Конспект лекций

Данный краткий конспект имеет указание (в скобках в названии лекций) номер лабораторной, которой касается данный материал. Для некоторых лабораторных **необходимо** знание данного материала.

Тема №1. (лаб. 2) Сравнение строк и основы нечеткой логики

Часто ИИ применяют для обработки поисковых запросов, преобразования речи в текст, распознавания рукописей и прочих взаимодействий с текстом. Очевидно, что сравнение текстов занимает большую роль в этой области, но не может быть реализовано в полной мере на python средствами одних лишь множеств. Большую гибкость для решения такой задачи дает наиболее простой раздел ИИ – методы нечеткой логики.

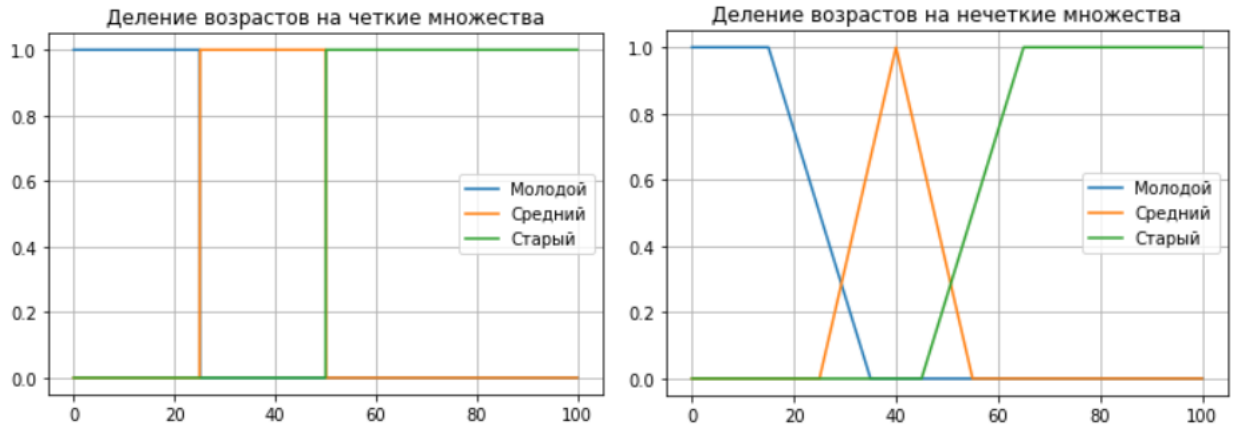
Нечеткая логика работает с так называемыми нечеткими множествами. В прошлой работе вы осуществляли операции над «четкими», булевыми множествами – если представить принадлежность к булевому множеству числом 1 (или True), а непринадлежность – числом 0 (или False), то получается, что элемент, принадлежащий к какому-то множеству, имеет булево значение 1, если же не принадлежит – булево 0. Так, по этой логике, чай может быть либо сладким (1) либо нет (0), то есть либо относиться к множеству «сладкие», либо нет.

В нечеткой же логике вместо характеристик 0 и 1 используется 0…1.0, то есть сменяется бинарный подход принадлежит/не принадлежит ли элемент к множеству на указание степени (в виде десятичной дроби), в которой тот или иной элемент к тому или иному множеству относится. По тому же примеру с чаем, в рамках нечеткой логики можно сказать, что чай несладкий (0), слегка сладковат (0.1), в меру сладкий (0.4), или переслащенный (1.0).

Тема №2. (лаб. 3) Нечеткие множества

Как уже было сказано ранее, нечеткая логика отличается от обычной (четкой) тем, что помимо принадлежности какого-то элемента к множеству появляется также степень его принадлежности к этому множеству.

Разница четкой и нечеткой логики представлена на графиках ниже в рамках множеств групп возрастов. Условно разделив возраста от 0 до 100 лет на 3 множества (молодой = 0…25, средний = 25…50, старый = 50…100), для четкого множества получаем, что человек может входить только в одно множество со значением принадлежности 1, или не входить в него со значением 0. К примеру, по данному примеру человек 30 лет будет иметь следующие значения принадлежности к представленным множествам (график слева): молодой = 0, средний = 1, старый = 0. На практике же, с возрастом человек не сразу (моментально) переходит из одной возрастной группы (множества) в другую, но постепенно меняет группу. В рамках нечеткой логики (график справа), появляются переходные зоны, в которых принадлежность объекта к множеству принимает значение 0…1. К примеру, тот же человек 30 лет по представленному графику будет иметь, по нечеткой логике, такие значения принадлежности: молодой = 0,25, средний = 0,25, старый = 0.



Функция, которая описывает график изменения значения принадлежности в множестве называется *функцией принадлежности*.

Процедура сопоставления значению какой-то величины (в примере выше – возраста) степени принадлежности к тому или иному нечеткому множеству называется *фаззификацией*.

Переход к нечетким множествам позволяет составлять правила, применимые для управления режимом работы различных объектов. В таком виде правила идентичны четкой логике, поскольку не учитывают степень принадлежности и, соответственно, позволяют выбирать только диапазон выходной величины, но не значение в нем.

В конечном виде, по правилам выходная величина определяется так:

1.1) задаются значения входных величин;

1.2) определяется, к какому множеству относится каждая входная величина (высокая ли температура или низкая, высокая ли влажность и т.п.), по нему определяется значение принадлежности для каждой из заданных величин;

2.1) проверяется, каким заданным правилам соответствуют множества, в которые попадают входные величины;

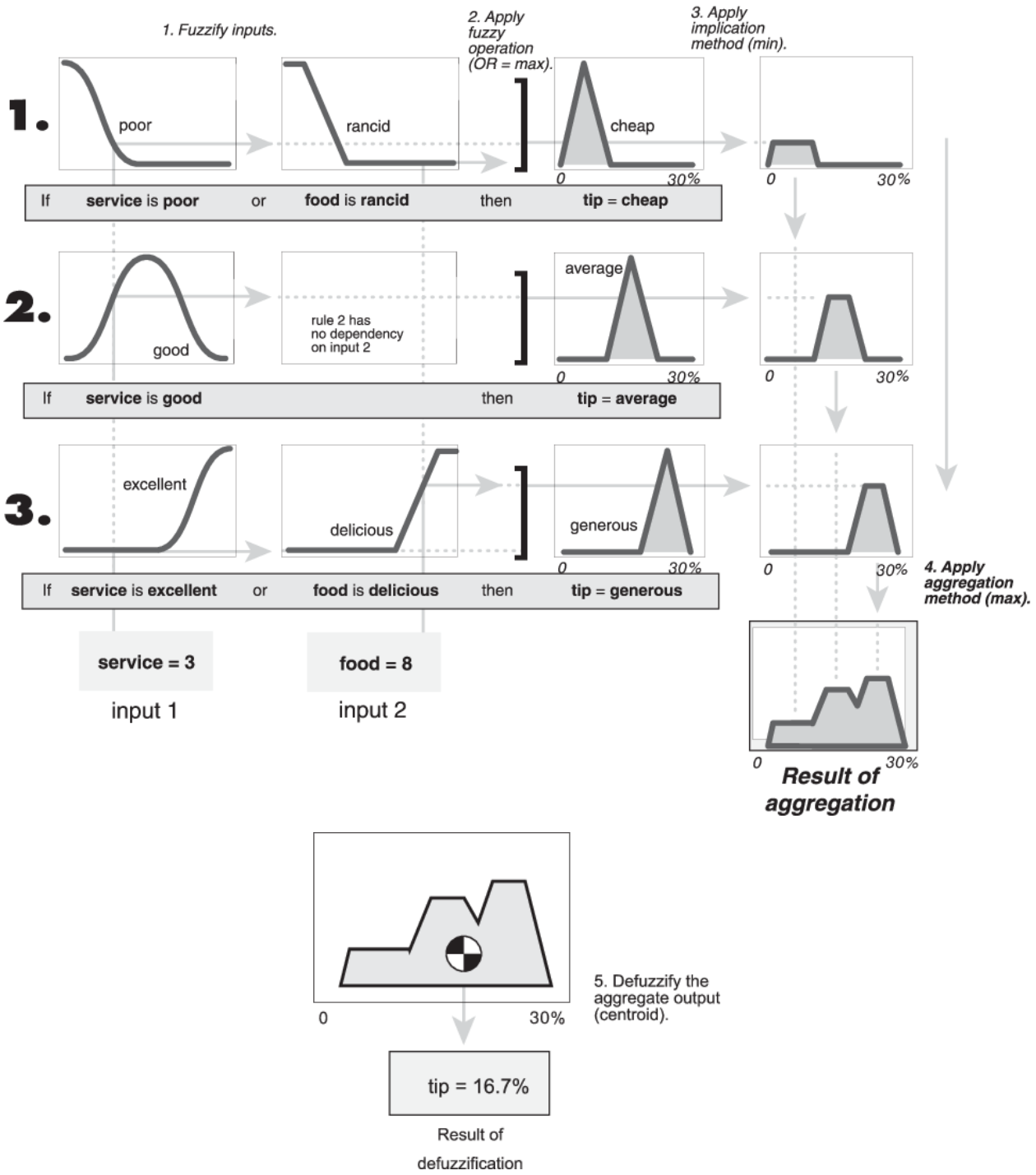
2.2) по правилам определяется выходное значение принадлежности – если в правиле величины сочетаются «И», то выбирается минимальное значение, если через «ИЛИ» – то максимальное;

3) функция принадлежности выходной величины «усекается» на уровне выходного значения принадлежности;

4) процедуры 2.2 и 3 выполняются для всех правил, все результаты объединяются (через пересечение всех «усеченных» множеств).

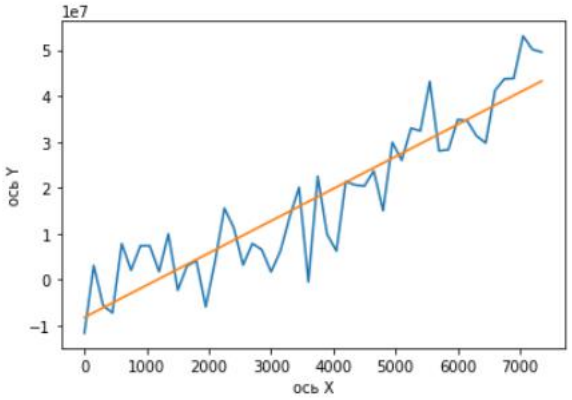
5) для полученной фигуры вычисляется так называемый центр масс. Именно центр масс и будет задавать значение выходной величины, подходящей под заданные правила. Процедура получения числового значения выходной величины после операций с нечеткой логикой называется *дефаззификацией.*

Фаззификация с последующей дефаззификацией представлена на примере вычисления чаевых, основываясь на качестве еды и сервиса в некотором ресторане.



Тема №3. (лаб. 4) Аппроксимация

Построение регрессии означает реализацию аппроксимации данных с введением какой-либо меры точности. Сама же аппроксимация – это процедура подбора оптимальных параметров какой-либо функции таким образом, чтобы она описывала исходные данные. В качестве примера на скриншоте приведена аппроксимация массива данных с шумом (синяя ломаная) прямой (оранжевая линия), при этом подбираются смещение и наклон последней так, чтобы расстояние между всеми точками исходных данных и проводимой прямой было бы минимально возможным (согласно методу наименьших квадратов, существуют и другие методы аппроксимации).



Аппроксимация, как уже было сказано, при наличии меры точности называется регрессией (регрессионным анализом). Причем, если аппроксимация производится прямой, то результат аппроксимации именуют линейной регрессией, если многочленом более высокого порядка – полиномиальной регрессией.

Тема №4. (лаб. 5) Формула Байеса для условной вероятности

При решении различных проблем машинного обучения возникает задача: в результате осуществления одной из гипотез B1, B2, …, Bn наступило событие A (вероятность верности гипотезы при происхождении какого-то события называют условной). Оценить вероятность того, что осуществилась конкретная гипотеза Bi. Для этого наиболее прочих подходит формула Байеса:

$$P_A(Bi) = \frac{P(Bi) \cdot P_{Bi}(A)}{\sum_{j=1}^{n} P(Bj) \cdot P_{Bj}(A)} \quad (1)$$

где Bi – несовместные гипотезы, может произойти только одна из них;

$P(Bi)$ - вероятность, что Bi гипотеза верна;

$P_{Bi}(A)$ - вероятность, что Bi гипотеза вызовет событие A (условная вероятность);

$P(Bi)\cdot P_{Bi}(A)$ - вероятность, что именно гипотеза Bi привела к A;

$\sum_{j=1}^{n} P(Bj)\cdot P_{Bj}(A)$ - вероятность, что A вообще могло произойти (полная вероятность – сумма всех условных вероятностей).

Для пояснения (1) проще всего рассмотреть пример. Допустим, тест на болезнь имеет точность 95% (и 5% ошибки, соответственно). Всего болезнь имеется лишь у 1% пациентов. Человек получил тест, который говорит о наличии у него болезни. Необходимо определить вероятность того, что человек действительно болен.

Применительно к формуле (1) задача расшифровывается следующим образом: A – гипотеза о том, что человек болен, B1 – тест верен (P(B1)=0,95), B2 – тест ошибочен (P(B2)=0,05). Выходит, что условных вероятностей для A две - $P_{B1}(A)=0{,}01$ (при верном тесте пациент действительно болен), $P_{B2}(A)=0{,}99$ (при ошибочном тесте пациент на самом деле здоров). Соответственно формула Байеса (1) имеет вид:

$$P_A(B1)=\frac{P(B1)\cdot P_{B1}(A)}{P(B1)\cdot P_{B1}(A)+P(B2)\cdot P_{B2}(A)}, \qquad (2)$$

И при подстановке числовых значений вероятностей в (2) имеем:

$$P_A(B1)=\frac{0{,}95\cdot 0{,}01}{0{,}95\cdot 0{,}01+0{,}05\cdot 0{,}99}\approx 0{,}16102=16{,}1\%.$$

Тем самым, положительный тест с точностью 95% еще не говорит о том, что человек болен с высокой вероятностью.

Задачи, решаемые по формуле (1), могут быть сформулированы иначе, без числовых значений вероятности. К примеру, на скриншоте представлена таблица – журнал игрока в теннис, где он 14 дней отмечал погодные условия и фиксировал, была ли игра удачной (Yes/No). Получив погодные условия на 15 день «**x**=(Sunny, Cool, High, Strong)» он хочет предсказать, будет ли игра успешной.

PlayTennis: training examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Здесь каждое погодное условие является гипотезой Bi, а PlayTennis=Yes – событием A. Для решения задачи требуется подсчитать сперва общее число успешных/проигрышных вариантов – соответственно 5 «No» и 9 «Yes». Сразу же можно оценить вероятность каждого из исходов 5/14 для «No» и 5/14 «Yes». Сразу стоит оговориться, что принятие решения о том, стоит ли играть или нет будет приниматься путем сравнения (1) вычисленной для гипотез PlayTennis=Yes и PlayTennis=No, тем самым придется вычислять оба исхода.

Для каждого погодного условия аналогично оцениваем вероятность, но для положительного и отрицательного исхода, очевидно, вероятности будут свои. Так, например, при Outlook=Sunny наблюдалось 2 из 9 выигрышей и 3 из 5 проигрышей в теннис, соответственно будем иметь P(Outlook=Sunny|Play=Yes)=2/9 и P(Outlook=Sunny|Play=No)=3/5. Подсчитав вероятности для всех условий, подставляем в (1) все условные вероятности для текущих условий 15 дня с исходом Yes, затем туда же для исхода No и сравниваем вероятности. В данном случае (см. скриншот ниже) получается, что играть не стоит.

$\mathbf{x}$=(Outlook=*Sunny*, Temperature=*Cool*, Humidity=*High*, Wind=*Strong*)

P(Outlook=*Sunny*|Play=*Yes*) = 2/9
P(Temperature=*Cool*|Play=*Yes*) = 3/9
P(Huminity=*High*|Play=*Yes*) = 3/9
P(Wind=*Strong*|Play=*Yes*) = 3/9
P(Play=*Yes*) = 9/14

P(Outlook=*Sunny*|Play=*No*) = 3/5
P(Temperature=*Cool*|Play==*No*) = 1/5
P(Huminity=*High*|Play=*No*) = 4/5
P(Wind=*Strong*|Play=*No*) = 3/5
P(Play=*No*) = 5/14

$P(Yes|\mathbf{x}) \approx [P(Sunny|Yes)P(Cool|Yes)P(High|Yes)P(Strong|Yes)]P(Play=Yes) = 0.0053$
$P(No|\mathbf{x}) \approx [P(Sunny|No)\ P(Cool|No)P(High|No)P(Strong|No)]P(Play=No) = 0.0206$

$P(Yes|\mathbf{x}) \gtrless P(No|\mathbf{x})$

Решение *"No"*.

Тема №5.1. Основы и истоки искусственного интеллекта

Сквозные технологии применяются для решения каких-то задач, такие технологии не самоцель. Во главе таких технологий – искусственный интеллект (ИИ). Если человек управляет роботом – это механизация, ИИ здесь нет. Киберфизическая система (CPS cyber-physical system) – исполняет по своему ИИ заданную оператором задачу.

Кибернетика – наука об общих закономерностях всего что происходит с инфой в сложных управляющих системах (обработка, передача и т.п). Изначально предлагался для живых систем (обществ), Роберт Винер предложил применить к механическим системам. Является пересечением теории управления, оптимизации, анализа данных, теории информации, теории математической коммуникации, ИИ и т.п.

Техническая кибернетика – наука об управлении техническими системами. Ее несколько неправомерно отождествляют с современной теорией автоматического регулирования и управления.

Тьюринг оказал существенное влияние на развитие информатике, в 1936 предложил теоретическую Машину Тьюринга, модель компьютера общего назначения.

ИИ – это слияние обучения (или самообучения) и принятия решений. Иллюстрация теста Тьюринга – CAPTCHA.

ИИ: наука и технология создания интеллектуальных систем, программ; свойство таких систем выполнять творческие функции. ИИ создается с целью превзойти человека там, где человек ошибается или не справляется. Тем самым ИИ – это еще попытка понять с помощью компьютера мышления человека, так как эта проблема не решена вовсе.

Сильный и слабый ИИ. Делает ли вывод ИИ на основе экспертного заключения? Сильный ИИ предполагает способность машиной мыслить и осознать себя. Слабый ИИ – не предполагает такой способности. Признаки сильного ИИ:

- принятие решений, стратегическое мышление, действие в неопределенности, решение задач;
- представление знаний, в том числе общих;
- планирование;
- обучение других систем;
- общение на естественном языке;
- и объединение всего этого,
- если все это есть, то это сильный ИИ, сейчас таких нет.

ImageNet – открытая выборка 15млн размеченных изображений. Сверточные НС – 2011 г.

ИИ состоит из разных отраслей – машинное обучение, обработка естественных языков, синтез речи, экспертные системы, планирование, робототехника, компьютерное зрение.

Тема №5.2. Машинное обучение и распознавание образов

Машинное обучение (МО) – машина проявляет поведение, которое не было явно в нее заложено. Дата майнинг (ДМ) используется для извлечения закономерностей из данных с целью обучить человека (к примеру, медицинская диагностика). В МО решение принимает машина, при ДМ – человек.

Распознавание образов (РО) – отнесение физического объекта или события к одной из предопределенных категорий. Большинство задач МО сводится к РО.

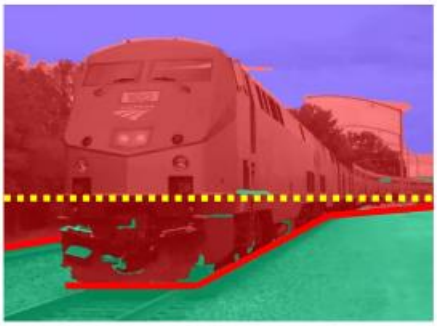
Пример работы РО – автоматическая сегментация сцены. Находятся похожие области.

Исходное изображение

Сегментированное изображение

$\{y_i\}_j = \{0,1,3,...,1\}$

Множество маркеров областей



Перенос разметки на изображение и её анализ

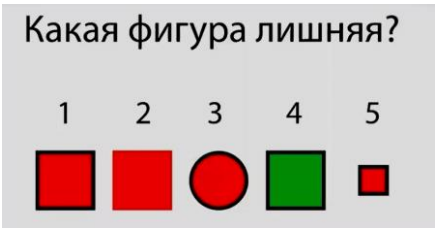
Сегментированное изображение переводится в множество признаков (массив, вектора), которые затем анализируются. Признаком, например, может быть яркость пикселей. По признакам математикой формируется множество маркеров областей (малое число значений, допустим, один вектор). На примере выше вместо множества сегментов по маркерам областей изображение делится на поезд, небо, траву.

Признак – результат измерения отличительной характеристики объекта (качественной или количественной). Формально признак – это отображение измеренного значения X на множество допустимых значений Df.

Распознаваемый объект – вектор (или можно сказать точка) в некотором пространстве.

Образ – совокупность признаков для объекта. В задачах классификации это пара переменных $(\vec{x}, \omega)$ где x это вектор признаков (набор наблюдений), омега – понятие за пределами наблюдения (метка наблюдения).

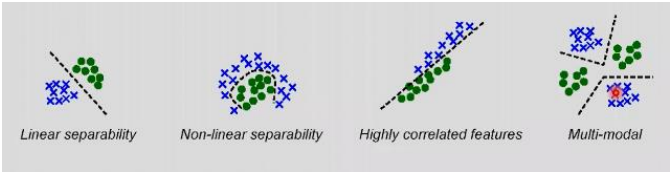
Задача с фигурами.



У первой фигуры нет оригинальности, поэтому это лишняя фигура, похожесть это и есть признак здесь.

Хороший вектор признаков имеет хорошую разделяющую способность, те объекты разных классов должны иметь различные вектора признаков и наоборот, то есть по вектору признаков можно четко отделить один класс от другого, тем самым признаки должны быть четко разделены. К примеру, у плоских фигур признак «круглость» плохо разделен (так как существует множество вариантов между кругом и овалом, то есть признак может принимать чуть ли не бесконечное множество значений), а вот форма – хорошо разделена (круг, квадрат, треугольник – дискретные значения признака, а значит признак хорошо разделен).

Граница разделения (разделимость) может быть прямой, нелинейной, высококоррелированной, множественной (мульти-модальной), лучше всего прямая (то есть можно четко объект отнести к одному или к другому классу).



Задачи распознавания образов – классификация (отнесение объекта к классу), обучение с учителем; регрессия (обобщение классификации, на выходе не целое число, а действительное число или вектор, попытаться

восстановить по входному вектору исходный); кластеризация (объединение объектов в осмысленные группы, выдает группировку, не число); описание (представление объекта в терминах набора примитивов, выдает структурное синтаксическое описание объекта, к примеру распознавание фотоизображений неплохо, похуже для аудио, еще хуже для текста, распознать, кто на фото – довольно реально, а в аудио распознать играющий инструмент уже сложно).

Подходы РО – статистический, нейросетевой, синтаксический, прочие (нечеткие, древа, тд). Для распознавания букв НС пытается отдельные части изображения буквы подавать на входной слой и на выходе загорается нейрон, соответствующий букве А; статистический подход – извлечение признаков из буквы (наклон сторон, горизонтальная линия, отверстие в букве и т.п.) и на их основе соотнесение с известной базой, моделью – если такие-то значения признаков, то такая-то модель наиболее вероятна; структурный (синтаксический) – разлагается буква по линиям на вектора и получаем структурное описание буквы А.

Цикл создания системы распознавания образов:

1.1) сбор данных – измерение для объекта реального мира датчиками, камерами, базами данных (для виртуальных данных, сгенерированных, задача пропущена по сути);

1.2) предобработка данных – фильтрация шума, извлечение признаков, нормализация;

2) снижение размерности – выбор признаков, проецирование признаков, в итоге построение пространство признаков;

3) выбор модели – НС, статистическая, структурная и т.п., выбор параметров модели;

4) обучение – классификация, регрессия, кластеризация, описание, адаптация модели по обучающей выборке, выбор подхода – с учителем или без;

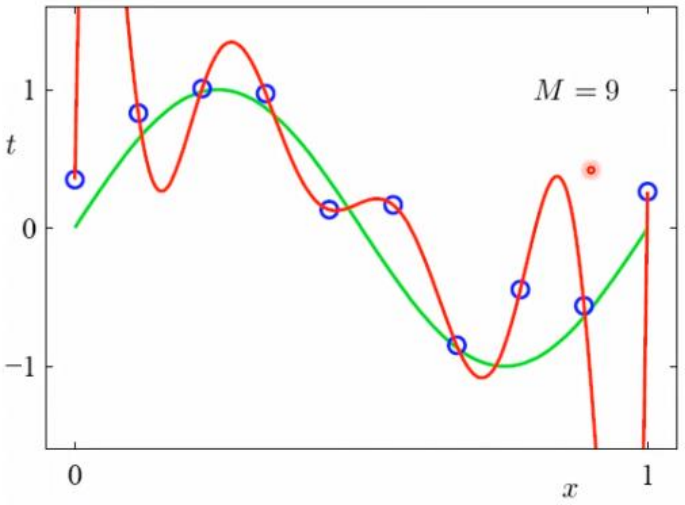
5) оценка результатов – насколько хорошо полученная модель описывает другие данные, подстройка модели или выбор другой модели.

Тема №5.3. Основы классификации

Классификаторы делят пространство признаков на области классов. Границы между классами – решающие правила (набор признаков, по которому объект можно отнести в тот или иной класс). Классификация не только определяет класс объекта, но еще и относит объект к нему. Классификатор можно представить множеством разделяющих (дискриминантных) функций, по значению этой функции проверяем, к какому классу объект относится.

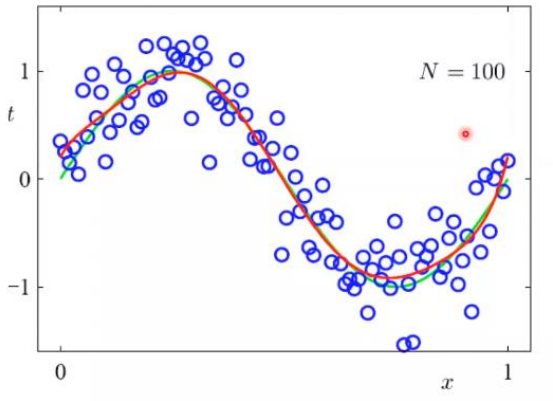
Классификация (контролируемая классификация) – распознавание по известным категориям, соотнесение объектов множества с этими категориями-классами, кластеризация (неконтролируемая классификация) – деление множества на категории (не известные заранее) по признакам объектов множества.

При обучении нужно достигать баланса между точностью и гибкостью модели. Например, исходный зеленый синус измерили с погрешностью синими точками и полином 9 порядка идеально проходит через все точки, но явно не совпадает с исходной синусоидой.



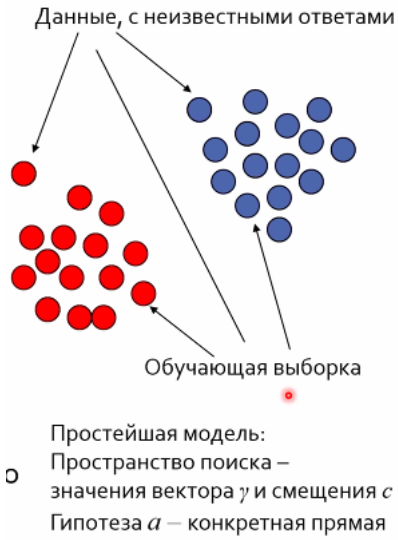
Но если точек было бы побольше, то уже результат аппроксимации через все точки был бы даже ужаснее, но, если проводить с учетом какого-то условия

минимума расстояния от аппроксимирующей кривой до всех точек, получается уже неплохой результат.



Тема №5.4. Обучение модели

Задача МО с учителем – предполагаем, что есть целевая функция, значения которой известны на конечном подмножестве объектов (множество пар объект-ответ, те известен вход и выход системы). Сама задача (или задача решения по прецендентам) – построить функцию, приближающую целевую функцию, те получить так называемую решающую функцию, при этом работать должно не только на обучающей, но и на тестовой выборке (на всем множестве, обучив на подмножестве).



Эмпирический риск (ошибка классификации). Эмпирический риск – реальная величина, ошибка классификации – теоретическая. К примеру, ошибочно отнесен объект к другому классу на этапе обучения – это ошибка

классификации или обучения. Эмпирический риск – тестируем каждый объект из обучающей выборки, оцениваем, насколько каждый объект из обучающей выборки похож на реальный.

L(a(xi,yi) – функция потерь, считаем для каждого элемента выборки.

Итого мы приходим к задаче оптимизации, по сути, РО = хитрая задача оптимизации, те найти $\mu : X^l \to a$ для которой эмпирический риск $R_{emp}(a, X^l) = \frac{1}{k}\sum_{i=1}^{l} L(a(x_i), y_i)$ будет минимален.

a*:

$L = Y \times Y \to R$ характеризует отличие правильного ответа от ответа данного построенным отображением

$L(y, y') = [y \neq y']$ - индикатор потери,

$L(y, y') = |y - y'|$ - отклонение

$L(y, y') = (y - y')^2$ - квадратичное отклонение

$L(y, y') = [|y - y'| > \delta]$ - индикатор существенного отклонения

3/20

Любая задача машинного обучения сводится к такой постановке задачи, усложняются только условия – число параметров и т.п.

Гипотез существует неограниченное количество, из них нас удовлетворить может тоже очень большое количество гипотез.

Обобщающая способность алгоритма – если алгоритм на тестовой выборке обеспечивает малую или хотя бы предсказуемую эмпирическую ошибку, то такой алгоритм можно считать способным к обобщению.

Проблема обобщения – нет гарантий, что малый эмпирический риск на тестовой выборке будет соответствовать ожидаемому, то есть истинный эмпирический риск будет мал.

В мат терминах разделяющая граница представляет собой уравнение гиперплоскости $y(x) = w^T x + w_0$ (линейная дискриминантная плоскость).

Подставляя в уравнение гиперплоскости вектор признаков объекта получаем число, характеризующее расстояние до плоскости – если положительное число, то объект с одной стороны гиперплоскости, если отрицательное – то с другой.

Что делать, если классов больше двух? Можно построить столько гиперплоскостей, сколько классов (один против всех). Построить гиперплоскость каждый против каждого – разделить каждый от каждого, очень много классификатором. Можно иначе, но тогда могут получиться области неопределенности – допустим две плоскости, три сектора мы знаем, а в четвертом не было объектов, то есть что там за класс, если попадет туда объект – не ясно. А могут плоскости не пересечься и получится, что в центре область неопределенности.

Для решения регрессий используется библиотека Scikit-learn (sklearn). Библиотека для алгоритмов машинного обучения, множество алгоритмов для обучения с учителем, много по регрессиям и тп.

Тема №5.5. (лаб. 6) Классификация и логистическая регрессия

Признак – результат измерения отличительной характеристики объекта (качественной или количественной). Формально признак – это отображение измеренного значения X на множество допустимых значений Df. Распознаваемый объект – вектор (или можно сказать точка) в некотором пространстве.

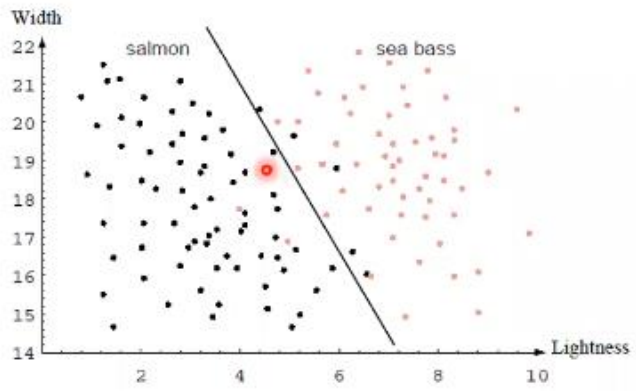
Образ – совокупность признаков для объекта. В задачах классификации это пара переменных $(\vec{x}, \omega)$ где x это вектор признаков (набор наблюдений), омега – понятие за пределами наблюдения (метка наблюдения).

Рассмотрим задачу классификации на примере рыб. Предположим, что требуется различать морских окуней (sea bass) и лососей (salmon), но не сказано, каким образом.

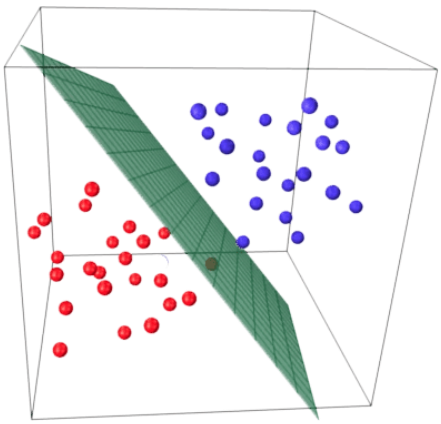
Можно попытаться по цвету, размеру глаз, рта, головы и т.п. Но проще всего ориентироваться на ширину и яркость чешуи рыб, что в случае с морским окунем и лососем дает достаточно четкие различия. В этом случае, формализуя задачу, вектор признаков для каждой рыбы будет состоять из ширины и яркости рыбы на изображении.

Примечание: в реальных задачах векторы признаков часто содержат куда большее число признаков.

Построим график, на котором по оси X будем откладывать яркость рыб, по оси Y – их ширину. Просмотрев множество фотографий для каждой рыбы будем ставить точку на этом графике (координата X точки равна яркости рыбы, координата Y – ее ширине), причем черным окрасим точки для лососей и красным – для окуней.

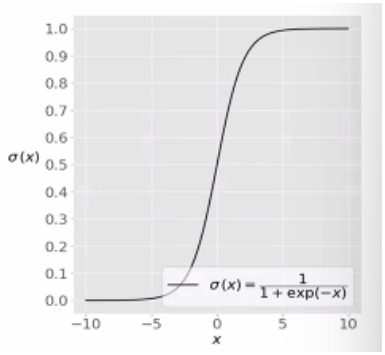


На графике также проведена прямая – это так называемая разделяющая граница классов. Все точки слева от нее считаем относящимися к лососям, справа – к окуням. Получить данную границу можно разными способами, в библиотеке sklearn для python множество способов ее построить, причем разной формы (не обязательно прямой). Вообще такую границу называют гиперплоскостью, что означает ее линейны (плоский) характер изменения в пространстве любой размерности. В случае 2D пространства, как в примере выше, гиперплоскость является прямой, но если бы у



нас были три признака для лососей, то гиперплоскость была бы плоскостью в 3D пространстве, как показано на скриншоте.

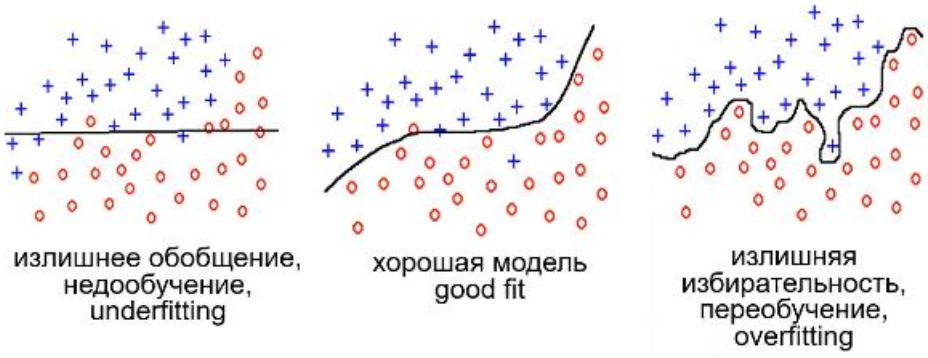
В данном примере используется так называемая логистическая регрессия, представленная на скриншоте. Особым математическим образом для всех объектов по их векторам признаков вычисляется их вероятность отнесения к одному из двух классов (скажем, ко второму), выбирается пороговое значение (часто 0,5), и если вероятность для объекта меньше этого порога, то данный объект не попадает в исходный класс (второй в данном примере) и, соответственно, принадлежит к оставшемуся (первому). Метод позволяет отделять 2 класса друг от друга, но, при необходимости, может разделять и больше – в этом случае проводится не одна, а большее число разделяющих гиперплоскостей.



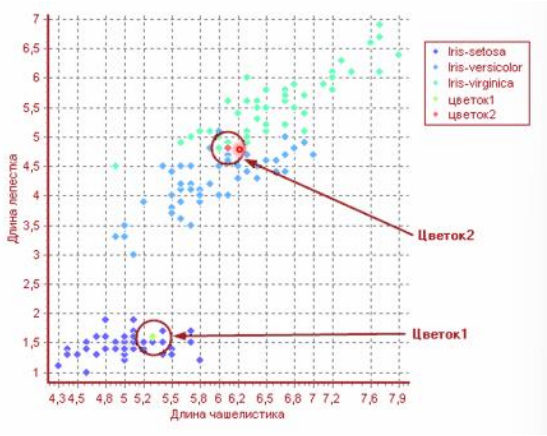
Для получения модели, выполняющей классификацию (=классификатор), требуется ее обучение. Для этого все доступные объекты делят на обучающую выборку, по которой модель «учится» различать классы (в этой выборке вы намеренно указываете, к какому классу какой объект относится) и тестовую, по которой проверяется точность работы модели (процент верно классифицированных объектов). В рамках данной задачи возникают два негативных понятия – пере- и недообучение.

Переобучение (overtraining, overfitting) – повышенная средняя ошибка при обработке тестовой выборки по сравнению с обучающей (модель слишком подстроилась под обучающие данные, которые могут содержать свои особенности, не содержащиеся в тестовой выборке, например, по случайности, в обучающую выборку попали только неширокие лососи, из-за чего любой широкий лосось из тестовой будет принят за окуня).

Недообучение (underfitting) – обучение не обеспечивает малую среднюю ошибку, возникает в недостаточно сложных моделях.



Тема №5.6. (лаб. 7) kNN



Рассмотрим вновь датасет ирисов Фишера. Напомним, в нем имеется 3 класса цветов, причем два верхних класса достаточно близки друг к другу.

Если ранее мы осуществляли классификацию линейной регрессией, при которой классы отделяются гиперплоскостью (прямой в данном двухмерном примере), то имеются иные методы, позволяющие получить куда более сложную конфигурацию границы.

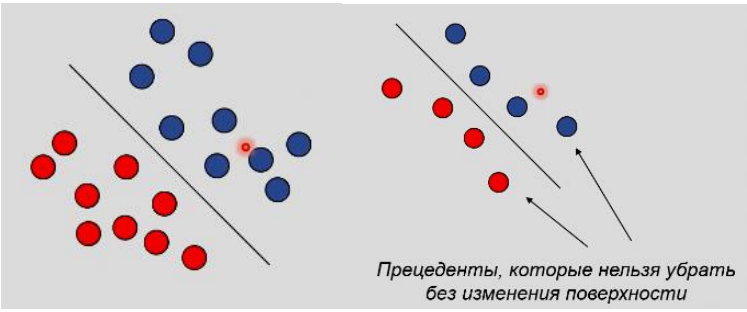
Один из самых простых классификаторов с нелинейной границей является классификатор k-ближайших соседей (чаще kNN – k-Nearest Neighbours). Согласно kNN, отнести отдельный объект следует к тому классу, которому принадлежит большинство из k его ближайших соседей в пространстве. То есть, классификатор «смотрит», сколько вокруг выбранного объекта на фиксированном расстоянии элементов разных классов и относит выбранный объект к тому классу, элементов которого оказывается больше вблизи объекта. Эти элементы и называют соседями.

Классификация по 1NN (k=1) ведет к ошибкам, одного соседа недостаточно. Типичное k – это нечетное число. Если классов более двух, то следует увеличивать k. Метод нестабилен для малых выборок, поскольку с увеличением k «по соседству» становится больше то элементов одного класса, то другого.

kNN устойчив к аномалиям, выбросам, его результаты возможно легко интерпретировать (объект отнесен к классу X, поскольку рядом с ним больше всего объектов класса X). Недостатки – нужна репрезентативная выборка, но большая выборка дает усложнение вычислений.

Тема №5.7. (лаб. 7) Метод опорных векторов

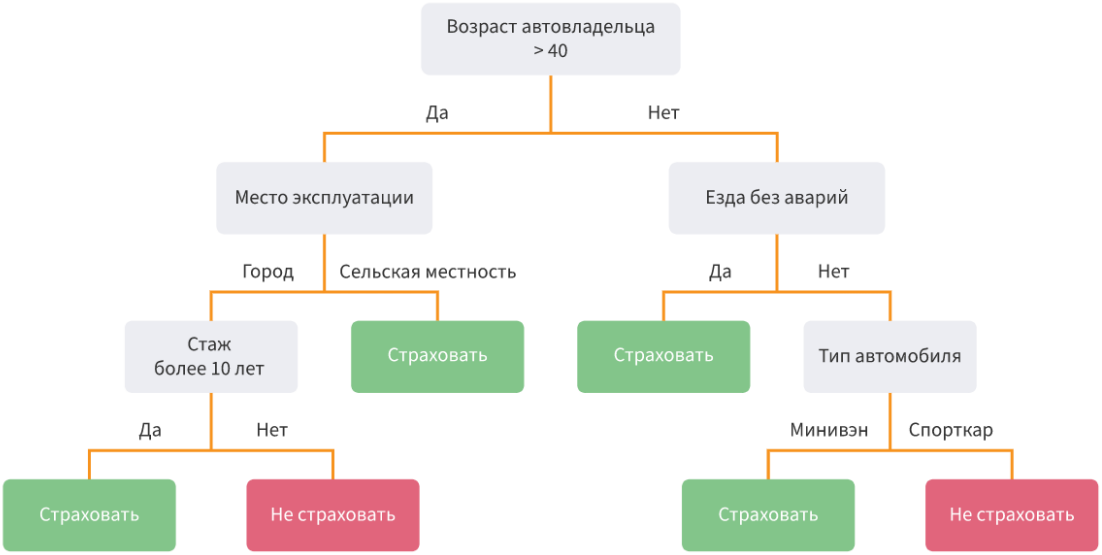
Если строить границу между двумя классами как равноудаленную от всех объектов гиперплоскость (или перпендикуляр при виде на плоскости), то не обязательно учитывать все объекты. Последовательно убирая объекты из выборки, начиная с самых удаленных, смотрим, меняется ли граница. Пока граница не меняется – ничего не делаем, когда она начинает деформироваться – останавливаемся, так как это свидетельствует о том, что исключили важные объекты, которые исключать нельзя было. Те объекты, которые убрать нельзя – опорные векторы (так как объекты в пространстве классификатора это и есть наборы или векторы признаков).



Метод опорных векторов с точки зрения математики – частный случай регуляризации по Тихонову. Вводится невязка, характеризующая ширину разделяющей полосы (вместо прямой, как гиперплоскости в проекции – полоса, прямоугольник), которая уменьшается, в пределе вырождаясь в прямую. Регуляризация по Тихонову – задаем невязку, минимизируем, получаем решение. Есть случаи линейной разделимости и линейной неразделимости. SVM дает наиболее точный результат среди линейных классификаторов. Также есть вариация для построения нелинейной разделяющей границы.

Тема №5.8. (лаб. 7) Древо решений

Древо решений – связанный итеративный алгоритм различения по атрибутам объектов. Оно классифицирует объекты по следующей логике: все правила отнесения к тому или иному классу выстраиваются в виде древа, по каждому признаку проверяется, выполняется ли то или иное условие, по проходу всех узлов формируется вывод, к какому классу относится объект. Иллюстрируется древо решений на примере классификатора возможности страхования автовладельца (два класса – страховать/не страховать), имеющего вектор признаков: возраст (число), место эксплуатации (город/село), стаж (число), езда без аварий (да/нет), тип автомобиля (минивэн/спорткар).



Алгоритм обладает множеством плюсов:

+ интуитивность;

+ можно извлекать правила на естественном языке;

+ не требует выбора входных атрибутов;

+ точность;

+ есть масштабируемые алгоритмы (параллельно на большие выборки);

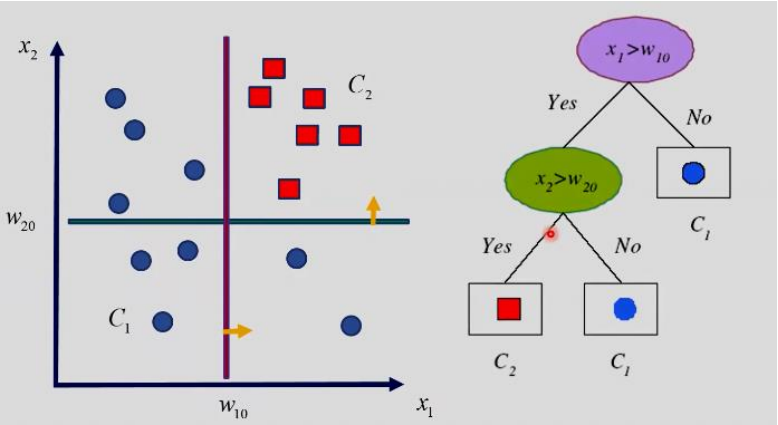
+ быстрое обучение;

+ классифицирует по числам и категориям.

Построение древа. Выбирается целевой атрибут, по нему правило – критерий расщепления выборки (допустим порог, свыше которого считается, что атрибут = правда), делим выборку на две части и исключаем из выборки атрибут расщепления. Сокращена размерность на 1 (в векторах признаков

далее не учитывается тот параметр, по которому расщепили выборку), но получили две выборки. Далее итерационно повторяем. Следуем от корня (начало), проверяем все правила-атрибуты и попадаем в лист (конец, указывает выявленный класс).

В примере сначала по признаку x1 задаем правило-атрибут, затем по x2. Тем самым формируется две границы, плоскость делится на 4 части, в 3 из которых лежит класс C1, в оставшейся четверти – класс C2.



Лучший показатель атрибута – насколько его введение расщепляет выборку, насколько приближает к концу разделения. Показатель информативности атрибута в древе – энтропия, мера неоднородности. Хороший атрибут снижает энтропию.

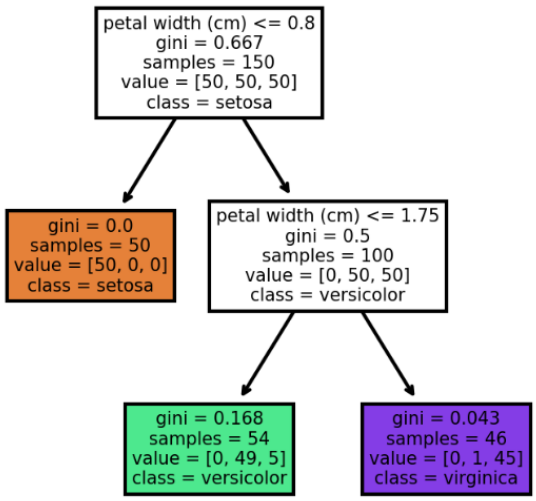
Индекс Джини (Gini) – показывает, насколько часто случайный объект выборки будет неверно распознан при условии взятия целевых значений из конкретного распределения. Показывает расстояние между целевым распределением и распределением предсказаний модели. Лучшим будет такое расщепление, для которого индекс Джинни минимален.

При проектировании древа решений можно задавать параметры остановки: ранняя остановка по достижении критерия (процента распознанных объектов); ограничение глубины дерева (максимальное число расщеплений для остановки); задание минимального числа узлов.

Если Джинни = 0, то останавливаемся в этой ветке, такой конец называют листком. Древо показывает, что задача классификации может решиться определением информативности признаков.

Ниже пример классификации датасета ирисов Фишера с учетом Джинни. На нем в каждом узле (если узел – не лист) первой строкой указано условие, по которому производится расщепление. В первом узле расщепление происходит по ширине лепестка (petal width). Согласно нему, если ширина лепестка<=0.8, то объект относится к классу setosa. На то, что выполненного условия (атрибута) достаточно для классификации указывает Джинни – в левом узле он равен нулю, соответственно все цветы с шириной лепестка<=0.8 можно однозначно отнести к setosa и данный узел расщеплять не требуется.

Если ширина лепестка>0.8 – алгоритм переходит во второй узел, где опять проверяется ширина лепестка, но по другому условию (<=1.75). По этому атрибуту выборка расщепляется на два узла, каждому из которых соответствует один из оставшихся классов. Величина Джинни, в случае идеальной классификации (точность=1.0) должен быть равен 0 во всех листах (конечных узлах) дерева.

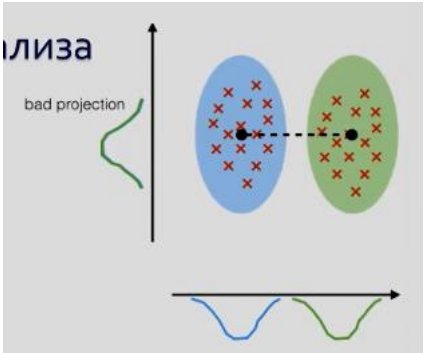


Тема №5.9. (лаб. 7) Наивный байесовский классификатор

Еще одним широко распространенным классификатором является так называемый наивный Байесовский классификатор. Он основывается на формуле Байеса из работы 5, причем его алгоритм работы допускает, что вероятности отнесения объекта к любому классу независимы (что называют наивным предположением, откуда классификатор и получил имя). Используется для поиска оптимальной линейной границы, работает только

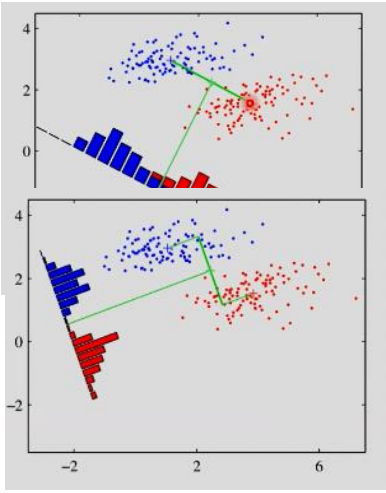
для реально независимых признаков (к примеру, цвет и форма – независимы, а форма и ширина могут зависеть друг от друга). Наивный байесовский классификатор является гауссовским классификатором, поэтому в литературе данные обозначения идентичны. Несмотря на наивный вид и очень упрощенные условия, наивные байесовские классификаторы часто работают намного лучше нейронных сетей во многих сложных жизненных ситуациях. Достоинством наивного байесовского классификатора является малое количество необходимых для обучения данных.

Тема №6. (лаб. 8) Нормализация и снижение размерности данных для классификации



Если делить выборку согласно проекции слева графика, то классифицировать объекты не удастся, так как классы накладываются друг на друга (как и признаки объектов в этих классах), из-за чего объекты становятся неотличимы. А по нижней проекции задача решается легко, так как видно четкое разделение синего и зеленого классов.

Классификация – метод построения информативного признака (наиболее четко разделяющего объекты разных классов). Самый простой метод построить такой признак – выбрать его. Фишер: классификация – метод радикального сокращения размерности (построение суперпризнака, по одному которому классифицировать объекты легко).

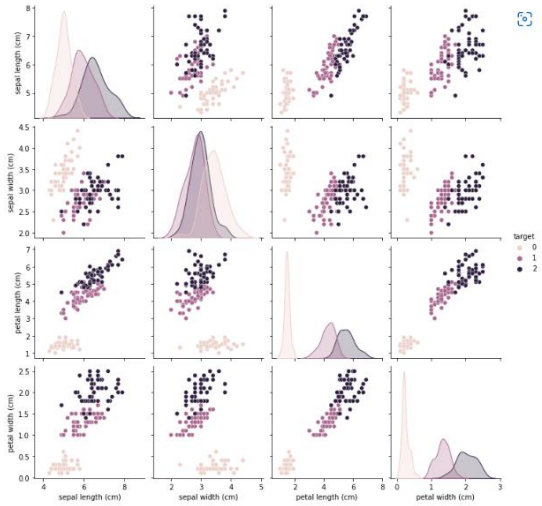


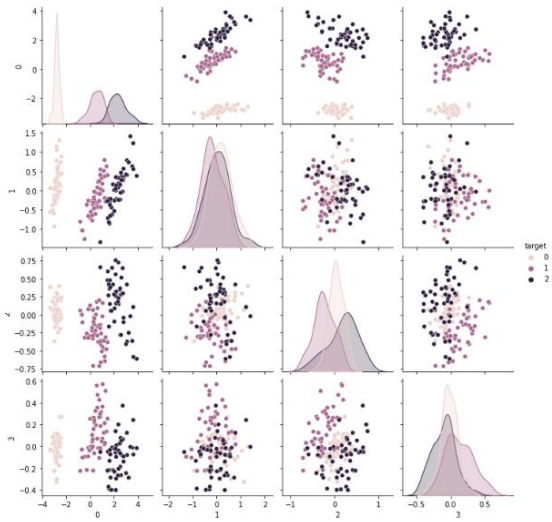
Как поделить два множества на примере двухмерного пространства признаков (то есть объекты классифицируются по двум их признакам)? Пополам – это самое простое. Но, поскольку плоскость – пространство двухмерное, можно разделяющую границу двигать, поворачивать, добиваясь лучшего разделения. На первом

скриншоте (сверху) видно, что зеленая разделяющая граница не обеспечивает лучшего разделения, хотя между точками красного и синего класса явно можно провести прямую, четко отделяющую точки одного класса от другого. О плохой границе говорят и наложенные гистограммы точек классов (то есть, при такой границе, в месте пересечения этих гистограмм классификатор будет ошибочно относить объект к неверному классу). Нижний скриншот показывает границу куда лучшую, поскольку гистограммы точек четко разделены.

Таким образом, для построения оптимальной границы можно взять такой серединный перпендикуляр (граница-прямая, перпендикулярная к отрезку, соединяющему по одной точке из разных классов), чтобы дисперсия каждого кластера (или класса) была меньше (а значит, меньше и ширина гистограмм).

Метод главных компонент (МГК, от англ. Principal Component Analysis, PCA) – алгебраическая процедура оптимального линейного сокращения размерности, реализует поиск оптимальной границы. По МГК строится проекция на пространство заданной размерности, в которой максимизируется дисперсия, строится проекция с минимальным суммарным расстоянием до всех точек. По МГК, если размерность оставить той же, что и у исходных данных, новые координаты получаются как сумма с некоторыми весами прежних признаков, тем самым и осуществляется поворот. То есть новый признак уже не является исходным (допустим шириной лепестка в ирисах Фишера), а комбинацией всех прежних признаков с некоторыми весами. Пример применения МГК к признакам ирисов Фишера с сохранением прежней размерности (было 4 признака, оставляем 4 после МГК, а можно уменьшить их число).



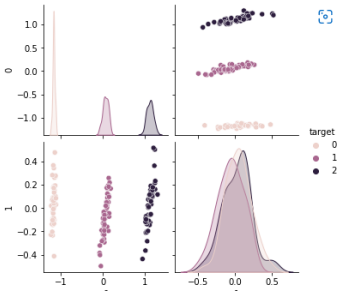


Исходный После МГК

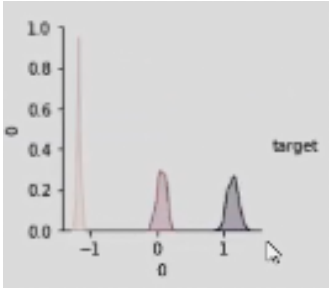
В ирисах Фишера 3 класса, после МГК отделяется хорошо один из них по первому признаку (обратите внимание, после МГК название признака удалено, поскольку первый признак, как и остальные, после МГК не совпадают с исходными, как было сказано выше).

И если первый класс отделен, то остальные не очень. МГК чувствителен к абсолютному значению признака, у ирисов Фишера признаки в разном масштабе (длина принимает значение в куда большем диапазоне значений, чем ширина, из-за чего даже после МГК длина имеет больший вес или вклад при классификации). Для решения проблемы требуется нормировка – приведение всех значений признаков к одному диапазону (обычно 0…1).

После нормировки и МГК со снижением размерности до 2 получаем график на скриншоте. Уже первый полученный признак можно назвать суперпризнаком, его достаточно, чтобы классификатор дал нулевую ошибку, так как все классы абсолютно четко отделены друг от друга.



А взяв МГК со снижением размерности до 1 получим только суперпризнак. МГК и нормировка используются на многих выборках, почти всегда дает положительный результат. Также МГК позволяет определить значимость признаков.



Тема №7. Виды ошибок и мер точности в задачах машинного обучения

Чаще всего существует основной класс, на выявление которого и направлена задача (не раскидать по классам все объекты, а обнаружить среди них лишь нужный класс). В этом случае остальные классы – вторичные.

Формула Байеса провоцирует резкий перевес в сторону того класса, где содержится больше всего объектов из выборки, поэтому при малом числе объектов классификатор на базе формулы Байеса сильно ошибается.

Ошибка первого рода (промах) – пропуск искомого объекта, ложноположительный результат.

Ошибка второго рода – классификация объекта из основного класса как вторичный класс, ложноотрицательный результат.

Погрешности 1 и 2 рода оценивают отдельно для несбалансированных классов (у которых сильный перевес по числу объектов в одну сторону, как в примере с больными всего 1% болен и здоровы 99%).

Допустим для того же примера с больным вероятность ошибки 1 рода 0,5, а 2го – 0, то по формуле Байеса вероятность общей ошибки (или просто общая ошибка) равна 0,005. Т.е. результаты неверны из-за дисбаланса.

Принятые сокращения: TP – True Positive, TN – True Negative, два верных исхода, FP – False Positive (ошибка 1 рода), FN – False Negative (2 рода).

Точность в самом интегральном смысле принято оценивать по формуле Accuracy.

$$Accuracy = \frac{1}{l}\sum_{i=1}^{l}[a(x_i) = y_i] = \frac{TP + TN}{TP + TN + FP + FN}$$

Такой подход никак не учитывает объем выборки и цену ошибки на разных классах (важнее бывает определенную ошибку к минимуму свести).

Устраняет недостатки другая формула точности Precision. Это доля от объектов, которые классификатор отнес к какому-то классу, которая реально относится к этому классу.

$$PRECISION = \frac{TP}{TP + FP}$$

Другая характеристика – полнота (recall). Показывает долю верно классифицированных объектов рассматриваемого класса (тут учитывается доля, которая улетела ошибочно в другие классы FN).

$$RECALL = \frac{TP}{TP + FN}$$

Recall – показывает способность обнаруживать алгоритмом класс вообще, Precision – показывает способность отличить искомый класс от остальных.

Усредняет обе оценки формула среднего гармонического. Она достигает максимум, когда обе метрики 1. Равна 0, если хотя бы одна из метрик равна 0.

- **F-мера** — среднее гармоническое:

$$F_\beta = (1 + \beta^2)\frac{precision * recall}{\beta^2 precision + recall}$$

β в данном случае определяет вес точности в метрике

Если выборка несбалансирована, прибегают к расширению этой выборки. Если расширить нечем – допустим нет больше фоток распознаваемой птицы, – то к выборке добавляют ее искаженную копию – с точки зрения классификатора это совершенно иные объекты будут, тем самым выборка расширяется оригинальными объектами. Для изображений – наложение шума, изменение геометрии, перспективы, цветности и т.п. Аугментация – искусственное расширение выборки, она улучшает выборку

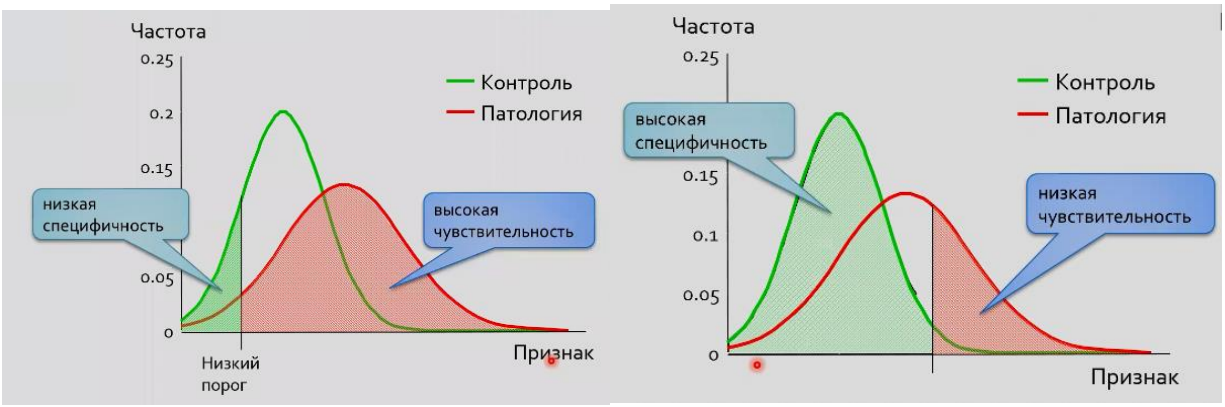
Чувствительность sensitivity (SE) – вероятность дать верный ответ на пример основного класса (верно распознать объект основного класса).

Избирательность specificity (SP) – вероятность дать верный ответ на пример вторичного класса (верно распознать объект вторичного).

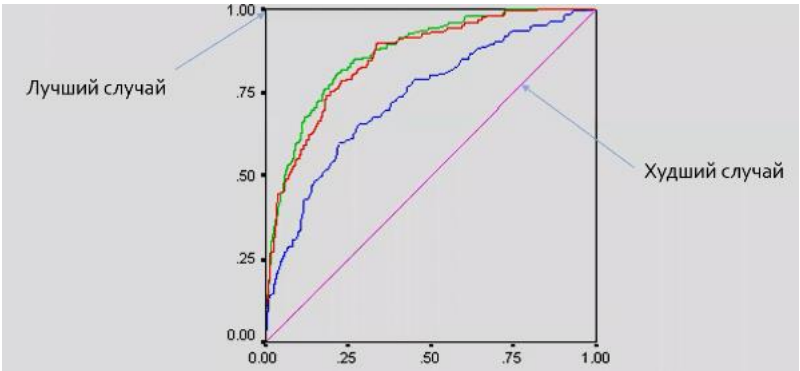
TPR – True Positive Rate – скорость отзыва положительных образцов, - соотношение положительного результата оценки среди всех положительных образцов.

FPR – False Positive Rate – скорость отзыва отрицательных образцов (уровень фальсификации), – отношение ошибочно оцененных положительных значений среди всех фактически отрицательных выборок.

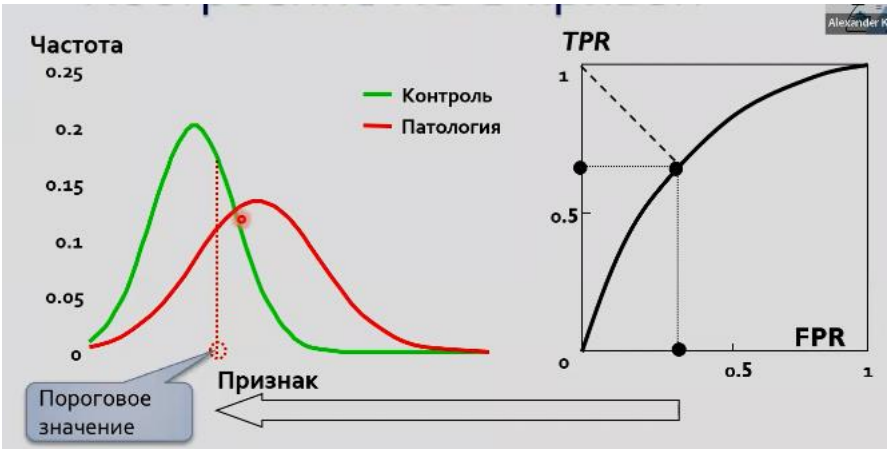
Положение границы между классами существенно влияет на озвученные параметры (как меры точности классификатора).



ROC curve – Receiver Operating Characteristic – зависимость TPR от FPR при плавном изменении порога. Если классификатор работает нормально, то он всегда будет давать ROC кривую выпуклую, чуть лучше, чем прямая в худшем случае. Под худшим случаем быть ничего не может.

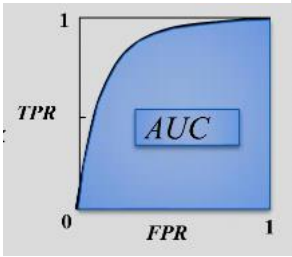


По ROC выбирается оптимальный порог – точка на данной кривой, которая ближе всего к левому верхнему углу с координатами 0,1. Так по этой кривой можно графически определить оптимум порога.



По форме ROC можно определить оптимальный признак. Как в примере ниже – в первом случае вообще не стоит использовать признак, во втором – можно ограничиться только признаком, обеспечивающим такую кривую, а третий пример – можно провести несколько тестов (по разным признакам) и сравнить по форме ROC точность – очевидно что ближайшая точка 1 теста ближе к 0,1, чем ближайшая второго теста, поэтому признак из первого теста предпочтительней.

Интегральный показатель прогностической эффективности признака AUC – Area Under Curve, площадь под ROC. Есть сопоставление AUC и эффективности, таблица ниже.



AUC	
0,5	Случайный классификатор
0,5-0,6	Плохой классификатор
0,6-0,7	Средний классификатор
0,7-0,8	Хороший классификатор
>0.8	Отличный классификатор

Экспериментальная оценка качества алгоритма классификации.

Теоретические оценки – завышены, рассчитываются трудно, малоинформативны. Так как параметры признаков – случайные величины, ограничиться одним наблюдением (опытом на одной выборке) для оценки точности не получится, если выборка не громадная.

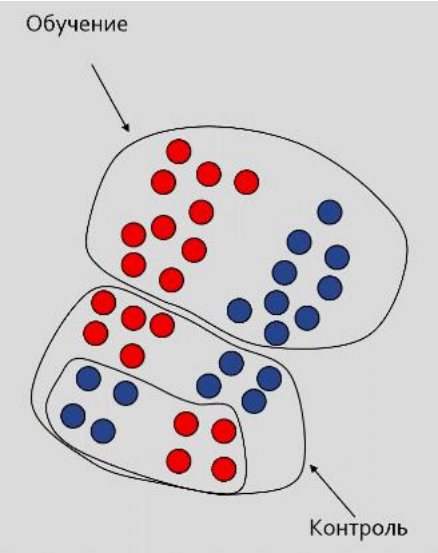
Для конкретной задачи лучше экспериментальные методы: удерживание, скользящий контроль и др.

Обучение по прецедентам – обучение по известным исходам (наличие обучающей и тестовой выборок).

Метод удерживания – часть изначальной выборки удерживается в качестве тестовой, те кусок выборки (подмножество) не используется при обучении, но только он используется при оценке точности. Целиком выборку лучше не использовать для обучения, так как скорее всего это приведет к переобучению, чрезмерной заточке классификатора именно на эту выборку, а не на интересующие признаки.

+ быстро и просто;

- если выделяющиеся случаи попадут только в одну выборку, то ошибка может быть высокой или, иначе, сильно смещенной (допустим сложные случаи изображений в тестовую попали, их всего пару штук и на них классификатор не обучался, в тестовой выборке он их не распознает).



Для исключения недостатка можно попробовать менять местами обучающую и тестовую выборки. Это позволяют методы повторного удерживания и скользящего контроля.

Повторное удерживание – меняем несколько раз выборку и усредняем результаты.

- остается вероятность непопадания особых случаев в подмножество;

- сильная случайность: как разбивать, сколько раз повторять чтобы точно исключить недообучение и т.п.

Скользящий контроль (CV, cross-validation) – развитие идеи повторного удержания, эмпирическое оценивание обучающей способности по прецедентам. Бьем выборку на множество частей и последовательно используем одну из частей для контроля, а остальные – для обучения. Статистика показывает, что CV лучше всего дает оценки точности. Результат – средняя ошибка по всем итерациям. Резко снижаем зависимость от нестандартных выборок.

Предельный случай CV – размер подмножеств = 1 объект выборки, дает очень хорошую точность.

Эвристические алгоритмы повышения точности оценок – бустинг (деление выборки на части, в одной из которых алгоритм работает хорошо, в другой – плохо; для части с плохо работающим алгоритмом текущим строим новый алгоритм, отделяя в этом подмножестве область, на которой новый

алгоритм плохо работает и т.д.); бэггинг (bootstrap aggregation) – отдельные классификаторы вводятся сразу и используются параллельно, не последовательно (как в бустинге), а ошибка компенсируется при голосовании (то есть в качестве результата выбирается лучшая оценка).

Тема №8.1. Возникновение нейронных сетей

Классические проблемы машинного обучения – отсутствие общего подхода к формальному описанию (недостаточно алгебры); нужны новые алгоритмы (более точные и долгие). В 20 веке для решения этой проблемы предложен метод опорных векторов.

Решения: создание общего подхода к описанию или анализ существующих алгоритмов и их проблем.

Математика 18 века рассчитана была на простые ручные вычисления, новые же алгоритмы нужны такие, чтобы точность не зависела от времени вычислений. Таким решением можно считать численные методы во многом.

Кибернетика – изначально как изучение структур общества, потом как изучение биологических систем, далее применительно к людям и в итоге возникла идея о том, что можно применить знания о работе мозга человека для распознавания образов.

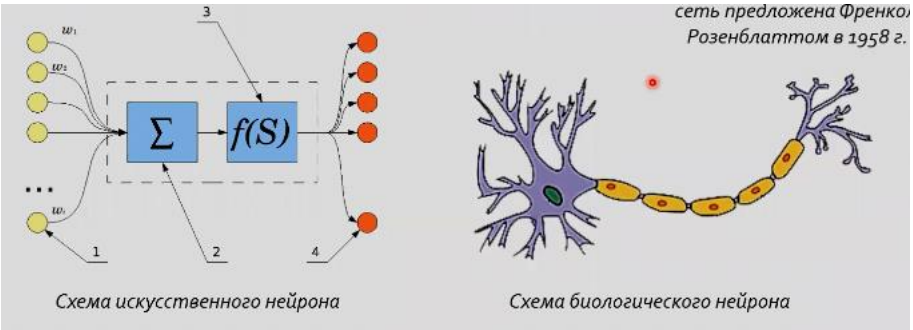
1872-1895 – понятие нейрона, первые предположения о принципе работы.

Искусственная нейронная сеть (ИНС, далее НС) – матмодель и ее воплощение в ПО или аппаратуре которая построена по принципу организации биологических нейронных сетей – сетей нервных клеток живого организма.

Тема №8.2. (лаб. 9) Модель нейрона и обучение нейросетей

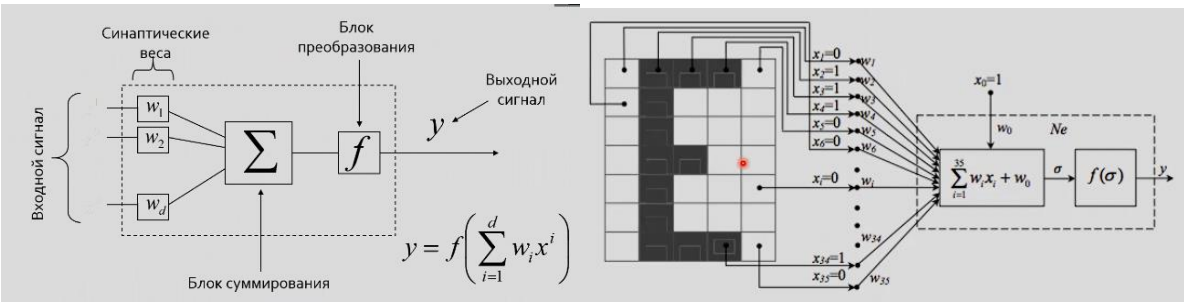
Нейрон – тело, дендриты (отростки) для ввода входных сигналов, отросток аксон для выходного сигнала. Нейрон активируется при подаче входных сигналов. Сформировано в 1943 У. Маккалохом и У. Питтсом, построили матмодель нейрона – входные сигналы суммируются с

синаптическими весами, далее нелинейное преобразование для формирования выхода.

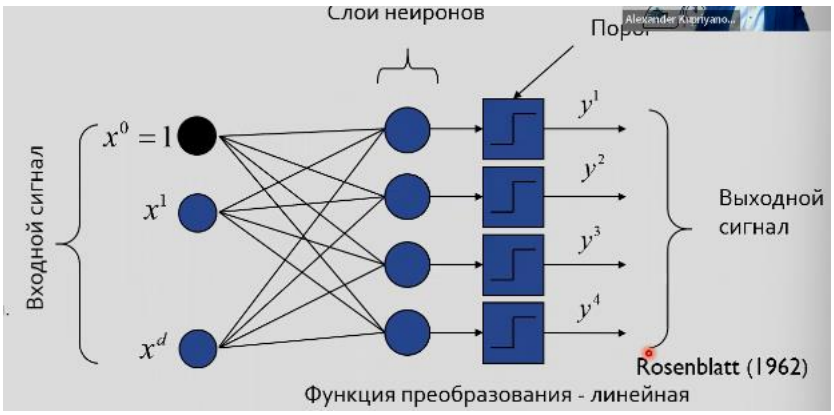


Первая НС предложена Френком Розенблаттом в 1958.

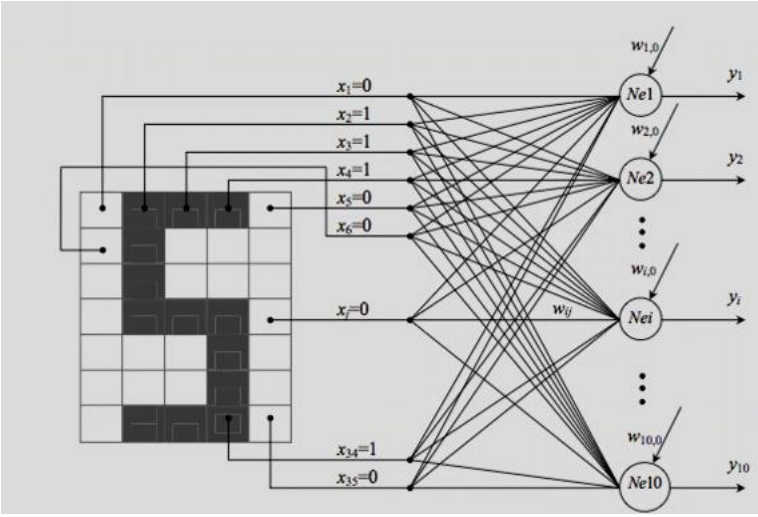
Параметрами модели нейрона являются веса, их нужно изменять. Обучение – итерационно уточняются веса.



Персептрон Розенблата (1962), модифицировал нейрон Маккалоха и Питтса, сделав ее способной обучаться. Изначально – однослойная структура с дискретным входом (фиксированные значения, не любые) и жесткой пороговой функцией. Первые персептроны могли распознавать некоторые буквы латинского алфавита. Исходный кибернетический нейрон размножается, получается персептрон, как на рисунке ниже.

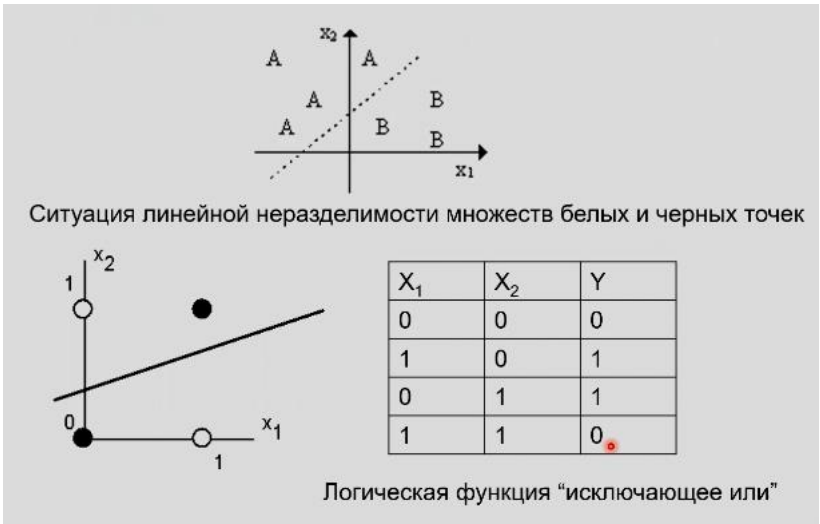


x0 здесь подается как константа сдвига, поскольку без нее не будет работать, она нужна для уравновешивания. Ниже однослойный персептрон для распознавания цифр.



Далее появляется понятие слой, так как логично, что выход персептрона можно подать на другой персептрон.

Персептрон может строить границу произвольно или как сложную логическую функцию, см. рисунок.

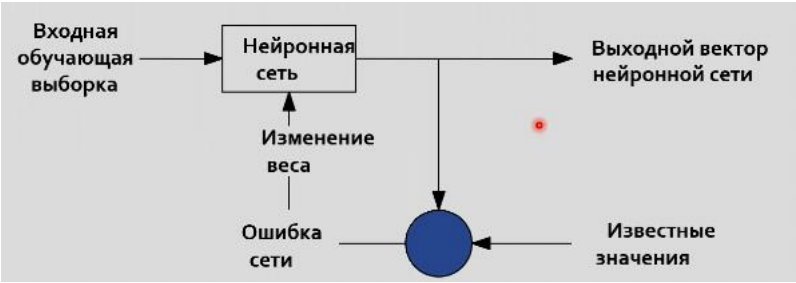


X_1	X_2	Y
0	0	0
1	0	1
0	1	1
1	1	0

Универсальная теорема аппроксимации (Цыбенко 1989) – ИНС прямой связи (feed-forward, без циклов) с одним скрытым слоем может аппроксимировать любую функцию с любой точностью.

Обучение НС заключается в простом алгоритме – считаем ошибку распознавания после теста НС и по ошибке изменяем веса нейронов сети. В первую очередь для решения этой задачи (оптимизаци) используется алгоритм

обратного распространения ошибки, который реализуется методом градиентного спуска. При методе обратного распространения сеть «запускается наоборот» и вместо сигнала распространяется ошибка, которая минимизируется градиентным спуском.



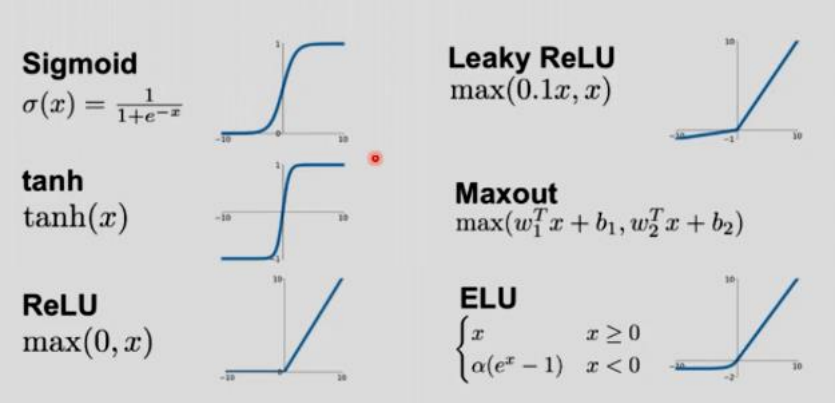
Параметры НС: число входов, функции активации, вид соединений нейронов (вперед, назад), какой вход/выход, что называть ошибкой (функция потерь). Поэтому при построении НС слишком много неясностей и чем дальше обучаем, тем медленнее сеть обучается.

Тема №8.3. (лаб. 9) Функции активации, алгоритмы оптимизации

У каждого нейрона может быть своя функция активации (то есть зависимость между входным сигналом нейрона и выходным). Практически всегда в нейронах одного слоя используют одинаковые функции активации.

Relu – одна из наиболее простых функций активации, из-за чего резко повышает скорость обучения (по сравнению, например, с сигмоидой, которая очень часто используется в качестве функции активации).

Сигмоида (в обобщенном виде называют softmax) – чаще всего использовалась в качестве функции активации долгое время. В отличие от Relu нелинейна. Примеры некоторых функций активаций представлены на скриншоте.



Алгоритмы оптимизации. Метод обратного распространения ошибок модифицировался. Сейчас есть алгоритмы адаптивный градиентный (AdaGrad) для повышения скорости, среднеквадратичное распространение (RMSProp) для повышения точности, Adam – смесь прошлых двух, часто рекомендуется.

+универсальность (можно подать что угодно и какой-то выход будет);

+возможность решать задачи со множеством классов, регрессии и тд;

+высокая параллельность вычислений;

+почти неограниченные возможности модифицирования;

-грубая оценка эмпирического риска – его можем только оценить, доказать какой-то предел информативности НС нельзя;

-проблема локальных минимумов – как и для всех алгоритмов на основе градиентного спуска, если мы попадем в локальный минимум, то из него скорее всего не выйдем, хотя локальный не равен глобальному минимуму;

-очень большая склонность к переобучению.

НС во многом – дилетантский подход к машинному обучению и с точки зрения теории и эксперимента являются ненадежным и неточным алгоритмом.

Тема №8.4. (лаб. 9) Создание нейросети с помощью tensorflow и keras

Опенсорс библиотеки для НС существуют – keras (надстройка tensorflow), Theano, pytorch, caffe, sklearn, tensorflow. Keras дает быстрее всего построить НС.

Создавая НС в keras мы импортируем библиотеку, выбираем модель слоев, потом последовательно добавляем слои. Тут же функции активации указываются.

```
from tensorflow.keras import layers
model = tf.keras.Sequential()
# Добавим к модели полносвязный слой с 64 узлами
model.add(layers.Dense(64, activation='relu'))
# Добавим другой слой:
model.add(layers.Dense(64, activation='relu'))
# Добавим слой softmax с 10 выходами:
model.add(layers.Dense(10, activation='softmax'))
```

Далее происходит обучение компиляцией. Указывается алгоритм оптимизации, функция ошибок, метрики для мониторинга обучения, максимальное количество эпох (итераций обучения). Эпоха – не совсем итерация, это проход по всей выборке, а итерацией в таком алгоритме скорее является обработка одного объекта выборки.

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

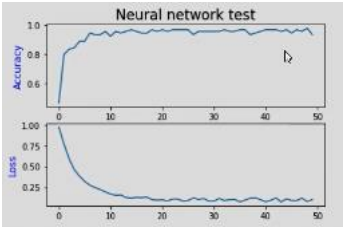
Batch показывает, после скольких объектов корректируются веса (больше батч – реже коррекция и как следствие накапливается ошибка, быстрее исполнение, но медленнее сходимость из-за накопления ошибки).

Можно задать функцию перебора параметров сети с выводом точности, оставить комп вычислять и по результатам выбрать лучший.

Параметр input_shape показывает число признаков в выборке – в случае с ирисом это 4 – две длины и две ширины.

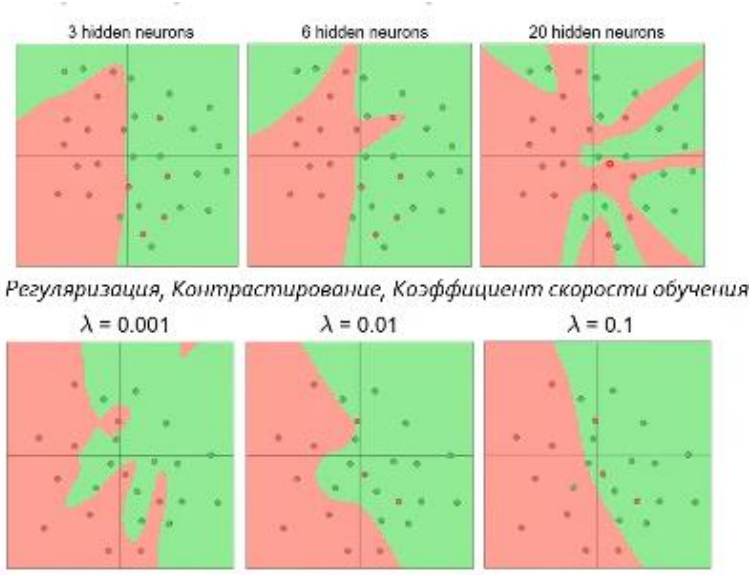
```
# Create the network
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(4,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(3, activation='softmax'))
```

При каком-то числе эпох точность перестает расти и начинает дребезжать, как с наложенной помехой. *При такой картине обучение нужно остановить, так как дальше будет только большая амплитуда дребезжания*.



До 90х НС были непопулярны, так как проигрывали методу опорных векторов в скорости. В 90х выпущена графическая карта для ПК, на ней додумались обучать НС, что существенно увеличило скорость вычислений.

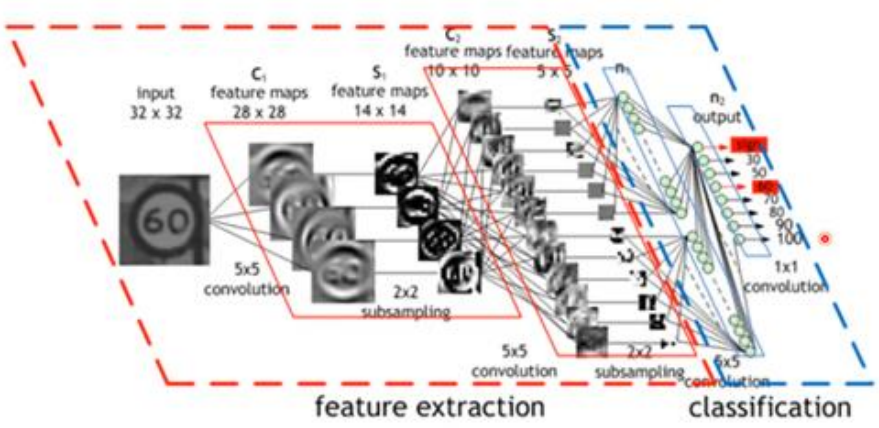
Для исключения переобучения используется регуляризация (вводится невязка с коэффициентом лямбда, МНК решается задача поиска оптимума, по сути регуляризация – как ФНЧ во многом). *Для борьбы с переобучением на определенной эпохе отключается нейрон (dropout) с определенной вероятностью (лямбда).*



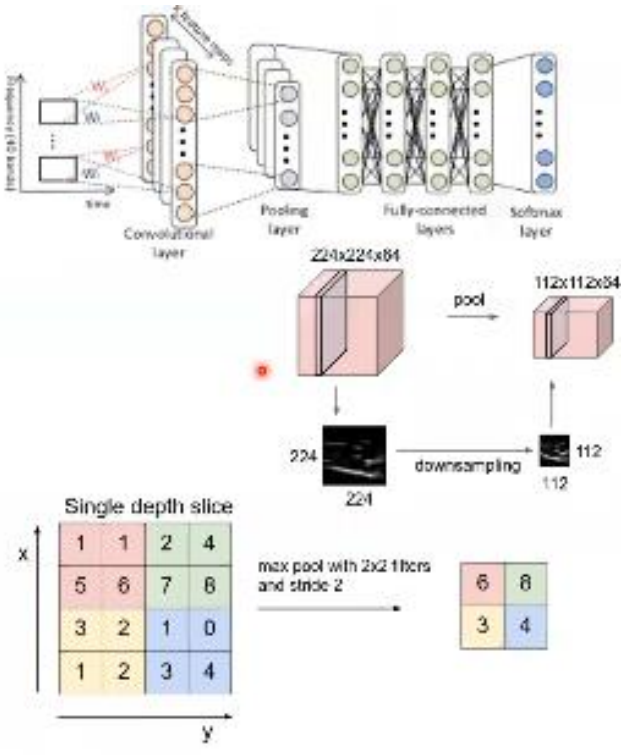
Тема №9.1. (лаб. 10) Сверточные нейронные сети

Для ПК изображение это массив из яркостей цветов (набор чисел). Проблема распознавания образов изображений – искажения объектов, перспективы, освещения, позы различные, ракурсы и т.п. (да и в принципе трудно отличить ту же чашку от шляпы), тем самым выявить признаки зачастую трудно выявить признаки.

Идея: нейроны сами могут формировать, извлекать признаки для классификации, не нужно их вручную выявлять. Это делают сверточные нейронные сети (связь нейронов все со всеми дает возможность выстраивать такие паттерны между весами нейронов в сети, чтобы НС могла сама выявлять признаки).



Вводятся *пулинг-слои (pool), уменьшающие размер задачи, то есть кратно уменьшающие размер изображения, поступившего с предыдущего слоя.*

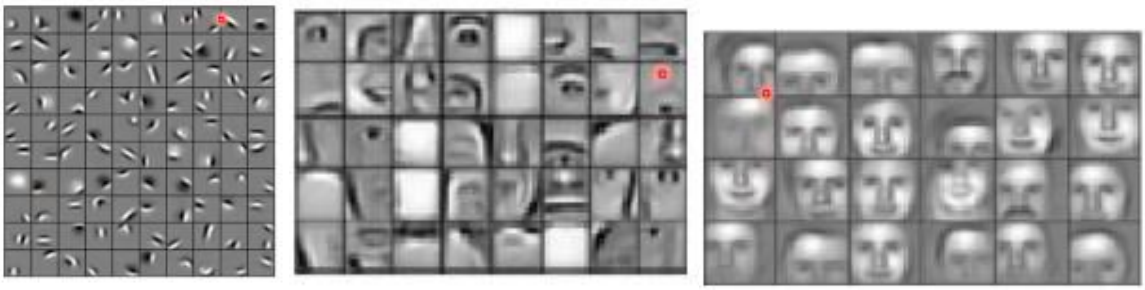


Чаще всего при уменьшении размерности из группы скажем 2x2 выкидываются 3 пиксела, остается один, у которого наибольшее значение (яркость).

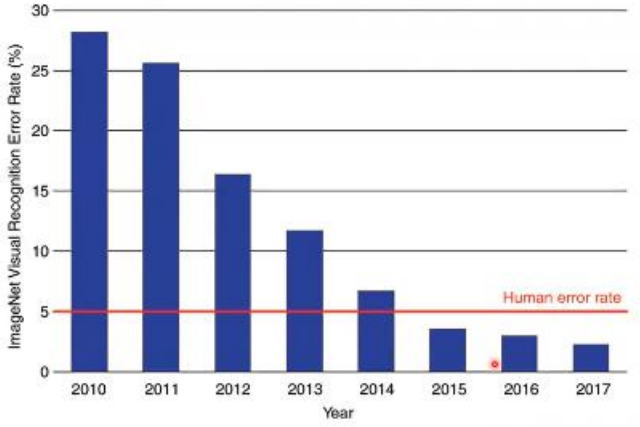
В результате сверточные НС могут делать сегментацию, находить на изображении объекты.

Сверточные НС решают задачи переобучения, выделения признаков. Ниже иллюстрация работы слоев – на первом выделяются характерные

области лица (тени/яркие зоны), на втором – уже проглядываются части лица, на третьем – непосредственно типажи лиц (при этом яркость иллюстрирует значения весов).



Ошибка точности распознавания образов человеком 5%, с 2015 г примерно этот порог пробит сверточной НС. Популярная сеть – AlexNet, ImageNet.



Тема №9.2. Кластеризация изображений

Машинное обучение с учителем – есть объекты и ответы на них, нужно лишь связь построить. Без учителя – есть данные и нужно сделать выводы какие-то.

Пример без учителя – кластеризация фоток дистанционного зондирования (выделение домов, дорог и т.п.).

Index	Class	OA (%)
1	Healthy grass	94.5
2	Stressed grass	88.7
3	Artificial turf	95.7
4	Evergreen trees	96.5
5	Deciduous trees	81.6
6	Bare earth	94.0
7	Water	90.8
8	Residential buildings	83.1
9	Commercial buildings	90.6
10	Roads	70.4
11	Sidewalks	60.3
12	Crosswalks	30.6
13	Major thoroughfares	35.7
14	Highways	72.4
15	Railways	93.2
16	Paved parking lots	65.6
17	Unpaved parking lots	0.0
18	Cars	97.0
19	Trains	93.4
20	Stadium Seats	92.4

Обучение без учителя – есть наблюдения, нужно сделать суждение о них, но заранее не знаем ответы и нельзя оценить качество решения, нечетко поставленная задача.

Задачи обучения без учителя – понижение размерности, анализ плотности распределения, кластеризация.

Кластеризация (кластерный анализ) – автоматическое разбиение элементов множества по принципу схожести, или задача разбиения заданной выборки объектов на непересекающиеся подмножества.

Особенности кластеризации – трудоемкость (часто выполняется чисто перебором всех возможных способов построения кластеров), может являться NP-полной задачей, требующей подбора различных параметров.

Применяются для анализа данных (data mining), группировки и распознавания объектов, поиска и извлечения информации.

Актуальная задача – построить алгоритм без дополнительных параметров и с автоматическим определением числа кластеров.

Использует меру схожести, важнейшая в кластеризации мера. Кластеризация – по сути задача оптимизация, мера схожести должна быть максимальна у одного кластера и минимальна у другого.

Схема кластеризации: 1) выделение характеристик (выбор свойств, уменьшение размерности и нормализация, представление в виде векторов признаков); 2) определение метрики сравнения; 3) разбиение на группы; 4) представление результатов.

Алгоритмов кластеризации много, мы рассмотрим k-средних.

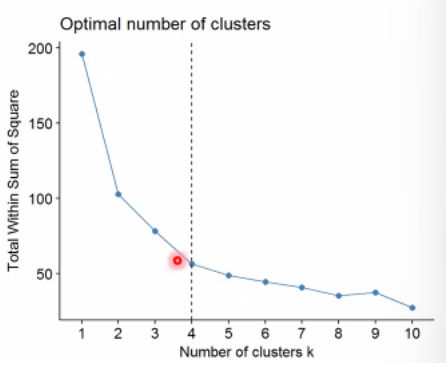
Метод ближайшего соседа KNN: 1) определить центр первого кластера, совпадающий с любым из объектов; 2) определить порог; 3) вычислить расстояние между другим образом и центру кластера; 4) если расстояние больше порога, то объект становится новым центром кластера; 5) далее для остальных образов.

Существенный недостаток – нужно задать порог, а также порядок выбора первого центра кластера.

K-средних устраняет недостатки KNN: 1) случайно выбирается k средних, являющимися центрами масс кластеров, любые k из всех объектов или случайных точек; 2) рассчитывается расстояние для каждого образа; 3) приписать к кластеру расстояние до минимального; 4) пересчитать центры масс; 5) повторить 2 и 3 пока кластеры перестанут меняться. Критерии остановки – 1) отсутствие изменений на шаге 3; 2) минимальное изменение среднеквадратичной ошибки. Бывает, что некоторые точки прыгают на итерациях и алгоритм не останавливается, для чего вводится параметр эпоха, ограничение по числу повторений алгоритма.

Алгоритм однопараметрический, только число классов. Чувствителен к начальному выбору центров масс, не учитывает строение кластеров.

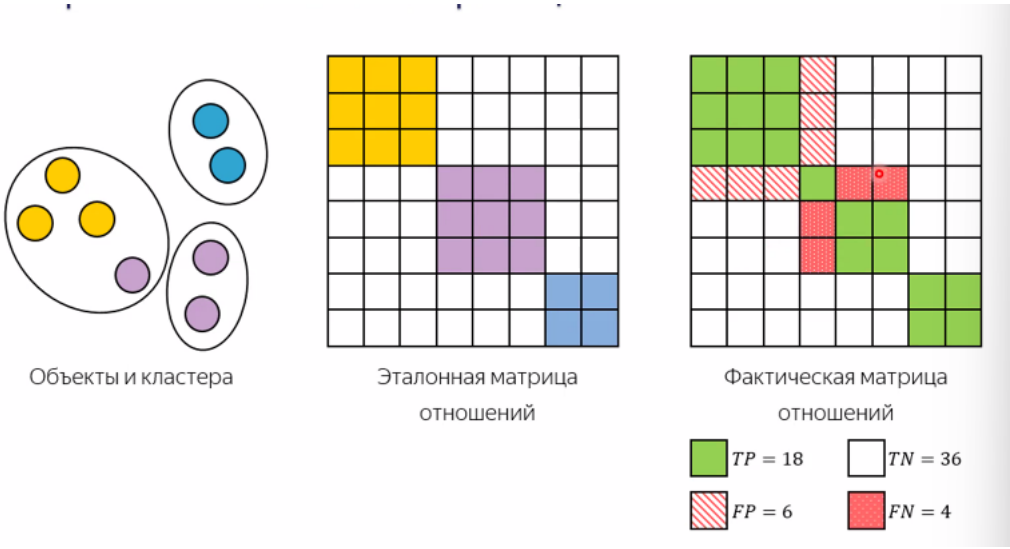
Число кластеров можно определить по методу локтя – характеру изменения дисперсии. Находим по графику дисперсии внутри кластеров в зависимости от числа кластеров дисперсию и находим точку перегиба на графике, это оптимум.



У алгоритма есть развития, например k++, более стабильное разбиение на кластеры.

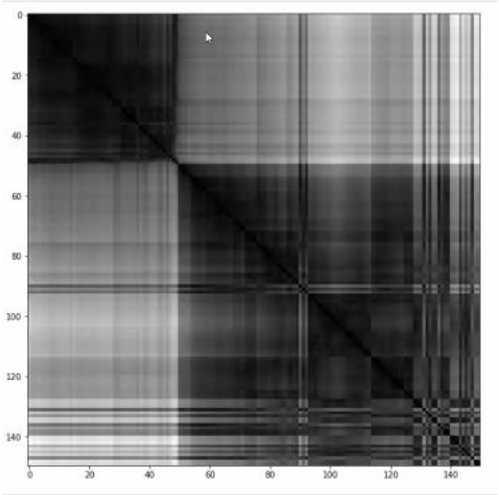
Оценка (метрики) качества кластеризации – однородность (homogeneity) уменьшается при объединении в один кластер двух эталонных; полнота (completeness) уменьшается при разделении эталонного кластера на части; Rag Bag. Это величины субъективные.

Есть подход сведения метрик кластеризации к полноте и точности – эталонная матрица отношений. Сравниваем результат кластеризации через эту матрицу, по разнице получаем TP, FP и тд.



Есть и другие оценки, такие как визуальная. По нему можно построить матрицу отношений.

Если метрика хопкинса меньше 0,25, то все должно делиться. Метрика vat дает картинку в виде матрицы.



Иерархическая кластеризация. Кластеризация смесью нормальных распределений. Expectation Maximization (EM) алгоритм. K-средних не аппроксимируемое гауссовской формой кластеризует плохо, но и k-средних может не учесть особенности кластеров по форме. Разные алгоритмы кластеризации дают разные преимущества.

Приложение. Подключение GPU Nvidia для обучения нейросети

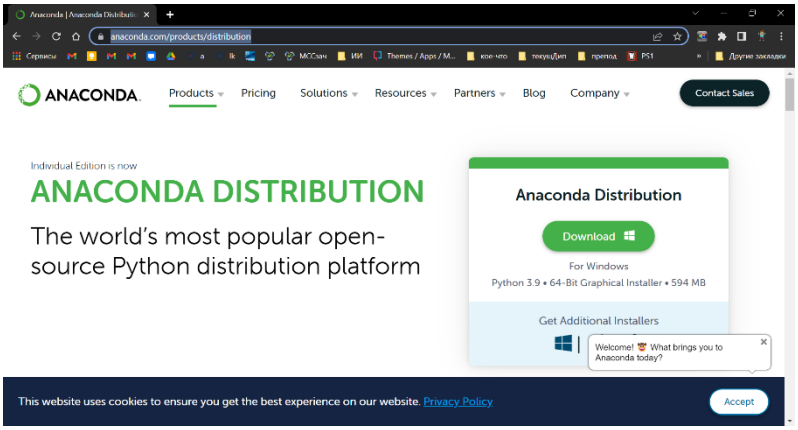
Позднее здесь будет соответствующая инструкция, пока можете прочитать на stackoverflow оригинал (Eng), второй по популярности ответ по ссылке:https://stackoverflow.com/questions/51002045/how-to-make-jupyter-notebook-to-run-on-gpu

Лабораторные работы

Установка ПО и азы работы в JupyterLab/Notebook

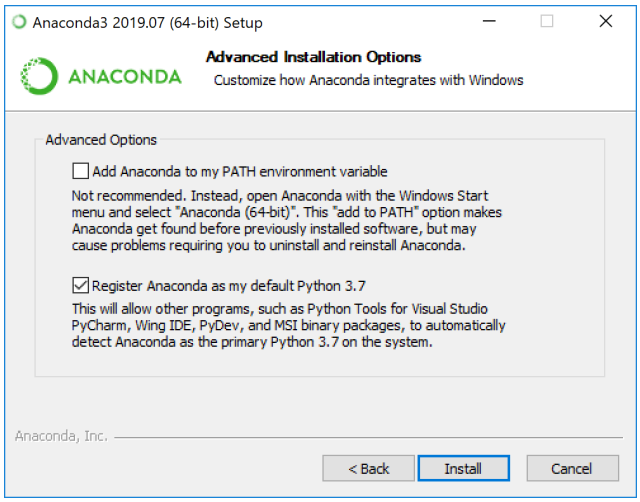
Подготовка к работе

Самый простой способ установить все необходимое для решения математических, логических и научно-инженерных задач на *Python* – использовать программу *Anaconda*. Она является сборником наиболее популярных библиотек и нескольких приложений для научных вычислений средствами Python.

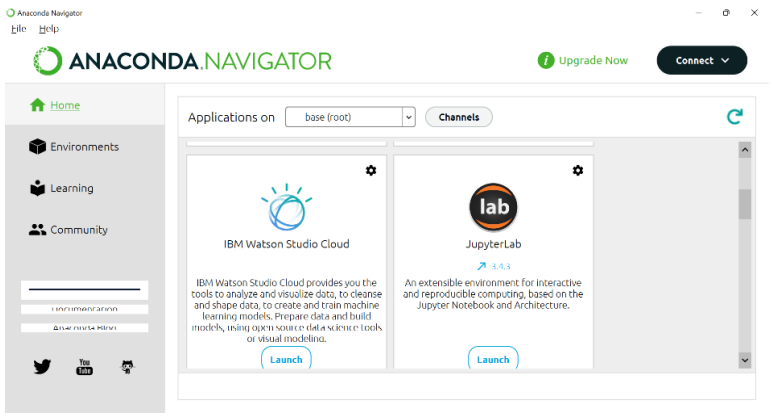


Загрузите пакет данной программы с официального сайта *https://www.anaconda.com/products/distribution*.

Запустите пакет, нажмите *Next*, затем *I Agree*. На следующем экране оставьте выбор на *Just Me* (в некоторых случаях установка дополнительных библиотек в Anaconda не удается при установке *All Users*, так как установщик не может получить права доступа на папку *ProgramData*, куда в этом случае устанавливается Anaconda). На последующих экранах, не меняя параметров, щелкайте *Next* и на последнем – *Install*. Убедитесь, что на последнем экране **поставлена** галочка *Register Anaconda as my default Python*, но **не поставлена** *Add Anaconda to my PATH environment variable*, это позволит вызывать Anaconda из консоли корректно. Anaconda потребует порядка 2,5 ГБ дискового пространства.



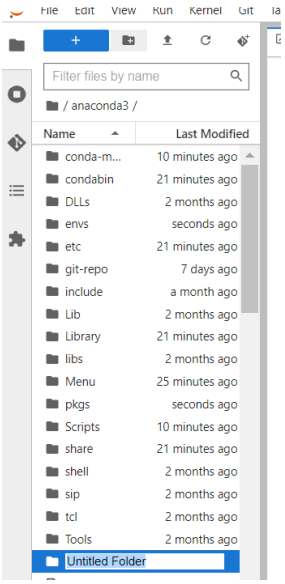
После установки на последнем экране уберите галочки в обоих чекбоксах и закройте инсталлятор. Теперь в меню пуск располагается папка *Anaconda3*. Для выполнения работ потребуется *Jupyter Lab* (расширенная версия *Jupyter Notebook*), которую необходимо установить. С этой целью открываем консоль команд Anaconda (Пуск – Все программы – папка «Anaconda3 (64-bit)» или подобным именем – Anaconda promt), в которой следует ввести и исполнить (Enter) команду:[1] *conda install -c conda-forge jupyterlab jupyterlab-git* (на запрос о согласии с установкой отвечаем латинской «y» с клавиатуры, после чего нажимаем Enter).



Имея все необходимое ПО, запустите из папки «Пуск – Все программы – Anaconda3 (64-bit)» программу Anaconda Navigator, дождитесь ее запуска и в открытом окне пролистайте вниз до JupyterLab, открыть который можно по кнопке *Launch*. Именно из данной панели проще всего открыть данную среду. Сам JupyterLab работает в браузере.

---

[1] в anaconda установка программ, дополнений, библиотек и прочего осуществляется через собственную консоль, как в описанном примере, либо через менеджер пакетов pip (как в консоли, так и внутри документов с вычислениями). Менеджер пакетов – программа для установки программ, хранящихся на сервере разработчика или сообщества (репозиторий). Google Play и AppStore – менеджеры пакетов, имеющие графический интерфейс и прочие функции, у pip же графического интерфейса нет, поэтому установка производится с помощью консольных команд. Синтаксис команд для pip выглядит так: «pip действие имя-пакета» (для консоли Anaconda заменить pip на conda), то есть можно установить/удалить пакет (библиотека python, программа на языке python из репозитория) введя вместо «действие» install/uninstall, также необходимо знать имя требуемого пакета.



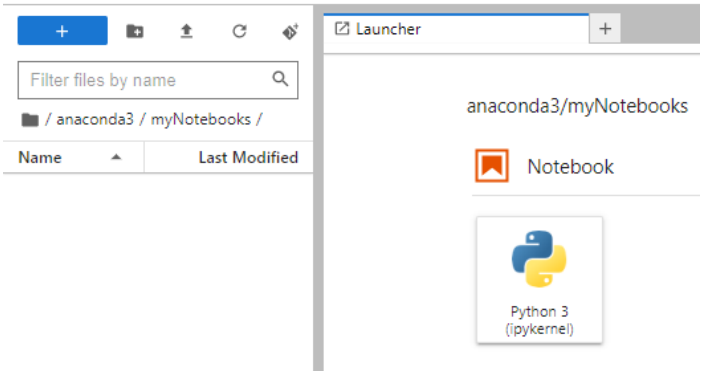
Запустив JupyterLab, перейдите на вкладку проводника (иконка папки) и выберете папку, в которую желаете сохранять программы и данные по вашим работам. Проводник по умолчанию отображает файлы из папки пользователя, где также содержится папка с настройками Anaconda (/anaconda3). Для примера создадим каталог для своих файлов в этой папке (для создания папки нажмите иконку папки с плюсом).

Основы python в JupyterLab/Notebook

JupyterLab, как уже было сказано, является расширенной версией Jupyter Notebook. Последний – IDE для работы с документами с исполняемым кодом (notebook-и), а JupyterLab имеет некоторые надстройки для удобства и возможности работы с не-notebook файлами (с python программами).

Главное отличие Jupyter Notebook от обычных IDE для python – работа в документах, разделенных на ячейки (Cells). Если в обычной IDE после написания кода в документе он исполняется целиком, то в Jupyter Notebook ячейки исполняются независимо, тем самым можно исполнять только те части кода, которые вам нужны в данный момент. Такая концепция построения документа с кодом названа REPL (read–eval–print loop – цикл чтения-вычисления-вывода), она удобна при математических вычислениях из-за чего используется, например, в системе компьютерной алгебры Wolfram Mathematica.

Откройте JupyterLab в папке, которую создали ранее (см. окончание подготовки к работе), на главной вкладке нажмите на кнопку нового файла Python в разделе Notebook.



Откроется новая вкладка с одной ячейкой In (для ввода). Попробуйте ввести в ней несколько простейших алгебраических операций. В python строки кода не требуют фигурных скобок для ограничения функций и «;» для ограничения строки кода, поэтому в единственную ячейку просто введите следующие строки:

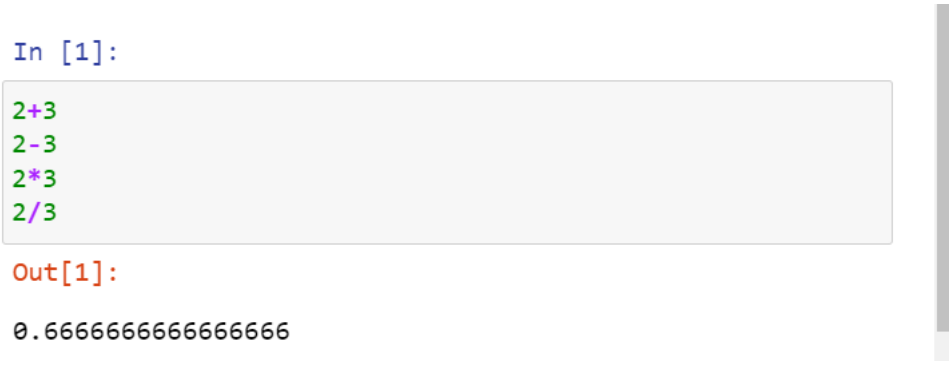
2+3

2-3

2*3

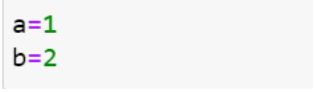
2/3

и исполните ячейку кнопкой Run сверху либо шорткатом (Shift+Enter или Ctrl+Enter).



Видим, что результат выводится только для 2/3, поскольку данная операция стоит последней, то есть *следующие друг за другом математические операции исполняются, но выводится только последняя из них*. Для вывода каждого из результатов необходимо воспользоваться стандартной функцией *print()*. Просто возьмите каждую заданную строку в скобки функции print() и повторите исполнение ячейки.

Переменным в python присваивается значение через «=». Зададим две переменных вместо прошлого кода в той же ячейке и исполним ее.



Видим, что результата (вывода) – нет, поскольку в ячейке только присваиваются значение, нет процедур с выводом. Если же добавить a/b в последнюю строку ячейки, то при исполнении результат появится.

```
In [17]:
a=1
b=2
a/b
Out[17]:
0.5
```

Массивы в python задаются в квадратных скобках, элементы перечисляются через запятую. Зададим простой массив из трех элементов строчного типа (записываются в кавычках – либо в одинарных, либо в двойных, **разницы нет**) и выведем их в цикле for. У этого цикла для нужной задачи будет следующая структура: «*for итератор in имя_массива:*», где итератор – и номер элемента в массиве, и ссылка на этот элемент. То есть python читает данную запись так «*для каждого элемента в массиве имя_массива исполни то, что записано после двоеточия*».

Двоеточие открывает внутреннее содержание (тело) какой-либо функции, то же самое, что в C++ и подобных языках делает «{». Если тело функции содержит лишь одну строку, то его можно записать в той же строке после двоеточия, иначе – с новой строки, *но обязательно с отступом от левого края (нажатием Tab на клавиатуре)*, все что на новых строках идет после двоеточия с отступом считается телом функции, после вызова которой стоит двоеточие.

```
In [3]:
words = ["я первое","а я второе","я последнее"]
for x in words:
    print(x)
я первое
а я второе
я последнее
```

В соответствии со сказанным выше, можно ту же процедуру реализовать и одной строкой.

```
In [6]:
words = ["я первое","а я второе","я последнее"]
for x in words: print(x)
я первое
а я второе
я последнее
```

Доступ к отдельным элементам заданного массива производится через «*имя_массива[индекс_в_массиве]*». Выведем первое (индекс массива начинается с нуля) значение заданного массива words в новой ячейке (для создания новой ячейки снизу нажать на панели сверху Insert – Insert Cell Below или нажать на ячейку вне поля ввода и нажать на клавиатуре латинскую B).

```
In [6]:
words = ["я первое","а я второе","я последнее"]
for x in words: print(x)
```

```
я первое
а я второе
я последнее
```

```
In [8]:
words[0]
Out[8]:
'я первое'
```

Функции в python задаются по конструкции «*def имя_функции(параметр1, параметр2, ...): тело функции*». Для примера ниже задана $f(x)=x^2+2$ и ее вызов для $x=0$ и $x=2$. Степень в python задается с помощью «**».

```
def f(x):
    return x**2+2
```

```
In [33]:
f(0)
Out[33]:
2

In [34]:
f(2)
Out[34]:
6
```

Перед дальнейшими действиями удалим ячейки с f(0) и f(2). Удалить ячейку можно из меню Edit-Delete Cells или нажать на ячейку вне поля ввода и двукратно нажать на клавиатуре D.

Функцию можно использовать для заполнения массивов. К примеру, зададим в массиве X значения от 1 до 10, а в массив Y запишем $f(X_i)$, то есть значение функции для каждого элемента X.

Для задания массива в виде диапазона значений в python используется *range(первый_элемент, последний_элемент, шаг)*. Если вызвать range без шага – то шаг будет равен 1, если вызвать с одним аргументом – сформируется массив из значений от 1 до заданного в виде аргумента значения с шагом 1.

```
X=range(10)
```

Массив Y инициализируем как *Y=[]*, после чего в цикле for заполним его значениями $f(X_i)$, прикрепляя на каждой итерации $f(X_i)$ к массиву Y, для чего используется функция «*имя_массива.append(элемент)*», добавляющая в конец массива выбранный элемент.

Заполним указанным образом Y и выведем его значения. Так как в одной ячейке и объявление Y=[], и наполнение данного массива с помощью .append(), данный массив (при исполнении ячейки) будет сперва обнуляться, а затем заполняться, тем самым не нужно беспокоиться, что многократное исполнение ячейки прицепит к концу Y слишком много элементов.

In [13]:

```
def f(x):
    return x**2+2
X=range(10)
Y=[]
for x in X:
    Y.append(f(x))
```

In [14]:

```
Y
```

Out[14]:

```
[2, 3, 6, 11, 18, 27, 38, 51, 66, 83]
```

Как было сказано, в тело цикла можно поместить более одной операции. Зададим новый пустой массив *Y1* вне цикла и наполним его в том же цикле значениями f(x), деленными надвое.

```
Y1=[]
for x in X:
    Y.append(f(x))
    Y1.append(f(x)/2)
```

Для вывода обоих массивов воспользуемся упомянутой ранее print() для каждого массива.

```
[2, 3, 6, 11, 18, 27, 38, 51, 66, 83]
[1.0, 1.5, 3.0, 5.5, 9.0, 13.5, 19.0, 25.5, 33.0, 41.5]
```

Часто при программировании требуется использовать условие. В python оно задается по конструкции «*if условие: операция*». Аналогично for, у if оператор может располагаться в той же строке либо в строчке под if с отступом слева (TAB) относительно if. Jupyter Notebook расставит отступы за вас сразу после переноса строки после двоеточия.

Зададим в цикле условие для наполнения значениями больше 7 в массив Y, меньше или равные 7 – в массив Y1. Ниже слева скриншот задания условия, справа – результат вывода Y и Y1.

```
for x in X:
    if f(x)>7:
        Y.append(f(x))
    if f(x)<=7:
        Y1.append(f(x))
```

```
[11, 18, 27, 38, 51, 66, 83]
[2, 3, 6]
```

То же действие можно совершить с помощью только одного условного оператора if, для чего оператор, который требуется исполнить при невыполнении заданного условия, помещается после конструкции «*else:*». При этом результат (Y и Y1) не будут отличаться от рассмотренных выше.

```
for x in X:
    if f(x)>7:
        Y.append(f(x))
    else:
        Y1.append(f(x))
```

До конца 2021 года в Python не было аналога функции *switch* для условий с несколькими вариантами, поэтому в сообществе принято использовать конструкцию «*if-elif-else*». В этом случае внутри одного условного оператора if после каждого elif (сокращение от else if) можно задавать новое условие,

конструкция аналогична последовательному исполнению множества операторов if.

Рассмотрим данную конструкцию на примере заполнения Y и Y1. Предположим, что требуется записать в Y все значения f(x) меньше 50, а также значение f(x), если оно равно 27, остальные же значения нужно записать в Y1. Для этого после if зададим условие «меньше 50» для заполнения Y, под ним после elif «равно 27» для той же операции, и после else (для всех оставшихся значений) будем заполнять Y1. Сравнение (равенство) в условии на языке python задается «= =»

```
for x in X:
    if f(x)<50:
        Y.append(f(x))
    elif f(x)==27:
        Y.append(f(x))
    else:
        Y1.append(f(x))
```

```
[2, 3, 6, 11, 18, 27, 38]
[51, 66, 83]
```

Ту же процедуру можно выполнить без elif, если в if записать оба условия, для чего их необходимо взять в скобку и поставить между ними: *or* – если необходимо, чтобы выполнилось хотя бы одно условие (ИЛИ); *and* – чтобы выполнялись оба условия (И). Y1 и Y после данных изменений будут иметь те же значения, так как формально условие не поменялось.

```
for x in X:
    if (f(x)<50 or f(x)==27):
        Y.append(f(x))
    else:
        Y1.append(f(x))
```

Вернем исходное состояние цикла for для наполнения значениями f(x) массив Y, при этом объявление Y1 можно удалить. *На данном этапе замените f(x) на ту, что задается вашим вариантом в таблице ниже.*

Варианты функций для f(x)

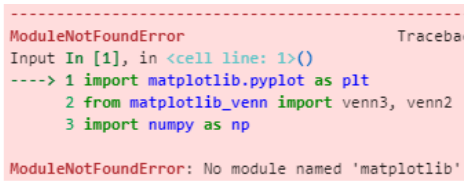
1	**2**	**3**	**4**	**5**	**6**
x^3-2	x^2-7	x^3+2x	x^2+2x	$x^3-0,5x^2$	$0,1x^3+0,4x$
7	**8**	**9**	**10**	**11**	**12**

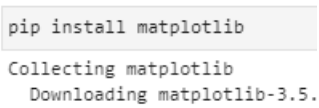
$0,1x-0,7x^3$	$-x^2+7$	$-x^3-0,4x$	$x^2-0,2x$	$x^3+0,5x^2$	$0,1x^3-0,4x$
13	**14**	**15**	**16**	**17**	**18**
$-x^3+2x^2$	$0,1x^2-7$	$0,6x^3+2,2x$	$-x^2+0,1x^3$	$0,3x^2-0,2x$	$0,7x+0,4x^2$
19	**20**	**21**	**22**	**23**	**24**
x^3-2	x^2-7	x^3+2x	x^2+2x	$x^3-0,5x^2$	$0,1x^3+0,4x$
25	**26**	**27**	**28**	**29**	**30**
$0,1x-0,7x^3$	$-x^2+7$	$-x^3-0,4x$	$x^2-0,2x$	$x^3+0,5x^2$	$0,1x^3-0,4x$
31	**32**	**33**	**34**	**35**	**36**
$-x^3+2x^2$	$0,1x^2-7$	$0,6x^3+2,2x$	$-x^2+0,1x^3$	$0,3x^2-0,2x$	$0,7x+0,4x^2$

Теперь построим график Y(X), для чего потребуется подключить библиотеку matplotlib.pyplot (точнее библиотека matplotlib, но из нее мы берем только класс pyplot, отвечающий за графики). Для этого создадим ячейку в начале документа (импорт библиотек принято производить в начале кода), в которой импортируем нужную библиотеку. Импорт в python осуществляется по конструкции «*import имя_библиотеки as краткое_название*», где краткого названия может (и as перед ним) и не быть, но с ним куда удобнее обращаться к классам и функциям библиотеки с длинным названием, как в данном случае.

```
In [16]:
import matplotlib.pyplot as plt
```

Если при импорте библиотеки возникает ошибка *ModuleNotFound*, значит импортируемая библиотека не установлена, следует исправить ситуацию исполнением команды «*pip install имя_библиотеки*» в отдельной ячейке, после чего повторить исполнение ячейки с импортом библиотеки.



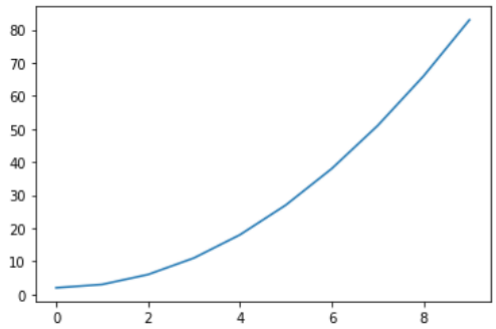


Не забываем исполнить ячейку с библиотекой для осуществления импорта и задаем параметры для построения графика Y(X). При кратком

названии библиотеки plt для построения графика используется конструкция «*plt.plot(массив_для_x, массив_для_y)*».

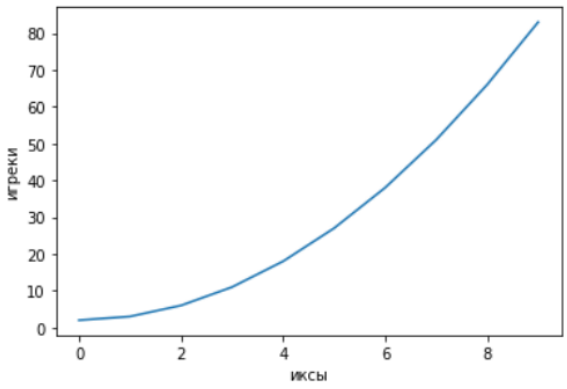
```
In [17]:
import matplotlib.pyplot as plt
plt.plot(X,Y)
```

```
Out[17]:

[<matplotlib.lines.Line2D at 0x25377babc40
>]
```



Чтобы задать обозначения осей нужно воспользоваться функциями *.ylabel()* и *.xlabel()* библиотеки, в скобках которых соответственно вписываем название осей в апострофах. Чтобы отображался только график, без текстовой информации о данных над графиком, также следует добавить в конец ячейки *plt.show()* (в консольной версии python эта команда выводит график, в Jupyter же график, как запрошенные к выводу данные, выводится и без данной команды).

```
In [30]:
import matplotlib.pyplot as plt
plt.plot(X,Y)
plt.xlabel('иксы')
plt.ylabel('игреки')
plt.show()
```



Оформление документа Jupyter и подготовка отчета

Тип ячеек в документе можно менять с исполняемого кода (*Code*) на простой текст (*Markdown*). Это дает возможность оформлять отчет по работе из документа Jupyter без переноса материалов в текстовый редактор, а также удобно для добавления комментариев, деления документа на озаглавленные блоки и т.п.

В документе по вводной работе создайте вверху ячейку, над импортом библиотек. Для смены типа ячейки нужно либо выбрать тип в выпадающем списке меню (между иконками перемотки и клавиатуры), либо щелкнуть по ячейке вне поля ввода и нажать на клавиатуре *M* (для смены на текст) или *Y* (для смены на код). Смените тип ячейки на текст. Неформатированный формат ячейки (Raw NBConvert) использоваться не будет, а заголовочный тип (Heading) заменен на текстовый с указанием уровня заголовка. В Jupyter доступно 6 уровней заголовков, все они отличаются размером (с увеличением уровня уменьшается размер), а последние два уровня имеют начертание курсивом.

Для задания заголовка в текстовой ячейке используется символ «#» в начале строки, причем количество идущих подряд таких символов задает уровень заголовка, а между текстом заголовка и последним таким символом должен стоять пробел. Задайте первой строкой название работы (заголовок уровня 1), второй – слово Вариант и номер варианта (уровень 2), третьей – ваше ФИО, курс-факультет-группу (простым текстом), четвертой – год выполнения. Обратите внимание, что перенос (Enter) между двумя строками текста не отображается после исполнения текстовой ячейки (две строки слипаются в одну, но после заголовка любого уровня перенос ставится автоматически), для отображения переноса требуется при редактировании текста делать два переноса между строками или добавить в конце строки HTML-тег переноса <*br*>. Чтобы отобразить текст после редактирования, ячейку с ним требуется исполнить. Для редактирования текстовой ячейки дважды щелкните по ней.

Чтобы сохранить оформленный отчет как PDF, нужно либо воспользоваться конвертером (File - Download as – PDF via LaTeX или PDF via HTML), который загружается отдельно, либо открыть режим печати (на клавиатуре Ctrl+P), в котором возможно сохранить файл как PDF или XPS (требуется выбрать такую опцию в поле принтеров).

Работа 1. Основные операции над множествами

Плановое время на выполнение: 2 занятия (4 ак. часа).

Примечание: перед началом работы создайте шапку отчета, как указано во вводной работе, в разделе оформления.

Одной из задач программ, использующих методы искусственного интеллекта, является сравнение каких-либо объектов (изображений, аудиозаписей и т.п.) с целью определить их схожести. Данная задача, в примитивном представлении, также может быть решена без средств интеллектуализации, лишь с помощью операций над множествами.

Множеством в python называют контейнер (или массив элементов), содержащий неповторяющиеся элементы в случайном порядке. То есть, данный тип данных позволяет легко определить неповторяющиеся элементы в строках, массивах (чисел, строк) и т.д. Создать множество из данных или строки можно с помощью команды *set(имя_массива_или_строки)*. Продемонстрируем работу set на элементарном примере. Можно видеть, что из массива data полученное множество сохранило только неповторяющиеся в нем числа, а из строки ‘data’ – только неповторяющиеся в ней символы.

```
data = [1,2,0,1,2,3,2,0,3]
set1 = set('data')
set2 = set(data)
print(set1)
print(set2)
```

```
{'t', 'd', 'a'}
{0, 1, 2, 3}
```

Множество можно задать непосредственно, для чего в переменную записывается массив в фигурных скобках вместо прямоугольных (при этом повторяющиеся элементы в множество, очевидно, записаны не будут). Для сортировки множества можно использовать метод *sorted(имя_множества)*. Удобно применять множество для удаления повторяющихся элементов, что продемонстрировано на примере массива случайных элементов rand, заданном через

```
plenty_a = {1,2,0,1,3,2}
print('Множество a = ', plenty_a)

plenty_data = set('data')
print('Множество data = ', sorted(plenty_data))

rand = np.random.randint(0, 5, 10)
print('\nМассив рандомных чисел', rand)
plenty_rand = set(rand)
print('Множесство из Массива rand', plenty_rand)
```

```
Множество a =  {0, 1, 2, 3}
Множество data =  ['a', 'd', 't']

Массив рандомных чисел [2 3 2 0 3 2 2 3 3 4]
Множесство из Массива rand {0, 2, 3, 4}
```

np.random.randint(мин, макс, число_элементов_массива), причем метод дает только целые случайные числа, равные или более *мин*, но менее *макс* (т.е. самому *макс* не могут быть равны). Не забудьте импортировать библиотеку numpy как np, чтобы метод *randint* заработал: `import numpy as np` (импортировать библиотеки принято в начале кода, поэтому создайте для данной строки ячейку в начале документа, под шапкой оформления).

Примечание: в примерах выше и далее результаты выводятся с помощью функции print(), причем для нее используется несколько выводимых значений (в скобках), отделенных запятой. Это позволяет выводить не только значения, но и текстовое их описание (в виде строк текста). Для переноса текста в строках применяется префикс «\n».

Множества также удобно использовать для определения, содержится ли в строке/массиве какой-либо элемент. Для этого полезной окажется конструкция *элемент in массив_или_строка*, которая выдает True, если элемент содержится в массиве, и False – если не содержится. И наоборот, если требуется узнать, отсутствует ли элемент в массиве/строке, используют конструкцию *элемент not in массив_или_строка*.

```
print('Есть ли 2 в plenty_a',2 in plenty_a)
print('Есть ли 5 в plenty_a',5 in plenty_a)

print('\nОтсутствует 2 в plenty_a',2 not in plenty_a)
print('Отсутствует ли 5 в plenty_a',5 not in plenty_a)
```

```
Есть ли 2 в plenty_a True
Есть ли 5 в plenty_a False

Отсутствует 2 в plenty_a False
Отсутствует ли 5 в plenty_a True
```

Для определения числа элементов в множестве используется тот же метод *len()*, что используется для обычного списка.

```
print('Длина множества ',plenty_a,' - ',len(plenty_a))
print('Длина множества data',plenty_data,' - ',len(plenty_data))
```

```
Длина множества  {0, 1, 2, 3}  -  4
Длина множества data {'d', 'a', 't'}  -  3
```

Для добавления элемента в множество используется метод *имя_множества.add(элемент_для_добавления)*, так, в данном примере к созданному ранее множеству plenty_data добавляется 100: `plenty_data.add(100)`. Для удаления используется *имя_множества.discard(элемент_для_удаления)*: `plenty_data.discard('d')` либо *имя_множества.remove(элемент_для_удаления)*: `plenty_data.remove('t')`, причем discard удалит элемент, если он имеется в множестве (и не сделает ничего, если элемента нет), remove же при удалении

несуществующего элемента выдаст ошибку. Очистить множество можно командой *имя_множества.clear()*.

Для объединения элементов множества можно либо создать новое множество, куда записать результат объединения с помощью команды *новое_множ = множ1.union(множ2)*, либо просто дописать в одно множество элементы другого командой *множ1.update(множ2)*.

```
# с созданием 3-го множества
a = {"а","в", "е"}
b = {"б", "г", "з"}
a_b = a.union(b)
print("Новое множество a_b - ",a_b)
# без создания 3-го множества
a.update(b)
print("Дополненное множество a - ",a)
```

```
Новое множество a_b -  {'а', 'в', 'г', 'е', 'б', 'з'}
Дополненное множество a -  {'а', 'в', 'г', 'е', 'б', 'з'}
```

Чтобы определить схожесть и различия множеств используют, методы определения, соответственно, пересечения и разности. Пересечение двух множеств задается по конструкции *новое_множ = множ1.intersection(множ2)* и результатом ее исполнения будет новое множество, содержащее только пересекающиеся (одинаковые) элементы из обоих множеств. Разность определяется аналогично: *новое_множ = множ1.difference(множ2)*, а результат содержит только отличающиеся элементы из обоих множеств. При этом, разность не существует, если множества содержат только одинаковые элементы, а пересечение – если одинаковых элементов в множествах нет. В этом случае результатом будет пустое множество (при выводе – set()), что демонстрируется на примере трех множеств из чисел на скриншоте.

```
a = np.random.randint(0, 5, 10)
b = np.random.randint(0, 5, 10)
c = [5, 6, 7]
print('a = ', a, '\nb = ', b, '\nc = ', c)
inter_ab = set(a).intersection(set(b))
inter_ac = set(a).intersection(set(c))
diff_ab = set(a).difference(set(b))
print('Пересечение множеств а и б - ', inter_ab)
print('Пересечение множеств а и ц - ', inter_ac)
print('Разность множеств а и б - ', diff_ab)
```

```
a =  [3 0 3 0 3 4 3 4 2 2]
b =  [4 2 2 4 2 4 3 3 3 4]
c =  [5, 6, 7]
Пересечение множеств а и б -  {2, 3, 4}
Пересечение множеств а и ц -  set()
Разность множеств а и б -  {0}
```

Если все элементы одного множества входят в другое, то последнее множество называется надмножеством, а первое – подмножеством для второго. Для определения, является ли *множ1*

```
a = [0,1,2,3,4]
b = [0,1,2,3,4,5]
print('a = ', set(a), '\nb = ', set(b))
print("а это надмножество б?", set(a).issuperset(set(b)))
print("б это надмножество а?", set(b).issuperset(set(a)))
print("а это подмножество б?", set(a).issubset(set(b)))
print("б это подмножество а?", set(b).issubset(set(a)))
```

```
a =  {0, 1, 2, 3, 4}
b =  {0, 1, 2, 3, 4, 5}
а это надмножество б? False
б это надмножество а? True
а это подмножество б? True
б это подмножество а? False
```

надмножеством *множ2* используется конструкция *множ1.issuperset(множ2)*, а для определения, является ли *множ1* подмножеством *множ2* – *множ1.issubset(множ2)*.

Рассмотрим пример визуализации пересечения и разности двух множеств. Для этого зададим три множества и добавим строки для их вывода.

```
a = {1,2,3,4,5,6}
b = {5,6,7,8,9,10}
c = {10,12,15,16,18,20}
print("Set1 - ", a)
print("Set2 - ", b)
print("Set3 - ", c)
```

Для визуализации нам потребуется библиотека *matplotlib_venn* с диаграммами Венна, которые отображают пересечения множеств в формате наложенных кругов разных цветов. Из нее потребуется лишь два модуля, посвященные диаграммам по пересечению трех множеств *venn3* и двух множеств *venn2*, поэтому импортируем не всю библиотеку, а только указанный модуль, что в python осуществляется с помощью конструкции *from библиотека import модуль1, модуль 2*. Добавьте соответствующую строку в ту же ячейку, где осуществляется импорт numpy, поскольку при программировании принято указывать все импортируемые элементы в одном месте.

```
from matplotlib_venn import venn3, venn2
import numpy as np
```

Зададим три множества одного размера (по 6 элементов в каждом), два из которых – пересекающиеся.

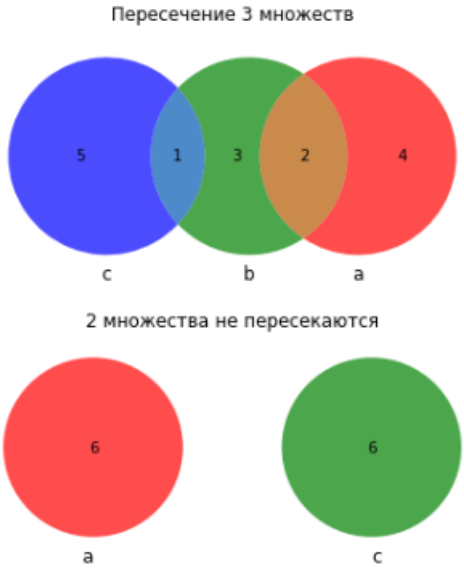
```
a = {1,2,3,4,5,6}
b = {5,6,7,8,9,10}
c = {10,12,15,16,18,20}
```

Затем добавим в ту же ячейку три строки для вывода диаграммы. У venn3 здесь используется 3 параметра: первый – массив (в квадратных скобках) сравниваемых множеств, второй – обозначения множеств на диаграмме (в круглых скобках через запятую в строковом формате), третий (необязательный) – параметр прозрачности кругов, по умолчанию 40%, здесь задается 70%. Вторая строка добавляет заголовок графику, а последняя должна быть вам известна из «основ python» в начале пособия. Полученный результат представлен ниже, здесь в отдельных частях кругов указывается число элементов, которые пересекаются (в пересекающихся частях) или не пересекаются (в «свободных» частях кругов).

```
venn3([a, b, c], ('a', 'b', 'c'), alpha=0.7)
plt.title("Пересечение 3 множеств")
plt.show()
```

Для отображения двух множеств используем модуль venn2 на примере непересекающихся множеств c и a, для чего добавим соответствующие строки в ячейку. При этом получим окончательный результат в виде двух диаграмм.

```
plt.title("2 множества не пересекаются")
venn2([a, c], ('a', 'c'), alpha=0.7)
plt.show()
```



Аналогично множествам, можно визуализировать пересечения массивов (совпадение их элементов). В качестве примера зададим два массива из 100 нолей каждый с помощью *a = np.zeros(100)* и *b = np.zeros(100)*. Теперь заменим на 1 в массиве a с 20 по 80 элементы с помощью *a[20:80] = 1.0*, а в массиве b – с 60 по 100-й элемент с помощью *b[60:100] = 1.0*. Также зададим массив *x*, содержащий значения для оси времени.

```
a = np.zeros(100)
b = np.zeros(100)
a[20:80] = 1.0
b[60:100] = 1.0
x = range(len(a))
```

Далее потребуется библиотека для построения плоских графиков, которую необходимо импортировать.

```
import matplotlib.pyplot as plt
from matplotlib_venn import venn3, venn2
import numpy as np
```

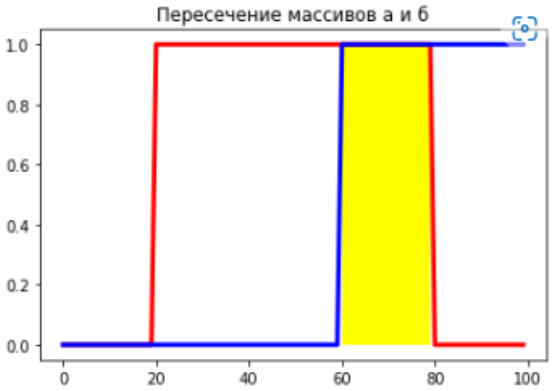
В той же ячейке настроим график для пересечения массивов. Первая строчка дает возможность добавлять в фигуру *fig* несколько графиков, содержащихся в *ax*, для чего используется метод *subplots*. Далее в ax записывается общий заголовок графиков, после чего – график каждого из массивов с параметрами цвета (color) и толщины линии (linewidth). Последняя строка отвечает за заливку пересекающихся областей (метод *.fill_between* – заливка между кривыми), где первые два параметра – кривые, между

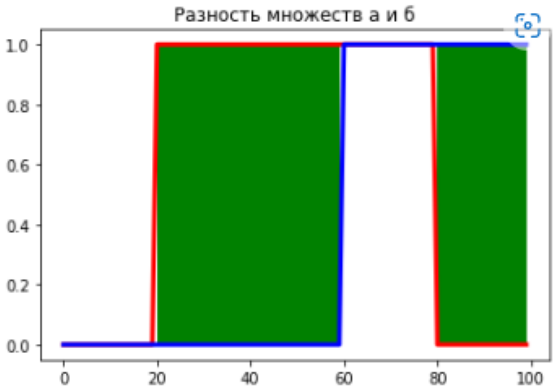
которыми осуществляется заливка (ось x и массив a), условие, по которому осуществляется заливка (*where=* требуется для обозначения условия, b == a говорит о том, что заливка между x и a будет происходить только в том случае, если a=b), а последний параметр (*facecolor*) задает цвет заливки.

```
fig, ax = plt.subplots()
ax.set_title("Пересечение массивов a и б")
ax.plot(a, color = 'r', linewidth = 3)
ax.plot(b, color = 'b', linewidth = 3)
ax.fill_between(x, b, where= (b == a), facecolor='yellow')
```

Подобным образом в той же ячейке описываем параметры графика разности массивов, сменив условия у ax.fill_between и добавив в конце всей ячейки plt.show() для вывода графика без служебной информации. Конечный результат приведен ниже.

```
fig, ax = plt.subplots()
ax.set_title("Разность множеств a и б")
ax.plot(a, color = 'r', linewidth = 3)
ax.plot(b, color = 'b', linewidth = 3)
ax.fill_between(x, a, b, where= (b != a), facecolor='green')
plt.show()
```





Самостоятельное задание

1) В таблице 1 заданы варианты строк – *str1* и *str2*, требуется:

1.1) преобразовать каждую из них в множество с названиями set1 и set2 соответственно, вывести с произвольным текстовым описанием значения

исходных массивов и полученных множеств (например: Строка str1 - …, Множество set1 - …);

1.2) вывести длину каждого множества с произвольным текстовым описанием;

1.3) проверить, имеется ли в каждом из множеств элемент *el1* из таблицы 1, вывести с произвольным текстовым описанием результаты сравнения;

1.4) проверить, является ли set1 подмножеством/надмножеством set2 и наоборот (set2 относительно set1), вывести с произвольным текстовым описанием результаты;

1.5) вывести с произвольным текстовым описанием результат вычисления пересечения и разности set1 с set2;

1.6) добавить к множеству set1 элемент *el2*, а к set2 – *el3* и *el4*, объединить в *set3* множества set1 и set2, вывести с произвольным текстовым описанием результат;

1.7) задайте строку *str3*, в которую (транслитерацией на латиницу) запишите собственную фамилию, преобразуйте ее в множество *set_surname*;

1.8) постройте диаграмму Венна (venn3) для всех трех множеств;

1.9) постройте диаграмму Венна (venn2) для множеств set1 и set_surname;

1.10) постройте диаграмму Венна (venn2) для множеств set2 и set_surname;

2.1) создайте 2 множества из массивов случайных чисел вида np.random.randint(0, 10*N, 10), где N – номер вашего варианта;

2.2) выполните для созданных множеств действия из 1.2, 1.4, 1.5;

3) создайте два массива нулей a и b, заполните их единицами в интервалах: a – от N+10 до 2*(N+10)-10, b – от N+20 до 2*(N+10), где N – вариант, постройте графики пересечения и разности данных массивов.

Таблица 1 – Варианты для первого задания

Вар.	str1	str2	el1	el2	el3	el4
1	‘ernest’	‘hemingway’	w	t	n	c
2	‘erich’	‘remarque’	q	s	d	n
3	‘jean’	‘baudrilliard’	u	v	t	d
4	‘soren’	‘kierkegaard’	d	r	s	t
5	‘samos’	‘pythagoras’	t	c	v	s
6	‘rene’	‘descartes’	s	n	r	v
7	‘pafnuty’	‘chebyshev’	v	d	c	r
8	‘erich’	‘hemingway’	w	t	n	c
9	‘jean’	‘remarque’	w	t	n	c
10	‘soren’	‘baudrilliard’	q	s	d	n
11	‘samos’	‘kierkegaard’	u	v	t	d
12	‘rene’	‘pythagoras’	d	r	s	t
13	‘pafnuty’	‘descartes’	t	c	v	s
14	‘ernest’	‘chebyshev’	s	n	r	v
15	‘jean’	‘hemingway’	v	d	c	r
16	‘soren’	‘remarque’	w	t	n	c
17	‘samos’	‘baudrilliard’	w	t	n	c
18	‘rene’	‘kierkegaard’	q	s	d	n
19	‘pafnuty’	‘pythagoras’	u	v	t	d
20	‘ernest’	‘descartes’	d	r	s	t
21	‘erich’	‘chebyshev’	t	c	v	s
22	‘soren’	‘hemingway’	s	n	r	v
23	‘samos’	‘remarque’	v	d	c	r
24	‘rene’	‘baudrilliard’	w	t	n	c
25	‘pafnuty’	‘kierkegaard’	w	t	n	c
26	‘ernest’	‘pythagoras’	q	s	d	n
27	‘erich’	‘descartes’	u	v	t	d
28	‘jean’	‘chebyshev’	d	r	s	t
29	‘samos’	‘hemingway’	t	c	v	s
30	‘rene’	‘remarque’	s	n	r	v
31	‘pafnuty’	‘baudrilliard’	v	d	c	r
32	‘ernest’	‘kierkegaard’	w	t	n	c
33	‘erich’	‘pythagoras’	q	s	d	n
34	‘jean’	‘descartes’	u	v	t	d
35	‘soren’	‘chebyshev’	d	r	s	t

Работа 2. Сравнение строк и основы нечеткой логики

Плановое время на выполнение: 2 занятия (4 ак. часа).

Зададим для начала две строки – в *str_A* запишем ‘Привет Мир’, в *str_B* – ‘привет мир’, то есть строки отличаются лишь тем, что у первой слова с заглавных букв, у второй – нет. Попробуем сравнить их в той же ячейке простым оператором сравнения (==), который даст True, если строки равны. Как видим, результат False, но с человеческой точки зрения строки почти идентичны. Более разумным будет сравнить множества *set_A* и *set_B*, соответственно полученные путем преобразования в множество каждой из строк. Выведем эти множества и результат их сравнения тем же образом (==). И вновь, визуально множества почти идентичны, но сравнение дает только True/False, нет результат «почти True», подходящего здесь.

```
print('Равны ли строки:', str_A == str_B)
```

```
Равны ли строки: False
```

```
print('Равны ли строки:', str_A == str_B)
print('Множества:\n', set_A, '\n', set_B)
print('Равны ли множества:', set_A == set_B)
```

```
Равны ли строки: False
Множества:
 {'р', 'в', 'т', ' ', 'и', 'П', 'е', 'М'}
 {'р', 'в', 'т', ' ', 'и', 'м', 'е', 'п'}
Равны ли множества: False
```

Но, как вы знаете, помимо прямого сравнения, для множеств можно оценить их пересечение, что даст новое множество из общих элементов. Применим соответствующий метод из прошлой работы, результат запишем в переменную *inter_AB*, и для оценки числового выражения «степени схожести» строк, поделим длину данного множества на длину одного из изначальных множеств. Получим 0.75 или 75%.

```
print('Пересечение множеств:', len(inter_AB)/len(set_A))
```

```
Равны ли строки: False
Множества:
 {'р', 'в', 'т', ' ', 'и', 'П', 'е', 'М'}
 {'р', 'в', 'т', ' ', 'и', 'м', 'е', 'п'}
Равны ли множества: False
Пересечение множеств: 0.75
```

Но этот результат не отражает истины, поскольку сравниваются множества, полученные из строк, а не сами строки, тем самым учитываются только неповторяющиеся буквы (подсчитав вручную процент идентичных символов в строках, получим 80%), плюс при разной длине строк (и множеств) возникает дополнительный вопрос к тому, как же вычислять схожесть в %.

Помимо задания строк непосредственно в коде, в python можно запросить текст для нее у пользователя, для чего в коде следует указать *имя_переменной = input()*. Для примера, попробуем сравнить с str_A текст, введенный пользователем (после исполнения ячейки с input() появится поле для ввода, на которое нужно кликнуть и ввести значение). Как видим в примере, строки в разном регистре (строчные/прописные) также не считаются равными.

```
str_input = input()
print("Исходная строка: " + str_A)
if str_A == str_input:
    print("Строки одинаковые")
else :
    print("Строки разные")
```

```
 ПРИВЕТ МИР
Исходная строка: Привет Мир
Строки разные
```

Примечание: в примере выше print() для текста «Исходная строка…» выводится как сумма двух строк, то есть в python прибавление одной строки к другой фактически «слепляет» строки в одну в указанном порядке.

Если для сравнения не важен регистр символов, можно преобразовать исходные строки к одному регистру: с помощью *имя_строки.lower()* – к нижнему регистру, с помощью *имя_строки.upper()* – к верхнему. Добавим в пример приведение обеих строк к нижнему регистру перед сравнением и увидим, как поменялся результат.

```
str_input = input()
print("Исходная строка: " + str_A)
if str_A.lower() == str_input.lower():
    print("Строки одинаковые")
else :
    print("Строки разные")
```

```
 ПРИВЕТ МИР
Исходная строка: Привет Мир
Строки одинаковые
```

Применение нечеткой логики позволяет организовать сравнение строк куда проще, причем результат выводится сразу в процентном соотношении. Для начала импортируйте *thefuzz.fuzz* – модуль одной из библиотек, реализующих методы нечеткой логики. Перед импортом, библиотеку thefuzz следует установить по известной вам команде *pip install имя_библиотеки*, исполненной в отдельной ячейке.

Соотношение равенства строк вызывается методом *fuzz.ratio(строка1, строка2)*. Вызвав его для приведенных выше строк, получим ранее рассчитанные 80%.

Другим методом нечеткой логики для сравнения строк является алгоритм Левенштейна, который позволяет найти так называемое редакционное расстояние между строк – то есть говорит о том, сколько

действий по редактированию нужно совершить над одной из сравниваемых строк, чтобы обе они стали идентичны друг другу. Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так: D (англ. delete) — удалить, I (англ. insert) — вставить, R (replace) — заменить, M (match) — совпадение.

Для реализации этого алгоритма построим собственную функцию, используя введенные выше обозначения *D, I, R*. Обозначим такую функцию следующим образом: `def leven(a, b):` , где *def* указывает на то, что далее идет определение функции, *leven* – название функции, *a* и *b* – сравниваемые строки.

В теле функции сперва запишем в переменные значения длин строк. Обратите внимание, в python можно только одним символом присвоения (=) записать значения сразу в несколько переменных, здесь в *n* записывается длина a, в *m* – длина b: `n, m = len(a), len(b)`. Чтобы избежать проблем с разной длиной строк, в первую очередь определяем, длиннее ли b, чем a и если да, то меняем их местами (перезаписываем в a строку b и наоборот),

```
if n > m:
    a, b = b, a
    n, m = m, n
```

а также значения их размеров, так как иначе сравнение строк закончится с концом строки b. Далее, используя массивы текущих и предыдущих строк (*cur* и *prev*) реализуем непосредственно алгоритм Левенштейна, не забывая выдать результат функции через *return()*.

```
cur = range(n + 1)
for i in range(1, m + 1):
    prev, cur = cur, [i] + [0] * n
    for j in range(1, n + 1):
        I, D, R = prev[j] + 1, cur[j - 1] + 1, prev[j - 1]
        if a[j - 1] != b[i - 1]:
            R += 1
        cur[j] = min(I, D, R)
return cur[n]
```

Опробовав полученную функцию на заданных ранее строках, получаем значение «2», то есть для получения из str_A строки str_B нужно в первой заменить 2 символа.

```
leven(str_A, str_B)
```

```
2
```

Примечание: в примере выше значение функции leven() выводится без print() поскольку вызов этой функции является последней строкой в ячейке, и в jupyter результат именно последней строки отображается на выводе, если для других результатов вывод не задается с помощью print().

Осталось добавить в заданную функцию возможность отображения процента схожести строк – своего рода аналог fuzzy.ratio. Его рассчитывают следующим образом: `perc = (2*(max(m,n) - cur[n])/(m+n))*100`. Таким образом, сменив прежнюю выходную переменную в return на *perc* получим значение в процентах.

```
    return perc
leven(str_A, str_B)
```

```
80.0
```

Если строки разного размера, то можно определить процент схожести меньшей строки относительно большей. При этом меньшая строка будет сравниваться с фрагментами большей, и если меньшая полностью найдена в большей – результат будет равен 100% и т.п. Реализуется данная процедура методом *fuzz.partial_ratio(строка1, строка2)*. На скриншоте представлен результат вызова fuzz.ratio и fuzz.partial_ratio для str_A (использованная ранее) и *str_C* = ‘Привет’.

```
Обычное сравнение 75 %
Частичное сходство строк 100 %
```

Помимо этого, в используемой библиотеке существует также метод *fuzz.token_sort_ratio(строка1, строка2)*, позволяющий определить схожесть строк в разной последовательности слов (то есть перед сравнением слова в строке сортируются в алфавитном порядке, а также игнорируется регистр символов). Для примера на скриншоте представлен результат сравнения строк *str_D* = ‘Привет наш мир’ и *str_E* = ‘Наш мир привет’ с помощью всех трех озвученных методов из fuzz.

```
Обычное сравнение 36 %
Частичное сходство строк 50 %
Сравнение в разном порядке 100 %
```

Для ручного разбиения строк на массив из составляющих слов (если слова отделены пробелами), можно использовать метод *строка.split()*. Отсортировать по порядку (не важно, числа или символы/слова/строки) элементы массива можно с помощью *sorted(массив)*. Чтобы получить из массива строк одну строку, можно воспользоваться методом *‘заполнитель’.join(массив_строк)*, где заполнитель – символ, подставляемый между словами из массива. Таким образом, если

```
str_D_low = str_D.lower()
str_D_split = str_D_low.split()
str_D_sort = sorted(str_D_split)
str_D_result = ' '.join(str_D_sort)

print(str_D)
print(str_D_low)
print(str_D_split)
print(str_D_sort)
print(str_D_result)
```

```
Привет мир наш
привет мир наш
['привет', 'мир', 'наш']
['мир', 'наш', 'привет']
мир наш привет
```

совместить все описанные методы, можно получить новую строку, которая даст наиболее точный результат при сравнении с другой, если порядок слов и регистр символов не важен. Пример приведения к такому, более оптимальному для сравнения виду строки str_D приведен на скриншоте.

Для поиска текста в массиве строк можно использовать модуль *process* из библиотеки thefuzz. Метод *process.extract(искомая_строка, массив_строк)* выдает процент схожести (вычисляемый, по умолчанию, по формуле *WRatio*, определяющей схожесть различными способами) искомой строки с каждым элементом из массива строк. Если же вызвать *process.extractOne(искомая_строка, массив_строк)*, то будет выведена только та строка, которая имеет наибольшее значение схожести. На примере ниже видно, что в двух строках с одинаковым процентом схожести найдена искомая строка (sir), и extractOne выводит первую из списка, выводимого excract.

```
str1 = "Sir, this young fellow's mother could: whereupon"
str2 = 'she grew round-wombed, and had, indeed, sir, a son'
str3 = 'for her cradle ere she had a husband for her bed.'
str4 = 'Do you smell a fault?'
str_arr = [str1, str2, str3, str4]
print("Выбранная строка: ", process.extractOne('sir', str_arr))
process.extract('sir', str_arr)
```

```
Выбранная строка:  ("Sir, this young fellow's mother could: whereupon", 60)
[("Sir, this young fellow's mother could: whereupon", 60),
 ('she grew round-wombed, and had, indeed, sir, a son', 60),
 ('Do you smell a fault?', 30),
 ('for her cradle ere she had a husband for her bed.', 20)]
```

Для этих методов можно сменить способ оценки схожести, для чего требуется указать параметр *scorer* в формате: *process.extract(искомая_строка, массив_строк, scorer = способ_оценки)*. В качестве scorer можно указать один из выше изученных из библиотеки thefuzz: ratio, partial_ratio, token_sort_ratio.

```
str1 = "Sir, this young fellow's mother could: whereupon"
str2 = 'she grew round-wombed, and had, indeed, sir, a son'
str3 = 'for her cradle ere she had a husband for her bed.'
str4 = 'Do you smell a fault?'
str_arr = [str1, str2, str3, str4]
print("Выбранная строка: ", process.extractOne('sir', str_arr, scorer = fuzz.partial_ratio))
process.extract('sir', str_arr, scorer = fuzz.partial_ratio)
```

```
Выбранная строка:  ("Sir, this young fellow's mother could: whereupon", 100)
[("Sir, this young fellow's mother could: whereupon", 100),
 ('she grew round-wombed, and had, indeed, sir, a son', 100),
 ('for her cradle ere she had a husband for her bed.', 33),
 ('Do you smell a fault?', 33)]
```

Самостоятельное задание

1.1) В таблице 2 заданы стихотворения согласно вариантам;

1.2) в переменные str1 – str4 с помощью input() запишите первые 4 строки указанного стихотворения;

1.3) выведите результат сравнения (в процентах) через пересечение множеств указанных в таблице 2 двух строк;

2.1) в функции leven сменить возвращаемую переменную на массив из двух значений: редакторского расстояния (curr[n]) и процента схожести;

2.2) вычислить leven для указанных в таблице 2 строк;

3.1) определить процент схожести заданного в таблице 2 слова для поиска и каждой указанных в таблице 2 строк трем рассмотренным методам из библиотеки thefuzz;

3.2) привести указанные в таблице 2 строки к сортированному виду (.lower, .split, sorted, .join), записать их в переменные strX_sort, где X – номер строки;

3.3) оценить схожесть сортированных строк по трем рассмотренным методам из библиотеки thefuzz;

4) реализовать поиск указанного в таблице 2 слова в массиве из str1 – str4 с использованием указанного там же scorer.

Таблица 2 – Варианты для задания

Вар.	Стихотворение	Строки	Слово для поиска	scorer
1	И. Бродский, Не выходи…	str1, str2	шип	ratio
2	В. Маяковский, Облако в штанах	str1, str2	шум	ratio
3	С. Есенин, Пой же, пой	str1, str2	пляж	ratio
4	Гёте, Фауст (гл.1)	str1, str2	лад	ratio
5	А. Блок, Ночь, улица…	str1, str2	лицо	ratio
6	М. Цветаева, Кто создан…	str1, str2	вера	ratio
7	А. Ахматова, Мы не умеем…	str1, str2	род	ratio
8	И. Бродский, Не выходи…	str2, str3	час	token_sort_ratio
9	В. Маяковский, Облако в штанах	str2, str3	рев	token_sort_ratio
10	С. Есенин, Пой же, пой	str2, str3	пол	token_sort_ratio
11	Гёте, Фауст (гл.1)	str2, str3	софа	token_sort_ratio
12	А. Блок, Ночь, улица…	str2, str3	масло	token_sort_ratio
13	М. Цветаева, Кто создан…	str2, str3	ребро	token_sort_ratio

14	А. Ахматова, Мы не умеем…	str2, str3	чин	token_sort_ratio
15	И. Бродский, Не выходи…	str3, str4	бор	ratio
16	В. Маяковский, Облако в штанах	str3, str4	раз	ratio
17	С. Есенин, Пой же, пой	str3, str4	след	ratio
18	Гёте, Фауст (гл.1)	str3, str4	медь	ratio
19	А. Блок, Ночь, улица…	str3, str4	вера	ratio
20	М. Цветаева, Кто создан…	str3, str4	изюм	ratio
21	А. Ахматова, Мы не умеем…	str3, str4	мерка	ratio
22	И. Бродский, Не выходи…	str1, str3	ком	partial_ratio
23	В. Маяковский, Облако в штанах	str1, str3	лень	partial_ratio
24	С. Есенин, Пой же, пой	str1, str3	хлеб	partial_ratio
25	Гёте, Фауст (гл.1)	str1, str3	долг	partial_ratio
26	А. Блок, Ночь, улица…	str1, str3	фон	partial_ratio
27	М. Цветаева, Кто создан…	str1, str3	мен	partial_ratio
28	А. Ахматова, Мы не умеем…	str1, str3	роща	partial_ratio
29	И. Бродский, Не выходи…	str1, str4	раз	token_sort_ratio
30	В. Маяковский, Облако в штанах	str1, str4	кров	token_sort_ratio
31	С. Есенин, Пой же, пой	str1, str4	рок	token_sort_ratio
32	Гёте, Фауст (гл.1)	str1, str4	дело	token_sort_ratio
33	А. Блок, Ночь, улица…	str1, str4	ход	token_sort_ratio
34	М. Цветаева, Кто создан…	str1, str4	морс	token_sort_ratio
35	А. Ахматова, Мы не умеем…	str1, str4	ум	token_sort_ratio

Работа 3. Нечеткие множества

Плановое время на выполнение: 2 занятия (4 ак. часа).

Сперва импортируем все необходимые модули из библиотеки *fuzzylogic*, еще одной реализации нечеткой логики на python, а также знакомой ранее mathplotlib. Символ * после import означает, что нужно импортировать все модули/функции из указанной до import библиотеки (в данной работе используется множество функций из *fuzzylogic.functions*, проще импортировать все; при этом, импортируемые функции вызываются сразу по названию, например, *sigmoid*, тогда как, импортировав весь модуль без *, через *import fuzzylogic.functions as fuzzfun* та же функция вызывалась бы менее удобно через *fuzzfun.sigmoid*).

```
from fuzzylogic.classes import Domain, Rule
from fuzzylogic.functions import *
import matplotlib.pyplot as plt
```

Сперва визуализируем два простейших множества. Для нечеткой логики из библиотеки fuzzylogic множества создаются внутри так называемого домена, который записывается в какую-либо переменную по принципу *переменная = Domain('имя', начало, конец, res = разрешение)*. Здесь имя – произвольная строка, начало – начальное значение интервала, на котором будут определены множества, конец – конечное значение этого интервала, разрешение – интервал дискретизации (шаг между соседними значениями в множествах).

```
T = Domain('My_domain', 0, 15, res = 0.1)
```

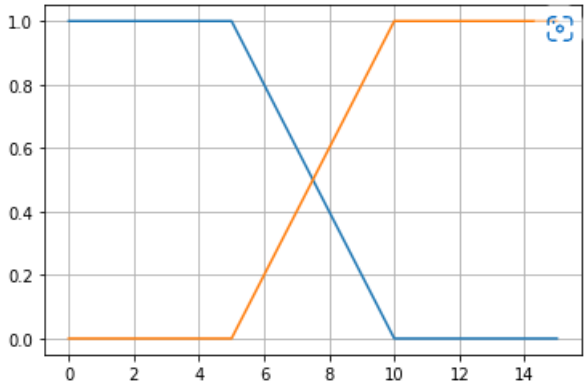
Множество в домене, записанном, как в примере, в переменную *T*, будет создаваться следующим образом: *T.имя_множества = функция_принадлежности(параметры)*, где функции принадлежности выбираются из fuzzylogic.functions (через *help(fuzzylogic.functions)* можете посмотреть имеющиеся функции), а параметры у каждой функции свои. Для примера возьмем простейшие из них – *S(начало_спада, конец_спада)* и *R(начало_роста, конец_роста)*. Первая спадает между указанным началом и концом (от 1 до 0), вторая – нарастает (от 0 до 1) в том же интервале, причем обе – по линейному закону. Назовем множества по порядку их задания, первое сформируем по функции

```
T.first = S(5, 10)
T.second = R(5, 10)
```

принадлежности R, второе – по S, причем для обеих зададим идентичные параметры.

Для вывода графика множества используется *домен.имя_множества.plot()*. Выведем оба множества, для удобства, с сеткой (*plt.grid()*).

```
plt.grid()
T.first.plot()
T.second.plot()
```

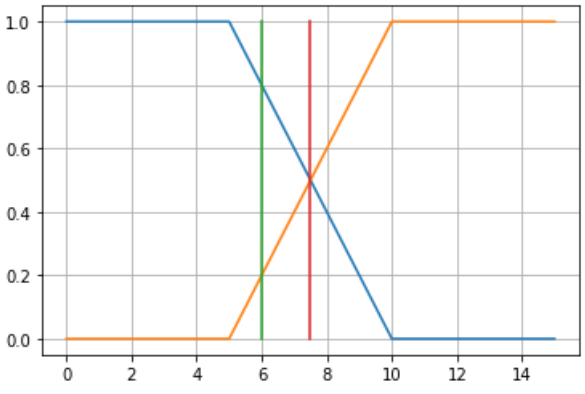


Для определения значения принадлежности конкретного значения (по оси абсцисс) к тому или иному множеству нужно исполнить команду *домен(значение)*. Чтобы визуализировать это значение (в виде столбика) на графике зададим простую функцию, как на скриншоте.

```
def plot_scatter(x):
    plt.plot([x,x],[0,1])
```

Добавим в код *point1 = 6* и *point2= 7.5*. Вызовем для каждой из них *print(домен(значение))[домен.имя_множества]* (к примеру, *T(point1)[T.first]*) для отображения принадлежности к каждому множеству, а также созданную выше *plot_scatter* для отображения точек на графике.

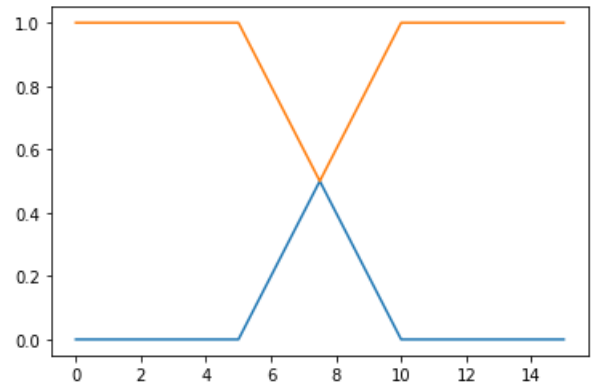
```
Принадлежность точки 6
к множ.1: 0.8
к множ.2: 0.2

Принадлежность точки 7.5
к множ.1: 0.5
к множ.2: 0.5
```



Множества можно объединять через логическое «И» (&). Для инверсии множества требуется перед ним добавить «~». Объединить множества T.first и T.second в R или S функцию не выйдет, поскольку обе они не предусматривают возможность задания получаемой конфигурации, но имеет

такую возможность функция ломаной *polygon*. Построим графики результата объединения и его инверсии.

```
T.polygon = T.first & T.second
T.polygon.plot()
T.inv_polygon = ~T.polygon
T.inv_polygon.plot()
```



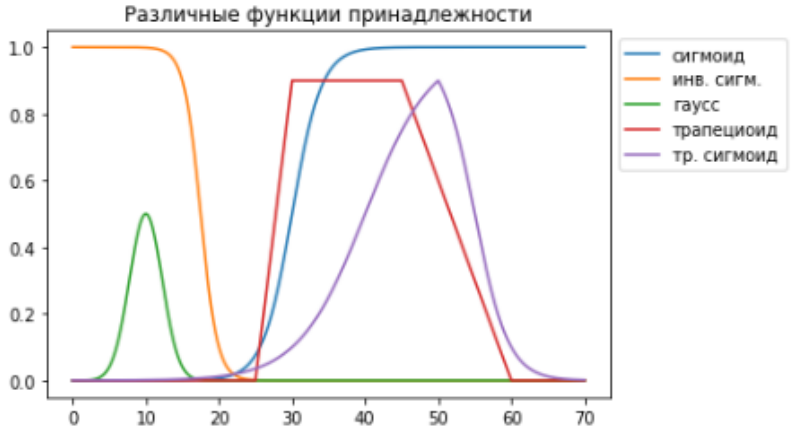
Иные функции принадлежности отражены на скриншоте ниже вместе с описанием их параметров.

```
T = Domain("test", 0, 70, res = 0.1)
# sigmoid(L, k, x0)
# L - высота, k - крутизна, x0 - положение центра
T.sigmoid = sigmoid(1, 0.5, 30)
T.sigmoid.plot()
# bounded_sigmoid(low, high, inverse = A)
# low - положение левой границы, high - положение правой,
# A - если True, то перевернут
T.bnd_sigmoid = bounded_sigmoid(15, 20, inverse = True)
T.bnd_sigmoid.plot()
# gauss(c, b, c_m=A)
# c - положение максимума, b - ширина, A - высота
T.gauss = gauss(10, .1, c_m = 0.5)
T.gauss.plot()
# trapezoid(low, c_low, c_high, high, c_m=A)
# low - начало левого фронта, c_low - конец левого,
# c_high - начало правого, high - конец правого, A - высота
T.trapezoid = trapezoid(25, 30, 45, 60, c_m = 0.9)
T.trapezoid.plot()
# triangular_sigmoid(low, high, c=M)
# low - начало, high - конец, M - положение максимума, высота=0.9
T.triangular_sigmoid = triangular_sigmoid(30, 60, c = 50)
T.triangular_sigmoid.plot()
```

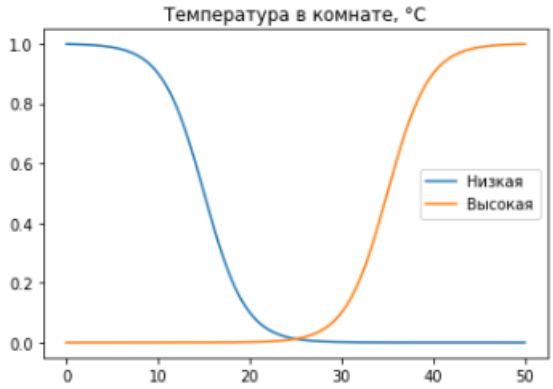
Ниже представлено продолжение той же ячейки, где задаются описания графиков (легенда) – *legends*, – в виде массива строк (порядок тот же, что при выводе графиков в коде выше). Для вывода легенды используется команда *plt.legend(строки_легенды, параметры)*. В данном случае у plt.legend используется один параметр (можно вызывать и без параметров вовсе),

который отвечает за смещение легенды вовне графика, чтобы она не накладывалась на кривые (*bbox_to_anchor = (сдвиг_по_x, сдвиг_по_y)*).

```
legends = ['сигмоид','инв. сигм.','гаусс','трапециоид','тр. сигмоид']
plt.legend(legends, bbox_to_anchor = (1.0, 1.0))
plt.title('Различные функции принадлежности')
plt.show()
```

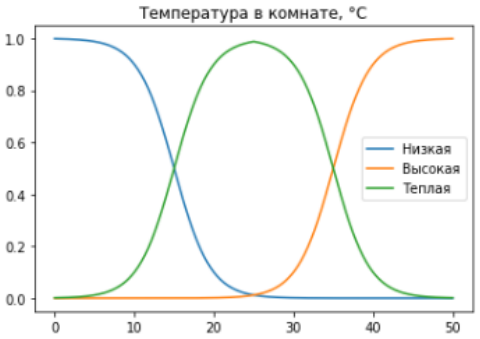


Инструмент объединения двух множеств можно использовать для построения третьего, среднего между первыми двумя. К примеру, задано, что низкой считается температура (приближенно) в комнате от 0 до 20, высокой – от 30 до 50, причем с ростом температуры «холодность» комнаты плавно (нелинейно) спадает, а «горячесть» - растет (также нелинейно). Требуется определить диапазон комфортной температуры в виде множества (для какой температуры выше принадлежность к этому множеству – та и комфортнее) и принадлежность к каждому множеству температуры 23.



Зададим множества под данную ситуацию функцией *bounded_sigmoid*, поскольку она позволяет задавать плавное изменение как с нарастанием, так и со спадом степени принадлежности. Для холодного множества выберем *bounded_sigmoid(10, 20, inverse = True)*, чтобы получить плавный спад

примерно с 10 до 20 градусов, для горячего - *bounded_sigmoid(30, 40)*, чтобы получить плавный рост с 30 до 40 градусов.




```
Принадлежность 23 градусов
к холодному: 0.028872748393428863
к теплому: 0.9711272516065711
к горячему: 0.005100318170652071
```

Третье множество теплой (или комфортной) температуры найдем пересечением первых двух через `T.warm = ~(T.hot + T.cold)`, где «+» аналогичен «ИЛИ», тогда как ранее сложение выполнялось через «&», что аналогично «И». Значение комфортности температуры 23 найдем как и прежде через *Домен(23)[Домен.теплое_множество]*

Также библиотека fuzzylogic позволяет задавать правила для проведения дефаззификации (но только с сочетанием двух входных множеств через «И», большим функционалом в этой области обладает библиотека *skfuzzy* и подобные). Правила задаются через *Rule({(величина1.множ, величина2.множ): выход_величина.множ})*, где величина 1 и 2 являются входными, а выход_величина – выходной, значение которой нужно определить. Множ здесь – множества, относящиеся к той или иной величине. В фигурных скобках можно указать несколько правил через запятую.

В таком виде запись аналогична четкой логике, к примеру, Rule((Температура.высокая, Влажность.низкая): Свет.средний}) можно понимать, как «ЕСЛИ температура в диапазоне высокого множества И влажность в диапазоне низкого множества, ТО свет должен быть из среднего множества».

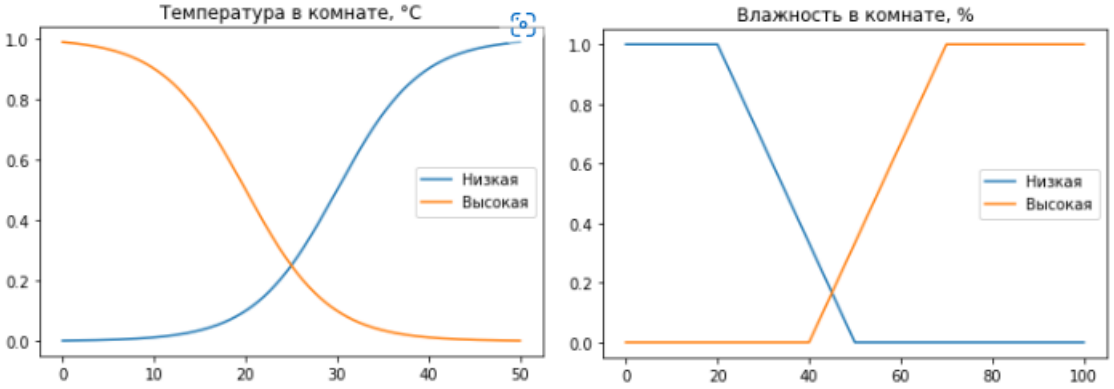
Для примера используем заданное ранее множества температуры, но без теплого множества и с измененными параметрами функции принадлежности: для низких температур от 10 до 30, для высоких – от 20 до 40. Также зададим домен для параметра влажность в диапазоне от 0 до 100, в котором низкая влажность описывается *S(20,50)*, а высокая – *R(40,70)*. Выходной величиной будет яркость света (предположим, что в зависимости от влажности и температуры в помещении меняется яркость света для комнатных растений).

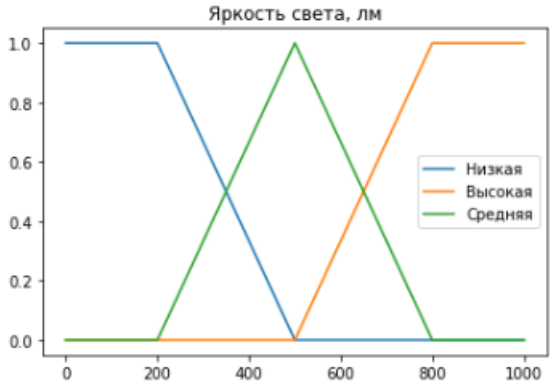
Яркость будет характеризоваться тремя диапазонами – слабый задается *S(200,500)*, сильный – *R(500,800)*, а средний – суммой высокого и низкого (через ИЛИ, +) с последующей инверсией (~).

Зададим следующие правила, связывающие эти множества:

1) ЕСЛИ температура = высокая И влажность = низкая ТО свет = средний;
2) ЕСЛИ температура = низкая И влажность = высокая ТО свет = средний;
3) ЕСЛИ температура = низкая И влажность = низкая ТО свет = слабый;
4) ЕСЛИ температура = высокая И влажность = высокая ТО свет = сильный.

Реализуем все правила в одной переменной и выведем полученное значение для влажность=50, температура=20. На скриншотах ниже также представлены графики полученных множеств.






```
my_rule = Rule({(T.hot, hum.dry): Light.med,
                (T.cold, hum.wet): Light.med,
                (T.cold, hum.dry): Light.low,
                (T.hot, hum.wet): Light.high})

values = {hum: 50, T: 20}
print('Выходное значение, лм:', my_rule(values))
```

```
Выходное значение, лм: 572.0167027091202
```

Самостоятельное задание

1.1) Задать нечеткие множества возрастов с 3 группами согласно параметрам из таблицы 3.1 (аналогично рисунку в начале работы);

1.2) определить принадлежность заданного в таблице 3.1 значения (возраста) к множествам, результат вывести с текстовым описанием;

1.3) построить график полученных множеств с наложенным на нее значением (в виде столбика), для графика использовать заголовок и легенду вне координатного поля;

2.1) задать нечеткие множества горячей и холодной температуры в комнате в соответствии с таблицей 3.2, используя только указанную функцию принадлежности;

2.2) найти множество теплой температуры в комнате на основе заданных в п. 2.1;

2.3) определить принадлежность заданной в таблице 3.2 точки к сформированным множествам, результат вывести с текстовым описанием;

2.4) построить график полученных множеств с наложенным на нее значением (в виде столбика), для графика использовать заголовок и легенду вне координатного поля;

3.1) продумать, какая величина может быть выходной (при дефаззификации), если входными являются рассмотренные выше две (возраст человека и температура в комнате);

3.2) задать диапазон выбранной выходной величины (для домена), а также не менее трех множеств в этом диапазоне (высокий/средний/низкий, сильный/средний/слабый и т.п.) с использованием той же функции, что была обозначена в таблице 3.2 для температуры;

3.3) продумать правила, связывающие все комбинации множеств входных величин с диапазонами выходной, реализовать эти правила через Rules();

3.4) вывести график множеств выходной величины (с заголовком и легендой), а также результат дефаззификации (применения Rules) к значениям входных величин, указанных в таблице 3.1 и 3.2 соответственно.

Таблица 3.1

Вариант	Молодой	Средний		Старый	Значение
	верх.гр.	ниж.гр.	верх.гр.	ниж.гр.	
1	34	30	45	47	46
2	35	34	52	56	35
3	32	30	49	51	31
4	32	31	45	45	35
5	32	28	54	55	30
6	31	30	47	51	30
7	27	22	51	51	37
8	28	28	41	45	38
9	33	30	45	50	31
10	29	29	48	53	49
11	34	34	41	45	25
12	25	24	44	48	39
13	22	19	54	59	48
14	30	27	47	51	25
15	28	25	44	45	41
16	26	25	48	51	40
17	33	31	53	58	44
18	22	18	50	53	31
19	33	33	53	55	25
20	26	22	44	49	35
21	32	32	51	55	36
22	27	24	55	57	48
23	35	32	42	45	25
24	29	29	55	58	30
25	30	26	56	58	48
26	24	24	46	46	38
27	31	31	58	59	37
28	35	32	46	47	41
29	31	29	44	45	31
30	35	32	53	57	32
31	30	27	51	53	27
32	28	26	45	50	28
33	26	24	45	47	40
34	27	24	46	49	42
35	23	18	51	56	39

Таблица 3.2

Вариант	Холодная		Горячая		Значение	Функция
	ниж.гр.	верх.гр.	ниж.гр.	верх.гр.		
1	-10	16	31	66	16	sigmoid
2	-11	5	21	67	15	bnd_sigmoid
3	-13	8	21	45	12	gauss
4	-14	16	28	78	21	trapezoid
5	-19	-2	33	83	25	triangular_sigmoid
6	-11	6	34	84	27	R
7	-15	20	43	84	30	S
8	-4	18	36	71	26	polygon
9	-4	15	39	86	32	sigmoid
10	-3	28	56	97	41	bnd_sigmoid
11	-6	21	36	80	27	gauss
12	-20	2	15	44	5	trapezoid
13	-15	18	32	69	24	triangular_sigmoid
14	-1	23	39	84	34	R
15	-4	11	46	93	32	S
16	0	30	58	105	53	polygon
17	-1	30	53	78	45	sigmoid
18	-11	7	34	75	26	bnd_sigmoid
19	-8	8	18	47	9	gauss
20	-13	14	33	83	27	trapezoid
21	-5	28	54	101	44	triangular_sigmoid
22	-3	13	34	73	21	R
23	-12	9	37	59	31	S
24	-20	10	29	56	14	polygon
25	-1	23	52	100	43	sigmoid
26	-1	29	47	76	36	bnd_sigmoid
27	-2	26	52	95	42	gauss
28	-4	28	56	100	45	trapezoid
29	-6	13	44	67	34	triangular_sigmoid
30	0	34	64	94	56	R
31	-20	12	32	76	21	S
32	-6	16	42	68	37	polygon
33	-15	17	27	71	22	sigmoid
34	-15	10	26	49	18	bnd_sigmoid
35	-17	6	28	51	18	gauss