

Machine Learning with Python

Model Evaluation and Improvement

Contacts

Haesun Park

Email : haesunrpark@gmail.com

Meetup: <https://www.meetup.com/Hongdae-Machine-Learning-Study/>

Facebook : <https://facebook.com/haesunrpark>

Blog : <https://tensorflow.blog>

Book

파이썬 라이브러리를 활용한 머신러닝, 박해선.

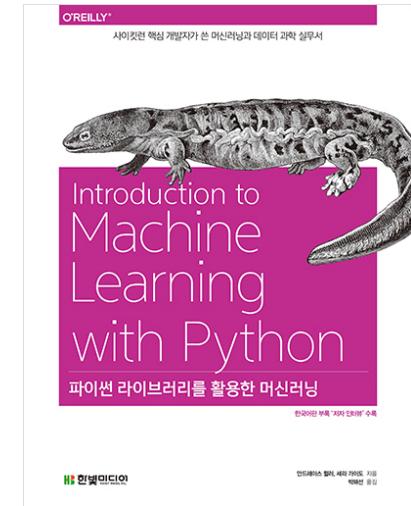
(Introduction to Machine Learning with Python, Andreas Muller & Sarah Guido의 번역서입니다.)

번역서의 1장과 2장은 블로그에서 무료로 읽을 수 있습니다.

원서에 대한 프리뷰를 온라인에서 볼 수 있습니다.

Github:

https://github.com/rickiepark/introduction_to_ml_with_python/



교차 검증

모델 평가

비지도 학습의 평가는 정성적이므로 지도 학습의 문제에 집중

train_test_split → fit → score

```
In [3]: from sklearn.datasets import make_blobs
        from sklearn.linear_model import LogisticRegression
        from sklearn.model_selection import train_test_split

        # 인위적인 데이터셋을 만듭니다
        X, y = make_blobs(random_state=0)
        # 데이터와 타깃 레이블을 훈련 세트와 테스트 세트로 나눕니다
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
        # 모델 객체를 만들고 훈련 세트로 학습시킵니다
        logreg = LogisticRegression().fit(X_train, y_train)
        # 모델을 테스트 세트로 평가합니다
        print("테스트 세트 점수: {:.2f}".format(logreg.score(X_test, y_test)))
```

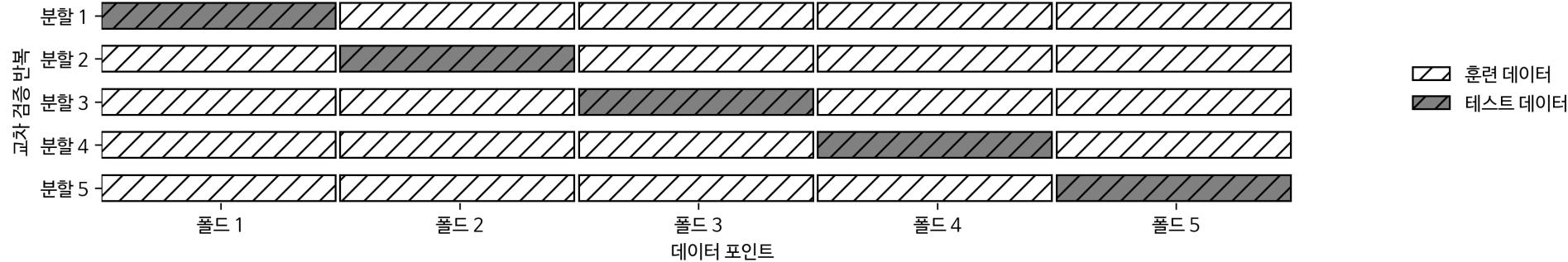
테스트 세트 점수: 0.88

교차 검증, R² 이외의 지표 등을 배웁니다.

교차 검증 cross-validation

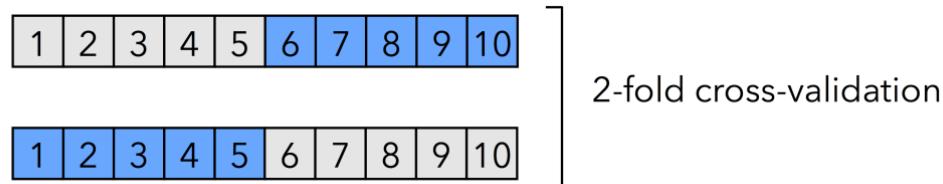
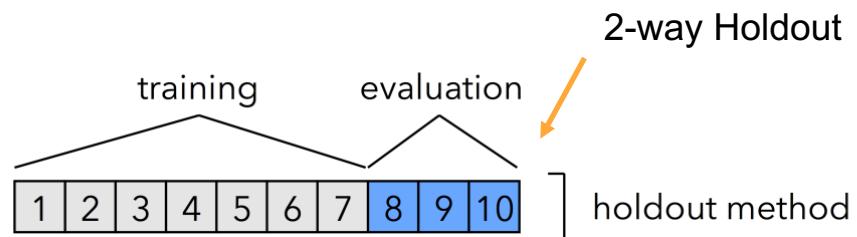
k-겹 교차 검증 k-fold cross-validation ($k=5$ or $k=10$)

1. 훈련 데이터를 k 개의 부분 집합(폴드)으로 나눕니다.
2. 첫 번째 폴드를 테스트 세트로하고 나머지 폴드로 모델을 훈련 시킵니다.
3. 테스트 폴드를 바꾸어 가며 모든 폴드가 사용될 때까지 반복합니다.

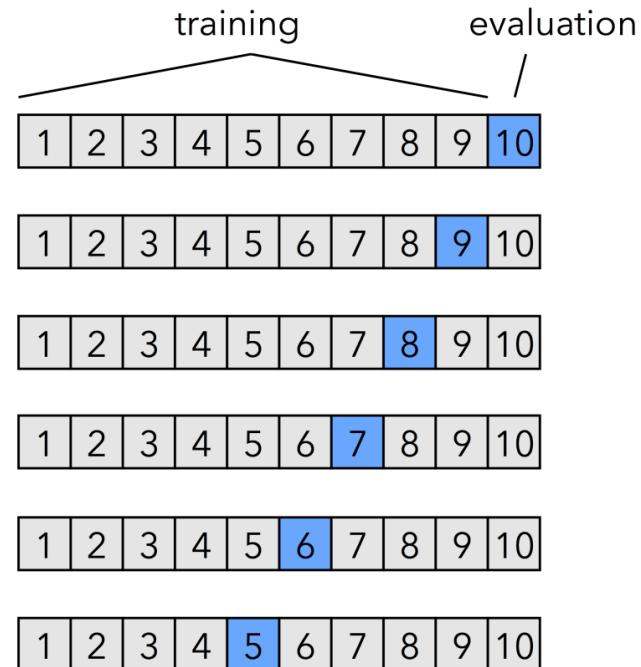


Others

홀드아웃 Holdout



LOOCV leave-one-out cross-validation
k=n CV



cross_val_score

```
In [5]: 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("교차 검증 점수: {}".format(scores))
```

교차 검증 점수: [0.961 0.922 0.958] ← 기본 폴드 수 : 3

모델, 데이터, 타깃

```
In [6]: scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
        print("교차 검증 점수: {}".format(scores))
```

교차 검증 점수: [1. 0.967 0.933 0.9 1.] ← 폴드 수 변경

```
In [7]: print("교차 검증 평균 점수: {:.2f}".format(scores.mean()))
```

교차 검증 평균 점수: 0.96

교차 검증의 장점

train_test_split는 무작위로 데이터를 나누기 때문에 우연히 평가가 좋게 혹은 나쁘게 나올 수 있음

→ 모든 폴드가 테스트 대상이 되기 때문에 공평함

train_test_split는 보통 70%~80%를 훈련에 사용함

→ 10겹 교차 검증은 90%를 훈련에 사용하기 때문에 정확한 평가를 얻음

모델이 훈련 데이터에 얼마나 민감한지 가늠할 수 있음

[1, 0.967, 0.933, 0.9, 1] → 90~100% Accuracy

[단점]: 데이터를 한 번 나누었을 때보다 k개의 모델을 만드므로 k배 느림

[주의]: cross_val_score는 교차 검증 동안 만든 모델을 반환하지 않습니다!

교차 검증의 함정

3-겹 교차 검증일 경우 테스트 세트에는 한 종류의 븎꽃만 포함됩니다.

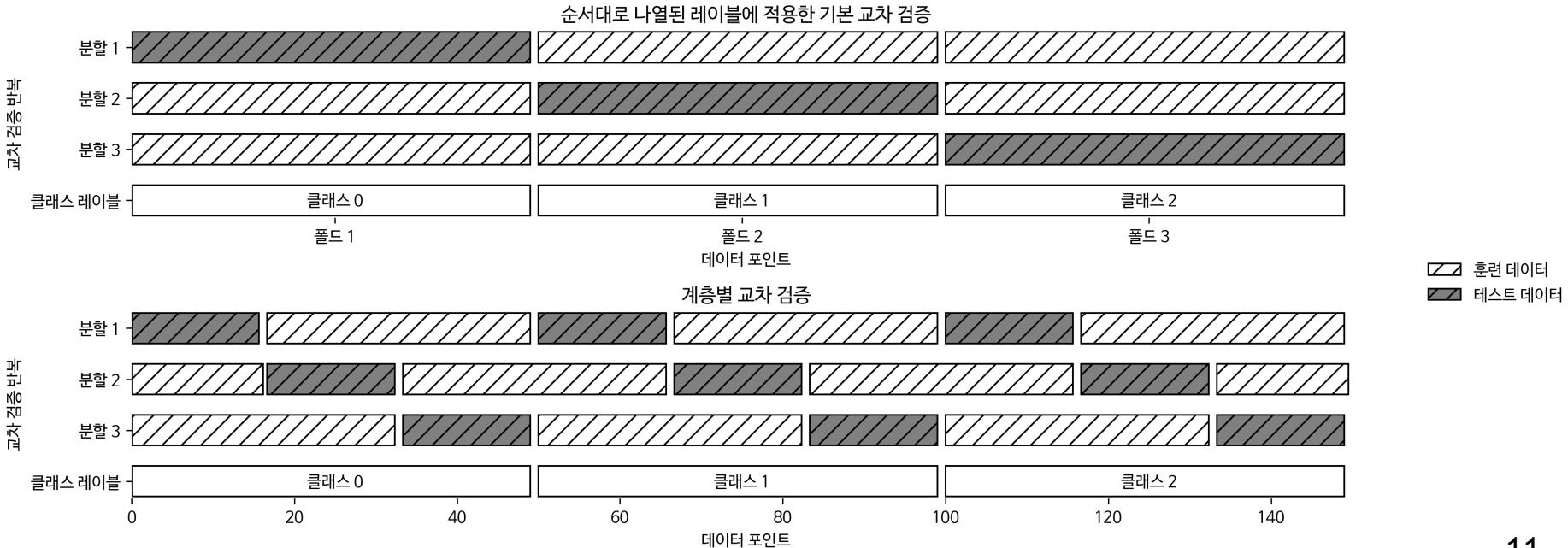
```
In [8]: from sklearn.datasets import load_iris  
iris = load_iris()  
print("Iris 레이블:\n{}".format(iris.target))
```

Iris 레이블:

KFold(n_splits=3)

계층별 k-겹 교차 검증 Stratified k-fold CV

분류 문제일 경우 `cross_val_score`는 기본적으로 `StratifiedKFold()`를 사용합니다.
회귀에서는 `KFold()`를 사용합니다.



교차 검증 분할기

분류에 KStratifiedFold() 대신 기본 KFold()를 적용하기 위해 cv 매개변수를 사용합니다.

```
In [10]: from sklearn.model_selection import KFold  
kfold = KFold(n_splits=5)
```

```
In [11]: print("교차 검증 점수:\n{}".format(  
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

교차 검증 점수:
[1. 0.933 0.433 0.967 0.433]

분류와 기본 KFold()

Iris 레이블:

```
In [12]: kfold = KFold(n_splits=3)
print("교차 검증 점수:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

교차 검증 점수:

[0. 0. 0.]

```
In [13]: kfolds = KFold(n_splits=3, shuffle=True, random_state=0)
print("교차 검증 점수:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfolds)))
```

교차 검증 점수:

[-0.9 0.96 0.96]

StratifiedKFold(n_splits=3): [0.961, 0.922, 0.958]

LOOCV\leave-one-out cross-validation

k=n 인 k-겹 교차 검증

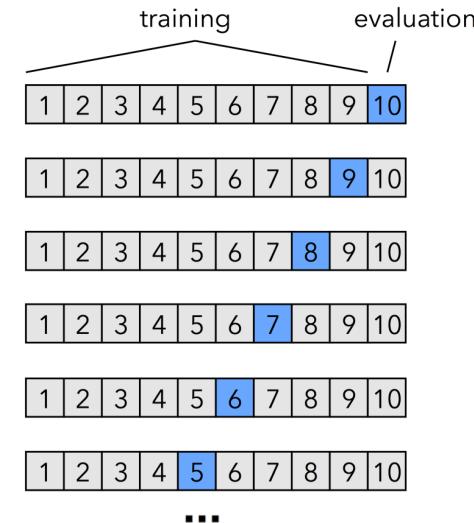
데이터셋이 클 때는 시간이 오래 걸리지만, 작은 데이터셋에서는 이따금 좋음.

In [14]:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("교차 검증 분할 횟수: ", len(scores))
print("평균 정확도: {:.2f}".format(scores.mean()))
```

교차 검증 분할 횟수: 150

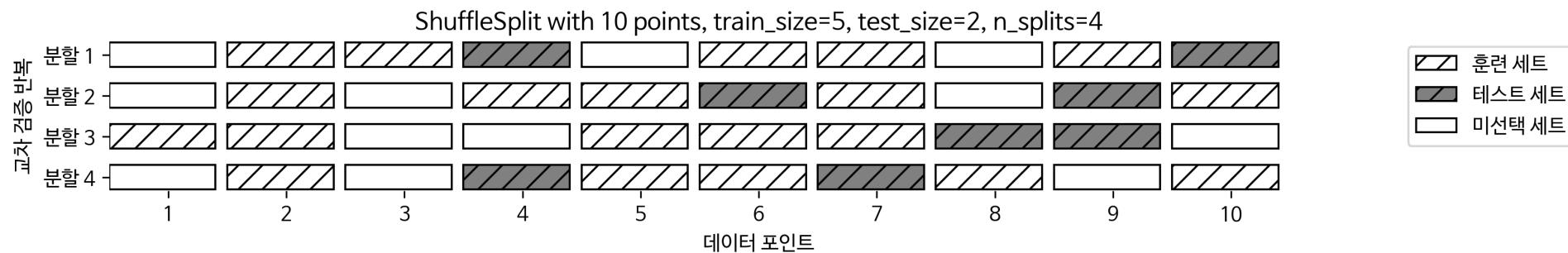
평균 정확도: 0.95



임의 분할 교차 검증 shuffle-split CV

훈련 세트 크기 : train_size, 테스트 세트 크기 : test_size, 폴드 수 : n_splits

ShuffleSplit(train_size=5, test_size=2, n_splits=4)



ShuffleSplit

test_size, train_size에 비율을 입력할 수 있음

test_size + train_size < 1 일 경우 부분 샘플링 subsampling이 됨

분류에 사용할 수 있는 StratifiedShuffleSplit도 있음

```
In [16]: 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("교차 검증 점수:\n{}".format(scores))
```

교차 검증 점수:

```
[ 0.867  0.907  0.973  0.973  0.893  0.96   0.987  0.893  0.933  0.973]
```

그룹별 교차 검증

타깃에 따라 폴드를 나누지 않고 입력 특성에 따라 폴드를 나누어야 할 경우

예를 들어 100장의 사진 데이터로 사람의 표정을 분류하는 문제에서 한 사람이 훈련 세트와 테스트 세트에 모두 나타날 경우 분류기의 성능을 정확히 측정할 수 없습니다.(의료 정보나 음성 인식 등에서도)

입력 데이터의 그룹 정보를 받을 수 있는 GroupKFold()를 사용합니다.

```
cross_val_score(model, X, y, groups, cv=GroupKFold(n_splits=3))
```

...

```
cv.split(X, y, groups)
```

...

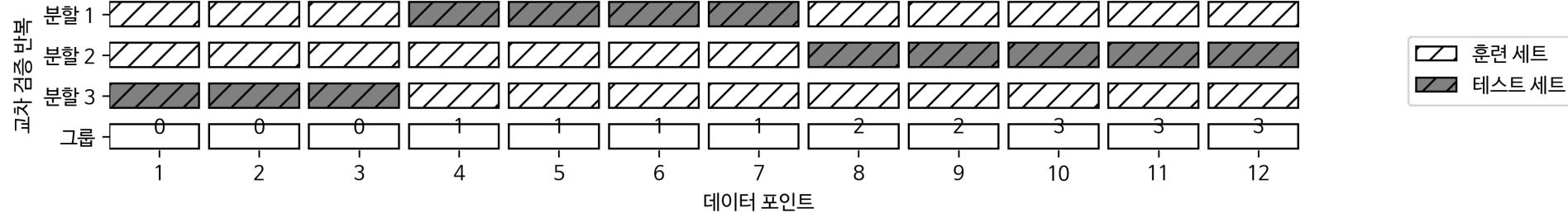
GroupKFold

```
In [17]: from sklearn.model_selection import GroupKFold  
# 인위적 데이터셋 생성  
X, y = make_blobs(n_samples=12, random_state=0)  
# 처음 세 개의 샘플은 같은 그룹에 속하고  
# 다음은 네 개의 샘플이 같습니다.  
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("교차 검증 점수:\n{}".format(scores))
```

교차 검증 점수:

[0.75 0.8 0.667]

GroupKFold



그리드 서치|Grid Search

매개변수 튜닝

매개변수를 튜닝하여 모델의 일반화 성능(교차 검증)을 증가시킵니다.

Scikit-Learn에는 GridSearchCV와 RandomizedSearchCV가 있습니다.

RBF 커널 SVM의 여러가지 매개변수 조합을 테스트합니다.

	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)

그리드 서치 구현

```
In [19]: # 간단한 그리드 서치 구현
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    random_state=0)
print("훈련 세트의 크기: {} 테스트 세트의 크기: {}".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]:
        # 매개변수의 각 조합에 대해 SVC를 훈련시킵니다
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # 테스트 세트로 SVC를 평가합니다
        score = svm.score(X_test, y_test)
        # 점수가 더 높으면 매개변수와 함께 기록합니다
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("최고 점수: {:.2f}".format(best_score))
print("최적 파라미터: {}".format(best_parameters))
```

gamma와 C에 대한
반복 루프

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
 for C in [0.001, 0.01, 0.1, 1, 10, 100]:

매개변수의 각 조합에 대해 SVC를 훈련시킵니다

svm = SVC(gamma=gamma, C=C)

svm.fit(X_train, y_train)

테스트 세트로 SVC를 평가합니다

score = svm.score(X_test, y_test)

점수가 더 높으면 매개변수와 함께 기록합니다

if score > best_score:

best_score = score

best_parameters = {'C': C, 'gamma': gamma}

36개의 모델이 만들어짐

가장 좋은 테스트 점수를 기록

훈련 세트의 크기: 112 테스트 세트의 크기: 38
최고 점수: 0.97
최적 파라미터: {'gamma': 0.001, 'C': 100}

검증 세트 validation set

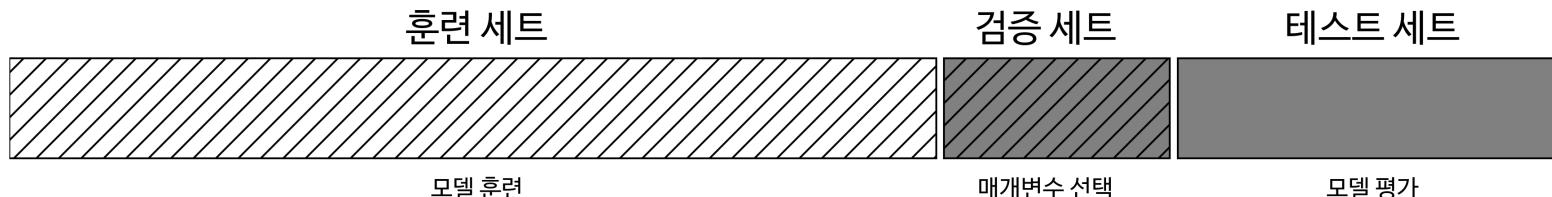
최고 점수: 0.97

최적 파라미터: {'gamma': 0.001, 'C': 100}

테스트 세트로 여러가지 매개변수 조합에 대해 평가했다면 이 모델이 새로운 데이터도 동일한 성능을 낸다고 생각하는 것은 매우 낙관적인 추정입니다.

최종 평가를 위해서는 독립된 데이터 세트가 필요합니다.

검증 세트 validation set 혹은 개발 세트 dev set



검증 세트 구현

```
In [21]: from sklearn.svm import SVC
# 데이터를 훈련+검증 세트 그리고 테스트 세트로 분할
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# 훈련+검증 세트를 훈련 세트와 검증 세트로 분할
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("훈련 세트의 크기: {} 검증 세트의 크기: {} 테스트 세트의 크기: "
      "{}\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]:
        # 매개변수의 각 조합에 대해 SVC를 훈련시킵니다
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # 검증 세트로 svc를 평가합니다
        score = svm.score(X_valid, y_valid)
        # 점수가 더 높으면 매개변수와 함께 기록합니다
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

훈련 데이터 → 훈련검증세트, 테스트 세트
훈련검증세트 → 훈련 세트, 검증 세트

두 구현의 결과 비교

```
훈련 세트의 크기: 112 테스트 세트의 크기: 38
최고 점수: 0.97
최적 파라미터: {'gamma': 0.001, 'C': 100}

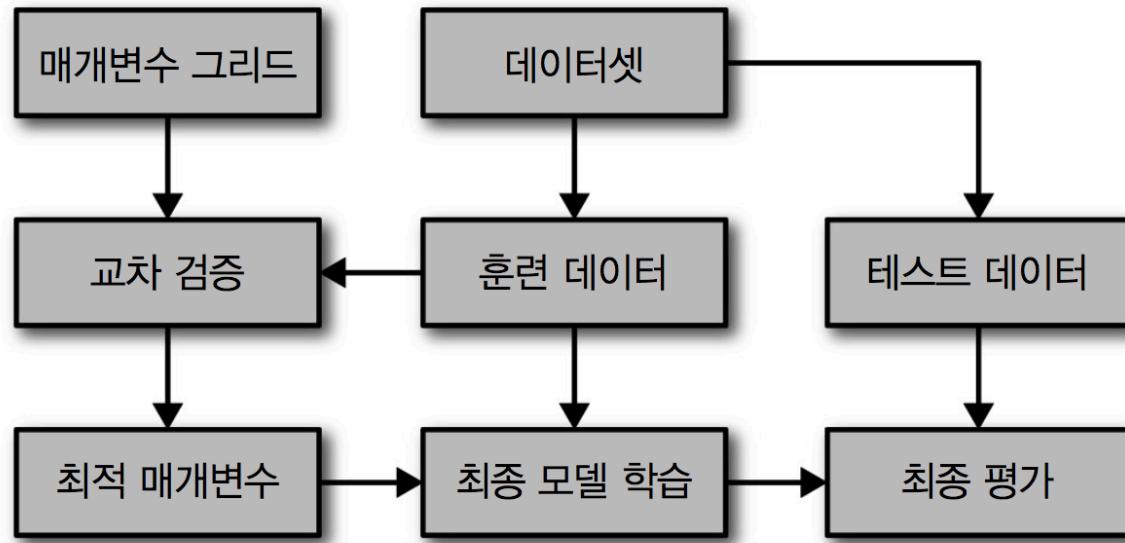
훈련 세트의 크기: 84 검증 세트의 크기: 28 테스트 세트의 크기: 38
검증 세트에서 최고 점수: 0.96
최적 파라미터: {'gamma': 0.001, 'C': 10}
최적 파라미터에서 테스트 세트 점수: 0.92
```

모델 시각화, 탐색적 분석, 모델 선택에 테스트 세트를 사용하면 안됩니다.

검증 세트 + cross_val_score

```
In [22]: for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # 매개변수의 각 조합에 대해 SVC를 훈련시킵니다
        svm = SVC(gamma=gamma, C=C)
        # 교차 검증을 적용합니다
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # 교차 검증 정확도의 평균을 계산합니다
        score = np.mean(scores)
        # 점수가 더 높으면 매개변수와 함께 기록합니다
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
        # 훈련 세트와 검증 세트를 합쳐 모델을 다시 만듭니다
        svm = SVC(**best_parameters)
        svm.fit(X_trainval, y_trainval)
```

매개변수 탐색의 전체 과정



GridSearchCV

교차 검증을 사용한 그리드 서치

검색하려는 매개변수를 키로 하는 딕셔너리를 사용함

```
In [25]: param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],  
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}  
print("매개변수 그리드:\n{}".format(param_grid))
```

매개변수 그리드:

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

```
In [26]: from sklearn.model_selection import GridSearchCV  
from sklearn.svm import SVC  
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

모델, 매개변수그리드, 폴드수
회귀:KFold, 분류:StratifiedKFold

```
In [27]: X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,  
                                                       random_state=0)
```

```
In [28]: grid_search.fit(X_train, y_train)
```

fit, predict, score,
predict_proba, decision_function 제공

그리드서치 모델 테스트

GridSearchCV에 사용하지 않은 테스트 세트로 평가

```
In [29]: print("테스트 세트 점수: {:.2f}".format(grid_search.score(X_test, y_test)))  
테스트 세트 점수: 0.97
```

```
In [30]: print("최적 매개변수: {}".format(grid_search.best_params_))  
print("최고 교차 검증 점수: {:.2f}".format(grid_search.best_score_))  
최적 매개변수: {'gamma': 0.01, 'C': 100}  
최고 교차 검증 점수: 0.97
```

훈련 데이터로 교차 검증한 점수

```
In [31]: print("최고 성능 모델:\n{}".format(grid_search.best_estimator_))  
최고 성능 모델:  
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)
```

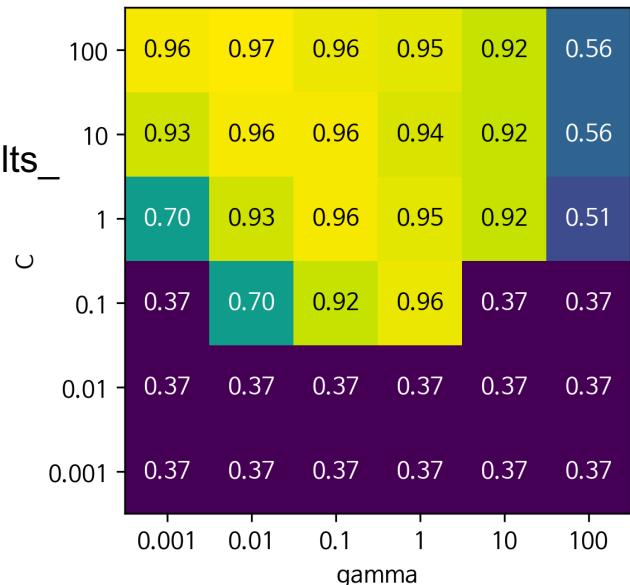
결과 분석

간격을 넓게 하여 그리드 서치를 시작하고 결과를 분석해 검색 영역을 조정합니다.

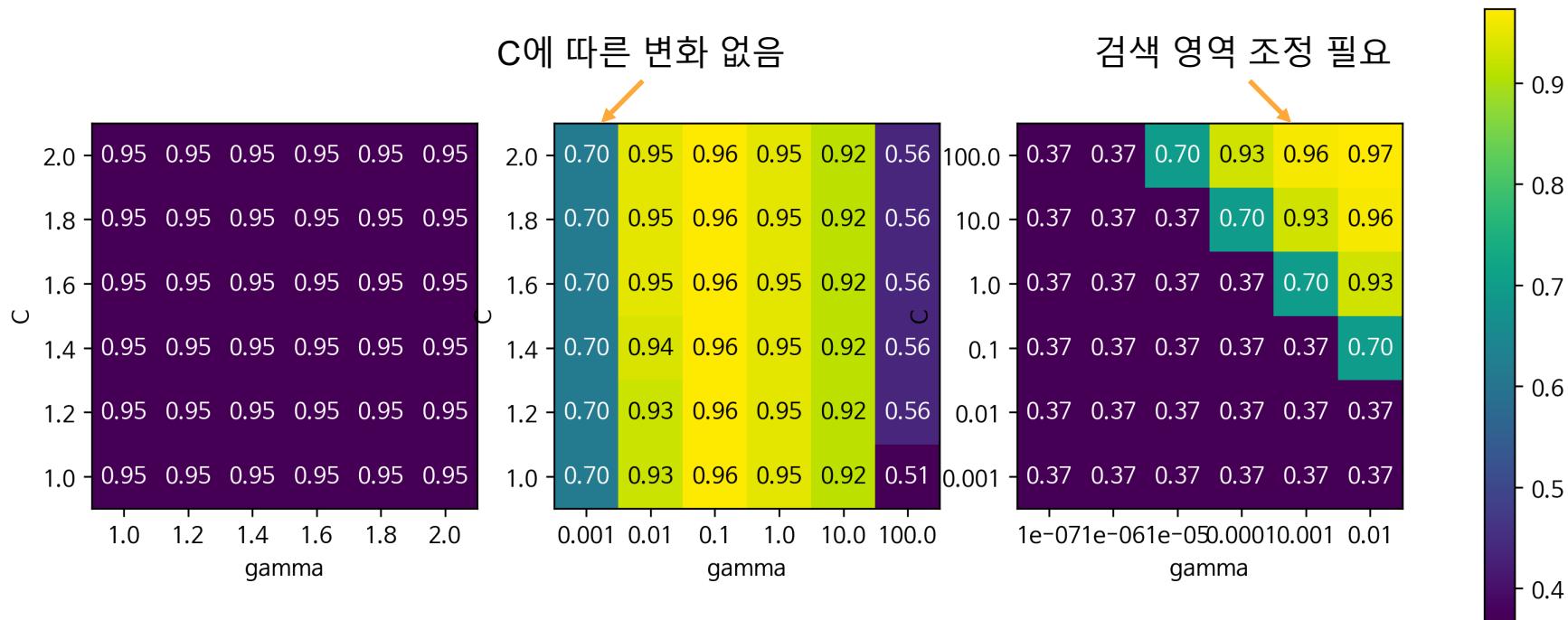
RandomizedSearchCV는 매개변수 조합이 매우 많거나 C와 같이 연속형 값을 조정할 때 많이 사용됩니다.

out[32]:					
	param_C	param_gamma	params	mean_test_score	
0	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	0.366	
1	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	0.366	
2	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	0.366	
3	0.001	1	{'C': 0.001, 'gamma': 1}	0.366	
4	0.001	10	{'C': 0.001, 'gamma': 10}	0.366	
	rank_test_score	split0_test_score	split1_test_score	split2_test_score	
0	22	0.375	0.347	0.363	
1	22	0.375	0.347	0.363	
2	22	0.375	0.347	0.363	
3	22	0.375	0.347	0.363	
4	22	0.375	0.347	0.363	
	split3_test_score	split4_test_score	std_test_score		
0	0.363	0.380	0.011		
1	0.363	0.380	0.011		
2	0.363	0.380	0.011		
3	0.363	0.380	0.011		
4	0.363	0.380	0.011		

grid_search.cv_results_



부적절한 그리드



비대칭 그리드 서치

다른 매개변수에 의해 선택적으로 그리드 서치를 수행할 수 있습니다.

매개변수 그리드를 딕셔너리의 리스트로 만듭니다.

```
In [35]: 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("그리드 목록:\n{}".format(param_grid))
```

linear 커널일 경우에는 gamma 매개변수를 지정하지 않습니다.

비대칭 그리드 서치 결과

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear
params	{'C': 0.001, 'kernel': 'rbf', 'gamma': 0.001}	{'C': 0.001, 'kernel': 'rbf', 'gamma': 0.01}	{'C': 0.001, 'kernel': 'rbf', 'gamma': 0.1}	{'C': 0.001, 'kernel': 'rbf', 'gamma': 1}	...	{'C': 0.1, 'kernel': 'rbf', 'gamma': 0.01}	{'C': 1, 'kernel': 'rbf', 'gamma': 0.01}	{'C': 10, 'kernel': 'rbf', 'gamma': 0.01}	{'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
mean_test_score	0.37	0.37	0.37	0.37					
rank_test_score	27	27	27	27					
split0_test_score	0.38	0.38	0.38	0.38					
split1_test_score	0.35	0.35	0.35	0.35					
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

```
In [36]: grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("최적 파라미터: {}".format(grid_search.best_params_))
print("최고 교차 검증 점수: {:.2f}".format(grid_search.best_score_))
```

최적 파라미터: {'kernel': 'rbf', 'gamma': 0.01, 'C': 100}
최고 교차 검증 점수: 0.97

ShuffleSplit()

중첩 교차 검증nested CV

훈련 데이터(cross_val_score) → 훈련 세트(SVC)와 테스트 세트 → 평가

훈련 데이터 → 훈련 세트(GridSearchCV) → 튜닝 → 모델 → 테스트 세트 → 평가

훈련 데이터(cross_val_score) → 훈련 세트(GridSearchCV)와 테스트 세트 → 평가

```
In [38]: param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],  
                    'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}  
scores = cross_val_score(GridSearchCV(SVC()), param_grid, cv=5)  
        iris.data, iris.target, cv=5)  
print("교차 검증 점수: ", scores)  
print("교차 검증 평균 점수: ", scores.mean())  
print(param_grid)
```

교차 검증 점수: [0.967 1. 0.967 0.967 1.]

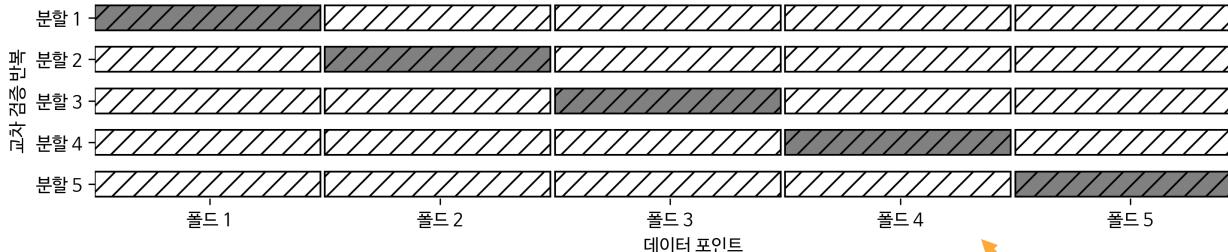
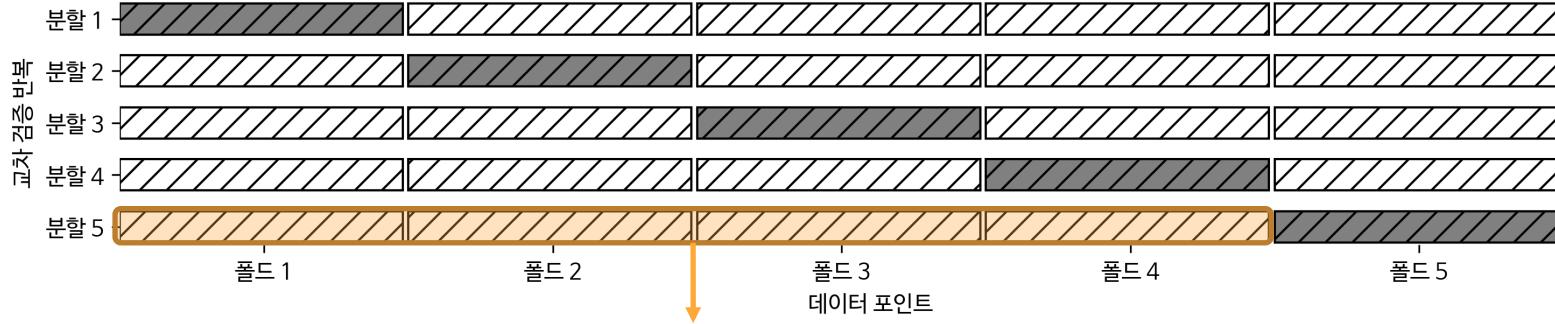
교차 검증 평균 점수: 0.98

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

cross_val_score + GridSearchCV

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

cross_val_score



$$36 \times 5 \times 5 = 900$$

GridSearchCV

병렬화

그리드 서치는 연산에 비용이 많이 들지만 매개변수 조합마다 쉽게 병렬화 가능

GridSearchCV와 cross_val_score의 n_jobs에 사용할 CPU 코어수를 지정할 수 있습니다(기본 1, 최대 -1).

하지만 GridSearchCV와 RandomForestClassifier 같은 모델이 동시에 n_jobs 옵션을 사용할 수 없습니다(이중 fork로 데몬 프로세스가 되는 것을 방지).

마찬가지로 cross_val_score와 GridSearchCV도 동시에 n_jobs 옵션을 사용할 수 없습니다.

평가 지표와 측정

평가 지표

일반적으로 회귀: R^2 , 분류: 정확도를 사용합니다.

비즈니스 임팩트를 고려하여 목표를 설정합니다(교통사고 회수, 입원 환자수 등)

비즈니스 지표를 얻으려면 운영 시스템에 적용해야 알 수 있는 경우가 많으므로 대리할 수 있는 평가 지표를 사용합니다(보행자 이미지를 사용한 자율 주행 테스트)

시스템의 목표에 따라 방문 고객이 10% 늘어나는 모델을 찾을 수 있지만 매출은 15% 줄어들 수 있습니다(경우에 따라 고객이 아니라 매출을 비즈니스 지표로 삼아야 합니다).

이진 분류

에러의 종류

양성 클래스(관심 대상)와 음성 클래스로 나뉩니다.

암진단: 암([양성 테스트](#), [악성](#)), 정상([음성 테스트](#))

[양성 테스트](#)(암)를 양성 클래스, [음성 테스트](#)(정상)을 음성 클래스라 할 때
정상을 양성 클래스로 분류(거짓 양성 false positive): 추가적인 검사 동반
암을 음성 클래스로 분류(거짓 음성 false negative): 건강을 해침

거짓 양성을 타입 I 에러, 거짓 음성을 타입 II 에러라고도 합니다.

거짓 양성과 거짓 음성이 비슷한 경우는 드물며 오류를 비용으로 환산하여
비즈니스적인 판단을 해야 합니다.

불균형 데이터셋

예) 광고 노출 수와 클릭 수: 99 vs 1 (현실에서는 0.1% 미만입니다)

무조건 ‘클릭 아님’으로 예측하면 99%의 정확도를 가진 분류기가 됩니다.

불균형한 데이터셋에서는 정확도만으로는 모델이 진짜 좋은지 모릅니다.

```
In [41]: 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)
```

숫자 9는 1, 나머지는 0인 타깃값

더미 vs 결정 트리

```
In [42]: 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("예측된 레이블의 고유값: {}".format(np.unique(pred_most_frequent)))  
print("테스트 점수: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

예측된 레이블의 고유값: [False]

테스트 점수: 0.90

무조건 9아님(0)으로 예측을 만듦

```
In [43]: from sklearn.tree import DecisionTreeClassifier  
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)  
pred_tree = tree.predict(X_test)  
print("테스트 점수: {:.2f}".format(tree.score(X_test, y_test)))
```

테스트 점수: 0.92

더미 vs 로지스틱 회귀

strategy 기본값: stratified
클래스 비율(10%:1, 90%:0) 대로 랜덤하게 예측

```
In [44]: from sklearn.linear_model import LogisticRegression

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

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

dummy 점수: 0.80
logreg 점수: 0.98

오차 행렬 confusion matrix

```
In [45]: from sklearn.metrics import confusion_matrix
```

```
confusion = confusion_matrix(y_test, pred_logreg)  
print("오차 행렬:\n{}".format(confusion))
```

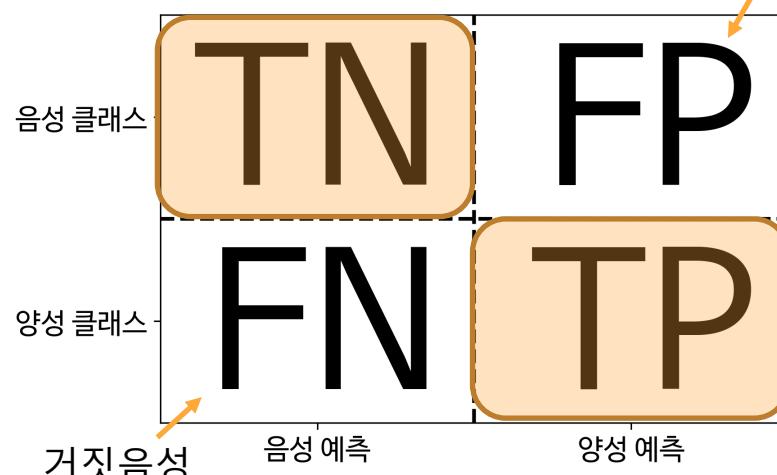
타깃값, 예측 결과

오차 행렬:

```
[[401  2]  
 [ 8 39]]
```

'9 아님' 정답	401	2
'9' 정답	8	39

'9 아님' 예측 '9' 예측



더미 분류기의 오차행렬

```
In [48]: print("빈도 기반 더미 모델:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\n무작위 더미 모델:")
print(confusion_matrix(y_test, pred_dummy))
print("\n결정 트리:")
print(confusion_matrix(y_test, pred_tree))
print("\n로지스틱 회귀")
print(confusion_matrix(y_test, pred_logreg))
```

비슷한 FP, FN

무조건 '9아님'으로 예측(TN,FN)

빈도 기반 더미 모델:
[[403 0]
 [47 0]]

무작위 더미 모델:
[[360 43]
 [40 7]]

결정 트리:
[[390 13]
 [24 23]]

로지스틱 회귀
[[401 2]
 [8 39]]

정확도 accuracy, 정밀도 precision, 재현율 recall

$$\text{정확도} = \frac{TP + TN}{TP + TN + FP + FN}$$

TN	FP
FN	TP

양성으로 예측된 것 중

진짜 양성의 비율(양성 예측도)

ex) 신약의 효과 측정

$$\text{정밀도} = \frac{TP}{TP + FP}$$

TN	FP
FN	TP

진짜 양성 중 양성으로 예측된 비율

(민감도, 적중률, 진짜양성비율-TPR)

ex) 암 진단

$$\text{재현율} = \frac{TP}{TP + FN}$$

TN	FP
FN	TP

정밀도 <> 재현율

모두 양성으로 예측하면
FP가 커져
정밀도가 낮아지고

FN은 0이 되어
재현율은 완벽해집니다.

$$\text{정밀도} = \frac{TP}{TP + FP}$$

양성 하나만을 제대로 예측하면
FP는 0이어서
정밀도는 1이 되지만

$$\text{재현율} = \frac{TP}{TP + FN}$$

FN이 커져
재현율은 낮아집니다.

f-점수 f-score

f_1 -점수라고도 합니다. 정밀도와 재현율의 조화평균으로 불균형 데이터셋에 좋지만 설명하기 어렵습니다.

$$F = 2 \times \frac{\text{정밀도} \times \text{재현율}}{\text{정밀도} + \text{재현율}}$$

f-점수의 분모가 0이 됨

빈도 기반 더미 모델:
[[403 0]
 [47 0]]

```
In [49]: from sklearn.metrics import f1_score
print("빈도 기반 더미 모델의 f1 score: {:.2f}".format(
    f1_score(y_test, pred_most_frequent)))
print("무작위 더미 모델의 f1 score: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("트리 모델의 f1 score: {:.2f}".format(f1_score(y_test, pred_tree)))
print("로지스틱 회귀 모델의 f1 score: {:.2f}".format(
    f1_score(y_test, pred_logreg)))
```

빈도 기반 더미 모델의 f1 score: 0.00

정확도

무작위 더미 모델의 f1 score: 0.14

0.80

트리 모델의 f1 score: 0.55

0.92

로지스틱 회귀 모델의 f1 score: 0.89

classification_report + pred_most_frequent

```
In [50]: from sklearn.metrics import classification_report  
print(classification_report(y_test, pred_most_frequent,  
                           target_names=["9 아님", "9"]))
```

양성 클래스를 '9아님'
으로 바꾸었을 때 점수

	precision	recall	f1-score	support
9 아님	0.90	1.00	0.94	403
9	0.00	0.00	0.00	47
avg / total	0.80	0.90	0.85	450

pred_most_frequent는
모든 데이터를 '9아님'으로 예측하기 때문에
양성 클래스를 맞춘 것이 없음

진짜 타깃값

classification_report + pred_dummy, pred_logreg

```
In [51]: print(classification_report(y_test, pred_dummy,  
target_names=["9 아님", "9"]))
```

어떤 클래스를
양성 클래스로
선택하는지가 중요

	precision	recall	f1-score	support
9 아님	0.90	0.89	0.90	403
9	0.14	0.15	0.14	47
avg / total	0.82	0.82	0.82	450

```
In [52]: print(classification_report(y_test, pred_logreg,  
target_names=["9 아님", "9"]))
```

	precision	recall	f1-score	support
9 아님	0.98	1.00	0.99	403
9	0.95	0.83	0.89	47
avg / total	0.98	0.98	0.98	450

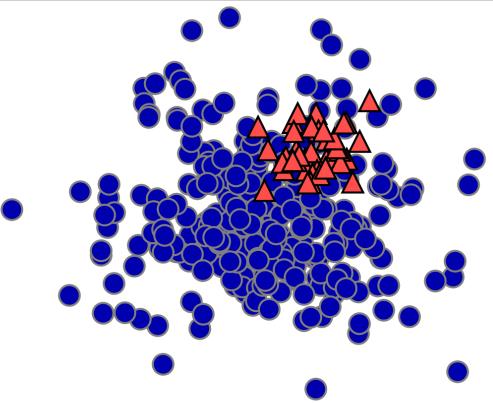
불확실성

`decision_function()`은 선형 판별식이 0보다 크면 양성으로,

`predict_proba()`는 선형 판별식에 시그모이드 함수를 적용하여 0.5보다 크면 양성으로 판단합니다.

이 함수들의 반환값이 클수록 예측에 대한 확신이 높다고 간주할 수 있습니다.

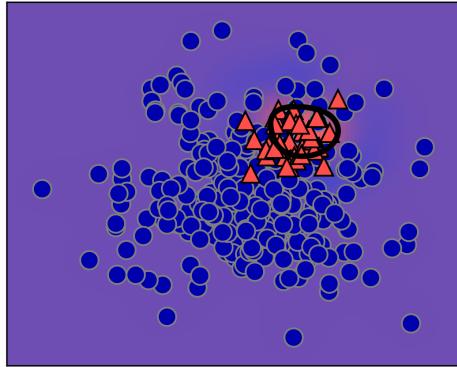
훈련 데이터



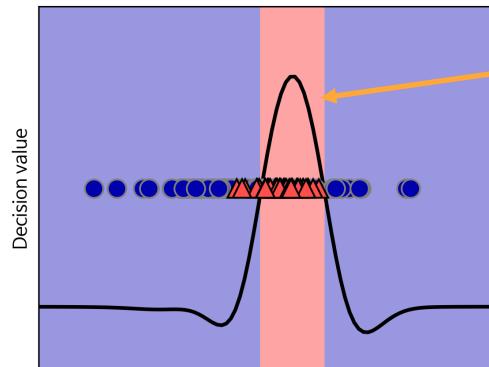
```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

decision_function() > 0

임계값 0일 때



임계값 0일 때 단면도



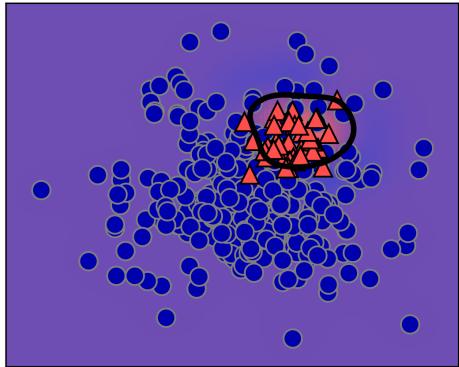
```
In [55]: print(classification_report(y_test, svc.predict(X_test)))
```

	precision	recall	f1-score	support
악성	0	0.97	0.89	104
	1	0.35	0.67	9
avg / total	0.92	0.88	0.89	113

악성이 양성 클래스일 경우
모든 악성을 잡아내기 위해 재현율을 높여야 합니다.

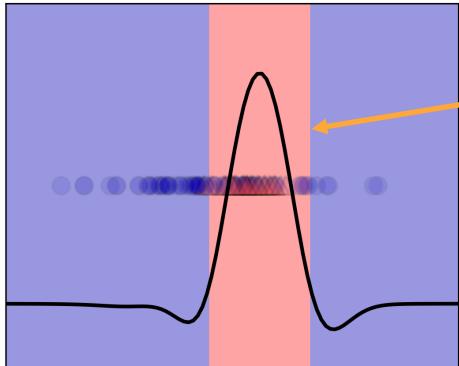
decision_function() > -0.8

임계값 -0.8일 때



임계값 -0.8일 때 단면도

Decision value



model.predict_proba(X_test) > 0.45

```
In [56]: y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```



```
In [57]: print(classification_report(y_test, y_pred_lower_threshold))
```

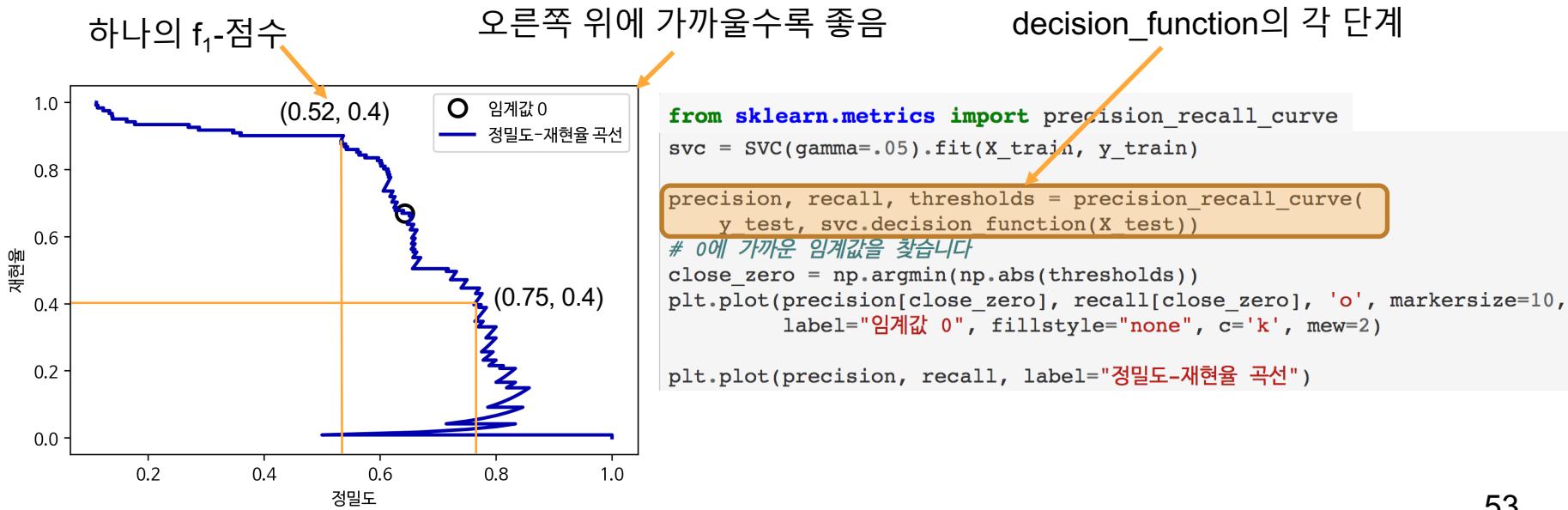
	precision	recall	f1-score	support
악성	0	1.00	0.82	104
	1	0.32	1.00	9
avg / total	0.95	0.83	0.87	113

실전에서는 테스트 세트를 사용해서는 안됩니다!

정밀도-재현율 곡선

정밀도와 재현율의 트레이드 오프를 조정

decision_function과 predict_proba 함수를 이용하여 운영 포인트 operating point 결정



RandomForestClassifier vs SVC

재현율이나 정밀도가 극단일 경우
랜덤 포레스트가 더 낫습니다.

In [60]:

```
from sklearn.ensemble import RandomForestClassifier

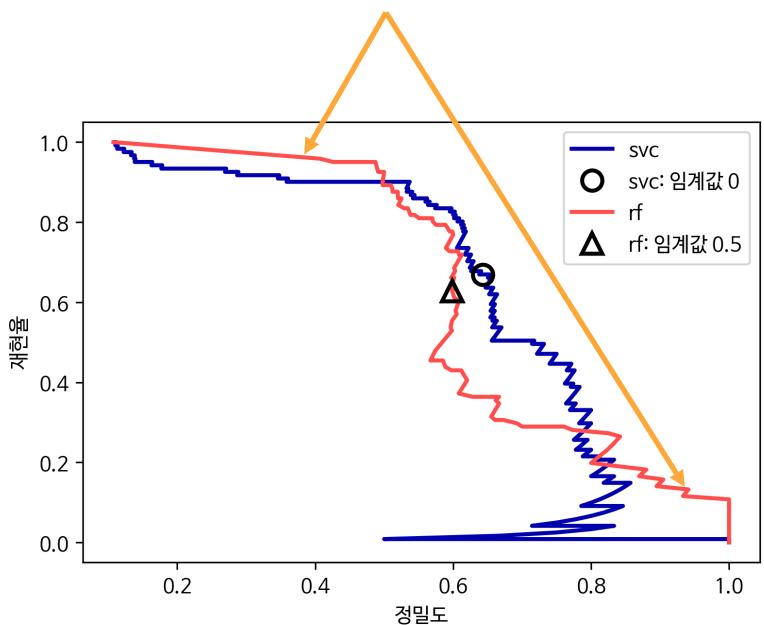
rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

# RandomForestClassifier는 decision function 대신 predict_proba를 제공합니다.
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="svc: 임계값 0", fillstyle="none", c='k', mew=2)

plt.plot(precision_rf, recall_rf, label="rf")



predict_proba의 결과중 양성 클래스(배열 인덱스 1)
에 대한 확률 값을 전달합니다.

평균 정밀도 average precision

정밀도-재현율 곡선의 아랫부분 면적을 나타냅니다.

```
In [61]: print("랜덤 포레스트의 f1_score: {:.3f}".format(
    f1_score(y_test, rf.predict(X_test))))
print("svc의 f1_score: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

랜덤 포레스트의 f1_score: 0.610
svc의 f1_score: 0.656

```
In [62]: 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("랜덤 포레스트의 평균 정밀도: {:.3f}".format(ap_rf))
print("svc의 평균 정밀도: {:.3f}".format(ap_svc))
```

랜덤 포레스트의 평균 정밀도: 0.660
svc의 평균 정밀도: 0.666

평균 정밀도는 거의 비슷한 수준

ROC와 AUC

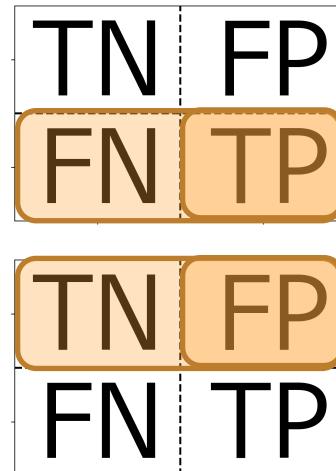
수신기 조작 특성(Receiver operating characteristics)

진짜 양성 비율(TPR, 재현율)에 대한 거짓 양성 비율(FPR)의 그래프

`sklearn.metrics.roc_curve`

$$\text{재현율} = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

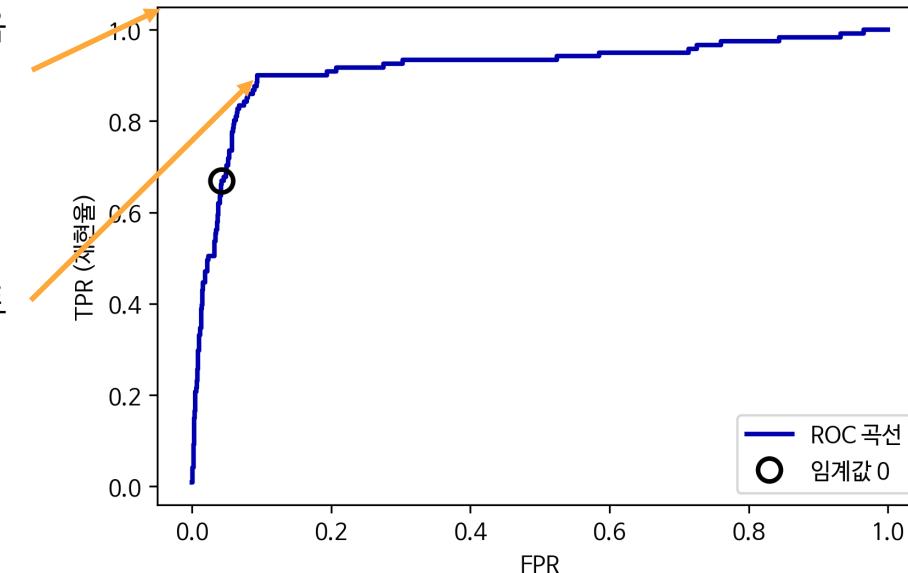


roc_curve + SVC

```
In [63]: from sklearn.metrics import roc_curve  
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))  
  
plt.plot(fpr, tpr, label="ROC 곡선")
```

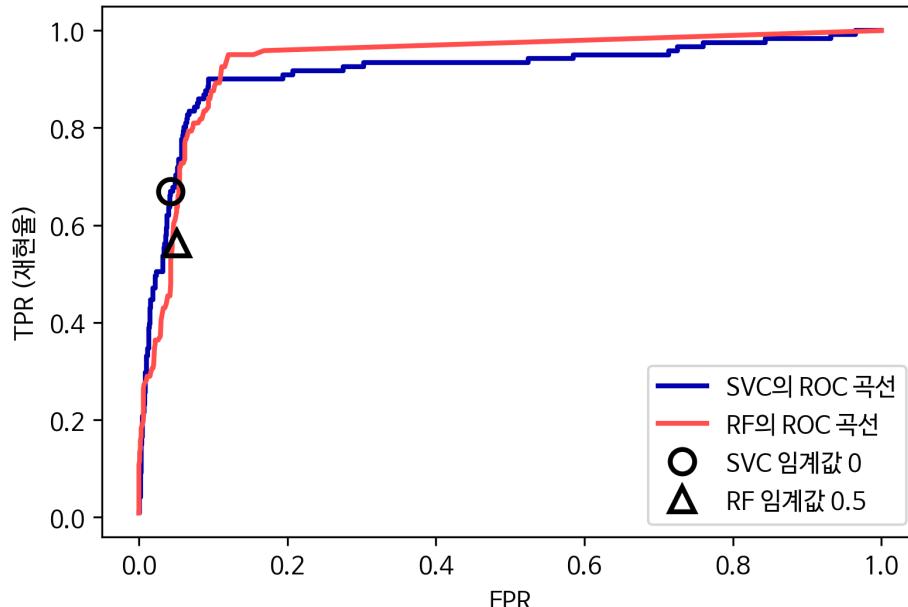
왼쪽 위에 가까울수록 좋음
거짓 양성(FP)은 적고
진짜 양성(TP)은 높게

적절한 운영 포인트



RandomForestClassifier vs SVC

```
In [64]: from sklearn.metrics import roc_curve  
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])  
  
plt.plot(fpr, tpr, label="SVC의 ROC 곡선")  
plt.plot(fpr_rf, tpr_rf, label="RF의 ROC 곡선")
```



AUC area under the curve

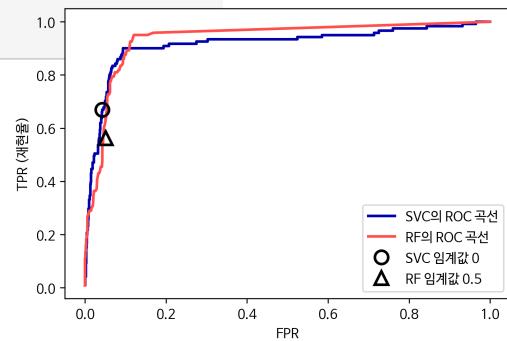
ROC 곡선 아래의 면적(like 평균 정밀도)

sklearn.metrics.roc_auc_score

타깃 레이블, 예측 확률

```
In [65]: 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("랜덤 포레스트의 AUC: {:.3f}".format(rf_auc))
print("SVC의 AUC: {:.3f}".format(svc_auc))
```

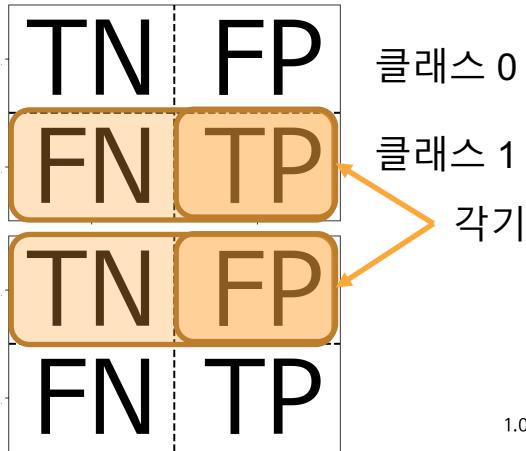
랜덤 포레스트의 AUC: 0.937
SVC의 AUC: 0.916



AUC 특징

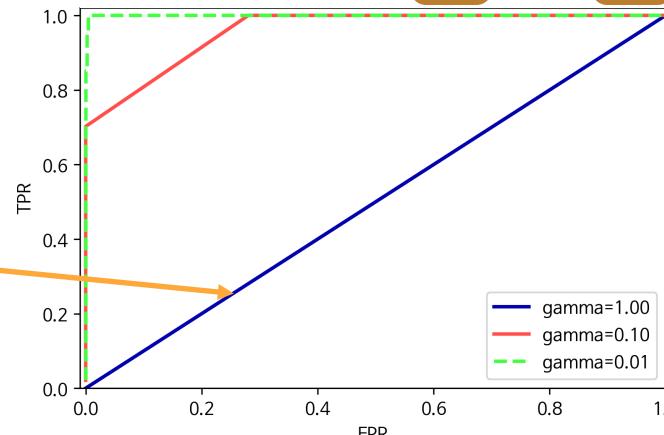
불균형한 데이터셋에도
안정적임

$$\text{재현율} = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{FP + TN}$$



불균형 데이터셋에서도 무작위로 선택하면
TPR, FPR이 비슷해지므로 ROC 곡선이
y=x 가 되어 AUC는 0.5에 가까워짐

gamma = 1.00	정확도 = 0.90	AUC = 0.50
gamma = 0.10	정확도 = 0.90	AUC = 0.96
gamma = 0.01	정확도 = 0.90	AUC = 1.00



다중 분류

다중 분류의 평가 지표

이진 분류 평가 지표에서 유도한 것으로 모든 클래스에 대해 평균을 냅니다.

다중 분류에서도 불균형한 데이터셋에서는 정확도가 좋은 지표가 아닙니다.

손글씨 숫자 10개에 대한 오차 행렬

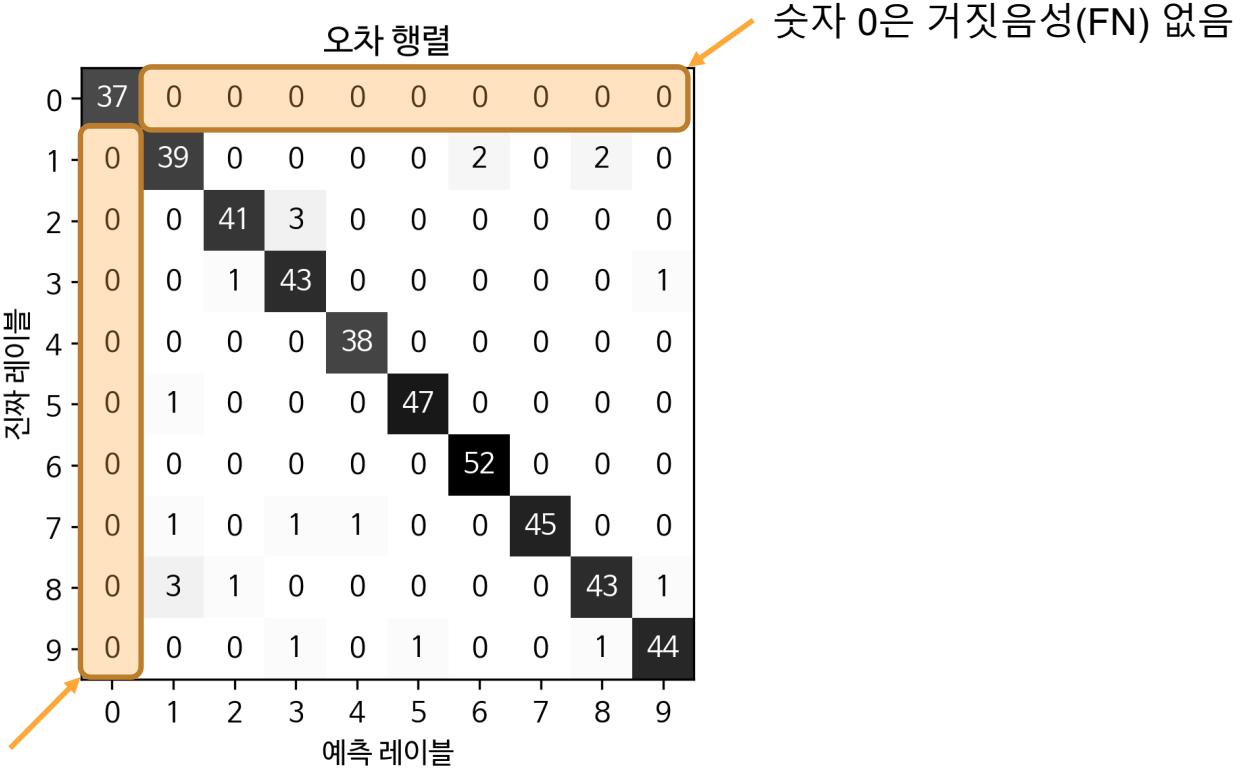
```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("정확도: {:.3f}".format(accuracy_score(y_test, pred)))
print("오차 행렬:\n{}".format(confusion_matrix(y_test, pred)))
```

정확도: 0.953

오차 행렬:

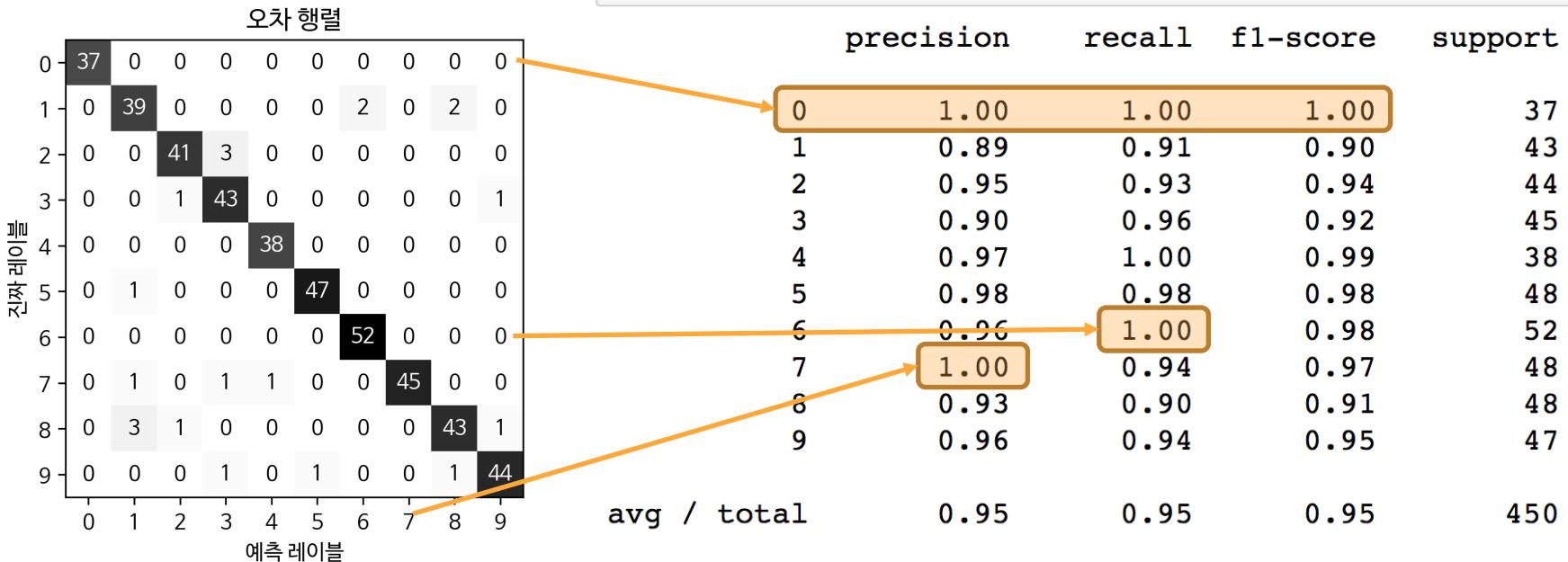
```
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  0  0  45  0  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

다중 분류의 오차 행렬



다중 분류 리포트

```
In [69]: print(classification_report(y_test, pred))
```



다중 분류 f₁-점수

다중 분류용 f₁-점수는 한 클래스를 양성 클래스로 두고 나머지를 음성으로 간주하여 각 클래스마다 f₁-점수를 계산합니다.

`f1_score(y_test, pred, average="...")`

- “macro”: 가중치 없이 평균을 냅니다.
- “weighted”: 클래스별 샘플 수로 가중치를 두어 평균을 냅니다.(분류 리포트)
- “micro”: 모든 클래스의 FP, FN, TP를 합하여 f₁-점수를 계산합니다.
- “binary”: 이진 분류에 해당, 기본값.

```
In [70]: print("micro 평균 f1 점수: {:.3f}".format(f1_score(y_test, pred, average="micro")))
      print("macro 평균 f1 점수: {:.3f}".format(f1_score(y_test, pred, average="macro")))
```

```
micro 평균 f1 점수: 0.953
macro 평균 f1 점수: 0.954
```

회귀의 평가 지표

R^2

회귀는 연속된 값을 예측하므로 R^2 만으로 충분합니다.

가끔 평균 제곱 에러나 평균 절댓값 에러를 사용하기도 합니다.

모델 선택과 평가 지표

cross_val_score

scoring 매개변수에 원하는 평가 지표를 지정할 수 있습니다. 기본값은 모델의 score() 메소드입니다.

```
In [71]: # 분류의 기본 평가 지표는 정확도 입니다
print("기본 평가 지표: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
# scoring="accuracy"의 결과는 같습니다.
explicit_accuracy = cross_val_score(SVC(), digits.data, digits.target == 9,
                                      scoring="accuracy")
print("정확도 지표: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9,
                           scoring="roc_auc")
print("AUC 지표: {}".format(roc_auc))
```

기본 평가 지표: [0.9 0.9 0.9]

정확도 지표: [0.9 0.9 0.9]

AUC 지표: [0.994 0.99 0.996]

roc_auc_score() 함수를 의미

GridSearchCV

In [73]: # AUC 지표 사용

```
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("AUC 지표를 사용한 그리드 서치")
print("최적의 파라미터: ", grid.best_params_)
print("최상의 교차 검증 점수 (AUC): {:.3f}".format(grid.best_score_))
print("테스트 세트 AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("테스트 세트 정확도: {:.3f}".format(grid.score(X_test, y_test)))
```

AUC 지표를 사용한 그리드 서치

최적의 파라미터: {'gamma': 0.01}

최상의 교차 검증 점수 (AUC): 0.997

테스트 세트 AUC: 1.000

테스트 세트 정확도: 1.000

scoring 옵션

- 분류: accuracy(기본값)
roc_auc(ROC 곡선 아래 면적)
average_precision(정확도-재현율 곡선 아래 면적)
f1_macro, f1_micro, f1_weighted
- 회귀: r2(R^2)
mean_square_error(평균 제곱 오차)
mean_absolute_error(평균 절댓값 오차)

```
In [74]: from sklearn.metrics.scorer import SCORERS
print("가능한 평가 방식:\n{}".format(sorted(SCORERS.keys()))))
```

요약 및 정리

중요한 주의 사항

교차 검증을 해야 합니다.

훈련 데이터: 모델 학습

검증 데이터: 모델과 매개변수 선택

테스트 데이터: 모델 평가

모델 선택과 평가에 적절한 지표를 사용합니다.

높은 정확도를 가진 모델 → 비즈니스 목표에 맞는 모델

현실에서는 불균형한 데이터셋이 아주 많음

거짓 양성(FP)과 거짓 음성(FN)이 많은 영향을 미침