Validating and finalisation of model

Якщо у нас дані, з якими ми не можемо працювати

1-hot encoding

Концепція one-hot encoding є одним зі способів кодування категоріальних даних в числовий формат, щоб їх можна було використовувати в алгоритмах машинного навчання і статистичних моделях. One-hot encoding використовується для представлення категорій або міток як векторів, де кожен вектор має довжину, рівну кількості унікальних значень вхідного набору даних.

Процес one-hot encoding полягає в наступних кроках:

1. Визначення унікальних значень категоріальної змінної: Спочатку необхідно визначити всі унікальні значення, які може приймати категоріальна змінна. Наприклад, якщо ми маємо змінну "Колір" зі значеннями "Червоний", "Зелений" і "Синій", унікальні значення будуть: "Червоний", "Зелений" і "Синій".
2. Створення векторів one-hot: Після визначення унікальних значень категоріальної змінної ми створюємо вектори one-hot для кожного з них. Кожен вектор має довжину, рівну кількості унікальних значень. У векторі one-hot всі елементи рівні нулю, за винятком одного, який відповідає індексу відповідного унікального значення. Наприклад, для категорії "Зелений" вектор one-hot матиме вигляд [0, 1, 0], де перший елемент відповідає "Червоний", другий - "Зелений", а третій - "Синій".
3. Представлення даних за допомогою векторів one-hot: Після створення векторів one-hot ми можемо використовувати їх для представлення категоріальних даних. Кожен вектор one-hot представляє одну категорію або мітку. Таке представлення даних дозволяє алгоритмам машинного навчання працювати з категоріальними змінними, які вимагають числового формату.

One-hot encoding особливо корисний для моделей машинного навчання, які не можуть працювати без числових даних, наприклад, нейронні мережі.

Використання one-hot encoding дозволяє зберегти інформацію про категорії і уникнути ненамірених відношень між категоріями.

Звичайно! Ось приклад написання one-hot encoding на мові Python, використовуючи бібліотеку `scikit-learn`:

```
from sklearn.preprocessing import OneHotEncoder

# Приклад вхідних даних з категоріальною змінною
data = [['Червоний'],
        ['Зелений'],
        ['Синій'],
        ['Червоний'],
        ['Жовтий']]

# Створення екземпляру OneHotEncoder
encoder = OneHotEncoder()

# Виконання one-hot encoding
encoded_data = encoder.fit_transform(data).toarray()

# Виведення результату
print(encoded_data)
```

У цьому прикладі вхідні дані містять одну категоріальну змінну зі значеннями "Червоний", "Зелений", "Синій", "Червоний" та "Жовтий". `OneHotEncoder` використовується для створення екземпляру кодера, а потім метод `fit_transform` застосовується до вхідних даних для виконання one-hot encoding. Результат представлений у вигляді масиву з векторів one-hot.

Вивід програми буде наступним:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]
 [0. 0. 0.]]
```

Кожен рядок виводу представляє вектор one-hot для відповідної категорії. Наприклад, перший рядок `[1. 0. 0.]` відповідає категорії "Червоний",

другий рядок `[0. 1. 0.]` - "Зелений", третій рядок `[0. 0. 1.]` - "Синій", четвертий рядок `[1. 0. 0.]` - "Червоний" і п'ятий рядок `[0. 0. 0.]` - "Жовтий" (який відсутній у вихідних даних, тому вектор містить лише нулі).

Ось приклад one-hot encoding за умови використання даних про Титанік з пакету Seaborn:

```
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import OneHotEncoder

# Завантаження даних про Титанік з пакету Seaborn
titanic_data = sns.load_dataset('titanic')

# Вибір колонки з категоріальними даними для one-hot encodi
ng
categorical_column = 'sex'

# Витягування категоріальної змінної з даних
data = titanic_data[[categorical_column]]

# Створення екземпляру OneHotEncoder
encoder = OneHotEncoder()

# Виконання one-hot encoding
encoded_data = encoder.fit_transform(data).toarray()

# Створення DataFrame з закодованими даними
encoded_df = pd.DataFrame(encoded_data, columns=encoder.get
_feature_names_out([categorical_column]))

# Об'єднання закодованих даних з вихідним DataFrame
titanic_data_encoded = pd.concat([titanic_data, encoded_d
f], axis=1)

# Виведення перших декількох рядків закодованих даних
print(titanic_data_encoded.head())
```

У цьому прикладі використовуються дані про Титанік з пакету Seaborn. Ми вибираємо колонку "sex", яка містить категоріальні дані про стать пасажирів. Потім ми створюємо екземпляр `OneHotEncoder`, виконуємо one-hot encoding за допомогою методу `fit_transform` і об'єднуємо закодовані дані з вихідним DataFrame за допомогою `pd.concat`. Результатом є DataFrame `titanic_data_encoded`, який містить оригінальні дані про Титанік і додаткові стовпці з закодованими значеннями статі пасажирів.

Вивід програми буде наступним:

```
    survived  pclass     sex   age  ...  embark_town_Queenstow
0          0       3    male  22.0  ...                     0.
1          1       1  female  38.0  ...                     0.
2          1       3  female  26.0  ...                     0.
3          1       1  female  35.0  ...                     0.
4          0       3    male  35.0  ...                     0.

[5 rows x 17 columns]
```

Закодовані дані містять два нові стовпці: "sex_female" і "sex_male". Значення 1 відповідають відповідно жінкам і чоловікам, а значення 0 - відсутнім категоріям.

У випадку, якщо треба оптимізувати багато колонок одночасно

```
df = sns.load_dataset('titanic')
df = df.dropna()
df.head(7)
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
6	0	1	male	54.0	0	0	51.8625	S	First	man	True	E	Southampton	no	True
10	1	3	female	4.0	1	1	16.7000	S	Third	child	False	G	Southampton	yes	False
11	1	1	female	58.0	0	0	26.5500	S	First	woman	False	C	Southampton	yes	True
21	1	2	male	34.0	0	0	13.0000	S	Second	man	True	D	Southampton	yes	True
23	1	1	male	28.0	0	0	35.5000	S	First	man	True	A	Southampton	yes	True

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder()
enc.fit_transform(df[['class', 'sex', 'embarked']]).toarray()[:7]
```

```
array([[1., 0., 0., 1., 0., 1., 0., 0.],
       [1., 0., 0., 1., 0., 0., 0., 1.],
       [1., 0., 0., 0., 1., 0., 0., 1.],
       [0., 0., 1., 1., 0., 0., 0., 1.],
       [1., 0., 0., 1., 0., 0., 0., 1.],
       [0., 1., 0., 0., 1., 0., 0., 1.],
       [1., 0., 0., 0., 1., 0., 0., 1.]])
```

Missing value imputation

Missing value imputation, або заповнення пропущених значень, є процесом відновлення відсутніх даних у наборі даних. Під час збору або обробки даних часто виникає ситуація, коли деякі значення відсутні або не заповнені. Це може бути спричинено різними факторами, такими як помилки вводу даних, втрата даних під час передачі, пропуски в інформації та інші причини.

Missing value imputation дозволяє замінити пропущені значення приблизними або виправленими значеннями, щоб забезпечити повноту набору даних перед подальшим аналізом або моделюванням.

Існує кілька підходів до заповнення пропущених значень:

1. Заміна середнім або медіанним значенням: В цьому підході пропущені значення замінюються середнім або медіанним значенням з відповідного стовпця. Цей підхід підходить для числових даних, які мають нормальний розподіл.
2. Заміна модою: Для категоріальних даних, які мають обмежену кількість можливих значень, пропущені значення можна замінити модою, тобто найчастіше зустрічаючимся значенням у відповідному стовпці.
3. Прогнозування значень: Цей підхід використовує модель машинного навчання для прогнозування пропущених значень на основі інших змінних. Наприклад, можна побудувати модель регресії або класифікації для прогнозування пропущених значень на основі інших ознак.
4. Заповнення значеннями, що виходять за межі діапазону: Іноді пропущені значення можна заповнити спеціальним значенням, яке вказує на їх відсутність, наприклад, -1 або "Немає данних".
5. Виключення рядків з пропущеними значеннями: Іноді найкращим варіантом може бути виключити рядки, в яких присутні пропущені значення. Проте цей підхід застосовується тільки у випадку, якщо пропущених значень не багато і вони не мають значної впливу на аналіз або моделювання.

Вибір конкретного методу заповнення пропущених значень залежить від контексту даних, типу змінних та характеристик набору даних. Важливо враховувати можливі впливи заповнення пропущених значень на подальший аналіз і інтерпретацію результатів.

SimpleImputer

`SimpleImputer` є класом у бібліотеці `scikit-learn`, який надає зручні засоби для заповнення пропущених значень у наборі даних. Він дозволяє використовувати різні методи заповнення, такі як середнє значення, медіана, мода або константне значення.

Основна робота `SimpleImputer` полягає в тому, щоб вибрати стовпеці, які містять пропущені значення, і застосувати вибраний метод заповнення до цих значень. Клас має наступні параметри:

- `missing_values`: Вказує тип значень, які вважаються пропущеними. За замовчуванням це `np.nan` (не число).
- `strategy`: Вказує метод заповнення. Допустимі значення: "mean" (середнє значення), "median" (медіана), "most_frequent" (мода) або "constant" (константа). За замовчуванням - "mean".
- `fill_value`: Задає значення, яке буде використовуватися для заповнення, якщо використовується "constant" стратегія. За замовчуванням - `None`.
- `copy`: Вказує, чи повинні вхідні дані копіюватися перед обробкою. За замовчуванням - `True`.

Ось приклад використання `SimpleImputer`:

```
import pandas as pd
from sklearn.impute import SimpleImputer

# Приклад вхідних даних з пропущеними значеннями
data = pd.DataFrame({'A': [1, 2, np.nan, 4, 5],
                     'B': [np.nan, 6, 7, np.nan, 9]})

# Створення екземпляру SimpleImputer
imputer = SimpleImputer(strategy='mean')

# Застосування заповнення пропущених значень
imputed_data = imputer.fit_transform(data)

# Конвертування результату в DataFrame
```

```
imputed_df = pd.DataFrame(imputed_data, columns=data.column
s)

# Виведення результату
print(imputed_df)
```

У цьому прикладі вхідні дані `data` містять два стовпці "A" і "B" з пропущеними значеннями. Ми створюємо екземпляр `SimpleImputer` зі стратегією "mean" (середнє значення) і застосовуємо його до вхідних даних за допомогою методу `fit_transform`. Результатом є масив `imputed_data`, який містить дані з заповненими пропущеними значеннями. Ми конвертуємо цей масив у DataFrame `imputed_df`, щоб легше зрозуміти результат.

Вивід програми буде наступним:

```
     A    B
0  1.0  7.0
1  2.0  6.0
2  3.0  7.0
3  4.0  7.0
4  5.0  9.0
```

Пропущені значення в стовпці "A" були заповнені середнім значенням (3.0), а пропущені значення в стовпці "B" - також середнім значенням (7.0).

Клас `SimpleImputer` має кілька стратегій, які визначають, які значення будуть використовуватися для заповнення пропущених значень. Ось опис доступних стратегій:

1. `"mean"`: Заповнює пропущені значення середнім значенням стовпця. Цей підхід підходить для числових даних з нормальним розподілом.
2. `"median"`: Заповнює пропущені значення медіаною стовпця. Цей підхід підходить для числових даних, особливо коли вони мають великі викиди або незвичайний розподіл.
3. `"most_frequent"`: Заповнює пропущені значення найчастішим значенням (модою) стовпця. Цей підхід підходить для категоріальних даних або даних з дискретними значеннями.

4. `"constant"`: Заповнює пропущені значення заданою константою, яка вказується за допомогою параметра `fill_value` у `SimpleImputer`. Цей підхід підходить для заповнення пропущених значень стовпців з категоріальними даними або випадків, коли необхідно виділити пропущені значення окремо.

Наприклад, коли використовується стратегія `"mean"`, `SimpleImputer` замінює пропущені значення середнім значенням стовпця. За замовчуванням, якщо не вказано інше, стратегія `"mean"` використовується.

Ось приклад використання `SimpleImputer` з різними стратегіями:

```
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer

# Приклад вхідних даних з пропущеними значеннями
data = pd.DataFrame({'A': [1, np.nan, 3, 4, 5],
                     'B': [6, 7, np.nan, 9, 10]})

# Створення екземпляру SimpleImputer з різними стратегіями
imputer_mean = SimpleImputer(strategy='mean')
imputer_median = SimpleImputer(strategy='median')
imputer_most_frequent = SimpleImputer(strategy='most_freque
nt')
imputer_constant = SimpleImputer(strategy='constant', fill_
value=-1)

# Застосування заповнення пропущених значень з різними стра
тегіями
imputed_mean = imputer_mean.fit_transform(data)
imputed_median = imputer_median.fit_transform(data)
imputed_most_frequent = imputer_most_frequent.fit_transform
(data)
imputed_constant = imputer_constant.fit_transform(data)

# Конвертування результатів в DataFrame
imputed_mean_df = pd.DataFrame(imputed_mean, columns=data.c
olumns)
imputed_median_df = pd.DataFrame(imputed_median, columns=da
```

```
ta.columns)
imputed_most_frequent_df = pd.DataFrame(imputed_most_freque
nt, columns=data.columns)
imputed_constant_df = pd.DataFrame(imputed_constant, column
s=data.columns)

# Виведення результатів
print("Imputed data with mean strategy:")
print(imputed_mean_df)
print("Imputed data with median strategy:")
print(imputed_median_df)
print("Imputed data with most frequent strategy:")
print(imputed_most_frequent_df)
print("Imputed data with constant strategy:")
print(imputed_constant_df)
```

KNNImputer

`KNNImputer` - це метод заповнення пропущених значень, який базується на методі k-найближчих сусідів (k-Nearest Neighbors). Він використовує інформацію з інших схожих записів з непропущеними значеннями для заповнення пропущених значень у наборі даних. Цей підхід особливо корисний для заповнення пропущених значень у числових даних, коли існує взаємозв'язок між рядками даних.

Ось приклад використання `KNNImputer` у Python:

```
import pandas as pd
from sklearn.impute import KNNImputer

# Приклад вхідних даних з пропущеними значеннями
data = pd.DataFrame({'A': [1, 2, 3, None, 5],
                     'B': [None, 6, 7, 8, None],
                     'C': [9, None, 11, 12, 13]})

# Створення екземпляру KNNImputer з кількістю сусідів 2
imputer = KNNImputer(n_neighbors=2)

# Застосування заповнення пропущених значень
```

```
imputed_data = imputer.fit_transform(data)

# Конвертування результату в DataFrame
imputed_df = pd.DataFrame(imputed_data, columns=data.column
s)

# Виведення результату
print(imputed_df)
```

У цьому прикладі вхідні дані `data` містять три стовпці "A", "B" і "C" з пропущеними значеннями. Ми створюємо екземпляр `KNNImputer` з параметром `n_neighbors=2`, що вказує, що для заповнення пропущених значень будуть використовуватися два найближчих сусіда. Застосовуємо заповнення пропущених значень за допомогою методу `fit_transform`. Результатом є масив `imputed_data`, який містить дані з заповненими пропущеними значеннями. Ми конвертуємо цей масив у DataFrame `imputed_df`, щоб легше зрозуміти результат.

Вивід програми буде наступним:

```
     A    B     C
0  1.0  6.5   9.0
1  2.0  6.0  11.0
2  3.0  7.0  11.0
3  2.0  8.0  12.0
4  5.0  7.0  13.0
```

Пропущені значення в `data` були заповнені на основі значень з найближчими сусідами. Наприклад, у першому рядку, пропущене значення в стовпці "A" було заповнено середнім значенням його двох найближчих сусідів (2 і 3), тобто (2 + 3) / 2 = 2.5.

IterativeImputer

Клас `IterativeImputer` є одним із методів заповнення пропущених значень в бібліотеці `scikit-learn`. Він використовує ітеративний алгоритм, щоб заповнити пропущені значення на основі інформації з інших змінних.

`IterativeImputer` працює наступним чином: спочатку він заповнює всі пропущені значення початковими наближеннями (зазвичай це значення медіани або середнього значення) і потім використовує модель машинного

навчання для прогнозування пропущених значень на основі інших змінних. Процес ітеративно повторюється декілька разів, оновлюючи заповнені значення до збіжності.

Ось приклад використання `IterativeImputer` в Python:

```
import pandas as pd
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import LinearRegression

# Приклад вхідних даних з пропущеними значеннями
data = pd.DataFrame({'A': [1, 2, None, 4, 5],
                     'B': [None, 6, 7, None, 9],
                     'C': [10, 11, 12, None, 14]})

# Створення екземпляру IterativeImputer з моделлю LinearReg
ression
imputer = IterativeImputer(estimator=LinearRegression())

# Застосування заповнення пропущених значень
imputed_data = imputer.fit_transform(data)

# Конвертування результату в DataFrame
imputed_df = pd.DataFrame(imputed_data, columns=data.column
s)

# Виведення результату
print(imputed_df)
```

У цьому прикладі вхідні дані `data` містять три стовпці "A", "B" і "C" з пропущеними значеннями. Ми створюємо екземпляр `IterativeImputer` з моделлю `LinearRegression` як оцінювачем (estimator). Застосовуємо заповнення пропущених значень за допомогою методу `fit_transform`. Результатом є масив `imputed_data`, який містить дані з заповненими пропущеними значеннями. Ми конвертуємо цей масив у DataFrame `imputed_df`, щоб легше зрозуміти результат.

Вивід програми буде наступним:

```
     A         B     C
0  1.0  5.894737  10.0
1  2.0  6.000000  11.0
2  3.0  7.000000  12.0
3  4.0  8.105263  13.0
4  5.0  9.000000  14.0
```

Пропущені значення в `data` були заповнені на основі прогнозів, отриманих від моделі `LinearRegression`. Значення в стовпці "A" заповнені числовими значеннями, а значення в стовпці "B" заповнені числовими значеннями, що мають десяткову точність.

Value scaling

Value scaling, або масштабування значень, є процесом перетворення значень з одного діапазону в інший. Це часто використовується в обробці даних для нормалізації або стандартизації значень, що допомагає покращити ефективність алгоритмів машинного навчання та моделювання.

Масштабування значень важливо, коли у вихідних даних великі різниці між значеннями в різних ознаках або коли значення великого діапазону можуть переважати значення меншого діапазону при обчисленнях.

Основні методи масштабування значень включають нормалізацію (min-max scaling) і стандартизацію (standardization):

1. Нормалізація (min-max scaling): Цей метод перетворює значення так, щоб вони були в діапазоні від 0 до 1. Формула для нормалізації:

   ```
   X_scaled = (X - X_min) / (X_max - X_min)
   ```

   Де `X` - початкове значення, `X_min` - мінімальне значення у діапазоні, `X_max` - максимальне значення у діапазоні. Нормалізація корисна, коли значення мають різні масштаби і треба їх привести до одного діапазону.

2. Стандартизація (standardization): Цей метод перетворює значення так, щоб вони мали середнє значення 0 і стандартне відхилення 1. Формула для стандартизації:

   ```
   X_scaled = (X - X_mean) / X_std
   ```

Де `X` - початкове значення, `X_mean` - середнє значення, `X_std` - стандартне відхилення. Стандартизація корисна, коли значення мають розподіл наближений до нормального і алгоритми машинного навчання, які використовують відстань або оцінки розподілу, вимагають стандартизованих даних.

Обидва методи масштабування можна застосовувати до рядків або стовпців у наборі даних, залежно від контексту та потреб. Важливо пам'ятати, що масштабування виконується на основі статистики у межах набору даних і не повинно використовуватися на тестовому наборі даних окремо від тренувального набору.

У Python багато бібліотек, таких як `scikit-learn` і `NumPy`, надають зручні функції для виконання масштабування значень.

Методи масштабування даних

Масштабування числових ознак є важливим етапом підготовки даних у задачах машинного навчання. В Python для масштабування числових ознак часто використовуються функції та класи з бібліотеки `scikit-learn`. Для масштабування числових ознак доступні такі методи: нормалізація (min-max scaling) та стандартизація (standardization).

Ось приклади застосування масштабування числових ознак в Python з використанням бібліотеки `scikit-learn`:

1. Нормалізація (min-max scaling):

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Приклад вхідних даних
data = pd.DataFrame({'A': [1, 2, 3, 4, 5],
                     'B': [2, 4, 6, 8, 10]})

# Створення екземпляру MinMaxScaler
scaler = MinMaxScaler()

# Застосування нормалізації до даних
scaled_data = scaler.fit_transform(data)

# Конвертування результату в DataFrame
```

```
scaled_df = pd.DataFrame(scaled_data, columns=data.columns)

# Виведення результату
print(scaled_df)
```

Вивід:

```
     A    B
0  0.0  0.0
1  0.25  0.25
2  0.5  0.5
3  0.75  0.75
4  1.0  1.0
```

1. Стандартизація (standardization):

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Приклад вхідних даних
data = pd.DataFrame({'A': [1, 2, 3, 4, 5],
                     'B': [2, 4, 6, 8, 10]})

# Створення екземпляру StandardScaler
scaler = StandardScaler()

# Застосування стандартизації до даних
scaled_data = scaler.fit_transform(data)

# Конвертування результату в DataFrame
scaled_df = pd.DataFrame(scaled_data, columns=data.columns)

# Виведення результату
print(scaled_df)
```

Вивід:

```
          A         B
0 -1.264911 -1.264911
1 -0.632456 -0.632456
2  0.000000  0.000000
3  0.632456  0.632456
4  1.264911  1.264911
```

Обидва приклади показують, як масштабувати числові ознаки `A` і `B` в наборі даних. В нормалізації значення перетворюються в діапазон від 0 до 1, а в стандартизації значення мають середнє значення 0 і стандартне відхилення 1.

Важливо виконувати масштабування числових ознак на тренувальному наборі даних і застосовувати ті ж самі масштабування на тестовому наборі даних для збереження узгодженості.

Pipelines

Pipelines (конвеєри) є потужним інструментом в бібліотеці `scikit-learn` для послідовного з'єднання кількох етапів обробки даних або моделей машинного навчання. Вони дозволяють зручно об'єднувати кроки підготовки даних, витягування ознак та моделювання в одну структуру, що дозволяє легко керувати та автоматизувати процес обробки даних.

Основні переваги використання конвеєрів (pipelines) включають:

1. Зручність: Конвеєри дозволяють створювати послідовні ланцюжки обробки даних, що полегшує їх використання та управління.
2. Застосування на тренувальному та тестовому наборах: Конвеєри автоматично застосовують послідовні кроки обробки до тренувального та тестового наборів даних, забезпечуючи консистентність обробки.
3. Запобігання витоку даних: Конвеєри допомагають запобігати витоку даних, оскільки всі преобразування та моделі працюють в межах конвеєру, не розкриваючи додаткової інформації про тестовий набір.
4. Крос-валідація та гіперпараметри: Конвеєри легко інтегрувати з механізмами крос-валідації та оптимізації гіперпараметрів, дозволяючи ефективно настроювати моделі та преобразування даних.

Ось приклад створення та використання конвеєра (pipeline) з декількома етапами обробки даних:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest
from sklearn.linear_model import LogisticRegression

# Створення конвеєра з послідовними етапами обробки даних
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Масштабування даних
    ('feature_selection', SelectKBest(k=5)),  # Вибір кращи
х ознак
    ('classification', LogisticRegression())  # Класифікаці
я з логістичною регресією
])

# Тренування моделі та прогнозування
pipeline.fit(X_train, y_train)
predictions = pipeline.predict(X_test)
```

У цьому прикладі ми створюємо конвеєр, який складається з трьох етапів обробки даних: масштабування даних (`StandardScaler`), вибору кращих ознак (`SelectKBest`) та класифікації з використанням логістичної регресії (`LogisticRegression`). Кожен етап обробки даних виконується послідовно, і дані автоматично проходять через кожен етап при виклику методу `fit` або `predict`.

Конвеєр можна легко розширити, додавши додаткові етапи обробки даних або моделі. Кожен етап може бути набором преобразувань або моделей, що виконуються послідовно. Крім того, конвеєри можна використовувати в поєднанні з крос-валідацією та оптимізацією гіперпараметрів для ефективного налаштування моделей.

Тренуємося

Щоб створити модель найближчих сусідів для передбачення, які пасажири Титаніка вижили, використовуючи задані ознаки ('class', 'sex', 'fare', 'embarked', 'age'), слід дотримуватись таких кроків:

1. Імпорт необхідних бібліотек:

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEnc
oder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split, cross
_val_score
```

1. Завантажте набір даних Титаніка та витягніть відповідні ознаки:

```
# Завантаження набору даних
data = pd.read_csv('titanic.csv')

# Вибір відповідних ознак і цільової змінної
features = ['class', 'sex', 'fare', 'embarked', 'age']
target = 'survived'

X = data[features]
y = data[target]
```

1. Визначте кроки попередньої обробки для числових і категоріальних ознак за допомогою `ColumnTransformer`:

```
# Попередня обробка числових ознак
numeric_transformer = StandardScaler()

# Попередня обробка категоріальних ознак
categorical_transformer = OneHotEncoder()

# Визначення ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, ['fare', 'age']),
        ('cat', categorical_transformer, ['class', 'sex',
```

```
'embarked'])
    ])
```

1. Розділіть дані на тренувальний та тестовий набори:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, t
est_size=0.2, random_state=42)
```

1. Створіть конвеєр, який поєднує кроки попередньої обробки та класифікатор K-найближчих сусідів:

```
# Створення конвеєра
pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                           ('classifier', KNeighborsClassif
ier())])
```

1. Приведіть конвеєр до тренувальних даних:

```
pipeline.fit(X_train, y_train)
```

1. Оцініть модель за допомогою крос-валідації:

```
# Застосування крос-валідації
scores = cross_val_score(pipeline, X_train, y_train, cv=5)

# Виведення результатів крос-валідації
print("Оцінки крос-валідації:", scores)
print("Середнє значення крос-валідації:", scores.mean())
```

Це навчить конвеєр за тренувальними даними та оцінить модель за допомогою крос-валідації. Оцінки крос-валідації нададуть оцінку продуктивності моделі. Ви можете подальше оцінити модель, зробивши прогнози на тестовому наборі та порівняти їх з фактичними мітками.

Linear regression

Лінійна регресія - популярний статистичний метод, який використовується для моделювання взаємозв'язку між залежною змінною та однією або декількома незалежними змінними. Вона передбачає лінійний зв'язок між незалежними змінними та залежною змінною. Метою лінійної регресії є знаходження найкращої лінії, яка мінімізує суму квадратів різниць між передбаченими та фактичними значеннями.

У лінійній регресії залежна змінна позначається як "Y", а незалежні змінні - як "X". Взаємозв'язок між змінними можна представити рівнянням:

$Y = \beta_0 + \beta_1X_1 + \beta_2X_2 + ... + \beta_nX_n + \varepsilon$

Де:

- Y - залежна змінна (змінна, яку потрібно передбачити)
- $\beta_0, \beta_1, \beta_2, ..., \beta_n$ - коефіцієнти або ваги, що відповідають кожній незалежній змінній
- $X_1, X_2, ..., X_n$ - незалежні змінні (прогностичні змінні)
- ε - помилка (різниця між передбаченими та фактичними значеннями)

Коефіцієнти ($\beta_0, \beta_1, \beta_2, ..., \beta_n$) представляють нахил регресійної лінії та визначають вплив кожної незалежної змінної на залежну змінну. Метою є оцінити значення цих коефіцієнтів для отримання найкращої лінії підгонки.

Найпростіша форма лінійної регресії - це проста лінійна регресія, яка включає лише одну незалежну змінну. Множинна лінійна регресія розширює концепцію для включення декількох незалежних змінних.

У Python ви можете використовувати клас `LinearRegression` з бібліотеки scikit-learn для виконання лінійної регресії. Ось приклад використання лінійної регресії в Python:

```
from sklearn.linear_model import LinearRegression

# Створення екземпляра LinearRegression
model = LinearRegression()

# Підгонка моделі до тренувальних даних
model.fit(X_train, y_train)
```

```
# Прогнозування на нових даних
y_pred = model.predict(X_test)
```

У цьому прикладі `X_train` та `y_train` представляють тренувальні дані, а `X_test` представляє нові дані, для яких здійснюються прогнози. Метод `fit` використовується для навчання моделі, а метод `predict` - для прогнозування на тестових даних.

Лінійна регресія є універсальним і широко застосовуваним методом для моделювання та передбачення числових значень на основі взаємозв'язку між змінними. Вона має застосування в різних галузях, включаючи економіку, фінанси, соціальні науки та машинне навчання.

Тренуємося

Щоб створити модель лінійної регресії для передбачення чайових на основі набору даних про чайові, з використанням відповідних кроків попередньої обробки, слід дотримуватись таких кроків:

1. Імпорт необхідних бібліотек:

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
```

1. Завантаження набору даних про чайові:

```
# Завантаження набору даних про чайові
data = pd.read_csv('tips.csv')
```

1. Витягнення ознак (незалежних змінних) та цільової змінної:

```
# Витягнення ознак і цільової змінної
features = data[['total_bill', 'size']]
target = data['tip']
```

1. Виконання необхідних кроків попередньої обробки, наприклад, масштабування ознак:

```
# Виконання кроків попередньої обробки
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)
```

1. Розбиття даних на тренувальний та тестовий набори:

```
# Розбиття даних на тренувальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(feature
s_scaled, target, test_size=0.2, random_state=42)
```

1. Створення екземпляру моделі LinearRegression:

```
# Створення екземпляру LinearRegression
model = LinearRegression()
```

1. Підгонка моделі до тренувальних даних:

```
# Підгонка моделі до тренувальних даних
model.fit(X_train, y_train)
```

1. Прогнозування на тестових даних:

```
# Прогнозування на тестових даних
y_pred = model.predict(X_test)
```

1. Оцінка моделі, обчислення метрик, таких як середньоквадратична помилка та коефіцієнт детермінації:

```
# Обчислення середньоквадратичної помилки
mse = mean_squared_error(y_test, y_pred)

# Обчислення коефіцієнта детермінації
r2 = r2_score(y_test, y_pred)

# Виведення метрик оцінки
```

```
print("Середньоквадратична помилка:", mse)
print("Коефіцієнт детермінації:", r2)
```

У цьому прикладі набір даних про чайові завантажується, ознаки 'total_bill' та 'size' витягуються як незалежні змінні, а стовпець 'tip' є цільовою змінною. Ознаки піддаються попередній обробці, масштабуючи їх за допомогою StandardScaler. Дані розділяються на тренувальний та тестовий набори. Створюється модель LinearRegression та підганяється до тренувальних даних. Прогнози робляться на тестових даних, а потім обчислюються метрики, такі як середньоквадратична помилка (MSE) та коефіцієнт детермінації (R^2), щоб оцінити продуктивність моделі.

Примітка: Кроки попередньої обробки можуть варіюватися в залежності від характеристик вашого набору даних та конкретних вимог вашого аналізу.

Ridge, Lasso та ElasticNet

Ridge, Lasso та ElasticNet - це методи регуляризації, що часто використовуються в лінійній регресії для уникнення перенавчання моделі та покращення її узагальнення. Ці методи додають до функції втрат додатковий термін регуляризації, який штрафує великі значення коефіцієнтів моделі. Ось пояснення кожного методу з прикладами коду на Python:

> ❗ Гіперпараметри - це параметри, які визначають поведінку алгоритму машинного навчання та моделі, але не вивчаються самим алгоритмом під час навчання. Вони задаються заздалегідь та використовуються для налаштування моделі з метою досягнення кращої продуктивності.

Гіперпараметри контролюють процес навчання моделі та її комплексність. Вони впливають на вибір архітектури моделі, регуляризацію, швидкість навчання та багато іншого. Добре налаштовані гіперпараметри можуть допомогти покращити продуктивність моделі та уникнути перенавчання або недонавчання.

Деякі приклади гіперпараметрів включають:

- `alpha` в Ridge, Lasso та ElasticNet регресії, який контролює силу регуляризації.
- `n_estimators` в моделях RandomForest та Gradient Boosting, який визначає кількість дерев у ансамблі.
- `learning_rate` в Gradient Boosting, який впливає на швидкість навчання моделі.
- `hidden_units` та `dropout_rate` в нейронних мережах, які визначають кількість та розмірність шарів.

Для вибору оптимальних гіперпараметрів можна використовувати методи, такі як крос-валідація та GridSearchCV, які дозволяють оцінювати модель з різними комбінаціями гіперпараметрів та знаходити найкращі налаштування для ваших даних та завдання.

1. Ridge Regression:
   Ridge регресія додає до функції втрат термін, який є сумою квадратів коефіцієнтів, помножених на гіперпараметр (alpha). Вона допомагає зменшити вплив незначущих ознак і запобігає перенавчанню.

```
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

# Завантаження набору даних
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
X = data[:, :-1]  # Ознаки
y = data[:, -1]   # Цільова змінна

# Розбиття даних на тренувальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(X, y, t
est_size=0.2, random_state=42)

# Масштабування ознак
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Створення екземпляру Ridge регресії
ridge = Ridge(alpha=0.5)

# Підгонка моделі до тренувальних даних
ridge.fit(X_train_scaled, y_train)

# Прогнозування на тестових даних
y_pred = ridge.predict(X_test_scaled)

# Обчислення середньоквадратичної помилки
mse = mean_squared_error(y_test, y_pred)
print("Середньоквадратична помилка (Ridge):", mse)
```

1. Lasso Regression:
   Lasso регресія виконує відбір ознак шляхом додавання до функції втрат абсолютного значення коефіцієнтів, помноженого на гіперпараметр (alpha). Вона спрямовує коефіцієнти незначущих ознак до нуля, ефективно виконуючи відбір ознак.

```
from sklearn.linear_model import Lasso

# Створення екземпляру Lasso регресії
lasso = Lasso(alpha=0.5)

# Підгонка моделі до тренувальних даних
lasso.fit(X_train_scaled, y_train)

# Прогнозування на тестових даних
y_pred = lasso.predict(X_test_scaled)

# Обчислення середньоквадратичної помилки
mse = mean_squared_error(y_test, y_pred)
print("Середньоквадратична помилка (Lasso):", mse)
```

1. ElasticNet Regression:
   ElasticNet регресія поєднує методи Ridge та Lasso, додаючи до функції втрат як суму квадратів коефіцієнтів, так і абсолютного значення

коефіцієнтів, помноженого на два гіперпараметри (alpha та l1_ratio). Вона надає баланс між методами Ridge та Lasso.

```
from sklearn.linear_model import ElasticNet

# Створення екземпляру ElasticNet регресії
elasticnet = ElasticNet(alpha=0.5, l1_ratio=0.5)

# Підгонка моделі до тренувальних даних
elasticnet.fit(X_train_scaled, y_train)

# Прогнозування на тестових даних
y_pred = elasticnet.predict(X_test_scaled)

# Обчислення середньоквадратичної помилки
mse = mean_squared_error(y_test, y_pred)
print("Середньоквадратична помилка (ElasticNet):", mse)
```

У всіх трьох прикладах дані завантажуються, розбиваються на тренувальний та тестовий набори, ознаки масштабуються за допомогою StandardScaler. Потім створюється екземпляр кожного методу регресії, і модель підгоняється до тренувальних даних. Прогнози робляться на тестових даних, і обчислюється середньоквадратична помилка для оцінки продуктивності моделі.

Ви можете експериментувати з різними значеннями гіперпараметрів (alpha та l1_ratio), щоб побачити їх

вплив на продуктивність моделі та значення коефіцієнтів.

Cross-validation

Cross-validation є методом оцінки продуктивності моделі, який дозволяє оцінити її здатність до узагальнення на нові дані. Він використовується для оцінки моделі на різних підмножинах даних, що дозволяє отримати більш надійну оцінку її продуктивності.

Основні принципи та концепції cross-validation такі:

1. Розділення даних: Спочатку дані розділяються на K рівних частин, які називаються "фолдами". Зазвичай використовують значення K=5 або

K=10.

2. Крос-валідаційні ітерації: Для кожної ітерації модель навчається на K-1 фолдах і оцінюється на залишковому фолді. Це означає, що кожен фолд використовується як тестовий набір один раз.
3. Оцінка продуктивності: Після кожної ітерації отримується оцінка продуктивності моделі, яка може бути усереднена для отримання загальної оцінки.
4. Вибір найкращої моделі: Після проведення всіх ітерацій можна порівняти продуктивність моделей та обрати ту, яка дає найкращі результати.

Принципи cross-validation можна реалізувати за допомогою різних методів. Один з найпоширеніших методів - це "K-fold Cross-Validation", який розділяє дані на K фолдів і проводить K ітерацій.

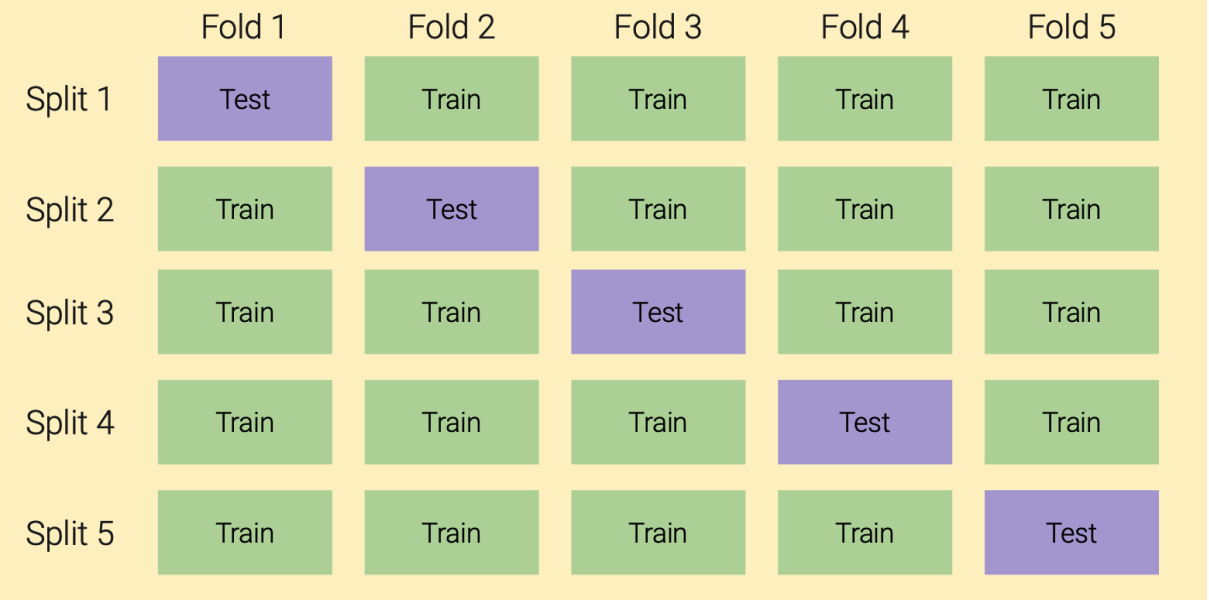
Ось приклади коду на Python для реалізації K-fold Cross-Validation з використанням бібліотеки scikit-learn:

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Завантаження даних
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
X = data[:, :-1]  # Ознаки
y = data[:, -1]   # Цільова змінна

# Ініціалізація KFold
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Ініціалізація моделі
model = LinearRegression()

# Список для збереження MSE
mse_scores = []

# K-fold Cross-Validation
```

```
for train_index, test_index in kfold.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Підгонка моделі
    model.fit(X_train, y_train)

    # Прогнозування на тестових даних
    y_pred = model.predict(X_test)

    # Розрахунок MSE
    mse = mean_squared_error(y_test, y_pred)
    mse_scores.append(mse)

# Обчислення середнього MSE
mean_mse = np.mean(mse_scores)

# Виведення результату
print("Середнє MSE: ", mean_mse)
```

У цьому прикладі дані розділяються на 5 фолдів за допомогою KFold, модель LinearRegression навчається на тренувальних даних кожного фолду, а потім оцінюється на відповідному тестовому фолді. Середнє значення MSE (середньоквадратичної помилки) обчислюється для оцінки продуктивності моделі.

Cross-validation допомагає отримати більш об'єктивну оцінку продуктивності моделі, оскільки вона враховує варіацію у даних та допомагає уникнути перенавчання.



GridSearchCV

GridSearchCV є інструментом в бібліотеці scikit-learn, який дозволяє автоматично шукати найкращі гіперпараметри для моделі, використовуючи крос-валідацію на заданій сітці гіперпараметрів. Це допомагає знаходити оптимальні значення гіперпараметрів, які найкраще підходять для вашої моделі і даних.

Основні принципи та концепції GridSearchCV:

1. Визначення сітки гіперпараметрів: Спочатку ви визначаєте набір гіперпараметрів, для яких потрібно провести пошук. Це можуть бути значення різних гіперпараметрів, які ви хочете перевірити.
2. Крос-валідація: Для кожної комбінації гіперпараметрів GridSearchCV виконує крос-валідацію, розбиваючи дані на тренувальний та тестовий набори та обчислюючи метрику продуктивності (наприклад, середньоквадратичну помилку) на кожному фолді.
3. Вибір найкращих гіперпараметрів: Після завершення пошуку GridSearchCV повертає комбінацію гіперпараметрів, яка має найкращу продуктивність відповідно до заданої метрики.

Ось приклад коду на Python для використання GridSearchCV для пошуку найкращих гіперпараметрів моделі:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge
```

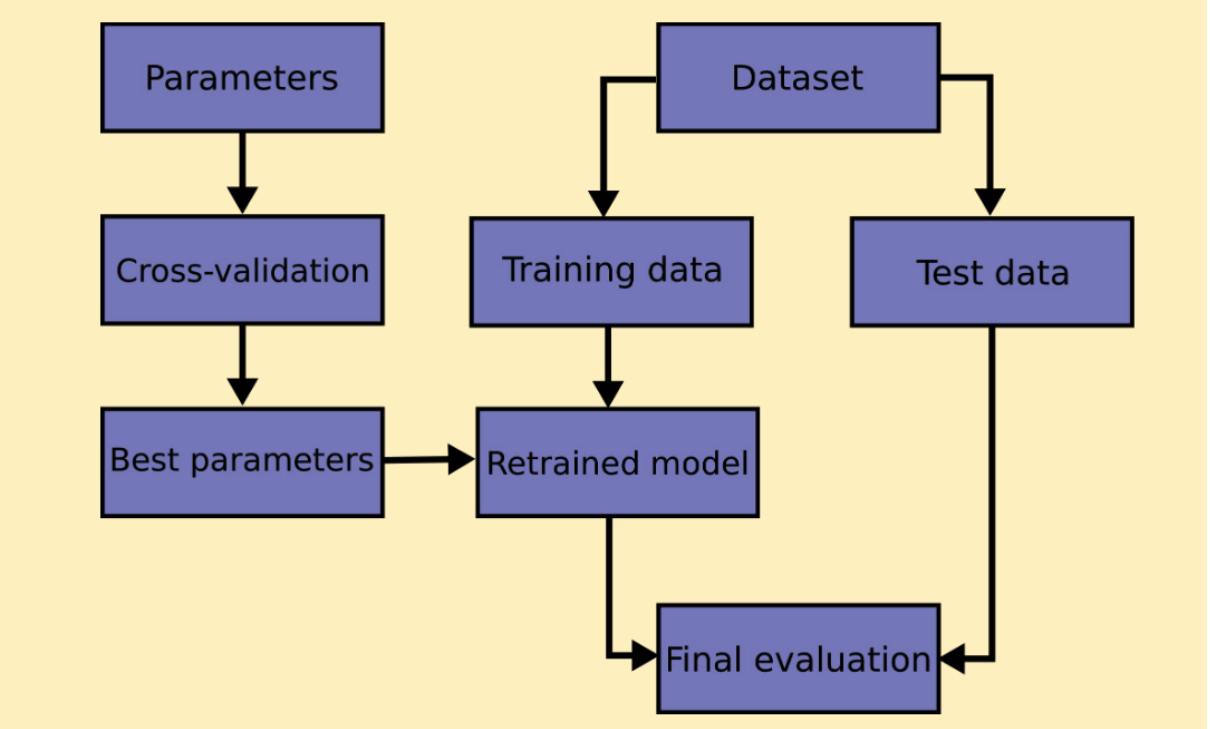
```
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
import numpy as np

# Завантаження даних
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
X = data[:, :-1]  # Ознаки
y = data[:, -1]   # Цільова змінна

# Масштабування ознак
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Визначення моделі
model = Ridge()

# Визначення сітки гіперпараметрів
param_grid = {'alpha': [0.1, 1.0, 10.0]}

# Використання GridSearchCV
grid_search = GridSearchCV(model, param_grid, cv=3)

# Пошук найкращих гіперпараметрів
grid_search.fit(X_scaled, y)

# Отримання найкращих гіперпараметрів
best_params = grid_search.best_params_

# Побудова моделі з найкращими гіперпараметрами
best_model = Ridge(alpha=best_params['alpha'])

# Підгонка моделі з найкращими гіперпараметрами
best_model.fit(X_scaled, y)

# Прогнозування на нових даних
new_data = np.array([[2, 3, 4]])
new_data_scaled = scaler.transform(new_data)
prediction = best_model.predict(new_data_scaled)
```

```
# Виведення результату
print("Прогнозоване значення:", prediction)
```

У цьому прикладі ми використовуємо модель Ridge для прикладу, але ви можете замінити його на будь-яку іншу модель, для якої ви шукаєте оптимальні гіперпараметри. Ми визначаємо сітку гіперпараметрів, в даному випадку - список значень параметра "alpha" для Ridge моделі. GridSearchCV використовує крос-валідацію з 3 фолдами для оцінки моделі з різними комбінаціями гіперпараметрів. Після пошуку ми отримуємо найкращі гіперпараметри та побудовуємо модель з цими гіперпараметрами. Нарешті, ми можемо використовувати цю модель для здійснення прогнозу на нових даних.

GridSearchCV дозволяє вам знайти найкращі гіперпараметри для вашої моделі автоматично, спрощуючи процес налаштування моделі та покращуючи її продуктивність.

How to get perfect data



Polynomial features

Polynomial features використовується для створення нових ознак шляхом перетворення початкових ознак у поліноміальні комбінації. Це дозволяє моделі здатніше розуміти та захоплювати нелінійні залежності між ознаками та цільовою змінною.

Поліноміальні ознаки створюються шляхом перетворення початкових ознак у всі можливі комбінації степенів. Наприклад, якщо ми маємо початкові ознаки `x1` і `x2`, то поліноміальні ознаки можуть включати `x1`, `x2`, `x1^2`, `x2^2`, `x1 * x2` і т.д. За допомогою цих поліноміальних ознак можна збільшити експресивність моделі та урахувати нелінійні взаємодії між ознаками.

Ось приклад коду на Python для створення поліноміальних ознак за допомогою PolynomialFeatures з бібліотеки scikit-learn:

```
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

# Початкові ознаки
X = np.array([[1, 2], [3, 4], [5, 6]])

# Ініціалізація PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)

# Трансформація початкових ознак в поліноміальні ознаки
X_poly = poly.fit_transform(X)

# Виведення результату
print(X_poly)
```

У цьому прикладі ми маємо початкові ознаки `X` з двома колонками. За допомогою PolynomialFeatures з параметром `degree=2`, ми створюємо поліноміальні ознаки другого ступеня. `include_bias=False` вказує на те, що необхідно виключити випадковий стовпчик з одиницями (доданий для моделей, які не мають власного зсуву). Результатом є матриця `X_poly`, що містить оригінальні ознаки та їх поліноміальні комбінації.

Створені поліноміальні ознаки можуть бути використані для навчання моделі, такої як лінійна регресія або метод опорних векторів, щоб узгодити більш складні залежності в даних. Важливо враховувати, що збільшення

ступеня поліноміальних ознак може призвести до зростання кількості ознак та обчислювальної складності моделі. Тому потрібно бути уважним і обирати оптимальний ступінь поліноміальних ознак в залежності від конкретної задачі та обсягу даних.

Тренуємося

Ок, для створення поліноміальних ознак на основі даних пакету seaborn про Титанік, спочатку ми мусимо завантажити дані та вибрати необхідні ознаки. Далі застосуємо PolynomialFeatures для створення поліноміальних ознак.

Ось приклад коду на Python для цього:

```
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures

# Завантаження даних про Титанік з пакету seaborn
titanic_data = sns.load_dataset('titanic')

# Вибір необхідних ознак
selected_features = ['class', 'sex', 'fare', 'embarked', 'a
ge']

# Відбір підмножини даних з вибраними ознаками
data_subset = titanic_data[selected_features]

# Заповнення відсутніх значень в даних
data_subset.fillna(data_subset.mean(), inplace=True)

# Ініціалізація PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)

# Трансформація початкових ознак в поліноміальні ознаки
data_poly = poly.fit_transform(data_subset)

# Виведення результату
print(data_poly)
```

У цьому прикладі ми спочатку завантажуємо дані про Титанік з пакету seaborn за допомогою `sns.load_dataset('titanic')`. Потім ми вибираємо певні ознаки для аналізу, які зберігаємо у змінній `selected_features`. Далі ми створюємо підмножину даних, що містить лише вибрані ознаки.

Перед застосуванням PolynomialFeatures, ми заповнюємо відсутні значення в даних шляхом заміни їх середніми значеннями за допомогою `data_subset.fillna(data_subset.mean(), inplace=True)`.

Після цього ми ініціалізуємо PolynomialFeatures з параметром `degree=2` та `include_bias=False`, щоб уникнути додавання зсуву. Потім застосовуємо `poly.fit_transform(data_subset)`, щоб отримати поліноміальні ознаки для вибраних ознак.

Результатом буде матриця `data_poly`, що містить початкові ознаки та їх поліноміальні комбінації.

Зверніть увагу, що перед використанням PolynomialFeatures бажано виконати передпроцесинг даних, такий як видалення відсутніх значень або масштабування ознак, в залежності від конкретної задачі та типу даних.

Заповнення відсутніх значень із середніми значеннями

```
import seaborn as sns
import matplotlib.pyplot as plt

# Завантаження даних про Титанік з пакету seaborn
titanic_data = sns.load_dataset('titanic')

# Вибір необхідних ознак
selected_features = ['class', 'sex', 'fare', 'embarked', 'a
ge']

# Відбір підмножини даних з вибраними ознаками
data_subset = titanic_data[selected_features]

# Заповнення відсутніх значень в даних
data_subset.fillna(data_subset.mean(), inplace=True)

# Побудова графіка до поліноміального аналізу
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
```

```
sns.scatterplot(data=titanic_data, x='fare', y='age', hue
='survived')
plt.title('До поліноміального аналізу')

# Побудова графіка після поліноміального аналізу
plt.subplot(1, 2, 2)
sns.scatterplot(data=data_subset, x='fare', y='age', hue='c
lass')
plt.title('Після поліноміального аналізу')

# Відображення графіків
plt.tight_layout()
plt.show()
```

Цей код побудує два графіки. Перший графік показує розподіл віку пасажирів в залежності від цін на квитки перед застосуванням поліноміального аналізу. Другий графік показує ті самі ознаки після застосування поліноміального аналізу. Обидва графіки використовують різні кольори, щоб показати додаткову інформацію, таку як виживання або клас пасажира.

Logistic regression

Логістична регресія (Logistic Regression) - це статистичний алгоритм машинного навчання, що використовується для моделювання ймовірності категоріального виходу залежно від вхідних ознак. Вона широко використовується для бінарної класифікації, коли потрібно визначити, до якої категорії належить об'єкт, але також може бути застосована до задачі мультикласової класифікації.

Основна ідея логістичної регресії полягає в тому, що вона моделює логарифм відношення шансів (логіт) для належності до певного класу в залежності від вхідних ознак. Відношення шансів відображає ймовірність відношення події до ймовірності неподії, і логіт перетворює його в лінійну функцію, що може бути використана для прогнозування категорії.

Логістична регресія може бути реалізована за допомогою методу максимальної правдоподібності, де модель навчається підбирати коефіцієнти, що максимізують ймовірність спостережень у навчальному наборі.

Ось приклад коду на Python для створення і навчання моделі логістичної регресії з використанням пакету scikit-learn:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pandas as pd

# Завантаження даних
data = pd.read_csv('data.csv')
X = data.drop('target', axis=1)
y = data['target']

# Розділення даних на тренувальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(X, y, t
est_size=0.2, random_state=42)

# Ініціалізація та навчання моделі логістичної регресії
model = LogisticRegression()
model.fit(X_train, y_train)

# Прогнозування на тестових даних
y_pred = model.predict(X_test)

# Оцінка точності моделі
accuracy = accuracy_score(y_test, y_pred)
print("Точність моделі: ", accuracy)
```

У цьому прикладі ми спочатку завантажуємо дані та розділяємо їх на вхідні ознаки `X` та цільову змінну `y`. Потім ми розділяємо дані на тренувальний та тестовий набори за допомогою `train_test_split`. Далі ми ініціалізуємо модель логістичної регресії, навчаємо її на тренувальних даних за допомогою `fit` та прогнозуємо значення на тестових даних за допомогою `predict`. Нарешті, ми оцінюємо точність моделі, порівнюючи прогнозовані значення з фактичними значеннями тестового набору.

Логістична регресія є потужним інструментом для класифікації та може бути використана в багатьох різних задачах, таких як прогнозування

виживання пасажирів Титаніка, медична діагностика, оцінка ризику та багато інших.

PCA

Principal Component Analysis (PCA) або аналіз головних компонентів є методом зменшення розмірності даних, який використовується для виявлення основних шаблонів або кореляцій у великій кількості змінних. Він дозволяє перетворити набір вхідних ознак в новий набір незалежних лінійних комбінацій, які називаються головними компонентами.

Основні принципи PCA:

1. Стандартизація даних: Спочатку дані стандартизуються, щоб забезпечити однаковий масштаб для всіх ознак. Це важливо, оскільки PCA базується на коваріаціях між ознаками.
2. Обчислення коваріаційної матриці: Потім обчислюється коваріаційна матриця, яка показує ступінь взаємозв'язку між ознаками. Вона використовується для визначення головних компонент.
3. Розрахунок головних компонент: Головні компоненти обчислюються шляхом розкладу коваріаційної матриці або за допомогою сингулярного розкладу. Кожна головна компонента є лінійною комбінацією вихідних ознак.
4. Вибір головних компонент: Головні компоненти вибираються згідно з їхньою внесеною дисперсією. Зазвичай вибираються перші к головних компонент, які пояснюють найбільшу частку дисперсії.

Ось приклад коду на Python для використання PCA з використанням бібліотеки scikit-learn:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Завантаження даних
data = pd.read_csv('data.csv')

# Вибір необхідних ознак
selected_features = ['feature1', 'feature2', 'feature3']
```

```
# Відбір підмножини даних з вибраними ознаками
data_subset = data[selected_features]

# Стандартизація даних
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data_subset)

# Використання PCA
pca = PCA(n_components=2)
principal_components = pca.fit_transform(data_scaled)

# Отримання головних компонент
principal_df = pd.DataFrame(data=principal_components, colu
mns=['PC1', 'PC2'])

# Додавання цільової змінної до головних компонент
principal_df['target'] = data['target']

# Виведення результату
print(principal_df.head())
```

У цьому прикладі ми спочатку завантажуємо дані та вибираємо необхідні ознаки. Потім ми стандартизуємо дані, щоб забезпечити однаковий масштаб для всіх ознак. Після цього ми використовуємо PCA з параметром `n_components=2`, щоб зберегти дві головні компоненти, які пояснюють найбільшу частку дисперсії. Нарешті, ми створюємо DataFrame `principal_df`, який містить головні компоненти та цільову змінну.

PCA є потужним інструментом для зменшення розмірності даних та виявлення основних залежностей між ознаками. Він може бути використаний для візуалізації даних, вилучення важливих ознак або передпроцесингу даних перед застосуванням моделей машинного навчання.

K-means clustering

K-means кластеризація - це алгоритм некерованої кластеризації, який використовується для групування схожих об'єктів у підмножини, які

називаються кластерами. Кожен кластер характеризується своїм центроїдом, який є середнім значенням всіх об'єктів у цьому кластері.

Принцип роботи K-means:

1. Вибір кількості кластерів: Спочатку необхідно визначити, скільки кластерів потрібно сформувати. Це може бути виконано заздалегідь, якщо ви маєте попередні знання про дані, або можна використовувати евристичні підходи, такі як метод "ліктя" (elbow method) або індекс силуету (silhouette index).
2. Ініціалізація центроїдів: Початкові центроїди вибираються випадково або за допомогою інших підходів. Вони можуть бути обрані серед вихідних об'єктів або за допомогою певного пошуку.
3. Призначення об'єктів до кластерів: Кожен об'єкт призначається до найближчого центроїда за допомогою метрики відстані, зазвичай евклідової відстані.
4. Перерахунок центроїдів: Після призначення об'єктів до кластерів, центроїди перераховуються шляхом обчислення середнього значення всіх об'єктів у кожному кластері.
5. Повторення кроків 3 і 4: Кроки призначення та перерахунку центроїдів повторюються досягнення збіжності або заданої кількості ітерацій.
6. Завершення алгоритму: Після збіжності або досягнення заданої кількості ітерацій, алгоритм завершується, і ми отримуємо кінцеві кластери та їхні центроїди.

Ось приклад коду на Python для використання K-means кластеризації з використанням бібліотеки scikit-learn:

```
from sklearn.cluster import KMeans
import pandas as pd

# Завантаження даних
data = pd.read_csv('data.csv')

# Вибір необхідних ознак
selected_features = ['feature1', 'feature2', 'feature3']

# Відбір підмножини даних з вибраними ознаками
```

```
data_subset = data[selected_features]

# Ініціалізація та навчання K-means моделі
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(data_subset)

# Отримання прогнозованих міток кластерів
cluster_labels = kmeans.labels_

# Виведення результату
print(cluster_labels)
```

У цьому прикладі ми спочатку завантажуємо дані та вибираємо необхідні ознаки. Потім ми ініціалізуємо K-means модель з параметром `n_clusters=3`, що вказує кількість кластерів, які ми хочемо сформувати. Після цього ми навчаємо модель на вибраних ознаках за допомогою `fit`. Нарешті, ми отримуємо прогнозовані мітки кластерів за допомогою `labels_`.

K-means кластеризація є корисним методом для групування даних, знаходження прихованих залежностей та аналізу схожих об'єктів. Вона широко використовується в багатьох областях, таких як сегментація зображень, аналіз соціальних мереж, маркетинговий аналіз та багато інших.