1 Principal Component Analysis and Latent Semantic Indexing

Lab Objective: *Understand the basics of principal component analysis and latent semantic indexing.*

Principal Component Analysis

Understanding the variance in complex data is one of the first tasks encountered in exploratory data analysis. For an example, consider the scatter plot displaying the sepal and petal lengths of 100 different irises shown in Figure 1.1.

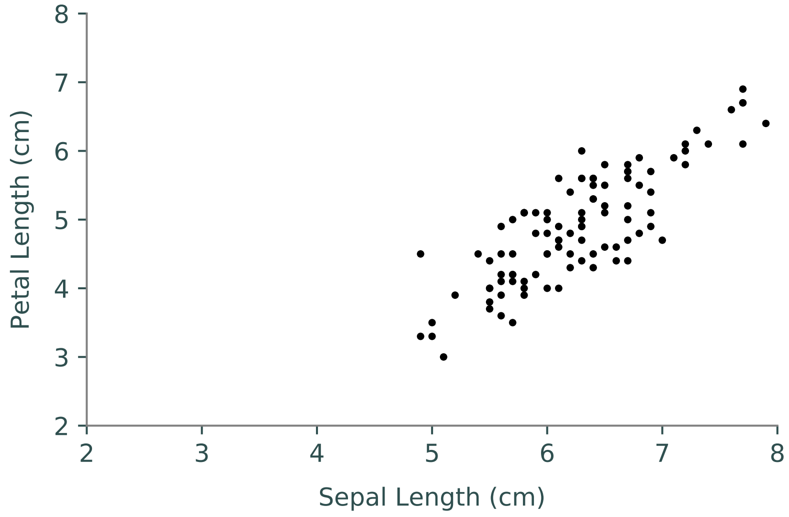

Figure 1.1: Sepal Length vs. Petal Length for 100 iris flowers. Note the strong correlation of these variables.

There are three distinct types of iris flowers present: *setosa*, *versicolor*, and *virginica*. Considering this data, we might ask how to best distinguish the different types of irises based on their given sepal and petal lengths. We can answer this question by finding the characteristic that causes the greatest variance in the data. (Greater variance implies a greater ability to distinguish between data points. If the variance is very small, the data are clustered tightly together, and it is difficult to distinguish well.)

Upon examination, we see that the petal length ranges between 3 and 7 cm, while the sepal length only ranges between 5 and 8 cm. We might be tempted to say that the most distinguishing aspect of irises is their petal length, but this is only considering the features of the data individually, and not collectively. The two features of the data are clearly correlated, and a more careful consideration would lead us to conclude that the most distinguishing aspect of irises is their overall size. Some irises are are much larger than others, while the sepal and petal lengths stay roughly in proportion.

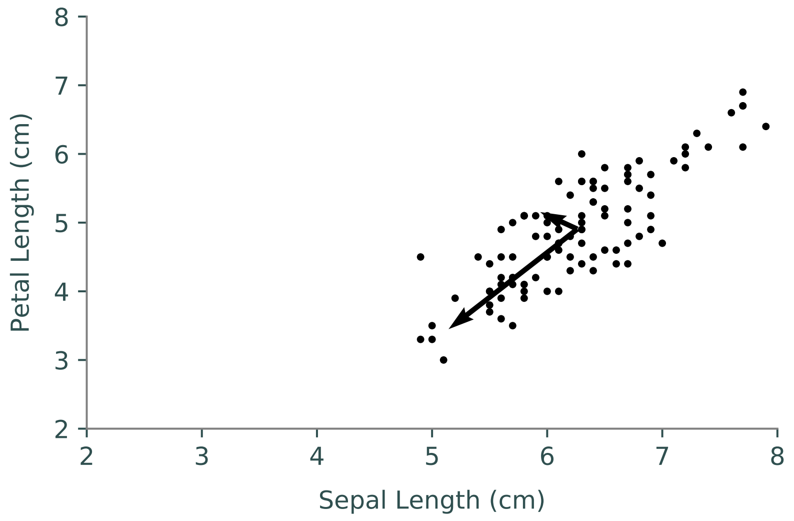

Figure 1.2: The vectors indicate the two principal components, which are weighted by their contribution to the variance.

Principal Component Analysis (PCA) is a multivariate statistical tool used to orthogonally change the basis of a set of observations from the basis of original features (which may be correlated) into a basis of uncorrelated (in fact, orthonormal) variables called the *principal components*. It is a direct application of the singular value decomposition (SVD) from linear algebra. More specifically, the first principal component will account for the greatest variance in the set of observations, the second principal component will be orthogonal to the first, accounting for the second greatest variance in the set of observations, etc. The first several principal components capture most of the variance in the observation set, and hence provide a great deal of information about the data. By projecting the observations onto the space spanned by the principal components, we can reduce the dimensionality of the data in a manner that preserves most of the variance.

In our iris example, the two principal components are shown in Figure 1.2. The first principal component, corresponding intuitively to iris size, accounts for 96% of the variance in the data. The second, which accounts for only 4% of the variance, corresponds to the relative sepal and petal length of irises of the same size.

Computing the Principal Components

We now explore how to use the SVD to compute the principal components of a dataset. Throughout this lab we will use the `sklearn` iris data set, which can be obtained as follows:

```
>>> import numpy as np
>>> from scipy import linalg as la
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
```

We represent the collection of observations as an $n \times m$ matrix X, where each row of X is an observation, and each column is a specific feature. Let $k = \min(m, n)$. We will use this later. In the iris example, X contains 150 observations, each consisting of 4 features (so $k = 4$), as shown below:

```
>>> X.shape
(150, 4)
>>> iris.feature_names
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

The first step in PCA is to pre-process the data. In particular, we first translate the columns of X to have mean 0. The data may then be optionally scaled to remove discrepancies arising from different units of measure (i.e. centimeters vs meters), and we call the new matrix containing the centered and scaled data Y. In this lab, we will not have any scaling issues, so we won't address this issue any further. Thus we can pre-process our iris data simply as follows:

```
>>> Y = X - X.mean(axis=0)
```

We next compute the truncated SVD of our centered and scaled data,

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

where U is $n \times k$, Σ is a $k \times k$ diagonal matrix containing the singular values of Y in decreasing order along the diagonal, and V is $m \times k$. The columns of V are the principal components (which form an orthonormal basis for the space spanned by the observations), and the corresponding singular values provide us information about how much variance is captured in each principal component. More specifically, let σ_i be the i-th non-zero singular value. Then the value

$$\frac{\sigma_i^2}{\sum_{j=1}^k \sigma_j^2}$$

is the percentage of the variance captured by the i-th principal component. We compute the truncated SVD of the iris data and show the variance percentages for each component below:

```
>>> U,S,VT = la.svd(Y, full_matrices=False)
>>> S**2/(S**2).sum() # variance percentages
array([ 0.92461621,  0.05301557,  0.01718514,  0.00518309])
```

In general, we are only interested with the first several principal components. But just how many principal components should we keep? There are a number of ways to decide this. One is to only keep the first two principal components, as these enable us to project the data into 2-dimensional space, which is easy to visualize. Another way is to only keep the set of principal components accounting for a certain percentage (say 80%) of the variance. A third method is to examine the *scree plot* of the variance percentages for each principal component, as in Figure 1.3.

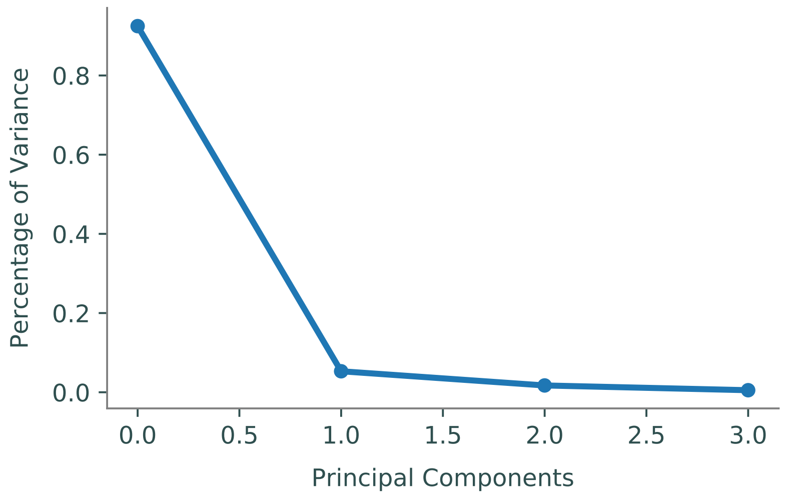

Figure 1.3: Scree plot of the percentage of variance for PCA on the iris dataset.

Upon examination of the iris scree plot, we see that there is a distinct change after the first principal component. This method is referred to as finding the "elbow" of the scree plot, and we keep all the principal components on the left of the elbow. In the case of the iris data, that is simply the first principal component, which accounts for 92% of the variance.

Once we have decided how many principal components to keep (say the first l), we can project the observations from the original feature space onto the principal component space by computing

$$\widehat{Y} = U_{:,:l}\Sigma_{:l,:l}$$

where $\Sigma_{:l,:l}$ is the first l rows and columns of Σ and $U_{:,:l}$ is the first l columns of U. Using the SVD formula, note that

$$\widehat{Y} = YV_{:,:l},$$

where $V_{:,:l}$ is the first l columns of V. In this way, we see that the i-th row of $\widehat{Y}$ is simply the projection of the i-th observation onto the orthonormal set of the first l principal components. Under this projection, the data is represented in fewer dimensions, and in such a way that accentuates the variance (which can help with finding patterns within the data).

In Figure 1.4 we display the transformed iris data set, plotting the first principal component against the second. This reduction helps us to see the distinctions between the three different species, using only two dimensions instead of the full four dimensions of the feature space.

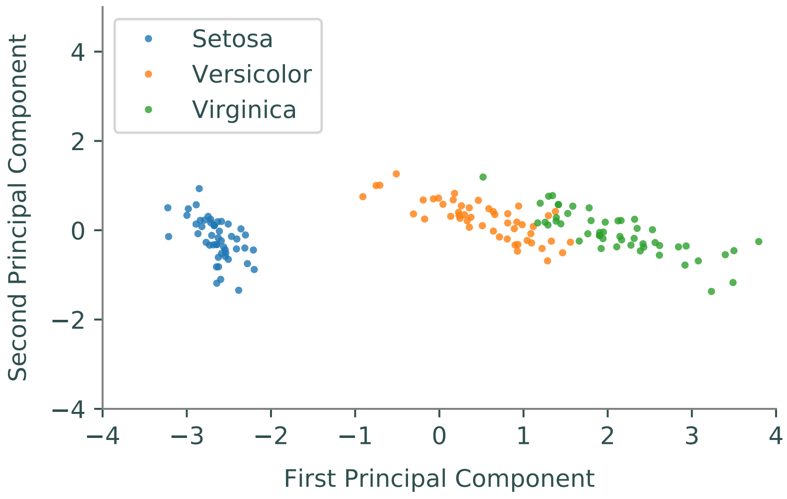

Figure 1.4: Plot of the transformed iris data, keeping only the first two principal components.

Problem 1. Write a function that recreates the plot shown in Figure 1.4 by performing PCA on the iris dataset, keeping the first two principal components.

Note: If `Yhat` is your 150×2 array of transformed observations, you can access the rows corresponding to the setosa flowers as follows:

```
>>> Yhat[iris.target==0]
```

To get the rows corresponding to versicolor and virginica specimens, simply replace the 0 with 1 and 2, respectively.

Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an application of PCA which applies the ideas we have discussed to the realm of natural language processing. In particular, LSI employs the SVD to reduce the dimensionality of a large corpus of text documents in order to enable us to evaluate the similarity between two documents. Many information-retrieval systems used in government and in industry are based on LSI.

To motivate the problem, suppose we have a large collection of documents dealing with various statistical and mathematical topics. How can we find an article about PCA? We might consider simply choosing the article which contains the acronym *PCA* the greatest number of times, but this is a crude method. A better way is to use a form of PCA on the collection of documents.

In order to do so, we need to represent the documents as numerical vectors. A standard way of doing this is to define an ordered set of words occurring in the collection of documents (called the *vocabulary*), and then to represent each document as a vector of word counts from the vocabulary. More formally, let our vocabulary be $V = \{w_1, w_2, \ldots, w_m\}$. Then a document is a vector $x = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$ such that x_i is the number of occurrences of word w_i in the document. In this setup, we represent the entire collection of m documents as an $n \times m$ matrix X, where m is the number of vocabulary words and n is the number of documents in our collection, each row being a document vector. As expected, we let $X_{i,j}$ be the number of times term j occurs in document i. Note that X is often a sparse matrix, as any one document likely doesn't contain most of the vocabulary words. This mode of representation is called the *bag of words* model for documents.

We calculate the SVD of X without centering or scaling the data so that we may retain the sparsity. We now have $X = U\Sigma V^T$. Once we have selected the number of principal components to keep, say l, we can represent the corpus of documents by the matrix

$$\widehat{X} = U_{:,:l}\Sigma_{:l,:l} = XV_{:,:l}.$$

Note that $\widehat{X}$ will no longer be a sparse matrix, but it has dimensions $n \times l$, which is much smaller than $n \times m$ when $l \ll m$.

Now that we have our documents represented in terms of the first l principal components, we can find the similarity between two documents. Our measure for similarity is just the cosine of the angle between the vectors; a small angle (and hence large cosine) indicates greater similarity, while a large angle (hence small cosine) indicates greater dissimilarity. Recall that we can use the inner product to find the cosine of the angle between two vectors. Under this metric, the similarity between document i and document j (represented by the i-th and j-th row of $\widehat{X}$, notated $\widehat{X}_i$ and $\widehat{X}_j$, respectively) is just

$$\frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\|\|\widehat{X}_j\|}.$$

To find the document most similar to document i, we simply compute

$$\text{argmax}_{j \neq i} \frac{\langle \widehat{X}_i, \widehat{X}_j \rangle}{\|\widehat{X}_i\|\|\widehat{X}_j\|}.$$

We now discuss some practical issues involved in creating the bag of words representation X from the raw text. Our dataset will consist of the US State of the Union addresses from 1945 through 2013, each contained in a separate text file in the folder `Addresses`. We would like to avoid loading in all of the text into memory at once, and so we will *stream* the documents one at a time.

The first thing we need to establish is the vocabulary set, i.e. the set of unique words that occur throughout the collection of documents. A Python set object automatically preserves the uniqueness of the elements, so we will create a set, and then iteratively read through the documents, adding the unique words of each document to the set. As we read in each document, we will remove punctuation and numerical characters and convert everything to lower case. The following code will accomplish this task:

```
>>> import os
>>> import string

# Get list of file paths to each text file in the folder
>>> folder = "./Addresses/"
>>> paths = [folder+p for p in os.listdir(folder) if p.endswith(".txt")]

# Helper function to get list of words in a string
>>> def extractWords(text):
...     ignore = string.punctuation + string.digits
...     cleaned = "".join([t for t in text.strip() if t not in ignore])
...     return cleaned.lower().split()
```

```
...
# Initialize vocab set, then read each file and add to the vocab set.
vocab = set()
>>> for p in paths:
...     with open(p, 'r') as infile:
...         for line in infile:
...             vocab.update(extractWords(line))
...
```

We now have a set containing all of the unique words in the corpus. However, many of the most common words do not provide important information. We call these *stop words*. Examples in English include *the, a, an, and, I, we, you, it, there*, etc; a list of common English stop words is given in `stopwords.txt`. We remove the stop words from our vocabulary set as follows, and then fix an ordering to the vocabulary by creating a dictionary whose key-value pairs are of the form (word, index):

```
>>> # Load stopwords.
>>> with open("stopwords.txt", 'r') as f:
...     stops = set([w.strip().lower() for w in f.readlines()])

>>> # Remove stopwords from vocabulary, create ordering.
>>> vocab = {w:i for i, w in enumerate(vocab.difference(stops))}
```

We are now ready to create the word count vectors for each document, and we store these in a sparse matrix X. It is convenient to use the `Counter` object from the `collections` module, as this object automatically counts the occurrences of each distinct element in a list.

```
>>> from scipy import sparse
>>> from collections import Counter
>>> counts = [] # holds the entries of X
>>> doc_index = [] # holds the row index of X
>>> word_index = [] # holds the column index of X

# Iterate through the documents.
>>> for doc, p in enumerate(paths):
...     with open(p, 'r') as f:
...         # create the word counter
...         ctr = Counter()
...         for line in f:
...             ctr.update(extractWords(line))
...         # Iterate through the word counter, storing counts
...         for word, count in ctr.items():
...             if word in vocab:
...                 word_index.append(vocab[word])
...                 counts.append(count)
...                 doc_index.append(doc)

# Create sparse matrix holding these word counts.
>>> X = sparse.csr_matrix((counts, [doc_index, word_index]),
                    shape=(len(paths), len(vocab)), dtype=np.float)
```

Problem 2. Using the techniques of LSI discussed above—applied to the word count matrix X, and keeping the first 7 principal components—write a function that takes in the path to a speech and returns a tuple of the most and least similar speeches. Test your answer on Ronald Reagan's 1984 speech.

Since X is a sparse matrix, you will need to use the SVD method found in `scipy.sparse.linalg`. This method operates slightly differently than the SVD method found in `scipy.linalg`, so be sure to read the documentation.

The simple bag of words representation is a bit crude, as it fails to consider how some words may be more important than others in determining the similarity of documents. Words appearing in few documents tend to provide more information than words occurring in every document. For example, while the word *war* might not be considered a stop word, it is likely to appear in quite a few addresses, whereas *Afghanistan* will not. Thus two speeches sharing the word *Afghanistan* ought to be considered more related than two speeches sharing the word *war*. So while $X_{i,j}$ is a good measure of the importance of term j in document i, we also need to consider some kind of global weight for each term j, indicating how important the term is over the entire collection. There are a number of different weights we could choose; we choose to employ the following approach:

Let t_j be the total number of times term j appears in the entire collection of documents. Define

$$p_{i,j} = \frac{X_{i,j}}{t_j}.$$

We then let

$$g_j = 1 + \sum_{i=1}^{m} \frac{p_{i,j}\log(p_{i,j}+1)}{\log m},$$

where m is the number of documents in the collection. We call g_j the *global weight* of term j. We replace each term frequency in the matrix X by weighting it globally. Specifically, we define a matrix A with entries

$$A_{i,j} = g_j \log(X_{i,j}+1).$$

We can now perform LSI on the matrix A, whose entries are both locally and globally weighted.

To calculate the matrix A in a streaming manner, we must alter our code above somewhat:

```
>>> from math import log

>>> t = np.zeros(len(vocab))
>>> counts = []
>>> doc_index = []
>>> word_index = []

# get doc-term counts and global term counts
>>> for doc, path in enumerate(paths):
...     with open(path, 'r') as f:
...         # create the word counter
...         ctr = Counter()
...         for line in f:
...             words = extractWords(line)
...             ctr.update(words)
...         # iterate through the word counter, store counts
...         for word, count in ctr.items():
...             if word in vocab:
...                 word_ind = vocab[word]
...                 word_index.append(word_ind)
...                 counts.append(count)
...                 doc_index.append(doc)
...                 t[word_ind] += count
...
# Get global weights.
>>> g = np.ones(len(vocab))
>>> logM = log(len(paths))
>>> for count, word in zip(counts, word_index):
...     p = count/float(t[word])
...     g[word] += p*log(p+1)/logM

# Get globally weighted counts.
>>> gwcounts = []
>>> for count, word in zip(counts, word_index):
...     gwcounts.append(g[word]*log(count+1))

# Create sparse matrix holding these globally weighted word counts
>>> A = sparse.csr_matrix((gwcounts, [doc_index,word_index]),
                      shape=(len(paths), len(vocab)), dtype=np.float)
```

Problem 3. Repeat Problem 2 using the matrix A. Do your answers seem more reasonable than before?