where extra computational burden isn't a big deal, you should run cross-validation.
* For larger datasets, a single validation set is sufficient. Your code will run faster, and you may have enough data that there's little need to re-use some of it for holdout.

There's no simple threshold for what constitutes a large vs. small dataset. But if your model takes a couple minutes or less to run, it's probably worth switching to cross-validation.

We obtain the cross-validation scores with the cross\_val\_score() function from scikit-learn. We set the number of folds with the cv parameter.

from sklearn.model\_selection import cross\_val\_score

*# Multiply by -1 since sklearn calculates \*negative\* MAE*

scores = -1 \* cross\_val\_score(my\_pipeline, X, y,

cv=5,

scoring='neg\_mean\_absolute\_error')

print("MAE scores:**\n**", scores)

scoring options: more details can be found in the below link

<https://scikit-learn.org/stable/modules/model_evaluation.html>

# Gradient Boosting:

Another ensemble model. Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble.

It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.)

Then, we start the cycle:

* First, we use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, we add the predictions from all models in the ensemble.
* These predictions are used to calculate a loss function (like mean squared error, for instance).
* Then, we use the loss function to fit a new model that will be added to the ensemble. Specifically, we determine model parameters so that adding this new model to the ensemble will reduce the loss. (Side note: The "gradient" in "gradient boosting" refers to the fact that we'll use gradient descent on the loss function to determine the parameters in this new model.)
* Finally, we add the new model to ensemble, and ...
* ... repeat!

## Boilerplate code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

# Read the data

X = pd.read\_csv('../input/train.csv', index\_col='Id')

X\_test\_full = pd.read\_csv('../input/test.csv', index\_col='Id')

# Remove rows with missing target, separate target from predictors

X.dropna(axis=0, subset=['SalePrice'], inplace=True)

y = X.SalePrice

X.drop(['SalePrice'], axis=1, inplace=True)

# Break off validation set from training data

X\_train\_full, X\_valid\_full, y\_train, y\_valid = train\_test\_split(X, y, train\_size=0.8, test\_size=0.2,

random\_state=0)

# "Cardinality" means the number of unique values in a column

# Select categorical columns with relatively low cardinality (convenient but arbitrary)

low\_cardinality\_cols = [cname for cname in X\_train\_full.columns if X\_train\_full[cname].nunique() < 10 and

X\_train\_full[cname].dtype == "object"]

# Select numeric columns

numeric\_cols = [cname for cname in X\_train\_full.columns if X\_train\_full[cname].dtype in ['int64', 'float64']]

# Keep selected columns only

my\_cols = low\_cardinality\_cols + numeric\_cols

X\_train = X\_train\_full[my\_cols].copy()

X\_valid = X\_valid\_full[my\_cols].copy()

X\_test = X\_test\_full[my\_cols].copy()

# One-hot encode the data (to shorten the code, we use pandas)

X\_train = pd.get\_dummies(X\_train)

X\_valid = pd.get\_dummies(X\_valid)

X\_test = pd.get\_dummies(X\_test)

X\_train, X\_valid = X\_train.align(X\_valid, join='left', axis=1)

X\_train, X\_test = X\_train.align(X\_test, join='left', axis=1)

# XGBoost:

XGBoost stands for extreme gradient boosting

we import the scikit-learn API for XGBoost (xgboost.XGBRegressor).

from xgboost import XGBRegressor

my\_model = XGBRegressor()

my\_model.fit(X\_train, y\_train)

from sklearn.metrics import mean\_absolute\_error

predictions = my\_model.predict(X\_valid)

print("Mean Absolute Error: " + str(mean\_absolute\_error(predictions, y\_valid)))

## Parameter Tuning:

**n\_estimators** specifies how many times to go through the modeling cycle described above. It is equal to the number of models that we include in the ensemble.

* Too low a value causes underfitting, which leads to inaccurate predictions on both training data and test data.
* Too high a value causes overfitting, which causes accurate predictions on training data, but inaccurate predictions on test data (which is what we care about).

Typical values range from 100-1000, though this depends a lot on the learning\_rate parameter discussed below.

my\_model = XGBRegressor(n\_estimators=500)

my\_model.fit(X\_train, y\_train)

**early\_stopping\_rounds** offers a way to automatically find the ideal value for n\_estimators. Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n\_estimators. It's smart to set a high value for n\_estimators and then use early\_stopping\_rounds to find the optimal time to stop iterating

we stop after 5 straight rounds of deteriorating validation scores.

When using early\_stopping\_rounds, you also need to set aside some data for calculating the validation scores - this is done by setting the eval\_set parameter.

my\_model = XGBRegressor(n\_estimators=500)

my\_model.fit(X\_train, y\_train,

early\_stopping\_rounds=5,

eval\_set=[(X\_valid, y\_valid)],

verbose=False)

**learning\_rate**

Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number (known as the learning rate) before adding them in.

This means each tree we add to the ensemble helps us less. So, we can set a higher value for n\_estimators without overfitting. If we use early stopping, the appropriate number of trees will be determined automatically.

In general, a small learning rate and large number of estimators will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle. As default, XGBoost sets learning\_rate=0.1.

my\_model = XGBRegressor(n\_estimators=1000, learning\_rate=0.05)

my\_model.fit(X\_train, y\_train,

early\_stopping\_rounds=5,

eval\_set=[(X\_valid, y\_valid)],

verbose=False)

**n\_jobs**

On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n\_jobs equal to the number of cores on your machine. On smaller datasets, this won't help.

my\_model = XGBRegressor(n\_estimators=1000, learning\_rate=0.05, n\_jobs=4)

my\_model.fit(X\_train, y\_train,

early\_stopping\_rounds=5,

eval\_set=[(X\_valid, y\_valid)],

verbose=False)

# Data Leakage:

Data leakage refers to a situation in machine learning where information from outside the training dataset is used to create the model. This can cause the model to perform unusually well on the training set but perform poorly on real-world or unseen data, essentially because it has inadvertently been given access to information it wouldn't normally have. Data leakage can occur at any stage of the data preparation, model training, or validation process, and it can lead to misleadingly optimistic performance metrics and poor generalization of the model to new data.

### Types of Data Leakage

1. **Leakage Through Predictors (Features):**
   * **Future Information:** Using data in features that would not be available at prediction time. For instance, using information from a future event to predict an outcome related to an earlier event.
   * **Incorrect Aggregation:** Aggregating data across multiple rows that include the target data. For example, using the average spending of a customer (including the target transaction) to predict whether they will spend above a certain amount on a new transaction.
2. **Leakage Through Preprocessing:**
   * **Preprocessing Before Train-Test Split:** Applying data transformation steps like normalization, filling missing values, or feature selection before splitting the data into training and test sets. This means the test set could influence how the transformations are applied to the training set.
   * **Improper Cross-validation:** When using cross-validation, it’s important that the data preprocessing (like scaling or imputing) is done within each fold separately to avoid leakage from the validation fold back into the training fold.

### Examples of Data Leakage

* **Time series forecasting:** When predicting future sales, including data from after the date of the prediction (like using the average of future sales as a feature) results in leakage.
* **Healthcare:** If a dataset to predict patient outcomes includes information gathered after the outcome occurred (like a treatment outcome), this constitutes leakage.
* **Financial applications:** Using the result of a market event (like a stock split or a merger announcement) before the event has occurred in your predictive model.

### How to Prevent Data Leakage

* **Careful Feature Selection:** Ensure all features used in training are available at the moment you want to make predictions in real-world scenarios.
* **Proper Data Splitting:** Apply any data preprocessing steps (like scaling or imputing missing values) after splitting the data into training and test sets. Use pipelines to ensure transformations are fit only on training data and then applied to any data subsequently (like test data).
* **Use of Pipelines:** In frameworks like scikit-learn, use pipelines to ensure that steps like scaling and normalization are confined to the data they should be applied to during cross-validation.
* **Domain Knowledge:** Leverage domain expertise to understand what data would realistically be available at prediction time and ensure the model only has access to appropriate information.