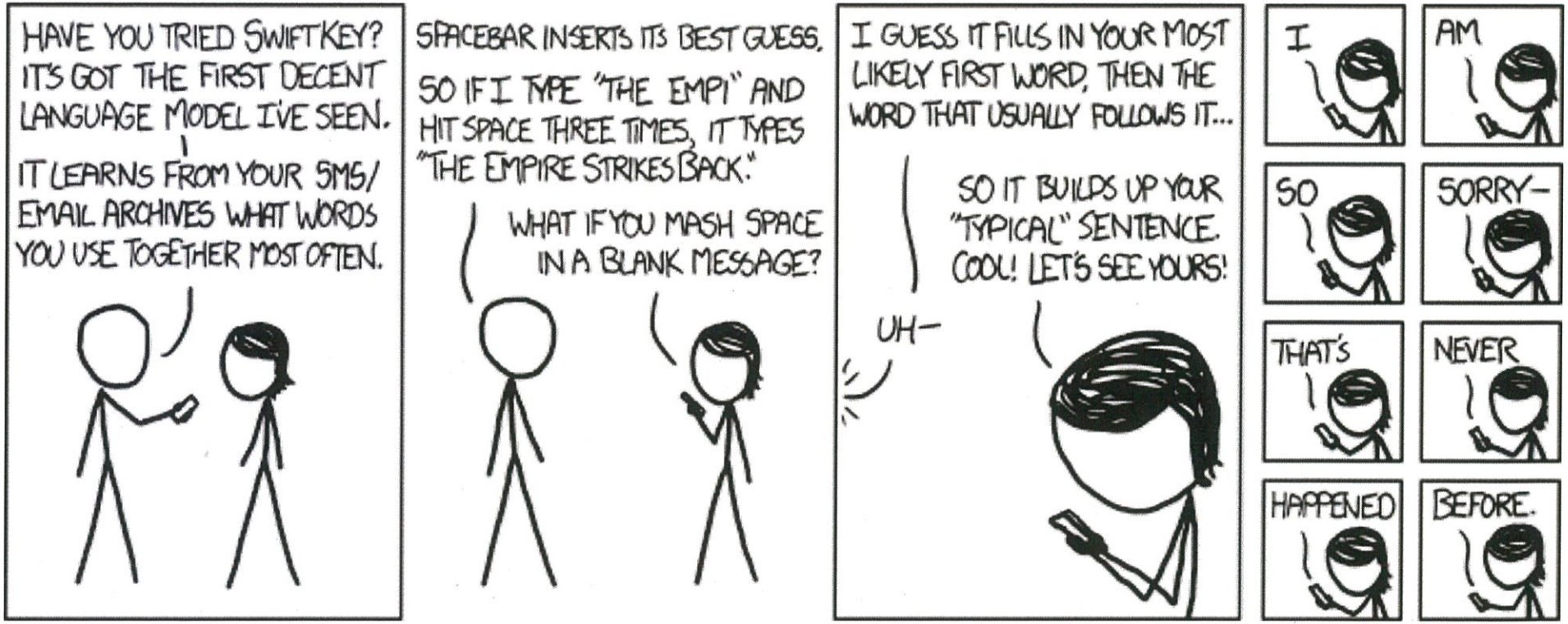
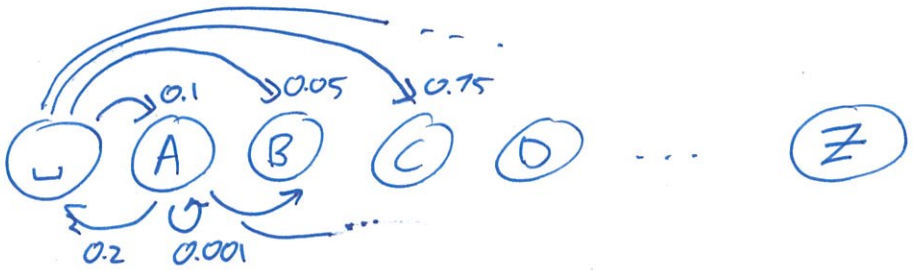


From https://xkcd.com/1068

How can we generate text predictions?

Let's use a Markov chain model:

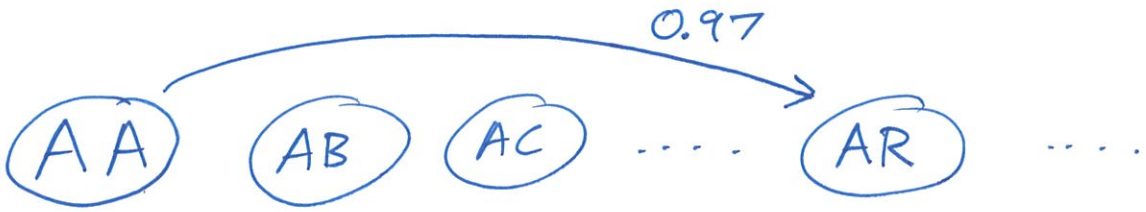


We can pick the probabilities by looking at a corpus, and measuring the frequency of certain letter pairs.

For example, above we've given the A→A transition probability as 0.001, because AA is rather infrequent in English. (e.g. AARDVARK, ...?)

If we use A-Z and a space, we'll produce a 27×27 stochastic matrix.

In a more advanced model, the possible states of the system reflect the last two characters seen:



Here, the transition probability from AA to AR is high, because in a typical English text if you see the letters AA, it's because you're reading the word AARDVARK.

Again, we can produce a stochastic matrix from a corpus.
(The probability of AB → CD will always be zero, since B ≠ C.)

The best description of Markov chains I've ever read is in chapter 15 of Programming Pearls:

> A generator can make more interesting text by making each letter a random function of its predecessor. We could, therefore, read a sample text and count how many times every letter follows an A, how many times they follow a B, and so on for each letter of the alphabet. When we write the random text, we produce the next letter as a random function of the current letter. The Order-1 text was made by exactly this scheme:
>
> > **t I amy, vin. id wht omanly heay atuss n macon aresethe hired boutwhe t, tl, ad torurest t plur I wit hengamind tarer-plarody thishand.**
>
> We can extend this idea to longer sequences of letters. The order-2 text was made by generating each letter as a function of the two letters preceding it (a letter pair is often called a digram). The digram TH, for instance, is often followed in English by the vowels A, E, I, O, U and Y, less frequently by R and W, and rarely by other letters.
>
> > **Ther I the heingoind of-pleat, blur it dwere wing waske hat trooss. Yout lar on wassing, an sit." "Yould," "I that vide was nots ther.**
>
> The order-3 text is built by choosing the next letter as a function of the three previous letters (a trigram).
>
> > **I has them the saw the secorrow. And wintails on my my ent, thinks, fore voyager lanated the been elsed helder was of him a very free bottlemarkable,**
>
> By the time we get to the order-4 text, most words are English, and you might not be surprised to learn that it was generated from a Sherlock Holmes story ("The Adventure of Abbey Grange").
>
> > **His heard." "Exactly he very glad trouble, and by Hopkins! That it on of the who difficentralia. He rushed likely?" "Blood night that.**

from http://blog.codinghorror.com/markov-and-you/

```
"""
From
http://agiliq.com/blog/2009/06/generating-pseudo-random-text-with-markov-chains-u/
"""

import random

class Markov(object):

    def __init__(self, open_file):
        self.cache = {}
        self.open_file = open_file
        self.words = self.file_to_words()
        self.word_size = len(self.words)
        self.database()


    def file_to_words(self):
        self.open_file.seek(0)
        data = self.open_file.read()
        words = data.split()
        return words


    def triples(self):
        """ Generates triples from the given data string. So if our string were
                "What a lovely day", we'd generate (What, a, lovely) and then
                (a, lovely, day).
        """

        if len(self.words) < 3:
            return

        for i in range(len(self.words) - 2):
            yield (self.words[i], self.words[i+1], self.words[i+2])

    def database(self):
        for w1, w2, w3 in self.triples():
            key = (w1, w2)
            if key in self.cache:
                self.cache[key].append(w3)
            else:
                self.cache[key] = [w3]

    def generate_markov_text(self, size=25):
        seed = random.randint(0, self.word_size-3)
        seed_word, next_word = self.words[seed], self.words[seed+1]
        w1, w2 = seed_word, next_word
        gen_words = []
        for i in xrange(size):
            gen_words.append(w1)
            w1, w2 = w2, random.choice(self.cache[(w1, w2)])
        gen_words.append(w2)
        return ' '.join(gen_words)
```

```
CMAMac1:Lecture24 scott$ python
Python 2.7.10 (default, Aug 28 2015, 07:26:38)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)] on d
Type "help", "copyright", "credits" or "license" for more i
>>> import markovgen
>>> markov = markovgen.Markov(open('harry-potter.txt'))
>>> markov.generate_markov_text(100)
think outside of the field. Chris suddenly whacked a
Bludger in the end of our date, when I was able to
play Aqua Nero. It deals damage, divided however I
like the Slytherin stands, though it was the fact that
he was sure Ron didnt say anything, he had extremely
loose rainbow suspenders, and over to the one he had
beaten just Voldemort, but with its rather dull ring
and not get much worse. "Meow..." came a very good
point," said Dumbledore, zipping up his chair at the
hand was not a wizard head on out and let her go..." Harry
```

Rooter: A Methodology for the Typical Unification of Access Points and Redundancy

Jeremy Stribling, Daniel Aguayo and Maxwell Krohn

ABSTRACT

Many physicists would agree that, had it not been for congestion control, the evaluation of web browsers might never have occurred. In fact, few hackers worldwide would disagree with the essential unification of voice-over-IP and public-private key pair. In order to solve this riddle, we confirm that SMPs can be made stochastic, cacheable, and interposable.

I. INTRODUCTION

Many scholars would agree that, had it not been for active networks, the simulation of Lamport clocks might never have occurred. The notion that end-users synchronize with the investigation of Markov models is rarely outdated. A theoretical grand challenge in theory is the important unification of virtual machines and real-time theory. To what extent can web browsers be constructed to achieve this purpose?

Certainly, the usual methods for the emulation of Smalltalk that paved the way for the investigation of rasterization do not apply in this area. In the opinions of many, despite the fact that conventional wisdom states that this grand challenge is continuously answered by the study of access points, we believe that a different solution is necessary. It should be noted that Rooter runs in $\Omega(\log \log n)$ time. Certainly, the shortcoming of this type of solution, however, is that compilers and superpages are mostly incompatible. Despite the fact that similar methodologies visualize XML, we surmount this issue without synthesizing distributed archetypes.

We question the need for digital-to-analog converters. It should be noted that we allow DHCP to harness homogeneous epistemologies without the evaluation of evolutionary programming [2], [12], [14]. Contrarily, the lookaside buffer might not be the panacea that end-users expected. However, this method is never considered confusing. Our approach turns the knowledge-base communication sledgehammer into a scalpel.

Our focus in our research is not on whether symmetric encryption and expert systems are largely incompatible, but rather on proposing new flexible symmetries (Rooter). Indeed, active networks and virtual machines have a long history of collaborating in this manner. The basic tenet of this solution is the refinement of Scheme. The disadvantage of this type of approach, however, is that public-private key pair and red-black trees are rarely incompatible. The usual methods for the visualization of RPCs do not apply in this area. Therefore, we see no reason not to use electronic modalities to measure the improvement of hierarchical databases.

The rest of this paper is organized as follows. For starters, we motivate the need for fiber-optic cables. We place our work in context with the prior work in this area. To address this obstacle, we disprove that even though the much-tauted autonomous algorithm for the construction of digital-to-analog converters by Jones [10] is NP-complete, object-oriented languages can be made signed, decentralized, and signed. Along these same lines, to accomplish this mission, we concentrate our efforts on showing that the famous ubiquitous algorithm for the exploration of robots by Sato et al. runs in $\Omega((n + \log n))$ time [22]. In the end, we conclude.

II. ARCHITECTURE

Our research is principled. Consider the early methodology by Martin and Smith; our model is similar, but will actually overcome this grand challenge. Despite the fact that such a claim at first glance seems unexpected, it is buffetted by previous work in the field. Any significant development of secure theory will clearly require that the acclaimed real-time algorithm for the refinement of write-ahead logging by Edward Feigenbaum et al. [15] is impossible; our application is no different. This may or may not actually hold in reality. We consider an application consisting of n access points. Next, the model for our heuristic consists of four independent components: simulated annealing, active networks, flexible modalities, and the study of reinforcement learning.

We consider an algorithm consisting of n semaphores. Any unproven synthesis of introspective methodologies will clearly require that the well-known reliable algorithm for the investigation of randomized algorithms by Zheng is in Co-NP; our application is no different. The question is, will Rooter satisfy all of these assumptions? No.

Reality aside, we would like to deploy a methodology for how Rooter might behave in theory. Furthermore, consider the early architecture by Sato; our methodology is similar, but will actually achieve this goal. despite the results by Ken Thompson, we can disconfirm that expert systems can be made amphibious, highly-available, and linear-time. See our prior technical report [9] for details.

III. IMPLEMENTATION

Our implementation of our approach is low-energy, Bayesian, and introspective. Further, the 91 C files contains about 8969 lines of SmallTalk. Rooter requires root access in order to locate mobile communication. Despite the fact that we have not yet optimized for complexity, this should be simple once we finish designing the server daemon. Overall,

Google Page Rank

(Brin, Page 1998: The anatomy of a large-scale hypertextual Web search engine.)

An early version of Google's ranking algorithm was based on Markov chains.

Consider a Markov chain with a state for each web page.

The transition probabilities are given by randomly following a link on the current page.

Additionally, there's a "damping factor", $d \approx 0.85$.

With probability $1-d$, instead of following a link we jump to a completely random page.

The "Page Rank" of a page is the fraction of the time (in the long run) that the Markov chain spends at that page.

Let's write the stochastic matrix as M, and let

$$x_{k+1} = M x_k.$$

From the description above, we have

$$(x_{k+1})_j = (1-d) + d \sum_{i=1}^{N} \frac{a_{ij} (x_k)_i}{\deg_i}$$

where: a_{ij} is the number of links from page i to page j,

$\deg_i$ is the total number of outgoing links on page i.

~~Notice: links from right~~

We could try to compute the eigenvectors for M, since we know the probabilities in the long run are the entries of the eigenvector with eigenvalue 1.

In practice, it's easier to just compute $M^{1000000} \begin{bmatrix} \frac{1}{N} \\ \frac{1}{N} \\ \vdots \\ \frac{1}{N} \end{bmatrix}$.

Notice: • links from high ranking pages make your Page Rank go up

• links from many pages make your Page Rank go up.

WELCOME TO THE GATHERING PLACE

Computer Networks and ISDN Systems 30 (1998) 107–117

COMPUTER NETWORKS and ISDN SYSTEMS

The anatomy of a large-scale hypertextual Web search engine [1]

Sergey Brin [2], Lawrence Page [*,2]

Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/

To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of Web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the Web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and Web proliferation, creating a Web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale Web search engine — the first such detailed public description we know of to date.

Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want.

Keywords: World Wide Web; Search engines; Information retrieval; PageRank; Google

1. Introduction

The Web creates new challenges for information retrieval. The amount of information on the Web is growing rapidly, as well as the number of new users inexperienced in the art of Web research. People are likely to surf the Web using its link graph, often starting with high quality human maintained indices such as **Yahoo!** [3] or with search engines. Human maintained lists cover popular topics effectively but are subjective, expensive to build and maintain, slow to improve, and cannot cover all esoteric topics. Automated search engines that rely on keyword matching usually return too many low quality matches. To make matters worse, some advertisers attempt to gain people's attention by taking measures meant to mislead

* Corresponding author.

[1] There are two versions of this paper — a longer full version and a shorter printed version. The full version is available on the Web and the conference CD-ROM.

[2] E-mail: {sergey, page}@cs.stanford.edu

[3] http://www.yahoo.com/

0169-7552/98/$19.00
PII S0169-7552(98)00110-X

Claim Any $m \times n$ matrix A has a singular value decomposition:

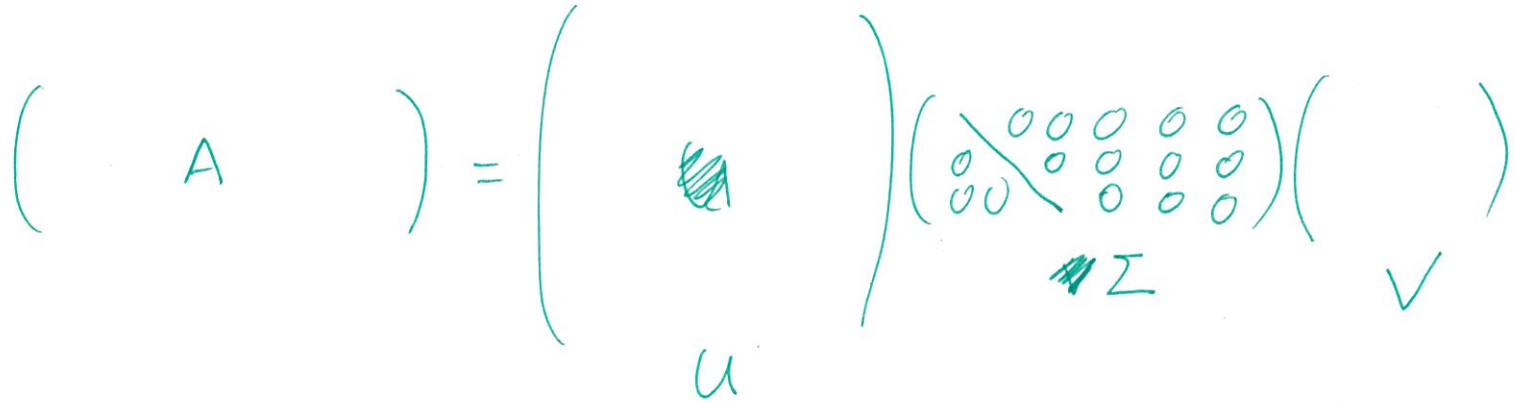
$$A = U \Sigma V^T$$

where U is an $m \times m$ orthogonal matrix

Σ is an $m \times n$ 'diagonal' matrix

with non-negative entries in decreasing order

V is an $n \times n$ orthogonal matrix.



Idea: $A^TA = V\Sigma^T U^T U \Sigma V^T = V\Sigma^T\Sigma V^T = V\Sigma^2 V^T$.

If we can diagonalise $A^TA = PDP^{-1}$, then $V = P$, $\Sigma = \sqrt{D}$, and we can solve for U.

Example

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} -0.57 & -0.82 \\ -0.82 & 0.57 \end{pmatrix} \begin{pmatrix} 6.5 & 0 & 0 \\ 0 & 0.37 & 0 \end{pmatrix} \begin{pmatrix} -0.34 & -0.55 & -0.76 \\ 0.85 & 0.17 & -0.50 \\ 0.41 & -0.82 & -0.40 \end{pmatrix}$$

This entry is pretty small.

What happens if we ignore it?

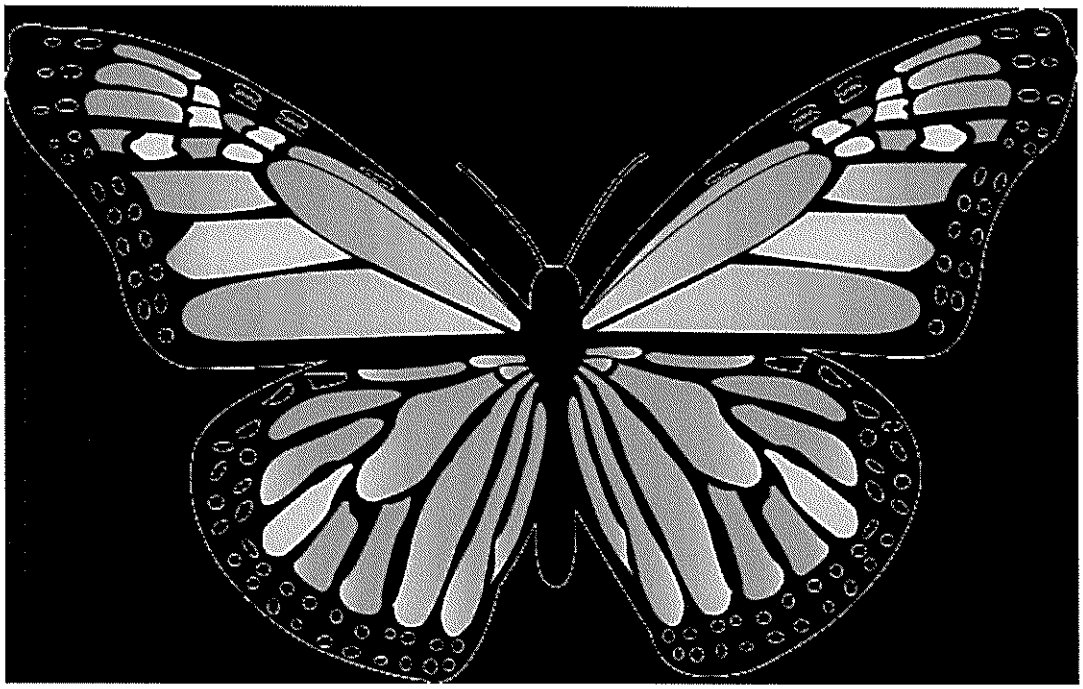
$$\approx \begin{pmatrix} -0.57 \\ -0.82 \end{pmatrix} (6.5) \begin{pmatrix} -0.34 & -0.55 & 0.76 \end{pmatrix}$$

$$= \begin{pmatrix} 1.3 & 2.1 & 2.8 \\ 1.8 & 3.0 & 4.1 \end{pmatrix}$$

Not so bad!

In[45]:=
```
Image[data = ImageData[
      ColorConvert[Import[FileNameJoin[{NotebookDirectory[], "monarch.gif"}]],
       "Grayscale"]][[All, All, 1]]]
```

Out[45]=



In[44]:=
```
ByteCount[data]
```

Out[44]= 1 828 952

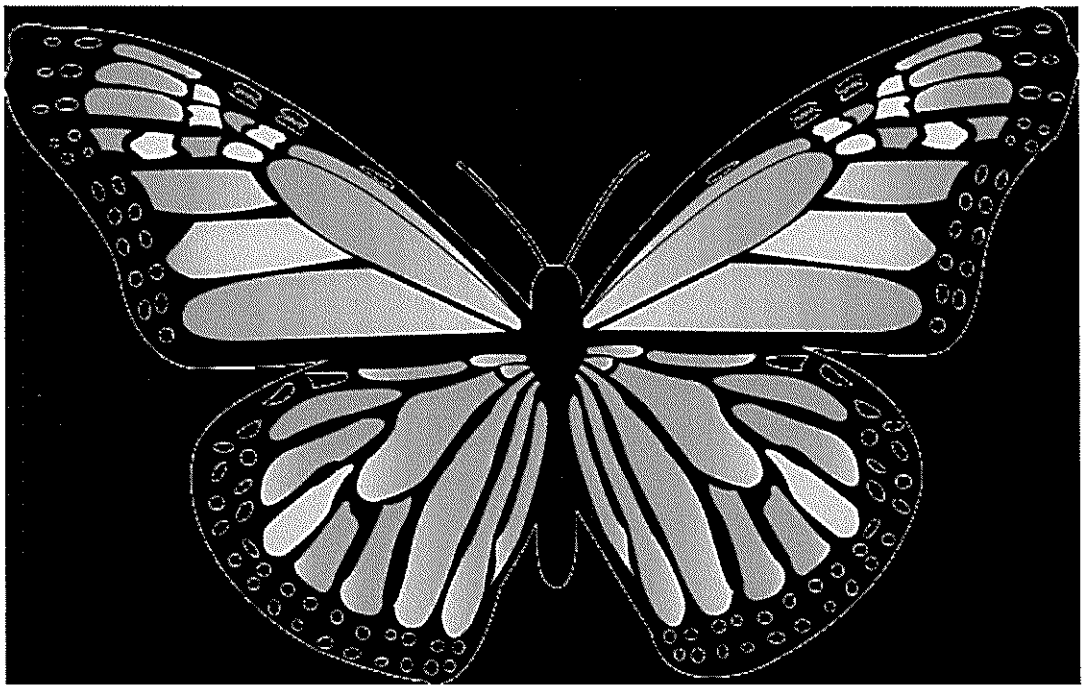
In[41]:=
```
{u, w, v} = SingularValueDecomposition[data];
```

In[42]:=
```
ByteCount[{u, w, v}]
```

Out[42]= 5 870 608

In[43]:=
```
Image[u.w.Conjugate[Transpose[v]]]
```

Out[43]=



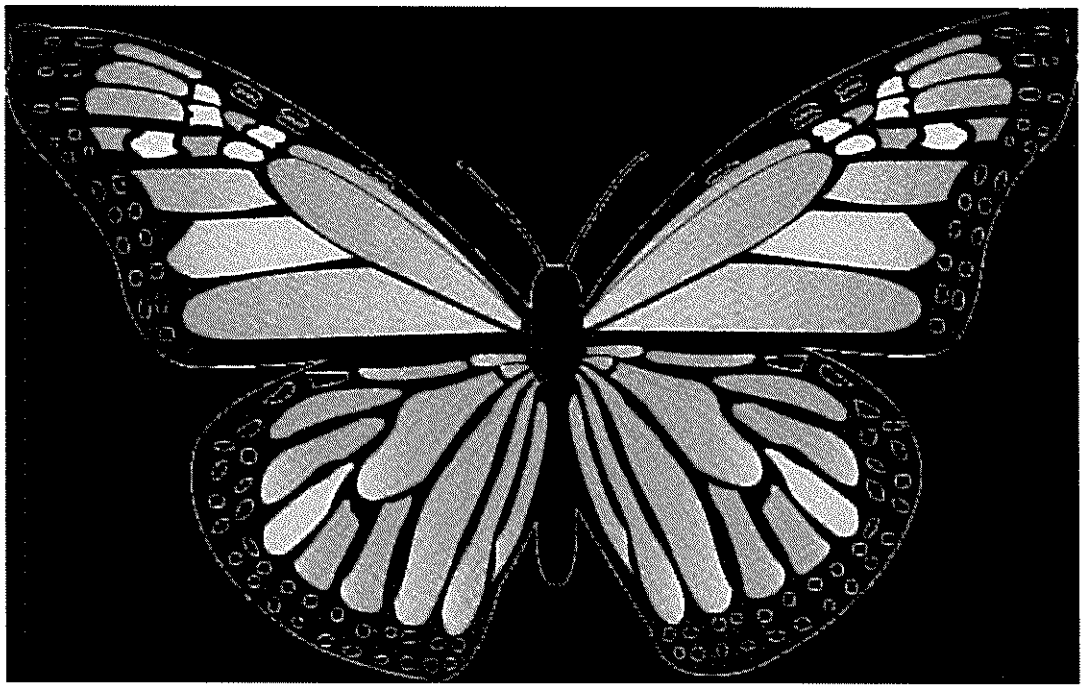
In[39]:=
```
{u, w, v} = SingularValueDecomposition[data, 100];
```

In[40]:=
```
ByteCount[{u, w, v}]
```

Out[40]= 865 320

In[30]:=
```
Image[u.w.Conjugate[Transpose[v]]]
```

Out[30]=



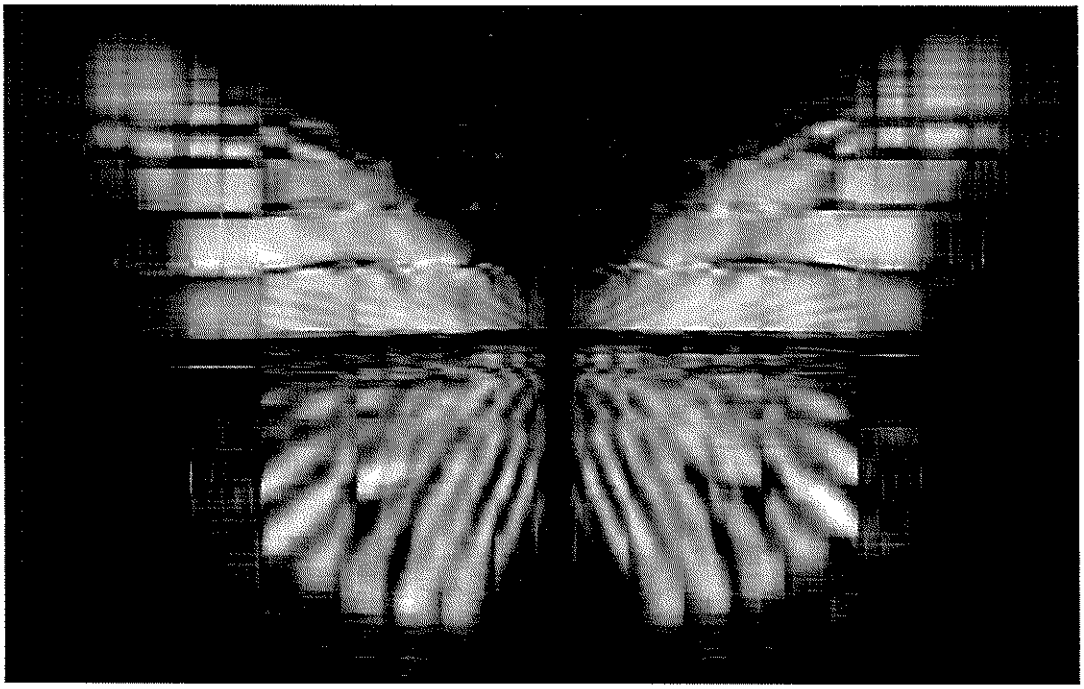
In[37]:=
```
{u, w, v} = SingularValueDecomposition[data, 10];
```

In[38]:=
```
ByteCount[{u, w, v}]
```

Out[38]= 81 120

In[32]:=
```
Image[u.w.Conjugate[Transpose[v]]]
```

Out[32]=



In[33]:=
```
{u, w, v} = SingularValueDecomposition[data, 1];
```

In[35]:=
```
ByteCount[{u, w, v}]
```

Out[35]=
```
8616
```

In[34]:=
```
Image[u.w.Conjugate[Transpose[v]]]
```

Out[34]=