Основи роботи в бібліотеці Scikit-Learn Python

Окремі глави з книги Дж. ВандерПлас. Python для сложных задач: наука о данных и машинное обучение. СПб.: Питер, 2018.

1 Представлення даних в Scikit-Learn

Бібліотека Scikit-Learn Python надає ефективні реалізації багатьох поширених алгоритмів машинного навчання. Перевага пакету Scikit-Learn в одноманітності програмного інтерфейсу, тобто, розібравшись в основах використання і синтаксисі одного типу моделей в Scikit-Learn, можна легко перейти до іншої моделі або алгоритму. Пакет Scikit-Learn відрізняє також зручна і детальна онлайн-документація.

Дані, які використовуються в Scikit-Learn, краще всього представляти у вигляді таблиць.

1.1 Дані у вигляді таблиці ознак

Найпростіша таблиця - двовимірний масив, в якому рядки представляють окремі елементи набору даних, а стовпці - ознаки або атрибути, пов'язані з кожним з цих елементів. Масив також називається **матрицею ознак**. Наприклад, розглянемо набір даних Iris, проаналізований Рональдом Фішером в 1936 році

(`https://en.wikipedia.org/wik /Iris_flower_data_set`).

Завантажимо його у вигляді об'єкта DataFrame бібліотеки Pandas за допомогою бібліотеки Seaborn:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

Отримаємо таблицю:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Кожен рядок даних відноситься до однієї з виміряних квіток. Будемо називати рядки цієї матриці **прикладами** (samples), вони завжди відповідають окремим об'єктам. Кількість рядків нехай дорівнює `n_samples`.

Кожен стовпець даних відноситься до кількісного показника, що описує приклад. Будемо називати стовпці матриці **ознаками** (features), а кількість стовпців вважати рівним `n_features`. Значення ознак зазвичай представляються дійсними числами, але в деяких випадках ознаки можуть бути булевими або мати дискретні значення.

Матриця ознак найчастіше зберігається в масиві NumPy або об'єкті DataFrame бібліотеки Pandas, хоча деякі моделі бібліотеки Scikit-Learn допускають використання також розріджених матриць з бібліотеки SciPy.

1.2 Цільовий масив

Крім матриці ознак x, часто маємо цільовий масив або масив міток, який прийнято позначати y. Цільовий масив представляє величину, значення якої ми хочемо спрогнозувати на основі наявних даних. Це залежна змінна (dependent variable).

Цільовий масив зазвичай одновимірний, довжиною `n_samples`. Його зберігають в масиві NumPy або об'єкті Series бібліотеки Pandas. Значення цільового масиву можуть бути неперервними числовими або дискретними класами / мітками. Слід зазначити, що окремі засоби бібліотеки Scikit-Learn можуть оперувати декількома цільовими величинами у вигляді двовимірного цільового масиву `[n_samples, n_targets]`.

Наприклад, для набору даних Iris залежною змінною є клас квітки, і цільовий масив - це стовпець species у наведеній вище таблиці. Використаємо бібліотеку Seaborn для візуалізації даних:

```
sns.set()
sns.pairplot(iris, hue='species', height=1.5);
```

Винесемо матрицю ознак і цільовий масив з об'єкта DataFrame, для зручності подальшого використання. Зробити це можна за допомогою операцій об'єкта DataFrame бібліотеки Pandas:

```
In: X_iris = iris.drop('species', axis=1)
X_iris.shape

Out: (150, 4)

In: y_iris = iris['species']
y_iris.shape

Out: (150, )
```

2 API статистичного оцінювання бібліотеки Scikit-Learn

У документації з API Scikit-Learn вказано, що він ґрунтується на наступних принципах:

- одноманітність - інтерфейс всіх об'єктів ідентичний і заснований на обмеженому наборі методів, причому документація теж однакова;
- контроль - видимість всіх заданих значень параметрів як відкритих атрибутів;
- обмежена ієрархія об'єктів - класи мови Python використовуються тільки для алгоритмів; набори даних представлені в стандартних форматах (масиви NumPy, об'єкти DataFrame бібліотеки Pandas, розріджені матриці бібліотеки SciPy), а для імен параметрів використовуються стандартні рядки мови Python;
- об'єднання - задачі машинного навчання виражаються у вигляді послідовностей алгоритмів нижчого рівня бібліотеки Scikit-Learn;
- розумні значення за замовчуванням - бібліотека задає для параметрів моделей відповідні значення за замовчуванням.

Найчастіше використання **API статистичного оцінювання бібліотеки Scikit-Learn включає наступні кроки**:

1. Вибір класу моделі за допомогою імпорту відповідного класу з бібліотеки Scikit-Learn.
2. Вибір гіперпараметрів моделі шляхом створення екземпляра цього класу з відповідними значеннями.
3. Компонування даних в матрицю ознак і цільовий вектор.
4. Навчання моделі на даних за допомогою виклику методу `fit()` екземпляра моделі.
5. Застосування моделі до нових даних:
    - в задачі навчання з учителем значення цільової змінної для невідомих даних зазвичай прогнозують за допомогою методу `predict()`;
    - в задачі навчання без вчителя виконується перетворення властивостей даних або виведення їх значень за допомогою методів `transform()` або `predict()`.

Розглянемо кілька прикладів.

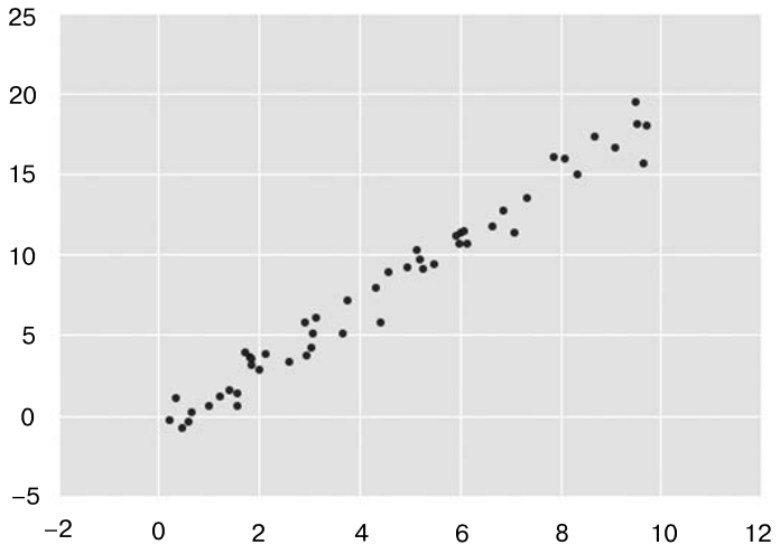

Рис. 1: Дані для лінійної регресії

2.1 Приклад навчання з учителем: лінійна регресія

Розглянемо лінійну регресію, коли потрібно підібрати розділяючу пряму для даних виду (x, y). Візьмемо наступні дані (рис. 1):

```
import matplotlib.pyplot as plt
import numpy as np
rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```

Розглянемо детально всі кроки використання API статистичного оцінювання бібліотеки Scikit-Learn.

1. Вибір класу моделі.

   Кожен клас моделі в бібліотеці Scikit-Learn представлено відповідним класом мови Python. Для розрахунку моделі простої лінійної регресії можна імпортувати клас лінійної регресії:

   ```
   from sklearn.linear_model import LinearRegression
   ```

   Існують також й інші, більш загальні моделі лінійної регресії (див. документацію до модуля `sklearn.linear_model` (`http://scikit-learn.org/stable/modules/linear_model.html`)

2. Вибір гіперпараметрів моделі.

Після вибору класу моделі є можливість вибрати гіперпараметри цієї моделі.

Гіперпараметрами називаються параметри моделі, які задаються до навчання моделі на даних.

Так, залежно від класу моделі може знадобитися відповісти на одне або кілька наступних питань.

- Чи хочемо ми виконати підбір зсуву прямої (тобто точки перетину з віссю координат)?
- Чи хочемо ми нормалізувати модель?
- Чи хочемо ми зробити модель більш гнучкою, виконавши попередню обробку ознак?
- Яка ступінь регуляризації повинна бути у нашій моделі?
- Скільки компонент моделі ми хотіли б використовувати?

Вибір гіперпараметрів в бібліотеці Scikit-Learn здійснюється шляхом передачі значень при створенні екземпляра моделі. Важливий момент: екземпляр моделі - це не те саме, що клас моделі.

Наприклад, створимо екземпляр класу `LinearRegression` і вкажемо за допомогою гіперпараметра `fit_intercept`, що буде виконуватися підбір точки перетину з віссю координат:

```
In: model = LinearRegression(fit_intercept=True)
model
```

```
Out: LinearRegression(copy_X=True, fit_intercept=True,
           n_jobs=None, normalize=False)
```

Зауважимо, що при створенні екземпляра моделі виконується тільки збереження значень гіперпараметрів. Поки ми не застосовували модель до жодних даних.

В бібліотеці Scikit-Learn відокремлено вибір моделі і застосування цієї моделі до конкретних даних.

3. Формування матриць ознак і цільового вектора.

   В даному прикладі цільова змінна y вже має потрібний вид - це масив довжиною `n_samples`. Проте, на основі даних x потрібно зробити матрицю розмірності `[n_samples, n_features]`:

   ```
   X = x[:, np.newaxis]
   ```

   Тепер масив X з навчальними даними дорівнює:

```
array([[3.74540119],
       [9.50714306],
       [7.31993942],
       [5.98658484],
       [1.5601864 ],
       [1.5599452 ],
       ...,
       [1.84854456]])
```

4. Навчання моделі.

   Виконується за допомогою методу `fit()` моделі:

   ```
   In: model.fit(X, y)

   Out: LinearRegression(copy_X=True, fit_intercept=True,
          n_jobs=None, normalize=False)
   ```

   Команда `fit()` викликає виконання множини обчислень, в залежності від моделі. Результати цих обчислень зберігаються в **атрибутах моделі**, які доступні для перегляду користувачем. Традиційно у бібліотеці Scikit-Learn всі параметри моделі, отримані в процесі виконання команди `fit()`, містять в кінці назви знак підкреслення. Наприклад, в розглянутій лінійній моделі атрибутами є:

   ```
   In: model.coef_

   Out: array([1.9776566])

   In: model.intercept_

   Out: -0.9033107
   ```

   Ці два параметри представляють собою кутовий коефіцієнт і точку перетину з віссю координат для лінійної апроксимації навчальних даних. Можна помітити, що знайдені оцінки параметрів моделі дуже близькі до відомих реальних значень параметрів, що дорівнюють 2 і -1.

   Про похибки у внутрішніх параметрах моделі можна дізнатися зі статистичного моделювання, використовуючи засоби пакету StatsModels мови Python (http://statsmodels.sourceforge.net/).

5. Прогнозування значень цільової змінної для нових даних.

   Після навчання моделі головна задача машинного навчання з учителем полягає в обчисленні з її допомогою значень для нових даних, які не є частиною навчальної множини. Для цього використовується метод `predict()`. Розглянемо наступні тестові дані:

```
xtest = np.linspace(-1, 11)
```

Як і раніше, значення xtest потрібно перетворити на матрицю ознак `[n_samples, n_features]`, після чого можна подати їх на вхід моделі:

```
Xtest = xtest[:, np.newaxis]
ytest = model.predict(Xtest)
```

Виконаємо візуалізацію результатів, намалювавши спочатку графік вхідних даних, а потім навчену модель (рис. 2):

```
plt.scatter(x, y)
plt.plot(xtest, ytest)
```

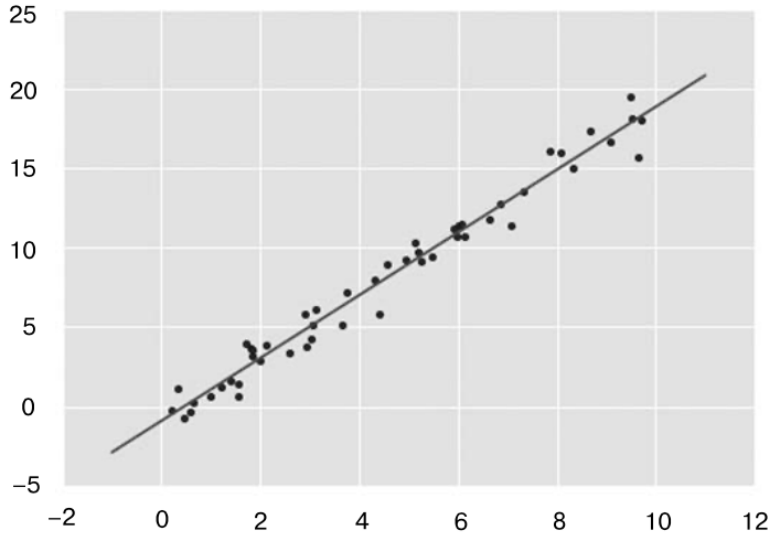

Рис. 2: Наближення даних за допомогою лінійної регресії

2.2 Приклад навчання з учителем: класифікація набору даних Iris

Розглянемо ще один приклад з набором даних Iris, в якому спробуємо дати відповідь на питання: як отримати прийнятний прогноз цільової змінної (міток) для інших даних, відмінних від тих, на яких навчалася модель?

Використаємо для цього просту модель, відому під назвою «Гаусівський наївний байесівський класифікатор». Ця модель не містить гіперпараметрів і має високу продуктивність.

Перевіримо роботу моделі на невідомих їй даних, тому необхідно розділити дані на **навчальну множину** і **тестову множину**. Для цього можна використати функцію `train_test_split`:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
    random_state=1)
```

Далі використаємо описані вище кроки для отримання прогнозу.

1. Вибір класу моделі:

```
from sklearn.naive_bayes import GaussianNB
```

2. Створення екземпляру моделі:

```
model = GaussianNB()
```

3. Навчання моделі на даних:

```
model.fit(Xtrain, ytrain)
```

4. Розрахунок прогнозу значень для нових даних:

```
y_model = model.predict(Xtest)
```

Щоб з'ясувати, яка частина спрогнозованих значень відповідає істинному значенню цільової змінної, використаємо `accuracy_score`:

```
In: from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

```
Out: 0.97368
```

Отримана точність перевищує 97%, тому для цього конкретного набору даних наївний алгоритм класифікації виявився ефективним.

Для того щоб краще зрозуміти, де наша модель помилилася, розраховують матрицю відмінностей (confusion matrix), в якій елементи поза головною діагоналлю показують кількість помилкових класифікацій за моделлю:

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

mat = confusion_matrix(ytest, y_model)

sns.heatmap(mat, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value');
```

2.3 Приклад навчання без учителя: зниження розмірності набору даних Iris

Як приклад задачі навчання без учителя розглянемо задачу зниження розмірності набору даних Iris з метою спрощення його візуалізації. Дані Iris, як відомо, є чотиривимірними: кожний приклад характеризується чотирма ознаками.

Задача зниження розмірності полягає у пошуку відповідного представлення більш низької розмірності, яке зберігає суттєві ознаки даних. Найчастіше зниження розмірності використовується для полегшення візуалізації даних.

Будемо використовувати метод головних компонент (principal component analysis, PCA), що представляє собою швидкий лінійний метод зниження розмірності. Нехай модель має повертати двовимірне представлення даних, тобто дві компоненти.

Виконаємо описані вище кроки побудови і оцінювання моделі.

1. Вибір класу моделі:

```
from sklearn.decomposition import PCA
```

2. Створення екземпляру моделі з гіперпараметрами:

```
model = PCA(n_components=2)
```

3. Навчання моделі на даних:

```
In: model.fit(X_iris)

Out: PCA(copy=True, iterated_power='auto', n_components=2,
    random_state=None, svd_solver='auto', tol=0.0, whiten=False)
```

4. Перетворення даних на двовимірні:

```
X_2D = model.transform(X_iris)
```

Побудуємо графік отриманих результатів (рис. 3): вставимо результати у початковий об'єкт `DataFrame` Iris і використаємо функцію `lmplot` для відображення результатів:

```
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False)
```

Бачимо, що у двовимірному представленні види квітів чітко розділені, хоча алгоритму PCA не відомі цільові значення видів квітів. Це показує, що навіть відносно проста класифікація на цьому наборі даних скоріш за все покаже досить добрі результати.

2.4 Навчання без вчителя: кластеризація набору даних Iris

Виконаємо кластеризацию набору даних Iris. Алгоритм кластеризації намагається виділити групи даних незалежно від міток. В даному прикладі використаємо потужний алгоритм кластеризації «Суміш гаусових розподілів (Gaussian mixture model, GMM)». Метод GMM полягає в спробі моделювання даних у вигляді набору гаусових скупчень.

Для навчання суміші гаусових розподілів використаємо визначені вище кроки.

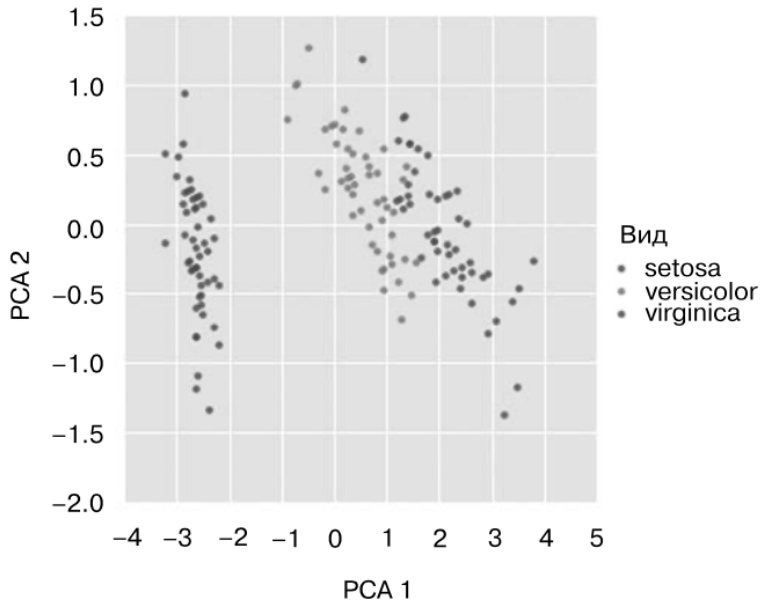

Рис. 3: Проекція даних набору Iris на двовимірний простір

1. Вибір класу моделі:

```
from sklearn.mixture import GaussianMixture
```

2. Створення екземпляру моделі з гіперпараметрами:

```
model = GaussianMixture(n_components=3, covariance_type='full')
```

3. Навчання моделі на даних:

```
In: model.fit(X_iris)
```

```
Out: GaussianMixture(covariance_type='full',
       init_params='kmeans', max_iter=100,
       means_init=None, n_components=3, n_init=1,
       precisions_init=None, random_state=None,
       reg_covar=1e-06, tol=0.001, verbose=0,
       verbose_interval=10, warm_start=False, weights_init=None)
```

4. Визначення міток кластерів:

```
y_gmm = model.predict(X_iris)
```

Додамо стовпець `cluster` в `DataFrame` Iris і використаємо бібліотеку Seaborn для побудови графіків результатів (рис. 4):

```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```

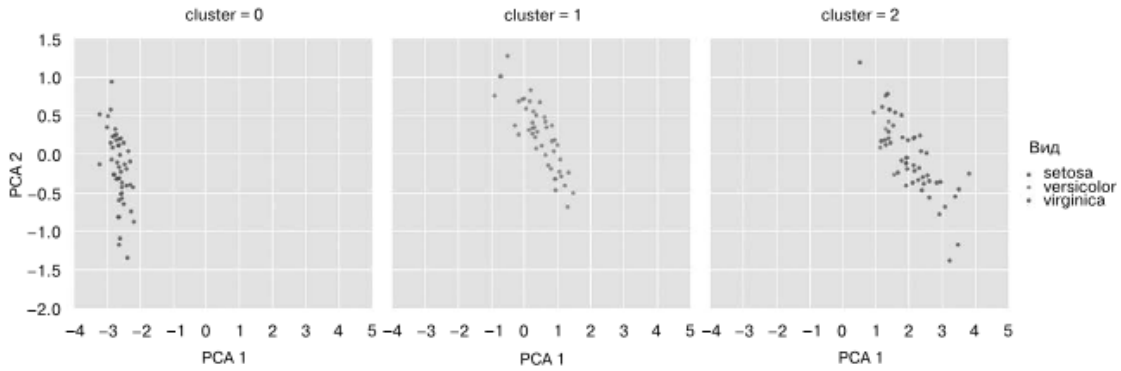

Рис. 4: Кластеризація даних Iris методом Gaussian mixture model

Виконавши кластеризацію і розподіливши дані за номерами кластерів, бачимо, що алгоритм Gaussian mixture model показав дуже добрі результати: один з класів повністю виділено в кластер 0, щоправда, невелика кількість прикладів двох інших класів дещо змішалися між собою. Таким чином, прикладів без учителя (без значень міток) достатньо для автоматичного розпізнавання різних різновидів квіток за допомогою алгоритму кластеризації Gaussian mixture model.

Подібний алгоритм може допомагати спеціалістам з предметної області виявити зв'язок між досліджуваними об'єктами.

3 Перевірка моделі. Оцінювання якості моделі

Основні кроки використання моделей машинного навчання з вчителем включають:

1. Вибір класу моделі.
2. Вибір гіперпараметрів моделі.
3. Навчання моделі на даних.
4. Застосування моделі до прогнозування на нових даних.

Вибір моделі та гіперпараметрів - це найбільш важливі кроки для ефективного використання цих методів. Для оптимального їх вибору необхідно виконати перевірку того, що конкретна модель з конкретними гіперпараметрами добре апроксимує конкретні дані.

Розглянемо спочатку **поганий підхід до перевірки моделі** на прикладі набору даних Iris. Завантажимо дані:

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
```

Виберемо метод класифікації *k найближчих сусідів* з гіперпараметром `n_neighbors=1`. Згідно з цією моделлю, мітка для невідомої точки розраховується такою як у найближчої до неї точки з навчальної вибірки.

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=1)
```

Навчимо цю модель і спрогнозуємо на її основі мітки для відомих даних. Обчислимо відсоток правильно класифікованих прикладів:

```
model.fit(X, y)
y_model = model.predict(X)

from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
```

Отримаємо в результаті 100% точність класифікації. Проте, це неправильна оцінка точності. 100% отримано, оскільки навчання і оцінка моделі виконувалися на одних і тих самих даних.

Правильний підхід до перевірки моделі базується на використанні **відкладених наборів даних** (holdout sets), а саме на **відокремленні перевірочних данних від навчальних**. Використовуючи функцію

`train_test_split`, виконується розбиття всіх наявних даних на дві підмножини:

- навчальна множина,
- перевірочна множина, на якій перевіряється якість моделі.

На прикладі даних Iris, нехай 50% віддамо на навчальну множину і 50% на перевірочну:

```
from sklearn.model_selection import train_test_split

X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
    train_size=0.5)
```

Навчаємо модель на одному наборі даних (на X1):

```
model.fit(X1, y1)
```

Оцінюємо роботу моделі на іншому наборі (на X2):

```
y2_model = model.predict(X2)

accuracy_score(y2, y2_model)
```

Отримано, що метод найближчого сусіда показує точність $90,67\%$ на перевірочному наборі даних Iris. Перевірочний набір даних подібний до невідомих даних, оскільки модель "не бачила"їх раніше.

3.1 Перехресна перевірка моделі

Один з недоліків використання відокремленого набору даних для перевірки моделі - це *втрата частини даних* для навчання моделі. У попередньому прикладі половина набору даних не бере участі в навчанні моделі. Це неоптимально і може стати причиною проблем, особливо якщо початковий набір даних невеликий.

Перехресна перевірка (cross-validation) полягає у побудові послідовності класифікацій, в яких кожна підмножина даних використовується як в якості навчальної, так і перевірочної множин. Наприклад, розіб'ємо дані на два набори і по черзі використаємо кожний з них в якості перевірочного (*двохблочна перехресна перевірка*):

```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)
```

Отримано числа 0.96 і 0.9067 - це дві оцінки точності, які можна узагальнити взявши від них *середнє значення* для отримання кращої оцінки загальної якості роботи моделі.

У **загальному випадку**, якщо k - кількість блоків, на які розбиваються дані для перевірки, то використовується функція `cross_val_score`:

```
from sklearn.model_selection import cross_val_score
k=5
cross_val_score(model, X, y, cv=k)
```

Наприклад, у випадку п'ятиблочної перехресної перевірки дані розбиваємо на п'ять груп і по черзі використовуємо кожну з них для оцінки навчання моделі на інших 4/5 даних. В результаті отримаємо

```
array([0.9666667, 0.9666667, 0.9333333, 0.9333333, 1. ])
```

Повторення перевірки за різними підмножинами даних дає краще уявлення про якість роботи алгоритму.

Розглянемо граничний випадок, при якому кількість блоків дорівнює кількості точок даних, і навчаємо модель в кожній спробі на всіх точках, крім однієї. Такий тип перехресної перевірки відомий під назвою **перехресної перевірки за окремими об'єктами** (leave-one-out cross-validation):

```
from sklearn.model_selection import LeaveOneOut
scores = cross_val_score(model, X, y, cv=LeaveOneOut())
scores
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
```

```
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
0., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Перехресна перевірка за окремими об'єктами дає результати за всіма 150 спробами, оскільки множина даних Iris налічує 150 прикладів. Результат у кожній спробі показує чи правильним (1.0), чи неправильним (0.0) був прогноз. Взявши середнє значення `scores.mean()` цих результатів, отримаємо загальну оцінку частоти помилок, вона для даного прикладу дорівнює 0.96.

3.2 Модуль model-selection бібліотеки Scikit-Learn

Бібліотека Scikit-Learn реалізує багато схем перехресної перевірки, зручних в певних конкретних ситуаціях. Вони реалізовані за допомогою ітераторів в модулі `model_selection`.

4 Вибір моделі і гіперпараметрів

Дана тема є однією з найбільш важливих у машинному навчанні. Важливим є наступне питання: що робити, якщо на тестовій (перевірочній) множині даних побудована модель показує недостатньо гарні результати? Існує кілька можливих відповідей:

- використовувати більш складну модель;
- застосовувати менш складну модель;
- зібрати більше прикладів для навчання;
- зібрати більше даних для додавання нових ознак до кожного із заданих прикладів.

Виявляється, що в окремих випадках використання більш складної моделі призводить до гірших результатів. Також додавання нових прикладів до навчальної вибірки інколи не призводить до покращення результатів!

4.1 Компроміс між систематичною помилкою і дисперсією

Вибір «оптимальної моделі» полягає у відшуканні найкращого компромісу між систематичною помилкою (зміщенням, bias) та дисперсією.

Розглянемо рис. 5, на якому представлено два випадки наближення набору даних за допомогою регресійної моделі.

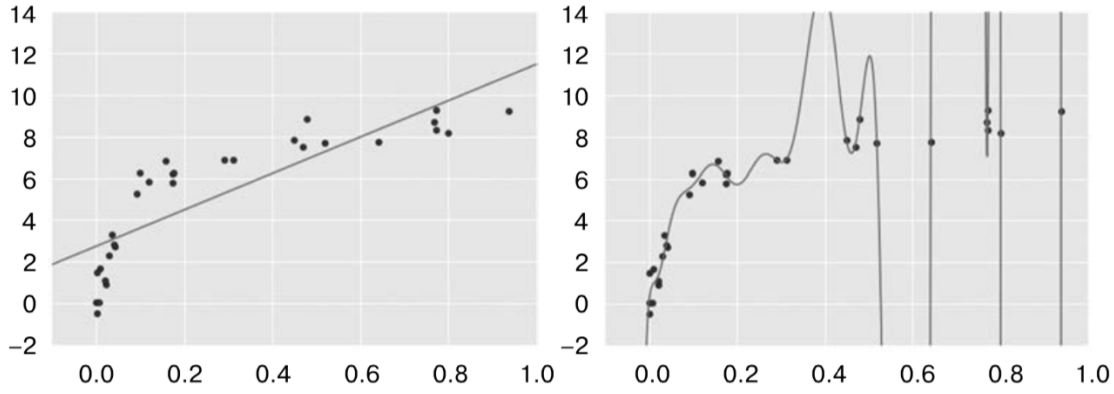

Рис. 5: Моделі регресії із значною систематичною помилкою і високою дисперсією

З рис.5 бачимо, що обидві моделі не дуже добре наближують дані. Модель, наведена зліва, намагається знайти прямолінійне наближення до даних. Але в силу того, що внутрішня структура даних складніша за пряму лінію, за допомогою прямолінійної моделі неможливо описати цей набір даних досить добре. Це приклад **недостатньо навченої (underfit)** моделі, тобто гнучкість моделі недостатня для задовільного врахування всіх ознак в даних. У цієї моделі **велика систематична помилка**.

Наведена справа модель описує дані многочленом високого ступеня. У цьому випадку модель досить гнучка, щоб практично ідеально відповідати всім нюансам даних, вона дуже точно описує навчальну множину. Подібну модель називають **перенавченою (overfit)** - гнучкість моделі така, що модель враховує не тільки початковий розподіл даних, а й випадкові помилки в них. Ця модель має **високу дисперсію**.

В якості оцінки ефективності моделі регресії використаємо R^2 - коефіцієнт детермінації або коефіцієнт змішаної кореляції. Він являє собою міру того, наскільки добре модель працює в порівнянні з простим середнім значенням цільових величин. $R^2 = 1$ означає ідеальний збіг прогнозу, а $R^2 = 0$ показує, що модель не є кращою за середнє значення даних, а від'ємні значення вказують на моделі, які працюють ще гірше. Для моделі зі значною систематичною помилкою (на рис. 5 зліва) маємо на навчальній множині значення $R^2 = 0.70$, на перевірочній множині - значення $R^2 = 0.74$. Для моделі зі значною дисперсією (на рис. 5 справа), в свою чергу, маємо на навчальній множині значення $R^2 = 0.98$, на перевірочній множині - значення $R^2 = -1.8 * 10^9$.

На основі оцінок ефективності вищенаведених двох моделей можна зробити наступне узагальнене спостереження:

- Для моделей зі значною систематичною помилкою ефективність моделі на перевірочному наборі даних не набагато гірша за її ефективність на навчальній множині.

- Для моделей з високою дисперсією ефективність моделі на перевірочному наборі даних є істотно гіршою за її ефективність на навчальній множині.

Якщо б ми могли управляти складністю моделі, то оцінки ефективності моделі на навчальній і перевірочній множинах вели б себе так як показано на рис. 6.

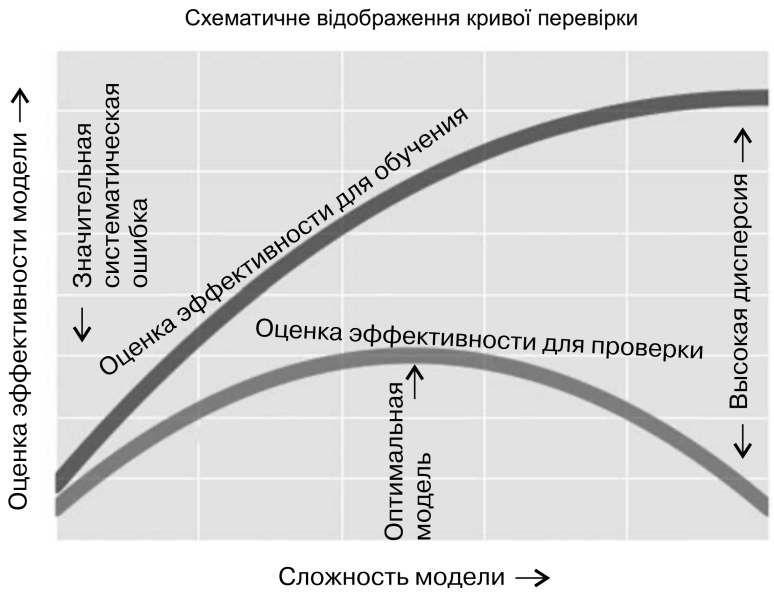

Рис. 6: Схематичне представлення залежності між складністю моделі і оцінками ефективності моделі на навчальній і перевірочній множинах

Наведений на рис.6 графік часто називають кривою перевірки (validation curve). На ньому можна спостерігати наступні важливі особливості:

- Оцінка ефективності на навчальній множині завжди перевищує оцінку ефективності моделі на перевірочній множині. Дійсно, модель краще підходить для вже бачених нею даних, ніж для тих, які вона ще не бачила.

- Оцінка ефективності для навчання зазвичай монотонно зростає із ростом складності моделі.

- Моделі з дуже низькою складністю (зі значною систематичною помилкою) є недостатньо навченими, тобто ці моделі будуть погано прогнозувати як дані навчальної множини, так і будь-які раніше не бачені ними дані.

- Моделі з дуже високою складністю (з високою дисперсією) є перенавченими, тобто будуть дуже добре прогнозувати дані навчальної множини, але на будь-яких раніше не бачених даних працювати дуже погано.

- Крива перевірки досягає максимуму в деякій проміжній точці; після цієї точки оцінка ефективності для перевірки різко спадає і модель стає перенавченою. Цей рівень складності означає прийнятний компроміс між систематичною помилкою і дисперсією.

Засоби регулювання складності моделі розрізняються залежно від моделі. Як здійснювати подібне регулювання для кожної з моделей, ми побачимо далі, коли розглянемо конкретні моделі докладніше.

4.2 Криві перевірки в бібліотеці Scikit-Learn

Розглянемо приклад перехресної перевірки і розрахунку кривої перевірки для моделі поліноміальної регресії.

Поліноміальна регресія - це узагальнена лінійна модель з параметризованим ступенем многочлена. Наприклад, многочлен 1-го ступеня апроксимує дані прямою лінією при параметрах моделі a і b:

$$y = ax + b.$$

Многочлен 3-го степеня апроксимує дані за допомогою кубічної кривої за параметрів моделі a, b, c, d:

$$y = ax^3 + bx^2 + cx + d.$$

Модель можна узагальнити на будь-яку кількість поліноміальних ознак. У бібліотеці Scikit-Learn реалізувати це можна за допомогою простої **лінійної регресії** в поєднанні з **поліноміальним препроцесором**. Використаємо **конвеєр (pipeline)** для поєднання цих операцій в єдиний ланцюг:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline

def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree),
                         LinearRegression(**kwargs))
```

Створимо дані для навчання випадковим чином:

```
import numpy as np

def make_data(N, err=1.0, rseed=1):

    rng = np.random.RandomState(rseed)
    X = rng.rand(N, 1) ** 2
    y = 10 - 1. / (X.ravel() + 0.1)
```

```
    if err > 0:
        y += err * rng.randn(N)
    return X, y

X, y = make_data(40)
```

Візуалізуємо створені дані разом з декількома їх апроксимаціями многочленами різного ступеня (рис. 7):

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()

X_test = np.linspace(-0.1, 1.1, 500)[:, None]

plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()
for degree in [1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X, y).
        predict(X_test)
    plt.plot(X_test.ravel(), y_test, label='degree={0}'.
        format(degree))
plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');
```

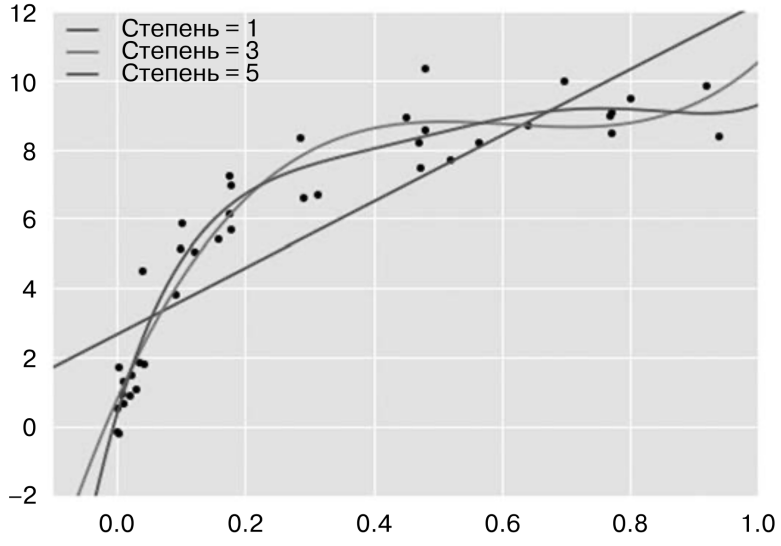

Рис. 7: Наближення набору даних трьома різними поліноміальними моделями

Параметром для управління складністю моделі в даному випадку є ступінь многочлена, яка може бути будь-яким невід'ємним числом. Виникає питання:

який ступінь многочлена забезпечує компроміс між систематичною помилкою (недонавчанням) і дисперсією (перенавчанням)?

Щоб вирішити це питання побудуємо **криву перевірки** для цих конкретних даних і моделей, використовуючи функцію `validation_curve`.

```
sklearn.model_selection.validation_curve(estimator, X, y,
    param_name, param_range, groups=None, cv=None,
    scoring=None, n_jobs=None, pre_dispatch='all',
    verbose=0, error_score=nan)
```

Основні вхідні параметри функції `validation_curve`: модель estimator, дані X, y, назва параметра `param_name` і діапазон для аналізу `param_range`. Функція обчислює (рис. 8):

- `train_score` - значення оцінки ефективності для навчання та
- `val_score` - значення оцінки ефективності на перевірочній множині у вказаному діапазоні.

```
from sklearn.model_selection import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(),
                                          X, y, 'polynomialfeatures__degree',
                                          degree, cv=7)
plt.plot(degree, np.median(train_score, 1), color='blue',
                                         label='training score')
plt.plot(degree, np.median(val_score, 1), color='red',
                                         label='validation score')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

Як і очікувалось цей графік демонструє наступну якісну поведінку: оцінка ефективності для навчання на всьому діапазоні перевищує оцінку ефективності для перевірки; оцінка ефективності для навчання монотонно зростає зі зростанням складності моделі, а оцінка ефективності для перевірки досягає максимуму перед різким спадом в точці, де модель стає перенавченою.

Як можна зрозуміти з наведеної кривої перевірки (рис. 8), оптимальний компроміс між систематичною помилкою і дисперсією досягається для многочлена третього ступеня. Обчислити і відобразити на графіку цю апроксимацію на початкових даних можна наступним чином (рис. 9):

```
plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```

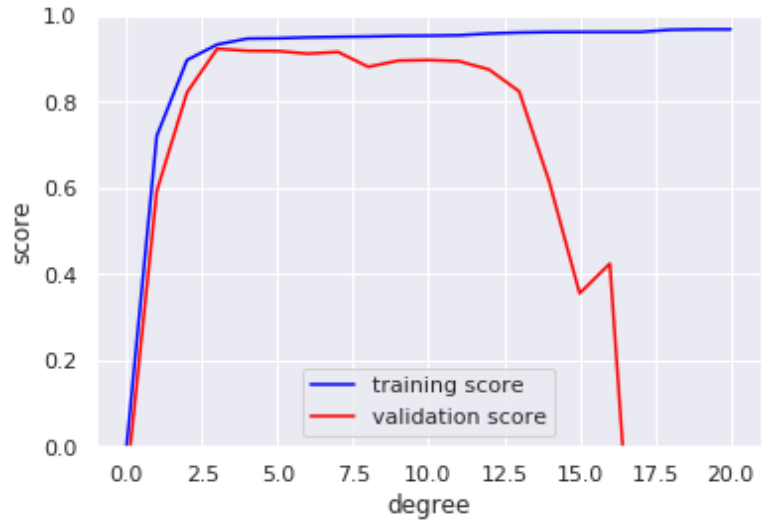

Рис. 8: Крива перевірки для даних, наведених на рис.7 (degree - ступінь, score - оцінка, training score - оцінка на навчальній множині, validation score - оцінка на перевірочній множині)

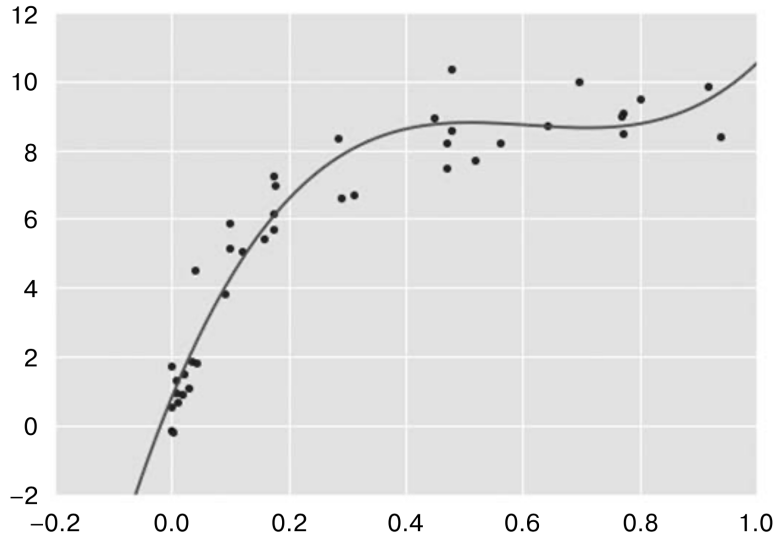

Рис. 9: Оптимальна поліноміальна модель отримана в результаті перехресної перевірки

Слід відмітити, що для знаходження оптимальної моделі не потрібно обчислювати оцінку ефективності на навчальній множині.

Проте вивчення залежності між оцінками ефективності на навчальній і перевірочній множинах дає нам корисну інформацію щодо ефективності моделі.

4.3 Криві навчання в бібліотеці Scikit-Learn

Поведінка кривої перевірки залежить не від одного, а від двох важливих факторів:

- складності моделі,
- кількості навчальних прикладів, тобто потужності навчальної множини.

Наприклад, згенеруємо новий набір даних з кількістю точок в п'ять разів більшою за попередній приклад (рис. 10):

```
X2, y2 = make_data(200)
plt.scatter(X2.ravel(), y2)
```

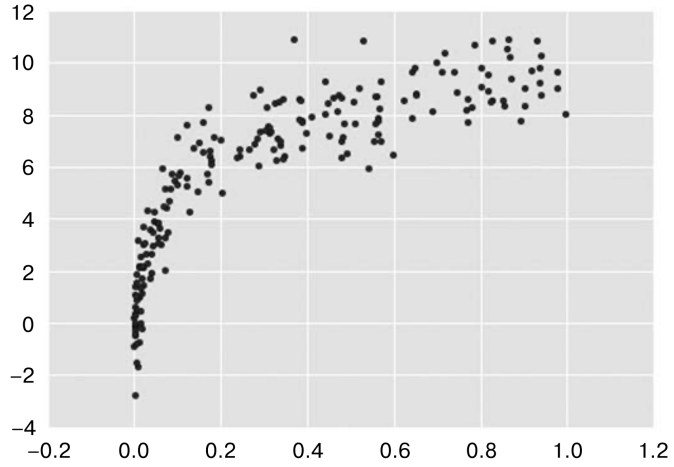

Рис. 10: Набір даних для демонстрації кривих навчання

Повторимо вищенаведений код і побудуємо графік кривої навчання для цього більшого набору даних (рис. 11):

```
degree = np.arange(21)
train_sc2, val_sc2 = validation_curve(PolynomialRegression(),
                        X2, y2, 'polynomialfeatures__degree',
                        degree, cv=7)
plt.plot(degree, np.median(train_sc2, 1), color='blue',
         label='training score')
plt.plot(degree, np.median(val_sc2, 1), color='red',
                            label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue',
         alpha=0.3, linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red',
         alpha=0.3, linestyle='dashed')
plt.legend(loc='lower center')
```

```
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

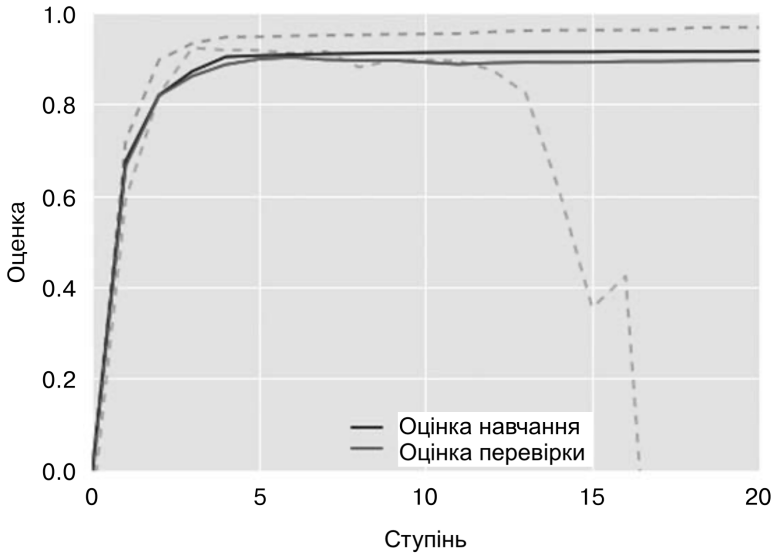

Рис. 11: Криві навчання для апроксимації даних, що наведені на рис.10 поліноміальною моделлю

Суцільні лінії на рис. 11 показують нові результати для 200 навчальних прикладів, а штрихові лінії - результати попереднього меншого набору даних з 40 прикладів. З цієї кривої навчання бачимо, що більший набір даних дозволяє використовувати набагато складнішу модель: максимум кривої знаходиться біля ступеня поліному рівного 6. Бачимо, що навіть модель зі ступенем 20 не є сильно перенавченою, оскільки оцінки ефективності для перевірки і навчання залишаються дуже близькими один до одного.

Щоб дослідити поведінку моделі залежно від кількості навчальних прикладів використовують прийом, в якому поступово збільшують підмножину даних для навчання моделі. Графік оцінок для навчання / перевірки залежно від розміру навчальної множини називається **кривою навчання (learning curve)**.

Модель заданої складності буде перенавченою на занадто маленькому наборі даних. Це означає, що оцінка ефективності на навчальному наборі буде відносно високою, а оцінка ефективності на перевірочній множині - відносно низькою.

Модель заданої складності виявиться недонавченою на занадто великому наборі даних. Це означає, що зі зростанням розміру набору даних оцінка ефективності для навчання буде знижуватися, а оцінка ефективності для перевірки - підвищуватися.

Враховуючи ці особливості, відобразимо схематично вигляд кривої навчання (рис. 12).

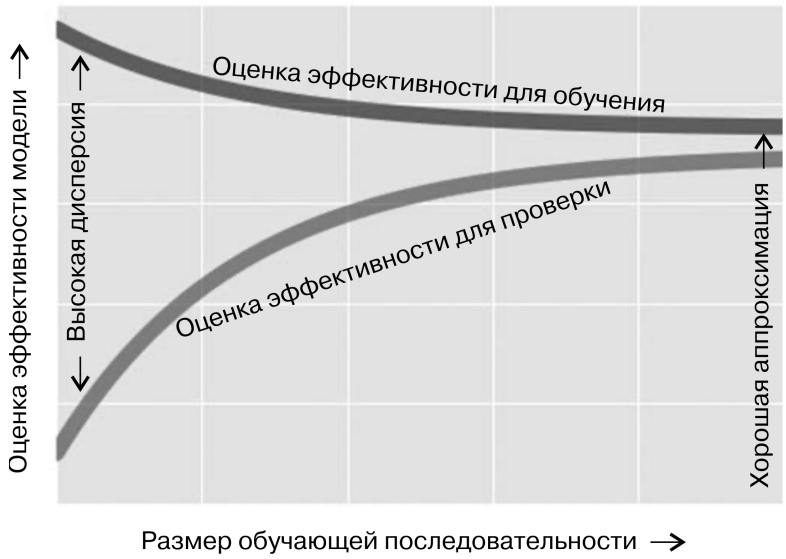

Рис. 12: Схематичний вигляд кривої навчання на навчальній і перевірочній множинах

Особливість кривої навчання - це її збіжність до конкретного значення оцінки ефективності при зростанні числа навчальних прикладів. У випадку «збіжності», додавання нових навчальних даних вже не призводить до підвищення оцінки ефективності! Єдиним способом поліпшити якість моделі в цьому випадку буде використання іншої, часто більш складної, моделі.

Побудуємо криві навчання для розглянутого прикладу для поліноміальних моделей другого і дев'ятого ступенів (рис. 13):

```
from sklearn.model_selection import learning_curve

fig, ax = plt.subplots(1, 2, figsize=(16, 6))

fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve
                (PolynomialRegression(degree),
                  X, y, cv=7, train_sizes=np.linspace(0.3, 1, 25))
    ax[i].plot(N, np.mean(train_lc, 1), color='blue',
                label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red',
                label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0],
                N[-1], color='gray', linestyle='dashed')
    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
```

```
ax[i].set_ylabel('score')
ax[i].set_title('degree = {0}'.format(degree), size=14)
ax[i].legend(loc='best')
```

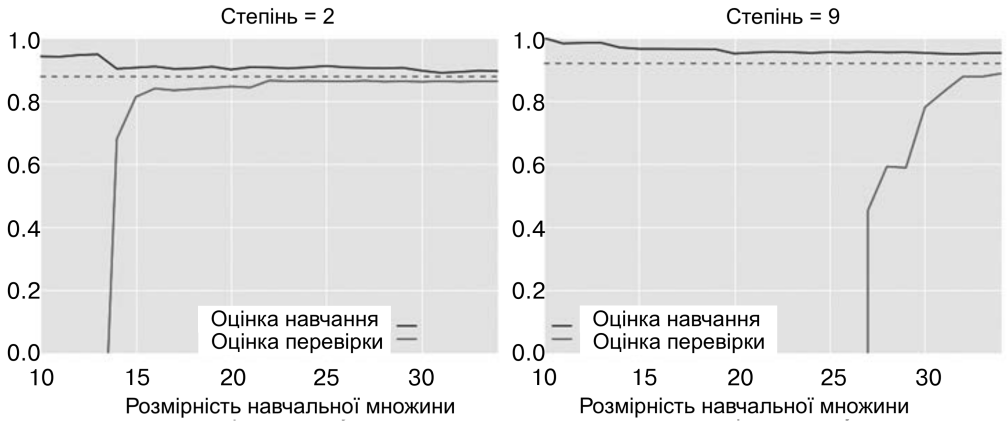

Рис. 13: Криві навчання для даних, наведених на рис.10. Зліва - для моделі низької складності. Справа - для моделі високої складності

На рис. 13 ми спостерігаємо як веде себе побудована модель зі збільшенням обсягу навчальних даних. Зокрема, після того моменту, коли крива навчання вже зійшлася до якогось значенням (тобто коли криві навчання і перевірки вже близькі один до одного), додавання додаткових навчальних даних не поліпшить апроксимацію істотно! Ця ситуація відображена на лівому малюнку з кривою навчання для моделі другого степеня.

Поліпшити оцінку кривої навчання, яка вже зійшлася, можна лише використанням іншої, зазвичай більш складної моделі. Це видно на правому малюнку: перейшовши до більш складної моделі дев'ятого степеня, ми покращуємо оцінку для точки збіжності, що показана штриховою лінією. Покращена модель має більш високу дисперсію - відстань між оцінками ефективності для навчання і перевірки. Якби нам довелося додати ще більше точок, крива навчання для більш складної з цих моделей все одно в підсумку б зійшлася.

Побудова графіка кривої навчання для конкретних моделі і набору даних полегшує прийняття рішення про те, в якому напрямку можна ще покращити аналіз даних.

4.4 Пошук по сітці, Grid Search

На практиці у моделей зазвичай буває більше одного параметра, тому графіки кривих перевірки і навчання перетворюються з двовимірних ліній в багатовимірні поверхні. Виконання подібних візуалізацій в таких випадках є непростим завданням, тому краще відшукати конкретну модель, при якій оцінка ефективності для перевірки досягає максимуму.

Бібліотека Scikit-Learn надає для цієї мети спеціальні автоматичні інструменти, що містяться в модулі `model_selection`. Розглянемо тривимірну сітку

ознак моделі - ступеня многочлена, флага, який вказує, чи потрібно підбирати точку перетину з віссю координат, і флага, який вказує, чи слід виконувати нормалізацію. Виконати ці настройки можна за допомогою мета-оцінщика GridSearchCV бібліотеки Scikit-Learn:

```
from sklearn.model_selection import GridSearchCV

param_grid = {'polynomialfeatures__degree': np.arange(21),
              'linearregression__fit_intercept': [True, False],
              'linearregression__normalize': [True, False]}

grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

На даному етапі цей оцінщик, як і будь-який інший оцінщик, ще НЕ був застосований до даних. Навчання моделі та відстеження проміжних оцінок ефективності в кожній з точок сітки проводиться шляхом виклику методу `fit()`:

```
grid.fit(X, y)
```

Після навчання можна дізнатися значення оптимальних параметрів:

```
In:  grid.best_params_

Out: {'linearregression__fit_intercept': False,
      'linearregression__normalize': True,
      'polynomialfeatures__degree': 4}
```

При необхідності можна скористатися цією оптимальною моделлю і графічно зобразити апроксимацію (рис.14):

```
model = grid.best_estimator_

plt.scatter(X.ravel(), y)
lim = plt.axis()

y_test = model.fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test, hold=True)
plt.axis(lim)
```

У пошуку по сітці є безліч опцій, включаючи можливості завдання користувальницької функції оцінки ефективності, розпаралелювання обчислень, виконання випадкового пошуку та ін. Для отримання додаткової інформації дивіться документацію бібліотеки Scikit-Learn, що присвячена пошуку по сітці:

`https://scikit-learn.org/stable/modules/grid_search.html`

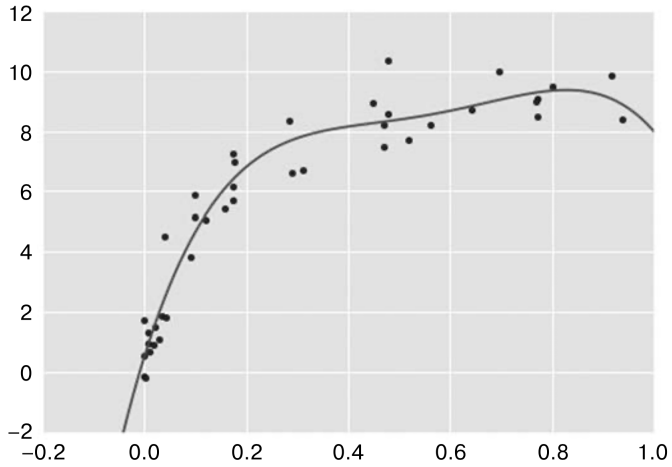

Рис. 14: Найкраща модель, визначена шляхом автоматичного пошуку по сітці