ON MODELING, MONEY, AND BANDITS

by

Benjamin P. Keefer

A thesis submitted in partial fulfillment of the requirements
for graduation with Honors in Mathematics.

Whitman College

2008

Certificate of Approval

This is to certify that the accompanying thesis by Benjamin P. Keefer has been accepted in partial fulfillment of the requirements for graduation with Honors in Mathematics.

____

Douglas R. Hundley, Ph.D.

Whitman College

May 06, 2008

Contents

List of Figures

1 Introduction

In this paper, we first review the modeling process and then apply it to a problem in portfolio analysis. We begin by proving the Best Basis Theorem and the Singular Value Decomposition and showing how we can use them to make a data set easier to analyze. From there, we discuss data clustering techniques and neural networks. Although we originally intended to use these clustering algorithms and neural networks to forecast the asset returns in our portfolio-analysis application, we instead choose to frame the application in terms of the n-armed bandit problem. We find however, that even this approach proves problematic and we then analyzed why. The conclusion will summarize our results.

2 Preprocessing Data

Preprocessing a data set $\mathbf{X}$, represented as an $m \times n$ matrix, generally has two purposes: to make it smaller and simpler. The two techniques discussed in this section are the Singular Value Decomposition (SVD) and the Best Basis Theorem. The SVD gives us a representation of $\mathbf{X}$ that is easier to work with. It allows us to decompose an $m \times n$ matrix into a product of an orthogonal $m \times m$ matrix, an $m \times n$ diagonal matrix, and another $n \times n$ orthogonal matrix. These matrices have other important properties as well, though we will discuss these properties more fully later. This decomposed product of $\mathbf{X}$ can then be used to find the Best Basis of the data set, which is simply a subset of elements of a basis of $\mathbf{X}$ that "best" (in terms of minimizing the least-squares error) reproduces the original data set.

2.1 Finding the Best Basis

The Basis Theorem allows us to take a multivariate data set and find a more compact representation. This means that if we have an $m \times n$ data set $\mathbf{X}$ with m data points in $\mathbb{R}^n$, then we would want to find a way to represent each data point in $\mathbb{R}^k$ with $k < n$ in order to reduce the complexity of our data. Of course there is a tradeoff between conveying most of the original information and reducing the number of coordinates.

In his textbook, David Lay [2] gives an example of how it can be applied to satellite images. He presents three images of Railroad Valley, Nevada using different spectral bands. Using the Best Basis Theorem (also called Principal Component Analysis), he creates a new picture, in which each pixel is a linear combination of the pixels from the three original pictures. This new picture also portrays a wider range of light than any of the original pictures. The general idea is to choose combinations of the original variables that seem important and ignore the elements of the basis that appear unimportant. For example, the line of best fit is the one-dimensional data set that best describes a higher dimensional data set. If we wanted to approximate a two-dimensional data set in a single dimension, all we would need is the line of best fit equation and the x-coordinates. We could then proceed to drop the y-coordinates from our data and still preserve our approximations. Condensing a data set in this fashion makes it easier to analyze and takes less memory because fewer coordinates are needed.

First, we will describe what we mean by the Best Basis and then show how to find it. Suppose we have an arbitrary array of data as defined below:

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix}$$

Consider the row vectors of $\mathbf{X}$, denoted $x^{(1)}, x^{(2)}, \dots, x^{(m)}$. Each of these row vectors is an element of $\mathbb{R}^n$.

Generally, when we talk about the Best Basis, we are referring to the "best" $k-$dimensional basis, for some k between 1 and $n-1$. Our goal is to find a specific basis Φ, with elements $\{\phi_1, \phi_2, \dots, \phi_n\}$, from which we can draw the k vectors that "best" represent $\mathbf{X}$. Remember that we want to reduce the number of coordinates in each data point, so k is at most $n-1$. As we will see in the proof of the Best Basis Theorem, it is easy to find the Best single-dimensional Basis of $\mathbf{X}$ and then to add the next $k-1$ vectors one at a time. So we will construct Φ one element at a time.

Last, we need to be able to determine which of two bases is better. Generally speaking, one basis is better than another if it allows us to more accurately reproduce the row vectors of $\mathbf{X}$. For example, suppose we have the data set

$$\mathbf{X} = \begin{pmatrix} 1 & 3 & 1 & 2 \\ 3 & 7 & 1 & 6 \\ 2 & 5 & 1 & 4 \end{pmatrix}$$

and the orthogonal basis

$$\Phi = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\},$$

with ϕ_i being the $i^{(\text{th})}$ element from the left in Φ. Notice that $\mathbf{X}$ is 3×4 and each data point $x^{(i)}$ is an element of $\mathbb{R}^4$. Also note that $\mathbb{R}^4$ is spanned by Φ. In order to reduce the dimensionality of the data set, we need to choose k between 1 and 4. Let $k = 3$. If we wanted to condense the data even further, we could also choose k to be 1 or 2. Suppose we wanted to represent $\mathbf{X}$ as best we could with the following three elements of Φ:

$$\left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}.$$

Using these vectors, we can approximate the three data points (i.e., row vectors of $\mathbf{X}$) as

$$\begin{aligned}\tilde{x}^{(1)} &= \phi_1 + 3\phi_2 + 2\phi_4 \\ \tilde{x}^{(2)} &= 3\phi_1 + 7\phi_2 + 6\phi_4 \\ \tilde{x}^{(3)} &= 2\phi_1 + 5\phi_2 + 4\phi_4\end{aligned}$$

with $\mathbf{X}$ then represented as

$$\begin{pmatrix} 1 & 3 & 0 & 2 \\ 3 & 7 & 0 & 6 \\ 2 & 5 & 0 & 4 \end{pmatrix}.$$

If we denote the error for the i^{th} observation as $x_{err}^{(i)} = |x^{(i)} - \tilde{x}^{(i)}|$, then it is easy to show that $\| \; x_{err}^{(i)} \; \| = 1$ and that $\| \; x_{err}^{(i)} \; \|^2 = 1$ for each i. We will determine whether one basis is better than another by comparing their associated average squared error. Had we tried to represent $\mathbf{X}$ using any other choice of three elements from Φ, our total error would have been even larger. You can see this for yourself if you try choosing other three element combinations from Φ. This means that $\{\phi_1, \phi_2, \phi_4\}$ is the three element subbasis of Φ that "best" represent $\mathbf{X}$. *Note that this does not mean that these elements are the best* $3-$*dimensional basis for* $\mathbf{X}$*.* All this shows is that of the 4 possible combinations of 3 element combinations considered here, this combination is best. The goal of the Best Basis theorem is to first find Φ and then to choose the k best elements of Φ.

We give a simplified form of the Best Basis Theorem below but first we will need a few final definitions. We define the centroid of the data set as the average point. If the data set is $m \times n$ with m observations in $\mathbb{R}^n$, then the average point is a vector in $\mathbb{R}^n$. We can calculate the centroid by averaging the entries in each of the columns, one column at a time. For example, in

$$\mathbf{X} = \begin{pmatrix} 1 & 3 & 1 & 2 \\ 3 & 7 & 1 & 6 \\ 2 & 5 & 1 & 4 \end{pmatrix}$$

the centroid is calculated as

$$\begin{bmatrix} \frac{1+3+2}{3} \\ \frac{3+7+5}{3} \\ \frac{1+1+1}{3} \\ \frac{2+6+4}{3} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 1 \\ 4 \end{bmatrix}$$

and the mean subtracted representation for $\mathbf{X}$ is

$$\begin{pmatrix} 1-2 & 3-5 & 1-1 & 2-4 \\ 3-2 & 7-5 & 1-1 & 6-4 \\ 2-2 & 5-5 & 1-1 & 4-4 \end{pmatrix} = \begin{pmatrix} -1 & -2 & 0 & -2 \\ 1 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

We first subtract the centroid in order to make it easier to find the best basis. Recall that in statistics, the line of best fit passes through the mean

(x, y) pair. Subtracting the mean forces the line of best fit to pass through the origin. Once we know that the line of best fit passes through the origin, then we know that the $y-$intercept is 0 and all that is left to find is the slope. An analogous simplification occurs in higher dimensional data sets that are mean subtracted.

We have one last remark. We will need what is called the covariance matrix $\mathbf{C}$ of $\mathbf{X}$. The formula for $\mathbf{C}$ is given by

$$\mathbf{C} = \frac{1}{m}\mathbf{X}^T\mathbf{X}, \tag{1}$$

where m is the number of observations in the data set. We will also assume that the best basis is orthonormal. If it was not, we could use the Gramm-Schmidt process to convert the basis to an orthogonal form and then normalize it. Now we are ready for the Best Basis Theorem:

The Best Basis Theorem. *Suppose that:*

- $\mathbf{X}$ *is an* $m \times n$ *mean-subtracted data matrix of* m *points in* $\mathbb{R}^n$.
- $\mathbf{C}$ *is the covariance matrix of* $\mathbf{X}$

Then the best k*-element basis* Φ *of* $\mathbf{X}$ *is found by taking the first* k *eigenvectors of* $\mathbf{C}$*, when arranged by eigenvalues from largest to smallest.*

Another way to think of the Best Basis is that it is a k-dimensional set that is associated with the smallest average squared error. Earlier, we introduced error as $\| x_{err}^{(i)} \|^2$, which was the squared Euclidean distance between each original data point and its new approximation. If $x^{(i)}$ is a row vector of $\mathbf{X}$, then $x^{(i)} \in \mathbb{R}^n$ because $\mathbf{X}$ is $m \times n$. Since Φ is a basis of $\mathbb{R}^n$, then we can write $x^{(i)}$ as a linear combination of the elements of Φ:

$$x^{(i)} = a_1\phi_1 + a_2\phi_2 + \ldots + a_n\phi_n,$$

for some real-valued constants $a_1, a_2, \ldots, a_n$ and where $\phi_1, \phi_2, \ldots, \phi_n$ are vectors in the basis Φ. If we choose the first k elements of Φ to approximate $x^{(i)}$, then we have

$$x^{(i)} \approx a_1\phi_1 + \ldots + a_k\phi_k.$$

Then we can write the residual, or the part left out of the approximation, of $x^{(i)}$ as $x_{err} = a_{k+1}\phi_{k+1} + \ldots + a_n\phi_n$. The error associated with the basis $\phi_1, \phi_2, \ldots, \phi_k$ is simply the squared magnitude of the residual:

$$\| x_{err}^{(i)} \|^2 = \| \sum_{j=k+1}^{n} a_j\phi_j \|^2 .$$

If we have m data points (or row vectors), the average squared error is

$$\frac{1}{m}\sum_{i=1}^{m} \| x_{err}^{(i)} \|^2.$$

In order to find the best k-element basis of $\mathbf{X}$, denoted Φ, we will have to minimize the average squared error.

The sketch of the proof of the Best Basis Theorem was provided by Doug

Hundley [3] from Whitman College. The proof consists of two parts:

1. Converting the minimization problem into a maximization problem that is easier to solve.

2. Solving the maximization problem.

To find the best basis, we will first show that the least-squares error associated with the ith observation in the data set can be measured as

$$\parallel x_{err}^{(i)} \parallel^2 = \sum_{j=k+1}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle .$$

Since for a general vector $\mathbf{v}$:

$$\parallel \mathbf{v} \parallel = \langle \mathbf{v}, \mathbf{v} \rangle = \mathbf{v}^T\mathbf{v},$$

then we note that the error for the ith observation, $\parallel x_{err}^{(i)} \parallel^2$ is equivalent to the product $\left\langle x_{err}^{(i)}, x_{err}^{(i)} \right\rangle$. Given that $x_{err}^{(i)} = a_{k+1}^{(i)}\phi_{k+1} + \ldots + a_n^{(i)}\phi_n$ with each ϕ_j orthonormal to the others, we can expand the error as

$$\parallel x_{err}^{(i)} \parallel^2 = \sum_{j=k+1}^{n} (a_j^{(i)})^2 \phi_j^T\phi_j = \sum_{j=k+1}^{n} (a_j^{(i)})^2$$

because $\phi_j^T\phi_j = 1$ and $\phi_j^T\phi_i = 0$ for $j \neq i$.

Each $a_j^{(i)}$, where $a_j^{(i)}$ corresponds to coefficients on the jth element of the basis Φ for the data point $x^{(i)}$, can be written as

$$(x^{(i)})^T\phi_j = \langle x^{(i)}, \phi_j \rangle = \left\langle a_1^{(i)}\phi_1 + a_2^{(i)}\phi_2 + \ldots + a_n^{(i)}\phi_n, \phi_j \right\rangle = a_j^{(i)}$$

since the elements of Φ are orthonormal. Then by rearranging the terms of the inner product, we can write the error as

$$\begin{aligned} \| \, x_{err}^{(i)} \, \|^2 &= \sum_{j=k+1}^{n} (a_j^{(i)})^2 = \sum_{j=k+1}^{n} \left\langle a_j^{(i)}, a_j^{(i)} \right\rangle \\ &= \sum_{j=k+1}^{n} \left\langle \left(x^{(i)}\right)^T \phi_j, \left(x^{(i)}\right)^T \phi_j \right\rangle \\ &= \sum_{j=k+1}^{n} \phi_j^T x^{(i)} \left(x^{(i)}\right)^T \phi_j. \end{aligned}$$

The average error is the mean of the m individual errors. From what we have shown thus far, the average error is equal to

$$\begin{aligned} \frac{1}{m}\sum_{i=1}^{m} \| \, x_{err}^{(i)} \, \|^2 &= \sum_{i=1}^{m} \frac{1}{m} \left(\sum_{j=k+1}^{n} (a_j^{(i)})^2 \right) \\ &= \sum_{i=1}^{m} \frac{1}{m} \left(\sum_{j=k+1}^{n} \phi_j^T x^{(i)} \left(x^{(i)}\right)^T \phi_j \right). \end{aligned}$$

In the above expressions, notice that we now have double summation. The summation from $i = 1$ to m is across all m data points. The summation $j = k+1$ to n is across the $n-k$ elements of Φ excluded from the k element sub-basis we are considering. Notice that for each j, with $k+1 \leq j \leq n$, we can rewrite $\sum_{i=1}^{m} \phi_j^T x^{(i)} (x^{(i)})^T \phi_j$ as

$$\phi_j^T \left[\sum_{i=1}^{m} x^{(i)} (x^{(i)})^T \right] \phi_j$$

because ϕ_j is independent of the choice of the i^{th} data point. Therefore, the inner product form is $\langle \phi_j, \left[\sum_{i=1}^{m} x^{(i)} (x^{(i)})^T\right] \phi_j \rangle$. We want to write the summation on the inside of the inner product in terms of the covariance matrix $\mathbf{C} = \frac{1}{m}\mathbf{X}^T\mathbf{X}$. Since $\mathbf{X} = [x^{(1)} \dots x^{(m)}]^T$, we can rewrite $\mathbf{C}$ as

$$\begin{aligned} \mathbf{C} &= \frac{1}{m}[x^{(1)} \dots x^{(m)}][x^{(1)} \dots x^{(m)}]^T \\ &= \frac{1}{m} \sum_{i=1}^{m} x^{(i)} (x^{(i)})^T \end{aligned}$$

through matrix multiplication. We are now ready to relate the average error to the covariance matrix $\mathbf{C}$. Remember from before that

$$\begin{aligned} \frac{1}{m} \sum_{i=1}^{m} \| \, x_{err}^{(i)} \, \|^2 &= \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{j=k+1}^{n} (a_j^{(i)})^2 \right) \\ &= \frac{1}{m} \sum_{j=k+1}^{n} \sum_{i=1}^{m} (a_j^{(i)})^2 \\ &= \sum_{j=k+1}^{n} \sum_{i=1}^{m} \frac{1}{m} \left(\phi_j^T x^{(i)} (x^{(i)})^T \phi_j \right). \end{aligned}$$

Since for each j we know that $\sum_{i=1}^{m} \phi_j^T x^{(i)} (x^{(i)})^T \phi_j = \langle \phi_j, \left[\sum_{i=1}^{m} x^{(i)} (x^{(i)})^T\right] \phi_j \rangle$, we can convert to inner product form and shift the summations as follows

$$\frac{1}{m}\sum_{i=1}^{m} \parallel x_{err}^{(i)} \parallel^2 = \sum_{j=k+1}^{n} \left(\left\langle \phi_j, \left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}(x^{(i)})^T\right]\phi_j \right\rangle \right).$$

Since the summation on the inside of the inner product is equivalent to $\mathbf{C}$, we can rewrite the average error as

$$\frac{1}{m}\sum_{i=1}^{m} \parallel x_{err}^{(i)} \parallel^2 = \sum_{j=k+1}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle .$$

To minimize the average error and find the best basis, we need to minimize

$$\sum_{j=k+1}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle .$$

To find the best 1-dimensional basis, let $k = 1$. Since

$$\sum_{j=1}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle = \sum_{j=1}^{1} \langle \phi_j, \mathbf{C}\phi_j \rangle + \sum_{j=1+1}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle ,$$

then minimizing $\sum_{j=2}^{n} \langle \phi_j, \mathbf{C}\phi_j \rangle$ is the same as maximizing $\phi_1^T\mathbf{C}\phi_1$. That is, to minimize the error all we need to do is to find the best projection.

How can we find the best projection? We will employ the *Spectral Theorem for Symmetric Matrices* from Lay's Linear Algebra text [2]. Recall that if $\mathbf{X}$ is an $m \times n$ matrix, then $\mathbf{C} = \frac{1}{m}\mathbf{X}^T\mathbf{X}$ must be $n \times n$ and therefore symmetric. By the Spectral Theorem, we can orthogonally diagonalize $\mathbf{C}$

into its eigenvectors $\{\mathbf{v}_i\}_1^n$ and eigenvalues $\{\lambda\}_1^n$ with

$$\mathbf{C} = \lambda_1 \mathbf{v}_1^T \mathbf{v}_1 + \lambda_2 \mathbf{v}_2^T \mathbf{v}_2 + \ldots + \lambda_n \mathbf{v}_n^T \mathbf{v}_n = V \Lambda V^T$$

and $VV^T = V^T V = I$. In the above decomposition, Λ is a diagonal matrix of eigenvalues and V has the eigenvectors of $\mathbf{C}$ for columns with transpose V^T. Assume that the eigenvectors and corresponding eigenvalues are ordered so that the eigenvalues are arranged from largest to smallest. Also, assume that the eigenvectors are orthonormal (since they are already orthogonal, it would be very easy to make them orthonormal if they were not already). Let ϕ be an arbitrary vector in $\mathbb{R}^n$ with $\parallel \phi \parallel = 1$. Since $\phi \in \mathbb{R}^n$, it must be some linear combination of the eigenvectors of $\mathbf{C}$. Furthermore, we can find a real-valued vector α such that

$$\phi = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n = V\alpha.$$

Remember that in order to find the best projection, we needed to maximize $\phi^T \mathbf{C} \phi$. If $\parallel \phi \parallel = 1$, then $\parallel \phi \parallel^2 = \phi^T \phi = 1$ and so we would need to maximize $\frac{\phi^T \mathbf{C} \phi}{\phi^T \phi}$. Since $\phi = V\alpha$ (where V is the matrix of eigenvectors), then

$$\phi^T \mathbf{C} \phi = \alpha^T V^T V \Lambda V^T V \alpha = \alpha^T \Lambda \alpha$$

which can be expanded to $\lambda_1\alpha_1^2 + \lambda_2\alpha_2^2 + \ldots + \lambda_n\alpha_n^2$. Since

$$\phi = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \ldots + \alpha_n\mathbf{v}_n$$

with the eigenvectors orthonormal, then

$$\phi^T\phi = \alpha_1^2 + \ldots + \alpha_n^2.$$

Putting the numerator and the denominator together, we obtain

$$\frac{\phi^T\mathbf{C}\phi}{\phi^T\phi} = \frac{\lambda_1\alpha_1^2 + \ldots + \lambda_n\alpha_n^2}{\alpha_1^2 + \ldots + \alpha_n^2}.$$

Since λ_1 is larger than $\lambda_2, \ldots, \lambda_n$, then

$$\frac{\lambda_1\alpha_1^2 + \ldots + \lambda_n\alpha_n^2}{\alpha_1^2 + \ldots + \alpha_n^2} \leq \frac{\lambda_1\alpha_1^2 + \ldots + \lambda_1\alpha_n^2}{\alpha_1^2 + \ldots + \alpha_n^2} \leq \lambda_1.$$

Observe that equality holds only when $\phi = \mathbf{v}_1$.

Thus, we have shown that the best 1-dimensional orthonormal basis is observed by taking the first *leading eigenvector* from the covariance matrix. To find the best 2-dimensional orthonormal basis, we would take the next leading eigenvector (with the second largest eigenvalue). Although this completes the proof, some explanation may be helpful. We started this proof by constructing a measurement of error for each possible basis. We argued earlier that the best basis is the one associated with the smallest average error. For a single dimensional basis, the error is minimized when $\phi_1^T \mathbf{C} \phi_1$ is maximized. Since we do not initially know what Φ looks like, ϕ_1 *can be any vector in* $\mathbb{R}^n$. So we need to consider all vectors in $\mathbb{R}^n$ in order to find the best single dimensional basis. If we let ϕ represent an arbitrary element in $\mathbb{R}^n$ with $\| \phi \| = 1$, then we have shown that

$$\frac{\phi^T \mathbf{C} \phi}{\phi^T \phi} \leq \lambda_1$$

and that equality holds only when $\phi = \mathbf{v}_1$. So the vector that maximizes $\phi_1^T \mathbf{C} \phi_1$ is $\mathbf{v}_1$.

Now we will let $\phi_1 = \mathbf{v}_1$. That is, we will choose the first element of the basis as $\mathbf{v}_1$. Then we will try to find the next vector in the best basis. By the argument we just made, we would want to choose $\phi_2 = \mathbf{v}_2$ and repeat this process until we have chosen the k leading eigenvectors.

So this means that we initialize the elements in the best basis one at a time. This is convenient because it means we can keep adding elements to the basis until we are satisfied. So we can choose k as we construct the basis; we stop adding elements when we are satisfied with the size of the average error

$$\frac{1}{m} \sum_{i=1}^{m} \| x_{err}^{(i)} \| .$$

So what does this all mean? In his original example, Lay found the 3×3 covariance matrix for the image of Railroad Valley (the original data had 4 million vectors) and created a new image by taking the leading eigenvector of the covariance matrix expansion. This image captured approximately 93 percent of information provided by the original data using much less memory. He also found that two pictures would capture all but 1.2 percent of the original information. When considering your own $m \times n$ data set, you can continue adding eigenvectors to the basis until you capture a previously determined percentage of the information in the data set. For example, to convert a 3-dimensional data set into a 2-dimensional data set, you would need to find the plane in $\mathbb{R}^2$ that best preserves your data and force each point not on the plane to its closest neighbor on the plane.

2.2 The Singular Value Decomposition

The Singular Value Decomposition (SVD) is an important result in applied linear algebra. David Lay and Doug Hundley ([2],[3]) say that some applications include finding the least-squares solution (or finding one of many generalized least-squares solutions), estimating the rank of a given matrix, finding the pseudo-inverse of any matrix (it need not be $n \times n$), and constructing bases for the four fundamental subsets of a matrix (i.e., the null space, the column space, the null space of the transpose, and the column space of the transpose). It is also helpful in finding the Best Basis. The rest of this section will be devoted to giving a proof of the SVD and explaining how it ties into the Best Basis.

Since we are talking about the Singular Value Decomposition, it would be nice to know what a singular value is. If we have an $m \times n$ matrix A, then we will call the singular values as the square roots of the eigenvalues of the matrix A^TA. In our proof of the SVD, we will show why these values are significant.

The Singular Value Decomposition. *Let A be any $m \times n$ matrix of rank r. Then*

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

where U is an orthogonal $m \times m$ matrix constructed from the eigenvectors of AA^T, V is an orthogonal $n \times n$ matrix constructed from the eigenvectors of A^TA, and Σ is an $m \times n$ diagonal matrix of singular values (with each singular value computed as the square root of an eigenvalue of A^TA).

The following proof was sketched by Hundley in his work [3], though any mistakes are my own. Let A be an $m \times n$ matrix. Then consider the $n \times n$ symmetric matrix A^TA. It is symmetric because $(A^TA)^T = A^TA$. By the spectral theorem of symmetric matrices, A^TA is orthogonally diagonalizable into VDV^T where the columns of V are orthogonal eigenvectors of A^TA and the corresponding entries of the diagonal matrix D are their eigenvalues. Denote the eigenvalues of A^TA (the entries of D) as $\{\lambda_i\}_{i=1}^n$ and the corresponding orthonormal eigenvectors as $\{v_i\}_{i=1}^n$. To obtain the orthonormal eigenvectors, simply divide the entries of the columns of V by the magnitude of the corresponding column vector.

First, we will need to show that all the eigenvalues of A^TA are non-negative. Let λ_i be an eigenvalue of A^TA with corresponding orthonormal eigenvector $\mathbf{v}_i$. Then

$$\parallel A\mathbf{v}_i \parallel^2 = (A\mathbf{v}_i)^T A\mathbf{v}_i = \mathbf{v}_i^T A^T A\mathbf{v}_i = \mathbf{v}_i^T(\lambda_i \mathbf{v}_i) = \lambda_i.$$

Since $\parallel A\mathbf{v}_i \parallel^2$ is nonnegative, so is λ_i.

Suppose that the rank of A^TA is r. Then the dimension of Null(A^TA) is $n-r$ because if A^TA has n columns, r of which are linearly independent, then it must have $n-r$ columns that are linearly dependent. The rank of the null space of A^TA is $n-r$. Since A has n columns, if the the null space has rank $n-r$, then the rank of A is likewise r.

To show that Null(A) has rank $n-r$, we will first suppose W is the subspace spanned by the eigenvectors corresponding to the eigenvalues of A^TA equal to zero. Clearly, if $\mathbf{w} \in W$. Then $\mathbf{w}$ is in Null(A^TA). Suppose now that $\mathbf{w} \in$ Null(A^TA), then $\mathbf{w}$ is an eigenvector with an eigenvalue of 0 because $A^TA\mathbf{w} = 0 \times \mathbf{w} = 0$. Therefore, $\mathbf{w} \in W$. So we can conclude that $W =$ Null(A^TA) and the two vector spaces must also share the same rank $n-r$. Since A^TA has rank r, then both Null(A^TA) and W have dimension $n-r$. This means we have $n-r$ eigenvectors of A^TA with eigenvalues equal to 0. If $\mathbf{w} \in W$, we have that

$$\parallel A\mathbf{w} \parallel^2 = \mathbf{w}^T A^T A\mathbf{w} = \mathbf{w}^T \mathbf{0}\mathbf{w} = \mathbf{0}$$

and so $\mathbf{w} \in \text{Null}(A)$. Moreover, if $\mathbf{w} \in \text{Null}(A)$, then

$$A^T A\mathbf{w} = A^T(A\mathbf{w}) = A^T\mathbf{0} = \mathbf{0}$$

and so $\text{Null}(A)$ must have rank $n - r$ and A must have rank of r.

We define the singular values of A as $\sigma_i = \sqrt{\lambda_i}$. Recall that we showed that $\| A\mathbf{v}_i \|^2 = \lambda_i$ earlier. This implies that the eigenvalues are non-negative and that σ_i is well-defined. Then construct the vectors

$$\mathbf{u}_i = \frac{1}{\sqrt{\| A\mathbf{v}_i \|^2}} A\mathbf{v}_i = \frac{1}{\sigma_i} A\mathbf{v}_i$$

and the matrix U having columns $\mathbf{u}_i$ as its ith column when $\lambda_i \neq 0$. Remember from the theorem that we wanted to construct U as an $m \times m$ orthogonal matrix. But we only have r vectors $\mathbf{u}_i$ where $\lambda_i \neq 0$. So we need $m - r$ more columns for U, with each column in $\mathbb{R}^m$.

We know that AA^T is $m \times m$ with rank r. If it has $m - r$ eigenvectors with eigenvalues equal of zero then we may be able to use these eigenvectors to fill in the missing columns of U. In fact, stealing the eigenvectors corresponding to the zero eigenvalues of AA^T will make things work out nicely, as we will see.

At this point, we have two objectives. To make U orthogonal, we will need to show that if $i \neq j$, then $\mathbf{u}_i$ and $\mathbf{u}_j$ are orthogonal. To make U have dimensions $m \times m$, we will need to show that AA^T has $m - r$ eigenvectors with an eigenvalue of 0. To prove the first, it is sufficient to show that the mapping $\mathbf{v} \to A\mathbf{v}$ preserves the orthogonality of the eigenvectors of A^TA. Suppose $\mathbf{v}_i$ and $\mathbf{v}_j$ are two such eigenvectors with $i \neq j$. Then

$$\begin{aligned}
\langle A\mathbf{v}_i, A\mathbf{v}_j\rangle &= \mathbf{v}_j^T A^T A\mathbf{v}_i \\
&= \mathbf{v}_j^T(\lambda_i\mathbf{v}_i) \\
&= \lambda_i \langle \mathbf{v}_i, \mathbf{v}_j\rangle \\
&= \lambda_i \cdot \mathbf{0} = \mathbf{0},
\end{aligned}$$

and the linear transformation $\mathbf{v} \to A\mathbf{v}$ preserves orthogonality of the eigenvectors of A^TA.

To show that AA^T has $m-r$ eigenvectors with a corresponding eigenvalue of 0, we will need to show that Null(AA^T) has rank $m-r$. Recall that we earlier assumed Null(A^TA) has rank $n-r$ and then proved that A must then have rank r. That means A has r pivot positions in reduced echelon form. Recall that the pivot columns of A provide a basis for Col$A$ and that the rows of A corresponding to the pivot rows of the reduced echelon form provide the basis for the Row$A$. Since Row$A$=ColA^T and since there are r pivot positions, then there are r pivot rows and A^T has rank r. This means that Null(A^T) has rank $m-r$. Now if Null(A^T) differed from Null(AA^T) we would arrive at a contradiction because we showed earlier that Null(A^TA) and Null(A) must have the same rank.

Therefore, we can find $m-r$ orthogonal eigenvectors from the orthogonal decomposition of AA^T (since it too is symmetric) with 0 eigenvalues and use these eigenvectors to fill the columns of U. To show that the original r vectors $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_r\}$ are indeed eigenvectors of AA^T (we already showed orthogonality), notice that

$$AA^T\mathbf{u}_i = AA^T\frac{1}{\sigma_i}A\mathbf{v}_i = A\frac{1}{\sigma_i}\lambda_i\mathbf{v}_i = \lambda_i\big(A\frac{1}{\lambda_i}\mathbf{v}_i\big) = \lambda_i\mathbf{u}_i.$$

Thus, the columns of U are also orthogonal eigenvectors of AA^T.

If we construct the diagonal $m \times n$ matrix from the n singular values of A^TA, with $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$, then all that is left to prove for the SVD is that $A = U\Sigma V^T$. We will now proceed with this last step. Consider AV. By matrix multiplication,

$$\begin{aligned} AV &= [A\mathbf{v}_1 \ldots A\mathbf{v}_n] \\ &= [\sigma_1\mathbf{u}_1 \ldots \sigma_n\mathbf{u}_n] \end{aligned}$$

since $\mathbf{u}_i = \frac{1}{\sigma_i} A\mathbf{v}_i$. Notice that

$$U\Sigma = [\mathbf{u}_1 \dots \mathbf{u}_n \dots \mathbf{u}_m] \begin{bmatrix} \sigma_1 & 0 & \dots & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \sigma_n & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix} = [\sigma_1\mathbf{u}_1 \dots \sigma_n\mathbf{u}_n].$$

So $AV = U\Sigma$. Since V is an $n \times n$ orthonormal matrix, it has inverse V^T, and so we have proved that $A = U\Sigma V^T$. $\square$

How does this relate to finding the Best Basis? Recall that the Best Basis for a data set $\mathbf{X}$ (with data entries as row vectors) was found by taking the leading eigenvectors of the covariance matrix $\mathbf{C}$. If we apply the Singular Value Decomposition to $\mathbf{X}$ and rewrite $\mathbf{X}$ as $U\Sigma V^T$ (and drop all zero rows and columns of Σ until it is a $k \times k$ matrix, with similar changes to U, V^T), then notice that

$$\mathbf{C} = \frac{1}{m}X^TX = \frac{1}{m}V\Sigma U^TU\Sigma V^T = V\big(\frac{1}{m}\Sigma^2\big)V^T.$$

Thus, the Best Basis for the row space of $\mathbf{X}$ comes from taking the first k leading eigenvectors of V. With the SVD, we no longer need to first find $\mathbf{C}$ and then its spectral decomposition. Instead, we can apply the SVD to $\mathbf{X}$

directly.

As an application of the SVD and the Best Basis, let us now consider the clown image in Figure 1 provided by MATLAB. This image is stored as

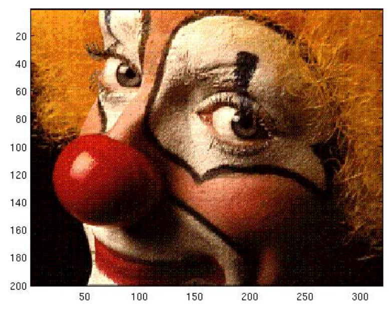


Figure 1: The original clown picture

a matrix of numbers. How big can these matrices get? In MATLAB, they can get very big [1]. A picture is composed of what we call pixels, which is short for "picture elements." If a picture has a lot of pixels, we say that it has a high resolution. Each pixel is a rectangular square, and the grid of these pixels form the picture. These pixels form the building blocks of the picture.

One example of a type of image with color is a true color image, which we can think of a matrix of size $m \times n$ with entries in $\mathbb{R}^3$. The m corresponds to the number of rows of pixels and n the number of columns of pixels in the image. For each point on the $m \times n$ matrix, we have a vector in $\mathbb{R}^3$ corresponding to the amount of red, green, and blue desired. So a 3.2 million pixel camera would need 9.6 million entries in $\mathbb{R}$ associated with it.

Suppose we want to know how many vectors are needed before we get an identifiable image. We can use the SVD to decompose the image's matrix and then create bases from the eigenvectors of the decomposition. Let us look first look at images produced by bases of dimensions 5 and 10.

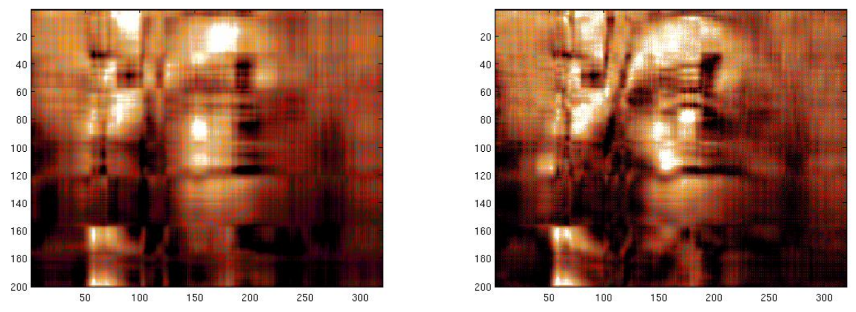


Figure 2: The reconstructed clown pictures with 5 and 10 vectors

The images in Figure 2 are not rendered well, although the image produced by ten vectors is at least recognizable. It looks like we will need to include more vectors in the bases. Let us now look at images produced by bases of dimensions 20 and 30 as shown in Figure 3.

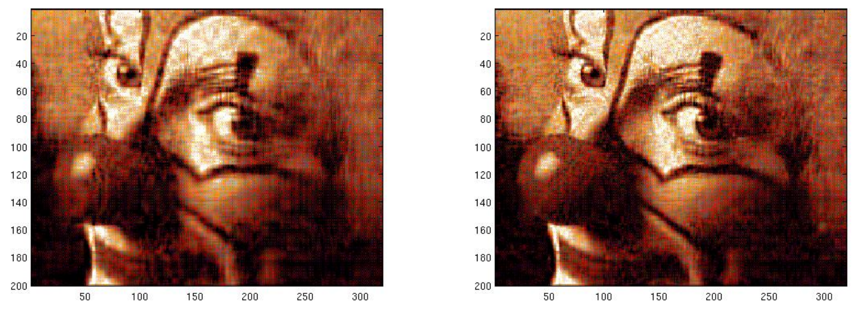


Figure 3: The reconstructed clown picture with 20 and 30 vectors

Notice that although these images are much clearer, the colors are noticeably muted (although readers in gray-scale may not be able to tell the difference). If we included more and more vectors, our image would slowly begin to look almost exactly like the original. However, since we are choosing the best vectors first, then each vector adds less and less to the picture and chances are that we would need a lot of vectors before we had an image that looked exactly like the original.

According to Lay [2], if we define the total variance of $\mathbf{X}$ as the sum of the entries along the diagonal of $\mathbf{C}$, denoted as the trace of $\mathbf{C}$ or $\mathrm{tr}(\mathbf{C})$, then we can think of $\frac{\lambda_i}{\mathrm{tr}(\mathbf{C})}$ as the information of $\mathbf{X}$ collected by the i^{th} eigenvector of $\mathbf{C}$. If we decide that we want to preserve 90 percent of the information of $\mathbf{X}$, then we can continue selecting eigenvectors of $\mathbf{C}$ until

$$\sum_{i=1}^{n} \frac{\lambda_i}{\mathrm{tr}(\mathbf{C})} \geq 0.90.$$

3 Neural Networks

A neural network is a function that assigns each input to an output that closely approximates a desired target. Since we generally use neural networks for classification or identification, then our target can be thought of the group or label we want to assign our input to. Although there are many different representations, the most typical form is called the three-layer feed-forward network, which we will focus on in this paper. For this type of neural network, the inputs, the outputs, and targets are generally vectors, with the targets and outputs having the same number of components.

Potential applications are numerous. One example is character recognition. Other applications mentioned by Steven Gonzalez [4] include converting printed text into audio form, playing backgammon, and "diagnosing automobile engine misfires." In MATLAB's implementation of the character recognition neural network, the neural network for character recognition is a function $f : \mathbb{N}^{35} \to \mathbb{N}^{26}$. It uses the method of steepest descent, a multivariate form of Newton's Method using the gradient instead of the slope, to estimate parameters that collectively represent a local minimum for the error of the estimates. This process is called training. The goal is for the computer to be able to identify which letter was drawn. To accomplish this task, it must first read an image, translate it into a vector in $\mathbb{N}^{35}$ and then find parameters that will successfully assign each vector to a vector in $\mathbb{N}^{26}$, which will represent a unique letter of the alphabet. In order to find such parameters, the computer needs to first be able to analyze a data bank of previous images of known letters and then develop a mechanism that correctly assigns the image to the correct letter. We will discuss how this is accomplished in more detail later. The rest of this section will be dedicated to understanding how a neural network functions, and we will use the character recognition example to illustrate the process.

In MATLAB's implementation, the image to be read is first broken up into thirty-five rectangles of equal size (see Figure 4 for an example). This character is saved as a vector $\vec{x} \in \mathbb{N}^{35}$, with each component of $\vec{x}$ corresponding to a rectangle in the image. If writing is present in the box, the corresponding component of $\vec{x}$ is assigned a value of 1. Otherwise, the component is assigned a value of 0. In the example of the letter A, we would take the image of the letter A given in Figure 4 and start from the top-left

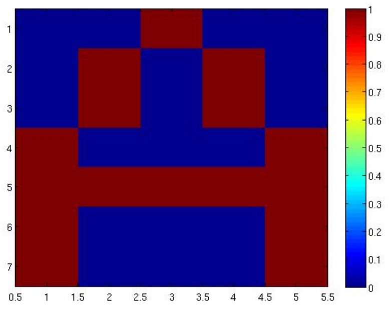


Figure 4: The letter A

corner (moving right and then down) and construct a vector $\vec{x} \in \mathbb{N}^{35}$ with an entry 1 if the corresponding rectangle in the image has writing in it and 0 in the absence of writing (the color red, or lighter shade of gray if printed in black-and-white, indicates writing). When translating the letter A, we would create the vector

$$\vec{x} = [\ 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 0 \mid 1 \mid 0 \mid 1 \mid 0 \mid 0 \mid 1 \mid 0 \mid 1 \mid 0 \mid 1 \mid 0 \mid 0 \mid \dots\ ]^T$$

Other letters are displayed in Figure 5.

After we have translated the image of the letter into a vector $\vec{x} \in \mathbb{N}^{35}$, we will need to teach the computer how to identify which letter is portrayed in the image using the vector $\vec{x}$. We will do this through a neural network, but we will also need a surjective function mapping the vector $\vec{y} \in \mathbb{N}^{26}$ produced

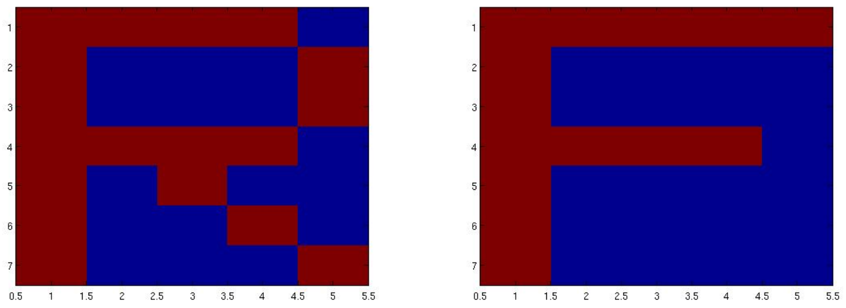


Figure 5: Other letters

by the neural network to a letter in the alphabet. This letter will represent the computer's best estimate of which letter is shown in the image.

MATLAB's algorithm for mapping the output of the neural network to a letter is relatively straight-forward. With 26 letters in the alphabet, MATLAB creates a numerical label $\vec{y} \in \mathbb{N}^{26}$, with a single entry having a value of 1 corresponding to the letter's position in the alphabet (for example, since A is the first letter in the alphabet, the corresponding vector in $\mathbb{N}^{26}$ has a 1 in its first entry). All other entries are coded 0. For example, the appropriate target for the letter A would be

$$\vec{y} = [\ 1 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, 0 \,|\, \ldots\ ]^T.$$

Since most people do not write letters as composed of thirty-five rectangles, how would we interpret partially-filled rectangles? Notice that in MATLAB's implementation, all letters are composed of rectangles. So this is a not an issue. If the entire rectangle is not filled in, we could also take the percentage of the area of each rectangle filled, and round to either 0 or 1.

Now that we have taken the letter A, assigned it a vector representation $\vec{x}$ and created a numerical label $\vec{y}$ for its target, we can begin training a neural network. This means, we need to find the parameters that best map each $\vec{x}$ to its target vector $\vec{y}$.

What form does a neural network take? The neural network discussed here has a three-layer feed-forward architecture and is generally non-linear. We say three-layer because we have an input layer, a processing–or hidden–layer, and an output layer. The neural network is also understood as a connected graph, with movement moving from the input layer to the hidden layer and finally to the output layer (hence feed-forward). In the input layer, we break up each input vector into its components. So if we have a vector $\vec{x} \in \mathbb{N}^{35}$ representing our input vector, we would take each component of $\vec{x}$ and assign it to a node, equivalent to a vertex in Graph Theory, in the input layer. In fact, it is important to know that each node in the graph of the neural network (see Figure 6) represents a real number. The output layer is where we recombine the results from the processing layer to form a vector $\hat{y} \in \mathbb{N}^{26}$ that will represent our output.

How does the neural network map $\mathbb{N}^{35} \to \mathbb{N}^{26}$? The process is not simple. The key to understanding how a neural network works is the processing layer. The processing layer is important because each node in the processing layer takes some linear combination of the entries in the input vector, adds them

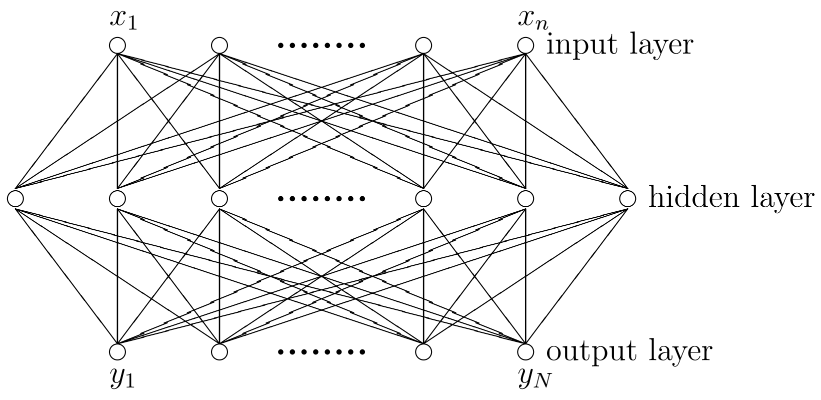


Figure 6: The neural network architecture.

together, and spits out a number close to either 1 or 0 depending on whether the linear combination is sufficiently large. This number is then passed to each of the output nodes, multiplied by a real constant, and then combined with the outputs of other nodes from the processing layer. If we focused instead on the signal (or input vector $\vec{x}$) instead of its entries, we would see that there are three steps involved in the processing layer:

- Amplifying each component of a signal $\vec{x}$ by linear weight from the vector $\vec{w}^T$ and summing the components of the resulting product

$$\vec{x} \to \vec{w}^T \vec{x}.$$

- Adding the product to the to base state b (if the total sum is sufficiently larger than b, then we want the processing node to return a 1, otherwise 0)

$$\vec{x} \to \vec{w}^T \vec{x} + b.$$

- Processing the combined signal using the formula

$$\sigma_\beta(x) = \frac{1}{1+e^{-\beta x}}.$$

The function $\sigma_\beta(x) = \frac{1}{1+e^{-\beta x}}$ (for an arbitrary constant β) is important because it is continuous and is also differentiable. Moreover, it will usually return a number close to 1 or close to 0. We use this form for theoretical calculations (e.g., when we want to take the derivative of σ_β). In actual MATLAB code, Doug Hundley [3] generally uses the inverse tangent function after normalizing the data (i.e., $\mu = 0, \sigma = 1$). The advantage of this approach is that for numbers close to zero, the inverse tangent function will return a number close to zero. For numbers between one and three, the inverse tangent function will return a number close to one. If the data is normally distributed, then over 99 percent of the data will fall within three standard deviations of the mean, which means that less than one percent of normally-distributed data would fall outside of the interval $[-3, 3]$.

A neural network is a mathematical simulation of how neurons in the brain work. Biologically, neurons have rest states, as represented by b, and when receiving a signal, will either fire a signal to the next neuron in the brain or will not. In this model, the processing layer is the neuron, the input is the signal from the previous neuron and the output is the signal sent to the next neuron. For more complex networks, we can have more than one hidden layer but one hidden layer is often sufficient.

The signal coming from each node in the hidden layer is either 1 or 0, representing whether the neuron fires a signal or not, respectively. The vector $\vec{w}$ contains the constants used to amplify the components of the input vector to a particular node (the entire signal $\vec{x}$ is sent to each hidden layer node). For our computations, we will need a different vector with real numbered entries for each node in the hidden layer. Then when we transfer the processed signal from the hidden layer to the output layer, we will need another set of vectors $\vec{w}$ to amplify the signal going to the output layer. The goal of training a neural network is to find entries for each of these weight vectors that minimize the error.

Let us now turn our attention to exactly how the signal is processed in a typical hidden layer node. If we turn back to Figure 6, we notice that each node in the hidden layer is connected to every input node. That means for each node in the hidden layer, we will need a real-valued constant to amplify each component of the inputted signal. In the character recognition example, the input has 35 entries. Thus, for each node we will need a vector of weights $\vec{w}$ with thirty five entries, each entry corresponding to the entry in the input vector. Then we will take the dot product of $\vec{x}$ and $\vec{w}$, add it to the real-valued rest state b, a real number, to get what we call the prestate. If this new sum is large enough, σ will map it to a number roughly equal to 1. Else, σ will map it to a number close to 0. The image of the combined signal under σ is called the state. In summary, if we consider the ith node in the hidden layer and a signal $\vec{x} \in \mathbb{R}^n$ with weight vector $\vec{w}_i$ (also in $\mathbb{R}^n$), we have:

$$\vec{x} \to \underbrace{\vec{w_i}^T\vec{x} + b_i}_{\text{Prestate}} \to \underbrace{\sigma\left(\vec{w_i}^T\vec{x} + b_i\right)}_{\text{State}}.$$

A graphical representation of this process is given in Figure 7.

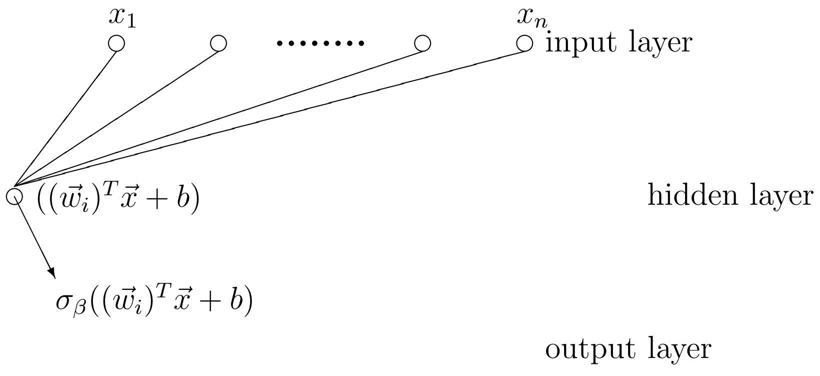

x_1
x_n
input layer
$((\vec{w}_i)^T\vec{x} + b)$
hidden layer
$\sigma_\beta((\vec{w}_i)^T\vec{x} + b)$
output layer

Figure 7: Processing the signal in the i^{th} cell

If we have more than one hidden layer node, we will need a separate weight vector $\vec{w}_i$ for each node. In this case, let $W^{(0)}$ denote the matrix of weight vectors for each of the k hidden layer cells with ${W^{(0)}}^T = [\vec{w}_1\vec{w}_2 \dots \vec{w}_k]^T$. Let the entries of $\vec{b}^{(0)}$ represent the initial states of each of the k hidden layer nodes. Since each of the signals coming out of the hidden layer toward the output layers will need to be similarly processed, we will need another matrix of weights and another vector of initial states for the output layer. Denote this matrix and this vector as $W^{(1)}$ and $\vec{b}^{(1)}$, respectively.

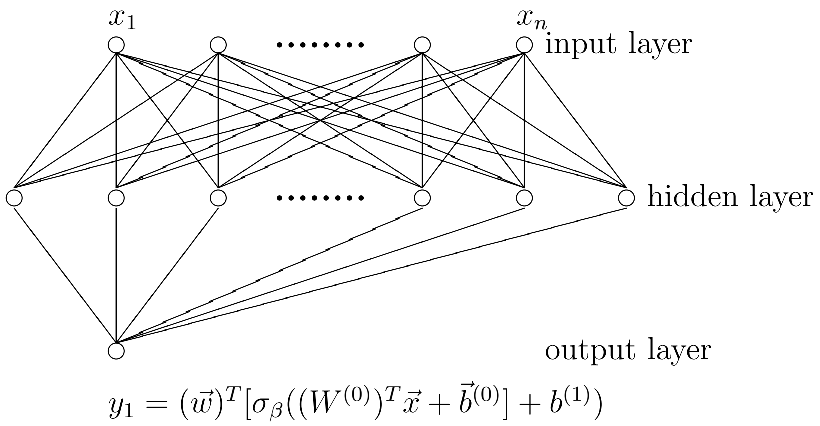
Then the first processing of the signal yields:

$$\vec{x} \to (W^{(0)})^T\vec{x} + \vec{b}^{(0)} \to \sigma((W^{(0)})^T\vec{x} + \vec{b}^{(0)}).$$

Then we can calculate our estimate, $\hat{y}$, of our target $\vec{y}$ using

$$\hat{y} = {W^{(1)}}^T[\sigma_\beta((W^{(0)})^T\vec{x} + \vec{b}^{(0)})] + \vec{b}^{(1)},$$

as shown in Figure 8.



x_1
x_n
input layer
hidden layer
output layer
$y_1 = (\vec{w})^T[\sigma_\beta((W^{(0)})^T\vec{x} + \vec{b}^{(0)}] + b^{(1)})$

Figure 8: Estimating the first component of the target

In Figure 8, we only estimated y_1, the first component of $\hat{y}$. To finish the mapping we would have to estimate the other $N-1$ components of $\vec{y}$ originally shown in Figure 6. Figure 6 and the explanation above describe how the signal is processed by a neural network to map the input vector $\vec{x}$ to some estimate of the target vector $\hat{y} \approx \vec{y}$. But how are the entries of $W^{(0)}$ and $W^{(1)}$ chosen? The entries are chosen to minimize the error. Observe we can write the error for the transformation of any one vector $\vec{x} \to \hat{y}$ as

$$E = \| \vec{y} - \hat{y} \|^2 .$$

Assuming no initial state vectors $\vec{b}^{(0)}$ and $\vec{b}^{(1)}$, the error will be a function of the weight matrices $W^{(0)}$ and $W^{(1)}$. If we have $1,000$ vector pairs (in this case, images of letters) in the data set, then we can calculate the average error by totaling the individual errors and dividing by $1,000$ (average error is by definition the total error divided by the number of observations). MATLAB uses a technique called back-propagation of error that gives you a way to calculate the gradient of the error function as you change the parameters of the weight matrices. The gradient is then used by the method of steepest descent to find entries for the weight vectors that minimize the error. Using this information, it is possible to then approximate entries that represent a local minimum of the error. With repeated initializations, it is possible to get close enough to entries associated with the global minimum. This process is called training. For more information, see Hundley [3].

In the character recognition example, we use a data bank of images of letters to train the neural network and obtain values for the entries of the weight matrices. Once we have trained the neural network, it is possible to scan new (and possibly illegible) images and correctly identify the character without human intervention. If we wanted to increase the accuracy of our network, we could consider using more than 35 rectangles, more hidden layer nodes, or a larger data bank of images, though at the cost of processing power.

What are the drawbacks to neural networks? Well for one, deciding upon the architecture and the number of hidden nodes needed is not trivial. In addition, neural networks are data-intensive, requiring an exhaustive amount of data. Consequently, they are primarily used for identification, characterization, and recognition purposes and less so for time series forecasting, though they may be applied to forecasting exchange rates and other data where availability is not an issue. A final drawback, as far as economics is concerned, is that statistical significance and values of coefficients are hard to tease out of the neural network.

4 Data Clustering

In the previous section, we learned about neural networks. In the character recognition sample, we knew ahead of time that there were twenty-six different letters in the alphabet. So each image could be mapped to one of only twenty-six vectors in $\mathbb{N}^{26}$. These vectors are called labels because they allows us to know how many groups or targets we will need. In the character recognition sample, we need to assign each input to one of twenty-six vectors.

Without labels, we would have more difficulty grouping letters. The best we could do is to group similar looking letters together. If we grouped a collection of letters, then we would expect to get at most 26 groups. Though typically, we do not know how many groups will be needed *a priori*. Data clustering is one way of tackling this problem. It is useful for reducing the dimensionality of a data set and classifying unknown data entries into groups.

Consider the data points in Figure 9. Each point lies in $\mathbb{R}^2$. The goal of data clustering is to assign each cluster to a single point in the cluster's neighborhood. In many circumstances, we want the centroid of the cluster. When the data points are already grouped together in clear and distinct bunches, our task is easy. However, this is not always the case.

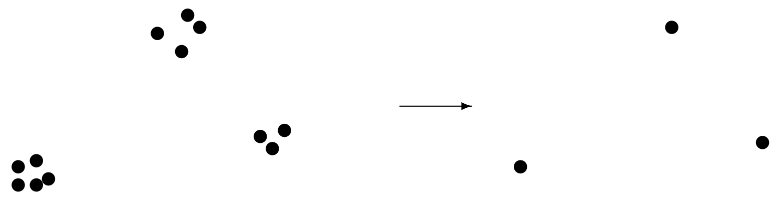


Figure 9: Example clustering

Consider data set $\mathbf{X}$, an $m \times n$ matrix. Then each row vector, $\mathbf{x}^{(i)} \in \mathbb{R}^n$. We will use *Voronoi Cells* to partition $\mathbb{R}^n$ into k cells, each cell with a specific center. Voronoi Cells will be instrumental when partitioning our data into groups. We give a definition below provided by Hundley [3].

Definition. *Let $\{\mathbf{c}^{(i)}\}_{i=1}^k$ be points in $\mathbb{R}^n$. These points define k Voronoi Cells, where the j^{th} cell is defined as:*

$$V_j = \{\mathbf{x} \in \mathbb{R}^n \textit{ such that } \parallel \mathbf{x} - \mathbf{c}^{(j)} \parallel \leq \parallel \mathbf{x} - \mathbf{c}^{(i)} \parallel, i = 1, 2, \ldots, k\}$$

If the distances of the data point to the centers of two different cells, we assign $\mathbf{x}$ to the cell with the smallest index.

In other words, we assign a data point to Voronoi Cell j if the distance (e.g., under the Euclidean metric) from the center of V_j to the data point is less than or equal to the distance of the data point to every other center.

If the data lie in $\mathbb{R}^2$ as in Figure 9, then we can partition a bounded rectangle around the points as given in Figure 10. Here, each data point is assigned to one of three cells, each with its own center. To draw Voronoi cells, first draw dashed lines connecting adjacent centers, followed by solid lines perpendicularly bisecting each of the dashed lines. These solid lines form the boundaries of the Voronoi Cells.

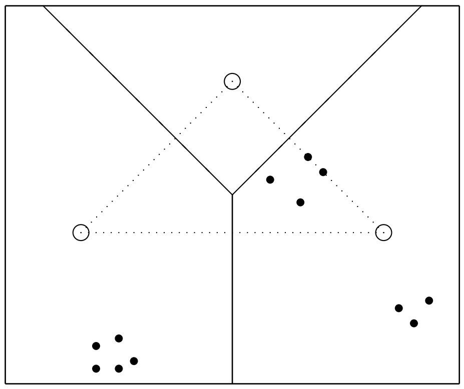


Figure 10: Example Voronoi diagram

Notice that the centers in Figure 10 are not well placed. Ideally, we would want to have a similar number of data points in each cell and to avoid empty cells. Fortunately, we have three algorithms to help us. They are the LBG algorithm, the Self-Organizing Map, and the Neural Gas Algorithm.

4.1 The LBG Algorithm

Of the three algorithms, the LBG is the simplest computationally. It uses a membership function and measurement of error in order to move the centers that represent the Voronoi Cells. The membership function allows us to partition the data set $\mathbf{X}$ into groups. For all three algorithms, the membership function assigns a data point $\mathbf{x} \in \mathbf{X}$ to the i^{th} Voronoi Cell if the distance between $\mathbf{x}$ and $\mathbf{c}^{(i)}$ is less than or equal to the distance between $\mathbf{x}$ and all of the other centers. Mathematically, if we are trying to assign each data point to one of p groups, we could write the membership function as:

$$m(\mathbf{x}) = i \text{ iff } \parallel \mathbf{x} - \mathbf{c}^{(i)} \parallel = \min_{j=1:\,p} \{ \parallel \mathbf{x} - \mathbf{c}^{(j)} \parallel \}. \tag{2}$$

We also need some measure of how good a center represents the data points in the cell. Denote the number of data points in the i^{th} cell as N_i and the total number of data points as N. Then for any one cluster, we define the *distortion error* for the cell i as:

$$E_i = \frac{1}{N_i} \sum_{k=1}^{N} \parallel \mathbf{x}^{(k)} - \mathbf{c}^{(i)} \parallel (\chi_i(\mathbf{x}^{(k)}))$$

where $\chi_i((\mathbf{x}^{(k)})) = 1$ if $\mathbf{x}^{(k)}$ is assigned to the i^{th} cell, otherwise $\chi_i((\mathbf{x}^{(k)})) = 0$. Because of the $\chi_i(\mathbf{x}^{(k)})$ term, only cells assigned to the i^{th} center by the membership function will contribute to the distortion error of the i^{th} cell. With the Euclidean metric on $\mathbb{R}^n$, the center that minimizes the error will be the centroid of the data in the cell. Also, we can define the total distortion error as the sum of the distortion errors of each of the p cells.

LBG Algorithm. *Let* $\mathbf{X}$ *be a matrix of* N *data points, and let* $\mathbf{C}$ *denote a matrix of* p *centers. We randomly initialize* p *centers and reassign each center to minimize the total distortion error. Then we:*

- *Assign each data point to center of cluster based on Equation 2.*
- *Make the new center the centroid of data in the cluster.*
- *Repeat as long as distortion error is decreasing.*

Note that the p centers are chosen randomly and then the cells are drawn. From there, the centers are then placed in the centroid of the cell. The cells are redrawn with the centroids chosen as the new centers. This process continues as long as the distortion error is declining.

Notice that in this algorithm there is no penalty for empty cells in the error calculation. So if a center originally placed with 0 data points, then the cell has no centroid and the center will not move unless by chance other data points are added to the redrawn cells. Moreover, the error for an empty Voronoi Cell is 0, even though empty cells are indicative of a poor grouping. So empty cells are not penalized and may remain empty throughout the iterations. While the LBG algorithm will certainly improve upon a randomly initialized assignment of p centers, the overall fit of the data can be heavily dependent on the initial clustering. For example, if we placed p centers throughout the plane, we may very well obtain 3 empty cells in one run while in another we may obtain none.

Although the LBG is the fastest of the three algorithms discussed here to implement, it is not without its drawbacks. For instance, the choice of p, the number of centers to use in the data set, is arbitrary and we could still end up with empty cells as in Figure 10. To prevent empty cells from occurring we could choose random data points to be our first centers. However, even this adjustment only guarantees that our initial cells have at least one data point, which is only slightly better than having empty cells. This is because the center will continually be assigned the value of the data point unless other data points are somehow added to the cell.

Together, the dependency of the quality of the clustering on the initial placement and the need for an arbitrary choice of p are two of the biggest drawbacks of the LBG. Fortunately, the Self-Organizing Map and the Neural Gas Algorithm will overcome these limitations.

4.2 Kohonen's Map

Kohonen's Map (also called the Self-Organizing Map), improves upon the LBG Algorithm because it gives us a powerful way to tailor the placement of centers to fit the data. In each iteration of the algorithm, the LBG only allows us to move the center within a given Voronoi Cell. Kohonen's Map, on the other hand, allows us to shift the centers, to move the boundaries, and to affect the membership of the data points.

The general idea of Kohonen's Map is that we will take a structure (e.g., a rectangular or hexagonal lattice) of vertices and edges and impose it upon a data set. Then we will invoke the algorithm to adjust the placement of the centers and the distances between the center until the structure is tailored to the distribution of the data.

Consider the picture displayed in Figure 11. If we suspect that the data

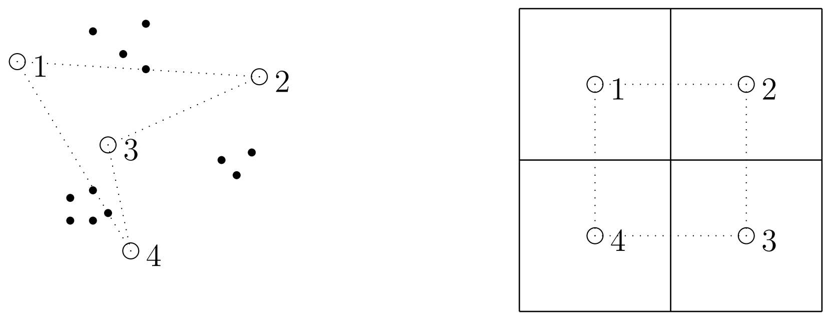


Figure 11: Initial allocation and lattice structure

can be clustered into a rectangular-like shape, then we would want a rectangular relationship defining our lattice of connections. After establishing our lattice, we would randomly place the centers in the data set as shown on the left. On the right side, the solid lines represent the boundaries of the Voronoi Cells and the dashed lines the connection between the cells. Notice that the connections are also rendered on the left side. If we choose the correct parameters for our algorithm, we should be able to get the clustering given in Figure 12. Notice that in Figure 12, neighbors in the lattice from Figure 11

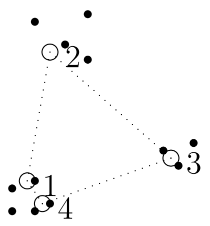


Figure 12: Hypothetical allocation

are neighbors in the diagram and that the distribution of the centers mirrors the distribution of the data points. This is what we mean when we say that we will impose the lattice on the data and tailor it to match the density of the data. We are preserving the relationships from the lattice in the data set. Ideally, we would like to also ensure that points close together in the data set will be assigned to centers close together in the original lattice. This kind of mapping is called a *topology preserving*. Kohonen's Map does not guarantee that the assignment is topology preserving, although it does preserve the data density.

So how does Kohonen's Map work? Suppose we had wanted to impose the lattice from Figure 11 on the data with the initialization of centers shown in the picture. Now we will need some way to update the centers. The general idea is that we will choose a random point $\mathbf{x}$ from the data, and move all centers toward it. However, we will want to move the closest center

$\mathbf{c}_w$ the most and centers farther away by a smaller amount. This allows us to increase the fit of the centers one at a time while having a minimal impact on the centers far away. However, we move all of the centers by some amount to ensure that all the centers will be placed inside the region containing the data points.

For Kohonen's Map, we will need two metrics: $d_I(i, w)$ and $d(\mathbf{x}$ and $\mathbf{c}_w)$. To measure distances between centers, we will use $d_I(i, j)$, which gives the distance between the i^{th} and j^{th} centers. The easiest metric is simply the number of edges connecting the centers in the lattice. In Figure 11, we would let $d_I(1, 1) = 0$, $d_I(1, 2) = 1$, and $d_I(1, 3) = 2$. The other metric $d(x, y)$ is what we will use to measure the distances between data points or between a data point and a center. For our purposes, we will use the standard Euclidean metric.

We will also need a little more notation in order to understand how the centers are updated. Let $\mathbf{c}^{(i)}(k)$ denote the center of the i^{th} cell at time k. Then $\mathbf{c}^{(i)}(k+1)$ refers to the center of the i^{th} cell at time $k+1$. Let $\mathbf{x}_k$ be the randomly chosen data point at time k with closest center $\mathbf{c}_w$. We will denote the distance between $\mathbf{c}_w$ and $\mathbf{c}^{(i)}(k)$ as $d_I(i, w)$. Also, we need two parameters $\epsilon(k)$ and λ, that control the size of the shift and the separation between the centers, respectively.

Then the formula for updating the i^{th} is:

$$\mathbf{c}^{(i)}(k+1) = \mathbf{c}^{(i)}(k) + \epsilon(k) * \exp\left(\frac{-d_I^2(i, w)}{\lambda(k)^2}\right)(\mathbf{x} - \mathbf{c}^{(i)}(k)),$$

with all terms in this formula as they were defined in the preceding paragraphs. This formula shifts all centers in the direction of $\mathbf{x}$ but shifts the closest center, $\mathbf{c}_w$, by $\epsilon(k) \times e^0 = \epsilon(k)$. The next closest center will be shifted by

$$\epsilon(k) \times e^{\frac{-1^2}{\lambda(k)^2}},$$

which will be less than $\epsilon(k)$ for values of $\lambda(k) < 1$. The centers further away in the lattice will hardly move at all. Thus, every iteration, we are moving each center in the direction of the randomly chosen data point and changing the distance between the centers. As we randomly cycle through the data points, our centers will generally move toward the centroids of the clusters, and will distribute themselves to match how the data is spread out in $\mathbb{R}^n$.

Generally, we desire adaptability early on, when we are tailoring the structure the the data; and stability later, when we wish to preserve the previously unfolded structure. Let ϵ_i and λ_i be the initial values of our parameters with ϵ_f and λ_f denoting the final values. We generally choose ϵ_i and λ_i to be relatively large and ϵ_f and λ_f to be relatively small. If we want each of these parameters to slowly approach their final values, we can let $\epsilon(k+1) = \epsilon_i(\frac{\epsilon_f}{\epsilon_i})^{\frac{k}{\texttt{tmax}}}$ and $\lambda(k+1) = \lambda_i(\frac{\lambda_f}{\lambda_i})^{\frac{k}{\texttt{tmax}}}$ where `tmax` is the number of iterations in the algorithm. This way, during the initial cycle, $k = 0$ and $(\epsilon(k), \lambda(k)) = (\epsilon_i, \lambda_i)$. During the last cycle, $k = \texttt{tmax}$, and $(\epsilon(k), \lambda(k)) = (\epsilon_f, \lambda_f)$. Now we are ready for Kohonen's Map algorithm.

Kohonen's Map.

- *Initialize* $\epsilon_i, \epsilon_f, \lambda_i, \lambda_f$, *tmax*
- *Choose random* $\mathbf{x} \in X$
- *Find closest center,* $\mathbf{c}_w$
- *Update all centers:*

$$\mathbf{c}^{(i)}(k+1) = \mathbf{c}^{(i)}(k) + \epsilon(k) * \exp\left(\frac{-d_I^2(i,w)}{\lambda(k)^2}\right)(\mathbf{x} - \mathbf{c}^{(i)}(k))$$

- *Update* λ, ϵ*.*
- *Repeat until* *tmax* *cycles have been achieved.*

After a sufficient number of cycles (there is no good way of estimating an optimal value for `tmax` beforehand), the centers will generally move toward each of the centroids of the data clusters. Kohonen's Map is sometimes called Self-Organizing Map because the original structure, whether a rectangular lattice or a sphere, will contort itself to match the data. In Figure 11, we saw that the original lattice spread out to match the data. The centers were placed inside clumps of data points, and the neighbors of the centers in the square lattice were also neighbors in the data set. Of course, this approach assumes we know an appropriate structure to impose upon the data, which may not always be the case (especially for data with higher dimension). For this reason, we will also want to consider the Neural Gas algorithm, which allows us to construct the connections between centers.

Before we begin discussing the Neural Gas Algorithm, let us instead consider an application of Kohonen's Map provided by Doug Hundley [3]. Suppose we were in charge of designing the layout of a zoo and that the zoo would house one of each of the following sixteen animals:

- Doves, hens, ducks, geese, owls, hawks, and eagles
- Foxes, dogs, and wolves
- Horses, zebras, and cows
- Cats, tigers, and lions

To simplify the layout for visitors, we would want animals with similar characteristics close by. That is, we would want to place the birds in one general cluster, the canines in another, the felines in a third, and the remaining large mammals in a fourth.

If we inspected each of the animals, we could easily make observations about size (small, medium, or large), structure (two legs or four legs, hair, hooves, mane, and feathers) and whether it likes to fly, run, or swim. With twelve characteristics, we could construct vectors in $\mathbb{R}^{12}$ for each of the animals, with each entry corresponding to a binary variable of whether the animal had a specified characteristic. So our data is in $\mathbb{R}^{12}$ but our zoo layout is in $\mathbb{R}^2$. We will want to use Kohonen's map to place animals that share similar vectors in $\mathbb{R}^{12}$ next to each other in our blueprints.

If we just randomly place the animals on the map, our blueprints would appear something like Figure 13. To generate Figure 13, we created a hexag-

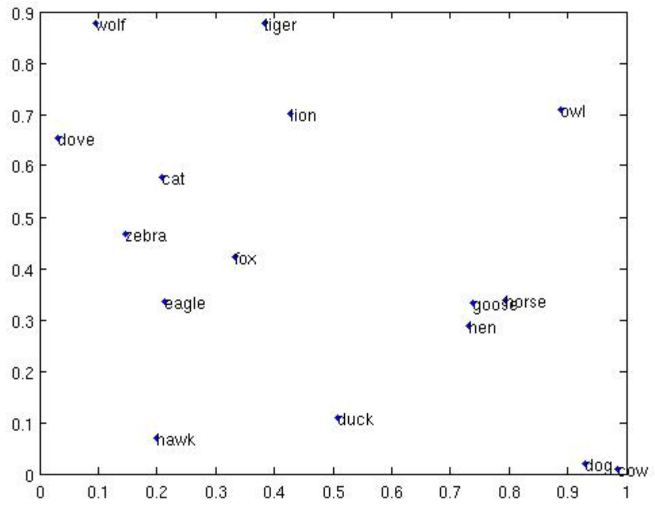


Figure 13: Initial placement

onal grid in $\mathbb{R}^2$ with each vertex representing a possible cage location. Then we assigned each cage to a random point in $\mathbb{R}^{12}$. Last, for each data point representing an animal, we found the closest center and wrote the animal's name in the location in the center's position in the lattice. Admittedly, the choice of hexagonal was arbitrary. We could have easily used a triangular or rectangular grid as well.

Observe that under the random initialization of centers, the dog is assigned to a Voronoi Cell next to the cow. Hence, on the grid, they appear as neighbors. Also, we see that the goose and the horse are adjacent, and the cat and the dove are nearby. Not only is this setup confusing for visitors but frankly, we might be concerned for the dove's safety.

In order to place similar animals next to each other, we could now proceed to update the centers according Kohonen's Map. Unfortunately, MATLAB does not provide a graphical interface to watch the centers get updated. However, from our understanding of Kohonen's Map, we know that somewhere in $\mathbb{R}^{12}$ centers adjacent in the grid will slowly be pulled closer together. Since animals sharing common features are generally close by in $\mathbb{R}^{12}$, the result is that the centers that are neighbors on the lattice will gradually be pulled closer to the locations of those same neighbors in $\mathbb{R}^{12}$. Even though the data points of the animals are stationary, the end result is that at the end of the algorithm, centers that are close according to the lattice will be placed around similar animals. Also, centers that are relatively far from each other in the lattice will drift apart. This helps ensure that similar animals are assigned to centers close together on the lattice.

The specific center to which each animal is assigned to is shown in Figure 14, though the hexagonal lattice showing the connections between centers is not visible. Neither are the centers corresponding to empty Voronoi Cells visible. From the picture, we can see that Kohonen's Map did a pretty good job. Notice how the birds are grouped in the lower right, the large mammals in the lower left, and the felines in the center left. Only the wolf, the dog, and the fox are spread out. Overall, Kohonen's Map arranged the animals by similarities relatively well. In fact, some very similar animals were placed on top of each other in Figure 14. This means that two similar animals were placed in the same Voronoi cell, though this might be something we would want to correct if we were in charge of the zoo.

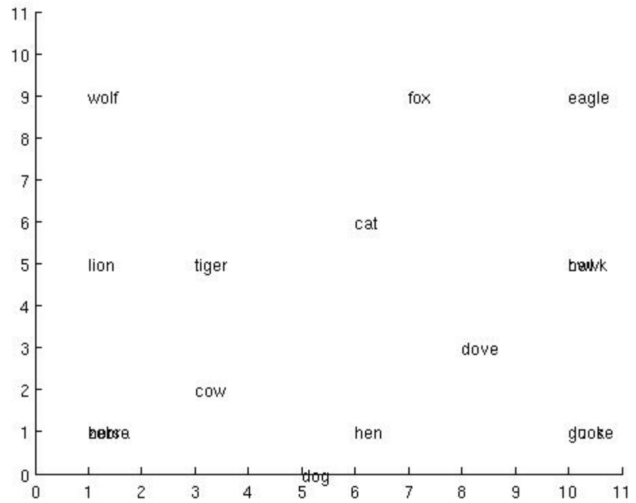


Figure 14: Placement after Kohonen's Map

4.3 The Neural Gas Algorithm

The Neural Gas Algorithm is the final data clustering algorithm we will consider. The main difference between Kohonen's Map and the Neural Gas Algorithm is that Kohonen's Map presumes a lattice to be imposed on the data set. The Neural Gas Algorithm finds a structure hidden in the data through a topology preserving map. The algorithm constructs the map by creating connections between neighboring centers during each iteration. We will only discuss the Neural Gas Algorithm from a general perspective because most of the details are similar to Kohonen's Map.

In the Neural Gas Algorithm, we will first need to randomly place the centers. We choose a random data point $\mathbf{x}$ and find its closest center $\mathbf{c}_w$, where w denotes the "winning" center because it is closest. We will use the distance metric again to find the center closest, but not equal, to $\mathbf{c}_w$, which we will denote as $\mathbf{c}_1$. Then we will connect the two centers and rank all centers based on their proximity to $\mathbf{c}_w$. In Figure 15 connecting two centers simply means that we draw an edge between them. We can construct an ordering on the p centers based on their proximity to $\mathbf{c}_w$, which we will denote as $V = \{w, i_1, i_2, i_3, \ldots, i_{p-1}\}$. At a given moment of time, $V(j)$ will represent the index of the $j^{(\text{th})}$ closest center to $\mathbf{c}_w$. Our metric for the Neural Gas Algorithm is then

$$d_{ng}(i, w) = j - 1$$

where i has position j in V. Alternatively, we could also say that $V(j) = i$. This means that $d_{ng}(w, w) = 1 - 1 = 0$, which is one of the requirements for d_{ng} to be a metric.

Next, we move all centers toward the data point, with those closest to $\mathbf{c}_w$ (according to d_{ng}) moving more than those further away. If $\mathbf{c}_w$ and $\mathbf{c}_1$ are not reconnected within a certain amount of time (as measured by cycling through data points), then we will disconnect them. In Figure 15 we would disconnect two vertices by removing the edge connecting them. After `tmax` iterations have transpired, the connections still present will be the ones in our topology preserving map.

When would we use the Neural Gas Algorithm instead of Kohonen's Map? Doug Hundley [3] provides the following example. Suppose we were trying to navigate an obstacle course where empty regions indicated obstacles and blue dots where it is safe to travel (see Figure 15). To navigate the course, all we need to know is a path between a finite number of points. In order to make it simpler to analyze, we would want to use the Neural Gas Algorithm to assign each point to a cluster and then connect neighboring clusters. When we run the Neural Gas Algorithm in MATLAB, we find that the path represented by the solid lines in Figure 15 is safe to travel. Notice that in this example, we

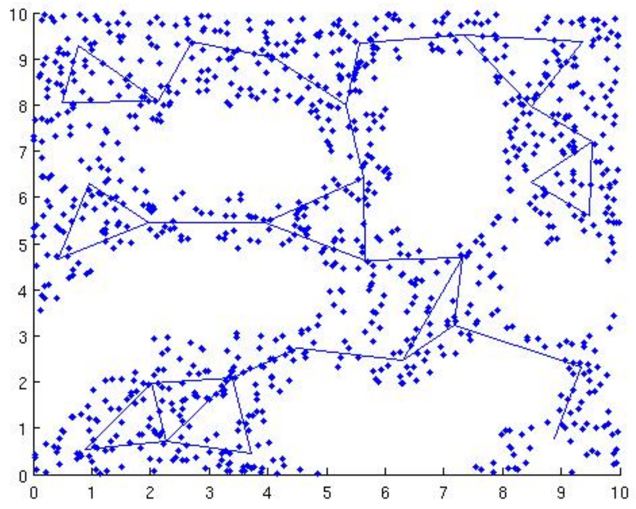


Figure 15: Obstacle course and the neural gas topology

are not concerned about the shapes of the Voronoi Cells, only the topology preserving map joining their centers.

5 The Modeling Approach: A Case Study

This application was originally conceived as an attempt to use learning algorithms to guide investment strategies. Our idea was that gambling paradigms could be used to select investments. We formulated strategies based on the n-armed bandit problem. After collecting some data–which is described in the appendix–we discovered that gambling paradigms are fundamentally ill-suited for investments. The rest of this section is dedicated to understanding why our gambling models provided poor results and adjusting our approach in light of our findings.

5.1 The n-armed Bandit Problem

What is the n-armed bandit problem? Suppose we enter a room that has n slot machines with the goal to make as much money as possible. To do so, we will have to estimate the payouts of the different machines and stick with those that are most profitable. We will need to balance the need to explore the payouts of all of the slot machines with the need to exploit the machine we think is optimal. This tradeoff is at the heart of the n-armed bandit problem. In his work [3], Hundley gives three strategies, each with varying measures of exploration and exploitation.

It is worth mentioning that this approach assumes that the payouts are what we call stationary, which simply means that the distribution of payouts is generally consistent over time. In contrast, a nonstationary time series is one in which the expected value or variance of the data changes over time. Figure 16 gives an example of a stationary sequence on the top and a non-

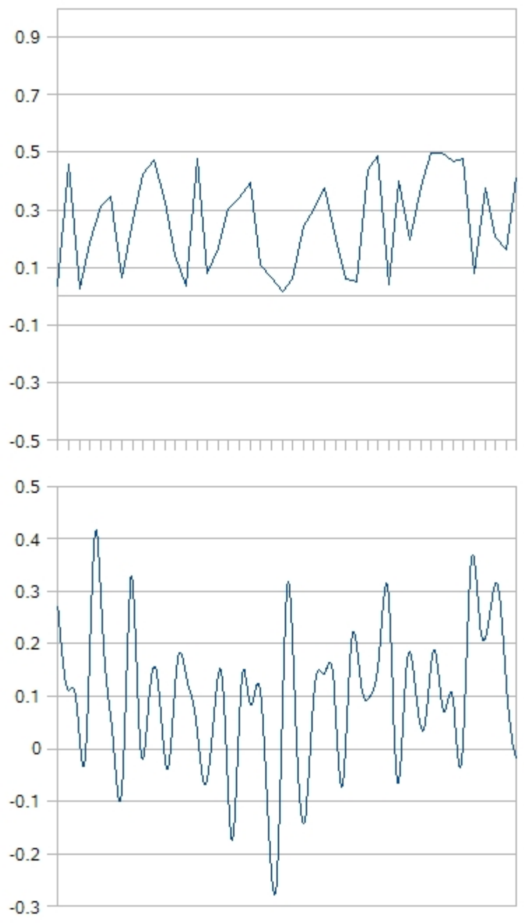


Figure 16: A stationary time series and a nonstationary one

stationary time series on the bottom. The upper series is stationary because the expected value and the variance of the series are generally constant. If we look at the lower time series, we can see that the distribution changes. You can even see a faint V-shape in the data while there are also instances of two or three sequential years that are characterized by extreme volatility. These two trends suggest that not only does the variance change, but so does the expected value. So the data is clearly nonstationary.

However, if the data set is stationary, then the n-armed bandit model works. So let us now introduce some some notation. Let

$$Q(a) = \text{The expected payout for slot machine a.}$$

Define $Q_t(a)$ as our average return for slot machine a. We could calculate

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{n_a}}{n_a}$$

with n_a indicating the number of times we have played slot machine a and r_i, for $1 \leq i \leq n_a$, denoting the return for the i^{th} trial of slot machine a. We will use $Q_t(a)$ as our estimate for $Q(a)$. If we assume that the probabilities do not change as we continue to play, then the law of large numbers states that the returns for a particular slot machine must converge to the expected return after arbitrarily many trials. That is

$$\lim_{t \to \infty} Q_t(a) = Q(a).$$

To execute our strategies, we will need to initialize $Q_0(a) = 0$ for all n slot machines. The first algorithm is the Greedy Algorithm, which says to exploit the slot machine with the highest payout. In the presence of a tie, then we would randomly select one. Define a_{t+1} as the machine we select at time $t+1$. By the Greedy Algorithm, we would then choose $a_{t+1} = i$ such that

$$Q_t(i) = \max\{Q_t(1), Q_t(2), \ldots, Q_t(n)\}.$$

If the expected returns are all negative, then this strategy will find the one that causes us to minimize our losses. If one or more of the expected returns are positive, this algorithm will likely cause us to exploit a single slot machine before we would even have the opportunity to estimate the returns of any of the other potential winners, which would clearly be undesirable.

The ϵ-Greedy algorithm attempts to correct this shortcoming. The algorithm is almost identical to the Greedy Algorithm except that we will choose $\epsilon > 0$ with ϵ being the probability that we play a random slot machine. One disadvantage of this strategy is that even after we are fairly confident we can identify the winning machine, we still explore all other machines with probability ϵ. Ideally, we would want $\epsilon \to 0$ as time progresses.

We need some mechanism that decreases the likelihood that we will explore other (non-winning) machines over time and that increases the likelihood that we exploit the winning machine. This approach is called positive reinforcement. Define $\pi_t(a)$ as the probability of choosing slot machine a at time t and a_t^* as the machine associated with the highest average return at time t. Then we will want to have $\pi_t(a^*) \to 1$ and $\pi_t(a) \to 0$ for $a \neq a^*$ as $t \to \infty$. One way to accomplish this is to move $\pi_t(a^*)$ toward one by some fixed percentage, $0 < \beta < 1$ of the difference between one and itself. Denote this percentage as β with $0 < \beta < 1$. Then we will update $\pi_t(a^*)$ using the formula

$$\pi_{t+1}(a^*) = \pi_t(a^*) + \beta[1 - \pi_t(a^*)]$$

and then moving all other probabilities, $\pi_t(a)$ toward 0 by the same β multiple of the difference separating $\pi_t(a)$ and 0

$$\pi_{t+1}(a) = \pi_t(a) + \beta[0 - \pi_t(a)].$$

This strategy allows us to sample all n machines early on but also ensures that we will eventually exploit the machine we feel most confident in, thus maximizing our winnings. Of the three strategies we considered, positive reinforcement is the one that best manages the tradeoff between exploration and exploitation. As such, it was the one we first used to guide the construction of our investment algorithms in our initial simulations.

5.2 Analytical Formulation

Analytically, we are constructing a function $f\colon \mathbb{R}^2 \to \mathbb{R}^2$. In the stocks and bonds example, our domain consists of the returns of the two investments at time t, denoted $r_s(t)$ and $r_b(t)$, respectively. Denote the percentage of our portfolio in stocks at time t as S_t. Likewise, B_t will represent the proportion of our portfolio invested in bonds at time t. We want our function to tell us how to change our portfolio, so we would write the following:

$$(S_{t+1} - S_t, B_{t+1} - B_t) = f(r_s(t), r_b(t))$$

If we state that the portfolio must be fully-invested in stocks and bonds, then we have the constraint that $S_t + B_t = 1$. Then $B_t = 1 - S_t$, and if we have a function that tells us how to update our stocks, then the constraint dictates how we must update bonds. So consider a new function $g\colon \mathbb{R}^2 \to \mathbb{R}$ such that

$$S_{t+1} = g(r_s(t), r_b(t)) + S_t$$

The function g will be used to change our allocation in our stock exposure between time periods. Like the n-armed bandit problem, g is the reward function. If we wanted static targets, then we would let $g(t) = 0$ for all t. Another possibility for g is

$$g(r_s(t), r_b(t)) = \begin{cases} \alpha & \text{when } r_s(t) > r_b(t) \\ -\alpha & \text{when } r_s(t) \leq r_b(t) \end{cases}$$

for $\alpha \in [-1, 1]$. For our empirical section, we will compare the returns generated when we set α to -1, 0, and 1.

Why are we interested in these specific forms? The premise of the n-armed bandit model, and reinforced learning in general, is that there is some feedback mechanism. The feedback mechanism is usually to converge to the "winner." Yet in the case of investments, there is some reason to believe that we might rather take advantage of value and converge to investment that underperformed in the last period. That is, we may rather want to buy low and sell high.

As an aside, notice that we only include the returns from the last period in the domain. If we included more than one period's return and we wanted to converge to the underperforming investment, the danger is that we might start to overallocate in the investment that cumulatively underperforms. If we just use the previous observation's returns, then it is easier to avoid this risk. In order to make comparisons easier, we want the domain of the function that converges to the winner to be the same as the one that converges to the loser. So we want to restrict the domain of both of these functions to $\mathbb{R}^2$. Notice that for the constant allocation functions (e.g., our standards), the choice of domain is irrelevant because the range is $\{0\}$.

Our goal is to maximize cumulative returns after accounting for costs. Although we will not introduce costs directly, we will constrain ourselves to switching a maximum of 1 percent of our assets across investments during a single time interval, measured as roughly one month. Under this assumption, $|\alpha| \leq 0.01$, and a maximum of 12 percent of our portfolio can be traded in any given year. So our expected holding time for a given investment would be over 8 years. Given that we can find no-load funds with no trading penalties if traded after a certain time-period (anywhere between three months and a year), an individual investor would have no trading costs using this approach.

We should mention that in our simulations, we use the performance of the indexes instead of the performance of the index funds because the data is more available. Since index fund returns closely mirror the returns of their index, our findings should also apply to index funds.

Let R denote the cumulative return of our portfolio at the end of our sample. If we assume negligible costs, then our cumulative return is given by

$$\begin{aligned} R &= \sum_{i=1}^{n} S_i r_s(i) + B_i r_b(i) \\ &= \sum_{i=1}^{n} S_i r_s(i) + (1 - S_i) r_b(i) \\ &= \sum_{i=1}^{n} S_i [r_s(i) - r_b(i)] + r_b(i). \end{aligned}$$

Since the rates of returns of stocks and bonds, $r_s(t)$ and $r_b(t)$, are predetermined by the market, the only way to maximize R is to choose an optimal time-path for S_t. We already assumed the recursion relation

$$S_t = S_{t-1} + g(r_s(t-1), r_b(t-1)),$$

which means that we will need to find a function g that provides superior returns, which will be our goal for the remainder of this section.

5.3 Initial Trials

Since we are trying to use the n-armed bandit model to inform our model, our hypothesis is that strategies using positive reinforcement will provide better returns. So if an asset posts a comparatively high return in one period, we would want to devote more of our portfolio to that asset in the next. Recall that for this approach to work, the returns of investments have to be stationary. Even if the payouts are not perfectly stationary, as long as the relative performance is consistent over time, our approach should work. To test whether the relative performance is consistent, we will run three quick simulations.

In our simulations, we will look at the returns of three assets: U.S. stocks, Japanese stocks, and U.S. bonds. We will further divide our total time period into three rounds covering the years 1970-1980, 1981-1990, and 1991-2008, respectively. Our goal is to assess the extent to which past performance predicts future performance.

Our first simulation is given in Figure 17. This graph shows the returns for the three assets over the first time period, which starts in 1970 and ends in 1980. The returns are shown as percentage increases. So if an asset gains 100 percent in the time period, then it essentially doubles. In the first round, we see that Japanese stocks performed the best and U.S. stocks performed the worst, with U.S. bonds performing in between.

In our second round, we observe that the relative returns of the three assets changed slightly but that the magnitudes changed severely. In Fig-

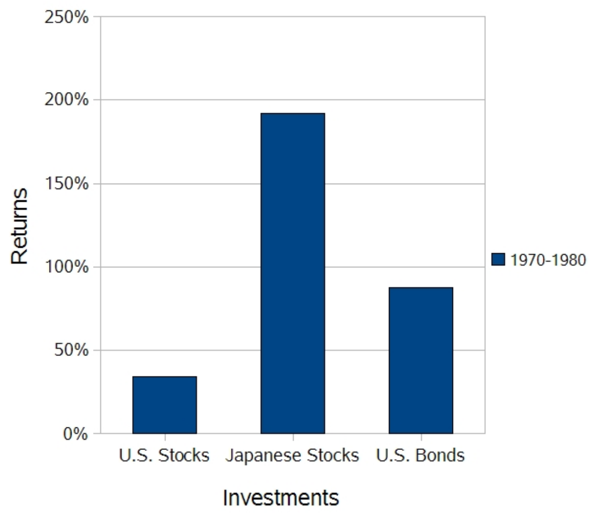


Figure 17: Returns for assets in round 1

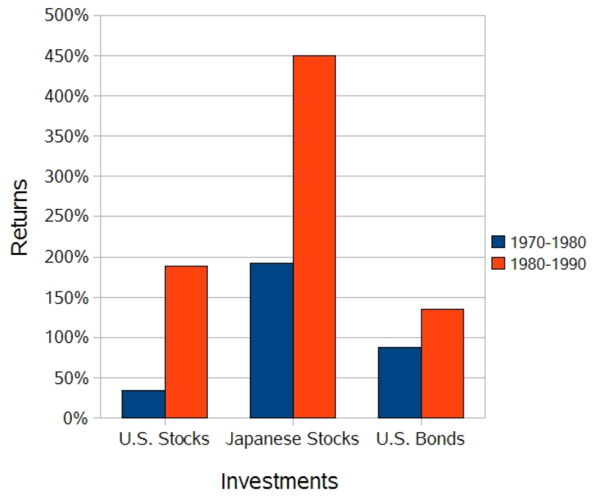


Figure 18: Returns for assets in rounds 1 and 2

ure 18, the returns for Japanese stocks in the second round were twice what they were in the first, while the ratio of the returns for U.S. stocks were even larger. U.S. bonds performed somewhat consistently but their returns also increased. In terms of relative performance, Japanese stocks posted the highest returns in both rounds but U.S. stocks overtook U.S. bonds in the second round. Overall, this suggests that the cumulative returns for assets are volatile, even if we aggregate the returns for each decade. This suggests that we may not be able to infer too much about future gains from the past.

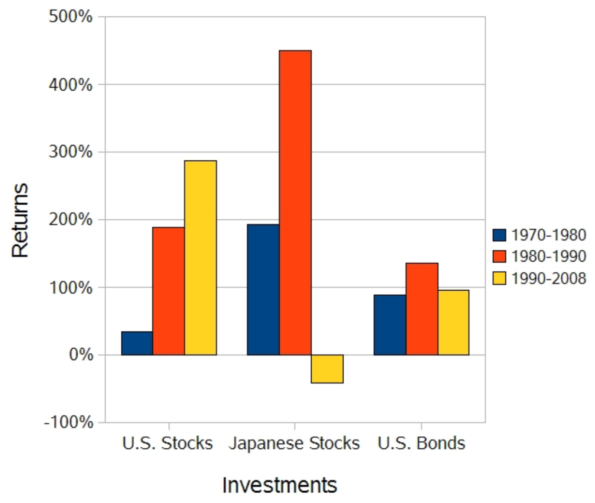


Figure 19: Returns for assets for all rounds

The returns for the last round are shown in Figure 19. Note that these returns are in dollars of the day and do not account for inflation. Since the third round lasted roughly twice as long as each of the first two, we would expect that the returns in the third round would be roughly equal to the sum of the returns in the first two rounds. Yet, only U.S. stocks met this expectation. U.S. bonds posted larger returns in the second round, which lasted ten years, than the third round, which lasted eighteen. Moreover, Japanese stocks posted significant losses, losing roughly half of their value in the third time period even though they were the strongest performers in the first two rounds.

Overall, the picture for the n-armed bandit approach is troubling. There appears to be little correlation between past returns and future returns. Moreover, the magnitudes of the returns and their relative rankings appear highly volatile. Even more disconcerting, the correlation between the past and present returns appears to be negative, which runs counter to our assumptions of stationarity. This suggests that we will need a different model, which we will develop in the next section.

5.4 Economic Theory

We saw in the last section that based on three quick simulations, our n-armed bandit model was likely to run into some difficulty. Our goal for this section is to find an economic theory that can explain the behavior of the financial assets under the last three simulations. The economic paradigm we will use is called informational cascades [11].

The premise of the model is essentially as follows. We assume that all of the information needed to accurately predict an asset's return is distributed among all investors, with no single investor having complete information. Instead, each investor has incomplete information about each asset's return.

Suppose that investor A, based on the information available to him, decides to invest in a given asset. Suppose also that investor B was indifferent between investing in that same asset and another prior to A's decision. Given the multitude of investors out there, it is reasonable to think that investor B is unsure what A knows that B does not. It is also reasonable to think that B assumes A knows something not available to him.

Under the information cascade paradigm, A's purchase will induce B to buy. After all, B was formerly indifferent but now thinks that A knows something he did not. So B will follow A's example. Then other formerly indifferent investors will purchase that same investment, generating a cascade. This cascade will continue until someone decides to sell, under which case we might find that the cascade reverses direction.

Under this paradigm, investors buy and sell in droves, suggesting that the prices of these assets will often fluctuate around their true worth, which we could define as the asset's hypothetical value if everyone had perfect information. Consequently, we will want to sell each asset at the peak of its cascade and purchase it at the trough, which is another way of saying "buy low and sell high."

So how will we implement this? Recall from the n-armed bandit section that we originally proposed shifting one percent of our portfolio from the underperforming asset to the overperforming asset, or from the losing asset to the winning one. Under this new framework of informational cascades, we will want to shift from the winning asset to the losing asset. In order to see which strategy is optimal, we will conduct formal simulations in the next section.

5.5 Further Scenarios

In this section, we will compare three different strategies. The first two should already be familiar. We will refer to the positive reinforcement strategy as the conventional approach because it was originally expected to work. We will refer to the informational cascade strategy as the contrarian approach because it runs contrary to the original model. In between is what we will call the constant approach, in which we will pick initial allocations at the beginning of the sample and stick with those allocations. So if we invested 25 percent in stocks initially, then at each trading period we will reallocate our portfolio to ensure 25 percent in stocks. If g is the function we use to update the percentages, then under the constant approach, we would set $g = 0$.

To review, we will implement the following three strategies:

- the contrarian (shift to loser)
- the constant (no shift)
- the conventional (shift to winner)

and assess how each performed under three different simulations.

In each simulation, we will allow our strategies to switch between two or three assets. We will also either trade every month or trade every 30 trading days, though we will specify which. In addition, we will measure performance in terms of the future gains for each dollar originally invested.

The first simulation assumes that our investor will choose to invest in U.S. stocks and bonds at each time period. Since Swensen [7] argues that bonds give inferior returns compared to stocks, we want to ensure that each of the three portfolios remains slanted toward equities. So we choose an initial allocation in stocks of 95 percent, with the remainder invested in bonds. However, if bonds systematically underperform stocks and our contrarian approach gravitates toward the underperforming asset, then without any correction, we will have a significant exposure in bonds. To prevent this, we will stipulate that a minimum of 90 percent of our portfolio must be invested in stocks. Note that this is only needed for the contrarian approach, because the conventional approach will gravitate toward stocks and the constant is by definition static.

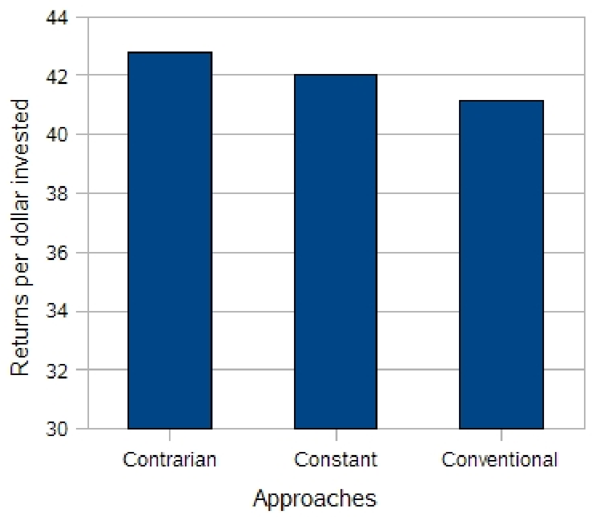


Figure 20: Performance of portfolios of U.S. stocks and bonds from 1958–2008

Figure 20 shows how each of these three strategies perform. We see that the contrarian approach outperformed the other two, as predicted by the informational cascade framework. Notice that the conventional approach performed the worst. The conventional approach underperformed the contrarian by roughly 15 percent. Since the time period was 50 years, then this means the contrarian approach only beat the conventional approach by less than 1 percent per year. At first, this may not appear that significant.

What we did not mention, however, was that the contrarian strategy not only posted superior returns but was safer as well. This is because during downturns, when stocks lose a significant portion of their value, the contrarian approach had generally maximized its bond exposure. Thus, investors who are near retirement or who desire to redeem their portfolio due to other financial circumstances may find the contrarian approach even more advantageous.

One caveat must be noted before we proceed. Although in the first simulation, we stipulated a minimum stock allocation of 90 percent, this number was somewhat arbitrary. Had we chosen a lower minimum allocation, our contrarian approach would have yielded lower returns because it would have invested more heavily in bonds. It would generally keep increasing its share of bonds until it hit the maximum allocation allowed. Since bonds perform worse than stocks over long time intervals, a larger bond exposure would likely have generated lower returns, though at lower risk. In order to compare strategies with similar average bond exposures, and hence risk, we stipulated that $0.90 \leq S_t \leq 1.00$. Had we stipulated that $0.50 \leq S_t \leq 0.60$, the relative returns would likely have been unchanged. We should note that portfolios that do not attempt to control changes in risk may produce different results.

In our second simulation, we construct portfolios consisting of U.S., U.K., and Japanese stocks from 1970–2008. We also trade every month, instead of every 45 days. Furthermore, we initially allocated fifty percent of our portfolio in U.S. stocks, with the remainder evenly split between Japanese and U.K. stocks. Since there is no reason to expect that any asset will outperform the other, we do not impose any stipulations, with the exception that we can only allocate somewhere between 0 and 100 percent of our portfolio in each asset. In the contrarian and conventional approach, we reallocate among the winning and losing assets each round, leaving the middle asset untouched.

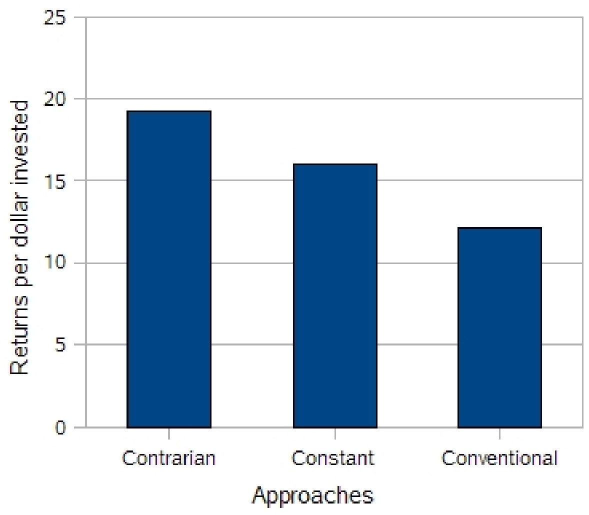


Figure 21: Performance of portfolios of U.S., U.K., and Japanese stocks from 1970–2008

From Figure 21, we see that the contrarian approach realized significantly better returns than the other two. In fact, the contrarian investor over this time period enjoyed nearly double the returns of the conventional investor. What accounted for this large difference? The answer lies in the occurrence of a Japanese stock market bubble during the 1980s. The contrarian approach performed the best because it was slowly divesting its Japanese stocks when they were increasing in value and so was only minimally exposed to the Japanese stock market when the bubble burst.

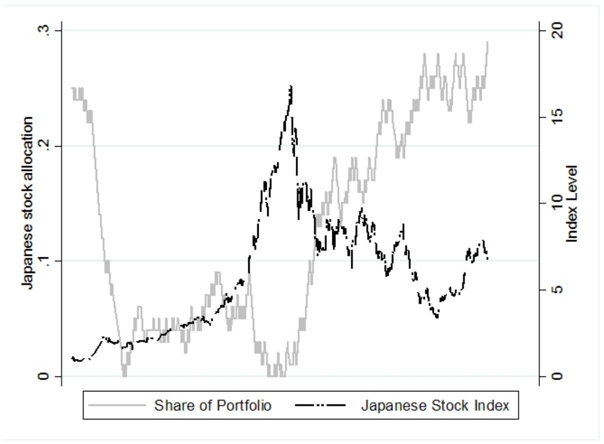


Figure 22: Allocation in Japanese stocks on left and in gray with Japanese stock index in black and on the right under the contrarian algorithm.

Figure 22 shows how our contrarian approach implemented its strategy. It depicts the level of the Japanese stock market index–with the index at 1970 equal to one–while simultaneously illustrating our contrarian approach's exposure over our time interval. There are two line graphs, each with its own corresponding axis. The gray line reveals the percentage of our portfolio invested in Japanese stocks and is measured on the left axis. The dark line shows how the Japanese stock index performed over the course of the simulation and its axis is on the right. We can see how the contrarian portfolio first decreases its exposure in the Japanese stock market as it continues to rise in the early part of our simulation. Just as the bubble burst, the average stock in the index was worth roughly fifteen times what it was worth in the beginning of the simulation. With almost no exposure during the crash, our contrarian approach experienced minimal losses and then proceeded regain its exposure in Japanese stocks once they were more realistically valued.

It is worth mentioning that because of the presence of the Japanese stock bubble, the returns of each of the three strategies depend on the time period chosen. Had we chosen our sample to start from 1980 or 1988, then the contrarian approach would have borne the brunt of the bubble and would have invested in Japanese stocks after they were corrected. Since Japanese stocks never recovered from the crash, this means that the contrarian approach would have underperformed the other two. So judgment about current conditions is critical for all investors, especially beginning investors. An investor would be wise to look at the performance over the last ten years or more before choosing initial allocations.

Our last simulation spans from 1970 to 2008 and concerns U.S. stocks and the Europe-Asian-and-Far-East financial index, which was designed by Morgan Stanley to capture the average returns in the European, Asian, and Pacific regions. Although it is impossible to invest in the index directly, there are index funds designed to mirror the index, though the availability of index fund data is much more limited, which is why we use the index values for our simulations. Our initial allocation in U.S. stocks was sixty percent, with the rest invested in the EAFE index.

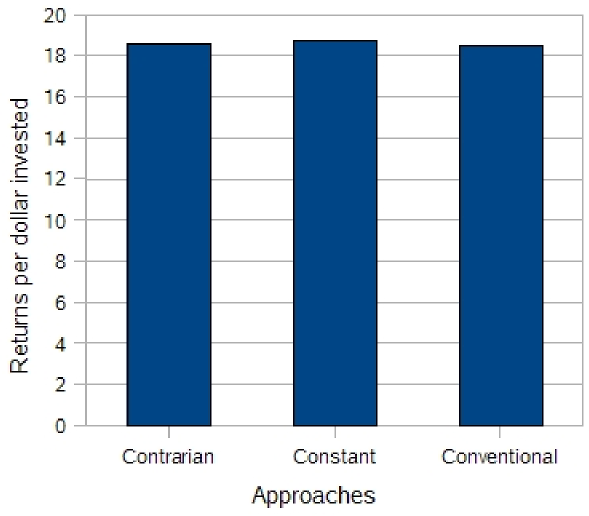
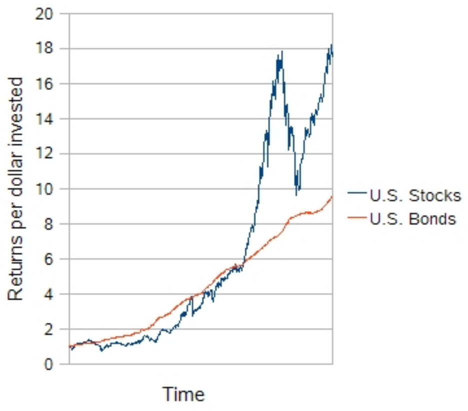
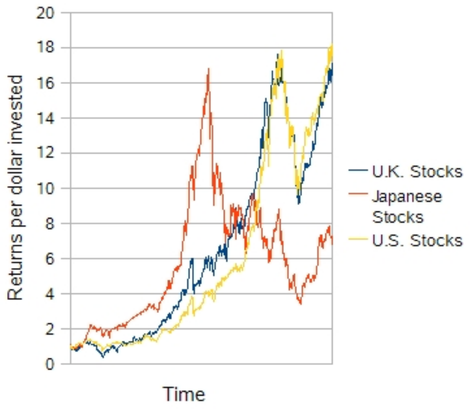


Figure 23: Performance of portfolios of U.S. stocks and the EAFE index from 1970–2008

Figure 23 shows that there is almost no difference between the returns of the three strategies after thirty years, which is rather surprising. How could this happen? Our theory of cascades tells us that we want to take advantages of the cascades to sell at the peaks and buy at the troughs. Yet if the cascades of both investments are synchronized, then there is no opportunity to switch out when desired. If we look at Figure 24, we can see that the EAFE index is more synchronized with U.S. stocks than the portfolios in the first two scenarios over the years 1970 to 2008. The synchronization





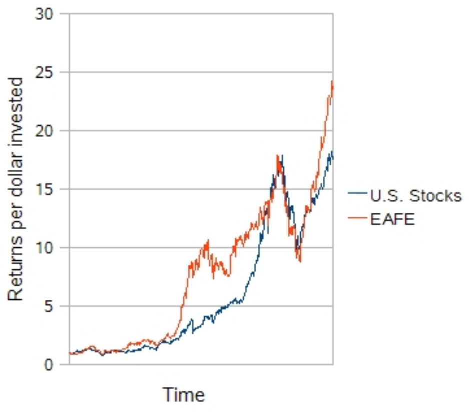


Figure 24: Cascades for simulations 1, 2, and 3, respectively

is apparent because both the EAFE and U.S. stocks hit peaks and troughs around the same time. Moreover, they exhibit similar cumulative returns for most of the time sample.

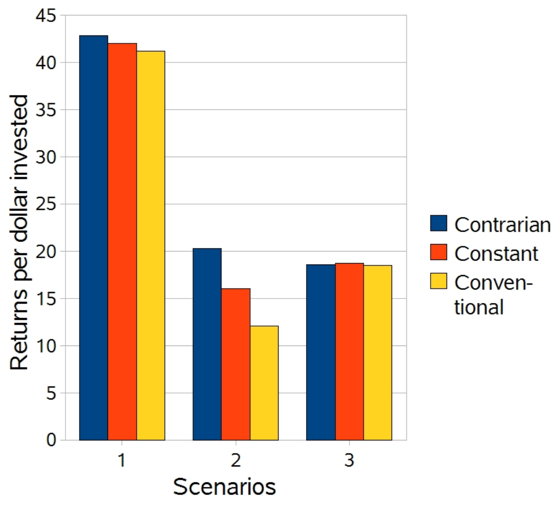


Figure 25: A summary of the three simulations

Figure 25 presents a summary of our three simulations. In two of the simulations, the contrarian approach performed best. In the last simulation, the differences were negligible because of reasons already discussed. These results suggest that our cascade model works and that the n-armed bandit model was flawed as an investment strategy.

So what can we conclude? Implemented correctly, the contrarian approach can provide higher yields and may reduce the risk from speculation, or stock bubbles. However, under certain conditions it may also provide no additional returns or may even yield lower returns. As we noted previously, the returns are highly sensitive to initial conditions and risk tolerance. Our advice for a beginning investor is to look at the last ten years or so before choosing initial investments and to use judgment when initializing starting allocations.

6 Conclusion

In our application, we first tried to frame the asset-decision problem for a portfolio in terms of the n-armed bandit model. However, this model assumes stationarity. Given the surprising nonstationary behavior evident in our assets' returns, we then turned to economic theory for guidance in formulating alternative approaches. We found that an algorithm inspired by the theory of informational cascades, which simply increases its exposure in recently underperforming assets, appears to outperform the other approaches considered in two out of the three simulations, with no significant difference in the third scenario. In the future, we would want to assess how to handle portfolios consisting of several assets instead of the two- or three-asset portfolios considered in this paper.

A Choice of α

One logical question to ask is how our contrarian algorithm would have performed with a different choice of α. Recall that α is the variable we are using to rebalance our portfolios in the contrarian and conventional strategies. Also, if α is too big, then the turnover in our assets may limit our performance.

We will consider how the returns in each of the scenarios varies for different values of α in $[0, 2]$. The first scenario we looked at involved a portfolio with U.S. stocks and bonds from 1958 to 2008. Figure 26 shows that the returns were not too sensitive on our choice of α. The optimal choice of α in this interval appears to be somewhere between 1 and 1.5 percent.

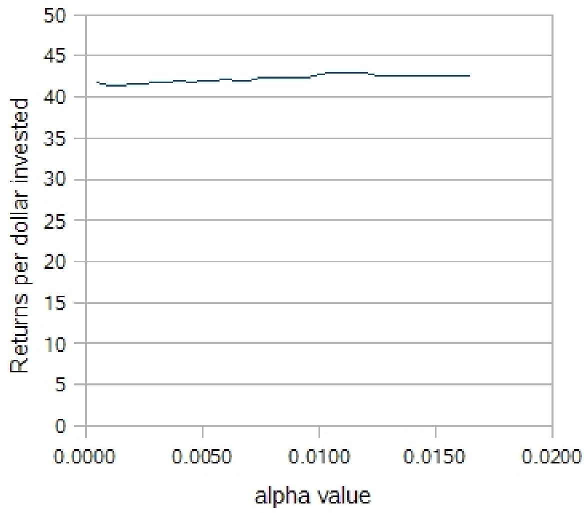


Figure 26: Returns for each α in scenario 1

In the second scenario, concerning Japanese, U.K., and U.S. stocks, the returns seemed to be more sensitive to α. Notice from Figure 27 that the returns increase relatively rapidly as α moves from 0 to 1. Just before 1, the slope decreases before increasing again. This is likely due to the fact that right around 1, we hit our constraint. Remember that we stipulated that we could not hold negative assets, or sell assets we did not have (in the real world, this can be done by "shorting" on an investment). For values of α near 1, we become completely divested of Japanese stocks just before the Japanese stock market bursts. One of the primary reasons that the contrarian strategy out performed the others in this scenario is because it whittled away its exposure to the Japanese stocks when they were surging. Thus, once we are divested from Japanese stocks, there are fewer opportunities for our algorithm to take advantage of, so the slope flattens. Note however that returns still increase as α increases, so there are clearly some further gains associated with an increasing α.

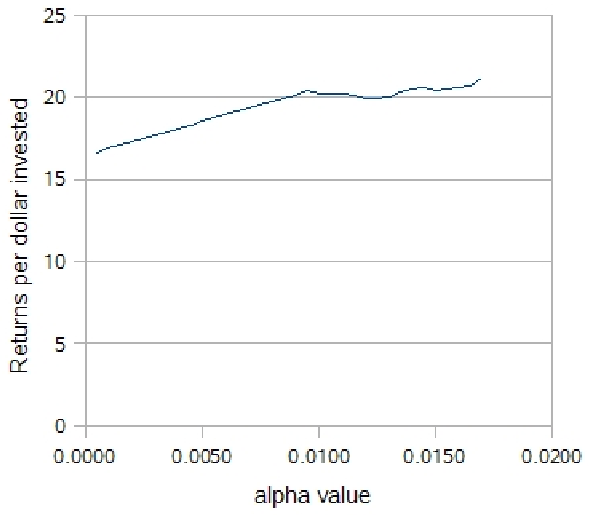


Figure 27: Returns for each α in scenario 2

In the third scenario, we see that increasing α actually lowers our returns, though the returns appear to be less sensitive to α compared to the second scenario. All in all, Figure 28 reveals that our contrarian approach is not infallible and that a smaller α minimizes our risk that the contrarian approach

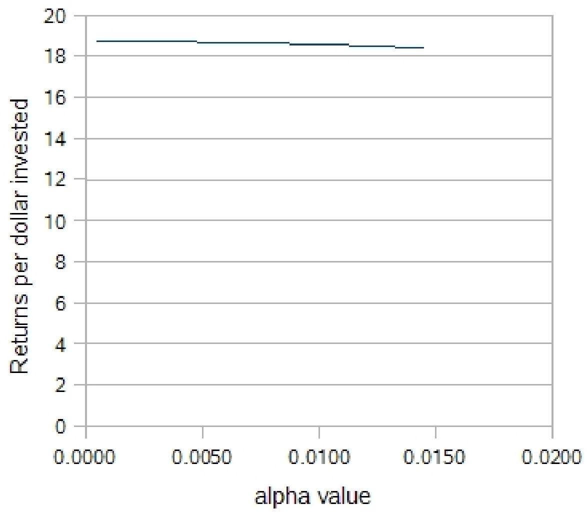


Figure 28: Returns for each α in scenario 3

will significantly underperform a portfolio with static targets.

B Data for the Empirical Application

The data used in the application came from a variety of sources. Most instrumental was Yahoo Finance. Daily data for the returns of bonds came from the website of the Federal Reserve Bank in St. Louis while the monthly data concerning the MSCI EAFE was obtained from the MSCIBarra website. Last, the data for the Nikkei 225 and the FTSE 100 was obtained in from a combination of Yahoo Finance and Wren Research, the latter of which is available at `http://www.wrenresearch.com.au/downloads/index.htm`.

The stock and bond daily data sets were combined. Any observation that was missing for one of the assets was dropped for the other and the data were merged by date. Then every 30 observations were combined and the cumulative return was calculated as the product of the daily returns. The monthly returns were calculated directly with minimal manipulation.

The bond market returns was calculated as follows. First, we calculated the value of newly-issued bonds (6-month Treasury Bills) on the secondary market by subtracting the interest rate from 100. (According to the author's research, short-term government interest rates are implicit. For example, a one-year bond carrying a 7% interest rate would be purchased for roughly $93 and redeemed for $100). Using this information, we can calculate the change in value over time according to changes in the interest rate for newly issued bonds. Then to find the total return for a bond, you add the percent change in value together with the amount of interest accrued while the bond was held. In total, this gives a total long-term return consistent with the returns posted on the bond markets.

References

[1] The Mathworks Inc. (n.d.). Image types in the toolbox. In Image processing toolbox. Retrieved at `http://www.mathworks.com/access/helpdesk/help/toolbox/images/f14-13543.html`.

[2] D. C. Lay, *Linear algebra and its applications.* Addison Wesley, 2003.

[3] D. R. Hundley *An introduction to empirical modeling.* Unpublished Available at `http://people.whitman.edu/hundledr/courses/M350.html`.

[4] S. Gonzalez, "Neural networks for macroeconomic forecasting: a complementary approach to linear regression models," Working Papers–Department of Finance Canada 2000–2007, Department of Finance

Canada, undated. Retrieved at `http://ideas.repec.org/p/fca/wpfnca/2000-07.html`.

[5] G. Dangelmayr, S. Gadaleta, D. Hundley, and M. J. Kirby, "Time series prediction by estimating Markov probabilities through topology preserving maps," in *Proc. SPIE Vol. 3812, p. 86-93, Applications of Science and Neural Networks, Fuzzy Systems, and Evolutionary Computation II, Bruno Bosacchi; David B. Fogel; James C. Bezdek; Eds.*, presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, pp. 86-93, Nov. 1999.

[6] G. Fabrikant, "Yale endowment grows 28%, topping $22 billion," *New York Times*, 2007.

[7] D. F. Swensen, *Unconventional success: a fundamental approach to personal investing*. Free Press, 2005.

[8] "Chimp beats stockbrokers in their trade," *Seattle Times*, 2007.

[9] B. G. Malkiel, "Returns from investing in equity mutual funds 1971 to 1991," *The Journal of Finance*, vol. 50, pp. 549–572, June 1995.

[10] A. Baker, "Money for old hope," *The Economist*, February 2008.

[11] S. Bikhchandani, D. Hirshleifer, and I. Welch, "Learning from the behavior of others: conformity, fads, and informational cascades," *The Journal of Economic Perspectives*, vol. 12, pp. 151-170, Summer 1998.