CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep dari pembelajaran mesin kemudian di terapkan pada permasalahan – (C2)
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin untuk mendapatkan pemecahan maslaah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 13
MODEL SELECTION

DESKRIPSI TEMA

Mahasiswa mempelajari cara pembangunan model sehingga model bisa tervalidasi dengan baik dan optimal

CAPAIAN PEMBELAJARAN MINGGUAN (SUB CPMK)

Mahasiswa dapat memahami dan mampu membuat model yang tervalidasi sehingga menghasilkan model pembelajaran yang optimal dengan Bahasa pemrogramam Python

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

Having discussed the fundamentals of supervised and unsupervised learning, and having explored a variety of machine learning algorithms, we will now dive more deeply into evaluating models and selecting parameters. We will focus on the supervised methods, regression and classification, as evaluating and selecting models in unsupervised learning is often a very qualitative process.

1. Import libraries

```
import pandas as pd
import os
import mglearn
import numpy as np
import matplotlib.pyplot as plt
```

2. To evaluate our supervised models, so far we have split our dataset into a training set and a test set using the train_test_split function, built a model on the training set by calling the fit method, and evaluated it on the test set using the score method, which for classification computes the fraction of correctly classified samples.

```
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# create a synthetic dataset
X, y = make_blobs(random_state=0)
# split data and labels into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instantiate a model and fit it to the training set
logreg = LogisticRegression().fit(X_train, y_train)
# evaluate the model on the test set
print("Test set score: {:.2f}".format(logreg.score(X_test, y_test)))
```

```
Test set score: 0.88
```

Remember, the reason we split our data into training and test sets is that we are interested in measuring how well our model generalizes to new, previously unseen data. We are not

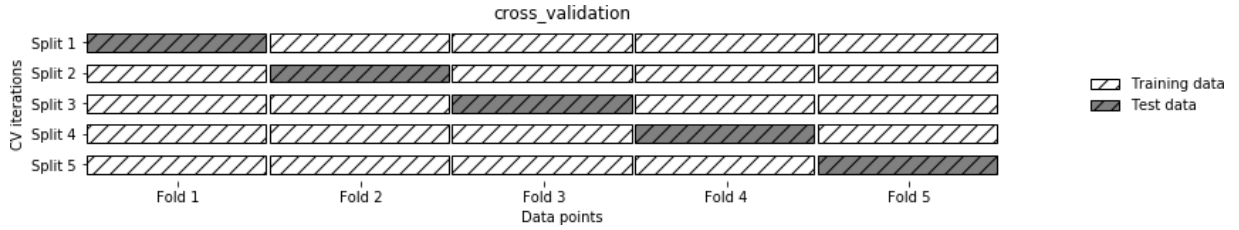
interested in how well our model fit the training set, but rather in how well it can make predictions for data that was not observed during training.

Cross-Validation

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

3. The most commonly used version of cross-validation is k-fold cross-validation, where k is a user- specified number, usually 5 or 10.

```
1 mglearn.plots.plot_cross_validation()
```



4. Cross-validation is implemented in scikit-learn using the cross_val_score function from the model_selection module. The parameters of the cross_val_score function are the model we want to evaluate, the training data, and the ground-truth labels.

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

```
Cross-validation scores: [0.96078431 0.92156863 0.95833333]
```

5. By default, cross_val_score performs three-fold cross-validation, returning three accuracy values. We can change the number of folds used by changing the cv parameter.

```
1 scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
2 print("Cross-validation scores: {}".format(scores))
3
```

```
Cross-validation scores: [1.         0.96666667 0.93333333 0.9        1.        ]
```

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

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

```
Average cross-validation score: 0.96
```

Using the mean cross-validation we can conclude that we expect the model to be around 96% accurate on average.

Stratified k-Fold Cross-Validation and Other Strategies

Splitting the dataset into *k* folds by starting with the first one-*k*-th part of the data, as described in the previous section, might not always be a good idea.

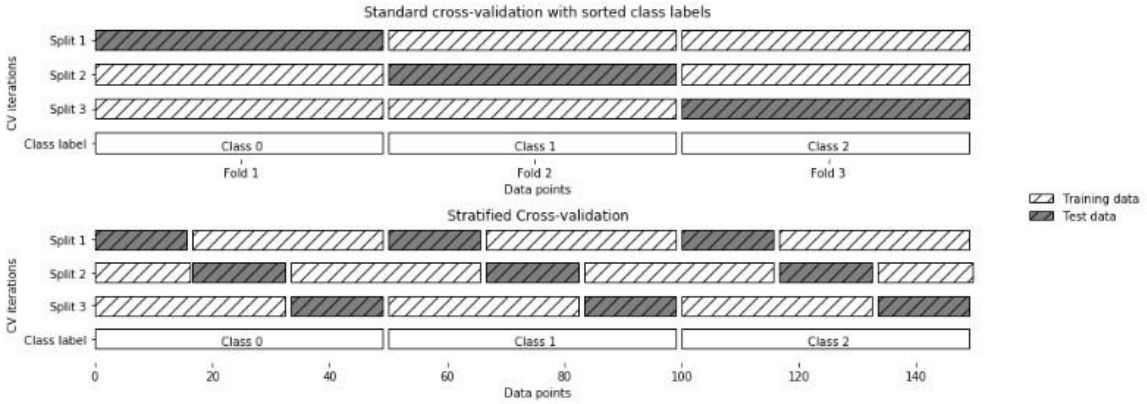
```
In [7]: from sklearn.datasets import load_iris
        iris = load_iris()
        print("Iris labels:\n{}".format(iris.target))
```

```
Iris labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

As you can see, the first third of the data is the class 0, the second third is the class 1, and the last third is the class 2. Imagine doing three-fold cross-validation on this dataset. The first fold would be only class 0, so in the first split of the data, the test set would be only class 0, and the training set would be only classes 1 and 2.

6. As the simple *k*-fold strategy fails here, scikit-learn does not use it for classification, but rather uses *stratified 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.

```
In [8]: mglearn.plots.plot_stratified_cross_validation()
```



It is usually a good idea to use stratified *k*-fold cross-validation instead of *k*-fold cross-validation to evaluate a classifier, because it results in more reliable estimates of generalization performance.

More Control Over Cross-Validation

7. Scikit-learn allows for much finer control over what happens during the splitting of the data by providing a *crossvalidation splitter* as the cv parameter. To do this, we first have to import the **KFold** splitter class from the model_selection module and instantiate it with the number of folds we want to use.

```
In [9]: from sklearn.model_selection import KFold
        kfold = KFold(n_splits=5)
```

```
In [10]: print("Cross-validation scores:\n{}".format(
             cross_val_score(logreg, iris.data, iris.target, cv=kfold)))

Cross-validation scores:
[1.         0.93333333 0.43333333 0.96666667 0.43333333]
```

This way, we can verify that it is indeed a really bad idea to use three-fold (nonstratified) cross-validation on the iris dataset.

```
In [11]: kfold = KFold(n_splits=3)
         print("Cross-validation scores:\n{}".format(
             cross_val_score(logreg, iris.data, iris.target, cv=kfold)))

Cross-validation scores:
[0. 0. 0.]
```

Another way to resolve this problem is to shuffle the data instead of stratifying the folds, to **remove** the **ordering** of the samples by label. We can do that by setting the **shuffle** parameter of KFold to **True**. If we shuffle the data, we also need to fix the **random_state** to get a reproducible shuffling.

```
In [12]: 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)))

Cross-validation scores:
[0.9  0.96 0.96]
```

Leave-one-out Cross-Validation

8. Another frequently used cross-validation method is *leave-one-out*. You can think of leave-one-out cross-validation as *k*-fold cross-validation where each fold is a single sample. For each split, you pick a single data point to be the test set.

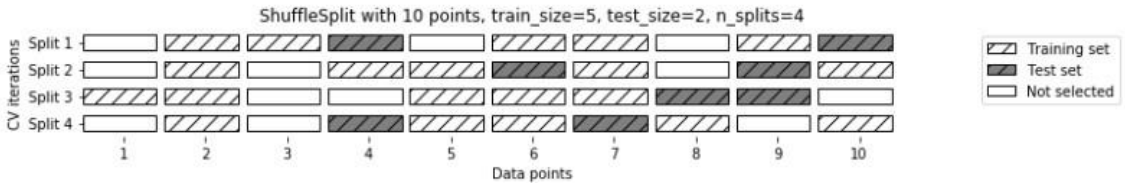
```
In [13]: from sklearn.model_selection import LeaveOneOut
         loo = LeaveOneOut()
         scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
         print("Number of cv iterations:", len(scores))
         print("Mean accuracy: {:.2f}".format(scores.mean()))

Number of cv iterations: 150
Mean accuracy: 0.95
```

Shuffle-split Cross-Validation

Another, very flexible strategy for cross-validation is *shuffle-split cross-validation*. In shuffle-split cross- validation, each split samples train_size many points for the training set and test_size many (disjoint) point for the test set. This splitting is repeated n_iter times.

```
In [14]: mglearn.plots.plot_shuffle_split()
```



9. The following code splits the dataset into 50% training set and 50% test set for 10 iterations.

```
In [15]: 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 scores:
[0.93333333 0.92       0.92       0.90666667 0.89333333 0.94666667
 0.93333333 0.82666667 0.97333333 0.93333333]
```

Cross-Validation with Groups

Another very common setting for cross-validation is when there are groups in the data that are highly related. Say you want to build a system to recognize emotions from pictures of faces, and you collect a dataset of pictures of 100 people where each person is captured multiple times, showing various emotions. The goal is to build a classifier that can correctly identify emotions of people not in the dataset.

10. The following is an example of using a synthetic dataset with a grouping given by the groups

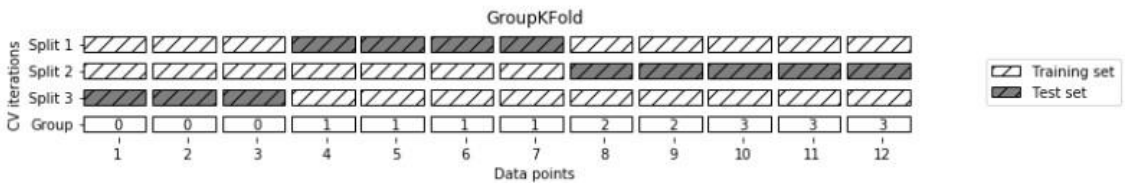
```
In [16]: from sklearn.model_selection import GroupKFold
         # create synthetic dataset
         x, y = make_blobs(n_samples=12, random_state=0)
         # assume the first three samples belong to the same group,
         # then the next four, etc.
         groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
         scores = cross_val_score(logreg, x, y, groups, cv=GroupKFold(n_splits=3))
         print("Cross-validation scores:\n{}".format(scores))
```

```
Cross-validation scores:
[0.75       0.8        0.66666667]
```

array.

As you can see, for each split, each group is either entirely in the training set or entirely in the test set.

```
In [17]: mglearn.plots.plot_group_kfold()
```



Grid Search

Now that we know how to evaluate how well a model generalizes, we can take the next step and improve the model's generalization performance by tuning its parameters. Finding the values of the important parameters of a model (the ones that provide the best generalization performance) is a tricky task, but necessary for almost all models and datasets. Because it is such a common task, there are standard methods in scikit-learn to help you with it. The most commonly used method is ***grid search***, which basically means trying all possible combinations

In [18]:
```
# 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))
```

```
Size of training set: 112   size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

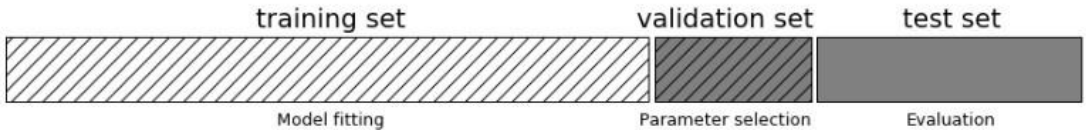
of the parameters of interest.

Danger of Overfitting the Parameters and Validation Set

One way to resolve this problem is to split the data again, so we have three sets: the **training** set to build the model, the **validation** (or development) set to select the parameters of the model, and the **test** set to evaluate the performance of the selected parameters.

After selecting the best parameters using the validation set, we can rebuild a model using the parameter settings we found, but now training on both the training data and the validation data. This way, we can use as much data as possible to build our model.

In [19]:
```
mglearn.plots.plot_threefold_split()
```



11. This leads to the following implementation:

In [20]:
```
# 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 test 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))
```

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

Best score on validation set: 0.96
Best parameters:  {'C': 10, 'gamma': 0.001}
Test set score with best parameters: 0.92
```

Grid Search with Cross-Validation

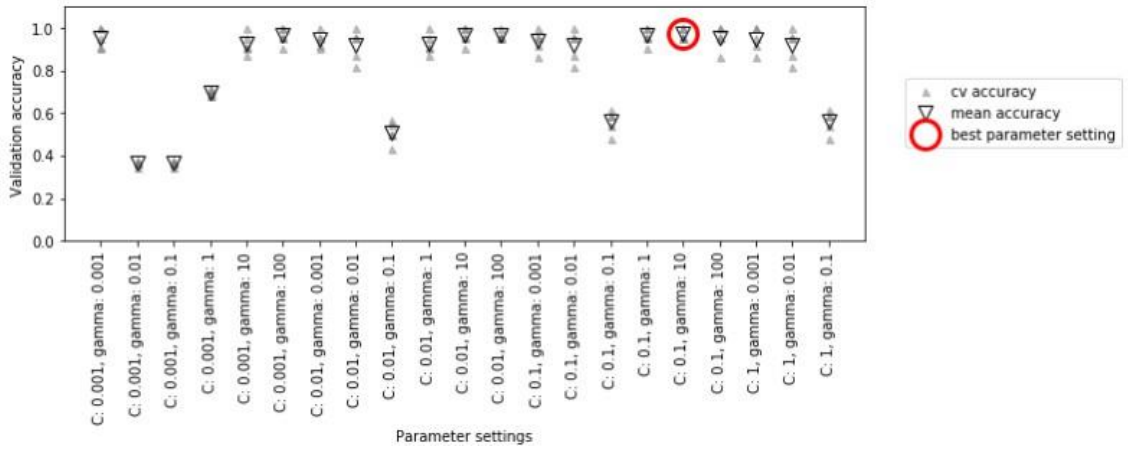
While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the previous code snippet we can see that GridSearchCV selects 'C': 10, 'gamma': 0.001 as the best parameters, while the output of the code in the previous section selects 'C': 100, 'gamma': 0.001 as the best parameters.

```
In [21]: 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)
```

```
Out[21]: SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

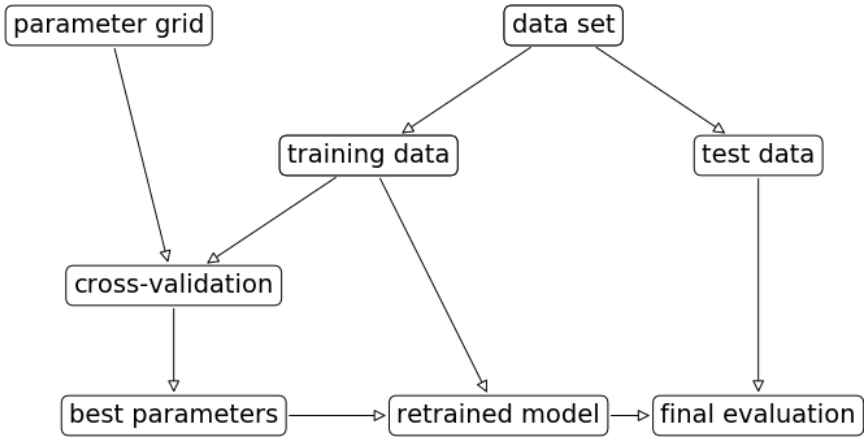
12. The following visualization (Figure 5-6) illustrates how the best parameter setting is selected in the preceding code.

```
In [22]: mglearn.plots.plot_cross_val_selection()
```



13. The overall process of splitting the data, running the grid search, and evaluating the final parameters.

```
In [23]: mglearn.plots.plot_grid_search_overview()
```



Because grid search with cross-validation is such a commonly used method to adjust parameters, scikit- learn provides the GridSearchCV class, which implements it in the form of an estimator. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model—in this case, C and gamma), and the values are the parameter settings we want to try out.

14. First Trying the values 0.001, 0.01, 0.1, 1, 10, and 100 for C and gamma translates to the following dictionary.

```
In [24]: 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))

         Parameter grid:
         {'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

We can now instantiate the GridSearchCV class with the model (SVC), the parameter grid to search (param_grid), and the cross-validation strategy we want to use (say, five-fold stratified

```
In [25]: from sklearn.model_selection import GridSearchCV
         grid_search = GridSearchCV(SVC(), param_grid, cv=5,
                                    return_train_score='True')
```

cross-validation).
We need to split the data into a training and a test set, to avoid overfitting the parameters.

```
In [26]: x_train, x_test, y_train, y_test = train_test_split(
             iris.data, iris.target, random_state=0)
```

```
In [27]: grid_search.fit(x_train, y_train)
```

```
Out[27]: GridSearchCV(cv=5, error_score='raise',
               estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False),
               fit_params=None, iid=True, n_jobs=1,
               param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
               pre_dispatch='2*n_jobs', refit=True, return_train_score='True',
               scoring=None, verbose=0)
```

To evaluate how well the best found parameters generalize, we can call score on the test set.

```
In [28]: print("Test set score: {:.2f}".format(grid_search.score(x_test, y_test)))
```

```
Test set score: 0.97
```

```
In [29]: print("Best parameters: {}".format(grid_search.best_params_))
         print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```

You can access the model with the best parameters trained on the whole training set using the **best_estimator**_ attribute.

```
In [30]: print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

Analyzing the Result of Cross-Validation

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

```
In [31]: # convert to DataFrame
         results = pd.DataFrame(grid_search.cv_results_)
         # show the first 5 rows
         display(results.head())
```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_C	param_gamma	params	r
0	0.0008	0.0004	0.366071	0.366079	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	
1	0.0006	0.0004	0.366071	0.366079	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	
2	0.0002	0.0008	0.366071	0.366079	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	
3	0.0004	0.0004	0.366071	0.366079	0.001	1	{'C': 0.001, 'gamma': 1}	
4	0.0000	0.0010	0.366071	0.366079	0.001	10	{'C': 0.001, 'gamma': 10}	

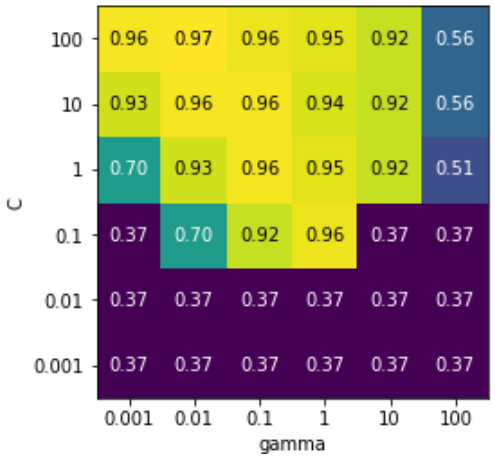
5 rows × 22 columns

First we extract the mean validation scores, then we reshape the scores so that the axes correspond to C and gamma.

In [32]:
```
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')
```

Out[32]:
```
<matplotlib.collections.PolyCollection at 0xc47ba58>
```



Now let's look at some plots where the result is less ideal, because the search ranges were not chosen properly.

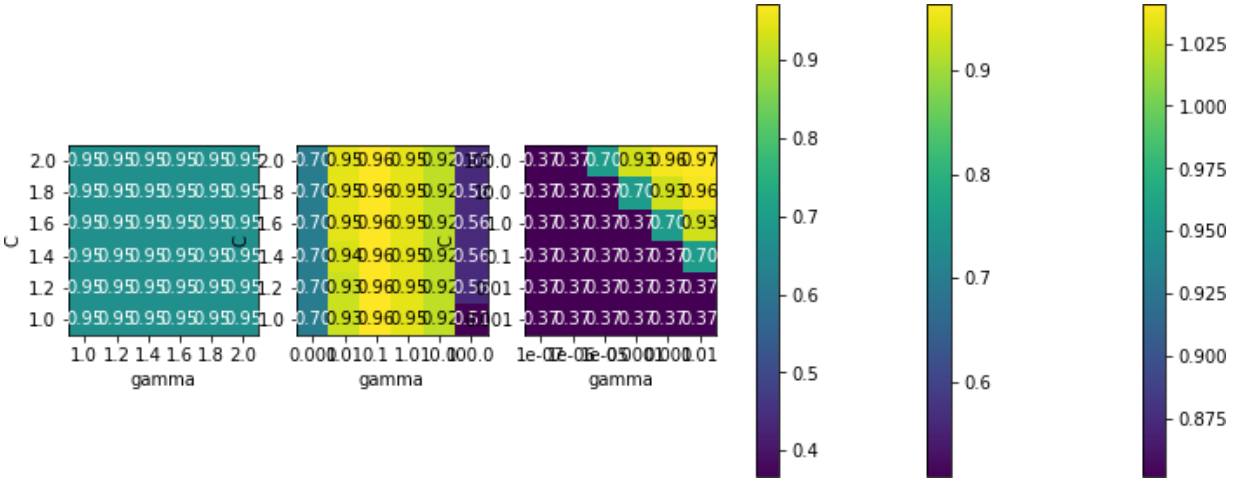
In [33]:
```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

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())
```



Search Over Spaces that are not Grids

In some cases, trying all possible combinations of all parameters as GridSearchCV usually does, is not a good idea. For example, SVC has a kernel parameter, and depending on which kernel is chosen, other parameters will be relevant. If kernel='linear', the model is linear, and only the C parameter is used. If kernel='rbf', both the C and gamma parameters are used (but not other parameters like degree). In this case, searching over all possible combinations of C, gamma, and kernel wouldn't make sense: if kernel='linear', gamma is not used, and trying different values for gamma would be a waste of time. To deal with these kinds of "conditional" parameters, GridSearchCV allows the param_grid to be a *list of dictionaries*. Each dictionary in the list is expanded into an independent grid.

15. A possible grid search involving kernel and parameters could look like this.

```
In [34]: 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))

         List of grids:
         [{'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]}]
```

16. Now let's apply this more complex parameter search.

```
In [35]: grid_search = GridSearchCV(SVC(), param_grid, cv=5, return_train_score='True')
         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_))

         Best parameters: {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'}
         Best cross-validation score: 0.97
```

17. Let's look at the **cv_results_** again.

```
In [36]: results = pd.DataFrame(grid_search.cv_results_)
         # we display the transposed table so that it better fits on the page:
         display(results.T)
```

	0	1	2	3	4	5	6	7
mean_fit_time	0.0006001	0.000600004	0.00100007	0.000399971	0.0006001	0.00100007	0.000999928	0.000600004
mean_score_time	0.000600004	0.000399971	0.000200033	0.000600052	0.000399971	0.000400114	0	0.000600052
mean_test_score	0.366071	0.366071	0.366071	0.366071	0.366071	0.366071	0.366071	0.366071
mean_train_score	0.366079	0.366079	0.366079	0.366079	0.366079	0.366079	0.366079	0.366079
param_C	0.001	0.001	0.001	0.001	0.001	0.001	0.01	0.01
param_gamma	0.001	0.01	0.1	1	10	100	0.001	0.01
param_kernel	rbf	rbf	rbf	rbf	rbf	rbf	rbf	rbf
params	{'C': 0.001, 'gamma': 0.001, 'kernel': 'rbf'}	{'C': 0.001, 'gamma': 0.01, 'kernel': 'rbf'}	{'C': 0.001, 'gamma': 0.1, 'kernel': 'rbf'}	{'C': 0.001, 'gamma': 1, 'kernel': 'rbf'}	{'C': 0.001, 'gamma': 10, 'kernel': 'rbf'}	{'C': 0.001, 'gamma': 100, 'kernel': 'rbf'}	{'C': 0.01, 'gamma': 0.001, 'kernel': 'rbf'}	{'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'}
rank_test_score	27	27	27	27	27	27	27	27
split0_test_score	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375
split0_train_score	0.363636	0.363636	0.363636	0.363636	0.363636	0.363636	0.363636	0.363636
split1_test_score	0.347826	0.347826	0.347826	0.347826	0.347826	0.347826	0.347826	0.347826

Nested Cross-Validation

We can go a step further, and instead of splitting the original data into training and test sets once, use multiple splits of cross-validation. This will result in what is called *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 parameters for each split in the outer loop). Then, for each outer split, the test set score using the best settings is reported.

18. We call cross_val_score with an instance of GridSearchCV as the model.

```
In [37]: scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                                  iris.data, iris.target, cv=5)
         print("Cross-validation scores:", scores)
         print("Mean cross-validation score:", scores.mean())
```

```
Cross-validation scores: [0.96666667 1.         0.9        0.96666667 1.        ]
Mean cross-validation score: 0.9666666666666668
```

19. It can be a bit tricky to understand what is happening in the single line given above, and it can be helpful to visualize it as for loops, as done in the following simplified implementation.

```
In [38]: def nested_cv(x, y, inner_cv, outer_cv, Classifier, parameter_grid):
             outer_scores = []
             # for each split of the data in the outer cross-validation
             # (split method returns indices)
             for training_samples, test_samples in outer_cv.split(x, y):
                 # find best parameter using inner cross-validation
                 best_params = {}
                 best_score = -np.inf
                 # iterate over parameters
                 for parameters in parameter_grid:
                     # accumulate score over inner splits
                     cv_scores = []
                     # iterate over inner cross-validation
                     for inner_train, inner_test in inner_cv.split(
                         x[training_samples], y[training_samples]):
                         # build classifier given parameters and training data
                         clf = Classifier(**parameters)
                         clf.fit(x[inner_train], y[inner_train])
                         # evaluate on inner test set
                         score = clf.score(x[inner_test], y[inner_test])
                         cv_scores.append(score)
                     # compute mean score over inner folds
                     mean_score = np.mean(cv_scores)
                     if mean_score > best_score:
                         # if better than so far, remember parameters
                         best_score = mean_score
                         best_params = parameters
                 # build classifier on best parameters using outer training set
                 clf = Classifier(**best_params)
                 clf.fit(x[training_samples], y[training_samples])
                 # evaluate
                 outer_scores.append(clf.score(x[test_samples], y[test_samples]))
             return np.array(outer_scores)
```

```
In [39]: from sklearn.model_selection import ParameterGrid, StratifiedKFold
         scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
                            StratifiedKFold(5), SVC, ParameterGrid(param_grid))
         print("Cross-validation scores: {}".format(scores))

         Cross-validation scores: [0.96666667 1.         0.96666667 0.96666667 1.
         ]
```

Referensi :
Raschka S, Mirjailili. 2017. Phython Machine Leraning 2nd edition. Packt Publishing