Functions and Data Fitting

August 31, 2022

1 Computations as Functions

Many computational tasks can be described by *functions*, that is, mappings from an input domain A to an output domain Y.

Examples:

- A **SPAM filter** for email needs to be able to tell if a certain piece of email is SPAM or not. At its core is a function f whose input is the email message and whose corresponding output is `true` (the message is SPAM) or `false` (not SPAM). The domain A of f is the set of all possible emails (a very large set!), and the codomain is the binary set

  $$Y = \{\texttt{true}, \texttt{false}\} ,$$

  so we write

  $$f \ : \ A \to Y \quad \text{and} \quad y = f(a) \in Y \quad \text{for } a \in A.$$

  Of course, it does not matter whether the values `true` and `false` are represented by strings, numerals (1 and 0, or perhaps 1 and -1), or something else.
- A game console uses a camera or a depth sensor such as a Kinect device to track the motions of someone playing a game of **virtual tennis**. At the core of this tracker is a function f that takes one video frame a from the camera or depth sensor and outputs a vector $\mathbf{y}$ of real numbers that somehow describe the configuration of the player's body. For instance, the first two numbers y_1, y_2 in $\mathbf{y}$ could be the values in degrees of two angles that specify the position of the player's left upper arm relative to her left shoulder. Other numbers in $\mathbf{y}$ specify other angles of the player's skeleton. For this application, A is the set of all possible video frames, and

  $$Y \subseteq \mathbb{R}^e ,$$

  a subset of all real-valued vectors with e components.
- A system for **medical diagnosis** is based on a function f that takes the description a of a patient's symptoms and returns the most likely ailment $y = f(a)$ out of a set Y of possible diseases.
- A **speech recognition** system is built around a function f that takes a snippet a of digitized audio samples and returns a word $y = f(a)$ out of a dictionary Y.
- A **movie recommendation** system relies on a function f that takes a list a of movies a certain person has seen in the past and returns a recommendation $y = f(a)$ that that person is likely to enjoy watching.

The virtual tennis task is one of *regression* because Y is a subset of real-valued vectors. All the other problems are *classification* tasks, because Y is a *categorical* set, that is, a finite set of values whose ordering does not matter.

Given one of these tasks, there are many ways in which a team of experts, mathematicians, and computer scientists can go about designing the key function f. Traditional methods *hand-craft* the function: Domain experts describe what is important for the task, and define a set of quantities that need to be computed from the input a. Mathematicians come up with some formulas for these quantities, and computer scientists write algorithms that compute numerical values based on these formulas. Each of these aspects depends strongly on what the task is, and it is difficult to say anything interesting about them abstractly.

More recently, *Machine Learning* (ML) has emerged as an alternative approach to hand design. In ML, one provides a large number of examples of input-output pairs $(a_1, y_1), \ldots, (a_N, y_N)$, and a class $\mathcal{F}$ of allowed functions. A predefined algorithm then computes $f \in \mathcal{F}$ so that

$$y_n \approx f(a_n) \quad \text{for} \quad n = 1, \ldots, N \; .$$

By itself, this is a *data fitting* problem, and you have seen several examples in your studies. For instance, if $A = \mathbb{R}$ and $Y = \mathbb{R}$, then $\mathcal{F}$ may be the set of all polynomials, and then one computes the coefficients of a polynomial that satisfies the approximate equalities above.

Machine learning is harder than data fitting: We want f to do well not only on the provided examples, but also on new inputs a. For the example with polynomials, we would like

$$y \approx f(a)$$

for any $a \in A$ that we are likely to encounter in the future. Even formalizing exactly what this means is a challenge, and we will spend quite a bit of time in this course doing so.

As you can imagine, things become trickier, even just for data fitting, as A becomes more complex: How do you fit a function to a set of examples when the domain is the set of all possible email messages? To insulate the complexities of the domain A from the problem of learning f, machine learning introduces the notion of a *feature vector* $\mathbf{x}$. The input a is not fed directly to f. Instead, some other function ϕ transforms a into a vector $\mathbf{x} \in X \subseteq \mathbb{R}^d$ with a pre-specified number d of components. The machine learning algorithm then takes a *training set*

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$$

(where the outputs y_n may be scalars or vectors) and computes a function h out of a predefined set $\mathcal{H}$ called the *hypothesis space* such that

$$y_n \approx h(\mathbf{x}_n) \quad \text{for} \quad n = 1, \ldots, N \; . \tag{1}$$

In this way,

$$f(a) = h(\phi(a)) \; ,$$

and the ML algorithm only sees vectors $\mathbf{x}$ of real numbers, rather than complicated objects such as a.

For instance, in the case of SPAM filtering, one could number all the words in the English dictionary from 1 to d (perhaps $d = 20{,}000$). Given an email message a, the corresponding feature $\mathbf{x}$ could be a vector with d components, with component k specifying how many times word number k in the dictionary occurs in a.[1]

Even if a itself is already a vector of numbers, like in the example of the body tracker above, there are still reasons for transforming a to a different vector $\mathbf{x}$. For instance, a may be unwieldily large, and one wants to somehow compress the information it contains into a more parsimonious representation.

[1]This would be a very sparse vector (that is, it would have very many zeros), because most English words do *not* come up in any one email message.

The introduction of features adds a burden for the designer (what exactly does ϕ do?), but allows machine learning to be agnostic of the specific structure of the domain A for the task at hand. The domain X of h is always a subset of $\mathbb{R}^d$, regardless of the application domain.[2]

The error in the approximation 1 is measured by a *loss*

$$\ell(y_n, h(\mathbf{x}_n)) \quad \text{where the } \textit{loss function } \ell \text{ has signature} \quad Y \times Y \to \mathbb{R}^+ \; .$$

This function depends on the application, and somehow measures how much we pay for a discrepancy between the true value y_n of the function output and the value $h(\mathbf{x}_n)$ returned by h.

We will see various definitions for ℓ in this course. As a preliminary example, if Y is binary (that is, if it contains two values), then a frequently used loss function is the *zero-one loss*,

$$\ell(y, \hat{y}) = \ell_{0\text{-}1}(y, \hat{y}) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

This loss assigns a unit penalty to a classification mistake ($y \neq \hat{y}$) and a zero penalty to a correct classification ($y = \hat{y}$).

To summarize, we consider the following two problems:

Assume that a training set

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R}^e$$

and a hypothesis space $\mathcal{H}$ of functions from $\mathbb{R}^d$ to $\mathbb{R}^e$ are given.

- *Data fitting* selects a function $h \in \mathcal{H}$ that minimizes the average loss $\ell(y_n, h(\mathbf{x}_n))$ over the examples $(\mathbf{x}_n, y_n)$ in the training set T.
- *Machine learning* selects a function $h \in \mathcal{H}$ that minimizes the average loss $\ell(y, h(\mathbf{x}))$ over previously unseen pairs $(\mathbf{x}, y)$.

The average loss

$$L_T(h) \overset{\text{def}}{=} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, h(\mathbf{x}_n))$$

of hypothesis h over the training set T is often called the *empirical risk*. The term "risk" comes from actuarial applications, in which average loss relates to the risk that an insurance company undertakes when it underwrites some item. The attribute "empirical" emphasizes that the average is computed numerically from data.

Before we give a more precise meaning to the notion of minimizing an average loss "over previously unseen data," the next Section examines data fitting in the familiar context of polynomial approximation.

[2]Some approaches to machine learning consider the features $\mathbf{x}$ as given, both at training time and when the learned system is deployed. Other approaches consider the design of ϕ as part of the problem. More about this point later in the course.

2 Polynomial Data Fitting

Data fitting is a good warmup problem: It is not machine learning, as discussed earlier, but it shares many characteristics with it. Examining how machine learning differs from data fitting will also highlight the key challenges of the former. In addition, if the hypothesis space $\mathcal{H}$ is the set of all polynomials and a quadratic loss (defined in Section 2.1 below) is used to measure the fitting error, the resulting data fitting problem is easy to solve: The coefficients of a polynomial h are the unknowns of the problem, and they appear linearly in h. Since the loss is quadratic, data fitting becomes a quadratic optimization problem, and the optimal coefficients can be found by solving a linear system of equations.

For simplicity, we will review these concepts first for polynomials in a single variable,

$$h \; : \; \mathbb{R} \rightarrow \mathbb{R} \; .$$

If the codomain is $\mathbb{R}^e$ for some $e > 1$, we can view h as a collection of e polynomials, and treat each polynomial separately, so there is nothing new there. If the domain is $\mathbb{R}^d$ for some $d > 1$, the modifications are straightforward in principle. We will examine those separately, after we build some intuition on the case $d = e = 1$.

2.1 Univariate Polynomials

A (real-valued) polynomial of degree k in the real-valued variable x is a linear combination of powers of x, up to and including power k:

$$h(x) = c_0 + c_1 x + \ldots + c_k x^k \quad \text{with} \quad c_i \in \mathbb{R} \quad \text{for} \quad i = 0, \ldots, k \quad \text{and} \quad c_k \neq 0 \; .$$

Given a training set

$$T = \{(x_1, y_1), \ldots, (x_N, y_N)\} \subset \mathbb{R} \times \mathbb{R} \; ,$$

the *empirical risk on T* is the average

$$L_T(h) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, h(x_n)) \tag{2}$$

where the *loss*

$$\ell(y, y') = (y - y')^2 \tag{3}$$

is a quadratic function of its two arguments.

Since the second argument of ℓ is given value $h(x_n)$ in the n-th term of the summation in equation 2, and since h is a linear function of its coefficients c_i, the empirical risk is a quadratic function of these coefficients. As a consequence, an optimal polynomial $\hat{h}$ of degree *up to* a given value k can be found by solving a system of linear equations. Specifically, the system is

$$A\mathbf{c} = \mathbf{b} \tag{4}$$

where the vector $\mathbf{c}$ gathers the unknown coefficients of the polynomial

$$\mathbf{c} = \begin{bmatrix} c_0 \\ \vdots \\ c_k \end{bmatrix}$$

and where the terms

$$A = \begin{bmatrix} 1 & x_1 & \dots & x_1^k \\ \vdots & \vdots & & \vdots \\ 1 & x_N & \dots & x_N^k \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

are computed from the training set. The matrix A is $N \times (k+1)$ and the vector $\mathbf{b}$ has N entries.

Beware: The reason why this system is linear is that the loss is quadratic and the coefficients appear linearly in a polynomial. Do not confuse this fact with the fact that polynomials are nonlinear functions of x: h is linear in c_i and nonlinear in x. In other words, fitting a nonlinear function of x can still be a linear problem in $\mathbf{c}$.

Also, while x and y are common names for *unknown* quantities in a problem, the values x_n and y_n are *known* when the problem is to find the coefficients c_i and the training set T is given. Thus, while x and y are natural names for the independent variable and the value of $y = h(x)$, the unknowns in the polynomial data-fitting problem are the entries c_i of $\mathbf{c}$, while the quantities x_n and y_n are known.

From the theory of linear algebra, we know that if $\mathbf{b}$ is in the range of A, there are one (if A is full rank) or infinitely many (if A is rank-deficient) solutions with zero loss (in other words, the approximation is perfect). If $\mathbf{b}$ is not in the range of A, then the system 4 admits *no* solution. However, in that case, the problem of solving system 4 is reinterpreted as the following minimization problem, called the *least-squares solution* to system 4:

$$\hat{\mathbf{c}} \in \arg\min_{\mathbf{c}} \|A\mathbf{c} - \mathbf{b}\|^2 . \tag{5}$$

Notation: The solution $\hat{\mathbf{c}}$ is *a member of* ($\in$) the $\arg\min$, rather than equal to it. This means that while there is a unique minimum *value* $\min_{\mathbf{c}} \|A\mathbf{c} - \mathbf{b}\|^2$, there are possibly (infinitely) many vectors $\mathbf{c}$ that achieve that minimum value. Therefore, $\arg\min_{\mathbf{c}} \|A\mathbf{c} - \mathbf{b}\|^2$ is generally a *set* of vectors, rather than a single vector, and we are typically content with any element $\hat{\mathbf{c}}$ of that set. Thus, the solution to a data fitting problem is not necessarily unique, even with a quadratic loss.

Since the squared norm of the difference between right-hand and left-hand side in row i of system 4 is equal to $\ell(y_i - h(x_i))$ (check this!), we have that

$$L_T(h) \;=\; \|A\mathbf{c} - \mathbf{b}\|^2$$

and therefore the least-squares solution to system 4, defined by expression 5, minimizes the empirical risk in definition 2 with the quadratic loss defined in equation 3.

The solution polynomial

$$\hat{h}(x) = \hat{c}_0 + \hat{c}_1 x + \ldots + \hat{c}_k x^k$$

has degree *up to* k, since $\hat{c}_k$ may happen to be zero.

Unknown Maximum Degree If the maximum degree k is not given, we need some way to determine it. In data fitting, we are only required to do well on the training set, so there is always the option of setting

$$k = N - 1 .$$

With this choice, the system above has N equations in N unknowns, so there is always at least one exact (that is, zero-residual) solution $\hat{\mathbf{c}}$ (and infinitely many if the equations are linearly dependent).

In other words, by making the degree of the polynomial high enough, we can achieve zero loss: Data fitting has become *interpolation* and the plot of $\hat{h}(x)$ goes exactly through all of the data points:

$$y_n = \hat{h}(x_n) \; .$$

More often, especially when the number N of data points is very large, we look for polynomials of lower degree. For now, the question of what degree to choose is purely subjective, as the following example illustrates. The theory of machine learning will make this and related concepts more precise.

Example:

- In Figure 1, the same ten red points (the training set T) are fit with polynomials of degree $k = 1$, 3, and 9. While a low degree ($k = 1$) gives a poor fit, the fit with $k = N - 1 = 9$ seems overkill: In order to go exactly through the ten points, the function $\hat{h}$ oscillates up and down between them in ways that seem somewhat arbitrary.

  In contrast, the intermediate degree $k = 3$ strikes a more pleasing balance between errors on the training set and overall smoothness or simplicity of the polynomial. We say that the polynomial with $k = 1$ *underfits* and the one with $k = 9$ *overfits*. Again, these notions will be made more quantitative later in this course.

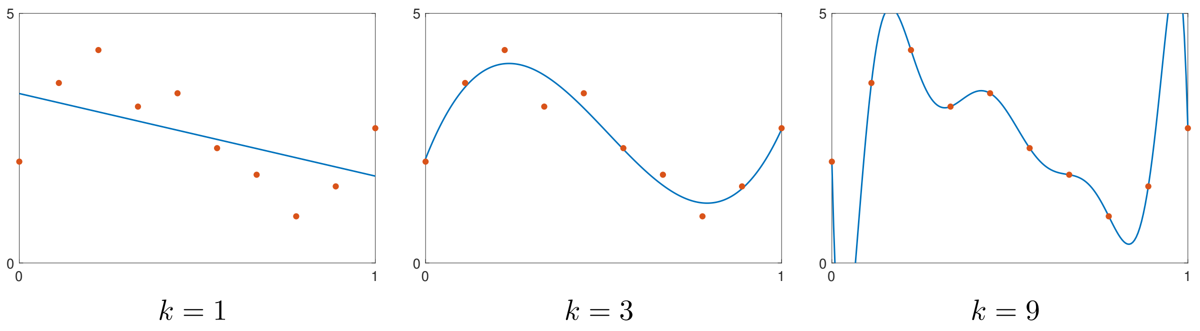

Figure 1: Ten data points are fitted with polynomials of varying degree k.

2.2 Multivariate Polynomials

In a multivariate polynomial, the independent variable is a vector $\mathbf{x}$ with d entries, rather than a single scalar. The theory of polynomial fitting is not very different for these polynomials and is outlined next.

A *monomial* of degree k' in the d real variables $x_1, \ldots, x_d$ is a product of powers of these variables:

$$x_1^{k_1} \ldots x_d^{k_d} \quad \text{with} \quad k_i \in \mathbb{N} \quad \text{and} \quad k_1 + \ldots + k_d = k' \; . \tag{6}$$

Keep in mind that the set $\mathbb{N}$ of natural numbers includes zero, so not all variables need to appear explicitly.

A *polynomial* of degree k is a linear combination of monomials of degrees up to and including k:

$$h(\mathbf{x}) = \sum_j c_j x_1^{k_1^{(j)}} \ldots x_d^{k_d^{(j)}} \quad \text{where} \quad \mathbf{x} = [x_1, \ldots, x_d]^T \quad \text{and} \quad c_j \in \mathbb{R} \quad \text{for all } j,$$

with the condition that the coefficient of at least one monomial of degree k is nonzero.[3]

[3]Without this condition, all polynomials could be said to have an arbitrarily large degree.

Given a training set

$$T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\} \subset \mathbb{R}^d \times \mathbb{R} \;,$$

the empirical risk

$$L_T(h) \overset{\text{def}}{=} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, h(\mathbf{x}_n)) \quad \text{where} \quad \ell(y, y') = (y - y')^2$$

is still a quadratic function of the coefficients.[4] For instance, if $k = 2$, then

$$h(\mathbf{x}) = c_0 + c_1 x_1 + c_2 x_2 + c_3 x_1^2 + c_4 x_1 x_2 + c_5 x_2^2$$

is a linear function of the coefficients, so that L_T is quadratic. If we are looking for a polynomial of degree *up to k*, then the vector $\hat{\mathbf{c}}$ of the coefficients of $\hat{h}$ can be found by solving the system

$$A\mathbf{c} = \mathbf{b}$$

where

$$\mathbf{c} = \begin{bmatrix} c_0 \\ \vdots \\ c_5 \end{bmatrix}$$

is the unknown and the terms

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{11}^2 & x_{11}x_{12} & x_{12}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{N1} & x_{N2} & x_{N1}^2 & x_{N1}x_{N2} & x_{N2}^2 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

are assembled from the training set. In these expressions, x_{n1} and x_{n2} are the two components of $\mathbf{x}_n$ in this example with data dimensionality $d = 2$.

It would be straightforward to write code that fits a polynomial up to any degree to a set of points in any dimension d. As we see in the next section, this code will not help much in machine learning.

2.3 Limitations of Polynomial Fitting

In principle, polynomials up to any degree can be fitted to any number of points. As long as the degree is high enough, the risk can be made as small as desired, including zero. In addition, data fitting can also be applied to classification problems: Just associate a natural number to each element of the codomain Y and proceed as above. Since $\hat{h}(\mathbf{x})$ is not necessarily integer, round it to the nearest element in Y for a categorical answer.

So, why don't we just use polynomial data fitting all the time and call it quits? There are several answers to this question.

[4] Indeed, all that has changed in these two expressions is a bold $\mathbf{x}_n$ rather than an italic x_n.

Complexity The first answer is complexity: In many cases, the sample space $X \subseteq \mathbb{R}^d$ has a high dimensionality d. The SPAM filter example is a case in point, with d of the order of thousands or tens of thousands (one dimension or so for each word in the English dictionary). As d increases, the number of coefficients in a polynomial of degree k in d variables grows very fast.

Specifically, the argument in Appendix A shows that there are

$$m(d,k) = \binom{d+k}{k}$$

possible monomials of degree up to k in d variables. The Appendix also shows that if the degree k is kept fixed, this function is $O(d^k)$, and if the dimensionality d is kept fixed instead, it is $O(k^d)$. Also, if both d and k grow at the same rate, the function m is $O(4^d/\sqrt{d})$. Thus, the rate of growth of m is polynomial when either d or k is kept fixed, and exponential if they are both allowed to grow at the same rate.

Thus, polynomial fitting becomes unwieldy for large d except when $k = 1$ or $d = 1$. The case $k = 1$ involves fitting an affine function[5], and the number of coefficients to estimate is then $O(d)$ (and more precisely $d+1$), which is very manageable. The case $d = 1$, regardless of the value of k, is the univariate case considered in Section 2.1, in which the number of coefficients to estimate is $O(k)$ (in fact, we know it's $k+1$), also entirely manageable.

For higher-degree polynomials, the number of coefficients grows at least polynomially (with power k) with the number d of variables. As d grows, the number N of data samples needed to estimate the unknowns grows as well (for instance, we want N about as large as $m(d,k)$ for an exact fit), and we soon run out of data. In other words, the *sample complexity* of polynomial data fitting grows polynomially with the dimensionality d of the sample space X, and for all but the lowest degrees the cost of collecting and annotating data samples[6] is prohibitive.

Overfitting Figure 1 hinted at a second reason why polynomial fitting won't do: In order to reduce the loss we need to increase the degree, and as we do so the polynomial swings more wildly between training samples. This difficulty becomes important for machine learning, in contrast with data fitting: In machine learning, we want $\hat{h}(\mathbf{x})$ to be a good predictor of y even *at previously unseen data points*, that is, at values of $\mathbf{x}$ that were not part of the training set T. While it is still unclear what this means exactly, it should be intuitively clear that the swings of a polynomial are not beneficial: After all, we hope that the training set T is enough to tell what will happen at the new points, so a tamer fit (such as the one for $k = 3$ in the Figure) would seem to be a safer bet. This is because the oscillations come from the polynomial, that is from our choice of hypothesis space $\mathcal{H}$, rather than from the data. While a polynomial (of degree 3) happens to do well in the figure, it is not clear that, given some training set T, the degree of the polynomial can be made large enough to obtain a satisfactorily small risk, *without at the same time causing excessive overfitting.* In other words, polynomials are not necessarily a natural choice of functions for any given fitting (or machine learning) problem. In fact, polynomials are used very rarely as the hypothesis space of a machine learning problem. The lone exception is affine polynomials ($k = 1$), which are very commonly used.

[5]Affine means linear plus a constant.

[6]Annotating a data point $\mathbf{x}$ means to specify what the corresponding value y is. Typically, this annotation is done manually by a person. More on this point later.

The Curse of Dimensionality The two factors above, complexity and overfitting, are polynomial-specific manifestations of a more fundamental and general difficulty. For the training set T to be representative of all possible data in the sample space $X \subset \mathbb{R}^d$, we would like the data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ to "fill X nicely." For instance, in Figure 1, the red points are taken at regular intervals, and are relatively closely spaced. Not much can really happen between two consecutive points, as long as the underlying phenomenon that generates the data is sufficiently smooth. As the dimensionality of X increases, however, it becomes very difficult very soon to "fill X nicely." Even when X is the unit cube in $\mathbb{R}^d$, that is, $X = [0, 1]^d$, if we wanted to sample X with a grid with only 10 points in each dimension we would end up with 10^d grid points: *The number of grid points grows exponentially with* d if the number of grid points per dimension is fixed. Considering that the number of atoms in the universe is around 10^{80}, we see that grids become completely infeasible for all but the smallest values of d. Even with $d = 10$ we would already need tens of billions (10^{10}) of data points. This fundamental difficulty is called the *curse of dimensionality.* Avoiding the curse calls for new ideas.

Appendix

A Counting Monomials

There are

$$m(d,k) = \binom{d+k}{k}$$

possible monomials of degree up to k in d variables.

Proof. First, for any nonnegative integer $k' \leq k$ we can write the monomial in expression 6 as follows:

$$1^{k_0}\, x_1^{k_1} \ldots x_d^{k_d} \quad \text{where} \quad k_0 + \ldots + k_d = k$$

and where the powers k_i are nonnegative (but possibly zero) integers. This expression corresponds bijectively to the following string of $d+k$ bits:

$$\underbrace{0,\ldots,0}_{k_0},1,\underbrace{0,\ldots,0}_{k_1},1,\ldots,1,\underbrace{0,\ldots,0}_{k_d}\ .$$

This string is constructed by writing a block of k_i consecutive zeros for each k_i with $i = 0,\ldots,d$, and then separating adjacent blocks with a single 1.

"Corresponds bijectively" in the sentence above means that there is exactly one such string for each such monomial, and counting these strings is easier than counting monomials directly. This is because there is one such string for each way of selecting the d positions for the d ones among all $k+d$ positions. This immediately yields the desired expression for $m(d,k)$. Δ

Rates of Growth How fast does the number $m(d,k)$ of coefficients grow with the degree k or the number of variables d? We can write

$$m(d,k) = \binom{d+k}{k} = \frac{(d+k)!}{d!k!} = \frac{(d+k)\,(d+k-1)\,\ldots\,(d+1)}{k!}\ .$$

Consider first keeping k fixed and varying d. Then, the denominator of the last expression above is a constant, and the numerator is the product of k factors, each of order d. Thus, if k is kept fixed then $m(d,k)$ is $O(d^k)$. Since $\binom{d+k}{k} = \binom{k+d}{d}$, we have that $m(d,k) = m(k,d)$, and therefore, by symmetry, if d is kept fixed then $m(d,k)$ is $O(k^d)$. In either case we have polynomial growth, and the growth rate is high if the fixed parameter is large.

What if both d and k grow at the same rate? Using the Stirling approximation to the factorial

$$q! \approx \sqrt{2\pi q}\left(\frac{q}{e}\right)^q \quad \text{as} \quad q \to \infty$$

in the expression

$$m(d,k) = \binom{d+k}{k} = \frac{(d+k)!}{d!\,k!}$$

with $k = d$ easily yields that $m(d,d)$ is $O(4^d/\sqrt{d})$, an exponential rate of growth.