Введение в машинное обучение. Метод ближайших соседей

Цель занятия

После освоения темы вы:

- узнаете, в каких сферах применяется машинное обучение;
- поймете назначение и принцип работы библиотеки NumPy;
- узнаете основные понятия машинного обучения;
- познакомитесь с формальной постановкой задачи машинного обучения с учителем;
- научитесь применять метод kNN для решения задач машинного обучения, узнаете его достоинства и недостатки.

План занятия

1. Введение. Сферы применения машинного обучения
2. Основные понятия машинного обучения
3. Формальная задача машинного обучения с учителем
4. Метод k ближайших соседей
5. Метрики классификации
6. Реализация kNN в Python

Конспект занятия

1. Введение. Сферы применения машинного обучения

Теория машинного обучения за последние 60 лет оформилась в самостоятельную математическую дисциплину, которая находится на стыке линейной алгебры, теории

вероятности, прикладной статистики, численных методов оптимизации, дискретного анализа, программирования.

Многие из нас ежедневно используют приложения, в основе которых лежат технологии искусственного интеллекта (ИИ) и машинного обучения. Все мы ежедневно пользуемся виртуальными помощниками, чтобы включить будильник и узнать прогноз погоды. И рекомендательными системами, чтобы решить, какой фильм посмотреть и какую еду приготовить. Сосредоточением решений машинного обучения можно считать беспилотный автомобиль, в котором внедрены системы компьютерного зрения, предсказания действий объектов на дороге, распознавания голоса и другие.

Помимо использования машинного обучения, человечество пытается с ним соревноваться. Так, в 2016 году корейски профессиональный игрок в Го Ли Седоля был побежден в этой игре программой AlphaGo. С тех пор прогресс в этой сфере зашел так далеко, что подобная программа считается простой. Ее уже можно давать студентам в качестве домашнего или семинарского задания.

Технологии машинного обучения активно используются не только для решения прикладных задач, но и в науке. Например, лектор этого курса Радослав Нейчев участвовал в команде Яндекс Cern, которая занималась проектами для Большого Адронного Коллайдера. Машинное обучение в проекте использовалось для оптимизации системы файлового хранения данных, собираемых коллайдером.

До недавнего времени единственным видом, который занимался порождением знаний из данных, был человек. Вот только делали это ученые самостоятельно. Например, Ньютон и Кеплер проанализировали огромное количество данных, определи взаимосвязи и сформулировали свои законы. Даже до н. э. Эратосфен смог определить, что Земля круглая, и определил ее радиус с 10%-й погрешностью.

Только с момента открытия машинного обучения мы перепоручили эту задачу компьютеру. Получив определенное количество данных и наблюдений и найдя логические паттерны, он может получить модель и сформулировать выводы. То есть технологии машинного обучения пытаются автоматизировать извлечение знаний из полученных данных. К сожалению, пока что это не просто рычаг, который можно дернуть и получить модель. Для этого специалистам машинного обучения приходится использовать сложные математические методы, сделать огромное количество априорных предположений.

В этом курсе мы будем разбираться, что такое машинное обучение, как оно работает и как его использовать.

2. Основные понятия машинного обучения

Введем основные понятия, которые пригодятся на протяжении всего курса. При работе с данными необходимо понять, где эти данные доступны.

Пример датасета (набора данных, выборки):

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Target (passed)
John	22	5	4	Brown	English	5	TRUE
Aahna	17	4	5	Brown	Hindi	4	TRUE
Emily	25	5	5	Blue	Chinese	5	TRUE
Michael	27	3	4	Green	French	5	TRUE
Some student	23	3	3	NA	Esperanto	2	FALSE

Датасет состоит из атомарных сущностей — объектов, наблюдений, точек.

Наблюдения предполагаются любыми, но обладают двумя важными свойствами:

- независимые друг от друга;
- **одинаково распределенные.**

По сути, мы предполагаем, что есть некоторая общая генеральная совокупность, некоторое общее распределение, из которого насэмплированы наши объекты.

Допустим, в нашем примере задача — предсказать оценку по машинному обучению у студента. Предполагаем, что все наблюдения **i.i.d.**

Почему нам необходима одинаковая распределенность? Потому что именно исходя из этого, мы имеем право использовать одну модель, которую впоследствии введем. Мы можем предполагать, что есть одна общая зависимость между наблюдениями и целевыми переменными. Если данные из различных распределений, то скорее всего

зависимости там будут тоже разные. И никакого права у нас использовать одну модель у нас нет. i.i.d. — очень важное свойство.

Каждое явление описывается различными признаками:

- числовыми,
- категориальными,
- строками,
- графами и др.

Категориальный признак — это признак, который принимает значение из некоторого множества. Причем есть различные категории, и они предполагаются неупорядоченными.

Все вместе признаки создают признаковое описание для каждого объекта. На текущий момент будем считать, что каждый объект описывается одними и теми же признаками, и все объекты находятся в признаковом пространстве. У нас есть n-мерное пространство, и каждое наблюдение — точка в этом пространстве.

Чтобы поставить задачу машинного обучения с учителем, нам необходимо не только признаковое описание объекта, но и **целевая переменная**.

Target (целевая переменная, зависимая переменная) позволяет нам поставить задачу — предсказать результат по признаковому описанию каждого объекта.

Как правило, выборка делится на две большие части:

1. **Матрица плана (матрица объект/признак, design matrix)** — та часть, которая известна про все объекты**.** Это матрица из всех объектов и всех признаковых описаний:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Target (passed)
John	22	5	4	Brown	English	5	TRUE
Aahna	17	4	5	Brown	Hindi	4	TRUE

Emily	25	5	5	Blue	Chinese	5	TRUE
Michael	27	3	4	Green	French	5	TRUE
Some student	23	3	3	NA	Esperanto	2	FALSE

2. **Целевые переменные**, где мы для каждого объекта знаем значение целевой переменной:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Target (passed)
John	22	5	4	Brown	English	5	TRUE
Aahna	17	4	5	Brown	Hindi	4	TRUE
Emily	25	5	5	Blue	Chinese	5	TRUE
Michael	27	3	4	Green	French	5	TRUE
Some student	23	3	3	NA	Esperanto	2	FALSE

Это столбец или матрица, если у нас target векторный.

Когда мы решаем задачу обучения с учителем, мы говорим, что у нас есть target.

Учитель — это наличие разметки, которая пришла извне: либо от эксперта, либо от естественных причин. Поэтому это называется обучение с учителем. В обучении без учителя никакого таргета у нас нет.

Про целевую переменную стоит отметить две важные вещи:

1. Она может быть числовой.
2. Она может быть категориальной.

Простой случай — таргет может быть бинарным:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Target (passed)
John	22	5	4	Brown	English	5	TRUE
Aahna	17	4	5	Brown	Hindi	4	TRUE
Emily	25	5	5	Blue	Chinese	5	TRUE
Michael	27	3	4	Green	French	5	TRUE
Some student	23	3	3	NA	Esperanto	2	FALSE

Обычно эта задача рассматривается отдельно, и это **задача бинарной классификации**. В этом случае это **label** или метка класса. В задаче **регрессии**, где у нас ответ — число, это значение целевой переменной.

Когда есть целевая переменная и матрица плана, возникает вопрос: что нам делать дальше? Дальше нужно строить некоторое отображение из описания объекта в значение целевой переменной. По сути, нужно строить модель.

Пример. Для примера разберем задачу регрессии — будем предсказывать число. Как можно предсказать оценку для студента?

У нас уже есть информация о студенте — как он закрыл предыдущий курс. Можно попробовать усреднить:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Predicted (mark)

John	22	5	4	Brown	English	5	4.5
Aahna	17	4	5	Brown	Hindi	4	4.5
Emily	25	5	5	Blue	Chinese	5	5
Michael	27	3	4	Green	French	5	3.5
Some student	23	3	3	NA	Esperanto	2	3

$\hat{mark}_{ML} = \frac{1}{2} mark_{Statistics} + \frac{1}{2} mark_{Python}$ — довольно неплохое **предсказание (predict)**, поскольку те, кто много уделяет время учебе, имеют стабильно высокие оценки. Если студент время от времени ленится, то оценки получаются «плавающие». Возникает вопрос: это хорошая модель или плохая?

Можем ввести другую модель:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Predicted (mark)
John	22	5	4	Brown	English	5	1
Aahna	17	4	5	Brown	Hindi	4	5
Emily	25	5	5	Blue	Chinese	5	2
Michael	27	3	4	Green	French	5	4
Some student	23	3	3	NA	Esperanto	2	3

Эта модель случайным образом расставляет числа от 1 до 5:

$\hat{mark}_{ML} = random(integer\ from\ [1;\ 5])$

Как сравнивать модели. Чтобы сравнивать модели, необходимо ввести **критерий качества** или **функцию потерь (loss function)**. Качество мы пытаемся получить большим, а функцию потерь пытаемся уменьшать.

Введем функцию потерь. Она формулируется для двух значений целевой переменной — одни истинные, другие предсказанные:

Square deviation	Target (mark)	Predicted (mark)
16	5	1
1	4	5
9	5	2
1	5	4
1	2	3

Например, можно использовать среднюю квадратичную ошибку (MSE):

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2 = \frac{1}{N}\sum_i (y_i - \hat{y}_i)^2$$

Это одна из классических мер ошибки в задаче регрессии.

Можно использовать среднюю абсолютную ошибку (MAE):

$$MAE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\|\mathbf{y} - \hat{\mathbf{y}}\|_1 = \frac{1}{N}\sum_i |y_i - \hat{y}_i|$$

Каждая функция потерь позволяет сравнить два предсказания. В зависимости от выбора функции потерь мы меняем оптимизационную задачу, которую решаем. Если неправильно выбрать функцию потерь, то мы решим не ту задачу. Поэтому при выборе функции потерь надо быть очень осторожным.

Когда выбираем модель и функцию потерь, мы автоматически формулируем некоторую **гипотезу** о наших данных. Например, если зависимость линейная, то мы предполагаем, что среднеквадратичные отклонения важны.

Параметры модели. Чтобы извлекать из данных знания, необходимо дать модели несколько степеней свободы. То есть параметры модели или какие-то способы менять саму себя.

Для простоты введем параметрическую модель:

Name	Age	Statistics (mark)	Python (mark)	Eye color	Native language	Target (mark)	Predicted (mark)
John	22	5	4	Brown	English	5	4.447
Aahna	17	4	5	Brown	Hindi	4	4.734
Emily	25	5	5	Blue	Chinese	5	5.101
Michael	27	3	4	Green	French	5	3.714
Some student	23	3	3	NA	Esperanto	2	3.060

$\hat{mark}_{ML} = \omega_1 \cdot mark_{Statistics} + \omega_2 \cdot mark_{Python}$ — теперь у нас не арифметическое среднее, а взвешенное среднее, где веса ω_1 и ω_2 — настраиваемые параметры. Мы не знаем, чему равны эти параметры. Мы предполагаем, что можем настроить эти параметры на основании данных.

Теперь хотим найти, с какими оптимальными весами нужно сложить оценки, чтобы максимально точно восстановить оценку по ML. Эти значения можно искать различными методами оптимизации. Например, случайным поиском, генетикой. Когда модель дифференцируема, можем использовать градиентные методы.

Гиперпараметр. Параметры настраиваем на основании наших данных. Гиперпараметры мы выбираем до начала настройки параметров. То есть гиперпараметры фиксируются до начала работы с данными, их нельзя подобрать

градиентами. Приходится подбирать вручную или поисками / переборами. Параметры же настраиваются автоматически.

Какие могут быть гиперпараметры:

1. Тип модели.
2. Типы используемых признаков.
3. Ограничения на модель — например, глубина решающего дерева.

Соберем термины воедино:

- Выборка (Dataset), набор наблюдений.
- Признак (observation, datum).
- Категория признака (Feature).
- Матрица объект/признак (Design matrix).
- Целевая переменная (target).
- Предсказание (Prediction).
- Модель (Model).
- Функция потерь (Loss function).
- Параметр (Parameter).
- Гиперпараметр (Hyperparameter).

3. Формальная задача машинного обучения с учителем

Поставим формальную задачу машинного обучения с учителем:

- Предположим, у нас есть некоторая выборка $L = \{x_i, y_i\}_{i=1}^{n}$, где x — признаковое описание объекта, y — значение целевой переменной

  Их набор образует весь наш набор данных, датасет. Множество x — матрица объект/признак, множество y — столбец или матрица таргетов.

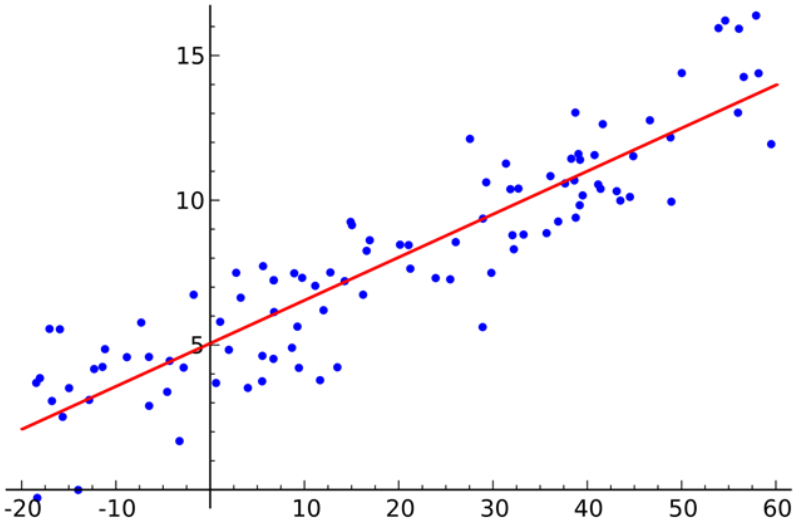
- $(x \in R^p,\ y \in R)$, для регрессии. R^p — векторы в p-мерном пространстве.
- $x_i \in R^p,\ y_i \in \{+1, -1\}$ для бинарной классификации.

- Ищем модель $f(x)$, которая будет предсказывать значение целевой переменной на основании x.
- Есть функция потерь $Q(x, y, f)$ или функция эмпирического риска, которая будет показывать, насколько хорошо данная модель описывает данную выборку.

По-хорошему начинать нужно с конца:

1. Сначала ответить на вопрос, как будем измерять качество? Что мы вообще оптимизируем?
2. Следующий вопрос — сколько данных уже доступно? Данных может быть много, мало, не быть совсем или есть возможность данные добрать. Данные могут быть дорогими.
3. Чтобы получить хороший результат, нужно использовать подходящие под задачу модели. Не стоит выбирать самую популярную.

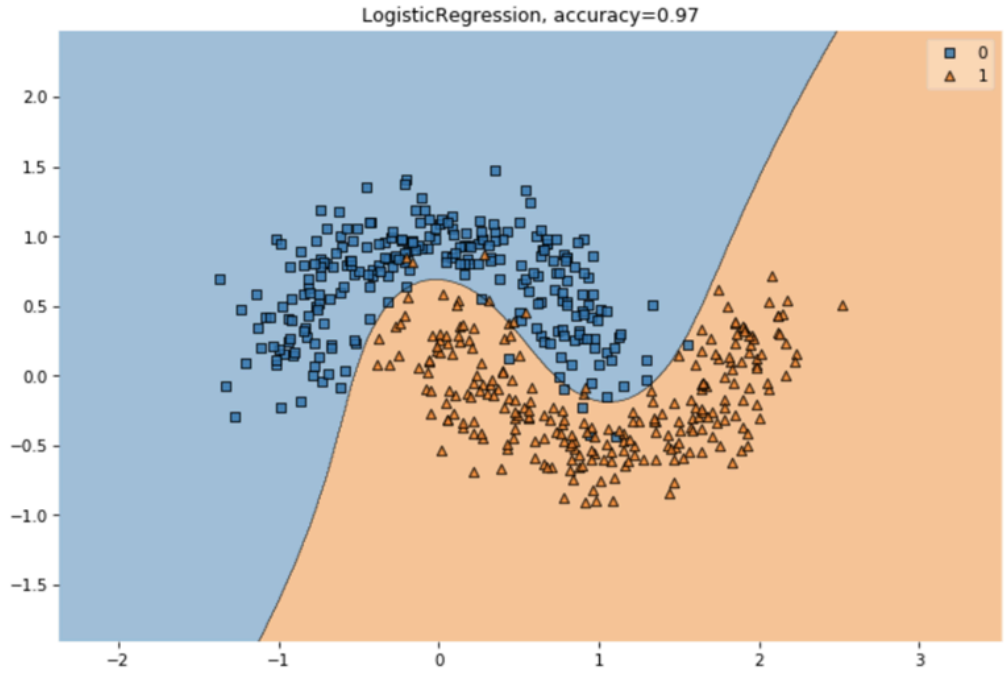
Задача регрессии. Допустим, есть набор целевых переменных. Мы хотим через них провести прямую:



Предположим, что наша модель линейная:

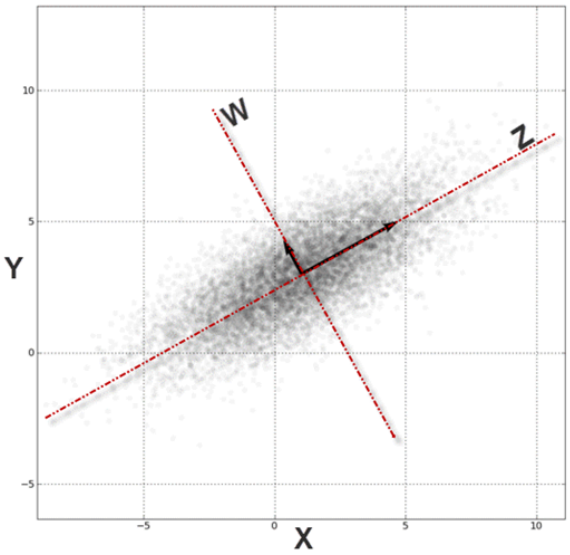
$$\hat{Y}_i = b_0 + b_1 X_i$$

Задача классификации. Есть набор синих и оранжевых точек. Мы хотим через них провести границу:



Задача уже нелинейная.

Задача обучения без учителя. Например, задача снижения размерности. Есть облако точек, и мы хотим снизить размерность данного облака:



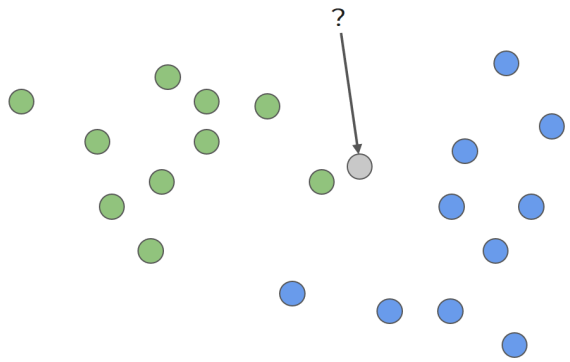
Задача без учителя, потому что нет прямых разметок, как снижать размерность.

4. Метод k ближайших соседей

Познакомимся с первым методом, который позволяет решать уже достаточно неплохо небольшие простые задачи.

Метод ближайших соседей (kNN) работает как в регрессии, так и в классификации. Работает он по принципу «скажи мне, кто твой друг / сосед, и я скажу, кто ты».

Пусть есть некоторое множество объектов:



Нам неизвестен цвет серой точки – она либо синяя, либо зеленая. Мы можем посмотреть на ее соседей. Ближайший сосед зеленый, значит, серая точка тоже зеленая. Но если посмотреть на 5 ближайших соседей, то там окажется 3 синих и 2 зеленых, и окажется, что серая точка синяя. Вопрос выбора числа соседей очень важен.

На практике такой подход неплохо работает:

- Мы хотим оценить среднюю цену за квадратный метр квартиры в некотором районе. Можно усреднить цены за метр соседних квартир.
- Хотим узнать курс студента, который живет в общежитии. Можно посмотреть на курс всех его соседей.
- Среднее потребление автомобилем бензина сопоставимо с машинами подобной модели.

Но возникает сразу несколько вопросов:

- Как выбирать число соседей?
- Как это работает на практике, а не на словах?

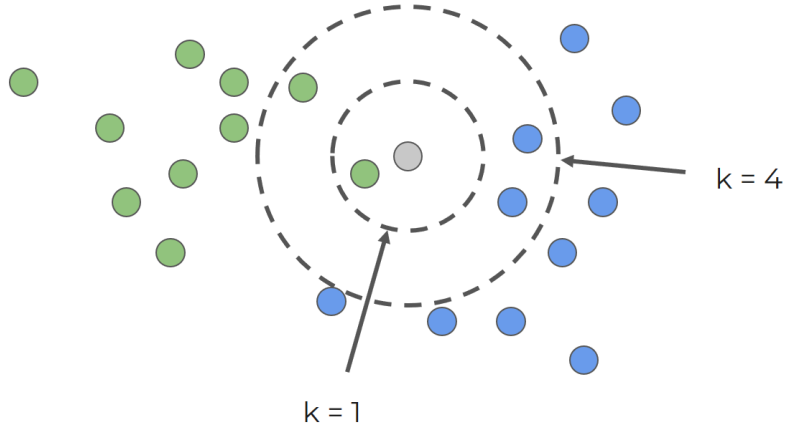
Допустим, у нас есть точка серого цвета. Как нам понять, она должна быть зеленой или синей? У нас есть обучающая выборка. Посчитаем расстояние от данной точки до всех объектов нашей обучающей выборки, отсортируем и посмотрим на k ближайших соседей. На основании k ближайших соседей будем принимать решение.

Получается, всю выборку надо обойти и со всеми посчитать расстояние. Именно поэтому этот подход работает только для небольших задач.

У нас есть два гиперпараметра:

1. Число соседей.
2. Метрика расстояния.

Чаще всего используется евклидова метрика. Можно использовать сумму модулей или что-то еще придумать. Главное, чтобы выполнялась аксиома метрики.



Можно получить результат для каждого из значений k.

Абсолютно так же этот подход будет работать и для задачи регрессии. В этом случае просто усредняем значения всех ближайших точек.

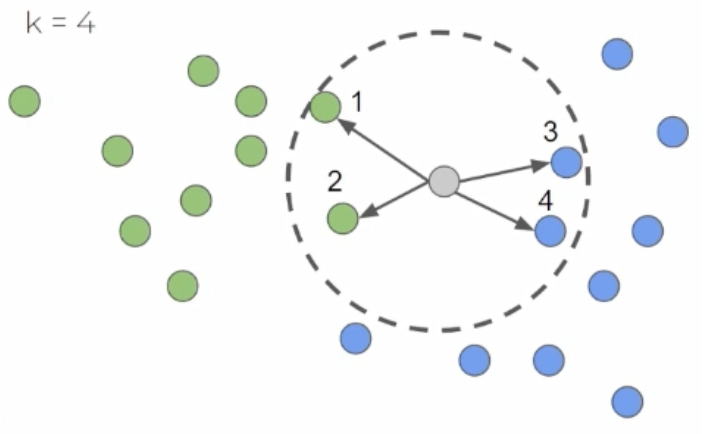
Сначала выбираем гиперпараметр, потом получаем предсказание. Мы можем перебрать различные значения гиперпараметра, оценить качество со всеми возможными вариантами и выбрать оптимальный.

Как сделать kNN лучше:

- Выбрать число соседей (перебираем гиперпараметр k).
- Выбрать подходящую метрику:
    - hamming;
    - евклидова метрика;
    - косинусная мера близости;
    - расстояние Минковского;
    - др.
- Можно соседям присвоить различные веса — взвешенный метод ближайших соседей.

Взвешенный метод ближайших соседей. Посчитаем расстояние до 4-х ближайших точек. И скажем, что каждая из них вносит свой вклад обратно пропорционально

расстоянию до серой точки. Чем дальше точка, тем меньше она оказывает влияние на серую точку.



Вес может быть 1/расстояние или функция от 1/расстояние:

- $\omega(x_{(i)}) = \omega_i$
- $\omega(x_{(i)}) = \omega(d(x, x_{(i)}))$

Предсказание теперь будет зависеть не просто от ближайших соседей, а от взвешенных ближайших соседей:

$$p_{green} = \frac{\omega(x_1)+\omega(x_2)}{\omega(x_1)+\omega(x_2)+\omega(x_3)+\omega(x_4)}$$

$$p_{blue} = \frac{\omega(x_3)+\omega(x_4)}{\omega(x_1)+\omega(x_2)+\omega(x_3)+\omega(x_4)}$$

Как именно зависят веса от расстояния d, должен определить эксперт.

kNN достаточно простой, но очень красивый подход. Если предположить, что есть неограниченное число объектов, то с помощью kNN можно решать практически любую задачу.

5. Метрики классификации

Поговорим о том, как можно измерить качество полученной модели, используя доступные данные, валидационную выборку. И как понять, подходит ли модель под ту или иную бизнес-задачу.

В задаче регрессии использовались те же самые функции ошибки, что использовались при обучении: MSE, MAE и др.

В классификации свои методы оценки качества модели.

Accuracy

$Accuracy = \frac{1}{n}\sum_{i=1}^{n}\left[y_i^t = y_i^p\right]$ – точность (accuracy) классификации. Какая доля объектов выборки была классифицирована правильно.

Пример:

target: 1 0 1 0 0 0 0 1 0 0

predicted: 0 0 1 0 0 0 0 1 1 0

$Accuracy = \frac{8}{10} = 0.8$

Accuracy имеет недостатки. Представим датасет, который очень сильно не сбалансирован. 99% объектов относятся к одному классу, а 1% — к другому. У нас есть алгоритм, который всегда предсказывает первый класс. При этом Accuracy будет 99%.

В этом случае используют Balanced accuracy:

$$Balanced\ accuracy = \frac{1}{C}\sum_{k=1}^{C}\frac{\sum_i\left[y_i^t=k\ and\ y_i^t=y_i^p\right]}{\sum_i\left[y_i^t=k\right]}$$

Balanced accuracy учитывает размер каждого класса. Мы смотрим на долю классификации каждого из классов и потом усредняем.

Точность и полнота

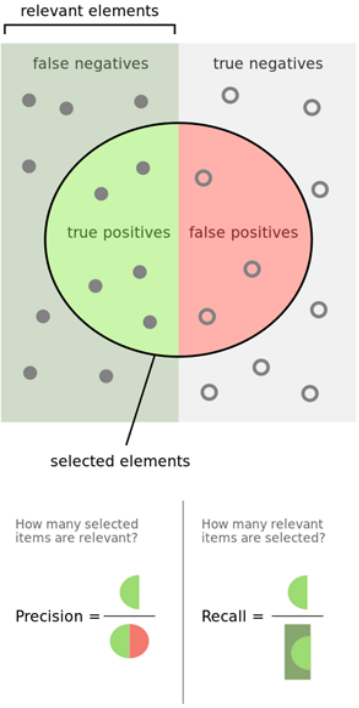
Точность и полнота (Precision and Recall) показывают нам оценку точности с точки зрения ошибки первого рода и второго рода.

У нас есть наблюдение, и мы можем правильно отклассифицировать объекты, принадлежащие к позитивному классу и к негативному классу. И неправильно отнести другие объекты к позитивному и негативному классам.

		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	**True positive**	**False positive**, Type I error
	Predicted condition negative	**False negative**, Type II error	**True negative**

$Precision = \frac{TP}{TP+FP}$ — насколько точно можно оценить метку положительного класса среди всех предсказанных объектов положительного класса.

$Recall = \frac{TP}{TP+FN}$ — показатель, насколько много объектов из истинно положительного класса мы смогли охватить.



Точность позволяют нам оценить, насколько мы уверены, что предсказанная метка положительного класса релевантна. Полнота позволяет оценить, насколько мы уверены, что мы не потеряли никакие объекты из положительного класса.

В оценке точности и полноты у нас есть целевой класс TP, на который мы в первую очередь обращаем внимание. При несбалансированной выборке точность будет 90%, полнота — 100%, а классификатор будет абсолютно бесполезен.

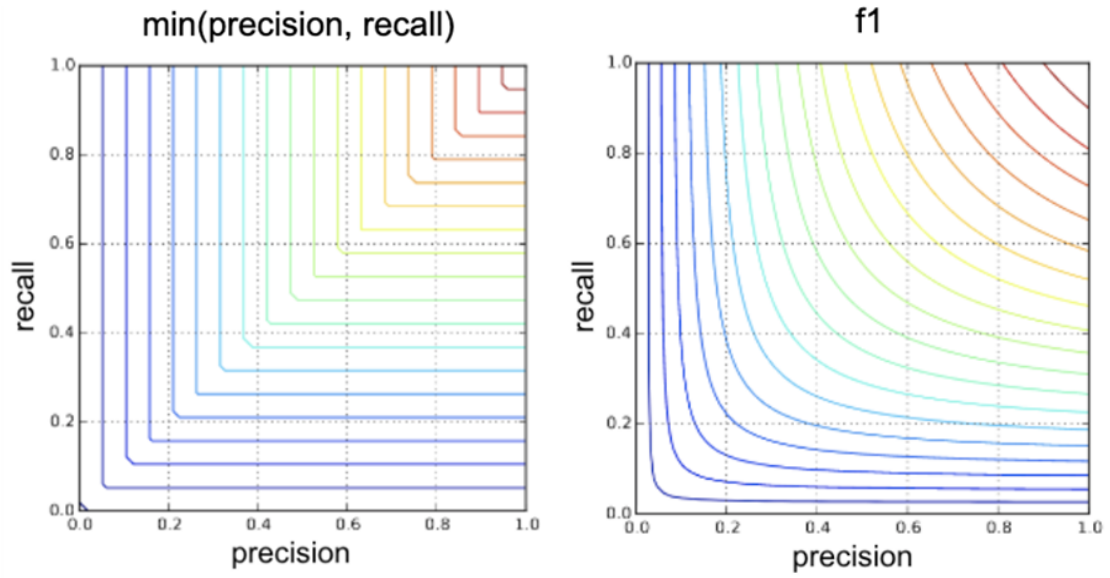
Точность и полнота — это два числа. Одновременно оптимизировать два числа трудно.

F-score

F-score — усреднение точности и полноты:

$$F_1 = \frac{2}{precision^{-1}+recall^{-1}} = 2\frac{Precision \cdot Recall}{Precision+Recall}$$

По сути, это попытка аппроксимировать минимум из точности и полноты.



F-score часто используют, чтоб измерять качество классификации.

В общем случае, когда точность и полнота могут быть неравнозначны в задаче:

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

Коэффициент β^2 позволяет сфокусироваться на точности или полноте.

6. Реализация kNN в Python

Посмотрим на первый в данном курсе алгоритм машинного обучения, а именно, на метод ближайших соседей kNN. Мы не будем заниматься собственноручной имплементацией данного подхода, вместо этого воспользуемся уже готовой реализацией из библиотеки `sklearn`. Воспользуемся базовыми принципами работы с библиотекой, глубокого обзора ее возможностей сейчас проводить не будем, сделаем это в следующих занятиях.

Импортируем все необходимые нам модули:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
```

```
from sklearn import datasets, neighbors
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Для чего импортируем:

- Библиотека `numpy` нам понадобится для работы с таблицами.
- `matplotlib` необходима для построения графиков.
- `accuracy_score` — способ оценки качества, он подсчитывает долю правильных ответов.
- `model_selection` позволяет разбить данные на две части: обучающую и валидационную.

В методе kNN считаем расстояния до всех точек из обучающей выборки, выбираем из них ближайшие. И как правило, выбираем наиболее частый или наиболее вероятный ответ, будь то метка класса или ответ в задаче регрессии.

Сгенерируем себе простой датасет:

```
classification_problem = datasets.make_classification(
    n_samples=100,
    n_features=2,
    n_informative=2,
    n_classes=3,
    n_redundant=0,
    n_clusters_per_class=1,
    random_state=3,
)
```

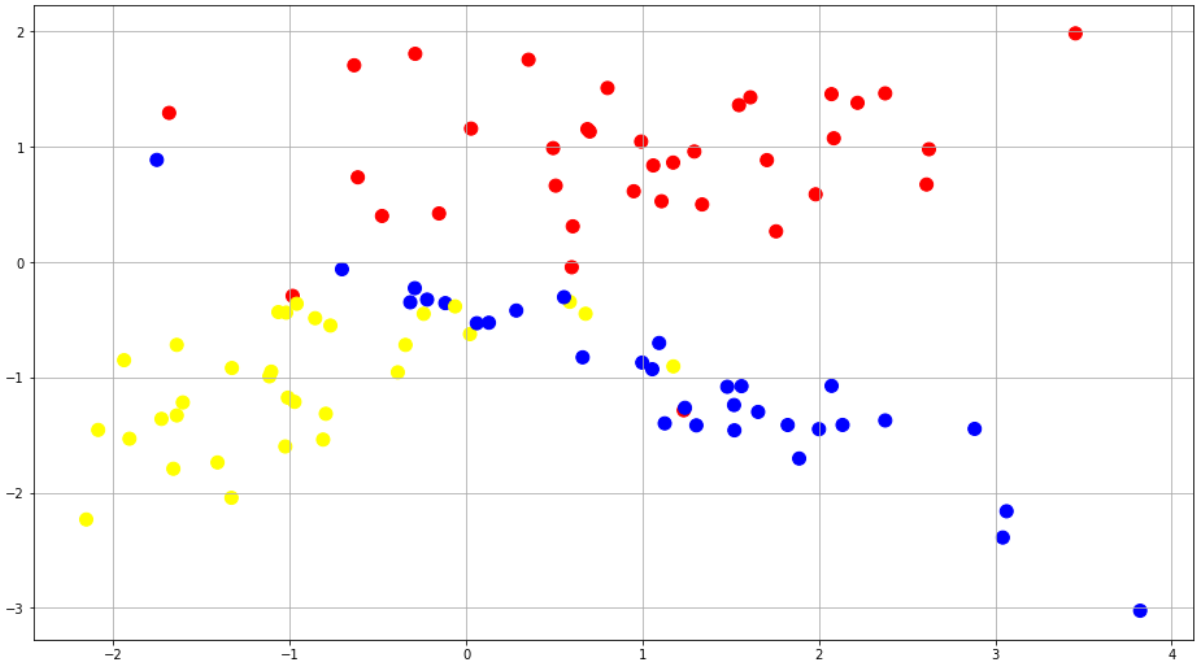
Создадим функцию, которая будет рисовать датасет, чтобы нам было удобней:

```
def plot_dataset(dataset: (np.ndarray, np.ndarray)):
    colors = ListedColormap(["red", "blue", "yellow"])
```

```
    plt.figure(figsize=(16, 9))
    plt.grid()
    plt.scatter(dataset[0][:, 0], dataset[0][:, 1], c=dataset[1],
cmap=colors, s=100)
    plt.show()
```

Теперь можно посмотреть на наш датасет:

```
plot_dataset(classification_problem)
```



Получилось три группы разных цветов. Обратим внимание, что выборка не является линейно разделимой, потому что присутствует некоторый «шум» из-за наличия «хвостов» у гауссовых распределений.

Разобьем данные на обучающую и валидационную части:

```
train_data, test_data, train_labels, test_labels = train_test_split(
    *classification_problem,
```

```
    test_size=0.3,
    random_state=1,
)
```

Что значит звездочка (*) в коде? Рассмотрим на примере:

```
def my_func(a, b, c):
       print(a)
       print(b)
       print(c)
       return
```

```
my_func(1, 2, 3)
```

Создадим список:

```
my_list = [4, 5, 6]
```

Писать так не очень удобно:

```
my_func(my_list[0], my_list[1], my_list[2])
```

Поэтому если есть объект, в котором упорядочены аргументы, можно написать:

```
my_func(*my_list)
```

Это гораздо удобнее, чем все переписывать.

То же самое можно делать с именованными аргументами. Если есть словарь, где каждому имени аргумента соответствует его значение, можно подавать его в следующем виде:

```
**my_dict
```

Теперь построим модель. Вызываем метод fit, чтобы обучиться на обучающих данных:

```
clf = neighbors.KNeighborsClassifier()
clf.fit(train_data, train_labels)
```

Делаем предсказание:

```
predictions = clf.predict(test_data)
accuracy_score(test_labels, predictions)
```

Получили, что на отложенных данных 9 из 10 объектов классифицируются правильно.

Посмотрим на предсказания:

```
predictions
```

Теперь визуализируем нашу задачу:

```
def make_meshgrid(
    data: np.ndarray,
    step: float = 0.05,
    border: float = 0.5,
):
    x_min, x_max = data[:, 0].min() - border, data[:, 0].max() + border
    y_min, y_max = data[:, 1].min() - border, data[:, 1].max() + border
    return np.meshgrid(np.arange(x_min, x_max, step), np.arange(y_min,
y_max, step))
```

plot_decision_surface показывает все возможные цвета, где должен быть ответ:

```
def plot_decision_surface(estimator, train_data, train_labels, test_data,
test_labels):
    colors = ListedColormap(["red", "blue", "yellow"])
```

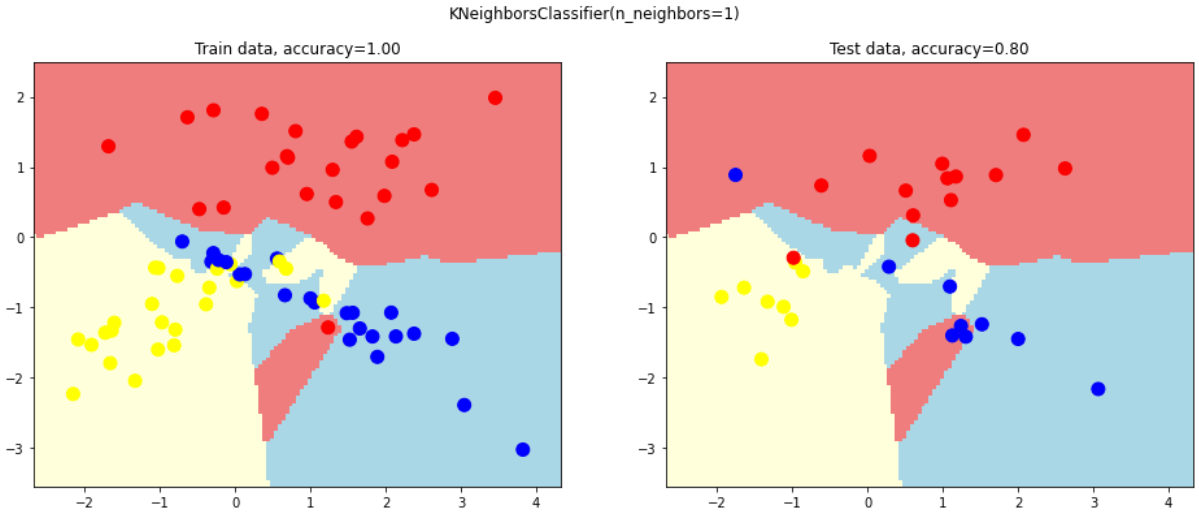
```
    light_colors = ListedColormap(["lightcoral", "lightblue",
"lightyellow"])


    # fit model
    estimator.fit(train_data, train_labels)


    # set figure size
    fig = plt.figure(figsize=(16, 6))
    fig.suptitle(estimator)


    # plot decision surface on the train data
    plt.subplot(1, 2, 1)
    xx, yy = make_meshgrid(train_data)
    mesh_predictions = np.array(estimator.predict(np.c_[xx.ravel(),
yy.ravel()])).reshape(xx.shape)
    plt.pcolormesh(xx, yy, mesh_predictions, cmap=light_colors,
shading="auto")
    plt.scatter(train_data[:, 0], train_data[:, 1], c=train_labels,
s=100, cmap=colors)
    acc = accuracy_score(train_labels, estimator.predict(train_data))
    plt.title(f"Train data, accuracy={acc:.2f}")


    # plot decision surface on the test data
    plt.subplot(1, 2, 2)
    plt.pcolormesh(xx, yy, mesh_predictions, cmap=light_colors,
shading="auto")
    plt.scatter(test_data[:, 0], test_data[:, 1], c=test_labels, s=100,
cmap=colors)
    acc = accuracy_score(test_labels, estimator.predict(test_data))
```

```
    plt.title(f"Test data, accuracy={acc:.2f}")
```

И можно посмотреть, что было бы, если бы мы построили классификатор, основываясь на одном ближайшем соседе:

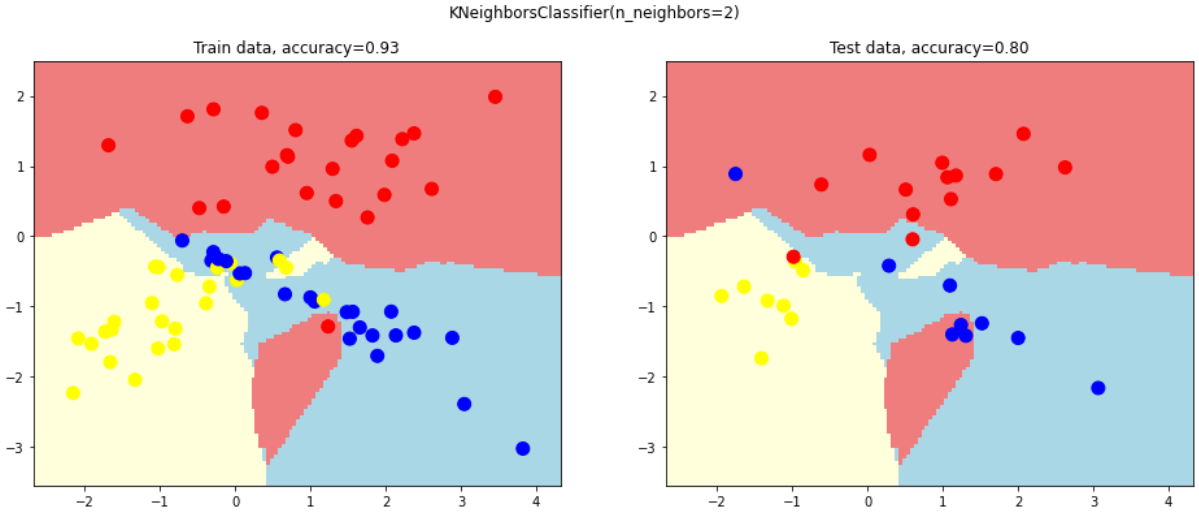
```
estimator = neighbors.KNeighborsClassifier(n_neighbors=1)


plot_decision_surface(estimator, train_data, train_labels, test_data,
test_labels)
```

Поверхности и на train, и на test абсолютно одинаковые:

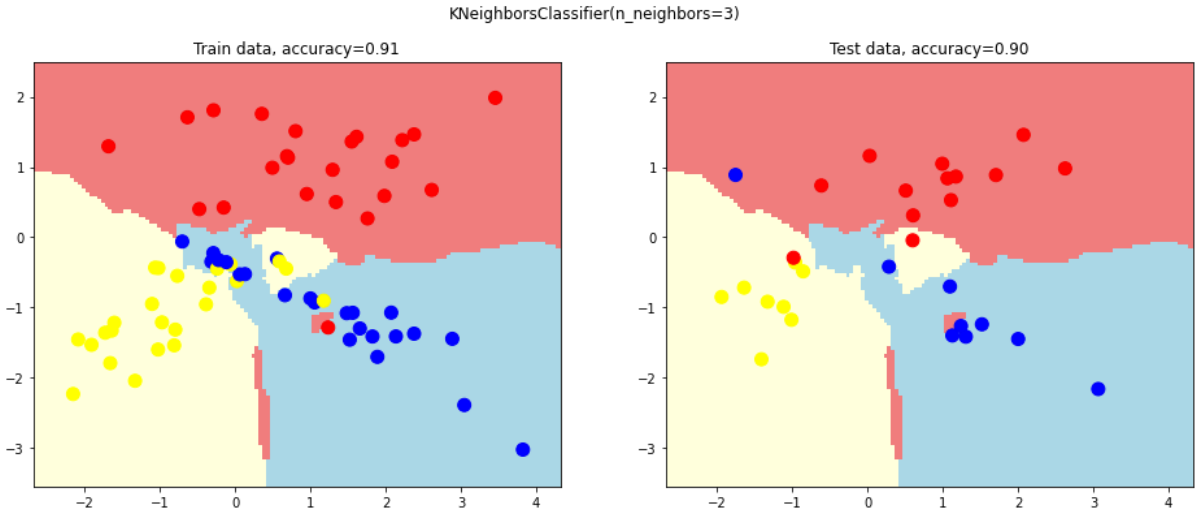


Попробуем увеличивать количество соседей и смотреть на качество классификации:

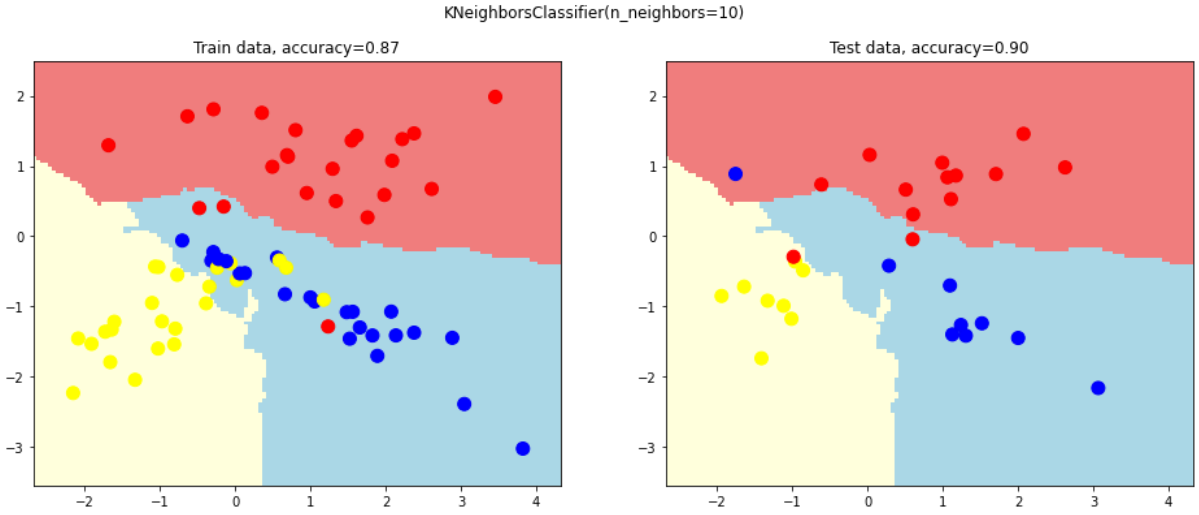
```
estimator = neighbors.KNeighborsClassifier(n_neighbors=2)


plot_decision_surface(estimator, train_data, train_labels, test_data,
test_labels)
```



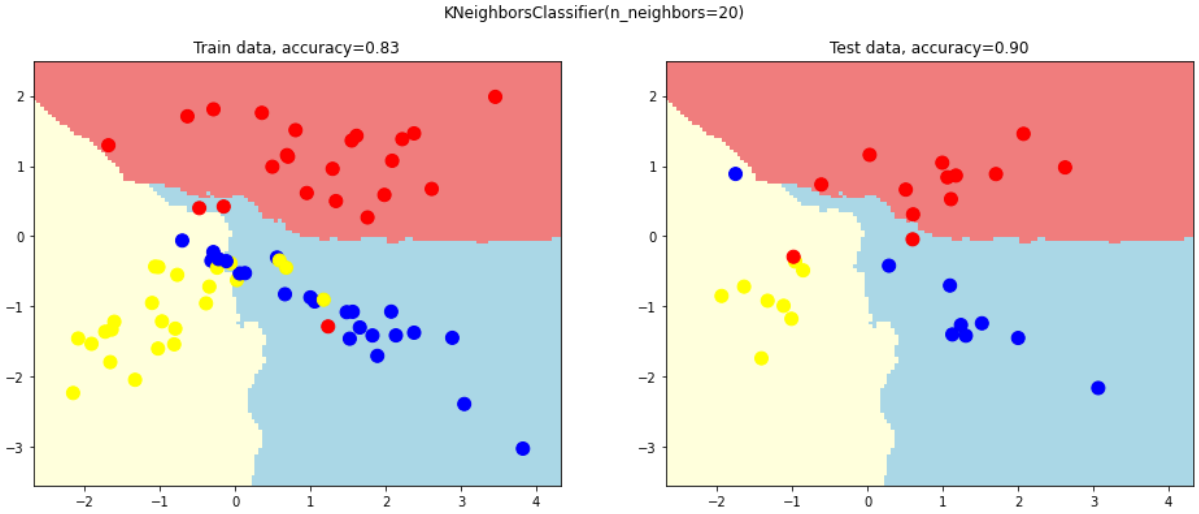
В случае трех соседей практически исчезает красная зона:



При 10 соседях практически ровные границы. Никаких лишних островов нет:



При 20 соседях получаются более плавающие границы, потому что здесь учитывается практически весь объем обучающей выборки:



Посмотрим размер нашей выборки:

```
train_data.shape
```

При 20 соседях границы выглядят естественно, и они соответствуют тому, как мы строили наши данные. При 30 и 40 соседях практически ничего не меняется.

Всегда можно проверить качество классификации на каждом из примеров. Можно добавлять подсчет accuracy_score на каждом шаге.

Получили, что на каждом шаге accuracy_score примерно 0,9 — очень точно. У нас задача достаточно простая; и крайне маленькая обучающая и валидационная выборка. Здесь хорошо использовать кросс-валидацию, о которой мы поговорим позже.

Посмотрим теперь на чуть более сложную задачу из 100 объектов и 100 признаков:

```
hard_problem = datasets.make_classification(
    n_samples=100,
    n_features=100,
    n_informative=50,
    n_classes=3,
    n_redundant=50,
    n_clusters_per_class=1,
    random_state=42,
)
```

Опять разделяем наши данные на train и валидацию:

```
train_data, test_data, train_labels, test_labels = train_test_split(
    *hard_problem,
    test_size=0.3,
    random_state=1,
)
```

Возьмем 5 соседей, обучим и посмотрим на качество:

```
clf = neighbors.KNeighborsClassifier(n_neighbors=5)
clf.fit(train_data, train_labels)
```

```
predictions = clf.predict(test_data)
```

```
accuracy_score(test_labels, predictions)
```

Получаем 60%. Почему качество так разительно упало?

Чем больше размерность нашего пространства, тем хуже будет нашему классификатору. kNN просто считает расстояния до всех возможных точек в обучающей выборке и выбирает из них наиболее близкие, чтобы усреднить или посчитать наиболее частый ответ.

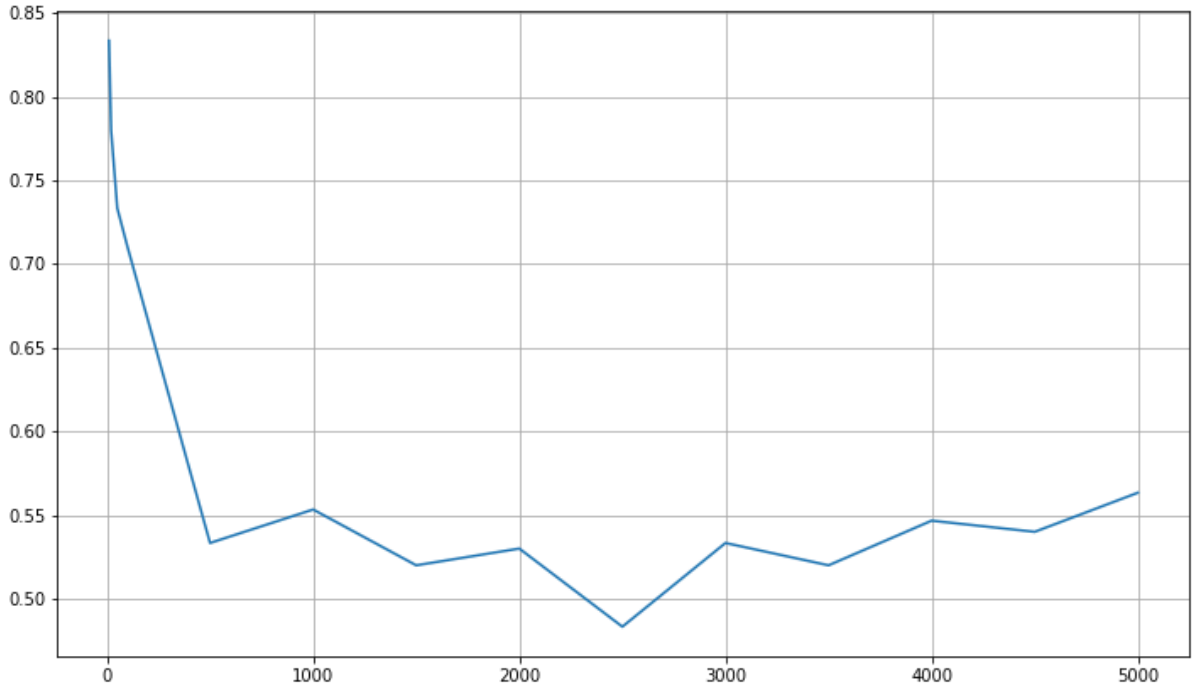
Проблема в том, что при использовании пространства высокой размерности, подсчет расстояний затрудняется и если некоторые признаки бесполезные, мы по факту начинаем обращать внимание лишь на некоторые признаки из обучающей выборки.

Посмотрим, как будет себя вести kNN, если мы будем варьировать количество признаков:

```
def train_knn_classifier(dimensions, n_classes):
    scores = []

    for dim in dimensions:
        problem = datasets.make_classification(
            n_samples=1000,
            n_features=dim,
            n_informative=dim // 2,
            n_classes=5,
            n_redundant=dim // 2,
            n_clusters_per_class=1,
            random_state=42,
        )

        train_data, test_data, train_labels, test_labels =
train_test_split(
            problem[0],
```

```
            problem[1],
            test_size=0.3,
            random_state=1,
        )

    clf = neighbors.KNeighborsClassifier(n_neighbors=5)
        clf.fit(train_data, train_labels)

        predictions = clf.predict(test_data)
        acc = accuracy_score(test_labels, predictions)

        scores.append(acc)

    plt.figure(figsize=(12, 7))
    plt.plot(dimensions, scores)
    plt.grid()
    plt.show()
```

```
train_knn_classifier([10, 20, 50, 100, 500, 1000, 1500, 2000, 2500, 3000,
3500, 4000, 4500, 5000], 5)
```



Видим, что качество разительно падает при увеличении количества признаков. И чем больше признаков, тем медленнее это все работает. kNN крайне неудобна с точки зрения увеличения размерности пространства и увеличения размера обучающей выборки. Для подсчета состояний нужно пройти по всем объектам и посчитать все зависимости от всех признаков.

Задача на подумать. Что происходит, когда меняешь random_state?

Посмотрим на что-то более сложное. Например, на kNN на чуть более серьезном датасете:

```
# If you are using colab, uncomment this cell

!curl
https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data
> wine.csv
```

Скачаем себе датасет про вина:

```
dataset = pd.read_csv("wine.csv", header=None)

dataset.head()
```

Это задача классификации. Нулевой столбец — метки классов. Есть 13 признаков.

Файл можно считать не только локально, но и сразу из интернета:

```
dataset =
pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data", header=None)

dataset.head()
```

Это удобно, если нет доступа к каким-либо утилитам командной строки.

Нулевой столбец заменим на 1:

```
x = dataset.drop(0, axis=1).to_numpy()

y = dataset[0].to_numpy()
```

Можно посмотреть, сколько объектов и какого класса:

```
from collections import Counter

Counter(y)
```

Разобьем на train и test:

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)


print(x_train.shape)

print(x_test.shape)
```

Пропишем, как нам обучать нашу модель:

```
clf = neighbors.KNeighborsClassifier(n_neighbores=5)


clf.fit(x_train, y_train)
```

Посмотрим дефолтную метрику качества для данной модели. Посмотрим на качество обучающей выборки:

```
clf.score(x_train, y_train)
```

То же самое можно сделать через accuracy:

```
accuracy_score(clf.predict(x_train), y_train)
```

Посмотрим на валидации на качество нашей модели:

```
from sklearn.metrics import balanced_accuracy_score
```

```
clf.score(x_test, y_test)
```

Будем использовать для разнообразия balanced_accuracy:

```
balanced_accuracy_score(clf.predict(x_test), y_test)
```

У balanced_accuracy качество стало ниже, так как там сложение происходит со стат. весами, которые соответствуют доле объекта данного класса в общей выборке.

Дальше выбор за слушателем. Можно попробовать:

- Построить модель чуть более подходящую.
- Перебрать различное количество соседей.
- Поварьировать используемую метрику. В нашем примере по умолчанию использовалась метрика Минковского.
- Использовать L1 или что-то еще.

Дополнительные материалы для самостоятельного изучения

1. docs, examples: http://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors
2. https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data