A New Model of Artificial Intelligence

Charles Davi

March 4, 2019

Abstract

In this article, I'll present a new model of artificial intelligence rooted in information theory that makes use of tractable, low-degree polynomial algorithms that nonetheless allow for the analysis of the same types of extremely high-dimensional datasets typically used in machine learning and deep learning techniques. Specifically, I'll show how these algorithms can be used to identify objects in images, predict complex random paths, predict projectile paths in three-dimensions, and classify three-dimensional objects, in each case making use of inferences drawn from millions of underlying data points, all using low-degree polynomial run time algorithms that can be executed quickly on an ordinary consumer device.[1] In short, the purpose of these algorithms is to commoditize the building blocks of artificial intelligence. All of the code necessary to run these algorithms, and generate the training data, is available on my researchgate homepage.[2]

1 Introduction

The core problem solved by my model of AI is how to operate without any prior information at all. This is a problem that is arguably beyond the scope of any traditional machine learning or deep learning techniques, since both of those models of AI generally require some form of training data. In contrast, my image feature recognition algorithm, and categorization algorithm, are both designed to operate without any training data at all, allowing them to serve as generalized pattern recognition algorithms.

[1]The run times referenced in this article are expressed in terms of the number of built-in Matlab functions and operations that are called by an algorithm. When appropriate, I will of course discuss the practical run times of the algorithms as well.

[2]I retain all rights, copyright and otherwise, to all of the algorithms, and other information presented in this paper. In particular, the information contained in this paper may not be used for any commercial purpose whatsoever without my prior written consent. All research notes, algorithms, and other materials referenced in this paper are available on my researchgate homepage, at https://www.researchgate.net/profile/Charles_Davi, under the project heading, *Information Theory*.

The core observation underlying my solution to this problem is that even in the absence of information about the processes that generated a given dataset, we can nonetheless partition the dataset in a manner that satisfies objective criteria. Specifically, we can measure the information content of each subset of a partition of a dataset using Shannon's equation $I = \log(\frac{1}{p})$, where p is the number of items in the subset in question divided by the total number of items in the entire dataset. This in turn allows us to measure the distribution of information among the subsets generated by a given partition of the dataset.

Superficially, this might seem like an academic curiosity, especially since the probability is in this case a density, and not the probability of an actual signal that we're trying to encode. However, it turns out that if we partition a dataset in a manner that maximizes the standard deviation of the information contents of the resultant subsets, we reveal a substantial amount of structure in the dataset, and at the same time, establish an objective criteria that can be evaluated given any dataset, even in the absence of any prior information about the dataset, thereby providing the spark that sets an otherwise autonomous set of algorithms into motion.

2 Image Feature Recognition

I'll begin with the image feature recognition algorithm, since its results are visual, allowing for a quick demonstration of how we can turn this abstract theory into concrete results. As an example, I've included a photo that I took in Stockholm, Sweden, at a sunny intersection in Södermalm.

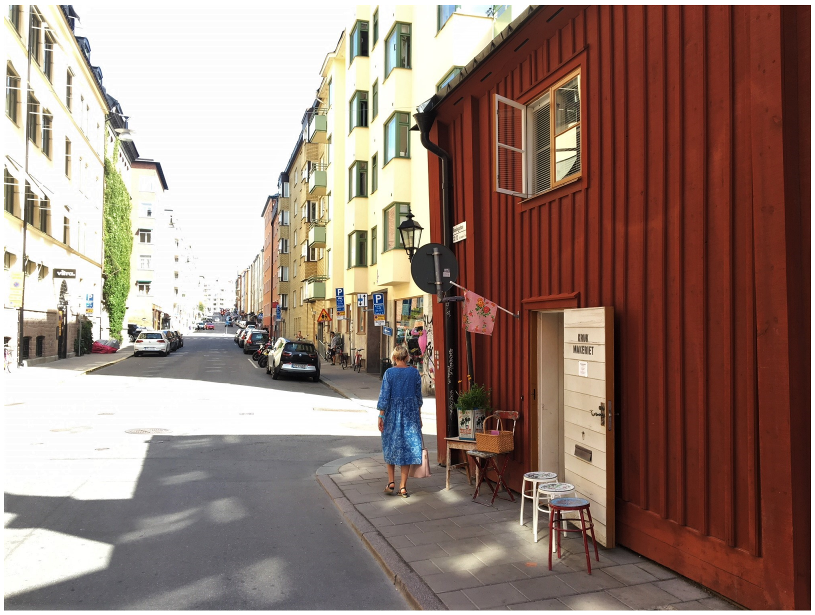
Figure 1: The original photo, taken in Södermalm, Stockholm.

Figure 2 shows the photo after being processed by the feature recognition algorithm, with the rectangular regions showing the borders of the features identified by the algorithm. The brighter a given region is, the more likely the algorithm thinks that the region in question is part of the foreground of the image. Figure 3 shows some of the individual features identified by the algorithm, which in this case include several contiguous objects, such as the series of stools on the right-hand side of the photo, the woman's dress, and the flag on the outside of the shop.

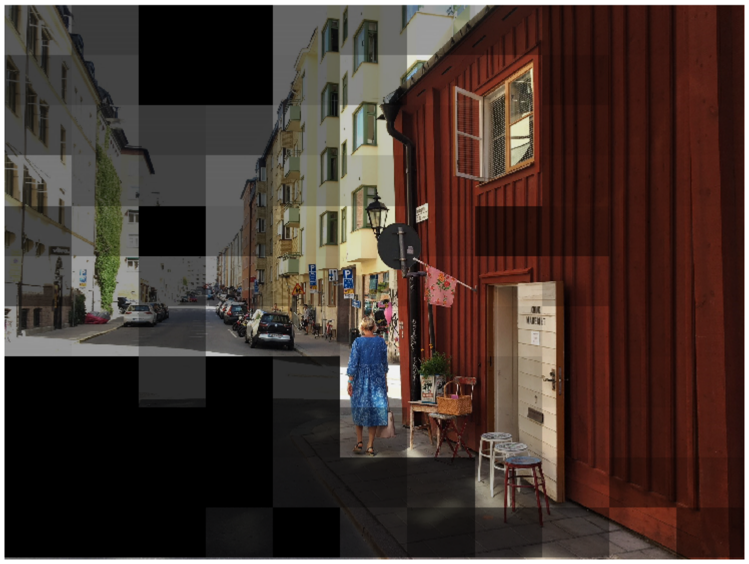

Figure 2: The photo after being processed by the feature recognition algorithm.

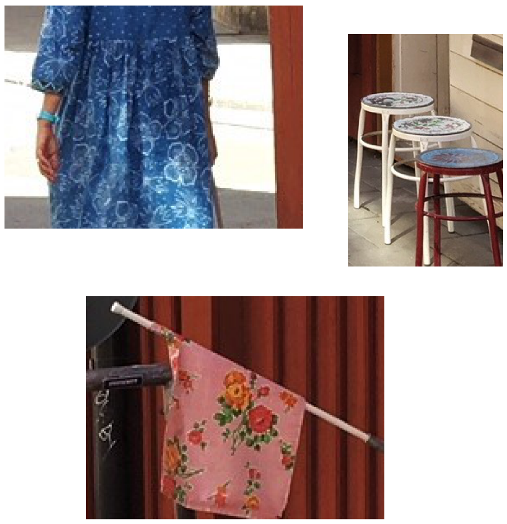

Figure 3: Three features identified by the feature recognition algorithm.

As you can see, the algorithm is able to identify objects of different shapes, sizes, and colors, in a busy scene, despite the fact that it has no prior information about the image. In fact, the algorithm has no prior information at all about anything. Instead, it treats every image as a case of first impression, and begins by subdividing the image into equally sized rectangular regions, iteratively decreasing the size of each region, until each region contains a maximally different amount of information. The goal of this process is to subdivide the image into maximally distinct features. In the subsections that follow, I'll explain in some detail how this algorithm works.

2.1 Machine Learning and Prior Data

Human beings are capable of recognizing objects in images, and in real life, often with no prior information at all about what's to be observed. This is generally a hard problem for machines, but there are obviously plenty of algorithms that can quickly recognize faces, cars, and other objects within images, even in real-time. Arguably the most successful techniques in this area are in essence statistical techniques, commonly thrown under the umbrella buzzword of "machine learning".

In general, machine learning techniques make use of prior data, meaning that an algorithm is trained on data that is presumed to be similar to the data that the algorithm will eventually be applied to. Machine learning techniques can be further categorized into **supervised learning**, where a human being provides some of form information about the dataset that is exogenous to the dataset, such as category labels, and **unsupervised learning**, where the algorithm is itself responsible for generating that type of information. In both cases, statistical techniques are applied to the data in order to uncover simple, possibly even closed form formula tests that can then be applied to new data, in order to categorize images, determine an expected price given a set of inputs, and in general, perform calculations for which there is otherwise no obvious, simple mathematical relationship between a set of inputs and a set of outputs. Though there could of course be exceptions, as a general matter, a fundamental component to any machine learning algorithm is a prior dataset from which information is extracted, and then applied to new data. This suggests that any algorithm rooted in machine learning is in some sense always dependent upon a human actor that will, at a minimum, select the prior data.

In its current form, the feature recognition algorithm doesn't make use of any prior data at all, but instead treats every image that it analyzes as a case of first impression. Nonetheless, it quickly produces high-quality partitions of images, identifying objectively distinct features within images (such as eyes, tires, bird beaks, and text). In short, it answers the question, "how should I partition an image that I know absolutely nothing about?" Another way to express the context in which this algorithm would be useful is when we don't know what we're looking for, but we know we should be looking for something.

2.2 The Information Entropy

In 1948, Claude Shannon introduced a remarkable theorem that relates the probability of a signal to its optimal code length, arguably inventing the entire discipline of information theory in the process.[3] In particular, Shannon showed that the optimal code length for a signal with a probability of p is $\log(\frac{1}{p})$. Taking the sum over the code lengths for each signal generated by a source, each weighted by the probability of the signal, we arrive at the **information entropy** of the source, which is a measure of the average information content per signal generated by the source.

Expressed as an equation, we have the following:

$$H = \sum_{i=1}^{n} p_i \log(\frac{1}{p_i}), \tag{1}$$

where p_i is the probability of signal i, and n is the total number of signals generated by the source.

The intuition for Equation (1) is straightforward: if we want to minimize the expected code length of a signal, then we should assign shorter codes to signals that occur frequently, and longer codes to signals that occur infrequently. Perhaps less obvious is the more general implication that low probability events carry more information than high probability events, assuming that we assign efficient codes to events.

Though originally intended for signals generated by sources over time, the same concepts can be applied to static systems that have fixed distributions. In particular, we can measure the information content of a distribution of colors within an image using this technique, which is a baked-in feature in both Matlab and Octave. That is, even though an image isn't an actual source that generates signals over time, it nonetheless has a distribution of colors, meaning that each color can be viewed as having a "probability" given by the number of instances of the color in question, divided by the total number of pixels in the image. By measuring the distribution of colors within an image, we can then measure the entropy of that distribution, which will give us the average information content of each pixel in the image. This does not mean that some pixels within an image require less storage than others, but rather, that an optimal encoding of the color distribution of the image would assign shorter codes to the most frequent colors, and longer codes to the least frequent colors.

[3]The original paper, *A Mathematical Theory of Communication*, is available here.

2.3 The Distribution of Information

If two regions within an image contain objectively distinct features, then they should have different color distributions, and therefore, different entropies. The converse of this observation implies that our expectation as to whether two regions contain objectively distinct features should be a function of the difference between the entropies of the two regions. Specifically, the ex ante probability that two regions contain two objectively distinct features should increase as a function of the difference in their respective entropies.[4]

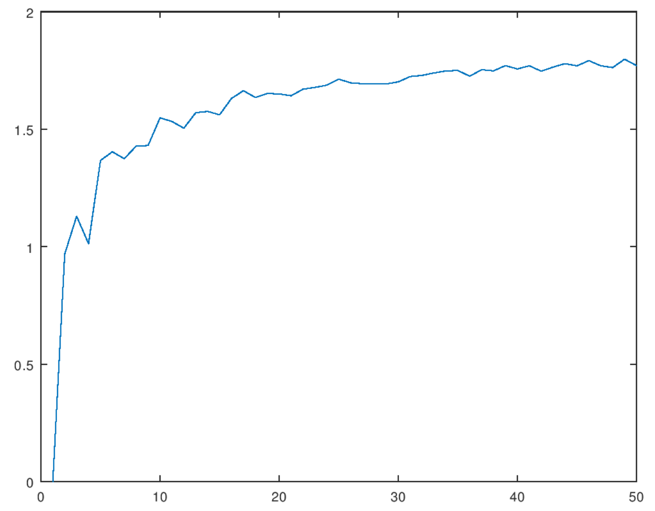

Figure 4: The standard deviation of the entropies as a function of N.

Beginning with this observation, as its first step, the algorithm partitions an image into $4, 9, 16, \ldots, N \times N$ equally sized regions, and tests the entropy of each resulting region. It then chooses the value of N for which the standard deviation of the entropies of the regions is maximized. That is, the algorithm partitions an image into smaller and smaller equally sized regions, until it finds the number of regions that maximizes the standard deviation of the entropies of the regions. This process is implemented by a subroutine that terminates the moment it finds a local maximum, with a worst-case run time of $O(m \log(m))$,

[4]Note that the entropy of a color distribution is not sensitive to the actual colors within an image, but only the distribution of those colors. So, for example, if we swap the positions of the blue pixels and green pixels in an image, the entropy of the image would be unchanged. In general, the entropy of an image is invariant under rotation, and any other operation that does not affect the distribution of colors, even if it changes the actual colors in the image.

where m is the number of pixels in the image.[5] In the case of the photo in Figure 1, the subroutine returned a value of $N = 11$, which is a local maximum. To give a bit more context, Figure 4 shows the standard deviation of the entropies of the regions of the photo shown in Figure 1 as a function of the number of subdivisions, for all $N \leq 50$. Note that we would have to iterate through a large number of possible values of N in order to be certain that we've found a global maximum, but this is unnecessary, and even counterproductive, because as a practical matter, the region sizes produced by low, local maxima are generally perfect for identifying macroscopic objects in images.[6]

So to summarize, the assumption underlying this first step of the feature recognition algorithm is that objectively distinct physical features within an image should contain different amounts of information, and therefore, have different entropies. By partitioning an image into a number of regions that maximizes the standard deviation of the entropies of the regions, we maximize the ex ante probability that each region will contain an objectively distinct feature.

2.4 Notability and Information

It might be tempting to assume that the regions with the most information will be the most notable, or somehow the most likely to contain a feature. However, counterexamples to this hypothesis can be constructed easily. That is, an image could have a complex background, and a simple subject. For example, imagine someone dressed in black in front of a Pollock painting. In that case, the background would probably contain more color information than the subject. As a result, we instead look to the variance of the entropy of a region as a proxy for its notability. That is, we calculate the square of the difference between (x) the average entropy over all regions, less (y) the entropy of the region in question. The greater this variance, the more of an outlier the region is in terms of its information content. We then build a distribution of the variance across all regions, which we use to determine how notable a region is by using this distribution to calculate the probability that another region has a lower variance than the region in question. A score of approximately 1 implies

[5] This subroutine makes exponentially increasing, or decreasing, guesses for the local maximum, ensuring that it never iterates more than $\log(m)$ times. Each test of the standard deviation requires a number of operations that is proportional to the number of pixels in the image, resulting in a total run time that is $O(m \log(m))$. However, the algorithm calls Matlab's entropy function for each region, which can, for large images, start to produce lag on a consumer device. If this becomes an issue, either due to the low quality of the machine, or the high resolution of the photo, we can instead approximate the standard deviation of the entropies by calculating the entropies for some representative subset of the regions.

[6] I have also developed a different version of this subroutine that produces much more fine-grained partitions of images, in turn producing detailed feature boundaries. This algorithm underlies another set of algorithms that are able to quickly extract three-dimensional depth information from two-dimensional images. See the research note entitled, "Fast Unsupervised 3-D Feature Extraction".

that nearly all of the other regions in the image have a lower variance than the region in question, and therefore, the region is probably notable. A score of approximately 0 implies that the region is probably not notable.

2.5 Background and Foreground

The method above works well at distinguishing between features within an image, but it does not necessarily distinguish between background and foreground. That is, even though the algorithm above will generally distinguish between subject and context, it doesn't tell us which is which. As a result, we make use of an additional test that measures how sensitive each region is to the removal of a single color, which we take as a proxy for distinguishing between the background and foreground of an image.

We begin by producing a temporary copy of each region within the image. We then reduce the number of colors in each region to 8, simply by multiplying the color matrix by scalars and taking the floor of the resultant matrix. This forces all of the colors within a given region into one of 8 buckets of colors. We then find the most frequent color for each region, and use the number of instances of this color divided by the total number of pixels in the region as a measure of how dominated the region is by that single color, which is really a band of colors that have all been forced into a single "bucket". We take this measure as a proxy for how likely the region is to be in the background. We then build a distribution of this measure across all regions, which we use to determine how likely a region is to be part of the background, by using this distribution to calculate the probability that a region is more sensitive to the removal of a single color than the region in question.

2.6 Identifying Notable Foreground Features

The greater the difference between the highest foreground probability and lowest foreground probability within a given image, the more confident we can be that there's a meaningful difference between the two regions that generated those probabilities. That is, if the difference between the probabilities is large, then we can be more confident in our selection of one of the two regions as more likely to be a foreground feature than the other. The same reasoning applies to the notability probability. Note that we're not really interested in the variance of the probabilities, but instead the spread between the largest probabilities and the smallest probabilities, since this is ultimately what will allow us to confidently separate wheat from chaff, and select regions as most likely to be notable foreground features. That is, as a general matter, the greater the difference between the largest and smallest probabilities across all regions, the more confident we can be in our identification of notable foreground features.

This difference is measured by a subroutine, which sorts a list, and then calculates the square of the difference between the largest and smallest entries in the list, the second largest and second smallest entries, and so on. We calculate this measure for both sets of probabilities across all regions, and then calculate a single weighted probability, which is named "symm", which I'll discuss in more detail below.

2.7 Assembling Regions into Larger Features

Because the process described above divides an image into equally sized regions, it's almost certainly going to subdivide what are in fact single features of the image. As a result, we need a process for reassembling these regions into larger contiguous features. This requires that we decide how similar two regions need to be in order to be combined, and how we measure similarity in the first instance. This process underlies all of the algorithms I'll discuss in this paper, and is not unique to the feature recognition algorithm. As a result, the feature recognition algorithm is really just a special case of a more general algorithm that constructs categories of mathematical objects based upon assumptions rooted in information theory.

The approach I take is to begin with an **anchor** region that is selected according to its weighted probability of being a notable foreground feature, with the highest probability region selected first, the next highest probability region after that, and so on, and then testing each of the four neighbors of that anchor region (up, down, left, and right) to see whether the notability score of each region is within some delta of the anchor. If so, that region is then placed in a queue, and all regions in the queue are subsequently analyzed similarly, by comparing neighbors of the regions in the queue to the anchor. If sufficiently similar regions are found, this will eventually generate a contiguous set of regions that together should contain a single feature, or part of a single feature.

The algorithm is obviously quite sensitive to the value of delta. If delta is zero, then the algorithm will not assemble any regions at all, and will simply return the original partition of the image described above, since it will in this case distinguish between every region. If delta is maximized, then the algorithm will return one giant feature - i.e., the entire image - since it will treat all regions as sufficiently similar to be reassembled. Somewhere in between these two extremes there is a value of delta that will generate a partition that assembles regions that actually contain the same features. Given that delta is a real number, there could of course be more than one value of delta that produces a reasonably correct partition. In fact, there could be multiple reasonable partitions of the image generated by different values of delta. However, there is an asymmetry to our selection of delta, in that choosing a value of delta that is smaller than one of the correct values of delta will not produce an incorrect partition, but will instead produce a partition that is unnecessarily subdivided. In contrast, choosing a value of delta that is larger than the maximum correct value of delta will produce an incorrect partition, since it will join regions together that do not contain the same features. As a result, it is rational to be more conservative in our selection of delta, since this should produce better partitions in general, understanding that we might miss out on optimal partitions in specific cases. That is, this algorithm is designed to always produce a quality result, rather than occasionally produce the best result at the risk of generating bad results in general. We can accomplish this by beginning with a small value of delta, and iterating up through larger values, and running the assemblage algorithm for each of these values of delta.

The assemblage algorithm generates a matrix for each iteration, that I call the **region matrix** (see Figure 5 below). This matrix is populated with the numbers assigned to the regions by the assemblage algorithm.

$$\begin{bmatrix}
76 & 90 & 0 & 0 & 71 & 70 & 46 & 41 & 69 & 61 & 39 \\
88 & 86 & 0 & 0 & 72 & 56 & 14 & 8 & 68 & 59 & 37 \\
84 & 81 & 89 & 89 & 67 & 27 & 21 & 13 & 68 & 58 & 35 \\
63 & 79 & 87 & 73 & 62 & 38 & 24 & 15 & 64 & 55 & 33 \\
85 & 66 & 0 & 75 & 44 & 17 & 19 & 77 & 64 & 53 & 29 \\
78 & 45 & 80 & 54 & 42 & 18 & 3 & 16 & 34 & 52 & 26 \\
49 & 47 & 65 & 28 & 20 & 1 & 7 & 22 & 11 & 51 & 25 \\
0 & 0 & 94 & 0 & 93 & 6 & 4 & 30 & 9 & 48 & 23 \\
0 & 0 & 0 & 0 & 91 & 40 & 2 & 32 & 5 & 43 & 23 \\
0 & 0 & 0 & 0 & 91 & 83 & 82 & 31 & 10 & 57 & 36 \\
0 & 0 & 0 & 95 & 0 & 83 & 92 & 74 & 50 & 12 & 60
\end{bmatrix}$$

Figure 5: The region matrix for the photo in Figure 1.

Figure 5 contains the region matrix ultimately generated for the photo in Figure 1. The 0 entries in the region matrix all have a zero probability of being notable foreground features, and are therefore never selected as anchors. As a result, the 0 feature should contain the background of the image, and is not necessarily contiguous, since it is everything that is left over by the algorithm. In this specific case, most of the 0 entries correspond to regions of the empty sidewalk on the bottom left-hand side of the photo in Figure 1, which in turn correspond to the black regions in Figure 2. Figure 6 contains regions 3, 4, and 7, which, despite sharing a boundary, the algorithm nonetheless treated as separate regions. As you can plainly see, each region contains different amounts of color information, which in this case, corresponds nicely to regions that contain three distinct subjects.

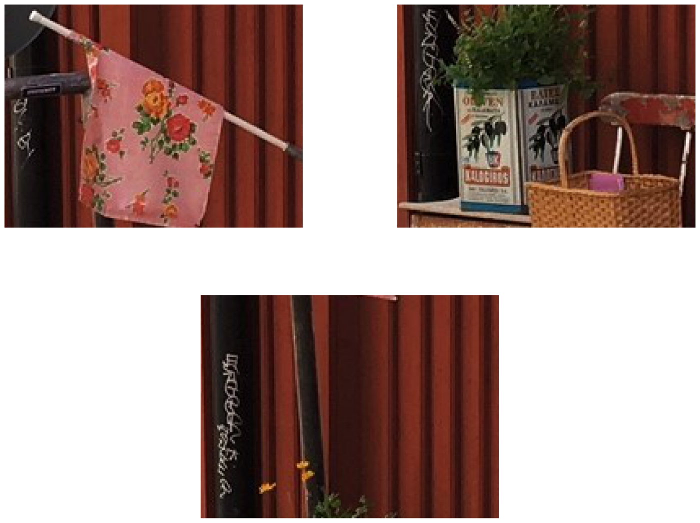

Figure 6: Three neighboring regions distinguished by the feature recognition algorithm.

This region matrix is the final result produced by the algorithm, generated by using the "correct" value of delta. But as mentioned above, the algorithm iterates through multiple values of delta, beginning with a small value, and iterating up through larger values. This means that we have to come up with appropriate minimum and maximum values for delta to iterate through.

As discussed above, we measure similarity by comparing the notability score of two regions. Note that as the standard deviation of the set of notability scores increases, the expected difference between the notability score of any two regions also increases. As a result, the equation for delta is proportional to the standard deviation of the set of notability scores. However, as mentioned above, the spread between the largest and smallest scores for each region being a notable foreground region is really the key determinant as to whether we can confidently separate an image into background and foreground. As a result, our selection of delta should also depend upon the variable symm we discussed above. Specifically, a small value of symm implies that there isn't much of a difference between background and foreground in our image, in turn implying that we need to be careful in distinguishing between regions. Similarly, a high value of symm implies that we can be a bit more aggressive in assembling regions.

To reconcile all of this, the first iteration of the assemblage algorithm uses a value of delta given by $\Delta = \frac{s}{divisor}$, where s is the standard deviation of the set of notability scores, and,

$$divisor = \frac{N^{N(1-symm)}}{10}, \tag{2}$$

where N^2 is the number of regions the image was partitioned into using the first step of the algorithm described above. That is, as the number of regions in the image grows, our initial choice of delta is reduced exponentially. This is because a high value of N implies that the image contains a large number of small regions that contain disparate amounts of information, in turn implying that we should be cautious in reassembling them back into larger regions. Similarly, a low value of symm implies that there isn't much of a difference between background and foreground, again causing our initial selection of delta to be reduced exponentially. The "10" is simply the product of experimentation and observation.

There is also the question of the appropriate maximum value of delta, and of course, the question of how we choose the "correct" region matrix from the set of matrices generated by iterating through different values of delta. We begin by addressing the appropriate maximum value of delta, which will inform our selection of the correct region matrix. Because the region matrix itself consists of integers that have some frequency, it will have an entropy. That is, the number of instances of an integer within the matrix divided by the size of the matrix can be viewed as a probability, which in turn will have some optimal code length. Further, note that before we assemble any of the regions, each region will have its own unique number assigned to it. As a result, if the first step of the algorithm partitions the image into N^2 regions, then the region matrix will initially consist of N^2 unique integers, each with a frequency of $\frac{1}{N^2}$. This implies that the initial entropy of the region matrix is $\log(N^2)$, which is the maximum entropy for a matrix of size $N \times N$. As a result, as we assemble regions into features, we will reduce the entropy of the region matrix. In the extreme case where the region matrix consists of a single feature, the entropy of the region matrix will be 0.

In short, by measuring the entropy of the region matrix, we can restate the initial problem discussed above in purely mathematical terms: by distinguishing between everything, we maximize entropy; by distinguishing between nothing, we minimize entropy. This means that finding the optimal partition of the image can be stated in terms of finding the region matrix that has the optimal entropy, which is by definition somewhere in between $H_{max} = \log(N^2)$ and 0.

Whatever our maximum value for delta is, it should not generate a region matrix that has an entropy of 0. That is, we should stop the algorithm once we end up producing a region matrix that contains a single feature. Ideally, we should probably stop long before we get to this point, unless we're very confident that the image contains a small number of very large, sprawling features. As a result, we use divisor, which is a measure of confidence in the distinction between background and foreground, to set the minimum acceptable entropy for the region matrix, which we use as a stopping condition for the algorithm. That is, once we breach the minimum acceptable entropy, the algorithm stops.

We set the minimum acceptable entropy to the following:

$$H_{min} = .47H_{max} + .53(1 - \frac{25}{divisor})H_{max}. \quad (3)$$

This equation is the product of both theory and experimentation, and so there are probably other formulas that will work just fine. The intuition underlying this particular equation is that as divisor increases, we're less confident in the distinction between background and foreground, and therefore, we increase the minimum required entropy. That is, if divisor is high, then we require the region matrix to have an entropy that is very close to the maximum possible entropy. This means that the algorithm will not assemble large, sprawling features, but will instead produce a large number of small features, since we are not confident in our distinction between background and foreground. In contrast, if divisor is low, then the region matrix can have a lower entropy, since we can be more confident in our distinction between background and foreground, allowing for large, sprawling features.

So in short, if we're not confident in the distinction between background and foreground, then we're going to begin with a very small value of delta, and stop the algorithm before it can produce a matrix that has large features. If we're very confident in the distinction between background and foreground, then we're going to begin with a larger value of delta, and allow the algorithm to run longer, allowing for the possibility of large, sprawling features.

We then select the correct value of delta by choosing the value of delta that causes the maximum change to the entropy of the region matrix. That is, as we iterate through values of delta, we measure the change in entropy as a function of delta, and choose the value of delta for which this rate of change is maximized. This is the value of delta that unlocked the greatest change in the structure of the partition. As noted above, given the natural asymmetry of this process, it is rational to be conservative and choose the smallest reasonable value of delta in scope. Consistent with this approach, we choose the value of delta that unlocks the greatest change in the structure of the region matrix, and ignore all incremental changes that occur after that point. It turns out that as a general matter, this approach consistently produces great partitions, uncovering actual objects in images. This part of the algorithm never iterates more than a fixed number of times, which I have set to 25, and as a result, the entire feature recognition algorithm has a run time that is $O(m \log(m))$.

The assumption underlying both the image feature recognition algorithm and the categorization algorithm is that the value of delta for which the rate of change in entropy is maximized is the value of delta for which the greatest amount of structure in the underlying object is revealed. Therefore, this is the value of delta that is best suited for distinguishing between the components of an object when no other prior information is available. This approach allows for an image, or a dataset, to generate its own value of delta, and therefore, generate its own context, which in turn allows for totally unsupervised pattern recognition with no prior information.[7]

3 Vectorized Categorization and Prediction

In a research note entitled, "Fast N-Dimensional Categorization Algorithm", I presented a categorization algorithm that can categorize a dataset of N-dimensional vectors with a worst-case run time that is $O(log(m)m^{N+1})$, where m is the number of items in the dataset, and N is the dimension of each vector in the dataset. The categorization algorithm I'll present in this article is identical to the algorithm I presented in the research note, except this algorithm skips all of the steps that cannot be vectorized. As a result, the vectorized categorization algorithm has a worst-case run time that is $O(m^2)$, where m is the number of items in the dataset. The corresponding prediction algorithm is basically unchanged, and has a run time that is worst-case $O(m)$. This means that even if our underlying dataset contains millions of observations, if it is nonetheless possible to structure these observations as high-dimensional vectors, then we can use these algorithms to produce nearly instantaneous inferences, despite the volume of the underlying data.

As a practical matter, on an ordinary consumer device, the dimension of the dataset will start to impact performance for $N \approx 10000$, and as a result, the run time isn't truly independent of the dimension of the dataset, but will instead depend upon on how the vectorized processes are implemented by the language and machine in question. Nonetheless, the bottom line is that these algorithms allow ordinary, inexpensive consumer devices to almost instantaneously draw complex inferences from millions of observations, giving ordinary consumer devices access to the building blocks of true artificial intelligence.

[7]This method uses the entropy of a mathematical object as a tractable measure of its complexity. We could of course use some other measures of complexity, and then measure the rate of change in that complexity, but the entropy is convenient because it is tractable, and a built-in feature of both Matlab and Octave. As a general matter, the core concept underlying these algorithms is that we measure the amount of structure in an object that is revealed over each iteration by measuring the change in the complexity of the object over each iteration.

3.1 Predicting Random Paths

Let's begin by predicting random-walk style data. This type of data can be used to represent the states of a complex system over time, such as an asset price, a weather system, a sports game, the click-path of a consumer, or the state-space of a complex problem generally. As we work through this example, I'll provide an overview of how the algorithms work, in addition to discussing their performance and accuracy.

The underlying dataset will in this case consist of two categories of random paths, though the categorization algorithm and prediction algorithm will both be blind to these categories. Specifically, our training data consists of 1,000 paths, each of which contains 10,000 points, for a total of 10,000,000 observations. Of those 1,000 paths, 500 will trend upward, and 500 will trend downward. The index of the vector represents time, and the actual entry in the vector at a given index represents the y value of the path at that time. We'll generate the upward trending paths by making the y value slightly more likely to move up than down as a function of time, and we'll generate the downward trending paths by making the y value slightly more likely to move down than up as a function of time.

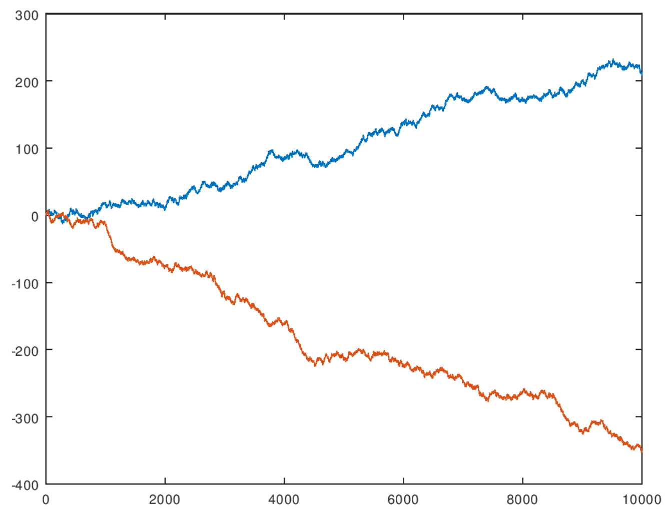

Figure 7: Two random paths.

For context, Figure 7 shows both a downward trending path and an upward trending path, each taken from the dataset. Figure 8 shows the full dataset of 1,000 paths, each of which begins at the origin, and then traverses a random path over 10,000 steps, either moving up or down at each step. The upward trending paths move up with a probability of .52, and the downward trending paths move up with a probability of .48.

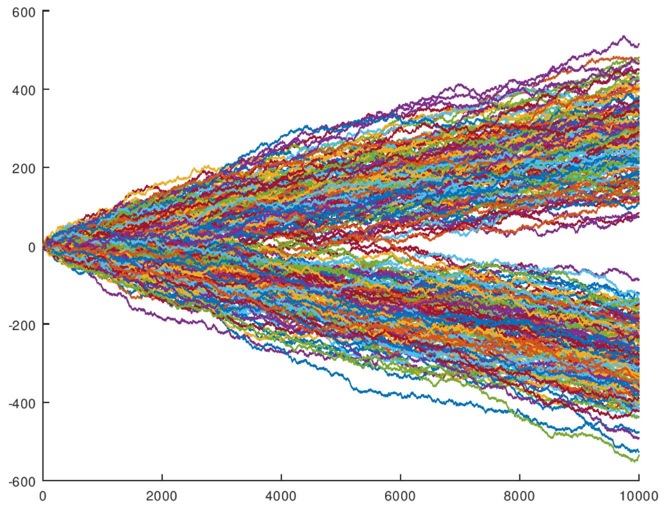

Figure 8: The dataset of random paths.

After generating the dataset, the next step is to construct two data trees that we'll use to generate inferences. One data tree is comprised of elements from the original dataset, called the **anchor tree**, and the other data tree is comprised of threshold values, called the **delta tree**.

The anchor tree is generated by repeatedly applying the categorization algorithm to the dataset. This will produce a series of subcategories at each depth of application. The top of the anchor tree contains a single nominal vector, and is used by the algorithm for housekeeping. The categories generated by the first application of the categorization algorithm have a depth of 2 in the anchor tree; the subcategories generated by two applications of the categorization algorithm have a depth of 3 in the anchor tree, and so on. The algorithm selects an anchor vector as representative of each subcategory generated by this process. As a result, each anchor vector at a depth of 2 in the anchor tree represents a category generated by a single application of the categorization algorithm, and so on.

As a simple example, assume our dataset consists of the integers $\{1, 2, 8, 10\}$. In this case, the first application of the categorization algorithm generates the categories $\{1, 2\}\{8\}\{10\}$, with the anchors being 1, 8, and 10. This is the actual output of the categorization algorithm when given this dataset, and it turns out that, in this case, the algorithm does not think that it's appropriate to generate any deeper subcategories. For a more detailed explanation of how this whole process works, see my research note entitled, "Predicting and Imitating Complex Data Using Information Theory".

As a result, in this case, the anchor tree will contain the nominal vector at its top, and then at a depth of 2, the anchors 1,8, and 10 will be positioned in separate entries, indexed by different widths.

Represented visually, the anchor tree is as follows:

(Inf)

(1) (8) (10)

Each application of the categorization algorithm also produces a value called delta, which is the maximum difference tolerated for inclusion in a given category. This is the same value used by the feature recognition algorithm, except in this case, we're using this value to distinguish between abstract vectors, and not images. Specifically, if x is in a category represented by the anchor vector a, then it must be the case that $||a - x|| < \delta_a$, where δ_a is the sufficient difference associated with the category represented by a. That is, δ_a is the difference between two vectors which, when met or exceeded, we treat as sufficient cause to categorize the vectors separately. The value of delta is determined by iterating through different values of delta, and choosing the particular value of delta that maximizes the change in the entropy of the categorization. As a result, delta is the natural, in-context difference between elements of the category in question. The value δ_a has the same depth and width position in the delta tree that the associated anchor vector a has in the anchor tree. In the example given above, delta is 1.0538. So in our example, 1 and 2 are in the same category, since $|1 - 2| < 1.0538$, but 8 and 10 are not, since $|8 - 10| > 1.0538$.

Since the categorization algorithm is applied exactly once in this case, only one value of delta is generated, resulting in the following delta tree:

(Inf)

(1.0538) (1.0538) (1.0538)

Together, these two trees operate as a compressed representation of the subcategories generated by the repeated application of the categorization algorithm, since we can take a vector from the original dataset, and quickly determine which subcategory it could belong to by calculating the difference between that vector and each anchor, and testing whether it's less than the applicable delta. This allows us to approximate operations on the entire dataset using only a fraction of the elements from the original dataset. Further, we can, for this same reason, test a new data item that is not part of the original dataset against each anchor to determine to which subcategory the new data item fits best. We can also test whether the new data item belongs in the dataset in the first instance, since if it is not within the applicable delta of any anchor, then the new data item is not a good fit for the dataset, and moreover, could not have been part of the original dataset. Finally, given a vector that contains M out of N values, we can then predict the $N-M$ missing values by substituting those missing values using the corresponding values in the anchors in the anchor tree, and then determining which substitution minimized the difference between the resulting vector and the anchors in the tree.

For example, if $N = 5$, and $M = 3$, then we would substitute the two missing values in the input vector $x = (x_1, x_2, x_3, \ldots)$, using the last two dimensions of an anchor vector $a = (a_1, a_2, a_3, a_4, a_5)$, producing the prediction vector $z = (x_1, x_2, x_3, a_4, a_5)$. We do this for every anchor in the anchor tree, and test whether $||a - z|| < \delta_a$. If the norm of the difference between the anchor vector and the prediction vector is less than the applicable delta, then the algorithm treats that prediction as a good prediction, since the resulting prediction vector would have qualified for inclusion in the original dataset. As a result, all of the predictions generated by the algorithm will contain all of the information from the input vector, with any missing information that is to be predicted taken from the anchor tree.

This implies that our input vector x could be within the applicable delta of multiple anchor vectors, thereby constituting a match for each related subcategory. As a result, the prediction algorithm returns both a single best-fit prediction, and an array of possible-fit predictions. That is, if our input vector produces prediction vectors that are within the applicable delta of multiple anchors, then each such prediction vector is returned in the array of possible fits. There will be, however, one fit for which the difference $||a - z||$ between the input vector and the applicable prediction vector is minimized, and this is returned by the algorithm as the best-fit prediction vector.

Returning to our example, each path is treated as a 10,000 dimensional vector, with each point in a path represented by a number in the vector. So in this case, the first step of the process will be to generate the data trees populated by repeated application of the categorization algorithm to the dataset of path vectors. Because the categorization algorithm is completely vectorized, this process can be accomplished on an ordinary consumer device, despite the high dimension of the dataset, and the relatively large number of items in the dataset. Once this initial step is completed, we can begin to make predictions using the data trees.

We'll begin by generating a new, upward trending path, that we'll use as our input vector, that again consists of $N = 10000$ observations. Then, we'll incrementally increase M, the number of points from that path that are given to the prediction algorithm. This will allow us to measure the difference between the actual path, and the predicted path as a function of M, the number of observations given to the prediction algorithm. If $M = 0$, giving the prediction algorithm a completely empty path vector, then the prediction algorithm will have no information from which it can draw inferences. As a result, every path in the anchor tree will be a possible path. That is, with no information at all, all of the paths in the anchor tree are possible.

In this case, the categorization algorithm compressed the 1,000 paths from the dataset into 282 paths that together comprise the anchor tree. The anchor tree has a depth of 2, meaning that the algorithm did not think that subdividing the top layer categories generated by a single application of the categorization algorithm was appropriate in this case. This implies that the algorithm didn't find any local clustering beneath the macroscopic clustering identified by the first application of the categorization algorithm. This in turn implies that the data is highly randomized, since each top layer category is roughly uniformly diffuse, without any significant in-category clustering.

This example demonstrates the philosophy underlying these algorithms, which is to first compress the underlying dataset using information theory, and then analyze the compressed dataset efficiently using vectorized operations. In this case, since the trees have a depth of 2 and a width of 282, each prediction requires at most $O(282)$ vectorized operations, despite the fact that the inferences are ultimately derived from the information contained in 10,000,000 observations.

Beginning with $M = 0$ observations, each path in the anchor tree constitutes a trivial match, which again can be seen in Figure 8. That is, since the algorithm has no observations with which it can generate an inference, all of the paths in the anchor tree are possible. Expressed in terms of information, when the input information is minimized, the output uncertainty is maximized. As a result, the prediction algorithm is consistent with common sense, since it gradually eliminates possible outcomes as more information becomes available.

Note that we can view the predictions generated in this case as both raw numerical predictions, and more abstract classification predictions, since each path will be either an upward trending path, or a downward trending path. I'll focus exclusively on classification predictions in another example below, where we'll use the same categorization and prediction algorithms to predict which class of three-dimensional shapes a given input shape belongs to.

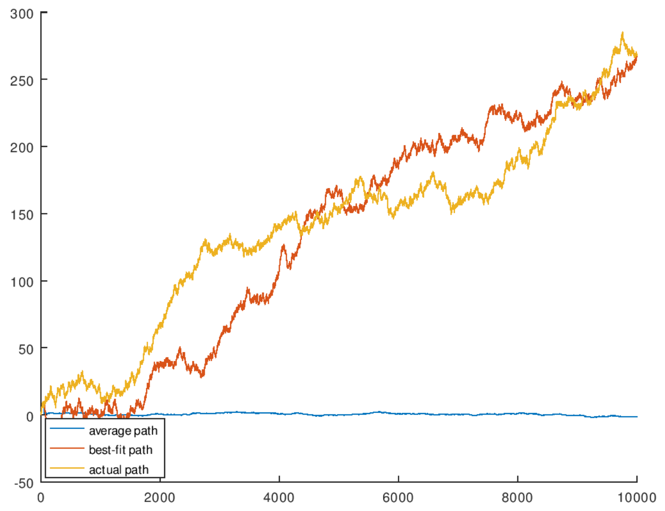

Figure 9: The actual path, average path, and best-fit path for $M = 50$.

Returning to the example at hand, I've set $M = 50$, and Figure 9 shows the actual path of the new input vector, the average path generated by taking the simple average of all of the predicted paths, and the best-fit path. In this case, the prediction algorithm has only .5 percent of the path information, and not surprisingly, the average of all predicted paths is flat, since the array of predicted paths contains paths that trend in both directions. In fact, in this case, the array of predicted paths contains the full set of 282 possible paths, implying that the first 50 points in the input path did not provide much information from which inferences can be drawn, though the best-fit path in this case does point in the right direction. This is not surprising, since generating the path requires new information to be generated at each point in the path, which is supplied by calling a random number generator that in turn determines whether the next point in the path is up or down, relative to the current path.

As a result, each point in the path conveys new information about the shape of the path. In contrast, in the case of an entirely deterministic curve, once the initial conditions of the curve are known, the remainder of the curve can be generated without any additional information, other than the underlying equation that generates the curve. As we'll see in the following sections, greater compression and accuracy can be achieved when predicting deterministic data, even though we're not making use of any interpolation. This in turn suggests that the compression ratio achieved by the categorization algorithm could be a measure of the complexity of the dataset, though I have not empirically tested this hypothesis.[8]

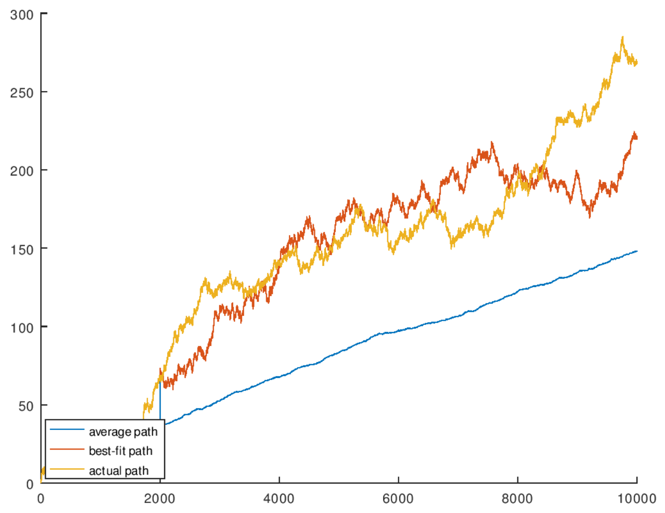

Figure 10: The actual path, average path, and best-fit path for $M = 2000$.

For $M = 2000$, the actual path, average path, and best-fit path, all clearly trend in the same, correct direction, as you can see in Figure 10. Figure 11 contains the full array of predicted paths for $M = 2000$, which consists of 173 possible paths. Though the average path, and best-fit path both point in the correct, upward trending direction, the set of possible paths clearly contains a substantial number of downward trending paths.

[8]To appreciate the intuition for the hypothesis, consider the extreme case where no compression is achieved, producing an anchor tree populated with the entire dataset. This implies that there was no clustering in the data. It also suggests that each data point contributes no information about the other data points in the dataset, and as a result, new data that is consistent with the dataset probably cannot be predicted using the information in the dataset, since it is comprised of data points that provide no information about each other. In the other extreme case, total compression is achieved, producing an anchor tree with a single value from the dataset, suggesting that the entire dataset is tightly clustered around a single data point. In this case, new data that is consistent with the underlying dataset will also be tightly clustered around that single anchor, in turn making prediction trivial. In contrast, the Shannon entropy does not consider structure information at all, and looks only to frequency, making it a poor measure of computational complexity. For example, the set $\{1, 1, 1, 2, 2, 2\}$ has a uniform statistical distribution, but an obvious structure. As a result, the Shannon entropy of the set is maximized, whereas the categorization algorithm compresses the set to two categories, achieving significant compression. I'm reluctant in taking this observation too far, since taken to the extreme, it suggests that the categorization algorithm could in some cases operate as a computable approximation of the Kolmogorov complexity, which is otherwise non-computable.

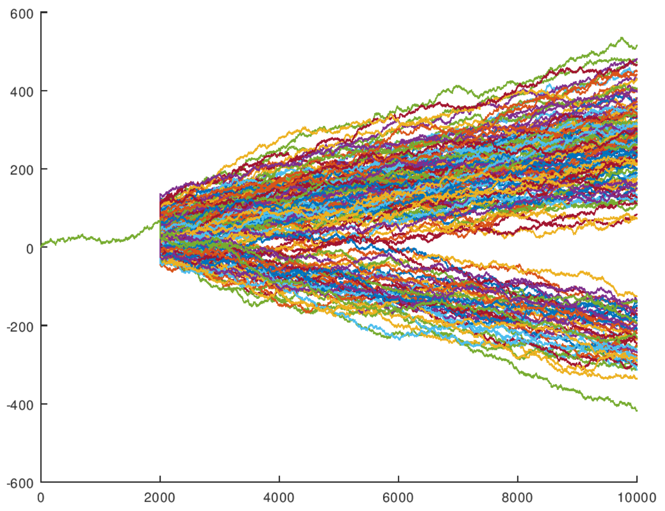

Figure 11: The full array of predicted paths for $M = 2000$.

Figure 12 shows the average error between the best-fit path and the actual path as a percentage of the y value in the actual path, as a function of M. In this case, Figure 12 reflects only the portion of the data actually predicted by the algorithm, and ignores the first M points in the predicted path, since the first M points are taken from the input vector itself. That is, Figure 12 reflects only the $N - M$ values taken from the anchor tree, and not the first M values taken from the input vector itself. Also note that the error percentages can and do in this case exceed 100 percent for low values of M.

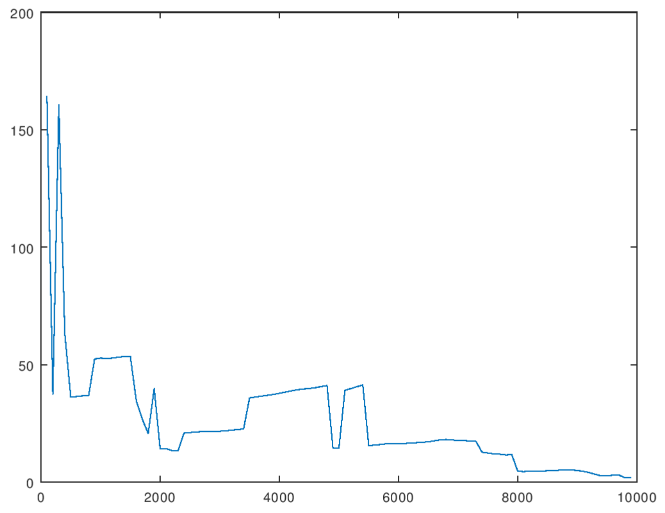

Figure 12: The average error as a function of M for a single input vector.

As expected, the best-fit path converges to the actual path as M increases, implying that the quality of prediction increases as a function of the number of observations. Figure 13 shows the same average error calculation for 25 randomly generated input paths.

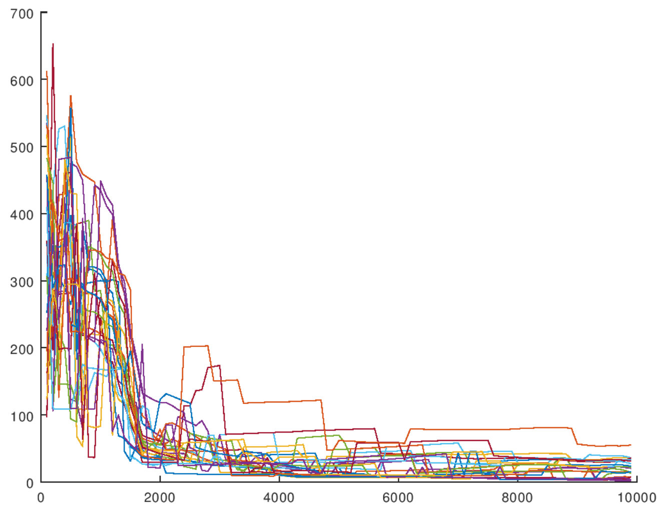

Figure 13: The average error as a function of M for 25 input vectors.

As noted above, the prediction algorithm is also capable of determining whether a given input vector is a good fit for the underlying data in the first instance. In this case, this means that the algorithm can determine whether the path represented by an input vector belongs to one of the categories in the underlying dataset, or whether the input vector represents some other type of path that doesn't belong to the original dataset. For example, if we generate a new input vector produced by a formula that has a .55 probability of decreasing at any given moment in time, then for $M \approx 7000$, the algorithm returns the nominal vector at the top of the anchor tree as its predicted vector, indicating that the algorithm thinks that the input vector does not belong to any of the underlying categories in the anchor tree.

This ability to not only predict outcomes, but also identify inputs as outside of the scope of the underlying dataset helps prevent bad predictions, since if an input vector is outside of the scope of the underlying dataset, then the algorithm won't make a prediction at all, but will instead flag the input as a poor fit. This also allows us to expand the underlying dataset by providing random inputs to the prediction function, and then take the inputs that survive this process as additional elements of the underlying dataset. For more on this approach, see my research note entitled, "Predicting and Imitating Complex Data Using Information Theory".

3.2 Higher Dimensional Spaces

3.2.1 Predicting Projectile Paths

The same algorithms that we used in the previous section to predict paths in a two-dimensional space can be applied to an N-dimensional space, allowing us to analyze high-dimensional data in high-dimensional spaces. We'll begin by predicting somewhat randomized, but nonetheless Newtonian projectile paths in three-dimensional space.

Our training data consists of 500 projectile paths, each containing 3000 points in Euclidean three-space, resulting in vectors with a dimension of $N = 9000$. The equations that generated the paths in the training data are Newtonian equations of motion, with bounded, but randomly generated x and y velocities, no z velocity other than downward gravitational acceleration, fixed initial x and y positions, and randomly generated, but tightly bounded initial z positions.

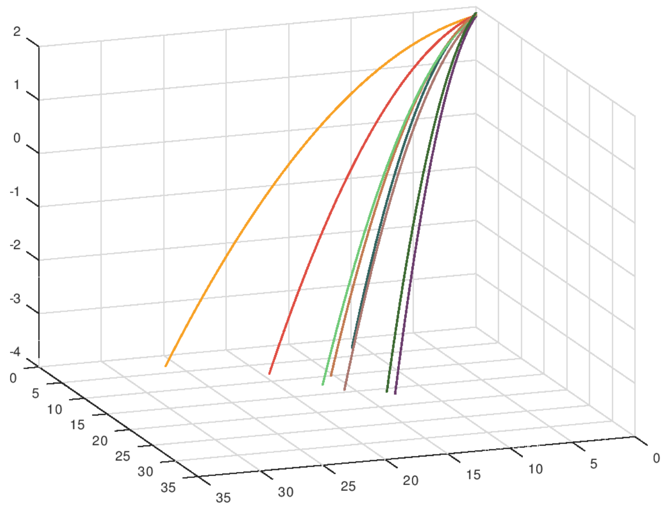

Figure 14: Projectile paths from the dataset.

Note that in this case, we're predicting a path that has a simple closed form formula without using any interpolation. Instead, the algorithm uses the exact same approach outlined above, which compresses the dataset into a series of anchor vectors, which in this case represent Newtonian curves, and then takes a new curve and attempts to place it in the resulting anchor tree of vectors. The anchor tree has a top level width of 193, suggesting that, despite the fact that the individual curves are highly structured, the dataset of curves is nonetheless quite randomized. As a result, the dataset consists of a fairly randomized collection of highly structured curves.

Just as we did above, first we'll generate a new curve, which will be represented as a 9,000 dimensional vector that consists of 3,000 points in Euclidean three-space. Then, we'll incrementally provide points from the new curve to the prediction algorithm, and measure the accuracy of the resultant predictions. Figure 15 shows the average path, best-fit path, and actual path for $M = 1750$.

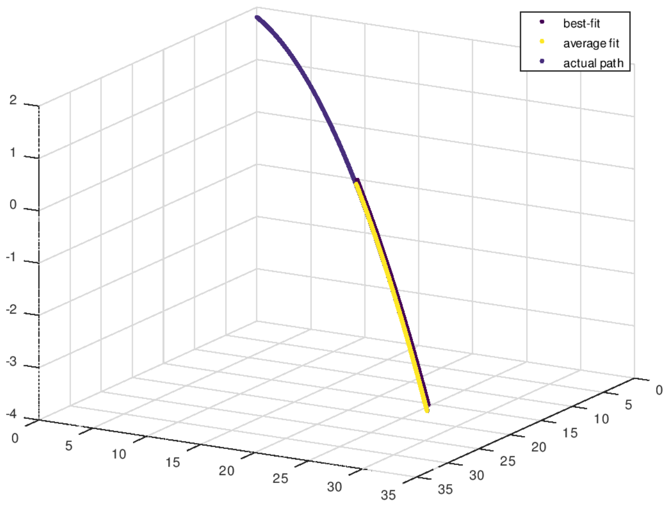

Figure 15: The average path, best-fit path, and actual path for $M = 1750$.

In this case, the percentage-wise average error between the predicted path and the actual path, shown in Figure 16, is much smaller than it was for the random path data. This is not surprising, given that each curve has a simple shape that is entirely determined by its initial conditions. As a result, so long as there is another curve in the anchor tree that has reasonably similar initial conditions, we'll be able to accurately approximate, and therefore, predict the shape of the entire input curve. In contrast, the random path data is generated by what is in reality a series of 10,000 initial conditions that are sensitive to the order in which they occur. As a result, the random path dataset literally contains more information than the projectile dataset, and therefore, we should expect to use a much larger dataset if we'd like to predict random paths to the level of precision achieved in this example, using only a modestly sized dataset.

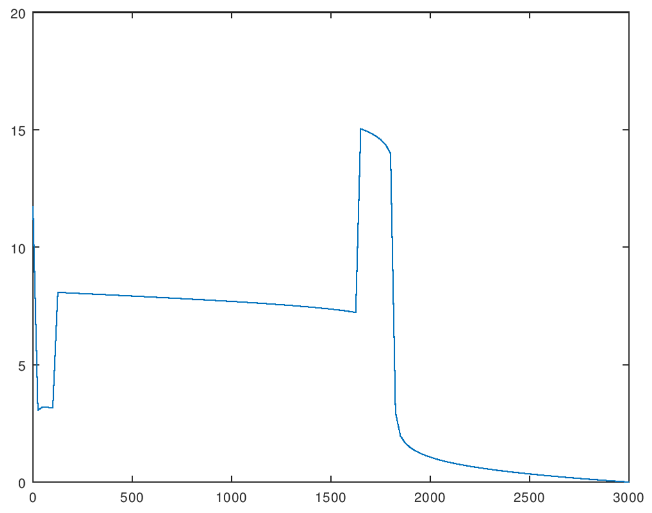

Figure 16: The average error as a function of M.

3.2.2 Shape and Object Classification

We can use the same techniques to categorize three-dimensional objects. We can then take the point data for a new object and determine to which category it belongs best by using the prediction algorithm. This allows us to take what is generally considered to be a hard problem, i.e., three-dimensional object classification, and reduce it to a problem that can be solved in a fraction of a second on an ordinary consumer device.

In this case, the training data consists of the point data for 600 objects, each shaped like a vase, with randomly generated widths that come in three classes of sizes: narrow, medium, and wide. That is, each class of objects has a bounded width, within which objects are randomly generated. There are 200 objects in each class, for a total of 600 objects in the dataset. Each object contains 483 points, producing a vector that represents the object with a total dimension of $N = 3 \times 483 = 1449$. Figure 17 shows a representative image from each class of object. Note that the categorization and prediction algorithm are both blind to the classification labels in the data, which are hidden in the $N + 1$ entry of each shape vector.

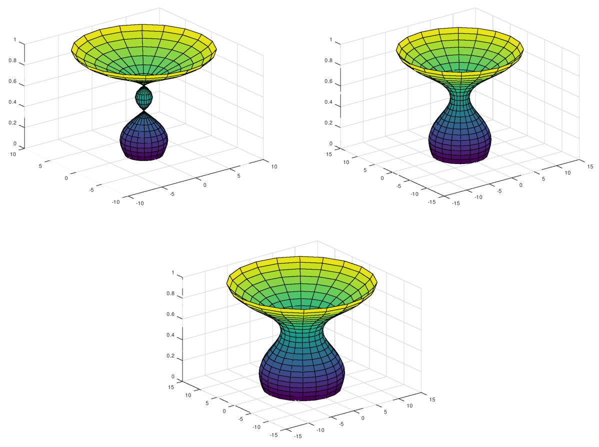

Figure 17: The three categories of object sizes from the dataset.

As we did above, we'll generate a new object, provide only part of the point data for that object to the prediction algorithm, and then test whether the prediction algorithm assigns the new object to a category with an anchor from the same class as the input object. That is, if the class of the anchor of the category to which the new input object is assigned matches the class of the input object, then we treat that prediction as a success.

I've randomly generated 300 objects from each class as a new input to the prediction algorithm, for a total of 900 input objects. I then provided the first $M = 1200$ values from each input vector to the prediction algorithm, producing 900 predictions. There are three possibilities for each prediction: success (i.e., a correct classification); rejection (i.e., the shape doesn't fit into the dataset); and failure (i.e., an incorrect classification). For this particular set of inputs, the success rate was 100% for all three classes of objects, with no rejections, and no incorrect classifications. Different datasets, and different inputs could of course produce different results. Nonetheless, the obvious takeaway is that the prediction algorithm is extremely accurate.

4 The Future of Computation

The Church-Turing Thesis is a hypothesis that asserts that all models of computation can be simulated by a Universal Turing Machine. In effect, the hypothesis asserts that any proposed model of computation is either inferior to, or equivalent to, a UTM. The Church-Turing Thesis is a hypothesis, and not a mathematical theorem, but it has turned out to be true for every known model of computation, though recent developments in quantum computing are likely to present a new wave of tests for this celebrated hypothesis.

In a research note entitled, "A Simple Model of Non-Turing Equivalent Computation", I presented the mathematical outlines of a model of computation that is on its face not equivalent to a UTM. That said, I have not yet built any machines that implement this model of computation, so, not surprisingly, the Church-Turing Thesis still stands.

The simple insight underlying my model is that a UTM cannot generate complexity, which can be easily proven.[9] This means that the Kolmogorov Complexity of the output of a UTM is always less than or equal to the complexity of the input that generated the output in question. This simple lemma has some alarming consequences, and in particular, it suggests that the behaviors of some human beings might be the product of a non-computable process, perhaps explaining how it is that some people are capable of producing equations, musical compositions,[10] and works of art,[11] that do not appear to follow from any obvious source of information. Originality is in this view the spontaneous generation of complexity, which is anomalous when viewed through the lens of computer theory.

In more general terms, any device that consistently generates outputs that have a higher complexity than the related inputs is by definition not equivalent to a UTM. Expressed symbolically, $y = F(x)$, and $K(x) < K(y)$, for some set of outputs y, where K is the Kolmogorov complexity. If the set of outputs is infinite, then the device can consistently generate complexity, which is not possible using a UTM. If the device generates only a finite amount of complexity, then the device can of course be implemented by a UTM that has some complexity stored in the memory of the device. In fact, this is exactly the model of computation that I've outlined above, where a compressed, specialized tree is stored in the memory of a device, and then called upon to supplement the inputs to the device, generating the outputs of the device.

The model of computation that I've presented above suggests that we can create simple, presumably cheap devices, that have specialized memories of the type I described above, that can provide fast, intelligent answers to complex

[9]Let $K(x)$ denote the computational complexity of the string x, and let $y = U(x)$ denote the output of a UTM when given x as input. Put informally, $K(y)$ is the length, measured in bits, of the shortest program that generates y on a UTM. Since x generates y when x is given as the input to a UTM, it follows that $K(y)$ cannot be greater than the length of x. This in turn implies that $K(y) \leq K(x) + C$. That is, we can generate y by first running the shortest program that will generate x, which has a length of $K(x)$, and then feed x back into the UTM, which will in turn generate y. This is simply a UTM that runs twice, the code for which will have a length of C that does not depend upon x, which proves the result. That is, there is a UTM that always runs twice, and the code for that machine is independent of the particular x under consideration.

[10]*Sonata for Cello and Piano, Op. 119*, by Sergei Prokofiev.

[11]*Judith II*, by Gustav Klimt.

questions, but nonetheless also perform basic computations. This would allow for the true commoditization of artificial intelligence, since these devices would presumably be both small and inexpensive to manufacture. This could lead to a new wave of economic and social changes, where cheap devices, possibly as small as transistors, have the power to analyze a large number of complex observations and make inferences from them in real-time.

In a research note entitled, "Predicting and Imitating Complex Data Using Information Theory", I showed how these same algorithms can be used to approximate both simple, closed-form functions, and complex, discontinuous functions that have a significant amount of statistical randomness. As a result, we could store trees that implement everyday functions found in calculators, such as trigonometric functions, together with trees that implement complex tasks in AI, such as image analysis, all in a single device that makes use of the algorithms that I outlined above. This would create a single device capable of both general purpose computing, and sophisticated tasks in artificial intelligence. We could even imagine these algorithms being hardwired as "smart transistors" that have a cache of stored trees that are then loaded in real-time to perform the particular task at hand.

All of the algorithms I presented above are obviously capable of being simulated by a UTM, but as a general matter, this type of non-symbolic, stimulus and response computation can in theory produce input-output pairs that always generate complexity, and therefore, cannot be reproduced by a UTM. Specifically, if the device in question maps an infinite number of input strings to an infinite number of output strings that each have a higher complexity than the corresponding input string, then the device is simply not equivalent to a UTM, and its behavior cannot be simulated by a UTM with a finite memory, since a UTM cannot, as a general matter, generate complexity.

This might sound like a purely theoretical model, and it might be, but there are physical systems whose states change in complex ways as a result of simple changes to environmental variables. As a result, if we are able to find a physical system whose states are consistently more complex than some simple environmental variable that we can control, then we can use that environmental variable as an input to the system. This in turn implies that we can consistently map a low-complexity input to a high-complexity output. If the set of possible outputs appears to be infinite, then we would have a system that could be used to calculate functions that cannot be calculated by a UTM. That is, any such system would form the physical basis of a model of computation that cannot be simulated by a UTM, and a device capable of calculating non-computable functions.