

## INTERMEDIATE MACHINE LEARNING

- Data types often found in datasets (**missing values**, **categorical variables**),
- Design **pipelines** to improve the quality of your machine learning code.
- Use advanced techniques for model validation (**cross-validation**),
- Build state-of-the-art models (**XGBoost**)
- Avoid common and important data science mistakes (**leakage**).

-----  
Three approaches to **dealing with missing values**.

Most machine learning libraries (including scikit-learn) give an error if you try to build a model using data with missing values. So you'll need to choose one of the strategies below.

### 1) A Simple Option: Drop Columns with Missing Values¶

The simplest option is to drop columns with missing values.

The model loses access to a lot of information with this approach.

```
# Get names of 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)
```

### 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.

The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely.

```
from sklearn.impute import SimpleImputer

# Imputation
my_imputer = SimpleImputer()
imputed_X_train = pd.DataFrame(my_imputer.fit_transform(X_train))
imputed_X_valid = pd.DataFrame(my_imputer.transform(X_valid))
```




### 3) An Extension To Imputation

We add a new column that shows the location of the imputed entries.

However, 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.

Bed	Bath	
1.0	1.0	
2.0	1.0	
3.0	2.0	
NaN	2.0	



Bed	Bath	Bed_was_missing
1.0	1.0	FALSE
2.0	1.0	FALSE
3.0	2.0	FALSE
2.0	2.0	TRUE

*# 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))*

As is common, imputing missing values (in **Approach 2** and **Approach 3**) yielded better results, relative to when we simply dropped columns with missing values (in **Approach 1**).

-----

A **categorical variable** takes only a limited number of values.

⇒ Consider a survey that asks how often you eat breakfast and provides four options: "Never", "Rarely", "Most days", or "Every day". In this case, the data is categorical, because responses fall into a fixed set of categories.

You will get an error if you try to plug these variables into most machine learning models in Python without preprocessing them first.

## 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¶

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).

This assumption makes sense in this example, because there is an indisputable ranking to the categories. Not all categorical variables have a clear ordering in the values, but we refer to those that do as **ordinal variables**.

Scikit-learn has a `OrdinalEncoder` class that can be used to get ordinal encodings

```
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  
)
```

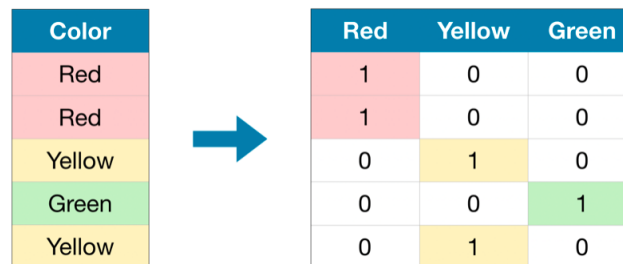
```
label_X_valid[object_cols] = ordinal_encoder.transform(X_valid[object_cols])
```

In the case that the validation data contains values that don't also appear in the training data, the encoder will throw an error, because these values won't have an integer assigned to them.

The simplest approach, is to drop the problematic categorical columns.

### 3) One-Hot Encoding

**One-hot encoding** creates new columns indicating the presence (or absence) of each possible value in the original data.



Color	Red	Yellow	Green
Red	1	0	0
Red	1	0	0
Yellow	0	1	0
Green	0	0	1
Yellow	0	1	0

In contrast to ordinal encoding, one-hot encoding does not assume an ordering of the categories. We refer to categorical variables without an intrinsic ranking as **nominal variables**.

- ⇒ Won't use it for variables taking more than 15 different values
- ⇒ For large datasets with many rows, one-hot encoding can greatly expand the size of the dataset. For this reason, we typically will only one-hot encode columns with relatively **low cardinality**. Then, high cardinality columns can either be dropped from the dataset, or we can use ordinal encoding.
- ⇒ We set **handle\_unknown='ignore'** to avoid errors when the validation data contains classes that aren't represented in the training data.
- ⇒ Setting **sparse=False** ensures that the encoded columns are returned as a numpy array

```
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)
```

**\* To obtain a list of all of the categorical variables in the training data.**

We do this by checking the data type (or dtype) of each column. The object dtype indicates a column has text (there are other things it could theoretically be, but that's unimportant for our purposes). For this dataset, the columns with text indicate categorical variables.

```
# Get list of categorical variables
```

```
s = (X_train.dtypes == 'object')
```

```
object_cols = list(s[s].index)
```

```
print("Categorical variables:")
```

```
print(object_cols)
```

-----

**Pipelines** are a simple way to keep your data preprocessing and modeling code organized.

1. **Cleaner Code:** With a pipeline, you won't need to manually keep track of your training and validation data at each step.
2. **Fewer Bugs**
3. **Easier to Productionize:** It can be surprisingly hard to transition a model from a prototype to something deployable at scale.
4. **More Options for Model Validation:** Cross-validation.

Notice that the data contains both categorical data and columns with missing values. With a pipeline, it's easy to deal with both!

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
- 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¶

Next, we define a random forest model

### Step 3: Create and Evaluate the Pipeline¶

Finally, we use the Pipeline class to define a pipeline that bundles the preprocessing and modeling steps.

⇒ With the pipeline, we preprocess the training data and fit the model in a single line of code.



⇒ 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)

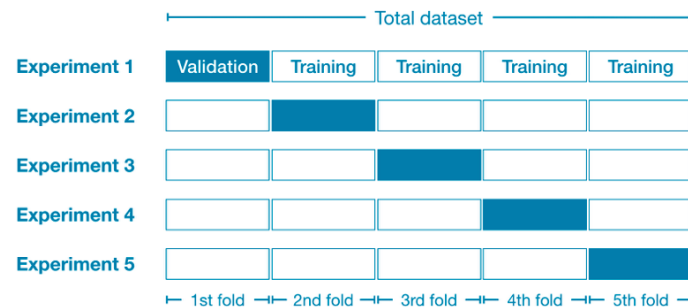
# Evaluate the model
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

---

The larger the validation set, the less the noise there is in our measure of model quality, and the more reliable it will be.

Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models!

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



However, it can take longer to run, because it estimates multiple models (one for each fold).

So, given these tradeoffs, when should you use each approach?

- ⇒ For small datasets, where extra computational isn't a big deal
- ⇒ 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.

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.ensemble import RandomForestRegressor
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer

my_pipeline = Pipeline(steps=[('preprocessor', SimpleImputer()),
                              ('model', RandomForestRegressor(n_estimators=50,
                                                              random_state=0))
                              ])
```

```
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)
```





\* Note: that we no longer need to keep track of separate training and validation sets.

\* We set the number of trees in the random forest model with the `n_estimators` parameter, and setting `random_state` ensures reproducibility.

**Higher number of trees in the random forest model better performance in MAE**

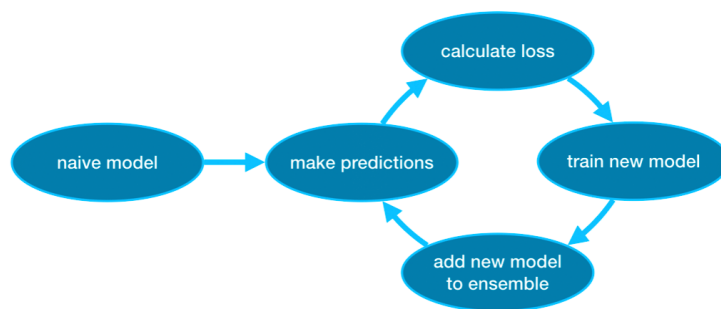
-----

How to build and optimize models with **gradient boosting**

By definition, **ensemble methods** combine the predictions of several models.

**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** subsequent additions to the ensemble will address those errors.



⇒ **XGBoost library**. XGBoost stands for **extreme gradient boosting**, which is an implementation of gradient boosting.



```
from xgboost import XGBRegressor
```

```
my_model = XGBRegressor()
```

**XGBoost has a few parameters:**

**n\_estimators** specifies how many times to go through the modeling cycle described above.

- ⇒ Too low a value causes underfitting
- ⇒ Too high a value causes overfitting, which causes accurate predictions on training data, but inaccurate predictions on test data.

**Typical values range from 100-1000**

```
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.

**Setting early\_stopping\_rounds=5** is a reasonable choice. In this case, we stop after 5 straight rounds of deteriorating validation scores.

```
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.

**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.

```
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)
```

**A small learning rate and large number of estimators will yield more accurate XGBoost models**

⇒ With parameter tuning, you can train highly accurate models

---

**Data leakage** (or **leakage**) happens when your training data contains information about the target, but similar data will not be available when the model is used for prediction.

This leads to high performance on the training set (and possibly even the validation data), but the model will perform poorly in production.



In other words, leakage causes a model to look accurate until you start making decisions with the model, and then the model becomes very inaccurate.

There are two main types of leakage: **target leakage** and **train-test contamination**.

**Target leakage** occurs when your predictors include data that will not be available at the time you make predictions. It is important to think about target leakage in terms of the timing or chronological order.

A different type of leak occurs when you aren't careful to distinguish training data from validation data.

Recall that validation is meant to be a measure of how the model does on data that it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavior. This is sometimes called **train-test contamination**.