Sample solutions

part0 (Score: 0.0 / 0.0)

1. Test cell (Score: 0.0 / 0.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

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 the eigenvectors of a certain matrix; one way to compute those eigenvectors is to use a special factorization from linear algebra called the *Singular Value Decomposition* (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 need to submit. However, you should try to understand all the code steps that appear in the subsection entitled, **Principal Components Analysis (PCA)**, as you will need to 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*, *classification*, and *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 ways. First, it can make the data set smaller and therefore faster to process or analyze. Secondly, choosing a clever representation can reveal hidden structure.

As a concrete example, consider the problem of *dimensionality reduction*: given a d-dimensional data set, we wish to transform it into a smaller k-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 examples at 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 the average 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 '{}'".forma
t(local_file,

body_checksum,

checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/uk-food/"
    DATA_PATH = "../resource/lib/publicdata/uk-food/"
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.)")
```

```
Downloading: https://github.com/cse6040/labs-fa17/raw/master/datasets/uk-food/uk-nutritio
n-data.csv ...

'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")
```

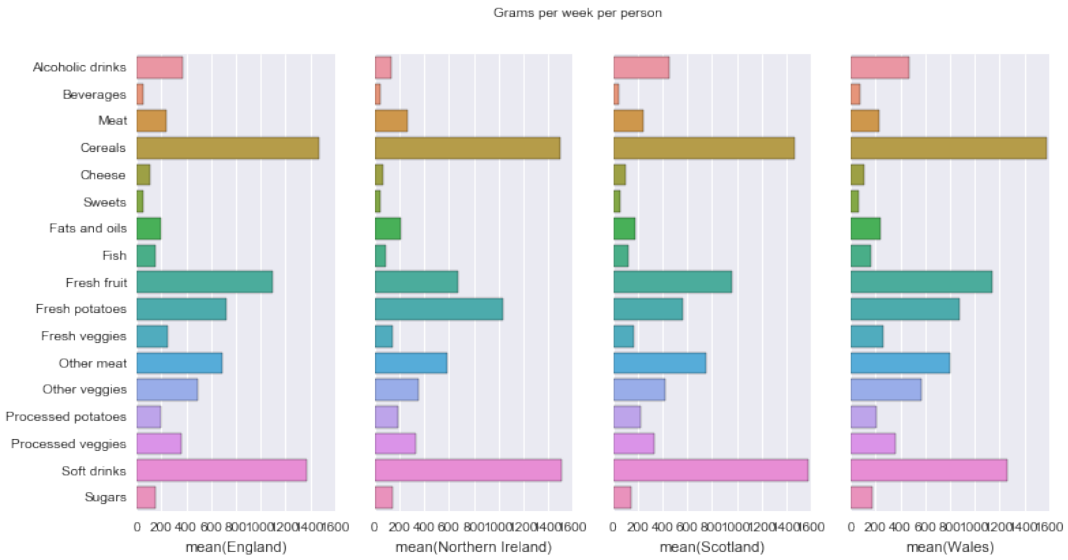
```
17 x 5 table of data:
```

	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]: <matplotlib.text.Text at 0x11a36a630>
```



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 any case, visual 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) data points, one for each country, embedded in a 17-dimensional space (one dimension per food category). The following page will help build your intuition for PCA; the notes 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/)

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. Each observation is represented by a d-dimensional real-valued vector corresponding to d measured predictors. As usual, we can stack these into a data matrix, denoted $X \equiv \begin{pmatrix} \hat{x_0}^T \\ \vdots \\ \hat{x_{m-1}}^T \end{pmatrix}$.

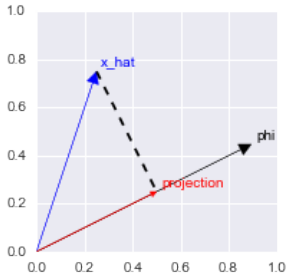
However, we'll add one more important assumption: these data should be *centered* about their mean, i.e., $\frac{1}{m} \sum_{i=0}^{m-1} \hat{x_i} = 0$. If the observations are not centered initially, then preprocess them accordingly.

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 the length of 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
project x_hat
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', len
gth_includes_head=True)
ax.arrow(0, 0, phi[0], phi[1], head_width=0.05, head_length=0.05, fc='k', ec='k', length_
includes_head=True)
ax.arrow(0, 0, x_hat_proj_phi[0], x_hat_proj_phi[1], head_width=0.025, head_length=0.025,
fc='r', ec='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 all the data 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:

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

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:

$$\frac{1}{\quad} \; x^T \left(\frac{1}{\quad} \sum_{\quad}^{m-1} \; x^T \right)$$

$$\begin{aligned} J(\phi) &= \frac{1}{2}\varphi^{\iota}\left(\frac{1}{m}\sum_{i=0} x_i^{\cdot} x_i^{\tau}\right)\varphi \\ &= \frac{1}{2}\varphi^T\left(\frac{1}{m}X^TX\right)\varphi \\ &\equiv \frac{1}{2}\varphi^T C\varphi. \end{aligned}$$

In the last step, we defined $C \equiv \frac{1}{m}X^TX$. In statistics, if X represents mean-centered data, then the matrix C is also known as the *sample covariance matrix* (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 on *maximizing 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 any minimization or maximization problem that has equality constraints. The idea is to modify the cost function in a certain way that effectively incorporates each constraint: 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:

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

> 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 $C = \frac{1}{m}X^TX$.

The matrix C will usually have many eigenvalues and eigenvectors. So which one do you want? Plug the eigenvector back into the original cost function. Then, $J(\varphi) = \frac{1}{2}\varphi^TC\varphi = \frac{\lambda}{2}\varphi^T\varphi = \frac{\lambda}{2}$. 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 and storage, 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 tool for simplifying linear algebra problems. It can also be somewhat expensive to compute accurately, but a lot of scientific and engineering effort has gone into 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 or columns. 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 $U^T U = I$ and $V^T V = I$; and the matrix Σ is an $s \times s$ diagonal matrix.

> 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 "column-view" of a 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 equations,

$$Xv_i = \sigma_i u_i.$$

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

$$X^T X = V \Sigma^T U^T U \Sigma V^T = V \Sigma^2 V^T,$$

or

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

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

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

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

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 instead want 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 suggests, you 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 corresponding to the k 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-truncated SVD is the product $U_k \Sigma_k V_k^T$.

Now consider the following alternative way to write the SVD:

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

Each term, $u_i \sigma_i v_i^T$ 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.,

$$A \approx U_k \Sigma_k V_k^T = \sum_{i=0}^{k-1} u_i \sigma_i v_i^T.$$

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, $Y_k Z_k$ has rank at most k. Then there is a theorem that says the smallest difference between A and the rank-k product $Y_k Z_k$, measured in the Frobenius norm, is

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

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 the sum of the squares of all the smallest $s - k$ singular values.

Applied to the covariance matrix, we may conclude that $C = \frac{1}{m} X^T X \approx \frac{1}{m} V_k \Sigma_k^2 V_k^T$ is in fact the best rank-k approximation of C, which justifies choosing 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 subspace.

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, $X \approx U_k \Sigma_k V_k^T$.
3. Choose $v_0, v_1, \ldots, v_{k-1}$ 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))
```

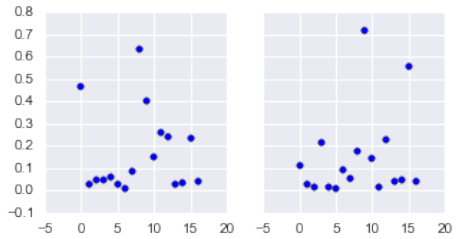
```
X_raw: (4, 17)
s = min(4, 17) == 4
```

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]])
```

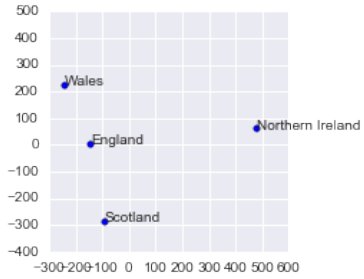
```
Entries of the 1st singular vector with the largest magnitude:
0    Alcoholic drinks
8         Fresh fruit
9      Fresh potatoes
Name: Product, dtype: object

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

In [11]:

Grade cell: `cell-d1cb2c73d30af839` Score: 0.0 / 0.0 (Top)

```
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')
```

Out[11]: (-300.0, 600.0, -400.0, 500.0)



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.

English

POWERED BY OPENedX