CSC 462: Machine Learning

5.2 Learning Algorithm Selection

5.3 Three Sets

5.7 Hyperparameter Tuning

5.7.1 Cross Validation

5.2 Learning Algorithm Selection

5.2 Learning Algorithm Selection

- Choosing a machine learning algorithm can be a difficult task.
  - If you have much time, you can try all of them. However, usually the time you have to solve a problem is limited.
- You can ask yourself several questions before starting to work on the problem.
  - Depending on your answers, you can shortlist some algorithms and try them on your data.

Explainability

In-memory vs. out-of-memory

Number of features and examples

Categorical vs. numerical features

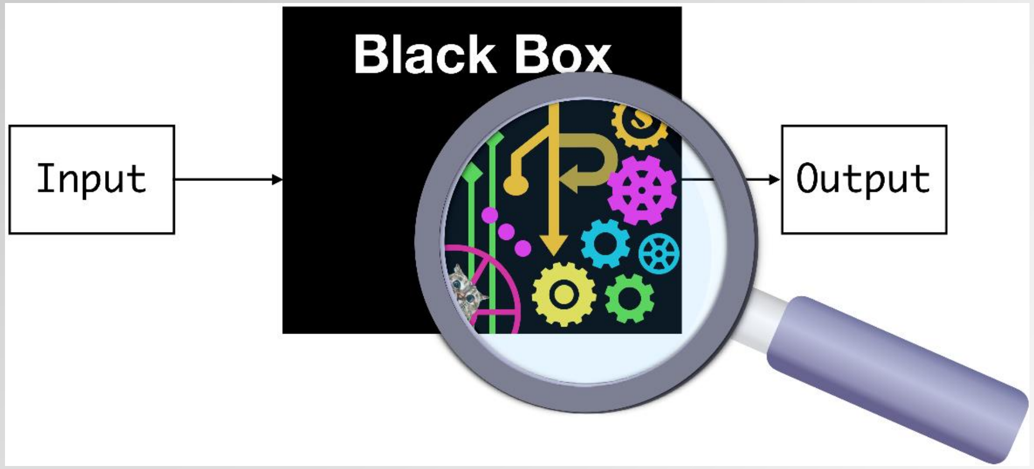
Nonlinearity of the data

Training speed

Prediction speed

Explainability

- Does your model have to be explainable to a non-technical audience?
- Most very accurate learning algorithms are so-called “black boxes.”
  - Examples of such models are neural networks or ensemble models.
- On the other hand, kNN, linear regression, or decision tree learning algorithms produce models that are not always the most accurate, however, the way they make their prediction is very straightforward.



In-memory vs. out-of-memory

- Can your dataset be fully loaded into the RAM of your server or personal computer?
  - If yes, then you can choose from a wide variety of algorithms.
- Otherwise, you would prefer **incremental learning algorithms** that can improve the model by adding more data gradually.

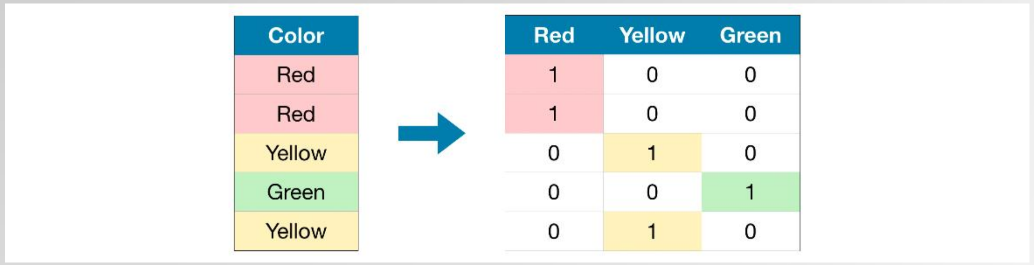
- List of Estimators with `partial_fit()` Method in Scikit-Learn
- Regression
  - sklearn.linear_model.SGDRegressor
  - sklearn.linear_model.PassiveAggressiveRegressor
  - sklearn.neural_network.MLPRegressor
- Classification
  - sklearn.naive_bayes.MultinomialNB
  - sklearn.naive_bayes.BernoulliNB
  - sklearn.linear_model.Perceptron
  - sklearn.linear_model.SGDClassifier
  - sklearn.linear_model.PassiveAggressiveClassifier
  - sklearn.neural_network.MLPClassifier

Number of features and examples

- How many training examples do you have in your dataset?
- How many features does each example have?
- Some algorithms, including neural networks and gradient boosting (we consider both later), can handle a huge number of examples and millions of features. Others, like SVM, can be very modest in their capacity.

Categorical vs. numerical features

- Is your data composed of categorical only, or numerical only features, or a mix of both?
- Depending on your answer, some algorithms cannot handle your dataset directly, and you would need to convert your categorical features into numerical ones.

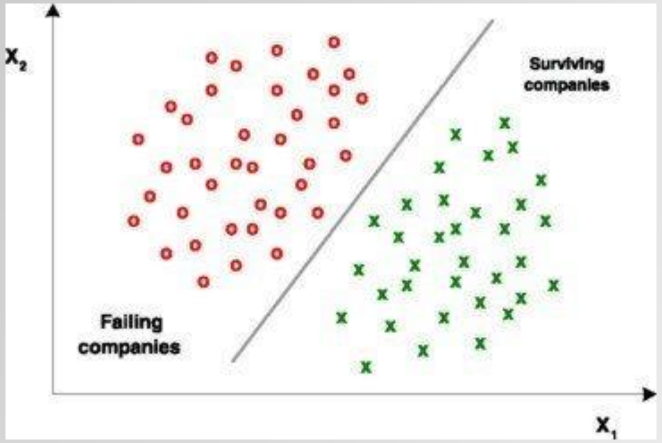


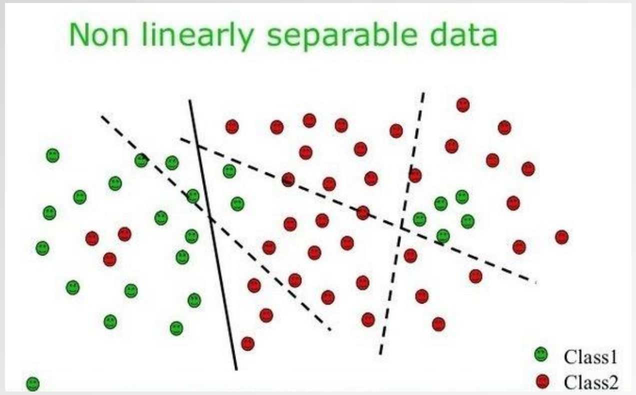
Color
Red
Red
Yellow
Green
Yellow

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

Nonlinearity of the data

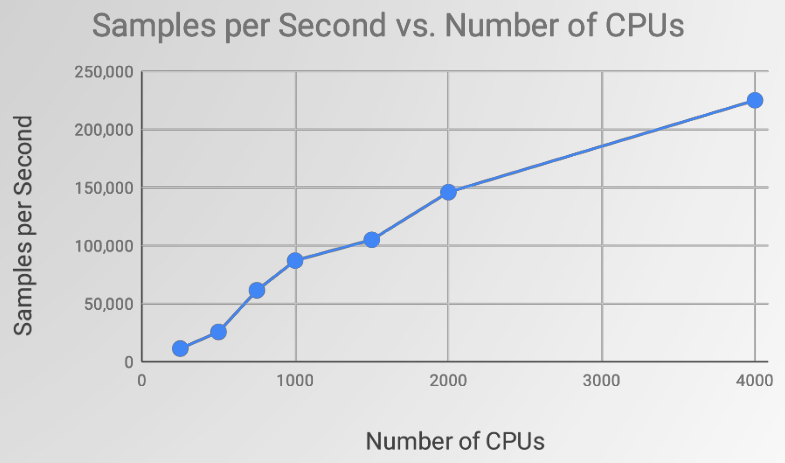
- Is your data linearly separable or can it be modeled using a linear model?
- If yes, SVM with the linear kernel, logistic or linear regression can be good choices.
- Otherwise, deep neural networks or ensemble algorithms might work better.





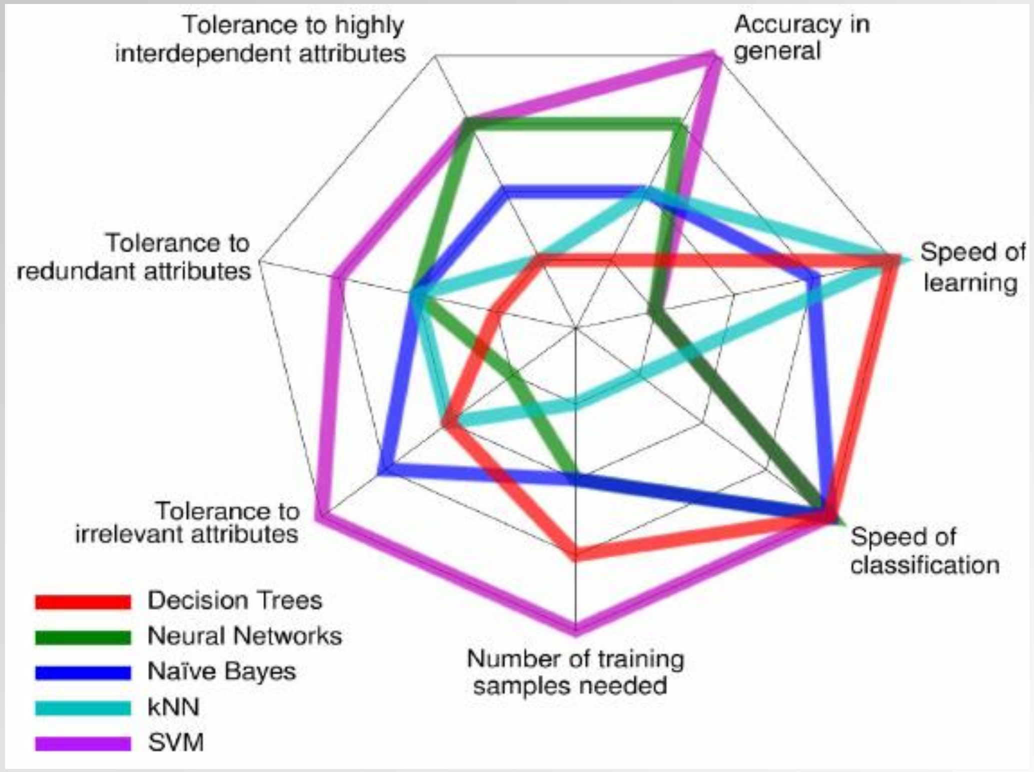
Training speed

- How much time is a learning algorithm allowed to use to build a model?
  - Neural networks are known to be slow to train.
  - Simple algorithms like logistic and linear regression or decision trees are much faster.
- Specialized libraries contain very efficient implementations of some algorithms; you may prefer to do research online to find such libraries.
- Some algorithms, such as random forests, benefit from the availability of multiple CPU cores, so their model building time can be significantly reduced on a machine with dozens of cores.

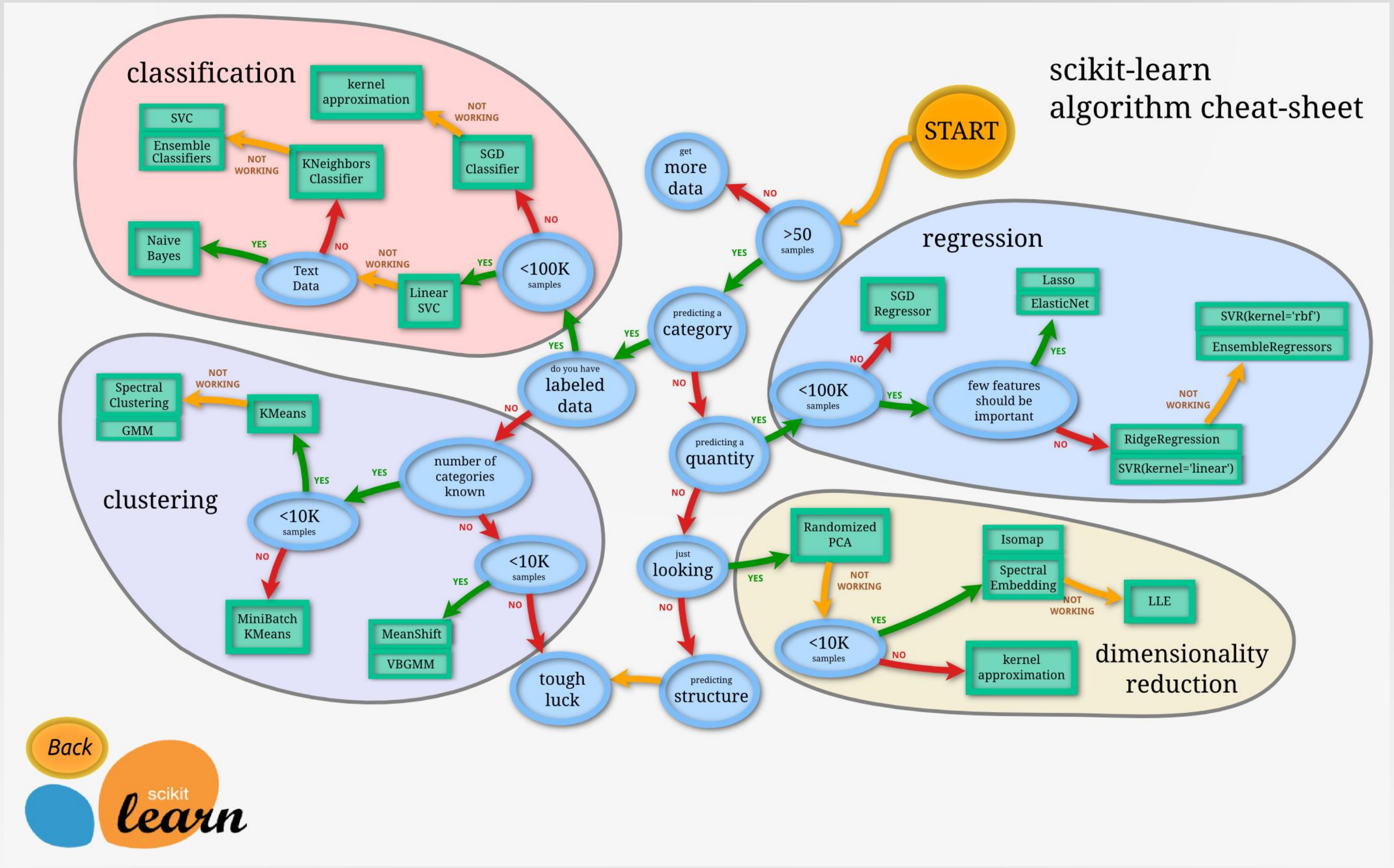


Prediction speed

- How fast does the model have to be when generating predictions?
- Will your model be used in production where very high throughput is required?
- Algorithms like SVMs, linear and logistic regression, and (some types of) neural networks, are extremely fast at the prediction time.
- Others, like kNN, ensemble algorithms, and very deep or recurrent neural networks, are slower



Machine learning algorithm selection for scikit-learn

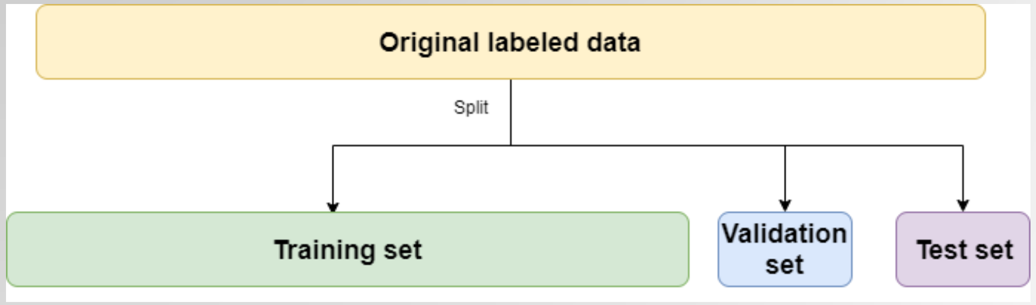


5.3 Three Sets

Three Sets

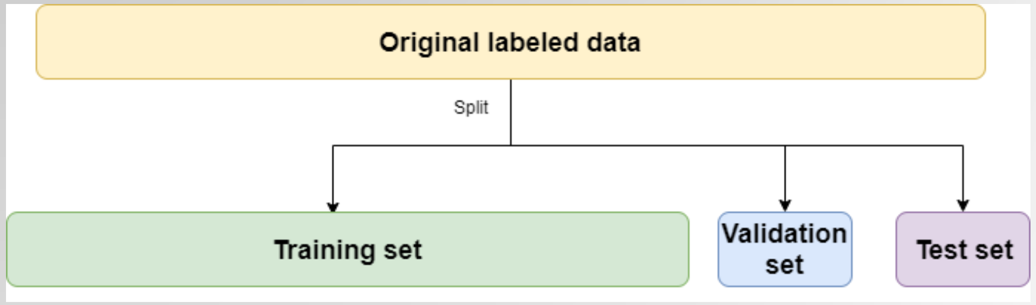
Data analysts work with three sets of labeled examples:

- Training set
- Validation set
- Test set



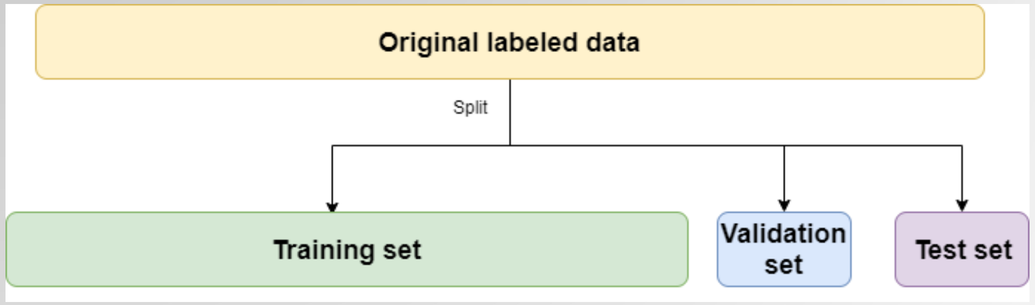
Validation & Test Sets

- The validation and test sets are roughly the same sizes
  - Much smaller than the size of the training set
- The learning algorithm cannot use examples from these two subsets to build a model
  - That is why those two sets are often called **hold-out sets**



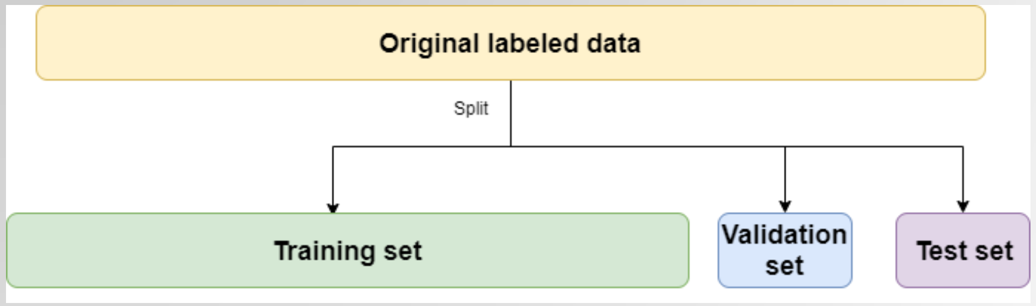
Validation & Test Sets

- There's no optimal proportion to split the dataset into these three subsets.
- In the past, the rule of thumb was to use:
  - 70% of the dataset for training
  - 15% for validation
  - 15% for testing
- In the age of big data, datasets often have millions of examples. In such cases, it could be reasonable to:
  - 95% for training
  - 2.5% for validation
  - 2.5% for testing



Why do we need two hold-out sets and not one?

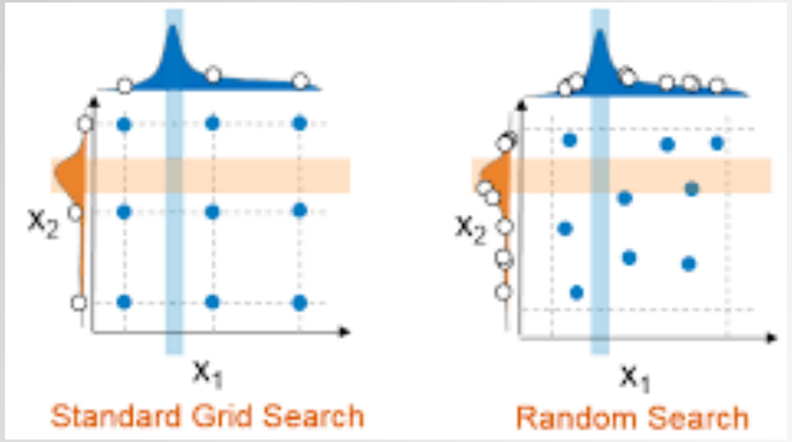
- Validation set is used to
  - Choose the learning algorithm
  - Find the best values of hyperparameters
- Test set is used to assess the model before delivering it to the client or putting it in production



5.7 Hyperparameter Tuning

Hyperparameter Tuning

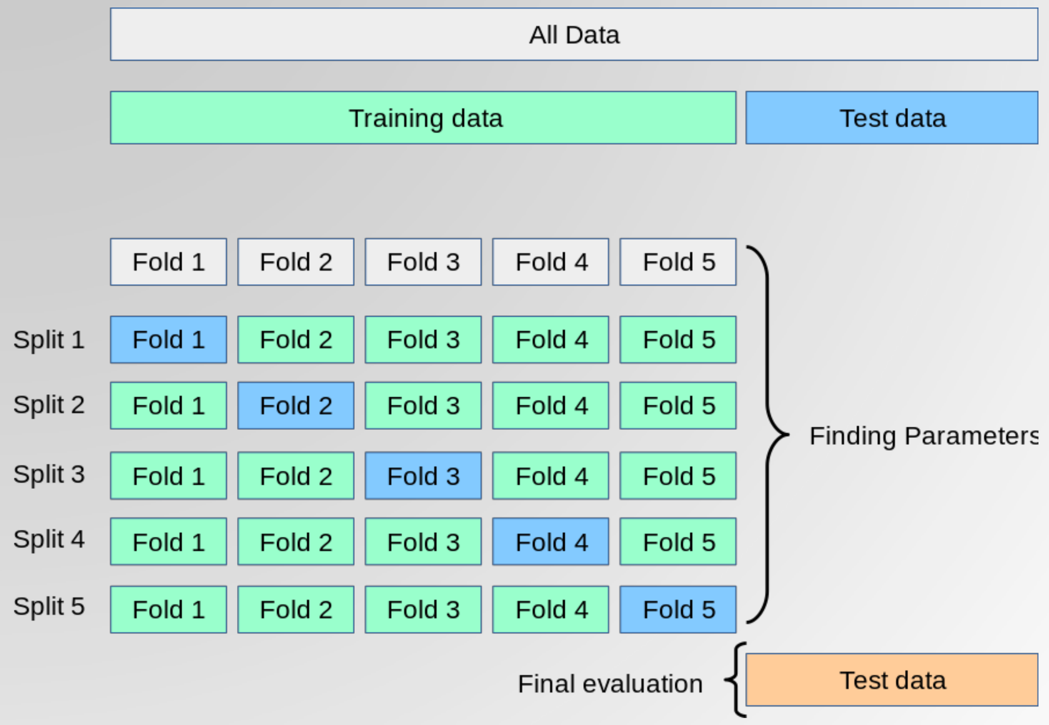
- Hyperparameters aren't optimized by the learning algorithm itself.
  - The data analyst has to "tune" hyperparameters by experimentally finding the best combination of values, one per hyperparameter.
- **Grid search** is the most simple hyperparameter tuning technique.
  - Grid search is a structured way of testing hyperparameter combinations and is often used when there are only three or less hyperparameters.
- Trying all combinations of hyperparameters, especially if there are more than a couple of them, could be time-consuming, especially for large datasets.
- There are more efficient techniques, such as **random search** and **Bayesian hyperparameter optimization**.



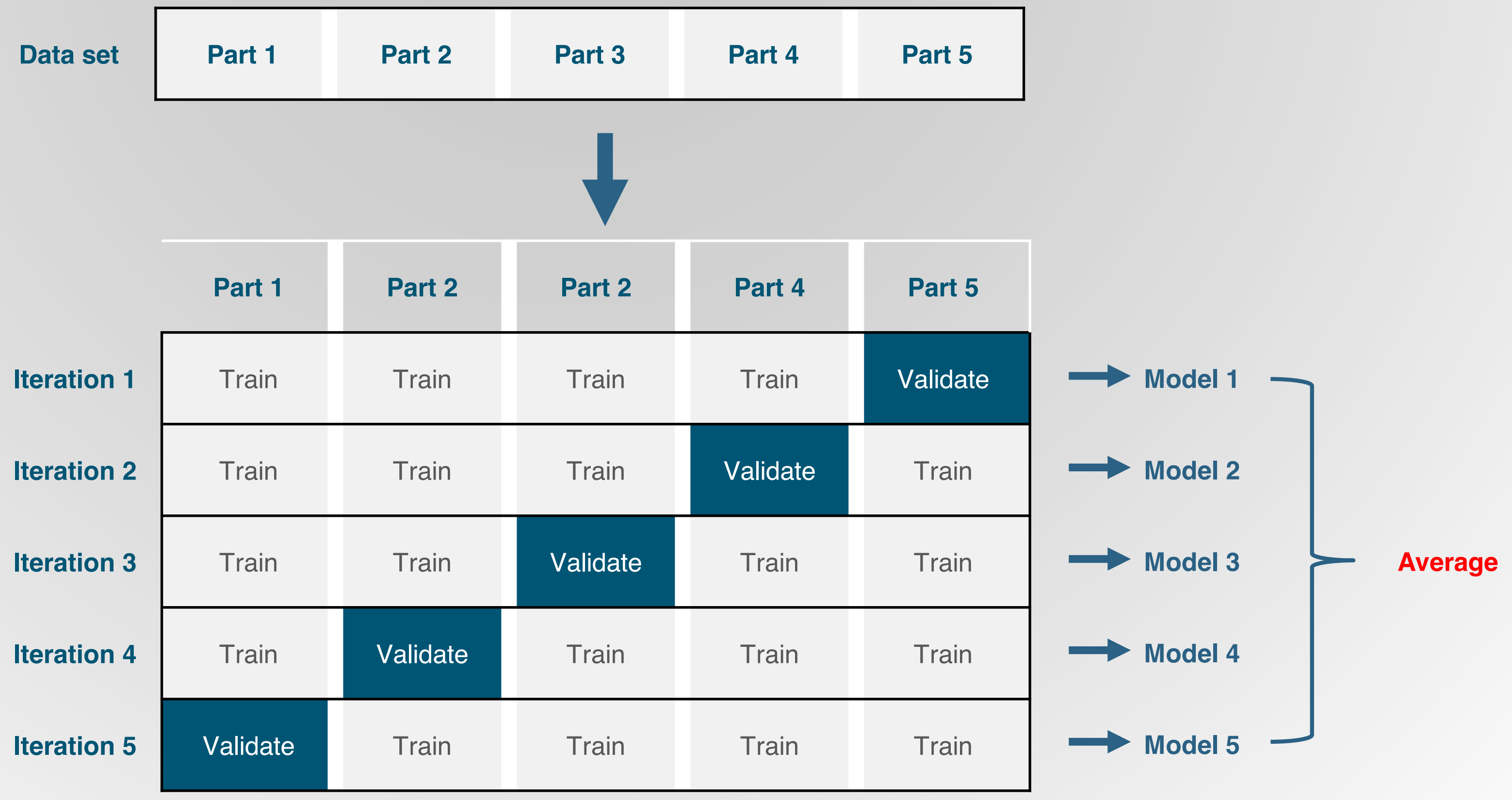
5.7.1 Cross Validation

Cross Validation

- Cross Validation is a very useful technique for assessing the effectiveness of your model, particularly in cases where you need to mitigate overfitting.
- It is also of use in determining the hyper parameters of your model, in the sense that which parameters will result in lowest test error.



Cross-validation



	Part 1	Part 2	Part 2	Part 4	Part 5
Iteration 1	Train	Train	Train	Train	Validate
Iteration 2	Train	Train	Train	Validate	Train
Iteration 3	Train	Train	Validate	Train	Train
Iteration 4	Train	Validate	Train	Train	Train
Iteration 5	Validate	Train	Train	Train	Train

scikit-learn cross validation functions

cross_val_score

cross_validate

KFold

StratifiedKFold

LeaveOneOut

cross_val_score

- The simplest way to use cross-validation is to call the `cross_val_score`
- By default, the score computed at each CV iteration is the `score` method of the estimator.
  - It is possible to change this by using the scoring parameter
- No Shuffle by default

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1, random_state=42)
>>> scores = cross_val_score(clf, X, y, cv=5)
>>> scores
array([0.96..., 1.  , 0.96..., 0.96..., 1.  ])
```

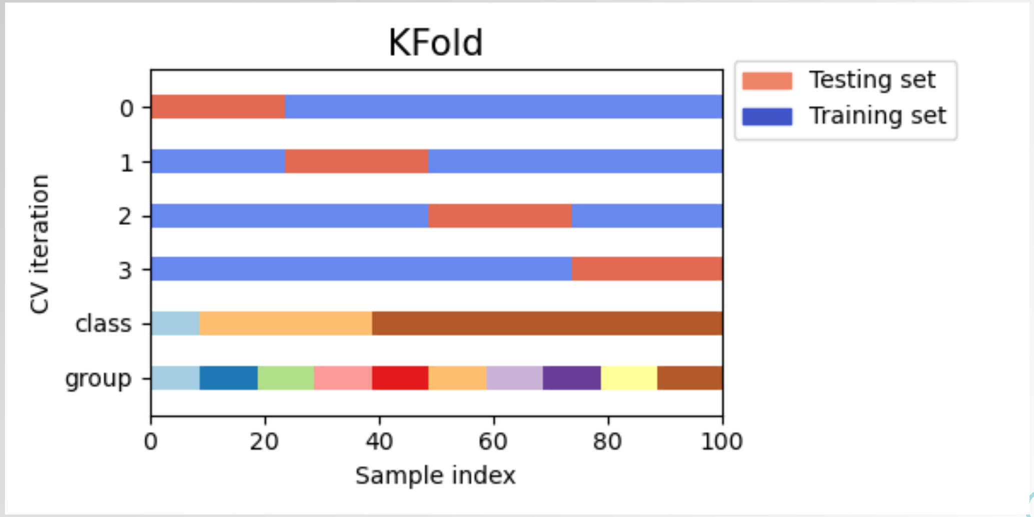
cross_validate

- The `cross_validate` function differs from `cross_val_score` in two ways:
  - It allows specifying **multiple metrics** for evaluation.
  - It returns a `dict` containing **fit-times, score-times** (and optionally training scores as well as fitted estimators) in addition to the test score.
- No Shuffle by default

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro’]
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, X, y, scoring=scoring)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro’]
>>> scores['test_recall_macro’]
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.  ])
```

KFold

- `KFold` divides all the samples in k groups of samples, called folds, of equal sizes (if possible).
- The prediction function is learned using k-1 folds, and the fold left out is used for test.
- **Parameters**
  - `n_splits` (default=5)
    - Number of folds. Must be at least 2.
  - `shuffle` (default=False)
    - Whether to shuffle the data before splitting into batches. Note that the samples within each split will not be shuffled.
  - `random_state` (default=None)
    - When shuffle is True, `random_state` affects the ordering of the indices, which controls the randomness of each fold. Otherwise, this parameter has no effect. Pass an int for reproducible output across multiple function calls.



KFold

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = KFold(n_splits=2)
>>> kf.get_n_splits(X)
2
>>> print(kf)
KFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in kf.split(X):
    ... print("TRAIN:", train_index, "TEST:", test_index)
    ... X_train, X_test = X[train_index], X[test_index]
    ... y_train, y_test = y[train_index], y[test_index]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

After getting the indices for both training and testing sets at each iteration,

you should train the model on each part (by calling the `fit` function), and then you should evaluate it using the test portion.