Линейная алгебра. Библиотека NumPy

Цель занятия

После освоения темы вы:

- вспомните понятия вектора и матрицы, векторного пространства, нормы вектора, ортогональности и гиперплоскости;
- вспомните, как выполнять базовые операции над векторами и матрицами;
- поймете назначение и принцип работы библиотеки NumPy;
- научитесь применять NumPy для работы с векторами и матрицами.

План занятия

1. Векторы. Основные операции над векторами
2. Матрицы. Основные операции над матрицами
3. Вычислительные функции библиотеки NumPy. Массивы
4. Векторы. Решение линейных уравнений в NumPy
5. Использование NumPy в задачах обработки данных. Генерация мелодии
6. Работа с табличными данными и векторами
7. Библиотека NumPy. Линейная алгебра в NumPy
8. Задача снижения размерности
9. Метод главных компонент

Конспект занятия

1. Векторы. Основные операции над векторами

Для изучения машинного обучения необходимо качественное и глубокое понимание линейной алгебры. Именно поэтому изучение машинного обучения стоит начать именно с повторения основ.

Начнем с базовых понятий.

Скаляр — число в действительной числовой оси. Например, -2.

$\alpha \in R$ — скаляр

Вектор — упорядоченный набор чисел. Вектор может быть одномерный, двумерный, n-мерный. Здесь n — действительное число, не бесконечность. Бесконечномерные и комплекснозначные вектора рассматривать не будем.

$x \in R^n$ – вектор

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Примеры векторов:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \in \mathbb{R}^3, \quad \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \in \mathbb{R}^5$$

где R^3 — трехмерное пространство.

Матрица — набор векторов, поставленных либо вертикально (столбцы), либо горизонтально (строки).

$A \in R^{m \times n}$ — матрица

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Примеры матриц:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{2\times 2}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

Эти понятия будем использовать практически на каждом занятии по машинному обучению. Векторы задают наши объекты (как правило, вектор-строки), матрицами

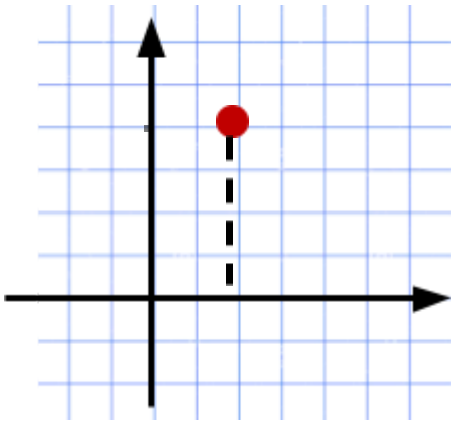
будем задавать либо линейное преобразование, либо обучающую выборку. Скалярами — веса, параметры, коэффициенты.

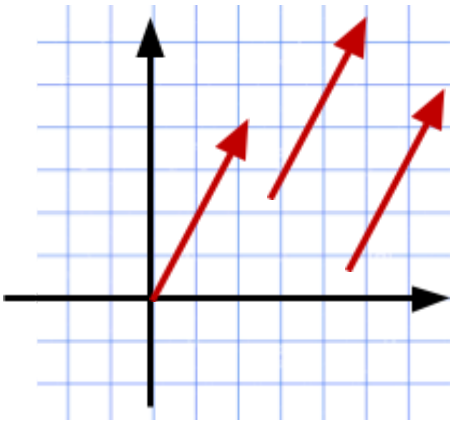
Физический смысл вектора. Вектор — набор упорядоченных чисел. Если мы рассмотрим декартовы координаты, то вектор — это точка в декартовых координатах. И в то же время вектору соответствует непосредственно вектор, направленный из начала координат в эту точку и перемещенный в любую другую точку.

Вектор задается направлением и длиной. Нам это понадобится в дальнейшем, потому что векторы мы будем рассматривать и как точки, и как направления.

Итак, вектор — это:

- упорядоченный набор чисел: $x = [1,\ 2]$;
- точка в декартовых координатах;
- направление + расстояние.





Векторные пространства

Далее мы будем работать не просто с векторами в вакууме, а с векторами в векторных, а чаще в линейных пространствах.

Векторное пространство — математическая структура, представляющая собой набор элементов, называемых векторами. Для них определены операции сложения друг с другом и умножения на число (скаляр).

Во-первых, для любой пары векторов мы можем получить их сумму, которая также будет принадлежать исходному множеству векторов:

$+\colon\ V \times V \to V$

Во-вторых, это операция умножения на скаляр. При умножении любого вектора на скаляр мы получаем другой вектор, но он также принадлежит исходному векторному пространству:

$\times: \ R \times V \to V$

Мы не можем выйти за пределы нашего множества векторов с помощью операций сложения или умножения. Эти операции удовлетворяют следующим свойствам:

	Свойство	**Пример**
1.	Ассоциативность +	$x + (y + z) = (x + y) + z$
2.	Коммутативность +	$x + y = y + x$
3.	Нейтральный элемент +	$\exists 0 \in V: \ \forall x \in V \quad 0 + x = x$
4.	Нейтральный элемент ·	$\forall x \in V \quad 1 \cdot x = x$
5.	Обратный элемент +	$\forall x \in V \ \exists - x \in V: \ \ x + (-x) = 0$
6.	Ассоциативность ·	$\alpha(\beta x) = (\alpha\beta)x$
7. 8.	Дистрибутивность	$(\alpha + \beta)x = \alpha x + \beta x$ $\alpha(x + y) = \alpha x + \alpha y$

Векторные операции

Сумма:

$x = \left[x_1 x_2 \ \vdots \ x_n\right] \in R^n, \ y = \left[y_1 y_2 \ \vdots \ y_n\right] \in R^n$

$x + y = \left[x_1 + y_1 x_2 + y_2 \ \vdots \ x_n + y_2\right] \in R^n$

Умножение на скаляр:

$x = \left[x_1 x_2 \ \vdots \ x_n\right] \in R^n, \ \alpha \in R, \ \alpha x = \left[\alpha x_1 \alpha x_2 \ \vdots \ \alpha x_n\right] \in R^n$

По сути, умножение на скаляр — это растяжение вектора вдоль его направления. Соотношение координат и наклон вектора при умножении на скаляр не меняется. Если умножить на отрицательное число, вектор будет развернут и будет смотреть в противоположную сторону, но все еще лежать на той же прямой.

Рассмотрим пример сложения и умножения в трехмерном пространстве:

$x, y \in R^3$:

$$x = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \qquad y = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

$$10x = \begin{bmatrix} 10 \\ 0 \\ 10 \end{bmatrix}$$

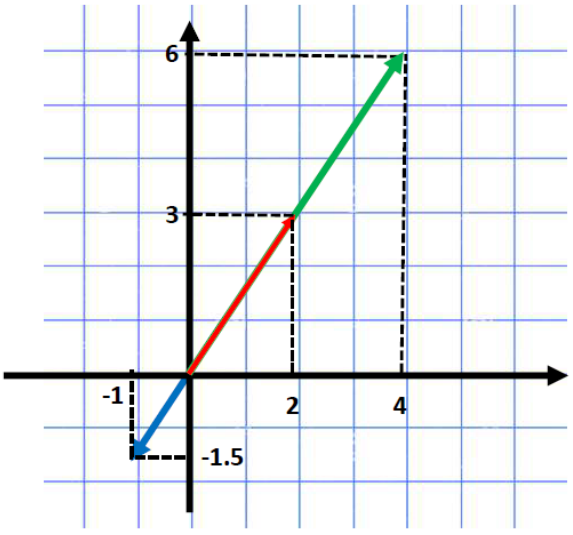
$$x + y = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

Рассмотрим геометрические интерпретации векторных операций. Геометрическая интерпретация умножения на скаляр:

$\vec{a} = [2, 3]$

$2\vec{a} = [4, 6]$

$-0.5\vec{a} = [-1, -1.5]$

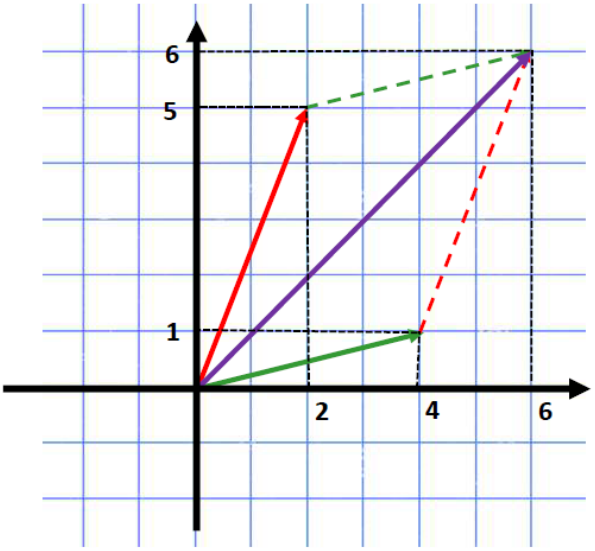


Геометрическая интерпретация сложения векторов:

$\vec{a} = [2, 5]$

$\vec{b} = [4, 1]$

$\vec{a} + \vec{b} = [6, 6]$



Вернемся к векторным пространствам.

Пример 1.

$(R^n, +, \cdot)$, $n \in N$ – векторное пространство, где

- сложение:
  $x + y = (x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n)$
- умножение на скаляр:
  $\alpha x = \alpha(x_1, x_2, \ldots, x_n) = (\alpha x_1, \alpha x_2, \ldots, \alpha x_n)$

Пример 2.

P^n – множество полиномов степени $\leq n$ с действ. коэф.

Например: $P^3 = \{ax^3 + bx^2 + cx + d \mid a, b, c, d \in R\}$

- Сумма:

$$\left(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0\right) + \left(b_n x^n + b_{n-1} x^{n-1} + \ldots + b_0\right) =$$

$$= \left(a_n + b_n\right) x^n + \left(a_{n-1} + b_{n-1}\right) x^{n-1} + \ldots + \left(a_0 + b_0\right) \in P^n$$

- Умножение на скаляр:

$$\lambda\left(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0\right) = \lambda a_n x^n + \lambda a_{n-1} x^{n-1} + \ldots + \lambda a_0 \in P^n$$

Аксиомы (1) – (8) выполнены → $(P^n, +, \cdot)$ **векторное пространство**

Скалярное произведение

Скалярное произведение — это операция над двумя векторами, результатом которой является скаляр.

Свойства скалярного произведения:

- Симметричность: $\forall x, y \in V \ \langle x, y \rangle = \langle y, x \rangle$
- Неотрицательная определенность: $\forall x \in V \backslash \{0\} \ \langle x, x \rangle > 0$ и $\langle x, 0 \rangle = 0$

Рассмотрим пример вычисления скалярного произведения:

$x = [x_1, ..., x_n], \ y = [y_1, ..., y_n] \in R^n$

$(x, y) = x_1 y_1 + x_2 y_2 + \ ... \ + x_n y_n$

Например:

$x = [1, \ 2, \ 3, \ 4], \ y = [-1, \ 0, \ 1, \ 2]$

$(x, y) = 1 \cdot (-1) + 2 \cdot 0 + 3 \cdot 1 + 4 \cdot 2 = \ -1 + 0 + 3 + 8 = 10$

Пространство $\left(R^n, \ +, \ \cdot\right)$ с указанным скалярным произведением $(\cdot, \cdot)$ называется **евклидовым**.

Нормы

Нормой в векторном пространстве V называется функция $\| \ \cdot \ \|: V \to R$, сопоставляющая вектору $x \in V$ его длину $\|x\| \in R$.

Норма отображает вектор в число.

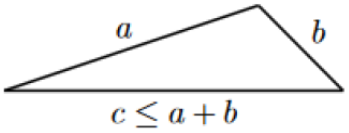
Свойства нормы:

1. $\forall x \in V, \ \forall \lambda \in R \ \ \|\lambda x\| = |\lambda| \|x\|$

При умножении вектора на скаляр мы можем вынести скаляр из самого вектора. Здесь используется модуль, потому что если мы поменяем направление вектора, длина остается той же.

2. Неотрицательно определена: $\forall x \in V \ \ \|x\| \geq 0$ and $\|x\| = 0 \Leftrightarrow x = 0$

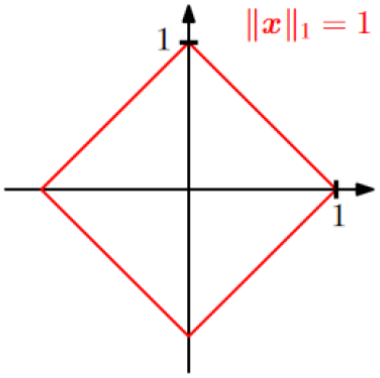
3. Неравенство треугольника: $\forall x, y \in V \ \|x + y\| \leq \|x\| + \|y\|$



Норму мы в основном будем использовать в двух целях — оценить норму вектора весов и оценить вектор ошибок. Во втором случае будем вычислять норму, чтобы посчитать, какую ошибку мы совершили при предсказании объектов.

Рассмотрим примеры норм.

Манхэттенская норма:



$x \in R^n$:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|[1,\ 2,\ 3]\|_1 = 1 + 2 + 3 = 5$$

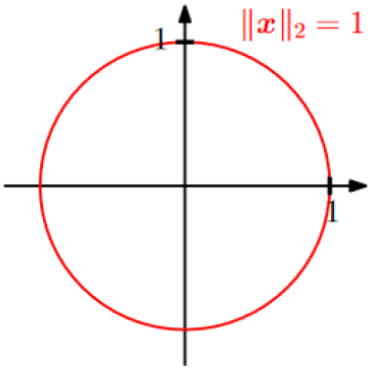
$$\|[1,\ 0]\|_1 = 1 + 0 = 1$$

$$\|[-\ 1,\ 0]\|_1 = |-\ 1| + 0 = 1$$

Ромбик на рисунке — на самом деле, сфера. Ведь сфера — это множество точек, равноудаленных от центра.

Давайте считать норму векторов (расстояние) как сумму модулей компонентов. Тогда получается, что все точки на ромбике равноудалены от центра, потому что сумма модулей их двух компонент порождает поверхность. Отсюда можно сделать вывод, что это сфера.

Евклидова норма:



$x \in R^n$:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$\|[1,\ 2,\ 3]\|_2 = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$$

$$\|[1,\ 0]\|_2 = \sqrt{1^2 + 0} = 1$$

$\|[-1, 0]\|_2 = \sqrt{(-1)^2 + 0} = 1$

Другие нормы:

Для $x = [x_1, ..., x_n] \in R^n$ l_p – норма задается как:

$\|x\|_p = \sqrt[p]{\sum_{i=1}^{n} |x_i|^p}$.

- l_1 – манхэттенская норма $\|\cdot\|_1$
- l_2 – евклидова норма $\|\cdot\|$
- l_∞: $\|x\|_\infty = |x_i|$

$\|[1, 2, 3]\|_\infty = 3, \quad \|[1, 0]\|_\infty = 1, \quad \|[-1, 0.5]\|_\infty = 1$

Обратите внимание на l_∞, которая показывает максимальный элемент по модулю в нашем векторе. Все, кроме максимального элемента, мы игнорируем.

Свяжем скалярное произведение и норму. Во-первых, скалярное произведение может индуцировать (порождать) норму: $\|x\| := \sqrt{\langle x, x \rangle}$

Например, евклидова норма:

$\sqrt{(x, x)} = \sqrt{x_1^2 + x_2^2 + ... + x_n^2} = \|x\|_2$

❗ **Не каждое скалярное произведение порождает норму.**

Например, манхэттенская норма.

Рассмотрим **неравенство Коши-Шварца**. Скалярное произведение и индуцированную им норму связывает следующее неравенство:

$|\langle x, y \rangle| \leq \|x\| \cdot \|y\|$

Например, для евклидовой нормы:

$|(x, y)| \leq \|x\|_2 \cdot \|y\|_2$

Обратите внимание, что оно работает только если норма индуцирована скалярным произведением.

Расстояние между векторами. Расстояние между векторами, или метрика, задается как норма разницы векторов.

Расстояние между векторами x и y определяется как

$$d(x,\ y) := \|x - y\| = \sqrt{\langle x - y,\ x - y \rangle}$$

Для евклидовой нормы получается привычное евклидово расстояние

$$d(x,\ y) = \|x - y\|_2 = \sqrt{(x - y,\ x - y)} = \sqrt{\left(x_1 - y_1\right)^2 + \left(x_2 - y_2\right)^2 + \ldots + \left(x_n - y_n\right)^2}$$

Углы между векторами

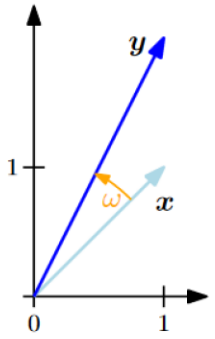
Угол между векторами может быть определен через скалярное произведение.

Согласно неравенству Коши-Шварца:

$|(x, y)| \leq \|x\| \cdot \|y\|,$

$-1 \leq \frac{(x,y)}{\|x\| \cdot \|y\|} \leq 1,$

ω: $\cos cos\ \omega = \frac{(x,y)}{\|x\| \cdot \|y\|}$ – угол между x и y



Зная косинус между векторами, мы всегда можем посчитать угол между ними.

Вне зависимости от того, какова мера пространства, посчитать можно по этой формуле.

Примеры расчета угла между векторами:

1. Найдем угол ω между $x = [5,\ 0]$ и $y = [1,\ 1]$:

$$\omega = arccos \frac{(x,y)}{\|x\|\|y\|} = arccos \frac{5 \cdot 1 + 0 \cdot 1}{\sqrt{5^2 + 0^2} \cdot \sqrt{1^2 + 1^2}} = arccos \frac{5}{5\sqrt{2}} = arccos \frac{\sqrt{2}}{4} = \frac{\pi}{4}$$

2. Найдем угол ω между $x = \left[\sqrt{3},\ 0, 1\right]$ и $y = [1,\ 0,\ 0]$:

$$\omega = arccos \frac{(x,y)}{\|x\|\|y\|} = arccos \frac{\sqrt{3}}{\sqrt{3+1} \cdot \sqrt{1}} = arccos \frac{\sqrt{3}}{2} = \frac{\pi}{6}$$

3. Найдем угол ω между $x = [1,\ 0,\ 0,\ 0,\ 1]$ и $y = [0,\ 1,\ 1,\ 0,\ 0]$:

$$\omega = arccos \frac{(x,y)}{\|x\|\|y\|} = arccos \frac{0}{\sqrt{2} \cdot \sqrt{2}} = arccos\ 0 = \frac{\pi}{2}$$

Ортогональность. Два вектора перпендикулярны или ортогональны друг другу, когда скалярное произведение между ними равно 0.

В зависимости от того, какое скалярное произведение мы выбираем, векторы будут по-разному ортогональны и не ортогональны друг другу.

Например, в $\mathbb{R}_n$ с заданным ранее скалярным произведением:

$x = [2,\ 3],\ y = [1,\ 0],\ (x, y) = 2 \neq 0 \rightarrow x$ и y не ортогональны.

$x = [1,\ 2,\ 3],\ y = [-2, 1,\ 0],\ (x, y) = -2 + 2 + 0 = 0 \rightarrow x$ и y ортогональны.

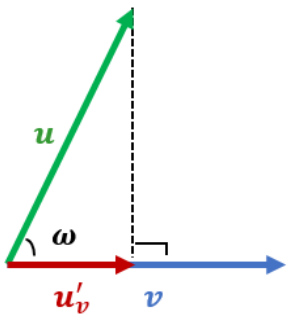
$x = [1,\ 0],\ y = [0, 1],\ (x, y) = 0,\ \|x\| = \|y\| = 1 \rightarrow x$ и y ортогональны.

Векторы ортогональны, только когда все компоненты при суммировании и умножении дают ноль.

Зачем ортогональность в машинном обучении. Понятие ортогональности в машинном обучении используется в методе главных компонент, который состоит в разложении пространства на ортогональные компоненты. Из них выбирается только некоторое подмножество, которое наиболее полезно для описания наших данных.

Угол между векторами в машинном обучении используется постоянно — например, косинус угла меры близости слов, эмбеддингов. При построении гиперплоскости нам также важно, на какой угол она отклоняется от чего-нибудь заранее заданного.

Ортогональная проекция. Пусть задана пара векторов u и v. u'_v — ортогональная проекция u на v.



Найдем длину u'_v:

$0 \leq \omega \leq 90,$

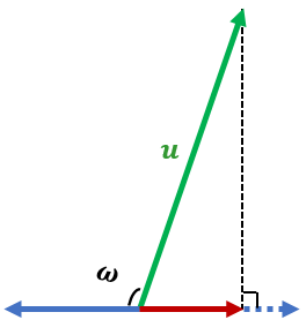
$(u, v) = \|u\|\|v\| \cos\cos \omega = \|u\|\|v\| \frac{\|u'_v\|}{\|u\|} =$

$= \|u'_v\|\|v\|$

Один вектор проецируем на другой, то есть берем только ту часть вектора u, которая сонаправлена вектору v. Причем неважно, в одну они сторону смотрят или в разную.

Рассмотрим случай вектора, который смотрит в другую сторону.

Пусть задана пара векторов u и v. u'_v — ортогональная проекция u на v.



Найдем длину $u_v^{'}$:

$90 \leq \omega \leq 180$,

$-(u,v) = \|u\|\|v\| \cos cos\ \omega = \|u\|\|v\| \frac{\|u_v^{'}\|}{\|u\|} =$

$= \|u_v^{'}\|\|v\|$

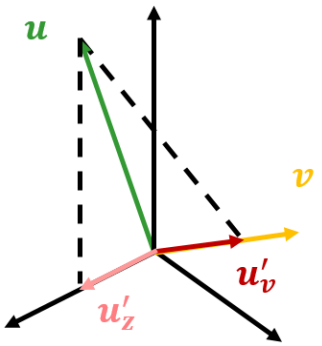
В общем случае формула угла между векторами выглядит следующим образом.

Пусть задана пара векторов u и v. $u_v^{'}$ — ортогональная проекция u на v.

Найдем длину $u_v^{'}$:

$|(u,v)| = \|u_v^{'}\|\|v\| \leftrightarrow \|u_v^{'}\| = \frac{|(u,v)|}{\|v\|}$,

$u_v^{'} = \frac{(u,v)}{(v,v)} v$



Рассмотрим примеры вычисления ортогональной проекции:

1. Найдем проекцию $u = [1,\ 3,\ 2]$ на $z = [0,\ 0,\ 1]$:

$u_z^{'} = \frac{(u,z)}{(z,z)} z = 2z = [0,\ 0,\ 2]$

2. Найдем проекцию $u = [1,\ 3,\ 2]$ на $v = [4,\ 1,\ 3]$:

$u_v^{'} = \frac{(u,v)}{(v,v)} v = \frac{4+3+6}{16+1+9} v = \frac{1}{2} v = [2,\ 0.5,\ 1.5]$

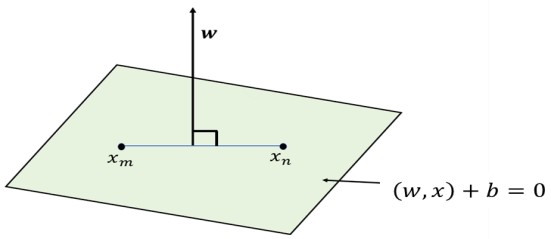
В машинном обучении мы будем использовать ортогональные проекции в задаче классификации, во всех линейных механизмах классификации, включая нейронные сети.

Гиперплоскость

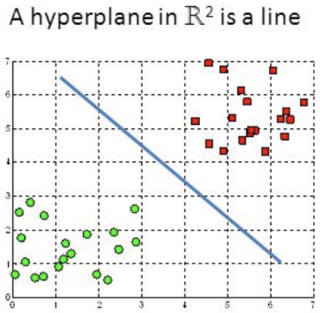
Гиперплоскость задается как $w_1x_1 + w_2x_2 + \ldots + w_nx_n + b = 0$, где хотя бы один элемент $w_i \neq 0$. В этой формуле x — все возможные координаты, а w — вектор нормали гиперплоскости.

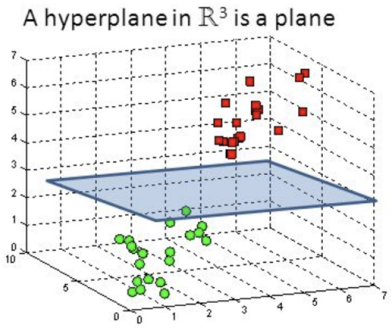
Более компактный вариант записи:

$(w, x) + b = 0,\ w = (w_1, \ldots, w_n)$.



Примеры гиперплоскостей:





Здесь уже неявно видна задача классификации, но о ней мы поговорим позже.

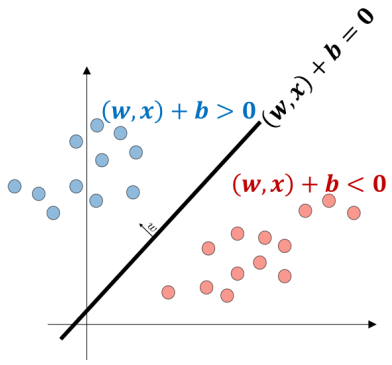
Вектор $w = (w_1, \ldots, w_n)$ определяет **нормаль (нормальный вектор)**, он ортогонален каждому вектору на гиперплоскости.

Посмотрим на пример из машинного обучения. Будем называть каждую точку объектом. Объекты = 2D векторы. Мы хотим разделить их на классы +1 и −1. Для этого нам нужно построить гиперплоскость, которая разделяет их: сверху — +1, снизу — −1.

Более формально:

- Объекты над гиперплоскостью: $(w, x) + b > 0$.

- Объекты под гиперплоскостью: $(w, x) + b < 0$.



Считаем скалярное произведение $(w, x) + b$. Если оно больше 0, класс сверху синего цвета; если меньше 0, то класс снизу красного цвета.

2. Матрицы. Основные операции над матрицами

Матрица — множество векторов, которые упорядочены внутри одной матрицы.

$A \in R^{m \times n}$ – матрица с m строками и n столбцами:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{2\times 2}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

Будем рассматривать только действительнозначные матрицы.

Особые матрицы:

Диагональная	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ $\left(a_{ii} \neq 0,\ a_{ij} = 0\ \forall i \neq j\right)$

Единичная	$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$ $(a_{ii} = 1,\ a_{ij} = 0\ \forall i \neq j)$
Симметричная	$\begin{bmatrix}1 & 4 & 5\\ 4 & 2 & 6\\ 5 & 6 & 3\end{bmatrix}$ $(a_{ij} = a_{ji})$
Верхнетреугольная и нижнетреугольная	$\begin{bmatrix}1 & 2 & 3\\ 0 & 4 & 5\\ 0 & 0 & 6\end{bmatrix}, \begin{bmatrix}1 & 0 & 0\\ 2 & 3 & 0\\ 4 & 5 & 6\end{bmatrix}$ $(a_{ij} = 0\ \forall i > j\ or\ \forall i < j)$

Операции над матрицами

Введем операции сложения и умножения на скаляр:

- Сложение:

  $A = \{a_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$, $B = \{b_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$, $A + B = \{a_{ij} + b_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$

- Умножение на скаляр:

  $A = \{a_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$, $\lambda \in R$, $\lambda A = \{\lambda a_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$

Матрицы и операции сложения и умножения матриц на скаляр задают векторное пространство. Так что в каком-то смысле матрицу можно рассматривать как вектор.

Введем теперь **матричное умножение**. Матричное умножение постоянно используется в машинном обучении. Все нейронные сети, GPU (графические процессоры) необходимы, потому что умеют быстро считать умножение матриц.

Линейная модель, которую мы разберем позже — это умножение матрицы объект-признак на матрицу весов. Нейронная сеть — это умножение матрицы

объект-признак на матрицу весов, только затем мы применяем активацию и повторяем многократно.

Итак, матричное умножение:

$A = \{a_{ij}\}_{i=1,\dots,m,\ j=1,\dots,n}$, $B = \{b_{ij}\}_{i=1,\dots,n,\ j=1,\dots,k}$

$$A \cdot B = \left\{\left(A_i, B^j\right)\right\}_{i=1,\dots,m,\ j=1,\dots,k} = \left\{\sum_{l=1,\dots,n} a_{il} \cdot b_{lj}\right\}_{i=1,\dots,m,\ j=1,\dots,k}$$

❗ Для умножения размерности матриц должны быть согласованы

Например $R^{2\times 2}$:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

❗ Матричное умножение не коммутативно в общем случае: AB ≠ BA

$$A = \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 6 & 2 \\ 3 & 2 \end{bmatrix}, \quad AB = \begin{bmatrix} 30 & 14 \\ 12 & 6 \end{bmatrix}, \quad BA = \begin{bmatrix} 20 & 28 \\ 11 & 16 \end{bmatrix}$$

Умножение коммутативно в следующих случаях:

- Умножение на единичную матрицу E:

  $AE = EA = A$

- Умножение на матрицу из нулей O:

  $AO = OA = O$

Поговорим о **транспонировании матриц**. Операция транспонирования меняет местами строки и столбцы матрицы:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}.$$

Обратим внимание на следующие свойства:

- A – симметричная матрица $\Rightarrow A^T = A$
- $\left(A^T\right)^T = A$

- $(A + B)^T = A^T + B^T$
- $(AB)^T = B^T A^T$

Линейные преобразования

Введем линейное преобразование, которое воздействует на вектор:

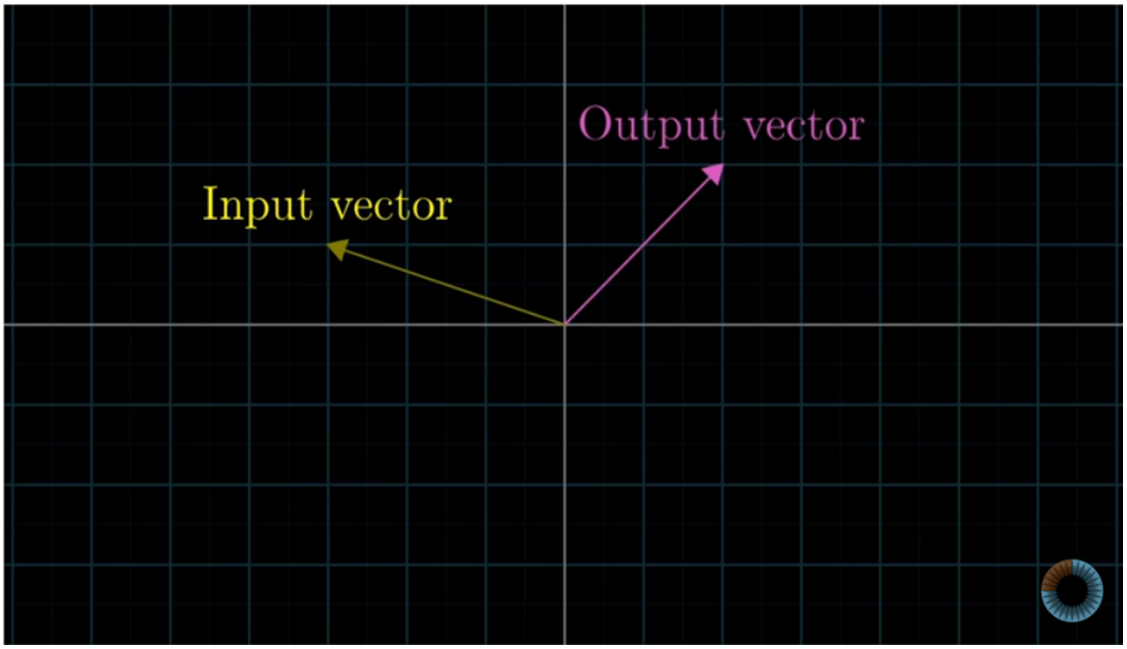
$x_{input} \rightarrow A \rightarrow x_{output}$

A – преобразование

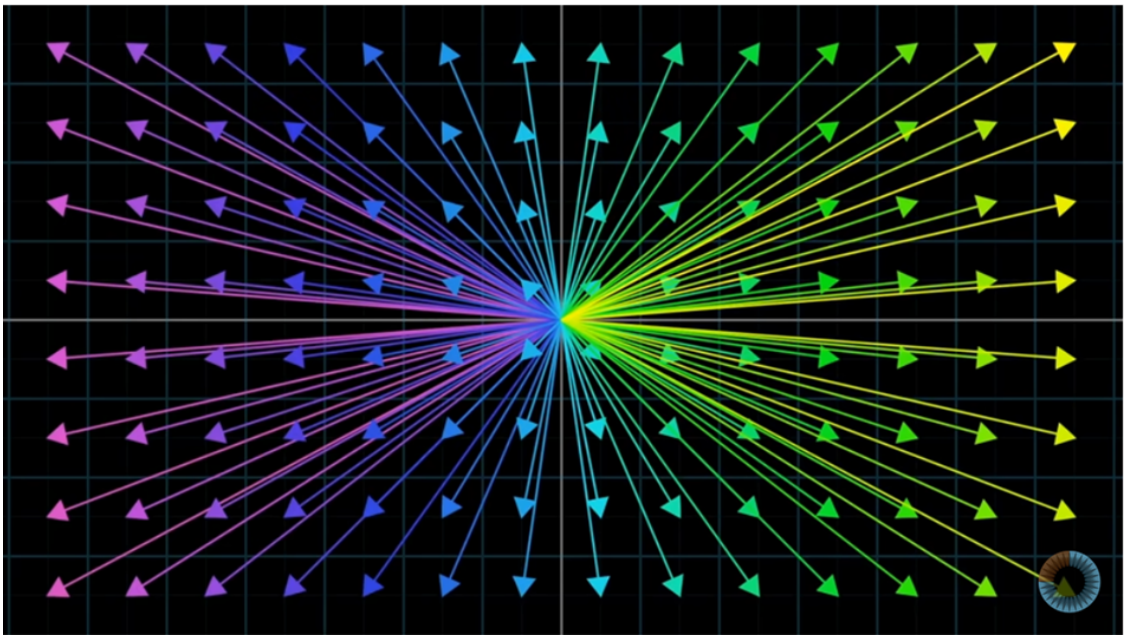
x_{input}, x_{output} – векторы

Пусть есть некий вектор. Мы воздействуем на него линейным преобразованием и получаем другой вектор в том же пространстве или в другом.

А что, если такое преобразование может называться матрицей? Как видите на рисунке, один вектор смотрит в одну сторону, другой в другую. Получается, мы его повернули, немного растянули, сжали.



Вы можете любой вектор, кроме нулевого, отобразить в любой из векторов на рисунке ниже и в любой между ними.



Свойства линейных преобразований:

1. $A(x + y) = A(x) + A(y)$
2. $A(\lambda x) = \lambda Ax$

Линейное преобразование можно произвести и при помощи матриц:

$x_{input} = x_1 e_1 + x_2 e_2 + \ldots + x_n e_n$, $e_1, \ldots e_n$ – базис, $x_1, \ldots, x_n$ – координаты

$$x_{output} = A\left(x_{input}\right) = A\left(x_1 e_1 + x_2 e_2 + \ldots + x_n e_n\right) = x_1 A\left(e_1\right) + x_2 A\left(e_2\right) + \ldots + x_n A\left(e_n\right)$$

$$A := \left[A\left(e_1\right) \mid A\left(e_2\right) \mid \ldots \mid A\left(e_n\right)\right]$$

$$\Longrightarrow x_{output} = A\left(x_{input}\right) = A \cdot x_{input}$$

Когда мы применяем линейное преобразование к вектору, мы по сути применяем его по отдельности к каждому из базисных векторов. Базисные векторы в ортогональном базисе ортогональны, поэтому они взаимодействуют независимо друг от друга. Итоговый результат — это преобразования над каждой из компонент по отдельности, совмещенные воедино. Таким образом получаем итоговый вектор.

Все линейные преобразования представляют собой поворот и растяжение-сжатие, а если меняем размерность — проекцию.

Рассмотрим пример линейного преобразования — поворот.

- Повернем векторы в R^2 на 90° против часовой стрелки.
- Базисные векторы повернутся:

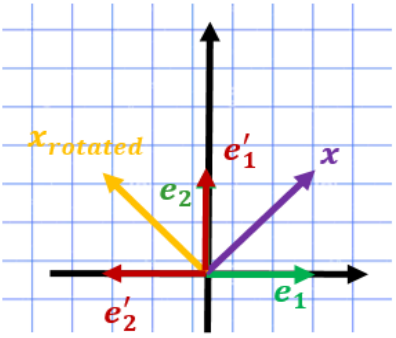
$$e_1 = \begin{bmatrix}1\\0\end{bmatrix} \to \begin{bmatrix}0\\1\end{bmatrix}, \qquad e_2 = \begin{bmatrix}0\\1\end{bmatrix} \to \begin{bmatrix}-1\\0\end{bmatrix}.$$

- Тогда матрица поворота:

$$R = \begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}.$$

- Повернем $x = [1,\ 1]^T$:

$$x_{rotated} = Rx = \begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}-1\\1\end{bmatrix}.$$



Обратите внимание:

1. Каждое линейное преобразование задается матрицей.
   Столбцы = представление базисных векторов после преобразования.
2. И наоборот: каждая квадратная матрица задает некоторое линейное преобразование.

3. Вычислительные функции библиотеки NumPy. Массивы

`NumPy` (Numeric Python) — одна из базовых библиотек Python, которая поддерживает линейно-алгебраические вычисления, в том числе работу с многомерными

массивами и соответствующими высокоуровневыми функциями. На основе `NumPy` работает множество других библиотек Python.

Одно из ключевых преимуществ `NumPy` — скорость работы. `NumPy` хорошо подходит для однотипных операций на очень большом объеме данных, например, при предобработке и «гармонизации» сырых данных. Это возможно благодаря **векторным вычислениям** — запараллеливанием однотипных операций вместо прохождения по большому циклу.

Допустим, дана последовательность чисел от 1 до 10000, где каждое число нужно возвести в квадрат. Зная основы Python, мы можем имплементировать этот алгоритм несколькими способами. Сравним время выполнения этой операции на примере встроенных списков и массивов `NumPy`.

Список + Циклы:

```
l = list(range(1,10001))
l1 = []
start1 = time.time()
for i in l:
 l1.append(i**2)
end1 = time.time() - start1
print(format(end1, '.8f'))
```

Получим время на операцию: 0,00376558 секунд.

Массивы `NumPy`:

```
arr = np.arange(1, 100001)
start2 = time.time()
arr = arr**2
end2 = time.time() - start2
print(format(end2, '.8f'))
```

Получим 0,00046968 секунд.

Разница во времени вычислений почти в 8 раз:

```
print(format((end1/end2), '.8f'))

>>> 8.01725888.
```

Массивы

В программировании под **массивом** мы понимаем структуру данных, которая хранит упорядоченный набор элементов одного типа. Любой упорядоченный набор данных поддерживает целочисленное индексирование.

Элементом в массиве `NumPy` (`numpy.ndarray`) может быть только число.

Прежде чем начать работать с библиотекой, необходимо ее установить:

```
!pip install numpy
```

После этого достаточно импортировать библиотеку (`np` — общепринятое сокращение названия):

```
import numpy as np
```

Самый простой способ создания массива — применение метода `np.array()` к списку.

Например, создадим одномерный массив из чисел 1, 2 и 3:

```
np.array([1,2,3])
```

Многомерный массив можно создать из вложенных списков. По-другому он называется **матрицей**. Например, создание двумерного массива (матрицы) размерности 2x3, где 2 — количество строк, 3 — количество столбцов:

```
np.array([[1, 2, 3], [4, 5.0, 6]])
```

Атрибуты массива

Важные атрибуты объектов `ndarray`:

- `ndarray.ndim` — число измерений, для строки — это 1, для матрицы — 2.
- `ndarray.shape` — размеры массива, его форма. Размер каждого из осей многомерного массива.
- `ndarray.size` — количество элементов массива.
- `ndarray.dtype` — объект, описывающий тип элементов массива.

В качестве примера создадим матрицу, состоящую из двух строк и трех столбцов:

```
a = np.array([[1, 2, 3], [4, 5.0, 6]])
```

Далее последовательно выведем каждый из атрибутов созданного объекта:

```
print(a.ndim)
print(a.shape)
print(a.size)
```

```
print(a.dtype)
```

Получим:

```
>> 2
>> (2, 3)
>> 6
>> float64
```

Способы создания массивов

Способы создания массива по какому-то простому закону:

- `np.arrange(начало, конец, шаг)` — генерация последовательности на интервале, аналогично функции `range()`.
- `np.linspace(начало, конец, число_элементов)` — генерация последовательности на интервале, вместо шага указываем, сколько элементов массива нужно создать (конец включается).

Сравним использование функций на примере:

```
a = np.arange(0,10,2)
print(a)
b = np.linspace(0,10,5)
print(b)
```

При использовании функции `np.arange` получим:

```
>>> [0 2 4 6 8]
```

Как видим, 10 не включена, верхняя часть интервала выбрасывается. При использовании функции `np.linspace` число 10 будет включено, расстояния между элементами одинаковые:

```
>>>> [ 0.   2.5  5.   7.5   10. ]
```

Частные способы создания массивов:

- `np.zeros()` — создание массива из нулей.
- `np.ones()` — создание массива из единиц.
- `np.eye()` — создание единичной матрицы. Это квадратная матрица, заполненная нулями, кроме главной диагонали — от левого верхнего до правого нижнего элемента, которая заполнена единицами.

Индексирование массивов. Операции с массивами

Система индексирования одномерных массивов аналогична спискам. Необходимо сослаться на массив и в квадратных скобках указать индекс или серию чисел — начало, конец и шаг:

```
your_array[начало:конец:шаг]
```

Индексирование многомерных массивов отличается лишь количеством аргументов внутри квадратных скобок. Например, дан двумерный массив — матрица из 6 чисел:

```
a = np.array([[1, 2, 3], [4, 5.0, 6]])
```

Чтобы сослаться на элемент первой строки первого столбца («1»), сошлемся на массив и на каждое измерение по очереди:

```
a[0,0]
```

Аналогично спискам мы можем заменять значения в массивах, присваивая элементу новое значение по индексу. Например, в известной нам матрице `a — [[1, 2, 3], [4, 5.0, 6]]` заменить элемент первой строки первого столбца можно так:

```
a[0][0] = 1.5
```

Получим матрицу `[1.5, 2, 3], [4, 5.0, 6].`

В многомерных случаях можем добавлять целые строки и столбцы в массив. Необходимо, чтобы вектор-заменитель по размеру совпадал с вектором-заменой.

Например, пусть дана двумерная матрица: `a [[1, 2, 3], [4, 5.0, 6]].`

Замена первого столбца:

```
a[:,0] = [0,1]
```

Результат: `[[0. 2. 3.] [1. 5. 6.]].`

Замену первой строки `([0,2,3])` можно сделать аналогично — `a[0,:]`, и заменить ее на список из трех элементов.

Удалить элементы из массива можно с помощью метода `np.delete(объект, индекс(ы), ось)`:

- объект — массив;

- индекс(ы) — индекс или список индексов;
- ось — по умолчанию `axis = 0`.

Другой способ — сделать срез и пересохранить массив. Например, дан массив `[1, 2, 3, 0]`, из которого нужно удалить последний элемент. Можем сделать это двумя способами.

Первый способ:

```
a = np.array([1,2,3,0])
a1 = np.delete(a, 3)
print(a1)
```

Второй способ — используем срез:

```
a2 = a[0:3]
print(a2)
```

В обоих случаях получим:

```
>>> [1,2,3]
>>> [1,2,3]
```

Метод `np.reshape()` позволяет изменить размерность массива. Допустим, дан одномерный массив (вектор) `[1, 2, 3, 4, 5, 6]`, который необходимо конвертировать (можно конвертировать в матрицу 3x2). Это можно сделать, применив метод `reshape()` с параметрами `(3, 2)`:

```
np.array([1, 2, 3, 4, 5, 6]).reshape(3,2)
```

Результат:

```
>>> [[[1 5]
     [3 7]]

     [[2 6]
     [4 8]]]
```

Транспонирование

В линейной алгебре транспонирование играет важную роль. На транспонировании основано большое количество оптимизационных алгоритмов анализа данных.

В `NumPy` можно транспонировать массив любой размерности с помощью метода `np.transpose()`:

- В одномерном случае ничего не изменится.
- В двумерном произойдет классическое транспонирование (смены индексов строк и столбцов для каждого элемента).
- В многомерном случае произойдет смена индексов между измерениями поочередно.

4. Векторы. Решение линейных уравнений

Понятие вектора

Необходимо различать два понимания вектора:

- В широком смысле — это любая последовательность однородных элементов, в том числе и чисел.
- В линейной алгебре вектор — направленный отрезок (в программировании реализуется).

Таким образом, массив в `NumPy` — это вектор в широком смысле.

Операции с векторами

Массивами `NumPy` поддерживаются арифметические операции основного Python:

- сложение (+);
- вычитание (—);
- нахождение равенства (=);
- деление (/);
- целочисленное деление (//);
- возведение в степень (**) и т. д.

Можно делать операции с массивом и отдельным числом. Рассмотрим пример проведения операции над векторами и массивами в `NumPy`.

Пусть дан вектор `x = [0 1 2 3]`. Прибавив число 5, мы прибавим это число к каждому элементу массива `x + 5 = [5 6 7 8]`.

Идентичные операции поддерживаются между массивами одинаковой размерности. Например мы возьмем векторы `x = [0 1 2 3]` и `y = [4 5 6 7]`. В результате сложения произойдет суммирование элементов, имеющих одинаковые индексы: `x + y = [4 6 8 10]`

Важно! Существуют специальные методы `NumPy`, поддерживающие векторные операции. Они работают быстрее, чем арифметические операции.

Метод	**Описание**
`add(x,y)`	Сложение
`substract(x,y)`	Вычитание
`negative(x)`	Негация
`multiply(x,y)`	Умножение
`divide(x,y)`	Деление
`floor_divide(x,y)`	Целочисленное деление
`power(x,y)`	Степень

Функции массивов

Метод	**Описание**
`sum`	Сумма всех элементов массива (или сумма всех элементов по одному из измерений)
`mean`	Среднее арифметическое
`std, var`	Стандартное отклонение, дисперсия
`min, max`	Минимум и максимум
`argmin, argmax`	Индексы минимального и максимального элемента
`cumsum`	Кумулятивная сумма элементов
`cumprod`	Кумулятивное произведение элементов

Важно выделить ряд методов, которые во многом объединяют массивы с множествами:

Метод	Описание
`unique()`	Подсчет отсортированных уникальных элементов в массиве
`intersect1d(x, y)`	Подсчет отсортированных общих элементов в двух массивах (пересечение)
`union1d(x, y)`	Объединение массивов-множеств
`setdiff1d(x, y)`	Асимметричная разница массивов (показывает все элементы первого массива, которых нет во втором массиве)
`setxor1d(x, y)`	Симметричная разница массивов (все элементы, которые есть только в одном из массивов)

Нормализация

Часто для упрощения работы многих алгоритмов анализа данных и/или для упрощения интерпретации оценок этих методов используется **нормализация переменных** — приведение вектора чисел к стандартизированному виду.

Допустим, дан вектор с последовательностью роста нескольких человек:

```
heights = np.array([187, 188, 190, 177, 156, 189])
```

Нужно использовать эти данные в относительном виде (важны не абсолютные значения, а вариация).

Нормализацию от 0 до 1 в `NumPy` можно осуществить следующим образом:

```
print((heights- np.min(heights)) / (np.max(heights) -
np.min(heights)))
```

Результат: `[0.91176471, 0.94117647, 1.0, 0.61764706, 0.97058824]`

Некоторые операции с векторами

Рассмотрим некоторые операции с векторами.

`np.linalg.norm(x)` — поиск нормы векторов.

Расстояние между векторами в `NumPy` можно посчитать как нормы разности векторов:

- `np.linalg.norm(x-y)`;
- `np.dot()` считает скалярное произведение двух векторов.

Например, в экономике часто можно наблюдать скалярное произведение вектора цен на вектор количества проданных товаров. Скалярное произведение в этом случае дает стоимость всех проданных товаров.

Пусть у нас есть 4 вида транзакций:

	1	**2**	**3**	**4**
Кол-во (`q`)	100	150	200	350
Цена (`p`)	5	2.5	3	0.5

Видим, что есть вектор проданных товаров `q = (100; 150; 200; 350)` и вектор цен `p = (5; 2.5; 3; 0.5)`. Сохраним эти вектора в виде массивов `NumPy`:

```
q = np.array([100, 150, 200,350])
p = np.array([5, 2.5, 3, 0.5])
np.dot(q, p)
```

В результате мы получим число 17650. Это итоговая стоимость всех товаров.

Решение системы линейных уравнений

Рассмотрим метод `np.linalg.solve()` для решения системы линейных уравнений. Задача оптимизации нескольких систем различных линейных и нелинейных уравнений тоже достаточно часто встречается в прикладных алгоритмах анализа данных.

Пусть задана система линейных уравнений:

$$20x + 10y = 350$$

$$17x + 22y = 500$$

Нужно представить ее в виде массивов:

```
A = np.array([[20, 10], [17, 22]])
B = np.array([350, 500])
```

Вектор `A` — вложенный список с коэффициентами при `x` и `y` из первого и второго уравнений соответственно.

Далее применим метод `solve()`:

```
X = np.linalg.solve(A,B)
print(X)
```

Результат:

```
>>> [10. 15.]
```

5. Использование NumPy в задачах обработки данных. Генерация мелодии

Рассмотрим работу с библиотекой `NumPy` применительно к аудиоданным. Выполним генерацию мелодии по заранее заданным нотам. Звучание нот реализуем по таблице частот.

Для реализации задачи выполним импорт библиотеки `NumPy` и библиотеки для озвучки аудио `IPython`. Чтобы создать звуковой ряд, необходима частота дискретизации и длительность звучания отдельной ноты. Длительность звучания равна одной секунде: `time = 1.0`. Частота дискретизации будет `samplerate = 44100` Гц. При этом **частота дискретизации** — это количество значений сигнала, записываемых в секунду.

Представим частоты в виде массива:

```
frequency = np.array([261.63, 293.66, 329.63, 349.23, 392, 440, 493])
```

Частоты соответствуют первой октаве от 261 до 493 Гц.

Полный текст программы:

```
import numpy as np
# библиотека для озвучки аудио
from IPython.display import Audio

#частота дискретизации (Гц)
samplerate = 44100

time = 1.0

# частоты нот (Гц) первой октавы: до, ре, ми, фа, соль, ля, си
```

```
frequency = np.array([261.63, 293.66, 329.63, 349.23, 392, 440,
493])

N_notes = frequency.size

N_notes
```

В результате программа выведет количество нот: 7.

Далее создадим мелодию с одинаковыми длительностями нот:

```
wave_sum = np.array([])

# мелодия с одинаковыми длительностями

for i in range(0,N_notes):
  # Генерация значений между 0 и 1 секундой
    samples = np.arange(44100 * time) / 44100.0
  # Синусоидальная функция частоты сигнала w(t) = A*sin(2*pi*f*t)
    wave = 10000 * np.sin(2 * np.pi * frequency[i] * samples)
  # конвертация в  wav формат (16 бит) и складывание сигналов
    wave_wave = np.array(wave, dtype=np.int16)
    wave_sum = np.append(wave_sum, wave_wave)

Audio(wave_sum, rate=samplerate)
```

В представленной программе сначала мы генерируем в цикле значения между 0 и 1, реализуем синусоидальную функцию частоты сигнала. Видно, что библиотека `NumPy` имеет собственную функцию для расчета синуса. После указанных действий результат синусоидальной функции мы конвертируем в wav-формат. В конце каждой итерации цикла мы складываем полученные звуковые ряды. После завершения цикла с помощью команды `Audio` у нас есть возможность воспроизвести полученный звуковой ряд.

Спектрограмма — графическое представление спектра сигнала в зависимости от времени. Спектрограмма может быть полезной для анализа звуковых сигналов, таких как речь, музыка или звуки окружающей среды. Также она может помочь выявить особенности изменения сигнала технических систем от времени.

Рассмотрим программу для построения спектрограммы:

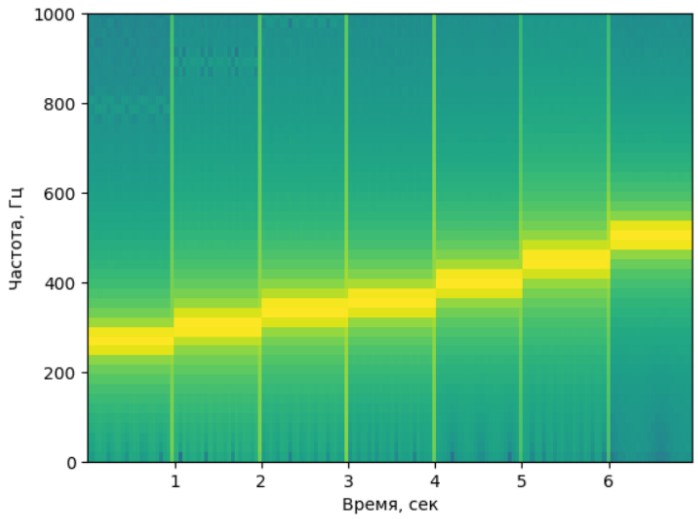
```
# Построение спектрограммы
```

```
# NFFT - количество точек для вычисления быстрого преобразования
Фурье при вычислении спектрограммы
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()
ax1.specgram(wave_sum,Fs = samplerate, NFFT = 2048)
plt.ylim([0, 1000])
plt.xlabel('Время, сек')
plt.ylabel('Частота, Гц')
plt.show()
```

Для построения спектрограммы мы будем использовать библиотеку `matplotlib`. С помощью команды specgram построим спектрограмму, при этом в качестве исходных данных используем `NFFT = 2048` — окно, которое позволяет вычислять быстрое преобразование Фурье.

Результат:



Поскольку в программе было 7 нот, из спектрограммы мы можем видеть, что частота увеличивается. В качестве эксперимента можно изменить окно и отследить его влияние на спектрограмму.

6. Работа с табличными данными и векторами

Разберем пример работы с табличными данными с помощью библиотеки `NumPy`. Дополнительно мы будем использовать библиотеку `Pandas` для загрузки данных из csv-файла:

```
import pandas as pd
data = pd.read_csv("temperature_russian_cities.csv")
data
```

Видно, что в таблице содержится информация о городах: широта, долгота и различные виды температур:

	City	Latitude	Longitude	The absolute minimum, °C	The temperature of the coldest month, °C	The average annual temperature, °C	The temperature of the warmest month, °C	The absolute maximum, °C	The average annual precipitation, mm
0	Абакан	53.717564	91.429317	-38.0	-17.5	3	25.0	45.0	323
1	Алдан	58.605723	125.396100	-48.7	-26.3	-5.5	16.6	35.2	718
2	Александров Гай	50.143771	48.549450	-39.9	-8.7	7.4	24.2	43.8	329
3	Александровск-Сахалинский	50.907923	142.174636	-41.0	-16.3	0.9	16.6	32.6	679
4	Анадырь	64.733661	177.496826	-46.8	-22.6	-6.9	11.6	30.0	384

При загрузке данных мы можем иметь дело не с числовыми данными, поэтому переведем данные столбца The temperature of the warmest month, °C в числовые:

```
temperatures = pd.to_numeric(data['The temperature of the warmest month, °C'])
```

Далее конвертируем формат `Pandas` в `NumPy`:

```
temperatures_numpy=temperatures.to_numpy()
temperatures_numpy
```

В результате получим массив температур:

```
array([25. , 16.6, 24.2, 16.6, 11.6, 23.2, 16.3, 25.6, 13.7, 18.4,  6. ,
       19.9, 21.8, 17.4, 21.1, 21.8, 19.1, 18.2, 19.4, 19. , 17.8, 18. ,
       17.3, 16.5, 18.7, 19.8, 20.6, 18. , 26. , 17.5, 13.1, 20.5, 18.3,
       18.9, 23.9, 26. ,  4.8, 13.8, 24.3, 19. , 19.3, 20. , 17.5, 18.1,
       19. , 18.2, 11.7, 20.2, 15.7, 18.1, 23.7, 14.6,  9. , 19. , 18.1,
       18.3, 18.9, 15.6, 16.3, 18.7, 17.6, 24. , 18.7, 19.8, 19.6, 20.4,
       12. , 19.1,  7.1, 24.7, 22.7, 19.2, 12.9, 13.5, 20.6, 16.1, 17.9,
       19.4, 16.5, 18.8, 19.2, 14.3, 17.4, 14.9, 18.4, 19.6, 16.9, 22.3,
       19.4, 13.5,  8.7, 19.9, 18.7, 17. , 12.5, 16.1, 18.3, 23.3, 18.8,
       19.2, 14.8, 21.4, 18.8, 19.3, 22.4, 22.4, 11.5, 11.1, 22.3, 17.9,
       23.6, 22. , 18.2, 17.5, 20.4, 18.8,  7.7, 18.7, 18.7, 19.4, 16.9,
       18.8, 19.8, 20.2, 19.6, 16.5, 21.3, 18.4, 12.4, 18.7, 16.5, 18.8,
       19.3, 17.6, 18.7, 10.6, 17.5, 24.9, 15.9, 17.3, 28.6, 19.5, 24.1,
       30.6])
```

Можно вычислить среднее значение температуры по городам:

```
np.mean(temperatures_numpy)
```

В результате получим: 18.30625.

Нормализация и стандартизация

Чтобы подготовить данные, для машинного обучения используются операции нормализации и стандартизации переменных.

Нормализация — это изменение масштаба данных из исходного диапазона, чтобы все значения находились в диапазоне от 0 до 1. Нормализация может быть полезной в некоторых алгоритмах машинного обучения, когда данные временных рядов имеют входные значения с различными масштабами. Это может потребоваться для алгоритмов, таких как k-Nearest соседей, которые используют вычисления расстояний, линейную регрессию и искусственные нейронные сети.

Пример реализации нормализации данных:

```
print((temperatures_numpy- np.min(temperatures_numpy)) /
(np.max(temperatures_numpy) - np.min(temperatures_numpy)))
```

Стандартизация набора данных включает изменение масштаба распределения значений так, чтобы среднее значение наблюдаемых значений было 0, а стандартное отклонение –1. Это можно рассматривать как вычитание среднего значения или центрирование данных.

Как и нормализация, стандартизация может быть полезной в некоторых алгоритмах машинного обучения, таких как машины опорных векторов, линейная и логистическая регрессия. Стандартизация предполагает, что ваши наблюдения соответствуют гауссовскому распределению со средним значением и стандартным отклонением.

Пример реализации стандартизации данных:

```
print((temperatures_numpy- np.min(temperatures_numpy)) /
(np.std(temperatures_numpy)))
```

Операции с векторами

Рассмотрим функции, реализующие операции с векторами:

- `np.linalg.norm(x)` — поиск нормы векторов;

- `np.linalg.norm(x-y)` — расстояние между векторами;
- `np.dot()` — скалярное произведение двух векторов;
- `np.linalg.solve()` — решения системы линейных уравнений;
- `np.linalg.lstsq()` — метод наименьших квадратов.

Выделим широту и долготу городов из таблицы и переведем их в числовой вид:

```
# широта городов
Latitude = pd.to_numeric(data['Latitude']).to_numpy()

# долгота городов
Longitude = pd.to_numeric(data['Longitude']).to_numpy()
```

Рассчитаем прямое расстояние между Новосибирском и остальными городами:

```
data[data['City']=='Новосибирск']
```

Из данных видим:

- широта Новосибирска — 55.008353 град;
- долгота Новосибирска — 82.935733 град.

Нам интересно вычисление расстояний от Новосибирска до других городов России. Такая задача может быть полезна при создании признакового пространства для анализа географических координат:

```
import math

R = 6373.0 # Расстояние до центра Земли

#Координаты Новосибирска переведенные в радианы
lat1 = 55.008353*np.pi/180
lon1 = 82.935733*np.pi/180

lon2 = Longitude*np.pi/180
lat2 = Latitude*np.pi/180

dlon = lon2 - lon1
dlat = lat2 - lat1
```

```
a = (np.sin(dlat/2))**2 + np.cos(lat1) * np.cos(lat2) *
(np.sin(dlon/2))**2
c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
distance_Novosibirsk = R * c
#расстояние от Новосибирска до городов России
distance_Novosibirsk
```

Мы можем определить максимальное расстояние:

```
np.max(distance_Novosibirsk)
```

Для сравнения вычислим максимальное геодезическое расстояние — расстояние, измеренное вдоль поверхности Земли с помощью специальной библиотеки, учитывающей несферичность земли. Для этого подгрузим библиотеку `geopy.distance`:

```
import geopy.distance

coords_1 = (lat1*180/np.pi, lon1*180/np.pi)
coords_2 = (lat2, lon2)

max_dist = 0
for i in range(1,len(lat2)):
    geodesic_distance = geopy.distance.geodesic(coords_1,
(lat2[i]*180/np.pi,lon2[i]*180/np.pi)).km
    if geodesic_distance > max_dist:
      max_dist = geodesic_distance
max_dist
```

Можно сравнить данные и увидеть, что они достаточно близки. В первом случае мы получили 4875,2 км; во втором — 4890,9 км.

Выполним аналогичный расчет расстояний всех городов от Москвы и определим максимальное расстояние.

Посчитаем норму для расстояний от Новосибирска и Москвы. Норма может быть рассчитана как корень квадратный из сумм квадратов — евклидово расстояние многомерного пространства:

```
np.linalg.norm(distance_Novosibirsk)
np.linalg.norm(distance_Moscow)
```

Также можно рассчитать норму расстояний между векторами вычисленных расстояний Новосибирска и Москвы:

```
#Норма разницы векторов Москвы и Новосибирска
np.linalg.norm(distance_Moscow - distance_Novosibirsk)
#Норма разницы векторов Новосибирска и Москвы
np.linalg.norm(distance_Novosibirsk- distance_Moscow)
```

В обоих случаях мы получим одинаковое значение, так как в формуле используется квадрат.

Метод наименьших квадратов

Теперь попробуем решить задачу прогнозирования температуры в самый теплый месяц в зависимости от широты. Для этого используем метод наименьших квадратов. Сначала выделим необходимые данные:

```
data[['Latitude','The temperature of the warmest month, °C']]
```

Переведем данные в числовой формат:

```
latitude_warm = pd.to_numeric(data['Latitude']).to_numpy()
temperatures_warm = pd.to_numeric(data['The temperature of the
warmest month, °C']).to_numpy()
x = latitude_warm
y = temperatures_warm
```

Мы можем переписать линейное уравнение как y = Ap, где A = [[x 1]] и p = [[m], [c]]. Коэффициенты m и c — это коэффициенты прямой.

Теперь используем `lstsq` для оценки `p`. Выполняем транспонирование:

```
A = np.vstack([x, np.ones(len(x))]).T
```

Реализуем процедуру метода наименьших квадратов:

```
m, c = np.linalg.lstsq(A, y, rcond=None)[0]
m, c
```

Таким образом, получаем коэффициенты `m` и `c`:

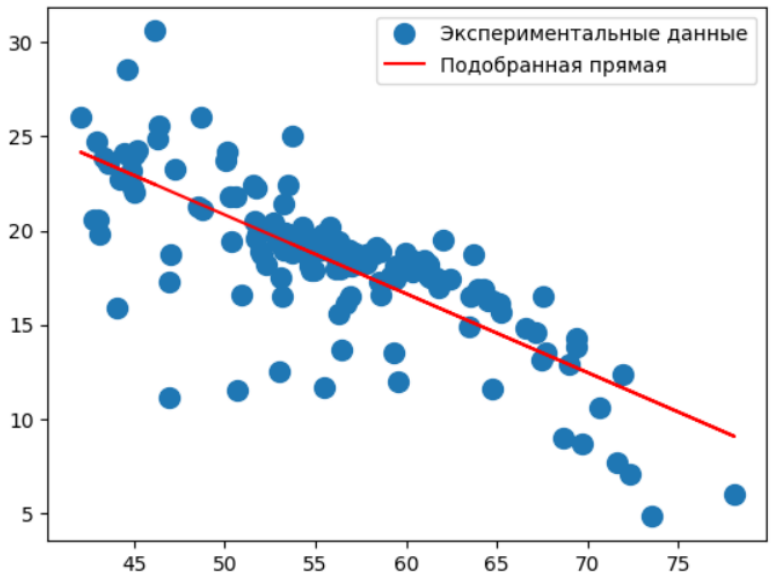
```
(-0.41926549657500517, 41.80544064668504)
```

Теперь покажем на графике, насколько данные таблицы близки к полученной прямой с учетом вычисленных коэффициентов:

```
import matplotlib.pyplot as plt
```

```
_ = plt.plot(x, y, 'o', label='Экспериментальные данные',
markersize=10)
_ = plt.plot(x, m*x + c, 'r', label='Подобранная прямая')
_ = plt.legend()
plt.show()
```

Полученный график:



Из графика мы видим, что прямая так или иначе прогнозирует температуру по широте. Более сложный пример позволит нам реализовать не только широту, но и долготу, а также учесть другие признаки.

7. Задача снижения размерности

В задаче снижения размерности мы говорим про снижение размерности признакового пространства, в котором находятся все объекты, с которыми мы будем работать.

Рассмотрим некоторую задачу, в которой есть многомерная выборка. В ней сотни или тысячи признаков данных. Это вызывает несколько проблем, так как далеко не все модели позволяют нам качественно работать с такими данными.

Какие могут быть проблемы:

- Эти данные **сложно визуализировать**. Мы не сможем адекватно визуализировать 1000-мерное пространство, не имея специальной математической подготовки.

- При работе с высокоразмерными данными модели могут **медленно обучаться**. Это может быть не только долго, но и дорого — как на этапе обучения, так и на этапе инференса.
- Некоторые модели хуже работают в случае обработки многомерных данных.

NB. **Проклятие размерности**. Почему невыгодно продавать бесконечномерные арбузы? Представим себе, что у нас есть некоторое пространство, которое равномерно заполнено точками.

Начнем с двумерного пространства. У нас есть сетка, и на пересечении линий находятся точки. Что такое арбуз? Есть центр, вокруг него мякоть (арбуз считаем шаром), на границе с ненулевой толщиной есть корка. Площадь корки и площади мякоти (в двумерном случае) друг с другом как-то соотносятся.

В трехмерном случае это уже объем мякоти и объем корки. Соотношение поменяется. При увеличении размерности отношение будет меняться. С какого-то момента обнаружится, что почти весь объем фокусируется в корке (можно взять предел от интеграла — объема корки). Поэтому бесконечномерные арбузы продавать невыгодно, никто не будет покупать только корку, когда мякоти бесконечно мало.

Вернемся к миру машинного обучения. Если у нас в пространстве точки распределены равномерно, то в случае большой размерности, эти точки будут находиться примерно на одинаковом удалении от центра. А значит, с помощью метрических алгоритмов мы не сможем их отличить.

Большая размерность зачастую сложна вычислительно и с точки зрения математики, с ней перестают работать некоторые методы.

8. Метод главных компонент

Понижение размерности

Данных может быть много. Признаковое пространство может быть очень высокой размерности. Мы можем обратиться к линейным методам для снижения размерности.

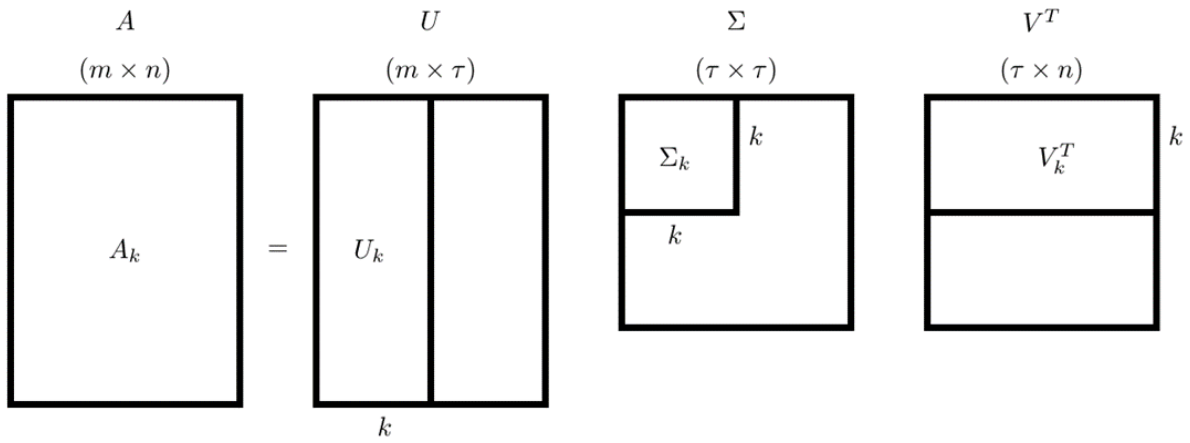
Посмотрим, что можно сделать с матрицей A (матрица, описываемая нашими признаками) для снижения размерности признакового пространства. Ее можно

разложить на несколько матриц. Матричные операции все линейные. И можно попытаться использовать матрицу меньшего ранга для описания данных.

Мы хотим матрицу A разложить на три матрицы:

$$A = U\Sigma V^T$$

$$A_k = U_k\Sigma_k V_k^T = \left(U_k\Sigma_k\right)V_k^T = U_k\left(\Sigma_k V_k^T\right)$$



Дополнительные ограничения:

$$U \in R^{m\times r};\ UU^T = I$$

$$V \in R^{n\times r};\ VV^T = I$$

$$\Sigma = diag(\sigma_1, \dots, \sigma_r);\ r = rank(M)$$

Матрицы U и V ортогональные, то есть обратные для самих себя в случае транспонирования. Из этого следует, что столбцы у них линейно независимы, норма у матриц U и V равна единице.

Если матрица обладает линейно независимыми столбцами, с единичной нормой, то это матрица поворота. То есть поворачивает наше признаковое пространство некоторым образом.

Итак, U и V — матрицы поворота.

Σ — диагональная матрица, на ее диагонали стоят некоторые числа. При умножении на такую матрицу мы будем растягивать или сжимать различные направления в

нашем пространстве. Если U и V — ортогональные матрицы, то каждому вектору соответствует лишь один диагональный элемент из матрицы Σ.

То есть мы разбиваем любое преобразование на поворот (матрицы U и V) и сжатие/растяжение (матрица Σ). В этом случае мы можем попытаться выбрать для себя оптимальные направления, вдоль которых у нас максимальная информация. Например, вдоль этих направлений максимальная дисперсия. После чего можно использовать эти направления с наибольшей дисперсией, а все остальные игнорировать.

Теперь мы можем понять, вдоль какого направления данные максимально разнородные, больше всего не похожи друг на друга. То есть лучше всего сможем различать точки между собой. При проекции на данное направление точки будут реже всего попадать в одну и ту же координату.

Поскольку вектора матриц U и V линейно независимы, то дисперсию мы можем измерять с помощью соответствующей σ. Более того, мы можем упорядочить σ по невозрастанию. То есть $\sigma_1 \geq \sigma_2$, $\sigma_2 \geq \sigma_3$ и т. д.

Если мы хотим получить k-мерное представление наших данных, достаточно взять первые k векторов матрицы U и соответствующие для них первые k сингулярных значений из матрицы Σ и домножить на первые k векторов матрицы V. Тогда мы получим некоторое приближение исходной матрицы A, теперь ранга k, и при этом оно будет оптимальным с точки зрения минимизации дисперсии.

Теорема (Экарта-Янга). Сингулярное разложение (SVD) дает нам оптимальное представление матрицы высокого ранга матрицей более низкого ранга k:

$A_k = U_k \Sigma_k V_k^T$

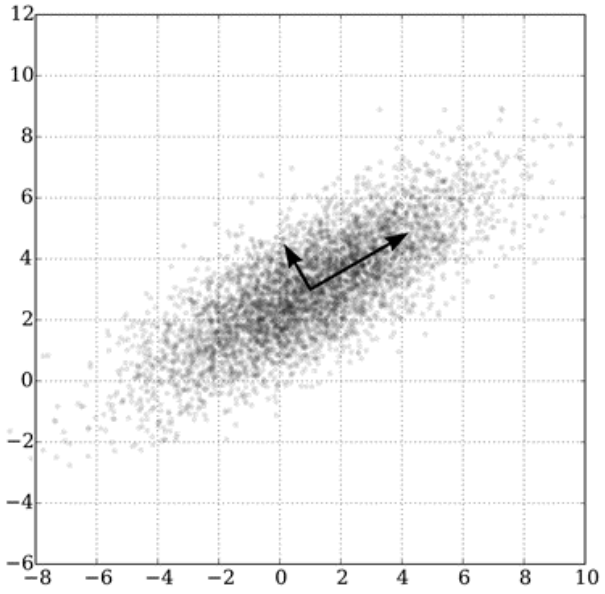
$\forall B_k: \; rank(B_k) = k$

$||A - B_k||_F \geq ||A - A_k||_F$ — ошибка по норме Фробениуса.

Норма Фробениуса — сумма квадратов разности элементов матриц.

PCA

Разберем, что это дает для машинного обучения. Допустим, у нас есть некоторые данные:



Данные обладают некоторой структурой в пространстве, в нашем случае это эллипс. Направление наибольшей дисперсии — направление главной (большой) полуоси. Вторая главная компонента направлена вдоль малой полуоси.

Что делать в случае, когда главные компоненты определены неоднозначно? Например, в случае центральной симметрии. Тут можно выбрать любое направление, как первую главную компоненту, а вторая будет ей ортогональна.

Метод главных компонент (PCA) по факту просто ищет направление наибольшей дисперсии последовательно.

Глядя на разложение

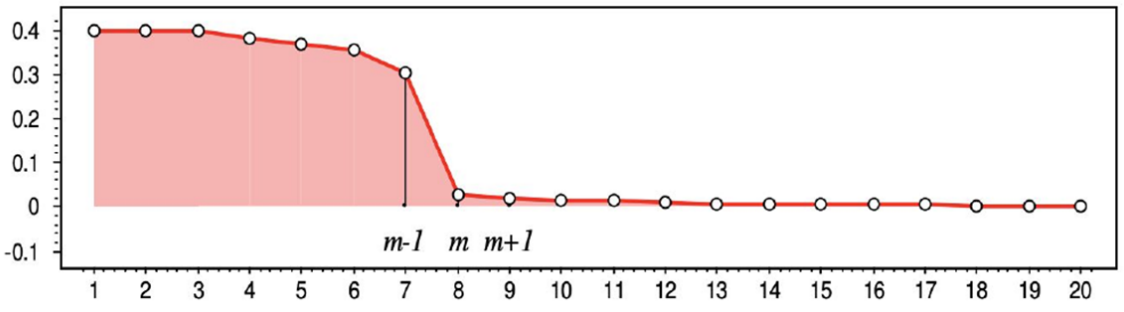
$X = U\Sigma V^T$,

можно понять:

- что матрица V указывает нам на оптимальные направления;
- $U\Sigma$ — новые главные компоненты, которые описывают наши данные.

Если мы хотим с помощью k-мерного представления выделить наиболее информативное k-мерное подпространство из нашего признакового пространства, нужно выбрать k компонент с наибольшей дисперсией, а все остальные занулить.

Как выбрать оптимальное k. Если данных много, то можно построить PCA по всем компонентам. Тогда мы уловим всю дисперсию. А потом можно посмотреть на матрицу Σ.



Мы увидим, что элементы матрицы Σ могут убывать некоторым образом. Здесь получается, что данные находятся в m-мерном подпространстве, которое содержит почти всю информацию о наших данных. Небольшие потери — цена за то, чтобы снизить размерность.

Из каких соображений выбирают число m.

$$E_m = \frac{||GU^T - F||^2}{||F||^2} = \frac{\lambda_{m+1} + \ldots + \lambda_n}{\lambda_1 + \ldots + \lambda_n} \leq \varepsilon$$

Смотрят, сколько элементов нужно отбросить, чтобы ошибка все еще не превышала заданное значение. Часто это называют **методом складного ножа**.

Таким образом строят PCA на все компоненты и смотрят, сколько компонент можно отбросить, чтобы их суммарный вклад в дисперсию был меньше, чем допустимая ошибка. Допустимую ошибку выбираем сами.

Применение на практике PCA

Рассмотрим применение PCA на практике.

- PCA зависит от того, в каких шкалах наши данные.

  Важно! Отнормируйте и отцентрируйте ваши данные.

- PCA позволяет снизить размерность, и мы уже не получаем исходное признаковое представление точек. Мы получаем представление уже в новом признаковом пространстве.

$X_k = U_k \Sigma_k$. Каждые k признаков — линейная комбинация исходных (поворот, сжатие/растяжение).

- Для возвращения в исходное пространство понадобится матрица V:

  $\overline{X} = U_k \Sigma_k V_k^T$. Но исходные данные будут восстановлены с некоторой ошибкой.

Пример 1. Есть множество точек. Каждая точка — векторное представление слова, которое из 300-мерной размерности было приведено к 2-мерной картинке:

Источник: github.com

Все эти точки были построены на основании контекстной близости слов.

Пример 2. PCA позволит снизить размерность данных, даже если это картинки. Каждая черно-белая картинка — это матрица.

Источник: towardsdatascience.com

С помощью SVD снизим размерность матрицы. Берем первые 16 компонент:

Если взять 50 компонент:

Если увеличить количество компонент до 250:

Таким образом, PCA — замечательный метод, который позволяет снизить размерность данных. И при этом он линейный и чувствителен к шкалам наших данных (так же, как и метод ближайших соседей, как и L1 и L2 регуляризации).

Дополнительные материалы для самостоятельного изучения

1. NumPy documentation — NumPy v1.24 Manual
2. Audio and Digital Signal Processing(DSP) in Python (pythonforengineers.com)
3. MP3 to Numpy and back | Kaggle