GTx CSE6040x

FA20: Computing for Data Analysis

Help mrajagopal6

Course Progress Dates Discussion Wiki

Course / Module 2: The Analysis of Data / Solution: Notebook 15

Previous Next

Sample solutions

Bookmark this page

Compression via the PCA and the SVD

The main topic of this lesson is a data analysis method referred to as *Principal Components Analysis* (PCA). The method requires computing th of a certain matrix; one way to compute those eigenvectors is to use a special factorization from linear algebra called the *Singular Value Decom* (SVD).

This notebook is simply a collection of notes with a little bit of code to help illustrate the main ideas. It does not have any exercises that you ne However, you should try to understand all the code steps that appear in the subsection entitled, **Principal Components Analysis (PCA)**, as yo apply the SVD in a subsequent part of this assignment.

Motivation: data "compression." In previous lessons, we've looked at a few of the major tasks in data analysis: *ranking*, *regression*, *classificat* *clustering*. Beyond these, the last problem you'll consider in our class is what we'll call *compression*.

At a high level, the term compression simply refers to finding any compact representation of the data. Such representations can help us in two can make the data set smaller and therefore faster to process or analyze. Secondly, choosing a clever representation can reveal hidden structu

As a concrete example, consider the problem of *dimensionality reduction*: given a d-dimensional data set, we wish to transform it into a smaller dimensional data set where $k \leq d$.

Choosing the k dimensions in a clever way might even reveal structure that is hard to see in all d original dimensions. For instance, look at the the "visualizing PCA" website:

http://setosa.io/ev/principal-component-analysis/ (http://setosa.io/ev/principal-component-analysis/)

Data: Nutrition in the UK

Here is one of those examples, which is nutritional data gathered in a study of four countries of the United Kingdom. (Researchers tabulated th number of grams consumed per week by an individual living in a particular country, broken down along various food and drink categories.)

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import display

%matplotlib inline
```

In [2]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(loca
                                                                                          body_
                                                                                          checl
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/uk-food/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/uk-food/"
    DATA_PATH = ""

datasets = {'uk-nutrition-data.csv': 'a6cdc2fb658bacfdf50797c625aa3815'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'uk-nutrition-data.csv' is ready!

(All data appears to be ready.)
```

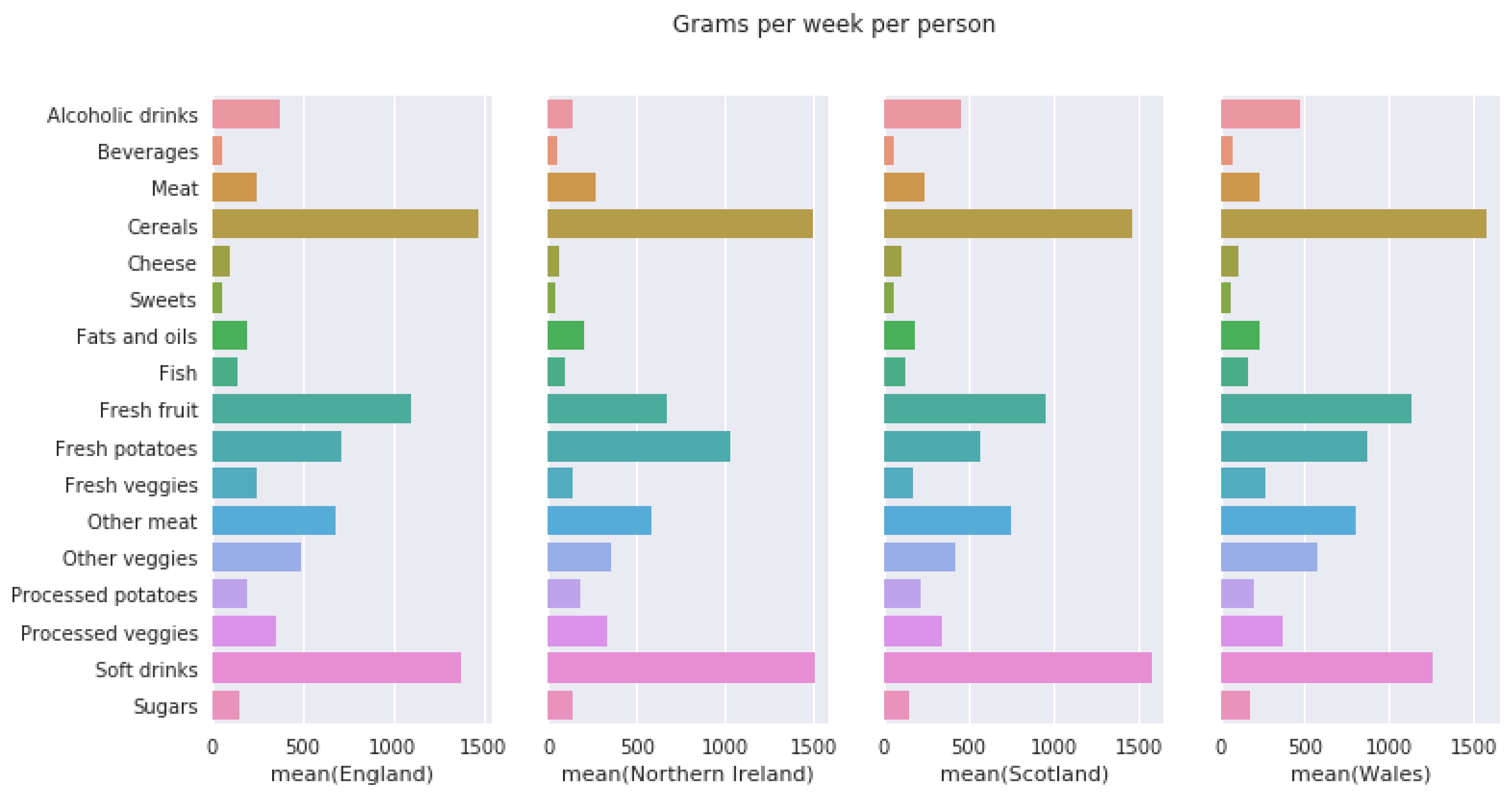
In [3]:
```
df_uk = pd.read_csv('{}uk-nutrition-data.csv'.format(DATA_PATH))
print("{} x {} table of data:".format(df_uk.shape[0], df_uk.shape[1]))
display(df_uk.head ())
print("...")

fig, axes = plt.subplots(1, 4, figsize=(12, 6), sharey=True)
countries = df_uk.columns.difference(['Product'])
for i in range(len(countries)):
    sns.barplot(x=countries[i], y='Product', data=df_uk, ax=axes[i])
    axes[i].set_ylabel("")
fig.suptitle("Grams per week per person")
```

```
17 x 5 table of data:
```

	Product	England	Northern Ireland	Scotland	Wales
0	Alcoholic drinks	375	135	458	475
1	Beverages	57	47	53	73
2	Meat	245	267	242	227
3	Cereals	1472	1494	1462	1582
4	Cheese	105	66	103	103

```
...
```

Out[3]: `Text(0.5,0.98,'Grams per week per person')`



Do the countries differ in any significant way? Looking only at the bar charts, it is probably hard to tell unless you are very perceptive, and in an inspection is a very *ad hoc* technique. Is there a more systematic way?

Principal components analysis (PCA)

The method of *principal components analysis* (PCA) is one such technique. For this example, it would start by viewing these data as four (4) da for each country, embedded in a 17-dimensional space (one dimension per food category). The following page will help build your intuition for I that then follow below show formally how PCA works and derives an algorithm to compute it.

[http://setosa.io/ev/principal-component-analysis/ (http://setosa.io/ev/principal-component-analysis/)](http://setosa.io/ev/principal-component-analysis/)

Basic definitions

Input data matrix, centered. Per our usual conventions, let $\hat{x}_0, \ldots, \hat{x}_{m-1}$ be th m data points, where each $x_i \in \mathbb{R}^d$ is a single observation. Ea is represented by a d-dimensional real-valued vector corresponding to d measured predictors. As usual, we can stack these into a data matrix,

$$X \equiv \left(\begin{array}{c} \hat{x}_0^T \\ \vdots \\ \hat{x}_{m-1}^T \end{array}\right).$$

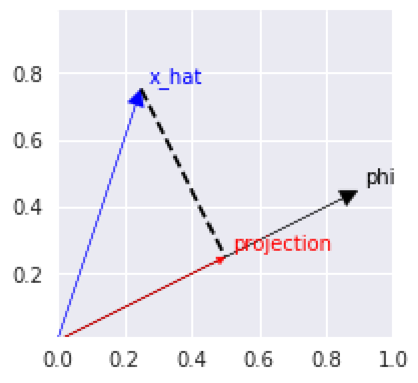
However, we'll add one more important assumption: these data should be *centered* about their mean, i.e., $\frac{1}{m}\sum_{i=0}^{m-1}\hat{x}_i = 0$. If the observations centered initially, then preprocess them accordingly.

Projections. Let $\varphi \in \mathbb{R}^d$ be a vector of unit length, i.e., $\|\varphi\|_2^2 = \varphi^T\varphi = 1$. The *projection* of a data point $\hat{x}_i$ onto φ is $\hat{x}_i^T\varphi$, which measures th the projected vector.

The following code cell illustrates a projection. Given a vector `x_hat` and a line represented by a unit vector `phi`, it computes the projection `x_hat_proj_phi` of `x_hat` onto `phi`.

```
In [4]: # Define a projection
        x_hat = np.array([0.25, 0.75]) # Vector to project
        phi = np.array([0.5, 0.25]) ; phi = phi / np.linalg.norm(phi) # Unit vector onto which to proje
        x_hat_proj_phi = x_hat.T.dot(phi) * phi # Carry out the projection
```

```
In [5]: # Visualize the projection (you don't need to understand this code cell in any detail)
        import matplotlib.lines as mlines

        plt.figure(figsize=(3, 3))
        ax = plt.axes()
        ax.arrow(0, 0, x_hat[0], x_hat[1], head_width=0.05, head_length=0.05, fc='b', ec='b', length_inc
        =True)
        ax.arrow(0, 0, phi[0], phi[1], head_width=0.05, head_length=0.05, fc='k', ec='k', length_include
        e)
        ax.arrow(0, 0, x_hat_proj_phi[0], x_hat_proj_phi[1], head_width=0.025, head_length=0.025, fc='r
        length_includes_head=True)

        perp_line = mlines.Line2D([x_hat[0], x_hat_proj_phi[0]],
                                  [x_hat[1], x_hat_proj_phi[1]],
                                  linestyle='--', color='k')
        ax.add_line(perp_line)
        ax.axis('equal') # Equal ratios, so you can tell what is perpendicular to what
        ax.axis([0, 1, 0, 1])

        dx, dy = 0.02, 0.02
        plt.annotate('x_hat', xy=(x_hat[0]+dx, x_hat[1]+dy), color='b')
        plt.annotate('phi', xy=(phi[0]+dx, phi[1]+dy), color='k')
        plt.annotate('projection', xy=(x_hat_proj_phi[0]+dx, x_hat_proj_phi[1]+dy), color='r')

        plt.show()

        msg = """* Black arrow: `phi` (len={:.3f})
        * Blue arrow: `x_hat` (len={:.3f})
        * Red arrow: projection of `x_hat` onto `phi` (len={:.3f})"""
        print(msg.format(np.linalg.norm(phi),
                         np.linalg.norm(x_hat),
                         np.linalg.norm(x_hat_proj_phi)))
```



```
* Black arrow: `phi` (len=1.000)
* Blue arrow: `x_hat` (len=0.791)
* Red arrow: projection of `x_hat` onto `phi` (len=0.559)
```

Maximizing projections

If the length of a projected data point is large, then intuitively, we have "preserved" its shape. So let's think of the total length of projections of a points as a measure of cost, which we can then try to maximmize.

Projection cost. Let $J(\varphi)$ be a cost function that is proportional to the mean squared projections of the data onto φ:

$$J(\phi) \equiv \frac{1}{2m}\sum_{i=0}^{m-1}(\hat{x}_i^T\varphi)^2.$$

The additional factor of "1/2" is for aesthetic reasons. (It cancels out later on.)

Let's also apply some algebra-fu to the right-hand side to put it into a more concise matrix form:

$$\begin{aligned} J(\phi) &= \frac{1}{2}\varphi^T \left(\frac{1}{m}\sum_{i=0}^{m-1} \hat{x}_i \hat{x}_i^T\right)\varphi \\ &= \frac{1}{2}\varphi^T \left(\frac{1}{m}X^T X\right)\varphi \\ &\equiv \frac{1}{2}\varphi^T C\varphi. \end{aligned}$$

In the last step, we defined $C \equiv \frac{1}{m}X^T X$. In statistics, if X represents mean-centered data, then the matrix C is also known as the *sample cova* (https://en.wikipedia.org/wiki/Sample_mean_and_covariance) of the data.

Principal components via maximizing projections. There are several ways to formulate the PCA problem. Here we consider the one based o *projections*.

Start by defining a *principal component* of the data X to be a vector, φ, of unit length that maximizes the sum of squared projections.

To convert this definition into a formal problem, there is a technique known as the *method of Langrange multipliers*, which may be applied to ar or maximization problem that has equality constraints. The idea is to modify the cost function in a certain way that effectively incorporates eacl for each constraint you will add to the cost function a term proportional to a dummy parameter times some form of the constraint.

Huh? It's easiest to see this formulation by example. In the case of a principal component, the modified cost function is

$$\hat{J}(\varphi, \lambda) \equiv J(\varphi) + \frac{\lambda}{2}(1 - \varphi^T \varphi),$$

where the second term captures the constraint: it introduces a dummy optimization parameter, λ, times the constraint that φ has unit length, i.e $\|\varphi\|_2^2 = \varphi^T \varphi = 1$, or $1 - \varphi^T \varphi = 0$.

> The reason to add the constraint in this way should become clear momentarily.
>
> As before, the factor of "1/2" is there solely for aesthetic reasons and will "cancel out," as you'll soon see.

The optimization task is to find the φ_* and λ_* that maximize $\hat{J}$:

$$(\varphi_*, \lambda_*) \equiv \arg\max_{\varphi,\lambda} \hat{J}(\varphi, \lambda).$$

To solve this optimization problem, you just need to "take derivatives" of $\hat{J}$ with respect to φ and λ, and then set these derivatives to 0.

Exercise (optional). Show that

$$\begin{aligned} \nabla_\varphi \hat{J} &= C\varphi - \lambda\varphi \\ \frac{\partial}{\partial\lambda}\hat{J} &= \tfrac{1}{2}(1 - \varphi^T \varphi). \end{aligned}$$

Setting these to zero and solving yields the following computational problem:

$$\begin{aligned} C\varphi = \tfrac{1}{m}X^T X\varphi &= \lambda\varphi \\ \|\varphi\|_2^2 &= 1. \end{aligned}$$

> Is it now clear why the constraint was incorporated into $\hat{J}$ as it was? Doing so produces a second equation that *exactly* captures the constraint!

This problem is an *eigenproblem*, which is the task of computing an eigenvalue and its corresponding eigenvector of $C = \frac{1}{m}X^T X$.

The matrix C will usually have many eigenvalues and eigenvectors. So which one do you want? Plug the eigenvector back into the original cos Then, $J(\varphi) = \frac{1}{2}\varphi^T C\varphi = \frac{\lambda}{2}\varphi^T \varphi = \frac{\lambda}{2}$. In other words, to maximize $J(\varphi)$ you should pick the φ with the largest eigenvalue λ.

Finding an eigenpair via the SVD

So how do you find the eigenvectors of C? That is, what algorithm will compute them?

One way is to form C explicitly and then call an off-the-shelf eigensolver. However, forming C explicitly from the data X may be costly in time a not to mention possibly less accurate. (Recall the condition number blow-up problem in the case of solving the normal equations.)

Instead, we can turn to the "Swiss Army knife" of linear algebra, which is the *singular value decomposition*, or SVD. It is an extremely versatile t simplifying linear algebra problems. It can also be somewhat expensive to compute accurately, but a lot of scientific and engineering effort has building robust and reasonably efficient SVD algorithms. So let's assume these exist -- and they do in both Numpy (http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html) and Scipy (http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.svds.html) -- and use them accordingly.

The SVD. Every real-valued matrix $X \in \mathbb{R}^{m \times d}$ has a *singular value decomposition*. Let $s = \min(m, d)$, i.e., the smaller of the number of rows c

Then the SVD of X is the factorization, $X = U\Sigma V^T$, where U, Σ, and V^T are defined as follows.

The matrices $U \in \mathbb{R}^{m\times s}$ and $V \in \mathbb{R}^{d\times s}$ are orthogonal matrices, meaning $U^TU = I$ and $V^TV = I$; and the matrix Σ is an $s \times s$ diagonal ma

> Note that V is taken to be $d \times s$, so that the V^T that appears in $U\Sigma V^T$ is $s \times d$.)

The columns of U are also known as the *left singular vectors*, and the columns of V are the *right singular vectors* of X. Using our usual "colum matrix, these vectors are denoted by u_i and v_i:

$$\begin{aligned} U &= \begin{bmatrix} u_0 & u_1 & \cdots & u_{s-1} \end{bmatrix} \\ V &= \begin{bmatrix} v_0 & v_1 & \cdots & v_{s-1} \end{bmatrix} \end{aligned}$$

Regarding the diagonal matrix Σ, its entries are, collectively, called the *singular values* of X:

$$\begin{bmatrix} \sigma_0 & & & \\ & \sigma_1 & & \\ & & \ddots & \\ & & & \sigma_{s-1} \end{bmatrix}.$$

From these definitions, the SVD implies that $XV = U\Sigma$. This form is just a compact way of writing down a *system* of independent vector equa

$$Xv_i = \sigma_i u_i.$$

Recall that in PCA, you want to evaluate $C = \frac{1}{m}X^TX$. In terms of the SVD,

$$X^TX = V\Sigma^TU^TU\Sigma V^T = V\Sigma^2V^T,$$

or

$$X^TXV = V\Sigma^2.$$

This relation may in turn be rewritten as the system of vector equations,

$$X^TXv_i = \sigma_i^2 v_i.$$

In other words, every pair $(\varphi, \lambda) \equiv \left(v_i, \frac{\sigma_i^2}{m}\right)$ is a potential solution to the eigenproblem, $C\varphi = \frac{1}{m}X^TX\varphi = \lambda\varphi$. The pair with the largest eigen $\left(v_0, \frac{\sigma_0^2}{m}\right)$.

Rank-k approximations: the truncated SVD

We motivated PCA by asking for a single vector φ, which effectively projects the data onto a one-dimensional subspace (i.e., a line). You might to represent the original d-dimensional data points on a k-dimensional surface or subspace, where $k \leq s \leq d$. As the previous discussion sug could choose the top-k right singular vectors of X, $v_0, \ldots, v_{k-1}$.

Indeed, there is another "principled" reason for this choice.

Let $A \in \mathbb{R}^{m\times d}$ be any matrix with an SVD given by $A = U\Sigma V^T$. Per the notation above, let $s \equiv \min(m, d)$.

Then, define the *k-truncated SVD* as follows. Consider any $k \leq s$, and let U_k, Σ_k, and V_k consist of the singular vectors and values correspon largest singular values. That is, U_k is the first k columns of U, V_k is the first k columns of V, and Σ_k is the upper $k \times k$ submatrix of Σ. The k-t is the product $U_k\Sigma_kV_k^T$.

Now consider the following alternative way to write the SVD:

$$A = U\Sigma V^T = \sum_{i=0}^{s-1} u_i\sigma_iv_i^T.$$

Each term, $u_i\sigma_iv_i^T$ is known as a *rank*-1 product. So the existence of the SVD means that A may be written as a sum of rank-1 products.

It would be natural to try to *approximate* A by truncating the SVD after k terms, i.e.,

$$A \approx U_k\Sigma_kV_k^T = \sum_{i=0}^{k-1} u_i\sigma_iv_i^T.$$

And in fact, there is *no* rank-k approximation of A that is better than this one!

In particular, consider *any* pair of k column vectors, $Y_k \in \mathbb{R}^{m\times k}$ and $Z_k \in \mathbb{R}^{d\times k}$; their product, Y_kZ_k has rank at most k. Then there is a theore the smallest difference between A and the rank-k product Y_kZ_k, measured in the Frobenius norm, is

$$\min_{Y_k,Z_k} \|A - Y_kZ_k^T\|_F^2 = \|A - U_k\Sigma_kV_k^T\|_F^2 = \sigma_k^2 + \sigma_{k+1}^2 + \sigma_{k+2}^2 + \cdots + \sigma_{s-1}^2.$$

In other words, the truncated SVD gives the best rank-k approximation to A in the Frobenius norm. Moreover, the error of the approximation is squares of all the smallest $s - k$ singular values.

Applied to the covariance matrix, we may conclude that $C = \frac{1}{m}X^TX \approx \frac{1}{m}V_k\Sigma_k^2V_k^T$ is in fact the best rank-k approximation of C, which justifi the k eigenvectors corresponding to the top k eigenvalues of C as the principal components.

Summary: The PCA algorithm

Based on the preceding discussion, here is the basic algorithm to compute the PCA, given the data X and the desired dimension k of the subs

1. If the data are not already centered, transform them so that they have a mean of 0 in all coordinates, i.e., $\frac{1}{m}\sum_{i=0}^{m-1}\hat{x}_i = 0$.
2. Compute the k-truncated SVD, $X \approx U_k\Sigma_kV_k^T$.
3. Choose $v_0, v_1, \dots, v_{k-1}$ to be the principal components.

Demo: PCA on the UK Nutrition Study data

Let's try this algorithm out on the UK Nutrition Study data from above.

In [6]:
```
countries = ['England', 'Northern Ireland', 'Scotland', 'Wales']
products = df_uk['Product']
X_raw = df_uk[countries].as_matrix().T
print("X_raw:", X_raw.shape)

s = min(X_raw.shape)
print("s = min({}, {}) == {}".format(X_raw.shape[0], X_raw.shape[1], s))
```

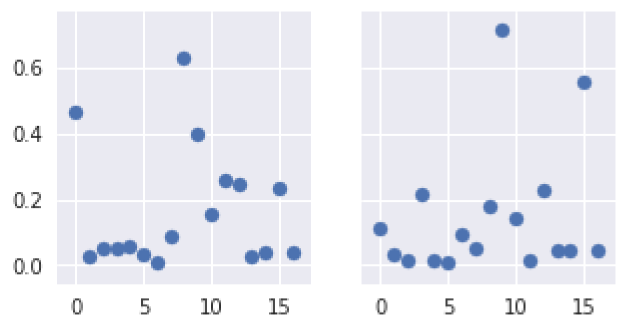
```
X_raw: (4, 17)
s = min(4, 17) == 4
```

In [7]:
```
X = X_raw - np.mean(X_raw, axis=0)
```

In [8]:
```
U, Sigma, VT = np.linalg.svd(X, full_matrices=False) # What does the `full_matrices` flag do?
print("U:", U.shape)
print("Sigma:", Sigma.shape)
print("VT:", VT.shape)
```

```
U: (4, 4)
Sigma: (4,)
VT: (4, 17)
```

In [9]:
```
m, d = X.shape
k_approx = 2
assert k_approx <= s

# Plot the components of the first k_approx=2 singular vectors
fig, axs = plt.subplots(1, k_approx, sharex=True, sharey=True,
                        figsize=(2.5*k_approx, 2.5))
for k in range(k_approx):
    axs[k].scatter(np.arange(max(m, d)), np.abs(VT[k, :].T))
```



In [10]:
```
print("Entries of the 1st singular vector with the largest magnitude:")
print(products[[0, 8, 9]])

print("\nEntries of the 2nd singular vector with the largest magnitude:")
print(products[[9, 15]])
```

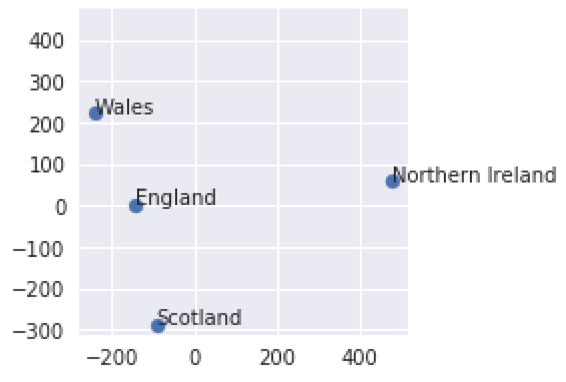
```
Entries of the 1st singular vector with the largest magnitude:
0    Alcoholic drinks
8         Fresh fruit
9      Fresh potatoes
Name: Product, dtype: object

Entries of the 2nd singular vector with the largest magnitude:
9     Fresh potatoes
15       Soft drinks
Name: Product, dtype: object
```

In [11]:
```
fig = plt.figure(figsize=(3, 3))
Y_k = X.dot(VT[0:2, :].T)
plt.scatter(Y_k[:, 0], Y_k[:, 1])
for x, y, label in zip(Y_k[:, 0], Y_k[:, 1], countries):
    plt.annotate(label, xy=(x, y))
```

```
ax = plt.axes()
ax.axis('square')
```

```
/usr/local/lib/python3.5/dist-packages/matplotlib/cbook/deprecation.py:106: MatplotlibDeprecatic
Adding an axes using the same arguments as a previous axes currently reuses the earlier instance
uture version, a new instance will always be created and returned.  Meanwhile, this warning can
ssed, and the future behavior ensured, by passing a unique label to each axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

```
Out[11]: (-276.44818576299815, 513.3106769439385, -311.64182973017, 478.1170329767666)
```



Fin! That's the end of these notes. If you've understood them, you are ready to move on to the next notebook in this assignment.

SVD-based (image) compression

As an exercise in applying the SVD, this notebook asks you to apply it to the literal problem of compression, in this case for images.

Note that this problem was original a final exam question in an earlier edition of the on-campus class, so if nothing else consider it good practic

Setup

Let's load some modules and an image, which you will use the SVD to compress.

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline

        import numpy as np

        from PIL import Image

        def im2gnp(image):
            """Converts a PIL image into an image stored as a 2-D Numpy array in grayscale."""
            return np.array(image.convert ('L'))

        def gnp2im(image_np):
            """Converts an image stored as a 2-D grayscale Numpy array into a PIL image."""
            return Image.fromarray(image_np.astype(np.uint8), mode='L')

        def imshow_gray(im, ax=None):
            if ax is None:
                f = plt.figure()
                ax = plt.axes()
            ax.imshow(im,
                      interpolation='nearest',
                      cmap=plt.get_cmap('gray'))
```

Let's download an image and represent it by a Numpy matrix, `img`.

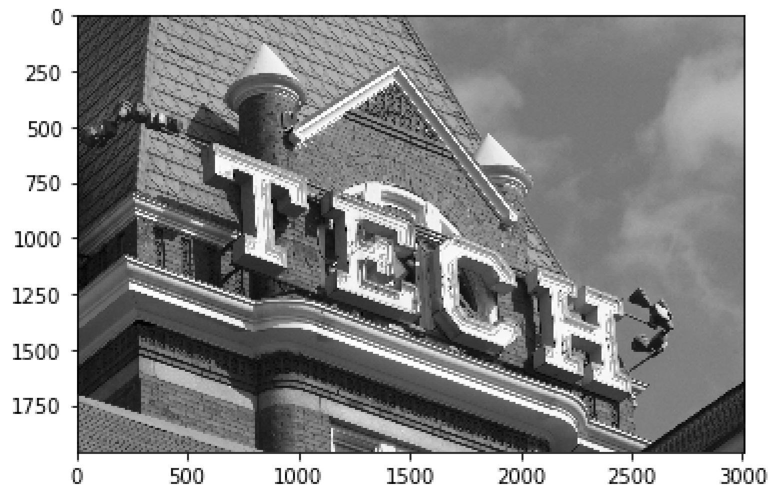
```
In [2]: import requests
        import os
        import hashlib
        import io
```

```
def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(loca
                                                                                            body_
                                                                                            checl
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/tech-tower/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/tech-tower/"
    DATA_PATH = ""

datasets = {'tt1.jpg': '380479dfdab7cdc100f978b0e00ad814'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'tt1.jpg' is ready!

(All data appears to be ready.)
```

Next, let's convert this image to grayscale and print some stats, e.g., its dimensions and size (in pixels).

In [3]:
```
pic_raw = Image.open('{}tt1.jpg'.format(DATA_PATH), 'r')
pic = im2gnp(pic_raw)
imshow_gray(pic)
```



Exercise 0 (1 point). Write a function that, given a Numpy array-representation of a (grayscale) image, returns its size in pixels.

In [4]:
```
def sizeof_image(I):
    assert type(I) is np.ndarray
    assert len(I.shape) == 2
    ### BEGIN SOLUTION
    return I.shape[0] * I.shape[1]
    ### END SOLUTION
```

In [5]:
```
# Test cell: `sizeof_image_test`

pic_pixels = sizeof_image (pic)

print ("The image uses about {:.1f} megapixels.".format (1e-6 * pic_pixels))

assert pic_pixels == 5895680
print ("\n(Passed!)")
```

```
The image uses about 5.9 megapixels.

(Passed!)
```

Compression via the truncated SVD

Recall that the k-truncated SVD gives the best rank-k approximation to a matrix A. That is, if the SVD of A is $A = U\Sigma V^T$, then we can approx

$$A \approx U_k \Sigma_k V_k^T,$$

where (U_k, Σ_k, V_k^T) is the k-truncated SVD, taken by retaining the k largest singular values (entries of diagonal matrix Σ), as well as the first k left and right singular vectors (U and V).

Key idea. For the rest of this notebook, if you understand the following idea, then the rest should make sense.

Suppose we represent the image as a matrix, A. Then we can also approximate A with its k-truncated SVD, (U_k, Σ_k, V_k^T). **If you can choose size of the representation, (U_k, Σ_k, V_k^T), is less than the size of original image, then we have compressed it.**

Exercise 1 (2 point). Write a function, `compress_image(I, k)`, that compresses an input image `I` by interpreting `I` as a matrix, computing tl SVD, and returning a compressed representation.

For this function, we want you to return a tuple, `(S, Uk, VkT)`, where `S` is an array of *all* the singular values (not just the top k) and `Uk` and `Vk` *truncated* singular vectors.

In [6]:
```
def compress_image(I, k):
    ### BEGIN SOLUTION
    U, Sigma, VT = np.linalg.svd(I, full_matrices=False)
    return (Sigma, U[:, :k], VT[:k, :])
    ### END SOLUTION

# Runs your function:
k = 10
Sigma, Uk, VkT = compress_image(pic, k)

print(Sigma.shape)
print(Uk.shape)
print(VkT.shape)
```

```
(1960,)
(1960, 10)
(10, 3008)
```

In [7]:
```
# Test cell: `compress_image_test`

assert Sigma.shape == (min(pic.shape),)
assert Uk.shape == (pic.shape[0], k)
assert VkT.shape == (k, pic.shape[1])
assert (Sigma[k:]**2).sum () <= 7e9

print ("\n(Passed!)")
```

```
(Passed!)
```

Exercise 2 (2 points). Write a function, `sizeof_compressed_image()`, that returns the number of "equivalent pixels" used by the compress SVD) representation.

What are "equivalent pixels?" Recall these facts:

- In the original grayscale image, only 1 byte is needed to store each pixel.
- In the (truncated) SVD representation, each matrix entry is a double-precision value, which requires 8 bytes.

In your implementation of this function, you may count just the k largest singular values even though we asked that your `compress_image()` *all* of the singular values.

In [8]:
```
def sizeof_compressed_image(Sigma, Uk, VkT):
    ### BEGIN SOLUTION
    m, k = Uk.shape
    k2, n = VkT.shape
    assert k == k2
    return 8 * (m*k + k + k*n)
    ### END SOLUTION
```

In [9]:
```
# Test cell: `sizeof_compressed_image_test`

cmp_pixels = sizeof_compressed_image(Sigma, Uk, VkT)

print("Original image required ~ {:.1f} megapixels.".format (1e-6 * pic_pixels))
print("Compressed representation retaining k={} singular values is equivalent to ~ {:.1f} megap
mat (k, 1e-6 * cmp_pixels))
print("Thus, the compression ratio is {:.1f}x.".format (pic_pixels / cmp_pixels))

assert cmp_pixels == 397520
print ("\n(Passed!)")
```

```
Original image required ~ 5.9 megapixels
Original image required ~ 5.9 megapixels.
Compressed representation retaining k=10 singular values is equivalent to ~ 0.4 megapixels.
Thus, the compression ratio is 14.8x.

(Passed!)
```

Exercise 3 (2 points). Recall that the error of the compressed representation, as measured in the squared Frobenius norm, is given by the sum singular values,

$$\|A - U_k \Sigma_k V_k^T\|_F^2 = \sigma_k^2 + \sigma_{k+1}^2 + \cdots + \sigma_{s-1}^2,$$

where $s = \min(m, n)$ if A is $m \times n$ and we assume that the singular values are sorted from largest (σ_0) to smallest (σ_{s-1}).

Write a function that returns the *relative* error, measured using the Frobenius norm, i.e.,

$$\frac{\|A - U_k \Sigma_k V_k^T\|_F}{\|A\|_F}.$$

```
In [10]: def compression_error (Sigma, k):
             """
             Given the singular values of a matrix, return the
             relative reconstruction error.
             """
             ### BEGIN SOLUTION
             S2 = Sigma**2
             S2_a = S2[:k].sum()
             S2_b = S2[k:].sum()
             return np.sqrt(S2_b / (S2_a + S2_b))
             ### END SOLUTION
```

```
In [11]: # Test cell: `compression_error_test`

         err = compression_error(Sigma, k)
         print ("Relative reconstruction (compression) error is ~ {:.1f}%.".format (1e2*err))
         assert 0.24 <= err <= 0.26
         print ("\n(Passed!)")
```

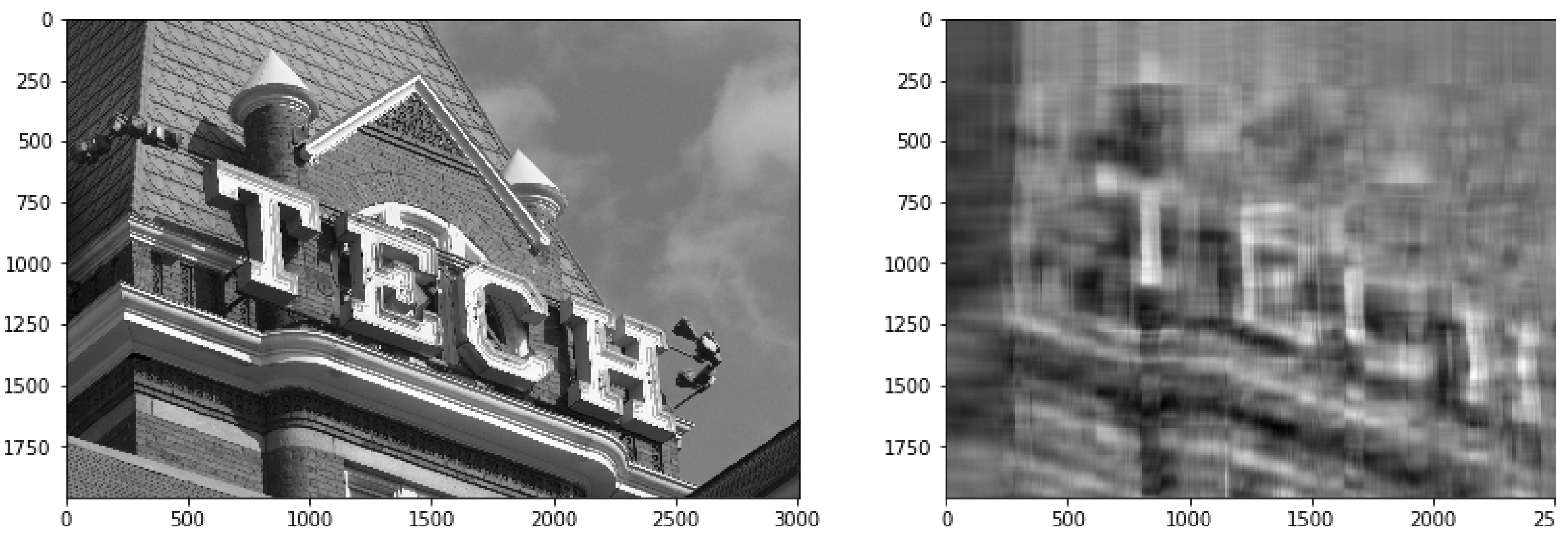
```
Relative reconstruction (compression) error is ~ 25.2%.

(Passed!)
```

Exercise 4 (2 points). Write a function that, given the compressed representation of an image, reconstructs it approximately.

```
In [12]: def uncompress_image(Sigma, Uk, VkT):
             assert Uk.shape[1] == VkT.shape[0]
             ### BEGIN SOLUTION
             k = Uk.shape[1]
             return (Uk * Sigma[:k]).dot(VkT)
             ### END SOLUTION
```

```
In [13]: # Test cell: `uncompress_image_test`

         pic_lossy = uncompress_image(Sigma, Uk, VkT)

         f, ax = plt.subplots(1, 2, figsize=(15, 30))
         imshow_gray(pic, ax[0])
         imshow_gray(pic_lossy, ax[1])

         abs_err = np.linalg.norm(pic - pic_lossy, ord='fro')
         rel_err = abs_err / np.linalg.norm(pic, ord='fro')
         print("Measured relative error is ~ {:.1f}%.".format(1e2 * rel_err))

         pred_rel_err = compression_error(Sigma, k)
         assert 0.95*pred_rel_err <= rel_err <= 1.05*pred_rel_err
         print("\n(Passed!)")
```

```
Measured relative error is ~ 25.2%.

(Passed!)
```



Exercise 5 (3 points). Write a function that, given the singular values of an image, determines the truncation point, k, that would reduce the rela below a desired threshold.

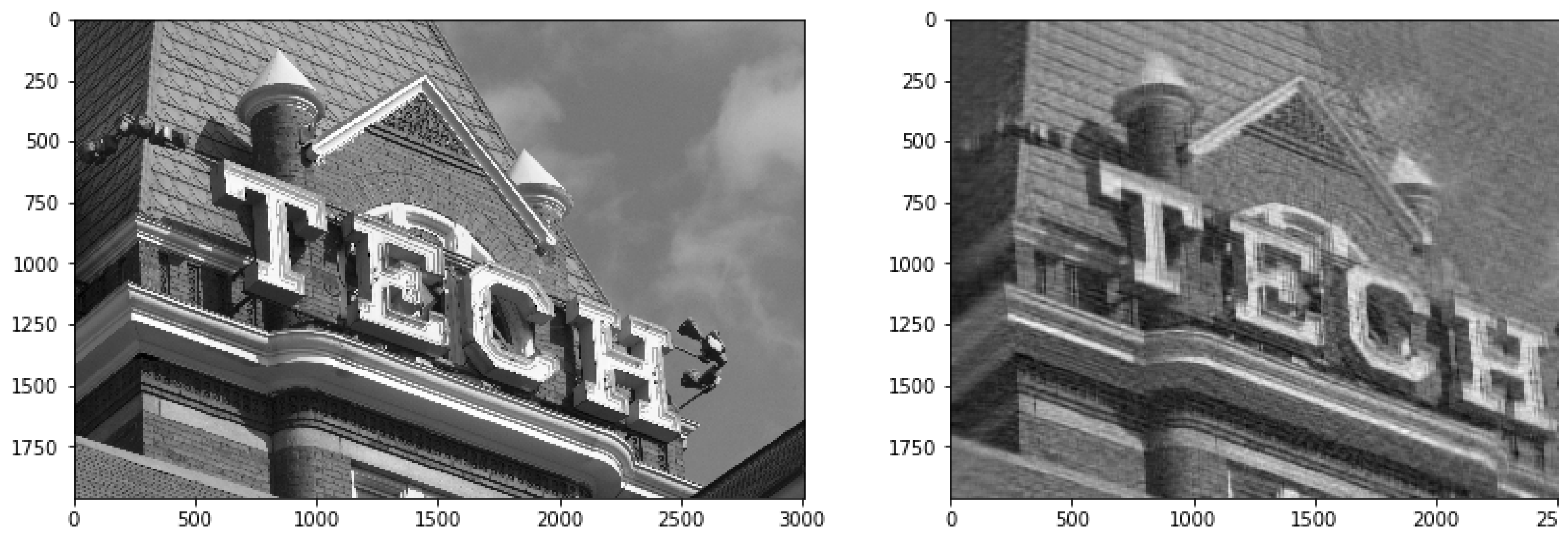
> *Hint:* You may find Numpy's cumulative sum (https://docs.scipy.org/doc/numpy/reference/generated/numpy.cumsum.html) handy.

In [14]:
```
def find_rank(rel_err_target, Sigma):
    ### BEGIN SOLUTION
    S_sq = Sigma**2
    abs_err_sq = np.cumsum(S_sq[::-1])[::-1]
    rel_err_sq = abs_err_sq / abs_err_sq[0]
    rel_err = np.sqrt(rel_err_sq)
    return np.min(np.where(rel_err < rel_err_target))
    ### END SOLUTION

rel_err_target = 0.15
k_target = find_rank(rel_err_target, Sigma)

print("Relative error target:", rel_err_target)
print("Suggested value of k:", k_target)
```

```
Relative error target: 0.15
Suggested value of k: 47
```

In [15]:
```
# Test cell: `find_rank_test`

print("Compressing...")
Sigma_target, Uk_target, VkT_target = compress_image(pic, k_target)
target_pixels = sizeof_compressed_image(Sigma_target,
                                        Uk_target,
                                        VkT_target)
target_ratio = pic_pixels / target_pixels
print("Estimated compression ratio: {:.1f}x".format(target_ratio))

pic_target = uncompress_image(Sigma_target, Uk_target, VkT_target)
f, ax = plt.subplots(1, 2, figsize=(15, 30))
imshow_gray(pic, ax[0])
imshow_gray(pic_target, ax[1])

assert compression_error(Sigma, k_target) <= rel_err_target
assert compression_error(Sigma, k_target-1) > rel_err_target

print("\n(Passed!)")
```

```
Compressing...
Estimated compression ratio: 3.2x

(Passed!)
```



‹ Previous

Next Up: Practice Problems for Midterm 2 ›
1 min + 1 activity

© All Rights Reserved

edX

About
Affiliates
edX for Business
Open edX
Careers
News

Legal

Terms of Service & Honor Code
Privacy Policy
Accessibility Policy
Trademark Policy
Sitemap

Connect

Blog
Contact Us
Help Center
Media Kit
Donate

GET IT ON Google play
Download on the App Store