Model Evaluation and Improvement

ESM3081 Programming for Data Science

Seokho Kang

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Model Evaluation and Improvement for Supervised Learning

- **We will focus on evaluating models and selecting hyperparameters for the supervised methods (regression and classification).**
  - Evaluating and selecting models in unsupervised learning is often a very qualitative process.

- **We are interested in measuring how well our model generalizes to new, previously unseen data.**
  - We are not interested in how well our model fit the training set, but rather in how well it can make predictions for data that was not observed during training.
  - We have split our dataset into a training set and a test set to evaluate supervised models.
    - Built a model on the training set by calling the *fit* method
    - Evaluated it on the test set using the *predict* method

Model Evaluation and Improvement for Supervised Learning

- **Model evaluation**
  - **Evaluation metrics:** to evaluate classification and regression performance that go beyond the default metrics
  - **Cross-validation:** a more robust way to assess generalization performance

- **Hyperparameter selection**
  - **Grid search:** an effective method for adjusting the hyperparameters in supervised models for the best generalization performance.

Topics

- **Evaluating Metrics and Scoring**
- **Cross-Validation**
- **Grid Search**

Evaluation Metrics and Scoring

Evaluation Metrics and Scoring

- There are many possible ways to summarize how well a supervised model performs on a given dataset.

- In practice, simple evaluation metrics like accuracy and R^2 score might not be appropriate for many applications.

- It is important to choose the right metric when selecting between models and adjusting hyperparameters.

Keep the End Goal in Mind

- **You should always have the end goal of the machine learning application in mind.**
  - In practice, we are usually interested not just in making accurate predictions, but in using these predictions as part of a larger decision making process.
  - Before picking a machine learning metric, you should think about the high-level goal of the application, often called the business metric.

- **When choosing a model,**
  - You should pick the model that has the most positive influence on the business metric.
  - Often this is hard, as assessing the business impact of a particular model might require putting it in production in a real-life system.

Keep the End Goal in Mind

- **In the early stages of development,**
  - It is often infeasible to put models into production just for testing purposes.
  - The high business or personal risks can be involved.

- **We often need to find some surrogate evaluation procedure, using an evaluation metric that is easier to compute.**
  - It pays off to find the closest metric to the original business goal that is feasible to evaluate.
  - The closest metric should capture the expected business impact.
  - Also, the closest metric should be used whenever possible for model evaluation and selection.
  - You should keep in mind that this is **only a surrogate**.

Metrics for Binary Classification

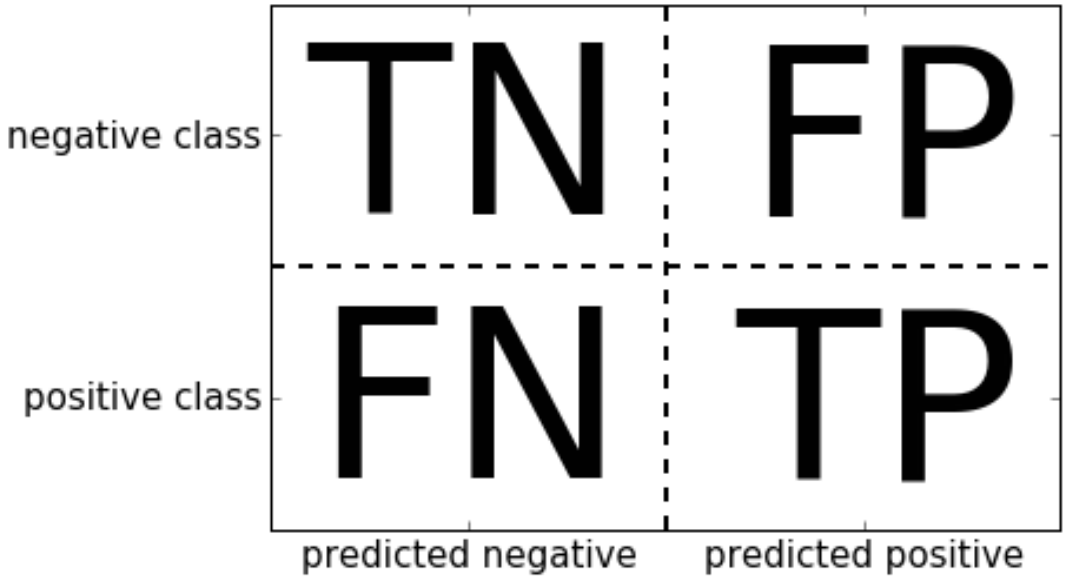
- **Binary classification is arguably the most common and conceptually simple application of machine learning in practice.**

- **For binary classification, we often speak of a positive class and a negative class.**
  - The positive class is generally the one we are looking for. (our main interest)
    - e.g., failure prediction, disease diagnosis, …
  - An incorrect positive prediction is called a false positive (FP) – **Type 1 Error**.
  - An incorrect negative prediction is called a false negative (FN) – **Type 2 Error**.

- **The consequence of false positives and false negatives are different.**
  - In many situations, we want to avoid false negatives as much as possible, while false positives can be viewed as more of a minor nuisance.
  - Often, measuring accuracy might be misleading, because the number of mistakes we make does not contain all the information we are interested in.

Metrics for Binary Classification

- **Confusion Matrix**
  - A comprehensive way to represent the result of evaluating binary classification.
  - The output of confusion matrix is a two-by-two array.
    - The rows correspond to the true classes.
    - The columns correspond to the predicted classes.
    - Each entry counts how often a data point that belongs to the class corresponding to the row was classified as the class corresponding to the column.



Metrics for Binary Classification

- **Summarization of the information in the confusion matrix**
  - **Accuracy:** The number of correct predictions divided by the number of all data points.

$$\frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

  - **Precision:** the number of positive predictions that are actually positive

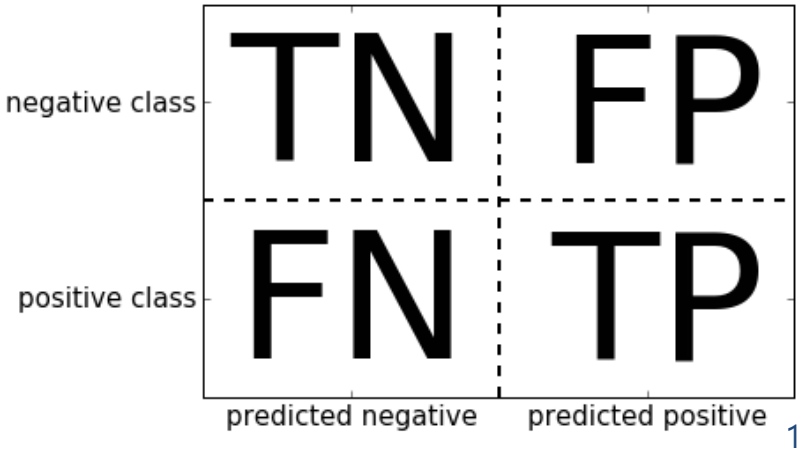
$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

  - **Recall(Sensitivity):** the number of positive data points captured by positive predictions

$$\frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

  - **Specificity:** the number of negative data points captured by negative predictions

$$\frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}}$$



Metrics for Binary Classification

- **Imbalanced Datasets**
  - Types of errors play an important role when one of two classes is much more frequent than the other one.
  - Datasets in which one class is much more frequent than the other are often called imbalanced datasets.
  - In reality, it is rare that the events of interest have equal or even similar frequency in the data.
  - Accuracy is not an appropriate metric to use when evaluating for imbalanced datasets.

Model 1

	Predicted Positive	Predicted Negative
Actual Positive	26	260
Actual Negative	214	2500

Accuracy = (26+2500)/3000 = 84.20%

Precision = 26/(26+214) = 10.83%

Recall = 26/(26+260) = 9.09%

Model 2

	Predicted Positive	Predicted Negative
Actual Positive	0	286
Actual Negative	0	2714

Accuracy = (0+2714)/3000 = 90.47%

Precision = 0/(0+0) = NaN

Recall = 0/(0+286) = 0%

Metrics for Binary Classification

- **The trade-off between optimizing recall and optimizing precision**
  - If you predict all data points to belong to the positive class,
    - You can trivially obtain a perfect recall.
    - However, it will result in many false positives, and therefore the precision will be very low.
  - If you predict all data points to belong to the negative class,
    - Precision will be NaN.
    - Recall will be very bad.
  - **F_1 Score**: the harmonic mean of precision and recall
    - It takes precision and recall into account
    - It can be a better measure than accuracy on imbalanced binary classification dataset

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Metrics for Binary Classification

- **The trade-off between optimizing sensitivity and specificity**
  - If you predict all data points to belong to the positive class,
    - Sensitivity will be 1
    - Specificity will be 0
  - If you predict all data points to belong to the negative class,
    - Sensitivity will be 0
    - Specificity will be 1
  - **Balanced Accuracy**: the arithmetic mean of Sensitivity and Specificity
    - It takes sensitivity and specificity into account
    - It can also be a better measure than accuracy on imbalanced binary classification dataset

$$\text{Balanced\_Accuracy} = \frac{\text{Sensitivity} + \text{Specificity}}{2}$$

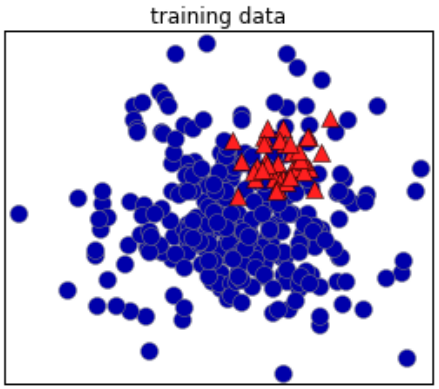
Metrics for Binary Classification

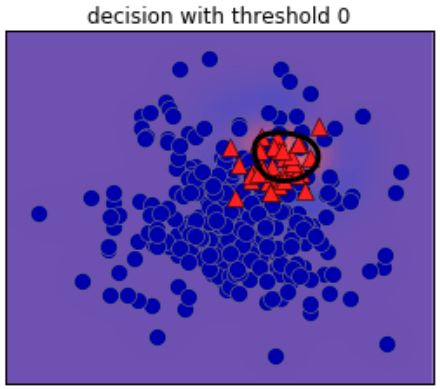
- **Taking Uncertainty into Account**
  - Most classifiers can assess degrees of certainty about predictions.
  - The outputs of *predict* themselves already threw away a lot of information that is contained in the model.
  - Making predictions can be seen as thresholding the output of *decision_function* or *predict_proba* at a certain fixed point
    - In binary classification, we often use 0 for *decision_function* and 0.5 for *predict_proba*.
    - We can change the threshold to adjust the predictions to focus on a certain class.
      e.g., Misclassifying rare class as majority class is often more expensive than misclassifying majority as rare class.
    - It is always hard to provide a rule of thumb regarding how to pick a threshold.

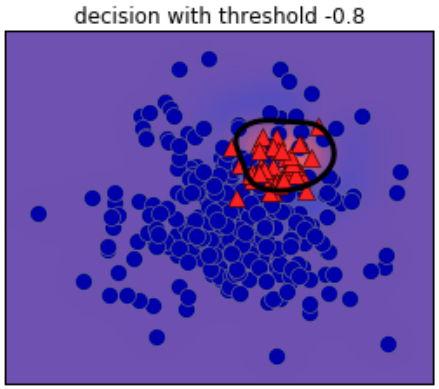
`predict` (self, X)	Predict the class labels for the provided data
`predict_proba` (self, X)	Return probability estimates for the test data X.
`decision_function` (self, X)	Evaluates the decision function for the samples in X.

Metrics for Binary Classification

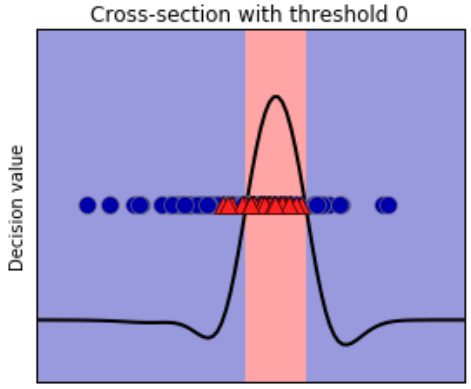
- **Taking Uncertainty into Account**
  - ***Example:*** Decision function of kernel SVC with different decision thresholds

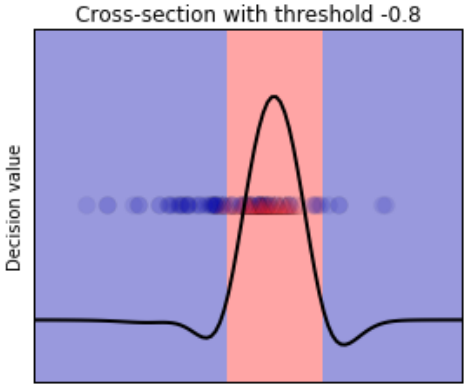






Red triangle – positive class
Blue circle – negative class





Performance
on the validation set

Precision = 0.35
Recall = 0.67
F-score = 0.46

Precision = 0.32
Recall = 1.00
F-score = 0.49

Metrics for Binary Classification

- **Taking Uncertainty into Account**
  - ***Example:*** Various thresholds for the binary classification of 24 data points

$P(y = 1|\boldsymbol{x})$

(1:positive, 0:negative)

Actual Class	Prob. of "1"	Actual Class	Prob. of "1"
1	0.996	1	0.506
1	0.988	0	0.471
1	0.984	0	0.337
1	0.980	1	0.218
1	0.948	0	0.199
1	0.889	0	0.149
1	0.848	0	0.048
0	0.762	0	0.038
1	0.707	0	0.025
1	0.681	0	0.022
1	0.656	0	0.016
0	0.622	0	0.004

when threshold=0

	Predicted Positive	Predicted Negative
Actual Positive	12	0
Actual Negative	12	0

when threshold=0.25

	Predicted Positive	Predicted Negative
Actual Positive	11	1
Actual Negative	4	8

when threshold=0.5

	Predicted Positive	Predicted Negative
Actual Positive	TP=11	FN=1
Actual Negative	FP=2	TN=10

when threshold=0.75

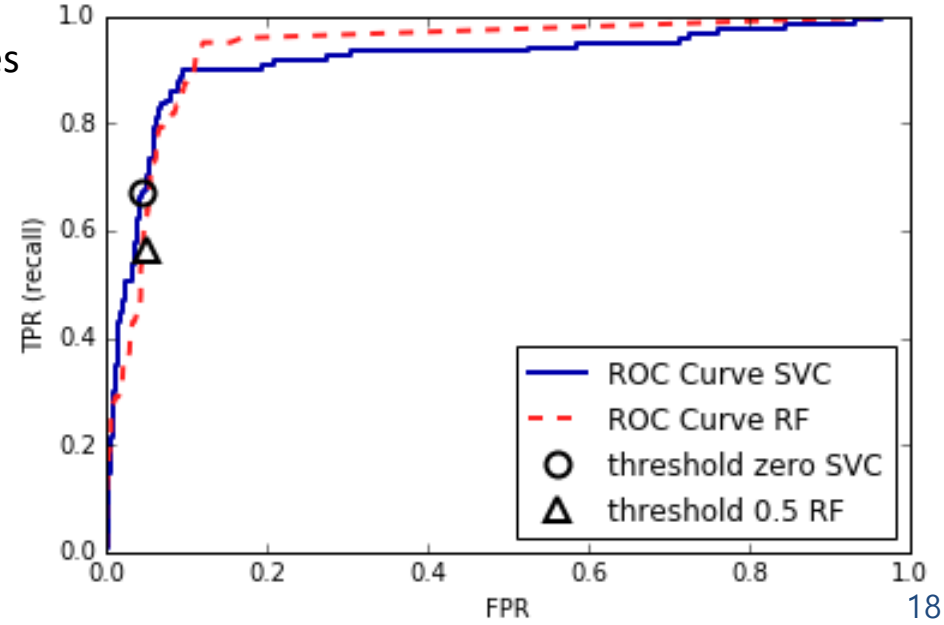
	Predicted Positive	Predicted Negative
Actual Positive	7	5
Actual Negative	1	11

when threshold=1

	Predicted Positive	Predicted Negative
Actual Positive	0	12
Actual Negative	0	12

Metrics for Binary Classification

- **Taking Uncertainty into Account**
  - **Receiver Operating Characteristics (ROC) Curve**
    - The ROC curve is commonly used to analyze the behavior of a classifier at different thresholds.
    - It considers all possible thresholds for a given classifier, and shows the false positive rate (FPR) against the true positive rate (TPR).
      - $\text{FPR} = \frac{\text{FP}}{\text{N}} = \frac{\text{FP}}{\text{FP+TN}}$
      - $\text{TPR(Recall)} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP+FN}}$
    - The area under the curve (AUC) summarizes the ROC curve using a single number.
      - AUC ranges from 0 (worst) to 1 (best)
      - The AUC can be interpreted as evaluating the *ranking* of positive data points.
      - It is highly recommend using AUC when evaluating models on imbalanced data.

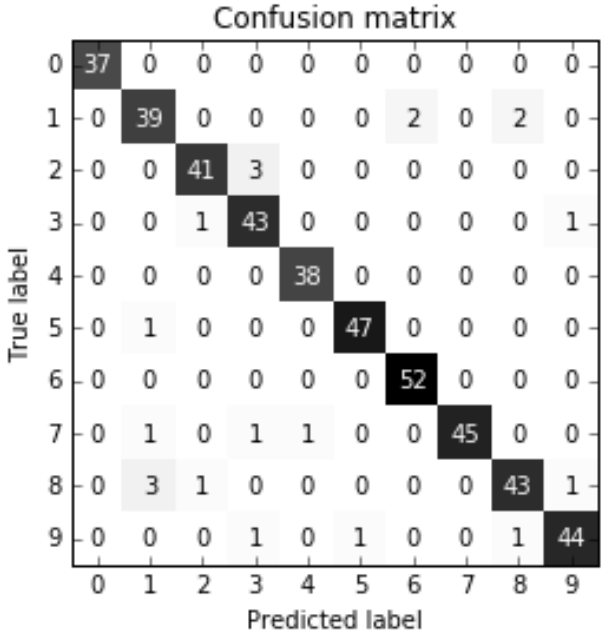


Metrics for Multi-Class Classification

- **In general, multiclass classification results are harder to understand than binary classification results.**
- **Most of the metrics for multiclass classification are derived from binary classification metrics, but averaged over all classes.**
  - Accuracy for multiclass classification is again defined as the fraction of correctly classified data points.
  - As for binary classification, the confusion matrix provides us with details.

Metrics for Multi-Class Classification

- **Example: Confusion matrix for the task of classifying ten classes**
  - **Accuracy = 95.3%**
  - **The precision, recall, and F_1-score can be computed for each class, with that class being the positive class and the other classes making up the negative class.**
    - Precision and recall are a perfect 1 for class 0, as there are no confusions with this class.
    - We can see that the model has particular difficulties with classes 1,3, and 8.



	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	0.89	0.91	0.90	43
2	0.95	0.93	0.94	44
3	0.90	0.96	0.92	45
4	0.97	1.00	0.99	38
5	0.98	0.98	0.98	48
6	0.96	1.00	0.98	52
7	1.00	0.94	0.97	48
8	0.93	0.90	0.91	48
9	0.96	0.94	0.95	47
avg / total	0.95	0.95	0.95	450

Metrics for Multi-Class Classification

- **The most commonly used metric for imbalanced datasets in the multiclass setting is the multiclass version of the F_1-score.**
  - **Micro Averaging:** computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and F_1-score using these counts.
  - **Weighted Averaging:** computes the weighted mean of the per-class F_1-scores, weighted by their support.
  - **Macro Averaging:** computes the unweighted mean of the per-class F_1-scores, giving equal weight to all classes.

Metrics for Multi-Class Classification

- **Considerations**
  - In multi-class classification, F_{micro} is equivalent to conventional classification accuracy, and thus can be dominated by majority classes.
  - F_{macro} is usually smaller than F_{micro} under class imbalance.
  - If you care about each data point equally much, it is recommended to use F_{micro}; if you care about each class equally much, it is recommended to use F_{macro}

$$P_{\text{micro}} = \frac{\sum_j \text{TP}^j}{\sum_j \text{TP}^j + \sum_j \text{FP}^j};$$

$$R_{\text{micro}} = \frac{\sum_j \text{TP}^j}{\sum_j \text{TP}^j + \sum_j \text{FN}^j};$$

$$F_{\text{micro}} = \frac{2 \times P_{\text{micro}} \times R_{\text{micro}}}{P_{\text{micro}} + R_{\text{micro}}};$$

$$P^j = \frac{\text{TP}^j}{\text{TP}^j + \text{FP}^j};$$

$$R^j = \frac{\text{TP}^j}{\text{TP}^j + \text{FN}^j};$$

$$F_{\text{macro}} = \frac{1}{c} \sum_j \frac{2 \times P^j \times R^j}{P^j + R^j}$$

Metrics for Regression

- **Evaluation for regression can be done by analyzing overpredicting the target versus underpredicting the target.**

  Given a validation or test set $D' = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$,

  - **Root Mean Squared Error (RMSE)**

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_i (y_i - \hat{y}_i)^2}$$

  - **Mean Absolute Error (MAE)**

$$\text{MAE} = \frac{1}{n}\sum_i |y_i - \hat{y}_i|$$

  - **Mean Absolute Percentage Error (MAPE)**

$$\text{MAPE} = \frac{1}{n}\sum_i \left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100\%$$

  - **Coefficient of Determination (R^2)**

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \qquad \bar{y} = \frac{1}{n}\sum_i y_i$$

** ratio scale vs interval scale*

Metrics for Regression

- **Considerations: Interval Scale vs Ratio Scale**
  - **Interval Scale** – has no absolute zero, allows for the degree of difference between magnitudes, but not the ratio (*e.g.*, temperature, …)
    - **Example:** If we use MAPE to evaluate temperature predictions,

      When the true value is 1°C and its prediction is 0°C → APE=100%

      When the true value is 100°C and its prediction is 101°C → APE=1%

      When the true value is 0°C?

  - **Ratio Scale** – possess an absolute zero, allows for the degrees of both difference and ratio between magnitudes (*e.g.*, length, weight, …)
    - **Example:** If we use MAE to evaluate weight predictions,

      When the true value is 1g and its prediction is 2g → AE=1g

      When the true value is 10,000g and its prediction is 10,001g → AE=1g

Metrics for Regression

- **Considerations: MAE vs RMSE**
  - RMSE more penalizes large errors

CASE 1: Evenly distributed errors

ID	Error	|Error|	Error^2
1	2	2	4
2	2	2	4
3	2	2	4
4	2	2	4
5	2	2	4
6	2	2	4
7	2	2	4
8	2	2	4
9	2	2	4
10	2	2	4

MAE	RMSE
2.000	2.000

CASE 2: Small variance in errors

ID	Error	|Error|	Error^2
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	3	3	9
7	3	3	9
8	3	3	9
9	3	3	9
10	3	3	9

MAE	RMSE
2.000	2.236

CASE 3: Large error outlier

ID	Error	|Error|	Error^2
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	20	20	400

MAE	RMSE
2.000	6.325

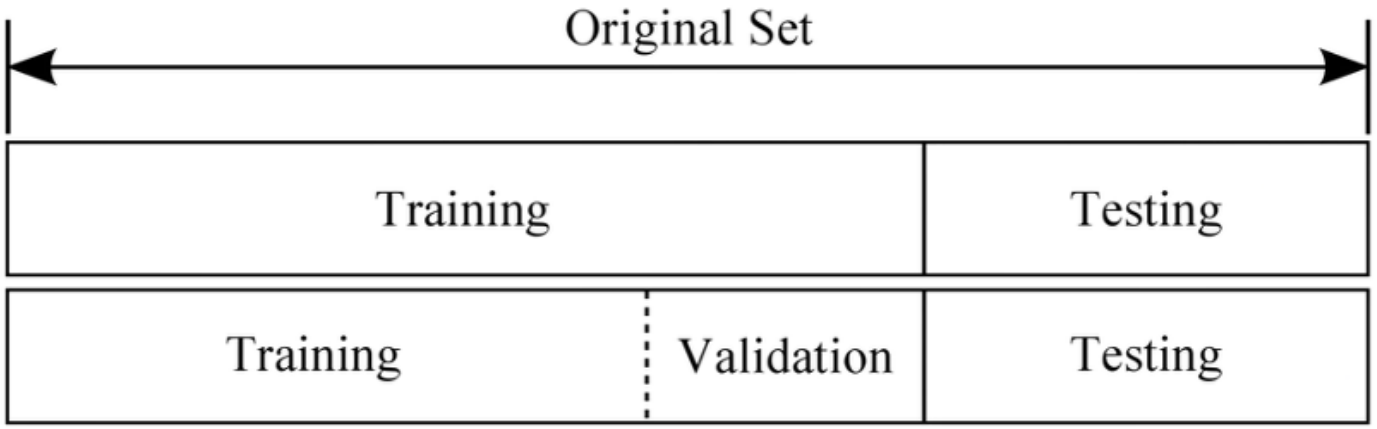
https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

Cross-Validation

Training, Validation, and Test

- **Training set:** to learn the **parameters** of the model
- **Validation set:** to choose the **hyperparameters** of the model
- **Test set:** for final evaluation of the **generalization error** of the model
  (How well will our model perform with new data that were not observed during training and validation?)

▪**흔한 실수 1**: training set에 대해서 최종 성능평가
▪**흔한 실수 2**: validation set과 test set을 구분하지 않음



"must be disjoint!"

Cross-Validation

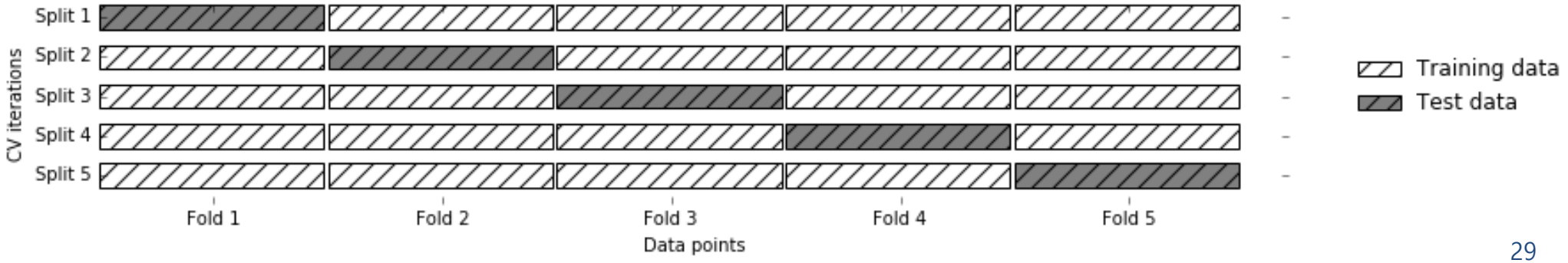
- **The reason we split our data into training/validation/test sets is that we are interested in measuring how well our model generalizes to new, previously unseen data.**

- **The results can depend on a particular random choice for the partition.**
  - By partitioning the available data into three sets, we drastically reduce the number of data points which can be used for learning the model.

- **Cross-validation is a more robust way of evaluating generalization performance.**
  - In cross-validation, the data are split repeatedly and multiple models are trained.
  - Cross-validation is more stable and thorough than using a split into a training and a test set.

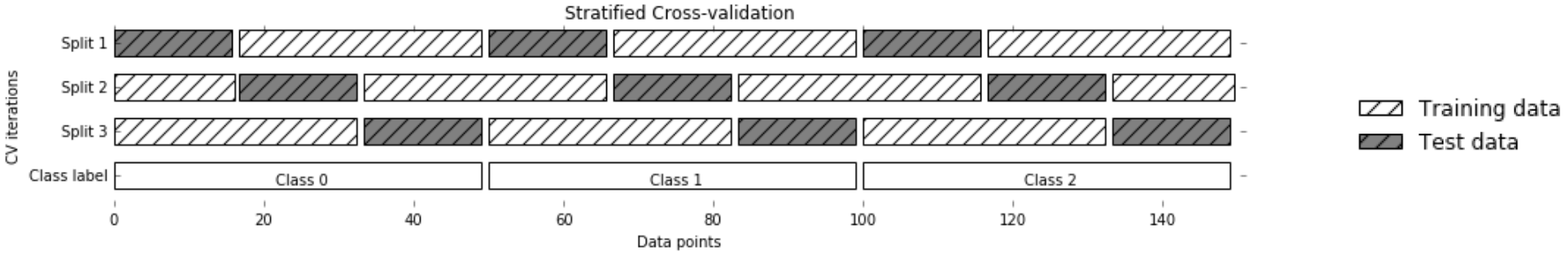
k-Fold Cross-Validation

- **k-fold cross-validation: the most commonly used version of cross-validation**
  - k is a user-specified number, usually 5 or 10.
  - When performing five-fold cross-validation, the data is first partitioned into five parts of (approximately) equal size, called folds.
  - Next, five models are trained.
    - The first model is trained using the first fold as the test set, and the remaining folds (2–5) are used as the training set.
    - The second model is trained using the second fold as the test set, and the remaining folds (1, 3-5) are used as the training set.
    - This process is repeated using folds 3, 4, and 5 as test sets.
  - We evaluate the performance for each of these five splits.
    - A common way to summarize the cross-validation accuracy is to compute the mean.



Stratified k-Fold Cross-Validation

- **For classification,** it is usually a good idea to use **stratified k-fold cross-validation** instead of standard k-fold cross-validation.
- In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset.
  - It results in more reliable estimates of generalization performance.
  - In the case of only 10% of samples belonging to class B, using standard k-fold cross-validation it might easily happen that one fold only contains samples of class A.



Leave-One-Out Cross-Validation

- Another frequently used cross-validation method is **leave-one-out cross-validation**.
- Leave-one-out cross-validation is identical to k-fold cross-validation where **each fold is a single data point.**
  - k = the number of data points in the dataset
  - For each split, you pick a single data point to be the test set.

- This can be very time consuming, particularly for large datasets, but sometimes provides better estimates on small datasets.

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html

sklearn.model_selection.cross_validate

sklearn.model_selection.**cross_validate**(*estimator, X, y=None, *, groups=None, scoring=None, cv=None, n_jobs=None, verbose=0, fit_params=None, pre_dispatch='2*n_jobs', return_train_score=False, return_estimator=False, error_score=nan*) [source]

Evaluate metric(s) by cross-validation and also record fit/score times.

Read more in the User Guide.

Parameters:

estimator : *estimator object implementing 'fit'*
The object to use to fit the data.

X : *array-like of shape (n_samples, n_features)*
The data to fit. Can be for example a list, or an array.

y : *array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*
The target variable to try to predict in the case of supervised learning.

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html

scoring : *str, callable, list, tuple, or dict, default=None*

Strategy to evaluate the performance of the cross-validated model on the test set.

If `scoring` represents a single score, one can use:

- a single string (see The scoring parameter: defining model evaluation rules);
- a callable (see Defining your scoring strategy from metric functions) that returns a single value.

If `scoring` reprents multiple scores, one can use:

- a list or tuple of unique strings;
- a callable returning a dictionary where the keys are the metric names and the values are the metric scores;
- a dictionary with metric names as keys and callables a values.

Scoring	Function
Classification	
'accuracy'	metrics.accuracy_score
'balanced_accuracy'	metrics.balanced_accuracy_score
'average_precision'	metrics.average_precision_score
'brier_score_loss'	metrics.brier_score_loss
'f1'	metrics.f1_score
'f1_micro'	metrics.f1_score
'f1_macro'	metrics.f1_score
'f1_weighted'	metrics.f1_score
'f1_samples'	metrics.f1_score
'neg_log_loss'	metrics.log_loss
'precision' etc.	metrics.precision_score
'recall' etc.	metrics.recall_score
'jaccard' etc.	metrics.jaccard_score
'roc_auc'	metrics.roc_auc_score
Regression	
'explained_variance'	metrics.explained_variance_score
'max_error'	metrics.max_error
'neg_mean_absolute_error'	metrics.mean_absolute_error
'neg_mean_squared_error'	metrics.mean_squared_error
'neg_mean_squared_log_error'	metrics.mean_squared_log_error
'neg_median_absolute_error'	metrics.median_absolute_error
'r2'	metrics.r2_score

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html

cv : *int, cross-validation generator or an iterable, default=None*

Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 5-fold cross validation,
- int, to specify the number of folds in a `(Stratified)KFold`,
- CV splitter,
- An iterable yielding (train, test) splits as arrays of indices.

For int/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer User Guide for the various cross-validation strategies that can be used here.

Changed in version 0.22: `cv` default value if None changed from 3-fold to 5-fold.

`model_selection.GroupKFold` ([n_splits])	K-fold iterator variant with non-overlapping groups.
`model_selection.GroupShuffleSplit` ([…])	Shuffle-Group(s)-Out cross-validation iterator
`model_selection.KFold` ([n_splits, shuffle, …])	K-Folds cross-validator
`model_selection.LeaveOneGroupOut`	Leave One Group Out cross-validator
`model_selection.LeavePGroupsOut` (n_groups)	Leave P Group(s) Out cross-validator
`model_selection.LeaveOneOut`	Leave-One-Out cross-validator
`model_selection.LeavePOut` (p)	Leave-P-Out cross-validator
`model_selection.PredefinedSplit` (test_fold)	Predefined split cross-validator
`model_selection.RepeatedKFold` ([n_splits, …])	Repeated K-Fold cross validator.
`model_selection.RepeatedStratifiedKFold` ([…])	Repeated Stratified K-Fold cross validator.
`model_selection.ShuffleSplit` ([n_splits, …])	Random permutation cross-validator
`model_selection.StratifiedKFold` ([n_splits, …])	Stratified K-Folds cross-validator
`model_selection.StratifiedShuffleSplit` ([…])	Stratified ShuffleSplit cross-validator
`model_selection.TimeSeriesSplit` ([n_splits, …])	Time Series cross-validator

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html

Returns:

scores : *dict of float arrays of shape (n_splits,)*

Array of scores of the estimator for each run of the cross validation.

A dict of arrays containing the score/time arrays for each scorer is returned. The possible keys for this `dict` are:

`test_score`

The score array for test scores on each cv split. Suffix `_score` in `test_score` changes to a specific metric like `test_r2` or `test_auc` if there are multiple scoring metrics in the scoring parameter.

`train_score`

The score array for train scores on each cv split. Suffix `_score` in `train_score` changes to a specific metric like `train_r2` or `train_auc` if there are multiple scoring metrics in the scoring parameter. This is available only if `return_train_score` parameter is `True`.

`fit_time`

The time for fitting the estimator on the train set for each cv split.

`score_time`

The time for scoring the estimator on the test set for each cv split. (Note time for scoring on the train set is not included even if `return_train_score` is set to `True`

`estimator`

The estimator objects for each cv split. This is available only if `return_estimator` parameter is set to `True`.

scikit-learn Practice

- **Example (*iris* dataset)**

```
In [2]: from sklearn.datasets import load_iris
        from sklearn.model_selection import cross_validate
        from sklearn.linear_model import LogisticRegression

        iris = load_iris()
```

We can change the number of folds used by changing the cv parameter.

```
In [3]: clf = LogisticRegression()
        scores = cross_validate(clf, iris.data, iris.target, scoring='accuracy', cv=5,
                                return_train_score=True, return_estimator=True)
```

```
In [4]: scores['train_score'], scores['test_score']
```

```
Out[4]: (array([0.96667, 0.96667, 0.98333, 0.98333, 0.975  ]),
         array([0.96667, 1.     , 0.93333, 0.96667, 1.     ]))
```

Looking at all five scores produced by the five-fold cross-validation, we can conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 90% accuracy.

This could imply that the model is very dependent on the particular folds used for training, but it could also just be a consequence of the small size of the dataset.

scikit-learn Practice

- **Example (*iris* dataset)**

```
In [5]: scores['estimator']

Out[5]: [LogisticRegression(),
         LogisticRegression(),
         LogisticRegression(),
         LogisticRegression(),
         LogisticRegression()]
```

The estimator objects for each cv split.

scikit-learn Practice

scikit-learn allows for much finer control over what happens during the splitting of the data by providing a cross-validation splitter as the cv parameter.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

sklearn.model_selection.KFold

class sklearn.model_selection.**KFold**(*n_splits=5, *, shuffle=False, random_state=None*) [source]

K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

sklearn.model_selection.StratifiedKFold

class sklearn.model_selection.**StratifiedKFold**(*n_splits=5, *, shuffle=False, random_state=None*) [source]

Stratified K-Folds cross-validator.

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Methods

get_n_splits (self[, X, y, groups])	Returns the number of splitting iterations in the cross-validator
split (self, X[, y, groups])	Generate indices to split data into training and test set.

scikit-learn Practice

- **Example (*iris* dataset)**

We import the *StratifiedKFold* splitter class from the *model_selection* module

```
In [6]: from sklearn.datasets import load_iris
        from sklearn.model_selection import cross_validate, StratifiedKFold
        from sklearn.linear_model import LogisticRegression

        iris = load_iris()
        kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
```

We need to fix the random_state to get a reproducible shuffling.

```
In [7]: clf = LogisticRegression()
        scores = cross_validate(clf, iris.data, iris.target, scoring='accuracy', cv=kfold,
                                return_train_score=True, return_estimator=True)
```

```
In [8]: scores['train_score'], scores['test_score']
```

```
Out[8]: (array([0.975  , 0.98333, 0.98333, 0.96667, 0.975  ]),
         array([0.96667, 1.     , 0.93333, 1.     , 0.9    ]))
```

```
In [9]: from sklearn.tree import DecisionTreeClassifier

        clf2 = DecisionTreeClassifier()
        scores2 = cross_validate(clf2, iris.data, iris.target, scoring='accuracy', cv=kfold,
                                 return_train_score=True, return_estimator=True)
```

```
In [10]: scores2['train_score'], scores2['test_score']
```

```
Out[10]: (array([1., 1., 1., 1., 1.]),
          array([0.96667, 1.     , 0.9    , 1.     , 0.86667]))
```

But, how do we apply data scaling?

scikit-learn Practice

- **Example (*iris* dataset) with data scaling**

In [13]:
```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedKFold
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
scaler = StandardScaler()
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
```

In [14]:
```
score_train = []
score_test = []

for train_idx, test_idx in kfold.split(iris.data, iris.target):

    X_train = iris.data[train_idx]
    y_train = iris.target[train_idx]
    X_test = iris.data[test_idx]
    y_test = iris.target[test_idx]

    scaler.fit(X_train)
    X_train_scaled = scaler.transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    clf = MLPClassifier(max_iter=1000, random_state=0)
    clf.fit(X_train_scaled, y_train)

    y_train_hat = clf.predict(X_train_scaled)
    score_train.append(accuracy_score(y_train, y_train_hat))
    y_test_hat = clf.predict(X_test_scaled)
    score_test.append(accuracy_score(y_test, y_test_hat))
```

The *split* method generates indices to split data into training and test set.

Now, we can use data scaling before building each model!

In [15]:
```
score_train, score_test
```

Out[15]:
```
([0.9833333333333333, 0.9833333333333333, 1.0, 0.975, 0.9833333333333333],
 [0.9, 1.0, 0.9666666666666667, 1.0, 0.9])
```

Benefits of Cross-Validation

- **Each data point will be in the test set exactly once.**
  - The model needs to generalize well to all of the data points in the dataset for all of the cross-validation scores (and their mean) to be high.
  - Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset.

- **Data are used more effectively.**
  - When using *train_test_split*, we usually use 75% of the data for training and 25% of the data for evaluation.
  - When using 10-fold cross-validation, we can use nine-tenths of the data (90%) to fit the model, and more data will usually result in more accurate models.

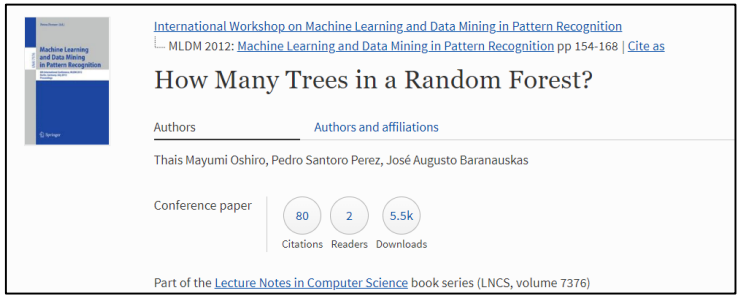
- **The main disadvantage is increased computational cost.**
  - As we train *k* models instead of a single model, cross-validation will be roughly *k* times slower than *train_test_split*.

Grid Search

Hyperparameter Search Space

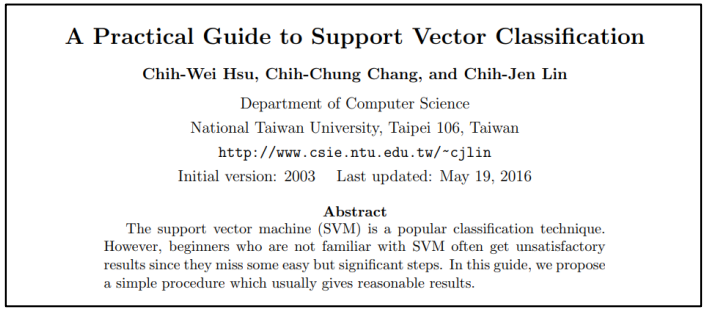
- **The smaller the better?**
  - set as small as possible

- **The larger the better?**
  - set as large as possible (*e.g.*, the number of trees in a random forest)



… … … The analysis of 29 datasets shows that from 128 trees there is no more significant difference between the forests using 256, 512, 1024, 2048 and 4096 trees. The mean and the median AUC values do not present major changes from 64 trees. Therefore, ***it is possible to suggest, based on the experiments, a range between 64 and 128 trees in a forest****. With these numbers of trees it is possible to obtain a good balance between AUC, processing time, and memory usage. … … …*

- **Otherwise (most cases)**
  - Hyperparameter search on a logarithmic scale
    (*e.g.*, regularization hyperparameter C and kernel bandwidth hyperparameter γ)



A Practical Guide to Support Vector Classification

Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin

Department of Computer Science
National Taiwan University, Taipei 106, Taiwan
http://www.csie.ntu.edu.tw/~cjlin

Initial version: 2003 Last updated: May 19, 2016

Abstract

The support vector machine (SVM) is a popular classification technique. However, beginners who are not familiar with SVM often get unsatisfactory results since they miss some easy but significant steps. In this guide, we propose a simple procedure which usually gives reasonable results.

We recommend a "grid-search" on C and γ using cross-validation. Various pairs of (C, γ) values are tried and the one with the best cross-validation accuracy is picked. We found that trying exponentially growing sequences of C and γ is a practical method to identify good parameters (for example, $C = 2^{-5}, 2^{-3}, \ldots, 2^{15}$, $\gamma = 2^{-15}, 2^{-13}, \ldots, 2^{3}$).

Grid Search

- We can improve the model's generalization performance by tuning its hyperparameters.
  - We discussed the hyperparameter settings of many of the learning algorithms.
  - It is important to understand what the hyperparameters mean before trying to adjust them.
  - Finding the values of the important hyperparameters of a model (the ones that provide the best generalization performance) is a tricky but necessary task.

- The most commonly used method is ***grid search***, which basically means trying **all possible combinations** of the hyperparameters of interest.

Grid Search

- **Example**: Kernelized SVM(SVC) with an RBF kernel
  - Two important hyperparameters
    - Regularization hyperparameter ***C***
    - Kernel bandwidth hyperparameter ***gamma***
  - Say we want to try the values 0.001, 0.01, 0.1, 1, 10, and 100 for the hyperparameter C, and the same for gamma.
  - We have 36 combinations of hyperparameters in total.

	C = 0.001	C = 0.01	…	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	…	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	…	SVC(C=10, gamma=0.01)
…	…	…	…	…
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	…	SVC(C=10, gamma=100)

scikit-learn Practice

- **Example (*iris* dataset)**

In [17]:
```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [18]:
```
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:

        clf = SVC(gamma=gamma, C=C)
        clf.fit(X_train_scaled, y_train)

        y_test_hat = clf.predict(X_test_scaled)
        score = accuracy_score(y_test, y_test_hat)

        if score > best_score:
            best_score = score
            best_hyperparameters = {'C': C, 'gamma': gamma}

print('Best score: {:.5f}'.format(best_score))
print('Best hyperparameters: {}'.format(best_hyperparameters))
```

We train an SVC for each combination of hyperparameters

If we got a better score, store the score and hyperparameters

```
Best score: 0.97368
Best hyperparameters: {'C': 100, 'gamma': 0.001}
```

The Danger of Overfitting the Hyperparameters

- Given the result, we might be tempted to report that we found a model that performs with 97% accuracy on our dataset.

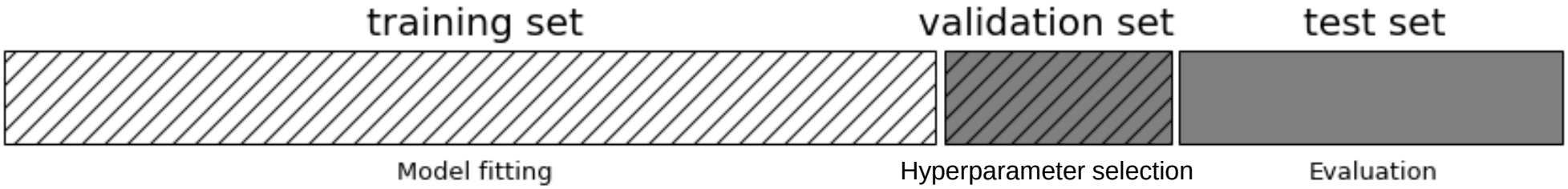
- However, this claim could be overly optimistic (or just wrong), for the following reason:
  - We tried many different hyperparameters and selected the one with best accuracy on the test set.
  - But this accuracy won't necessarily carry over to new data.
  - Therefore, we can no longer use the test set to assess how good the model is.

- It is important to keep a separate test set, which is only used for the final evaluation.
- We need an independent dataset to select hyperparameters → **validation set**

Validation Set

- One way to resolve this problem is to split the data into three sets:

- **Training set:** to build the model
- **Validation set:** to select the hyperparameters of the model
- **Test set:** to evaluate the performance with the selected hyperparameters.



scikit-learn Practice

- **Example (*iris* dataset) with data scaling**

In [19]:
```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = load_iris()
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, test_size=0.25, random_state=0)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, test_size=0.25, random_state=1)

print('Size of training set: {} size of validation set: {} size of test set:'
      ' {}₩n'.format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))
```

```
Size of training set: 84 size of validation set: 28 size of test set: 38
```

We split the data into three sets: training, validation, and test sets.

In [20]:
```
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
```

The data scaler is fitted on the training set (X_train)

scikit-learn Practice

- **Example (*iris* dataset) with data scaling**

In [21]:
```
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        clf = SVC(gamma=gamma, C=C)
        clf.fit(X_train_scaled, y_train)

        y_valid_hat = clf.predict(X_valid_scaled)
        score = accuracy_score(y_valid, y_valid_hat)

        if score > best_score:
            best_score = score
            best_hyperparameters = {'C': C, 'gamma': gamma}

print('Best score on validation set: {:.5f}'.format(best_score))
print('Best hyperparameters: {}'.format(best_hyperparameters))
```

```
Best score on validation set: 0.92857
Best hyperparameters: {'C': 100, 'gamma': 0.001}
```

We select the best hyperparameters using the validation set.

In [22]:
```
scaler.fit(X_trainval)
X_trainval_scaled = scaler.transform(X_trainval)
X_test_scaled = scaler.transform(X_test)

clf = SVC(**best_hyperparameters)
clf.fit(X_trainval_scaled, y_trainval)

y_test_hat = clf.predict(X_test_scaled)
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))
```

```
Test set score with best hyperparameters: 0.97368
```

The data scaler is fitted on X_trainval.

We build a model on both the training and validation data.

Grid Search with Cross-Validation

- The method of splitting the data into a training set, a validation set, and a test set is commonly used.

- But the outcome is quite sensitive to how exactly the data is split, especially when the data size is small.

- For a better estimate of the generalization performance, we can use cross-validation to evaluate the performance of each hyperparameter combination.
  - Cross-validation is often used in conjunction with hyperparameter search methods like grid search.
  - The main downside of the use of cross-validation is the time it takes to train all these models.

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

sklearn.model_selection.GridSearchCV

class sklearn.model_selection.**GridSearchCV**(*estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False*) [source]

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method. It also implements "score_samples", "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Read more in the User Guide.

Parameters:

estimator : *estimator object.*
This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid : *dict or list of dictionaries*
Dictionary with parameters names (`str`) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

refit : *bool, str, or callable, default=True*

Refit an estimator using the best found parameters on the whole dataset.

For multiple metric evaluation, this needs to be a `str` denoting the scorer that would be used to find the best parameters for refitting the estimator at the end.

Where there are considerations other than maximum score in choosing a best estimator, `refit` can be set to a function which returns the selected `best_index_` given `cv_results_`. In that case, the `best_estimator_` and `best_params_` will be set according to the returned `best_index_` while the `best_score_` attribute will not be available.

The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GridSearchCV` instance.

Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

See `scoring` parameter to know more about multiple metric evaluation.

scikit-learn Practice

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Attributes:

best_estimator_ : *estimator*

Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if `refit=False`.

See `refit` parameter for more information on allowed values.

best_score_ : *float*

Mean cross-validated score of the best_estimator

For multi-metric evaluation, this is present only if `refit` is specified.

This attribute is not available if `refit` is a function.

best_params_ : *dict*

Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is present only if `refit` is specified.

scikit-learn Practice

- **Example (*iris* dataset) with data scaling**

In [23]:
```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

iris = load_iris()
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, test_size=0.25, random_state=0)

scaler = StandardScaler()
scaler.fit(X_trainval)
X_trainval_scaled = scaler.transform(X_trainval)
X_test_scaled = scaler.transform(X_test)
```

The data scaler is fitted on X_trainval.

In [24]:
```
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
hyperparam_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
                   'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid_search = GridSearchCV(SVC(), hyperparam_grid, scoring='accuracy', refit=True, cv=kfold)
grid_search.fit(X_trainval_scaled, y_trainval)

print('Best score on validation set: {:.5f}'.format(grid_search.best_score_))
print('Best hyperparameters: {}'.format(grid_search.best_params_))
```

```
Best score on validation set: 0.97312
Best hyperparameters: {'C': 10, 'gamma': 0.1}
```

We specify the hyperparameter candidates to search over using a dictionary.

In [25]:
```
y_test_hat = grid_search.predict(X_test_scaled)
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))
```

```
Test set score with best hyperparameters: 0.97368
```

The *predict* method employs the model trained on both the training and validation data.

GridSearchCV allows the *hyperparam_grid* to be a list of dictionaries. *e.g.*,

```
param_grid = [{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

scikit-learn Practice

- **Example (*iris* dataset) with data scaling, without GridSearchCV**

In [27]:
```
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:

        clf = SVC(gamma=gamma, C=C)
        scores = cross_validate(clf, X_trainval_scaled, y_trainval, scoring='accuracy', cv=kfold,
                                return_train_score=True)
        score = scores['test_score'].mean()

        if score > best_score:
            best_score = score
            best_hyperparameters = {'C': C, 'gamma': gamma}

print('Best score on validation set: {:.5f}'.format(best_score))
print('Best hyperparameters: {}'.format(best_hyperparameters))
```

```
Best score on validation set: 0.97312
Best hyperparameters: {'C': 100, 'gamma': 0.01}
```

With five-fold cross-validation, we need to train 36 * 5 = 180 models.

The mean validation accuracy is computed for each hyperparameter setting.

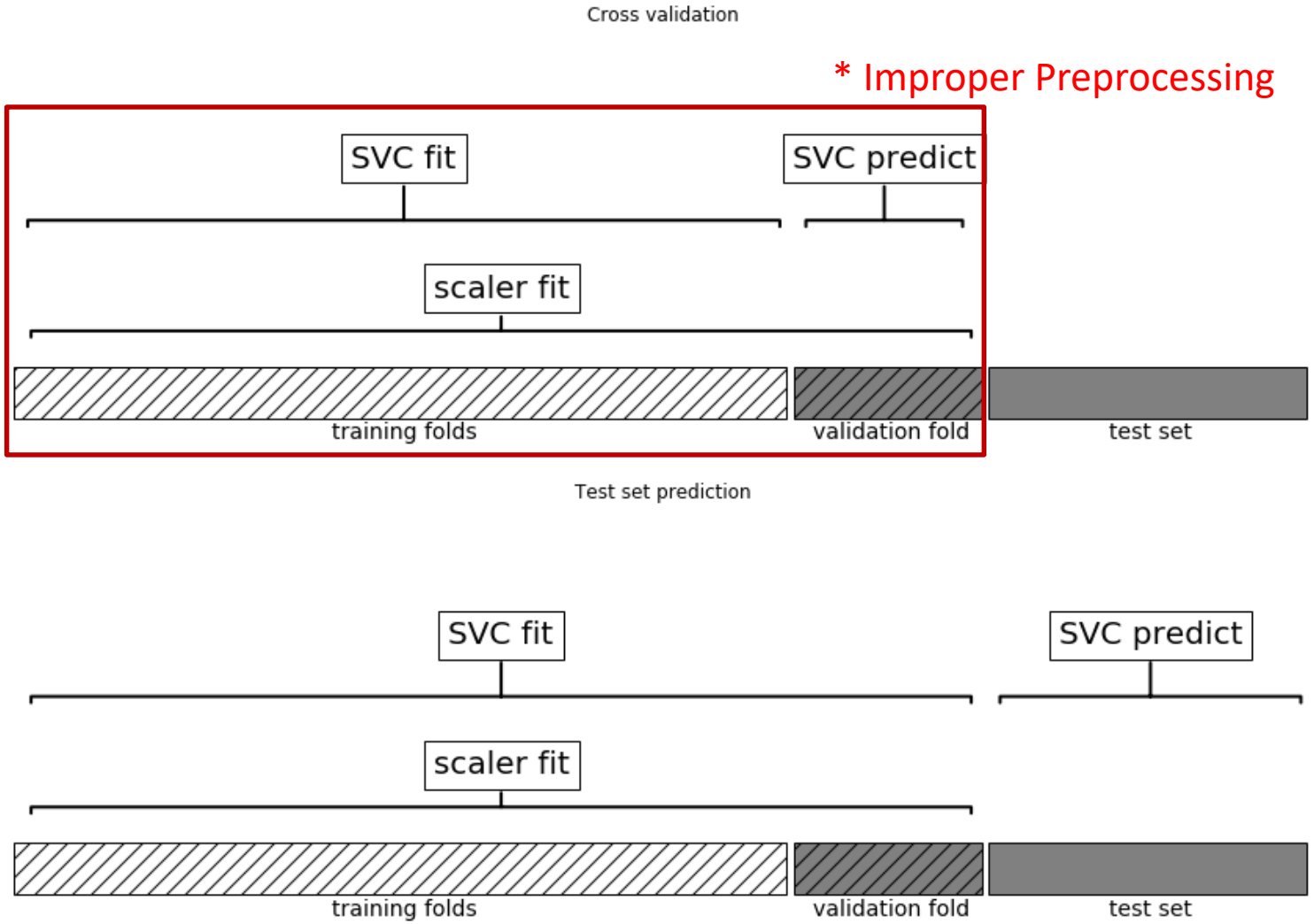
In [28]:
```
clf = SVC(**best_hyperparameters)
clf.fit(X_trainval_scaled, y_trainval)

y_test_hat = clf.predict(X_test_scaled)
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))
```

```
Test set score with best hyperparameters: 0.97368
```

We build a model on both the training and validation data.

scikit-learn Practice

- **Example (*iris* dataset) with data scaling**



Cross-Validated Committees

- After grid search with k-fold cross-validation, the *k* models for the best hyperparameter setting are provided.

- Instead of rebuilding a model on both training and validation data, we construct an ensemble of the *k* models and use it to evaluate the performance on the test set.

- In the case of classification,
  - **Hard-voting:** uses the predicted class labels of the *k* models for majority rule voting.
  - **Soft-voting:** predicts the class label based on the average per-class probability estimate of the *k* models.
    - The output of *predict_proba* or *decision_function* method.

scikit-learn Practice

- **Example (*iris* dataset) without GridSearchCV**

In [29]:
```
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:

        clf = SVC(gamma=gamma, C=C)
        scores = cross_validate(clf, X_trainval_scaled, y_trainval, scoring='accuracy', cv=kfold,
                                return_train_score=True, return_estimator=True)
        score = scores['test_score'].mean()

        if score > best_score:
            best_score = score
            best_models = scores['estimator']
            best_hyperparameters = {'C': C, 'gamma': gamma}

print('Best score on validation set: {:.5f}'.format(best_score))
print('Best hyperparameters: {}'.format(best_hyperparameters))
```

```
Best score on validation set: 0.97312
Best hyperparameters: {'C': 100, 'gamma': 0.01}
```

If we got a better score, store the models trained during the cross-validation

In [30]:
```
from scipy.stats import mode

y_test_hats = []
for baseclf in best_models:
    y_test_hats.append(baseclf.predict(X_test_scaled))

y_test_hat = mode(y_test_hats, axis=0)[0]
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))
```

```
Test set score with best hyperparameters: 0.97368
```

The **hard-voting** ensemble of five models is used to evaluate the performance on the test set.

scikit-learn Practice

- **Example (*iris* dataset) without GridSearchCV**

In [29]:
```
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
best_score = 0
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:

        clf = SVC(gamma=gamma, C=C)
        scores = cross_validate(clf, X_trainval_scaled, y_trainval, scoring='accuracy', cv=kfold,
                                return_train_score=True, return_estimator=True)
        score = scores['test_score'].mean()

        if score > best_score:
            best_score = score
            best_models = scores['estimator']
            best_hyperparameters = {'C': C, 'gamma': gamma}

print('Best score on validation set: {:.5f}'.format(best_score))
print('Best hyperparameters: {}'.format(best_hyperparameters))
```

If we got a better score, store the models trained during the cross-validation

```
Best score on validation set: 0.97312
Best hyperparameters: {'C': 100, 'gamma': 0.01}
```

In [31]:
```
classes = best_models[0].classes_

y_test_hats = []
for baseclf in best_models:
    y_test_hats.append(baseclf.decision_function(X_test_scaled))

y_test_hat = classes[np.argmax(np.mean(y_test_hats, axis=0), axis=1)]
test_score = accuracy_score(y_test, y_test_hat)
print('Test set score with best hyperparameters: {:.5f}'.format(test_score))
```

The **soft-voting** ensemble of five models is used to evaluate the performance on the test set.

```
Test set score with best hyperparameters: 0.97368
```

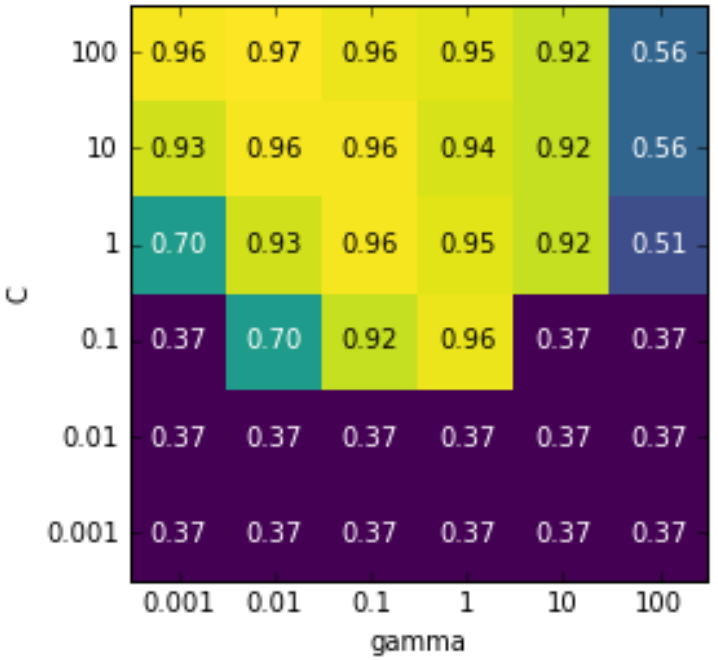
Analyzing the Result of Cross-Validation

- As grid searches are quite computationally expensive to run, it is often a good idea to start with a relatively coarse and small grid.
  - We can inspect the results of the cross-validated grid search, and possibly expand our search.
  - It is helpful to visualize the results of cross-validation to understand how the model generalization depends on the hyperparameters we are searching.

- If we are searching a two-dimensional grid of hyperparameters, this is best visualized as a heat map.
  - Each point in the heat map corresponds to one run of cross-validation, with a particular hyperparameter setting.
  - The color encodes the cross-validation accuracy, with light colors meaning high accuracy and dark colors meaning low accuracy.

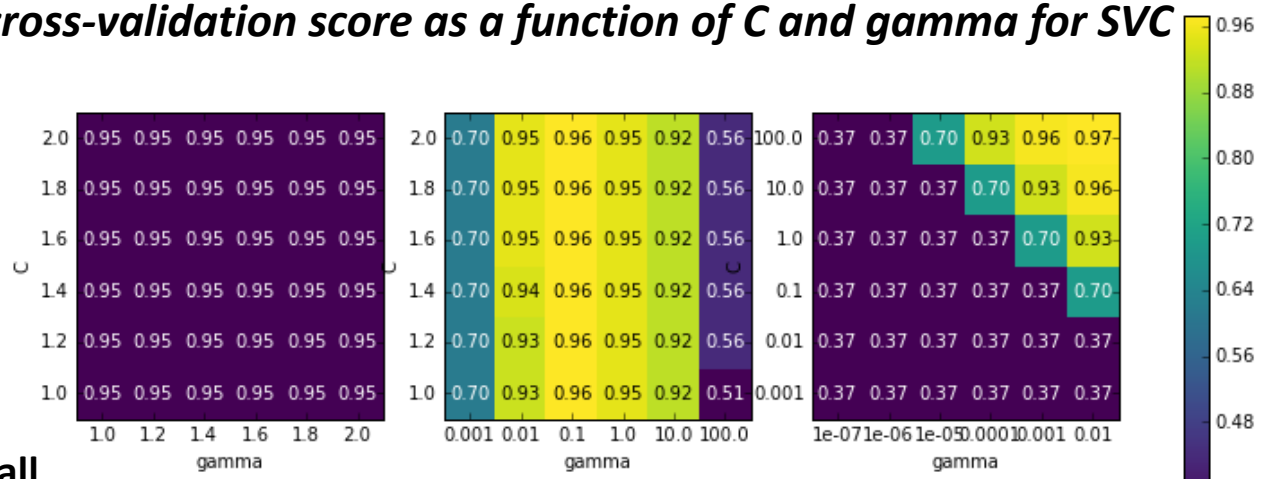
Analyzing the Result of Cross-Validation

- **Example:** ***Heat map of mean cross-validation score as a function of C and gamma for SVC***
  - You can see that SVC is very sensitive to the setting of the hyperparameters.
    - For many of the hyperparameter settings, the accuracy is around 40%, which is quite bad.
    - For other settings the accuracy is around 96%. We can take away from this plot several things.
  - Both hyperparameters (C and gamma) matter a lot, as adjusting them can change the accuracy from 40% to 96%.



Analyzing the Result of Cross-Validation

- **Example:** ***Heat map of mean cross-validation score as a function of C and gamma for SVC***



- **The first panel: no changes at all**
  - This is often caused by improper scaling and range of the hyperparameters C and gamma.
  - If no change in accuracy is visible over the different hyperparameter settings, it could also be that a hyperparameter is just not important at all.
- **The second panel: a vertical stripe pattern**
  - This indicates that only the setting of the gamma hyperparameter makes any difference.
  - This could mean that the C hyperparameter is not important.
- **The third panel: changes in both C and gamma**
  - We can probably exclude the very small values from future grid searches.
  - As the current optimum hyperparameter setting is at the top right, we can expect that there might be even better values beyond this border.

Analyzing the Result of Cross-Validation

- Tuning the hyperparameter grid based on the cross-validation scores a good way to explore the importance of different hyperparameters.

- Evaluation of the test set should happen only once we know exactly what model we want to use.

- Again, you should not test different hyperparameter ranges on the final test set.

Nested Cross-Validation

- In the preceding examples, we split the data into training and test sets and performed cross-validation on the training set.
  - We still have a single split of the data into training and test sets.
  - This might make our results unstable and make us depend too much on this single split of the data.

- We can also use cross-validation to evaluate the test performance

→ **Nested Cross-Validation**

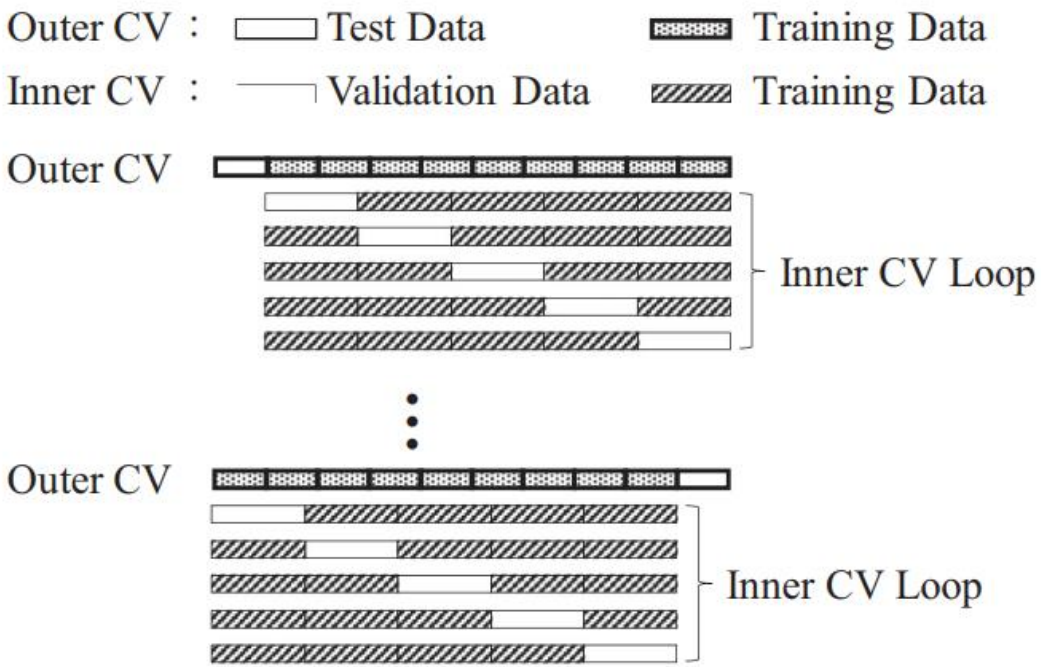
  - In nested cross-validation, there is an outer loop over splits of the data into training and test sets.
  - For each of them, a grid search is run (which might result in different best hyperparameters for each split in the outer loop).
  - Then, for each outer split, the test set score using the best settings is reported.

Nested Cross-Validation

- **"Nested Cross-Validation" (*a.k.a.*, Double Cross-Validation)**

  - **Outer CV:**
    (Training/Validation) vs Test

  - **Inner CV:**
    Training vs Validation



Nested Cross-Validation

- The result of nested cross-validation tells us how well a model generalizes, given the best hyperparameters found by the grid.

- It can be useful for evaluating how well a given model works on a dataset, especially when the dataset is small.

- Implementing nested cross-validation in scikit-learn is straightforward
  – Use *cross_validate* with an instance of *GridSearchCV* as the model

- As it doesn't provide a model that can be used on new data, nested cross-validation is rarely used when looking for a predictive model to apply to future data.

scikit-learn Practice

- **Example (*iris* dataset)**

In [32]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn.svm import SVC

iris = load_iris()
```

In [33]:
```
hyperparam_grid = [{'kernel': ['rbf'],
                    'C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
                   {'kernel': ['linear'],
                    'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

inner_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
outer_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
grid_search = GridSearchCV(SVC(), hyperparam_grid, scoring='accuracy', refit=True, cv=inner_kfold)
scores = cross_validate(grid_search, iris.data, iris.target, scoring='accuracy', cv=outer_kfold,
                        return_train_score=True, return_estimator=True)

print('Outer cross-validation score: %.5f'%scores['test_score'].mean())
```

```
Outer cross-validation score: 0.96667
```

We use stratified five-fold cross-validation in both the inner and the outer loop.

But, how do we apply data scaling?

scikit-learn Practice

- **Example (*iris* dataset) with data scaling**

In [34]:
```
from sklearn.preprocessing import StandardScaler

hyperparam_grid = [{'kernel': ['rbf'],
                    'C': [0.001, 0.01, 0.1, 1, 10, 100],
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
                   {'kernel': ['linear'],
                    'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

inner_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
outer_kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)

scaler = StandardScaler()
score_test = []
for trainval_idx, test_idx in outer_kfold.split(iris.data, iris.target):

    X_trainval = iris.data[trainval_idx]
    y_trainval = iris.target[trainval_idx]
    X_test = iris.data[test_idx]
    y_test = iris.target[test_idx]

    scaler.fit(X_trainval)
    X_trainval_scaled = scaler.transform(X_trainval)
    X_test_scaled = scaler.transform(X_test)

    grid_search = GridSearchCV(SVC(), hyperparam_grid, scoring='accuracy', refit=True, cv=inner_kfold)
    grid_search.fit(X_trainval_scaled, y_trainval)

    y_test_hat = grid_search.predict(X_test_scaled)
    score_test.append(accuracy_score(y_test, y_test_hat))

print('Outer cross-validation score: %.5f'%np.mean(score_test))
```

```
Outer cross-validation score: 0.94667
```

To apply data scaling, we can use the *split* method again.

Discussion

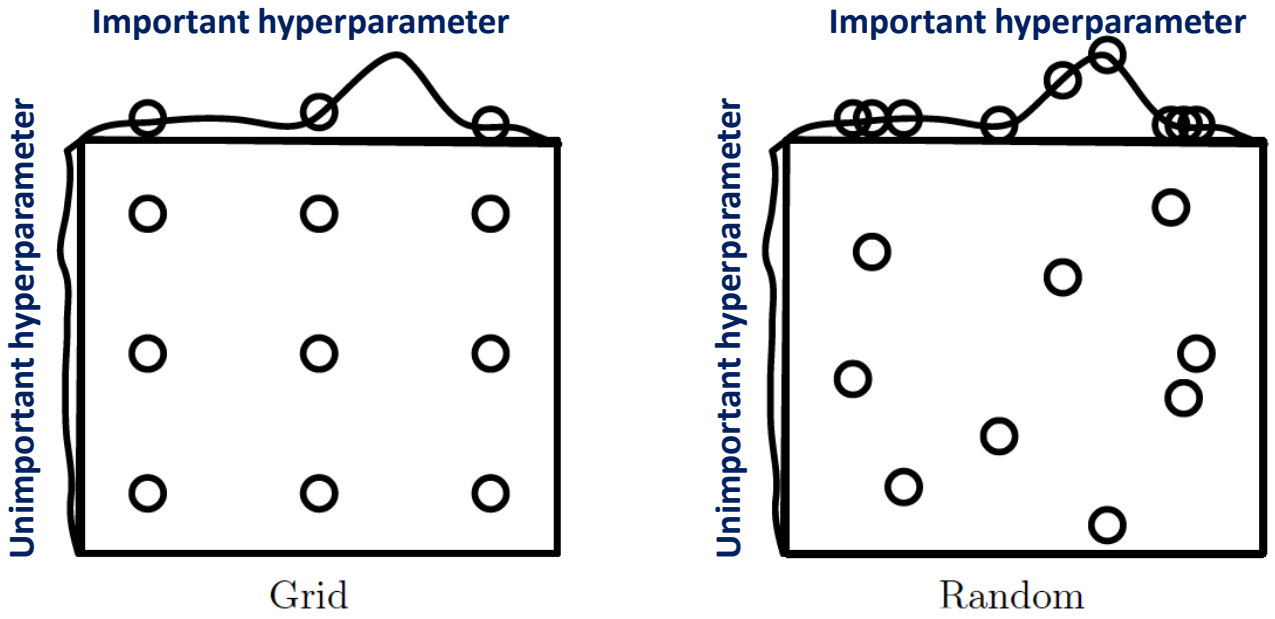
- **The distinction between the training set, validation set, and test set is fundamentally important to applying machine learning methods in practice.**
  - The most commonly used form is a training/test split for evaluation, and using cross-validation on the training set for model selection.
  - When the original dataset is too small, use “Nested Cross-Validation”.

- **Any choices made based on the test set “leak” information from the test set into the model.**

- **We can parallelize grid search and cross-validation**
  - Building a model using a particular hyperparameter setting on a particular cross-validation split can be done completely independently from the other hyperparameter settings and models
  - If you are using a multi-core CPU, you can use the *n_jobs* parameter to adjust the number of CPU cores to use.

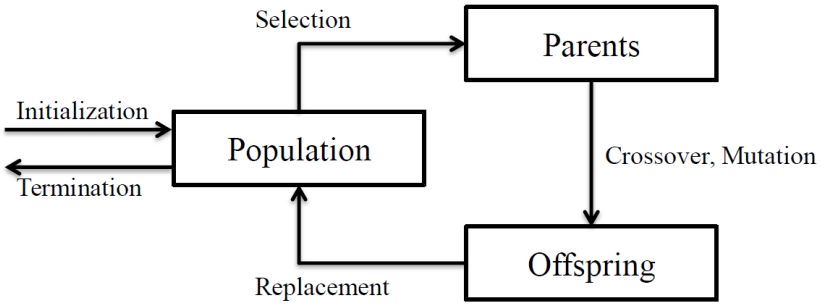
Advanced Hyperparameter Selection Methods

- **Random Search**
  - Search by random sampling from the hyperparameter space
  - Random search is often more efficient than grid search.
  - Prior knowledge can be incorporated by specifying the distribution from which to sample.
  - All the trials are independent → Easy to parallelize

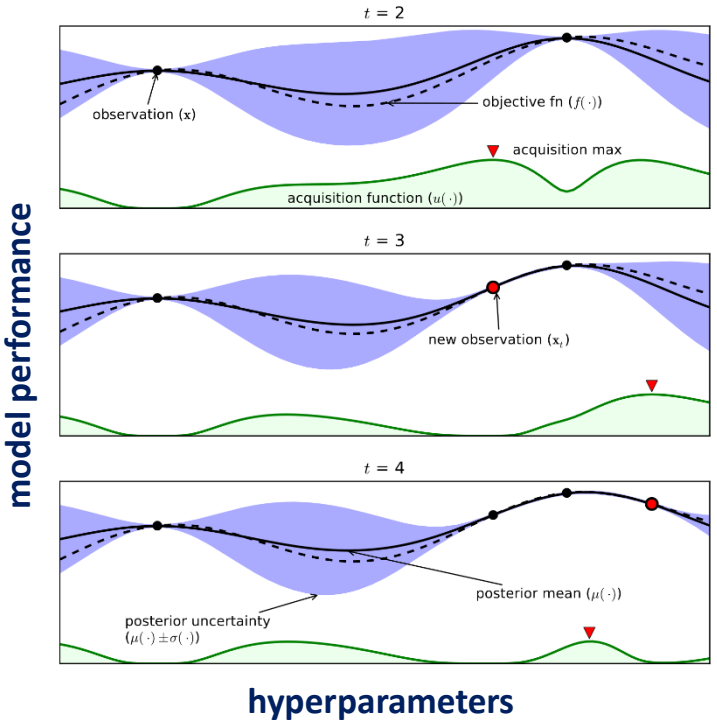


Advanced Hyperparameter Selection Methods

- **Meta-heuristics**
  - efficiently explore the hyperparameter space to find near-optimal solution (*e.g.,* Genetic algorithm, Particle swarm optimization)



- **Model-based Optimization**
  - find the functional relationship between hyperparameters and model performance – surrogate model (*e.g.,* Bayesian optimization)



Summary and Outlook

Evaluation Metrics and Scoring

Scoring	Function
Classification	
'accuracy'	metrics.accuracy_score
'balanced_accuracy'	metrics.balanced_accuracy_score
'average_precision'	metrics.average_precision_score
'brier_score_loss'	metrics.brier_score_loss
'f1'	metrics.f1_score
'f1_micro'	metrics.f1_score
'f1_macro'	metrics.f1_score
'f1_weighted'	metrics.f1_score
'f1_samples'	metrics.f1_score
'neg_log_loss'	metrics.log_loss
'precision' etc.	metrics.precision_score
'recall' etc.	metrics.recall_score
'jaccard' etc.	metrics.jaccard_score
'roc_auc'	metrics.roc_auc_score
Regression	
'explained_variance'	metrics.explained_variance_score
'max_error'	metrics.max_error
'neg_mean_absolute_error'	metrics.mean_absolute_error
'neg_mean_squared_error'	metrics.mean_squared_error
'neg_mean_squared_log_error'	metrics.mean_squared_log_error
'neg_median_absolute_error'	metrics.median_absolute_error
'r2'	metrics.r2_score

Using Evaluation Metrics in Model Selection

- **Calculate it directly using a scoring function, *e.g.*,**

sklearn.metrics.**mean_absolute_error**(*y_true, y_pred, *, sample_weight=None, multioutput='uniform_average'*) [source]

sklearn.metrics.**roc_auc_score**(*y_true, y_score, *, average='macro', sample_weight=None, max_fpr=None, multi_class='raise', labels=None*) [source]

sklearn.metrics.**confusion_matrix**(*y_true, y_pred, *, labels=None, sample_weight=None, normalize=None*) [source]

- **Or, you can simply provide a string describing the evaluation metric you want to use as the *scoring* parameter of the following classes.**

sklearn.model_selection.cross_validate

sklearn.model_selection.**cross_validate**(*estimator, X, y=None, *, groups=None, scoring=None, cv=None, n_jobs=None, verbose=0, fit_params=None, pre_dispatch='2*n_jobs', return_train_score=False, return_estimator=False, error_score=nan*) [source]

sklearn.model_selection.GridSearchCV

class sklearn.model_selection.**GridSearchCV**(*estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False*) [source]

Summary and Outlook

- **We've discussed …**
  - The cornerstones of evaluating and improving machine learning algorithms: *evaluation metrics, cross-validation,* and *grid search.*
  - The model evaluation and selection techniques are the most important tools in a data scientist's toolbox.

Practical Guideline

- **The importance of the evaluation metric used for model selection and evaluation**
  - Make sure that the metric you choose to evaluate and select a model for is a good stand-in for what the model will actually be used for.
  - In reality, classification problems rarely have balanced classes, and often false positives and false negatives have very different consequences.
  - You need to understand what these consequences are, and pick an evaluation metric accordingly.

Practical Guideline

- **The use of a test set**
  - The test set allows us to evaluate a machine learning model as it will perform in the future.
  - **If we use the test set to select model hyperparameters, using the same data to evaluate how well our model will do in the future will lead to overly optimistic estimates.**
  - We therefore need to resort to a split into:
    (1) training data for model building,
    (2) validation data for model selection,
    (3) test data for model evaluation.
  - We can replace each of these splits with cross-validation.