

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	Градиентный бустинг	2
1.1	Общие сведения	2
1.2	Особенности реализации в <code>sklearn</code>	2
1.3	Особенности реализации в <code>XGBoost</code>	2
1.3.1	Установка пакета <code>xgboost</code> на Windows	2
1.4	Простой пример работы с <code>xgboost</code> и <code>shap</code>	3
1.5	Особенности реализации в <code>LightGBM</code>	4
1.6	Особенности реализации в <code>CatBoost</code>	4
2	Приемы интерпретации моделей машинного обучения	4
3	Области видимости в языке Python	4
4	Замыкания/фабричные функции в Python	7
4.1	Области видимости и значения по умолчанию применительно к переменным цикла	7
5	Калибровка классификаторов	8
5.1	Непараметрический метод гистограммной калибровки (Histogram Binning)	9
5.2	Непараметрический метод изотонической регрессии (Isotonic Regression)	9
5.3	Параметрическая калибровка Платта (Platt calibration)	9
5.4	Логистическая регрессия в пространстве логитов	9
5.5	Деревья калибровки	9
5.6	Температурное шкалирование (Temperature Scaling)	9
6	Приемы работы с менеджером пакетов <code>conda</code>	10
6.1	Создание виртуального окружения	10
6.2	Активация/деактивация виртуального окружения	11
6.3	Обновление виртуального окружения	12
6.4	Вывод информации о виртуальном окружении	12
6.5	Удаление виртуального окружения	12
6.6	Экспорт виртуального окружения в <code>environment.yml</code>	12
7	Приемы работы с пакетом <code>Vowpal Wabbit</code>	13

8	Приемы работы с библиотекой pandas	13
8.1	Число уникальных значений категориальных признаков в объекте DataFrame . . .	13
8.2	Число пропущенных значений в объекте DataFrame	13
9	Интерпретация моделей и оценка важности признаков с библиотекой SHAP	13
9.1	Общие сведения о значениях Шепли	13
9.2	Перестановочная важность признаков в библиотеке eli5	13
	Список литературы	14

1. Градиентный бустинг

1.1. Общие сведения

1.2. Особенности реализации в sklearn

1.3. Особенности реализации в XGBoost

1.3.1. 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 (см. рис. 1), по которым будет проводиться поиск интересующего пакета (в данном случае пакета `xgboost`), а затем можно воспользоваться конструкцией

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

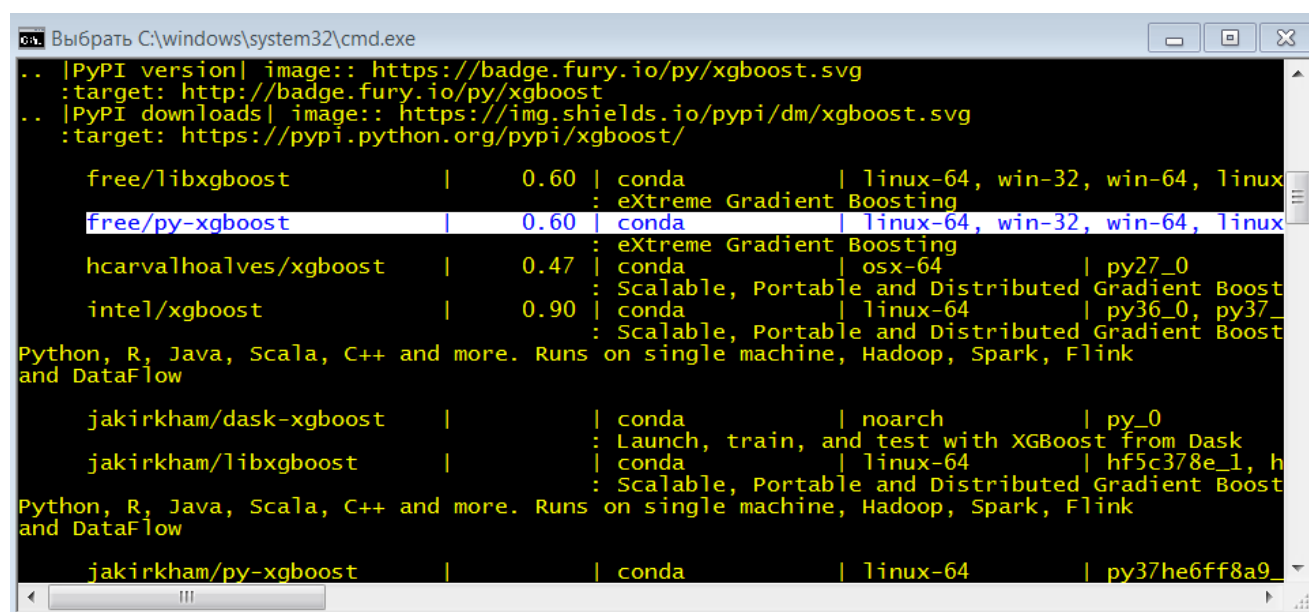


Рис. 1. Окно командной оболочки `cmd.exe` со списком доступных каналов, по которым будет проводиться поиск пакета `xgboost`

После, выбрав канал, можно приступать к установке пакета

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

1.4. Простой пример работы с xgboost и shap

Решается задача бинарной классификации. Требуется построить модель, предсказывающую годовой доход заявителя по порогу \$50'000 (то есть больше или меньше \$50'000 зарабатывает заявитель в год). Используется набор данных UCI Adult income

```
import xgboost
import shap # для оценки важности признаков вычисляются значения Шепли (Shapley value)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

shap.initjs()

X, y = shap.datasets.adult()
X_display, y_display = shap.datasets.adult(display=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7)
d_train = xgboost.DMatrix(X_train, label=y_train)
d_test = xgboost.DMatrix(X_test, label=y_test)

params = {
    'eta' : 0.01,
    'objective' : 'binary:logistic',
    'subsample' : 0.5,
    'base_score' : np.mean(y_train),
    'eval_metric' : 'logloss'
}
model = xgboost.train(params, d_train,
                      num_boost_round = 5000, # число итераций бустинга
                      evals = [(d_test, 'test')],
                      verbose_eval=100, # выводит результат на каждой 100-ой итерации бустинга
                      early_stopping_rounds=20)

xgboost.plot_importance(model)
```

На рис. 2, рис. 3 и рис. 4 изображены графики важности признаков.

Следует иметь в виду, что в библиотеке **xgboost** поддерживается три варианта вычисления важности признаков (см. [Interpretable Machine Learning with XGBoost](#)):

- **weight**: общее число сценариев по всем деревьям, когда i -ый признак используется для расщепления обучающего набора данных,
- **cover**: общее число сценариев по всем деревьям, когда i -ый признак используется для расщепления набора данных, взвешенное по числу точек обучающего набора данных, которые проходят через эти расщепления,
- **gain**: среднее снижение потерь на обучающем наборе данных, полученное при использовании i -ого признака.

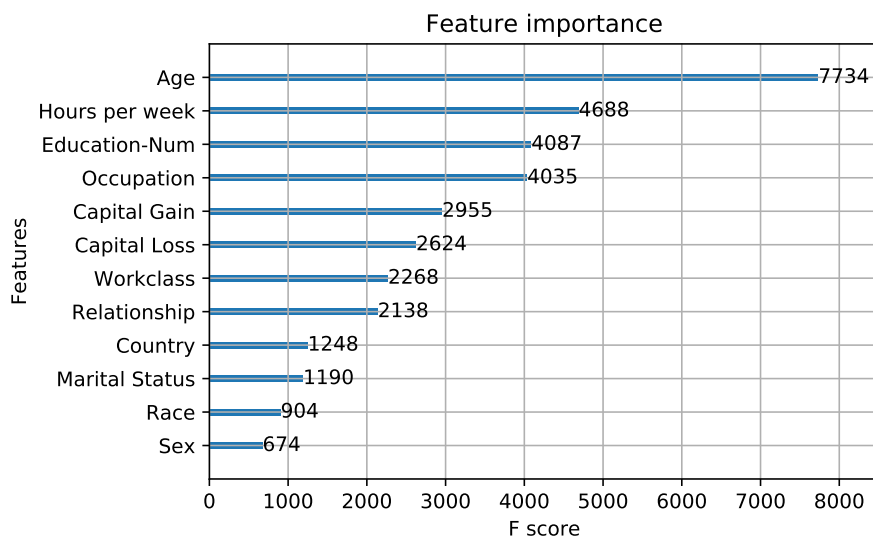


Рис. 2. График важности признаков `xgboost.plot_importance(model)`, построенный с помощью пакета `xgboost`

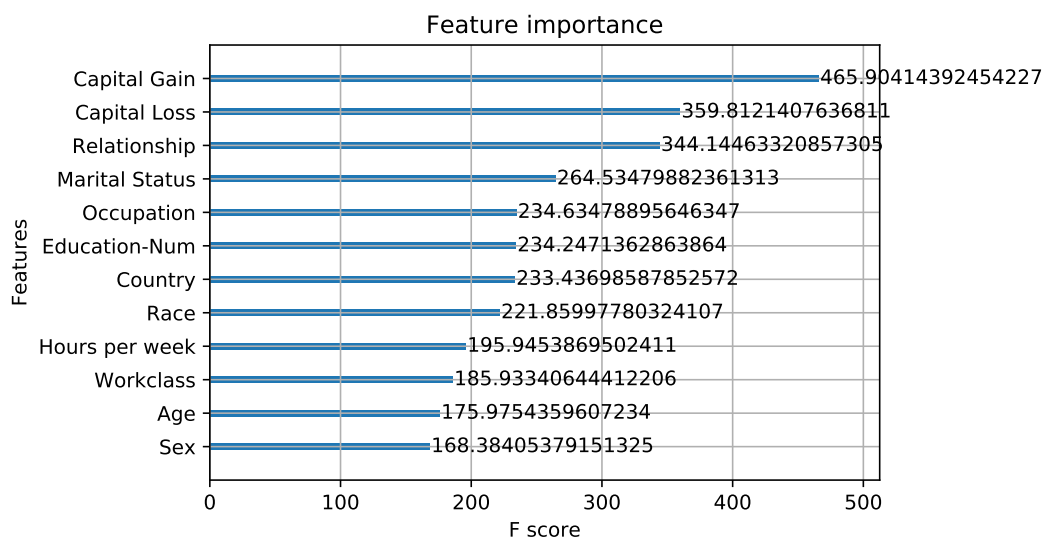


Рис. 3. График важности признаков `xgboost.plot_importance(model, importance_type='cover')`, построенный с помощью пакета `xgboost`

1.5. Особенности реализации в LightGBM

1.6. Особенности реализации в CatBoost

2. Приемы интерпретации моделей машинного обучения

3. 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.

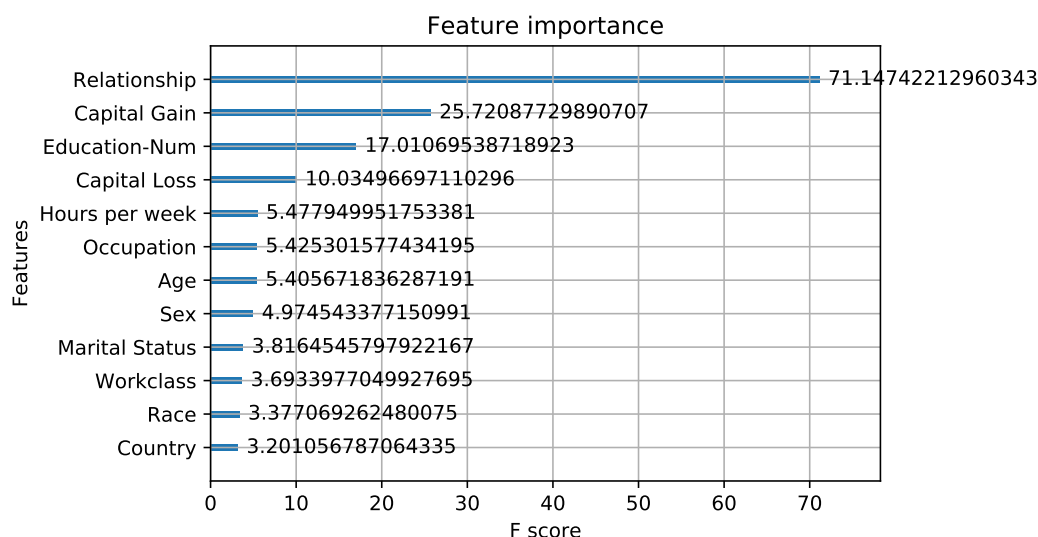


Рис. 4. График важности признаков `xgboost.plot_importance(model, importance_type='gain')`, построенный с помощью пакета `xgboost`

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 обволакивающей функции* или в выражении `lambda`, затем в *глобальной* и, наконец, во *встроенной*. Поиск завершается, как только будет найдено первое подходящее имя.
- 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

¹Local, Enclosing, Global, Built-in

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`, натывается 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 имеет ссылок на какой-либо объект и потому возбуждается исключение `UnboundLocalError`

4. Замыкания/фабричные функции в Python

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].

4.1. 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 получают 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

```

Обобщение по вопросу

Значения аргументов по умолчанию вложенных функций, динамически создаваемых в цикле на уровне области видимости объемлющей функции, вычисляются в момент *создания* этих вложенных функций, а не в момент их вызова, поэтому `lambda x, i=i: ...` работает корректно

5. 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* (ассигасу) 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.

5.1. Непараметрический метод гистограммной калибровки (Histogram Binning)

Изначально в методе использовались бины одинаковой ширины, но можно использовать и равномошные 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.](#)

5.2. 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.

5.3. 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.](#)

5.4. Логистическая регрессия в пространстве логитов

5.5. Деревья калибровки

Стандартный алгоритм 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.

5.6. 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.

6. Приемы работы с менеджером пакетов conda

6.1. 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
```

6.2. 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})
```

²В данном случае чтобы удалить виртуальную среду достаточно просто удалить директорию `envs`

и теперь имя окружения будет (`envs`).

Деактивировать виртуальное окружение

```
conda deactivate
```

6.3. 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.

6.4. 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
```

6.5. Удаление виртуального окружения

Удалить виртуальное окружение `heroku_env`

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

6.6. Экспорт виртуального окружения в `environment.yml`

Экспортировать активное виртуальное окружение в `yml`-файл

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

³Этот файл должен находиться в той же директории что и директория окружения `envs`

7. Приемы работы с пакетом Vowpal Wabbit

8. Приемы работы с библиотекой pandas

8.1. Число уникальных значений категориальных признаков в объекте DataFrame

Для того чтобы вывести информацию по числу уникальных значений в каждом категориальном признаке некоторого объекта `pandas.DataFrame` можно воспользоваться конструкцией

```
X.select_dtypes('category').apply(lambda col: col.unique().shape[0])
```

8.2. Число пропущенных значений в объекте DataFrame

Информацию по числу пропущенных значений в каждом столбце можно вывести следующим образом

```
X.isna().any(axis=0)
```

9. Интерпретация моделей и оценка важности признаков с библиотекой SHAP

9.1. Общие сведения о значениях Шепли

В библиотеке SHAP <https://github.com/slundberg/shap> для оценки *важности признаков* используются значения Шепли (Shapley value) https://en.wikipedia.org/wiki/Shapley_value.

Формула для вычисления значения Шепли i -ого признака имеет вид (здесь сумма распространяется на все подмножества S из множества признаков N , не содержащие i -ого признака)

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} (v(S \cup \{i\}) - v(S)),$$

где n – общее число признаков; $v(S \cup \{i\})$ – прогноз модели с учетом i -ого признака; $v(S)$ – прогноз модели без i -ого признака; S – произвольный набор признаков без i -ого признака.

Значение Шепли для i -ого признака вычисляется для каждой точки данных (например, для каждого клиента в выборке) на всех возможных комбинациях признаков (в том числе и для пустых множеств S). Затем полученные оценки суммируются по модулю, что в итоге дает важность i -ого признака.

Библиотека SHAP позволяет интерпретировать и модель в целом, и результаты отдельных примеров <https://habr.com/ru/company/homecredit/blog/479712/>.

9.2. Перестановочная важность признаков в библиотеке eli5

Еще важность признаков можно оценивать с помощью так называемой *перестановочной важности* (permutation importances) <https://www.kaggle.com/dansbecker/permutation-importance>.

Идея проста: нужно в заранее отведенном для исследования важности признаков наборе данных (валидационном наборе) перетасовать значения признака, влияние которого изучается на данной итерации, оставив остальные признаки (столбцы) и целевой вектор без изменения.

Признак считается «важным», если метрики качества модели падают, и соответственно – «неважным», если перестановка не влияет на значения метрик.

Замечание

Перестановочная важность вычисляется после того как модель будет обучена

Рассмотрим задачу построения регрессионной модели на наборе данных `load_boston`

```
import eli5
import pandas as pd
from eli5.sklearn import PermutationImportance
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
from pandas import DataFrame, Series

boston = load_boston()

X_train, X_test, y_train, y_test = train_test_split(boston['data'],
                                                    boston['target'],
                                                    random_state=2)

X_train_sub, X_valid, y_train_sub, y_valid = train_test_split(X_train,
                                                            y_train,
                                                            random_state=0)

# модель случайного леса, как обычно, обучается на обучающей выборке
rf = RandomForestRegressor(n_estimators=500).fit(X_train_sub, y_train_sub)

# модель перестановочной важности обучается на валидационном наборе данных
perm = PermutationImportance(rf, random_state=42).fit(X_valid, y_valid)

eli5.show_weights(perm, feature_names = boston['feature_names']) # визуализирует перестановочны
                          е важности признаков
```

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.