*Дата последней сборки документа:*
*31 марта 2020 г. в 20:35*

# Машинное обучение и анализ данных. Заметки

*Подвойский А.О.*

Здесь приводятся заметки по некоторым вопросам, касающимся машинного обучения, анализа данных, программирования на языках `Python`, `R` и прочим сопряженным вопросам так или иначе, затрагивающим работу с данными.

## Содержание



## 1. Установка пакета `xgboost` на `Windows`

Устанавливать пакет `xgboost` рекомендуется с помощью следующей команды

```
conda install -c anaconda py-xgboost
```

Существует альтернативный способ установки пакета `xgboost` (разумеется он работает и для других пакетов). Для начала требуется вывести список доступных каналов 1, по которым будет проводиться поиск интересующего пакета (в данном случае пакета `xgboost`), а затем можно воспользоваться конструкцией

```
anaconda search -t conda xgboost
```

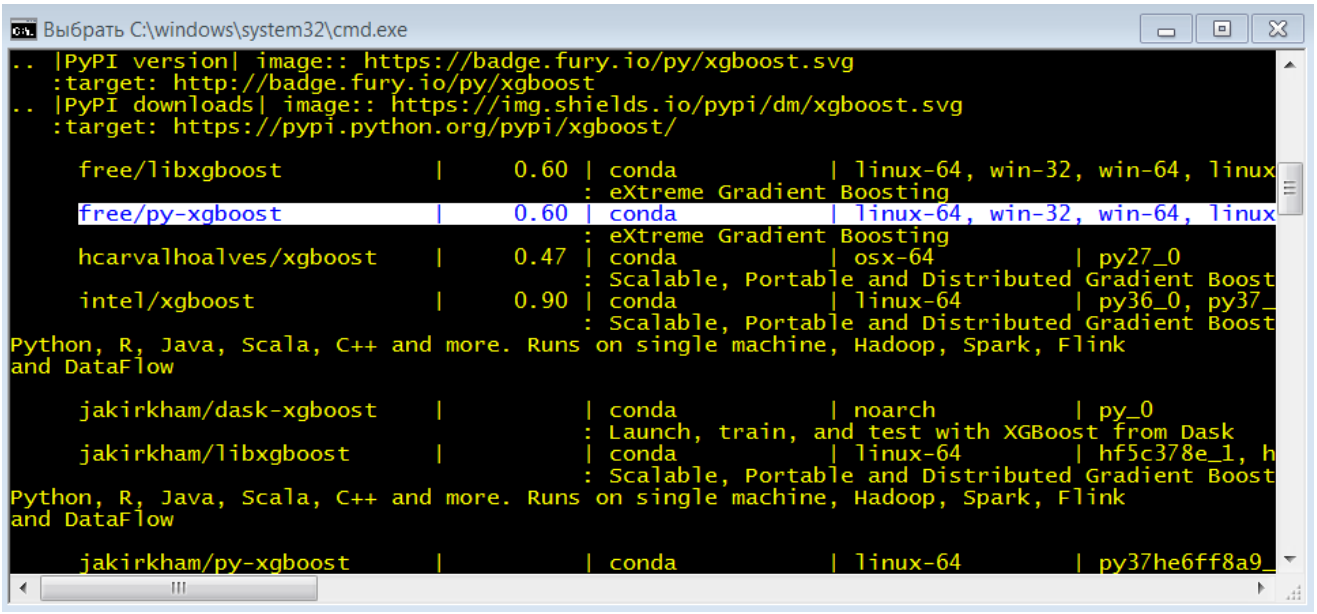


Рис. 1. Окно командной оболочки `cmd.exe` со списком доступных каналов, по которым будет проводиться поиск пакета `xgboost`

После, выбрав канал, можно приступать к установке пакета

```
conda install -c free py-xgboost
```

## 2. Области видимости в языке Python

Когда мы говорим о поиске значения имени применительно к программному коду, под термином *область видимости* подразумевается *пространство имен* – то есть место в программном коде, где имени было присвоено значение [1].

В любом случае область видимости переменной (где она может использоваться) всегда определяется местом, где ей было присвоено значение.

---

*Замечание*

Термины «*область видимости*» и «*пространство имен*» можно использовать как синонимичные

---

При каждом вызове функции создается новое *локальное пространство имен*. Это пространство имен представляет локальное окружение, содержащее имена параметров функции, а также имена переменных, которым были присвоены значения в теле функции.

По умолчанию операция присваивания создает локальные имена (это поведение можно изменить с помощью `global` или `local`).

Схема разрешения имен в языке `Python` иногда называется *правилом LEGB*[1] [1, стр. 477]:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – в *локальной*, затем в *локальной области любой объемлющей функции* или в выражении `lambda`, затем в *глобальной* и, наконец, во *встроенной*. Поиск завершается, как только будет найдено первое подходящее имя.

[1] Local, Enclosing, Global, Built-in

- Когда внутри функции выполняется операция присваивания `a=10` (а не обращения к имени внутри выражения), интерпретатор всегда создает или изменяет имя в *локальной области видимости*, если в этой функции оно не было объявлено глобальным или нелокальным.

Пример

```
# глобальная область видимости
X = 99

def func(Y):  # Y и Z локальные переменные
    # локальная область видимости
    Z = X + Y # X - глобальная переменная
    return Z

func(1)  # Y = 1
```

Переменные `Y` и `Z` являются *локальными* (и существуют только во время выполнения функции), потому что присваивание значений обоим именам осуществляется внутри определения функции: присваивание переменной `Z` производится с помощью инструкции `=`, а `Y` – потому что аргументы всегда передаются через операцию присваивания.

Когда внутри функции выполняется операция присваивания значения переменной, она всегда выполняется в *локальном пространстве имен функции*

```
a = 10  # глобальная область видимости

def f():
    a = 100  # локальная область видимости
    return a
```

В результате переменная `a` в теле функции ссылается на совершенно другой объект, содержащий значение 100, а не тот, на который ссылается внешняя переменная.

Переменные во вложенных функциях привязаны к *лексической области видимости*. То есть поиск имени переменной начинается в *локальной области видимости* и затем последовательно продолжается во всех *объемлющих областях видимости внешних функций*, в направлении от внутренних к внешним.

Если и в этих *пространствах имен* искомое имя не будет найдено, поиск будет продолжен в *глобальном пространстве имен*, а затем во *встроенном пространстве имен*, как и прежде.

При обращении к локальной переменной до того, как ей будет присвоено значение, возбуждается исключение `UnboundLocalError`. Следующий пример демонстрирует один из возможных сценариев, когда такое исключение может возникнуть

```
i = 0
def foo():
    i = i + 1  # приведет к исключению UnboundLocalError
    print(i)
```

В этой функции переменная `i` определяется как *локальная* (потому что внутри функции ей присваивается некоторое значение и отсутствует инструкция `global`).

При этом инструкция присваивания `i = i + 1` пытается прочитать значение переменной `i` еще до того, как ей будет присвоено значение.

Хотя в этом примере существует глобальная переменная `i`, она не используется для получения значения. Переменные в функциях могут быть либо *локальными*, либо *глобальными* и не могут произвольно изменять *область видимости* в середине функции.

Например, нельзя считать, что переменная `i` в выражении `i + 1` в предыдущем фрагменте обращается к глобальной переменной `i`; при этом переменная `i` в вызове `print(i)` подразумевает локальную переменную `i`, созданную в предыдущей инструкции.

---

*Обобщение по вопросу*

Когда интерпретатор, сканируя определение функции `def`, натыкается на строку `i = i + 1`, он заключает что переменная `i` является *локальной*, так как ей присваивается какое-то значение в теле функции. А когда функция вызывается на выполнение и интерпретатор снова доходит до строки `i = i + 1` выясняется, что переменная `i`, стоящая в правой части, не имет ссылок на какой-либо объект и потому возбуждается исключение `UnboundLocalError`

---

## 3. Замыкания/фабричные функции

Под термином *замыкание* или *фабричная функция* подразумевается объект функции, который сохраняет значения в *объемлющих областях видимости*, даже когда эти области могут прекратить свое существование [1, стр. 488].

Замыкания и вложенные функции особенно удобны, когда требуется реализовать концепцию отложенных вычислений [2].

Рассмотрим в качестве примера следующую функцию

Пример замыкания

```
def maker(N):
    def action(X):
        return X**N  # функция action запоминает значение N в объемлющей области видимости
    return action
```

Здесь определяется внешняя функция, которая просто создает и возвращает вложенную функцию, не вызывая ее. Если вызвать внешнюю функцию

```
>>> f = maker(2)  # запишет 2 в N
>>> f  # <function action at 0x0147280>
```

она вернет ссылку на созданную ею вложенную функцию, созданную при выполнении вложенной инструкции `def`. Если теперь вызвать то, что было получено от внешней функции

```
>>> f(3)  # запишет 3 в X, в N по-прежнему хранится число 2
>>> f(4)  # 4**2
```

будет вызвана вложенная функция, с именем `action` внутри функции `maker`. Самое необычное здесь то, что вложенная функция продолжает хранить число 2, значение переменной `N` в функции `maker` даже при том, что к моменту вызова функции `action` функция `maker` уже *завершила свою работу и вернула управление*.

Когда функция используется как вложенная, в замыкание включается все ее окружение, необходимое для работы внутренней функции [2, стр. 137].

### 3.1. Области видимости и значения по умолчанию применительно к переменным цикла

Существует одна известная особенность для функций или lambda-выражений: если lambda-выражение или инструкция `def` вложены в цикл внутри другой функции и вложенная функция

ссылается на переменную из объемлющей области видимости, которая изменяется в цикле, все функции, созданные в этом цикле, будут иметь одно и то же значение – значение, которое имела переменная на последней итерации [1, стр. 492].

Например, ниже предпринята попытка создать список функций, каждая из которых запоминает текущее значение переменной `i` из объемлющей области видимости

Эта реализация работать НЕ будет

```
def makeActions():
    acts = []
    for i in range(5):  # область видимости объемлющей функции
        acts.append(
            lambda x: i**x  # локальная область видимости вложенной анонимной функции
        )
    return acts

acts = makeActions()
print(acts[0](2))  # вернет 4**2, последнее значение i
print(acts[3](2))  # вернет 4**2, последнее значение i
```

Такой подход не дает желаемого результата, потому что поиск переменной в объемлющей области видимости производится позднее, *при вызове вложенных функций*, в результате все они получат одно и то же значение (значение, которое имела переменная цикла на последней итерации).

Это один из случаев, когда необходимо явно сохранять значение из объемлющей области видимости в виде аргумента со значением по умолчанию вместо использования ссылки на переменную из объемлющей области видимости.

То есть, чтобы фрагмент заработал, необходимо передать текущее значение переменной из объемлющей области видимости в виде значения по умолчанию. Значения по умолчанию вычисляются в момент *создания вложенной функции* (а не когда она *вызывается*), поэтому каждая из них сохранит свое собственное значение `i`

Правильная реализация

```
def makeActions():
    acts = []
    for i in range(5):
        acts.append(
            lambda x, i=i: i**x  # сохранить текущее значение i
        )
    return acts

acts = makeActions()
print(acts[0](2))  # вернет 0**2
print(acts[2](2))  # вернет 2**2
```

## 4. Калибровка классификаторов

Подробности в статье А. Дьяконова «Проблема калибровки уверенности».

Ниже описываются способы оценить качество калибровки алгоритма. Надо сравнить *уверенность* (confidence) и *долю верных ответов* (accuracy) на тестовой выборке.

Если классификатор «хорошо откалиброван» и для большой группы объектов этот классификатор возвращает вероятность принадлежности к положительному классу 0.8, то среди этих

объектов будет приблизительно 80% объектов, которые в действительности принадлежат положительному классу. То есть, если для группы точек данных общим числом 100 классификатор возвращает вероятность положительного класса 0.8, то приблизительно 80 точек на самом деле будут принадлежать положительному классу и доля верных ответов тогда составит 0.8.

### 4.1. Непараметрический метод гистограммной калибровки (Histogram Binning)

Изначально в методе использовались бины одинаковой ширины, но можно использовать и равномощные бины.

Недостатки подхода:

- число бинов задается наперед,
- функция деформации не непрерывна,
- в «равноширинном варианте» в некоторых бинах может содержаться недостаточное число точек.

Метод был предложен Zadrozny B. и Elkan C. Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers.

### 4.2. Непараметрический метод изотонической регрессии (Isotonic Regression)

Строится монотонно неубывающая функция деформации оценок алгоритма.

Метод был предложен Zadrozny B. и Elkan C. Transforming classifier scores into accurate multiclass probability estimates.

Функция деформации по-прежнему не является непрерывной.

### 4.3. Параметрическая калибровка Платта (Platt calibration)

Изначально этот метод калибровки разрабатывался только для метода опорных векторов, оценки которого лежат на вещественной оси (по сути, это расстояния до оптимальной разделяющей классы прямой, взятые с нужным знаком). Считается, что этот метод не очень подходит для других моделей.

Предложен Platt J. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods.

### 4.4. Логистическая регрессия в пространстве логитов

### 4.5. Деревья калибровки

Стандартный алгоритм строит строит суперпозицию дерева решений на исходных признаках и логистических регрессий (каждая в своем листе) над оценками алгоритма:

- Построить на исходных признаках решающее дерево (не очень глубокое),
- В каждом листе – обучить логистическую регрессию на одном признаке,
- Подрезать дерево, минимизируя ошибку.

### 4.6. Температурное шкалирование (Temperature Scaling)

Этот метод относится к классу DL-методов калибровки, так как он был разработан именно для калибровки нейронных сетей. Метод представляет собой простое многомерное обобщение шкалирования Платта.

# 5. Приемы работы с менеджером пакетов conda

## 5.1. Создание виртуального окружения

Создать виртуальное окружение dashenv

```
conda create --name dashenv
```

Создать виртуальное окружение с указанием версии Python

```
conda create --name testenv python=3.6
```

Создать виртуальное окружение с указанием пакета

```
conda create --name testenv scipy
```

Создать виртуальное окружение с указанием версии Python и нескольких пакетов

```
conda create --name testenv python=3.6 scipy=0.15.0 astroid babel
```

---

*Замечание*

Рекомендуется устанавливать сразу несколько пакетов, чтобы избежать конфликта зависимостей

---

Для того чтобы при создании нового виртуального окружения не требовалось каждый раз устанавливать базовые пакеты, которые обычно используются в работе, можно привести их список в конфигурационном файле .condarc в разделе create_default_packages

.condarc

```
ssl_verify: true
channels:
 - conda-forge
 - defaults
report_errors: true
default_python:
create_default_packages:
 - matplotlib
 - numpy
 - scipy
 - pandas
 - seaborn
```

Если для текущего виртуального окружения не требуется устанавливать пакеты из набора по умолчанию, то при создании виртуального окружения следует указать специальный флаг --no-default-packages

```
conda create --no-default-packages --name testenv python
```

Создать виртуальное окружение можно и из файла environment.yml (первая строка этого файла станет именем виртуального окружения)

environment.yml

```
name: stats2
channels:
 - conda-forge
 - defaults
dependencies:
 - python=3.6   # or 2.7
```

```
 - bokeh=0.9.2
 - numpy=1.9.*
 - nodejs=0.10.*
 - flask
 - pip:
  - Flask-Testing
```

```
conda env create -f environment.yml
```

При создании виртуального окружения можно указать путь до целевой директории, где будут размещаться файлы окружения. Следующая команда создаст виртуальное окружение в поддиректории текущей рабочей директории `envs`[2]

```
conda create --prefix ./envs jupyterlab matplotlib
```

С помощью файла спецификации можно создать *идентичное виртуальное окружение* (i) на той же платформе операционной системы, (ii) на той же машине, (iii) на какой-либо другой машине (перенести настройки окружения).

Для этого предварительно требуется создать собственно файл спецификации

```
conda list --explicit > spec-file.txt
```

Имя файла спецификации может быть любым. Файл спецификации обычно не является кросс-платформенным и поэтому имеет комментарий в верхней части файла (`#platform: osx-64`), указывающий платформу, на которой он был создан.

Теперь для того чтобы *создать* окружение достаточно воспользоваться командой

```
conda create --name myenv --file spec-file.txt
```

Файл спецификации можно использовать для установки пакетов в существующее окружение

```
conda install --name myenv --file spec-file.txt
```

## 5.2. Активация/деактивация виртуального окружения

Активировать виртуальное окружение `dashenv`

```
conda activate dashenv
```

Активировать виртуальное окружение в случае, когда оно создавалось с `--prefix`, можно указав полный путь до окружения

```
conda activate E:\[WorkDirectory]\[Python_projects]\directory_for_experiments\envs
```

В этом случае в строке приглашения командной оболочки по умолчанию будет отображаться полный путь до окружения. Чтобы заменить длинный префикс в имени окружения на более удобный псевдоним достаточно использовать конструкцию

```
conda config --set env_prompt ({name})
```

которая добавит в конфигурационный файл `.condarc` следующую строку

.condarc

```
...
env_prompt: ({name})
```

[2] В данном случае чтобы удалить виртуальную среду достачно просто удалить директорию `envs`

и теперь имя окружения будет `(envs)`.

Деактивировать виртуальное окружение

```
conda deactivate
```

### 5.3. Обновление виртуального окружения

Обновить виртуальное окружение может потребоваться в следующих случаях:

- обновилась одна из ключевых зависимостей,
- требуется добавить пакет (добавление зависимости),
- требуется добавить один пакет и удалить другой.

В любом из этих случаев все что нужно для того чтобы обновить виртуальное окружение это просто обновить файл `environment.yml`[3], а затем запустить команду

```
conda env update --prefix ./envs --file environment.yml --prune
```

Опция `--prune` приводит к тому, что `conda` удаляет все зависимости, которые больше не нужны для окружения.

### 5.4. Вывод информации о виртуальном окружении

Вывести список доступных виртуальных окружений

```
conda env list
```

Вывести список пакетов, установленных в указанном окружении

```
conda list --name myenv
```

Вывести информацию по конкретному пакету указанного окружения

```
conda list --name dashenv matplotlib
```

### 5.5. Удаление виртуального окружения

Удалить виртуальное окружение `heroku_env`

```
conda env remove --name heroku_env
```

### 5.6. Экспорт виртуального окружения в `environment.yml`

Экспортировать активное виртуальное окружение в yml-файл

```
conda env export > environment.yml
```

## Список литературы


1. *Лутц М.* Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.
2. *Бизли Д.* Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с.
3. *Чакон С.*, *Штрауб Б.* Git для профессионального программиста. – СПб.: Питер, 2020. – 496 с.


[3] Этот файл должен находится в той же директории что и директория окружения `envs`