Homework 6 solutions

9.1360. A simple population model. We consider a certain population of fish (say) each (yearly) season. $x(t) \in \mathbb{R}^3$ will describe the population of fish at year $t \in \mathbb{Z}$, as follows:

- $x_1(t)$ denotes the number of fish less than one year old
- $x_2(t)$ denotes the number of fish between one and two years old
- $x_3(t)$ denotes the number of fish between two and three years

(We will ignore the fact that these numbers are integers.) The population evolves from year t to year $t+1$ as follows.

- The number of fish less than one year old in the next year $(t+1)$ is equal to the total number of offspring born during the current year. Fish that are less than one year old in the current year (t) bear no offspring. Fish that are between one and two years old in the current year (t) bear an average of 2 offspring each. Fish that are between two and three years old in the current year (t) bear an average of 1 offspring each.
- 40% of the fish less than one year old in the current year (t) die; the remaining 60% live on to be between one and two years old in the next year $(t+1)$.
- 30% of the one-to-two year old fish in the current year die, and 70% live on to be two-to-three year old fish in the next year.
- All of the two-to-three year old fish in the current year die.

Express the population dynamics as an autonomous linear system with state $x(t)$, *i.e.*, in the form $x(t+1) = Ax(t)$. **Remark:** this example is silly, but more sophisticated population dynamics models are very useful and widely used.

Solution. We have that

$$\begin{aligned} x_1(t+1) &= 2x_2(t) + x_3(t), \\ x_2(t+1) &= 0.6x_1(t), \\ x_3(t+1) &= 0.7x_2(t). \end{aligned}$$

Thus, we have that $x(t+1) = Ax(t)$, where

$$A = \begin{bmatrix} 0.0 & 2.0 & 1.0 \\ 0.6 & 0.0 & 0.0 \\ 0.0 & 0.7 & 0.0 \end{bmatrix}.$$

9.1420. Iterative solution of linear equations. In many applications we need to solve a set of linear equations $Ax = b$, where A is nonsingular (square) and x is very large (*e.g.*, $x \in \mathbb{R}^{100000}$). We assume that Az can be computed at reasonable cost, for any z, but the standard methods for computing $x = A^{-1}b$ (*e.g.*, LU decomposition) are not feasible. A common approach is to use an *iterative* method, which computes a sequence $x(1), x(2)$, ... that *converges* to the

solution $x = A^{-1}b$. These methods rely on another matrix $\hat{A}$, which is supposed to be 'close' to A. More importantly, $\hat{A}$ has the property that $\hat{A}^{-1}z$ is easily or cheaply computed for any given z. As a simple example, the matrix $\hat{A}$ might be the diagonal part of the matrix A (which, presumably, has relatively small off-diagonal elements). Obviously computing $\hat{A}^{-1}z$ is fast; it's just scaling the entries of z. There are many, many other examples. A simple iterative method, sometimes called *relaxation*, is to set $\hat{x}(0)$ equal to some approximation of x (*e.g.*, $\hat{x}(0) = \hat{A}^{-1}b$) and repeat, for $t = 0, 1, \ldots$

$$r(t) = A\hat{x}(t) - b; \qquad \hat{x}(t+1) = \hat{x}(t) - \hat{A}^{-1}r(t);$$

(The hat reminds us that $\hat{x}(t)$ is an approximation, after t iterations, of the true solution $x = A^{-1}b$.) This iteration uses only 'cheap' calculations: multiplication by A and $\hat{A}^{-1}$. Note that $r(t)$ is the residual after the tth iteration.

a) Let $\beta = \|\hat{A}^{-1}(A - \hat{A})\|$ (which is a measure of how close $\hat{A}$ and A are). Show that if we choose $\hat{x}(0) = \hat{A}^{-1}b$, then $\|\hat{x}(t) - x\| \leq \beta^{t+1}\|x\|$. Thus if $\beta < 1$, the iterative method works, *i.e.*, for any b we have $\hat{x}(t) \to x$ as $t \to \infty$. (And if $\beta < 0.8$, say, then convergence is pretty fast.)

b) Find the exact conditions on A and $\hat{A}$ such that the method works for any starting approximation $\hat{x}(0)$ and any b. Your condition can involve norms, singular values, condition number, and eigenvalues of A and $\hat{A}$, or some combination, etc. Your condition should be as explicit as possible; for example, it should not include any limits. Try to avoid the following two errors:

- Your condition guarantees convergence but is too restrictive. (For example: $\beta = \|\hat{A}^{-1}(A - \hat{A})\| < 0.8$)
- Your condition doesn't guarantee convergence.

Solution. Substituting $r(t)$ into the expression for $\hat{x}(t+1)$ we get

$$\hat{x}(t+1) = \hat{x}(t) - \hat{A}^{-1}r(t) = (I - \hat{A}^{-1}A)\hat{x}(t) + \hat{A}^{-1}b.$$

Now we form the error $\tilde{x}(t) = \hat{x}(t) - x$, where $x = A^{-1}b$. We have

$$\tilde{x}(t+1) = (I - \hat{A}^{-1}A)\tilde{x}(t) = \hat{A}^{-1}(\hat{A} - A)\tilde{x}(t). \tag{1}$$

This shows that the error is the solution of an autonomous linear dynamical system. We can therefore answer part b immediately: In order for $\tilde{x}(t)$ to converge to zero regardless of the initial estimate, the linear dynamical system above must be stable, *i.e.*, the eigenvalues of $I - \hat{A}^{-1}A$ must have magnitude smaller than one. The eigenvalues of $I - \hat{A}^{-1}A$ have the form $1 - \lambda_i$, where λ_i is an eigenvalue of $\hat{A}^{-1}A$. We can express the condition as: $|1 - \lambda_i| < 1$. This means the eigenvalues λ_i of $\hat{A}^{-1}A$ lie in the open disk of radius one, centered at 1 in the complex plane. Now let's answer part a. Taking the norm of both sides of (1) we have

$$\|\tilde{x}(t+1)\| = \|\hat{A}^{-1}(\hat{A} - A)\tilde{x}(t)\| \leq \|\hat{A}^{-1}(\hat{A} - A)\|\|\tilde{x}(t)\| = \beta\|\tilde{x}(t)\|.$$

Hence we have $\|\tilde{x}(t)\| \leq \beta^t\|x - \hat{A}^{-1}b\|$. Now

$$x - \hat{A}^{-1}b = x - \hat{A}^{-1}Ax = \hat{A}^{-1}(\hat{A} - A)x$$

so $\|x - \hat{A}^{-1}b\| \leq \beta\|x\|$. All together we have the desired result:

$$\|\tilde{x}(t)\| \leq \beta^{(t+1)}\|x\|.$$

16.2560. Blind signal detection. A binary signal $s_1, \ldots, s_T$, with $s_t \in \{-1, 1\}$ is transmitted to a receiver, which receives the (vector) signal $y_t = as_t + v_t \in \mathbb{R}^n$, $t = 1, \ldots, T$, where $a \in \mathbb{R}^n$ and $v_t \in \mathbb{R}^n$ is a noise signal. We'll assume that $a \neq 0$, and that the noise signal is centered around zero, but is otherwise unknown. (This last statement is vague, but it will not matter.)

The receiver will form an approximation of the transmitted signal as

$$\hat{s}_t = w^\mathsf{T} y_t, \quad t = 1, \ldots, T,$$

where $w \in \mathbb{R}^n$ is a weight vector. Your job is to choose the weight vector w so that $\hat{s}_t \approx s_t$. If you knew the vector a, then a reasonable choice for w would be $w = a^\dagger = a/\|a\|^2$. This choice is the smallest (in norm) vector w for which $w^\mathsf{T} a = 1$.

Here's the catch: You don't know the vector a. Estimating the transmitted signal, given the received signal, when you don't know the mapping from transmitted to received signal (in this case, the vector a) is called *blind signal estimation* or *blind signal detection*.

Here is one approach. Ignoring the noise signal, and assuming that we have chosen w so that $w^\mathsf{T} y_t \approx s_t$, we would have

$$(1/T)\sum_{t=1}^{\mathsf{T}} (w^\mathsf{T} y_t)^2 \approx 1.$$

Since $w^\mathsf{T} v_t$ gives the noise contribution to $\hat{s}_t$, we want w to be as small as possible. This leads us to choose w to minimize $\|w\|$ subject to $(1/T)\sum_{t=1}^{\mathsf{T}} (w^\mathsf{T} y_t)^2 = 1$. This doesn't determine w uniquely; we can multiply it by -1 and it still minimizes $\|w\|$ subject to $(1/T)\sum_{t=1}^{\mathsf{T}} (w^\mathsf{T} y_t)^2 = 1$. So we can only hope to recover either an approximation of s_t or of $-s_t$; if we don't know a we really can't do any better. (In practice we'd use other methods to determine whether we have recovered s_t or $-s_t$.)

a) Explain how to find w, given the received vector signal $y_1, \ldots, y_T$, using concepts from the class.

b) Apply the method to the signal in the file `bs_det_data.json`, which contains a matrix `Y`, whose columns are y_t. Give the weight vector w that you find. Plot a histogram of the values of $w^\mathsf{T} y_t$ using `using Plots; histogram(w'*Y, bins=60)`. You'll know you're doing well if the result has two peaks, one negative and one positive. Once you've chosen w, a reasonable guess of s_t (or, possibly, its negative $-s_t$) is given by

$$\tilde{s}_t = \text{sign}(w^\mathsf{T} y_t), \quad t = 1, \ldots, T,$$

where $\text{sign}(u)$ is $+1$ for $u \geq 0$ and -1 for $u < 0$. The file `bs_det_data.json` contains the original signal, as a row vector `s`. Give your error rate, *i.e.*, the fraction of times for which $\tilde{s}_t \neq s_t$. (If this is more than 50%, you are welcome to flip the sign on w.)

Solution. We can write

$$\frac{1}{T}\sum_{t=1}^{\mathsf{T}}(w^{\mathsf{T}}y_t)^2 = (1/T)\|Y^{\mathsf{T}}w\|^2,$$

where $Y = [y_1 \cdots y_T]$. We must minimize $\|w\|$ subject to $\|Y^{\mathsf{T}}w\| = \sqrt{T}$. Both of these are homogeneous in w, so we could just as well maximize $\|Y^{\mathsf{T}}w\|$, subject to $\|w\| = 1$, and then scale the solution so that $\|Y^{\mathsf{T}}w\| = \sqrt{T}$. To maximize $\|Y^{\mathsf{T}}w\|$ subject to $\|w\| = 1$ is easy: we take w to be v_1, the right singular vector associated with the largest singular value of Y^{T}. (This is also, by the way, the left singular vector associated with the largest singular value of Y.) We then scale v_1 by α, so that $\|Y^{\mathsf{T}}(\alpha v_1)\| = \alpha\sigma_1 = \sqrt{T}$, where σ_1 is the largest singular value (*i.e.*, the norm) of Y (or Y^{T}). This yields

$$w = (\sqrt{T}/\sigma_1)v_1.$$

Note that we could just as well take the negative of this vector, which would also minimize $\|w\|$ subject to $\|Y^{\mathsf{T}}w\| = \sqrt{T}$.

Here is the code to do this:

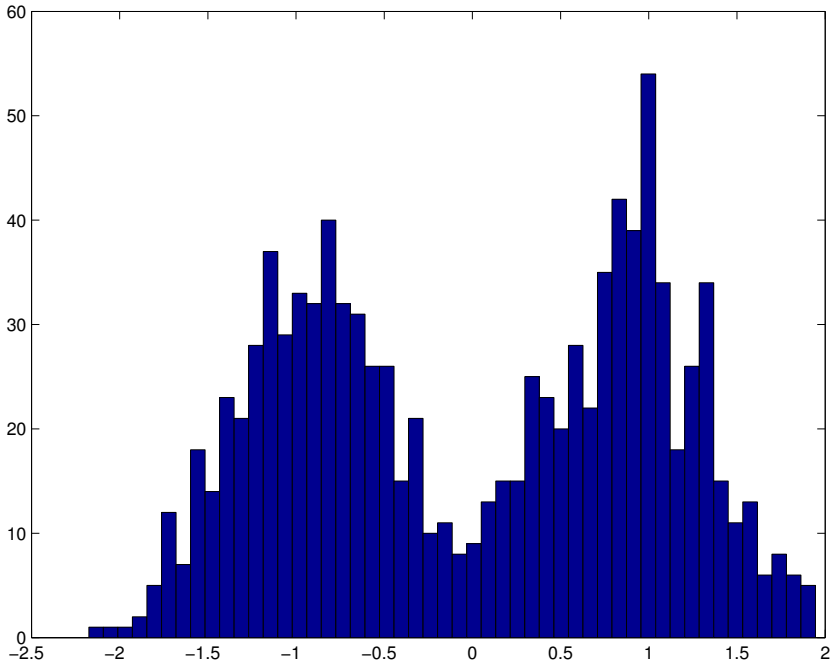
```
% blind signal detection exercise
% solution

bs_det_data;
[U,S,V]=svd(Y');
w = (sqrt(T)/S(1,1))*V(:,1)

% now form estimate of original binary signal
shat = (Y'*w)';
hist(shat,50);
print -deps bs_det_hist

% error rate
stilde = sign(shat);
error_rate = sum(s~=stilde)/T
```

The error rate is 2.9%. The resulting histogram is shown below.



16.2670. Regularization and SVD. Let $A \in \mathbb{R}^{n\times n}$ be full rank, with SVD

$$A = \sum_{i=1}^{n} \sigma_i u_i v_i^{\mathsf{T}}.$$

(We consider the square, full rank case just for simplicity; it's not too hard to consider the general nonsquare, non-full rank case.) Recall that the regularized approximate solution of $Ax = y$ is defined as the vector $x_{\text{reg}} \in \mathbb{R}^n$ that minimizes the function

$$\|Ax - y\|^2 + \mu\|x\|^2,$$

where $\mu > 0$ is the regularization parameter. The regularized solution is a linear function of y, so it can be expressed as $x_{\text{reg}} = By$ where $B \in \mathbb{R}^{n\times n}$.

a) Express the SVD of B in terms of the SVD of A. To be more specific, let

$$B = \sum_{i=1}^{n} \tilde{\sigma}_i \tilde{u}_i \tilde{v}_i^{\mathsf{T}}$$

denote the SVD of B. Express $\tilde{\sigma}_i$, $\tilde{u}_i$, $\tilde{v}_i$, for $i = 1, \ldots, n$, in terms of σ_i, u_i, v_i, $i = 1, \ldots, n$ (and, possibly, μ). Recall the convention that $\tilde{\sigma}_1 \geq \cdots \geq \tilde{\sigma}_n$.

b) Find the norm of B. Give your answer in terms of the SVD of A (and μ).

c) Find the worst-case relative inversion error, defined as

$$\max_{y\neq 0} \frac{\|ABy - y\|}{\|y\|}.$$

Give your answer in terms of the SVD of A (and μ).

Solution.

a) The regularized least-squares solution is given by $x_{\mathrm{rls}(\mu)} = \left(A^\mathsf{T}A + \mu I\right)^{-1} A^\mathsf{T}y$, and thus

$$\begin{aligned}
B &= \left(A^\mathsf{T}A + \mu I\right)^{-1} A^\mathsf{T} \\
&= \left(\left(U\Sigma V^\mathsf{T}\right)^\mathsf{T} U\Sigma V^\mathsf{T} + \mu I\right)^{-1} \left(U\Sigma V^\mathsf{T}\right)^\mathsf{T} \\
&= \left(V\Sigma U^\mathsf{T} U\Sigma V^\mathsf{T} + \mu I\right)^{-1} V\Sigma U^\mathsf{T} \\
&= \left(V\left(\Sigma^2 + \mu I\right)V^\mathsf{T}\right)^{-1} V\Sigma U^\mathsf{T} \\
&= \left(V\left(\Sigma^2 + \mu I\right)^{-1} V^\mathsf{T}\right) V\Sigma U^\mathsf{T} \\
&= V\left(\Sigma^2 + \mu I\right)^{-1}\Sigma U^\mathsf{T} \\
&= V\,\mathrm{diag}\left(\frac{\sigma_i}{\sigma_i^2 + \mu}\right) U^\mathsf{T}.
\end{aligned}$$

This is almost the SVD of B, except for one detail: the numbers

$$\frac{\sigma_i}{\sigma_i^2 + \mu}$$

aren't necessarily ordered from largest to smallest. Thus we have

$$\tilde{\sigma}_i = \frac{\sigma_{[i]}}{\sigma_{[i]}^2 + \mu} \quad \tilde{u}_i = v_{[i]}, \qquad \tilde{v}_i = u_{[i]},$$

where the notation $x_{[i]}$ means the ith largest element of x. (We accepted all sorts of descriptions of this!) One common misconception was that the numbers

$$\frac{\sigma_i}{\sigma_i^2 + \mu}$$

were simply in reverse order, so all that had to be done was to reverse the ordering. That isn't true; just sketch the function $\sigma/(\sigma^2 + \mu)$ as a function of μ to see that it is not always decreasing. (It increases first, then decreases.)

b) The norm of B is its largest singular value, *i.e.*,

$$\|B\| = \max_i \frac{\sigma_i}{\sigma_i^2 + \mu}.$$

c) The worst-case relative inversion error is the matrix norm of $AB - I$:

$$\begin{aligned}
AB - I &= U\Sigma V^\mathsf{T} V\left(\Sigma + \mu\Sigma^{-1}\right)^{-1} U^\mathsf{T} - I \\
&= U\Sigma\left(\Sigma + \mu\Sigma^{-1}\right)^{-1} U^\mathsf{T} - I \\
&= U\left(\left(I + \mu\Sigma^{-2}\right)^{-1} - I\right) U^\mathsf{T} \\
&= U\,\mathrm{diag}\left(\frac{1}{1 + \mu/\sigma_i^2} - 1\right) U^\mathsf{T} \\
&= -U\,\mathrm{diag}\left(\frac{\mu}{\sigma_i^2 + \mu}\right) U^\mathsf{T}
\end{aligned}$$

This is the SVD of $AB - I$ (to within reordering). Its largest singular value, *i.e.*, the norm of $AB - I$, is given by

$$\|AB - I\| = \frac{\mu}{\sigma_n^2 + \mu}.$$

Note that we had to absorb the negative sign in the lefthand orthogonal matrix; one common error was to keep the negative sign in the norm. Obviously that couldn't be right because norms are always nonnegative!

16.2710. The EE263 search engine. In this problem we examine how linear algebra and low-rank approximations can be used to find matches to a search query in a set of documents. Let's assume we have four documents: **A**, **B**, **C**, and **D**. We want to search these documents for three terms: *piano*, *violin*, and *drum*. We know that:
in **A**, the word *piano* appears 4 times, *violin* 3 times, and *drum* 1 time;
in **B**, the word *piano* appears 6 times, *violin* 1 time, and *drum* 0 times;
in **C**, the word *piano* appears 7 time, *violin* 4 times, and *drum* 39 times; and
in **D**, the word *piano* appears 0 times, *violin* 0 times, and *drum* 5 times.
We can tabulate this as follows:

	A	**B**	**C**	**D**
piano	4	6	7	0
violin	3	1	4	0
drum	1	0	39	5

This information is used to form a *term-by-document* matrix A, where A_{ij} specifies the frequency of the ith term in the jth document, *i.e.*,

$$A = \begin{bmatrix} 4 & 6 & 7 & 0 \\ 3 & 1 & 4 & 0 \\ 1 & 0 & 39 & 5 \end{bmatrix}.$$

Now let q be a *query vector*, with a non-zero entry for each term. The query vector expresses a criterion by which to select a document. Typically, q will have 1 in the entries corresponding to the words we want to search for, and 0 in all other entries (but other weighting schemes are possible.) A simple measure of how relevant document j is to the query is given by the inner product of the jth column of A with q:

$$a_j^T q.$$

However, this criterion is biased towards large documents. For instance, a query for *piano* ($q = [1\ 0\ 0]^T$) by this criterion would return document **C** as most relevant, even though document **B** (and even **A**) is probably much more relevant. For this reason, we use the inner product normalized by the norm of the vectors,

$$\frac{a_j^T q}{\|a_j\|\,\|q\|}.$$

Note that our criterion for measuring how well a document matches the query is now the cosine of the angle between the document and query vectors. Since all entries are non-negative, the

cosine is in $[0, 1]$ (and the angle is in $[-\pi/2, \pi/2]$.) Define $\tilde{A}$ and $\tilde{q}$ as normalized versions of A and q (A is normalized column-wise, *i.e.*, each column is divided by its norm.) Then,

$$c = \tilde{A}^\mathsf{T} \tilde{q}$$

is a column vector that gives a measure of the relevance of each document to the query. And now, the question. In the file `term_by_doc.json` you are given m search terms, n documents, and the corresponding term-by-document matrix $A \in \mathbb{R}^{m \times n}$. (They were obtained randomly from Stanford's *Portfolio* collection of internal documents from the 1990s.) The variables `term` and `document` are lists of strings. The string `term[i]` contains the ith word. Each document is specified by its former URL, *i.e.*, the jth document used to be at the URL `document[j]`; the documents are no longer available online, but you might be able to find some of them on some internet archive (like the Wayback Machine) if you're curious. (You don't need to in order to solve the problem.) The matrix entry `A[i,j]` specifies how many times term i appears in document j.

When you specify documents in your results, please just specify the indices, that is, give us `j` rather than `document[j]`.

a) Compute $\tilde{A}$, the normalized term-by-document matrix. Compute and plot the singular values of $\tilde{A}$.

b) Perform a query for the word *students* ($i = 53$) on $\tilde{A}$. What are the 5 top results?

c) We will now consider low-rank approximations of $\tilde{A}$, that is

$$\hat{A}_r = \min_{\hat{A},\, \mathbf{rank}(\hat{A}) \le r} \|\tilde{A} - \hat{A}\|.$$

Compute $\hat{A}_{32}$, $\hat{A}_{16}$, $\hat{A}_8$, and $\hat{A}_4$. Perform a query for the word *students* on these matrices. Comment on the results.

d) Are there advantages of using low-rank approximations over using the full-rank matrix? (You can assume that a very large number of searches will be performed before the term-by-document matrix is updated.)

Note: Variations and extensions of this idea are actually used in commercial search engines (although the details are closely guarded secrets . . .) Issues in real search engines include the fact that m and n are enormous and change with time. These methods are very interesting because they can recover documents that don't include the term searched for. For example, a search for *automobile* could retrieve a document with no mention of *automobile*, but many references to cars (can you give a justification for this?) For this reason, this approach is sometimes called *latent semantic indexing*.

Julia hints: You may find the command `sortperm` useful. It returns the index permutation that would sort its argument into an ascending order, or if the `rev=true` argument is supplied, into a descending order. Here's some sample code that prints the indices of the two largest elements of a vector `c`:

```
p = sortperm(c, rev=true)
@show p[1:2]                # indices of two largest elements
c_sorted = c[p]             # equivalent of sort(c, rev=true)
@show c[p[1:2]]             # two largest elements of c
```

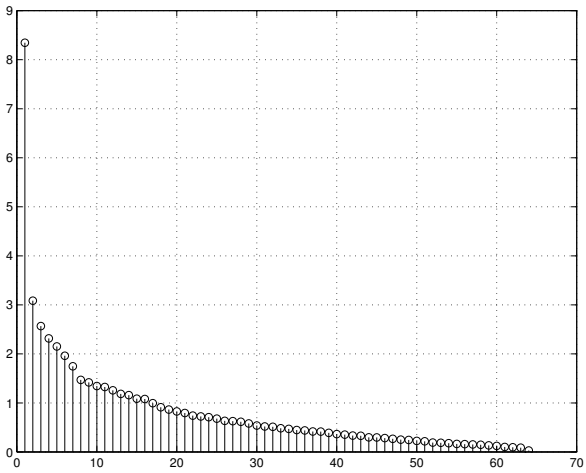


Figure 1: Singular values of $\tilde{A}$

Solution. The singular values of the normalized matrix are shown in the figure. The full-rank search yields the documents:

```
[106, 105, 107, 115, 111]
```

The corresponding weights are $c = [\,0.60 \quad 0.55 \quad 0.50 \quad 0.49 \quad 0.41\,]^\mathsf{T}$. Given the SVD of $\tilde{A} = U\Sigma V^\mathsf{T}$, the nearest low-rank approximation is easily computed:

$$\hat{A}_r = \min_{\hat{A},\,\mathbf{rank}(\hat{A})\leq r} \|\tilde{A} - \hat{A}\| = \sum_{k=1}^{r} \sigma_k u_k v_k^\mathsf{T} = U_1 W_1^\mathsf{T},$$

where σ_k are the singular values, u_k are the columns of U, v_k are the columns of V. We have also written the low rank approximation as the product of two smaller full-rank matrices, $U_1 = [\,u_1 \;\cdots\; u_r\,]$, and $W_1 = [\,\sigma_1 v_1 \;\cdots\; \sigma_r v_r\,]$. The search results are as follows (the table includes the document numbers only, *i.e.*, the last part of the URL):

	rank 32	rank 16	rank 8	rank 4
1st	106	106	115	115
2nd	105	107	106	105
3rd	107	105	120	107
4th	115	115	107	66
5th	111	111	111	63

And the corresponding weights are:

	rank 32	rank 16	rank 8	rank 4
1st	0.6445	0.5474	0.3751	0.2212
2nd	0.4952	0.4519	0.3666	0.1961
3rd	0.4726	0.4312	0.3462	0.1912
4th	0.4299	0.3910	0.3322	0.1894
5th	0.3846	0.3745	0.2998	0.1824

Note the "loss of resolution" for the lower rank searches, as the weights become less differentiated. Down to rank 16 the low-rank search is essentially equivalent to the full-rank search. For rank 8, the top 5 results are still very similar, although their ordering is somewhat changed. For rank 4 the results start differing significantly, although there are still some top 5 documents in common with the full-rank search. However, if you look at the content of the documents (something that almost no-one seems to have done), you'll see that the ones returned by the low-rank searches appear to be more relevant! Of course, this is a subjective appreciation... But we'll get back to this in a while, when discussing the advantages of the low-rank search. The matlab code for this problem is as follows:

```
term_by_doc  % run the script that defines matrix A
t=53;  % index of term(s) to search for (t=[53 64] also works)
q=zeros(m,1);  % query vector
q(t)=1;  % put 1 in the entries indexed by the vector term
disp(' ') disp('query:')
for k=1:length(t), disp(term{t(k)}), end  % print words in query
for j=1:n, An(:,j)=A(:,j)/norm(A(:,j)); end  % normalize columns
q=q/norm(q);  % normalize query
c0=An'*q;
[w0,i0]=sort(c0);  % sort in ascending order
w0=flipud(w0);  % flip to get descending order
i0=flipud(i0); disp(' ');
disp('full-rank top 5:')  % print top 5 docs
for k=1:5, disp(document{i0(k)}), end
disp(w0(1:5)')  % print weights of the top 5 docs
[U,S,V]=svd(A);
stem(diag(S)) % plot of singular values
grid on for p=1:4
r=2^(6-p);   % r=2.^(5:-1:2)=[32 16 8 4]
U1=U(:,1:r);
S1=S(1:r,1:r);
V1=V(:,1:r);
c1=(V1*S1)*(U1'*q);
[w1,i1]=sort(c1);
w1=flipud(w1);
i1=flipud(i1);
disp(' ')
disp('low-rank top 5:')  % print top 5 docs
for k=1:5, disp(document{i1(k)}) end
disp(w1(1:5)')  % print weights of the top 5 docs
end
```

Now, for the advantages of low-rank search. The most obvious one is query speed, which can be achieved by computing the query using the factored form of $\hat{A}_r$:

$$c = W_1(U_1^\mathsf{T} q)$$

(with U_1 and W_1 as defined above.) The full-rank query (and the low-rank without using the factored form) requires about nm operations. The factored low-rank query requires about $r(m+n)$ operations. Of course, computing the SVD for a large matrix requires a large numbers of operations, so the number of searches performed between updates of the matrix must be large for this to pay off. (Note that there are techiques for doing an approximate update of the low-rank approximation when a new document is added without recomputing the whole SVD.) Also, note that the SVD can be done off-line. Even if it's overall more expensive (*i.e.*, requires more operations) to do the SVD plus the low-rank searches than to do all searches on the full-rank matrix, the user will be happy to have faster searches. Meanwhile another computer can be used to work in parallel on the next updating of the low-rank approximation. The only part of this problem that most people didn't get was the "latent semantic indexing" property, which can be interpreted as a form of "de-noising." More than query speed, this is actually the driving motivation for the use of low rank approximations in database searches. The best way to explain this is by example, so here's a good one (from a student's exam solution!) Consider the following term-by-document matrix:

	Doc 1	Doc 2	Doc 3	Doc 4
car	30	15	1	2
automobile	0	15	0	1
penguin	0	0	20	1
coffee	0	0	3	14

The SVD of the normalized $\tilde{A}$ yields the singular values:

$$\sigma = [\,1.33\ 1.08\ 0.88\ 0.54\,].$$

Note that the fourth is smaller, but not by that much. Also from the SVD, we get:

$$U = \begin{bmatrix} 0.89 & -0.23 & 0.09 & -0.38 \\ 0.37 & -0.11 & -0.01 & 0.92 \\ 0.13 & 0.71 & 0.69 & 0.03 \\ 0.23 & 0.66 & -0.72 & -0.02 \end{bmatrix}.$$

We can give a "semantic" interpretation of each dyad. The first vector weights the terms *car* and *automobile*. The second vector weights the terms *penguin* and *coffee*. The third vector puts differential weights on *penguin* and *coffee*. The fourth vector puts differential weights on *car* and *automobile*. If we remove the fourth dyad, the result is that the differentiation between *car* and *automobile* is removed! Here's the resulting low-rank approximation of $\tilde{A}$:

$$\hat{A}_3 = \begin{bmatrix} 0.85 & 0.85 & 0.05 & 0.14 \\ 0.35 & 0.35 & -0.01 & 0.08 \\ 0.01 & -0.01 & 0.99 & 0.07 \\ -0.01 & 0.01 & 0.15 & 0.98 \end{bmatrix}.$$

We see that *car* and *automobile* are now weighted equally in documents 1 and 2. A search for *automobile* (*i.e.*, $q = [\,0\ 1\ 0\ 0\,]^\mathsf{T}$) on the original $\tilde{A}$ yields zero relevance for the first document:

$$c = \tilde{A}^\mathsf{T} q = [\,0.00\ \ 0.71\ \ 0.00\ \ 0.07\,]^\mathsf{T}.$$

The same search on the low-rank approximation $\hat{A}_3$ produces the much more sensible result:

$$c = \hat{A}_3^{\mathsf{T}} q = [\,0.35 \;\; 0.35 \;\; -0.01 \;\; 0.08\,]^{\mathsf{T}}.$$

Documents 1 and 2 are now considered to have equal relevance for the word *automobile*! The conclusion is that even though the search results of a low-rank approximation may be "less accurate" (in the sense that they differ from those of a search with the original matrix), they may actually be better from the user's point of view.