Метод опорных векторов. Оценка качества классификации. Методы кросс-валидации

Цель занятия

В результате обучения на этой неделе:

- вы познакомитесь с методом опорных векторов, который используется для задач классификации и регрессионного анализа;
- узнаете способ создания нелинейного классификатора с помощью так называемого ядерного трюка (kernel trick);
- продолжите изучение метрик оценки качества классификации, разберете ROC-AUC;
- научитесь использовать метод кросс-валидации для оценки качества модели.

План занятия

1. Метод опорных векторов
2. Нелинейный метод опорных векторов
3. Оценка качества классификации
4. Методы кросс-валидации
5. Cross-validation riddle

Конспект занятия

1. Метод опорных векторов

Метод опорных векторов (Support Vector Machine, SVM) будет первым шагом в сторону обобщения всех уже рассмотренных подходов на нелинейные случаи.

Повторим задачу классификации и линейный классификатор. Рассмотрим для простоты задачу бинарной классификации:

$X^l = \left(x_i,\ y_i\right)_{i=1}^{l},\ x_i \in R^n,\ y_i \in \{-1,\ 1\}$

Модель линейной классификации — гиперплоскость, которая отделяет точки одного класса от другого:

$$a\left(x;\ w,\ w_0\right) = sign\left(< x, w > - \ w_0\right),\ w \in R^n,\ w_0 \in R.$$

Здесь $< x, w > - \ w_0$ — метка класса,

w — вектор нормали к гиперплоскости.

Функция потерь — эмпирический риск. Мы его минимизируем:

$$\sum_{i=1}^{\ell}\left[a(x_i; w, w_0) \neq y_i\right] = \sum_{i=1}^{\ell}\left[M_i(w, w_0) < 0\right] \ \rightarrow \ \min_{w, w_0}.$$

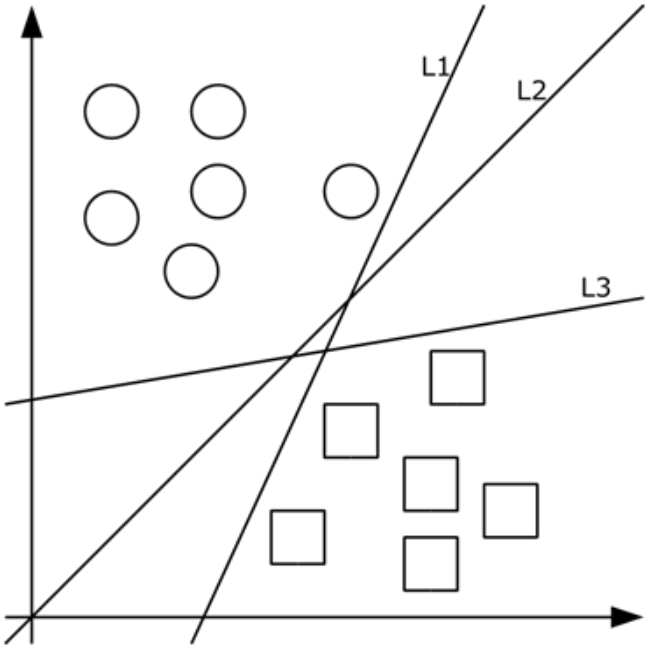
Отступ (margin) $M_i\left(w,\ w_0\right) = \left(< x_i, w > - w_0\right)y_i$, который показывает, насколько глубоко в собственном классе находится объект.

Метод опорных векторов появился в 1963 году, был представлен Владимиром Вапником и Алексеем Червоненкис. Данная техника позволяла хорошо и устойчиво решать задачи. В дальнейшем была обобщена на нелинейные случаи, благодаря kernel trick или смене ядра в 1992 году.

Линейно разделимая выборка

Пусть наша выборка линейно разделимая:

$\exists w,\ w_0 \colon\ M_i\left(w,\ w_0\right) = y_i\left(< w,\ x_i > - w_0\right) > 0,\ i = 1,\ldots, l$



И разделяющих плоскостей может быть больше одной. Ответить на вопрос, какая из них лучшая, мы ответить не можем. Нужно вводить дополнительные критерии качества.

Глядя на рисунок выше, кажется, что все-таки оптимальной является гиперплоскость L2. Она находится посередине между классами объектов. И если ее немного отклонить, разделение объектов не изменится.

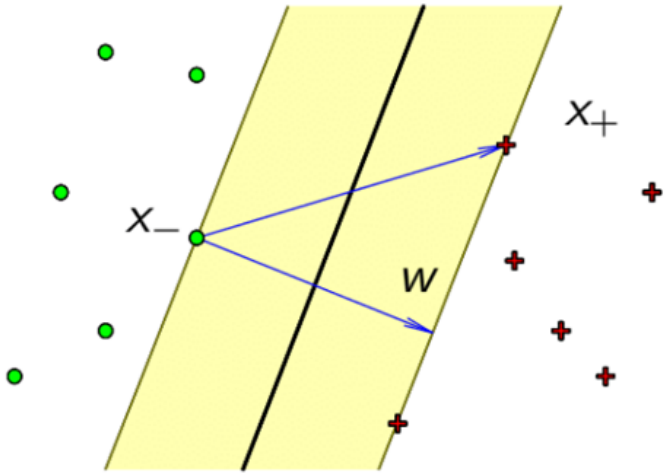
Модель является **неустойчивой**, если при незначительном изменении ее параметров происходит значительное изменение предсказания модели.

Зададим формально оптимальную гиперплоскость.

Раз выборка линейно разделима, мы можем провести гиперплоскость и таким образом перенормировать веса, чтобы выполнялись следующие неравенства:

$$\forall x_+ : \ < w, x_+ > \ - w_0 \geq 1$$

$$\forall x_- : \ < w, x_- > \ - w_0 \leq - 1$$



Минимальный отступ равен единице:

$$\min_{i=1,\dots,\ell} M_i(w, w_0) = 1$$

Тогда

$$\frac{< x_+ - x_-, w >}{||w||} \geq \frac{2}{||w||} \to max$$

$< w, x_+ >$ и $< w, x_- >$ — расстояние от гиперплоскости до положительных и отрицательных объектов соответственно. Если мы будем максимизировать расстояние, то классы друг от друга будут раздвигаться. То есть мы подбираем такую гиперплоскость, которая максимально удалена от объектов первого и второго класса.

$\frac{2}{||\omega||}$ — оценка сверху. Она от точек не зависит, зависит только от параметра. То есть мы, минимизируя параметр, максимизируем расстояние между классами.

Формальная постановка задачи классификации с помощью метода опорных векторов для линейно разделимой выборки:

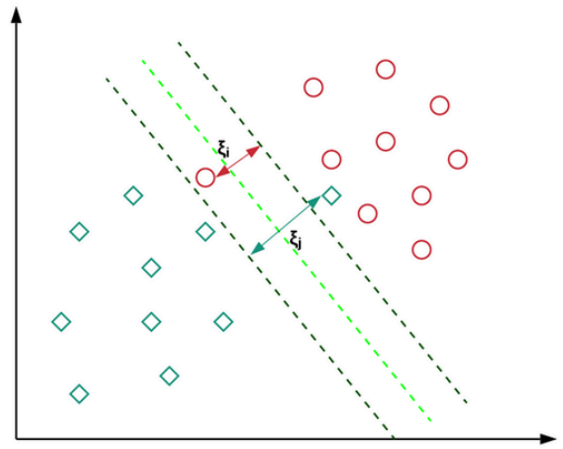
$$\begin{cases} \frac{1}{2}\|w\|^2 \to \min\limits_{w, w_0}; \\ M_i(w, w_0) \geqslant 1, \quad i = 1, \dots, \ell. \end{cases}$$

Но линейно разделимые выборки встречаются редко.

Что делать, если выборка линейно неразделима

Попытаемся обобщить метод опорных векторов.

Разрешим некоторым точкам иметь отрицательный отступ, то есть попадать в чужой класс:



Тогда для каждого объекта отступ:

$M_i(w, w_0) \geq 1 - \xi_i, \; i = 1, \ldots, l$

ξ_i должны удовлетворять следующим условиям:

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{\ell}\xi_i \to \min\limits_{w,w_0,\xi}; \\ M_i(w, w_0) \geqslant 1 - \xi_i, \quad i = 1, \ldots, \ell; \\ \xi_i \geqslant 0, \quad i = 1, \ldots, \ell. \end{cases}$$

Получаем постановку условной оптимизационной задачи.

Откуда безусловная оптимизационная задача:

$$C\sum_{i=1}^{\ell}(1 - M_i(w, w_0))_+ + \frac{1}{2}\|w\|^2 \to \min_{w,w_0}.$$

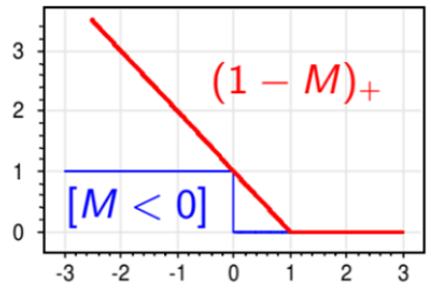
Решается такая задача градиентными методами, методами линейного и квадратичного программирования, методом множителей Лагранжа (поскольку это задача с ограничениями).

Hinge loss

Введем понятие Hinge loss, еще одной функции потерь:

$$Q(w, w_0) = \sum_{i=1}^{l} [M_i(w, w_0) < 0] \leq \sum_{i=1}^{l} (1 - M_i(w, w_0))_+ + \frac{1}{2C}||w||^2 \to min$$

Это опять оценка сверху на функционал эмпирического риска.



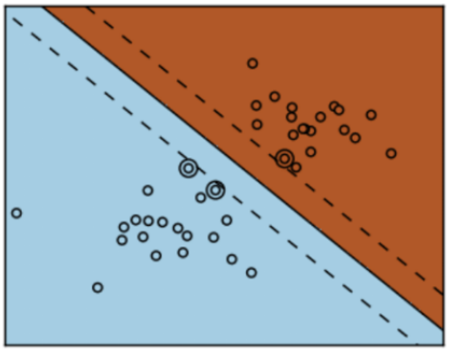
$\sum_{i=1}^{l} (1 - M_i(w, w_0))_+$ – аппроксимация, ошибка, которая штрафует модель за неправильное предсказание, прорелаксированное ограничение на задачу;

$\frac{1}{2C}||w||^2$ – регуляризация, исходная оптимизационная задача с некоторым коэффициентом, увеличение расстояния между классами.

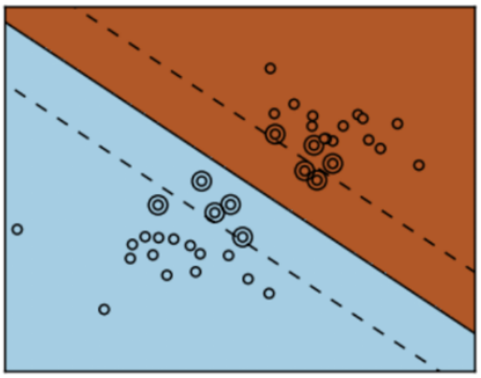
Полученное решение оптимизационной задачи будет смещено относительно исходной задачи. То есть мы уже решаем не ту задачу, которую хотели бы, но получаем более устойчивое решение.

Следует обратить внимание на константу C:

- Если константа C большая, то регуляризация слабая:



- Если константа C маленькая, то регуляризация сильная:



При маленькой константе мы раздвигаем границы, делаем отступ большим, начинаем учитывать при обучении большее число объектов.

Почему все-таки название метода — «метод опорных векторов»? Если точка находится в глубине своего класса, у нее функция ошибки (Hinge loss) нулевая. Только те точки, которые попали внутрь разделяющей полосы, по сути, влияют на решение оптимизационной задачи. Это те самые опорные вектора. И это позволяет эффективно решать задачи при малых вычислительных мощностях.

2. Нелинейный метод опорных векторов

SVM позволяет решать задачи, когда разделяющая поверхность уже не является линейной.

Рассмотрим задачу, когда выборки (положительного и отрицательного классов, по-прежнему рассматриваем бинарную классификацию) не могут быть разделены

простой линейной плоскостью. Мы можем попытаться изменить само признаковое пространство, изменить в нем меру близости. Например, заменить скалярное произведение на что-то еще. Мы можем ввести какую-то другую функцию, которая обладает свойствами **ядра** и позволяет оценить, насколько точки похожи друг на друга.

Ядра бывают разные, и они отвечают за смену скалярного произведения в пространстве на что-то иное.

Свойства функции ядра:

$K(x,\ x')\colon X \times X \to R$

$\exists\psi\colon X \to H\colon K(x,\ x') =< \psi(x),\ \psi(x') >,\ H$ — Гильбертово пространство.

Таким образом, ядро позволяет оценивать расстояние между точками по-другому.

Примеры ядер, которые удовлетворяют этим ограничениям:

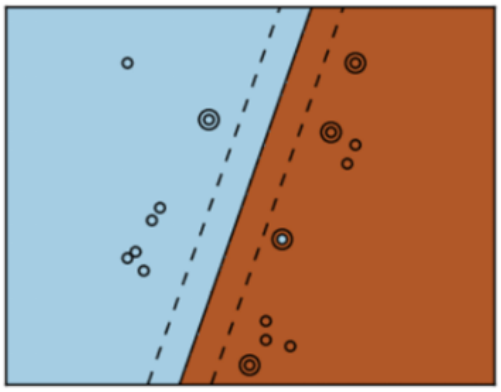
$K(x,\ x') =< x,\ x'>^2$ — квадратичное.

$K(x,\ x') =< x,\ x'>^d$ — полиномиальное степени d.

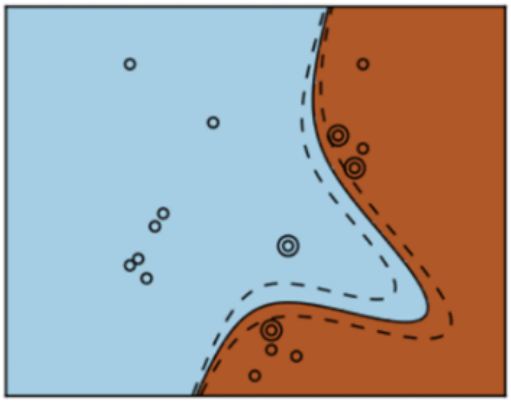
$K(x,\ x') = (< x,\ x' >+\ 1)^d$ — полиномиальное степени $\leq d$.

$K(x,\ x') = exp\left(-\ \gamma||x\ -\ x'||^2\right)$ — ядро радиальных базисных функций (RBF).

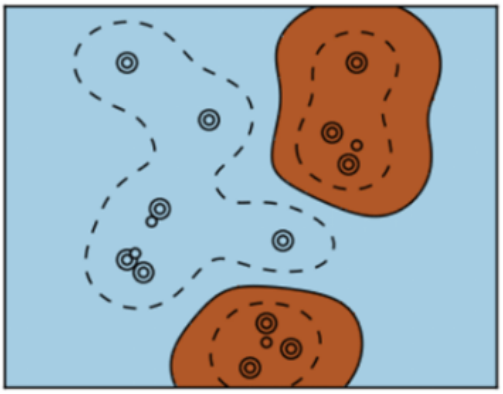
Примеры. $< x,\ x' >$:



$(< x, x' >+ 1)^d$, $d = 3$:



$exp(-\gamma||x - x'||^2)$:



Получаем, что при смене скалярного произведения (признакового описания) получаем другую метрику и норму.

Таким образом, метод SVM позволяет решать задачи, где у нас существенно нелинейные зависимости. Но!

Важно! Ядро приходится выбирать руками. По сути, это делает эксперт на основании своего опыта, интуиции, предположений, свойств задачи и т. д.

Что, если ядер не 1, а 25? Вручную их перебирать долго и трудно, особенно если учесть, что задача может быть большая, и точек в ней много. Обучаться все это может сутками.

SVM требует доли экспертизы в своей предметной области и в некоторой степени везения. Ядро — по сути, еще один гиперпараметр, который нужно как-то подбирать.

Тем не менее подведем некоторые итоги, свойства SVM:

- Позволяет найти линейную и нелинейную разделяющую поверхность в признаковом пространстве.
- Исходные признаки за счет использования ядра могут учитываться нелинейным образом.
- Ядро нужно выбирать аккуратно.
- SVM естественным образом включает в себя регуляризацию.

3. Оценка качества классификации

ROC-кривая

Мы уже разобрали, что такое Accuracy, Precision, Recall, F-score. Все эти меры качества обращают внимание именно на метки класса, предсказываемые классификатором.

Многие классификаторы умеют предсказывать чуть больше информации. Например, логистическая регрессия или наивный байесовский классификатор могут предсказывать еще и вероятность правильного решения.

Вероятность каждого из классов — чуть более информативная величина, так как мы можем в зависимости от порога по-разному приписывать метки класса, используя одни и те же вероятности.

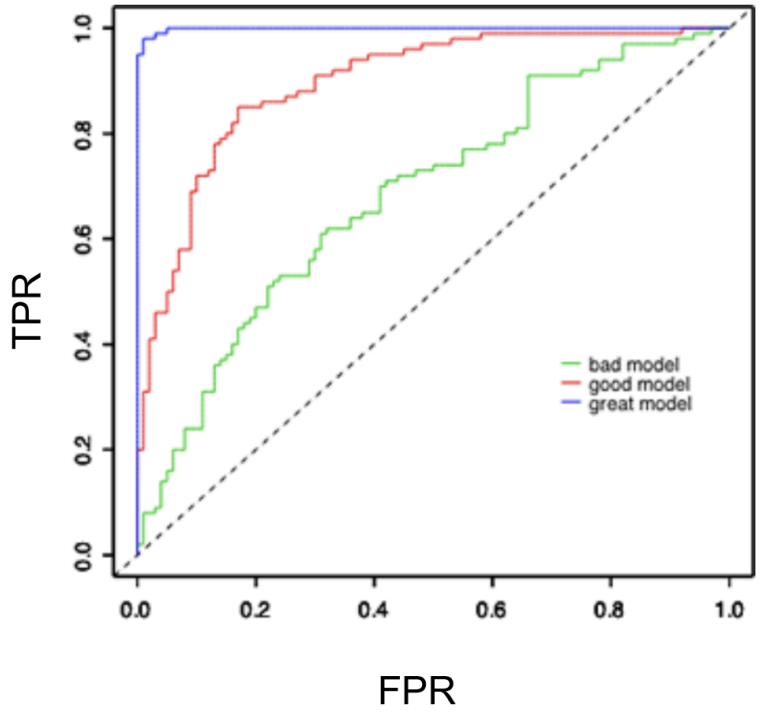
Чтобы сравнивать классификаторы, которые используют не только метки класса, но и вероятности, мы можем воспользоваться **ROC-кривой**.

ROC-кривая (receiver operating characteristic) — кривая, которая показывает нам в осях FPR и TPR (False Positive Rate и True Positive Rate), насколько хорошо наш классификатор работает:

		Actual Class	
		Yes	No
Predicted Class	Yes	**T**rue **P**ositive	**F**alse **P**ositive
	No	**F**alse **N**egative	**T**rue **N**egative

$$TPR = \frac{True\ positives}{True\ positives + False\ negatives}$$

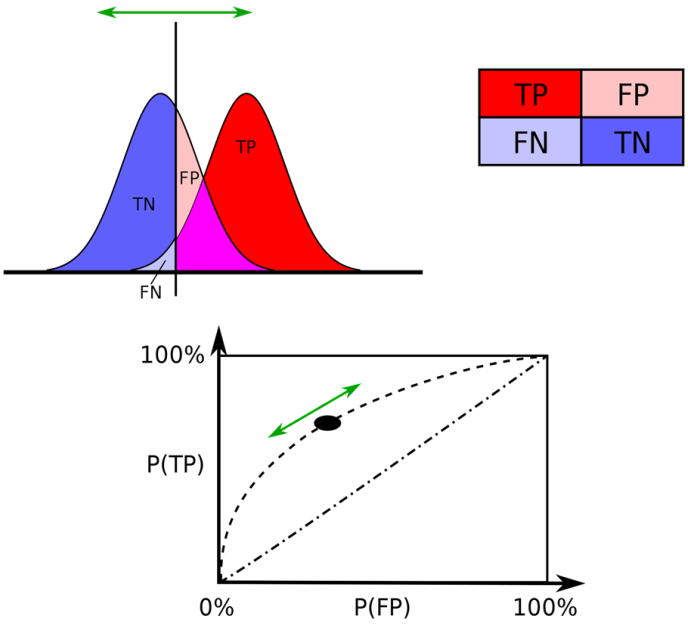
$$FPR = \frac{False\ positives}{False\ positives + True\ negatives}.$$



В зависимости от порога у нас по-разному будут стоять метки, а значит, будут разные FPR и TPR.

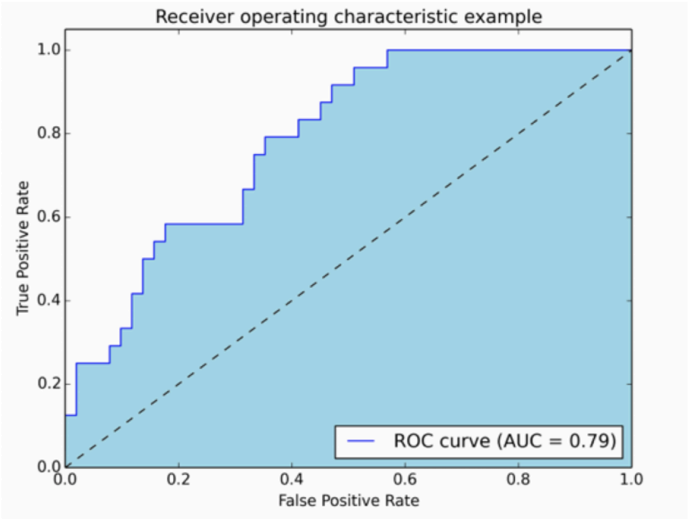
ROC-кривую получают следующим образом: варьируют порог и получают вероятности на положительные и отрицательные классы.

Как оценить такой классификатор? Нужно посчитать площадь под кривой.



Меняем порог, каждый раз считаем TPR и FPR нашей выборки. Получаем точку на графике. Получив всю кривую, можно увидеть, насколько классификатор хорошо работает.

Для самого плохого случая классификатор будет работать случайным образом, TPR и FPR будут расти одновременно и одинаково. Получим диагональную прямую.



Площадь всего данного квадрата, на котором расположена ROC-кривая, равна 1. Площадь под случайным классификатором (под диагональной прямой) равна 0,5.

Таким образом, площадь позволяет оценить качество классификатора независимо от того, какой порог мы выбрали, чтобы выставлять метки класса. Чем быстрее растет TPR и медленнее при этом FPR, тем лучше наш классификатор, потому что он позволяет все точки отнести к правильному классу. Для идеального классификатора площадь равна 1.

NB. Ни для какого классификатора площадь под ROC-кривой не должна быть меньше 0,5. Если площадь меньше 0,5, значит, где-то в расчетах меток класса закралась ошибка. Если площадь меньше 0,5, можно поменять местами метки классов. Тогда график отразится относительно диагонали, площадь под кривой станет больше 0,5.

Площадь под ROC-кривой называется **ROC-AUC** (area under curve).

С помощью оценки TPR и FPR можно найти оптимальное значение порога для нашего решения задачи. Почитать об этом больше можно в материале по ссылке.

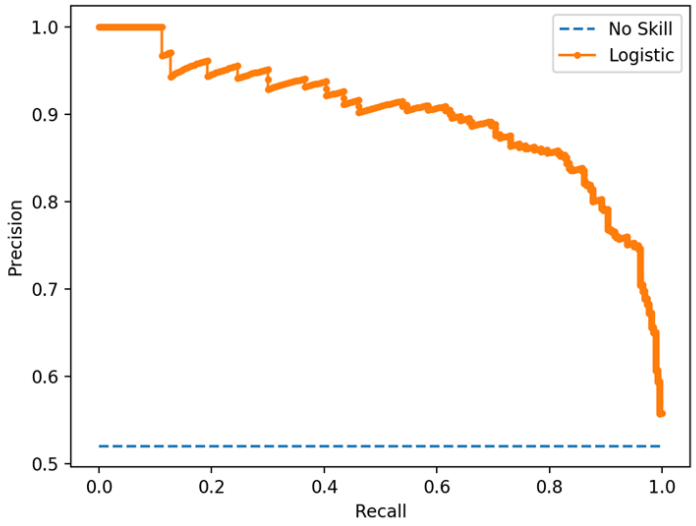
Precision-Recall кривая

Precision и Recall тоже зависят от порога

$$Precision = \frac{tp}{tp+fp}$$

$$Recall = \frac{tp}{tp+fn}$$

Можно тоже построить кривую в зависимости от порога:



Precision-Recall AUC тоже лежит в диапазоне (0, 1). С помощью этой кривой также можно подобрать оптимальное значение порога.

Подробнее про Precision-Recall AUC можно почитать в материале по ссылке.

Precision-Recall и ROC-кривая говорят про целевой класс. Если выборка не сбалансирована, нужно быть очень осторожным.

Мультиклассовые метрики

В случае нескольких классов придется усреднять. Усреднять можно по-разному:

average	Precision	Recall	F_beta
"micro"	$P(y, \hat{y})$	$R(y, \hat{y})$	$F_\beta(y, \hat{y})$
"samples"	$\frac{1}{\lvert S\rvert}\sum_{s\in S} P(y_s, \hat{y}_s)$	$\frac{1}{\lvert S\rvert}\sum_{s\in S} R(y_s, \hat{y}_s)$	$\frac{1}{\lvert S\rvert}\sum_{s\in S} F_\beta(y_s, \hat{y}_s)$
"macro"	$\frac{1}{\lvert L\rvert}\sum_{l\in L} P(y_l, \hat{y}_l)$	$\frac{1}{\lvert L\rvert}\sum_{l\in L} R(y_l, \hat{y}_l)$	$\frac{1}{\lvert L\rvert}\sum_{l\in L} F_\beta(y_l, \hat{y}_l)$
"weighted"	$\frac{1}{\sum_{l\in L}\lvert\hat{y}_l\rvert}\sum_{l\in L}\lvert\hat{y}_l\rvert P(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l\in L}\lvert\hat{y}_l\rvert}\sum_{l\in L}\lvert\hat{y}_l\rvert R(y_l, \hat{y}_l)$	$\frac{1}{\sum_{l\in L}\lvert\hat{y}_l\rvert}\sum_{l\in L}\lvert\hat{y}_l\rvert F_\beta(y_l, \hat{y}_l)$

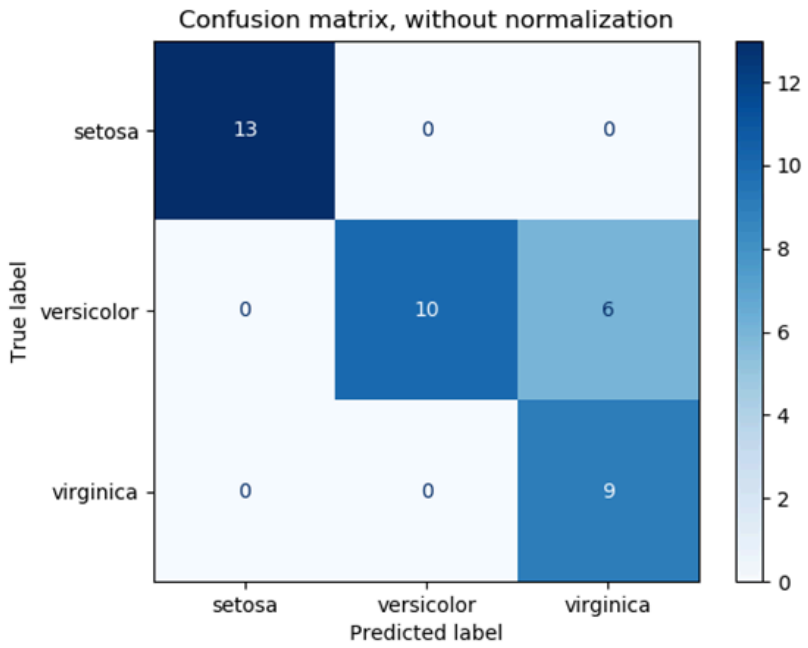
Источник таблицы: документация

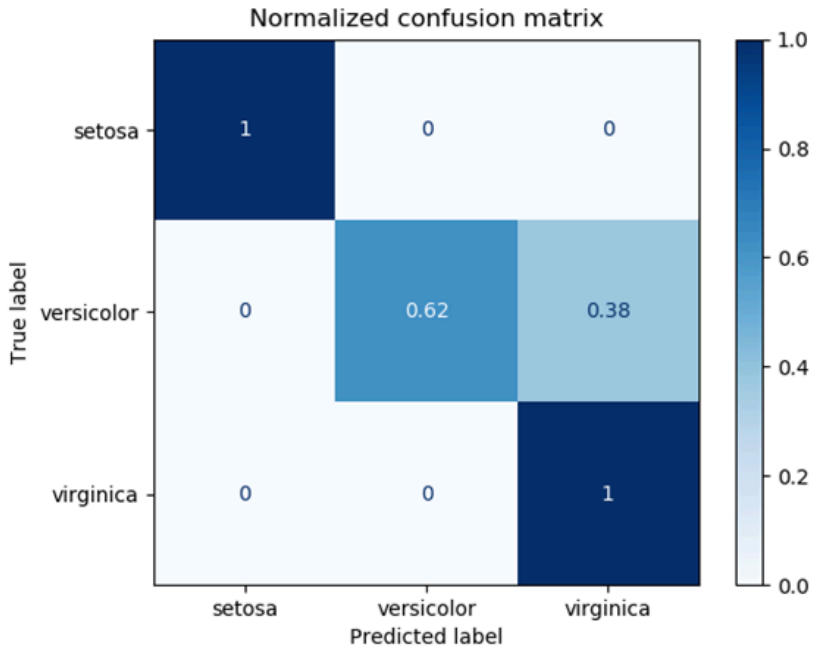
Выбор усреднения зависит от задачи.

Например, можно взять метод «один против всех», построить множество ROC-кривых, и усреднить под ними площадь.

Матрица ошибок

Матрица ошибок или Confusion matrix — матрица, у которой по одной оси лежат истинные метки классов, по другой — предсказанные метки классов.





Задача: мы хотим предсказать метки классов наиболее точным образом. То есть чтобы матрица была диагональной.

В ненормированной матрице на пересечении получаем число предсказанных объектов. В нормированной матрице — долю от всего числа объектов.

Если матрица получается недиагональной, можно увидеть, какие метки классов мы путаем.

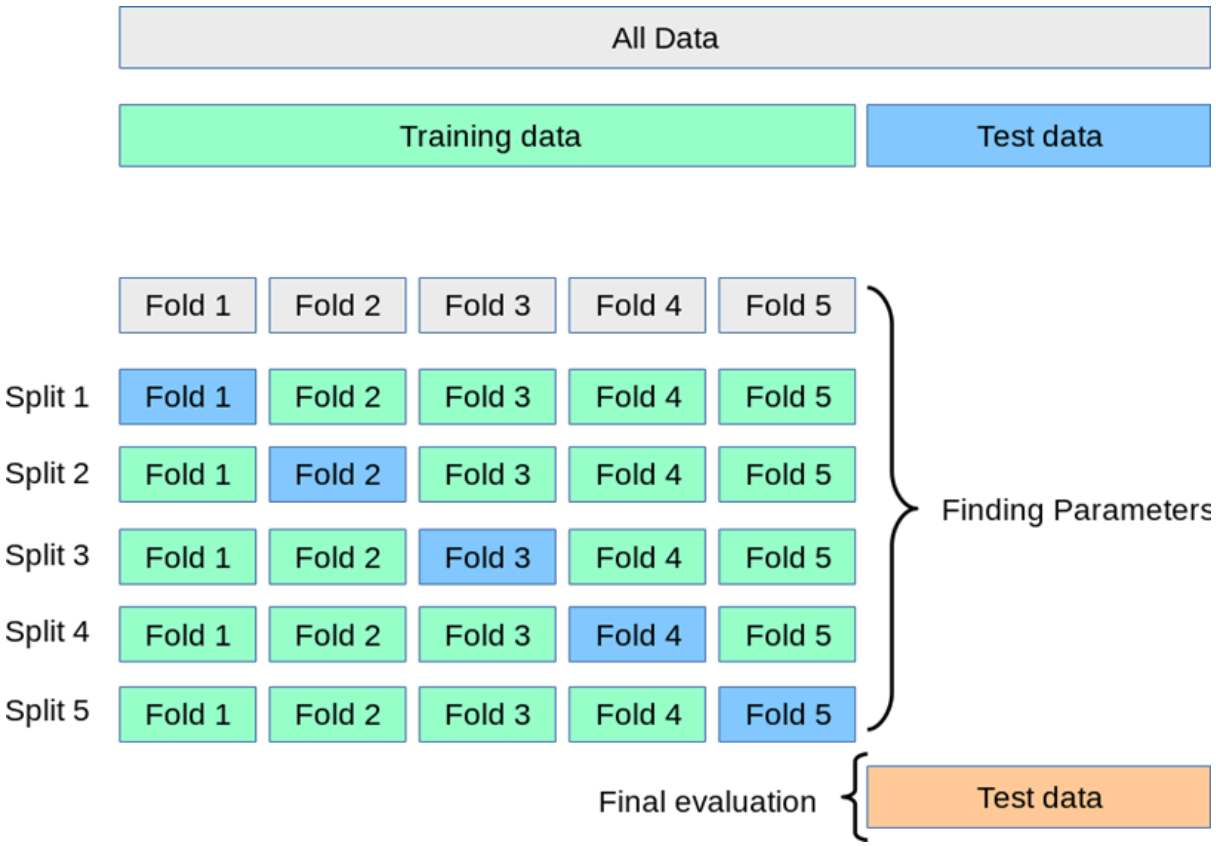
Матрица ошибок позволяет визуально оценить качество классификации.

4. Методы кросс-валидации

Теперь, используя все изученные метрики качества, можно оценить качество модели.

Метод кросс-валидации позволяет оценить качество модели и подобрать некоторые гиперпараметры.

Пусть у нас есть выборка — все наши данные. Поделим все наши данные на две части — train и test:



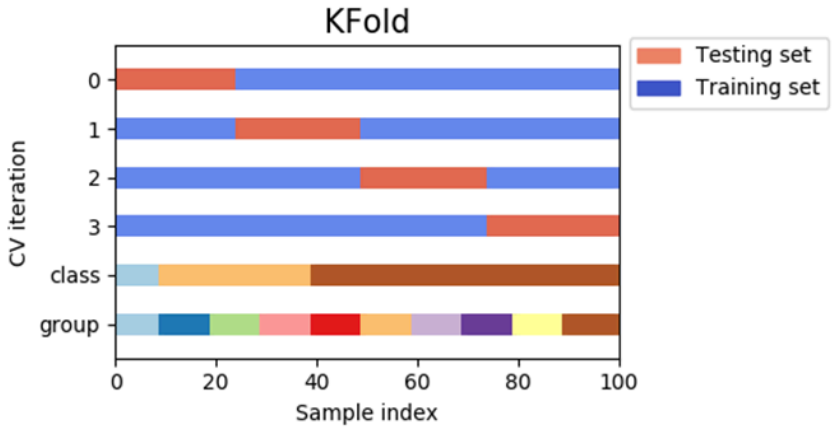
Данные части Test отложим в сторону до того, пока не будет готова финальная модель. Test часть будет финальной точкой проверки адекватности модели.

Часть Train делим на две части: train и валидацию (на картинке синие — валидационные фолды, зеленые — трейновые фолды).

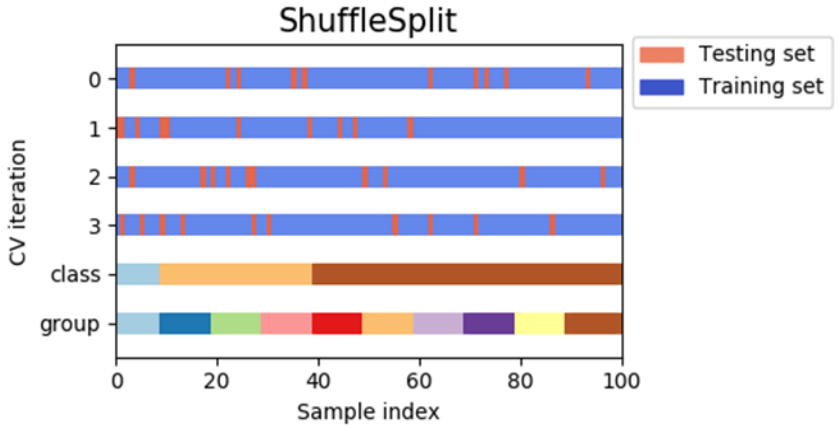
Обучаем модель на зеленых фолдах, валидируем на синих. Делаем это случайным образом несколько раз. Оцениваем качество, получаем финальный параметр, подобранный по кросс-валидации, и отправляем в часть Test.

Кросс-валидация может быть достаточно специфичной.

Мы можем бить на фолды не случайным образом:



И случайным образом:

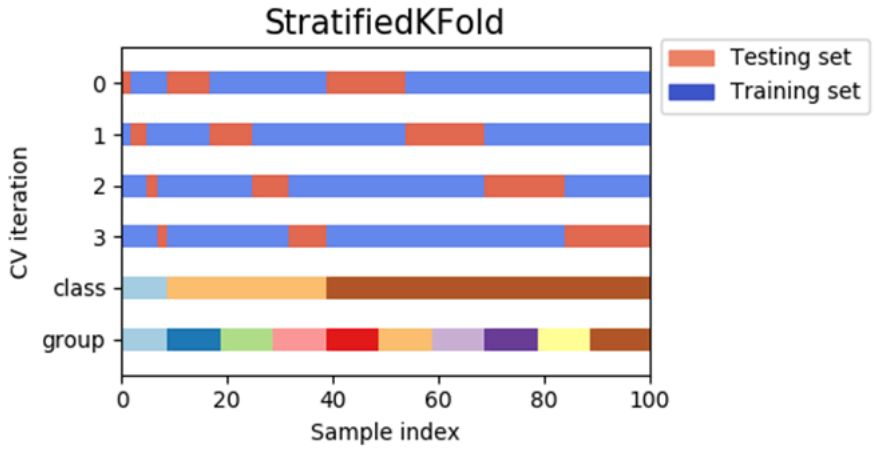


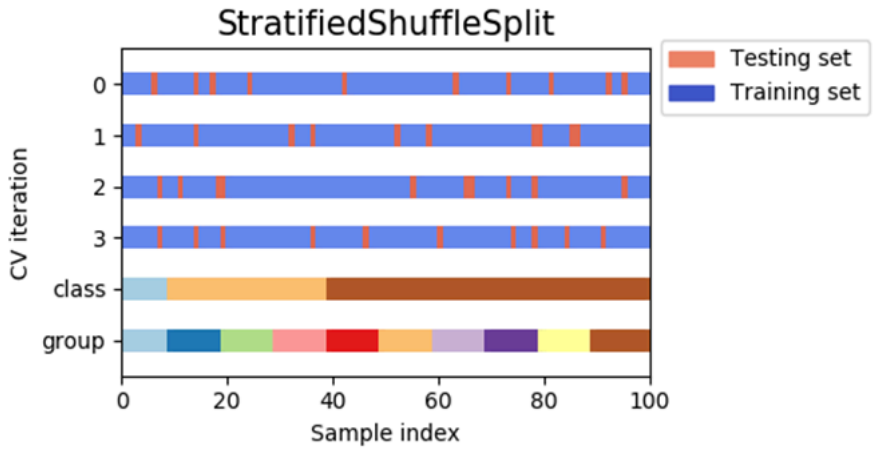
Можно вообще взять крайний случай, когда модель обучается на всей выборке, а тестируется на одном объекте. Это работает на маленьком датасете, называется **Leave One Out (LOO)** (выброс один объект).

При использовании кросс-валидации точки могут быть неравномерно распределены по классам.

Чтобы сбалансировать классы, можно использовать **стратификацию**: из каждого из итоговых классов выбираем по кусочку и соединяем их в одну кучу. Стратификация позволяет нам учитывать, что данные должны обладать тем же самым распределением, той же частотой, как и в исходной выборке. Дисбаланс классов, таким образом, отображается и на валидации.

Стратификацию можно проводить по значениям признаков.





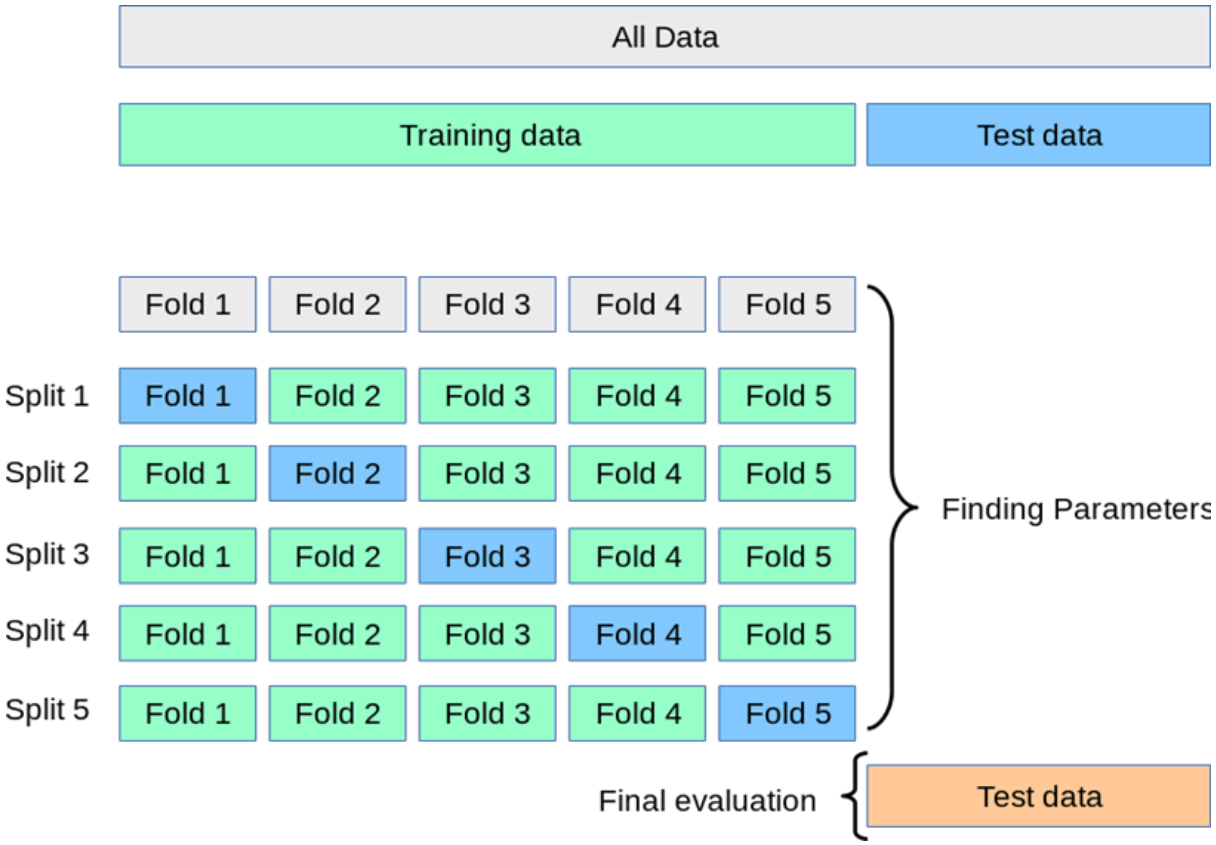
Нужно быть аккуратными с кросс-валидацией в случае с данными, в которых есть некоторая упорядоченность. Например, временные ряды, и данные в будущем некоторым образом обусловлены на данные в прошлом.

Важно! Нельзя валидироваться на прошлом, а обучаться на будущем.

Pipeline валидации в данном случае должен учитывать этот порядок.

5. Cross-validation riddle

Кросс-валидация — разделение обучающих данных на n частей. Для обучения берем (n-1) часть. Валидируемся на последней части:



Когда может пригодиться кросс-валидация? Например, когда нет тестового датасета, но нужно проверить, что модель работает корректно. Или когда данные очень различны. В этом случае модель может обучаться на разных подмножествах по-разному.

Напишем кросс-валидацию, будем решать задачу классификации. Предоставленные данные и классы будут распределены случайным образом.

Будем использовать датасет, который не будет нести никакого смысла:

```
# Some imports...
```

```
import numpy as np
import matplotlib.pyplot as plt


from sklearn.svm import LinearSVC
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import accuracy_score
```

Решаем задачу бинарной классификации некоторых случайных объектов:

```
class FeatureSelector:
    def __init__(self, num_features):
        self.n = num_features # number of best features to select


    def fit(self, X, y):
        # Select features that describe the targets best, i.e. have
        # highest correlation with them:
        covariance = ((X - X.mean(axis=0)) * (y[:,np.newaxis] -
y.mean())).mean(axis=0)
        self.best_feature_ids = np.argsort(np.abs(covariance))[-self.n:]


    def transform(self, X):
        return X[:,self.best_feature_ids]


    def fit_transform(self, X, y):
        self.fit(X, y)
        return self.transform(X)
```

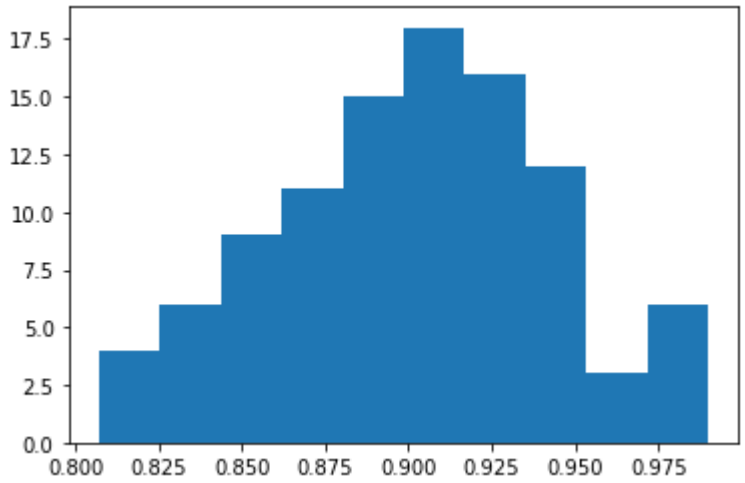
```
num_features_total = 1000

num_features_best = 100


N = 100


# Dataset generation

X = np.random.normal(size=(N, num_features_total))

y = np.random.randint(2, size=N)


# Feature selection:

X_best = FeatureSelector(num_features_best).fit_transform(X, y)


# Simple classification model

model = LinearSVC()


# Estimating accuracy using cross-validation:

cv_score = cross_val_score(model, X_best, y, scoring='accuracy', cv=10,
n_jobs=-1).mean()

print(f"CV score is {cv_score}")
```

В классе FeatureSelector выбираем, какое количество характеристик наиболее сильно коррелированы с вектором ответов. В результате отбираем 100 лучших features для таргетного вектора. Будем обучать модель не на всех features, а только на некотором подмножестве features, про которое мы знаем, что оно скоррелировано с ответом.

Получаем точность на кросс-валидации 87%, очень хорошую. Может ли быть такая высокая точность при заданных условиях датасета? Нет, не может. У нас *x* и *y* — случайные величины, которые никак друг с другом не связаны. Ожидаемо, что кросс-валидация должна была просто угадывать, и точность должна быть 0,5.

Рассмотрим функцию experiment:

```
num_features_total = 1000
num_features_best = 100


N = 100
def experiment():
    # Dataset generation
    X = np.random.normal(size=(N, num_features_total))
    y = np.random.randint(2, size=N)


    # Feature selection:
    X_best = FeatureSelector(num_features_best).fit_transform(X, y)


    # Simple classification model
    model = LinearSVC()


  # Estimatin accuracy using cross-validation:
  return cross_val_score(model, X_best, y, scoring='accuracy', cv=10,
n_jobs=-1).mean()


results = [experiment() for _ in range(100)]
plt.hist(results, bins=10);
```



Точность классификации будет лежать на промежутке от 0,8 до 1.

На первых шагах мы дали нашей модели намного больше информации, чем мы думаем.

Подаем на вход данные в FeatureSelector и смотрим, насколько они скоррелированы с ответом, отбираем 100 лучших features. При кросс-валидации в нашу обучающую выборку попадает уже информация о тестовой части выборки. И так для каждого сплита кросс-валидации.

Сделаем теперь кросс-валидацию правильно:

```
num_features_total = 1000

num_features_best = 100


N = 100

# Dataset generation

X = np.random.normal(size=(N, num_features_total))

y = np.random.randint(2, size=N)
```

Разбиваем данные на 10 фолдов:

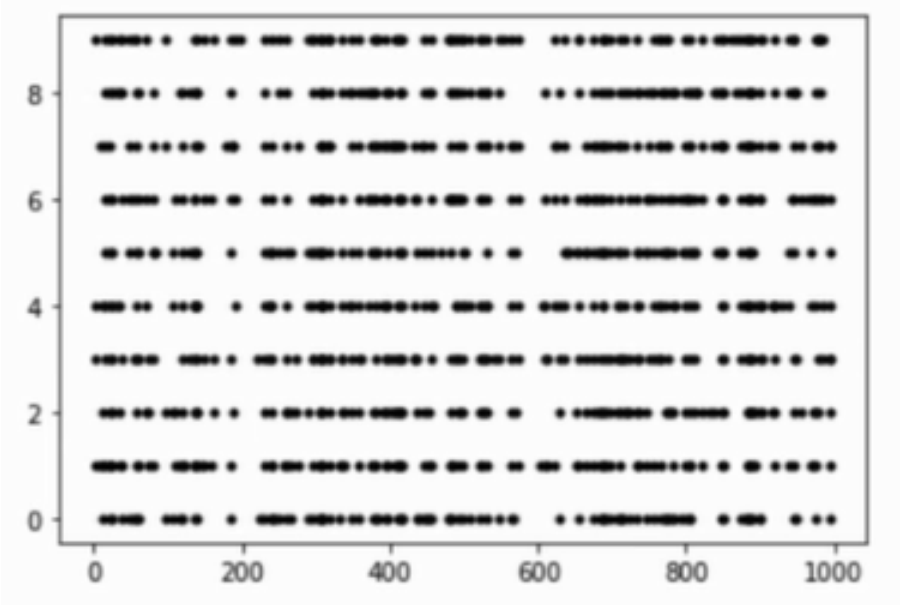
```
batches = np.random.permutation(N).reshape(10, -)
mask = np.ones(N, dtype=bool)


A = []
best_features = []
for b in batches:
      mask[b] = False #exclude samples


      selector = FeatureSelector(num_features_best)
      X-best = selector.fit_transform(X[mask], y[mask])
      model = LinearSVC()
      model.fit(X_best, y[mask])
      prediction = model.predict(X[b,:][:,  self.best_feature_ids])
      accuracy = sum(prediction==y[b])/len(y[b])
      mask[b] = True # fix the mask
      A.append(accuracy)
      best_features.append(selector.best_feature_ids)


cv_score = np.mean(A)
print(cv_score)
```

Мы маскируем один фолд, на остальных выделяем лучшие features. Обучаем модель, и посмотрим теперь на точность предсказаний.

Получаем точность 0,4.

Посмотрим, как распределены наши фолды:

```
for i, best_feature in enumerate(best_features):
# print(len(best_feature))
      plt.plot(best_feature, best_feature*0 + i, ".", color="black")
```



При каждом нажатии видим, что фолды на каждой итерации разные. А в задаче с полученной аномально высокой точностью набор features на каждом шаге был одинаков.

Напишем теперь кросс-валидацию проще:

```
from sklearn.pipeline import Pipeline
```

```
fs = FeatureSelectore(num_features_best)
estimator = LinearSVC()
steps = [
      ('selector', fs).
```

```
      ('estimator', estimator)
]
pipeline = Pipeline(steps)
```

```
cv_score = cross_val_score(pipeline, X, y, scoring='accuracy', cv=10,
n_jobs=-1).mean()
cv_score
```

>>> 0.3999999999997

Pipeline позволяет собрать модель, которая будет состоять не только из самой модели, но и из шагов препроцессинга. Теперь мы не можем ошибиться, когда делаем кросс-валидацию, и точность будет приближена к реальной ситуации о случайных данных.

Дополнительные материалы для самостоятельного изучения

1. https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe
2. https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5
3. https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/
4. https://scikit-learn.org/stable/modules/model_evaluation.html#from-binary-to-multiclass-and-multilabel