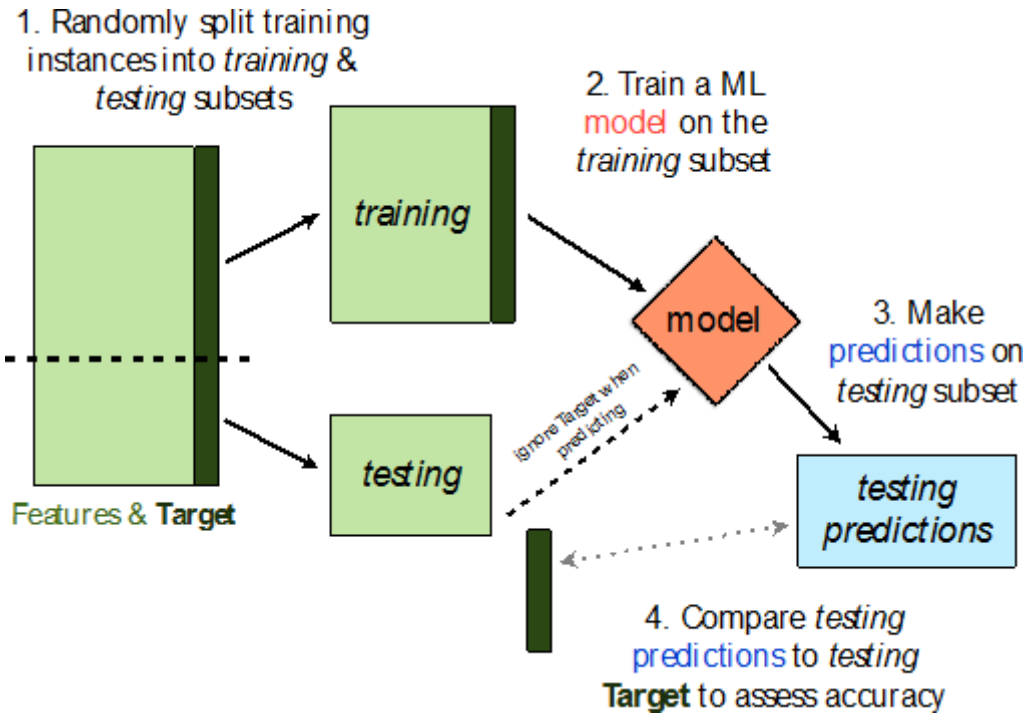
Walidacja krzyżowa (Cross-Validation)

Poznanie parametrów funkcji predykcji i przetestowanie jej na tych samych danych jest błędem metodologicznym. Otrzymalibyśmy model, który po prostu powtórzyłby etykiety czy dane liczbowe, które właśnie zobaczył, miałby doskonały wynik, ale nie zdołałby przewidzieć niczego użytecznego na nieznanych, nowych danych (unseen data). Ta sytuacja nazywa się przeuczeniem (**overfitting**). Aby tego uniknąć, powszechną praktyką jest przeprowadzanie (nadzorowanego) eksperymentu z uczeniem maszynowym, aby część dostępnych danych udostępnić jako **zestaw testowy** `X_test, Y_test`. Zauważ, że słowo "eksperyment" nie jest przeznaczone wyłącznie do użytku akademickiego, ponieważ nawet w warunkach komercyjnych uczenie maszynowe zwykle rozpoczyna się eksperymentalnie. Oto schemat typowego przepływu pracy z walidacją krzyżową w szkoleniu modelowym. Najlepsze parametry można określić za pomocą technik wyszukiwania siatki.



Rysunek 1https://www.developer.com/mgmt/real-world-machine-learning-model-evaluation-and-optimization.html

W scikit-learn - bibliotece do uczenia maszynowego dla języka Python - losowy podział na szkolenia i zestawy testów można szybko obliczyć za pomocą funkcji `train_test_split`. Załadujmy zestaw danych kwiatów Irysu tak, aby pasował do liniowej maszyny wektorów nośnych (Support Vector Machine – SVM):

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> from sklearn import datasets
>>> from sklearn import svm

>>> iris = datasets.load_iris()
>>> iris.data.shape, iris.target.shape
((150, 4), (150,))
```

Możemy teraz szybko przetestować zestaw szkoleniowy, a jednocześnie przekazać 40% danych do testowania (oceny) naszego klasyfikatora:

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)

>>> X_train.shape, y_train.shape
((90, 4), (90,))
>>> X_test.shape, y_test.shape
((60, 4), (60,))

>>> clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.96...
```

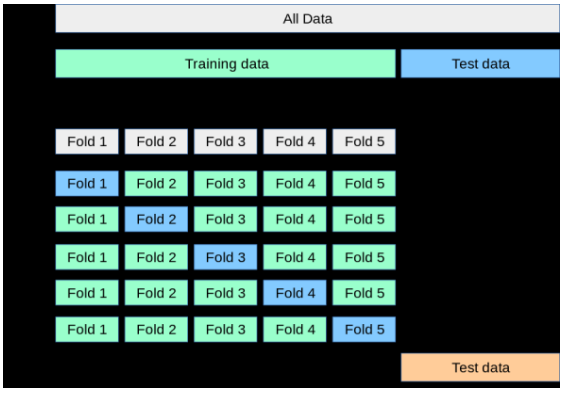
Podczas oceny różnych ustawień ("hiperparametrów") dla estymatorów, takich jak ustawienie C, które musi zostać ustawione ręcznie dla maszyny SVM, nadal istnieje ryzyko nadmiernego dopasowania zestawu testowego, ponieważ parametry mogą być modyfikowane, dopóki estymator nie zostanie wykonany optymalnie. W ten sposób wiedza na temat zestawu testowego może "przeciekać" do modelu, a dane oceny nie będą już raportować o wydajności generalizacji. Aby rozwiązać ten problem, kolejna część zestawu danych może być przechowywana jako tzw. "zestaw walidacji": trening odbywa się na zbiorze treningowym (1), po którym następuje ocena zestawu walidacyjnego (2) i kiedy eksperyment wydaje się być udany , ocena końcowa może być wykonana na zestawie testowym (3).

Jednakże, dzieląc dostępne dane na trzy zestawy, drastycznie zmniejszamy liczbę próbek, które można wykorzystać do nauki modelu, a wyniki mogą zależeć od konkretnego losowego wyboru dla pary zestawów treningowy/walidacyjny.

Rozwiązaniem tego problemu jest procedura zwana krzyżową walidacją (w skrócie CV – Cross-Validation). Zestaw testowy powinien nadal być dostępny do oceny końcowej, ale zestaw walidacyjny nie jest już potrzebny podczas wykonywania CV. W podstawowym podejściu, zwanym k-krotnym CV (**k-Fold CV**), zestaw treningowy jest podzielony na k mniejszych zestawów (inne podejścia są opisane poniżej, ale generalnie są zgodne z tymi samymi zasadami). Następująca procedura jest przestrzegana dla każdego k-foldu:

- Model jest wyszkolony przy użyciu ***k – 1*** foldów jako danych treningowych;
- wynikowy model jest walidowany na pozostałej części danych (tj. jest używany jako zestaw testowy do obliczenia miary wydajności, takiej jak dokładność - accuracy).

Miara wydajności zgłaszana przez *k-fold* sprawdzania krzyżowego jest wtedy średnią z wartości wyliczonych w pętli. Takie podejście może być kosztowne obliczeniowo, ale nie powoduje marnowania zbyt dużej ilości danych (jak w przypadku ustalania arbitralnego zestawu walidacyjnego), co jest główną zaletą w przypadku problemów, takich jak wnioskowanie odwrotne, gdy liczba próbek jest bardzo mała.



Obliczanie wartości walidowanych krzyżowo

Najprostszym sposobem użycia sprawdzania krzyżowego jest wywołanie funkcji pomocniczej `cross_val_score` na estymatorze i zestawie danych.

Poniższy przykład demonstruje, jak oszacować dokładność liniowej maszyny wektorów wspierających jądro na zestawie danych Irysów, dzieląc dane, dopasowując model i obliczając wynik 5 razy z rzędu (z różnymi podziałami za każdym razem):

```
>>> from sklearn.model_selection import cross_val_score
>>> clf = svm.SVC(kernel='linear', C=1)
>>> scores = cross_val_score(clf, iris.data, iris.target, cv=5)
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Wartość średniej i 95-procentowy przedział ufności oszacowania wyników są zatem następujące:

```
>>> print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Accuracy: 0.98 (+/- 0.03)
```

Domyślnie wynik obliczany przy każdej iteracji CV jest metodą oceny estymatora zwaną `score`. Można to zmienić za pomocą parametru punktacji (score):

```
>>> from sklearn import metrics
>>> scores = cross_val_score(
...     clf, iris.data, iris.target, cv=5, scoring='f1_macro')
>>> scores
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Gdy argument `cv` jest liczbą całkowitą, `cross_val_score` domyślnie używa strategii `KFold` lub `StratifiedKFold`, przy czym ta druga jest używana, jeśli estymator pochodzi z `ClassifierMixin`.

Można również użyć innych strategii sprawdzania poprawności krzyżowej, przekazując zamiast tego iterator weryfikacji krzyżowej, na przykład:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> n_samples = iris.data.shape[0]
>>> cv = ShuffleSplit(n_splits=5, test_size=0.3, random_state=0)
>>> cross_val_score(clf, iris.data, iris.target, cv=cv)
array([0.977..., 0.977..., 1.  ..., 0.955..., 1.        ])
```

Inną opcją jest użycie powtarzalnych podziałów (trening, test) jako tablic indeksów, na przykład:

```
>>> def custom_cv_2folds(X):
...     n = X.shape[0]
...     i = 1
...     while i <= 2:
...         idx = np.arange(n * (i - 1) / 2, n * i / 2, dtype=int)
...         yield idx, idx
...         i += 1
...
>>> custom_cv = custom_cv_2folds(iris.data)
>>> cross_val_score(clf, iris.data, iris.target, cv=custom_cv)
array([1.        , 0.973...])
```

Transformacja danych z przetrzymywanymi danymi

Podobnie, jak ważne jest przetestowanie predyktora na danych nie podanych do trenowania, przetwarzanie wstępne (takie jak standaryzacja, wybór funkcji itp.) i przekształcenia danych powinny być uczone na zestawie treningowym i zastosowane na wstrzymanej do przewidywań części danych:

```
>>> from sklearn import preprocessing
>>> X_train, X_test, y_train, y_test = train_test_split(
...     iris.data, iris.target, test_size=0.4, random_state=0)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train_transformed = scaler.transform(X_train)
>>> clf = svm.SVC(C=1).fit(X_train_transformed, y_train)
>>> X_test_transformed = scaler.transform(X_test)
>>> clf.score(X_test_transformed, y_test)
0.9333...
```

`Pipeline` ułatwia komponowanie estymatorów, zapewniając takie zachowanie pod kontrolą krzyżową:

```
>>> from sklearn.pipeline import make_pipeline
>>> clf = make_pipeline(preprocessing.StandardScaler(), svm.SVC(C=1))
>>> cross_val_score(clf, iris.data, iris.target, cv=cv)
...
array([0.977..., 0.933..., 0.955..., 0.933..., 0.977...])
```

Funkcja cross_validate i sposoby oceny

Funkcja `cross_validate` różni się od `cross_val_score` na dwa sposoby:

- Pozwala określić wiele metryk do oceny.
- Zwraca zestawienie zawierające czasy dopasowania, czasy punktacji (i opcjonalnie wyniki treningowe oraz dopasowane estymatory) oprócz wyniku testu.

W przypadku pojedynczej oceny metryki, gdzie parametrem oceny jest łańcuch, wywoływalny (callable) lub nie (None), komendami będą `['test_score', 'fit_time', 'score_time']`

Dla wielokrotnej oceny metryki wartością zwracaną jest zestaw z następującymi komendami `['test_<scorer1_name>', 'test_<scorer2_name>', 'test_<scorer...>', 'fit_time', 'score_time']`

`return_train_score` jest domyślnie ustawiony na `True`. Dodaje on klucze wyników dla wszystkich ocen. Jeśli wyniki trenowania nie są potrzebne, należy je wyraźnie ustawić na `False`.

Możesz również zachować estymator dopasowany do każdego zestawu treningowego, ustawiając `return_estimator = True`.

Wiele metryk można określić jako listę, krotkę (tuple) lub zestaw predefiniowanych nazw ocen:

```
>>> from sklearn.model_selection import cross_validate
>>> from sklearn.metrics import recall_score
>>> scoring = ['precision_macro', 'recall_macro']
>>> clf = svm.SVC(kernel='linear', C=1, random_state=0)
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5, return_train_score=False)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_precision_macro', 'test_recall_macro']
>>> scores['test_recall_macro']
array([0.96..., 1.  ..., 0.96..., 0.96..., 1.        ])
```

Lub jako zestaw mapujący nazwy ocen do predefiniowanych lub własnych funkcji oceniających:

```
>>> from sklearn.metrics.scorer import make_scorer
>>> scoring = {'prec_macro': 'precision_macro',
...            'rec_micro': make_scorer(recall_score, average='macro')}
>>> scores = cross_validate(clf, iris.data, iris.target, scoring=scoring,
...                         cv=5, return_train_score=True)
>>> sorted(scores.keys())
['fit_time', 'score_time', 'test_prec_macro', 'test_rec_micro',
 'train_prec_macro', 'train_rec_micro']
>>> scores['train_rec_micro']
array([0.97..., 0.97..., 0.99..., 0.98..., 0.98...])
```

Poniżej przykład dla `cross_validate` używający pojedynczej metryki:

```
>>> scores = cross_validate(clf, iris.data, iris.target,
...                         scoring='precision_macro', cv=5,
...                         return_estimator=True)
>>> sorted(scores.keys())
['estimator', 'fit_time', 'score_time', 'test_score', 'train_score']
```

Uzyskanie przewidywania przez walidację krzyżową

Funkcja `cross_val_predict` ma podobny interfejs do `cross_val_score`, ale zwraca, dla każdego elementu na wejściu, prognozę, która została uzyskana dla tego elementu, gdy był w zestawie testowym. Można stosować tylko strategie sprawdzania poprawności krzyżowej, które przypisują wszystkie elementy do zestawu testowego dokładnie raz (w przeciwnym razie zgłaszany jest wyjątek).

Uwaga !

Informacja dotycząca niewłaściwego użycia `cross_val_predict`

Wynik `cross_val_predict` może być inny niż wynik uzyskany przy użyciu `cross_val_score`, ponieważ elementy są pogrupowane na różne sposoby. Funkcja `cross_val_score` pobiera średnią z foldów walidacji krzyżowej, podczas gdy `cross_val_predict` po prostu zwraca etykiety (lub prawdopodobieństwa) z kilku odrębnych niezidentyfikowanych modeli. Zatem `cross_val_predict` nie jest odpowiednią miarą błędu generalizacji.

Funkcja `cross_val_predict` jest odpowiednia dla:

- Wizualizacja prognoz uzyskanych z różnych modeli.
- Mieszanie modelu: Gdy przewidywania jednego nadzorowanego estymatora są wykorzystywane do szkolenia innego estymatora w metodach zespołowych.

Iteratory walidacji krzyżowej

Poniższe sekcje zawierają listę narzędzi do generowania indeksów, które mogą być używane do generowania podziałów zbiorów danych zgodnie z różnymi strategiami walidacji krzyżowej.

1. Iteratory walidacji krzyżowej dla danych IID

Zakładając, że niektóre dane są niezależne, i identycznie rozłożone (IID - Independent and Identically Distributed) co zakłada, że wszystkie próbki pochodzą z tego samego procesu generatywnego i zakłada się, że proces generatywny nie ma pamięci wcześniej wygenerowanych próbek.

W takich przypadkach można użyć następujących walidatorów krzyżowych.

UWAGA

Pomimo, że dane z przestrzeni IID są powszechnym założeniem teorii uczenia maszynowego, rzadko występują w praktyce. Jeśli ktoś wie, że próbki zostały wygenerowane przy użyciu procesu zależnego od czasu, bezpieczniej jest użyć schematu walidacji krzyżowej opartego na szeregach czasowych. Podobnie, jeśli wiemy, że proces generatywny ma strukturę grupową (próbki z różnych przedmiotów, eksperymentów, urządzeń pomiarowych) bezpieczniej jest stosować krzyżową walidację grupową.

1.1. K-Fold

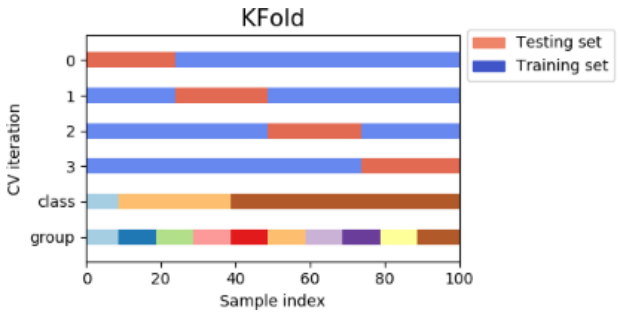
`KFold` dzieli wszystkie próbki w ***k*** grupach próbek, zwanych foldami (jeśli ***k = n*** jest to odpowiednik strategii *Leave One Out*), jeśli to możliwe o jednakowych rozmiarach. Funkcja przewidywania jest nauczana przy użyciu foldów, a pominięty fold jest używany do testowania.

Przykład 2-krotnej walidacji krzyżowej na zestawie danych z 4 próbkami:

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold

>>> X = ["a", "b", "c", "d"]
>>> kf = KFold(n_splits=2)
>>> for train, test in kf.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[0 1] [2 3]
```

Oto wizualizacja zachowania walidacji krzyżowej. Zauważ, że na klasy `KFold` nie mają wpływu ani klasy ani grupy.



Każdy fold składa się z dwóch tablic: pierwsza jest związana z zestawem treningowym, a druga z zestawem testowym. W ten sposób można utworzyć zestawy treningowe/testowe przy użyciu indeksowania `numpy`:

```
>>> X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
>>> y = np.array([0, 1, 0, 1])
>>> X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
```

1.2. Powtarzany K-Fold

`RepeatedKFold`d powtarza K-Fold ***n*** razy. Można go użyć, gdy trzeba uruchomić `KFold` ***n*** razy, tworząc różne podziały w każdym powtórzeniu.

Przykład 2-krotnego składania K powtórzonego 2 razy:

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> random_state = 12883823
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)
>>> for train, test in rkf.split(X):
...     print("%s %s" % (train, test))
...
[2 3] [0 1]
[0 1] [2 3]
[0 2] [1 3]
[1 3] [0 2]
```

Podobnie, `RepeatedStratifiedKFold` powtarza Stratified K-Fold ***n*** razy z różną randomizacją w każdym powtórzeniu.

1.3. Leave One Out (LOO)

LeaveOneOut (lub LOO) to prosta weryfikacja krzyżowa. Każdy zestaw do nauki jest tworzony przez pobranie wszystkich próbek z wyjątkiem jednej, a zestaw testowy jest pominięty. Tak więc dla próbek mamy różne zestawy treningowe i różne testy. Ta procedura walidacji krzyżowej nie marnuje zbyt wiele danych, ponieważ tylko jedna próbka jest usuwana z zestawu szkoleniowego:

```
>>> from sklearn.model_selection import LeaveOneOut

>>> X = [1, 2, 3, 4]
>>> loo = LeaveOneOut()
>>> for train, test in loo.split(X):
...     print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Potencjalni użytkownicy LOO do wyboru modelu powinni rozważyć kilka znanych zastrzeżeń. W porównaniu z walidacją krzyżową KFold buduje się ***n*** modeli z ***n*** próbkami zamiast ***k*** modeli, gdzie ***n > k***. Ponadto każdy jest szkolony na ***n – 1*** próbkach, a nie na ***(k – 1)n/k***. W obu przypadkach, zakładając, że ***k*** nie jest zbyt duże i ***k < n***, LOO jest bardziej kosztowny obliczeniowo niż weryfikacja krzyżowa KFold.

Pod względem dokładności LOO często powoduje wysoką wariancję jako estymator błędu testu. Intuicyjnie, ponieważ ***n – 1*** z ***n*** próbek jest używanych do budowy każdego modelu, modele zbudowane z foldów są praktycznie identyczne ze sobą i z modelem zbudowanym z całego zestawu treningowego.

Jeśli jednak krzywa uczenia się jest stroma dla danego rozmiaru szkolenia, wówczas 5- lub 10-krotna walidacja krzyżowa może zawyżać błąd generalizacji.

Zasadniczo większość autorów i dowody empiryczne sugerują, że 5- lub 10-krotna walidacja krzyżowa powinna być preferowana zamiast LOO.

1.4. Leave P Out (LPO)

`LeavePOut` jest bardzo podobny do `LeaveOneOut`, ponieważ tworzy wszystkie możliwe zestawy treningowe / testowe, usuwając próbki z całego zestawu. W przypadku próbek tworzy to pary trening-test. W przeciwieństwie do `LeaveOneOut` i `KFold` zestawy testowe będą się na siebie nakładać.

Przykład Leave-2-Out na zestawie danych z 4 próbkami:

```
>>> from sklearn.model_selection import LeavePOut

>>> X = np.ones(4)
>>> lpo = LeavePOut(p=2)
>>> for train, test in lpo.split(X):
...     print("%s %s" % (train, test))
[2 3] [0 1]
[1 3] [0 2]
[1 2] [0 3]
[0 3] [1 2]
[0 2] [1 3]
[0 1] [2 3]
```

1.5. Walidacja losowa permutacji krzyżowych Shuffle & Split

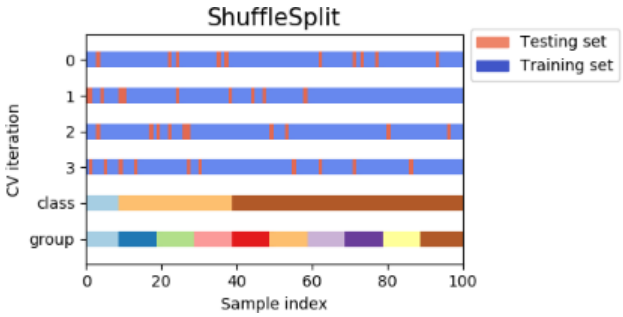
`Iterator ShuffleSplit` wygeneruje zdefiniowaną przez użytkownika liczbę niezależnych podziałów zestawu danych trening- test. Próbki są najpierw tasowane, a następnie dzielone na parę zestawów treningowych i zestawów testowych.

Możliwe jest kontrolowanie losowości dla powtarzalności wyników poprzez jawne zaszczepienie generatora liczb losowych pseudolosowych `random_state`.

Oto przykład użycia:

```
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.arange(10)
>>> ss = ShuffleSplit(n_splits=5, test_size=0.25,
...     random_state=0)
>>> for train_index, test_index in ss.split(X):
...     print("%s %s" % (train_index, test_index))
[9 1 6 7 3 0 5] [2 8 4]
[2 9 8 0 6 7 4] [3 5 1]
[4 5 1 0 6 9 7] [2 3 8]
[2 7 5 8 0 3 4] [6 1 9]
[4 1 0 6 8 9 3] [5 2 7]
```

Oto wizualizacja zachowania walidacji krzyżowej. Zauważ, że na `ShuffleSplit` nie mają wpływu klasy ani grupy.



`ShuffleSplit` jest zatem dobrą alternatywą dla walidacji krzyżowej KFold, która pozwala na dokładniejszą kontrolę liczby iteracji i proporcji próbek po każdej stronie podziału trening / test.

2. Iteratory walidacji krzyżowej ze stratyfikacją opartą na etykietach klas

Niektóre problemy z klasyfikacją mogą wykazywać dużą nierównowagę w rozkładzie klas docelowych: na przykład może istnieć kilka razy więcej próbek negatywnych niż próbek pozytywnych. W takich

przypadkach zaleca się stosowanie próbkowania warstwowego, jak zaimplementowano w StratifiedKFold i StratifiedShuffleSplit, aby zapewnić, że względne częstotliwości klas są w przybliżeniu zachowane w każdym treningu i foldzie walidacyjnym.

2.1. Stratified k-fold

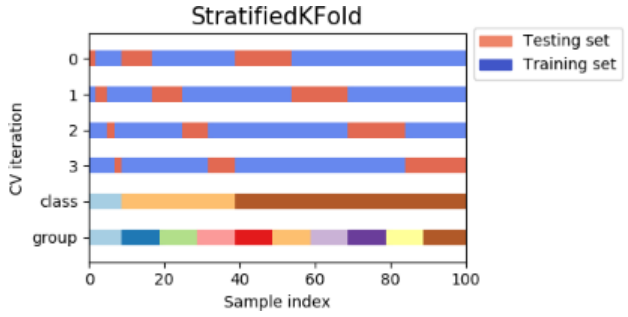
`StratifiedKFold`jest odmianą Kfold, która zwraca stratyfikowane foldy: każdy zestaw zawiera w przybliżeniu taki sam procent próbek każdej klasy docelowej, jak kompletny zestaw.

Przykład trójwarstwowej walidacji krzyżowej na zestawie danych z 10 próbkami z dwóch lekko niezrównoważonych klas:

```
>>> from sklearn.model_selection import StratifiedKFold

>>> X = np.ones(10)
>>> y = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
>>> skf = StratifiedKFold(n_splits=3)
>>> for train, test in skf.split(X, y):
...     print("%s %s" % (train, test))
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

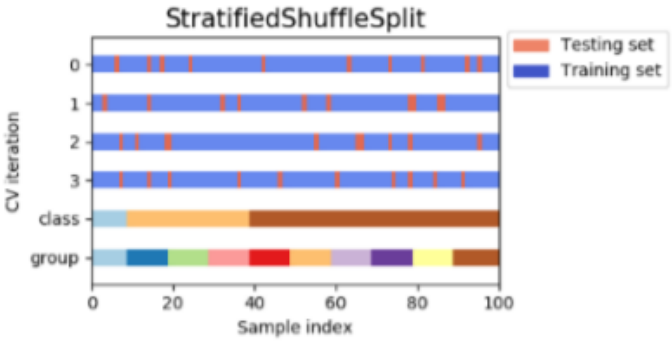
Oto wizualizacja zachowania walidacji krzyżowej:



`RepeatedStratifiedKFold` może być użyty do powtórzenia Stratified K-Fold ***n*** razy z różną randomizacją w każdym powtórzeniu.

2.2. Stratified Shuffle Split

`StratifiedShuffleSplit` jest odmianą `ShuffleSplit`, która zwraca podzielone warstwy, czyli tworzy podziały, zachowując taki sam procent dla każdej klasy docelowej, jak w całym zestawie. Oto wizualizacja zachowania walidacji krzyżowej.



3. Iteratory walidacji krzyżowej dla zgrupowanych danych.

Założenie danych IID zostaje zerwane, jeśli podstawowy proces generatywny daje grupy zależnych próbek.

Takie grupowanie danych jest specyficzne dla domeny. Przykładem może być gromadzenie danych medycznych od wielu pacjentów z wieloma próbkami pobranymi od każdego pacjenta. Takie dane mogą być zależne od indywidualnej grupy. W naszym przykładzie identyfikator pacjenta dla każdej próbki będzie identyfikatorem grupy.

W tym przypadku chcielibyśmy wiedzieć, czy model wyszkolony na określonym zestawie grup uogólnia dobrze na niewidoczne grupy. Aby to zmierzyć, musimy upewnić się, że wszystkie próbki w zakładce walidacji pochodzą od grup, które w ogóle nie są reprezentowane w spasowanym treningu.

W tym celu można użyć następujących splitterów do sprawdzania poprawności krzyżowej. Identyfikator grupowania próbek jest określany za pomocą parametru `groups`.

3.1. Grupowanie k-fold

`GroupKFold` to odmiana k-fold, która zapewnia, że ta sama grupa nie jest reprezentowana zarówno w zestawach testowych, jak i szkoleniowych. Na przykład, jeśli dane są uzyskiwane od różnych podmiotów z kilkoma próbkami na podmiot i jeśli model jest wystarczająco elastyczny, aby uczyć się na podstawie cech wysoce specyficznych dla osoby, może nie uogólnić się na nowe tematy. `GroupKFold` umożliwia wykrycie tego rodzaju sytuacji związanych z przeuczeniem.

Wyobraź sobie, że masz trzy przedmioty, każdy z przypisanym numerem od 1 do 3:

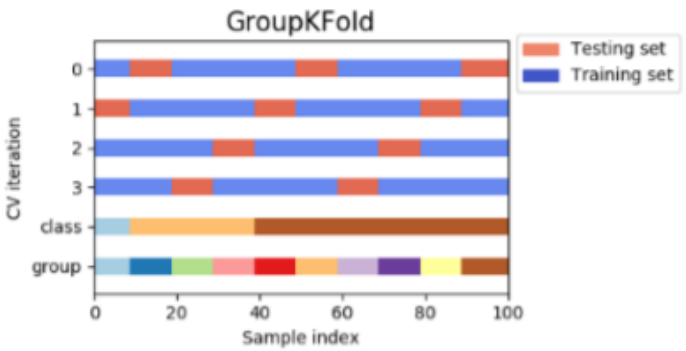
```
>>> from sklearn.model_selection import GroupKFold

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 8.8, 9, 10]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "d", "d", "d"]
>>> groups = [1, 1, 1, 2, 2, 2, 3, 3, 3, 3]

>>> gkf = GroupKFold(n_splits=3)
>>> for train, test in gkf.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[0 1 2 3 4 5] [6 7 8 9]
[0 1 2 6 7 8 9] [3 4 5]
[3 4 5 6 7 8 9] [0 1 2]
```

Każdy temat jest w innej próbie, a ten sam temat nigdy nie jest testowany i szkolony. Zauważ, że foldy nie mają dokładnie tego samego rozmiaru z powodu braku równowagi w danych.

Oto wizualizacja zachowania walidacji krzyżowej:



3.2. Opuść jedną grupę

`LeaveOneGroupOut` to schemat sprawdzania poprawności krzyżowej, który przechowuje próbki zgodnie z dostarczoną przez inną firmę tablicą grup całkowitych. Ta informacja o grupie może być użyta do zakodowania dowolnie określonych predefiniowanych foldów walidacji krzyżowej.

Każdy zestaw treningowy składa się zatem ze wszystkich próbek z wyjątkiem tych związanych z określoną grupą.

Na przykład, w przypadku wielu eksperymentów, LeaveOneGroupOut może zostać użyty do utworzenia walidacji krzyżowej na podstawie różnych eksperymentów: tworzymy zestaw treningowy wykorzystując próbki wszystkich eksperymentów z wyjątkiem jednego.

```
>>> from sklearn.model_selection import LeaveOneGroupOut

>>> X = [1, 5, 10, 50, 60, 70, 80]
>>> y = [0, 1, 1, 2, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3, 3]
>>> logo = LeaveOneGroupOut()
>>> for train, test in logo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

Innym powszechnym zastosowaniem jest wykorzystanie informacji o czasie: na przykład grupy mogą być rokiem zbierania próbek, a tym samym pozwalają na walidację krzyżową z podziałami opartymi na czasie.

3.3. Leave P Groups Out

`LeavePGroupsOut` jest podobny do `LeaveOneGroupOut`, ale usuwa próbki powiązane z grupami dla każdego zestawu treningowego / testowego.

Przykład dla Leave-2-Group Out:

```
>>> from sklearn.model_selection import LeavePGroupsOut

>>> X = np.arange(6)
>>> y = [1, 1, 1, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3]
>>> lpgo = LeavePGroupsOut(n_groups=2)
>>> for train, test in lpgo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[4 5] [0 1 2 3]
[2 3] [0 1 4 5]
[0 1] [2 3 4 5]
```

3.4. Group Shuffle Split

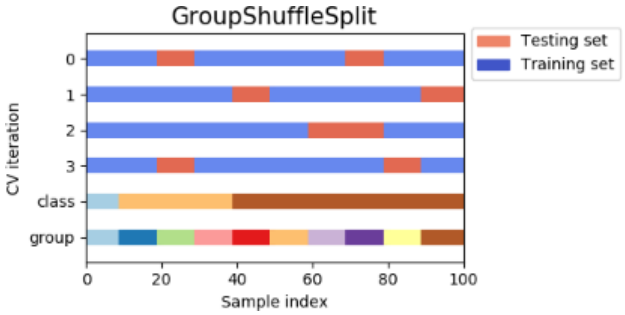
Iterator `GroupShuffleSplit` zachowuje się jak kombinacja `ShuffleSplit` i `LeavePGroupsOut` i generuje sekwencję losowych partycji, w których podzbiór grup jest utrzymywany dla każdego podziału.

Oto przykład użycia:

```
>>> from sklearn.model_selection import GroupShuffleSplit

>>> X = [0.1, 0.2, 2.2, 2.4, 2.3, 4.55, 5.8, 0.001]
>>> y = ["a", "b", "b", "b", "c", "c", "c", "a"]
>>> groups = [1, 1, 2, 2, 3, 3, 4, 4]
>>> gss = GroupShuffleSplit(n_splits=4, test_size=0.5, random_state=0)
>>> for train, test in gss.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
...
[0 1 2 3] [4 5 6 7]
[2 3 6 7] [0 1 4 5]
[2 3 4 5] [0 1 6 7]
[4 5 6 7] [0 1 2 3]
```

Oto wizualizacja zachowania walidacji krzyżowej.



Ta klasa jest przydatna, gdy pożądane jest zachowanie LeavePGroupsOut, ale liczba grup jest na tyle duża, że generowanie wszystkich możliwych partycji z wstrzymanymi grupami byłoby zbyt drogie. W takim scenariuszu GroupShuffleSplit dostarcza losową próbkę (z wymianą) podziałów pociągu / testu wygenerowanych przez LeavePGroupsOut.

3.5. Predefiniowane zestawy podziałów / walidacji

W przypadku niektórych zestawów danych istnieje już predefiniowany podział danych na foldy treningowe i walidacyjne lub na kilka foldów walidacji krzyżowej. Używając `PredefinedSplit` można użyć tych foldów, np. podczas wyszukiwania hiperparametrów.

Na przykład, gdy używasz zestawu sprawdzania poprawności, ustaw wartość `test_fold` na 0 dla wszystkich próbek, które są częścią zestawu sprawdzania poprawności, i dla -1 dla wszystkich innych próbek.

3.6. Walidacja krzyżowa danych szeregów czasowych

Dane szeregów czasowych charakteryzują się korelacją między obserwacjami, które są bliskie w czasie (autokorelacja). Jednak klasyczne techniki walidacji krzyżowej, takie jak `KFold` i `ShuffleSplit`, zakładają, że próbki są niezależne i identycznie rozmieszczone, co skutkowałoby nieuzasadnioną korelacją między instancjami szkoleniowymi i testowymi (uzyskując słabe szacunki błędu generalizacji) na danych szeregów czasowych. Dlatego bardzo ważne jest, aby ocenić nasz model dla danych szeregów czasowych na obserwacjach „przyszłych" najmniej podobnych do tych, które są wykorzystywane do szkolenia modelu. Aby to osiągnąć, jednym rozwiązaniem jest `TimeSeriesSplit`.

3.7. Seria czasowa Split

`TimeSeriesSplit` to odmiana k-fold, która zwraca pierwsze ***k*** foldy jako zestaw treningowy i ***(k + 1)***-ty fold jako zestaw testowy. Zauważ, że w przeciwieństwie do standardowych metod walidacji

krzyżowej, kolejne zestawy treningowe są zestawami tych, które są przed nimi. Ponadto dodaje wszystkie nadwyżkowe dane do pierwszej partycji treningowej, która jest zawsze używana do szkolenia modelu.

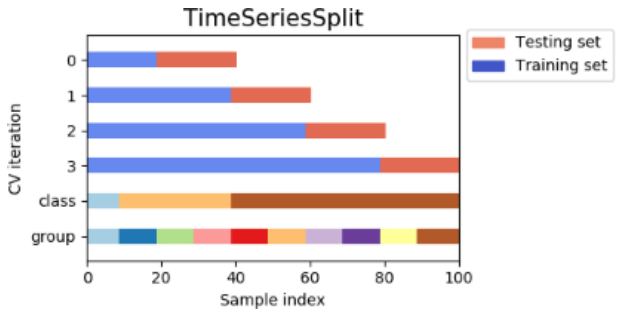
Ta klasa może być używana do krzyżowej walidacji próbek danych szeregów czasowych, które są obserwowane w stałych odstępach czasu.

Przykład 3-częściowej walidacji krzyżowej szeregów czasowych na zestawie danych z 6 próbkami:

```
>>> from sklearn.model_selection import TimeSeriesSplit

>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit(n_splits=3)
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=3)
>>> for train, test in tscv.split(X):
...     print("%s %s" % (train, test))
[0 1 2] [3]
[0 1 2 3] [4]
[0 1 2 3 4] [5]
```

Oto wizualizacja zachowania walidacji krzyżowej.



4. Tasowanie (shuffling)

Jeśli porządkowanie danych nie jest arbitralne (np. próbki o tej samej etykiecie klasy są przyległe), przetasowanie w pierwszej kolejności może być niezbędne, aby uzyskać znaczący wynik walidacji krzyżowej. Jednakże może być odwrotnie, jeśli próbki nie są niezależnie i identycznie rozmieszczone. Na przykład, jeśli próbki odpowiadają artykułom prasowym i są uporządkowane według czasu publikacji, to przetasowanie danych prawdopodobnie doprowadzi do nadpisania modelu i zawyżenia wyniku walidacji: zostanie wykonane testowanie na próbkach, które są sztucznie podobne (bliskie w czasie) do próbek szkoleniowych.

Niektóre iteratory sprawdzania poprawności krzyżowej, takie jak KFold, mają wbudowaną opcję przetasowania indeksów danych przed ich podziałem. Zauważ, że:

- Zużywa to mniej pamięci niż bezpośrednie przetasowanie danych.
- Domyślnie nie jest przeprowadzane tasowanie, nawet dla walidacji krzyżowej (stratified) K-fold przeprowadzanej przez podanie `cv = some_integer` w `cross_val_score`, przeszukiwaniu siatki, itp. Pamiętaj, że `train_test_split` nadal zwraca losowy podział.
- Parametr `random_state` ma domyślnie wartość `None`, co oznacza, że tasowanie będzie różne za każdym razem, gdy `KFold(..., shuffle = True)` jest iterowany. Jednak `GridSearchCV` użyje tego samego tasowania dla każdego zestawu parametrów zatwierdzonych przez pojedyncze wywołanie jego metody dopasowania w komendzie `fit`.
- Aby uzyskać identyczne wyniki dla każdego podziału, ustaw `random_state` na liczbę całkowitą.