KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 6 урок > Оконные функции и интерактивные графики

> Оглавление

> Оконные функции

1. Скользящее среднее

Иногда (например, при работе с временными данными) нужно произвести агрегирующие вычисления, захватывающие определённый промежуток данных (не всю колонку). То есть мы будем работать в определённом «окне» значений колонки. Окна бывают разные, простой вариант — скользящее.

Посмотрим на скользящее среднее более подробно. Это один из способов сгладить временной ряд, чтобы избавиться от шума в данных и более точно увидеть линию общего тренда.

В pandas есть уже готовая реализация скользящего окна — метод `rolling()`. Он принимает несколько параметров, первый и самый важный из которых `window` — размер окна, также называемый шириной окна. Данный параметр отвечает за число наблюдений, которые используются для подсчета скользящего значения (обычно — скользящего среднего). К примеру,

$$SMA_t = \frac{1}{n}\sum_{i=0}^{n-1} x_{t-i} = \frac{x_t + x_{t-1} + ... + x_{t-(n-1)}}{n}$$

где

n — размер окна,

t — момент времени.

Предположим, имеется пять наблюдений:

```
df = pd.DataFrame({'value': [0, 1, 2, 3, 4]})
df
```

```
  value
0     0
1     1
```

```
2     2
3     3
4     4
```

Для подсчета скользящего среднего с размером окна 2 вычисления будут выглядеть следующим образом:

$$SMA_t = \frac{x_t + x_{t-1}}{2}$$

где *xt* — значение в текущий момент времени,

xt−1 — в предыдущий.

```
df.rolling(window=2).mean()
```

```
   value
0    NaN
1    0.5
2    1.5
3    2.5
4    3.5
```

Обратите внимание, что теперь на месте самого первого значения стоит NaN, поскольку указанный размер окна 2 подразумевает наличие двух значений для подсчета. Вполне логично, что скользящее среднее для первого наблюдения вычислить не получится, поскольку других значений перед ним нет. Если увеличить размер окна до 3, то NaN будет стоять и вместо второго наблюдения, и так далее.

Другой пример:

```
conversion_df['Conversion_rate'].head(10)
```

```
Date
2014-01-01    0.229167
2014-01-02    0.229968
2014-01-03    0.192201
2014-01-04    0.188793
2014-01-05    0.179035
2014-01-06    0.170029
2014-01-07    0.172693
2014-01-08    0.186263
2014-01-09    0.185567
2014-01-10    0.181296
Name: Conversion_rate, dtype: float64
```

Для скользящего окна с периодом 5 значение 2014-01-05 будет вычисляться по значениям с 01 по 05 включительно. А для 2014-01-06 — по значениям с 02 по 06.

```
conversion_df['Conversion_rate'].rolling(5).mean().head(10)
```

```
Date
2014-01-01         NaN
2014-01-02         NaN
2014-01-03         NaN
2014-01-04         NaN
2014-01-05    0.203833
2014-01-06    0.192005
2014-01-07    0.180550
2014-01-08    0.179363
2014-01-09    0.178717
2014-01-10    0.179170
Name: Conversion_rate, dtype: float64
```

Еще один вариант использования `rolling()` — **центрированное скользящее среднее**. В таком случае используются наблюдения до, после и во время t*t*. В pandas за это отвечает параметр `center`, который по умолчанию равен False.

Для подсчета центрированного среднего с размером окна 3:

$$SMA_t = \frac{x_{t-1} + x_t + x_{t+1}}{3}$$

где x_t — значение в текущий момент времени, x_{t-1} — в предыдущий, x_{t+1} — последующий.

```
df.rolling(window=3, center=True).mean()
# (0+1+2)/3=1
# (1+2+3)/3=2
```

```
  value
0   NaN
1   1.0
2   2.0
3   3.0
4   NaN
```

С помощью метода `rolling()` можно посчитать не только скользящее среднее, но и, например, скользящую медиану. Для этого после метода `rolling()` используется метод `median()` вместо `mean()`.

Избавление от NaN'ов

Если расчёт каждого нового значения ровно по периоду не важен, можно использовать параметр `min_periods`. Он принимает число, которое указывает минимальное количество значений, которое необходимо, чтобы посчитать результат.

```
conversion_df['Conversion_rate'].rolling(5, min_periods=1).mean().head(7)
```

```
Date
2014-01-01    0.229167
2014-01-02    0.229567
2014-01-03    0.217112
2014-01-04    0.210032
2014-01-05    0.203833
2014-01-06    0.192005
2014-01-07    0.180550
Name: Conversion_rate, dtype: float64
```

Документация

> 2. Экспоненциальное сглаживание

Следующий шаг — экспоненциальное сглаживание. В предыдущем подходе (скользящее среднее) использовались *n* последних наблюдений, все они имели равный вес. В данном случае веса экспоненциально уменьшаются, т.е. более «старые» события имеют меньший вес, а для подсчета используются все имеющиеся наблюдения.

$$EMA_t = \alpha * P_t + (1 - \alpha) * EMA_{t-1}$$

где:

- *α* — веса от 0 до 1; чем выше, тем больший вес имеют новые значения и меньше старые;
- *Pt* — значение в момент времени *t;*
- *EMAt*−1 — значение скользящего среднего в момент *t*−1.

Более подробно про расчет EMA (Exponential Moving Average, экспоненциального скользящего среднего) можно почитать здесь.

Для использования экспоненциального скользящего окна в pandas есть метод `ewm()`. Например,

```
  value
0     0
1     1
2     2
3     3
4     4
```

```
df.ewm(span=2).mean()
```

```
      value
0  0.000000
1  0.750000
2  1.615385
```

```
3    2.550000
4    3.520661
```

Параметр `span` определяет, насколько сильно учитываются прошлые значения в расчете текущего среднего: больший `span` означает, что в расчет включается больше прошлых значений, что приводит к более плавному сглаживанию. Меньший `span` означает, что в расчет включается меньше прошлых значений, а значит, скользящее значение быстрее реагирует на изменения в данных.

> Итерация по нескольким спискам одновременно

Время от времени вам будет нужно пройтись по нескольким спискам одновременно — то есть получать элементы с одним индексом из каждого из них. Можно сделать это так:

```
nums = [3, 5, 7]
letters = ['all', 'for', 'exoplanets colonization']

for i in range(len(nums)):
    num = nums[i]
    letter = letters[i]
    # дальше делаем что-то с этими переменными
```

Но есть более удобный способ — функция `zip()`. По сути она позволяет перебирать элементы списков одновременно, возвращая на каждой итерации кортеж с элементами из одинаковой позиции соответствующих списков.

```
nums = [3, 5, 7]
letters ['all', 'for', 'exoplanets colonization']

for i in zip(nums, letters):
    num = i[0]
```

```
    letter = i[1]
    # дальше делаем что-то с этими переменными
```

Кажется, что лучше не стало — в такой записи действительно стало ненамного лучше. Но мы можем сразу извлечь `num` и `letter` из кортежа с помощью записи:

```
nums = [3, 5, 7]
letters ['all', 'for', 'exoplanets colonization']

for num, letter in zip(nums, letters):
    # дальше делаем что-то с этими переменными
```

Связано это с тем, что в питоне можно распаковывать кортежи и списки в переменные, например:

```
num, letter = (3, 'all')  # num будет равно 3, а letter - 'al
l'
```

Здесь мы присваиваем элементы из кортежа `(3, 'all')` в соответствующие переменные.

Больше информации

> Преобразование чисел в категории

Если вам нужно перейти от чисел к категориям, воспользуйтесь функцией `pd.cut`. Она принимает массив значений и число интервалов/список из границ интервалов:

```
values.head()
```

```
0    1
1    6
2    1
3    2
4    4
dtype: int64
```

```
pd.cut(values.head(), 3)
```

```
0    (0.995, 2.667]
1      (4.333, 6.0]
2    (0.995, 2.667]
3    (0.995, 2.667]
4    (2.667, 4.333]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]
```

```
pd.cut(values.head(), [0, 3, 5, 10])
```

```
0     (0, 3]
1    (5, 10]
2     (0, 3]
3     (0, 3]
4     (3, 5]
dtype: category
Categories (3, interval[int64]): [(0, 3] < (3, 5] < (5, 10]]
```

Как видите, числа заменились на интервалы.

Изменение названий

Для добавления своих названий используется аргумент `labels`, куда подаётся список из названий интервалов:

```
pd.cut(values.head(), [0, 3, 5, 10], labels=['low', 'medium', 'high'])
```

```
0       low
1      high
2       low
3       low
4    medium
dtype: category
Categories (3, object): [low < medium < high]
```

Документация

> Объединение таблиц по нескольким полям

Если указать в параметре `on` функции `pd.merge` список из колонок, то произойдёт объединение по их комбинации. То есть будут объединены строки, в которых совпадают значения во всех указанных в `on` колонках:

```
# Объединение по комбинации колонок Company Id и Company Name
orders_with_sales_team = pd.merge(order_leads, sales_team, on
=['Company Id','Company Name'])
```

Больше информации

> Общая настройка графиков

Многие настройки рисования можно установить 1 раз для скрипта (обычно в его начале), избавившись таким образом от повторов:

```
# Будет увеличен размер шрифта и графиков, фон рисунков стане
т белым и добавится сетка

sns.set(
    font_scale=2,
    style="whitegrid",
```

```
    rc={'figure.figsize':(20,7)}
        )
```

Документация

> Кастомизация pandas графика

Хорошие графики обладают подписанными осями, заголовком и начинаются от 0 при сравнении между собой нескольких значений (например, на барплоте)!

Есть несколько способов провести кастомизацию графика, построенного с помощью метода `plot()` из библиотеки `pandas`.

Через объект графика

При создании графика возвращается объект, который хранит о нём информацию. Через него можно задать параметры кастомизации:

```
ax = conversion_df.plot()  # создадим объект графика

ax.set_xlabel('Date of orders')  # Имя оси x
ax.set_ylabel('Conversion rate')  # Имя оси y
ax.set_title('Conversion rate by date')  # Заголовок графика

y_labels = [str(int(i * 100)) + '%' for i in ax.get_yticks()]
# Подготовим пользовательские метки для значений оси y

ax.set_yticklabels(y_labels)  # Установим новые метки оси y
ax.set_xticklabels(labels=conversion_df.index, rotation=90)
# Установим новые метки оси x и повернем подписи перпендикуля
рно

sns.despine()  # Избавимся от части рамок на графике
```

Через методы matplotlib

```
ax = conversion_df.plot()

plt.xlabel('Date of orders')
plt.ylabel('Conversion rate')
plt.title('Conversion rate by date')

y_labels = [str(int(i * 100)) + '%' for i in ax.get_yticks()]

ax.set_yticklabels(y_labels)
ax.set_xticklabels(labels=conversion_df.index, rotation=90)

sns.despine()
```

Больше информации

> Функция визуализации sns.distplot

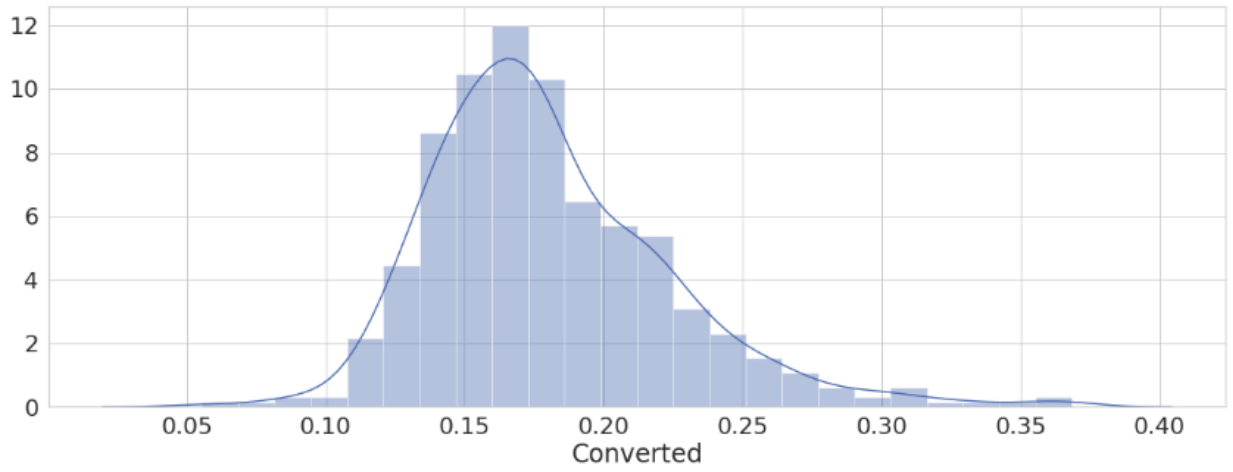
Чтобы отрисовать распределение значений (гистограмму), можно использовать функции `sns.displot()` и `sns.dis t plot()`. Вторую разработчики seaborn пометили как устаревшую, но она ещё работает и имеет интересный параметр `kde`.

kde

`kde` (Kernel Density Estimation) — аргумент добавляет аппроксимацию (упрощение) распределения, по умолчанию `True`. При этом распределение шкалируется (масштабируется) и становится графиком плотности распределения вероятности (probability density estimation). По оси `y` идет плотность вероятности:

```
sns.distplot(conversion_by_sales['Converted'])
```

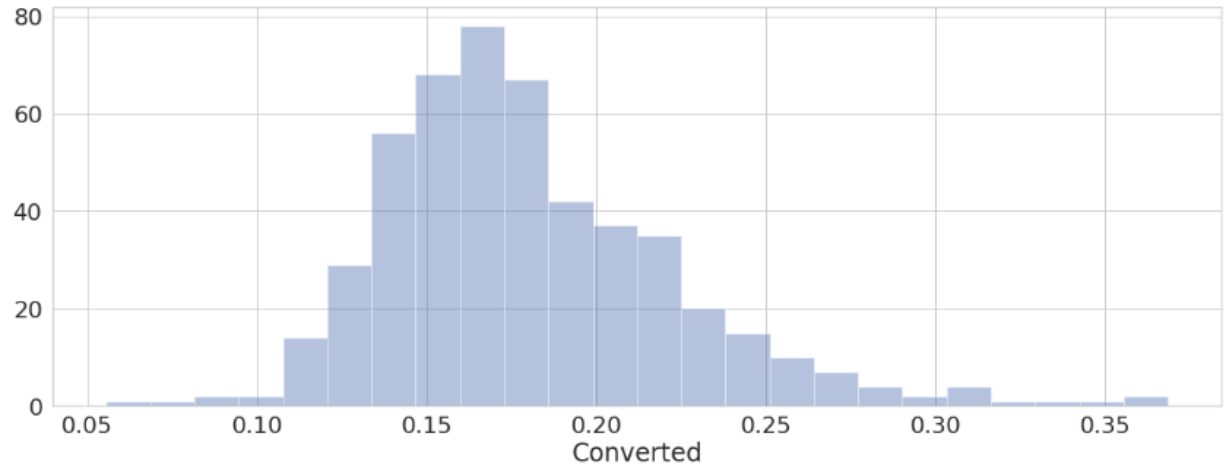
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5e82ce6cc0>
```



Если параметр `kde` равен `False`, этого не происходит, значение на оси `y` означает количество:

```
sns.distplot(conversion_by_sales, kde=False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5e9bec8470>
```



аналогичный график можно получить с помощью `sns.histplot()` и `sns.displot()`.

Документация

> Библиотека plotly

Библиотека для создания интерактивных графиков. Например:

```
import plotly.express as px


px.line(conversion_df, conversion_df.index, conversion_df['Co
nversion_rate'])
```

Документация

Больше информации

> Модули

Модульность — важное свойство программ, которое обеспечивается языком программирования. Мы можем разбить код программы по нескольким обособленным по смыслу файлам. Это значительно облегчает разработку и тестирование сколько-нибудь сложных приложений.

Небольшой пример: вы написали 10 функций, решающих ваши задачи. Держать их в одном файле и там же вызывать — не очень хорошая идея, потому что получается бардак. Хорошим шагом будет разделение функций от скриптов, где вы непосредственно применяете их

Реализация

Каждый питоновский скрипт (с расширением `py`, не юпитер ноутбук) является модулем. Чтобы получить доступ к его содержимому необходимо произвести импорт. При импортировании файла он целиком исполняется — как если бы вы выполнили его в юпитер ноутбуке.

Для получения своего модуля:

- создайте файл
- напишите там код, который хотите (стандартный вариант — функцию)
- сохраните файл с английским названием, чтобы он начинался с буквы и не содержал пробелов, в конце `.py`

Расположение модулей

Чтобы модуль можно было импортировать, он должен быть виден питону. Он смотрит модули в нескольких местах:

- питоновская папка, куда устанавливаются библиотеки — она находится в глубинах питона;
- папка, откуда запущен питон — просто рабочая папка, где вы работаете с ноутбуком;
- кастомные места, прописанные в `sys.path` — любое место, которое вы запишете.

Самый простой вариант — держать ваш файл-модуль в той же папке, где работаете

Виды импорта

Помимо импорта видов

`import pandas as pd`

и

`import os`

Существуют другие варианты:

- `from my_module import my_function` — импортировать функцию `my_function`, содержащуюся в `my_module.py`. После этого её можно использовать в коде как `my_function`

- `from my_module import *` Краткая запись для импорта всех имён из модуля, `my_function` также можно использовать сразу в коде по её имени. Все остальные переменные/функции тоже были выгружены из модуля. Не рекомендуем эту практику при импорте большинства модулей, так как скрипт захламляется именами из модуля (меньше свободных имён для ваших переменных). Но вполне валидно использовать для модулей с небольшим числом имён.

Больше информации