MACHINE LEARNING AND FRACTAL GEOMETRY

PETER BLOEM

MACHINE LEARNING
AND
FRACTAL GEOMETRY

PETER BLOEM

March 30, 2010

Master's thesis in Artificial Intelligence
at the University of Amsterdam

supervisor

Prof. Dr. Pieter Adriaans

committee

Dr. Jan-Mark Geusebroek
Prof. Dr. Ir. Remko Scha
Dr. Maarten van Someren

CONTENTS

SUMMARY

The main aim of this thesis is to provide evidence for two claims. First, there are domains in machine learning that have an inherent fractal structure. Second, most commonly used machine learning algorithms do not exploit this structure. In addition to investigating these two claims, we will investigate options for new algorithms that are able to exploit such fractal structure.

The first claim suggests that in various learning tasks the input from which we wish to learn, the dataset, contains fractal characteristics. Broadly speaking, there is detail at all scales. At any level of 'zooming in' the data reveals a non-smooth structure. This lack of smoothness at all scales can be seen in nature in phenomena like clouds, coastlines, mountain ranges and the crests of waves.

If this detail at all scales is to be exploited in any way, the object under study must also be *self-similar*, the large-scale features must in some way mirror the small-scale features, if only statistically. And indeed, in most natural fractals, this is the case. The shape of a limestone fragment will be closely related to the ridges of the mountainside where it broke off originally, which in turn will bear resemblance to the shape of the mountain range as a whole.

Finding natural fractals is not difficult. Very few natural objects are at all smooth, and the human eye has no problem recognizing them as fractals. In the case of datasets used in machine learning, finding fractal structure is not as easy. Often these datasets are modeled on a Euclidean space of dimension greater than three, and some of them are not Euclidean at all, leaving us without our natural geometric intuition. The fractal structure may be there, but there is no simple way to visualize the dataset as a whole. We will analyze various datasets to investigate their possible fractal structure.

Our second claim is that when this fractal structure and self-similarity exists, most commonly used machine learning algorithms cannot exploit it. The geometric objects that popular algorithms use to represent their hypotheses are always Euclidean in nature. That is, they are non-fractal. This means that however well they represent the data at a narrow range of scales, they cannot do so at all scales, giving them an inherent limitation on how well they can model any fractal dataset.

While the scope of the project does not allow a complete and rigorous investigation of these claims, we can provide some initial research into this relatively unexplored area of machine learning.

CHAPTER 1 · INTRODUCTION

This chapter introduces the two basic concepts on which this thesis is built: fractals and machine learning. It provides some reasons for attempting to combine the two, and highlights what pitfalls may be expected.

The history of fractals begins against the backdrop of the Belle Époque, the decades following the end of the Franco-Prussian war in 1871. Socially and politically, a time of peace and prosperity. The nations of Europe were held in a stable balance of power by political and military treaties, while their citizens traveled freely. The academic world flourished under this relative peace. Maxwell and Faraday's unification of magnetism and electricity had left a sense in the world of physics that the basic principles of the universe were close to being understood fully. Cantor's discoveries in set theory sparked the hope that all of mathematics could be brought to a single foundation. Fields such as history, psychoanalysis and the social sciences were all founded in the final decades of the 19th century.

As tensions rose in Europe, so they did in the academic world. Einstein's 1905 papers suggested that physics had far from reached its final destination, offering the possibility that space and time were not Euclidean. Paradoxes in the initial foundations of set theory fueled a crisis of faith in the foundations of mathematics. In the postwar period, these tensions culminated in the development of quantum mechanics and general relativity, disconnecting physics forever from human intuition. The mathematical world, having resolved its paradoxes, was struck by the twin incompleteness theorems of Gödel, and forced to abandon the ideal of a complete and consistent mathematics.

Considering all these revolutions and paradigm shifts, it is understandable that the development of fractal geometry, which also started during this era, was somewhat overshadowed, and its significance wasn't fully understood until more than half a century later. Nevertheless, it too challenged fundamental principles, the cornerstones of Euclidean geometry. Now, one hundred years later, it is difficult to find a single scientific field that has not benefited from the development of fractal geometry.

The first fractals were described more than 50 years before the name fractal

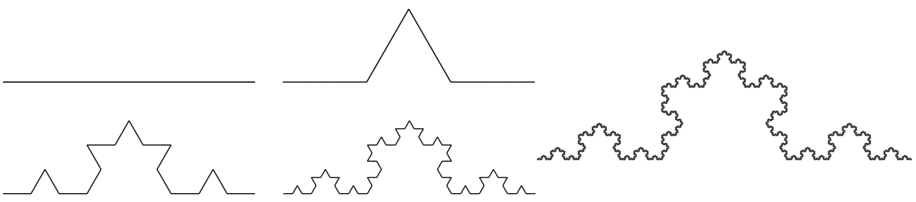


FIGURE 1.1: The Koch curve. The initial image, a line segment, is replaced by four line segments, creating a triangular extrusion. The process is repeated for each line segment and continuously iterated.

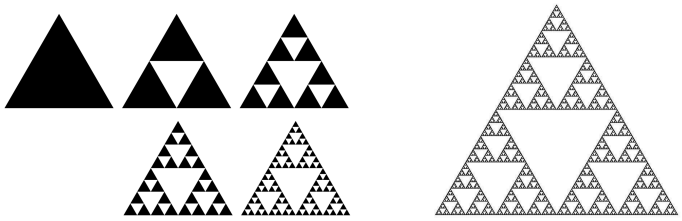


FIGURE 1.2: The Sierpinski triangle.

itself was coined. They were constructed as simple counterexamples to commonly held notions in set theory, analysis and geometry. The Koch curve shown in figure 1.1, for instance, was described in 1904 as an example of a set that is continuous, yet nowhere differentiable. Simply put, it is a curve without any gaps or sudden 'jumps', yet at no point does it have a single tangent, a unique line that touches it only at that point. For most of the nineteenth century it was believed that such a function could not exist.

The Koch curve is constructed by starting with a single line segment of length 1, cutting out the middle third and placing two line segments of length one third above it a in a triangular shape. This leaves us with a curve containing four line segments, to each of which we can apply the same operations, leaving us with 16 line segments. Applying the operation to each of these creates a curve of 64 segments and so on. The Koch curve is defined as the shape that this iterative process tends to, the figure we would get if we could continue the process indefinitely, the process' *limit set*. The construction in figure 1.1 shows the initial stages as collections of sharp corners and short lines. It is these sharp corners that have no unique tangent. As the process continues, the lines get shorter and the number of sharp points increases. In the limit, every point of the Koch curve is a corner, which means that no point of the Koch curve has a unique tangent, even though it is continuous everywhere.

Another famous fractal is the Sierpinski triangle, shown in figure 1.2. We start with a simple equilateral triangle. From its center we can cut a smaller up-side-down triangle, so that we are left with three smaller triangles whose corners are just touching. We can apply the same operation to each of these three triangles, leaving us with nine even smaller triangles. Apply the operation again and we are left with 27 triangles and so on. Again, we define the Sierpinski triangle to be the limit set of this process. As an example of the extraordinary properties of fractals, consider that the number of triangles in the figure after applying the transformation n times is 3^n. If the outline of the original triangle has length 1, then after the first operation the triangles each have an outline of length $\frac{1}{2}$ (since each side is scaled by one half), and after n steps each triangle has an outline of length $\frac{1}{2^n}$. As for the surface area, it's easy to see that each operation decreases the surface area by $\frac{1}{4}$, leaving $(\frac{3}{4})^n$ after n iterations. If we consider what happens to these formulas as the number of operations grows to infinity, we can see that the Sierpinski set contains an infinite number of infinitely small triangles, the sum of their outlines is infinite, yet the sum of their surface areas is zero.

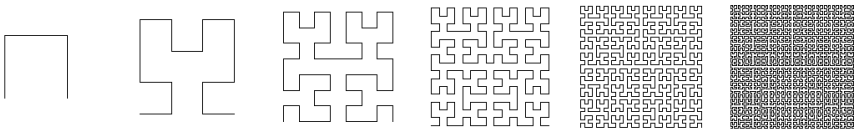


FIGURE 1.3: The Hilbert curve. See E.2 for attribution.

For another unusual property of these fractal figures we turn our attention to the Hilbert curve, shown in figure 1.3. The curve starts as a 'bucket' shape made up of three line segments. Each line segment is then replaced by a smaller bucket, which is rotated at right angles according to simple rules, and the buckets are connected. The process is repeated again, and a complicated, uninterrupted curve emerges. We can see from the first few iterations that the curve progressively fills more and more of the square. It can be shown that in the limit, this curve actually fills all of the square. That means that for every point in the square, there is a point on the Hilbert curve, and the total set of points on the Hilbert curve is indistinguishable from the total set of points inside the square. This means that while intuitively, we might think of the Hilbert curve as one dimensional—it is after all, a collection of line segments—in fact it is precisely equal to a two dimensional figure, which means that it must be two dimensional as well.

In the early years of the 20^{th} century, many figures like these were described. They all have three properties in common. First, they were all defined as the limit set of some iterative process. Second, they are all *self similar*, a small part of the figure is exactly equal to a scaled down version of the whole. And finally, they all showed groundbreaking properties, sometimes challenging notions that had been held as true since the beginnings of geometry. Despite the far reaching consequences, they were generally treated as pathological phenomena. Contrived, atypical objects that had no real impact other than to show that certain axioms and conjectures were in need of refinement. Ultimately, they were to be got rid of rather than embraced.

It was only after the Second World War that fractals began to emerge as a family of sets, rather than a few isolated, atypical counterexamples. And more importantly, as a family that can be very important. This development was primarily due to the work of Polish-born mathematician Benoît Mandelbrot. The development of Mandelbrot's ideas is best exemplified by his seminal 1967 paper, *How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension* (Mandelbrot, 1967). In this paper, Mandelbrot built on the work of polymath Lewis Fry Richardson. Richardson was attempting to determine whether the length of the border between two countries bears any relation to the probability of the two countries going to war. When he consulted various encyclopedias to get the required data, he found very different values for the lengths of certain borders. The length of the border between Spain and Portugal, for instance, was variously reported as 987 km and 1214 km. Investigating further, Richardson found that the length of the imaginary ruler used to measure the length of a border or coastline influences the results. This is to be expected when measuring the length of any curve, but when we measure say, the outline of a circle with increasingly small rulers, the resulting measurements will converge to the

true value. As it turns out, this doesn't hold for coastlines and borders. Every time we decrease the size of the ruler, there is more detail in the structure of the coastline, so that ultimately the sum of our measurement keeps growing as our measurements get more precise.

Richardson's paper was largely ignored, but Mandelbrot saw a clear connection with his own developing ideas. He related the results of Richardson to the early fractals and their unusual properties. As with the Hilbert curve we are tempted to think of coastlines as one-dimensional. The fact that they have no well-defined length is a result of this incorrect assumption. Essentially, the set of points crossed by a coastline fills more of the plane than a one-dimensional curve can, but less than a two-dimensional surface. Its dimension lies between one and two. In these days the dimension of the length of the British coast is determined to be about 1.25.[1]

As Mandelbrot developed these ideas, it became clear that structures with non-integer dimensions can be found everywhere in nature. For thousands of years, scientists had used Euclidean shapes with integer dimensions as approximations of natural phenomena. Lines, squares, circles, balls and cubes all provided straightforward methods to model the natural world. The inaccuracies resulting from these assumptions were seen as simple noise, that could be avoided by creating more complicated combinations of Euclidean shapes.

Imagine, for instance, a structural analysis of a house. Initially, we might represent the house as a solid cube. To make the analysis more accurate, we might change the shape to a rectangular box, and add a triangular prism for the roof. For even greater accuracy, we can replace the faces of this figure by flat boxes to represent the walls. The more simple figures we add and subtract, the more accurate our description becomes. Finally, we end up with a complicated, but Euclidean shape. If we follow the same process with natural elements, we notice a familiar pattern. To model a cloud, we may start with a sphere. To increase accuracy we can add spheres for the largest billows. Each of these will have smaller billows emerging from it, and they have smaller billows emerging from them, and so on. To fully model the cloud, we need to repeat the process indefinitely. As we do so, the structure of the cloud changes from Euclidean to fractal.

The array of fractal phenomena found in nature is endless. Clouds, coastlines, mountain ranges, ripples on the surface of a pond. On the other hand, the closer we look, the more difficult it becomes to find natural phenomena that are truly without fractal structure. Even our strictly Euclidean house becomes decidedly fractal, when we want to model the rough surface of the bricks and mortar it is made of, the swirls in the smoke coming from the chimney, the folded grains in the wood paneling, even the minute imperfections on the surface of the windows.

1.0.1 **Fractals**

This section provides a quick overview of some of the fractals and fractal families that have been described over the past century.

[1] From measurements by Richardson, although he didn't interpret the value as a dimension.

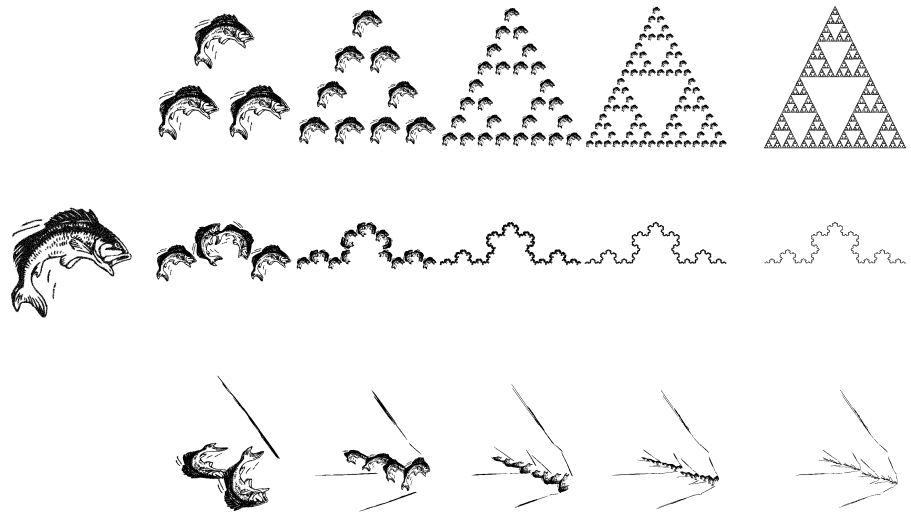


FIGURE 1.4: Three fractals defined by transformations of an arbitrary initial image. The Sierpinski triangle, the Koch curve and a fractal defined by three randomly chosen affine transformations.

Iterated function systems

One of the most useful families of fractals are the iterated function systems. This family encompasses all sets that are exactly self similar. That is, all sets that are fully described by a combination of transformed versions of itself. The Sierpinski triangle, for instance, can be described as three scaled down copies of itself. This approach leads to a different construction from the one we have seen so far. We start with any initial image (a picture of a fish, for instance) and arrange three scaled down copies of it in the way that the Sierpinski triangle is self similar. This gives us a new image, to which we can apply the same operation, and so on. The limit set of this iteration is the Sierpinski triangle. The interesting thing here is that the shape of the initial image doesn't matter. The information contained in the initial image disappears as the iteration goes to infinity, as shown in figure 1.4

Any set of transformations that each reduce the size of the initial image defines its own limit set (although a different set of transformations may have the same limit set). Most of the classic fractals of the early 20^{th} century can be described as iterated function systems.

There is another way to generate the limit set of a set of transformations. Start with a random point in the plane. Choose one of the operations at random, apply it to the point, and repeat the process. Ignore the first 50 or so points generated in this way, and remember the rest. In the limit, the sequence of points generated this way will visit all the points of the limit set. This method is known as the *chaos game*.

Figure 1.5 shows some fractals generated in this way.

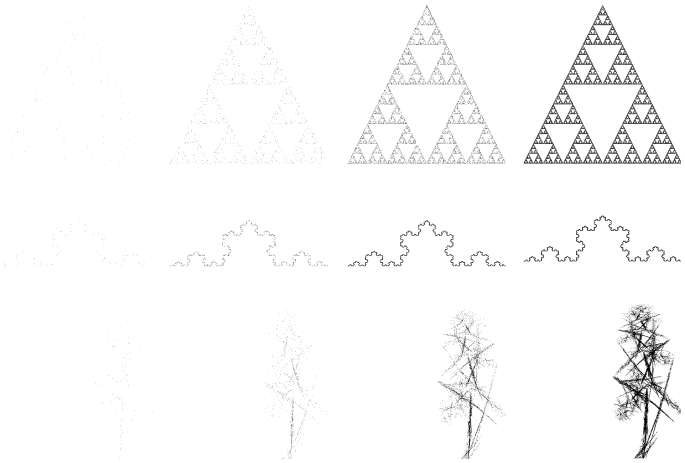


FIGURE 1.5: Three fractals generated using the chaos game. The Sierpinski triangle, the Koch curve and a fractal defined by five randomly chosen affine maps.

Strange attractors and basins of attraction

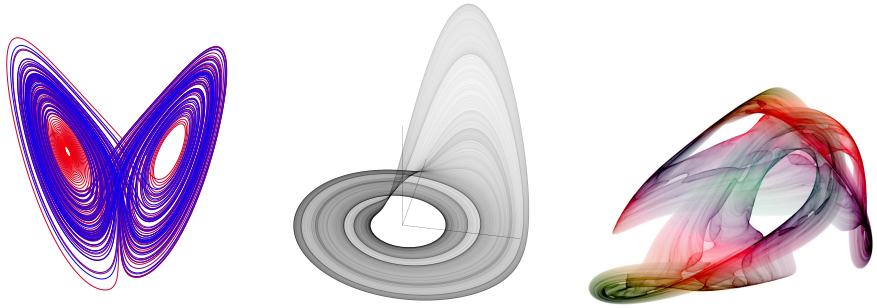
The development of the study of non-linear dynamical systems—popularly known as chaos theory—has gone hand in hand with that of fractal geometry. The theory of dynamical systems deals with physical systems whose state can be fully described by a given set of values. Consider, for example, a swinging pendulum. Every point in its trajectory is fully defined by its position and its velocity, just two numbers. From a given pair of position/velocity values, the rest of its trajectory can be calculated completely [2] We call the set of values that describes the system its *state*. The set of all possible states is called the *state space*, and set of states visited by a given system at different points in time is known as its *orbit*.

In the case of the pendulum, the state is described by two real values, which makes the state space a simple two-dimensional Euclidean space. As the pendulum swings back and forth, the point describing its state traces out a pattern in state space. If the pendulum swings without friction or resistance, the pattern traced out is a circle. If we introduce friction and resistance, the system's state forms a spiral as the pendulum slowly comes to rest.

There are two types of dynamical systems. Those that have a continuous time parameter, like the pendulum, and those for whom time proceeds in discrete steps. Continuous dynamical systems can be described with differential equations, discrete dynamical systems are described as functions from the state space to itself, so that the point at time t can be found by applying the function iteratively, t times, starting with the initial point: $x_5 = f^5(x_0) = f(f(f(f(x_0))))$.

The trajectory of a point in state space, its *orbit*, can take on many different forms. Dynamical systems theory describes several classes of behavior that an orbit can tend towards, known as *attractors*. The simplest attractors are just

[2] Under the assumptions of classical mechanics.



(A) The Lorenz attractor (B) The Rössler attractor (C) A Pickover attractor

FIGURE 1.7: Three examples of strange attractors. A and B are defined by differential equations in a three dimensional state space, C is defined by a map in a two dimensional state-space. See E.2 for attributions.

points. The orbit falls into the point and stays there, like the pendulum with friction tending towards a resting state. A slightly more complicated attractor is the periodic orbit. The orbit visits a sequence of states over and over again, like the frictionless pendulum. Note that any orbit that hits a state it has visited before must be periodic, since the state fully determines the system, and the orbit from it. Both of these attractors also have limit cycling variants, orbits that continually approach a point, or a periodic orbit, but never exactly reach it.

A more complicated type of attractor, and the phenomenon that gives chaos theory both its name and its popularity, is the strange attractor. An orbit in the trajectory of a strange attractor stays within a finite part of state space, like a periodic attractor, but unlike the periodic attractor, it never visits the same state twice, and unlike the limit cycling attractors, it never tends towards any kind of simple behavior. The orbit instead follows an intricately interleaved pattern and complicated tangles. Figure 1.7 shows three examples. Almost all strange attractors have a non-integer dimension, and so can be seen as fractals.

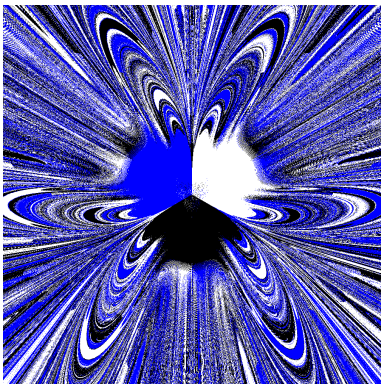


FIGURE 1.6: Basins of attraction for the magnetic pendulum.

Fractals can be found in dynamical systems in another place. Consider the following dynamical system: a pendulum with an iron tip is allowed to swing free in three dimensions. Below it, three magnets are mounted in a triangular formation. The magnets are strong enough to pull the pendulum in and allow it to come to rest above one of them. The question is, given a starting position from which we let the pendulum go, which of the three magnets will finally attract the pendulum? If we assign each magnet a color, we can color each starting point in the plane by the magnet that will finally bring the pendulum to a halt. Figure 1.6 shows the result.

It's clear that for this dynamical system, each magnet represents a simple point

attractor. The colored subsets of the plane represent the *basins of attraction* of each attractor. The set of initial points that will lead the orbit to that attractor. As is clear from figure 1.6, the figure has a complicated fractal structure.

The type of self similarity exhibited by strange attractors and fractal basins is far less well-defined than that of iterated function systems. Zooming in on one of the wings of the Lorenz attractor will show layered bands of orbits, and zooming in further will reveal that these are themselves made up of ever thinner bands. In that sense there is self similarity, but there is no simple transformation that will turn the whole attractor into a subset of itself. The fractals resulting from dynamical systems are likely difficult to learn. Far more so than the iterated function systems or even the random fractals found in nature.

1.1 Machine learning

The phrase *Machine Learning* describes a very broad range of tasks. Speech recognition, controlling a robot, playing games or searching the internet, all tasks that can be implemented on a computer and where the computer can learn to perform better by trying different strategies, observing user behavior or analyzing data. And each task requires its own domain knowledge, different algorithmic complexities, and different hardware.

To bring together all machine learning research, so that research in one field can benefit from achievements in another, a kind of framework has emerged which emphasizes the elements common to all learning tasks, so that general algorithms can be designed for a wide range of learning tasks and algorithms can be effectively compared with each other.

While general elements are common to all machine learning research, there is of course no single standardized framework, and we'll define here precisely the type of machine learning on which this thesis will focus. This definition will be general enough to allow us to make a broad point about fractals in machine learning, yet tuned to make sure that the concept of *fractal data* has a meaningful interpretation.

We will limit ourselves to off line learning. We present an algorithm with a dataset, the algorithm will build a model and we will then test the model on unseen data. The dataset has the following structure. A dataset X consists of n *instances* $x^1, ...x^n$. Each of these instances represents an example of the phenomenon we are trying to learn about. This could range from people, to states of the stock market at given moments, to handwritten characters we are trying to recognize, to examples of sensory input to which a robot must respond. We must represent these instances in a uniform way so that a general learning algorithm can process them. For this purpose we represent each instance by fixed number, d, of features.

Each feature $x^i_1, \ldots, x^i_d$ can be thought of as a 'measurement' of instance x^i. If the instances in our dataset represent people, we might measure such things as height, hair color or the date of birth. However, in this thesis, we will constrain the datasets to numeric features—measurements that can be expressed as a single number per instance. Note that dates and nominal features (like hair

color) can be converted to numeric features in many ways, so our requirement does not necessarily rule them out. Thus, our dataset can be seen as an n by d table, where each column represents a type of measurement, each row represents a measured instance, and each cell contains a number.

This representation, with the constraint of numeric features, allows us to model a dataset as a set of points in a d-dimensional space. Each point represents an instance and each dimension represents a type of measurement. We will call d the *embedding dimension* of the dataset.

Now that we have a well defined dataset, we can define learning tasks. We will limit ourselves to two common learning tasks.

density estimation This task requires only the dataset as described. We assume that the instances in X are random variables, drawn from some distribution which we wish to model. In other words, given some new instance x what is the probability, or probability density, of seeing that instance?

For example, what is the probability of seeing a two meter tall man with 10 cm ring fingers. Or, what is the probability of seeing the Dow Jones index close above 10300 and the NASDAQ Composite below 2200?

classification For this task, each instance x^i is also assigned a *class* $y^i \in \{C_1, C_2, ..., C_m\}$. For instance, each person in a dataset may be sick or healthy, or each day in a dataset may be rainy, cloudy or sunny. The task is, given a new instance, to predict its class.

1.2 Fractals and machine learning

In this section we will highlight some of the advantages and pitfalls that the application of fractal geometry to the problem of machine learning may bring.

1.2.1 Fractal learning

When we talk about fractals and machine learning, we must distinguish between two scenarios. The first is the learning of fractal data, using any kind of machine learning algorithm. How well do common machine learning algorithms, such as neural networks and decision trees, perform on data that has a well defined fractal character? The second scenario is the use of fractal models to design new machine learning algorithms. If, for instance, we have a particular family of fractal probability distributions, we can search for the distribution that best describes a given dataset.

Intuitively, the two are likely to complement one another. If the fractal data that we are learning is in the set of fractals we are using to model the data, then our algorithm has an obvious advantage to other algorithms, because it can theoretically fit the data perfectly. If we generate a large number of points on the Sierpinski triangle, using the chaos game, we know that the set of iterated function systems contains a perfect model for this data, because the Sierpinski triangle can be described as a an iterated function system. This model will provide a far better fit than any non-fractal model can, simply because the exact same model was also used to generate the points.

The main caveat here is that the type of fractal most likely to be encountered in real-life datasets is the random fractal. Random fractals are fractals that are driven by a random process and have no simple finite description. Examples of random fractals are natural phenomena like coastlines, clouds or the frequency of floods in the Nile delta, but also man-made phenomena like the change of currency conversion rates or commodity prices over time. We cannot reasonably expect any finite model to be able to describe random fractal data exactly. We can only hope to find a deterministic model that fits the data well, or a finite description for the total family of random fractals under investigation.

Our initial attempts to do the former can be found in chapters 3 and 4, for the latter, see chapter 5.

Of course, even if deterministic fractal models cannot hope to model random fractals precisely, it is still conceivable that their performance on fractal datasets is competitive with that of non-fractal models.

1.2.2 The curse of dimensionality

The *curse of dimensionality* refers to various related problems which occur when we wish to use statistical techniques on datasets with a high dimensionality. (Bishop *et al.*, 2006)

Consider for instance, that to sample a one dimensional interval of length one with a distance of 0.01 between sample points, a set of 100 points will suffice. In two dimensions, we might initially expect that a grid of points spaced at a distance of 0.01 produces an analogous sampling, but in fact these points are only 0.01 units apart along one of the two dimensions. Along the diagonals, the distance between the points is $\sqrt{0.01^2 + 0.01^2} \approx 0.014$. To get a distance of 0.01 along the diagonal, we need points spaced about 0.007 units along each dimension. As the dimensionality increases, the problem gets worse.

As we shall see in chapter 2, many datasets that have a high embedding dimension (number of features) have a much lower intrinsic dimension. All non-fractal models are fixed to a single dimension, taken to be the embedding dimension. Fractal models would theoretically be able to adapt their dimensionality to that of the dataset, which could be an important step to solving the curse of dimensionality.

The fact that fractal data can have a high embedding dimension and a low intrinsic dimension has been called the *blessing of self similarity*.(Korn & Pagel, 2001)

CHAPTER 2 · DATA

In this chapter, we investigate the notion that datasets can have fractal properties. We consider the concepts of fractal dimension and self-similarity as they relate to machine learning.

2.1 Dimension

2.1.1 Introduction of concepts

In chapter 1 we saw that certain shapes like coastlines, mountain ridges and clouds, have strange geometric properties, and that these properties can be explained from the fact that they do not have an integer dimension. This solves one problem, but creates another; how do we generalize the concept of dimension so that Euclidean shapes retain their intuitive integer dimensions, and fractals get a new fractional dimension that fits existing geometric laws?

There are many definitions for fractal dimension, but they don't always agree. Thankfully, the disagreements are well understood, and the choice of which definition of dimension to use generally depends on the task at hand.

The idea behind most methods is always the same. Two types of attributes of the object being measured are related by a power law

$$s_1 \propto {s_2}^D \tag{2.1}$$

where the dimension is the exponent D (or in some cases, can be derived from D). For example, the measurement of the length of a coastline is related to the size of the ruler by this kind of relation.

To explain where the power laws come from, we will look at objects as sets of points in $\mathbb{R}^m$ (common objects like lines and cubes are then just sets of infinitely many points). We define an operation S_σ of *scaling by* $\sigma \in \mathbb{R}$ for individual points as $\sigma \cdot x$. When the operation is applied to a set we define it as $S_\sigma(X) = \{\sigma x \mid x \in X\}$. That is, the image of this function is the set of all points in X multiplied by σ. The important thing to realize is that this operation scales all one-dimensional measures by σ. All lengths, distances, radii and outlines are all multiplied by σ.

The Euclidean scaling law (also known as the square-cube law) states that if we measure some m-dimensional aspect of a set (like the volume of a cube or the area of one of its faces) and we scale the cube by σ, the m-dimensional measurement s_m will scale by ${s_m}^m$. If we take some one-dimensional measurement s_1 (like the length of an edge) we get:

$$s_1\sigma = s_m\sigma^m$$

In words, the scaling factor of the measure of some set (length, area, volume or

analog) is determined by a power law, where the exponent is the dimension of the set.

This scaling law was first described and proved by Euclid for specific geometric figures such as cubes and cones. A general version was first demonstrated by Galileo, and has since become one of the cornerstones of science and mathematics. Given the fundamental nature of these scaling laws, it's no surprise that the discovery of fractal counterexamples in the early twentieth century came as a shock. It is because of this (apparent) contradiction, that fractals are still referred to as *non-euclidean*.

The contradiction was finally resolved when the notion of non-integer dimension was introduced and made rigorous. Our task, when faced with a set, is to find this fractional dimension, which makes the scaling laws work. Because of the Euclidean scaling law, there are many relations to be found of the form

$$s_1 \propto {s_D}^D$$

where s_1 is a one-dimensional measurement of a figure, and s_D is a measurement related to the figure's intrinsic dimension. An example of this is the method of measuring the length of the coast of Britain by ever smaller rulers, as mentioned in chapter 1. The size of the ruler is a one-dimensional measure and the scaling of the resulting length measurement is determined by the coastline's intrinsic dimension. In this case of course, the measurement is one-dimensional (which does not match the 1.25 dimensional coastline) so the measurement itself goes to infinity, but the scaling law still holds.

We can use this relation to determine the figure's intrinsic dimension, based on measurements. If we take the measurement of the British coastline as an example, then our basic assumption is that the length of the ruler, ϵ is related to the resulting length measurement L as

$$L = c \cdot \epsilon^D$$

To determine the dimension, we take the logarithm of both sides:

$$\ln L = \ln \left[c\epsilon^D \right] \tag{2.2}$$
$$\ln L = D \ln \epsilon + \ln c \tag{2.3}$$

This is a linear relation between $\ln L$ and $\ln \epsilon$, where the slope is determined by D. If we measure L for various values of ϵ and plot the results in log-log axes, the points will follow a line of slope D. This approach—plotting measurements in log-log axes and finding the slope of a line fitting the points—is the most common way of determining the fractal dimension from measurements. We will describe variations of this method that work on general Euclidean sets of any dimension, and on probability measures.

It should be noted that while the scaling law holds exactly for Euclidean figures, it may hold only approximately for fractals. To capture any deviations from a precise scaling law, we write the relation as

$$L = \Phi(\epsilon)\epsilon^D$$

where the function $\Phi(\epsilon)$ is called the *pre-factor*. Taking the logarithm on both sides again, we can rewrite to

$$D = \frac{\ln L}{\ln \epsilon} - \frac{\ln \Phi(\epsilon)}{\ln \epsilon}$$

If we assume that the contribution of the pre-factor to the scaling is negligible compared to the factor ϵ^D, more specifically, if

$$\lim_{\epsilon \to 0} \frac{\ln \Phi(\epsilon)}{\ln \epsilon} = 0 \tag{2.4}$$

we can still find the dimension using

$$D = \lim_{\epsilon \to 0} \frac{\ln L}{\ln \epsilon}$$

This suggests that if we choose small values of ϵ, we can minimize the error caused by a non-constant pre-factor. Note that 2.4 is quite a general requirement. Under this requirement the pre-factor may oscillate or even become unbounded. The behavior of oscillating deviation from an exact scaling law is called *lacunarity*. Lacunarity is used as a measure of 'texture' of fractals, and to differentiate between two fractals with the same dimension.

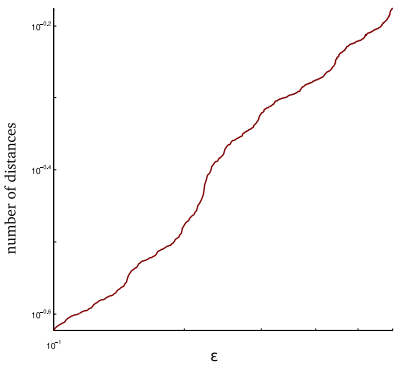


FIGURE 2.1: The power law relation for the cantor set is not exact. In log-log axes it shows periodicity. (The power law relation shown here is explained in section 2.1.5)

It has been suggested that deviation from the scaling law (ie. a lack of correlation between the measurements in the log-log plot) can be taken as an indication of low self-similarity. This is an incorrect assumption (Andrle, 1996). A case in point is the Cantor set. It has perfect self-similarity, but its pre-factor is periodic in the log-log axes. (Chan & Tong, 2001)

2.1.2 Hausdorff dimension

The Hausdorff dimension (Theiler, 1990; Hutchinson, 2000) is generally considered to be the 'true' dimension in a mathematical sense. For most purposes, however, its formulation makes it impossible to work with, so many other definitions have been devised. These serve as approximations and bounds to the Hausdorff dimension. We will provide only a summary explanation of the Hausdorff dimension for the sake of completeness.

We will first introduce some concepts. We will define the Hausdorff dimension of a set of points $X = \{x_i\}$, $x_i \in \mathbb{R}^d$. Let a *covering* E of X be any set of sets so that $X \subseteq \bigcup_{e \in E} e$, ie. the combined elements of E cover X. Let the *diameter* of a set be the largest distance between two points in the set: $\mathrm{diam}(S) = \sup_{x,y \in S} d(x, y)$ [1]

[1] As indicated by the sup operation, where open sets and partially open sets are concerned, we

Let $\mathcal{E}(X, r)$ be the set of all coverings of X with the additional constraint that $\text{diam}(e) < r$ for all elements e in all coverings in $\mathcal{E}(X, r)$.

We can now define the following quantity for X:

$$H^d(X, r) = \inf_{E \in \mathcal{E}(X,r)} \sum_{e \in E} \text{diam}(e)^d$$

If we let r go to zero, we get the Hausdorff measure:

$$H^d(X) = \lim_{r \to 0} H^d(X, r)$$

It can be shown that there is a single critical value $\widehat{d}$ for which

$$H^d = \begin{cases} 0, & d < \widehat{d} \\ \infty, & d > \widehat{d} \end{cases}$$

This value is the Hausdorff dimension, $D_h = \widehat{d}$

The idea behind the Hausdorff dimension can be explained in light of the experiments measuring natural fractals like coastlines. As noted in those experiments, when measures of integer dimension such as length are used, the measurement either goes to infinity or to zero (eg, if we were to measure the surface area of a coast line). If we define a measure that works for any real-valued dimension, it will have a non-zero finite value only for the set's true dimension.

2.1.3 Box counting dimension

The box counting dimension is one of the simplest and most useful definitions of dimension. It is defined for any subset of a Euclidean space.

To determine the box counting dimension, we divide the embedding space into a grid of (hyper)boxes with side length ϵ.[2] The size of the side length of the boxes is related to the number of non-empty boxes $N(\epsilon)$ by a power law:

$$N(\epsilon) \propto \left(\frac{1}{\epsilon}\right)^{D_b}$$

where D_b is the box counting dimension.

For most sets, the box counting dimension is equal to the Hausdorff dimension. If not, the box counting dimension provides an upper bound.

---

take the diameter to the largest value that can be approached arbitrarily closely by the distance between two points in the set.

[2] If the dataset is represented as one dimensional, we divide the space into line segments, a two dimensional space is divided into squares, a three dimensional space into cubes and so on analogously for higher dimensions.

2.1.4 Dimensions of probability measures

When we are considering the dimension of a dataset (a finite set of points), we need to explain clearly what it is we are actually talking about. After all, the dimension—by any definition—of a finite set of points is always zero. What we are actually measuring is the dimension of the underlying probability distribution. In statistical terms the methods described here are estimators of the dimension of a probability distribution.

We assume that each point x in the dataset X is drawn independently from a single probability distribution over $\mathbb{R}^d$. Under this probability distribution $p(A)$ represents the probability that a point x will fall in a region $A \in \mathbb{R}^d$. $p(x)$ represents the probability density of point x.[3] We will refer to the probability distribution itself as p.

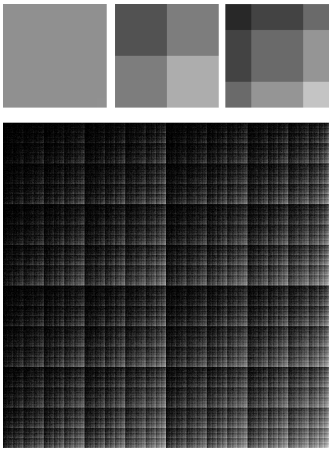


FIGURE 2.2: A fractal probability distribution with a non-fractal support. The brighter a point is, the greater is its probability density. The three images above show its construction from a simple uniform distribution.

We could simply create an analogy of the box counting dimension for probability measures by dividing the instance space into ever finer boxes and counting the number of boxes that have a non-zero probability. For certain distributions, this would work very well. For example, we can define a probability distribution by choosing random points on the Sierpinski triangle. For this distribution, the box counting method would return the same value as it would for the Sierpinski triangle. More generally, this method returns the dimension of the probability distribution's support, the region of non-zero probability.[4].

For some distributions, this behavior is undesirable. Consider for instance the probability distribution shown in figure 2.2. The distribution's support is the full square pictured. No region of the image has zero probability. This means that its box counting dimension by the method described above is precisely 2. This is unsatisfactory because the distribution clearly has a fractal structure (in fact it's a simple iterated function system). If we cover the image with boxes of a small side length, a large percentage of the probability mass will be in a small percentage of the boxes. This is the structure we are actually interested in, because it conveys the fractal nature of the distribution, but because no box will ever have probability truly zero, we do not get it from the box counting dimension.

To deal with this problem, a 'spectrum' of dimensions was introduced, known as the *Renyi dimensions* or the *generalized dimension*. (Theiler, 1990) Let C^ϵ be the

[3] In the whole of this text, we will use $p(\cdot)$ for both probability densities when the argument is a point and probabilities when the argument is a set. We trust that context will suffice to distinguish between the two.

[4] Formally, the support is the smallest set whose complement has zero probability

set of boxes with side-length ϵ used to cover the distribution in the definition of the box counting dimension. We then define

$$I_q(\epsilon) = \sum_{c \in C^\epsilon} p(c)^q$$

This function is the analog of the box count in the box counting dimension. Here, instead of counting non-empty boxes, we count the probability mass of all the boxes, raised to a weighting parameter $q \in \mathbb{R}$.

We can then define a dimension for every value of q:

$$D_q = \frac{1}{q-1} \lim_{\epsilon \to 0} \frac{\log I_q(\epsilon)}{\log \epsilon}$$

The value of q in this function determines how much importance we attach to the variance in probability between boxes. It can be shown that D_q is a non-increasing function of q (Ott *et al.*, 1994). Distributions for which D_q varies with q are called *multifractal* distributions. (Or multifractal measures, which is a more common phrase).

Some specific values of q are particularly interesting. D_0 is the box counting method described earlier. For $q = 1$, the dimension becomes undefined, but we can let $q \to 1$ and solve using de l'Hôpital's rule, which gives us

$$D_1 = \lim_{\epsilon \to 0} \frac{1}{\ln \epsilon} \sum_{c \in C^\epsilon} p(c) \log p(c)$$

The last part of this equation is the negative of the information entropy $H(X) = -\sum p(x) \log p(x)$. Because of this, D_1 is known as the *information dimension*. What this equation effectively states is that as we discretize the instance space into smaller boxes, the entropy of the set of boxes and their probabilities increases by a power law, where the exponent represents the dimension of the probability distribution.

If we take $q = 2$, we get

$$D_2 = \lim_{\epsilon \to 0} \frac{1}{\log \epsilon} \log \sum_{c \in C^\epsilon} p(c)^2$$

This is known as the correlation dimension. It can be very useful because it is easy to approximate, as discussed in section 2.1.6.

Dynamical systems

We can easily adapt this approach to define a dimension for dynamical systems. The following treatment, will give us a definition of the probability of points in an orbit of a dynamical system that can be used to calculate its dimension.

For the first time period t from initial state x_0 we will call the amount of time that the system spends in region A, $\eta(A, x_0, t)$. We then define a measure $\mu(A, x_0)$ as

$$\mu(A, x_0) = \lim_{t \to \infty} \frac{\eta(A, x_0, t)}{t}$$

For many point processes $\mu(A, x_0)$ is the same for all x_0, with the possible exception of a set of initial conditions of zero Lebesgue measure, (ie. a zero chance of being chosen at random from a uniform distribution). In those cases, we can discard the x_0 argument, and call $\mu(A)$ the *natural measure* of the point process. This measure fits all the requirements of a probability measure. We will use this as our probability measure to determine the dimension of a dynamical system, $p(A) = \mu(A)$.

For a completely deterministic process like a dynamical system, it may seem counter-intuitive to call this a probability measure. The whole orbit is after all, completely determined by a single initial state, without any probabilistic elements. In chaotic attractors, however, the slightest error in initial conditions will be amplified exponentially with time, making the theoretically fully predictable orbit practically unpredictable. If we represent this uncertainty of the initial conditions as a uniform probability distribution over a small neighbourhood of x_0, which is transformed by the dynamical system's map, then after a short amount of time our uncertainty about the orbit will have spread out over phase space into a stable, fractal probability distribution. In other words, under the iteration of the dynamical system, the probability distribution converges to the system's natural measure. In this sense it is natural to consider μ a probability distribution, because it represents our uncertainty of the system when the initial conditions are not known with absolute precision.

2.1.5 Point-wise dimension and correlation dimension

If we take any point x on the support of the probability distribution, we define the pointwise dimension (Theiler, 1990) $D_p(x)$ as:

$$D_p(x) = \lim_{\epsilon \to 0} \frac{\ln p\left(B_\epsilon(x)\right)}{\ln \epsilon} \tag{2.5}$$

Where $B_\epsilon(x)$ is a ball of radius ϵ centered on x. For many distributions, the pointwise dimension is independent of x. For those cases where it isn't, we can define a weighted average over all points:

$$D_{\tilde{p}} = \int D_p(x)\ dp(x)$$

It is more practical, however, to average before taking the limit in 2.5. To this end we define the correlation integral $C(\epsilon)$:

$$C(\epsilon) = \int p(B_\epsilon(x))\ dp(x)$$

And we define the correlation dimension as

$$D_c = \lim_{\epsilon \to 0} \frac{\ln C(\epsilon)}{\ln \epsilon}$$

The usefulness of the correlation integral should become apparent from the

following identities:

$$\begin{aligned} C(\epsilon) &= \int p\left(B_\epsilon(x)\right) dp(x) \\ &= E\left(p\left(B_\epsilon(X)\right)\right) \\ &= p\left(d(X,Y) \leqslant \epsilon\right) \end{aligned}$$

From top to bottom, the correlation integral represents the weighted average probability over all balls with radius ϵ, which is equal to the expected probability of a ball with radius ϵ centered on a random variable X distributed according to p. The final line states that this is equivalent to the probability that the distance between two points chosen randomly according to p, is less than or equal to ϵ (because the ball $B_\epsilon(X)$ is placed at a point chosen according to p, and Y is distributed according to p, $p\left(B_\epsilon(X)\right)$ is the probability that Y falls within a distance of ϵ to X).

The main advantage of the correlation dimension is that it can be easily approximated from data, and that it makes good use of that data. A dataset of n points has $(n^2 - n)/2$ distinct pairs with which to estimate $p\left(d(X,Y) \leqslant \epsilon\right)$, and the approximation is quite simple. The process is further discussed in section 2.1.6

Since we rely on the (generalized) box counting dimension for our interpretation of these measurements, it's important to know how the correlation dimension relates to it. The answer is that the correlation dimension is an approximation to D_2.

To show why, we write the definition of D_2:

$$D_2 = \lim_{\epsilon \to 0} \frac{I_2(\epsilon)}{\ln \epsilon}$$

This is the same as the definition of the correlation dimension, but with $I_2(\epsilon)$ taking the place of $C(\epsilon)$. So what we want to show, is that these two quantities are alike in the limit. We rewrite I_2:

$$\begin{aligned} I_2(\epsilon) &= \sum_{c \in C^\epsilon} p(c)^2 \\ &= \sum_{c \in C^\epsilon} p(c) \lim_{|X| \to \infty} \frac{|\{x \mid x \in X, x \in c\}|}{|X|} \end{aligned}$$

Where X is a set with elements drawn independently from the distribution p, and the numerator simply denotes the number of points in this set that fall in c. The limit's argument "$|X| \to \infty$" denotes that we increase the size of X by sampling more, so that $|X|$ goes to infinity.

We can also write this as

$$I_2(\epsilon) = \lim_{|X| \to \infty} \frac{1}{|X|} \sum_{x \in X} p(c_\epsilon(x))$$

where $c_\epsilon(x)$ is the box in the ϵ-grid that contains x.

If we assume that for small ϵ,

$$p\left(c(x)\right) \backsimeq p\left(B_\epsilon(x)\right) \tag{2.6}$$

then we can approximate I_2 using balls of radius ϵ around the points in a dataset:

$$I_2(\epsilon) \approx \lim_{|X|\to\infty} \frac{1}{|X|} \sum_{x\in X} p(B_\epsilon(x))$$
$$I_2(\epsilon) \approx C(\epsilon)$$

The move from cubes to balls in 2.6 can be justified by noting that the correlation dimension does not change when we switch from the Euclidean distance to the Chebyshev distance—in fact it is the same for all L^p norms (Theiler, 1990, p. 1059). For two points $x, y \in \mathbb{R}^d$, the Chebyshev distance is defined as

$$d_t(x, y) = \max_{i\in(1,d)} |x_i - y_i|$$

When we use this as the distance between points, the notion of a ball centered at point x, with radius ϵ is still defined as $B_\epsilon(x) = \{y | d(x, y) \leqslant \epsilon\}$. The points that form a ball under the taxicab distance actually form a cube under the Euclidean distance, which tells us that the distinction between balls and cubes is moot. When using the Chebyshev distance the only difference between $I_2(\epsilon)$ and $C(\epsilon)$ is that the former are aligned to a grid, and the latter are centered at data points.

For a generalized version of the correlation dimension (ie. for all values of q), we refer the reader to (Theiler, 1990) and the references therein.

2.1.6 Measuring D_q

We can use the definitions of dimension that we have seen so far to estimate the dimension of a probability distribution by a large sample of points drawn from it. In all cases, we can assume without loss of generality that our finite dataset X fits inside a bounding box $[0, 1]^d$, because affine scaling will not affect the dimension.

The box counting estimator

The most straightforward estimator is the box counting estimator. To estimate D_0 from a dataset $X = \{x_1, \ldots, x_n\}$, we simply count $N(\epsilon)$ the number of boxes in a grid with box-sidelength ϵ that contain one or more points in our dataset. By the definition of box counting dimension:

$$D_0 = \lim_{\epsilon\to 0} \lim_{|X|\to\infty} \frac{\ln N(\epsilon)}{\ln \epsilon^{-1}}$$

Where "$|X| \to \infty$" indicates that the size of our dataset goes to infinity (ie. we sample an infinite number of points). As mentioned in the previous section, this relation means that if we plot N_ϵ for a range of relatively small ϵ's in log-log axes, the data will form a straight line. The slope of this line represents the dimension.

A problem that occurs for all estimators is that in a finite dataset, the power law only holds for a limited range of scales. Because our dataset is finite, the box count will become constant for ϵ smaller than a certain value. At some point each data point has its own box. There will also be a largest scale, above which $N(\epsilon)$ will not follow the scaling law anymore, because our dataset is bounded. In practice, this means that we must create the log-log plot and look for a range in which the points follow a line. We then perform a linear least-squares fit on the points within that range to find the slope.

To create an efficient implementation of the algorithm, we choose the values for ϵ from $1, \frac{1}{2}, \frac{1}{4}, \cdots, \frac{1}{2^n}$. This has the convenience that each successive step divides the previous boxes into precisely 2^d boxes each, where d is the embedding dimension.

The partitioning this creates in our space has a tree structure, where the whole bounding box represents the root note, the first 2^d boxes with $\epsilon = \frac{1}{2}$ represent its children and they each have 2^d children for the boxes with $\epsilon = \frac{1}{4}$ and so on, down to some maximum depth n.

We can very quickly generate this tree in memory by iterating over all $x \in X$ and for each point calculate its index in the tree $\sigma = \{\sigma_1, \cdots, \sigma_n\}$. This allows us to create just that part of the tree that represents boxes with points in it. From this tree we can get $N(\epsilon)$ for each ϵ. The algorithm we used is described in section C.6. Further optimizations are described in (Traina Jr *et al.*, 2000).

For different values of q, it is straightforward to maintain a count of the number of points per box and find an estimate of I_q from the data.

The box counting estimator has the advantage of being fast, but it is known for slow convergence, particularly for large dimensions and values of q below one (Grassberger & Procaccia, 1983; Greenside *et al.*, 1982; Theiler, 1990)). Ironically, this means that the box counting estimator may be a poor choice for estimating the box-counting dimension in many situations. It's primary virtues are that it can be implemented efficiently and it provides simple estimators for arbitrary values of q.

The correlation integral estimator

To estimate D_2 we can find an estimate of the correlation integral. We can estimate the correlation integral simply as

$$\bar{C}(\epsilon) = \frac{\text{number of distinct pairs in the dataset with distance} < \epsilon}{\text{total number of distinct pairs}}$$

We again plot a range of values for ϵ against their correlation integrals in log-log axes and find a line though the points.

More precisely, we define $\bar{C}(\epsilon)$ as

$$\bar{C}(\epsilon) = \frac{1}{\frac{1}{2}(|X|^2 - |X|)} \sum_{i=1}^{|X|} \sum_{j=i+1}^{|X|} [d(x_i, x_j) \leqslant \epsilon]$$

where $[\cdot]$ are Iverson brackets (representing 1 if the argument is true and 0 otherwise). We can approximate the limit on the right with a sufficiently large dataset.

In some publications the sums are taken over all pairs, instead of all distinct pairs. Besides being more computationally expensive, this introduces an unnecessary bias when points paired with themselves are counted. The idea behind this estimate of the correlation integral is that

$$p\left(B_\epsilon(y)\right) \approx \frac{1}{|X|} \sum_{x \in X} [d(x,y) \leqslant \epsilon]$$

Averaging this approximation over all points in the dataset leads to $\bar{C}(\epsilon)$. But when the point y for which we are approximating $p(B_\epsilon(y))$ appears in the dataset, this introduces a bias, as our estimate will not go to zero with ϵ. To avoid this bias we must exclude the center point from the dataset:

$$p\left(B_\epsilon(y)\right) \approx \frac{1}{|X/y|} \sum_{x \in X/y} [d(x,y) \leqslant \epsilon]$$

The results section shows a small experiment to see the effect of the distinction.

The Takens estimator

The drawback of both the box counting and the correlation integral estimator is the requirement of choosing a range of scales for which to calculate the slope. Floris Takens derived an estimator which under certain assumptions is the maximum likelihood estimator of the correlation dimension and requires an upper limit ϵ_0 to the scales, but no further human intervention. (Takens, 1985; Chan & Tong, 2001)

The first assumption is that the scaling law is exact. That is, for $\epsilon \leqslant \epsilon_0$, $C(r) = c \cdot \epsilon^D$, where c is a constant and D is the dimension. This is usually not perfectly true for fractals, but as noted before, c is generally at least asymptotically constant.

We now define the set containing all distances between point pairs less than ϵ_0:

$$A = \{d(x,y) \mid x, y \in X, x \neq y, d(x,y) \leqslant \epsilon_0\}$$

The second assumption is that these distances are iid. If so their distribution can be derived from the first assumption as

$$p\left(d \leqslant \epsilon \mid d \leqslant \epsilon_0\right) = \frac{p(d \leqslant \epsilon)}{p(d \leqslant \epsilon_0)} = \frac{C(\epsilon)}{C(\epsilon_0)} = \frac{c\epsilon^{D_t}}{c{\epsilon_0}^{D_t}} = \left(\frac{\epsilon}{\epsilon_0}\right)^{D_t}$$

Without loss of generality we can assume that the data has been scaled so that $\epsilon_0 = 1$ (this can be achieved in the algorithm by dividing every distance by ϵ_0). We take the derivative of the probability function above to get the probability density function for single distance values:

$$p(\epsilon) = D_t \epsilon^{D_t - 1}$$

We can then find the log-likelihood of the data A with respect to the dimension:

$$l(D_t) = \prod_{a \in A} D_t a^{D_t - 1}$$
$$\ln l(D_t) = \sum_{a \in A} \ln D_t + (D_b - 1) \ln a$$

The derivate of the log likelihood is known as the *score*. Setting it to zero will give us the maximum likelihood estimate of D_t:

$$\frac{d \ln l(D_t)}{dD_t} = \frac{|A|}{D_t} + \sum_{a \in A} \ln a$$
$$\frac{|A|}{D_t} + \sum_{a \in A} \ln a = 0$$
$$D_t = -\frac{|A|}{\sum_{a \in A} \ln a}$$

Therefore, under the assumptions stated, the negative inverse of the average of the logarithm of the distances below ϵ_0 is a maximum likelihood estimator for the correlation dimension. [5]

Besides being a maximum likelihood estimator under reasonable assumptions, the Takens estimator has the advantage of being defined in a sufficiently rigorous manner to determine its variance. For this and other statistical properties of the Takens estimator we refer the reader to the original paper (Takens, 1985), and the perhaps more accessible treatments in (Chan & Tong, 2001) and (Theiler, 1990).

An additional advantage of the Takens estimator is that it can be largely evaluated automatically. This allows us, for instance, to plot a graph of the dimension estimate against some parameter at a high resolution (whereas with the other estimators, we would have had to manually fit a slope through the data for each value of the parameter). Another use would be to train an algorithm with a bias towards a certain dimension, where the Takens estimator is calculated many times during the training process. All that is required in these cases is a value of ϵ_0 that is reasonable for the whole range of measures we are considering. It has been suggested that the mean of D plus one standard deviation is a reasonable value for general cases. (Hein & Audibert, 2005)

An important drawback of the Takens estimator is the reliance on an exact scaling law. The estimator will still work (though not optimally) when the pre-factor is asymptotically constant. When the pre-factor oscillates (as it does with the ternary cantor set) the Takens estimator has been reported to fail (Chan & Tong, 2001) (though in our experiments, the effect seems to be negligible compare to other factors, see section 2.3.3).

[5] Note also that the choice of base of the logarithm is not arbitrary. For other bases than the natural logarithm, the definition of the maximum likelihood estimator changes.

2.1.7 The use of measuring dimension

At this point, a valid question to ask is what to do with a measurement of dimension. Why has so much attention been given to this subject in the literature? What can we do with the results of all these experiments?

The original answer lies mostly in physics. As we have seen in chapter 1, the field of dynamical systems often encounters fractals in the form of strange attractors. The basic principle of physics requires that a theory makes measurable predictions which can then be tested by taking measurements. For theories that predict fractals, measurements become problematic, because most measurements are based on Euclidean concepts such as length, area or volume. Simply put, dimension is one of the few verifiable things a theory can predict about a fractal. In cases where fractals are studied by physicists, the task of the theorist is to predict a fractal dimension, and the task of the experimentalist is to measure it.

A more statistical use of dimension measurement is the task of differentiating between noise and structured data, specifically between chaos and noise. In common parlance the words chaos and noise are often used interchangeably, but in mathematics and physics they have specific and strictly different meanings. Both are signals that appear 'messy', but where chaos differs is that a chaotic high dimensional system is driven by a low-dimensional strange attractor, whereas noise tends to have a simple probability distribution with an integer dimension equal to the embedding dimension. Measuring the dimension of data can tell us whether we are dealing with chaos, noise, or a combination of the two.

In the context of machine learning, measurements of dimension have been used in dimensionality reduction. The fractal dimension of a point set can serve as an important guideline in determining the optimal number of dimensions to choose when applying dimensionality reduction. (Kumaraswamy, 2003; Traina Jr *et al.*, 2000)

2.2 Self similarity

The word fractal is not precisely defined. Mandelbrot, who coined the term, originally defined fractals as sets whose Hausdorff dimension does not equal their topological dimension (he later revised this, saying that he preferred to think of the word as not precisely defined). Under this definition, we could simply use one of the estimators above and show that many important natural phenomena likely have a non-integer fractal dimension.

While this satisfies the letter of our thesis that there are naturally occurring datasets[6] that contain fractal structure, it neglects the spirit, which is that datasets contains fractal structure that can be exploited in machine-learning scenarios. For fractal structure to be exploitable, the datasets must be self-similar. Unfortu-

[6] In this text we often speak of 'natural data', or 'naturally occurring datasets'. This simply means non-synthetic data. Ie. data that is generated by an unknown process, which is relevant to some real-world learning problem. It does not mean data that is necessarily confined to the domain of nature.

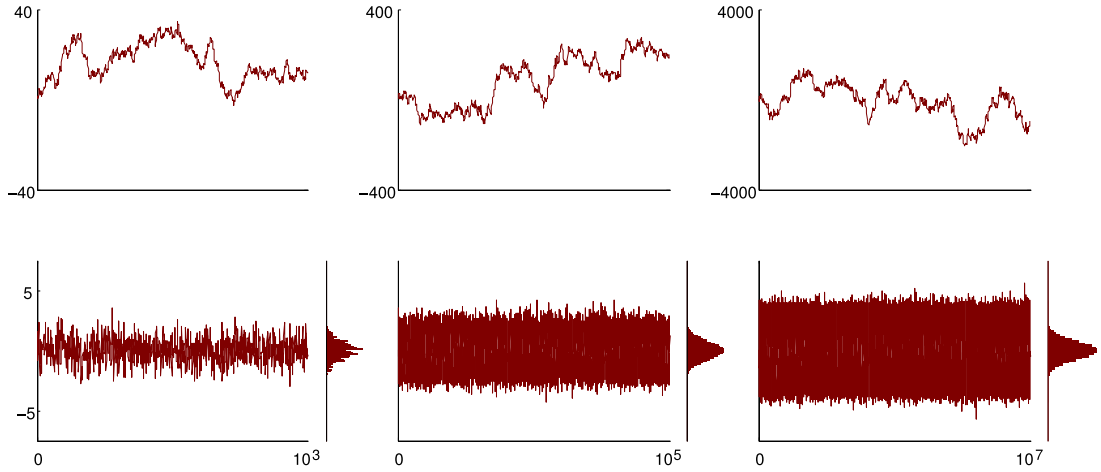


FIGURE 2.3: Three random walks (above) with normally distributed increments (below). A histogram of the increments is plotted sideways. Each plot increases the timescale by a factor 100. Because the vertical axes are rescaled by $100^{1/2}$ between plots, the graphs look statistically similar. (The vertical axes of the increment plots are not rescaled between plots)

nately, very little research has been done into measuring the notion of self similarity independent of dimension, and there is (to our knowledge) no method to quantify exactly the level of self similarity in a set or measure.

What we *can* do, is look at 'temporal' self-similarity instead of 'spatial' self-similarity. This is a phenomenon that has been studied in depth for a wide variety of domains, like hydrology, geophysics and biology. It is used in the analysis of time-series, which makes this section an excursion from the type of data described in chapter 1, but a necessary one to show that self similarity does occur abundantly in natural data.

Let x_t be a series of measurements with $x_t \in \mathbb{R}$ indexed by a finite set of values for t (which we will assume are equally spaced in time). For a thorough investigation of time series it is usually necessary to model them as stochastic processes, but for our light introduction we can forgo such rigor.

A simple time series is generated by a process called a random walk. At each time t the value of x increases by a value drawn randomly from $N(0, 1)$. Figure 2.3 shows some plots of this process over various timescales. Here, we can see exactly why these processes are considered self similar. If we rescale the time axis to a particular range in time, and rescale the vertical axis to the range of the values within the timescale, the plots become statistically similar. The process lacks an inherent scale. This type of process—and the related *Brownian motion*, which is achieved by letting the number of timesteps go to infinity and the variance of the increments to zero—are commonly used in fields like finance to model the behavior of stock returns or price charts.

Since the random walk is such a simple model, it is also easy to predict optimally. The value x_{t+n}, n timesteps after some observed value x_t, is normally distributed with mean x_t and variance n. This may not result in very precise predictions, but under the assumption of a random walk with a particular dis-

tribution for the increments, the prediction cannot be made any more specific. Under this assumption, a market analyst might take a timeseries for say, the price of cotton, estimate the distribution of the increments (assuming that they are normally distributed) and based on the assumption that the increments are drawn independently each time step, make a prediction of the range in which the price will stay given its current position,with a given level of certainty. For some timeseries, this method would prove disastrous. The first problem is that the increments may not be distributed normally. There are many timeseries where the distribution of the increments has a *fat tail*, which means that compared to the Gaussian distributions, far more probability mass is located in the tails of the distribution. In the Cauchy distribution, for instance, the tails follow the now familiar power law. But even when the increments *are* distributed normally, the assumption of a random walk can be fatal, because for some phenomena the increments are not independent.

We can divide the timeseries with dependent increments into two classes. For the *persistent* timeseries, a positive increment is likely to be followed by another positive increment. For the *anti-persistent* timeseries, a positive change is more likely to be followed by a negative change. Figure 2.4 shows three time series with their increments. As we can see, the persistent timeseries is marked by great peaks and deep valleys. For the market analyst and his random walk assumption, events that should only occur once every millennium.

There is a specific property that can help us determine the level of persistence in a time series, called the *rescaled range*. We will call the range $R(a, b)$ of a section of our timeseries $a \leqslant t \leqslant b$ the difference between the maximum and the minimum value after the trend (the straight line between x_a and x_b) has been subtracted. We will call the sample standard deviation of the values in our section $S(a, b)$. For self-similar processes the rescaled range $\frac{R(a,b)}{S(a,b)}$ has the following property:

$$\frac{R(a, b)}{S(a, b)} \sim (b - a)^H$$

where H is known as the *Hurst exponent*, named for Harold Edwin Hurst, the hydrologist who first described this principle in relation to the yearly flood levels of the Nile. For $H = 0.5$, the increments are independent and the random walk assumption is correct. For $H > 0.5$ the process is persistent, and for $H < 0.5$ the process is anti-persistent.

The Hurst exponent is related to the Hausdorff dimension D_h of the timeseries (or rather, the underlying stochastic process) by

$$D_h = 2 - H$$

At the heart of both persistence and self-similarity lies a principle known as *long dependence*. To show this effect, we can compute the correlation of the series' increments against a time-lagged version of itself. This is known as the *autocorrelation*. For processes with small increments, the autocorrelation for a small timelag will be large, but the correlation will generally decay with the time-lag. For some processes the decay will be so slow that it can be said never to reach zero. These processes are persistent and have a Hurst exponent above

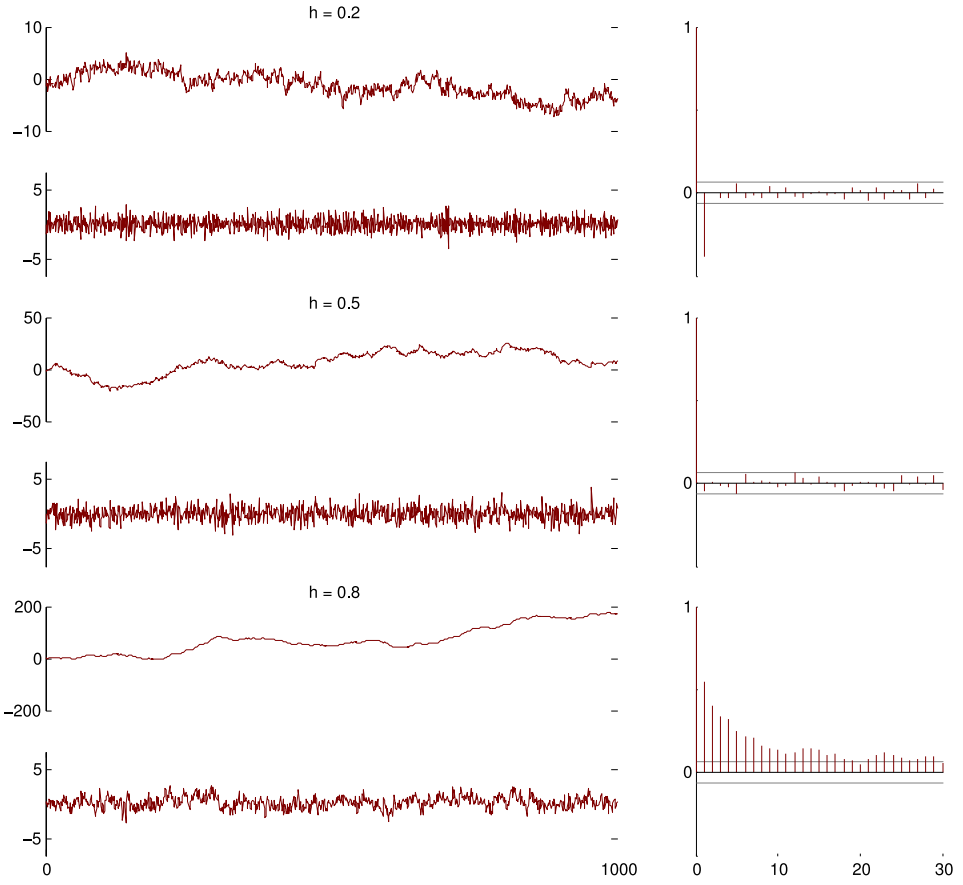


FIGURE 2.4: Three timeseries of 1000 steps with increasing Hurst exponent. For each we have plotted the timeseries (top), the increments (bottom) and the correlogram (right). The gray lines in the correlograms show bounds for a confidence level of 0.05.

0.5. A plot of the correlation versus the lag is called a correlogram. Figure 2.4 shows correlograms for three timeseries of varying persistence.

The way to measure the Hurst exponent should sound familiar. We measure the rescaled range for a variety of sections and plot the resulting values in log-log axes. The slope of the resulting line gives us the Hurst exponent. This is precisely the method we used earlier to determine the dimension of sets and measures. As with the dimension, this method should only be taken as an estimator. Many more have been devised since the Hurst exponent was first introduced and all have their strengths and weaknesses. As with the dimension, measurement of the Hurst exponent often means using many different estimators to make sure the estimate is accurate.

Using the Hurst exponent, correlograms and other statistical methods, many natural timeseries have been shown to be self-similar. For example, the number of packets arriving at a node of a busy ethernet(Leland *et al.*, 1994), biological signals like the heartbeat (Meyer & Stiedl, 2003) or an EEG (Linkenkaer-Hansen

et al., 2001), the human gait (Linkenkaer-Hansen *et al.*, 2001), financial time-series (Peters, 1994) and the yearly flood levels of rivers (Hurst *et al.*, 1965; Eltahir, 1996)

For a more complete and technical treatment of the subject of self-similar processes, we refer the reader to (Embrechts & Maejima, 2002).

2.3 Dimension measurements

2.3.1 Correlation integral

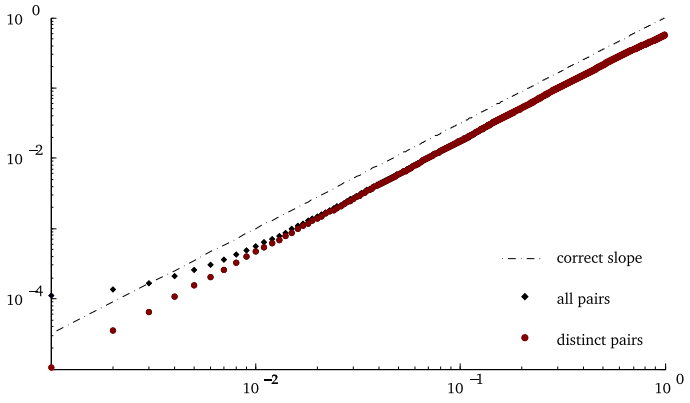


FIGURE 2.5: A log-log plot for the two versions of the correlation integral. The dashed line shows the correct slope (with a different offset). The black diamonds show the method of including all pairs, the red circles show the method of only including distinct pairs.

Early treatments of the correlation integral have given its definition as

$$C'(\epsilon) = \lim_{|X|\to\infty} \frac{1}{|X|^2} \sum_{i=0}^{|X|} \sum_{j=0}^{|X|} u(d(x_i, x_j) - \epsilon)$$

As we discussed, the following is more correct

$$C(\epsilon) = \lim_{|X|\to\infty} \frac{1}{|X|^2 - |X|} \sum_{i=0}^{|X|} \sum_{j=i+1}^{|X|} u(d(x_i, x_j) - \epsilon)$$

To show the importance of this difference, we used both methods on a set of 10 000 random points on the Sierpinski triangle, generated using the chaos game. Figure 2.5 shows the results plotted in log-log axes.

The figure shows that both versions fit the same line, but for the smaller distances C' shows a deviation, while C along the correct slope. This means that for C', the lowest points must be discarded to find the slope, whereas for C all points can be used. Removing the 100 smallest points, we found a value of 1.497 using C', and 1.499 using C. Using all points,we found 1.496 using C' and 1.540 using C.

In addition to the increase in accuracy and usable datapoints, using only distinct pairs and eliminating points paired with themselves also leads a drop of more than 50% in the number of pairs that need to be evaluated.

2.3.2 The box counting and correlation integral estimators

×	Size	E	CI	BC	HD
Sierpinski Triangle	30000	2	1.57	1.53	$\frac{\log 3}{\log 2} \approx 1.585$
Koch	30000	2	1.28	1.15	$\frac{\log 4}{\log 3} \approx 1.26$
Cantor	30000	2	0.53	0.6	$\frac{\log 2}{\log 3} \approx 0.63$
Population	20000	2	1.47	1.73	×
Road Intersections	27282	2	1.76	1.73	×
Basketball	19112	18	2.52	2.37	×
Galaxies	5642	3	2.12	1.98	×
Sunspots	2820	1	0.92	0.88	×
Currency	4774	2	1.63	1.85	×

TABLE 2.1: The results of measuring the dimension of various point sets. Size describes the number of points in the dataset. E refers to the embedding dimension, that is, the dimension in which the points are represented. CI refers to the dimension as measured by the Correlation Integral method. BC is the Box Counting method. HD refers to the Hausdorff dimension, an analytically derived value that can be regarded as correct. The datasets are described in appendix B

2.3.3 The Takens estimator

For the Takens estimator we can plot the estimate against a range of values for the parameter ϵ_0. For datasets with known dimension we can plot the error of the estimate. The results are shown in figure 2.6.

Plots for datasets with unknown dimension are shown in 2.7.

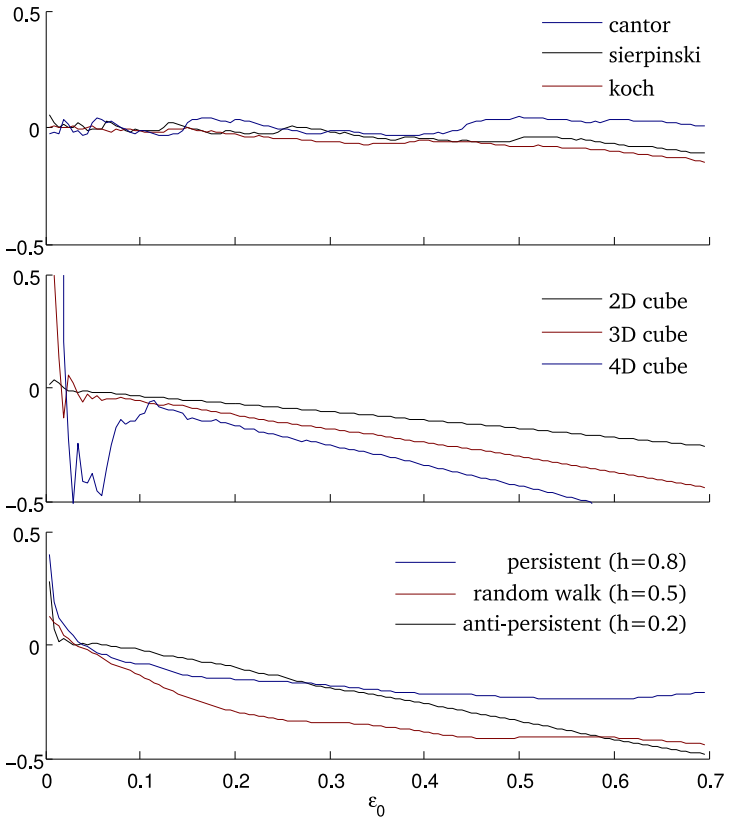


FIGURE 2.6: Plots of the error of the takens estimator against it parameter ϵ_0. The top plot shows the error of three common fractals (scaled to fit the bi-unit square). The middle plot shows the error for a uniform distribution over $(-1,1)^n$. The final plot show the error for the examples of fractional brownian motion with varying Hurst exponents. The top two plots were made with 10000 points each, the bottom plot with 2000 points per dataset. The three plots in the top experiments all have strong lacunarity. As these plots show, this causes some error, but the influence of dimension is much greater. For prints without color, the order of items in the legend is the same as the order of the graphs at $\epsilon_0 = 0.7$.

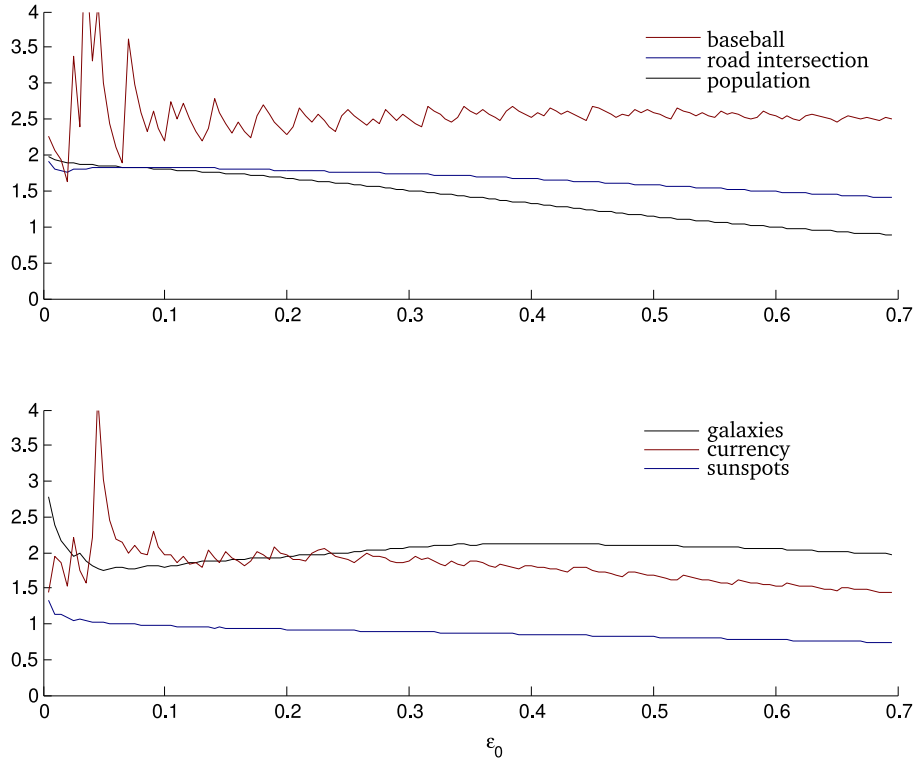


FIGURE 2.7: Dimension estimates using the takens estimator, for varying values of ϵ_0. Datasets of size larger than 10000 we reduced to 10000 points by random sampling (with replacement). For prints without color, the order of items in the legend is the same as the order of the graphs at $\epsilon_0 = 0.7$.

CHAPTER 3 · DENSITY ESTIMATION

In this chapter, we analyze the problem of density estimation from a fractal perspective. We first give a more rigorous explanation of the iterated function system concept, described in section 1.0.1, and its use in modeling measures as well as sets. We introduce methods for learning such models to represent a given set, and compare it to the commonly used Gaussian mixture model.

As we saw in chapter 1, we can create fractals by taking an initial image, combining several (contractively) transformed versions of it to create a new image, and repeating the process. The limit of this iteration is a fractal set. Fractals created in this way are called *Iterated Function Systems*. Examples of fractals which can be generated with this method are the Sierpinski triangle, the Koch curve and the Cantor set. We will define iterated function systems and their properties more precisely. The proofs for the properties described here can be found in (Hutchinson, 2000) (as well as many others).

We will define the IFS concept first for sets in $\mathbb{R}^d$.[1]We will then generalize to (probability) measures over $\mathbb{R}^d$. Finally, in chapter 5, we will discuss an extension of the model known as *random iterated function systems*.

3.1 The model: iterated function systems

3.1.1 IFS sets

Iterated function systems are usually presented by their construction, as above and in the first chapter. To define them mathematically, however, it makes more sense to base our definition on the concept of a *scaling law*.

Call a function $F : \mathbb{R}^d \to \mathbb{R}^d$ a *map*, and let $F(X)$ be the set of points constructed by applying the transformation to all points in the set X (with $X \subset \mathbb{R}^d$).

A set of k maps $S = \langle S_1, \cdots, S_k \rangle$ defines a scaling law. We say that a set K satisfies the scaling law iff:

$$K = S_1(K) \cup S_2(K) \cup \ldots \cup S_k(K)$$

That is, the set is exactly made up of small transformed copies of itself. We will call the maps S_i the *components* of the scaling law.

It can be shown that for a given scaling law, if all components S_i are contractive, there is a unique set which satisfies the scaling law. The reverse is not true. One set may satisfy several different scaling laws. The Koch curve, for instance, can be described by a scaling law of four transformations, and a scaling law of two transformations.

[1]The definition can be generalized to metric spaces.

We can also treat S as a map itself, so that for any $X \subset \mathbb{R}^n$

$$S(X) = S_1(X) \cup S_2(X) \cup \ldots \cup S_k(X)$$

We can iterate S, so that $X_{n+1} = S(X_n)$, with some initial set X_0 called the *initial image*. We can also write this iteration of a functions as $S^{n+1}(X) = S\left(S^n(X)\right)$ with $S^0(X) = S(X)$.

A second fundamental property of iterated function systems is that for any nonempty initial image X_0, the iteration will converge to the single unique set K, which satisfies the scaling law:

$$\lim_{m \to \infty} S^m(X_0) = K$$

for any $X_0 \neq \emptyset$ with

$$S(K) = \bigcup_{i \in (1,k)} S_i(K)$$

3.1.2 Parameter space and code space

If we select the components for a scaling law from a specific family, we can often represent them as a vector in some $\mathbb{R}^m$. For instance, if we limit ourselves to affine maps—maps that can be represented by a transformation matrix and a translation vector—we can represent each S_i as a vector in $\mathbb{R}^{d^2+d}$. A complete scaling law composed of k transformations can then be represented as a vector in $\mathbb{R}^m$ with $m = k(d^2 + d)$. We will call $\mathbb{R}^m$ *parameter space*. It will become useful when we search for a particular scaling law to fit given data.

Let σ be a sequence $\sigma = \langle \sigma_1, \sigma_2, \cdots, \sigma_q \rangle$, with $\sigma_i \in (1, k)$. Let S_σ represent the composition of the components denoted by σ: $S_\sigma = S_{\sigma_1} \circ S_{\sigma_2} \circ \cdots \circ S_{\sigma_q}$. Since all components S_i are contractive, it's easy to see that as $q \to \infty$, $\operatorname{diam}(S_\sigma(K)) \to 0$. This means that every point in S's limit set is denoted by an infinite sequence σ (though not necessarily by a unique one).

3.1.3 IFS measures

Ultimately, the modeling of sets isn't very interesting in machine learning. A more relevant task would be modeling probability measures. Luckily, the IFS concept has been extended to measures as well. As with the sets in the previous section, we will let S be a set of k functions from $\mathbb{R}^d$ to $\mathbb{R}^d$.

We will extend the concept of a map to accept a measure ν as an argument:

$$(F(\nu))(X) = \nu(F^{-1}(X))$$

That is, if ν is a probability measure, $F(\nu)$ is also a probability measure, which assigns the same mass to X as ν would assign to $F^{-1}(X)$.

This allows a straightforward extension of the IFS concept to measures. A measure ν satisfies the scaling law S iff

$$\nu = S_1(\nu) + S_2(\nu) + \ldots + S_k(\nu)$$

If we wish to use probability measures, we must make sure that the total mass over $\mathbb{R}^d$ remains 1. For this purpose we introduce to the scaling law k weights $\rho_i \in (0,1)$, with the additional constraint $\sum \rho_i = 1$. A probability measure p is said to satisfy a scaling law of this type iff:

$$p = \sum_{i \in (1,k)} \rho_i S_i(p)$$

As with IFS Sets, each scaling law can be used as a function, in this case a function of a measure.

$$S(\nu) = \sum_{i \in (1,k)} \rho_i S_i(\nu)$$

The same results hold for IFS measures as do for IFS sets. For a given scaling law over measures with contractive components, there is a unique measure k which satisfies the scaling law.

If ν_0 is an initial measure whose support is compact (ie. closed and bounded) and nonempty, then the iteration S^m will converge to k regardless of the initial measure:

$$\lim_{m \to \infty} = S^m(\nu_0) = k$$

where k is the unique measure satisfying the scaling law.

Generating random points

If the initial measure is a probability measure, then so is the resulting fractal measure. We can draw points from this probability distribution in two ways.

First, we can draw a point x_0 from some initial distribution (with compact [2] measure). We draw a random function S_{r_0} from the probabilities suggested by $\rho_1, \ldots, \rho_k$ and let $x_1 = S_{r_0}(x_0)$. We repeat this process n times, so that $x_n = S_{r_{n-1}}(x_{n-1})$. As n goes to infinity the probability distribution of x_n converges to S's limit set. Practically, $n = 5$ is already enough for many applications. The process works for certain non-compact initial distributions as well (for instance, a multivariate Gaussian distribution).

To generate a large set of points drawn independently from the limit of S we need not draw each point by the method described above. We can use the chaos game described in chapter 1. Start with some initial point x_0 and generate a sequence of points by the same method described above. It can be shown that after some finite initial number of points (usually 50 is enough) the sequence of points behaves as though the points are drawn iid from the IFS's limit measure.

3.2 Learning IFS measures

The task at hand is to find a probability measure $p_\theta(x)$, defined by the parameters θ, which is somehow a good model for a given datasets $X = \langle x_1, \ldots, x_r \rangle$, with $x_i \in \mathbb{R}^n$. To approach this problem we need to define three things: a fitness function to determine which of two models is the best, a representation of

[2] Closed and bounded.

IFS probability distribution (ie. what does θ look like) and a way to search the parameter space for models with a high fitness. We will discuss each of these components individually.

3.2.1 The fitness function

The most common way of determining how well a probability model p_θ fits a given collection of points is to measure the likelihood of the data, given the model. If the points are independently drawn and identically distributed (as we assume they are) the likelihood function of a dataset X is simply given by:

$$l_X(\theta) = \prod_{x \in X} p_\theta(x)$$

The likelihood function works well as a fitness function in many learning contexts where the parameters are gradually changed to slowly find a better and better model. However, a problem emerges when the probability distributions under consideration are not continuous. Consider for instance the family of probability distributions over $\mathbb{R}^d$ that only afford a single point $\theta \in \mathbb{R}^d$ any probability density. All other points have zero probability density. Clearly, for a dataset consisting of only a single point x, the maximum likelihood model has $\theta = x$. However no gradual learning algorithm, like hill climbing, will find these parameters, since any gradual change to a given hypothesis $\theta \neq x$ will yield a likelihood of zero, unless the algorithm stumbles on $\theta = x$ by pure chance. In other words, the fitness landscape has no smooth gradient.

This is a problem for our family of IFS probability models as well. Not only can they be highly discontinuous, but a small change in parameters can cause all points of non-zero density to shift away from the dataset, even if the change is towards the optimal model. What is needed is a fitness function that relates points in the dataset to regions with high probability (under p_θ) by their distance, so that moving towards regions of high probability will count towards the model's fitness even if the actual likelihood of the data doesn't increase. For this purpose we shall use one of the most commonly used tools in fractal geometry, the *Hausdorff distance*.

The Hausdorff distance between two point sets X and Y is defined as

$$d_H(X, Y) = \max \left[\max_{x \in X} \min_{y \in Y} d(x, y), \max_{y \in Y} \min_{x \in X} d(x, y) \right]$$

[3] Imagine a game where we must travel from one set to the other in as short a distance as possible, and a malevolent second player chooses our starting point. The Hausdorff distance will be the distance we travel.

We use the Hausdorff distance to evaluate our models by generating a set of random points from the model under consideration and calculating the Hausdorff distance to the dataset. Since the algorithm to calculate the Hausdorff distance is quite slow, and the dataset will often contain in the order of 10 000 points, we will use a subsample of the dataset (drawn with replacement).

[3] For non-finite sets max and min are replaced by sup and inf respectively. In our case this isn't necessary.

Because the use of Hausdorff distance as a statistical fitness function seems to be unexplored, we investigate more deeply in appendix D.

3.2.2 Representation of the model

To define an IFS probability model in d dimensions, we need two elements. A set of k transformations $\{S_i\}$ and a set of k prior probabilities $\{\rho_i\}$ that sum to one.

We want our model to be expressible as a vector θ in some $\mathbb{R}^m$. We can easily represent each prior probability as an element of θ, but we must ensure that they sum to one and fall in the interval $(0, 1)$. If the first k elements of θ are used to represent the prior probabilities, then we get ρ_i as

$$\rho_i = \frac{|\theta_i|}{\sum_{j \in (1,k)} |\theta_j|}$$

For the transformations, we will limit ourselves to affine transformations. The easiest way to represent these is as a $d \times d$ transformation matrix R and a d-sized translation vector t, so that the transformation S_i becomes $S_i(x) = R_i x + t_i$. This allows us to model each transformation in $d^2 + d$ real values, giving us a total of $kd^2 + kd + k$ parameters for the full model. We will call this the *simple* representation.

The main problem with the simple parameter mode is that it combines several distinct operations into a single set of parameters: the transformation matrix combines scaling, rotation and shearing and folds them into d^2 parameters in a complicated way. Since learning will likely benefit from being able to adjust the parameters for these elementary operations independently, we define an additional parameter mode where they are separated. We split the affine transformation into a scaling vector s of size n, a set α of $\frac{1}{2}n^2 - \frac{1}{2}n$ angles and a translation vector t of size n. The full transformation is then defined as $S_i(x) = R_{\alpha_i} \mathrm{diag}(s_i)x + t_i$, where R_{α_i} is the rotation matrix defined by α_i (an algorithm for constructing this matrix is given in appendix C) and $\mathrm{diag}(s_i)$ represents a matrix with s_i at the diagonal and zero everywhere else. This construction defines a subset of the total set of affine transformations (skewing cannot be represented), but has the advantage of using fewer parameters for the full model ($k(0.5n^2 + 1.5n + 1)$). We will refer to this as the *TSR* representation (for scaling, translation, rotation).

For certain purposes, it helps to only allow uniform scaling. This makes each component of the model a similitude (ie. all distances are scaled by a unique factor under the transformation). This type of representation requires only $k(0.5n^2 + 0.5n + 2)$ parameters to represent. Its main advantage is that for models like these, the dimension is well-defined. We will call this the *similitude* representation.

Alternate model: mixture of Gaussians

To give an idea of its performance in describing the dataset, we compare the iterated function systems to the mixture-of-Gaussians (MOG) model. The MOG

model describes a probability measure as the sum of a number of multivariate Gaussian distributions. If we have a model with k components, we describe each model by its mean μ_i, its covariance Σ_i, and a prior probability ρ_i (so that all priors sum to one). The probability density of a point under this distribution is then

$$p(x) = \sum_{i \in (1,k)} \rho_i N_d(x \mid \mu_i, \Sigma_i)$$

Where N_d represents the d-dimensional Gaussian with the given parameters.

The great advantage of comparing IFS models against MOG models, is that we can make the description the same. It can be shown that any affine transformation of a multivariate Gaussian, is itself a multivariate Gaussian. This means that we can encode the components of a MOG model in a very natural way, as an affine transformation of the standard Gaussian (μ at the origin and $\Sigma = I_d$). If the affine transformation is represented as a transformation matrix R and a translation vector t, then the Gaussian has $\mu = t$ and $\Sigma = RR^T$.

Since this is the exact same way we represent our IFS models (k affine transformations with k priors) we can keep everything else (learning algorithm, fitness functions) the same and have a fair comparison.

3.2.3 Learning

Evolution strategies

The main learning algorithm used is known as *evolution strategies*. We use a specific subset of the full range of possible algorithms. For a more complete introduction, see (Beyer & Schwefel, 2002) and (Schwefel & Rudolph, 1995).

It is one of a family of models inspired by genetic evolution. The algorithm operates on a collection of models known as the population. Every generation, the worst models are discarded and a new population is created from the remaining models, by a process of recombination and mutation. After continuing this process for many generations, the population will (likely) converge to a model that provides a good fit for for the data.

the population The population consists of ν models. Each model in the population is defined by a vector $\theta \in \mathbb{R}^m$ as described above. In addition to these model parameters, each model contains a set of strategy parameters σ, used in the mutation.

Each model's fitness is determined as described above. The top u models are retained, and $\nu - u$ new models are created to restore the population to its original size.

recombination We use the method of *uniform* recombination. For each child to be created, z parents are chosen at random from the remaining models. Each parameter in the child (both the model and strategy parameters) is then chosen at random from one of the parents.

mutation Mutation is the process that drives learning. The idea is to slightly change the parameters of the child after it is created, so that models can

slowly converge to a better fit.

The strategy parameters determine the mutation of the model parameters. They are split up into an m dimensional scaling vector s_σ and an $\frac{1}{2}m^2-\frac{1}{2}m$ set of angles (Where m is the length of the model parameters), which are converted to an $m \times m$ rotation matrix R_σ (see C.4). These define a multivariate Gaussian distribution from which a vector is drawn (if $X \sim N(0, I)$, then $R \cdot \mathrm{diag}(s) \cdot X$ has the required distribution). This vector is then added to the child's model parameters.

The child's strategy parameters are also mutated. To each of the angles a random value is added, drawn (for every parameter) from $N(0, \tau)$. To each of the scaling parameters a value is added that is drawn from $N(0, \upsilon)$. υ can be regarded as the convergence speed; increasing it will cause the algorithm to converge faster, but may cause it to overshoot some values, and it may limit how closely the algorithm can converge to an optimal solution. Decreasing υ will increase the number of generations required to find a solution, and will decrease the likelihood that the algorithm will escape local optima. (Schwefel & Rudolph, 1995) gives an elaborate method for determining υ (from a new parameter), but we had better results without this method (possibly due to implementation errors).

A few notes on this process:

- Per generation the fitness is only calculated once per model. For the next generation, however, we recalculate the fitness, because we calculate it on a random subsample of the data. This way, a model that was lucky in the current generation, in the sense that many well fitting points were chosen from the data, will have to prove itself again in the next generation.
- The order of mutating the strategy parameters and mutating the model parameters has no effect on performance.

3.3 Results

3.3.1 Comparison to mixture of Gaussians

We trained models for various datasets with 2 to 6 components, training for 1000 generations, taking the best model of the final generation as the result. Table 3.2 shows the result for the optimum number of models. We used the same approach to learn a mixture-of-Gaussians model. The MOG models were trained using both the log-likelihood and the Hausdorff distance as a fitness function. We report both the resulting Hausdorff distance and the log likelihood.

Some notes on the experiment:

- Both models were trained using evolution strategies, using the same representation of n affine transformations with a prior probability for each. For the affine transformations, the TSR representation was used. Both versions were trained for 4000 generations.

	d	similitude $\frac{1}{2}d^2+\frac{1}{2}d$	tsr $\frac{1}{2}d^2+\frac{3}{2}d$	straight d^2+d
Three	2	0.15	**0**.13	0.14
Sierpinski	2	**0**.13	0.15	0.18
Koch	2	**0**.05	0.10	0.06
Cantor	2	**0**.01	0.02	0.02
Sunspots	1	**0**.10	0.21	0.51
Population	2	0.63	**0**.12	0.49
Road intersections	2	**0**.30	0.34	0.30
Basketball	18	1.82	2.17	**1**.73
Galaxies	3	**0**.43	0.44	0.47
Currency	8	**0**.71	0.98	0.78

TABLE 3.1: Results for an experiment comparing the three parameter modes. In each experiment, models with four components were trained for the various datasets.

- The IFS transformations were trained with the Hausdorff distance as a fitness measure, using a sample of 250 points from the dataset and the model. A third model was the MOG model with log-likelihood as a fitness function.

- The MOG models were trained with log likelihood as a fitness function (again on a sample of 250 points). It may seem obvious that each model performs better on its own fitness measure, but the MOG models performed no better on the Hausdorff distance when trained with it as a fitness measure. The log likelihood was chosen as a fitness measure because it seems to provide better results.

- The results reported were derived from a test set of 10% of points withheld from the total set. The full set was used to measure the log likelihood. 1000 random points were samples from the test set with replacement to find the Hausdorff distance (to 1000 points generated from the best model at the final generation).

- For the currency and basketball datasets, the rotation angles in the strategy parameters were removed to speed up the algorithm in this and all other experiments.

- The results show positive log-likelihood for some experiments. This may seem impossible as a probability value must be below 1, and thus its logarithm must be negative. However, since we're dealing with probability density values, positive log-likelihoods are quite possible.

3.3.2 **Model representation**

This experiment tests which parameter representation is most successful (see section 3.2.2). Each mode uses a different number of parameters. Fewer parameters make it easier for the model to find a good solution, but they also reduce the number of allowed solutions. The results are shown in table **??**.

	HD			LL		
	ifs_{hd}	mog_{hd}	mog_{ll}	ifs_{hd}	mog_{hd}	mog_{ll}
Three	**0.20**	0.38	0.51	−1503185	−305798	**24303**
Sierpinski	**0.24**	0.33	0.74	−51480	−43885	**−34648**
Koch	**0.05**	0.19	0.92	**7419**	−32813	−858
Cantor	**0.01**	0.11	0.87	−∞	−234362	**183961**
Sunspots	**0.20**	0.33	0.29	−992	−3297	**−654**
Population	**0.17**	0.24	0.43	3842	−32132	**14299**
Road intersections	**0.24**	0.32	0.45	−21291	−28675	**−14972**
Basketball	**1.54**	1.98	4.19	−∞	−∞	−∞
Galaxies	**0.47**	0.49	1.25	−11689	−8451	**−7114**
Currency	**0.67**	0.73	2.15	−∞	−∞	−∞

(A) 2 components

	HD			LL		
	ifs_{hd}	mog_{hd}	mog_{ll}	ifs_{hd}	mog_{hd}	mog_{ll}
Three	**0.14**	0.79	0.74	−∞	−58048	**23179**
Sierpinski	**0.13**	0.35	0.95	**−19691**	−49414	−26764
Koch	**0.09**	0.15	0.38	**23586**	−54437	5078
Cantor	**0.01**	0.12	1.00	−∞	−60682	**145415**
Sunspots	0.15	0.35	**0.12**	−7390	−2980	**−416**
Population	**0.16**	0.73	0.61	−7098	7484	**14679**
Road intersections	**0.22**	0.30	0.58	−37210	−35003	**−15392**
Basketball	**1.36**	1.62	4.20	−∞	−∞	−∞
Galaxies	**0.45**	0.49	0.98	−9611	−8963	**−6780**
Currency	**0.69**	0.71	2.37	−∞	−∞	−∞

(B) 3 components

	HD			LL		
	ifs_{hd}	mog_{hd}	mog_{ll}	ifs_{hd}	mog_{hd}	mog_{ll}
Three	**0.11**	0.11	0.15	−∞	22187	**54793**
Sierpinski	**0.15**	0.31	0.56	**−18162**	−53087	−25938
Koch	**0.10**	0.17	1.24	−669189	−48564	**6917**
Cantor	**0.01**	0.13	0.80	−∞	−6418	**132780**
Sunspots	**0.12**	0.29	0.27	−7888	−4262	**−456**
Population	**0.28**	0.41	0.61	−55502	−24219	**15179**
Road intersections	**0.22**	0.28	1.30	−32236	−36522	**−15294**
Basketball	**1.33**	2.09	4.21	−∞	−∞	−∞
Galaxies	**0.47**	0.66	0.84	−22009	−9732	**−6960**
Currency	**0.68**	0.69	2.25	−∞	−∞	−∞

(C) 4 components

	HD			LL		
	ifs_{hd}	mog_{hd}	mog_{ll}	ifs_{hd}	mog_{hd}	mog_{ll}
Three	**0.09**	0.12	1.20	−∞	**32382**	19602
Sierpinski	**0.17**	0.25	0.97	**−20494**	−54459	−25763
Koch	**0.08**	0.12	0.45	−191268	−74447	**9010**
Cantor	**0.02**	0.05	1.36	−∞	−570	**118131**
Sunspots	0.26	**0.17**	0.20	−1186	−1294	**−407**
Population	0.76	**0.27**	0.34	−3936	−10541	**15176**
Road intersections	0.27	**0.26**	0.33	−37940	−24158	**−17484**
Basketball	**1.37**	1.49	4.20	−∞	−∞	−∞
Galaxies	**0.41**	0.52	0.81	−21080	−8900	**−6923**
Currency	**0.52**	0.70	2.13		**−101244**	−∞

(D) 5 components

TABLE 3.2: Approximating various datasets with IFS and MOG models.

	E	d	2	3	4	5
Three	2	2.00	1.19	0.62	0.66	0.73
Sierpinski	2	1.58	1.52	1.38	1.63	1.47
Koch	2	1.26	1.12	1.27	1.20	1.31
Cantor	2	0.63	0.63	0.65	0.67	0.72
Population	2	1.73	1.35	1.32	1.35	1.48
Road intersections	2	1.73	1.80	1.46	1.67	1.96
Basketball	18	2.37	1.42	1.03	2.24	1.60
Galaxies	3	1.98	3.79	3.99	3.16	3.24
Currency	8	1.85	1.46	1.44	1.65	2.01

TABLE 3.3: Dimensions of models learned (using the TSR parameter mode and varying numbers of components) for datasets. The first two columns show the dataset's embedding dimension (E) and the best estimate we have for the Hausdorff dimension (d). The other columns show the scaling dimension of the learned model.

3.3.3 Dimension of the models

To test the assertion made in chapter 1, that fractal models are capable of varying their intrinsic dimension, we perform we following experiment. We replace the scaling vector of each model by a single scaling value. This makes each component transformation, now represented by a scalar, a translation vector and a rotation matrix, a *similitude*. In short, each pair of points is mapped to a second pair of points so that the distance between the two is always scaled by a fixed value r_i (for component S_i). The advantage of knowing that our model is made up of similitudes is that we know that the *scaling dimension* d_s of the model is defined by the following equation:

$$\sum_{i=1}^{k} r_i^{d_s} = 1$$

Which can be solved numerically. We say that an IFS satisfies the *open set condition* if there exists a non-empty open set $O \subseteq R^n$ such that

$$\bigcup_{i=1}^{k} S_i(O) \subseteq O$$

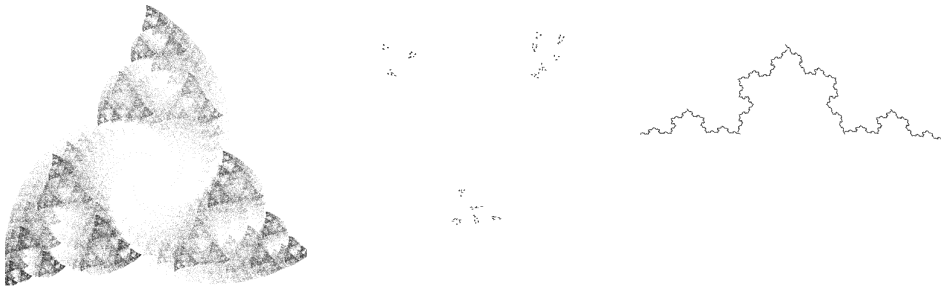
$$S_i(O) \cap S_j(O) = \emptyset \quad \text{if} \quad i \neq j$$

If the IFS satisfies the open set condition the scaling dimension is equal to the Hausdorff dimension.

Table **??** shows the dimensions of the best IFS models after training for 4000 generations.

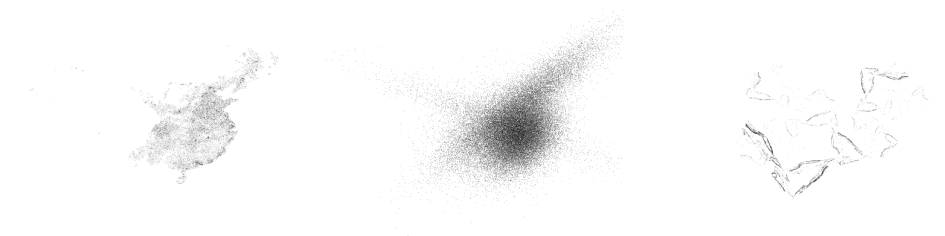
3.4 Gallery

Figures 3.1 – 3.4 provide a selection of the images generated in the experiments shown above.



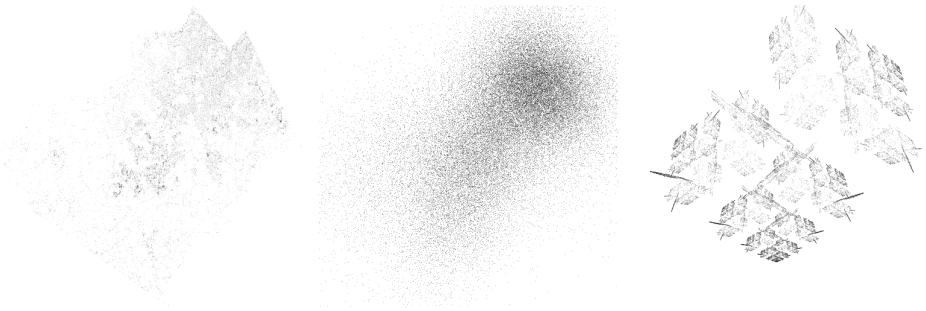
(A) (B) (C)

FIGURE 3.1: Three approximations of synthetic datasets: (3.1a) a 2-component approximation of the Sierpinski triangle, (3.1b): a 5-components approximation of the 'three' dataset and (3.1c) a 2-component approximation of the Koch curve



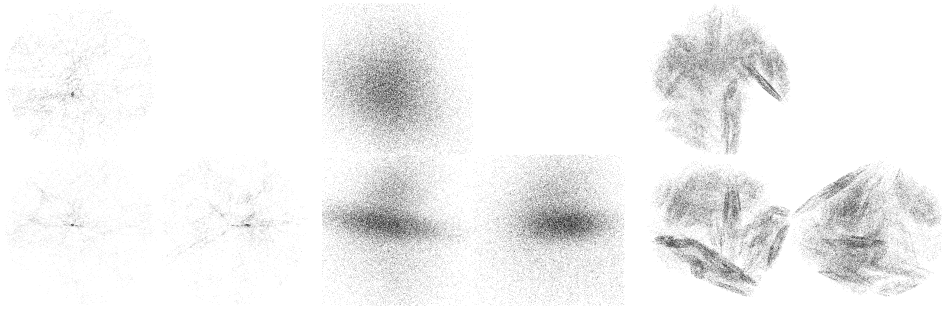
(A) (B) (C)

FIGURE 3.2: MOG and IFS models after 1000 generations of training with 4 components for the population dataset: (3.2a) the dataset, (3.2b): the MOG model trained with log-likelihood as a fitness function (3.2c) the IFS model, trained with Hausdorff distance as a fitness function.



(A) (B) (C)

FIGURE 3.3: MOG and IFS models after 1000 generations of training with 4 components for the road intersection dataset: (3.3a) the dataset, (3.3b): the MOG model trained with log-likelihood as a fitness function (3.3c) the IFS model, trained with Hausdorff distance as a fitness function.



(A) (B) (C)

FIGURE 3.4: MOG and IFS models after 1000 generations of training with 4 components for the galaxies dataset: (3.4a) the dataset, (3.4b): the MOG model trained with log-likelihood as a fitness function (3.4c) the IFS model, trained with Hausdorff distance as a fitness function. (The levels of these images were adjusted to make the detail more visible.)

3.5 Extensions

This section provides some additional ideas and methods for working with IFS models.

3.5.1 Using the model

Once we have learned our model, which hopefully fits the data well, we will want to use it in some setting, to analyze properties of unseen points. Since we learned the model using a fitness measure based on the Euclidean distance, it makes sense to use the point-set distance from the unseen point to a set randomly drawn from the learned measure to approximate how much a point 'belongs' to the set. In some cases, however we are simply interested in the probability density of a point or the probability of a region under the measure we've learned. This becomes particularly relevant when we use the IFS models to build classifiers in chapter 4.

Call the result of our learning the probability measure p. If we want to know the probability density of point x, $p(x)$, we will need to iterate the model. Recall

that we can generate p by starting with some initial measure, and iteratively applying its scaling law S^p. Call p^0 the initial image. We will use the (multivariate) standard normal distribution as our initial image: $p^0(x) = N(x \mid 0, I)$. After one iteration of S^p, the probability density of x will be a weighted combination of the probability densities of x under p^0 transformed by S_i^p:

$$p^1(x) = \sum_{i=0}^{k} \rho_i(S_i{}^p(p^0))(x)$$

Or more generally,

$$p^{n+1}(x) = \sum_{i=0}^{k} \rho_i(S_i^p(p^n))(x) \tag{3.1}$$

The limit of this iteration gives our probability density, but for practical purposes the value often becomes usable after as little as 5 iterations. The reason for choosing a standard normal distribution as p^0 is that a Gaussian distribution under an affine transformation becomes another Gaussian distribution. This means that at every step of the iteration $p^n(x)$ can be calculated directly.

If we want to calculate the probability of a region A under our measure, we can apply the same logic. To approximate 3.1 for regions, we need a way to calculate the probability of a region under a given Gaussian. A method for ellipsoids of arbitrary dimension is given in C.2. This method is necessarily numerical, so that a number of samples need to be taken to get an accurate estimate. The advantage of our model is that as the iteration goes deeper, fewer samples are needed, so that at an iteration depth above 5, a single sample per estimation of $p^n(A)$ can suffice.

CHAPTER 4 · CLASSIFICATION

This chapter investigates the use of fractal models in classification. It builds on the analogue of density models described in the previous chapter, to create a similar analogue to a Bayesian classifier. We also investigate the behavior of some commonly used classification algorithms on simple fractal data.

4.1 Experiments with popular classifiers

In this section, we try to learn the Newton fractal and the Mandelbrot set using common classification algorithms.

The training set for each was created by drawing 100 000 random points uniformly from an enclosing rectangle, and letting the iteration run far enough to approximate the class. In the case of the Newton fractal each of the three solutions represents a class. For the Mandelbrot set, points in the Mandelbrot set are assigned one class, points outside it the other.

We test each algorithms in three ways:

Error This is the straightforward ratio of misclassified points, tested on a second set of 100 000 random points.

MDL ratio This measures how well the model compresses the test set. Ideally, this is the ratio of the dataset represented by the model (including the exceptions required to correct the model's errors) to the size of the dataset represented normally.

A full MDL treatment of the performance of these algorithms is slightly beyond the scope of this section. As the intention of these experiments is more to provide an intuition for the behavior of common learning methods, rather than prove anything conclusively, we use the following imperfect method to give a reasonable idea of the MDL ratio: We take the model M in some representation, the class vector of the dataset V and the same vector with the datapoints that the model classified correctly set to zero, V_e. We then calculate the MDL ratio as:

$$\frac{z(M) + z(V_e)}{z(V)}$$

Where $z(\cdot)$ is a general-purpose compression function (in our case, java's GZIP implementation). The idea here is that M and V_e are sufficient to reconstruct the data, and therefore equivalent to V in terms of information content. Note that we are only considering the class-vector and not the instances themselves. The implementations used are those provided by the WEKA machine learning library. The model's toString() method is taken as a representation of the model (except in the case of the kNN

classifier where we use the dataset itself). As noted, this methodology has its flaws, but we will assume that the resulting measurements are at least sufficiently accurate to give a general idea of the MDL performance.

Visual inspection Because our dataset has two numeric features we can plot the model in two dimensions and inspect its behavior. To create these images, the domain was partitioned into pixels. For each pixel, we took the point at its center and used the model to classify that point. We then colored the pixel according to its class, using arbitrary colors for the classes.

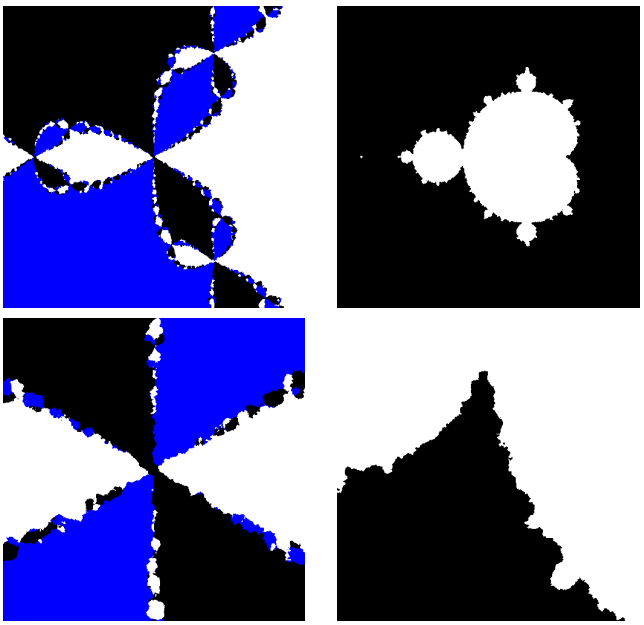
A second image was also generated, 'zooming in' on a particular region of the domain, using the same resolution as the first image. For the Newton fractal, the zoomed images represents the domain $x, y \in (-0.25, 0.25)$, for the Mandelbrot set we use $x \in (-1.0, -0.5)$ and $y \in (-0.5, 0)$.

4.1.1 Overview

×	Newton		Mandelbrot	
×	error	MDL	error	MDL
k-Nearest Neighbours	**0.033**	1.221	**0.006**	1.184
Naive Bayes	0.240	0.831	0.168	0.845
Ada Boost	0.458	0.846	0.087	0.788
C4.5 (unpruned)	**0.033**	0.644	0.009	0.356
C4.5 (pruned)	**0.033**	0.618	0.009	**0.353**
MLP, 5 nodes	0.125	0.571	0.037	0.426
MLP, 10 nodes	0.105	**0.529**	0.026	0.366
MLP, 50 nodes	0.130	0.708	0.024	0.490
MLP, 2 layers	0.152	0.713	0.026	0.366

4.1.2 k-Nearest neighbours

The classifiers were trained with the parameter $k = 7$.



Unsurprisingly, the models look good at a large scale, and deteriorate upon zooming in. For a dataset this size, KNN is a well-performing—if slow—algorithm. KNN of course generalizes very little at all. It is however one of the few algorithms that naturally maintains the fractal structure of the data.

4.1.3 Naive Bayes

This is a straightforward model, using a single one-dimensional normal distribution per feature as a posterior class distribution. Naive Bayes serves as an example of a very smooth model, with a small representation.

4.1.4 Adaboost

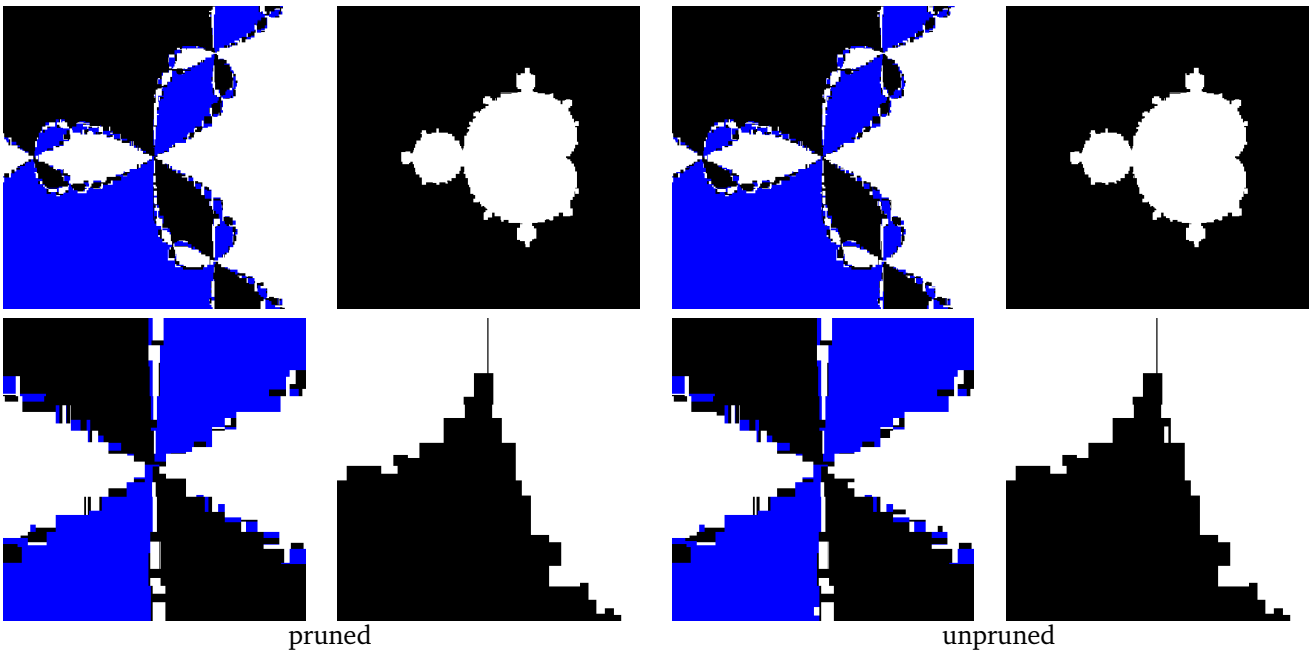
We trained a boosted classifier with decision stumps, training 25 stumps. The reweighting of the dataset performed in boosting is relevant to our problem,

as it allows regions of high complexity to receive greater attention. In other words, a boosting classifier can approximate a fractal with a succession of smooth classifiers, much like fractals are often are generated by starting with a Euclidean shape and removing, adding or modifying increasingly detailed Euclidean shapes.

Unfortunately, the simple voting mechanism used by adaboost after training makes it impossible for the classifier to be exploited in this way.

4.1.5 C4.5

The C4.5 decision tree algorithm shows a great deal of promise. While its behavior is similar to that of KNN in that it increases its model size to deal with increasing complexity, it does a better job of generalizing (in an MDL sense). We still cannot expect C4.5 to generalize across scales (ie. display detail at smaller scales that the data provides for) but the translation from flat data to a decision tree may serve as a stepping stone to new algorithms.

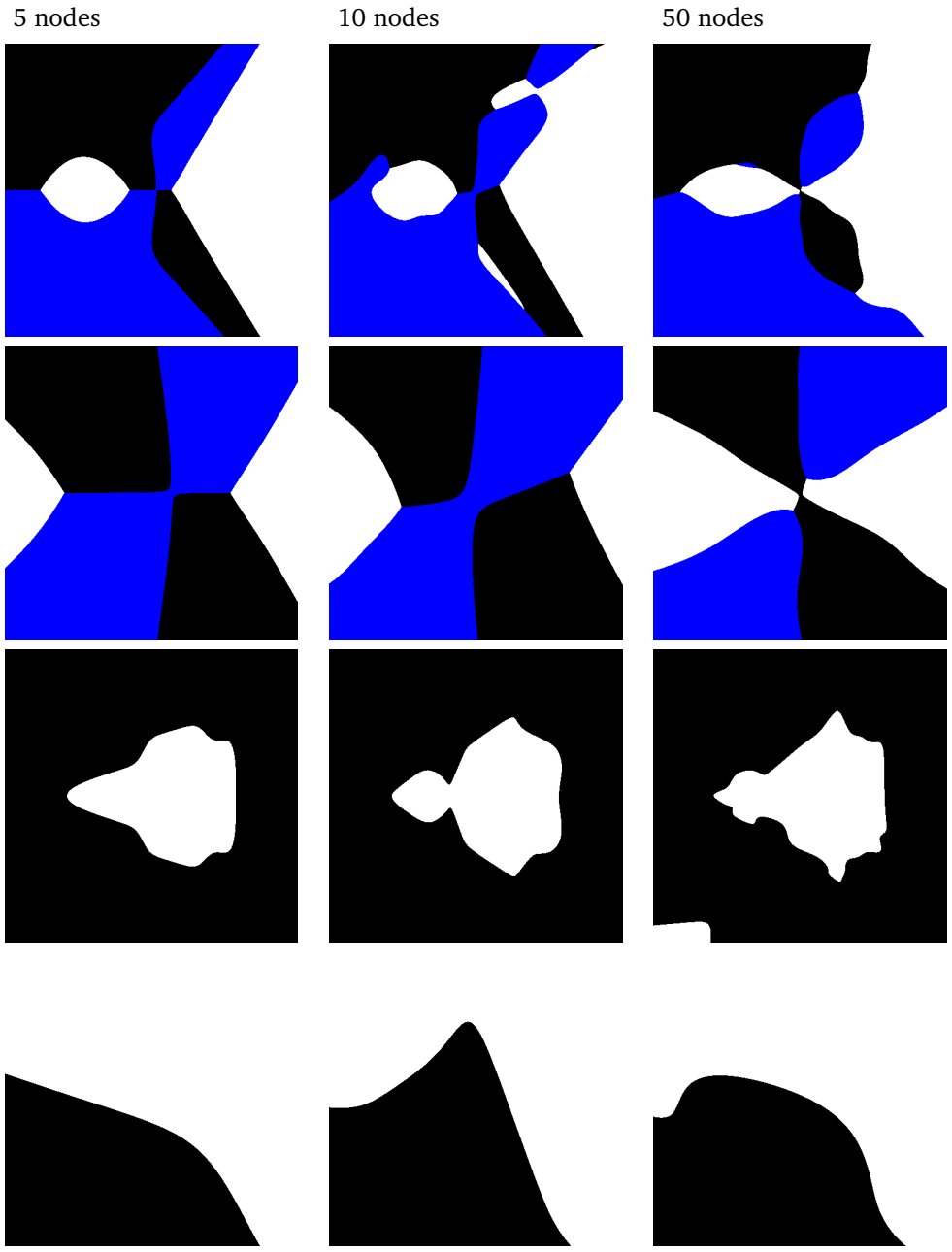


pruned unpruned

4.1.6 Neural networks

Each network has two input nodes, a single layer of 5, 10 or 50 hidden nodes and output nodes represented the target classes (two for the Mandelbrot task, three for the Newton task). All networks were trained for 10000 epochs.

The hidden layers have a sigmoidal activation function, the output layers have linear activations



4.2 Using IFS models for classification

In the previous chapter we found that we can reliably train well-fitting IFS models for a given datasets. Compared to Gaussian mixture models, these models fit well (by the Hausdorff distance fitness measure). We will first describe how to construct a simple Bayesian classifier from the rival MOG model, and then we will adapt this method for our IFS density models.

4.2.1 A Bayesian classifier with MOG models

We are faced with the following problem: we have a set X of points $x_i \in \mathbb{R}^d$, where each x_i has an associated class $c_i \in \{C_1, \ldots, C_q\}$. We want to construct a model that can predict the class for unseen points.

To build a Bayesian classifier, we want to estimate the probability of a certain class, given the point x to be classified: $p(c \mid x)$. The class for which this probability is the highest, is our class prediction $\widehat{c}$. We use Bayes' theorem to rewrite this:

$$\begin{aligned}\widehat{c} &= \arg\max_{c \in C} \; p(c \mid x) \\ \widehat{c} &= \arg\max_{c \in C} \; \frac{p(x \mid c)\, p(c)}{p(x)} \\ \widehat{c} &= \arg\max_{c \in C} \; p(x \mid c) p(c)\end{aligned}$$

This tells us what is required for our model. For each class we need a class prior probability $p(c)$ and an estimate of the probability density of the point, given that class. We can estimate the prior from the dataset (or use some other knowledge about the class distribution) and we can build a MOG model for each class to represent $p(x \mid c)$.

Now that we have a way to build a classifier, we do of course still need a way to find a set of models which fit the data well, models which represent each $p(x \mid c)$ accurately. This is discussed in section 4.2.3.

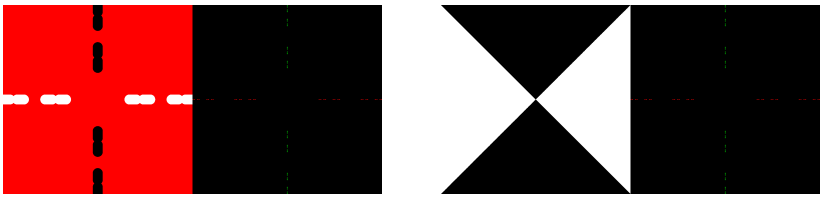
4.2.2 IFS classifier

The basic principle behind the IFS classifier is simple. We simply create a Bayesian classifier, as we did in the previous section, and we use IFS density models to approximate $p(x \mid c)$. We have seen in section 3.5.1, how to calculate this for a given IFS model. Instead of the recursive definition given there, we will define the density as a single sum.

Let $S = \langle S_1, \ldots, S_k, \rho_1, \ldots, \rho_k \rangle$ be the model that we are trying to evaluate. Let σ be a finite integer sequence $\langle \sigma_1, \ldots, \sigma_r \rangle$ with $\sigma \in (1, k)$. Let $(1, k)^r$ be the set of all such sequences of length r.

Let S^σ be the composition of the transformations indicated by the sequence σ:

$$S^\sigma(x) = (S_{\sigma_1} \circ \ldots \circ S_{\sigma_r})(x)$$



(A) (B)

FIGURE 4.1: (a) A straightforward two-class IFS classifier evaluated to depth 6, each class (colored black and white) is represented by a cantor set (green and blue on the right). The points colored red could not be classified because the probability density was too low. (b) The more robust (and faster) IFS classifier with the same models.

and let ρ_σ be the weight associated with that transformation:

$$\rho_\sigma = \prod_{\sigma_i \in \sigma} \rho_{\sigma_i}$$

Our IFS model, evaluated to depth r is now just the sum of k^r initial probability measures transformed by S^σ, each weighted by p_σ. As before, our initial probability measure is the standard Gaussian $N_d(0, I)$. We will again restrict the component transformations to affine transformations, so that the combined transformations S^σ will also be affine transformations, and that each can be represented by a translation vector t, and a transformation matrix R: $S^\sigma = \langle R_\sigma, t_\sigma \rangle$.

It can be shown that the affine transformation of the standard Gaussian by $\langle R, t \rangle$ is itself a Gaussian distribution $N(t, RR^T)$. Combining these elements, we can express the probability density of a point x under our IFS measure S as

$$p(x \mid S) = \lim_{r \to \infty} \sum_{\sigma \in (1,k)^r} \rho_\sigma N(x | t_\sigma, R_\sigma {R_\sigma}^T)$$

Which, of course, we will approximate with a finite value for r (usually $r = 5$ will suffice).

There are two problems with this method which we need to address.

The first is shown in figure 4.1. Unlike the MOG model, IFS models usually have a support that doesn't cover the entire instance space. A MOG model will assign each point a non-zero value. For most IFS models, almost all of the instance space has probability density zero. Now, this technically only holds in the limit, and for any finite approximation of an IFS model, using the standard Gaussian as an initial image, all points will have a non-zero probability density. However these probability densities very quickly become so small that in the standard 64-bit floating point representation used in most computers, they are indistinguishable from zero. This means that when we have, say, two IFS measures, each representing a posterior class probability, then points far removed from either measure's support will have probabilities so close to zero that they are indistinguishable and the classifier can't make a judgement.

The second problem is speed. For example, to evaluate the density of a single point for a model with 3 components to depth 5, the method from chapter 3

will need to evaluate $N(x \mid \mu, \Sigma)$ 243 times, and it will need to apply 364 transformations to find the parameters of the Gaussians. In the course of learning, we will need to evaluate the probability density of many points, so we need to optimize this process as much as we can.

The first thing we will do to remedy these issues, is to limit the component transformations to similitudes. As shown in section 3.3.2, this is often the best choice. The fact that it uses fewer parameters than the other models means that the ES algorithm is faster, finds good solutions quicker, and the uniform scaling it offers will allow some important simplifications in determining probability density.

We can express a similitude transformation as $S_i = \langle t, A, s \rangle$, where $t \in \mathbb{R}^d$ is a translation vector, A is a transformation matrix which only rotates and $s \in \mathbb{R}^+$ is a scaling parameter. The actual transformation can then be expressed as $S_i(x) = Asx + t$.

The two important points about using similitudes as component transformations are that, (1) a composition of two similitudes is a similitude itself, so that each S^σ is a similitude and (2) for N(0, I) transformed by a similitude we have:

$$\begin{aligned} N(t, (As)(As)^T) &= N(t, (As)(sA^T)) \\ &= N(t, AA^T s^2) \\ &= N(t, AA^T s^2) \\ &= N(t, Is^2) \end{aligned}$$

The last step follows from the fact that A is orthogonal.

We can now rewrite our classifier function for the IFS model with similitude components:

$$\begin{aligned} \widehat{c} &= \arg\max_{c \in C} \; p(x \mid c)p(c) \\ &= \arg\max_{c \in C} \; p(c) \sum_{\sigma \in (1,k)^r} N(x|t_\sigma, R_\sigma {R_\sigma}^T) \; \rho_\sigma \\ &= \arg\max_{c \in C} \; p(c) \sum N(x|t_\sigma, I{s_\sigma}^2) \; \rho_\sigma \end{aligned}$$

We can write $N(x|\cdot)$ out in full, and simplify

$$N(x \mid \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} \det(\Sigma)^{\frac{1}{2}}} \exp\left[-\frac{1}{2}(x-\mu)\Sigma^{-1}(x-\mu)^T\right]$$

$$\widehat{c} = \arg\max_{c \in C} \; p(c) \sum \frac{1}{(2\pi)^{\frac{d}{2}} {s_\sigma}^d} \exp\left[-\frac{1}{2} d(x, t_\sigma)^2 {s_\sigma}^{-2}\right] \rho_\sigma$$

Where $d(x, y)$ is the Euclidean distance between x and y (the square of which is cheaper to compute than the distance itself). We can eliminate any scalars that do not depend on the class:

$$\widehat{c} = \arg\max_{c \in C} \; p(c) \sum {s_\sigma}^{-d} \exp\left[-\frac{1}{2{s_\sigma}^2} d(x, t_\sigma)^2\right] \rho_\sigma$$

This tells us that the only elements we need to evaluate $p(x \mid c)$ are t_σ, s_σ and ρ_σ for each $\sigma \in (1,k)^r$. We have found that the most efficient way to construct this classifier is to compute and cache these three values for each σ and evaluate the above expression for each point encountered, using the cached values.

This expression goes to zero less quickly than the earlier version, but it does not solve the problem of points outside the support of all IFS models. If we encounter such a point (ie. we get $p(c|x) = 0$ for all c), we make the assumption that for these points, the influence of the scaling factors s_σ is negligible, and we set them equal to a global constant s:

$$\begin{aligned}\widehat{c} &= \arg\max_{c \in C}\ p(c) \sum s^{-d} \exp\left[-\frac{1}{2s^2} d(x, t_\sigma)^2\right] \rho_\sigma \\ &= \arg\max_{c \in C}\ p(c) \sum \exp\left[-\frac{1}{2} d(x, t_\sigma)^2\right]^{s^{-2}} \rho_\sigma\end{aligned}$$

For every σ, $\exp\left[-\frac{1}{2} d(x, t_\sigma)^2\right]^{s^{-2}} \rho_\sigma$ is a monotone, nondecreasing function in s, which means that the sum, and the sum weighted by s are monotone nondecreasing functions in s as well. As such the ordering does not depend on s, and we can choose s to suit our needs. If we set $s = \sqrt{2}$, we get

$$\widehat{c} = \arg\max_{c \in C}\ p(c) \sum \exp\left[-\frac{1}{2} d(x, t_\sigma)^2\right] \rho_\sigma$$

This function can be used to evaluate points far away from any of the class models.

4.2.3 Learning

One common way to learn a good Bayesian classifier from MOG models is to partition the dataset by class, and learn a MOG model for each point set. These MOG models can then be combined into a Bayesian classifier as described above.

For a dataset X with a set C of classes, we split the dataset X into $|C|$ subsets X^c, so that X^c contains all points that have class c: $X^c = \{x_i \mid c_i = c\}$. For each of these, we train a MOG model (either using the methods described in chapter 3 or some other method, like Expectation-Maximization). We use these to estimate $p(x \mid c)$ and we estimate $p(c)$ as

$$p(c) = \frac{|X^c|}{|X|}$$

This gives us a quick and straightforward way to build a classifier. Using the algorithms from chapter 3 to train a model for each point set, we can replace the MOG models with IFS models, and create an IFS classifier.

A second way to learn MOG and IFS classifier models is to express the whole classifier as a vector of real values, and find a good model using evolution strategies (described in chapter 3). Expressing the classifier as a vector is simple, we already know how to express the MOG and IFS models as vectors, so all we

need is |C| additional values for the class priors, which are encoded just like the component priors of the individual models (ie. they are stored as real values of any magnitude, and the absolute, normalized value is taken to build the model).

We need to change one more element of the learning algorithm to accommodate our classifier models, the fitness function. When learning classifiers, we use the symmetric error (the ratio of misclassified samples to the total number of samples). Since the classifier is still relatively slow, we use a random subsample of the dataset to evaluate a model's fitness (and as before, we re-calculate the fitness every generation).

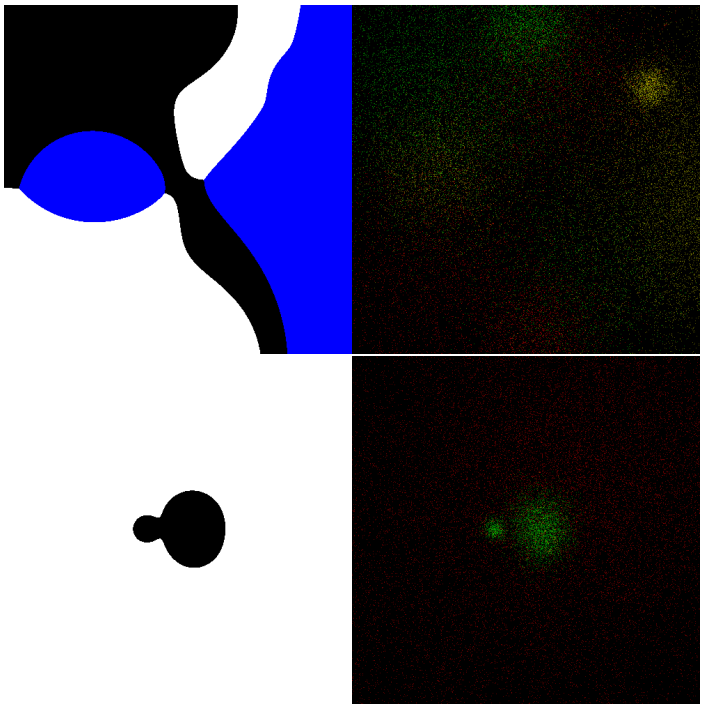
In our experiments, we combine both these methods. The first method is used to create a 'seed' model. We then learn the full classifier models based on these. In the first generation of the final learning stage all models take their parameters from the seed.

4.2.4 Results

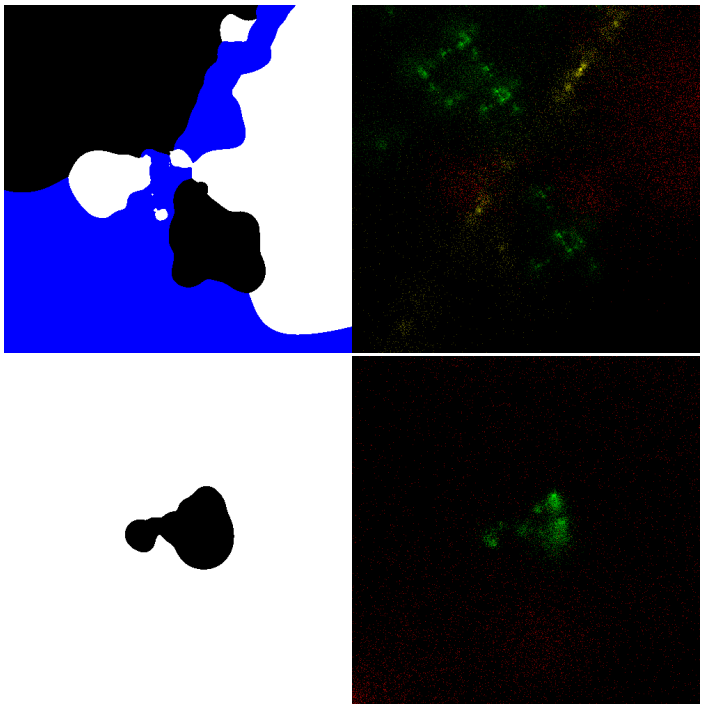
We ran the experiments from the first section on the two classifiers described above. We trained models for each class with 3 components, the seeds were trained for 5000 generations, as were the classification models. We sampled 5000 points to determine the fitness of each model. We evaluated the IFS models to depth 5. The results are shown below.

×	Newton		Mandelbrot	
×	error	MDL	error	MDL
MOG	0.219	**0**.601	**0**.009	**0**.464
IFS	**0**.161	0.663	0.0114	0.519

MOG classifier



IFS classifier



	size	dim	classes	ifs	mog	ann	knn	c45
Iris	150	4	3	0.05	**0**.03	**0**.03	0.08	**0**.03
US Income Census	65534	8	2	0.05	0.06	0.05	0.06	**0**.05
Forest Cover	65536	10	7	0.22	0.24	0.16	0.11	**0**.09
Abalone	4177	8	3	**0**.43	0.44	0.47	0.49	0.49
Shuttle	58000	9	7	0.00	0.01	0.00	0.00	**0**.00

TABLE 4.1: The results (the ratio of misclassified instances in the test set) of classification on five natural datasets. A randomly chosen section of 25% of the full dataset was withheld as a test set.

The IFS classifier does not seem to fulfill the promise of a model that can apply large scale structure at ever smaller scales (or at least not the correct structure).

Natural datasets

Table **??** shows the performance of the IFS classifier against various others. Performance is measured by the symmetric error (the simple ratio of misclassified instances in the testset to the total size of the testset). We withheld 25% of the total data as a test set. More accurate methods like cross-validation, or measurements of the variance of the error, are currently not feasible for reasons of performance. The datasets are described in appendix 2.

For the IFS and MOG models we trained models separately for each class (see section 4.2.3) for 2000 generations, after which we trained the combined model for a further 4000 generations. For the combined learning stage, we disabled the ES option of rotation in the strategy parameters to improve performance. Each IFS and MOG model (one per class) consisted of 3 components. agent fitness was determined in both stages by a sample of 500 hundred points (using the Hausdorff distance in the first stage, and the symmetric error in the second. Similitude representation was used in all experiments.

Despite the fact that the IFS classifier does not live up to our expectations of a fractal classifier, these results are encouraging. The performance shown in **??** is clearly competitive, in some cases equal to, or better than other classifiers.

There are two caveats to that conclusion. First, The competing algorithms were chosen without a great deal of care, and using more or less default parameters, while the IFS classifiers has received a great deal more attention to fine tune its parameters. Second, the performance of the IFS classifier is currently not comparable to that of the other algorithms. The experiments in the three rightmost columns (those that do not rely on evolution strategies) took a little over an hour to complete, whereas all experiments combined required around three weeks to finish.

The performance problems are mostly due to the inefficient and indirect learning method of using an evolutionary algorithm. If a more direct method of learning IFS models can be found, it should bring learning speed in line with common algorithms.

CHAPTER 5 · RANDOM FRACTALS

The final chapter of this thesis discusses random fractals, the type of fractals that we are most likely to find in 'natural' datasets. We discuss the problems that these kinds of sets introduce to both classification and density estimation tasks. We take a popular framework for describing random fractals—an extension of the IFS concept—and construct a learning algorithm for it. A learning algorithm that works not on single datasets, but on collections of datasets.

5.1 Fractal learning

Considering that we have learning tasks and datasets with fractal structure, as shown in chapter 2, it may seem a simple conclusion that a fractal learning model would be ideally suited for these tasks. But whether such models can actually be useful depends heavily on context. As an example, consider the task of learning the structure of a coastline. We wish to classify points within in a rectangular coastal area as "sea" or "land". We have our dataset of 2-dimensional coordinates, each with an associated class. The decision boundary of the perfect model would follow the coastline exactly. We will assume that our coastline is quite rough, and has strong statistical self-similarity. Our usual learners will, as we've seen, approximate the coastline as a smooth line. (Some, like decision trees, will approximate the coastline quite well, but all will be smooth at scales where the data stops providing information).

Our hypothetical fractal learner, however, would fit a line through the data and apply the structure found at large scale at smaller scales as well. But a coastline is only a statistical fractal. Unlike the perfectly self-similar fractals such as the Koch curve and the Sierpinski triangle, its small scale structure does not follow perfectly from its large scale structure. There is a random element which makes the small scale structure impossible to predict perfectly. This poses a problem

(A) (B) (C)

FIGURE 5.1: Learning the structure of a coastline: (a) The dataset, (b) A smooth approximation using a neural network (c) a fractal approximation using an IFS classifier.

when interpreting the results. When a neural network draws a smooth line to approximate the fractal coast we understand that it has 'generalized away' the fractal structure. When a fractal learner presents its best fit, it has replaced the fractal structure of the coastline with a fractal structure that resembles it. Both learners are doing the same thing, finding the best fitting model, but since the fractal learner's result doesn't look like generalization to us, we interpret the results as a far more specific prediction.

We almost expect to be able to take the model as a map, and sail into all the little fjords that the model tells us are there. A mistake we would never make with the neural network's unnaturally smooth coastline. How we should read the fractal result is not as "there is a fjord here", but as "this region has a fjord-like structure which may look something like this". Ideally, we would want a learner to present not only a crisp fractal decision boundary, but also some indication of uncertainty, telling us at what level the structure shown becomes a guess based on assumptions of self-similarity.

The reason that we don't have this problem with Euclidean learners is that as we zoom into their decision boundary, it gets closer and closer to a straight line. We can instantly equate smooth with a lack of knowledge and determine the model's uncertainty at a given level.

5.2 Information content

A concept that is central to the discussion of these and other problems, is the information content of a fractal. There is a strong difference in what we can ideally hope to learn about a natural fractal such as the distribution of ore in a region, when compared to fractals generated by a purely deterministic process, such as the Koch curve.

A concept that will aid our intuition is Kolmogorov complexity. Simply put, the Kolmogorov complexity of an object (represented by a bit-string) is the length of the shortest program that produces it on a given universal computer. [1] Since it only takes a program of small finite size size to translate a program from one universal computer to another, we speak about Kolmogorov complexity without explicitly defining the actual universal computer used. We simply accept that the value may differ by some small constant amount, depending on the choice of universal computer. It should be noted that Kolmogorov complexity is a value that cannot be computed (the statement "a has a Kolmogorov complexity of b" is undecidable). We can, however, derive very close upper and lower bounds for many situations. In this context we will not go into the technical details.

When we consider deterministic synthetic fractals like the Koch curve, the Sierpinski triangle, or even the Mandelbrot set, it's plain to see that their Kolmogorov complexity is small. They can all be defined fully by a very small computer program. We cannot say the same for fractals with a random element. Some synthetic fractals are generated with a random element and almost all natural fractals are driven by processes of such great complexity that they can

[1] A computer that can simulate any other computer correctly when provided with the right program.

be considered random for all intents and purposes. There is a caveat in both cases. A particular computer-generated random walk (a synthetic random fractal) will usually be generated using a pseudo-random number generator, giving the resulting dataset a very short generating program (including the seed of the random number generator). This means that the Kolmogorov complexity of a computer generated random fractal is in the same order as that of deterministic fractals. Similarly, it may be true that the interactions of the atoms in a cirrus cloud follow purely deterministic rules and the whole cloud can theoretically be described precisely by some relatively small starting state combined with a perfect simulator. Both points are moot, of course, because we cannot induce the seed of a proper random number generator from its output any more than we can hope to perfectly measure the initial state of a particular cloud. For this reason we will view the random element in the computer programs as truly random in the sense of Kolmogorov complexity; ie. the random component cannot be compressed by any significant amount.

The interesting point is that we can replace this random component by another like it to create a fractal of the same type. For instance, in the case of our computer generated random walk, we can replace the string of integers produced by the random number generator with another string of random numbers (by changing the seed) to produce another random walk. These are completely different fractals, but they belong to the same family of fractals, defined by the deterministic part of the generating algorithm. This deterministic program functions as a kind of *sufficient statistic*. It represents everything we can reasonably hope to learn about our fractal dataset.

An interesting example of a random fractal in this respect is the chaos game method of generating the Sierpinski triangle. The generating algorithm has a strong random component, and as the algorithm begins to produce points, the dataset remains relatively uncompressable. However, in the limit the algorithm generates a simple deterministic fractal with a very small Kolmogorov complexity. In fact, if we use a discrete space of points (such as the floating point values used in real-world computers) and we delete points that occur more than once in the dataset, so that the limit set becomes finite, there will be a point during the generation of the dataset where the most efficient way of describing the dataset switches from a straightforward description of the generated points to a description of the Sierpinski triangle and the set of points that haven't been generated yet.

The same can be said for a discrete random walk on the line or in the plane. We know that in the limit, these processes fill their respective embedding spaces. For the three-dimensional random walk, this doesn't hold anymore. (Schroeder, 1996) In the limit the 3D random walk produces a different result for any random string used to drive it. When we plot the Kolmogorov complexity of a process generating the Sierpinski triangle as a function of time, we will see a 'hump' as the complexity increases and then returns to a small value. In the case of a 3D random walk, the complexity only increases.

5.3 Random iterated function systems

One way of dealing with the problems described above, is to learn a family of fractals, rather than a single probability distribution or set. A framework for defining random fractals in this way is described in (Hutchinson, 2000) and related publications. As before, we describe the basic model and the main result and refer the reader to the citations to find a more technical treatment (including proofs).

5.3.1 RIFS sets

In the following it is helpful to define a *random fractal* not as a single set, but as a probability distribution over compact Euclidean sets. By the same token, we refer to a distribution over sets as a random set.

We define a random set $\mathcal{E}$ as a probability distribution over compact[2] subsets of $\mathbb{R}^d$. Let $E = \langle E_1, \ldots, E_k \rangle$ represent k independent draws from $\mathcal{E}$.

Let S_i again be a simple function from $\mathbb{R}^d$ to itself: $S_i : \mathbb{R}^d \to \mathbb{R}^d$. Let $\mathcal{S}$ be a probability distribution over all k-tuples $\langle S_1, \ldots, S_k \rangle$ of functions $S_i : \mathbb{R}^d \to \mathbb{R}^d$. Note that the S_i are not drawn individually or independently (as the E_i); rather the distribution is over complete k-tuples.

Consider now the set

$$\bigcup_{i \in (1,k)} S_i(E_i)$$

Because we've selected E_i and S_i through a random process from given distributions $\mathcal{S}$ and $\mathcal{E}$, this union represents another set drawn from a random distribution. We will denote this distribution by

$$\mathcal{S}(\mathcal{E})$$

It is of particular importance to note that when $\mathcal{S}(\cdot)$ is constructed, we must draw k independent sets from the argument to construct a new set analogous to E.

$\mathcal{S}$ functions as a random scaling law on random sets. We say that a random set $\mathcal{K}$ satisfies the scaling law $\mathcal{S}$ iff

$$\mathcal{K} = \mathcal{S}(\mathcal{K})$$

In other words, we have a probability distribution on compact sets $\mathcal{K}$. If we choose k compact sets according to this distribution and transform each by an S_i from a k-tuple chosen from $\mathcal{S}$, then we get a new set $S(K)$. If the probability distribution over these resulting sets is equal to $\mathcal{K}$, then we say that $\mathcal{K}$ satisfies the random scaling law $\mathcal{S}$.

The basic properties of deterministic IFS models hold for random IFS models as well. There is a unique random set $\mathcal{K}$ that satisfies a given scaling law. [3]

[2] Closed and bounded

[3] So long as every S_i in any tuple resulting from $\mathcal{S}$ satisfies the open set condition.

As with the deterministic IFS models, we can iterate the scaling transformation to approximate $\mathcal{K}$. As noted before, generating $\mathcal{S}(\mathcal{E})$ requires k independent draws from $\mathcal{E}$. This means that when we iterate $\mathcal{S}$ once, ie. we generate a random draw from

$$\mathcal{S}^2(\mathcal{E}) = \mathcal{S}(\mathcal{S}(\mathcal{E}))$$

we must generate k *independent* draws from $S(E)$ (and thus k^2 independent draws from $\mathcal{E}$). As with the deterministic case it has been shown that

$$\lim_{m\to\infty} \mathcal{S}^m(\mathcal{E}) = \mathcal{K}$$

for any distribution $\mathcal{E}$ over compact subsets of $\mathbb{R}^d$.

5.3.2 RIFS measures

Let a random measure $\mathcal{V}$ be a probability distribution over the set of all compactly supported [4] measures on $\mathbb{R}^d$.

Since we will treat these measures as probability measures, things can get complicated very quickly. To clarify, let $\mathcal{P}$ be a probability distribution over the set of compactly supported probability measures on $\mathbb{R}^d$. From $\mathcal{P}$, we can choose a single probability distribution p. From p we can then draw a set of random points in $\mathbb{R}^d$.

As noted in chapter 3, we can apply a transformation S_i to a measure. Let $V_1, \cdots, V_k$ be a set of measures drawn independently from $\mathcal{V}$. Define $\mathcal{S}$ as a probability distribution over 2k-tuples $\langle S_1, \cdots, S_k, \rho_1, \cdots, \rho_k \rangle$, where S_i are functions as we have used them and ρ_i are positive real values that sum to one, representing the probability of each S_i. Let $S = \langle S_1, \cdots, S_k, \rho_1, \cdots, \rho_k \rangle$ be one such tuple drawn according to $\mathcal{S}$.

Applying the scaling law to the randomly drawn measures as follows

$$\sum_{i=1}^{k} \rho_i S_i(V_i)$$

determines a new random measure

$$\mathcal{S}(\mathcal{D})$$

The fundamental properties of iterated function systems have again been proved for random measures. For a given random scaling law $\mathcal{S}$, there is a single random measure $\mathcal{K}$ which satisfies it. Repeated application of the scaling transformation to some initial random measure, will converge to $\mathcal{K}$.

Section C.5 describes an algorithm for drawing a set of random points from a random probability measure from a given $\mathcal{S}$, which may help to elucidate the concepts described here.

[4] That is, measures whose support is a compact set.

The mean measure

Drawing a single random point from a single instance of a random measure defines a probability distribution that is a weighted mixture of all the probability distribution that the random IFS can generate. We will call this the *mean measure*. To generate random points from the mean measure to depth r, we can simply start with a point x_0 chosen randomly from the initial distribution, choose a random scaling operator S^1 from the the RIFS (according the distribution over scaling operators that defines the RIFS), choose from it a random transformation S^1_i. We can then define $x_1 = S^1_i(x_0)$ and repeat the process until we get x_r.

If we have a RIFS that only accords a discrete number of discrete scaling operators a non-zero probability, the mean instance is itself a discrete IFS. For each scaling operator $S^\nu = (S^\nu_1, \ldots, S^\nu_k, \rho^\nu_1, \ldots, \rho^\nu_k)$, with probability $p(\nu)$, the probability that transformation S^ν_i is chosen in the generation of the mean instance is $\rho^\nu_i p(\nu)$.

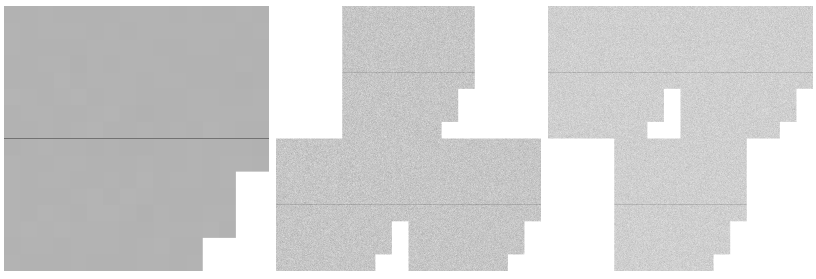
This also tells us that we can generate points in the mean instance using the chaos game method by repeatedly applying randomly chosen transformations.

5.3.3 Examples

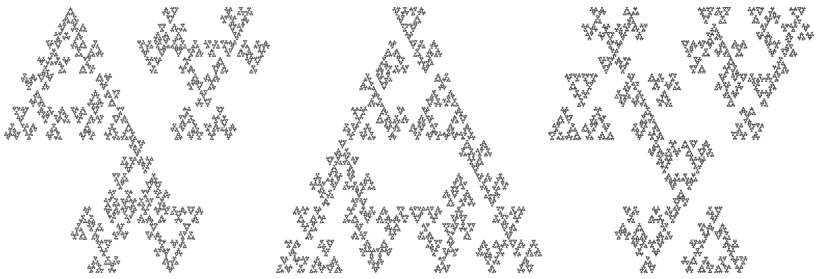
Sierpinski triangles

At this point, it should be helpful to illustrate the concept of Random Iterated Function Systems, and their expressive power with some examples.

We begin with a very simple example. For this example we use two deterministic fractals, the first is the Sierpinski triangle pointing upwards, the second is the same, but pointing down. The following images show the transformations that make up the two deterministic IFS models. The image on the left shows the initial distribution (shaped so that the transformations are easy to identify).

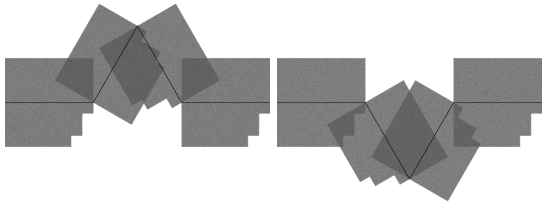


We define the random scaling law $\mathcal{S}$ as a choice between these two with equal probability. The following images show points drawn from three instance of $\mathcal{S}$.

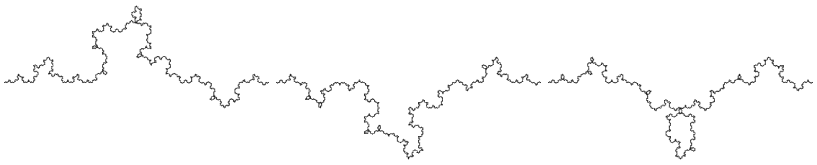


Koch curves

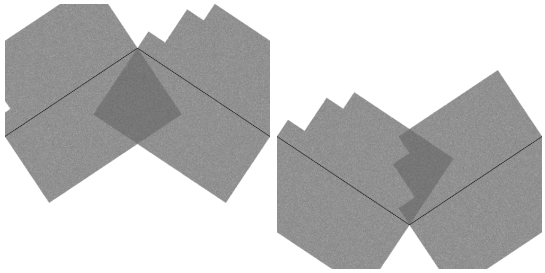
For this RIFS we use the Koch curve as a basis. As with the previous example, we take the regular version, and an upside-down version, and define $\mathcal{S}$ as a choice between the two with equal probability.



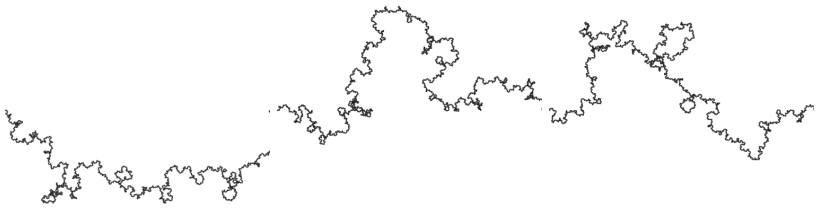
The following are three instances of this random fractal. here, we begin to see how random iterated function systems can mimic natural, organic shapes.



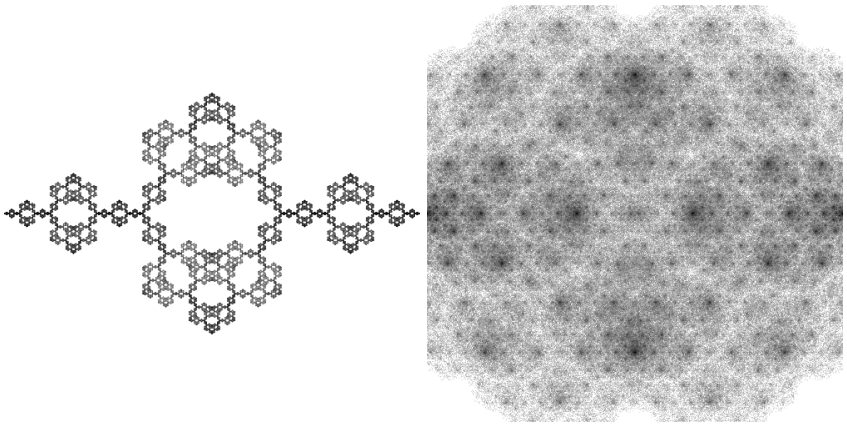
An interesting observation can be made when we use a different, but equivalent construction for the model's components (note the rotations).



Using two components instead of four, these constructions result in the Koch curve as well. In regular IFS terms, they are equivalent to the 4-component constructions shown above. However, when we use them to construct a random fractal, the result is considerably different.



The difference becomes especially clear when we study the mean measures of both models (the four component model is on the left).



Coast lines

The examples of the Koch curve shown above show a natural structure that is reminiscent of coast lines. To pursue this notion further, we can change our model from a distribution over a discrete set of IFS models to a continuous distribution.

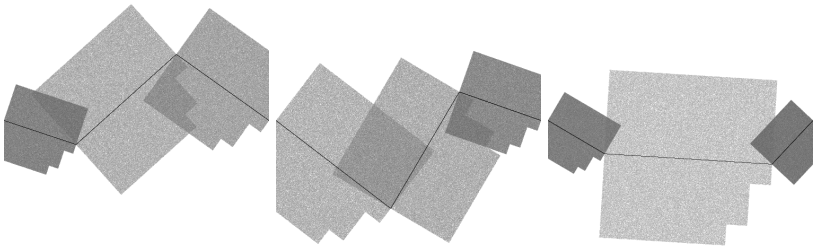
We define our distribution over IFS models as follows

- Choose two random points $x \sim N\left[(-\frac{1}{3}, 0), \sigma\right]$ and $y \sim N\left[(\frac{1}{3}, 0), \sigma\right]$.
- Define the line segments u as $(-1, 0)$ to x, v as x to y and w as y to $(1, \sigma)$.
- Define the line segment a as $(-1, 0)$ to $(1, 0)$
- Find the transformations U, V and W consisting of rotation, scaling and translation, so that U maps a to u and V maps a to v and W maps a to w.

- let $\mathcal{S} = \langle U, V, W, \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \rangle$

This procedure defines a probability distribution over 3-component IFS models, and thus, a random iterated function system.

The following are three IFS models returned by this model for $\sigma = 0.3$.



Which leads to measures that form a continuous line (since every line segment is always replaced by three segments which connect the previous corner points. The following images are three instances:

Interestingly, when we increase σ the measure loses its line-like character. Figure **??** shows the progression as σ increases. The random number generator used was reset to the same seed for each image generated, so that the instances drawn are the same for each image, but with increasingly greater variance.

Random graphs

We can achieve a different class of images by subtly changing the method described above.

- We choose a point x with $x_1 = 0.5$, and $x_2 \sim N(0, \sigma)$
- We again define the line segments u as $(-1, 0)$ to x and v x to $(1, \sigma)$.
- As before, a is $(-1, 0)$ to $(1, 0)$
- We now find the transformations, using *only scaling, translation and shearing* so that U maps a to u and V maps a to v.
- Let $\mathcal{S} = \langle U, V, \frac{1}{2}, \frac{1}{2} \rangle$

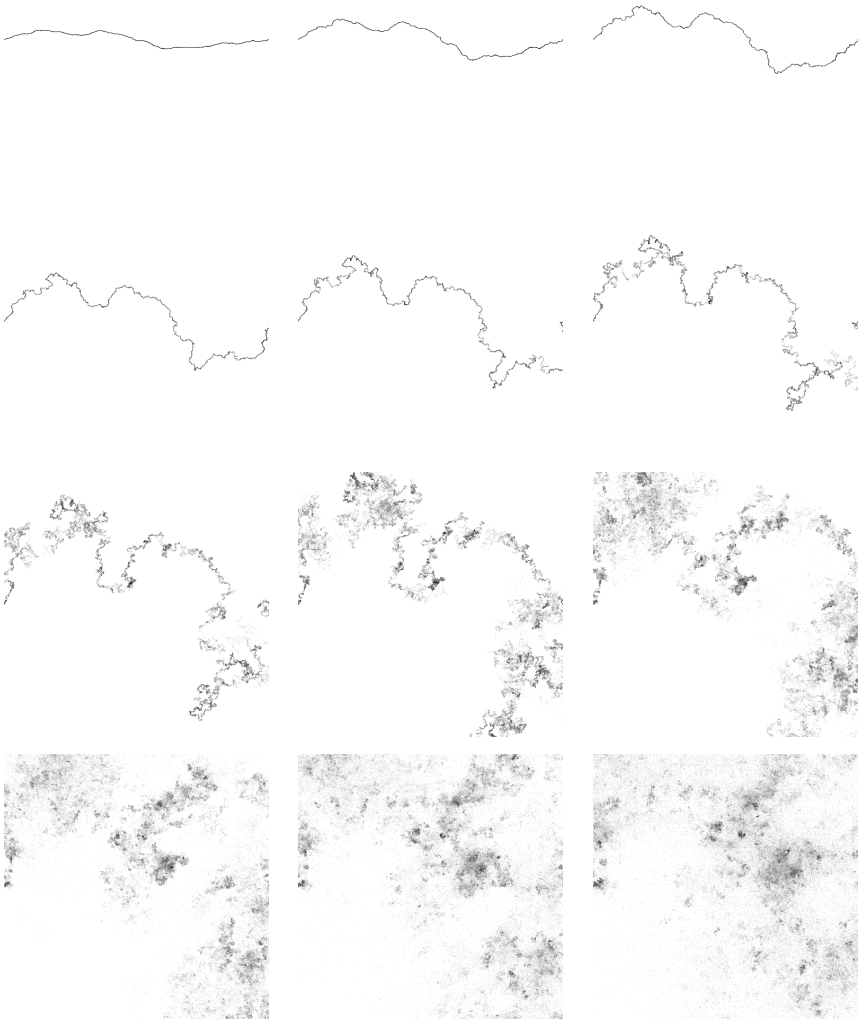
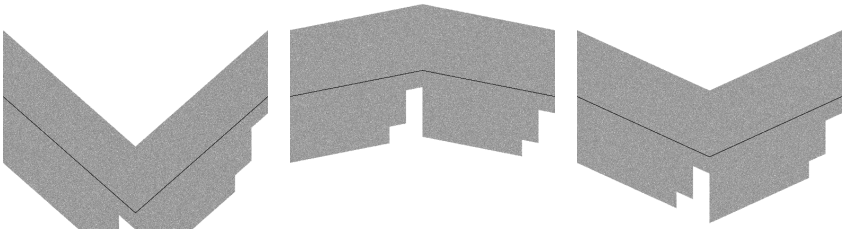
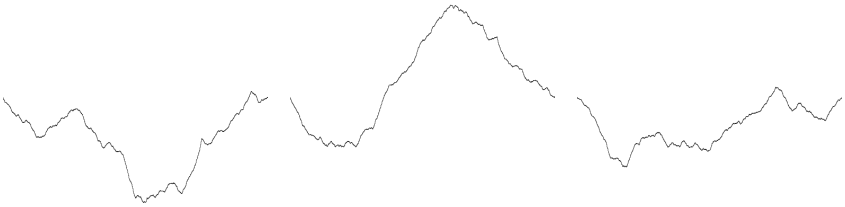


FIGURE 5.2: The same instance of the 'coastline' RIFS for increasing values of σ. The last image has $\sigma = 0.9$. The other images are equally spaced between 0 and 0.9.

This produces IFS components such as these (for $\sigma = 0.6$)



Which leads to the following RIFS instances

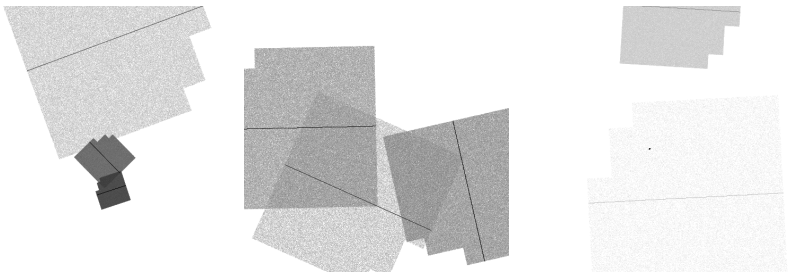


The are quite reminiscent of the persistent timeseries of chapter 2. The only difference is that these are necessarily *bridges*, ie. their endpoints are fixed.

Gaussian models

In chapter 3, we used several representations of IFS models as vectors in a space $\mathbb{R}^m$ for some m. We can define a Gaussian probability distribution $N_m(\mu, I\sigma)$ over this space to draw random discrete IFS models. This approach defines a RIFS model for each value of σ.

This produces IFS components such as these (for $\sigma = 0.45$, with three discrete IFS models, with three components per model, and using the similitude representation)



Which leads to the following RIFS instances[5]

[5]These triplets do not represent a fair random choice. They were chosen as the three most visually interesting from a fair set of 10 random instances.

Figure **??**, shows instances for increasing values of σ.

5.3.4 Learning RIFS measures

We want to be able to learn RIFS measures. We will be learning from a set Ψ of sets X_i of points. The points in set X_i are assumed to be drawn independently from a single instance of a RIFS measure (which we will attempt to approximate).

As in chapter 3, we require three elements to construct a learning method. A representation for a hypothesis, a fitness function and a learning algorithm. For our learning algorithm we will use evolution strategies again, exactly as described in chapter 3.

To define a fitness function for these models, we use two particularly helpful properties of the Hausdorff distance—our fitness function in the experiments for learning regular IFS models. Firstly, that it is defined on point sets in metric spaces, rather than just Euclidean spaces. Secondly, that it is itself a metric. This means that the Hausdorff distance defines a metric space over sets of points. For instance, in our earlier tests in chapter 3, it was used to define a metric space of point sets in $\mathbb{R}^d$.

Combining these two aspects we can use the Hausdorff distance on itself, that is, define a distance between sets of point sets. Consider the task of finding a random IFS that accurately models coastlines. We've seen that a random simple IFS model can output things that look like coast lines, so the natural question is, can we feed a learning algorithm a set of coastlines and find a random IFS model that produces similar results. In this scenario, each set $X_i \in \Psi$ is a set of random points on a given coastline, with a different coastline for each X_i. We will try to learn a RIFS $\mathcal{S}$ that fits Ψ well. To determine the fitness of a given $\mathcal{S}$ we generate a set of sets $\Lambda = Y_1, \ldots, Y_w$. We define the distance between two instances as $d_H(X_i, Y_i)$ and we will define the distance between the RIFS $\mathcal{S}$ and the data X_i as $d_H(\Psi, \Lambda)$ with $d_H(X_i, Y_i)$ as a metric.

Using this fitness function, the rest of the algorithm is relatively trivial. We will focus on learning random IFS models that consist of a finite number of t discrete IFS models, each with k transformations. The transformations can be represented—using the TSR representation—in $2n + \frac{n^2-n}{2}$ real values. We define a prior probability for each transformation, and for each discrete IFS, so a random IFS model can be described in $t(k(2n + \frac{n^2-n}{2} + 1) + 1)$ real values.

With this description and the fitness function, we can use evolution strategies

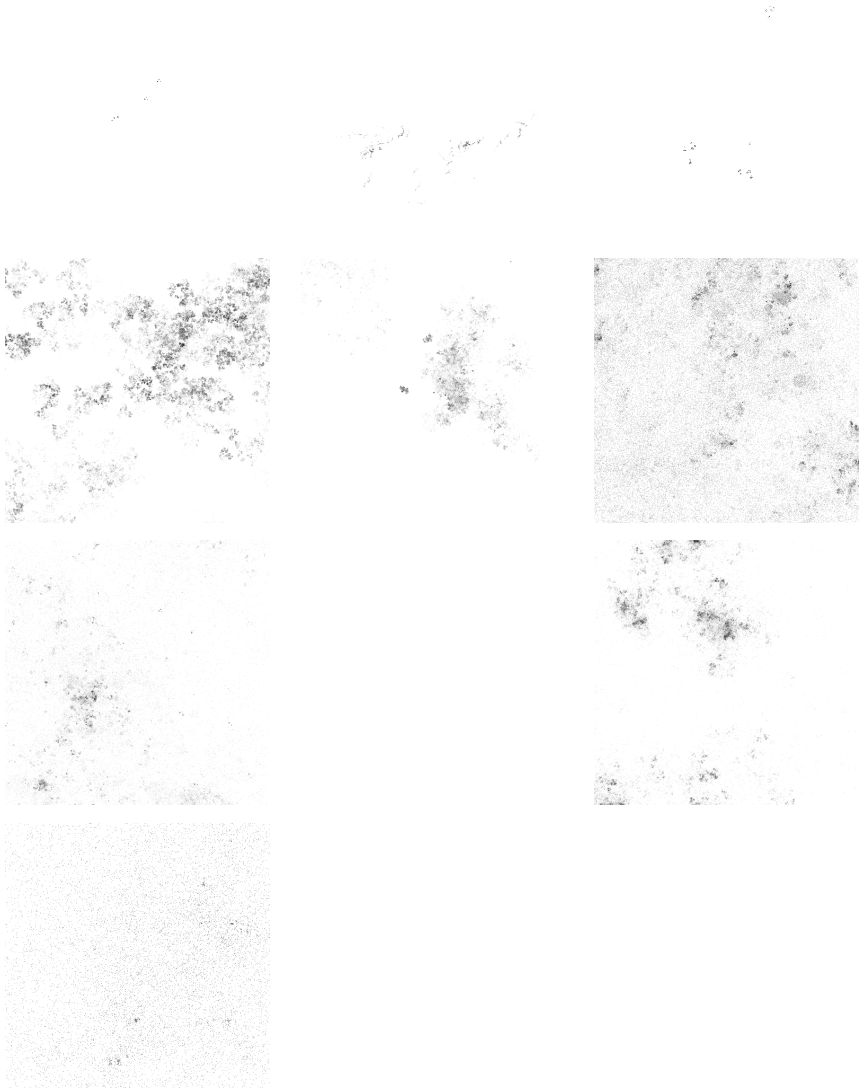


FIGURE 5.3: The same instance of the 'Gaussian' RIFS for increasing values of σ. The last image has $\sigma = 1.5$. The other images are equally spaced between 0 and 1.5. Some of the examples for larger σ are empty, likely because the transformations drawn were not, on average, contractive.

to find a model that fits the dataset well.

5.3.5 Results

The main issue with this approach is its computational complexity. Calculating the Hausdorff distance between point sets is already expensive and calculating the Hausdorff distance between sets of point sets adds a a factor of q^2 to that, where q is the number of point sets in the dataset. Because of this constraint we will limit ourselves to simple two dimensional datasets.

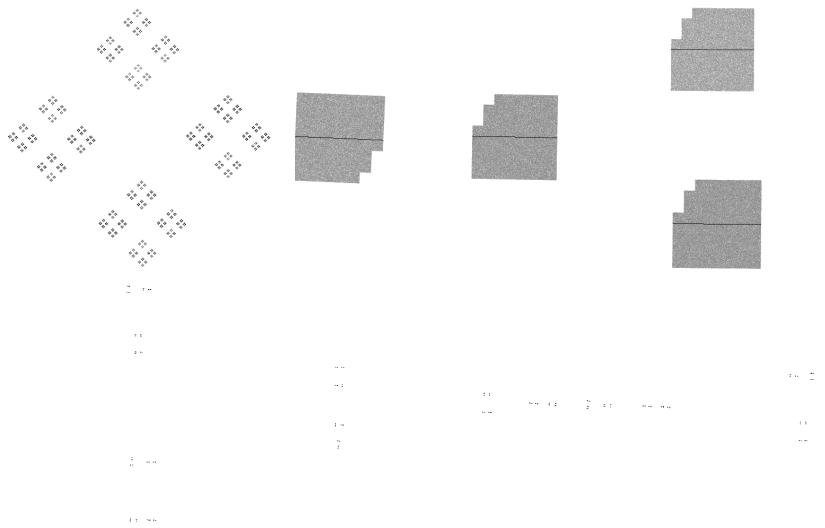
Since this is a new learning task, we cannot really compare the result numerically to any existing algorithm. We will simply limit ourselves to visual inspection of the resulting models to show that, in principle, the method works.

Cantor RIFS

The first RIFS model we attempt to reconstruct consists of two discrete IFS models with equal probability. One cantor set arranged along the horizontal axis and one along the vertical, both between -1 and 1.

The following images show the IFS models, the mean instance and three random instances.

The datasets consisted of three hundred instances with 10000 point drawn from each. The fitness of each agent was determined (once per generation) on a random sample of 50 datasets and 50 points per dataset. After 400 generations, the following model emerged.

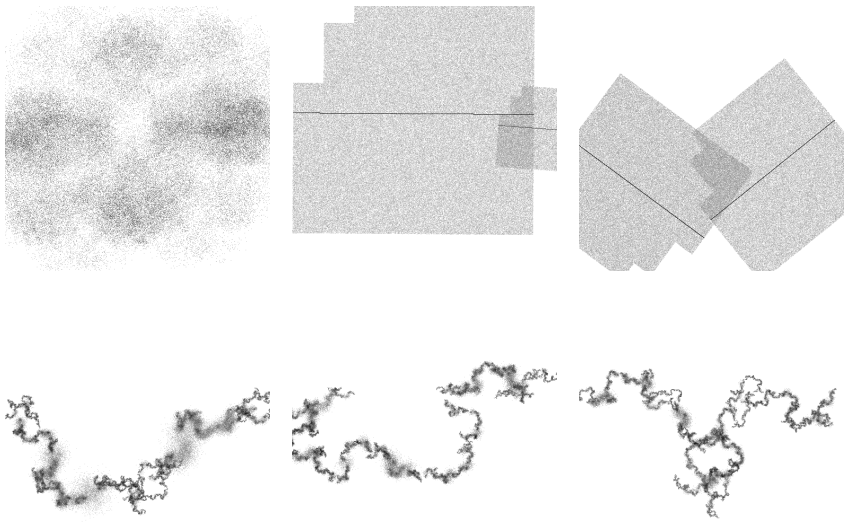


The population size was 25, with 25 additional children generated each generation. Other parameters were chosen as in C.7.

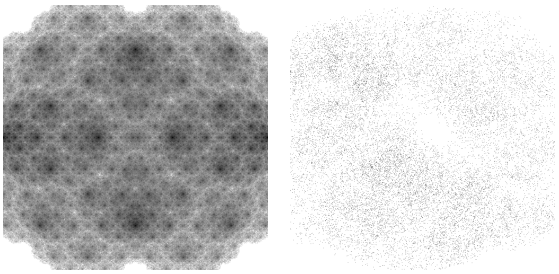
Koch RIFS

The learning task already increases in difficulty without increasing the dimensionality, the number of models or the number of components per model. Training with the same parameters as the previous experiment, we attempt to learn the model from section 5.3.3.

At generation 2000, we get the following results (from left to right, the mean measure, the two component models and three random instances):

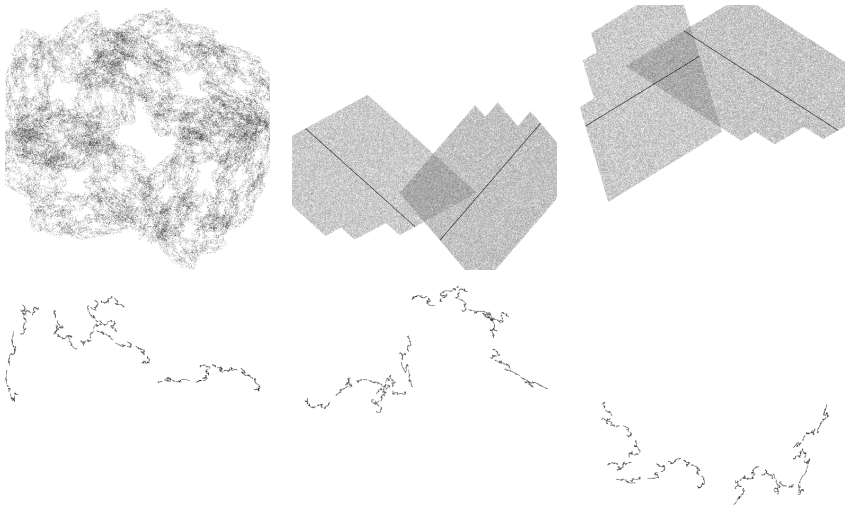


One method we can use to improve performance is to seed the model. We train a regular discrete 4 component model on the mean measure, using the methods from chapter 3. In our case we can generate the mean measure directly from the model we are trying to learn. In the case of natural data—where would would only have several point sets representing random instances—we would take a sample of points from all datasets to approximate a sample from the mean measure (remember, the mean measure dictates the probability of a point averaged over all datasets). After training the seed for 2000 generations, we found the following model (the target mean model is shown on the right):



We now create the initial population for our RIFS learning experiment from this final model. Each agent in the initial population requires 4 four components

(two models, two components per model), which are chosen at random with replacement from the four components of the final model. Using the same parameters as in the previous experiment, we achieve the following result after 2000 generations.



Discussion

We have achieved a measure of success in learning very simple examples of random fractals. Clearly, the method is not ready to tackle natural datasets yet, but these results show, that the basic principles work. In particular, the notion of applying the Hausdorff distance twice, is in principle successful as a fitness function. The major drawbacks of the method are the complexity of the fitness function: $O(q^2r^2)$, where q is the number of point sets tested and r is the number of points per set. Our experiments suggest varying q and r does not change the performance very much, if the value q^2r^2 is kept fixed (in other words we can't use clever choices for q and r to improve performance).

As with the previous two chapter, hope for improvement lies in three elements. Optimization should be able to improve the range of this method very quickly. While the core methods were implemented as efficiently as possible, our implementation as a whole lacked any sort of parallelization. The calculation of Hausdorff distance should be very simple to parallelize and there are many algorithms available to parallelize Evolution strategies. Furthermore, the version of ES used in our implementation is somewhat dated, and later versions are likely to improve performance and reduce the number of parameters. Finally there is always the hope that future hardware improvements will bring a greater number of learning tasks within the range of this method.

Another option is to reduce the complexity of the models. Instead of searching the space of all models consisting of all possible affine transformations, we can make the simplifying assumption that our data fits the type of model described in section 5.3.3. Under this assumption we would only need to fin the parameter

σ that optimizes the fitness function.

Another way of improving the method would be to restrict the data to two dimensional images and define the distance between two point sets as the mean squared error (or something similar), while maintaining the use of Hausdorff dimension for the distance between sets of sets. This approach may lead to a more accurate (and possibly faster) fitness function. The downside is that it doesn't scale well to higher dimensions.

If these improvements do indeed increase the range of this method, datasets like the following might be analyzed using RIFS models

- Financial timeseries may be an early contender. Their low dimensionality make the complexity of the task a likely contender to work without a great deal of improvement to our method. The only drawback is that we need to allow shearing in our family of transformations, which only allows the simple representation (see section 3.2.2).
- In a similar vein, we might attempt to analyze EEG recordings and other timeseries data (see also the references at the end of section 2.2).
- Snowflakes. The crystallizing process behind snowflakes generates a structure that is not perfectly fractal, but can likely be decribed well by a fractal model. We only require two dimensions, but the the sixfold symmetry present in almost all snowflakes suggest a requirement of at least 6 components. Our initial tests to performs this task yielded no usable results.
- Bacterial colonies. Bacteria are often analyzed by taking swabs of some object and cultivating the swab in a petri dish. When bacteria are present in the original swab, they will grow out into a colony. The size of the colony indicates the number of bacteria on the original swab, but the structure of the colony can be an indicator of the type of bacteria present. RIFS models may be a helpful way to analyze images of bacterial colonies automatically.
- Clouds are a typical example of fractals. Using satellite data it's possible to construct 3d images of clouds.(Wittenbrink *et al.*, 1996) If these methods can be transformed to generate point sets, the resultant data can be modeled as random fractals.
- Elevation data and fracture surfaces. Modern technology can give us elevation data from scales as large as mountain ranges to micrometer-scale images of fracture surfaces. While these are not point sets as we have used, they can be seen as an elevation function, and thus as a measure, which means they can be modeled as random fractals.

So far we have only attempted to learn simple RIFS models of synthetic datasets and inspected the results visually. This suggests that a model trained on a dataset of, for instance, snowflakes, can give us a fractal based simulation of snowflakes generation that may lead to new insight in the original data. while interesting, this has limited applicability (snowflake formation, for instance,

is already very well understood). However, once we have found that we can successfully model our phenomenon with random fractals, many new options become available.

As an example, we might build a classifier by training one RIFS model on cumulus clouds and another RIFS model on cirrus clouds. Using the Hausdorff distance, we can then classify unseen clouds based on the distance to data generated by these models.

Even when the data cannot be successfully modeled by a random fractal model, we can generalize our assumption to say that we assume the data to be generated by some random measure. Say for instance that we have a large collection of EEG measurements. Without training a random fractal model, we can still apply the Hausdorff distance, and use a host of clustering algorithms on the resulting distance matrix.

In short, the combined notions of random measures and Hausdorff distance suggests a new area of learning tasks, where RIFS models are a natural method of modeling.

CONCLUSIONS

The primary goal of this thesis was to show that fractal geometry could play an important part in the field of machine learning.

First, we investigated the fractal structure already present in naturally occurring datasets. Chapter 2 shows, first and foremost, that there are many well-established methods to measure fractal dimension—the most important qualifier of fractal structure. Our results show that few datasets have an intrinsic dimension that is close to their embedding dimension. Furthermore, in the limited sample of datasets that we tested, there did not seem to be any bias for data with an integer-valued dimension, suggesting that fractal data may be the rule rather than the exception. Current methods of measuring dimension do not have sufficient accuracy to test these suggestions more rigorously. A short description of relevant research in time-series analysis shows that the idea of self similarity in natural data is by no means novel or controversial.

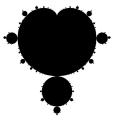
Our second claim was that modern learning algorithms were Euclidean in nature, and could therefore not adequately deal with fractal data. The first half of chapter 4 provides visual evidence to this effect. An important point to remember when analyzing these results is that this behavior is often more of a blessing than a curse. In many situations, we expect the concept under study to be a Euclidean one, complicated with fractal noise. In these situations, the Euclidean learners help us to generalize the noise away. The problem is that this assumption has so far been implicit. By making it explicit, we can begin to investigate exactly when it is appropriate to see fractal structure as noise and Euclidean structure as meaningful, and when these assumptions break down.

Most of our efforts were devoted to investigating possibilities for fractal modeling. Our attempts in chapter 3 show that there are fundamental problems when using IFS models (and likely all families of fractals) as probability models. These problems can be overcome by slightly augmenting the approach, specifically by using a different fitness function. Under this fitness function, IFS models perform well compared to mixture of Gaussian models. When we use these modified probability models to construct a classifier (chapter 4), they clearly show competitive behavior when compared with common machine learning methods, at the price of a far greater learning time. Since this is a very new method with only a basic implementation, there is every possibility that new insights, optimization and hardware improvements can make the method a viable option for approaching certain real-world problems.

Our final chapter discusses the random nature of natural fractals. This poses new problems not adequately addressed by the deterministic fractal models. Using the random iterated function system model, this leads naturally to a new class of learning tasks, involving distributions over multiple probability measures. We provide a few modest results in conquering such learning tasks, with the hope that these can be expanded to make the method viable for real-world learning tasks.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Pieter Adriaans for the many interesting discussions and insightful advice, without which this thesis wouldn't be half as interesting. My thanks also go out to the members of the defense committee, specifically Maarten van Someren for his efforts in guiding this thesis to completion.



APPENDIX A · COMMON SYMBOLS

$[\cdots]$: When the argument is a logical expression, these are Iverson brackets, representing 1 if the argument is true, and 0 otherwise. If the argument is numeric, square brackets are simply synonymous with parentheses: $(\cdots)$.

$\lambda(\cdot)$: The Lebesgue measure. Generalization of length, surface, volume, etc.

$d(a, b)$: The distance between points a and b.

$B_\epsilon(x)$: A ball of radius ϵ centered on point x: $B_\epsilon(x) = \{y : \text{distance}(x, y) \leqslant \epsilon\}$.

B_ϵ : Shorthand for $B_\epsilon(0)$, a ball of radius ϵ centered on the origin.

$\sim$: Used to denote that a random variable has a particular distribution, eg. $X \sim U(0, 1)$ means that X is uniformly distributed between 0 and 1. In the context of pseudocode, it can also mean that a variable's value is drawn from a particular distribution.

$U(a, b)$, $U(x|a, b)$: A uniform distribution over the interval $[a, b]$.

$N(\mu, \sigma)$, $N(x|\mu, \sigma)$: The normal distribution with mean μ and standard deviation σ.

$N_d(\mu, \Sigma)$, $N_d(x|\mu, \Sigma)$: A d-dimensional normal distribution with mean μ and covariance matrix Σ. The subscript may be dropped when the dimensionality is apparent form the context.

I_d : The d by d identity matrix. The subscript may be dropped when the dimensionality is apparent form the context.

$|x|$: The length (norm) of vector x. If the argument is a set, this represents its cardinality.

diam(A) ; the diameter of a set A. The maximum distance between any two points in A: $\max\{d(a, b) \mid a, b \in A\}$

diag(M) : A vector representing the diagonal entries of the (square) matrix M: $v_i = M_{ii}$.

diag(v) : A matrix M, whose diagonal entries have are taken from vector v ($M_{ii} = v_i$) with all other entries 0.

$\min A$, $\max A$: Respectively the smallest and largest elements in the set A (assuming that all elements in A are comparable by some natural ordering).

$\min(x_1, x_2, \cdots, x_n)$, $\max(x_1, x_2, \cdots, x_n)$: Respectively the smallest and largest values of the input arguments: $\min\{x_1, x_2, \cdots, x_n\}$.

$\min_{x \in X} f(x)$: The smallest of all the values obtained by applying the elements of X to the function $f(\cdot)$: $\min\{f(x) \mid x \in X\}$.

$\max_{x \in X} f(x)$: Analogous to $\min_{x \in X} f(x)$

D_q The dimension of a probability distribution, where q is a weighting parameter.

D_0, D_B The box counting dimension.

D_1, D_I The information dimension.

D_2, D_C The correlation dimension.

$p(A)$ The probability of a set $A \subset \mathbb{R}^d$.

$p(x)$ The probability density of a point $x \in \mathbb{R}^d$

S, S_i, ρ_i The elements that describe a scaling law. A scaling law for for sets is a k-tuple $S = \langle S_1, \ldots, S_k \rangle$ of maps $S_i : \mathbb{R}^d \to \mathbb{R}^d$. A scaling law for sets is a 2k-tuple $S = \langle S_1, \ldots, S_k, \rho_1, \ldots, \rho_k \rangle$ with S_i as before, and $\rho_i \in (0, 1), \mathbb{R}$, with $\sum \rho_i = 1$.

iid Independent identically distributed

iff If and only if

APPENDIX B · DATASETS

B.1 Point sets

B.1.1 Synthetic datasets

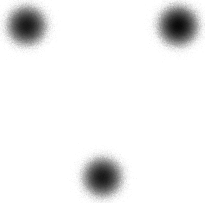


FIGURE B.1: The dataset 'three'; a simple mixture of Gaussians distribution.

The following are datasets generated from simple algorithms. The advantage of these datasets is that they are free of noise, and many properties, such as fractal dimension, have been analytically established. We will mainly use these to determine the behavior of many learning algorithms before moving on to real-world learning tasks.

In some of the images in this chapter, the levels have been manipulated to bring out the detail.

Three

A very simple, specifically non-fractal dataset. It contains random points, generated from a 2D Gaussian mixture model, with three components, each symmetric with equal variance, and arranged in a triangle.

IFS sets

Simple IFS fractals. Specifically, the Sierpinski triangle, the Koch curve and the Cantor set.

B.1.2 Natural datasets

Population density

The distribution of people across a stretch of land follows a kind of fractal generating process, whereby people are both attracted to and repelled by centers of population.

The dataset used is the Gridded Population of the World, version three available from `http://sedac.ciesin.columbia.edu/gpw`. This dataset was produced by the Center for International Earth Science Information Network (CIESIN) and the Centro Internacional de Agricultura Tropical (CIAT). It is a histogram of the population of China. It provides population density data at a resolution of 2.5 arcminutes.

FIGURE B.2: The population density of China.

We transform this into a point set by drawing random points from the histogram.

Originator	Center for International Earth Science Information Network (CIESIN), Centro Internacional de Agricultura Tropical (CIAT).
Publication Date	2005
Title	Gridded Population of the World, Version 3 (GPWv3) Data Collection
Geospatial Data Presentation Form	raster digital data, map
Publication Place	Palisades, NY
Publisher	CIESIN, Columbia University
Online Linkage	`http://sedac.ciesin.columbia.edu/gpw/index.jsp`

Road intersections

A pointset used in (Wong *et al.*, 2005; Kumaraswamy, 2003) and many other papers regarding the fractal dimension of finite data. This dataset records as two dimensional points the road intersections in Montgomery County in Maryland, with 9552 points.

The dataset was generated from the US census' TIGER database. The processed version was downloaded from Leejay Wu's site at `http://www.cs.cmu.edu/~lw2j/`.

FIGURE B.3: Road intersections in Montgomery County.

Basketball

18 numeric attributes such as points scored and penalty minutes for around 19000 instances. Each instance represents a year that a player played for a given team. The years used are 1946-2004. The data was downloaded from `databasebasketball.com` (formerly `basketballreference.com`).

Galaxy cluster distribution

The large scale structure of the universe, seen at the scale of superclusters of galaxies exhibits a fractal structure, where galaxies are gravitationally bound in filaments, the largest known structures in the universe. The dataset is a numeric point set, containing redshift data, and position in the sky for around 18000 galaxies. These coordinates are converted to cartesian coordinates and scaled so that the full set fits into the bi-unit cube.

The dataset containing cartesian data was downloaded from the website of John Huchra at `http://www.cfa.harvard.edu/~huchra/seminar/lsc/`. It was generated from the 2 Micron All-Sky Survey Redshift Survey.

bt

FIGURE B.4: The large scale structure of the universe.

This publication makes use of data products from the Two Micron All Sky Survey, which is a joint project of the University of Massachusetts and the Infrared Processing and Analysis Center/California Institute of Technology, funded by the National Aeronautics and Space Administration and the National Science Foundation.(Skrutskie *et al.*, 2006)

Sunspots

A one-dimensional time series of the number of sunspots per month from 1749-1983. Downloaded from `http://robjhyndman.com/TSDL/` (Hyndman, 2009), originally from (Hipel & McLeod, 1994).

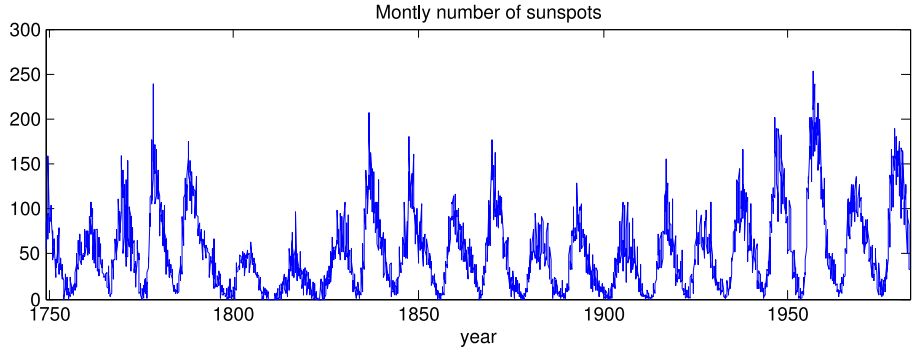


FIGURE B.5: A monthly count of the number of sunspots.

Currency

The global economy is notorious for producing a variety of chaotic processes and fractal structures.

This dataset records exchange rates for the Australian dollar, British pound, Canadian dollar, German DeutschMark, Dutch guilder, French franc, Japanese yen and the Swiss franc.

The data was downloaded from the time series library at `http://robjhyndman.com/TSDL/` (Hyndman, 2009), originally from (Franses & Van Dijk, 2000).

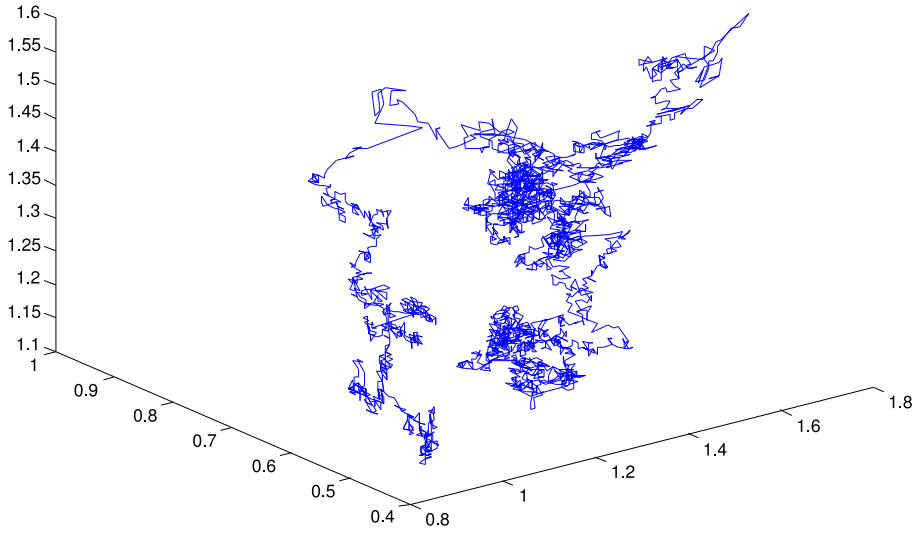


FIGURE B.6: The exchange rates of the Australian dollar, the British pound and the Canadian dollar, from 1979 to 1998.

B.2 Classification datasets

These datasets were used for classification tasks. Each contained points with a number of numerical features, and a single nominal feature, to be predicted.

B.2.1 Synthetic datasets

The Mandelbrot set

Arguably the most famous fractal of all. Using the points c and z in the complex plane, and starting with $z = 0$, we iterate the function $f(z) = z^2 + c$. If the point remains within some fixed radius, c is in the Mandelbrot set. Practically, we iterate for at least 50 steps and see whether the point has traveled outside the circle with radius 10. (For the familiar fractal zoom, much larger values are required, but for our purposes, these numbers suffice.

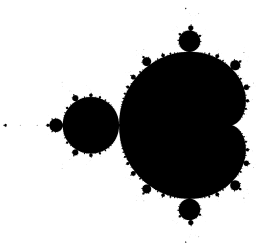


FIGURE B.7: The Mandelbrot set

Newton fractal

Newton's method is a way of finding numerical approximations of optimization problems, in the context of the Newton fractal, it usually refers to the problem of finding the roots of a complex polynomial. Using, for instance the polynomial $f(z) = z^3 - 1$, there are three possible solutions. Which solution Newton's method converges to, depends on the chosen starting point. If we color each

starting point according to the solution it converges to (assigning each an arbitrary color), we get the fractal of figure B.8.

If we treat this as a learning task, we get a set of points with two numerical attributes (defining a point in the plane) and one of three classes, determining the solution to which Newton's method converges.

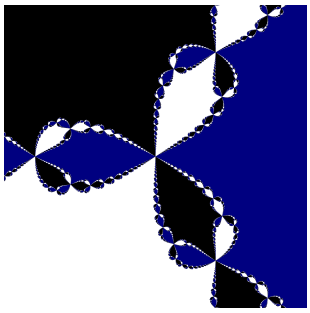


FIGURE B.8: A Newton fractal. Basins of attraction of the Newton method, applied to finding the complex roots of $f(z) = z^3 - 1$.

The magnetic pendulum

The fractal that was discussed in section 1.0.1

B.2.2 Natural datasets

These were all downloaded from the UCI Machine Learning repository (Asuncion & Newman, 2007).

Iris

The classic dataset introduced by Fisher in 1936 (Psychol *et al.*, 1936). It records four features of three species of iris, for 50 specimens per species.

US income census

This dataset contains census data for American households. The target class is whether the annual income is above or below 50000 dollars. Non-numeric features were removed.

Downloaded from `http://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD)`.

The dataset was clipped at about 65 thousand instances for the experiments in this thesis.

Forest cover

Contains 581012 instances each representing a 30×30 meter patch of ground in a US forest, described by 54 numeric attributes. One of seven classes describes the predominant type of forest cover (for instance: ponderosa pine, douglas fir or cottonwood/willow).

Downloaded from `http://archive.ics.uci.edu/ml/datasets/Covertype`, courtesy of Jock A. Blackard and Colorado State University.

The dataset was clipped at about 65 thousand instances for the experiments in this thesis.

Abalone

Contains 4177 instances, each representing a single specimen of abalone, with 8 numeric features per specimen. The classes are male, female and infant.

Downloaded from `http://archive.ics.uci.edu/ml/datasets/Abalone`.

Shuttle

A NASA dataset of measurements on the shuttle. Contains 58000 instances, 9 numeric features and 7 classes.

Downloaded from `http://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle)`.

APPENDIX C · ALGORITHMS AND PARAMETERS

This chapter provides technical descriptions of algorithms that are useful in conducting the experiments described in the main text. Some are not part of the final algorithms described, but were used in earlier versions or side experiments. They are described because they may be of use to anyone conducting similar research.

C.1 Volume of intersecting ellipsoids

Given two ellipsoids P and Q, both of dimension m, we wish to estimate the value

$$\frac{\lambda(P \cap Q)}{\lambda P}$$

In order to accomplish this we draw N random points uniformly from P, and count the number N' that fall inside Q, so that for large N, N'/N approximates the required value.

Each ellipse is represented by a rotation matrix R and a translation t vector so that $R \cdot B_1 + t$ is the ellipsoid, in other words the ellipsoids are defined as a transformation of the unit ball.

Algorithm 1 Estimating the intersection ratio for two ellipsoids.

$N' \leftarrow 0$
repeat N times
    # random uniform draw from the unit ball [1]
    $x = \{x_1, \ldots, x_m\}$, with $x_i \sim N(0, 1)$
    $u \sim U(0, 1)$
    $r \leftarrow u^{\frac{1}{m}} \frac{x}{\|x\|}$
    # turn it into a random point in P
    $p \leftarrow R_P \cdot r + t_P$
    # Coordinates relative to Q*'s center*
    $p \leftarrow p - t_Q$
    # Transform coordinates so that Q *is a ball*
    $p \leftarrow \text{inv}(Q) \cdot p$

    if $\|p\| \leqslant 1$ **then** $N' = N' + 1$

return $\frac{N'}{N}$

[1] The algorithm to generate this was taken from (Rubinstein & Kroese, 2007). Note that rejection sampling from a uniform distribution over $(-1, 1)^m$—rejecting those points with distance to the origin greater than one—will also generate a uniform distribution over the unit ball, but as the size of the rejection region grows exponentially with the dimension, this isn't a viable algorithm for dimensions greater than 5.

C.2 Probability of an ellipsoid under a multivariate Gaussian

Given an ellipsoid E of dimension d, we wish to estimate the probability of E under the distribution $N_d(0, I_d)$. In other words, what is the probability that a random point chosen from $N_d(0, I_d)$ falls in E?

This method is adapted from the more general algorithms described in (Somerville, 1998). We can describe any epsilon E as an affine transformation $E = \langle R, t \rangle$ of the unit ball:

$$E(B_1) = RB_1 + t$$

The inverse of this operation is $E^{-1} = \langle R^{-1}, -R^{-1}t \rangle$.

The basic idea of this algorithm is to choose random directions, or lines through the origin, and to get an estimate of E's probability from the section of this line that intersects E and the probability of this section given the distribution $N(0, I)$. We average these estimates over many random directions to get a good approximation.

We choose a random direction r (a unit vector, see C.1) and we define the random variable R as a point from $N_d(0, I_d)$ which lies on the line r. It can be shown that $R \sim \chi^2(d)$, ie. R is distributed according to a chi-squared distribution with d degrees of freedom. We know that r must intersect the boundary of E in zero, one or two points.

Once we have the intersection points (the method for finding these is explained below) we can use the following properties.

If the origin is inside E (ie. if $\|E^{-1}(x)\| \leqslant 1$) we have one intersection point $r\alpha$, where α is the distance to the boundary of E and we have $p(R \leqslant \alpha) = F_{\chi^2}(\alpha, d)$, where F_{χ^2} is the cumulative distribution function of the chi-squared distribution. In this case, we have

$$N(E|0, I) = \lim_{n\to\infty} \sum_{i\in(1,n)} \frac{p(R(r_i) \leqslant r_i\alpha) + p(R(-r_i) \leqslant -r_i\alpha)}{n}$$

Where $R(\cdot)$ and $\alpha(\cdot)$ represent the random variables and distances to the boundary of E for a given direction. r_i represent independently drawn random directions, and we sum the probability for r and $-r$ because we want to use a full line per sample, and not just the half-line from the origin.

If the origin is outside E, our direction can intersect the boundary at zero, one or two distances. In the first two cases, we count the probability for this line as zero. If there are two intersection points, $\alpha_1 < \alpha_2$ we use

$$p(\alpha_1 < R \leqslant \alpha_2) = F_{\chi^2}(\alpha_2, d) - F_{\chi^2}(\alpha_1, d)$$

As before, we average over this value for n random directions.

We will now explain how to find the intersections between r and the boundary of E, $bd(E)$. We know that $bd(B_1) = \{x \mid d(0, 1)\}$ so that

$$\begin{aligned} bd(E) &= \left\{x \mid d(0, E^{-1}(x)) = 1\right\} \\ &= \left\{x \mid \|R^{-1}x + R^{-1}t\| = 1\right\} \end{aligned}$$

since the points are on r, we also know that $x = r\alpha$ where α is a scalar. This gives us the following equation to solve:

$$\|R^{-1}r\alpha + R^{-1}t\| = 1$$

We square both sides, giving us the inner product on the left:

$$\begin{aligned}\left[R^{-1}r\alpha + R^{-1}t\right]^T \left[R^{-1}r\alpha + R^{-1}t\right] &= 1\\ \alpha^2\langle R^{-1}r\rangle - \alpha\left(\langle R^{-1}r, R^{-1}t\rangle + \langle R^{-1}r, R^{-1}t\rangle\right) - \langle R^{-1}t\rangle - 1 &= 0\end{aligned}$$

Where $\langle x, y\rangle$ is the inner product and $\langle x\rangle = \langle x, x\rangle$. We can solve for α using the quadratic formula, using the discriminant to determine the number of intersection points.

Finally, we can adapt this method for arbitrary Gaussian distributions by defining the required MVN as as an affine transformation of the standard distribution: $G(x) = Sx + u$, which makes the resulting distribution $N(u, SS^T)$.We can then say that $N(E \mid u, SS^T) = N(G^{-1}(E) \mid 0, I)$.

Algorithm 2 Estimating the probability of an ellipsoid under $N(0, I)$.

N: Number of samples
$E = \langle R, t\rangle$: An ellipsoid with dimension d.

$s = 0$
repeat N times
    $r \leftarrow$ *A random unit vector*

    # The parameters for the quadratic formula
    $a \leftarrow \langle R^{-1}r\rangle$
    $b \leftarrow \langle R^{-1}r, R^{-1}t\rangle + \langle R^{-1}r, R^{-1}t\rangle$
    $c \leftarrow \langle R^{-1}t\rangle - 1$

    # The discriminant
    $d \leftarrow b^2 - 4ac$
    if $d \geqslant 0$
        $\alpha_1, \alpha_2 \to -(b \pm \sqrt{d})/2a$

    if $\|R^{-1}x - R^{-1}t\| \leqslant 1$ *# if* x *is in* E
        $s \leftarrow s + F_{\chi^2}(|\alpha_2|, d) + F_{\chi^2}(|\alpha_1|, d)$
    else
        $s \leftarrow s + F_{\chi^2}(\alpha_2, d) - F_{\chi^2}(\alpha_1, d)$
return $\dfrac{s}{N}$

C.3 Optimal similarity transformation between point sets

Given two point sets $X = \langle x_1, x_2, \ldots, x_n \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ with $x_i, y_i \in \mathbb{R}^d$ we can calculate directly an affine transformation (consisting of a rotation matrix R, a translation vector t and a single scaling parameter c) which minimizes the mean squared error:

$$E(R, t, c) = \frac{1}{n} \sum_{i \in (1,k)} \|y_i - (cRx_i + t)\|^2$$

We only detail the method itself here. A proof is given in the original paper describing the method. (Umeyama, 1991)

This algorithm isn't used in the final text of this thesis, but it is a particularly relevant tool for the task of finding IFS models. If we have some indication of which points in a dataset are likely to map to one another under one of the IFS models transformations, we can use this algorithm to find that transformation.

Algorithm 3 Finding the optimal similarity transformation between two point sets

---

X: A matrix $X = \langle x_1, x_2 \ldots, x_n \rangle$ with $x_i \in \mathbb{R}^d$
Y: A matrix $Y = \langle y_1, y_2, \ldots, y_n \rangle$ with $y_i \in \mathbb{R}^d$

 # The means
$\bar{x} = \frac{1}{n} \sum_{x \in X} x$
$\bar{y} = \frac{1}{n} \sum_{y \in Y} y$

 # Standard deviations
$\sigma_x{}^2 = \frac{1}{n} \sum_{x \in X} \|x - \bar{x}\|^2$
$\sigma_y{}^2 = \frac{1}{n} \sum_{y \in Y} \|y - \bar{y}\|^2$

 # Covariance matrix
$\Sigma = \frac{1}{n} \sum_{i \in (1,n)} [x_i - \bar{x}][y_i - \bar{y}]^T$

 # Singular value decomposition
$\langle U, D, V^T \rangle = \text{svd}(\Sigma)$

$S \leftarrow I_d$ *# the identity matrix*
if $\det(\Sigma) < 0$
    $S_{dd} \leftarrow -1$

 # The optimal parameters
$R = USV^T$
$t = \bar{y} - cR\bar{x}$
$c = \frac{1}{\sigma_x{}^2} \text{trace}(DS)$

C.4 A rotation matrix from a set of angles

A vector of $(d^2 + d)/2$ angles is sufficient to describe any rotation in a d dimensional Euclidean space. The following algorithm describes how to transform such a vector into a rotation matrix R. This method is taken from (Rudolph, 1992) and (Schwefel & Rudolph, 1995). It is required to use the TSR and Similitude parameter representations and in the ES algorithm, to generate random normal mutations based on the strategy parameters of the agent.

Algorithm 4 Constructing a d-dimensional rotation matrix from a set of angles.

α: A vector of size $(d^2 + d)/2$ with $\alpha_i \in \mathbb{R}^d$

$R \leftarrow I_d$ *# The identity matrix*
for $i \in (1, d-1)$
  for $j \in (i+1, d)$
    # Construct an elementary rotation matrix E
    $k \leftarrow (2d - i)(i + 1)/2$
    $E \leftarrow I_d$
    $E_{ii}, E_{jj} \leftarrow \cos(\alpha_k)$
    $E_{ij}, E_{ji} \leftarrow -\sin(\alpha_k)$
    $R \leftarrow R \cdot E$

return R

C.5 Drawing a point set from an instance of a random measure

Section 5.3.2 describes the concept of random measures, specifically random probability distributions. As explained in that section every probability distribution over $2k$-tuples (such that one instance contains k maps and k priors) determines a unique random fractal (where a random fractal is a probability distribution over Euclidean sets with statistical self-similarity).

$\mathcal{S}$ can be described in many ways, We can select a finite number of discrete IFS model, and simply choose from them at random, we can describe the $2k$-tuple as a vector in a very large parameter space (as described in chapter 3) and choose a random parameter according to some simple distribution. There are many possibilities. For this algorithm all that is required is that we can generate a random instance according to $\mathcal{S}$.

Since we are approximating the process

$$\lim_{m \to \infty} \mathcal{S}^m(\mathcal{E})$$

we will use a recursive function, that ends at a given depth with a random draw from $\mathcal{E}$. Since the influence of $\mathcal{E}$ diminishes as m grows, we will take $\mathcal{E}$ to be the distribution over measures which always returns the standard multivariate normal distribution $N(0, I)$.

Algorithm 5 Drawing a set of iid points, distributed according to a single instance of some $\mathcal{S}$

$\mathcal{S}$: A probability distribution over k tuples as described above
n: The number of points to return.
s: A seed for a random number generator $r(s)$ as described above
d: The depth to which to evaluate.

function instance($\mathcal{S}, n, s, d$)
    return instanceInner($\mathcal{S}, n, r(s), d$)

function instanceInner($\mathcal{S}, n, r, d$)
    if $d = 0$
        return [probRound(n) *points drawn from* $N(0, I)$]

    # Draw a random discrete scaling law
    $S \sim \mathcal{S}$

    for $i \in (1, k)$
        # Draw a random instance at a lower recursion depth
        $I_i \leftarrow$ instanceInner($\mathcal{S}, n \cdot \rho_i, r, d - 1$)

    return $\sum_{i \in (1,k)} S_i(I_i)$

We include a parameter s in the function to represent a seed to a standard random number generator. We use this random number generator when generating the draws from $\mathcal{S}$, but not when drawing points from $\mathrm{N}(0, \mathrm{I})$ at the end of the recursion, so that when the function is called twice with the same seed, it returns two different point sets drawn from the same instance of the random fractal defined by $\mathcal{S}$

In most situations, n will not be an integer at the end of recursion, and in many scenario's it will always be below 0.5. Because of this, it is useful to use probabilistic rounding. A probabilistic round rounds a given real valued number r at random, in such a way that the expectation of such a rounding operation equals r. Since the line where n is rounded is called many times, the functions returns on average, about the right number of points.

The following algorithm describes how to implement probabilistic rounding.

Algorithm 6 Probabilistic rounding

r: A real valued number.

function probRound(r)
    $u \sim \mathrm{U}(0, 1)$
    if $u < r - \mathrm{floor}(r)$
        return ceil(r)

    return floor(r)

C.6 Box counting dimension

Estimating $I_0(\epsilon)$ for the range $\epsilon \in \{1, \frac{1}{2}, \cdots, \frac{1}{2^n}\}$.

Algorithm 7 Building a tree to estimate the box counting dimension

```
n: Maximum depth
X: A dataset of points in ℝ^d scaled to fit (0,1)^d
global ϵ ← 0^n # A vector of n elements that will hold the box counts

r ← new node # Root node
for x ∈ X
    # 0^d and 1^d are vectors of length d filled with values 0 and 1 respectively
    σ = code(x, 0^d, 1^d, ⟨⟩, n)
    r.add(σ, 1)
return ϵ

function code(x, m⁻, m⁺, σ, k)
    if d = 0 return σ

    p = (m⁻ + m⁺)/2 # Calculate center of parent box
    c ← 0
    n⁻ ← 0^d, n⁺ ← 1^d
    for i in [0, d)
        if x_i > m_i^m
            c ← c + 2^i
            n_i⁻ ← p_i, n_i⁺ ← m_i⁺
        else
            n_i⁻ ← m_i⁻, n_i⁺ ← p_i
    σ.add(c) # add the new codon to the sequence
    return code(x, n⁻, n⁺, σ, k − 1)

class node
    field c ← new map # Children: a map from codons (integers) to nodes
    field k ← 0 # The depth of this node in the tree
    function add (σ, i)
        if ¬c.contains(σ_i)
            n ← new node
            n.k ← k + 1
            ϵ_{k+1} ← ϵ_{k+1} + 1 # Increment the box count
            c.add(σ_i, n)
        c.get(σ_i).add(σ, i + 1)
```

C.7 Common parameters for evolution strategies

Unless noted otherwise, these parameters were used for evolution strategies.

τ	0.8	The mutation of the angles section of the strategy parameters are mutated by $N(0, \tau)$.
υ	0.001	The mutation of the scaling section of the strategy parameters are mutated by $N(0, \upsilon)$.
u	50	The population size after the bottom agents are removed.
ν	100	The population size after the children have been created.
g	4000	Training is stopped after g generations, and the top model of the last population is selected.
s	250	The sample size. Calculating the Hausdorff distance is far to expensive an operation to perform on the whole dataset, every model, every generation. We sample s points from the dataset with replacement, and s points from the model, and calculate the Hausdorff distance for between these as the model's fitness. The value is stored for a single generation and recalculated on fresh samples the next.
o	2	The number of parents for each child. These are chosen at random from the population, with replacement.
Crossover	uniform	All parameters are mutated by uniform crossover. For each parameter p_i in the child, a random parent is chosen, and its p_i is copied.
$\sigma_o, \sigma_s, \sigma_a$	$0.01, 0, 0$	When generating the first population, we choose the parameters from a normal distribution. The model parameters are chosen from $N(0, \sigma_o)$, the scaling parameters are chosen from $N(0, \sigma_s)$ and the mutation angles are chosen from $N(0, \sigma_a)$.
		There is no maximum lifespan. Each model is allowed to remain in the population so long as its performance puts it in the top ν models. Its fitness is recalculated each generation.

APPENDIX D · HAUSDORFF DISTANCE

Our attempts to model data with fractals led very quickly to the concept of Hausdorff distance. Hausdorff distance has been a staple in the field IFS approximation almost from day one, but is almost entirely unused within the field of Machine Learning. To show that Hausdorff distance may have a wide applicability in in many Machine Learning problems, we investigate its use in more detail.

D.1 Learning a Gaussian mixture model

To show the correlation between the Hausdorff distance and the likelihood, we perform a simple experiment. We create a two-dimensional dataset from a basic mixture model, and learn it with the ES model as described in chapter 3.

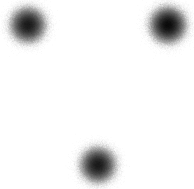


FIGURE D.1: The dataset used in the experiment

For each run we create a collection of each agent's Hausdorff distance and log likelihood on the dataset, plotting the first against the second in a two dimensional log histogram [1]

We found that if this is done without a learning algorithm, by simply sampling random models, there is no significant correlation between Hausdorff distance and likelihood. But as the algorithm progresses a clear (inverse) correlation can be seen. For this reason we have discarded the first 100 generations (of a total of 500).

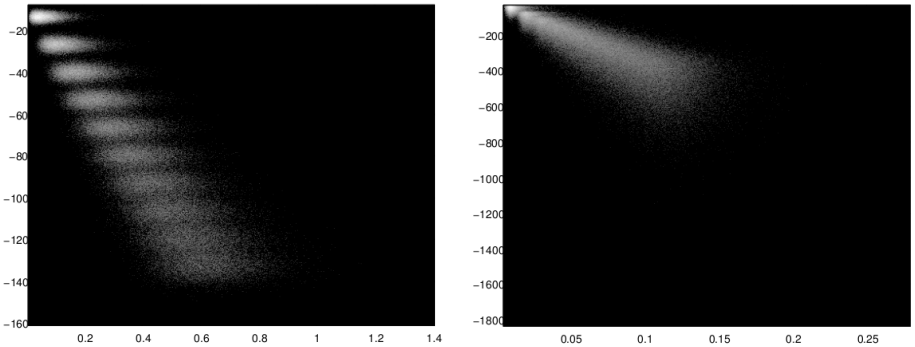
Figure D.2 shows the results. A clear inverse correlation can be seen.

D.1.1 Discussion

These experiments performed here show that this technique is promising. Since the class of models for which it is applicable is distinct from the class of models for which likelihood is applicable as a fitness funtion, different fields may benefit from the use of Hausdorff distance as a fitness function. Two properties are important for the use of Hausdorff distance:

- The model must be defined over a metric space. This is more general than the Euclidean space over which many probability models are defined. This opens the door to applying Hausdorff distance to probability models over, for instance, strings of text or protein sequences.
- The model must be generative. If we can use the model to generate a set

[1] we divide the rectangle bounding all the datapoints into 500×500 bins, and we plot the log value of the number of points hitting each bin according to a grayscale colormap.



(A) Learning by likelihood (B) Learning by Hausdorff distance

FIGURE D.2: The results of two experiments, using ES to learn a simple Gaussian mixture model using the log-likelihood and the Hausdorff distance as a fitness function. The vertical axes show the log likelihood, the vertical axes show the Hausdorff distance.

of random points, independently drawn from its probability distribution, there is no need to be able to calculate the probability density of a given point. For some models, like the IFS models in chapter 3, generating random points is far easier than finding the density of a single point, or the probability over a region.

APPENDIX E · NEURAL NETWORKS

This appendix describes a line of research which diverges from the main line of this thesis, but is nevertheless interesting enough to report. The basic principle behind it is to use a simple neural network in an iteration to generate fractal probability distributions and classifications.

As we saw in chapter 1, dynamical systems can be a rich source of fractals. A simple function $f : \mathbb{R}^d \to \mathbb{R}^d$ can be iterated to describe the orbit of some initial point $x_0 \in \mathbb{R}^d$, so that $x_1 = f(x_0)$ and $x_{n+1} = f(x_n)$. For just the right kind of function, this orbit becomes chaotic, it fills a large region of the phase space and never hits the same point twice. Furthermore, any uncertainty about the position of x_0 quickly balloons out until the only knowledge we have of the points position at x_n (for some n) is a general probability distribution over $\mathbb{R}^d$ that is entirely independent of the initial point [1].

This distribution is equivalent to the probability distribution we get when we choose a point x by choosing x_t for a random point t chosen uniformly from $(0, n)$, where we let n go to infinity. This is called the attractor's *natural measure*. For additional discussion of these concepts see section 1.0.1 and section 2.1.4.

In addition to an attractor's natural measure, we can also use basins of attraction to model fractals. If our function has multiple attractors, we can partition the space based on where each initial point ends up under the iteration. For a map with two attractors (chaotic or otherwise), we can color all points for which the orbit tends to the first black and all point for which the orbit tends to the second white. The black and white subsets of state space are called the basins of attraction of the attractors. For various maps, the boundaries of these basins have intricate fractal structure. The Mandelbrot set, the Newton fractal and the magnetic pendulum are examples of such dynamic systems.

The basic principle behind the ideas in this appendix is that we can use a neural network to represent the function f, and iterate that network to generate fractals. Since a neural network is completely determined by a set of real values, we can use evolution strategies to fit the network to data.

E.1 Density estimation

To make clear what we're trying to do we will look at some examples. Our first example, the Hénon attractor is described by the following iteration

$$x_{n+1} = y_n + 1 - \alpha {x_n}^2$$
$$y_{n+1} = \beta x_n$$

[1] with the exception of a vanishingly small set of initial points

Where the model with $\alpha = 1.4$ and $\beta = 0.3$ has a strange attractor. (Alligood *et al.*, 1996)

FIGURE E.2: The Henon attractor.

For a more visually appealing example, we turn to the *Pickover attractors*, described by the map:

$$x_{n+1} = \sin(\alpha x_n) + \beta \cos(\alpha y_n)$$
$$y_{n+1} = \sin(\gamma x_n) + \delta \cos(\gamma y_n)$$

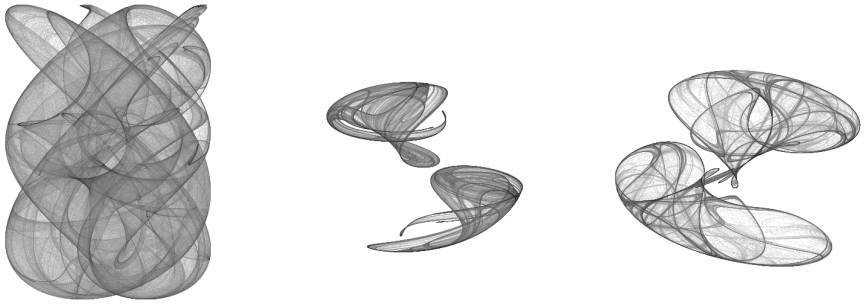
Where α, β, γ and δ are parameters. This map can show a wide variety of structure for different parameters. (Krawczyk, 2003)

The central notion behind these fractals (and many others), is that of an iterated map. What we attempt in this section is to generalize this map to a neural network, and to learn the parameters so that the attractor's natural measure fits given data.

We use a three-layer neural network with $2d$ input nodes, h hidden nodes, and d output nodes, where d is the dimensionality of the space for which we are trying to learn a probability distribution, and h is a model parameter. The connections from the input layer to the hidden layer have a linear activation, the connections from the hidden layer to the output have a sigmoidal activation function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The first two layers also have bias nodes. During our iteration we continually take the output from the network and use it as the values for the first d input nodes. The other d input nodes remain constant (these are model parameters, we include these so that the network can simulate the function behind the Mandelbrot set, see the next section). To draw a random point from our model, we first choose a random point from $N(\mu, Is)$, where μ and s are model parameters. We then iterate the model with this point as input. After n iterations, we take the d output values, which gives us a single point in $\mathbb{R}^d$. Controlling the distribution of the input points allows us make sure that all input points lie in the basin of attraction of the required attractor.



(A) $-2.03, -2.9, 1.44, 0.7$ (B) $-1.32, -1.99, 0.66, 0.66$ (C) $-1.12, -1.63, 0.9, -1.1$

FIGURE E.1: Three pickover attractors. Captions show the values of α, β, γ and δ respectively.

We also assume that the points visited by a single orbit have the same frequency as those returned by as many iid draws[2] (if we exclude the first 50 or so points to let the orbit get to its attractor). Under this assumption we can simply generate a large dataset by generating a single long orbit, and ignoring the first few points.

A single model is described by the weights of the network, the parameters μ and s and the values of the d constant nodes. These are all real values, so a single model is described by a single vector in $\mathbb{R}^m$ for some m. We use evolution strategies, as described in chapter 3 to train this model. As in that chapter, we use Hausdorff distance as a fitness function.

The results are shown in figure E.3

The results show some promise. The algorithm is able to learn strange attractors to successfully cover a more than 1 dimensional region of the instance space. Unfortunately it breaks down for the Hénon attractor. Our implementation of the ES algorithm is unable to find the map that perfectly describes the attractor.

It may still seem tempting to try and construct a classifier from this model (as we did with the IFS models in chapter 4. Unfortunately, the main drawback of this model is that it is only generative. There is no simple way of calculating the probability density of a point or the probability of a region under a given attractor. This makes the model much less useful for machine learning purposes than the IFS model. Its main use would be to reconstruct a discrete dynamical system from observed data. As we have seen from the Hénon experiment, there is still some way to go.

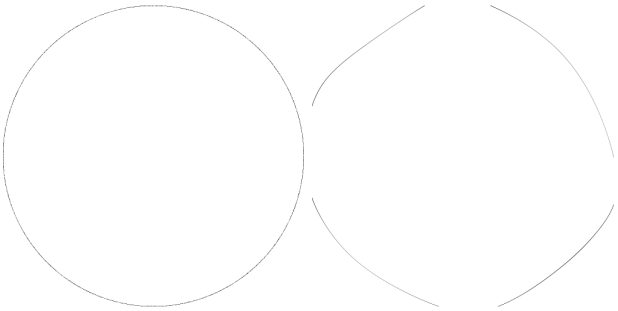
E.2 Classification

As noted before, fractals manifest themselves not only as the natural measures of strange attractors, but also as the basin boundaries of attractors (strange or otherwise). One example of this is the Mandelbrot set. Under iteration of the complex map $z_{n+1} = z_n^2 + c$, some points are attracted to infinity (in whatever direction), and for some points, the iteration remains bounded. Another example is the Newton fractal, under its map, all points are attracted to one of three point attractors.

The key insight here, is that these maps define a partition of the instance space. In other words, they are classifiers. If we can generalize this concept using neural networks, we can search for a map which fits a given dataset.

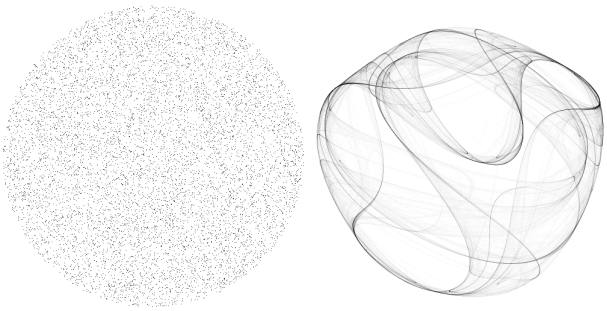
To cast this model into the mold of neural networks, we need two ingredients. First we need a neural network to approximate the map. For this purpose, we use exactly the same topology as described in the previous section. In this context, we won't need the distribution over initial points, as our initial point is the input point we wish to classify. After iterating the map some 50 or 100 times, we can be reasonably sure that the input point has reached its attractor. All we need now is a neural network to map attractors to classes. Since attractors tend to have wide margins between them, a simple network should be capable of

[2] This property is related to the notion of *ergodicity*

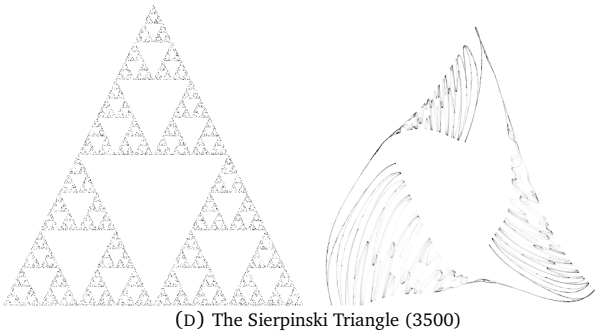


(A) A Sphere (500)

(B) The Hénon attractor (4000)

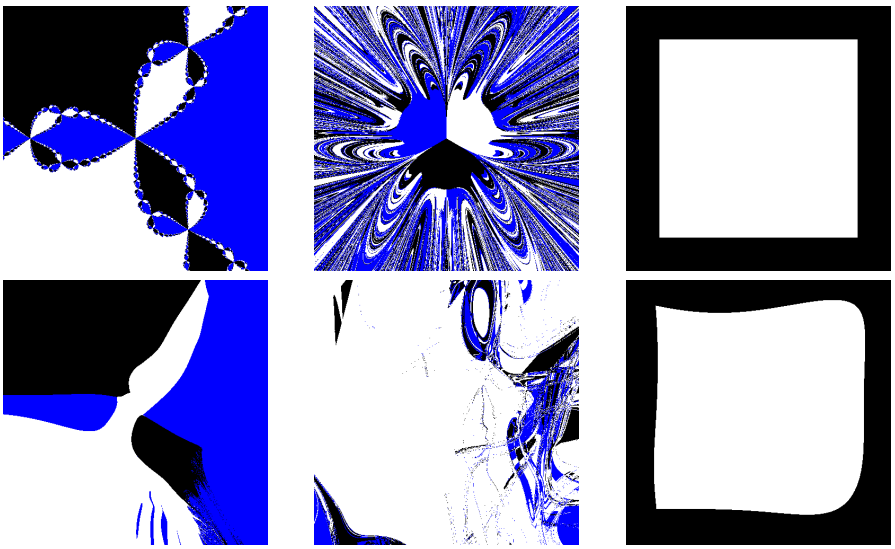


(C) A ball (4500)



(D) The Sierpinski Triangle (3500)

FIGURE E.3: Four results of density estimation. The data is shown on the left. The number in brackets shows the generation that was chosen to display (either because some generation had a particularly good model, or to show how early a good model appeared). For some of the images the levels have been tweaked to bring out the detail.



(A) The Newton fractal (9000) (B) The magnet fractal (1250) (C) A simple square dataset (9000)

FIGURE E.4: Three results of classification tasks.

representing this function.

In this case, the entire network is described by the weights of the neural network. We learn, as before, using evolution strategies. This time we use the symmetric error (over a subsample of the data) as the fitness function. Results are shown in figure E.4.

Learning the Mandelbrot set

The results of the classifier are not very encouraging. Either the model is not capable of expressing the required fractals, or the fractal solutions are too difficult to find. To shed some light on this, we try the following experiment on the Mandelbrot set. We split the model into two neural networks and we train each independently using backpropagation. For the map we draw a dataset a of random points in the plane, and apply the function to find the required targets. For the second part, we use the rule that point at a distance of 10 units from the origin can be assumed to have diverged to infinity, and we train on a dataset of points with these targets.

After 50 epochs of backpropagation over a dataset 10 000 points, we combine the two models and create a plot of the resulting classifier. The results(figure E.5) are remarkably close to the Mandelbrot set. This suggests that the neural network model can successfully approximate fractals like the Mandelbrot set, but these solutions are difficult to find for the evolutionary algorithm [3]

[3] We also tried using backpropagation-through-time, with no better results.

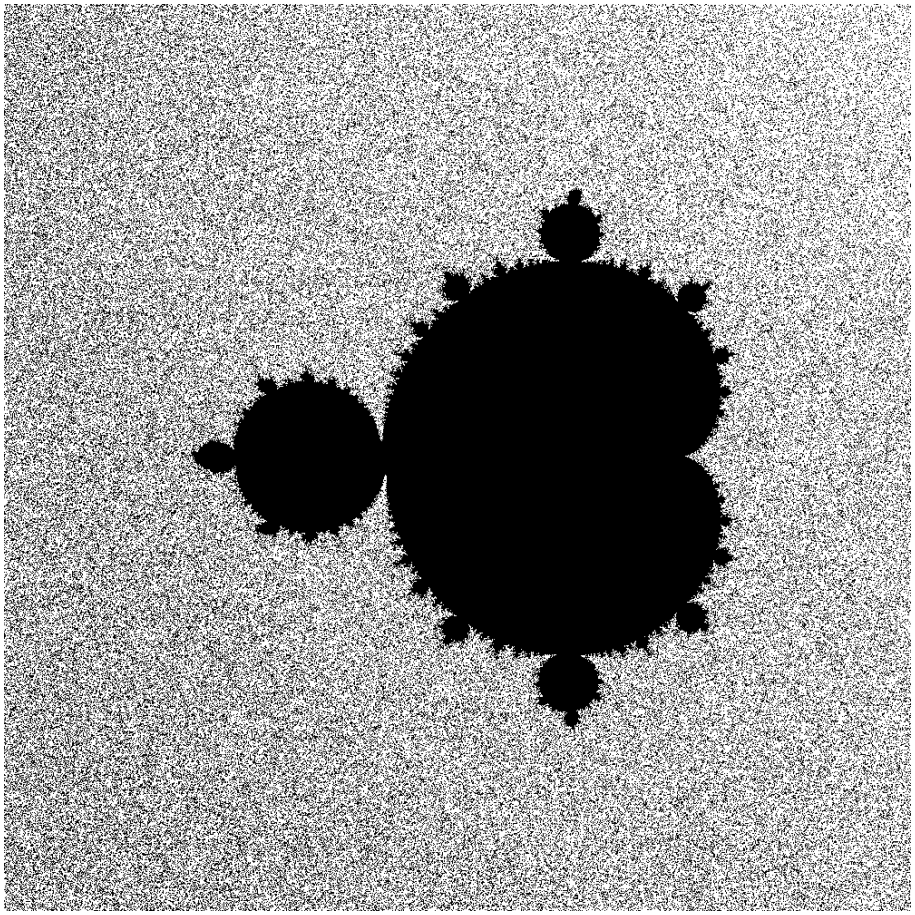


FIGURE E.5: A neural network approximation of the Mandelbrot set. We sampled 10 000 points from $\mathcal{N}_4(0, \mathrm{I})$ as out input set (to represent the two inputs) and applied the Mandelbrot set's map to them to create a target set. On this set we trained a neural network with 20 hidden nodes for 50 epochs. For the second part of the network we created a dataset of another 10 000 points drawn from $\mathcal{N}_2(0, \mathrm{I} \cdot 25)$, and assigned classified each as "inside the Mandelbrot set" if the point was within a radius of 10 units from the origin. Connecting these two networks, and running it for 20 iterations, we get the image shown above.

REFERENCES

ALLIGOOD, K.T., SAUER, T., & YORKE, J.A. 1996. *Chaos: an introduction to dynamical systems*. Springer Verlag.

ANDRLE, R. 1996. The west coast of Britain: statistical self-similarity vs. characteristic scales in the landscape. *Earth surface processes and landforms*, **21**(10), 955–962.

ASUNCION, A., & NEWMAN, D.J. 2007. *UCI Machine Learning Repository*.

BEYER, H.G., & SCHWEFEL, H.P. 2002. Evolution strategies–A comprehensive introduction. *Natural Computing*, **1**(1), 3–52.

BISHOP, C.M., *et al.* 2006. *Pattern recognition and machine learning*. Springer New York:.

CHAN, K.S., & TONG, H. 2001. *Chaos: a statistical perspective*. Springer Verlag.

CHANG, S.H., CHENG, F.H., HSU, W., & WU, G. 1997. Fast algorithm for point pattern matching: invariant to translations, rotations and scale changes. *Pattern Recognition*, **30**(2), 311–320.

DELIGNIERES, D., & TORRE, K. 2009. Fractal dynamics of human gait: a reassessment of Hausdorff et al.(1996)'s data. *Journal of Applied Physiology*, 90757–2008.

ELTAHIR, E.A.B. 1996. El Niño and the natural variability in the flow of the Nile River. *Water Resources Research*, **32**(1), 131–137.

EMBRECHTS, P., & MAEJIMA, M. 2002. *Selfsimilar processes*. Princeton Univ Pr.

FRANSES, P.H., & VAN DIJK, D. 2000. *Nonlinear time series models in empirical finance*. Cambridge Univ Pr.

GRASSBERGER, P., & PROCACCIA, I. 1983. Measuring the strangeness of strange attractors. *Physica D: Nonlinear Phenomena*, **9**(1-2), 189–208.

GREENSIDE, HS, WOLF, A., SWIFT, J., & PIGNATARO, T. 1982. Impracticality of a box-counting algorithm for calculating the dimensionality of strange attractors. *Physical Review A*, **25**(6), 3453–3456.

GRUNWALD, P. 2005. A tutorial introduction to the minimum description length principle. *Advances in minimum description length: theory and applications*.

HEIN, M., & AUDIBERT, J.Y. 2005. Intrinsic dimensionality estimation of submanifolds in R d. *Page 296 of: Proceedings of the 22nd international conference on Machine learning*. ACM.

HIPEL, K.W., & MCLEOD, A.I. 1994. *Time series modelling of water resources and environmental systems*. Elsevier Science Ltd.

HURST, H.E., BLACK, RP, & SIMAIKA, YM. 1965. *Long-term storage: An experimental study*. Constable.

HUTCHINSON, J.E. 2000. Deterministic and random fractals. *Complex systems*.

HYNDMAN, R. (N.D.). 2009. *Time series data library*.

KORN, F., & PAGEL, B. 2001. On the "dimensionality curse" and the "self-similarity blessing". *IEEE Transactions on Knowledge and Data Engineering*, **13**(1), 96–111.

KRAWCZYK, R.J. 2003. Dimension of Time in Strange Attractors. *In: ISAMA and Bridges 2003 Joint 2003 Conference, edited by R. Sarhangi*. Citeseer.

KUMARASWAMY, K. 2003. *Fractal Dimension for Data Mining*.

LELAND, W.E., TAQQU, M.S., WILLINGER, W., & WILSON, D.V. 1994. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking (ToN)*, **2**(1), 1–15.

LINKENKAER-HANSEN, K., NIKOULINE, V.V., PALVA, J.M., & ILMONIEMI, R.J. 2001. Long-range temporal correlations and scaling behavior in human brain oscillations. *Journal of Neuroscience*, **21**(4), 1370.

MANDELBROT, B. 1967. How long is the coast of Britain? Statistical self-similarity and fractional dimension. *Science*, **156**(3775), 636.

MEYER, M., & STIEDL, O. 2003. Self-affine fractal variability of human heartbeat interval dynamics in health and disease. *European journal of applied physiology*, **90**(3), 305–316.

OTT, E., SAUER, T., & YORKE, J.A. 1994. *Coping with chaos*. Wiley New York.

PEITGEN, H.O., JÜRGENS, H., & SAUPE, D. 2004. *Chaos and fractals: new frontiers of science*. Springer.

PETERS, E.E. 1994. *Fractal market analysis: applying chaos theory to investment and economics*. Wiley.

PSYCHOL, J.E., GENERALIS, A.O.O., GENET, S.A., BIOL, M., BIOINFORMATICS, BMC, ANAL, C.S.D., & CHEMOM, J. 1936. 1. Fisher R: The use of multiple measurements in taxonomic problems. *Ann of Eugenics*, **7**, 179–188.

RUBINSTEIN, R.Y., & KROESE, D.P. 2007. *Simulation and the Monte Carlo method*. Wiley-Interscience.

RUDOLPH, G. 1992. On correlated mutations in evolution strategies. *Parallel problem solving from nature*.

SCHROEDER, M. 1996. *Fractals, chaos, power laws: Minutes from an infinite paradise*. Freeman New York.

SCHWEFEL, H.P., & RUDOLPH, G. 1995. Contemporary evolution strategies. *Lecture Notes in Computer Science*, 893–893.

SKRUTSKIE, MF, CUTRI, RM, STIENING, R., WEINBERG, MD, SCHNEIDER, S., CARPENTER, JM, BEICHMAN, C., CAPPS, R., CHESTER, T., ELIAS, J., *et al.* 2006. Two Micron All Sky Survey (2MASS). *The Astronomical Journal*, **131**, 1163–1183.

SOMERVILLE, P.N. 1998. Numerical computation of multivariate normal and multivariate-t probabilities over convex regions. *Journal of Computational and Graphical Statistics*, 529–544.

TAKENS, F. 1985. On the numerical determination of the dimension of an attractor. *Lecture notes in mathematics*, **1125**, 99–106.

THEILER, J. 1990. Estimating fractal dimension. *Journal of the Optical Society of America A*, **7**(6), 1055–1073.

TRAINA JR, C., TRAINA, A., WU, L., & FALOUTSOS, C. 2000. Fast feature selection using fractal dimension. *In: XV Brazilian Symposium on Databases (SBBD)*.

UMEYAMA, S. 1991. Least-squares estimation of transformation parameters between twopoint patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **13**(4), 376–380.

WERBOS, PJ. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, **78**(10), 1550–1560.

WITTENBRINK, C.M., LANGDON, G.G.J., & FERNANDEZ, G. 1996. Feature extraction of clouds from GOES satellite data for integrated model measurement visualization. *Pages 212–222 of: Proceedings of SPIE- The International Society for Optical Engineering.*, vol. 2666. Citeseer.

WONG, A., WU, L., & GIBBONS, P.B. 2005. Fast estimation of fractal dimension and correlation integral on stream data. *development*, **5**, 10.

IMAGE ATTRIBUTIONS

figure 1.3 The Hilbert curve. The image was adapted from the file available at Wikimedia Commons: http://commons.wikimedia.org/wiki/File:Hilbert_curve.png, originally created by Zbigniew Fiedorowicz. This image is licensed under the Creative Commons Attribution ShareAlike 3.0 License. The license can be found here: http://creativecommons.org/licenses/by-sa/3.0/

figure 1.7a The Lorenz attractor. This image was taken from Wikimedia Commons: http://commons.wikimedia.org/wiki/File:Lorenz_attractor.svg as created by user Dschwen. It is distributed under the Creative Commons Attribution ShareAlike 3.0 License. The license can be found here: http://creativecommons.org/licenses/by-sa/3.0/

figure 1.7b The Rössler attractor. This image was taken from Wikimedia Commons http://commons.wikimedia.org/wiki/File:Roessler_attractor.png as created by user Wofl. It is distributed under the Creative Commons Attribution ShareAlike 2.5 License. The license can be found here: http://creativecommons.org/licenses/by-sa/2.5/deed.en

figure 1.7c A Pickover attractor. This image was adapted from a file available at MediaWiki Commons: http://commons.wikimedia.org/wiki/File:Atractor_Poisson_Saturne.jpg as added by user Alexis Rufatt. It is distributed under the Creative Commons Attribution ShareAlike 2.5 License. The license can be found here: http://creativecommons.org/licenses/by-sa/2.5/deed.en

INDEX