Cross-Validation

Learn about K-Fold cross-validation and why it's used.

Chapter Goals:

- Learn about the purpose of cross-validation
- Implement a function that applies the K-Fold cross-validation algorithm to a model

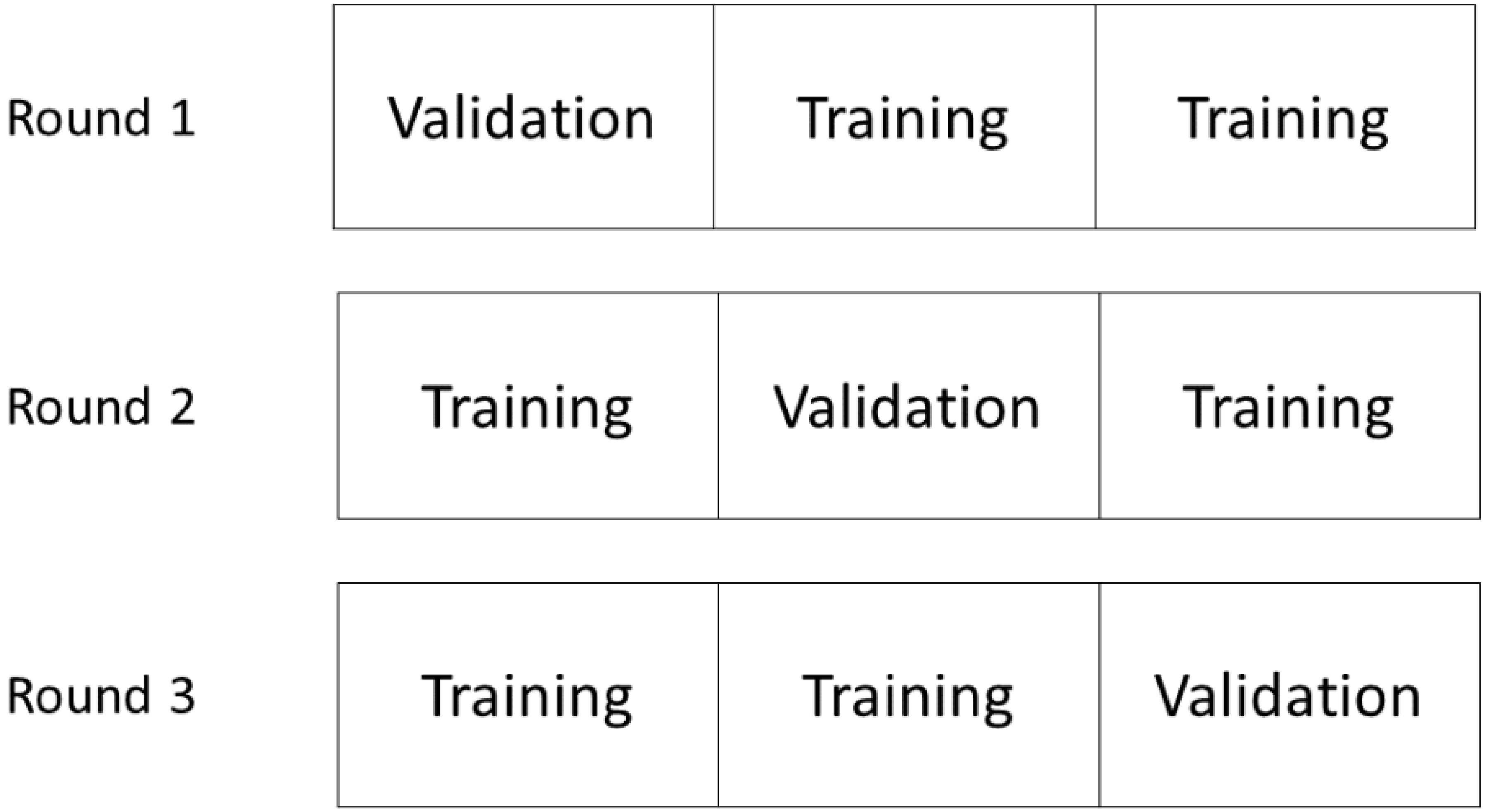
A. Additional evaluation datasets

Sometimes, it's not enough to just have a single testing set for model evaluation. Having additional sets of data for evaluation gives us a more accurate measurement of how good the model is for the original dataset.

If the original dataset is big enough, we can actually split it into three subsets: training, testing, and validation. The validation set is about the same size as the testing set, and it is used for evaluating the model after training. The testing set is then used for final evaluation once the model is done training and tuning.

However, partitioning the original dataset into three distinct sets will cut into the size of the training set. This can reduce the performance of the model if our original dataset is not large enough. A solution to this problem is cross-validation (CV).

Cross-validation creates synthetic validation sets by partitioning the training set into multiple smaller subsets. One of the most common algorithms for cross-validation, K-Fold CV, partitions the training set into **k** approximately equal sized subsets (referred to as *folds*). There are **k** "rounds" of the algorithm, and each "round" chooses one of the **k** subsets for the validation set (a different subset is chosen each round), while the remaining **k - 1** subsets are aggregated into the round's training set and used to train the model.



The K-Fold cross-validation process with 3 folds (k=3)

Each round of the K-Fold algorithm, the model is trained on that round's training set (the combined training folds) and then evaluated on the single validation fold. The evaluation metric depends on the model. For classification models, this is usually classification accuracy on the validation set. For regression models, this can either be the model's mean squared error, mean absolute error, or R^2 value on the validation set.

B. Scored cross-validation

In scikit-learn, we can easily implement K-Fold cross-validation with the `cross_val_score` function (also part of the `model_selection` module). The function returns an array containing the evaluation score for each round.

The code below demonstrates K-Fold CV with 3 folds for classification. The evaluation metric is classification accuracy.

```python
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
clf = linear_model.LogisticRegression()
# Predefined data and labels
cv_score = cross_val_score(clf, data, labels, cv=3)  # k = 3

print('{}\n'.format(repr(cv_score)))
```

RUN SAVE RESET

The code below demonstrates K-Fold CV with 4 folds for regression. The evaluation metric is R^2 value.

```python
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
reg = linear_model.LinearRegression()
# Predefined data and labels
cv_score = cross_val_score(reg, data, labels, cv=4)  # k = 4

print('{}\n'.format(repr(cv_score)))
```

RUN SAVE RESET

Note that we don't call `fit` with the model prior to using `cross_val_score`. This is because the `cross_val_score` function will use `fit` for training the model each round.

For classification models, the `cross_val_score` function will apply a special form of the K-Fold algorithm called *stratified* K-Fold. This just means that each fold will contain approximately the same class distributio as the original dataset. For example, if the original dataset contained 60% class **0** data observations and 40% class **1**, each fold of the stratified K-Fold algorithm will have about the same 60-40 split between class **0** and class **1** data observations.

While cross-validation gives us a better measurement of the model's fit on the original dataset, it can be ver time-consuming when used on large datasets. For large enough datasets, it is better to just split it into training, validation, and testing sets, and then use the validation set for evaluating the model before it is finalized.

`max_depth` value, we print the 95% confidence interval for the cross-validated scores across the 5 folds.

For the most part, the maximum depth of 4 produces the best 95% confidence interval of cross-validated scores. This would be the value of `max_depth` that we choose for the final decision tree.

If the confidence interval had consistently continued to improve for maximum depths of 5, 6 and 7, we would have continued applying the cross-validation process to evaluate larger maximum depth values.

Time to Code!

The coding exercise for this chapter is to complete the aforementioned `cv_decision_tree` function. The function's first argument defines whether the decision tree is for classification/regression, the next two arguments represent the data/labels, and the final two arguments represent the tree's maximum depth and number of folds, respectively.

First, we'll create the decision tree (using the `tree` module imported in the backend).

Initialize `d_tree` with `tree.DecisionTreeClassifier` if `is_clf` is `True`, otherwise use `tree.DecisionTreeRegressor`. In either case, initialize with keyword argument `max_depth` set to `max_depth`.

Then we'll use the `cross_val_score` function (imported in the backend) to obtain the CV scores.

Set `scores` equal to `cross_val_score` applied with `d_tree`, `data`, and `labels` for the first three arguments. Use `cv=cv` for the keyword argument, then return `scores`.

```python
def cv_decision_tree(is_clf, data, labels,
                     max_depth, cv):
  # CODE HERE
  pass
```

TEST   SHOW SOLUTION   SAVE   RESET