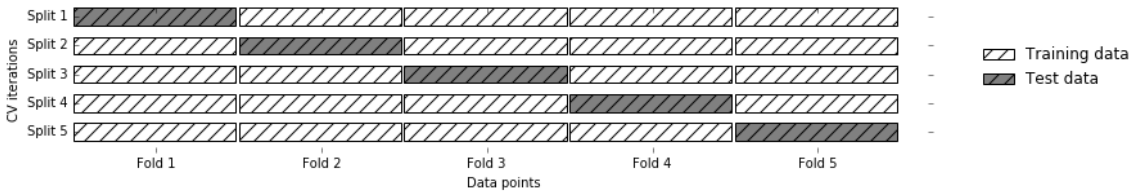
L7 – Model Evaluation and Improvement

- Evaluating models and selecting parameters with focus on:
  - Supervised methods: regression and classification
  - The method we have learned:
    1) Split our dataset: the train_test_split function
    2) Build a model on the training set: the fit method
    3) Evaluate on the test set: the score method
  - We are interested in measuring how well our model generalizes to new, previously unseen data
- Here we expend on two aspects of this evaluation:
  - cross-validation: a more robust assessment of generalization
  - grid search: an effective method for adjusting paramters

Cross-Validation

- Statistical method of evaluating generalization performance
  - The dataset is split repeatedly and multiple models are trained
  - Commonly used: *k-**fold cross-validation***
  - With *k* a user-specified number, usually 5 or 10
- Steps of *k-**fold cross-validation***
  - The data is first partitioned into k parts of (approximately) equal size, called folds
  - Next, a sequence of models is trained
  - In the end, we can collect *k* accuracy values

```
import mglearn
mglearn.plots.plot_cross_validation()
```



- Cross-validation is implemented in scikit-learn using the cross_val_score function from model_selection module
  - Parameters:
    - The model want to evaluate
    - The training data
    - The ground-truth labels

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
iris = load_iris()                          logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

  - Change the number of folds by changing the cv parameter

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=10)
print("Cross-validation scores: {}".format(scores))
```

  - A common way to summarize the cross-validation accuracy: to compute the mean

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

  - Observe a relatively high variance in the accuracy between folds
  - This is caused by the small size of the dataset
- A second function for cross-validation is cross_validate, which returns a dictionary containing:
  - The training and test times
  - The training score (optional) and the test score

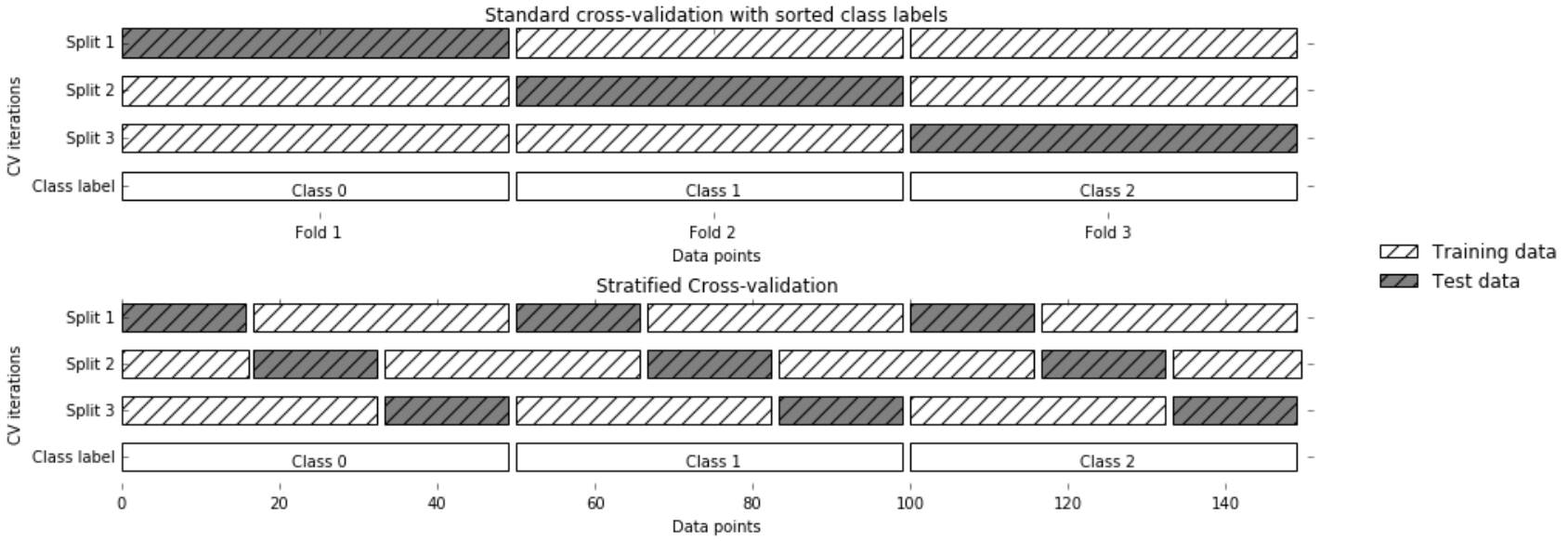
```
from sklearn.model_selection import cross_validate
res = cross_validate(logreg, iris.data, iris.target, cv=5, return_train_score=True)
display(res)
# using pandas, the results can be nicely displayed
import pandas as pd
res_df = pd.DataFrame(res)
display(res_df)
print("Mean times and scores:\n", res_df.mean())
```

- Benefit of Cross-Validation
  - Avoid the unrealistic generate by the “lucky” or “unlucky” caused by the random splitting a dataset into training and test sets
  - The data is used more effectively
    - For 5-fold cross-validation, 80% data are used for training
    - For 10-fold, 90% data are used
- Disadvantage
  - Increased computational cost
  - Not a way to build a model that can be applied to new data
  - The purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset
  - It will be a problem when there is strong order in the dataset, e.g.

```
from sklearn.datasets import load_iris
iris = load_iris()
print("Iris labels:\n{}".format(iris.target))
```

- To solve this problem, stratified *k*-fold cross-validation
  - Simple *k*-fold strategy failed on the datasets with strong order
  - Stratified *k*-fold cross-validation results in more reliable estimates of generalization performance
  - See how the cross-validation is generated in the stratified one

mglearn.plots.plot_stratified_cross_validation()



- More control of cross-validation can be realized
  - Using the KFold splitter class from model_selection module

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
print("Cross-validation scores:\n{}".format(cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

  - When using three-fold, we can verify it is indeed a very bad idea

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

  - Another way to resolve this problem is to shuffle the data instead of stratifying the folds
    - Setting the shuffle parameter of KFold to be True
    - Setting a fixed value of random_state to get a reproducible shuffling

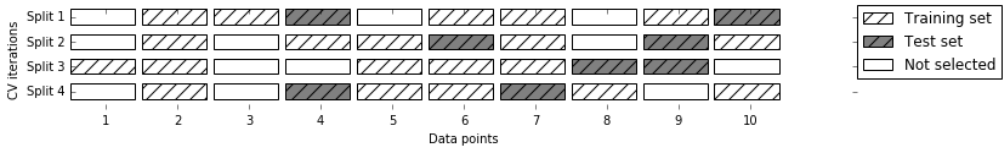
```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

  - Another frequently used cross-validation method is *leave-one-out*
    - Consider as k-fold cross-validation where each fold is a single sample
    - Time-consuming; but may provides better estimates on small datasets

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

- Shuffle-split Cross-Validation
  - Each split samples train_size many points from the training set
  - Each split samples test_size many (disjoint) points from the test set
  - This splitting is repeated n_splits times

mglearn.plots.plot_shuffle_split()



  - This shows a demo of four iterations of splitting a dataset consisting of 10 points, with a training set of 5 points and test tests of 2 points each
  - You can use integers for the absolute size or floating-point numbers to specify the fractions of the whole dataset

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

- Cross-validation with groups
  - When there are groups in the data that are highly related
    - Collect a dataset of pictures of 100 people, where each person is captured multiple times
    - Random splitting is likely let pictures of the same person in both the training and the test sets
    - Must ensure the training and test sets contain images of different people
    - This example of groups in the data is common in medical applications and also in speech recognition
  - Use GroupKFold, which takes an array of groups as argument

```
from sklearn.model_selection import GroupKFold
# create synthetic dataset
X, y = make_blobs(n_samples=12, random_state=0)
# assume the first three samples belong to the same group, then the next four, etc.
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
mglearn.plots.plot_group_kfold()  # visualize each group is entirely in the training or the test set
```

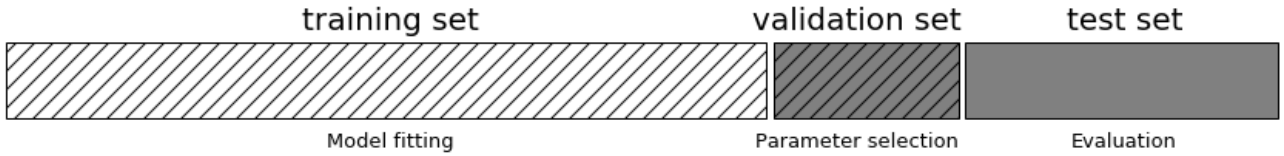
Grid Search

- Finding the values of the important parameters of a model is a tricky task, but necessary for almost all models
- The most common method is Grid Search
  - Basically means trying all possible combinations of parameters
  - To improve the model's generalization performance
  - Consider the case of a kernel SVM with an RBF kernel
  - Two parameters:
    1) The kernel bandwidth, gamma
    2) The regularization parameter, C
  - We can implement a simple grid search by for-loops

```
# naive grid search implementation
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, random_state=0)
print("Size of training set: {} size of test set: {}".format(X_train.shape[0], X_test.shape[0]))

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]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)

        # evaluate the SVC on the test set
        score = svm.score(X_test, y_test)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

- **Danger** of overfitting the parameters and the validation set
  - We used the test data to adjust the parameters
  - We can no longer use it to assess how good the model is
  - We need an independent dataset to evaluate, one that was not used to create the model
  - Solution: to split the data again into three sets
    1) The training set to build the model
    2) The validation (or development) set to select parameters
    3) The test set to evaluate the performance of selected parameters

mglearn.plots.plot_threefold_split()



- After selecting the best parameters using the validation set, we can rebuild a model by training on both the training data and the validation data
- In this way, we can use as much as possible to build our model

```
from sklearn.svm import SVC
# split data into train+validation set and test set
X_trainval, X_test, y_trainval, y_test = train_test_split(iris.data, iris.target, random_state=0)
# split train+validation set into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(X_trainval, y_trainval, 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]))

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]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the validation set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

```
# rebuild a model on the combined training and validation set, and evaluate it on the test set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Best score on validation set: {:.2f}".format(best_score))
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

- The score on the test set (e.g., 92%) is lower than the best score on the validation set (e.g., 96%)
- Thus, we can only claim the accuracy of 92%
- The distinction between the training set, the validation set and the test set is fundamentally important
- It is important to keep a separate test set only for final evaluation
- To enhance the robustness of splitting method, we can use cross-validation to evaluate the performance of each parameter combination

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# rebuild a model on the combined training and validation set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

- The main downside of using cross-validation is computing time
- This visualization illustrate how the best parameter is selected

```
mglearn.plots.plot_cross_val_selection()
```

- Grid search with cross-validation as commonly used
  - scikit-learn provides the GridSearchCV class
  - We first define the grid of parameters

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],  'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))
```

  - Then instantiate the GridSearchCV class with the SVC model

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

  - Fitting the GridSearchCV object will
    - Not only search for the best parameters but also automatically fits a new model with the best performance

```
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, random_state=0)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
```

    - Convenient interface to access the trained model by predict & score

```
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))
```

- Access to the actual model was found by the best_estimator_ attribute

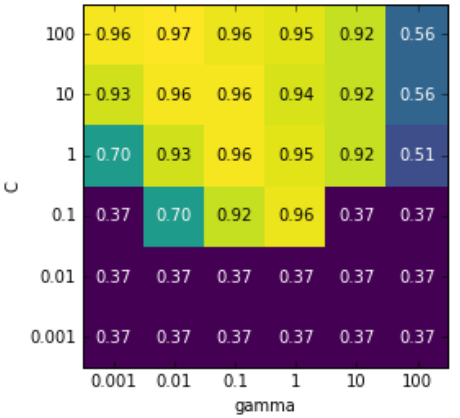
```
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

- Analyzing the result of cross-validation
  - Grid searches are quite computational expensive, often it is a good idea to start with a relatively coarse and small grid
  - The result can be found in the cv_results_ attribute

```
import pandas as pd
# convert to DataFrame
results = pd.DataFrame(grid_search.cv_results_)
# show the first 5 rows
display(results.head())
```

  - As we were searching a two-dimensional grid, which can be visualized as a heat map

```
scores = np.array(results.mean_test_score).reshape(6, 6)
# plot the mean cross-validation scores
mglearn.tools.heatmap(scores, xlabel='gamma',
                      xticklabels=param_grid['gamma'], ylabel='C',
                      yticklabels=param_grid['C'], cmap="viridis")
```



- Range of search is very important
- See some less meaningful ones below

```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))
param_grid_linear = {'C': np.linspace(1, 2, 6), 'gamma': np.linspace(1, 2, 6)}
param_grid_one_log = {'C': np.linspace(1, 2, 6), 'gamma': np.logspace(-3, 2, 6)}
param_grid_range = {'C': np.logspace(-3, 2, 6), 'gamma': np.logspace(-7, -2, 6)}
for param_grid, ax in zip([param_grid_linear, param_grid_one_log, param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)
    # plot the mean cross-validation scores
    scores_image = mglearn.tools.heatmap(
                scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
                yticklabels=param_grid['C'], cmap="viridis", ax=ax)
plt.colorbar(scores_image, ax=axes.tolist())
```

- First panel shows no change at all
- Second panel shows a vertical strip pattern
- Third panel shows changes in both C and gamma but nothing happens

- Search over spaces that are not grids
  - GridSearchCV allows the para_grid to be a list of dictionaries

```
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]}]
print("List of grids:\n{}".format(param_grid))

grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

- Parallelizing cross-validation and grid search
  - By setting the n_jobs parameter to the number of CPU cores
  - The n_jobs parameter is available for both GridSearchCV and cross_val_score
  - You can set n_jobs = -1 to use all available cores

Evaluating Metrics and Scoring

- We have learned to evaluate
  - Classification performance using accuracy (the fraction of correctly classified samples)
  - Regression performance using R^2
  - These are only two of the many possible ways to summarize how well a supervised model performs on a given dataset
- Keep the End Goal in Mind
  - Need to think about the high-level goal of the application, often called the *business metric*
  - Application-based preference needs to be considered

- Metrics for Binary Classification
  - Let's look at the ways in which accuracy might be misleading
  - For binary classification, we often speak of a *positive* class and a *negative* class
  - Classifiers will make mistakes; we need to ask *what the consequences of these mistakes might be* in the real world
    - A healthy patient will be classified as positive, leading to additional testing (some costs and an inconvenience for the patient) – an incorrect positive prediction is called a ***false positive*** (also known as type I error)
    - A sick patient will be classified as negative, the undiagnosed cancer might lead to serious health issues – such an incorrect negative prediction is called a ***false negative*** (also known as type II error)
  - The consequence of false positives and false negatives are rarely the same

- Imbalanced datasets
  - Datasets in which one class is much more frequent than the other; these datasets are often called *imbalanced datasets* or *datasets with imbalanced classes*
  - In reality, imbalanced data is quite normal
  - Influence of imbalanced dataset, an example:
    - For a dataset with 99 positive and 1 negative samples, let's say you build a classifier that is 99% accurate on the positive sample.
    - 99% accuracy sounds very impressive but this doesn't take the class imbalance into account
    - You can achieve 99% accuracy without building a machine learning model – i.e., by always 'predicting' positive
  - In summary, accuracy doesn't allow us distinguish the constant "positive" model from a potentially good model
  - New evaluation method is needed!!!

- Let's create a 9:1 imbalance dataset from the digits dataset by classifying the digital 9 against the nine other classes

```
from sklearn.datasets import load_digits
digits = load_digits()
y = digits.target == 9
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=0)
```

  - First, we can use the DymmyClassifier to always predict the majority class (here "not nine")

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Unique predicted labels: {}".format(np.unique(pred_most_frequent)))
print("Test score: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

  - Compare this against using an actual classifier

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Test score: {:.2f}".format(tree.score(X_test, y_test)))
```

Result of *DecisionTree* is only slightly better, possible reason:
1) Sth wrong when using DecisionTree
2) Accuracy is in fact not a good measurement here

- Let's try two more classifiers on the same dataset

```
dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("dummy score: {:.2f}".format(dummy.score(X_test, y_test)))

from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("logreg score: {:.2f}".format(logreg.score(X_test, y_test)))
```

  - LogisticRegression produces very good results
  - However, random classifier yields over 80% accuracy

- The problem here is that accuracy is an inadequate measure for quantifying predictive performance in this imbalanced setting
  - Other measurements / metrics are needed
  - One of the most comprehensive way: confusion matrices

```
from sklearn.metrics import confusion_matrix
confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

- 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

```
mglearn.plots.plot_confusion_matrix_illustration()
```

- According to four different terms
  - True Negative (TN) :: False Positive (FP)
  - False Negative (FN) :: True Positive (TP)

```
mglearn.plots.plot_binary_confusion_matrix()
```

- Now we can compare the performance of different classifiers

```
print("Most frequent class:")          print(confusion_matrix(y_test, pred_most_frequent))
print("\nDummy model:")                 print(confusion_matrix(y_test, pred_dummy))
print("\nDecision tree:")               print(confusion_matrix(y_test, pred_tree))
print("\nLogistic Regression")          print(confusion_matrix(y_test, pred_logreg))
```

- Idea result: more TN & TP and less FN & FP
  - Therefore, Logistic Regression performs the best in these tests

- Several ways to summarize the info. in confusion matrix
  - Relationship to accuracy: $\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN + FP + FN}}$
  - Precision: $\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$
    - Measure how many of the samples predicted as positive are true positive
    - Used as a performance metric when the goal is to limit the number of false positive
  - Recall: $\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$
    - Measure how many of the positive samples are captured by the positive prediction
    - Used as performance metric when need to identify all positive samples; i.e., when it is important to avoid false negative (e.g., cancer diagnosis)
  - f-score (or f-measure): $\text{F} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision+recall}}$
    - As a trade-off between optimizing the recall and the precision
    - With the harmonic mean of precision and recall
    - Is also known as the f_1-score – the higher the better
    - A disadvantage: is harder to interpret and explain

```
from sklearn.metrics import f1_score
print("f1 score most frequent: {:.2f}".format(f1_score(y_test, pred_most_frequent)))
print("f1 score dummy: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("f1 score tree: {:.2f}".format(f1_score(y_test, pred_tree)))
print("f1 score logistic regression: {:.2f}".format(f1_score(y_test, pred_logreg)))
```

- Comprehensive summary can be generated by classification_report
  - Majority

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent, target_names=["not nine", "nine"]))
```

  - Dummy model

```
print(classification_report(y_test, pred_dummy, target_names=["not nine", "nine"]))
```

  - Logistic regression

```
print(classification_report(y_test, pred_logreg, target_names=["not nine", "nine"]))
```

- Both classes need to be checked:
  - When looking at the “not nice” class, the difference between the dummy models and a very good model are not very clear
  - However, the difference is clear when looking at the “nine” class

- Taking value of decision-function into account
  - Most classifiers provide a decision_function or a predict_prob method to assess degrees of certainty about prediction
  - Using different decision thresholds leads to different performance

```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2], random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
mglearn.plots.plot_decision_threshold()
```

  - We can use the classification_report function to evaluate precision and recall for both classes

```
print(classification_report(y_test, svc.predict(X_test)))
```

  - Let's assume in our application it is important to have a high recall for class 1 (e.g., the cancer screening) – i.e., more points to be classified as class 1, so we decrease the threshold

```
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
print(classification_report(y_test, y_pred_lower_threshold))
```

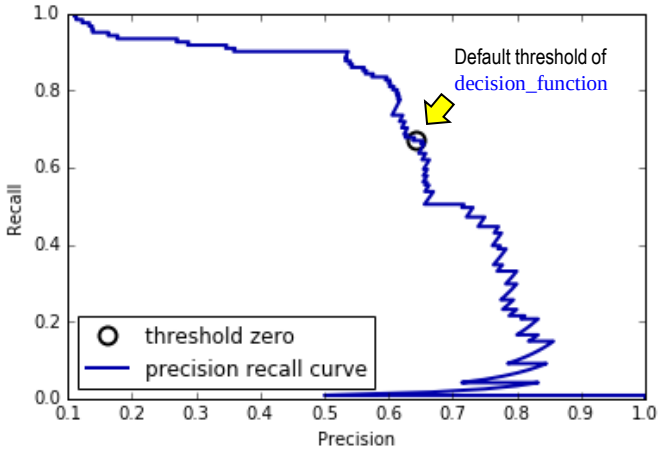
- Picking a threshold for models that implement the predict_proba method can be easier, as it is on a fixed 0 to 1 scale
  - By default, the threshold of 0.5 means that if more than 50% "sure" a point will be classified as positive
  - Increasing the threshold mean that the model needs to be more confident to make a positive decision (and less confident to make negative decision)

- Precision-Recall curves and ROC curves
  - Changing the threshold of decision-function is a way to adjust the trade-off of precision and recall for a given classifier
  - Setting a requirement on a classifier like 90% recall is often called setting the operating point
    - Fixing an operating point is often helpful in business settings to make performance guarantees to customers
    - Be more instructive to check all possible trade-offs of precision & recall at once
  - Using a tool called the *precision-recall curve*

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(y_test, svc.decision_function(X_test))
```

- The precision_recall_curve function returns a list of precision and recall values for all possible threshold in sorted order
- We can plot a curve

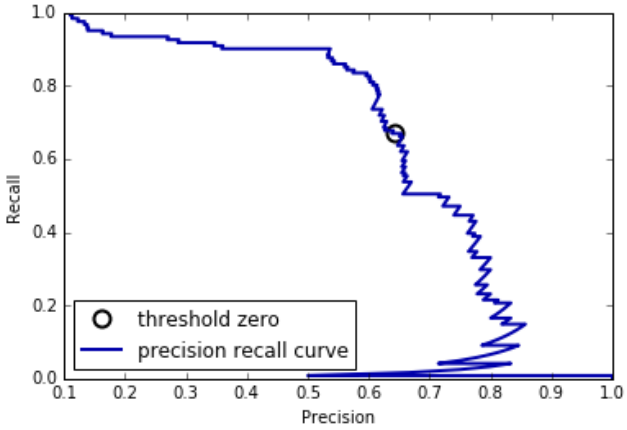
```
# Use more data points for a smoother curve
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2], random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(y_test, svc.decision_function(X_test))
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)
plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
plt.legend(loc="best")
```



- The closer a curve stays to the upper-right corner (i.e., both precision and recall are high), the better the classifier is
  - Raising the threshold moves the operation point toward higher precision but also lower recall
  - The model above is able to get a precision of up to 0.5 with very high recall
- Different classifiers can work well in different parts of the curve
  - Let's compare SVM with a random forest
  - RandomForestClassifier doesn't have a decision_function but only predict_proba
  - The precision_recall_curve function expects its 2nd argument a certain measure for the positive class (class 1) – so we pass the probability of a sample being class 1 as rf.predict_proba(X_test)[:,1]
  - Default threshold for predict_proba as 0.5 is marked as point on curve

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)
# RandomForestClassifier has predict_proba, but not decision_function
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(y_test, rf.predict_proba(X_test)[:, 1])
```

```
plt.plot(precision, recall, label="svc")
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero svc", fillstyle="none", c='k', mew=2)
plt.plot(precision_rf, recall_rf, label="rf")
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
         markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)
plt.xlabel("Precision")   plt.ylabel("Recall")   plt.legend(loc="best")
```



- The random forest performs better at the extremes
- Around the middle, the SVM performs better
- Check the f_1-score again, which only captures one point on the precision-recall curve (the one with default threshold)

```
from sklearn.metrics import f1_score
print("f1_score of random forest: {:.3f}".format(f1_score(y_test, rf.predict(X_test))))
print("f1_score of svc: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

- Differently, comparing two precision-recall curves provides a lot of detailed insight

- One particular way to summarize the precision-recall curve is by computing the integral or area under the curve of the precision-recall curve, also known as the *average precision*

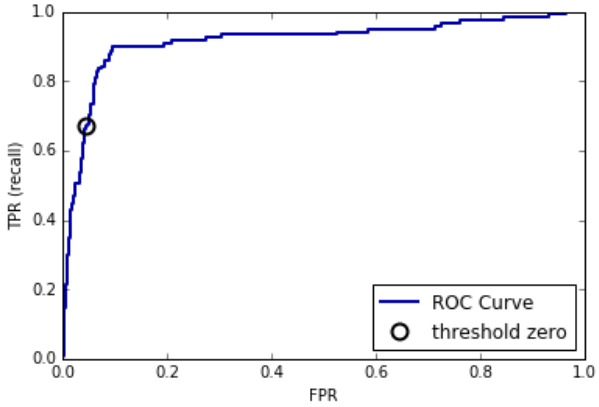
```
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Average precision of random forest: {:.3f}".format(ap_rf))
print("Average precision of svc: {:.3f}".format(ap_svc))
```

  - The random forest and SVC perform similarly well
  - The random forest even slightly ahead
  - This is quite different from the result we got from f1_score earlier

- **R**eceiver **O**perating **C**haracteristics (ROC)
  - Another tool to analyze the behavior of classifier at different thresholds as a curve (named as ROC curve)
  - It shows the *false positive rate* (FPR) against the *true positive rate* (TPR); TPR is also named as recall

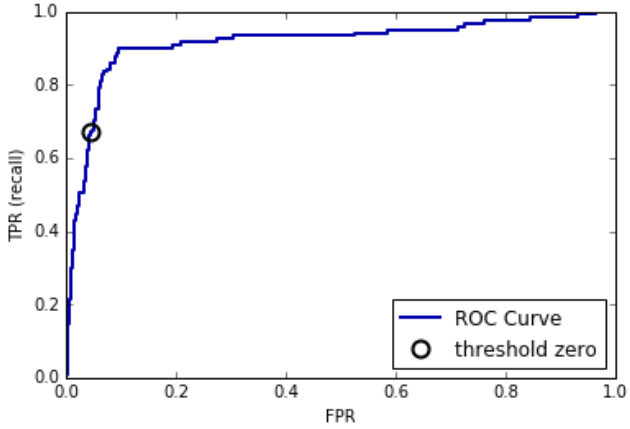
$$FPR = FP / (FP + TN)$$

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10, label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```



- For the ROC curve, the ideal curve is close to the top left: you want a classifier that produces a high recall while keeping a low false positive rate.
  - We can achieve a significantly higher recall (around 0.9) while only increasing the FPR slightly – a good way to optimize the threshold
  - The point closest to the top left might be a better operating point than the one chosen by default
- Let's conduct a comparison of the random forest and the SVM using ROC curves

```
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])
plt.plot(fpr, tpr, label="ROC Curve SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC Curve RF")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero SVC", fillstyle="none", c='k', mew=2)
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr[close_default_rf], '^', markersize=10,
         label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```



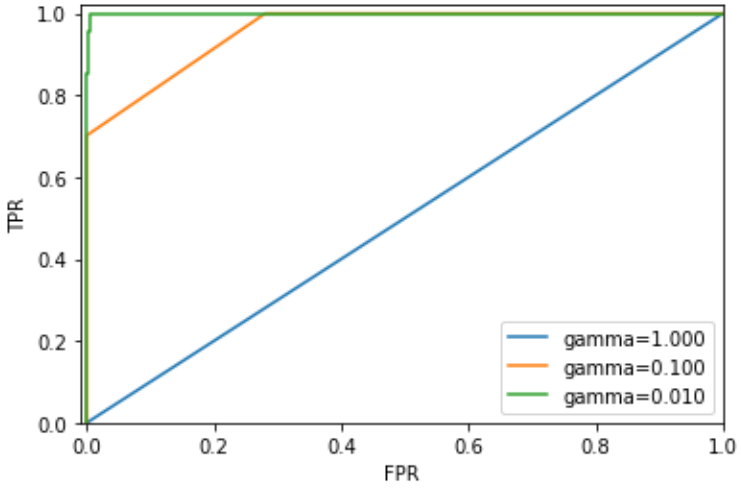
- As for the precision-recall curve, we often want to summarize the ROC curve using a single number, ***A**rea **U**nder the **C**urve* (AUC)

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

- AUC is a much better metric for imbalanced classification problems than accuracy
  - A perfect AUC of 1 means that all positive points have a higher score than all negative points
  - For classification problems with imbalanced classes, using AUC for model selection is often much more meaningful
- Let's go back to the problem we studied earlier of classifying all nines in the digits dataset versus all other digits
  - Using SVM with three different settings of the kernel bandwidth, gamma

```
from sklearn.datasets import load_digits          digits = load_digits()
y = digits.target == 9
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=0)
plt.figure()
for gamma in [1, 0.1, 0.01]:
        svc = SVC(gamma=gamma).fit(X_train, y_train)
        accuracy = svc.score(X_test, y_test)
        auc = roc_auc_score(y_test, svc.decision_function(X_test))
        fpr, tpr, _ = roc_curve(y_test , svc.decision_function(X_test))
        print("gamma = {:.2f} accuracy = {:.2f} AUC = {:.2f}".format(gamma, accuracy, auc))
        plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))
plt.xlabel("FPR")   plt.ylabel("TPR")   plt.xlim(-0.01, 1)   plt.ylim(0, 1.02)   plt.legend(loc="best")
```

- The accuracy of all three settings of gamma is the same, 90%
  - This might be the same as chance performance, or it might not
  - With gamma=0.1, performance drastically improves to an AUC of 0.96
- Finally, with gamma=0.01, we get a perfect AUC of 1.0
  - That means that all positive points are ranked higher than all negative points according to the decision function.
  - In other words, with the right threshold, this model can classify the data perfectly!
- We highly recommend using AUC when evaluating models on imbalanced data
- Adjusting the decision threshold might be necessary to obtain useful classification results from a model with a high AUC

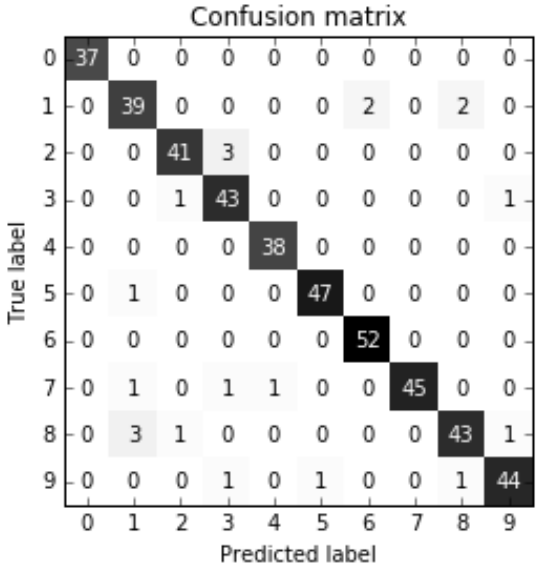


- Metrics for Multiclass Classification
  - All are derived from binary classification metrics (e.g., average)
  - When classes are imbalanced, accuracy is not a good measure
  - Common tools: the confusion matrix and the classification report
  - See the handwriting digits example below

```
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Accuracy: {:.3f}".format(accuracy_score(y_test, pred)))
print("Confusion matrix:\n{}".format(confusion_matrix(y_test, pred)))
```

  - You can find a visually more appealing plot

```
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Predicted label',
    ylabel='True label', xticklabels=digits.target_names,
    yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))
```

- The most commonly used metric for imbalanced datasets is the multiclass version of the f-score
- Idea behind: to compute one binary f-score per class, with that class being the positive class and the other classes making up the negative classes
- Then, these per-class f-scores are averaged using one of the following strategies
  - "macro" averaging computes the unweighted per-class f-scores. This gives equal weight to all classes, no matter what their size is.
  - "weighted" averaging computes the mean of the per-class f-scores, weighted by their support as what is reported in the classification report.
  - "micro" averaging computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and f-score using these counts.

- If you care about each sample equally much, it is recommended to use the "micro” average f1-score
- if you care about each class equally much, it is recommended to use the "macro" average f1-score

```
from sklearn.metrics import f1_score
print("Micro average f1 score: {:.3f}".format(f1_score(y_test, pred, average="micro")))
print("Macro average f1 score: {:.3f}".format(f1_score(y_test, pred, average="macro")))
```

Using Evaluation Metrics in Model Selection

- We often want to use metrics like AUC in model selection using GridSearchCV or cross_val_score
  - Can be realized easily by changing the score from the default (accuracy) to roc_auc

```
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

# default scoring for classification is accuracy
print("Default scoring: {}".format(cross_val_score(SVC(), digits.data, digits.target == 9)))
# providing scoring="accuracy" doesn't change the results
explicit_accuracy = cross_val_score(SVC(), digits.data, digits.target == 9, scoring="accuracy")
print("Explicit accuracy scoring: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9, scoring="roc_auc")
print("AUC scoring: {}".format(roc_auc))
```

– Similarly, we can change the metric used to pick the best parameters in GridSearchCV

```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target == 9, random_state=0)
# we provide a somewhat bad grid to illustrate the point:
param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
# using the default scoring of accuracy:
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Grid-Search with accuracy")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (accuracy)): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

– Then, we change to select by AUC

```
# using AUC scoring instead:
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc") # "roc_auc"=>"average_precision"
grid.fit(X_train, y_train)
print("\nGrid-Search with AUC")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (AUC): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

- In summary, when using accuracy and AUC, different values of the parameter gamma are selected
- Using AUC found a better parameter setting in terms of AUC and even in terms of accuracy

- Summary
  - We discussed cross-validation, grid search & evaluation metrics
  - Two important particular points:
    - The cross-validation is often overlooked by new practitioners
    - The importance of the evaluation metric or scoring function used for model selection and model evaluation
  - Imbalanced Dataset
    - Always keep in mind the influence
    - Better evaluation metrics can improve
    - But in practice, we still need preprocessing as what we learned before