Kaggle: Intermediate Machine Learning

**General Approach of ML**

1. Understand ML object & data description

2. Read training/test data

3. Clean data

- Drop missing values or null values or

- Replace missing values or null values with mean of the column/meaningful values

4. Separate target from predictors

5 Divide data into training and validation subsets

6. Preprocess data

7. Generate ML model & fit the ML model to training data

8. Evaluate the ML model with error measure

**Categorical Variables**

approach1: drop columns with categorical data

approach2: ordinal encoding

approach3: one-hot encoding

**Pipelines**

|  |
| --- |
| Step1: Define Preprocessing Steps  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)  ])  Step2: Define model  from sklearn.ensemble import RandomForestRegressor  model = RandomForestRegressor(n\_estimators=100, random\_state=0)  Step3: Create and Evaluate the Pipeline  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**

the drawbacks of data-driven way measuring model quality with a validation set

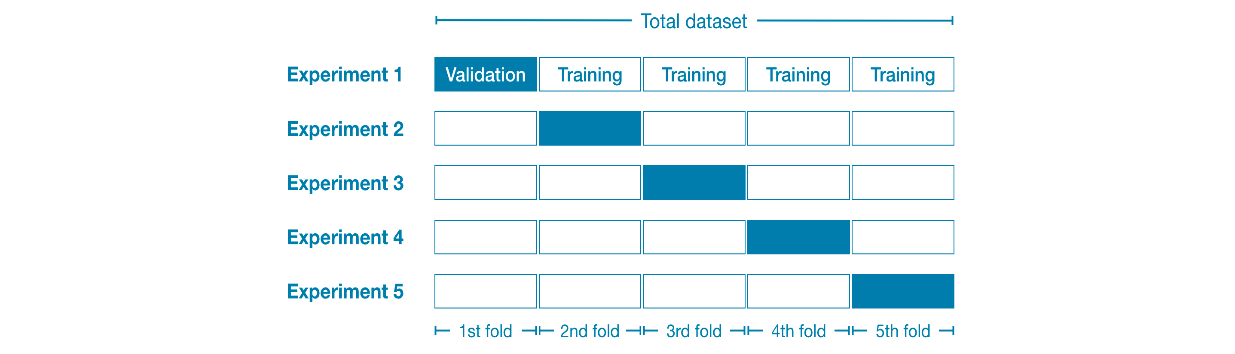
* In general, the larger the validation set, the less randomness (aka "noise") there is in our measure of model quality, and the more reliable it will be. Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models!

Solution for the drawbacks of data-driven way measuring model quality with a validation set

Cross-Validation for better measurements of model performance

What is cross-validation?

* run our modeling process on different subsets of the data to get multiple measures of model quality
* For example, we could begin by dividing the data into 5 pieces, each 20% of the full dataset. In this case, we say that we have broken the data into 5 "**folds**".



Then, we run one experiment for each fold:

* In **Experiment 1**, we use the first fold as a validation (or holdout) set and everything else as training data. This gives us a measure of model quality based on a 20% holdout set.
* In **Experiment 2**, we hold out data from the second fold (and use everything except the second fold for training the model). The holdout set is then used to get a second estimate of model quality.
* We repeat this process, using every fold once as the holdout set. Putting this together, 100% of the data is used as holdout at some point, and we end up with a measure of model quality that is based on all of the rows in the dataset (even if we don't use all rows simultaneously).

When should you use cross-validation?

Cross-validation gives a more accurate measure of model quality, which is especially important if you are making a lot of modeling decisions. 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 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.

Alternatively, you can run cross-validation and see if the scores for each experiment seem close. If each experiment yields the same results, a single validation set is probably sufficient.

\* While it's possible to do cross-validation without pipelines, it is quite difficult! Using a pipeline will make the code remarkably straightforward

|  |
| --- |
| 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') |

If you'd like to learn more about [hyperparameter optimization](https://en.wikipedia.org/wiki/Hyperparameter_optimization), you're encouraged to start with **grid search**, which is a straightforward method for determining the best combination of parameters for a machine learning model. Thankfully, scikit-learn also contains a built-in function [GridSearchCV()](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html" \t "_blank) that can make your grid search code very efficient!

**XGBoost(Extreme Gradient Boosting)**

The most accurate modeling technique for structured data.

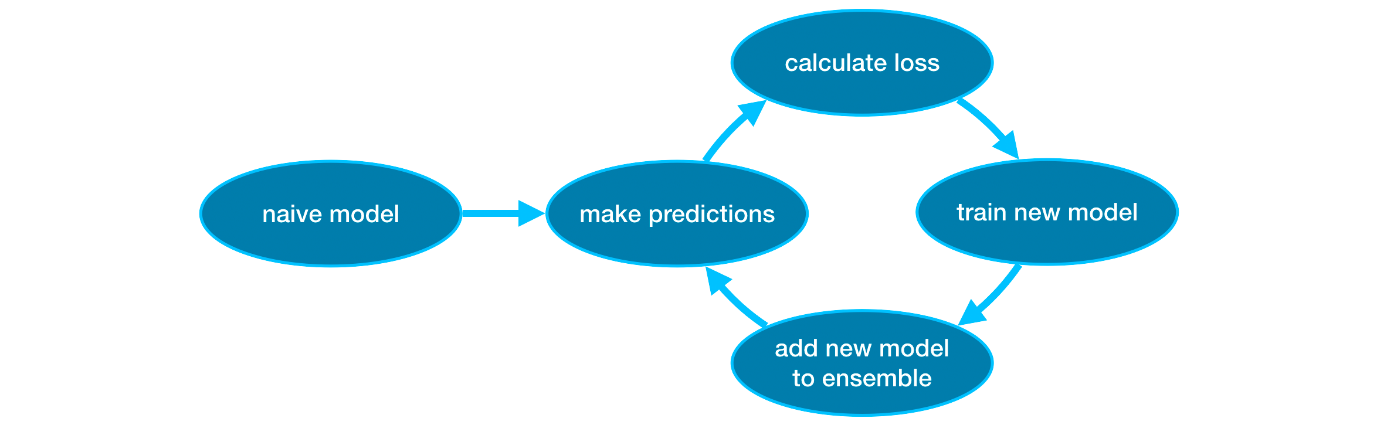
Leading software library for working with standard tabular data (the type of data you store in Pandas DataFrames, as opposed to more exotic types of data like images and videos). With careful parameter tuning, you can train highly accurate models.

Ensemble 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](https://en.wikipedia.org/wiki/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*](https://en.wikipedia.org/wiki/Gradient_descent)*on the loss function to determine the parameters in this new model.*)
* Finally, we add the new model to ensemble, and ...
* ... repeat!



Parameter tuning

the parameters that can dramatically affect accuracy and training speed

* n\_estimators

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.

* early\_stopping\_rounds

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.

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.

If you later want to fit a model with all of your data, set n\_estimators to whatever value you found to be optimal when run with early stopping.

* 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.

* 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.

The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.

**Data Leakage**

**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.**

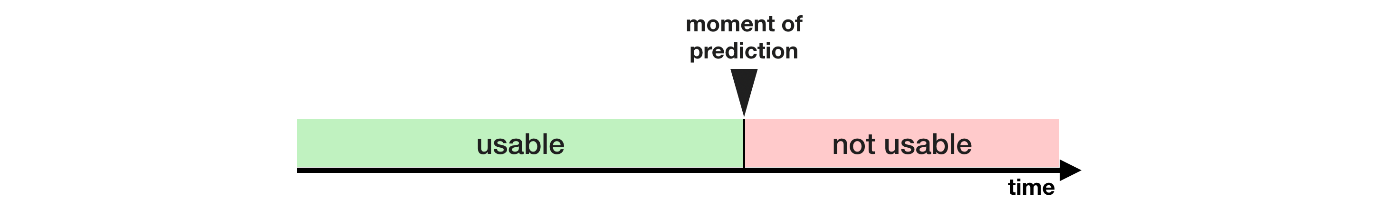
### Target leakage

* **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 that data becomes available, not merely whether a feature helps make good predictions.
* **Example:** Imagine you want to predict who will get sick with pneumonia. The top few rows of your raw data look like this

| got\_pneumonia | age | weight | male | took\_antibiotic\_medicine | ... |
| --- | --- | --- | --- | --- | --- |
| False | 65 | 100 | False | False | ... |
| False | 72 | 130 | True | False | ... |
| True | 58 | 100 | False | True | ... |

People take antibiotic medicines after getting pneumonia in order to recover. The raw data shows a strong relationship between those columns, but took\_antibiotic\_medicine is frequently changed after the value for got\_pneumonia is determined. This is target leakage.

* To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded.



Train-Test Contamination

* 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**.
* If your validation is based on a simple train-test split, exclude the validation data from any type of fitting, including the fitting of preprocessing steps. This is easier if you use scikit-learn pipelines. When using cross-validation, it's even more critical that you do your preprocessing inside the pipeline!

Data leakage can occur at different stages of the machine learning process:

1. **Training Phase:** If information from the test set or future data is inadvertently included in the training data, the model may learn patterns that do not generalize well. For example, using features that would not be available at the time of prediction can lead to overfitting.
2. **Feature Selection:** Leakage can occur when features are selected based on information from the entire dataset, including the test set. This can result in the model learning patterns that do not exist in new, unseen data.
3. **Preprocessing:** Inappropriate data preprocessing steps that involve information from the test set or that rely on knowledge of the target variable can introduce leakage. For instance, scaling features based on statistics computed from the entire dataset can lead to data leakage.

To avoid data leakage in machine learning, it's important to follow best practices:

1. **Split Data Properly:** Divide the dataset into training and testing sets before any preprocessing or feature engineering. Ensure that the test set is not used in any way during model training.
2. **Cross-Validation:** Use techniques like k-fold cross-validation to assess model performance without leakage. This involves splitting the data into multiple folds and training the model on different subsets while evaluating on the remaining folds.
3. **Feature Engineering:** Be cautious when creating new features or transforming existing ones. Ensure that these processes are based solely on information available in the training set.
4. **Temporal Data Handling:** If dealing with time-series data, ensure that the training set includes only information available up to a certain point in time, and the test set includes data from later periods.
5. **Strict Validation Procedures:** When tuning hyperparameters or selecting models, use a separate validation set that is not part of the training set.

Data leakage in machine learning can lead to overly optimistic performance estimates during development but result in poor performance on real-world, unseen data. Therefore, it is crucial to be vigilant about preventing leakage at every stage of the machine learning pipeline.