

Vector space

vector : 我们将把向量视为实数的有序元组 $[x_1, x_2, \dots, x_n]$, $x_i \in \mathbb{R}$ 一个向量有固定的维数 n, 即元组的长度。我们可以把向量的每个元素想象成代表与所有其他元素正交的方向上的距离。

We consider vectors to be ordered tuples of real numbers. A vector has a fixed dimension n, which is the length of the tuple. We can imagine each element of the vector as representing a distance in an **direction orthogonal** to all the other elements.

- 考虑 3D 向量 $[5, 7, 3]$ 。这是 \mathbb{R}^3 中的一个点，它由
- Consider the 3D vector $[5, 7, 3]$. This is a point in \mathbb{R}^3 , which is formed of:

$$\begin{aligned} 5 * [1, 0, 0] + \\ 7 * [0, 1, 0] + \\ 3 * [0, 0, 1] \end{aligned}$$

每个向量 $[1,0,0]$ 、 $[0,1,0]$ 、 $[0,0,1]$ 都指向一个独立的方向（正交方向），长度为 1。向量 $[5,7,3]$ 可以看作是这些正交单位向量（称为“基向量”）的加权和。向量空间有三个独立的基，因此是三维的。

Each of these vectors $[1,0,0]$, $[0,1,0]$, $[0,0,1]$ is pointing in an independent direction (orthogonal direction) and has length one. The vector $[5,7,3]$ can be thought of a weighted sum of these orthogonal unit vectors (called "**basis vectors**"). The vector space has three independent bases, and so is three dimensional.

- **标量乘法 scalar multiplication** so that ax is defined for any scalar a. For real vectors, $ax = [ax_1, ax_2, \dots, ax_n]$, elementwise scaling.
 - $(\mathbb{R}, \mathbb{R}^n) \rightarrow \mathbb{R}^n$
- **向量加法 vector addition** so that $x + y$ vectors x, y of equal dimension. For real vectors, $x + y = [x_1 + y_1, x_2 + y_2, \dots, x_d + y_d]$ the elementwise sum
 - $(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}^n$
- **norm** $\|x\|$ which allows the length of vectors to be measured 主要用于强调在二维或三维空间中的直观几何含义。
 - $\$mathbb{R} \rightarrow |mathbb{R}|geq 0} \sim \sim \sim |mathbf{x}|_2 = sqrt{x_1^2 + x_2^2 + dots + x_n^2}$
 - **numpy.linalg.norm**
- **vector addition** so that $x + y$ vectors x, y of equal dimension. For real vectors, $x + y = [x_1 + y_1, x_2 + y_2, \dots, x_d + y_d]$ the elementwise sum
 - $(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}^n$
- 内积和norm求夹角 $\theta = arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|}\right)$ 内积基本就反应了他们的夹角，因为norm可以被看作为常量

- `np.degrees(np.arccos(np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))))`
- `np.inner(x,y)`
- 线性插值 **linear interpolation** between two vectors. Linear interpolation of two values is governed by a parameter α , and is just: 线性插值就是构造x和y中所有的向量

$$\text{lerp}(\vec{x}, \vec{y}, \alpha) = (1 - \alpha)\vec{x} + (\alpha)\vec{y}$$

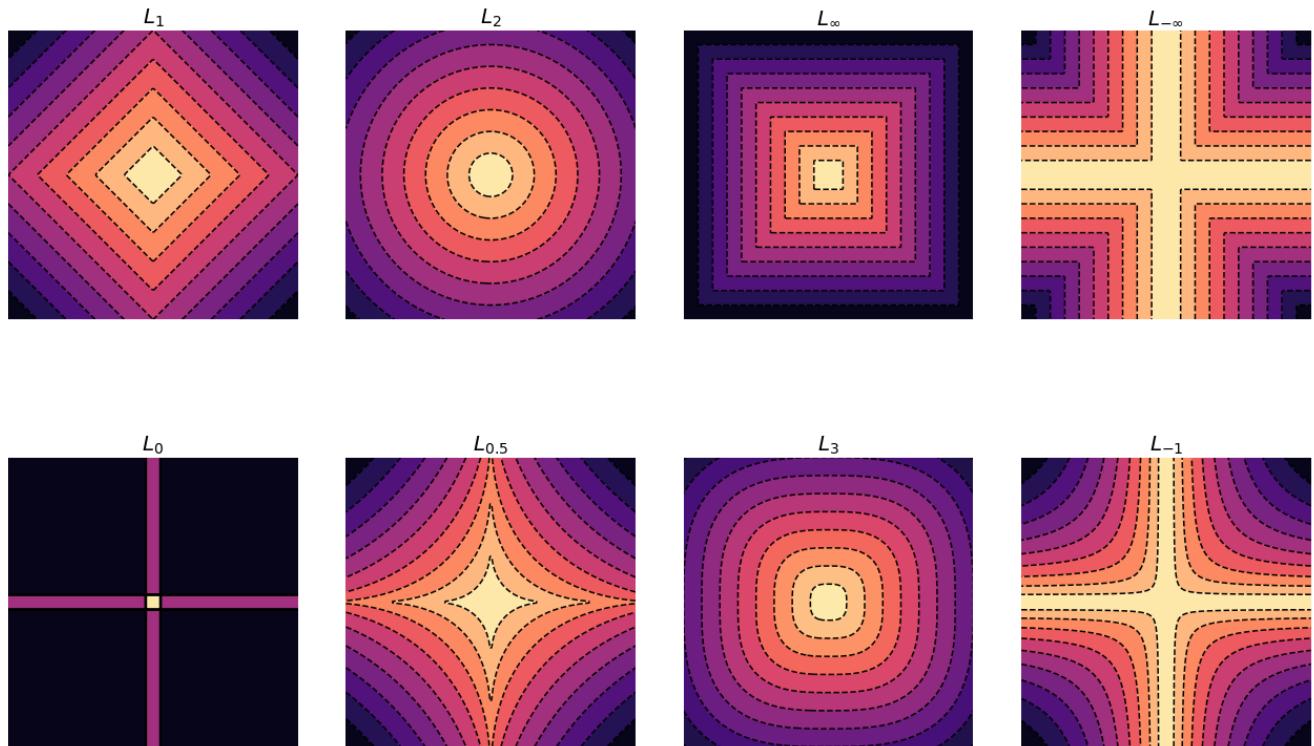
Different Norm

- L0 norm : 通常被定义为向量中非零元素的数量 $||\mathbf{x}||_0 = \text{非零元素的个数}$ is usually defined as the number of non-zero elements in the vector.
- L1 norm : 是向量中各个元素绝对值之和 $||\mathbf{x}||_1 = |\sum_{i=1}^n |x_i||$ is the sum of the absolute values of the elements of the vector
- L2 norm : 通常被称为欧几里得范数, 是向量元素平方的和再开平方根。Often referred to as the Euclidean paradigm, it is the sum of the squares of the vector elements re-squared to the square root.
- Infinity norm: 又称最大范数, 是向量的各个分量绝对值的最大值 $||\mathbf{x}||_\infty = \max(|x_1|, |x_2|, \dots, |x_n|)$ Also known as the maximum paradigm number, it is the maximum value of the absolute value of each component of a vector
- L_p -norm : 是欧几里得范数的一种泛化形式, 用于测量向量在 p 维空间中的长度: $||\mathbf{x}||_p = \left(\sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}}$ is a generalised form of the Euclidean paradigm for measuring the length of a vector in p -dimensional space
- L_p distance L_p 距离用于度量两个向量间的距离, 适用于文档向量 \mathbf{d} 和 \mathbf{d}' 。具体来说, L_p 距离定义如下:

$$L_p(\mathbf{d}, \mathbf{d}') = \left(\sum_{i=1}^n |d_i - d'_i|^p \right)^{\frac{1}{p}}$$

其中, \mathbf{d} 和 \mathbf{d}' 是两个文档向量, d_i 和 d'_i 分别是这两个向量中的对应元素, n 是向量的维数, p 是一个正实数, 决定了距离的类型:

- 当 $p = 1$, 这就是曼哈顿距离 (Manhattan distance), 即向量元素差的绝对值之和。
- 当 $p = 2$, 这就是欧几里得距离 (Euclidean distance), 即向量元素差的平方和的平方根, 是最常用的距离度量。就可以判断相似度
- 当 p 趋向于无穷大时, L_p 距离趋向于最大范数 (maximum norm), 即两个向量间差的绝对值的最大值。
- When $p = 1$, this is the Manhattan distance, which is the sum of the absolute values of the differences of the vector elements.
- When $p = 2$, this is the Euclidean distance, which is the square root of the sum of the squares of the differences of the elements of the vectors, and is the most commonly used distance measure.
- As p tends to infinity, the L_p distance tends to the maximum norm, i.e., the maximum value of the absolute value of the difference between two vectors.



Matrice

线性变换 linear transform

- 旋转 (Rotation) : 旋转变换改变向量的方向而保持其长度不变。在二维空间中，一个围绕原点旋转角度 θ 的旋转变换可以通过以下矩阵实现：

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

- 缩放 (Scaling) : 缩放变换改变向量的长度。在二维空间中，一个沿x轴和y轴分别缩放 a 和 b 倍的缩放变换可以通过以下矩阵实现：

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \end{pmatrix}$$

- 剪切 (Shearing) : 剪切变换会使对象倾斜，改变其形状而保持面积（或体积）不变。在二维空间中，一个沿x轴的剪切可以通过以下矩阵实现：

$$\begin{pmatrix} 1 & k & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

- 反射 (Reflection) : 反射变换会使向量在某个轴上翻转。例如，在二维空间中，一个关于x轴的反射可以通过以下矩阵实现：

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

矩阵乘法 Multiplying a Vector by Matrix

`numpy.dot()` 矩阵乘法的定义是：如果 A 表示线性变换 $f(\vec{x})$ ，而 B 表示线性变换 $g(\vec{x})$ ，那么 $B\vec{A}\vec{x} = g(f(\vec{x}))$ 。

- A is $p \times q$ and
- B is $q \times r$.
-

If $C = AB$ then $C_{ij} = \sum_k a_{ik}b_{kj}$

This is the **outer product** of two vectors, every possible combination of their elements:

$$\vec{x} \otimes \vec{y} = \vec{x}^T \vec{y}$$

and the product of a $1 \times N$ with an $N \times 1$ vector is a 1×1 matrix; a scalar. This is exactly the **inner product** of two vectors: 内积用于计算相似性，如余弦相似度。

$$\vec{x} \cdot \vec{y} = \vec{x}^T \vec{y},$$

and is only defined for vectors \vec{x}, \vec{y} of the same length.

vranice 方差

数据大小的密度

$$\sigma^2 = \frac{1}{N-1} \sum_{i=0}^{N-1} (x_i - \mu_i)^2$$

covranice 协方差

$$\Sigma_{ij} = \frac{1}{N-1} \sum_{k=1}^N (X_{ki} - \mu_i)(X_{kj} - \mu_j)$$

`np.cov()` 两个变量的协方差：

矩阵中的非对角线元素表示这两个变量之间的协方差。如果我们将这两个变量标记为 X 和 Y，则这两个元素分别是 $\text{Cov}(X,Y)$ 和 $\text{Cov}(Y,X)$ 。在协方差的情况下， $\text{Cov}(X,Y)$ 等于 $\text{Cov}(Y,X)$ 。

$$\begin{pmatrix} \text{Var}(X) & \text{Cov}(X, Y) \\ \text{Cov}(Y, X) & \text{Var}(Y) \end{pmatrix}$$

五维情况下：

$$\begin{pmatrix} \text{Var}(X_1) & \text{Cov}(X_1, X_2) & \text{Cov}(X_1, X_3) & \text{Cov}(X_1, X_4) & \text{Cov}(X_1, X_5) \\ \text{Cov}(X_2, X_1) & \text{Var}(X_2) & \text{Cov}(X_2, X_3) & \text{Cov}(X_2, X_4) & \text{Cov}(X_2, X_5) \\ \text{Cov}(X_3, X_1) & \text{Cov}(X_3, X_2) & \text{Var}(X_3) & \text{Cov}(X_3, X_4) & \text{Cov}(X_3, X_5) \\ \text{Cov}(X_4, X_1) & \text{Cov}(X_4, X_2) & \text{Cov}(X_4, X_3) & \text{Var}(X_4) & \text{Cov}(X_4, X_5) \\ \text{Cov}(X_5, X_1) & \text{Cov}(X_5, X_2) & \text{Cov}(X_5, X_3) & \text{Cov}(X_5, X_4) & \text{Var}(X_5) \end{pmatrix}$$

作用

- 衡量变量间的线性关系：协方差矩阵提供了数据集中各个变量之间线性关系的度量。矩阵中的每个元素代表了一对变量之间的协方差，表明它们是否同时增减。正协方差表示两个变量正相关（一个增加时另一个也增加），负协方差表示它们负相关（一个增加时另一个减少）。
- Measuring linear relationships between variables: the covariance matrix provides a measure of the linear relationship between the variables in the data set. Each element of the matrix represents the covariance between a pair of variables, indicating whether they increase or decrease simultaneously.
- 数据特征的提取：通过对协方差矩阵进行特征分解，可以提取数据的主要成分或方向，这是主成分分析（PCA）的基础。PCA通过协方差矩阵识别数据中的主要变化方向，帮助减少数据的维度，同时尽可能保留重要的信息。Extraction of data features: The main components or directions of the data can be extracted by feature decomposition of the covariance matrix, which is the basis of Principal Component Analysis (PCA).
- 数据的多维度分布理解：协方差矩阵反映了多维数据集中各维度的联合变异性。通过分析协方差矩阵，可以理解数据各维度间的相互作用，这对于多变量分析非常重要。Understanding the multidimensional distribution of data: The covariance matrix reflects the joint variability of dimensions in a multidimensional dataset.

Linear algebra

Adjacency matrix 邻接矩阵 : with edges being the weights of the connection between sites.

Matrix powers: Since we have already defined matrix multiplication, we can now define $A^2 = AA$, $A^3 = AAA$, $A^4 = AAAA$, etc. These are the **powers** of a matrix, and are only defined for square matrices(正方形矩阵).

Eigenvalues and eigenvectors

A matrix represents a special kind of function: a **linear transform**; an operation that performs rotation and scaling on vectors. However, there are certain vectors which don't get rotated when multiplied by the matrix. 矩阵代表一种特殊的函数：线性变换；一种对向量进行旋转和缩放的运算。不过，有些向量与矩阵相乘时不会发生旋转。

Special vectors: They only get scaled (stretched or compressed). These vectors are called **eigenvectors**, and they can be thought of as the "fundamental" or "characteristic" vectors of the matrix, as they have some stability. The prefix **eigen** just means **characteristic** (from the German for "own"). The scaling factors that the matrix applies to its eigenvectors are called **eigenvalues**. 特殊矢量：它们只会被缩放（拉伸或压缩）。这些向量被称为特征向量，它们可以被视为矩阵的“基本”或“特征”向量，因为它们具有一定的稳定性。矩阵应用于其特征向量的缩放因子称为特征值。`evals, evecs = np.linalg.eig(A)`

np的eigenvalues缺陷

- 数值稳定性问题：在处理非常大或非常小的数值时，计算特征值和特征向量可能会出现数值稳定性问题。这可能导致结果的精度不高或者计算过程中出现数值上的不稳定。
- 计算复杂性：对于非常大的矩阵，计算特征值和特征向量的算法可能非常耗时。尤其是在需要高精度结果的情况下，计算量会显著增加。
- Numerical stability issues: Numerical stability issues may arise in the calculation of eigenvalues and eigenvectors when dealing with very large or very small values. This may lead to poor accuracy of the results or numerical instability in the computation. Computational complexity:
- For very large matrices, the algorithms for computing eigenvalues and eigenvectors can be very time consuming. Especially if high precision results are required, the computational effort can increase significantly.

PCA 主成分分析 Principal Component Analysis

```
# Step 1: 数据标准化
# 计算每个特征的均值和标准差
mean = np.mean(X, axis=0)
std = np.std(X, axis=0)

# 标准化数据
X_normalized = (X - mean) / std
```

```

# Step 2: 计算协方差矩阵
cov_matrix = np.cov(X_normalized.T)

# Step 3: 计算协方差矩阵的特征值和特征向量
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Step 4: 对特征向量进行排序，并选择主成分
# 对特征值从大到小排序，获取排序后的特征值的索引
sorted_indices = np.argsort(eigenvalues)[::-1]

# 选择前k个特征向量，k是你想要的主成分数量
k = 2 # 例如，选择前2个主成分
principal_components = eigenvectors[:, sorted_indices[:k]]

# Step 5: 将原始数据转换到新的特征空间
X_pca = X_normalized.dot(principal_components)

# X_pca 是降维后的数据

```

特征分解

- **定义：** 特征分解是将一个矩阵分解为特征值和特征向量。它只适用于方阵。It only applies to square matrices.
- **优点：**
 - 提供了矩阵的直接分解，可以揭示矩阵的基本特性，如可逆性、秩等。
 - 特征值和特征向量的概念在理解线性变换中非常重要。
 - Provides a direct decomposition of a matrix that reveals the basic properties of a matrix such as invertibility, rank, etc.
 - The concepts of eigenvalues and eigenvectors are important in understanding linear transformations.
- **缺点：**
 - 仅限于方阵，不能应用于非方阵（例如，大多数现实世界的数据集是非方阵）。
 - 计算上可能不够稳定，特别是对于大型矩阵。
 - Limited to square matrices, cannot be applied to non-square matrices (e.g., most real-world datasets are non-square).
 - May not be computationally stable, especially for large matrices.

主成分分析 (PCA)

- **定义：** PCA是一种统计方法，通过正交变换将可能相关的变量转换为线性不相关的变量集合。它不限于方阵，并通常用于降维。
- **优点：**
 - 可以应用于任何大小的矩阵，特别适用于高维数据集。
 - 有效的数据压缩工具，可以减少数据集的维度，同时保留最重要的信息。
 - 有助于去除噪声，强化数据集中最重要的信号。
 - Can be applied to matrices of any size and is particularly suitable for high-dimensional datasets.
 - Effective data compression tool that reduces the dimensionality of a dataset while retaining the most important information.

- Helps to remove noise and enhance the most important signals in the dataset.
- 缺点:
 - PCA依赖于线性假设，对于非线性数据结构不是最佳选择。
 - 结果的解释可能不直观，特别是在高维数据上。
 - 对数据的标准化或规范化高度敏感。
 - PCA relies on linear assumptions and is not optimal for non-linear data structures.
 - Interpretation of results may not be intuitive, especially on high-dimensional data.
 - Highly sensitive to standardisation or normalisation of data.

比较

- 特征分解是PCA的数学基础。实际上，PCA涉及到协方差矩阵或数据矩阵的奇异值分解（SVD），这可以视为特征分解的一种形式。
- PCA通常被视为特征分解在数据分析和降维方面的实际应用。
- 在实际操作中，PCA更加普遍，因为它适用于非方阵，并且与特定的应用（如数据压缩、特征提取、噪声减少）密切相关。In practice, PCA is more common because it is applicable to non-square matrices and is closely related to specific applications (e.g., data compression, feature extraction, noise reduction).

Trace(迹)

The trace of a square matrix can be computed from the sum of its diagonal values:

$$\text{Tr}(A) = a_{1,1} + a_{2,2} + \cdots + a_{n,n}$$

It is also equal to the sum of the eigenvalues of A

$$\text{Tr}(A) = \sum_{i=1}^n \lambda_i$$

The trace can be thought of as measuring the **perimeter** of the parallelotope of a unit cube transformed by the matrix. [Strictly, it is *proportional* to the perimeter, with the constant of proportionality being Perimiter(A) = $2^{n-1} \text{Tr}(A)$].

Determinant 行列式

The determinant $\det(A)$ is an important property of square matrices. It can be thought of as the **volume** of the parallelotope of a unit cube transformed by the matrix -- it measures how much the space expands or contracts after the linear transform.

It is equal to the product of the eigenvalues of the matrix.

$$\det(A) = \prod_{i=1}^n \lambda_i$$

If any eigenvalue λ_i of A is 0, the determinant $\det(A) = 0$, and the transformation collapses at least one dimension to be completely flat. This means that the transformation **cannot be reversed**; information has been lost.

Definite and semi-definite matrices

A matrix is called

- **positive definite** if all of its eigenvalues are greater than zero: $\lambda_i > 0$. 正定矩阵
- **positive semi-definite** if all of its eigenvalues are greater than or equal to zero: $\lambda_i \geq 0$. 半正定矩阵
- **negative definite** if all of the eigenvalues are less than zero: $\lambda_i < 0$, - 负定矩阵
- **negative semi-definite** if all the eigenvalues are less than or equal to zero: $\lambda_i \leq 0$. 半负定矩阵

在统计数据分析中，协方差矩阵通常是正定的，这意味着它保留了数据的某些基本性质和结构。换句话说，正定矩阵在变换空间中保持了向量的“方向”。在数值优化中，海森矩阵被用于判断函数的凸性，其正定性是找到最优解的重要条件。In statistical data analysis, the covariance matrix is usually positive definite, which means that it preserves some of the fundamental properties and structure of the data. In other words, a positive definite matrix maintains the "direction" of the vectors in the transformation space. In numerical optimisation, the Hessian matrix is used to determine the convexity of a function, and its positive definiteness is important for finding the optimal solution.

Matrix Inversion

We have seen four basic algebraic operations on matrices:

- scalar multiplication cA ;
- matrix addition $A + B$;
- matrix multiplication BA
- matrix transposition A^T

There is a further important operation: **inversion** A^{-1} , defined such that:

- $A^{-1}(Ax) = \vec{x}$,
- $A^{-1}A = I$
- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$

`np.linalg.inv()` 矩阵可逆的条件为是方阵且 $\det(A)$ 也就是行列式不为0 $\det(A) \neq 0$ 。如果 $\det(A)=0$ ，那么变换 A 至少折叠了一个维度，这意味着它不是双射的。If $\det(A)=0$, then the transformation A collapses at least one dimension, which means it's not bijective.

奇异矩阵与非奇异矩阵 Singular and non-singular matrices

A matrix with $\det(A) = 0$ is called **singular** and has no inverse.

A matrix which is invertible is called **non-singular**.

The geometric intuition for this is simple. Going back to the parallelogram model, a matrix with zero determinant has at least one zero eigenvalue. This means that at least one of the dimensions of the parallelepiped has been squashed to nothing at all. Therefore it is impossible to reverse the transformation, because information was lost in the forward transform.

All of the original dimensions must be preserved in a linear map for inversion to be meaningful; this is the same as saying $\det(A) \neq 0$.

BUT only square matrices can be inverted !

Inversion is only defined for square matrices, representing a linear transform $\mathbb{R}^n \rightarrow \mathbb{R}^n$. This is equivalent to saying that the determinant of the matrix must be non-zero: $\det(A) \neq 0$. Why?

A matrix which is non-square maps vectors of dimension m to dimension n . This means the transformation collapses or creates dimensions. Such a transformation is not uniquely reversible.

For a matrix to be invertible it must represent a **bijection** (a function that maps every member of a set onto exactly one member of another set).

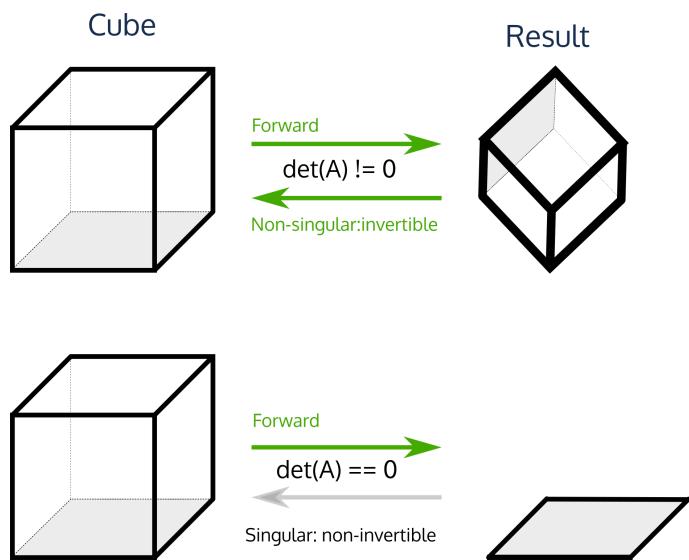
Singular and non-singular matrices

A matrix with $\det(A) = 0$ is called **singular** and has no inverse.

A matrix which is invertible is called **non-singular**.

The geometric intuition for this is simple. Going back to the parallelogram model, a matrix with zero determinant has at least one zero eigenvalue. This means that at least one of the dimensions of the parallelepiped has been squashed to nothing at all. Therefore it is impossible to reverse the transformation, because information was lost in the forward transform.

All of the original dimensions must be preserved in a linear map for inversion to be meaningful; this is the same as saying $\det(A) \neq 0$.



Time complexity

Matrix inversion, for a general $n \times n$ matrix, takes $O(n^3)$ time. It is *provable* that no general matrix inversion algorithm can ever be faster than $O(n^3)$ (one of the few problems for which a tight polynomial time bound is known). 矩阵运算涉及大量重复的浮点运算（舍入累加）--反演特别难以直接以稳定的形式计算，许多理论上可以反演的矩阵无法使用浮点表示法进行反演。时间复杂性--对于一般的 $n \times n$ 矩阵，矩阵反演需要 $O(n^3)$ 时间。可以证明，没有一种通用矩阵反演算法能比 $O(n^3)$ 更快。

Special cases

- orthogonal matrix (rows and columns are all orthogonal unit vectors): $O(1)$, $A^{-1} = A^T$ 正交矩阵

- diagonal matrix (all non-diagonal elements are zero): $O(n)$, $A^{-1} = \frac{1}{A}$ (i.e. the reciprocal of the diagonal elements of A). 对角矩阵
- positive-definite matrix: $O(n^2)$ via the *Cholesky decomposition*. We won't discuss this further. 正定矩阵
- triangular matrix (all elements either above or below the main diagonal are zero): $O(n^2)$, trivially invertible by **elimination algorithms**. 三角形矩阵（主对角线上下的所有元素均为零）： $O(n^2)$ \$, 通过消除算法可反转。

SVD Singular Value Decomposition 奇异值分解

The **singular value decomposition** (SVD) is a general approach to decomposing any matrix A. It is the powerhouse of computational linear algebra.

The SVD produces a decomposition which splits **ANY** matrix up into three matrices:

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

where

- A is any $m \times n$ matrix,
- U is a **square unitary** $m \times m$ matrix, whose columns contain the **left singular vectors**, 左奇异向量
- V is an **square unitary** $n \times n$ matrix, whose columns contain the **right singular vectors**, 右奇异向量
- Σ is a diagonal $m \times n$ matrix, whose diagonal contains the **singular values**. 其对角线上的元素是奇异值，这些奇异值是矩阵 A 的非负实数，通常按降序排列。

一个单元矩阵的共轭转置等于其逆矩阵。如果 A 为实数，那么 U 和 V 将是正交矩阵 ($U^T = U^{-1}$)，其行都具有单位矩，其列也都具有单位矩。A **unitary** matrix is one whose conjugate transpose is equal to its inverse. If A is real, then U and V will be **orthogonal** matrices ($U^T = U^{-1}$), whose rows all have unit norm and whose columns also all have unit norm.

矩阵 Σ 的对角线是奇异值的集合，与特征值密切相关，但并不完全相同（特殊情况除外，如当 A 是正半有限对称矩阵时）！奇异值总是正实数。The diagonal of the matrix Σ is the set of **singular values**, which are closely related to the eigenvalues, but are not quite the same thing (except for special cases like when A is a positive semi-definite symmetric matrix)! The **singular values** are always positive real numbers.

We can compute the SVD with `np.linalg.svd`:

与PCA的关系

PCA是一种统计方法，用于通过正交变换将可能相关的变量转换为线性不相关的变量集合，通常用于降维。

- PCA可以通过SVD实现。具体来说，对数据矩阵 (A) 进行SVD，得到 ($A = U\Sigma V^T$)。
- 在PCA中，主成分就是SVD中的右奇异向量 (V) (或 (V^T) 的行)。
- 通过SVD，可以更稳定和高效地计算PCA，尤其是在处理大型数据集时。

Fractional powers

We can use the SVD to compute interesting matrix functions like the square root of a matrix $A^{1/2}$.

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

Note: $A^{1/2}$ is not the elementwise square root of each element of A!

Rather, we must compute the elementwise square root of Σ , then compute $A^{1/2} = U\Sigma^{1/2}V^T$.

Inversion - relation to SVD

We can efficiently invert a matrix once it is in SVD form. For a non-symmetric matrix, we use:

$$A^{-1} = V\Sigma^{-1}U^T$$

Rank of a matrix 矩阵的秩

矩阵的秩等于非零奇异值的个数。The **rank** of a matrix is equal to the number of non-zero singular values.

- 如果非零奇异值的个数等于矩阵的大小，那么矩阵就是全秩。
- 全秩矩阵的行列式不为零，并且可以反转。
- 秩告诉我们变换所代表的平行矩阵有多少维。
- 如果矩阵没有满秩，它就是奇数（不可反转），具有缺秩。
- 如果非零奇异值的数量远小于矩阵的大小，则该矩阵为低阶矩阵。

Condition number of a matrix 矩阵的条件数

矩阵的条件数是最大奇异值与最小奇异值的比值。The **condition number** number of a matrix is the ratio of the largest singular value to the smallest.

- 这只针对全秩矩阵。
- 条件数衡量矩阵的反转对微小变化的敏感程度。
- 条件数小的矩阵称为条件良好，不太可能引起数值问题。
- 条件数大的矩阵是条件差的，数值问题可能会很严重。

条件不良的矩阵几乎是奇异的，因此对其进行反演将导致无效结果，因为浮点舍入错误。

```
def pca_using_svd(X, n_components):
    """
    Perform PCA using SVD.

    # Centering the data (subtract the mean of each feature)
    X_centered = X - np.mean(X, axis=0)

    # Performing SVD
    U, S, VT = np.linalg.svd(X_centered)

    # Selecting the top n_components principal components
    principal_components = VT[:n_components, :]

    # Projecting the data onto principal components
    X_pca = np.dot(X_centered, principal_components.T)

    return X_pca, principal_components
```

白化 Whitening

使用SVD进行白化的步骤

1. 中心化数据:

- 对数据进行中心化，即从每个特征中减去其均值。这样做的目的是使数据在各个维度上的平均值为零。

2. 计算SVD:

- 对中心化后的数据矩阵 X 进行奇异值分解。SVD将数据矩阵分解为三个矩阵的乘积：
$$X = U\Sigma V^T$$
- 其中， U 和 V 是正交矩阵，而 Σ 是对角矩阵，对角线上的元素是奇异值。

3. 构建白化矩阵:

- 白化矩阵通常是通过取 Σ 中奇异值的逆平方根来构建的，记为 $\Sigma^{-\frac{1}{2}}$ 。

4. 应用白化变换:

- 使用白化矩阵对原始数据矩阵 X 进行变换，得到白化后的数据 $X_{\text{whitened}} = U\Sigma^{-1/2}U^TX$

白化的目的

• 去除相关性 Removal of correlation:

- 白化的主要目的是去除数据特征间的相关性。通过这种变换，数据的协方差矩阵将变为单位矩阵，意味着变换后的特征彼此统计独立。*The main purpose of whitening is to remove the correlation between the features of the data. With this transformation, the covariance matrix of the data is changed to a unit matrix, meaning that the transformed features are statistically independent of each other.*

• 方差归一化 Covariance Normalisation:

- 白化过程中，每个特征的方差都被标准化为1。这样做可以确保没有任何一个特征在数值上主导整个数据集。*During whitening, the variance of each feature is normalised to 1. This ensures that no single feature numerically dominates the entire data set.*

• 改善算法性能 Improved algorithm performance:

- 白化后的数据有助于改善许多机器学习算法的性能，特别是在涉及距离计算的算法（如k-means聚类）和深度学习模型中。*Whitened data helps to improve the performance of many machine learning algorithms, especially in algorithms that involve distance calculations (e.g. k-means clustering) and deep learning models.*

• 数据可视化和进一步处理 Data visualisation and further processing:

- 白化处理可以使数据更适合进行可视化和进一步的分析处理，因为它确保了数据在所有维度上具有相似的规模。*Whitening makes the data more suitable for visualisation and further analytical processing as it ensures that the data has a similar scale in all dimensions.*

优化 optimisation

概念 notion of word

derivative of function 导函数

polynomial 多项式

parameters and objective function 参数和目标函数

synthesiser 合成器

constraint 约束

geometric median 几何中值

Introduction

Optimisation is the process of adjusting things to make them better. In computer science, we want to do this *automatically* by a algorithm. An enormous number of problems can be framed as optimisation, and there are a plethora of algorithms which can then do the automatic adjustment *efficiently*, in that they find the best adjustments in few steps. In this sense, *optimisation is search*, and optimisation algorithms search efficiently using mathematical structure of the problem space.

优化 是调整事物以使其变得更好的过程。在计算机科学中，我们希望通过算法“自动”做到这一点。大量的问题可以被视为优化，并且有大量的算法可以“有效”地进行自动调整，因为它们只需几个步骤即可找到最佳调整。从这个意义上说，优化就是搜索，优化算法使用问题空间的数学结构进行有效搜索。

parameters: the things we can adjust, which might be a scalar or vector or other array of values, denoted θ . The parameters exist in a **parameter space** -- the set of all possible configurations of parameters denoted Θ . This space is often a **vector space** like \mathbb{R}^n , but doesn't need to be.

参数：我们可以调整的事物，可能是一个标量、向量或其他值数组，表示为 θ 。参数存在于一个参数空间中——所有可能的参数配置集合表示为 Θ 。这个空间通常是一个向量空间，如 \mathbb{R}^n ，但不必是。

the objective function: a function that maps the parameters onto a *single numerical measure* of how good the configuration is. $L(\theta)$. The output of the objective function is a single scalar. The objective function is sometimes called the *loss function*, the *cost function*, *fitness function*, *utility function*, *energy surface*, all of which refer to (roughly) the same concept. It is a quantitative ("objective") measure of "goodness".

目标函数：一个将参数映射到配置好坏的单一数值度量上的函数，表示为 $L(\theta)$ 。目标函数的输出是一个单一的标量。目标函数有时被称为损失函数、成本函数、适应度函数、效用函数、能量面，所有这些都是指（大致）相同的概念。它是“好坏”的定量（“客观”）度量。

$$\theta^* = \operatorname{argmin}_{\theta \in \Theta} L(\theta)$$

- θ^* is the configuration that we want to find; the one for which the objective function is lowest.
- Θ is the set of all possible configurations that θ could take on, e.g. \mathbb{R}^N .

θ^* 是我们想要找到的配置；也就是使目标函数最小值

Θ 是所有可能的 θ 配置的集合，例如 \mathbb{R}^N

在这我们为了避免混淆，常用loss函数来代替目标函数这一概念

constraints: the limitations on the parameters. This defines a region of the parameter space that is feasible, the **feasible set** or **feasible region**. For example, the synthesizer above has knobs with a fixed physical range, say 0-10; it isn't possible to turn them up to 11. Most optimisation problems have constraints of some kind;

约束：参数的限制。这定义了参数空间中可行的区域，即可行集或可行区域。例如，上面的合成器有固定物理范围的旋钮，比如 0-10；它不可能调到 11。大多数优化问题都有某种约束；

Loss的基本定义

正如这个例子所示，通常会把问题表达成目标函数是输出与参考之间的距离测量的形式。不是每个目标函数都有这种形式，但许多目标函数确实如此。

也就是说，我们有某个函数 $y' = f(\vec{x}; \theta)$ ，它从输入 \vec{x} 产生输出，该输入受一组参数 θ 的控制，并且我们测量输出和某个参考值 y 之间的差异（例如，使用向量范数）：

$$L(\theta) = |y' - y| = |f(\vec{x}; \theta) - y|$$

这里， $L(\theta)$ 是根据参数 θ 定义的损失函数， $|\cdot|$ 是某种向量范数，用于度量 $f(\vec{x}; \theta)$ （模型的预测或输出）与目标 y （实际值或期望值）之间的差异或距离。在优化过程中，我们的目标是找到参数 θ 的配置，以使得这个损失函数的值最小，从而使输出尽可能接近参考值。

这在逼近问题中非常常见，我们想找到一个函数来逼近一组测量观察值。这是机器学习的核心问题。

注意，记号 $f(\vec{x}; \theta)$ 意味着函数 f 的输出既依赖于某个（向量）输入 \vec{x} ，也依赖于参数向量 θ 。优化过程只调整 θ ，而在优化期间认为向量 \vec{x} 是固定的（例如，它可能代表一系列实世界的测量值）。

验证

对于验证loss函数来说，验证会消耗一定的计算能力，因此会有时间成本。

这意味着一个好的优化算法将通过少量的查询（评估目标函数）找到参数的最优配置。为了做到这一点，必须有数学上的结构可以帮助指导搜索。如果完全没有任何结构，最好的办法可能就是随机猜测参数配置，并在一定次数的迭代后选择成本最低的配置。这通常不是一个可行的方法。

因此，优化算法经常利用目标函数的某种结构来指导搜索，这可能包括使用梯度（在可微分的情况下）、凸性（在凸优化问题中）、或者是其他一些可以指示参数调整方向的特性。算法可能还会利用历史信息来预测参数的更好配置，从而减少目标函数评估的次数。在现代优化算法中，如贝叶斯优化等，甚至会构建目标函数的代理模型（surrogate model），在代理模型上进行评估和搜索，从而减少对实际目标函数评估的需求。

Discrete vs. continuous 离散 vs 连续

确实，当参数位于连续空间（通常是 \mathbb{R}^n ）中时，问题就是连续优化问题；如果参数是离散的，那么问题就是离散优化问题。连续优化通常更容易处理，因为我们可以利用平滑性和连续性的概念。

Indeed, a problem is a continuous optimisation problem when the parameters lie in a continuous space (usually \mathbb{R}^n); if the parameters are discrete, then the problem is a discrete optimisation problem. Continuous optimisation is usually easier to handle because we can make use of the concepts of smoothness and continuity.

优化的属性 每个优化问题都有两个部分：

- 参数，可以调整的事物。
- 目标函数，用来衡量一组特定参数的好坏。

一个优化问题通常也包括：

- 约束，定义了参数的可行集。
- 目标函数是，一个关于参数的函数，它返回一个单一的标量值，表示该参数集的优良程度。

Every optimisation problem has two parts:

- Parameters, things that can be tuned.
- The objective function, which measures how good a particular set of parameters are.

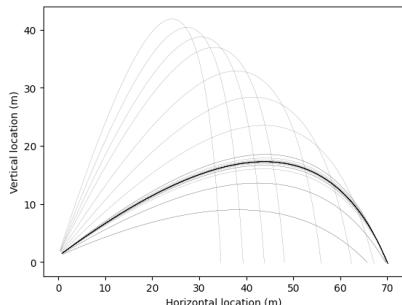
An optimisation problem usually also includes:

- Constraints, which define the feasible set of parameters.
- The objective function is, a function of the parameters that returns a single scalar value indicating how good the set of parameters is.
-

Example 1: throwing a stone

例如，如果我想要优化我能将石头扔多远，我可能能够调整投掷角度。这是我可以调整的参数（在这种情况下，只有一个参数 $\theta = [\alpha]$ ）。

目标函数必须是一个依赖于这个参数的函数。我将不得不模拟投掷球以计算它投掷的距离，并尝试让它投得越来越远。



Focus: continuous optimisation in real vector spaces

重点：实向量空间中的连续优化

本课程将专注于在 \mathbb{R}^n 中连续问题的优化。即

$$\theta \in \mathbb{R}^n = [\theta_1, \theta_2, \dots, \theta_n],$$

并且优化问题是：

$$\theta^* = \operatorname{argmin}_{\theta \in \mathbb{R}^n} L(\theta), \text{ subject to constraints}$$

这是在连续向量空间中搜索的问题，以找到使 $L(\theta)$ 最小的点。我们通常会遇到目标函数在该向量空间中是平滑和连续的问题；注意，参数是连续空间的元素，并不必然意味着目标函数在该空间中是连续的。This is

the problem of searching through a continuous vector space to find the point that minimises $L(\theta)$. We usually encounter problems where the objective function is smooth and continuous in that vector space; note that the fact that the arguments are elements of a continuous space does not necessarily mean that the objective function is continuous in that space.

一些优化算法是迭代的，它们生成越来越接近解决方案的近似值。其他方法是直接的，比如线性最小二乘法，涉及一步找到最小值。在本课程中，我们将主要关注迭代的、近似的优化。

Geometric median: optimisation in \mathbb{R}^2

几何中值：在 \mathbb{R}^2 的优化

找到一个 >1D 数据集的中位数。标准的中位数是通过排序然后选择中间元素计算的（对于偶数大小的数据集有各种规则）。这对于更高维度不适用，而且没有简单直接的算法。但中位数有一个简单的定义：它是最小化到数据集中所有向量的距离和的向量。

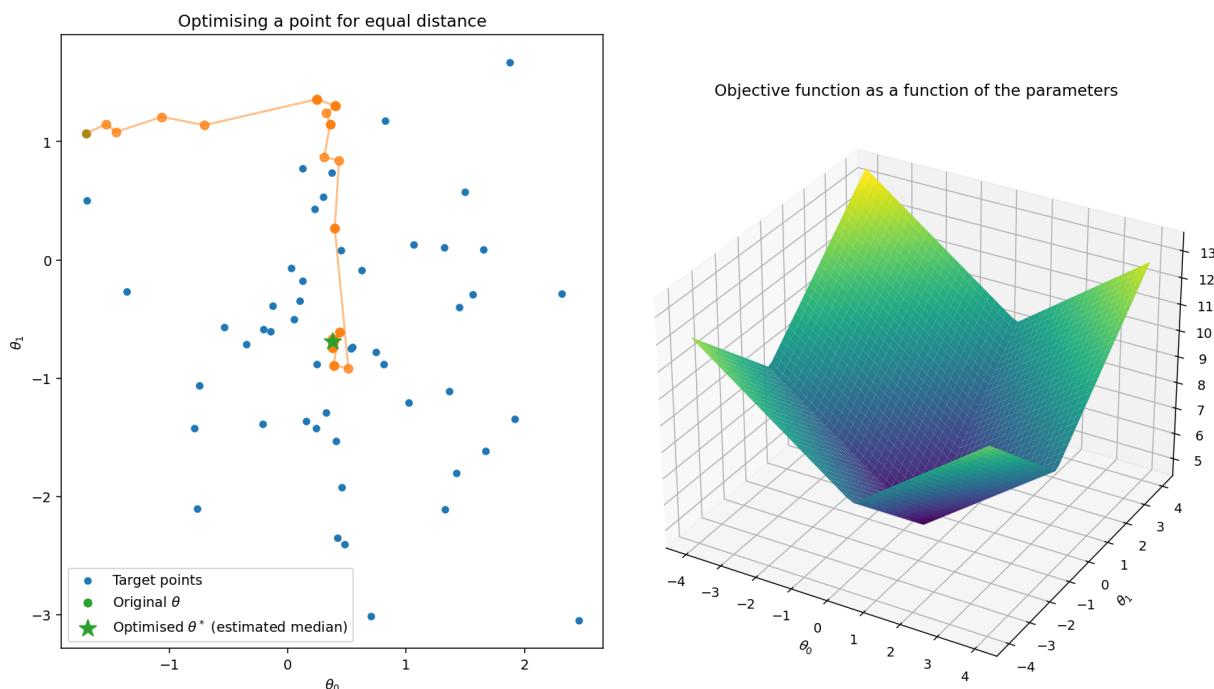
一个非常简单的优化例子是找到一个点，该点最小化到一组其他点的距离（关于某种规范）。我们可以定义：

参数 $\theta = [x, y \dots]$, 二维空间中的位置。

目标函数 一个点与一系列目标点 \vec{x}_i 之间距离的总和：

$$L(\theta) = \sum_i \|\theta - \vec{x}_i\|_2$$

这将尝试找到空间中的一个点（表示为 θ ），该点最小化到目标点的距离。我们可以从某个随机初始条件（对 θ 的猜测）开始解决这个问题：



上述图： θ_1 对应 x ; θ_2 对应 y

optimisation in \mathbb{R}^n

在 \mathbb{R}^n 的优化

我们可以同样容易地在更高维度中工作。一个略有不同的问题是尝试找到一个点的布局，使得这些点是均匀分布的（针对某种规范）。在这种情况下，我们必须优化一整组点，我们可以通过将它们全部合并到一个单一的参数向量中来做到这一点。

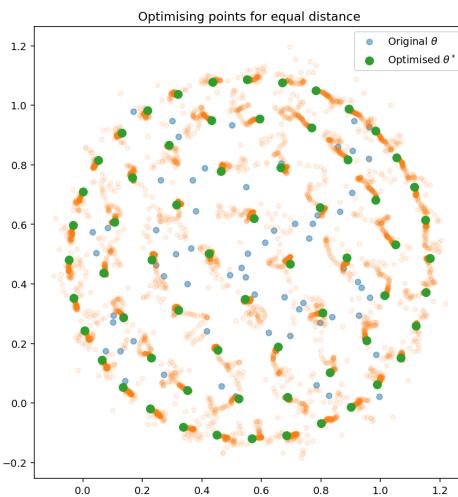
我们可以定义：

参数 $\theta = [x_1, y_1, x_2, y_2, \dots]$, 二维中点位置的数组, 注意: 我们已将二维点序列 "解包" 为高维向量, 因此点的整体配置就是向量空间中的一个点。

损失函数 点与某个目标欧氏距离之差的平方和:

$$\sum_i \sum_j (\alpha - \|x_i - x_j\|_2)^2$$

这将尝试找到所有点之间相隔 α 单位的点的配置。



原始点是蓝色的，最后优化的点是绿色的。最后进行了优化，绿色和橙色的是优化后的效果。每个点的迭代轨迹。

- 很明显，最后绿色的点比蓝色的点间距更均匀。所以优化的效果似乎不错。
- 橙色轨迹表明，确实是逐步有效地移动到这些位置的。这并不是瞬间解决的。它是通过将这些蓝色点逐渐移向橙色、绿色的最佳位置。

约束优化 Constrained optimisation

A constrained optimisation might be written in terms of an equality constraint:

$$\theta^* = \underset{\theta \in \Theta}{\text{argmin}} L(\theta) \text{ subject to } c(\theta) = 0,$$

or an inequality:

$$\theta^* = \underset{\theta \in \Theta}{\text{argmin}} L(\theta) \text{ subject to } c(\theta) \leq 0,$$

where $c(\theta)$ is a function that represents the constraints.

等式约束可以被看作是将参数约束在一个表面上，来代表一种权衡。例如， $c(\theta) = \|\theta\|_2 - 1$ 强制参数位于单位球面上。等式约束可能用于在总值必须保持不变的情况下进行权衡（例如，卫星的有效载荷重量可能事先就被固定）。

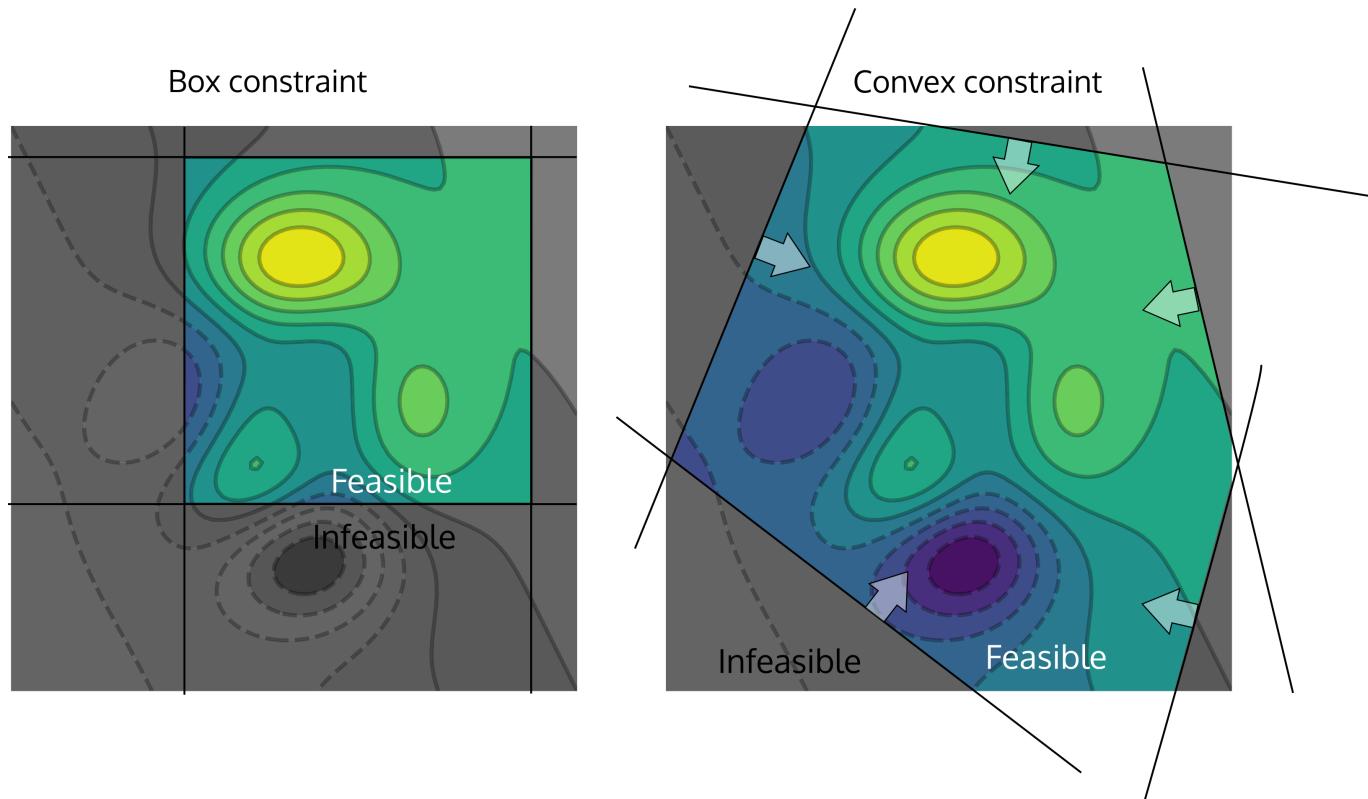
不等式约束可以被看作是将参数约束在一个体积内，来代表值的范围限制。例如， $c(\theta) = \|\theta\|_\infty - 10$ 强制参数位于以原点为中心，范围为(-10, 10)的盒子内——这也许是合成器旋钮的范围。

常见的优化类型

box constraint 是一种简单的约束类型，它只是要求 θ 在 R^n 内的一个盒子里；例如，每个元素 $0 < \theta_i < 1$ （所有参数在正单元立方体内）或 $\theta_i > 0$ （所有参数在正象限内）。这是一个形式简单的不等式约束 $c(\theta)$ 。许多优化算法支持盒约束。

convex constraint 是另一种简单的约束，其中约束是参数 θ 的凸和的一系列不等式。**box constraint** 是 **convex constraint** 的一个特定子类。这相当于可行集被许多平面/超平面（在曲线凸约束的情况下可能是无限多个）的交集所限制。

Unconstrained optimization 不对参数施加任何约束，搜索空间中的任何参数配置都是可能的。在许多问题中，纯无约束优化会导致无用的结果



约束与惩罚 constraints and penalties

无约束优化很少能单独给出有用的答案。虽然我们通常将 θ 表示为在 R^N 中，但可行集通常不是整个向量空间。有两种方法来处理这个问题：

约束优化

使用本身支持硬约束的优化算法。这对某些类型的优化很直接，但对一般优化来说比较棘手。直接使用 convex 或者 box 约束

优点：

- 保证解决方案将满足约束。
- 可能能够利用约束来加速优化。
- Assurance that the solution will satisfy the constraints.
- May be able to use constraints to accelerate optimisation.

缺点：

- 可能比无约束优化效率低。
- 可用于优化的算法较少。
- 可能很难用优化器中可用的参数指定可行区域。
- May be less efficient than unconstrained optimisation.
- Fewer algorithms are available for optimisation.
- May be difficult to specify feasible regions with parameters available in the optimiser.

软约束

对目标函数应用惩罚，以“阻止”违反约束的解决方案。Apply penalties to the objective function to "block" solutions that violate the constraints.

$$L(\theta') = L(\theta) + \lambda(\theta),$$

其中 $\lambda(\theta)$ 是一个惩罚函数，随着约束的违反程度的增加而值增加。

优点：

- 任何优化器都可以使用
- 能够合理地处理软约束
- Can be used by any optimiser
- Can handle soft constraints reasonably well

缺点：

- 可能不服从一些重要的约束
- 可能难以将约束形式化为惩罚
- 不能利用在空间受限区域中的高效搜索
- May not obey some important constraints
- May be difficult to formalise constraints into penalties
- Cannot take advantage of efficient search in spatially constrained regions

目标函数的松弛

解决离散优化和约束优化问题可能更困难；有些算法尝试寻找类似的连续或无约束优化问题来替代解决。这称为松弛；解决的是问题的松弛版本，而不是原始的难优化问题。例如，有时可以将问题中的约束吸收进目标函数，以将一个受约束问题转换为一个无约束问题。

Solving discrete and constrained optimisation problems can be more difficult; some algorithms try to find similar continuous or unconstrained optimisation problems to solve instead. This is called relaxation; a relaxed version of the problem is solved rather than the original hard optimisation problem. For example, it is sometimes possible to absorb the constraints in a problem into the objective function in order to convert a constrained problem into an unconstrained one.

惩罚

惩罚 指的是增加目标函数的项以最小化解决方案的某些其他属性，通常用来近似约束优化。这在近似问题中广泛使用，以找到能够良好泛化的解决方案；也就是调整得既能近似一些数据，但又不是太近。

这是对需要专门算法的具有硬约束问题的松弛，变为一个简单的目标函数问题，该目标函数适用于任何目标函数。拉格朗日乘子法就是一个例子

惩罚函数

惩罚函数只是一个增加到目标函数中的项，它会不利于“坏的解决方案”。

目标函数：石头落地有多远？

$$L(\theta) = \text{throw_distance}(\theta)$$

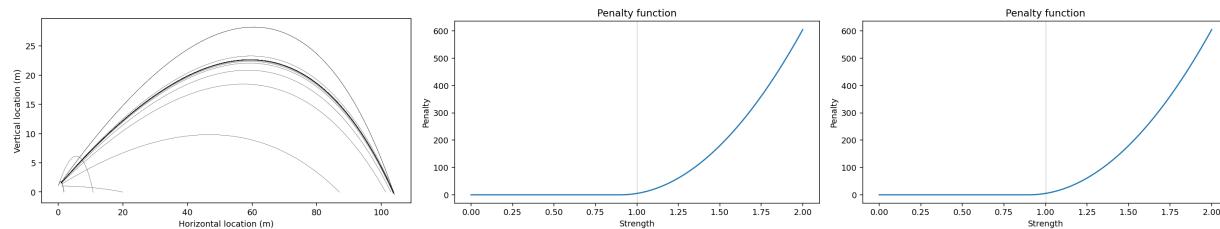
参数：投掷角度 α 和投掷力度 v （退出速度）， $\theta = [\alpha, v]$ 约束：投掷力度 $0 \leq v \leq v_k$ ，大于零且小于某个最大力量。

有两个选项：

- 使用受约束的优化算法，这种算法甚至不会搜索超过最大力量的解决方案。
- 改变目标函数，使过度的投掷力度变得不可接受。

受约束的优化算法

加入惩罚函数



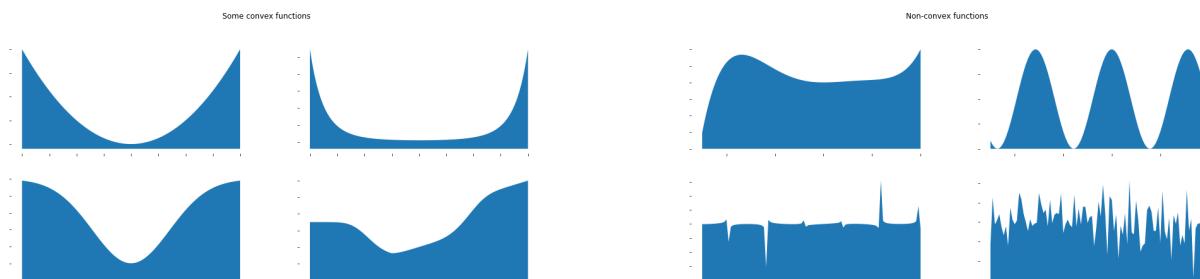
目标函数的属性

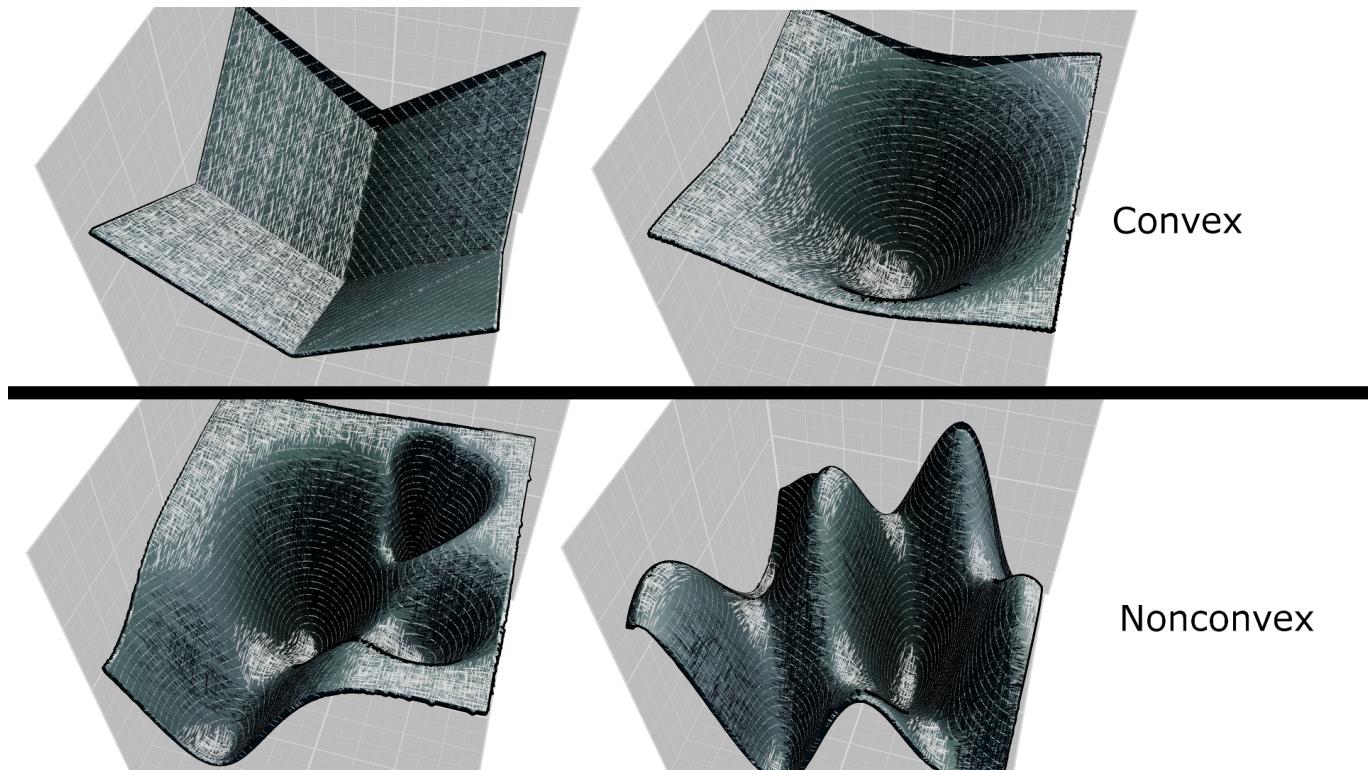
凸性，全局最小值和局部最小值

一个目标函数可能有局部最小值。局部最小值是指在该点周围的每个方向上目标函数都在增加的任何点（那个参数设置）。在该点改变参数会增加目标函数。

如果目标函数有一个单一的、全局最小值，那么它就是凸函数。例如，每个二次函数都是一个抛物线（在任何数量的维度中），因此恰好有一个最小值。其他函数可能有具有局部最小值的区域，但这些最小值不是函数可能取得的最小可能值。

凸性意味着找到任何最小值等同于找到全局最小值——保证的最佳可能解决方案。这个最小值是全局最小值。在一个凸问题中，如果我们找到了一个最小值，我们可以停止搜索。如果我们可以证明不存在最小值，我们也可以停止搜索。





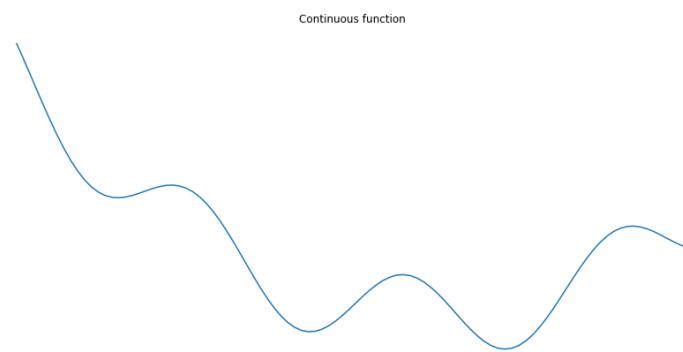
凸优化

如果目标函数是凸函数，并且任何约束形成了搜索空间的凸部分，那么这个问题就是凸优化问题。即便是对于有成千上万变量的问题，也有非常有效的方法来解决凸优化问题，这些方法包括：

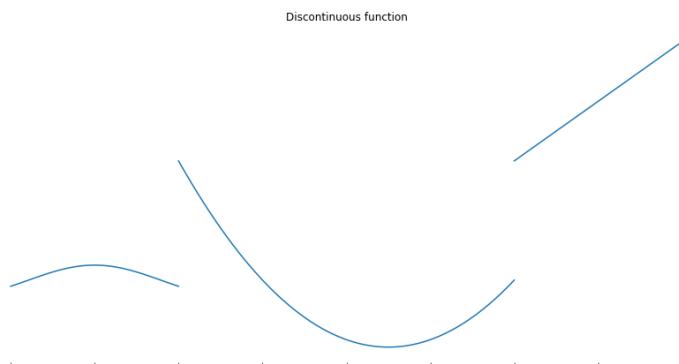
- 约束和目标函数都是线性的（线性规划）
- 二次目标函数和线性约束（二次规划）
- 或者一些特殊情况（半二次规划，二次约束的二次规划）。

连续性

如果对于 θ 的一些非常小的调整，存在一个任意小的 $L(\theta)$ 的变化，那么目标函数是连续的。



如果一个函数是不连续的，局部搜索方法不能保证收敛到一个解。对于不连续目标函数的优化通常比连续函数的优化要困难得多。这是因为对参数的任何调整都可能导致目标函数发生任意变化。最好的情况时离散且



可微

直接凸优化：最小二乘法

有时我们有一个优化问题，可以指定为一步求解。一个例子是线性最小二乘，它解决的是形如下面的目标函数：

$$\operatorname{argmin}_x L(\vec{x}) = \|\vec{Ax} - \vec{y}\|_2^2,$$

也就是说，它找到的 \vec{x} 最接近于解 $A\vec{x} = \vec{y}$ ，在最小化平方范数的意义上。平方范数只是为了使得代数推导更简单。

这个方程是凸的——它是一个二次函数，即使在多个维度中它也必须有一个唯一的全局最小值，可以直接找到。我们知道它是凸的是因为它没有高于2的次幂项（没有 x^3 等），所以它是二次的。二次函数最多只有一个最小值。

The solution is given by solving the system of **normal equations**:

$$(A^\top A) \vec{x} = A^\top \vec{y}$$

and therefore our solution is

$$\vec{x}^* = (A^\top A)^{-1} A^\top \vec{y}$$

which can also be written as

$$\vec{x}^* = A^+ \vec{y}$$

where A^+ is the **Pseudo-Inverse** of A .

解最小二乘问题 $\min_{\mathbf{w}} \psi(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 :$

$$\begin{aligned}\psi(\mathbf{w}) &= (\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y}) \\ &= \mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{w}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\mathbf{w} + \mathbf{y}^T \mathbf{y}\end{aligned}$$

注意上式中所有项均为标量，故 $\mathbf{w}^T \mathbf{X}^T \mathbf{y} = (\mathbf{y}^T \mathbf{X}\mathbf{w})^T = \mathbf{y}^T \mathbf{X}\mathbf{w}$ ，

$$\Rightarrow \psi(\mathbf{w}) = \mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \mathbf{y}^T \mathbf{y}$$

假设 \mathbf{X} 为列满秩矩阵，则 $\mathbf{X}^T \mathbf{X}$ 正定，目标函数 $\psi(\mathbf{w})$ 为凸函数。

对 $\psi(\mathbf{w})$ 求梯度并令其为 $\mathbf{0}$ ：

$$\frac{\partial \psi}{\partial \mathbf{w}} = 2\mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{X}^T \mathbf{y} = \mathbf{0}$$

$$\Rightarrow \mathbf{X}^T \mathbf{X}\mathbf{w} = \mathbf{X}^T \mathbf{y} \tag{2}$$

(2)式称为最小二乘问题的正规方程(normal equations)。由(2)式：

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

定义矩阵 \mathbf{X} 的伪逆(pseudo-inverse, Moore-Penrose inverse)为：

$$\mathbf{X}^\dagger = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$$

对于表达式 $\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w}$ ，我们可以将其看作是 $\mathbf{u}^T \mathbf{v}$ 的形式，其中 $\mathbf{u} = \mathbf{X}\mathbf{w}$ 和 $\mathbf{v} = \mathbf{X}\mathbf{w}$ 。这种情况下，导数是 $2\mathbf{X}^T \mathbf{X}\mathbf{w}$ 。

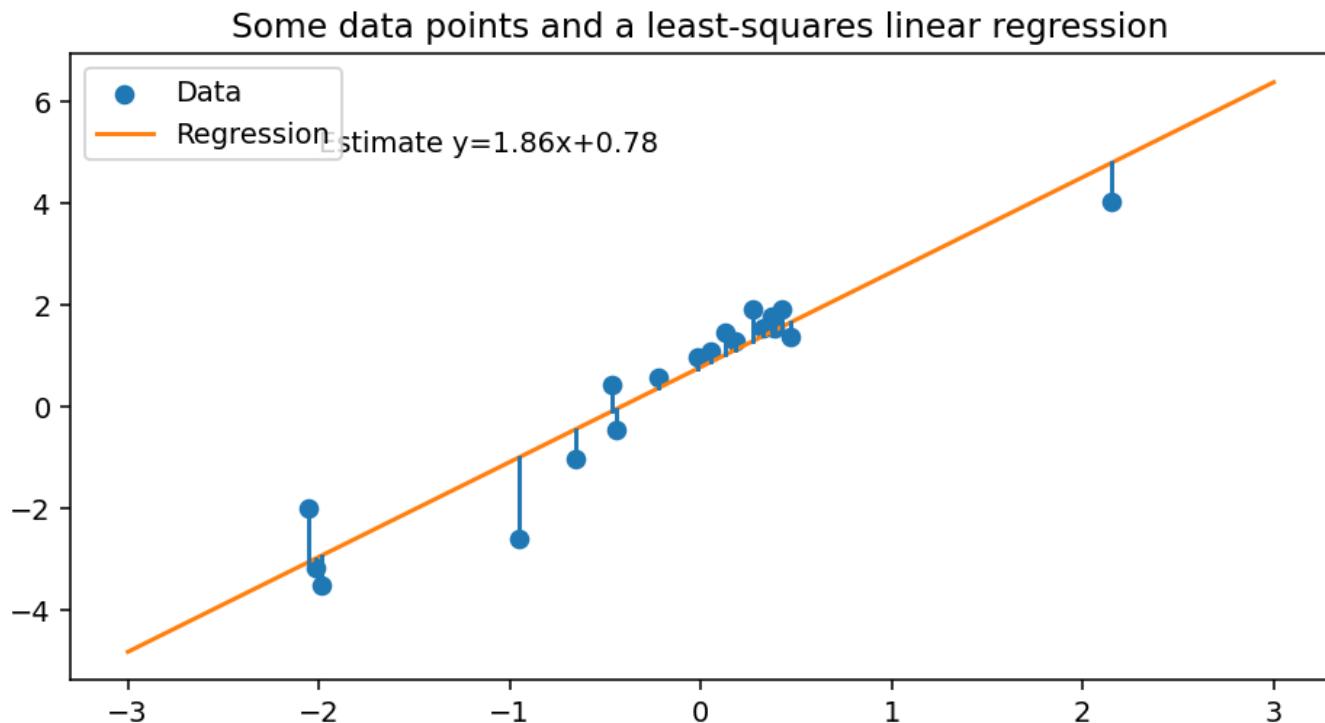
线性拟合

我们将检查这个过程中最简单的线性回归示例：找到线性方程 $y = mx + c$ 的梯度 m 和偏移量 c ，使得与一组观测到的 (x, y) 数据点的平方距离最小化。这是在 $\theta = [m, c]$ 空间中的搜索；这些是参数。

目标函数是 $L(\theta) = \sum_i (y - mx_i - c)^2$ ，对于一些已知的数据点 $[x_0, y_0], [x_1, y_1]$ ，等等。

我们可以通过 SVD 直接使用伪逆来解决这个问题。这是一个可以直接一步解决的问题。

作为示范，我们将使用方程式 $y = 2x + 1$, $m = 2$, $c = 1$ 的线，其中我们有从这个函数获得的一些带噪声的观测数据。



迭代优化

迭代优化涉及在参数空间中进行一系列步骤。有一个当前参数向量（或它们的集合），在每次迭代中调整，希望能减少目标函数的值，直到在满足一些终止条件后优化终止。

迭代优化算法：

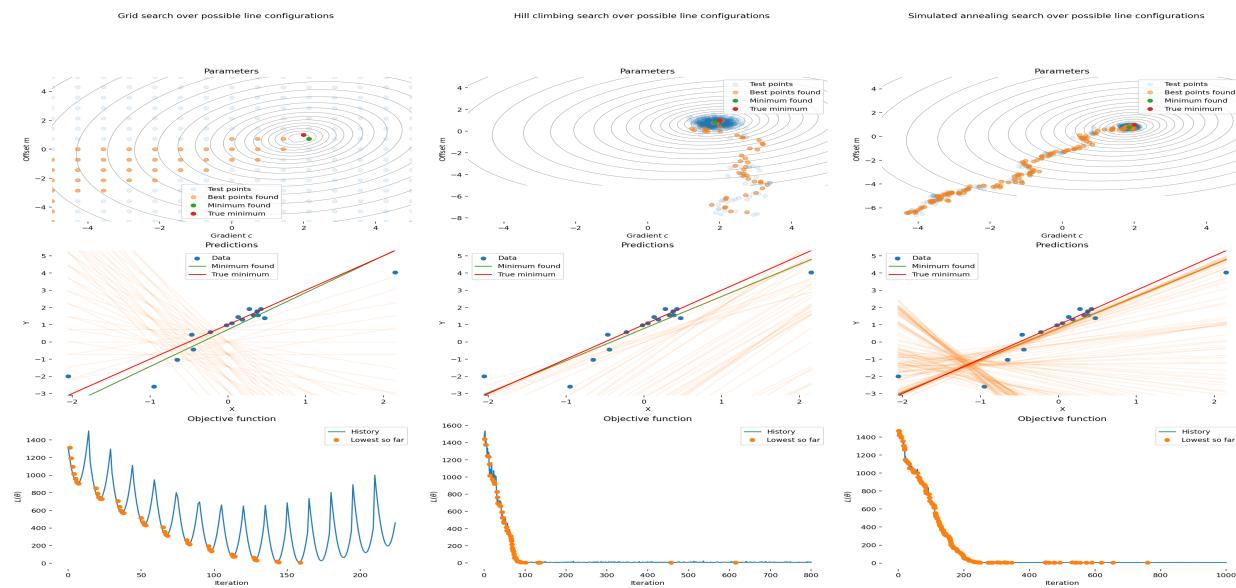
1. 选择一个起始点 x_0
2. 当目标函数在变化时
 1. 调整参数
 2. 评估目标函数
 3. 如果找到比迄今为止更好的解决方案，记录下来
3. 返回找到的最佳参数集

常规搜索：网格搜索

网格搜索是一种直接但效率不高的多维问题优化算法。参数空间通过在每个维度上均等划分可行集来进行简单采样，通常每个维度采用固定数量的划分。

在这个网格上的每个 θ 处都会评估目标函数，并追踪到目前为止发现的最低损失 θ 。这种方法简单，并且可以用于一维优化问题。它有时用于优化机器学习问题的超参数，在这些问题中目标函数可能很复杂，但找到绝对最小值并不是必需的。

网格 爬山 模拟退火

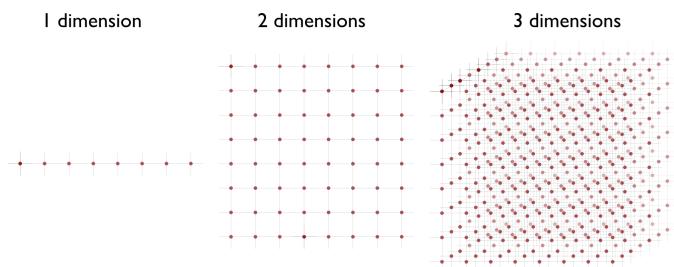


维数的爆炸

为什么要费心优化？为什么不搜索每一种可能的参数配置？

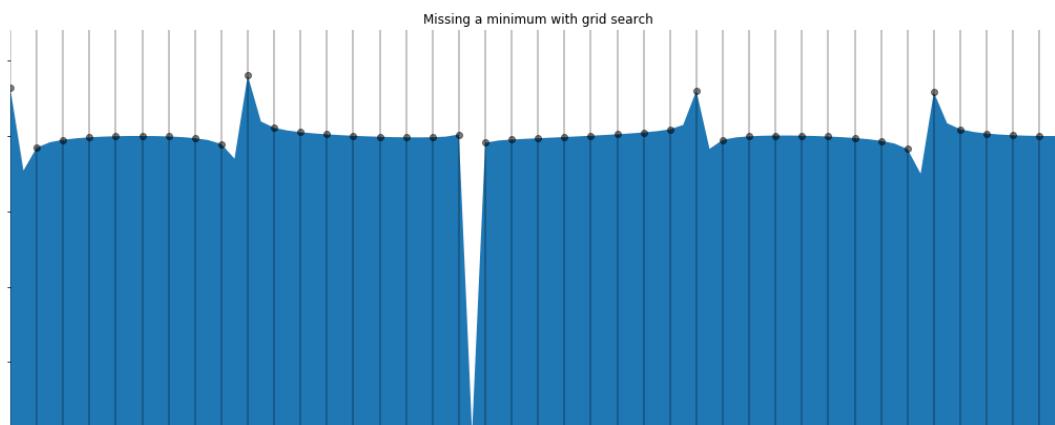
即使在相对较小的参数空间中，以及目标函数被认为是平滑的情况下，这种方法的扩展性也不好。简单地将每个维度划分为若干点（比如8个），然后尝试这样形成的点网格上的每一种组合，选择最小的结果。

虽然在一维（只检查8个点）和二维（只检查64个点）中这是可行的，但如果你有一个100维的参数空间，这种方法就完全行不通了。这将需要 8^{10}



网格搜索的密度

如果目标函数不是非常平滑，那么需要一个更密集的网格来捕捉所有的极小值。



实际的优化问题可能有数百个、数千个甚至数十亿个参数（在大型机器学习问题中）。网格搜索和类似的方法在参数空间的维度上是指数级的。

优点

- 适用于任何连续的参数空间。
- 不需要了解目标函数的知识。
- 实现起来非常简单。
- Applies to any contiguous parameter space.
- No knowledge of the objective function is required.
- Very simple to implement.

缺点

- 极其低效
- 必须提前指定搜索空间的界限。
- 极易偏向于在空间的“前角”附近找到东西。
- 高度依赖于选择的划分数量。
- 难以调整以免完全错过极小值。
- Extremely inefficient.
- Must specify bounds of search space in advance.
- Highly biased towards finding things near the "front corners" of the space.
- Highly dependent on the number of divisions chosen.
- Highly dependent on the number of divisions chosen. Difficult to adjust so as not to miss the minima completely.

超参数

网格搜索依赖于被搜索的范围以及网格划分的间距。大多数优化算法都有类似的特性，这些特性可以进行调整。

这些影响优化器寻找解决方案方式的属性称为 **超参数**。它们不是目标函数的参数，但它们确实会影响得到的结果。

一个完美的优化器将没有超参数——解决方案不应该依赖于如何找到它。但实际上，所有有用的优化器都有一些数量的超参数，这些超参数会影响它们的性能。超参数较少通常更好，因为这样调整优化器的工作就不那么繁琐。

简单随机搜索

最简单的这类算法，它除了我们可以从参数空间中随机抽取样本之外，不做任何假设，这就是**随机搜索**。

过程很简单：

- 随机猜测一个参数 θ
- 检查目标函数
- 如果 $L(\theta) < L(\theta^*)$ (之前最佳的参数 θ^*)，则设置 $\theta^* = \theta$

终止条件有很多可能性，比如在最佳损失最后一次变化后的一定次数迭代后停止。下面的简单代码使用了一个固定的迭代计数，因此不保证它会找到一个好的解决方案。

优点

- 随机搜索不会陷入局部最小值，因为它不使用任何局部结构来指导搜索。
- 不需要了解目标函数的结构 - 甚至不需要拓扑结构。
- 非常简单实现。
- 几乎总是比网格搜索好。
- Random search does not fall into local minima because it does not use any local structure to guide the search.
- No need to know the structure of the objective function - not even the topology.
- Very simple to implement.
- Almost always better than grid search.

缺点

- 极度低效，通常只在没有其他数学结构可以利用的情况下适用。
- 必须能够从参数空间中随机抽样（通常不是问题）。
- 结果不一定随时间改善。最佳结果可能在第一步或一百万步之后找到。没有办法预测优化将如何进行。
- Extremely inefficient, usually only applicable when no other mathematical structure is available.
- Must be able to sample randomly from parameter space (usually not a problem).
- Results do not necessarily improve over time. The best results may be found after the first step or after a million steps. There is no way to predict how the optimisation will proceed.

元启发优化 Metaheuristics

有许多标准的元启发式方法可以用来改善随机搜索。

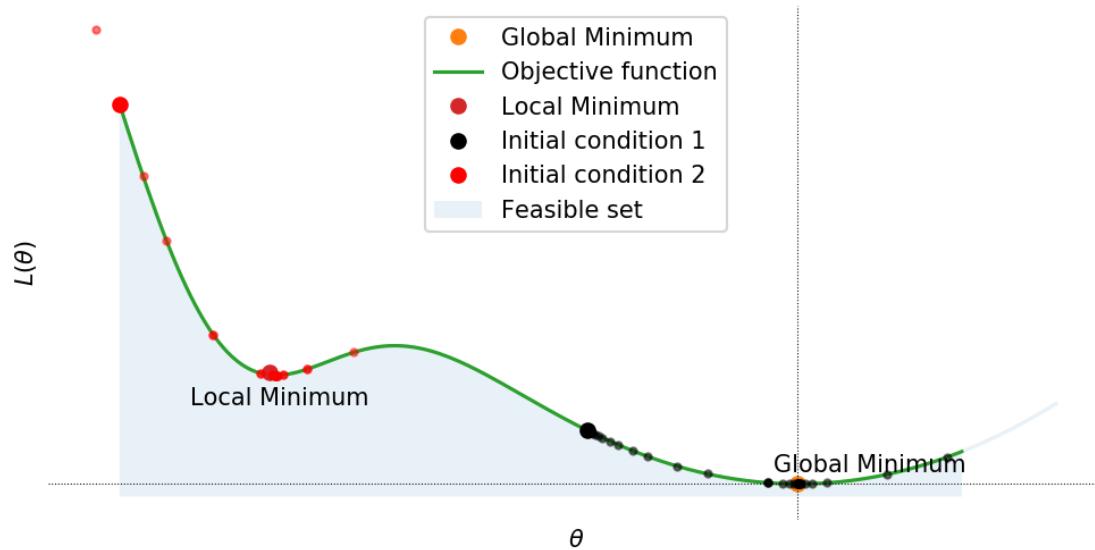
这些方法包括：

- 局部性，它利用了一个事实：对于相似的参数配置，目标函数可能会有相似的值。这假设了目标函数的连续性。
- 温度，它可以在优化过程中改变参数空间中移动的速率。这假设了局部最优解的存在。
- 种群，它可以跟踪多个同时的参数配置，并在它们之间进行选择/混合。
- 记忆，它可以记录过去的好坏的步骤，并避免/重访它们。

局部性

局部搜索 指的是那些对解进行增量式改变的算法。当目标函数具有一定的连续性时，这些算法可以比随机搜索或网格搜索更加高效。然而，它们可能会陷入局部最小值而无法达到全局最小值。由于它们通常专门用于非凸问题，这可能会成为一个问题。

这意味着优化的输出取决于**初始条件**。结果可能会从一个位置找到一个局部最小值，而从另一个起始参数集找到不同的局部最小值。



局部搜索可以被视为在参数空间中形成轨迹（一条路径），这条路径应该希望从高损失向低损失移动。

爬山算法：局部搜索

爬山算法是随机搜索的一个变种，它假设参数空间具有某种拓扑结构，因此有一个有意义的邻域概念，我们可以对参数向量进行增量式改变。爬山算法是**局部搜索**的一种形式，与其从参数空间中随机抽样，不如说是在当前最佳参数向量的附近随机抽取配置。它进行增量调整，只有在改善损失时才保留到邻近状态的转换。

简单爬山算法一次只调整参数向量的一个元素，轮流检查每个“方向”，如果情况有所改善就进行一步。**随机爬山算法**对参数向量进行随机调整，然后根据结果是否有改善来接受或拒绝这一步。

爬山这个名字来自于算法随机漫步的事实，它只采取上坡（或下坡，用于最小化）步骤。因为爬山算法是一种**局部搜索算法**，它容易被困在局部最小值中。基本的爬山算法对最小值没有防御措施，如果存在较差的解决方案很容易被困住。简单的爬山算法也可能被山脊阻挡，所有形式的爬山算法在损失函数变化缓慢的**稳定阶段**都会遇到困难。

优点

- 不比随机搜索复杂多少
- 可以比随机搜索快得多

缺点

- 难以选择调整的幅度
- 容易陷入最小值
- 在目标函数较为平坦的区域难以应对
- 要求目标函数（近似地）连续

同样，这种基础算法有许多调整方法：

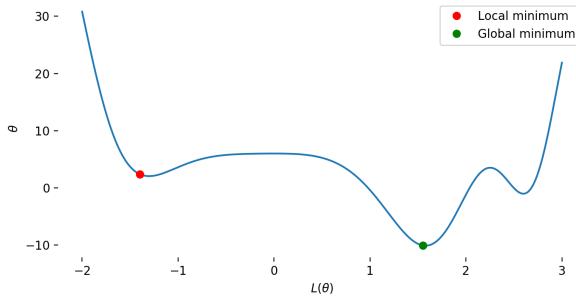
- 自适应局部搜索，其中邻域的大小可以适应性地调整（例如，如果n次迭代没有改进，增加随机步骤的大小）

- 多重启动可以用来尝试避免陷入局部最小值，方法是对随机初始猜测运行多次过程。这是另一种元启发式——应用于搜索算法本身的启发式。

模拟退火：温度时间表和逃离局部最小值

模拟退火通过有时随机上坡（而不是总是下坡）的能力来扩展爬山算法。它使用温度时间表在优化开始时允许更多上坡步骤，在过程的后期则减少这种步骤。这用于克服山脊和避免被困在局部最小值中。

这个想法是，允许在过程早期进行随机的“坏跳跃”，有助于找到更好的整体配置。



图片：爬山算法会被困在左边的局部最小值中。模拟退火有时会接受“不良”的局部变化以跨越山丘，达到更好的最小值。

“温度时间表”的概念来自于退火金属。熔融的金属分子在各处跳跃。当它们冷却时，随机的跳跃变得越来越小，因为分子锁定在一个紧密的晶格中。快速冷却的金属结构不如慢冷却的金属结构良好。

可以更多的向上动作

模拟退火使用了接受概率的概念。它不仅仅接受任何能减少损失的随机变化，还会随机接受可能暂时增加损失的跳跃的某些比例，并且随着时间的推移逐渐减少这些比例。

假定当前的损失为 $l = L(\theta)$ 和提出的新损失 $l' = L(\theta + \Delta\theta)$ ，其中 $\Delta\theta$ 代表 θ 的一个随机扰动，我们可以定义一个概率 $P(l, l', T(i))$ ，这是在第 i 次迭代时从 θ 跳到 $\Delta\theta$ 的概率。

一个常见的规则是：

如果 $l' < l$ ，则 $P(l, l', T(i)) = 1$ ，即总是下坡。 $P(l, l', T(i)) = e^{-(l'-l)/T(i)}$ ，即如果相对减少很小，也会接受向上的跳跃。

$T(i)$ 通常是迭代次数的指数衰减函数，这样一开始即使是大的跳跃也会被接受，即使它们是向上的，但随着时间的推移，向上跳跃的倾向会减少。

例如， $T(i) = e^{\frac{-i}{r}}$ ，其中 i 是迭代次数， r 是冷却率， T 是温度。

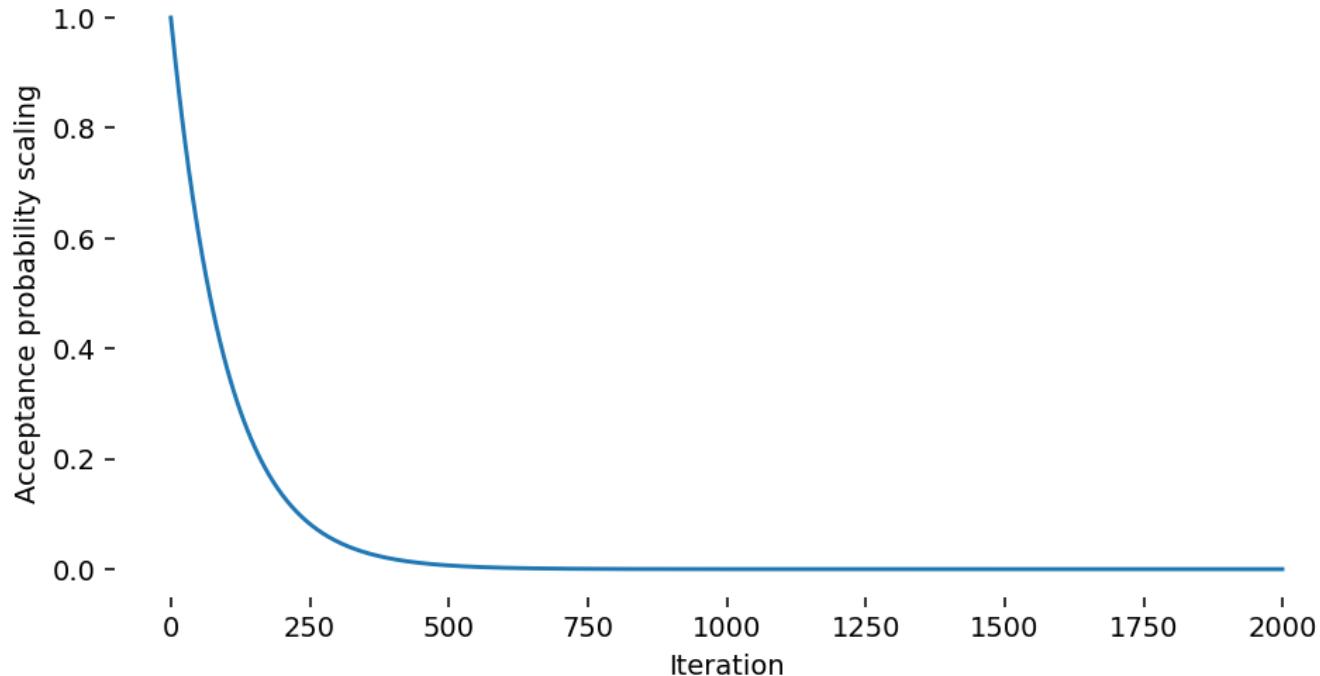
优点

- 比爬山算法对被困在局部最小值中的敏感性小得多
- 易于实施
- 经验上非常有效
- 即使在连续/离散混合设置中也相当有效。

缺点

- 取决于温度时间表和邻域函数的良好选择，这些是额外的自由参数，需要担心。
- 没有收敛性保证
- 如果不需要向上的步骤，会显得很慢。

A possible simulated annealing temperature schedule



更复杂的示例：寻找等距点

对于线性拟合来说，这并不是非常令人印象深刻，因为它是一个非常简单的凸函数；它没有局部最小值来让你陷入困境。我们可以看看找到一组等间距点的问题。这是非凸的（并且有无数个相同的最小值），解决起来比拟合一些点的线要困难得多。这是一个特别适合模拟退火风格方法的任务。

种群

另一个受自然启发的随机搜索变体是使用一个种群，即多个竞争潜在解决方案，以及应用某种类似进化的方法来解决优化问题。这包括一些：

变异（引入随机变化） **自然选择**（解决方案选择） **繁殖**（解决方案之间的交换）

这类算法通常被称为遗传算法，原因显而易见。所有遗传算法都维护着某种潜在解决方案的种群（一组向量 $\vec{\theta}_1, \vec{\theta}_2, \vec{\theta}_3, \dots$ ），以及某种规则，用于保留种群中的某些成员并淘汰其他成员。参数集合被称为解决方案的基因型。

简单的种群方法仅仅使用小的随机扰动和一个简单的选择规则，比如“保留前25%的解决方案，按损失排序”。每次迭代都会通过随机变异略微扰动解决方案，淘汰最弱的解决方案，然后复制剩余的“最适应”的解决方案多次，以产生下一步的后代。种群大小从迭代到迭代保持不变。这只是带有种群的随机局部搜索。其想法是，这可以比简单的局部搜索探索更大的空间区域，并在此期间维护关于什么可能是好的多个可能的假设。

遗传算法：种群搜索

优点

- 易于理解，适用于许多问题。
- 只需要对目标函数有最基本的了解。
- 可以应用于具有离散和连续部分的问题。
- 对局部最小值有一定的鲁棒性，尽管难以控制。
- 在参数化方面有极大的灵活性：变异方案、交叉方案、适应度函数、选择函数等。

缺点

- 需要调整许多极大影响优化性能的“超参数”。你应该如何选择它们？
- 无收敛保证；随意。
- 与使用对目标函数更深刻理解的方法相比（非常）慢。
- 需要进行大量的目标函数评估：每个种群成员每次迭代一次。

记忆

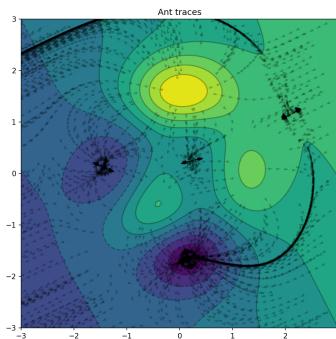
我们迄今为止看到的优化算法都是无记忆的。它们探索解空间的某些部分，检查损失，然后继续前进。他们可能会一次又一次地检查相同或非常相似的解决方案。使用某种形式的记忆可以缓解这种低效率，其中优化器记住参数空间中“好”的和“坏”的部分，并使用这个记忆做出决策。特别是，我们想要记住解空间中好的路径。

记忆+种群

蚁群优化

蚂蚁非常擅长寻找食物（探索），然后引导整个蚁群到食物来源去探索和提取所有食物（开发）。它们能够做到这一点，而无需任何复杂的协调。相反，蚂蚁四处游荡，直到它们找到一些东西吃。然后，它们在回到蚁丘的路上留下了信息素（气味）的痕迹。其他蚂蚁可以跟随这个痕迹找到食物并检查整个区域是否有任何特别美味的东西。

蚁群优化结合了记忆和种群启发式方法。它使用了信息素机制来优化问题：



信息素机制：一种自发的、间接的协调机制，通过行为或动作在环境中留下的痕迹刺激随后的动作执行。

在优化方面，这意味着：

- 拥有一群参数集合（“蚂蚁”）
- 记忆穿过空间的好路径（“信息素”）

找到空间中好部分（即低目标函数值）的蚂蚁会留下积极的“信息素”，通过存储标记向量。其他蚂蚁将朝向这些信息素移动，并最终跟随通往好解决方案的路径。随着时间的推移（即迭代次数增加），信息素会蒸发，以免蚂蚁被限制在空间的一小部分。我们使用辅助数据结构来记忆参数空间中的好路径，而不是利用物理环境，以避免重复搜索。

ACO(蚁群优化)特别适合寻径和路线查找算法，其中信息素痕迹的记忆结构对应于解决方案结构。

优点

- 在被大而狭窄的谷地分隔开的空间中可能非常有效。
- 如果信息素有效，相比遗传算法可以使用更少的目标函数评估次数。
- 当它有效时，其效果通常非常显著。

缺点

- 实现起来算法复杂度适中。
- 没有收敛性保证；特设的。
- 相比遗传算法有更多的超参数。

优化质量

收敛性

优化算法被称为收敛到一个解决方案。在凸优化中，这意味着已找到全局最小值，问题已解决。在非凸优化中，这意味着找到了局部最小值，算法无法摆脱该局部最小值。

一个好的优化算法能够快速收敛。这意味着目标函数的下降应该是陡峭的，以便每次迭代都能带来较大的变化。一个不好的优化算法根本不会收敛（它可能会永远徘徊，或发散到无穷大）。许多优化算法只在某些条件下才会收敛；收敛性取决于优化的初始条件。

收敛性保证

一些优化算法如果存在解决方案则保证收敛；而其他一些（如大多数启发式优化算法）即使问题有解也不保证会收敛。例如，随机搜索可能会永远漫游在可能性空间中，永远找不到最小化（或甚至减少）损失的特定配置。

对于迭代解决方案，目标函数值对迭代次数的图表是诊断收敛问题的有用工具。理想情况下，损失应该尽可能快地下降。

深度神经网络

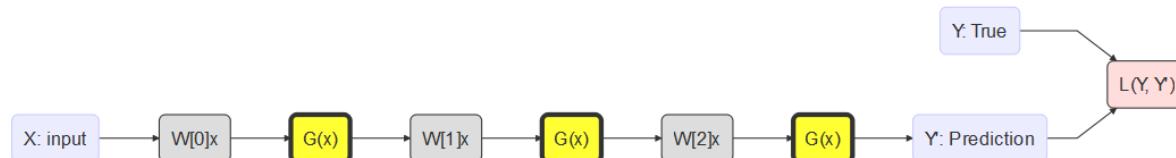
深度学习或深度神经网络已成为现代机器学习研究的重要组成部分。它们在语音识别、机器翻译、图像分类和图像合成等领域取得了惊人的成功。深度学习的基本问题是寻找一个近似函数。在一个简单的模型中，这可能如下工作：给定一些观察值 $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n$ 和一些相应的输出 y_1, y_2, \dots, y_n ，找到一个函数 $y' = f(\vec{x}; \theta)$ ，带有参数 θ ，以便我们拥有

$$\theta^* = \operatorname{argmin}_{\theta} \sum_i ||f(\vec{x}_i; \theta) - y_i||$$

其中距离是衡量 f 的输出和期望输出 y_i 有多接近的某种度量（具体使用的距离会随问题而变）。其思想是我们可以学习 f ，使我们能够将变换泛化到我们尚未看到的 \vec{x} 。这显然是一个优化问题。但是深度神经网络可以有数十亿的参数——一个非常长的 θ 向量。在如此庞大的参数空间内，如何在合理的时间内进行优化？上述例子中有数千万个参数需要调整。

反向传播 Backpropagation

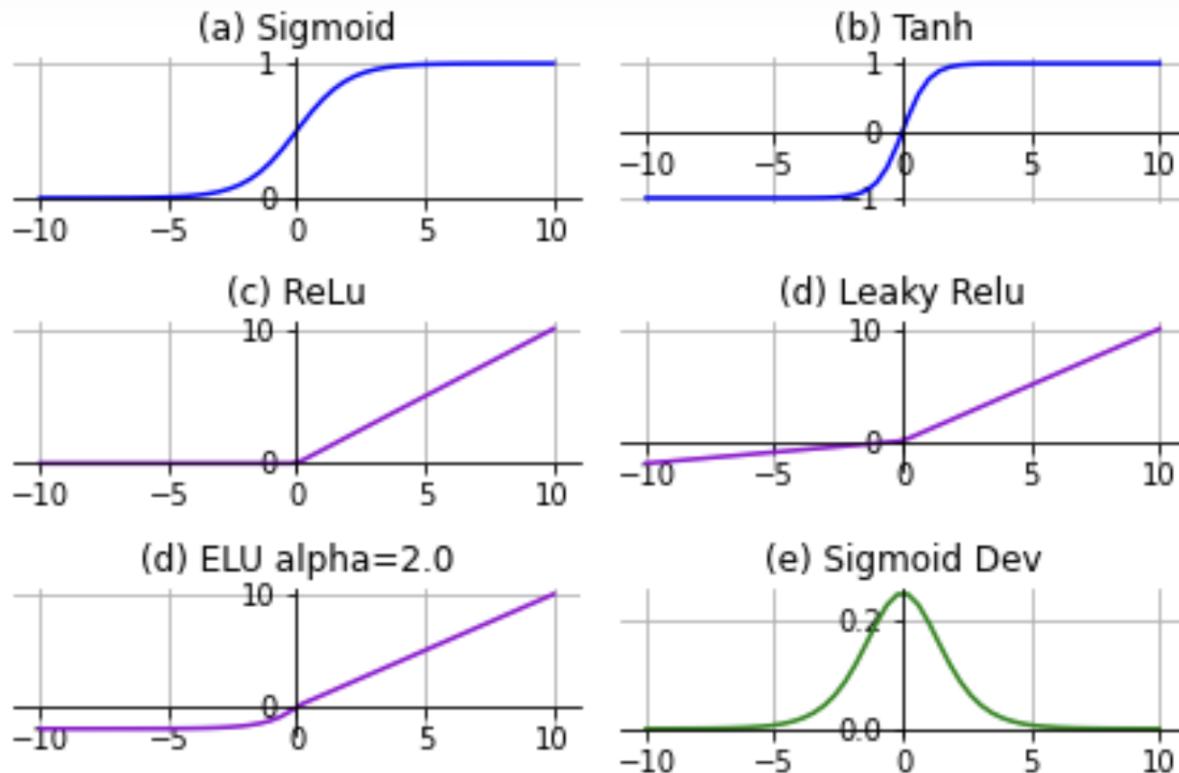
答案是这些网络以一种非常简单（但聪明）的方式构建。传统的神经网络由一系列层组成，每一层都是一个线性映射（仅仅是矩阵乘法）后面跟着一个简单的、固定的非线性函数。想象一下：旋转、伸展（线性映射）和折叠（简单的固定非线性折叠）。一个层的输出是下一个层的输入。



图像：一个3层深度网络。每一层由应用于前一层输入 x 的线性映射 W 组成，然后是一个固定的非线性函数 $G(\vec{x})$

每一层中的线性映射（当然）由一个矩阵指定，这个矩阵被称为权重矩阵。网络完全由每层的权重矩阵的条目参数化（这些矩阵的所有条目可以视为参数向量 θ ）。非线性函数 $G(\vec{x})$ 对于每一层都是固定的，不能变化；它通常是一个以某种方式“压缩”输出范围的简单函数（比如 \tanh 、 relu 或 sigmoid ——你不需要了解这些）。只有在优化过程中权重矩阵会发生变化。

$$\vec{y}_i = G(W_i \vec{x}_i + \vec{b}_i)$$



而这种特定的构造（在某些条件下）具有巨大的优势，即目标函数对权重的导数可以同时为网络中的每一个权重计算，即使是多个层组合在一起的情况。执行此操作的算法称为反向传播，它是一种自动微分算法。“对权重的导数”意味着我们可以一次性知道每个权重对预测的影响程度，对于网络中的所有权重。

如果我们想象将所有权重矩阵的元素串联成一个单一向量 θ ，我们就可以获得目标函数相对于 θ 的梯度。这使得优化变得“容易”；我们只需朝着最快下降的方向前进即可。

为什么不使用启发式搜索算法

启发式搜索方法如随机搜索、模拟退火和遗传算法易于理解、实现简单，且几乎没有限制，它们可以适用于哪些问题。那么为什么不总是使用这些方法呢？

- 它们可能非常慢；可能需要很多次迭代才能接近最小值，并且每次迭代都需要大量计算。无法保证收敛，甚至无法保证进步。搜索可能会卡住，或者在高原地区缓慢漂移。
- 有大量的超参数可以调整（温度计划、种群大小、记忆结构等）。这些参数应该如何选择？这些参数的最佳选择本身就是一个优化问题。
- 对于像深度神经网络这样的优化问题，启发式搜索完全不够用。在训练有数百万参数的网络时，它简直是太慢了，无法取得进展。相反，应用了一阶优化。我们今天将讨论的一阶算法可以比启发式搜索快几个数量级。

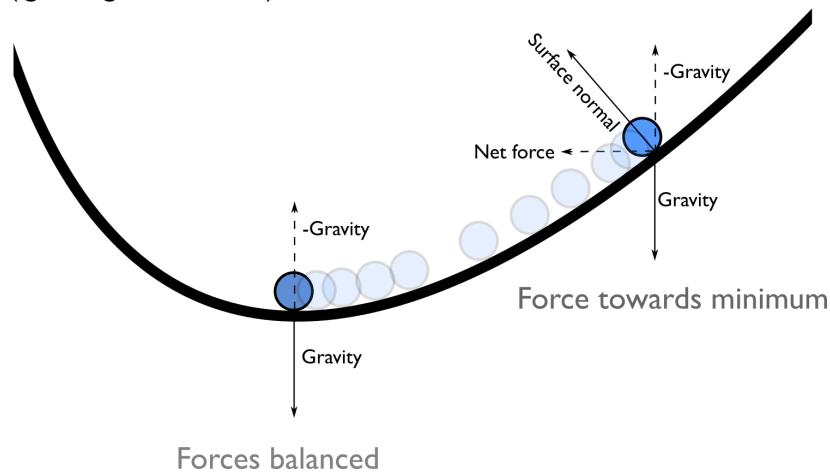
注：如果目标函数已知是凸的，约束也是凸的，那么还有更好的优化方法，这些方法通常可以非常快速地运行，并保证收敛。

扔一个球：物理层面上的搜索

通过考虑一个在（光滑）表面上滚动的球来形成对高阶优化的直觉，这个表面代表了一个2D域上目标函数的值（即，如果我们有一个包含两个元素的参数向量 θ ）。

重力对球施加一个垂直于表面平面的力。表面沿着表面法线的方向对球施加力，表面法线是一个从表面“直接指出”的向量。这导致了在表面最陡峭斜坡方向上的力分量，使球朝那个方向加速。这个梯度向量总是指向最陡斜坡的方向。

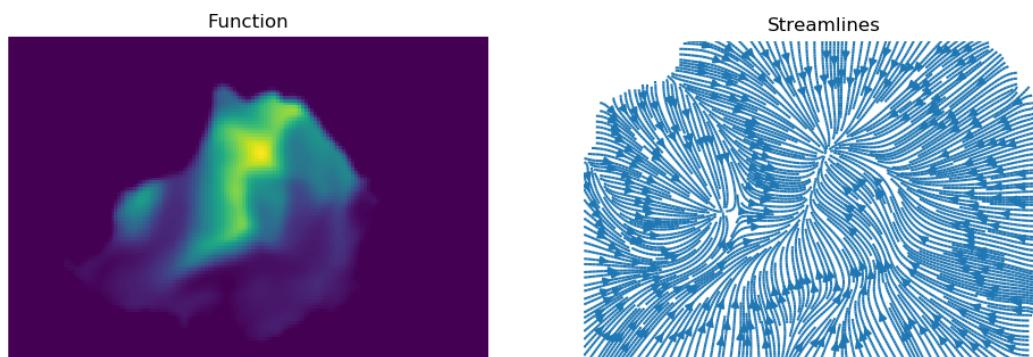
Physical optimisation (ignoring momentum)



球最终会稳定在一个力平衡的配置中。例如，如果球在一个最小点稳定下来，表面的法线与重力平行，就会发生这种情况。

Attractors: flowing towards a solution

我们可以将其视为一个**Attractors**，它将我们的搜索吸引到解决方案。搜索的轨迹与目标函数的“流场”平行。我们的物理直觉是，我们可以通过沿着这些流线滚动进入“盆地”来找到最小值。



雅可比矩阵：导数矩阵

如果 $f(x)$ 是一个标量 x 的标量函数， $f'(x)$ 是 f 关于 x 的一阶导数，即 $\frac{d}{dx} f(x)$ 。二阶导数写作 $f''(x) = \frac{d^2}{dx^2} f(x)$ 。

如果我们将其推广到向量函数 $\vec{y} = f(\vec{x})$ ，那么我们在任意特定输入 \vec{x} 时，有输入的每个分量与输出的每个分量之间的变化率（导数）。我们可以将这些导数信息收集到一个称为雅可比矩阵的矩阵中，它描述了*在特定点

\vec{x}^* 的斜率。如果输入 $\vec{x} \in \mathbb{R}^n$ 且输出 $\vec{y} \in \mathbb{R}^m$, 那么我们有一个 $m \times n$ 矩阵:

$$f'(\vec{x}) = \vec{J} = \begin{bmatrix} \frac{\partial y_0}{\partial x_0} & \frac{\partial y_0}{\partial x_1} & \cdots & \frac{\partial y_0}{\partial x_n} \\ \frac{\partial y_1}{\partial x_0} & \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \cdots \\ \frac{\partial y_m}{\partial x_0} & \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

这简单地告诉我们, 当我们改变输入的任何一个分量时, 输出的每个分量变化了多少——向量值函数的广义“斜率”。这是在点 \vec{x} 处表征向量函数变化的一种非常重要的方法, 并且在许多上下文中广泛使用。在 f 将 \mathbb{R}^n 映射到 \mathbb{R}^m (从一个向量空间映射到相同的向量空间) 的情况下, 我们有一个正方形的 $n \times n$ 矩阵 \vec{J} , 我们可以用它来做标准的操作, 比如计算行列式, 进行特征分解或 (在某些情况下) 求逆。

在许多情况下, 我们有一个非常简单的雅可比矩阵: 只有一个单独的行。这适用于我们有一个标量函数 $y = f(\vec{x})$ 的情况, 其中 $y \in \mathbb{R}$ (即从 n 维输入到一维输出)。这是我们有一个损失函数 $L(\theta)$ 作为向量输入的标量函数的情况。在这种情况下, 我们有一个单行雅可比矩阵: 梯度向量。

梯度向量: 雅可比方程的一行

- $\nabla f(\vec{x})$ 是向量函数的 (标量) 函数的梯度向量, 相当于向量函数的一阶导数。我们对 \vec{x} 的每个分量都有一个 (偏) 导数。这告诉我们, 如果我们对每个维度独立地进行微小的变化, $f(\vec{x})$ 会变化多少。注意, 在这门课程中我们只处理输出为标量但输入为向量的函数 $f(x)$ 。我们将处理参数向量 θ 的标量目标函数 $L(\theta)$

$$\nabla L(\vec{\theta}) = \left[\frac{\partial L(\vec{\theta})}{\partial \theta_1}, \frac{\partial L(\vec{\theta})}{\partial \theta_2}, \dots, \frac{\partial L(\vec{\theta})}{\partial \theta_n} \right]$$

- 如果 $L(\theta)$ 是一个映射 $\mathbb{R}^n \rightarrow \mathbb{R}$ (即一个标量函数, 比如一个普通的目标函数), 那么 $\nabla L(\theta)$ 是一个值为向量的映射 $\mathbb{R}^n \rightarrow \mathbb{R}^n$;
- 如果 $L(\theta)$ 是一个映射 $\mathbb{R}^n \rightarrow \mathbb{R}^m$, 那么 $\nabla L(\theta)$ 是一个值为矩阵的映射 $\mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}$;

$\nabla L(\theta)$ 是一个指向 $L(\theta)$ 变化最陡的方向的向量。

海森矩阵: 梯度向量的雅可比矩阵

Hessian: Jacobian of the gradient vector

- $\nabla^2 f(\vec{x})$ 是向量函数的 (标量) 函数的海森矩阵, 相当于向量函数的二阶导数。按照我们上面的规则, 它只是一个值为向量的函数的雅可比矩阵, 所以我们知道:
- $\nabla^2 L(\theta)$ 是一个值为矩阵的映射 $\mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$ 这很重要, 因为我们可以看到, 即使是标量值函数的二阶导数也与其输入的维度呈二次方地增长!

(如果原始函数是一个向量，我们将得到一个海森张量而不是矩阵)。

$$H(L) = \nabla \nabla L(\vec{\theta}) = \nabla^2 L(\vec{\theta}) = \begin{bmatrix} \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1^2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2^2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_n} \\ \vdots & & & & \\ \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n^2} \end{bmatrix}$$

可微的目标函数

对于某些目标函数，我们可以计算出目标函数相对于参数 θ 的（精确的）导数。例如，如果目标函数有一个单一的标量参数 $\theta \in \mathbb{R}$ ，函数为：

$$L(\theta) = \theta^2$$

那么，从基础微积分知识来看，关于 θ 的导数就是：

$$L'(\theta) = 2\theta.$$

如果我们知道导数，我们可以使用它来朝着“好的方向”移动——沿着目标函数的斜坡向下移动到最小值。

对于多维目标函数（其中 θ 有多个分量），问题会稍微复杂一些，我们会有一个**梯度向量**而不是简单的标量导数（写作 $\nabla L(\theta)$ ）。然而，同样的原理适用。

阶数：零阶、一阶、二阶 迭代算法可以根据它们所需的导数阶数进行分类：

零阶优化算法只需要评估目标函数 $L(\theta)$ 。例子包括随机搜索和模拟退火。

一阶优化算法需要评估 $L(\theta)$ 及其导数 $\nabla L(\theta)$ 。这一类包括**梯度下降方法家族**。

二阶优化算法需要评估 $L(\theta)$ 、 $\nabla L(\theta)$ 以及 $\nabla^2 L(\theta)$ 。这些方法包括**类牛顿优化**。

利用导数的优化

如果我们知道（或可以计算）目标函数的梯度，我们就知道在任何给定点的函数的斜率。这给了我们两个信息：

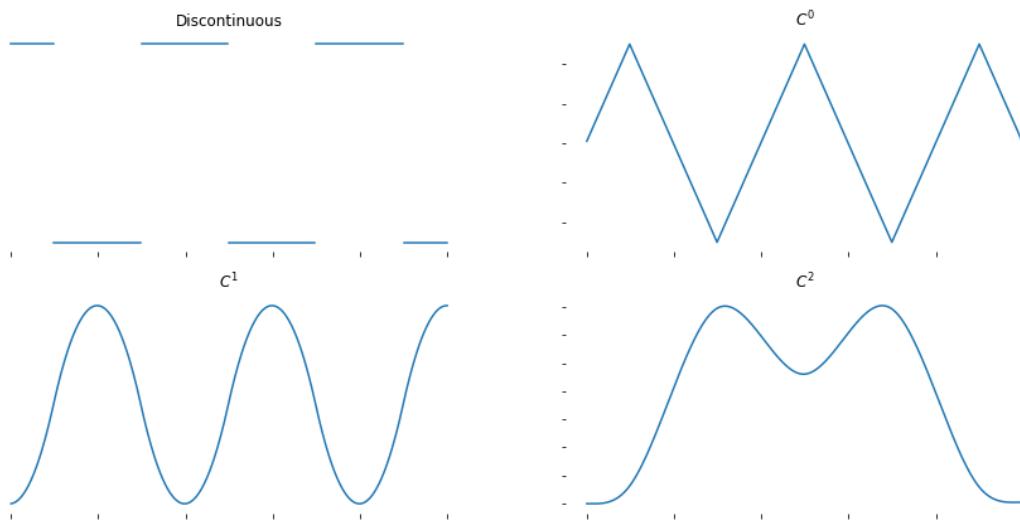
- 最快增长的方向和
- 该斜率的陡峭程度。这是微积分的主要应用之一。

知道目标函数的导数足以大大提高优化的效率。

状态

可微性

一个光滑函数具有连续的导数，直到某个阶数。光滑函数通常更容易进行迭代优化，因为当前近似的小变化很可能导致目标函数的小变化。如果函数的第n阶导数是连续的，我们称这个函数是** C^n 连续的**。



图片：从左到右，从上到下分别是不连续的、 C^0 、 C^1 、 C^2 连续函数

拥有连续导数与知道这些导数是什么之间有区别。

一阶优化使用目标函数相对于参数的（一阶）导数。这些技术只能在目标函数至少是：

C^1 连续的，即函数及其导数中无处有阶跃变化 可微的，即梯度在任何地方都是定义好的（尽管我们会看到，这些约束在实践中可以有所放宽）。

许多目标函数满足这些条件，一阶方法比零阶方法有效得多。对于特定类别的函数（例如凸函数），已知特定一阶优化器收敛所需的步骤数量有明确的界限。

利普希茨连续性

一阶（和更高阶）连续优化算法对函数的要求比仅仅是 C^1 连续性更高，它们要求函数具有利普希茨连续性。

对于函数 $\mathbb{R}^n \rightarrow \mathbb{R}$ （即我们关心的目标函数 $L(\theta)$ ），这等同于说梯度是有界的，函数的变化速率不能超过某个常数；存在一个最大的陡度。对于所有的 i 和某个固定的 K ，有 $\frac{\partial L(\theta)}{\partial i} \leq K$ 。

Lipschitz 常数

我们可以想象在一个表面上滑动一个特定陡度的圆锥体。我们可以检查它是否曾经触摸到表面。这是衡量函数陡峭程度的一个指标；或者等价于一阶导数的上界。函数 $f(x)$ 的利普希茨常数 K 是一次只触摸函数一次的圆锥体的宽度的度量。这是一个衡量函数平滑程度的指标，或者等价于目标函数在其定义域上任何一点的最大陡度。它可以定义为：

$$K = \sup \left[\frac{|f(x) - f(y)|}{|x - y|} \right],$$

其中 \sup 是上确界；比这个函数的每一个值都大的最小值。

较小的 K 意味着函数更加光滑。 $K = 0$ 完全平坦。我们将假设我们要处理的函数有一些有限的 K ，尽管其值可能不会被精确知道。

Lipschitz 连续 是比一致连续更强的条件，但比Lipschitz 连续更强的约束是可导，这都是建立在数学的基础上。

分析导数

如果我们有分析导数（即我们知道函数的导数的封闭形式；我们可以直接写下数学公式），你可能还记得高中数学优化的过程：

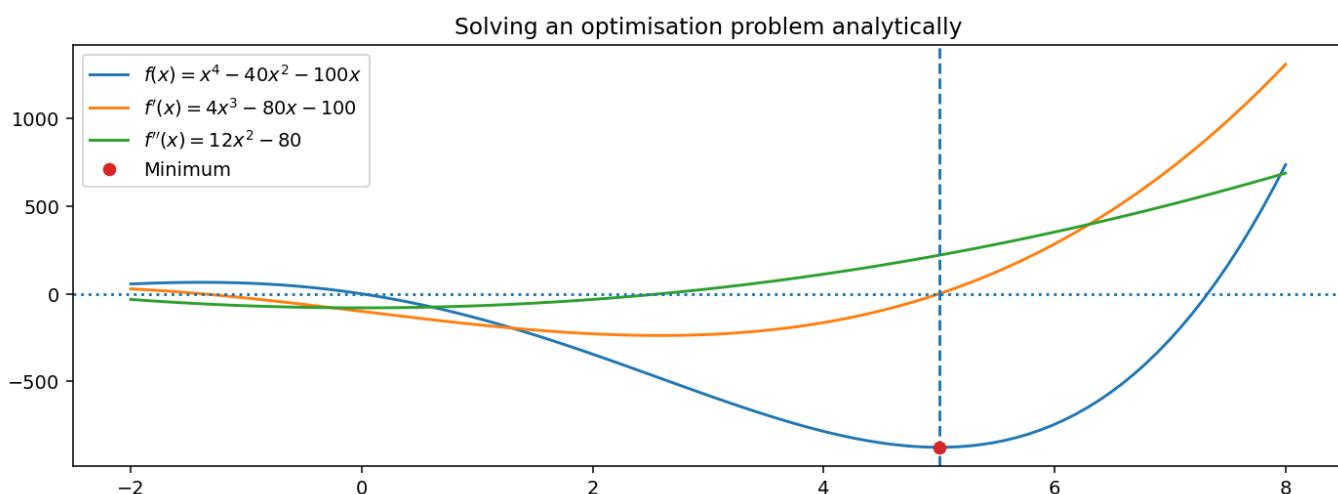
- 计算导数 $f'(x) = \frac{d}{dx} f(x)$
- 解导数为零的方程（即解 x 使得 $f'(x) = 0$ ）。这找到了函数的所有转折点或极值点。
- 然后检查解中是否有二阶导数为正的 $f''(x) > 0$ ，这表示解是最小值。

Example

This is how we might find the minimum of $f(x) = x^4 - 40x^2 - 100x$. The derivative is:

$f'(x) = 4x^3 - 80x - 100$ and the second derivative is $f''(x) = 12x^2 - 80$. We can solve for:

$f'(x) = 4x^3 - 20x - 100 = 0$ and check the sign of $f''(x) = 12x^2 - 20$ to find if we have a minimum.



可计算的精确导数

解析导数方法根本不需要迭代。只要解出导数，我们就能立刻得到解。但通常我们没有简单的导数解，我们可以在特定点评估导数；我们有精确的逐点导数。我们可以评估函数 $f'(x)$ 在任何 x 处的值，但不能用封闭形式写下它。在这种情况下，我们仍然可以通过采取步骤使我们“尽可能快地下山”来大大加速优化。这要求我们能够在目标函数的任何点计算梯度。

梯度：导数向量

我们将处理输入向量并输出标量的目标函数：

```
# vector -> scalar
def objective(theta):
    ...
    return score
```

我们希望生成函数：

```
# vector -> vector
def grad_objective(theta):
    ...
    return score_gradient
```

数学上，我们可以将导数向量写为：

$$\nabla L(\vec{\theta}) = \left[\frac{\partial L(\vec{\theta})}{\partial \theta_1}, \frac{\partial L(\vec{\theta})}{\partial \theta_2}, \dots, \frac{\partial L(\vec{\theta})}{\partial \theta_n} \right]$$

注意： $\frac{\partial L(\vec{\theta})}{\partial \theta_1}$ 仅表示在点 $\vec{\theta}$ 处，沿 θ_1 方向上 L 的变化。

这个向量 $\nabla L(\vec{\theta})$ 被称为梯度或梯度向量。在任意给定点，函数的梯度指向函数增长最快的方向。这个向量的大小是函数变化的速率（“陡度”）。

梯度下降

基本的一阶算法称为梯度下降，它非常简单，从某个初始猜测 $\theta^{(0)}$ 开始：

$$\theta^{(i+1)} = \theta^{(i)} - \delta \nabla L(\theta^{(i)})$$

其中 δ 是一个缩放超参数——步长。步长可能是固定的，也可能根据像线搜索这样的算法自适应地选择。

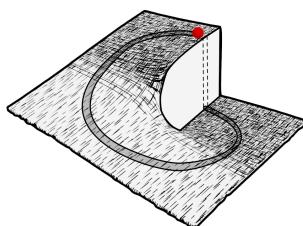
这意味着优化器会进行移动，在那里目标函数下降得最快。

用更简单的术语来说：

从某个地方开始 $\theta^{(0)}$ 重复以下步骤：检查在每个方向上地面有多陡 $v = \nabla L(\theta^{(i)})$ 在最陡的方向 v 上移动一小步 δ ，以找到新的位置 $\theta^{(i+1)}$ 。符号说明： $\theta^{(i)}$ 并不意味着 θ 的第 i 次幂，而只是迭代序列中的第 i 个 $\vec{\theta}$ ：
 $\vec{\theta}^{(0)}, \vec{\theta}^{(1)}, \vec{\theta}^{(2)}, \dots$

下坡不总是最短路径

虽然梯度下降可以非常快，但沿梯度方向并不一定是达到最小值的最快路线。在下面的例子中，从红点到最小值的路线非常短。然而，沿梯度方向，却需要绕一个很远的路才能到达最小值。



图像：梯度下降意味着沿最陡峭的坡下降——但这不总是最短的路线

不过，它通常比盲目地四处跳跃希望到达底部要快得多！

Implementing gradient descent

The implementation follows directly from the equations:

```

import utils.history
import imp; imp.reload(utils.history)
from utils.history import History

#
# note: fixed step size isn't a good idea!

def gradient_descent(L, dL,
                      theta_0,
                      delta, tol=1e-4):
    """
    L: scalar loss function
    dL: gradient of loss function w.r.t parameters
    theta_0: starting point
    delta: step size
    tol: termination condition;
          when change in loss is less than tol, stop iterating
    """
    theta = np.array(theta_0) # copy theta_0
    o = History()
    o.track(np.array(theta), L(theta))

    # while the loss changes
    while np.sum(np.abs(dL(theta)))>tol:
        # step along the derivative
        theta += -delta * dL(theta)
        o.track(np.array(theta), L(theta))

    return o.finalise()

def L(theta):
    return theta**2

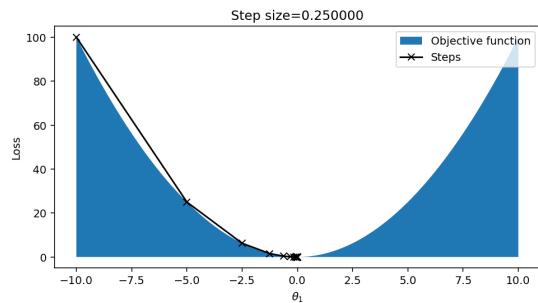
def dL(theta):
    # we can differentiate L(theta) in our heads :)
    return 2*theta

def plot_gradient_descent(xs, L, dL, x_0, step):
    # do descent
    res = gradient_descent(L, dL, x_0, step)

    # plot
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.fill_between(xs, L(xs), label="Objective function")
    ax.set_title("Step size=%f" % step)
    ax.set_xlabel("$\theta_1$")
    ax.set_ylabel("Loss")
    ax.plot(res.best_thetas, res.best_losses, 'k-x', label='Steps')
    ax.legend()
    print("Converged in {} steps".format(res.iters))

```

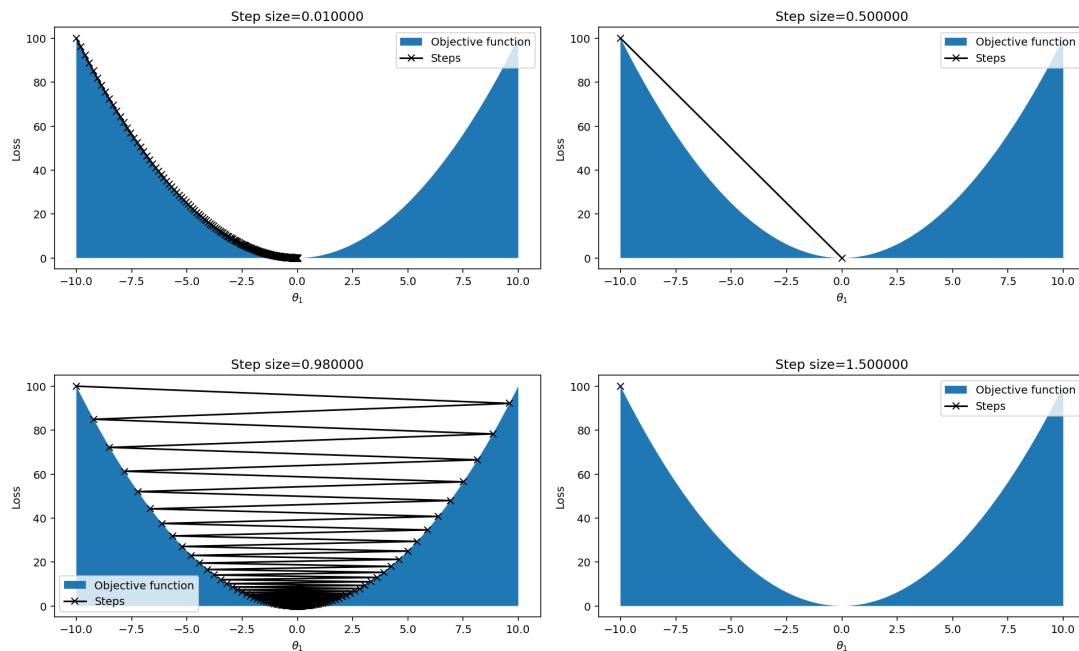
```
xs = np.linspace(-10, 10, 200)
```



步长的重要性

步长 δ 对成功至关重要。如果它太小，步伐会非常小，收敛会很慢。如果步伐太大，优化的行为可能变得相当不可预测。如果在一步的空间内梯度函数变化显著（例如，如果梯度在步长中改变符号），就会发生这种情况。The step size δ is critical to success. If it is too small, the pace will be very small and convergence will be slow. If the pace is too large, the behaviour of the optimization can become quite unpredictable. This can happen if the gradient function changes significantly in the space of one step (e.g., if the gradient changes sign in the step).

步长为0.01 步长为0.5 步长过长 - 震荡现象 步长太大 -发散



与Lipschitz 常数的联系

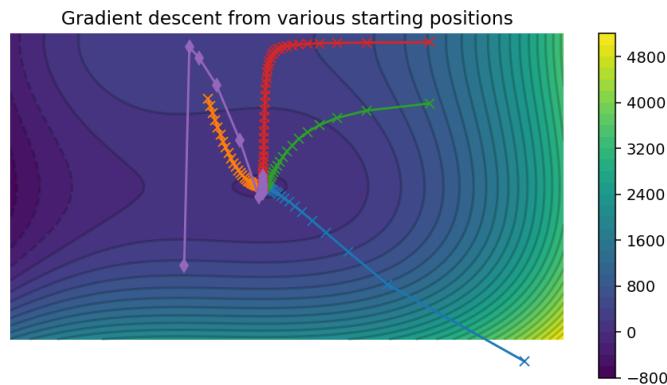
的确，步长 δ 与目标函数的Lipschitz常数 K 直接相关。理论上，如果我们知道Lipschitz常数，我们可以选择一个保证收敛性的步长，不会太大以致于引起振荡，也不会太小导致收敛过慢。具体来说，步长通常选择为Lipschitz常数的倒数。

然而，实际情况是在许多现实世界的优化任务中，我们很少能精确知道 K ，因此步长往往是通过近似方法（如线搜索）来设定的。线搜索是一种动态调整步长的方法，它在每次迭代中尝试找到一个能够减少目标函数值的步长。这样做的好处是可以适应目标函数的局部性质，但也增加了每次迭代的计算负担。

在二维空间中的梯度下降

这种技术可以扩展到任意数量的维度，只要我们能够在参数空间的任意点获得梯度向量。我们只是有一个梯度向量 $\nabla L(\theta)$ 而不是简单的一维导数。代码不需要改变。

在二维空间中，梯度下降算法会考虑两个参数的梯度，并在这两个方向上同时进行优化。这意味着每一次迭代中，算法都会根据两个参数的偏导数来更新这两个参数的值。在高维空间中，算法的这个性质是一样的，每个参数都根据它的偏导数进行更新，尽管在更高的维度中，直观理解和可视化就更加困难。



目标函数的梯度

为了能够进行一阶优化，必须能够得到目标函数的导数。显然，这并不直接适用于经验优化（例如，在实验中测试组件质量的真实世界制造过程——没有导数），但在许多我们拥有可优化的计算模型的情况下，这种方法是可行的。这也是在优化过程中倾向于构建模型的另一个原因。

当目标函数是一个已知的数学表达式时，我们可以直接计算它的梯度。在机器学习等领域，目标函数通常是一个损失函数，它衡量的是模型预测和真实数据之间的差异。在这些情况下，可以使用自动微分工具（如 TensorFlow 或 PyTorch）来计算梯度，即使是对非常复杂的函数。

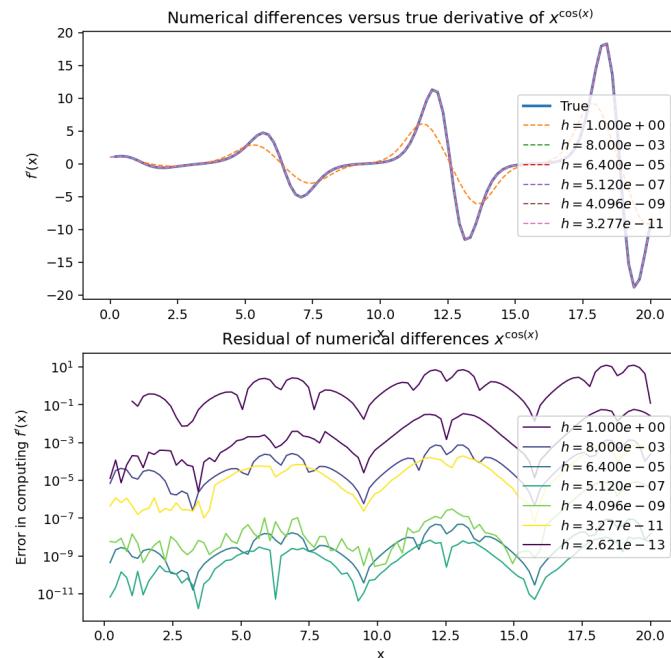
在那些无法直接计算梯度的情况下，比如实验或制造过程优化，可能会采用基于梯度的估计方法，如有限差分法，或者采用其他类型的优化方法，如进化算法或模拟退火，它们不需要目标函数的梯度。

为什么不使用数值差分法？

函数 $f(x)$ 微分的定义是众所周知的公式：

鉴于这个定义，如果我们能够在任何地方评估 $L(\theta)$ ，为什么我们需要知道真正的导数 $\nabla L(\theta)$ 呢？为什么不仅仅是评估 $L(\theta + h)$ 和 $L(\theta - h)$ 对于一些小的 h 呢？这种方法称为数值微分，这些是有限差分。

$$\frac{d}{dx} f(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x - h)}{2h}$$



对于相当平滑的一维函数，这种方法效果不错：

数值问题

选择一个既不会导致函数表示不准确又不会让数值问题主导结果的 h 也是困难的。记住，有限差分违反了所有良好的浮点结果规则：

$$\frac{f(x + h) - f(x - h)}{2h}$$

- (a) 它将一个小数 h 加到可能大得多的数 x 上 (量级误差)
- (b) 然后它减去两个非常相似的数 $f(x + h)$ 和 $f(x - h)$ (消除误差)
- (c) 然后它将结果除以一个非常小的数 $2h$ (除法放大)

由于这些问题，有限差分方法虽然在数学上是可行的，但在实际的计算实践中往往是次优的。特别是在机器学习等领域，其中包含大量参数的复杂函数，采用自动微分技术来准确且高效地计算梯度是非常重要的。自动微分利用链规则，可以避免有限差分中的数值不稳定性和高计算成本，因此在需要频繁和准确地计算梯度的情况下，它成为了首选方法。

维数诅咒的复仇

然而，即使我们能够处理数值问题，在高维中这也是不可行的。**维数诅咒**再次出现。为了在点 \vec{x} 处评估梯度，我们需要在每一个维度 x_i 计算数值差分。如果 $\vec{\theta}$ 有一百万个维度，那么每一个单独的导数评估都将需要两百万次 $L(\theta)$ 的计算！这是一个完全不合理的开销。与零阶方法相比，一阶方法的加速会因为梯度的评估而消失。

正因为这样，在实践中，对于大型机器学习模型和其他高维优化问题，自动微分成为了一种必不可少的工具。自动微分技术，如反向传播算法，使我们能够有效地同时计算所有维度的导数，从而显著减少了所需的函数评估次数。这样一来，一阶优化算法就能够有效地应用于具有大量参数的复杂系统，使得深度学习等高维机器学习任务成为可能。

改进梯度下降

梯度下降可以非常有效，并且通常比零阶方法好得多。然而，它也有缺点：

损失函数 $L'(\theta) = \nabla L(\theta)$ 的梯度必须能够在任何点 θ 处计算。自动微分在这方面提供了帮助。

梯度下降可能会陷入局部最小值。这是梯度下降方法的固有特点，除非函数是凸的并且步长是最优的，否则它不会找到全局最小值。随机重新启动和动量方法可以减少对局部最小值的敏感性。

如果目标函数（和/或梯度）评估起来很慢，梯度下降可能会非常慢。如果目标函数可以表示为许多子问题的简单总和，随机梯度下降可以大大加速优化。

这些改进措施可以帮助梯度下降方法克服一些基本的限制，从而在更广泛的问题上更为有效和可靠。通过结合这些技术，可以改善梯度下降方法的鲁棒性，尤其是在复杂或非标准的优化问题中。

自动微分

如果我们能够以封闭形式精确地知道目标函数的解析导数，这个问题就可以解决。例如，我们在上一讲中看到的最小二乘线性回归的导数（相对而言）很容易确切地作为公式计算出来。然而，手动计算目标函数的导数似乎非常限制性，对于一个复杂的多维问题，这可能确实非常复杂。这是**算法微分**（或**自动微分**）的主要动机。

自动微分可以接受一个函数，通常是用一个完整编程语言的子集编写的，并自动构造一个在任何给定点评估精确导数的函数。这使得执行一阶优化变得可行。

编程语言的进步

在这个模块中，我们将看到支持数据科学的三个主要的编程进步。

- 向量化编程

- 例如: NumPy, eigen, nd4j, J
- 提供对张量的原生操作，可能有GPU加速。
- 可微分编程

- 可微分编程

- 例子: autograd, JAX, pytorch, tensorflow
- 自动微分向量化代码，生成张量算法的精确导数。

- 概率编程

- 例子: pymc, stan, edward, Uber, Pyro, webppl
- 允许值是不确定的，具有（张量，可微分）的随机变量作为一等值

这些进步中的每一个都为语言的表达能力提供了巨大的飞跃，使得能够优雅而紧凑地处理那些否则会繁琐且晦涩的算法。

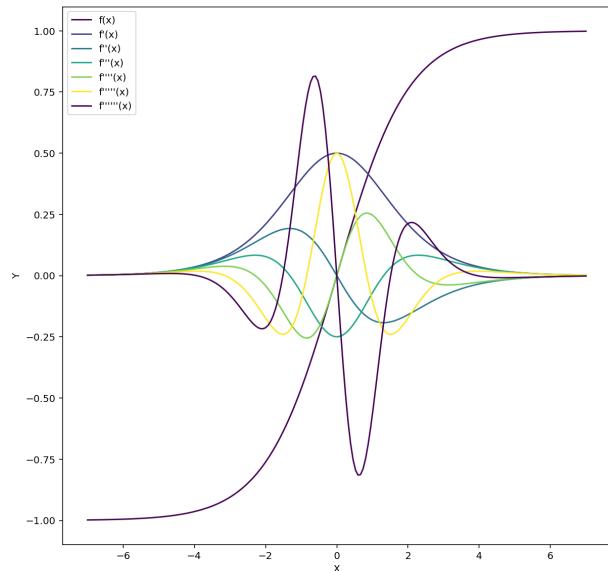
数据科学现代进步背后的魔法

一些最常见的实现技术体现在像TensorFlow、Theano、Torch/PyTorch这样的深度学习软件包中。这些软件提供了**定义计算图**的方法（隐式或显式），该计算图定义了要执行的操作，并且可以从中推导出相应的导数计算。这些实现往往专注于在神经网络中使用的操作，如矩阵乘法和逐元素非线性函数，并且它们通常不包括分支或迭代（或仅支持有限形式的条件/循环表达式）。

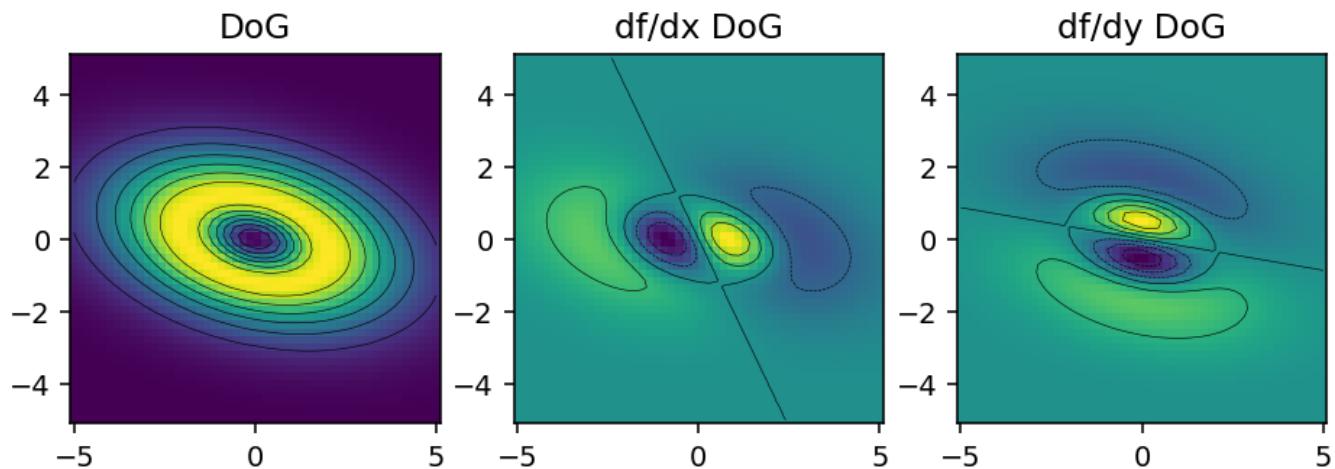
Autograd

例如autograd，为几乎所有NumPy代码提供自动微分。下面的例子来自autograd文档。它是一个可以直接替换的工具，它可以“神奇地”估计导数（尽管只支持某些操作）。

autograd 现在已经发展成为 Google JAX，可能是最有前途的机器学习库。JAX支持使用自动微分的GPU和TPU计算。我在这里没有使用它，因为它安装起来比较困难。



Vectorised example



在优化中使用自动微分

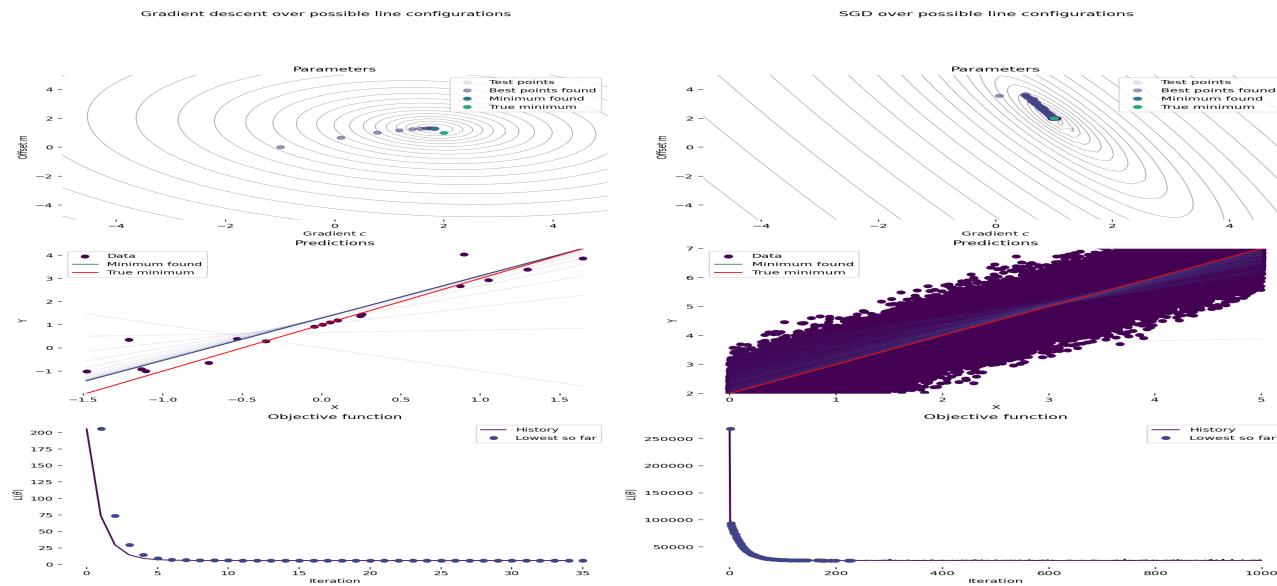
利用自动微分技术，我们可以将目标函数的形式直接写成计算过程，并且“免费”获得该函数的导数。这使得执行一阶优化变得极为高效。

这正是机器学习库所做的事情。它们让编写在GPU/TPU硬件上运行的向量化、可微代码变得简单。其余的只是附加操作

一阶线性拟合

让我们重新解决第6讲中的最佳拟合直线问题。我们想要找到 m 和 c ；即直线的参数，使得直线与一组数据点之间的平方差最小化（目标函数）。

一阶线性拟合 和 SGD



自动微分的限制

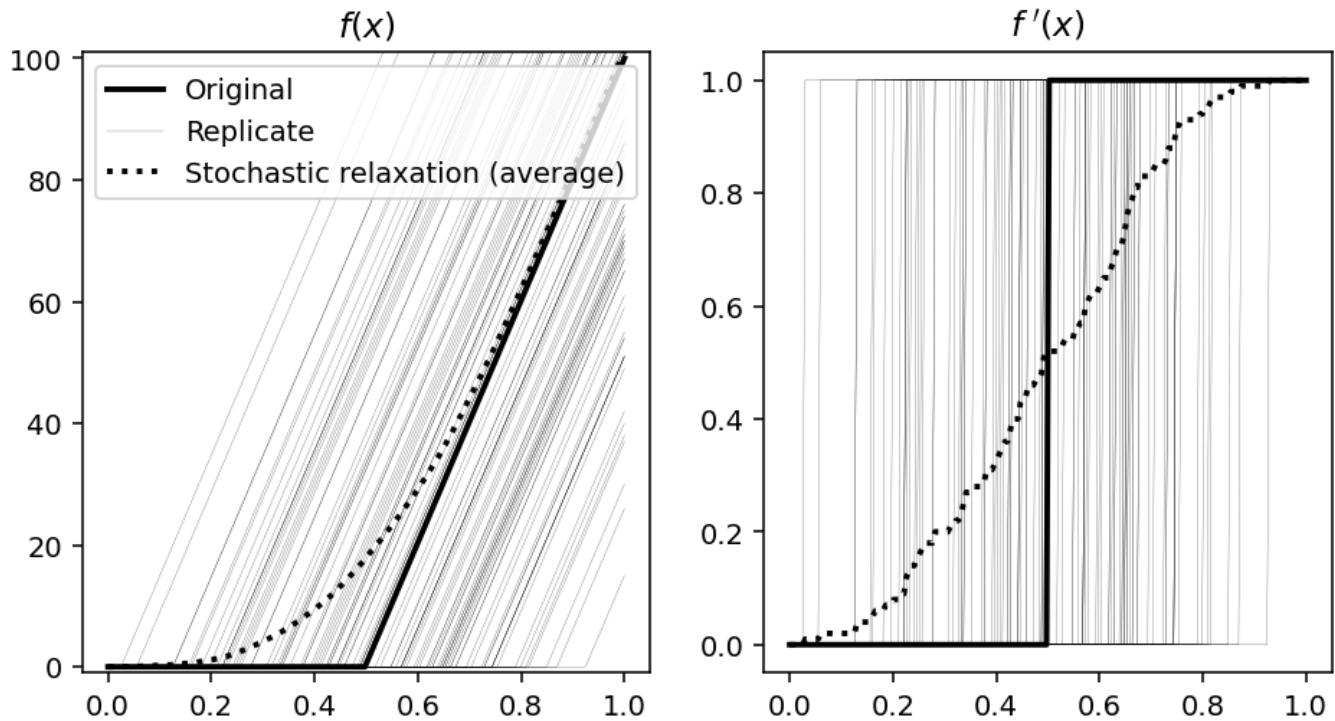
显然，微分只适用于可微分的函数。虽然一阶梯度向量通常可以在合理的时间内计算，但正如我们稍后将看到的，计算多维函数的二阶导数变得非常困难。

"The Blind Watchmaker"一书中提出并讨论了如何动物进化出伪装能力的问题。进化是一个逐步的优化过程，而为了能够被接受的步骤，需要从"较差的适应性"到"良好的适应性"之间有一个平滑的路径。

那么，像蛾这样的动物是如何进化出伪装的呢？它要么被看到而被捕食者吃掉，要么捕食者没看到它。这是一个二元函数，没有梯度。虽然进化不需要梯度，但它确实需要存在大致连续的适应性函数。

解决方案 争论点是，尽管每一个特定的案例都是简单的二元选择，但它是平均在许多随机情况下，其中条件会略有不同（可能接近黑暗，可能捕食者视力不佳，可能天气多雾）并且平均所有这些情况，一些非常微小的颜色变化可能会带来优势。从数学角度讲，这是随机弛豫；表面上不可能的陡峭梯度通过整合许多不同的随机条件被渲染成（大约）Lipschitz连续的。

这适用于进化之外的许多问题。例如，一个非常陡峭的函数在某点有一个非常大的导数；或者它在部分区域内可能导数为零。但如果我们将许多情况进行平均，其中步骤位置略有不同，我们就会得到一个平滑的函数。



随机梯度下降

梯度下降在每次迭代之前都会评估目标函数及其梯度，然后再进行一步操作。这在优化大数据集的函数近似时尤其昂贵（例如，在机器学习中）。

如果目标函数可以分解为小的部分，优化器可以独立地对随机选择的部分进行梯度下降，这可能会快得多。这称为随机梯度下降（SGD），因为它采取的步骤取决于目标函数部分的随机选择。

如果目标函数可以写成和的形式：

$$L(\theta) = \sum_i L_i(\theta),$$

即目标函数由许多简单的子目标函数 $L_1(\theta), L_2(\theta), \dots, L_n(\theta)$ 的和组成。

这种形式在参数匹配观测值的近似问题中经常出现，如在机器学习应用中。在这些情况下，我们有许多训练样本 \vec{x}_i 与已知的输出 y_i 匹配，我们希望找到参数向量 θ

$$L(\theta) = \sum_i \|f(\vec{x}_i; \theta) - y_i\|$$

使得上述式子被最小化，即模型输出和期望输出之间的差异被最小化，对所有训练样例求和。

微分是一个线性运算符。这意味着我们可以交换求和、标量乘法和微分

$$\frac{d}{dx}(af(x) + bg(x)) = a \frac{d}{dx}f(x) + b \frac{d}{dx}g(x), \text{ 我们有：}$$

$$\nabla \sum_i \|f(\vec{x}_i; \theta) - y_i\| = \sum_i \nabla \|f(\vec{x}_i; \theta) - y_i\|$$

在这种情况下，我们可以取任何一组子集的训练样本和输出，计算每个样本的梯度，然后根据子集的计算梯度进行移动。随着时间的推移，随机子集选择将（希望）平均分布。每个子集称为一个小批量

(minibatch)，并且遍历整个数据集的一次运行（即足够的小批量，以便每个数据项都被优化器“看到”）称为一个轮次 (epoch)。

内存优势

SGD(随机梯度下降)在内存消耗方面具有主要优势，因为计算可以应用于小批量的少量样本。我们只计算一个子样本的梯度，并朝着那个方向移动。这不会是完全正确的整个目标函数的导数，但它将是一个很好的近似。

特别是在内存受限的设备上，如GPU（即使是强大的GPU，也可能只有12-16GB的RAM），将整个数据集存储在设备上可能是不可能的。分成批次可以绕过这个限制。这也可以在内存层次结构方面具有优势——小批量数据可能会引起更少的缓存未命中，从而提高性能。

启发式增强

SGD(随机梯度下降)不仅在内存效率上有优势，实际上在目标函数下降方面也可以提高优化性能，特别是通过减少陷入极小值的可能性。这是因为在每个小批量中对目标函数的随机划分向优化过程中增加了噪声。这一开始看起来像是一个问题；我们不希望优化是不准确的。但噪声意味着梯度下降有可能不下降，而是可能上升并越过一个极大值。

虽然增加噪声是一种**启发式搜索方法**（没有保证它会改善情况，甚至不会让情况变得更糟），但通常非常有效。我们本质上得到了一种有限形式的**随机松弛**的好处——通过对随机子样本进行平均，我们的目标函数可以被“平滑化”，所以即使它不是完全的Lipschitz连续（或者有一个不好的Lipschitz常数），SGD也可以工作得很好。

使用SGD

SGD并没有保证会朝着正确的方向移动。在实践中，它对许多现实世界问题来说可以非常高效。例如，让我们再次考虑将一条线拟合到观察结果（线性回归）的问题。

给定一组对 x_i, y_i ，我们想要找到参数 $\vec{\theta} = [m, c]$ 来最小化函数 $y' = f(x; m, c) = mx + c$ 和真实已知输出 y 之间的平方误差。目标函数是：

$$\begin{aligned} & \sum_i \|f(\vec{x}_i; \vec{\theta}) - y_i\|_2^2 \\ &= \sum_i \|(mx_i + c) - y_i\|_2^2 \end{aligned}$$

我们可以计算一个随机子集上的梯度，而不是计算整个和。代码实现：

```
import autograd.numpy as np
def sgd(L, dL, theta_0, xs, ys, step=0.1, batch_size=10, epochs=10):
    """L: Objective function, in the form L(theta, xs, ys)
       dL: derivative of objective function in form dL(theta, xs, ys)
       theta_0: starting guess
       xs: vector of inputs
       ys: vector of outputs
       step: step size
       batch_size: batchsize
    """
    for epoch in range(epochs):
        for i in range(0, len(xs), batch_size):
            batch_xs = xs[i:i+batch_size]
            batch_ys = ys[i:i+batch_size]
            grad = dL(theta_0, batch_xs, batch_ys)
            theta_0 -= step * grad
```

```

theta = np.array(theta_0)
grad = np.zeros_like(theta_0)

o = History()

# One epoch is one run through the whole dataset
for epoch in range(epochs):
    batch = np.arange(len(xs))
    # randomize order
    np.random.shuffle(batch)
    subset_index = 0
    # iterate over subsets

    ## One batch
    while subset_index + batch_size <= len(xs):

        # accumulate partial gradient
        i,j = subset_index, subset_index+batch_size

        # compute gradient on the random subset
        # accumulating the gradient direction as we go
        grad = dL(theta, xs[batch[i:j]], ys[batch[i:j]]) / (j-i)
        # next batch please!
        subset_index += batch_size

        # make a step
        theta = theta - grad * step
        o.track(theta, L(theta, xs, ys))

#o.track(theta, L(theta, xs, ys))
return o.finalise()

```

使用SGD的线性回归

我们可以使用10000个点来进行线性回归示例——并且只通过数据一次就找到了一个很好的拟合。这是因为我们可以将问题分解为许多小问题的和（一次在几个随机点上拟合线），这些都是一个大问题的一部分（拟合所有点的线）。

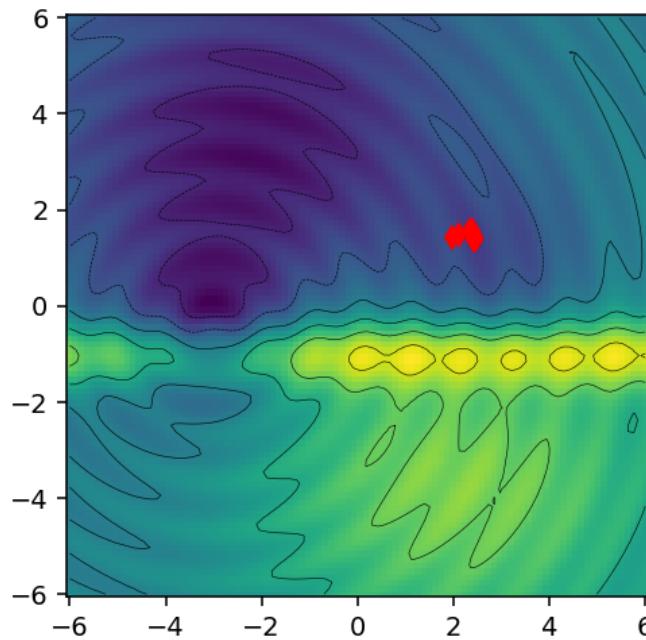
当这种情况成为可能时，它比计算我们可用的全部数据集的梯度要高效得多。

一个噩梦般的函数

这个函数包含了所有困难的特性：

- 到处是狭窄的山谷
- 多个局部最小值
- 中间穿过一个巨大的山脊

梯度下降几乎无望——它会被困在一个山谷中，并在一个巨大的弧形路径中徘徊，甚至永远不会接近任何一个局部最小值



梯度下降法不能很好地解决问题。调整步长也无济于事；问题并不在于我们选择了一个糟糕的步长。**弹性梯度下降**也无济于事；我们的目标函数并不是一个简单的子目标函数之和。我们能做什么呢？

随机重启

梯度下降法很容易陷入局部极小值。一旦陷入局部极小值的**吸引盆地**，就很难脱身。梯度下降法可以增加一点噪音，使优化器越过小的山脊和山峰，但无法脱离深度最小值。一个简单的启发式方法就是运行梯度下降法，直到它卡住为止，然后随机地以不同的初始条件重新开始，再试一次。如此反复多次，希望其中一条优化路径最终能达到全局最小值。这种元启发式适用于任何局部搜索方法，包括爬山法、模拟退火法等。

简单记忆：动量项

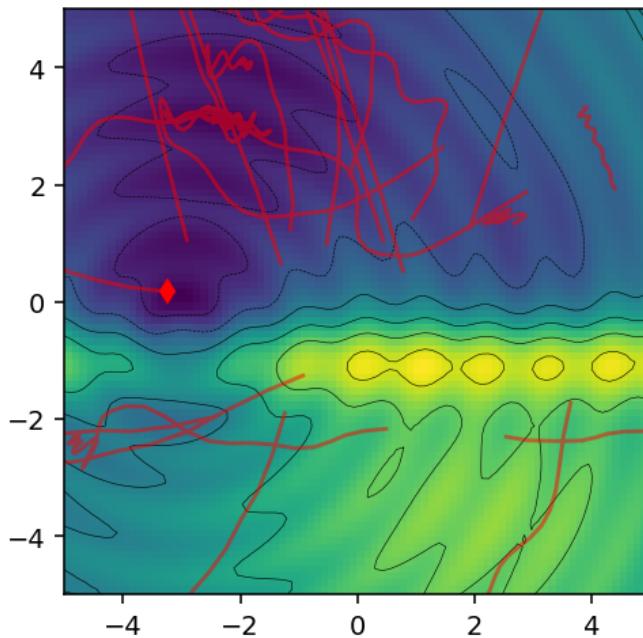
一个物理上的球在表面滚动时不会因为表面的小不平整而停下来。一旦它开始下坡滚动，它可以跳过小坑坑和小凹陷，穿过平坦的平原，并且稳定地沿着狭窄的山谷向下，而不会从边缘弹开。这是因为它有动量；它会倾向于继续向它刚才前进的方向移动。这是记忆启发式的一种形式。与其拥有蚁群风格的路径，优化器只记住一条简单的路径——它当前的前进方向。

同样的想法可以用来减少（随机）梯度下降被目标函数中的小波动所困住的几率，并且“平滑”下降过程。思路很简单；如果你现在正朝着正确的方向前进，即使梯度不总是完全下降，也要继续朝那个方向前进。

我们引入一个速度 v ，并让优化器朝这个方向移动。我们逐渐调整 v 以与导数对齐。

$$v = \alpha v + \delta \nabla L(\theta) \quad \theta^{(i+1)} = \theta^{(i)} - v$$

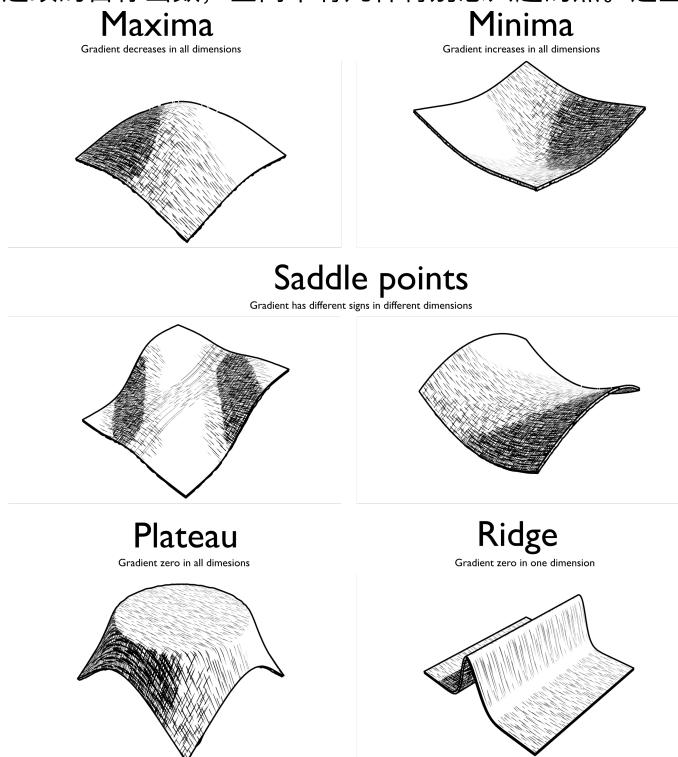
这由参数 α 控制； α 越接近 1.0，系统中的动量就越大。 $\alpha = 0.0$ 相当于普通的梯度下降。



关键点的类型

这种对物理表面的直观理解让我们思考如何描述目标函数的不同部分。

对于平滑连续的目标函数，空间中有几种特别感兴趣的点。这些是**关键点**，在这些点上，梯度向量的分量是



零向量。

图片：关键点的分类。每一种都对应于Hessian矩阵的特征值的不同配置。

二阶导数

如果一阶导数代表函数的“斜率”，那么二阶导数代表函数的“曲率”。

对于每一个参数分量 θ_i ，海森矩阵（Hessian）存储了其他每一个 θ_j 的“陡峭程度”如何变化。

想象我在一座山上

- 我所在的海拔高度对应于目标函数的值。
- 我可以改变的参数是我的北/南和东/西方向的位置。
- 梯度向量是我向北或向东走一步时海拔高度的变化，这些就是两个参数。这表示了我在山上所处位置的局部陡峭程度。
- 海森矩阵（Hessian）捕捉到当我向北走时，向北一步陡峭程度的变化量，同时当我向北走时东面陡峭程度的变化量；向东走同理。因此，有一个 2×2 矩阵描述了这些陡峭程度的变化。

对于向量值函数 $f(\vec{x})$ ，我们有以下关系：

$$\nabla L(\vec{\theta}) = \left[\frac{\partial L(\vec{\theta})}{\partial \theta_1}, \frac{\partial L(\vec{\theta})}{\partial \theta_2}, \dots, \frac{\partial L(\vec{\theta})}{\partial \theta_n} \right],$$

这是梯度向量，而二阶导数存储在海森矩阵中：

$$\nabla \nabla L(\vec{\theta}) = \nabla^2 L(\vec{\theta}) = \begin{bmatrix} \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1^2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2^2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_2 \partial \theta_n} \\ \vdots & & & & \\ \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n \partial \theta_3} & \cdots & \frac{\partial^2 L(\vec{\theta})}{\partial \theta_n^2} \end{bmatrix},$$

这就是海森矩阵。注意，在海森矩阵中，我们为函数的每一对维度都有一个条目。你可能会注意到它与协方差矩阵的相似性，后者捕捉了数据坐标如何相互变化；海森矩阵则捕捉了函数的梯度如何相互变化。

对于一个二维表面，梯度向量指定了在给定点切于表面的平面的法线。海森矩阵指定了一个二次函数（一个有一个极小值的平滑曲线），遵循表面的曲率。

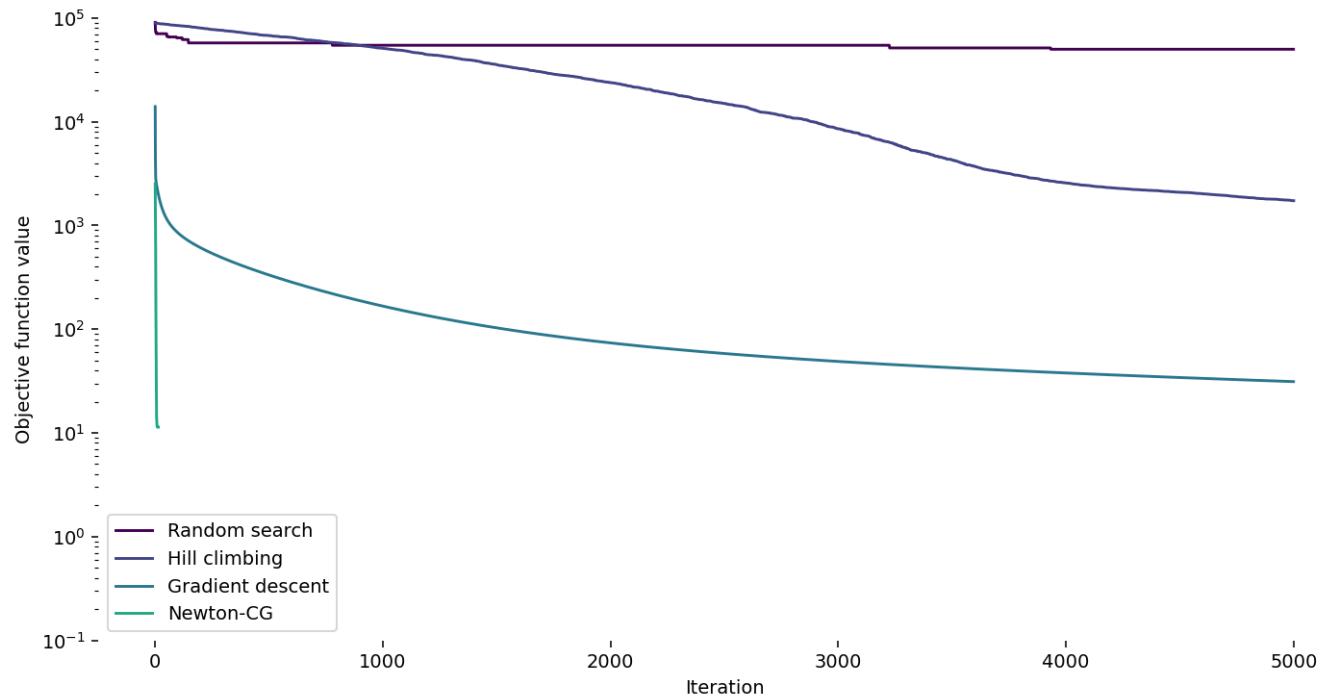
海森矩阵的特征值

海森矩阵的特征值捕捉了我们在前面讲座中看到的关于**关键点类型**的重要特性。特别是，海森矩阵的**特征值**告诉我们我们所拥有的关键点是什么类型的。

- 如果所有特征值都严格为正，矩阵被称为**正定的**，该点是一个极小值点。
- 如果所有特征值都严格为负（**负定的**），那么该点是一个极大值点。
- 如果特征值有混合的符号，那么该点是一个鞍点。
- 如果特征值都是正或都是负，但其中一些是零，那么矩阵是**半正定的**，该点是高地/脊线。

二阶优化

二阶优化使用 Hessian 矩阵，一步跳到每个局部二次逼近的底部。这样就可以跳过平原，避开会减慢梯度下降速度的鞍点。一般来说，二阶方法要比一阶方法快得多。



维度诅咒（再次提及）

然而，简单的二阶方法在高维空间中不适用。计算 Hessian 矩阵需要 d^2 次计算以及 d^2 的存储空间。许多机器学习应用中的模型具有 $d > 100$ 万个参数。仅仅存储一次优化迭代的 Hessian 矩阵就需要：

二阶优化在穿越鞍点和平原时比梯度下降等一阶方法更快。它在低维问题中特别有效，但是维度诅咒总是存在的。想象一下，我们有一个包含 100 万个参数的问题需要优化。梯度向量的元素数量与参数向量相同，因此有 100 万个元素，或者对于 float64，约占用 8 兆字节的内存。但 Hessian 矩阵必须存储每一对参数的变化。这将需要：8000000000000 bytes of memory 即 8 TB！这是无法承受的计算负担。有一些特殊的、内存有限的二阶方法，它们使用近似海森矩阵（如广泛使用的 L-BFGS 算法），但这不在本课程的范围之内。

概率

本课程部分涉及随机元素；不确定性、随机性和统计在计算中的作用。基本的数学原理来源于概率论，它为我们提供了操作不确定值的简单而强大的方法，并且允许我们执行像根据一些观测推断最可能的假设这样的有用操作。概率论是一种简单、一致且有效的操纵不确定性的方法。

什么是概率？

- **实验**（或试验）一个有不确定结果的发生。
 - 例如，失去一个潜艇——潜艇的位置现在未知。
- **结果** 实验的结果；世界的一种特定状态。
 - 例如：潜艇在海洋网格[2,3]。
- **样本空间** 实验所有可能结果的集合。
 - 例如，海洋网格 $\{[0,0], [0,1], [0,2], [0,3], \dots, [8,7], [8,8], [8,9], [9,9]\}$ 。
- **事件** 一些具有共同属性的可能结果的子集。
 - 例如，位于赤道以南的网格。
- **概率** 事件关于样本空间的概率是样本空间中属于事件的结果数量除以样本空间中所有可能结果的总数。因为它是一个比例，概率总是一个介于0（代表不可能发生的事件）到1（代表必然发生的事件）之间的实数。
 - 例如，潜艇位于赤道以下的概率，或潜艇在网格[0,0]中的概率（在这个例子中事件就是单一的结果）。
- **概率分布** 将结果映射到总和为1的概率的映射。这是因为从一次试验中必定会发生一个结果（概率为1），因此所有可能结果的总和将是1。一个随机变量有一个概率分布，它将每个结果映射到一个概率。
 - 例如 $P(X = x)$ ，潜艇位于特定网格平方 \vec{x} 的概率
- **随机变量** 表示一个未知值的变量，我们确实知道它的概率分布。该变量与试验的结果相关联。
 - 例如， X 是一个随机变量，代表潜艇的位置。
- **概率密度/质量函数** 通过将每个结果映射到概率 $f_X(x)$, $x \rightarrow \mathfrak{M}$ 来定义概率分布的函数。这可以是连续的 x (密度) 或离散的 x (质量)。
 - 例如 $f_X(x)$ 是潜艇的概率质量函数，它将每个网格映射到代表其概率的实数。
- **观察** 我们直接观察到的结果；即数据。
 - 例如，潜艇被发现在网格[0,5]中
- **样本** 我们根据概率分布模拟的结果。我们说我们从分布中抽取了一个样本。
 - 例如，如果我们相信潜艇是根据某种模式分布的，生成可能的、遵循这种模式的具体网格位置。
- **期望/期望值** 随机变量的“平均”值。
 - 潜艇平均位于网格[3.46, 2.19]中

概念

一个随机变量 X 具有一个概率分布 $P(X)$ ，它为属于样本空间 \mathfrak{X} 的结果 x 分配概率 $0 \leq P(X = x) \leq 1$ 。该概率分布由一个概率密度/质量函数 $f_X(x)$ 定义，它为结果分配概率，使得所有结果上的概率之和为 1， $\sum_{x \in \mathfrak{X}} f_X(x) = 1$ 。我们可以观察通过试验从分布中得到的特定结果 x_i 。我们可以根据分布 $P(X)$ 抽样（模拟）新的结果 x'_j 。假设结果有价值，我们可以在无数次试验中评估平均期望值 $E[X]$ 。

概率哲学

关于概率及其应用，存在两种思考学派。我们将（主要）遵循贝叶斯解释，但理解这涉及什么是有价值的。

贝叶斯/拉普拉斯关于概率的观点

贝叶斯主义者将概率视为信念的运算；在这种思维模型中，概率是置信程度的衡量。 $P(A) = 0$ 表示相信事件 A 不可能是真的，而 $P(A) = 1$ 则是相信事件 A 绝对确定的信念。在贝叶斯观点中，说“外面下雨的概率是0.3”是有意义的

(概率量化了我们根据我们所拥有的信息对天气的信念)。注意，这并不是说我们相信天气有0.3的可能性是下雨的(不管那意味着什么)

贝叶斯主义者允许通过概率规则结合并操纵对状态的置信度。贝叶斯逻辑中的关键过程是置信度的更新。给定一些：
* 先验概率（这是格拉斯哥，可能不会是晴天）和一些 * 新的证据（室内似乎有明亮的反射） * 更新我们的来计算后验——我们认为外面是晴天的新概率。

贝叶斯推理要求我们接受对事件的先验设定，即我们必须用概率分布明确量化我们的假设。它是对不确定信息的逻辑的扩展。

USS Scorpion

例如，在搜寻潜艇的例子中，先验可能是潜艇很可能在南大西洋（根据收到的最后一次无线电广播）。证据可能是来自搜寻船只的声纳调查结果。在每次调查后，潜艇位于调查区域的后验概率可能被更新。这代表了我们关于船只可能位置的置信度。

频率主义者对概率的观点

有一种替代的学派认为概率仅仅是重复事件长期行为的表现（例如，硬币正面朝上的概率是0.5，因为从长期来看，这将是发生此事件的平均比例）。

频率主义者不接受像“现在是晴天的概率是多少？”这样的说法，因为没有涉及到长期行为（“现在”只有一次）。在这种世界观中，讨论只能发生一次的事件的概率是没有意义的。在频率主义者看来，询问像“任何给定日子是晴天的概率是多少？”这样的问题是很有意义的，因为我们可以测量许多不同天数的这个事件（晴天或不晴天）。例如，频率主义者不会给USS Scorpion在特定网格中的概率赋值；这不是一个可以重复的实验。

客观性与主观性

频率主义者与贝叶斯主义者之间的辩论很快就会进入哲学领域。这里无法充分展示观点的多样性和论点的深度。

简而言之，贝叶斯概率论有时被认为是主观的，因为它需要指定先验信念，而频率主义的概率模型不承认先验的概念，因此是客观的。

另一种观点是，贝叶斯模型明确编码了不确定知识，并为操作该知识制定了普遍的正式规则，正如形式逻辑对于确定知识所做的那样。频率主义方法在它们对普遍真理（例如，渐近行为）的陈述中是客观的，但它们不构成信念的演算，因此不能直接回答许多重要的问题。

贝叶斯

- 包含先验 (priors)
- 概率是置信度 (degree of belief)
- (将群体参数视为随机变量，数据为已知)

频率主义

- 没有先验 (priors)
- 概率是事件的长期频率 (long-term frequency of events)
- (假定群体参数为固定的，数据为随机的)

概率模型的优越性

不管你认同哪一种哲学模型，有一点你可以确信的是：概率是最好的。

虽然有时会使用概率论以外的其他不确定性模型，但所有其他表示不确定性的方法与概率方法相比都是严格劣于的，在这个意义上，一个人、代理或计算机在不确定的情况下使用概率模型对未来事件进行“下注”，在所有决策系统中可以得到最好的可能回报。

任何与使用概率论所能达到的同样好的赌博结果的理论都等同于概率论。

生成模型：正向和逆向概率

概率模型的一个关键思想是生成过程；即存在某个未知过程，我们可以观察到其结果。该过程本身受我们不知道的未观察变量的控制，但我们可以推断。

一个典型的例子是罐子问题。设想一个罐子，一些球被倒入罐中（比如由某个神秘实体）。每个球可能是黑色或白色。

你随机从罐子中取出四个球，并观察它们的颜色。你得到了四个白球。

现在你可以问很多问题：

- 下一个被抽出的球是白色的概率是多少？
 - 这是一个正向概率问题。它问的是与观察结果分布相关的问题。
- 罐中白球和黑球的分布是怎样的？
 - 这是一个逆向概率问题。它问的是与生成观察结果的过程所受控制的未观察变量相关的问题。
- 神秘实体是谁？
 - 这是一个无法知晓的问题。我们的观察结果无法解决这个问题。

有大量的过程可以被构建为罐子问题（比如球被替换的罐子问题，有多个罐子但你不知道球来自哪个罐子的问题，球可以在罐子之间移动的问题，等等）。

概率论的形式基础

概率公理

概率的基本公理很少，其他一切都可以从这些公理推导出来。用 $P(A)$ 表示事件 A 的概率（注意：这些适用于事件（结果的集合），而不仅仅是结果！）

- 有界

$$0 \leq P(A) \leq 1$$

对于所有可能的事件 A —— 概率为0，或者是正数且小于1。

- 统一

$$\sum_x P(x) = 1$$

对于样本空间 σ 中的完整的可能结果集合（不是事件！） $x \in \sigma$ —— 总会发生某些事情。

- 求和规则

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

即事件 A 或事件 B 发生的概率是两者独立概率之和减去两者同时发生的概率。（符号注释： \vee 表示“或”， \wedge 表示“且”）

- 条件概率

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

条件概率 $P(A|B)$ 被定义为在已知事件 B 发生的条件下，事件 A 发生的概率。

随机变量和分布

一个随机变量是可以取不同值的变量，但我们不知道它的具体值；即一个“未指定”的变量。然而，我们有一些知识，能够捕捉这个变量可能取的不同状态及其对应的概率。概率论允许我们在不需要给随机变量指定一个具体值的情况下，对其进行操作。

随机变量用一个大写字母表示，比如 X 。

随机变量可能代表：

- 投掷骰子的结果（离散的）；
- 室外是否下雨（离散的：二元的）；
- USS Scorpion 潜艇的纬度（连续的）；
- 我们还没遇到的某个人的身高（连续的）。

分布

一个概率分布定义了随机变量不同状态的可能性。

我们可以将 X 看作是实验，将 x 看作是结果，有一个函数将每一个可能的结果映射到一个概率上。我们写作 $P(X = x)$ （注意大小写！），并使用以下简写表示：

$P(X = x)$, 随机变量 X 取值 x 的概率 $P(X)$, 表示 $X=x$ 的概率的简写 $P(x)$, 表示特定值 $X=x$ 的概率的简写 P_x

我们可以将结果视为随机变量取一个特定值，即 $P(X = x)$ 。请注意，按照惯例我们使用 $P(A)$ 表示事件 A 的概率，而不是随机变量 A 的概率（一个事件是结果的集合；随机变量仅为结果分配概率）。

离散和连续

随机变量可以是连续的（例如一个人的身高）或者是离散的（骰子面上显示的值）。

离散变量 离散随机变量的分布用一个概率质量函数（PMF）描述，它为每个结果赋予一个特定的值；想象一个Python字典，它将结果映射到概率上。PMF通常写作 $f_X(x)$ ，其中 $P(X = x) = f_X(x)$ 。

连续变量 连续变量有一个概率密度函数（PDF），它指定了作为连续函数 $f_X(x)$ 的结果上的概率分布。对于PDF来说，不是 $P(X = x) = f_X(x)$ 这种情况。

积分为一

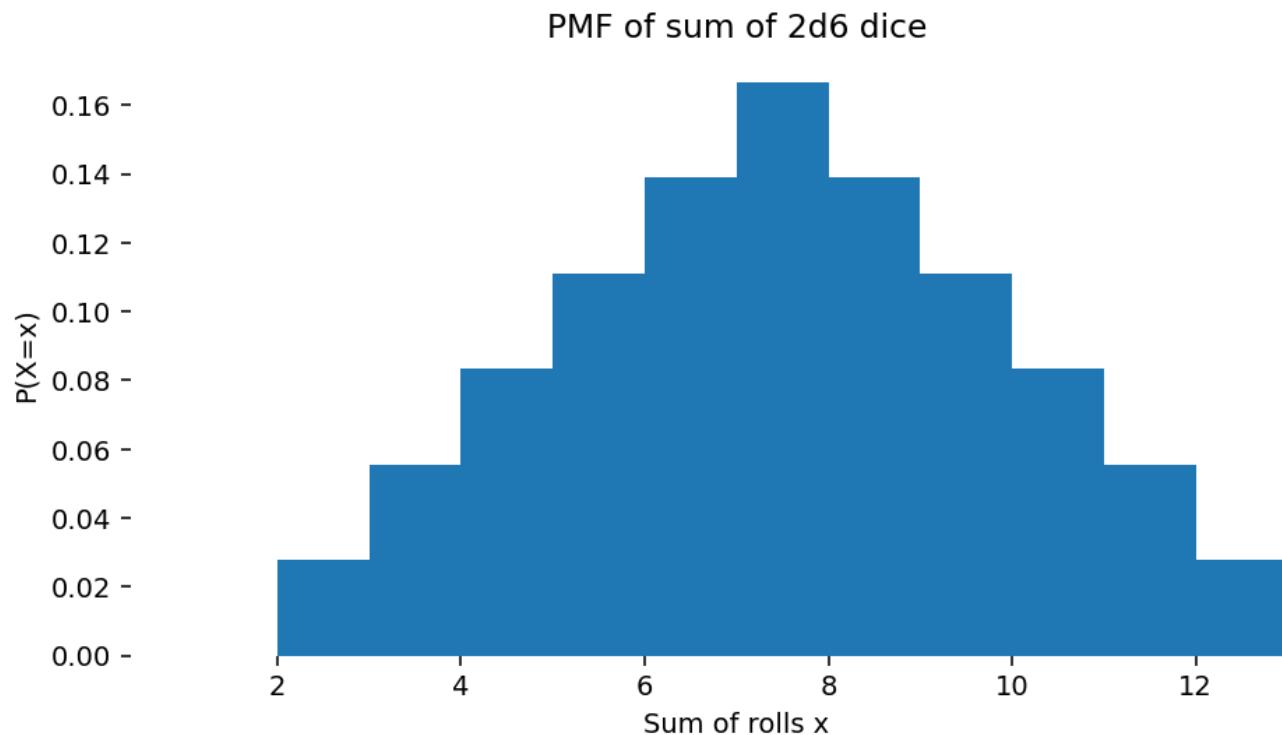
概率质量函数或概率密度函数的和/积分必须正好为 1，因为所考虑的随机变量必须取值；这是单一性的结果。实验的每一次重复都只有一个结果。

$$\sum_i f_X(x_i) = 1 \quad \text{for PMFs of discrete RVs}$$

$$\int_x f_X(x) dx = 1 \quad \text{for PDFs of continuous RVs}$$

PMF 示例：掷骰子的总和

一个非常简单的离散 PMF 是两个六面骰子之和的期望值。 $P(X = x) = f_X(x)$ 对每种可能的结果 x 取值 (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)



期望值

如果一个随机变量取数值，那么我们可以定义这个随机变量的**期望值或期望** $E[X]$ 为：

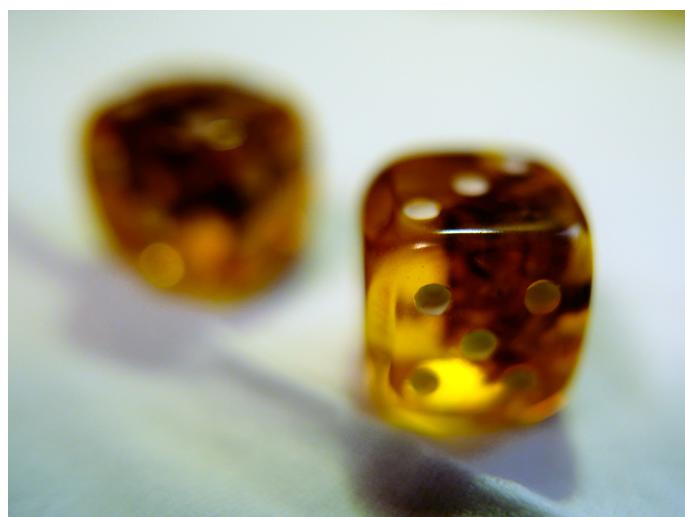
$$\checkmark = \int x f_X(x) dx$$

对于具有概率质量函数 $P(X = x) = f_X(x)$ 的离散随机变量，我们可以将其写成求和形式：

$$\checkmark = \sum x f_X(x)$$

如果只有有限个可能的结果，那么这可以表示为： $\checkmark = P(X=x_1) x_1 + P(X=x_2) x_2 + \dots + P(X=x_n) x_n$

期望值是随机变量的“平均值”。非正式地说，它代表我们“期望发生的情况”；最有可能的总体“得分”。它可以被认为是一个实验所有可能结果的加权和，其中每个结果按该结果发生的概率进行加权。



例如，在“一对骰子”的场景中，我们可以计算掷骰子后显示的“点数”总数的期望值。我们计算显示每个点数的概率，并乘以该点数，然后将结果相加。这就是平均显示的点数，或者说是期望值。

期望值与均值

期望值对应于平均或平均结果的概念。随机变量的期望值是所有结果值的**真实平均值**，如果我们无限次进行实验，就会观察到这些结果。这是**总体均值**——随机变量的整个可能无限的总体的平均值。

许多关于随机变量的重要性质可以用期望值来定义。

- 随机变量 X 的均值就是 $\mathbb{E}[X]$ 。它是**中心趋势**的度量。
- 随机变量 X 的方差是 $\text{var}(X) = \mathbb{E}[X - \mathbb{E}[X]^2]$ 。它是**分散程度**的度量。

对 X 的函数的期望值

我们可以对随机变量应用函数，例如，随机变量的平方。

连续随机变量 X 的任何函数 $g(X)$ 的期望值定义为：

$$\mathbb{E}[g(X)] = \int_x f_X(x)g(x)dx$$

或者

$$\mathbb{E}[g(X)] = \sum_x f_X(x)g(x)$$

对于离散随机变量。

例如，我们可以计算像这样的简单期望值：

$$\mathbb{E}[2X^2] = \sum_x f_X(x)2x^2dx$$

或者

$$\mathbb{E}[\sin(X)] = \sum_x f_X(x)\sin(x)dx$$

也就是说，我们只是取每个结果通过函数 $g(x)$ 后的和/积分，按结果 x 发生的概率加权。 $g(x)$ 可以被认为是一个“评分”函数，它为 X 的每个结果分配一个实数。注意 $g(x)$ 不会影响概率密度/质量函数 $f_X(x)$ 。它不影响结果的概率，只影响分配给这些结果的值。例如，如果我们玩一个骰子游戏，其中掷出的分数是显示点数的平方（所以显示2点值4分，显示8点值64分等），那么我们会计算 $\mathbb{E}[X^2]$ 如下：

```
sqr_expected_value = np.sum(pmf * (np.arange(2,13))**2)
```

计算经验分布

对于离散随机变量，我们总是可以从一系列观察中计算出经验分布；例如，从文本语料库中特定单词的计数中计算（例如，1994年每篇报纸文章中的计数）。我们只需计算每个单词出现的次数并除以单词总数。

$$P(X = x) = \frac{n_x}{N},$$

其中 n_x 是观察到结果 x 的次数， N 是试验的总次数。

注意，经验分布是一种近似未知真实分布的分布。对于离散变量的非常大的样本，只要我们看到的样本是以不带偏见的方式抽取的，经验分布将越来越接近**真实的概率质量函数（PMF）**。然而，这种方法对于连续随机变量并不实用，因为我们只会看到每个观察值一次（想想为什么！）。

随机抽样程序

均匀抽样

有一些算法可以生成在一个区间内**均匀分布**的连续随机数，例如从 0.0 到 1.0。这些实际上是**伪随机数**，因为计算机是（希望如此）确定性的。它们被设计来近似真实随机序列的统计特性。本质上，所有这些生成器都生成离散符号（比特或整数）的序列，然后将这些符号映射到特定范围内的浮点数；这是相当难以做到准确的。

我们必须小心：计算机生成的是伪随机浮点数；而不是真正的随机实数。虽然大多数时候这种差异并不重要，但它们确实是不同的事物。

均匀分布的数在其区间内取任何值的概率都相等，并且在其他地方的概率为零。尽管这是从连续概率密度函数（PDF）中抽样，但它是从任意概率质量函数（PMF）中抽样的关键基础。均匀分布记为 $X \sim U(a, b)$ ，意味着随机变量 X 可以在 a 和 b 之间取值，且该区间内任何数值出现的可能性都相等。符号 \sim 读作“分布为”，即“ X 分布为区间 $[a, b]$ 中的均匀分布”。

注意在实践中，如果我们使用浮点数，这些值不会在给定区间的实数中均匀分布，因为我们只能抽样有效的浮点值。虽然浮点数是非均匀分布的，但对于大多数应用来说，这种差异并不重要。

例如，我们知道 NumPy 函数 `np.random.uniform(a, b, [shape])` 用 a 和 b 之间的“连续”随机数填充数组。

离散抽样

对于离散概率质量函数，我们可以通过划分单位区间来按照任意给定的 PMF 抽样结果。这就像在墙上贴上面积与每个结果的概率成比例的海报，然后向墙上投掷飞镖。

算法：

- 为结果 x_1, x_2, \dots 选择任意排序
- 为每个结果分配一个“区间”，这是区间 $[0,1]$ 的一部分，等于其概率，这样区间就被划分为连续的不重叠区域 $[P(x_1) \rightarrow P(x_1) + P(x_2), P(x_1) + P(x_2) \rightarrow P(x_1) + P(x_2) + P(x_3), \dots]$
- 在范围 $[0,1]$ 内抽取一个均匀样本
- 它落在的“结果区间”即为要抽取的样本

根据 PMF 的定义，所有概率的总和将为 1.0，因此它将完美地填满区间 $[0,1]$ ，没有间隙。

联合概率、条件概率、边际概率 Joint conditional marginal probability

两个随机变量的联合概率记为 $P(X, Y)$ ，表示 X 和 Y 同时取特定值的概率（即 $P(X = x) \wedge P(Y = y)$ ）。

边际概率是从 $P(X, Y)$ 通过对所有可能的 Y 结果进行积分（求和）得到的 $P(X)$ ：

$$P(X) = \int_y P(X, Y) dy \text{ 对于 PDF 而言。}$$

$$P(X) = \sum_y P(X, Y) \text{ 对于 PMF 而言。}$$

这使我们能够通过对涉及的另一个变量的所有可能结果求和，从联合分布中计算出一个随机变量的分布。

边际化只是指从联合分布中对一个或多个变量进行积分：它移除了这些变量的分布。

如果两个随机变量相互独立，则表示它们之间没有任何依赖关系。如果是这种情况，那么联合分布就是各自分布的乘积： $P(X, Y) = P(X)P(Y)$ 。在变量相互依赖的一般情况下，这不成立。

给定随机变量 Y 的情况下随机变量 X 的条件概率记为 $P(X|Y)$ ，可以计算为

$$P(X|Y) = \frac{P(X, Y)}{P(Y)}.$$

这告诉我们如果我们已经知道（或确定）了 Y 的结果， X 的结果有多大可能性。读作“在 Y 取 y 值的情况下， X 取 x 值的概率”。如果 X 和 Y 是独立的，则条件概率 $P(X|Y) = P(X)$ 和 $P(Y|X) = P(Y)$ 。

当两个随机变量 X 和 Y 不是相互独立的，他们的联合概率 $P(X, Y)$ 就不能简单地通过乘以各自的边缘概率来计算。相反，需要使用其他方法来确定这种依赖关系下的联合概率分布。有几种方法可以做到这一点：

1. 直接观察或数据统计：

- 如果可以直接观测 X 和 Y 的联合行为，或者有关于它们的历史数据，可以通过统计方法来估计它们的联合概率分布。

2. 条件概率公式：

- 一个重要的工具是条件概率公式， $P(X, Y) = P(Y|X) \times P(X)$ 或 $P(X, Y) = P(X|Y) \times P(Y)$ 。
- 这里， $P(Y|X)$ 是在 X 发生的情况下 Y 发生的概率，而 $P(X)$ 是 X 发生的边缘概率。
- 类似地， $P(X|Y)$ 是在 Y 发生的情况下 X 发生的概率，而 $P(Y)$ 是 Y 的边缘概率。

3. 利用已知的概率模型：

- 如果 X 和 Y 的依赖关系遵循某种已知的概率模型（如线性回归、贝叶斯网络等），可以使用这些模型来计算联合概率。

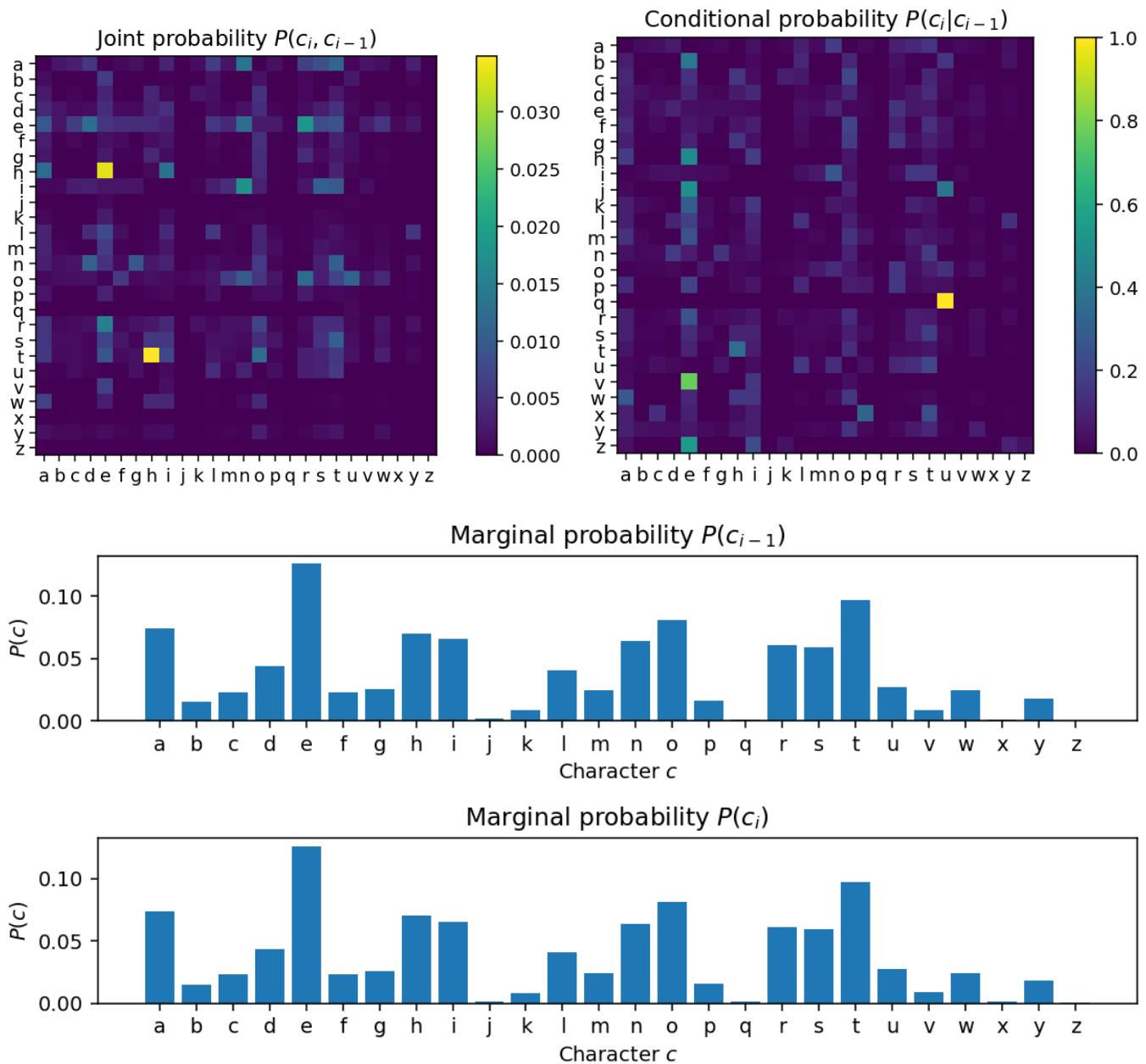
4. 模拟和计算实验：

- 在一些情况下，可以通过模拟实验来估计 X 和 Y 的联合概率。这通常涉及到生成大量 X 和 Y 的样本，然后计算它们一起出现的频率。

二元模型

我们可以在之前看到的文本字符模型的情况下考虑这些。从组成“变形记”（Metamorphosis）的字符代码向量中，我们可以取出每一对按顺序出现的字符对。也就是说，我们考虑某个索引 i 处的两个字符 c_{i-1} 和 c_i 。这被称为二元模型，还有一元、三元、 n 元模型的概念推广。一个“gram”指的是一个单位，如一个字符或单词，我们正在讨论的是字符二元模型。

- 二元模型的联合分布 $P(C_i = c_i, C_{i-1} = c_{i-1})$ 由每个字符对的归一化计数给出。
- 边际分布 $P(C_i = c_i)$ 可以通过对每个可能的字符 c_{i-1} 求和，从 $P(C_i = c_i, C_{i-1} = c_{i-1})$ 中计算得出，同样地可以边际化以找到 $P(C_{i-1} = c_{i-1})$
- 条件分布 $P(C_i = c_i | C_{i-1} = c_{i-1})$ 由联合分布除以 $P(C_{i-1} = c_{i-1})$ 的计数得出。它告诉我们在观察到一个字符 c_{i-1} 之后，观察到特定字符 c_i 的可能性有多大。



赔率、对数赔率 odds log odds

具有概率 p 的事件的赔率定义为：

$$\text{odds} = \frac{p}{1-p}$$

赔率是讨论不太可能的情景时更有用的单位（1:999的赔率，或999:1反对，比 $p = 0.001$ 更容易理解）。

对数赔率或**logit**对于非常不太可能的情景特别有用：

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

对数赔率的刻度与赔率分子中零的数量成比例。

对数概率

多个独立随机变量取一组值的概率可以通过乘积计算：

$$P(X, Y, Z) = P(X)P(Y)P(Z)$$

通常情况下

$$P(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n P(X_i = x_i)$$

我们经常需要计算这样的乘积，但是乘以许多小于1的值会导致数值问题：我们会得到浮点数下溢。相反，操纵对数概率在数值上更可靠，可以相加而不是相乘：

$$\log P(x_1, \dots, x_n) = \sum_{i=1}^n \log P(x_i)$$

The log-likelihood does not have this problem with underflow:

$$\log \square(c_1, c_2, \dots, c_n) = \sum_i \log P(c_i)$$

先验概率、似然概率、后验概率 Prior, likelihood, posterior

反转条件分布

我们经常想要知道在某个事件 B 发生的情况下，另一个事件 A 发生的概率；即 $P(A|B)$ 。但我们通常只能计算 $P(B|A)$ 。

这种情况通常是：

- 我们知道神秘实体的行为 $P(B|A)$ ；
- 我们知道我们看到的数据 $P(B)$ ；
- 我们知道神秘实体通常可能做什么 $P(A)$ ；
- 我们想要弄清楚神秘实体正在做什么 $P(A|B)$ 。

一般来说 $P(A|B) \neq P(B|A)$ ；这两个表达式可以完全不同。

通常，这类问题出现在：

- 我们想要知道在有一些 **证据**的情况下某个事件发生的概率（我的血液检测呈阳性，我患这种病的可能性有多大？）
- 但我们只知道在事件发生时观察到证据的概率（如果你患有这种病，血液检测会有 95% 的时间呈阳性）。

贝叶斯定理提供了正确反转概率分布的方法：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

这直接遵循概率的公理。贝叶斯定理是一个非常重要的定理，并且有一些令人惊讶的后果。

术语

- $P(A|B)$ 被称为**后验概率**——我们想要知道的，或在计算后会知道的
- $P(B|A)$ 被称为**似然概率**——事件 A 产生我们看到的证据的可能性
- $P(A)$ 是**先验概率**——不考虑证据时事件 A 的可能性
- $P(B)$ 是**证据**——不考虑事件时证据 B 的可能性。

贝叶斯定理提供了一个一致的规则，可以将某些先验信念与观察到的数据结合起来，估计一个结合了它们的新分布。

我们经常将这表述为一些我们想要知道的**假设 H**，给定我们观察到的**数据 D**，我们将贝叶斯定理写为：

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)}$$

H 和 D 在这个表达式中是随机变量。

对证据的积分

我们可以说后验概率与先验概率和似然概率的乘积成比例。但要评估其值，我们需要计算 P(D)，即证据。

一开始很难理解这代表什么。但一种思考方式是将其视为从联合分布 $P(H, D)$ 中边际化 $P(D)$ 的结果；即对每个可能的 D，对 H 的每个可能结果进行积分。

因为概率必须加起来等于 1，我们可以将 $P(D)$ 写为：

$$P(D) = \sum_i P(D|H_i)P(H_i)$$

对于一组离散结果 H_i ，或者 $P(D) = \int_H P(D|H)P(H)dH$ 对于连续结果的分布。

一般情况下这可能难以计算。对于只有两个可能结果的简单二元案例（H 只能是 0 或 1），贝叶斯定理可以写成：

$$P(H=1|D) = \frac{P(D|H=1)P(H=1)}{P(D|H=1)P(H=1) + P(D|H=0)P(H=0)}$$

自然频率

有一种解释此类问题的方法，可以大大降低做出错误判断的可能性。自然频率的解释涉及想象具有固定大小的具体人群（例如，房间里有10000人），并将人群的比例视为计数（房间里有多少人患有瘟疫？）。

我们可以使用它来可视化上述问题并解释明显的悖论。下面的图表展示了 $P(\text{瘟疫}) = 0.005$ (1在200) 的情况，再次使用5%准确的测试。

贝叶斯定理用于合并证据

贝叶斯定理是将先验信念和观察结合起来更新信念的正确方法。我们总是从一个概率分布（先验）转换为新的信念（后验），使用一些观察到的证据。这可以用于“学习”，其中“学习”意味着根据观察更新概率分布。它在必须将不确定信息融合在一起的任何地方都有巨大的应用，无论是来自多个来源（例如传感器融合）还是随时间变化（例如概率过滤）。

熵 Entropy

概率分布的一个关键特性是熵。直观上，这是一个衡量观察者在观察分布中的抽样时的“惊讶”程度的指标，或者换句话说，是分布可能代表的不同“状态”数量的（对数）测量。一个平坦、均匀的分布是非常“令人惊讶”的，因为其值很难预测；一个狭窄、尖峰的分布则不令人惊讶，因为其值总是非常相似。

这是一个精确的量化——它给出了分布中的信息。信息的单位通常是比特；1比特的信息告诉你一个是否问题的答案。熵告诉你确切需要多少比特（最少）来将一个分布中的值传达给一个已经知道该分布的观察者。或者，你可以将分布描述的不同状态数视为 $p = 2^{H(X)}$ —— 这个值被称为困惑度，它可以是分数。

熵只是对数概率的期望值

随机变量 X 的（离散）分布的熵可以计算为：

$$H(X) = \sum_x -P(X=x) \log_2(P(X=x))$$

这只是随机变量对数概率的期望值（对数概率的“平均”值）。

抛硬币

考虑抛硬币：这是从可以取两种状态，正面和反面的离散随机变量进行采样。

如果我们将两种可能的状态称为 0（正面）和 1（反面），我们可以用一个参数 q 来描述这个，其中 $P(X = 0) = q$ 和 $P(X = 1) = (1 - q)$ （这是因为 $P(X = 0) + P(X = 1)$ 必须等于 1——硬币必须落在一边或另一边。我们忽略边缘着陆！）。

如果这个过程非常有偏见，并且正面比反面更有可能 ($q \ll 0.5$)，观察者大多数时间都不会感到惊讶，因为预测正面将是一个好的赌注。如果这个过程是公正的 ($q = 0.5$)，观察者将无法预测正面或反面更有可能。我们可以写出这个分布的熵：

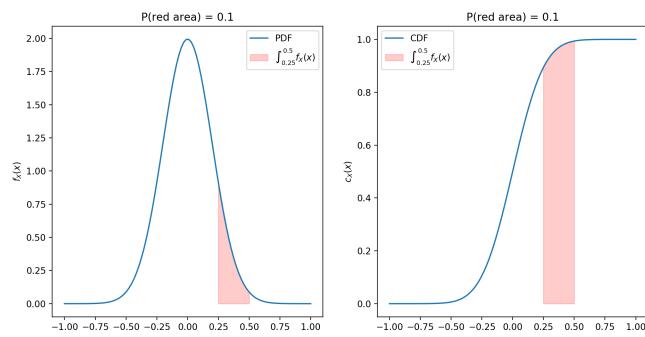
$$\begin{aligned} H(X) &= P(X = 0) \log_2 P(X = 0) + P(X = 1) \log_2 P(X = 1) \\ &= -q \log_2 q - (1 - q) \log_2 (1 - q) \end{aligned}$$

连续随机变量

连续变量的问题

连续随机变量由概率密度函数 (PDF) 定义，而不是概率质量函数 (PMF)。PMF 本质上只是一系列值的集合，但 PDF 是将其定义域中的任何输入映射到概率的函数。这看起来是一个微妙的区别（随着 PMF 的“区间”越来越多，它越来越接近 PDF，对吗？），但它带来了许多复杂性。

- 任何特定值的概率都是 $P(X = x) = 0$: 对于每个可能的 x 都是零，但分布函数的支持集 (PDF 不为零的任何地方) 中的任何值都是可能的。
- 无法像对 PMF 那样直接从 PDF 中抽样。但有几种技巧可以从连续分布中抽样。
- 我们无法像对经验分布那样仅通过观察次数来估计真实 PDF。这永远无法“填满”PDF，因为它只适用于具有零“宽度”的单个值。
- 贝叶斯定理易于应用于离散问题。但我们如何使用贝叶斯定理对连续 PDF 进行计算？
- 简单的离散分布没有维度的概念。但我们可以将 \mathbb{R} 中或在向量空间 \mathbb{R}^n 中有连续值，代表随机变量取向量值的概率，甚至是其他泛化（矩阵、复数或四元数等其他领域，甚至更复杂的结构如黎曼流形）



上的分布。

概率分布函数

随机变量 X 的 PDF $f_X(x)$ 将一个值 x （可能是一个实数、向量或任何其他连续值）映射到一个数字，即该点的密度。它是一个函数（假设是实向量上的分布） $\mathbb{R}^n \rightarrow \mathbb{R}^+$ ，其中 \mathbb{R}^+ 是正实数，且 $\int_x f_X(x) = 1$

- 虽然 PMF 的结果的概率最多为 1，但 PDF 的最大值不是 $f_X(x) \leq 1$ ——只是 PDF 的积分为 1。

PDF 在任何点的值不是概率，因为连续随机变量取任何特定数字的概率必须为零。相反，我们可以说连续随机变量 X 落在区间 (a, b) 内的概率是：

$$P(X \in (a, b)) = (a < X < b) = \int_a^b f_X(x) dx$$

支持集

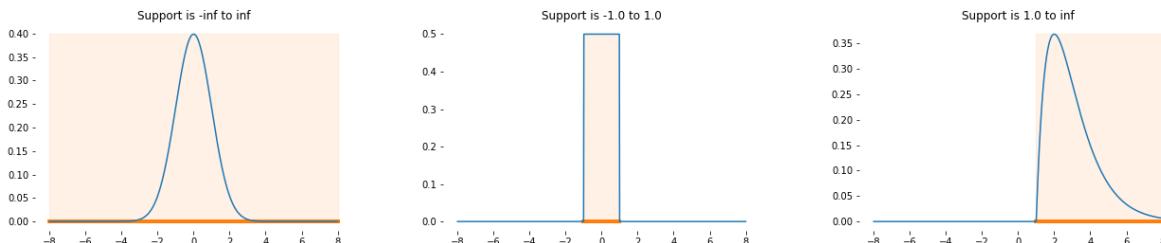
PDF 的支持集是其映射的非零密度的域。

$$\text{supp}(x) = x \text{ 使得 } f_X(x) > 0$$

一些 PDF 在固定区间内有密度，并在其他地方密度为零。均匀分布就是这样，它在固定范围内密度恒定，并且在其他地方为零。这被称为**紧凑支持**。我们知道从这种 PDF 的随机变量中抽样将总是在这个支持范围内给出值。一些 PDF 在无限域上有非零密度。正态分布就是这样。从正态分布中抽样可能会得到任何实际值；只是更有可能靠近分布的均值而不是远离。这是**无限支持**。

示例

无限支持 紧凑支持 半无限支持



累积分布函数

实值随机变量的累积分布函数 (CDF) 为

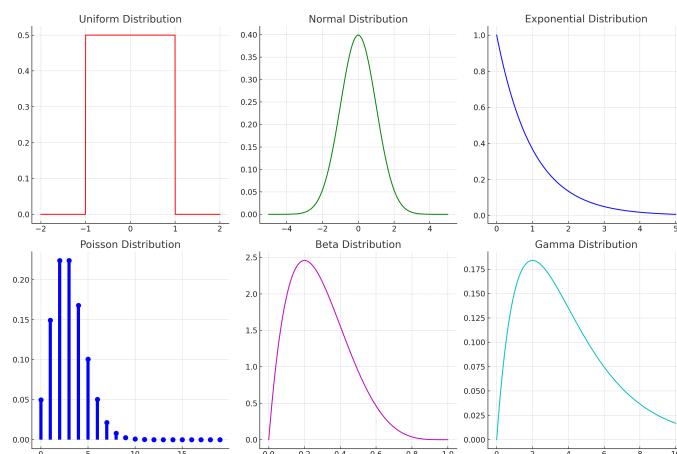
$$F_X(x) = \int_{-\infty}^x f_X(x) = P(X \leq x)$$

与 PDF 不同，CDF 总是将 x 映射到 $[0,1]$ 的值域中。对于任何给定的值 $F_X(x)$ 告诉我们小于或等于 x 的概率质量有多少。给定一个 CDF，我们现在可以回答问题，比如：随机变量 X 取 3.0 到 4.0 之间的值的概率是多少？

$$P(3 \leq X \leq 4) = F_X(4) - F_X(3)$$

这是一个概率。有时使用 CDF 而不是 PDF 进行计算更方便或有效。

概率密度分布



上图展示了六种常见的概率密度分布的图形及其公式和参数解释：

1. 均匀分布 (Uniform Distribution):

- 公式: $f(x) = \frac{1}{b-a}$ for $a \leq x \leq b$
- 参数: a (最小值), b (最大值)

- The graph shows a uniform distribution with the formula $f(x) = \frac{1}{b-a}$ for $a \leq x \leq b$.
Parameters: a (minimum value), b (maximum value).

2. 正态分布 (Normal Distribution) :

- 公式: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
- 参数: μ (平均值), σ (标准差)
- The graph depicts a normal distribution with the formula $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$.
Parameters: μ (mean), σ (standard deviation).

3. 指数分布 (Exponential Distribution) :

- 公式: $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$
- 参数: λ (率参数)
- The exponential distribution graph illustrates the formula $f(x) = \lambda e^{-\lambda x}$ for $x \geq 0$. Parameter: λ (rate parameter).

4. 泊松分布 (Poisson Distribution) :

- 公式: $f(x) = \frac{\lambda^x e^{-\lambda}}{x!}$ for $x = 0, 1, 2, \dots$
- 参数: λ (事件发生的平均次数)
- The Poisson distribution graph is described by the formula $f(x) = \frac{\lambda^x e^{-\lambda}}{x!}$ for $x = 0, 1, 2, \dots$.
Parameter: λ (average number of events).

5. 贝塔分布 (Beta Distribution) :

- 公式: $f(x) = \frac{x^{\alpha-1} (1-x)^{\beta-1}}{B(\alpha, \beta)}$ for $0 \leq x \leq 1$
- 参数: α 和 β (形状参数)
- The beta distribution is defined with the formula $f(x) = \frac{x^{\alpha-1} (1-x)^{\beta-1}}{B(\alpha, \beta)}$ for $0 \leq x \leq 1$. Parameters: α and β (shape parameters).

6. 伽马分布 (Gamma Distribution) :

- 公式: $f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}$ for $x > 0$
- 参数: k (形状参数), θ (尺度参数)
- The gamma distribution graph uses the formula $f(x) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-x/\theta}$ for $x > 0$. Parameters: k (shape parameter), θ (scale parameter).

PDF 示例：正态分布

所有连续 PDF 中最普遍的是正态或高斯分布。它将概率分配给实数值 $x \in \mathbb{R}$ (换句话说, 由所有实数组成的样本空间)。它的密度由 PDF 给出:

$$f_X(x) = \frac{1}{Z} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

您不需要记住这个公式, 但了解其本质上只是 e^{-x^2} 加上一些缩放因子以标准化它是很有用的 —— 这被称为平方指数函数。

旁注： $Z = \sqrt{2\pi}\sigma^2$ ，但您不需要记住这一点。

我们使用简写符号来表示连续随机变量的分布， $\text{variable} \sim \text{distribution}(\text{parameters})$ ，其中 \sim 读作“按照分布”。对于正态分布，这是：

$$X \sim \mathcal{N}(\mu, \sigma^2),$$

读作

“随机变量 X 按照 [N]ormal 分布，均值为 μ ，方差为 σ^2 ”

这意味着 X 的密度函数由具有特定参数 μ 和 σ^2 的正态密度函数类定义； $f_X(x) = \frac{1}{Z} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ 。

还有其他符号用于其他连续分布，包括 $\Gamma(\alpha, \beta)$, $\beta(\alpha, \beta)$, $t(v)$, $\chi^2(k)$, …，在这个课程中我们不会涉及。如果您想了解更多，请查阅参考资料。

位置和规模

正态分布将最高密度点放在其中心 μ (“均值”) 处，以 σ^2 (“方差”) 定义的扩散。这可以被认为是密度函数的位置和规模。大多数标准的连续随机变量 PDF 都有位置（密度集中的地方）和规模（密度的分布范围）。

正态模型 Normal model

这似乎是一个非常有局限性的选择，但有两个很好的理由使它在许多情况下都能很好地用作模型：

1. 正态变量具有很好的数学特性，易于分析处理（即无需依赖数值计算）。
2. 中心极限定理告诉我们，随着求和变量数量的增加，任何随机变量之和（无论它们如何分布）都将趋向于列维稳定分布*。对于大多数随机变量来说，这意味着正态分布（一种特定的列维稳定分布）。

中心极限定理 Central limit theorem

如果我们将 $Y = X_1 + X_2 + X_3 + \dots$ ，组成许多随机变量的和，那么对于 X_1, X_2, \dots 中的每个变量可能具有的几乎任何 PDF， Y 的 PDF 都将是近似正态的，即 $Y \sim \mathcal{N}(\mu, \sigma^2)$ 。这意味着，任何涉及许多随机成分混合物的过程，在各种条件下都将趋于高斯。

多元分布：在 \mathbb{R}^n 上的分布

连续分布将离散变量（概率质量函数）（例如在 \mathbb{Z} 上）概括为通过概率密度函数在 \mathbb{R} 上的连续空间。

概率密度可以进一步推广到向量空间，特别是到 \mathbb{R}^n 。这扩展了 PDF 以便在整个向量空间内分配概率，同时受到（多维积分）的约束

$$\int_{\mathbf{x} \in \mathbb{R}^n} f_X(\mathbf{x}) = 1.$$

这与下面的表达式相同：

$$\int_{x_0=-\infty}^{x_0=\infty} \int_{x_1=-\infty}^{x_1=\infty} \dots \int_{x_n=-\infty}^{x_n=\infty} f_X([x_0, x_1, \dots, x_n]), dx_0 dx_1 \dots dx_n = 1.$$

在向量空间上具有 PDF 的分布称为**多元分布**（这个名字并不是很好；向量分布可能更清晰）。在许多方面，它们的工作方式与单变量连续分布相同。然而，它们通常需要更多参数来指定其形式，因为它们可以在更多维度上变化。

多元均匀分布

多元均匀分布特别容易理解。它在向量空间 \mathbb{R}^n 的某个（轴对齐的）盒子内分配相等的密度 $f_X(x_i) = f_X(x_j)$ ，使得

$$\int_{\mathbf{x}} f_X(\mathbf{x}) = 1, \mathbf{x} \in \text{a box.}$$

从中抽样非常简单；我们只需从范围 $[0,1]$ 的一维均匀分布中独立抽样，以获得我们向量样本的每个元素。这是从单位盒子内的 n 维均匀分布中抽取的。

正态分布

正态分布（如上所述）被广泛用作连续随机变量的分布。它可以为任何维度的随机变量定义；在统计术语中称为**多元正态**。在第5单元中，我们看到了**均值向量 μ** 和**协方差矩阵 Σ** 的概念，这些概念以椭圆的形式捕捉了数据集的“形状”。这些实际上是多元正态分布的参数化。

多元正态分布完全由均值向量 μ 和协方差矩阵 Σ 指定。如果你将正态分布想象成空间中的球形质量，均值平移质量，协方差应用变换矩阵（缩放、旋转和剪切）于球体。

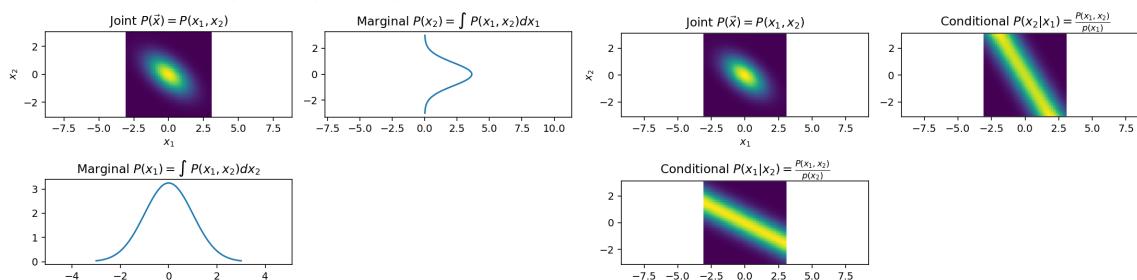
就像均匀分布一样，我们可以想象从具有每个维度独立正态分布的“单位球”中抽取样本。这些样本由协方差矩阵 Σ 和均值向量 μ 线性变换，就像上面的 A 和 b 一样（尽管 Σ 实际上是 $A^{-\frac{1}{2}}$ 出于技术原因）

联合和边际分布

现在我们可以讨论**联合概率密度函数**（在所有维度上的密度）和**边际概率密度函数**（在某些维度的子集上的密度）。

例如，考虑 $X \sim N(\mu, \Sigma), X \in \mathbb{R}^2$ ，一个二维（“双变量”）正态分布。我们可以看一些 PDF 的例子，展示：

- 联合 $P(\mathbf{X})$
- 边际 $P(X_1)$ 和 $P(X_2)$
- 条件概率 $P(X_1|X_2)$ 和 $P(X_2|X_1)$



期望计算

例如，计算随机变量函数的期望对于连续随机变量来说通常很难。积分：

$$\mathbb{E}[g(\mathbf{X})] = \int_{\mathbf{x}} f_{\mathbf{X}}(\mathbf{x}) g(\mathbf{x}) d\mathbf{x}$$

可能是不可解的。然而，计算任何可能的 x 的 $g(x)$ 通常非常容易。如果我们能够以某种方式从分布 $P(X = x)$ 中抽样，那么我们可以非常容易地近似这个值：

$$\mathbb{E}[g(X)] = \int_x f_X(x)g(x) dx \approx \frac{1}{N} \sum_{i=1}^N g(x_i),$$

其中 x_i 是从 $P(X = x)$ 中随机抽样得到的，由 PDF $f_X(x)$ 定义。随着 N 的增大，这个近似值会变得更好。

三种方法

我们将看到三种不同的推断方法：

- **直接估计参数**，其中我们定义观察的函数，直接估计分布参数的值。这需要我们假设控制神秘实体的分布的形式。这种方法非常高效，但只适用于非常特定类型的模型。我们需要为我们想要估计的每个特定分布有**估计函数**，这些函数将观察映射到参数估计。
- **最大似然估计参数**，其中我们使用**优化**来找到使观察结果看起来尽可能可能的参数设置。我们可以将其视为调整某个预定义模型的参数，直到它们与我们拥有的观察结果“最佳对齐”。这需要一个迭代优化过程，但它适用于任何已知似然函数（即我们可以计算观察结果被该模型生成的可能性）的模型。
- **贝叶斯、概率**方法明确地使用概率分布对神秘实体的行为进行编码。在贝叶斯模型中，我们假设参数本身的分布，并将参数视为随机变量。我们对这些参数有一个初始假设（“先验”），我们使用观察结果更新这一信念，以将我们对参数的估计调整到更紧密（希望如此）的分布（“后验”）。与其他方法不同，我们不估计单个“参数设置”，而是总是有一个随观察数据变化的可能参数的分布。这种方法更加健壮，可以说是更加连贯的推断方式，但它更难表示和计算。我们需要参数的**先验**，以及一个**似然函数**，告诉我们数据在特定参数设置下被生成的可能性。

估计器

与离散分布不同，离散分布的 PMF 可以通过使用经验分布直接从观察中估计（就像我们对《罗密欧与朱丽叶》所做的那样），但对于连续分布并没有类似的直接程序。

对于许多连续分布，统计学家已经开发出**估计器**；这些函数可以应用于观察集合，以估计定义可能生成这些观察的分布的概率密度函数的最可能参数。

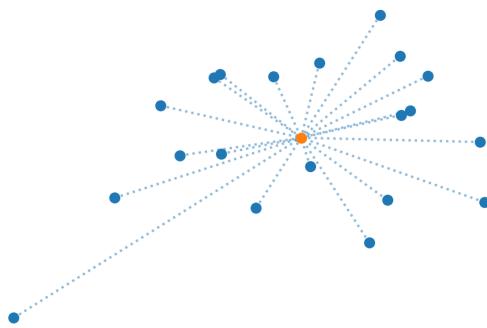
必须提前决定分布的形式（例如，假设数据由近似正态分布生成）；这通常称为**模型**。然后可以在这个模型的假设下计算特定参数。

直接估计

进行推断的一种方法是，如果我们假设分布的特定形式（例如假设它是正态的），使用**估计器**来估计这种人群分布的参数（例如均值和方差）。这些**估计器**是通过我们可以应用于数据的汇总函数——**统计量**来计算的。这些**估计器**需要为每个特定类型的问题特别推导。

例如，一组观察样本的算术平均值和标准偏差是**统计量**，它们是 μ 和 σ 正态分布的**估计器**。如果我们有（被认为已经）从正态分布中抽取的观察结果，我们可以仅通过计算均值和标准偏差来估计该分布的参数 μ 和 σ 。

换句话说，我们可能想要知道“真实”的平均应用评分——产生应用评分的分布的人群均值。我们假设一个随机过程正在创建这些评分，其操作特性（参数）我们可以从样本中学习。但我们只能通过测量用户实际评价应用的特定子集的一些评分，并计算结果的统计量——样本均值，来观察有限的样本评分。



图片：直接估计使用数据的函数，即统计量，直接估计参数；例如，点云的算术平均值估计人群的均值向量。

标准估计器

均值

算术平均值是样本值 x_1, x_2, \dots, x_n 的总和除以值的数量：

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i$$

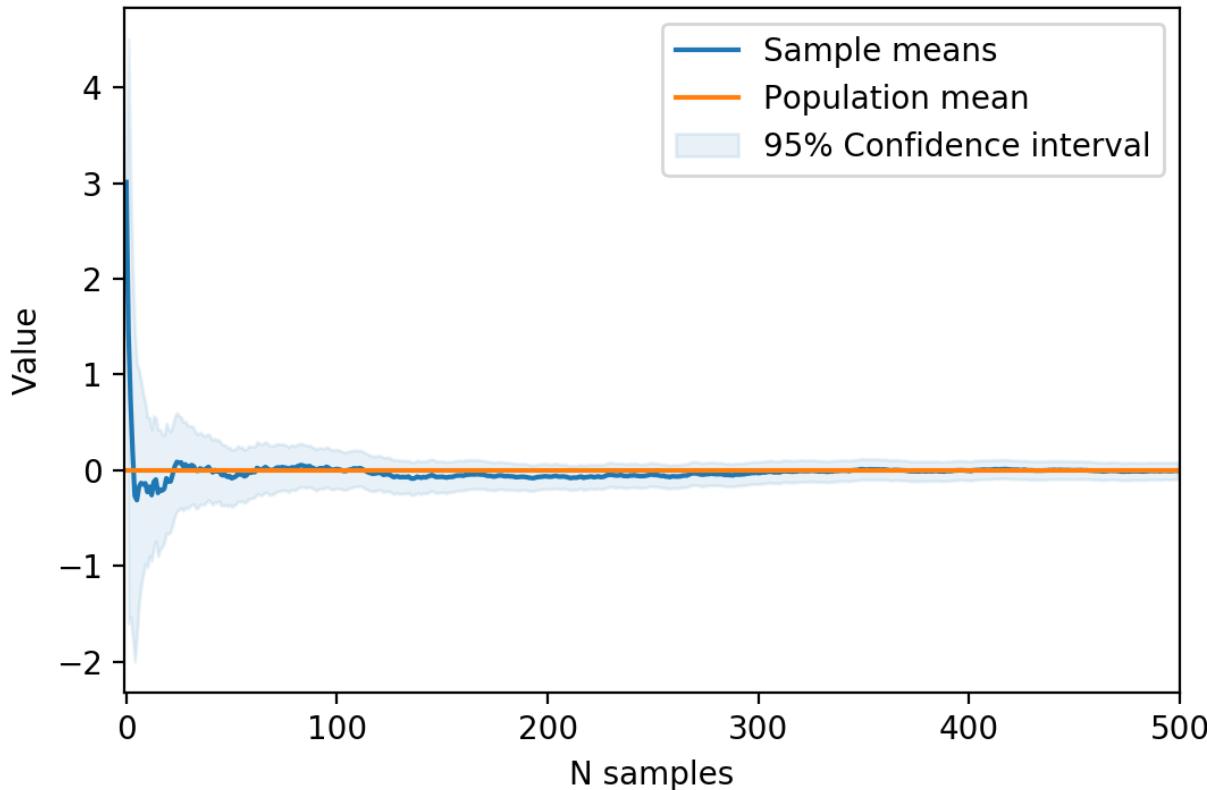
样本均值

随机变量 X 的总体均值是 $\mu = E[X]$ 。事实证明，观察样本的算术平均值或样本均值（我们用一个小帽子 $\hat{\mu}$ 表示）是真实总体均值 μ 的一个好的（脚注：统计学家称之为“无偏”的）估计器。随着样本数量的增加，我们对总体均值 μ 的估计 $\hat{\mu}$ 会越来越好。

重要的是区分以下观念：

- 总体均值 μ ，通常是存在的但不能直接知道。它是随机变量的期望值 $E[X]$ 。
- 样本均值 $\hat{\mu}$ ，只是我们看到的样本的算术平均值（例如，通过 `np.mean(x, axis=0)` 计算得到）。

样本均值是一个统计量（观察值的函数），它是总体均值的估计器（可能是分布的一个参数）。可以对这个估计设定特定的界限；标准误差给出了我们预期样本的算术平均值与总体均值接近程度的度量，尽管其解释并不直接。



均值测量的是数值集合的**中心倾向**。均值向量**将这一概念推广到更高维度。

方差和标准偏差

样本方差是序列中每个值与该序列均值的差的平方：

$$\hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu})^2.$$

它是总体方差的估计器， $\mathbb{E}[(X - \mathbb{E}[X])^2]$

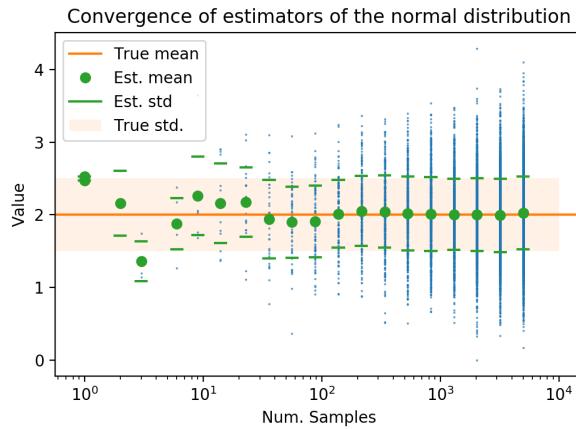
样本标准偏差只是这个值的平方根。

$$\hat{\sigma} = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \hat{\mu})^2}$$

方差和标准偏差衡量一组值的扩散程度。协方差矩阵 Σ 将这个概念推广到更高维度。

与正态分布的关系

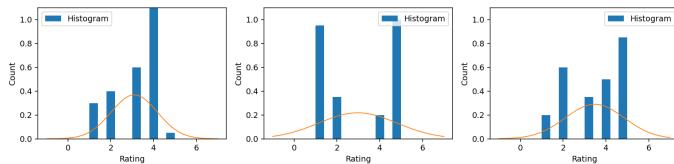
如果我们假设我们的数据是由正态分布生成的，那么统计量均值 $\hat{\mu}$ 和方差 $\hat{\sigma}^2$ 估计该正态分布的参数 μ, σ , $\Sigma(\mu, \sigma)$ 。即使底层过程不完全是正态的，由于中心极限定理，它可能非常接近正态。即使这不适用，均值和方差仍然是有用的描述性统计量。



拟合

估计可能产生应用评分的正态分布的参数意味着什么？我们正在拟合一个受 PDF 控制的分布到一组观察结果。在我们的离散示例中，我们可以通过计算经验分布（假设我们有足够的样本）来简单地拟合分布。但估计 PDF 需要一些结构，即具有一些参数化的函数空间。

我们可以可视化这一点：



从模型中取样

我们可以从拟合分布中抽取样本，并与结果进行比较。它们不会是一个很好的代表，因为我们的数据显然不是正态的。但它们显示了我们驯服的神秘实体所产生的结果，让我们可以评估我们的**建模假设**--即应用程序评分的特征仅为平均值和标准偏差。

最大似然估计：通过优化进行估计

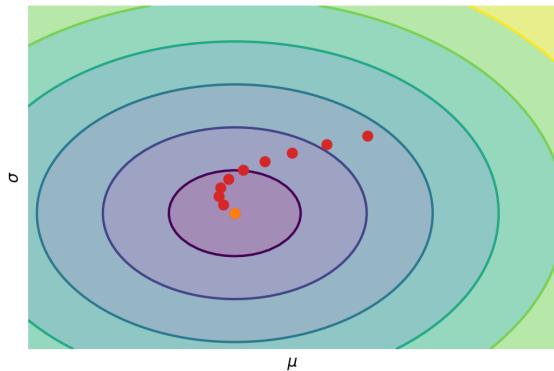
如果我们没有现成的估计器来估计我们想要的参数，该怎么办？我们如何进行推断？我们如何将分布参数拟合到观察结果上？

在许多情况下，我们可以计算特定潜在随机分布生成观察结果的似然。这就是我们之前看到的似然。对于 PDF，值 x 的似然只是 PDF 在 x 处的值： $f_X(x)$ 。似然是数据的函数，基于某些特定参数的假设。

许多独立观察的似然是各个似然的乘积，对数似然是各个对数似然的总和。

$$\log \square(x_1, \dots, x_n) = \sum_i \log f_X(x_i)$$

想象一下，我们有一个我们不知道参数的**估计器**的分布。如果有一些数据，我们该如何估计它们可能是什么？我们可以将所有参数写成向量 θ ；例如，正态分布将有 $\theta = [\mu, \sigma]$ 。



图片：最大似然估计使用优化来最大化数据的似然函数，并找到给定数据的参数的最优值。它不需要特殊的估计器函数；只需要一个似然。

优化解决所有问题

尽管我们没有固定的、闭合形式的函数来估计参数，但有了似然函数，我们可以应用优化来计算在此参数设置下我们实际观察到的数据最有可能的情况。这相当于在我们的“神秘实体”机器上转动旋钮，直到我们找到一个在我们向其输入样本时输出最大似然值的设置。

如果似然取决于分布的某些参数 θ ，那么我们写作：

$$\square(\theta|x)$$

然后，我们可以定义一个目标函数：最大化对数似然，或等价地最小化负对数似然。

$$\theta^* = \arg \min_{\theta} L(\theta)$$

$$L(\theta) = -\log \square(\theta|x_1, \dots, x_n) = -\sum_i \log f_X(x_i; \theta),$$

假设我们的 $f_X(x_i)$ 可以写作 $f(x; \theta)$ 来表示在给定 θ 的特定参数选择下 f 的 PDF。

最大似然估计

这与我们之前看到的近似目标函数 $|f(x; \theta) - y|$ 非常相似，但我们有 $y = 0$ 并且 f 只有标量输出，所以不需要范数。我们已经知道如何解决这种问题；只需优化。这被称为**最大似然估计**，是确定给定一些观察结果的分布参数的一般技术。它将找到解释观察结果如何产生的参数的最佳设置。

如果我们幸运的话，这将是可微的，我们可以使用梯度下降（或随机梯度下降——注意目标函数是简单子目标函数的总和）。如果我们不幸，我们可以退回到效率较低的优化器。在这种情况下，我们不需要特殊的估计器，只要我们能够评估任何参数设置 θ 下的 PDF $f(x; \theta)$ 。这适用于更广泛的概率分布类别。

使用 MLE 拟合正态分布

例如，我们可以考虑从一组（假设是独立的）样本中估计正态分布的均值和方差的问题，而不使用估计器；比如我们的应用评分。为此，我们需要能够计算任何给定样本的似然，并取所有这些样本的乘积（或者更确切地说是对数似然的总和）。

这给了我们目标函数。如果我们改变符号，使之最小化负对数似然，我们将寻找使数据最有可能的参数向量。

对于单变量正态分布，参数只是 μ 和 σ ，所以 $\theta = [\mu, \sigma]$ 。

在这种情况下，当然，我们确实有估计器；但即使我们只有似然函数，这个程序也同样有效。

混合模型

但如果我们的模型比单纯的正态分布更复杂呢？我们可以想象以某种其他方式建模，也许能够捕捉到应用 B 似乎在两侧都有“峰值”的事实。一个非常简单的模型是**高斯混合**，我们只是说我们期望我们试图拟合的分布的 PDF 是 N 个不同正态分布（“成分”） $\square_i(\mu_i, \sigma_i)$ 的加权组合（凸和），每个都有自己的 μ_i, σ_i ，以及表示这个“成分”重要性的权重因子 λ_i ，其中 $\sum_i \lambda_i = 1$ 。这使我们能够表示“多峰”的分布。

这个模型让我们想象评分可能属于一个“簇”或另一个。每个簇的位置和大小由该成分的 μ_i 和 σ_i 给出， λ_i 给出数据落入该簇的可能性。

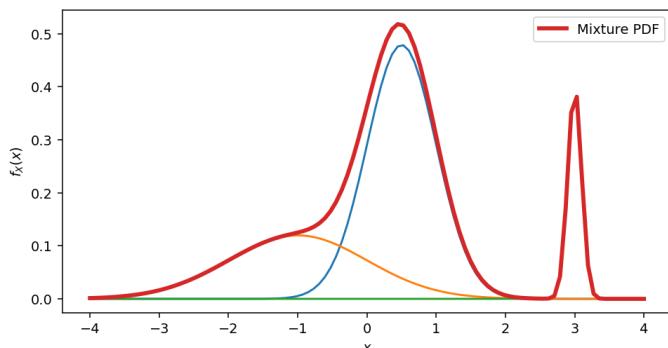
我们可以轻松地绘制这个函数的 PDF；它只是：

$$f_X(x) = \sum_i \lambda_i n_X(x; \mu_i, \sigma_i),$$

其中

$$n_X(x; \mu, \sigma) = \frac{1}{Z} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

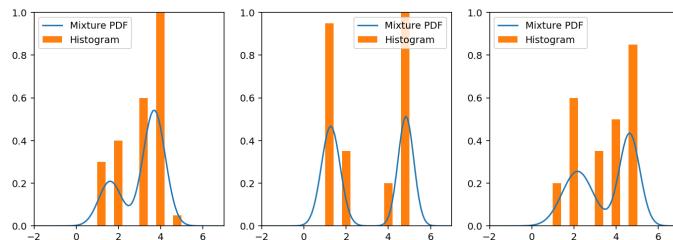
是标准正态 PDF 函数。



拟合混合模型

这是对我们应用评分更为合理的模型，并可能是一个更好的模型。但我们如何拟合它？即使我们提前固定 N ，我们肯定没有任何直接的估计器可以估计正态 PDF 和的均值、标准偏差（和权重）。这不是我们知道如何做的事情。

但是，（对数）似然在代码中写起来很简单。对于每个观察值 x ，我们只需计算每个成分的加权 PDF 之和，结果就是该观察值的似然。这是数据的函数 $\square(\theta|x)$ ，我们的参数向量是 $\theta = [\mu_1, \sigma_1, \lambda_1, \mu_2, \sigma_2, \lambda_2, \dots]$ 。



贝叶斯推断

贝叶斯推断涉及到以一种截然不同的方式思考问题。贝叶斯人将他们正在估计的分布的参数视为本身是随机变量，具有自己的分布。

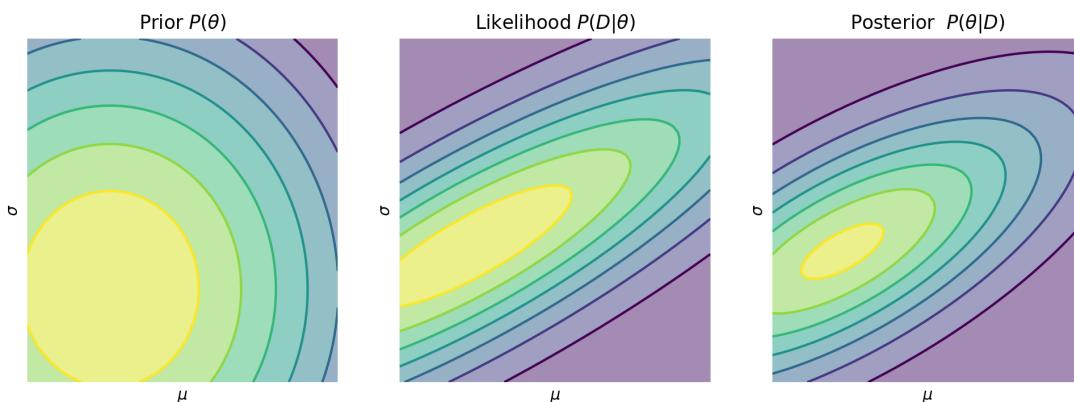
先验分布定义在这些参数上（例如，我们可能相信平均应用评分可以以相同的概率取任何值 1.0-5.0），随着更新到来的证据被贝叶斯规则结合起来，以精炼我们对参数分布的信念。我们再次认为我们的分布由某个参数向量 θ 描述，我们想要精炼一个关于可能的 θ 的分布。

我们不考虑估计器或它们的抽样分布，谈论找到最佳参数设置是没有意义的；我们只能对参数设置有信念，这必须以概率方式表示。我们不寻求找到最有可能的参数设置（如在直接估计或 MLE 中），而是推断一个与数据兼容的可能参数设置的分布。

我们谈论的是在给定一些**先验信念**和一些**证据**的情况下，推断参数的**后验分布**。我们假设我们有一个似然函数 $P(D|\theta)$ 和一个关于参数的先验 $P(\theta)$ ，然后我们可以使用贝叶斯规则：

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

这给了我们一个关于 θ 的新分布，给定一些观察结果。贝叶斯规则同样适用于连续分布和离散分布，但在“闭合形式”（即代数上）中的计算要困难得多。



在某些情况下，可以用闭合形式找到 $P(\theta|D)$ ，但代数通常很复杂，模型选择有限；我们不会讨论如何做到这一点。然而，当这是可能的，它更具计算效率。相反，我们将从计算角度出发，找到一种从后验分布 $P(\theta|D)$ 中抽取样本的方法。

示例

记住，我们假设应用评分是由这个函数生成的：

```
return np.random.normal(mu, sigma)
```

我们想要推断关于 μ 和 σ 的分布（不是关于观察结果的分布！）。也就是说，我们将把参数本身视为具有自己分布的随机变量，并使用贝叶斯推理（即应用贝叶斯规则）在给定一些先验和一些观察到的证据的情况下推断参数的后验分布。

参数和样本

我们有一组观察结果 $D = x_1, x_2, \dots$ ，代表实际的应用评分。我们不确定这些评分真正告诉我们关于未见潜在用户人群的信息有多少。我们将分布参数表示为 $\theta = [\mu, \sigma]$ ，并可以讨论 $P(\theta)$ ，即参数向量上的分布。

先验

假设我们对 θ 有一些先验信念 $P(\theta)$ ：例如，这可能是一个非常简单的假设，即我们的先验是 μ 和 σ 均匀分布

- $\mu \sim U(1, 5)$
- $\sigma \sim U(0, 10)$

这给了我们 $P(\theta)$ 的形式。我们可以编写代码中从这个分布中抽样的方法，以及评估任何给定参数设置的先验概率的方法。正如我们稍后将看到的，我们需要这两个函数来进行贝叶斯推断。

```
def sample_prior():
    return [np.random.uniform(1,5),
            np.random.uniform(0,3)]

def log_prior(theta):
    mu, sigma = theta
    # p(x) = 1/x if x ~ U(0,x), and 0 otherwise
    return (scipy.stats.uniform(1.0, 4.0).logpdf(mu) +
            scipy.stats.uniform(0.0, 3.0).logpdf(sigma))
```

似然

我们需要能够定义一个似然函数。这是一个给定某些参数设置的数据函数，在这种情况下，它与用于最大似然估计的似然函数相同：一个样本的似然只是在该点评估的正态 PDF，所有样本的似然是这些似然的乘积。

```
def log_likelihood(samples, theta):
    # for a *known* mu, theta
    # compute the log PDF at each sample, and sum them
    return np.sum(scipy.stats.norm(theta[0],
                                   theta[1]).logpdf(samples))
```

推断

我们如何计算后验分布 $P(\theta|D)$ ？我们不会讨论如何以闭合形式（作为一个函数）找到这个——有时这是可能的，但在数学上涉及到复杂性，因为我们需要处理关于 θ 的分布——而是讨论如何在给定先验、似然和一些观察结果的情况下从这个后验中抽取样本。

关于如何解决这个问题有大量文献，其中有一些棘手的部分：

- $P(D|\theta)$ 需要为关于 θ 的分布计算，而不仅仅是一些数字。仅仅计算一个特定 θ 的概率是不够的；我们必须处理分布函数。
- $P(D) = \int_{\theta} P(D|\theta)P(\theta)$ 可能是难以解决的。

使其可解决

有许多方法可以简化这个问题，使其可解决。我们将使用两种：

样本就够了

我们通常无法计算 $P(\theta|D)$ ，因为我们不知道如何对函数的乘积进行操作。但对于具体的、具体的 θ 值来说，这通常是微不足道的。例如，对于一个给定的固定 θ ，我们可以计算那个特定示例的似然和先验。

这让我们想到了从后验分布 $P(\theta|D)$ 抽取样本的想法，而不是试图精确计算分布。

仅关注相对概率

我们可以做一个简化的假设：我们只关心具有我们实际拥有的数据 D 的不同参数设置的相对概率。即我们有

$$P(\theta|D) \propto P(D|\theta)P(\theta)$$

并忽略这是由某个未知常数 $Z = \frac{1}{P(D)}$ 缩放的后验事实。这只有在我们在这个例子中只考虑一个模型和一组数据的情况下才有意义。

马尔可夫链蒙特卡洛

我们可以通过对模拟退火算法进行非常简单的修改，实现从（相对）后验分布中抽样的程序。

这定义了一种通过参数设置空间的随机游走，提出对参数设置的小的随机调整，并在使估计更有可能的情况下接受“跳跃”，或者如果不是，则按照 $P(D|\theta)P(\theta)$ 变化的比例接受。这种方法的优点是我们可以使用来自 θ 的确定样本，我们不必进行任何复杂的积分。这种方法称为马尔可夫链蒙特卡洛

我们所需的只是评估 $P(\theta)$ （先验）和 $P(D|\theta)$ （似然）的方法，对于任何特定的 θ 。

实践中的 MCMC：抽样问题

我们将使用马尔可夫链蒙特卡洛来解决贝叶斯推断问题。MCMC 方法的伟大之处在于，你基本上可以直接写下模型然后直接进行推断。无需推导复杂的近似，也不必限制我们自己只使用可以解析计算答案的有限模型。本质上，无需手工计算；一切都是算法完成的。

MCMC 允许我们从任何我们无法直接抽样的分布 $P(X = x)$ 中抽取样本。特别是，我们将能够从参数的后验分布中抽样。

MCMC 方法的坏处在于，即使它渐近地会做出“正确的事情”，但对于实际执行的样本运行来说，抽样策略的选择有很大的影响。贝叶斯推断应该只依赖于先验和观察到的证据；但 MCMC 方法还取决于用于近似后验的抽样策略。

我们从哪个分布中抽样？

$$\text{在贝叶斯推断的情况下, } P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} = \frac{P(D|\theta)P(\theta)}{\int_{\theta} P(D|\theta)P(\theta)}.$$

- $P(\theta|D)$ 是后验, 即在数据 (观察结果) D 给定的情况下, 参数 θ 的分布, 使用:
- 似然 $P(D|\theta)$,
- 先验 $P(\theta)$ 和
- 证据 $P(D)$ 。

换句话说, 观察结果和先验给定的情况下, 参数的分布是什么? 如果我们假设, 如上所述, 我们不关心 $P(D)$, 因为我们只比较 θ 的不同可能值, 那么我们可以从与 $P(D|\theta)P(\theta)$ 成比例的分布中抽样。

Metropolis-Hastings

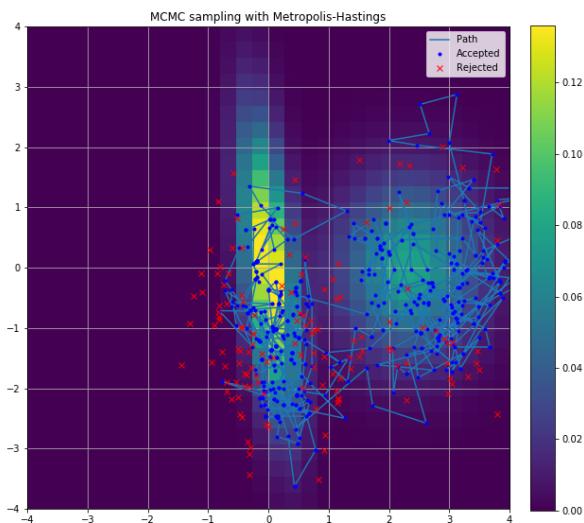
Metropolis-Hastings (或者简称 Metropolis) 是一种非常优雅且相对有效的实现 MCMC 算法的方法, 并且能够在高维空间中工作 (即当 θ 复杂时)。

Metropolis 抽样使用一个简单的辅助分布, 称为提议分布 $Q(\theta'|\theta)$, 来帮助从难以处理的后验分布 $P(\theta|D)$ 中抽样。这类似于我们在优化部分称之为邻域函数的东西。

Metropolis-Hastings 使用这个来漫游分布空间, 在使用 $Q(\theta'|\theta)$ 随机抽样 $P(\theta|D)$ 空间的位置时, 接受跳跃到新位置。这个随机游走 (一个马尔可夫链, 因为我们做出的随机跳跃只取决于我们当前的位置) 是“马尔可夫链蒙特卡洛”的“马尔可夫链”部分。

这就像模拟退火算法, 只是现在有一个函数 $f_X(\theta)$ 使某些步骤比其他步骤更有可能, 而不是似然函数。我们只需取当前位置 θ , 并提出一个新位置 θ' , 即从 $Q(\theta'|\theta)$ 中随机抽取的样本。通常这是一些非常简单的东西, 比如以 x 为均值、某个预设的 σ 为正态分布: $Q(\theta'|\theta) = \mathcal{N}(\theta, \sigma^2)$

We can show a simple demo of this, drawing samples from a tricky 2D probability distribution.



轨迹

MCMC 过程接受的样本历史被称为轨迹。例如, 我们可以通过查看轨迹的直方图来估计模型参数。轨迹是样本序列 $[x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(n)}]$, 通过 MCMC (大致) 从后验分布 $P(\theta|D)$ 中抽取。

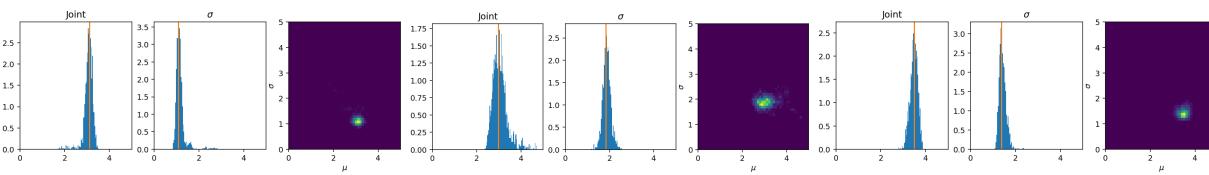
应用 MCMC

让我们将这个应用到我们的应用评分问题上。我们想要估计参数 μ 和 σ 的分布，给定一些观察（证据/数据） x_1, x_2, x_3, \dots ，假设这些观察来自我们认为是 $\mathcal{N}(\mu, \sigma)$ 的分布。

我们已经有了一个先验函数 `prior_pdf` 和一个似然函数 `likelihood`。所以我们可以使用与上面相同的程序来做这个，但这次从先验中随机抽取的各种随机初始条件开始随机过程，以确保我们不会卡在空间中的某个不好的部分。

预测后验：从模型中抽样

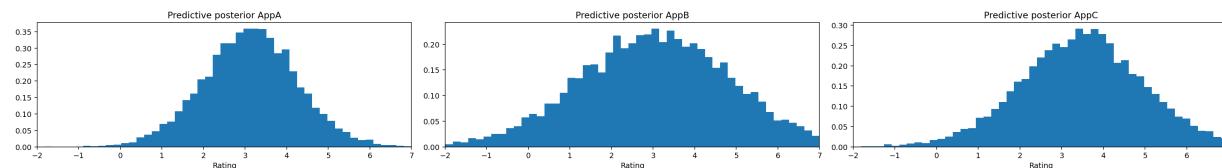
我们所绘制的是模型参数的后验分布的样本；即我们期望模型参数在观察到的数据和我们的先验下所呈现的值。



预测后验是我们预期看到的观察结果的分布；对未来样本的预测。这意味着从模型中抽取样本，同时在参数的后验分布上进行整合。通过从预测后验中抽样，我们正在生成新的合成数据，如果我们的模型是好的，这些数据应该具有与数据相同的统计特性。

我们可以通过一个两步骤的嵌套过程来做到这一点：

- 对于 n 次重复
 - 从我们的参数后验分布中抽取样本，以给出一个具体的分布
- 对于 m 次重复
 - 从这个具体的分布中抽取样本



Linear regression

线性回归

Linear regression is the fitting of a line to observed data. It assumes the mysterious entity generates data where one of the observed variables is scaled and shifted version of another observed variable, corrupted by some noise; a linear relationship. It is a very helpful lens through which different approaches to data modelling can be seen; it is pretty much the simplest useful model of data with relationships, and the techniques we use easily generalise from linear models to more powerful representations. 线性回归是对观察数据进行直线拟合的过程。它假设有一个神秘的实体生成数据，其中一个观察变量是另一个观察变量的缩放和移位版本，受到一些噪声的干扰；即线性关系。线性回归是通过这种方式来看待数据建模的不同方法的一个非常有用的角度；它几乎是最简单有用的数据关系模型，并且我们使用的技术可以从线性模型轻松泛化到更强大的表示形式。

The problem is to estimate what that scaling and shifting is. In a simple 2D case, this is the gradient m and offset c in the equation $y = mx + c$. It can be directly generalised to higher dimensions to find A and b in $y = Ax + b$, but we'll use the simple "high school" $y = mx + c$ case for simplicity.

问题是估计缩放和移位是什么。在一个简单的二维案例中，这是方程式 $y = mx + c$ 中的梯度 m 和偏移量 c 。它可以直接泛化到更高维度以在 $y = Ax + b$ 中找到 A 和 b ，但为了简单起见，我们将使用简单的“高中”案例 $y = mx + c$ 。

We assume that we will fit a line to *noisy* data. That is the process that we assume that is generating the data is $y = mx + c + \epsilon$, where ϵ is some noise term. We have to make assumptions about the distribution of ϵ in order to make inferences about the parameters.

我们假设我们将对带噪声的数据拟合一条线。也就是说，我们假设生成数据的过程是 $y = mx + c + \epsilon$ ，其中 ϵ 是一些噪声项。我们必须对 ϵ 的分布做出假设，以便对参数进行推断。

One simple assumption is that $\epsilon \sim \mathcal{N}(0, \sigma^2)$ i.e. that we have normally distributed variations in our measurements. So our full equation is: 一个简单的假设是 $\epsilon \sim \mathcal{N}(0, \sigma^2)$ ，即我们的测量中有正态分布的变化。所以我们的完整方程是：

$$y = mx + c + \mathcal{N}(0, \sigma^2),$$

or equivalently, putting the $mx + c$ as the mean of the normal distribution: 或者等价地，将 $mx + c$ 作为正态分布的均值：

$$y \sim \mathcal{N}(mx + c, \sigma^2)$$

Note that we assume that y is a random variable, x is known, and that m, c, σ are parameters that we wish to infer from a collection of observations: 注意，我们假设 y 是一个随机变量， x 是已知的，而 m, c, σ 是我们希望从一系列观察中推断出来的参数：

$$[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$$

In code, we could write down what we *assume* is generating data, our tame mysterious entity: 在代码中，我们可以写下我们假设正在生成数据的，我们的温顺的神秘实体：

```
def model(x, theta):
    m, c, sigma = theta
    y = np.random.normal(x * m + c, sigma)
    return y
```

通过直接优化进行线性回归

We saw how this problem could be solved as a **function approximation** problem using optimisation. We can write an objective function: 我们看到了如何通过优化将这个问题解决为一个**函数逼近**问题。我们可以写一个目标函数：

$$L(\theta) = |f(x; \theta) - y|,$$

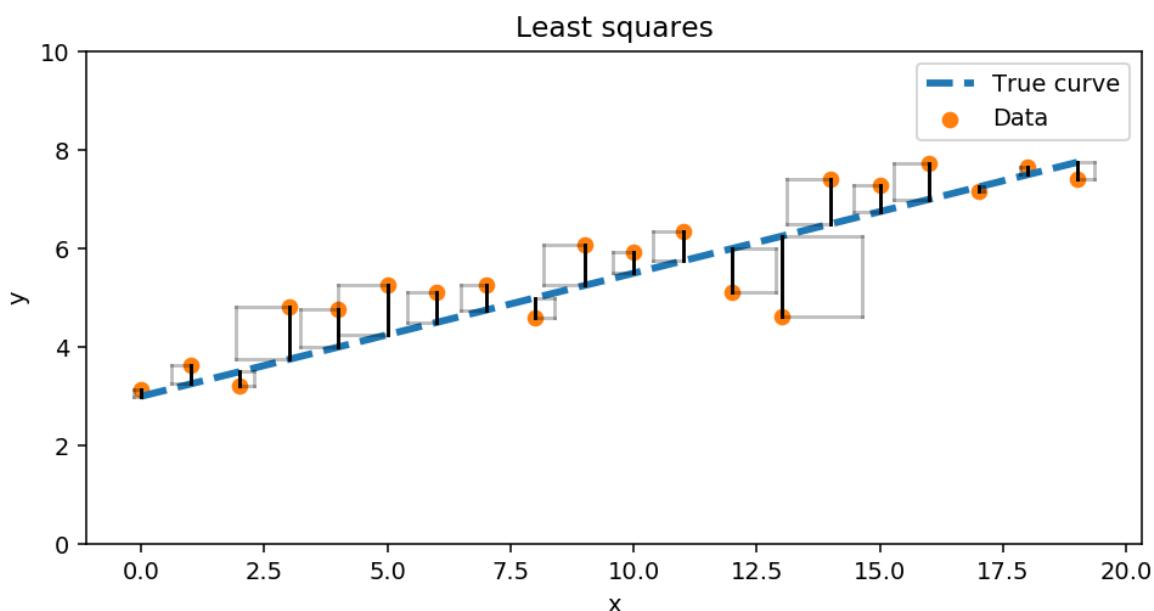
where $\theta = [m, c]$ and $f(x; \theta) = \theta_0 x + \theta_1$. 其中 $\theta = [m, c]$ 且 $f(x; \theta) = \theta_0 x + \theta_1$ 。

If we choose the squared Euclidean norm, then we have, for the simple $y = mx + c$ case : 如果我们选择平方欧几里得范数，那么对于简单的 ($y=mx+c$) 情况：

$$L(\theta) = |f(x; \theta) - y|$$

$$L(\theta) = |\theta_0 x + \theta_1 - y|^2 = (\theta_0 x + \theta_1 - y)^2,$$

which we can easily minimise, e.g. by gradient descent, since computing ($\nabla L(\theta)$) turns out to be easy. This is **ordinary linear least-squares**. 我们可以轻松地将其最小化，例如通过梯度下降，因为计算 $\nabla L(\theta)$ 结果很简单。这是**普通线性最小二乘法**。



Linear least squares tried to make the size of the squares nestled between the line and data points as small as possible 线性最小二乘法试图使线与数据点之间的正方形的大小尽可能小

In fact, we can find a closed form solution to this problem, without doing any iterative optimisation. This is because we have an **estimator** that gives us an estimate of the parameters of the line fit directly from observations. We can derive this, for example, by setting $\nabla L(\theta) = 0$ and solving directly (high-school

optimisation). 实际上，我们可以找到这个问题的闭式解，而不需要进行任何迭代优化。这是因为我们有一个估计器，它可以直接从观测数据中给出线性拟合参数的估计值。例如，我们可以通过设置 $\nabla L(\theta) = 0$ 并直接求解来推导出这一点。

通过最大似然估计进行线性回归

Linear regression via maximum likelihood estimation

我们也可以将这视为一个推断问题。我们可以明确假设我们正在观察来自其参数我们希望估计的分布的样本。这是一种最大似然方法。这要求我们能够以随机变量的分布来描述问题。We could also consider this to be a problem of inference. We could explicitly assume that we are observing samples from a distribution whose parameters we wish to estimate. This is a **maximum likelihood approach**. This requires that we can write down the problem in terms of the distribution of random variables.

如果我们假设“误差”是破坏完美的 $y = mx + c$ 关系的正态分布值，我们可能有一个模型 $Y \sim \mathcal{N}(mx + c, \sigma^2)$; Y 有均值 $mx + c$ 和某个标准偏差 σ . If we assume that "errors" are normally distributed values which are corrupting a perfect $y = mx + c$ relationship, we might have a model $Y \sim \mathcal{N}(mx + c, \sigma^2)$; Y has mean $mx + c$ and some standard deviation σ .

我们可以将其写为一个最大似然问题 (MLE)，在该问题中我们最大化 $\mathbb{P}(\theta|x_1, y_1, x_2, y_2, \dots, x_n, y_n)$ 。为了避免下溢，我们使用似然的对数并最小化负对数似然。独立样本 x_i 的对数似然由以下给出：We can write this as a maximum likelihood problem (MLE), where we maximise $\mathbb{P}(\theta|x_1, y_1, x_2, y_2, \dots, x_n, y_n)$. To avoid underflow, we work with the log of the likelihood and minimise the negative log-likelihood. The log-likelihood of independent samples x_i is given by:

$$\log \mathbb{P}(\theta|x_1, y_1, \dots, x_n, y_n) = \log \prod_i f_Y(x_i, y_i) = \sum_i \log f_Y(x_i, y_i),$$

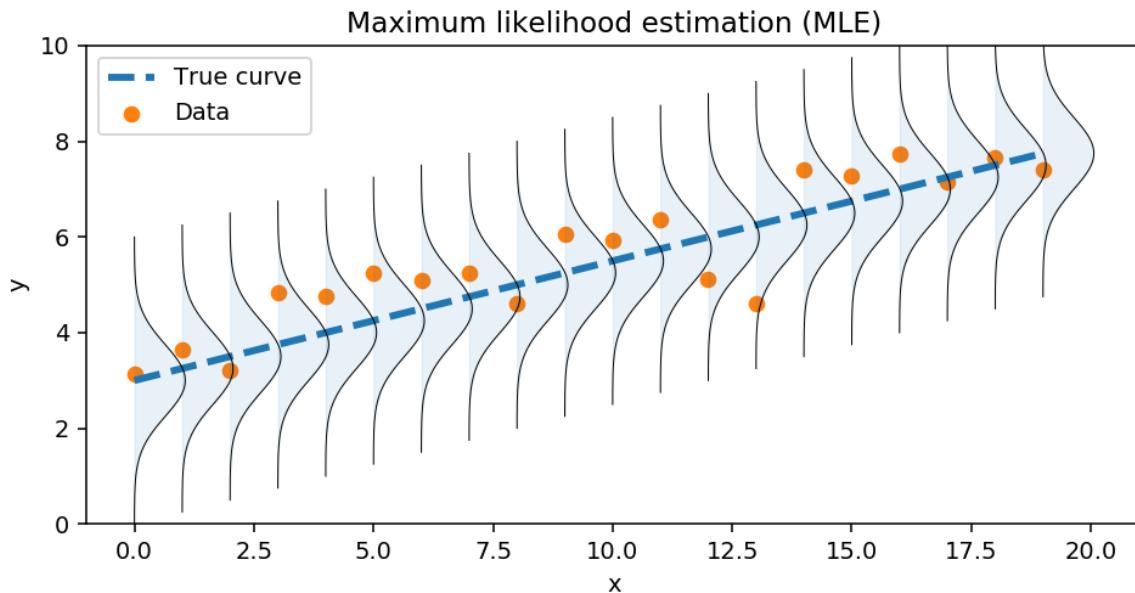
$$f_Y(x_i, y_i) = \frac{1}{Z} e^{-\frac{(y_i - \mu)^2}{2\sigma^2}}, \quad \mu = mx_i + c$$

我们可以最小化负对数似然来找到 $\theta = [m, c, \sigma]$ 的“最有可能”的设置，如果我们愿意在 LaTeX 中写出长方程式，我们可以将其写为一个目标函数：We can then minimise the negative log-likelihood to find the "most likely" setting for $\theta = [m, c, \sigma]$, which (if we feel like writing out long equations in LaTeX), we could write as an objective function:

$$L(\theta) = - \sum_i \log \left[\frac{1}{Z} e^{-\frac{(y_i - \theta_0 x_i - \theta_1)^2}{2\sigma^2}} \right],$$

在我们有线性回归的正态分布噪声的情况下，这实际上与直接优化线性最小二乘法完全相同，尽管我们除了找到 m 和 c 之外，还会找到误差的标准偏差 σ 。这是**最大似然线性回归**。In the case where we have normally distributed noise for linear regression, this turns out to be exactly equivalent to the direct optimisation with linear least-squares, although we will also find the standard deviation of the error σ in addition to m and c . This is **maximum likelihood linear regression**.

[注意：你绝对不需要记住这些方程或能够推导出它们。然而，你应该理解它们背后的逻辑。] [Note: you definitely do not need to remember these equations or be able to derive them. You should understand the logic behind them, however].



最大似然估计试图找到一条直线的参数，使得观察结果看起来可能 Maximum likelihood estimation tried to find parameters of a line that made the observations likely

贝叶斯线性回归

Bayesian linear regression

如果我们想知道我们对 m 和 c (以及 σ) 的估计有多确定怎么办? MLE 会告诉我们最可能的设置, 但它不会告诉我们与数据兼容的可能设置。What if we wanted to know how sure our estimates of m and c (and σ) were? MLE will tell us the most likely setting, but it won't tell us the possible settings that are compatible with the data.

贝叶斯方法是让参数本身成为随机变量。我们不想优化。我们不想找到最有可能的参数。我们反而想推导出关于参数的信念作为概率分布。这就是贝叶斯人做的; 他们用概率来表示信念。The Bayesian approach is to let the parameters themselves by random variables. We don't want to optimise. We don't want to find the most likely parameters. We instead want to derive a belief about the parameters as a probability distribution. This is what Bayesians do; they represent belief with probability.

因此我们可以将 $\theta = [m, c, \sigma]$ 写成一个随机变量, 并尝试推断出它的分布。我们可以使用贝叶斯规则来做到这一点。以这种形式写 (D=数据, H=假设; 假设的参数设置) : So we can write $\theta = [m, c, \sigma]$ as a random variable, and try and infer the distribution over it. We can do this using Bayes' rule. Writing in the form (D=data, H=hypothesis; hypothesised parameter settings):

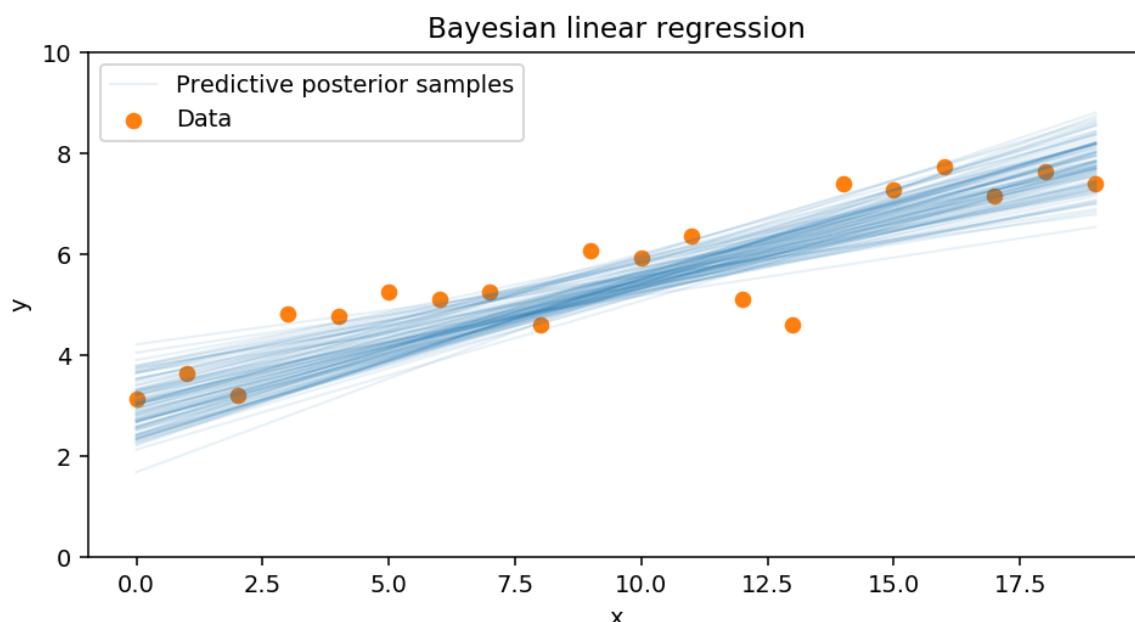
$$P(H|D) = \frac{P(D|H)P(H)}{P(D)}$$

假设我们的假设 H 是由 θ 参数化的, 那么我们想要知道 $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$, 其中 D 代表数据 $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ 。在线性回归中, θ 可以被看作是数据是由 θ 指定的参数的一条线生成的假设。Assuming our hypotheses H are parameterised by θ , then we want to know $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$, where D stands for the data $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$. In linear regression θ can be seen as the hypothesis that the data was generated by a line with parameters specified by θ .

我们需要：

- 关于参数的先验 $P(\theta)$ 。在线性回归的情况下，关于可能的梯度 m 、偏移 c 和噪声水平 σ 的初始信念。
- 计算似然 $P(D|\theta)$ 的方法。
- 使用贝叶斯规则结合这些的方法。通常这是不可能精确计算的（特别是 $P(D)$ 项通常是难以处理的），但我们可以使用马尔可夫链蒙特卡洛等方法从中抽样。We need:
- a **prior** over the parameters $P(\theta)$. An initial belief about the possible gradient m , offset c and noise level σ , in the linear regression case.
- a way of calculating the **likelihood** $P(D|\theta)$.
- a way of combining these using Bayes Rule. In general this is impossible to compute exactly (in particular the $P(D)$ term is often intractable), but we could sample from it using **Markov Chain Monte Carlo**, for example.

这将给我们从后验分布 $P(\theta|D)$ 中的样本，因此我们可以看到我们对神秘实体的参数的信念有多确定。This will give us samples from the posterior distribution of $P(\theta|D)$, so we can see how sure we should be about our beliefs about the parameters of the mysterious entity.



贝叶斯回归尝试在给定证据的情况下更新线参数的分布 Bayesian regression tries to update a distribution over line parameters given evidence