8 Linear Unsupervised Learning

8.1 Fixed spanning sets, orthonormality, and projections

In this Section we review the rudimentary concepts from linear algebra that are crucial to understanding unsupervised learning techniques. The interested reader needing a refresher in basic vector and matrix operations (which are critical to understanding the concepts presented here) may find a review of such topics in Sections 8.8 and 8.9.

8.1.1 Notation

As we have seen in previous Chapters, data associated with *supervised* tasks of regression and classification always comes in as input/output pairs. Such dichotomy does not exist with *unsupervised* learning tasks wherein a typical dataset is written simply as a set of P (input) points

$$\{\mathbf{x}_1,\ \mathbf{x}_2, \ldots, \mathbf{x}_P\} \tag{8.1}$$

or $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ for short, all living in the same N-dimensional space. Throughout the remainder of this Section we will assume that our dataset has been *mean-centered*: a simple and completely reversible operation that involves subtracting off the mean of the dataset along each input dimension so that it straddles the origin.

As illustrated in Figure 8.1 when thinking abou points in a multi-dimensional vector space we can picture them either as *dots* (as shown in the left panel), or as *arrows* stemming from the origin (as shown in the middle panel). The former is how we have chosen to depict our regression and classification data thus far in the book since it is just more aesthetically pleasing to picture linear regression as fitting of a line to a *scatter of dots* as opposed to a *collection of arrows*. The equivalent latter perspective however (i.e., viewing multi-dimensional points as arrows) is the conventional way vectors are depicted, e.g., in any standard linear algebra text. In discussing unsupervised learning techniques it is often helpful to visualize points living in an N-dimensional space using both of these conventions, some as *dots* and some as *arrows* (as shown in the right panel of Figure 8.1). Those vectors drawn as arrows are particular points, often called a

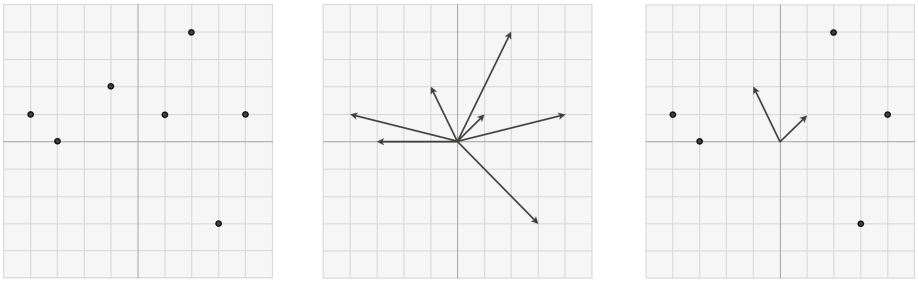

Figure 8.1 Two-dimensional points illustrated as dots (left panel), arrows (middle panel), and a mixture of both (right panel). Those shown as arrows on the right are a *basis* or *spanning set* over which we aim to represent every point in the entire space.

basis or *spanning set of vectors*, over which we aim to represent every other point in the space.

Notationally, we denote a spanning set by

$$\{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\} \tag{8.2}$$

or, $\{\mathbf{c}_k\}_{k=1}^{K}$ for short.

8.1.2 Perfect representation of data using fixed spanning sets

A spanning set is said to be capable of perfectly representing all P of our points if we can express each data point $\mathbf{x}_p$ as some linear combination of our spanning set's members, as

$$\sum_{k=1}^{K} \mathbf{c}_k w_{p,k} = \mathbf{x}_p, \qquad p = 1, ..., P. \tag{8.3}$$

Generally speaking two simple conditions, if met by a spanning set of vectors $\{\mathbf{c}_k\}_{k=1}^{K}$, guarantee all P equalities in Equation (8.3) to hold regardless of the dataset $\left\{\mathbf{x}_p\right\}_{p=1}^{P}$ used: (i) $K = N$, or in other words, the number of spanning vectors matches the data dimension[1], and (ii) all spanning vectors are linearly independent[2]. For such a spanning set Equation (8.3) can be written more compactly as

$$\mathbf{C}\mathbf{w}_p = \mathbf{x}_p, \qquad p = 1, ..., P \tag{8.4}$$

where the spanning matrix $\mathbf{C}$ is formed by stacking the spanning vectors column-wise

[1] otherwise if $K < N$, some portions of the space will definitely be out of the spanning vectors' reach.

[2] see Section 8.8.6 if unfamiliar with the notion of linear independence.

$$\mathbf{C} = \begin{bmatrix} | & | & \cdots & | \\ \mathbf{c}_1 & \mathbf{c}_2 & & \mathbf{c}_N \\ | & | & \cdots & | \end{bmatrix} \tag{8.5}$$

is formed by stacking the spanning vectors column-wise, and where the linear combination weights are the stacked into column vectors $\mathbf{w}_p$

$$\mathbf{w}_p = \begin{bmatrix} w_{p,1} \\ w_{p,2} \\ \vdots \\ w_{p,N} \end{bmatrix} \tag{8.6}$$

for all $p = 1, ..., P$.

To tune the weights in each $\mathbf{w}_p$ we can form an associated Least Squares cost function (as we have done multiple times previously, e.g., with linear regression in Section 5.1) that, when minimized, forces the equalities in Equation (8.4) to hold

$$g(\mathbf{w}_1, ..., \mathbf{w}_P) = \frac{1}{P}\sum_{p=1}^{P} \left\| \mathbf{C}\mathbf{w}_p - \mathbf{x}_p \right\|_2^2. \tag{8.7}$$

This Least Squares cost function can be minimized via any local optimization method. In particular we can use the first order condition (Section 3.2) in each weight vector $\mathbf{w}_p$ *independently*, with the corresponding first order system taking the form

$$\mathbf{C}^T\mathbf{C}\,\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p. \tag{8.8}$$

This is an $N \times N$ symmetric linear system that can be easily solved numerically (see e.g., Section 3.2.2 and Example 3.6). Once solved, the optimally tuned weight vector $\mathbf{w}_p^\star$ for the point $\mathbf{x}_p$ is often referred to as the *encoding* of the point over the spanning matrix $\mathbf{C}$. The actual linear combination of the spanning vectors for each $\mathbf{x}_p$ (i.e., $\mathbf{C}\mathbf{w}_p$) is likewise referred to as the *decoding* of the point.

Example 8.1 Data encoding

In the left panel of Figure 8.2 we show an $N = 2$ dimensional toy dataset centered at the origin, along with the spanning set $\mathbf{C} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ shown as two red arrows. Minimizing the Least Squares cost in Equation (8.7), in the right panel we show the *encoded* version of this data, plotted in a new space whose coordinate axes are now in line with the two spanning vectors.

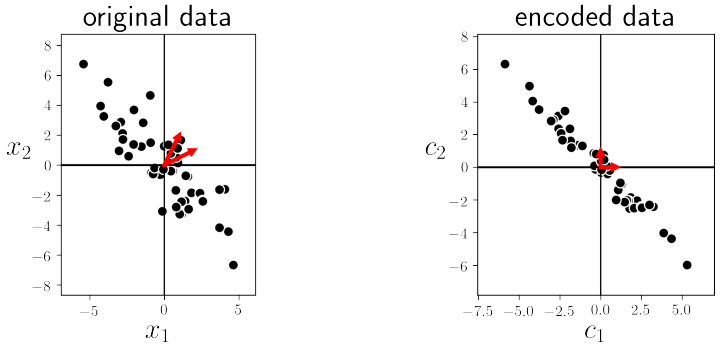

Figure 8.2 Figure associated with Example 8.1. A toy dataset (left panel) with spanning vectors shown as red arrows, and its encoding (right panel). See text for further details.

8.1.3 Perfect representation of data using fixed orthonormal spanning sets

An *orthonormal* basis or spanning set is a very special kind of spanning set whose elements (i) have unit length, and (ii) are perpendicular or orthogonal to each other. Algebraically this means that vectors belonging to an orthonormal spanning set satisfy the following condition

$$\mathbf{c}_i^T\mathbf{c}_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{8.9}$$

which can be expressed equivalently but more compactly, in terms of the spanning matrix $\mathbf{C}$, as

$$\mathbf{C}^T\mathbf{C} = \mathbf{I}_{N\times N}. \tag{8.10}$$

Because of this very special property of orthonormal spanning sets we can solve for the ideal weights $\mathbf{w}_p$ (or *encoding*) of the point $\mathbf{x}_p$ immediately, since the the first order solution in Equation (8.8) simplifies to

$$\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p. \tag{8.11}$$

In other words, encoding is enormously cheaper when our spanning set is orthonormal since there is no system of equations left to solve for, and we can get the encoding of each data point directly via a simple matrix-vector multiplication.

Substituting this form of the encoding into the set of equalities in Equation (8.4) we have

$$\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p = \mathbf{x}_p, \qquad p = 1, ..., P. \tag{8.12}$$

We call this the *autoencoder* formulae since it expresses how a point $\mathbf{x}_p$ is first *encoded* (via $\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p$) and then *decoded* back to itself ($\mathbf{C}\mathbf{w}_p = \mathbf{C}\mathbf{C}^T\mathbf{x}_p$). This is because with orthonormal spanning sets we also have that $\mathbf{C}\,\mathbf{C}^T = \mathbf{I}_{N\times N}$, and the two transformations we apply to the data, the *encoding* transformation $\mathbf{C}^T$ and the *decoding* transformation $\mathbf{C}$, are inverse operations.

8.1.4 Imperfect representation of data using fixed spanning sets

In the previous two Subsections, in order to be able to represent data perfectly, we assumed that the number of linearly independent spanning vectors K and the ambient input dimension N were identical. When $K < N$ we can no longer perfectly represent every possible data point in an input space. Instead we can only hope to *approximate*, as well as possible, our dataset as

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p, \qquad p = 1, ..., P. \tag{8.13}$$

This is analogous to Equation (8.4) except now $\mathbf{C}$ and $\mathbf{w}_p$ are an $N \times K$ matrix and a $K \times 1$ column vector, respectively, where $K < N$.

To learn the proper encodings for our data we still aim to minimize the Least Squares cost in Equation (8.7) and can still use the first order system to independently solve for each $\mathbf{w}_p$ as in Equation (8.8). Geometrically speaking, in solving the Least Squares cost we aim at finding the best K dimensional subspace on which to project our data points, as illustrated in Figure 8.3. When the *encoding* $\mathbf{w}_p$ is optimally computed for the point $\mathbf{x}_p$ its *decoding* $\mathbf{C}\mathbf{w}_p$ is precisely the projection of $\mathbf{x}_p$ onto the subspace spanned by $\mathbf{C}$. This is called a *projection* because, as illustrated in the Figure, the representation is given by projecting or dropping $\mathbf{x}_p$ perpendicularly onto the subspace.

Akin to what we saw in the previous Subsection, if our spanning set of K elements is orthonormal the corresponding formulae for each encoding vector $\mathbf{w}_p$ simplifies to Equation (8.11), and the *autoencoder* formulae shown previously in Equation (8.12) become

$$\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p \approx \mathbf{x}_p \qquad p = 1, ..., P. \tag{8.14}$$

In other words, since $K < N$ the encoding $\mathbf{C}^T$ and decoding $\mathbf{C}$ transformations are no longer quite inverse operations of one another.

8.2 The Linear Autoencoder and Principal Component Analysis

The most fundamental unsupervised learning method, known as Principal Component Analysis or PCA for short, follows directly from our discussion in the previous Section regarding fixed spanning set representations with one crucial caveat: instead of just learning the proper weights to best represent input data over a given *fixed* spanning set we *learn* a proper spanning set as well.

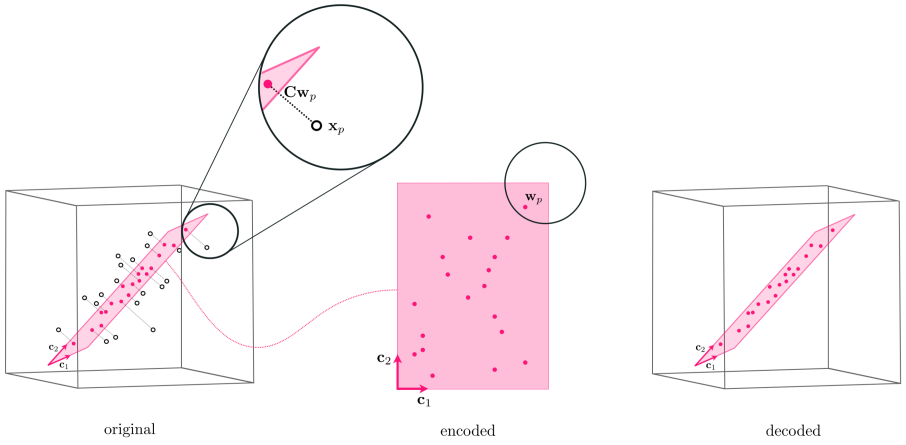

Figure 8.3 (left panel) A dataset of points $\mathbf{x}_p$ in $N = 3$ dimensions along with a linear subspace spanned by $K = 2$ vectors $\mathbf{c}_1$ and $\mathbf{c}_2$, colored in red. (middle panel) The encoding space spanned by $\mathbf{c}_1$ and $\mathbf{c}_2$, where our encoded vectors $\mathbf{w}_p$ live. (right panel) The projected or decoded versions of each data point $\mathbf{x}_p$ shown in the subspace spanned by $\mathbf{C} = [\mathbf{c}_1 \ \mathbf{c}_2]$. The decoded version of the original point $\mathbf{x}_p$ is expressed as $\mathbf{C}\mathbf{w}_p$.

8.2.1 Learning proper spanning sets

Imagine we returned to the previous Section, but instead of assuming we were given $K \leq N$ *fixed* spanning vectors over which to represent our mean-centered input as

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p \qquad p = 1, ..., P \tag{8.15}$$

we aimed to *learn* the best spanning vectors to make this approximation as tight as possible. To do this we could simply add $\mathbf{C}$ to the set of parameters of our Least Squares function in Equation (8.7) giving

$$g\left(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\|_2^2 \tag{8.16}$$

which we could then minimize to learn the best possible set of weights $\mathbf{w}_1$ through $\mathbf{w}_P$ as well as the spanning matrix $\mathbf{C}$. This Least Squares cost function, which is generally non-convex, can be properly minimized using any number of local optimization techniques including gradient descent (see Section 3.6) and coordinate descent (see Section 3.2.2).

Example 8.2 **Learning a proper spanning set via gradient descent**

In this example we use gradient descent to minimize the Least Squares cost

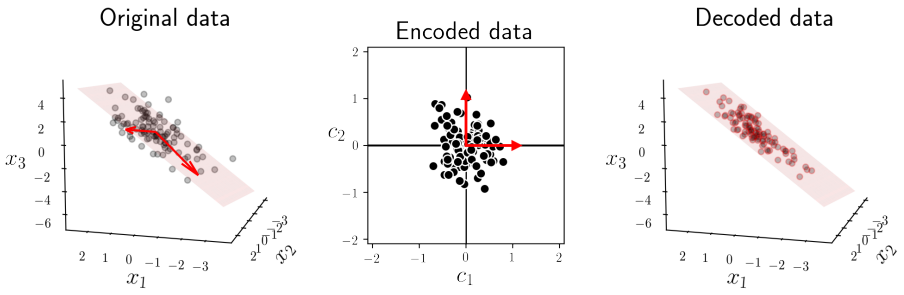

Figure 8.4 Figure associated with Example 8.2. See text for details.

in Equation (8.16) in order to learn the best $K = 2$ dimensional subspace for the mean-centered $N = 3$ dimensional dataset of $P = 100$ points shown in the left panel of Figure 8.4.

In addition to the original data, the left panel of the Figure shows the learned spanning vectors as red arrows, and corresponding subspace colored in light red. This is the very best two-dimensional subspace representation for the input data. In the middle panel we show the corresponding learned encodings $\mathbf{w}_p$ of the original input $\mathbf{x}_p$ in the space spanned by the two recovered spanning vectors. In the right panel of the Figure we show the original data space again as well as the decoded data, i.e., the projection of each original data point onto our learned subspace.

8.2.2 The linear autoencoder

As detailed in Section 8.1.4, if our K spanning vectors concatenated column-wise to form the spanning matrix $\mathbf{C}$, are orthonormal then the encoding of each $\mathbf{x}_p$ may be written simply as $\mathbf{w}_p = \mathbf{C}^T\mathbf{x}_p$. If we plug in this simple solution for $\mathbf{w}_p$ into the p^{th} summand of the Least Squares cost in Equation (8.16), we get a cost that is a function of $\mathbf{C}$ *alone*

$$g\left(\mathbf{C}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2. \tag{8.17}$$

We can think of this Least Squares as enforcing the *autoencoder* formulae shown in Equation (8.14) to hold when properly minimized, and thus it is often referred to as the *linear autoencoder*. Instead of being given an encoding/decoding scheme for each data point, by minimizing this cost function we *learn* one.

Even though we were led to the linear autoencoder by assuming our spanning matrix $\mathbf{C}$ is orthonormal, we need not constrain our minimization of Equation

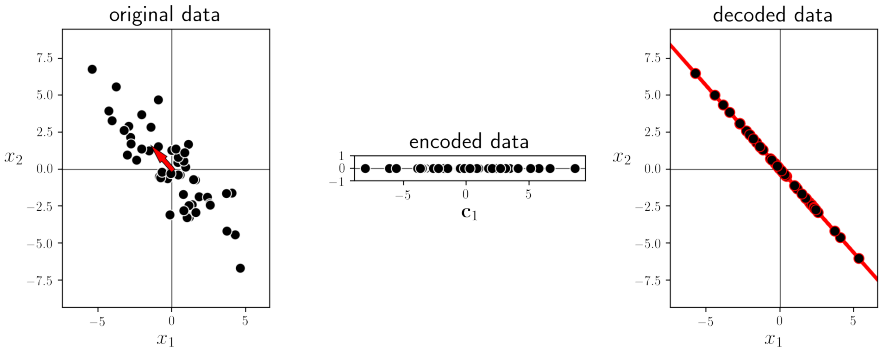

Figure 8.5 Figure associated with Example 8.3. See text for details.

(8.17) to enforce this condition because, as is shown in Section 8.7, the minima of the linear autoencoder are *always* orthonormal (see Section 8.7.1).

Example 8.3 **Learning a Linear Autoencoder using gradient descent**

In the left panel of Figure 8.5 we show a mean-centered two dimensional dataset, along with a single spanning vector (i.e., $K = 1$) *learned* to the data by minimizing the linear autoencoder cost function in Equation (8.17) using gradient descent. The optimal vector is shown as a red arrow in the left panel, the corresponding encoded data is shown in the middle panel, and the decoded data in the right panel along with the optimal subspace for the data (a line) shown in red.

8.2.3 Principal Component Analysis

The Linear Autoencoder cost in Equation (8.17) may have many minimizers, of which the set of *principal components* is a particularly important one. The spanning set of principal components always provide a consistent *skeleton* for a dataset, with its members pointing in the dataset's *largest directions of orthogonal variance*. Employing this particular solution to the Linear Autoencoder is often referred to as *Principal Component Analysis*, or PCA for short, in practice.

This idea is illustrated for a prototypical $N = 2$ dimensional dataset in Figure 8.6, where the general elliptical distribution of the data is shown in light grey. A scaled version of the first principal component of this dataset (shown as the longer red arrow) points in the direction in which the dataset is most spread out, also called its largest direction of variance. A scaled version of the second principal component (shown as the shorter of the two red arrows) points in the next most important direction in which the dataset is spread out that is *orthogonal* to the first.

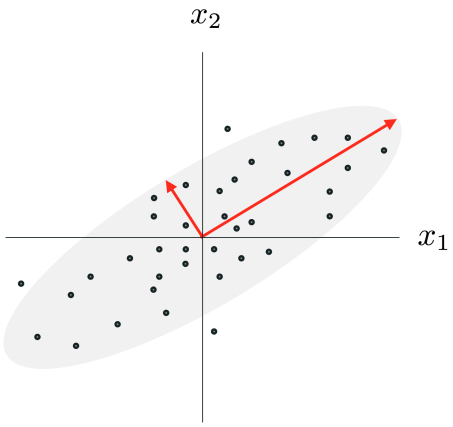

Figure 8.6 A prototypical dataset with scaled versions of its first and second principal components shown as the longer and shorter red arrows, respectively. See text for further details.

As we show in Section 8.7.2, this special orthonormal minimizer of the Linear Autoencoder is given by the eigenvectors of the so-called *covariance matrix* of the data. Denoting by $\mathbf{X}$ the $N \times P$ data matrix consisting of our P input points stacked column-wise

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_P \\ | & | & & | \end{bmatrix} \tag{8.18}$$

the covariance matrix is defined as the $N \times N$ matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T$. Denoting the eigen-decomposition (see Section 8.10) of the covariance matrix as

$$\frac{1}{P}\mathbf{X}\mathbf{X}^T = \mathbf{V}\,\mathbf{D}\,\mathbf{V}^T \tag{8.19}$$

the *principal components* are given by the orthonormal eigenvectors in $\mathbf{V}$, and the variance in each (principal component) direction is given precisely by the corresponding non-negative eigenvalues in the diagonal matrix $\mathbf{D}$.

Example 8.4 **Principal components**

In the left panel of Figure 8.7 we show the mean-centered data first displayed in Figure 8.5, along with its two principal components (pointing in the two orthogonal directions of greatest variance in the dataset) shown as red arrows. In the right panel we show the *encoded* version of the data in a space where the principal components are in line with the coordinate axes.

Example 8.5 **A warning example!**

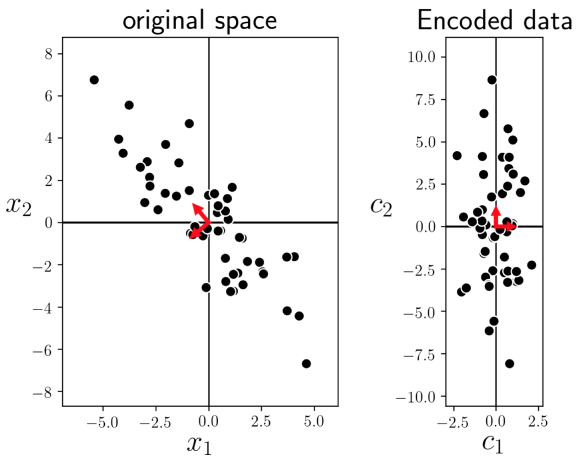

Figure 8.7 Figure associated with Example 8.4.

While PCA can technically be used to reduce the dimension of data in a predictive modeling scenario (in hopes of improving accuracy, computation time, etc.) it can cause severe problems in the case of classification. In Figure 8.8 we illustrate feature space dimension reduction via PCA on a simulated two-class dataset where the two classes are linearly separable. Because the ideal one-dimensional subspace for the data in this instance runs (almost) parallel to the ideal linear classifier, projecting the complete dataset onto this subspace completely destroys the inter-class separation. For this very reason, while it is common place to *sphere* classification data using PCA as detailed in Example 6, one needs to be extremely careful using PCA as a dimension reduction tool with classification or when the data does not natively live in or near a linear subspace.

8.2.4 Python implementation

Below we provide a `Python` implementation involved in computing the principal components of a dataset - including data centering, principal component computation, and the PCA encoding. This implementation extensively leverages `numpy`'s linear algebra submodule.

First we center the data using the short implementation below.

```
# center an input dataset X
def center(X):
    X_means = np.mean(X,axis=1)[:,np.newaxis]
    X_centered = X - X_means
    return X_centered
```

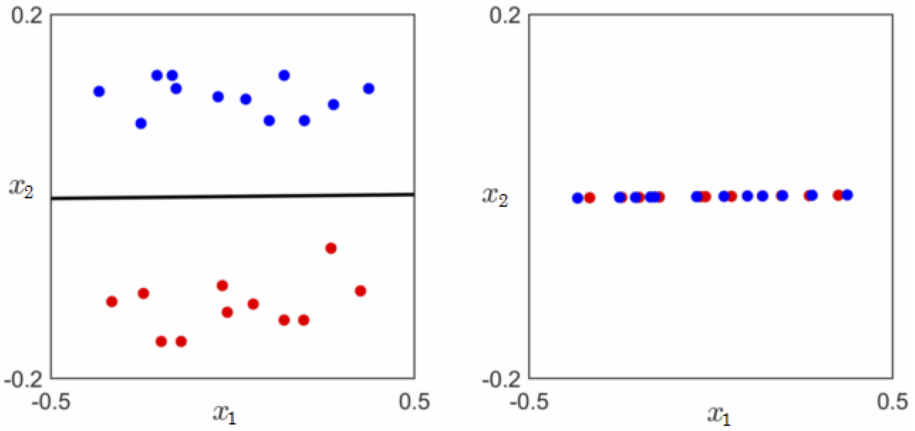

Figure 8.8 Figure associated with Example 8.5. (left panel) A toy classification dataset consisting of two linearly separable classes. The ideal one-dimensional subspace produced via PCA is shown in black. (right panel) Reducing the feature space dimension by projecting the data onto this subspace completely destroys the original separability of the data.

Next we compute the principal components of the mean-centered data.

```
1 # function for computing principal components of input dataset X
2 def compute_pcs(X,lam):
3     # create the data covariance matrix
4     P = float(X.shape[1])
5     Cov = 1/P*np.dot(X,X.T) + lam*np.eye(X.shape[0])
6
7     # use numpy function to compute eigenvectors / eigenvalues
8     D,V = np.linalg.eigh(Cov)
9     return D,V
```

Note that in practice it is often helpful to slightly *regularize* a matrix prior to computing its eigenvalues/vectors to avoid natural numerical instability issues associated with their computation. Here this means adding a small weighted identity $\lambda \mathbf{I}_{N\times N}$, where $\lambda \geq 0$ is some small value (like e.g., 10^{-5}), to the data covariance matrix prior to computing its eigenvalues/vectors. In short, in order to avoid computational trouble we typically compute principal components of the regularized covariance matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T + \lambda \mathbf{I}_{N\times N}$ instead of the raw covariance matrix itself. Thus the addition of the term `lam * np . eye(X . shape[0])` in line 5 of the implementation above.

8.3 Recommender Systems

In this Section we discuss the fundamental linear *recommender system*, a popular unsupervised learning framework commonly employed by businesses to help automatically recommend products and services to their customers. From the vantage of machine learning however, the basic recommender system detailed here is simply a slight twist on our core unsupervised learning technique: Principal Component Analysis.

8.3.1 Motivation

Recommender systems are heavily used in e-commerce today, providing customers with personalized recommendations for products and services by using a consumer's previous purchasing and rating history, along with those of similar customers. For instance, a movie provider like Netflix with millions of users and tens of thousands of movies, records users' reviews and ratings (typically in the form of a number on a scale of $1-5$ with 5 being the most favorable rating) in a large matrix such as the one illustrated in Figure 8.9. These matrices are very sparsely populated, since an individual customer has likely rated only a small portion of the movies available.

With this sort of product ratings data available, online movie and commerce sites often use the unsupervised learning technique we discuss in this Section as their main tool for making personalized recommendations to customers regarding what they might like to consume next. With the technique for producing personalized recommendations we discuss here we aim to first intelligently guess the values of missing entries in the ratings matrix. Next, in order to recommend a new product to a given user, we examine our filled-in ratings matrix for products we have predicted the user would highly rate (and thus enjoy), and recommend those.

8.3.2 Notation and modeling

With a recommender system we contiune to use our familiar notation $\{\mathbf{x}_1, ..., \mathbf{x}_P\}$ to denote input data, each of which has dimension N. In this application the point $\mathbf{x}_p$ denotes our p^{th} customer's rating vector of all N possible products available to be rated. The number of products N is likely quite large, so large that each customer only has the chance to purchase and review a very small sampling of them, making $\mathbf{x}_p$ a very sparsely populated vector (with whatever ratings user p has input into the system). We denote the index set of these non-empty entries of $\mathbf{x}_p$ as

$$\Omega_p = \left\{ (j,p) \mid j^{th} \text{ entry of } \mathbf{x}_p \text{ is filled in} \right\}. \tag{8.20}$$

Since our goal is to fill in the missing entries of each input vector $\mathbf{x}_p$ we have no

Figure 8.9 A prototypical movie rating matrix is very sparsely populated, with each user having rated only a very small number of films. In this diagram movies are listed along rows with users along columns. In order to properly recommend movies for users to watch we try to intelligently guess the missing values of this matrix, and then recommend movies that we predict users would highly rate (and therefore enjoy the most).

choice but to make assumptions about how users' tastes behave in general. The simplest assumption we can make is that every user's tastes can be expressed as a linear combination of some small set of fundamental user taste profiles. For example, in the case of movies these profiles could include the prototypical romance movie lover, prototypical comedy movie lover, action movie lover, etc. The relatively small number of such categories or user types compared to the total number of users provides a useful framework to intelligently guess missing values present in a user ratings dataset. Formally this is to say that we assume that some ideal spanning set of K fundamental taste vectors (which we can package in an $N \times K$ matrix $\mathbf{C}$) exist so that each vector $\mathbf{x}_p$ can be truly expressed as the linear combination

$$\mathbf{C}\mathbf{w}_p \approx \mathbf{x}_p, \qquad p = 1, ..., P. \tag{8.21}$$

In order to then learn both the spanning set $\mathbf{C}$ and each weight vector $\mathbf{w}_p$ we could then initially propose to minimize a Least Squares cost similar to one shown in Equation (8.16). However our input data is now *incomplete* as we only have access to the entries indexed by Ω_p for $\mathbf{x}_p$. Therefore we can only minimize that Least Squares cost over these entries, i.e.,

$$g\left(\mathbf{w}_1, ..., \mathbf{w}_P, \mathbf{C}\right) = \frac{1}{P}\sum_{p=1}^{P}\left\|\left\{\mathbf{C}\mathbf{w}_p - \mathbf{x}_p\right\}\Big|_{\Omega_p}\right\|_2^2. \tag{8.22}$$

The notation $\{\mathbf{v}\}|_{\Omega_p}$ here denotes taking only those entries of $\mathbf{v}$ in the index

set Ω_p. Because the Least Squares cost here is defined only over a select number of indicies we cannot leverage any sort of orthonormal solutions to this cost, or construct a cost akin to the Linear Autoencoder in Equation (8.17). However we can easily use gradient (see Section 3.6) and coordinate descent (see Section 3.2.2) based schemes to properly minimize it.

8.4 K-means clustering

The subject of this Section, the *K-means algorithm*, is an elementary example of another set of unsupervised learning methods called *clustering algorithms*. Unlike PCA that was designed to reduce the ambient dimension (or feature dimension) of the data space, clustering algorithms are designed to (properly) reduce the number of points (or data dimension) in a dataset, and in doing so help us better understand its structure.

8.4.1 Representing a dataset via clusters

One way to simplify a dataset is by grouping together nearby points into *clusters*. Take the following set of two-dimensional data points shown in the left panel of Figure 8.10. When you carefully examine the data shown there you can see that it naturally falls into three groups or *clusters* because you have something along the lines of a clustering algorithms built in to your brain.

In the right panel of Figure 8.10 we show a visual representation of each cluster, including each cluster's boundary drawn as a uniquely colored solid curve. We also draw the center of each cluster using a star symbol that matches the unique boundary color of each cluster. These cluster centers are often referred to in the jargon of machine learning as cluster *centroids*. The centroids here allow us to think about the dataset in the big picture sense - instead of $P = 10$ points we can think of our dataset grossly in terms of these $K = 3$ cluster centroids, as each represents a chunk of the data.

How can we describe, mathematically speaking, the clustering scenario we naturally see when we view the points/clusters like those shown in Figure 8.10?

First some notation. As in the previous Sections we will denote our set of P points generically as $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_P$. To keep things as generally applicable as possible we will also use the notation K to denote the number of clusters in a dataset (e.g., in the dataset of Figure 8.10 in $K = 3$). Because each cluster has a centroid we need notation for this as well, and we will use $\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_K$ to denote these where $\mathbf{c}_k$ is the centroid of the k^{th} cluster. Finally we will need a notation to denote the set of points that belong to each cluster. We denote the set of indices of those points belonging to the k^{th} cluster as

$$\mathcal{S}_k = \left\{p \mid \text{if } \mathbf{x}_p \text{ belongs to the } k^{th} \text{ cluster}\right\}. \tag{8.23}$$

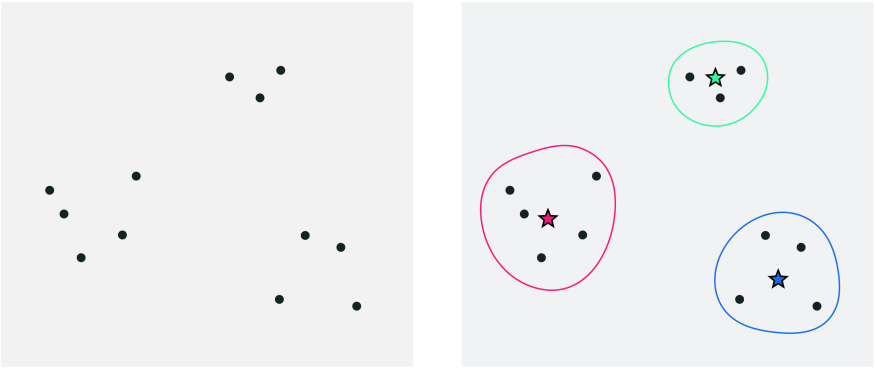

Figure 8.10 (left) A two-dimensional toy dataset with $P = 10$ data points. (right) The data shown naturally clustered into $K = 3$ clusters. Points that are geometrically close to one another belong to the same cluster, and each cluster boundary is roughly marked using a uniquely colored solid curve. Each cluster center - also called a *centroid* - is marked by a star symbol colored to match its cluster boundary.

With all of our notation in hand we can now better describe the prototype clustering scenario shown in the Figure 8.10. Suppose we have identified each cluster and its centroid 'bye eye', as depicted in the right panel of the Figure. Because the centroid denotes the center of a cluster, it seems intuitive that each one can be expressed as *the average of the points assigned to its cluster* as

$$\mathbf{c}_k = \frac{1}{|\mathcal{S}_k|} \sum_{p \in \mathcal{S}_k} \mathbf{x}_p. \tag{8.24}$$

This formula confirms the intuition that each centroid represents a chunk of the data - being the average of those points belonging to each cluster.

Next we can state mathematically an obvious and implicit fact about the simple clustering scenario visualized in Figure 8.10: that each point belongs to the cluster whose centroid it is closest to. To express this algebraically for a given point $\mathbf{x}_p$ is simply say that the point must belong to the cluster whose distance to the centroid $\left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2$ is *minimal*. In other words, the point $\mathbf{x}_p$ belongs to or is *assigned* to cluster $k^\star$ if

$$a_p = \underset{k=1,\ldots,K}{\text{argmin}} \ \left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2. \tag{8.25}$$

In the jargon of machine learning these are called cluster *assignments*.

8.4.2 Learning clusters to represent data

Of course we do not want to have to rely on our visualization abilities to identify clusters in a dataset. We want an algorithm that will do this for us automatically. Thankfully we can do this rather easily using the framework detailed above for mathematically describing clusters, the resulting algorithm being called the *K-means clustering algorithm*.

To get started, suppose we want to cluster a dataset of P points into K clusters automatically. Note here that we will fix K, and address how to properly decide on its value later.

Since we do not know where the clusters nor their centroids are located we can start off by taking a random guess at the locations of our K centroids (we have to start somewhere). This 'random guess' - our initialization - for the K centroids could be a random subset of K of our points, a random set of K points in the space of the data, or any number of other types of initializations. With our initial centroid locations decided on we can then determine cluster assignments by simply looping over our points and for each $\mathbf{x}_p$ finding its closest centroid using the formula we saw above

$$a_p = \underset{k=1,...,K}{\text{argmin}} \left\|\mathbf{x}_p - \mathbf{c}_k\right\|_2. \tag{8.26}$$

Now we have both an initial guess at our centroids and clustering assignments. With our cluster assignments in hand we can then update our centroid locations - as the average of the points recently assigned to each cluster

$$\mathbf{c}_k = \frac{1}{|\mathcal{S}_k|}\sum_{p\in\mathcal{S}_k}\mathbf{x}_p. \tag{8.27}$$

These first three steps - initializing the centroids, assigning points to each cluster, and updating the centroid locations - are illustrated in the top row of Figure 8.11 with the dataset shown above in Figure 8.10.

To further refine our centroids / clusters we can now simply repeat the above two-step process of a) re-assigning points based on our new centroid locations and then b) updating the centroid locations as the average of those points assigned to each cluster. We can halt doing so after e.g., a pre-defined number of maximum iterations or when the cluster centroids to not change location very much from one iteration to the next.

Below we provide a pseudo-code for the K-Means algorithm based on the disucssion above.

Example 8.6 **Choosing the ideal number of clusters K**

The result of the algorithm reaching poor minima can have significant impact on the quality of the clusters learned. For example in Figure 8.12 we use a 2-D

Algorithm 2 The K-means algorithm

input: dataset $\mathbf{x}_1, ..., \mathbf{x}_P$, initializations for centroids $\mathbf{c}_1, ..., \mathbf{c}_K$, and maximum number of iterations J
for $j = 1, \ldots, J$
    # Update cluster assignments
    for $p = 1, \ldots, P$
        $a_p = \underset{k=1,...,K}{\text{argmin}} \left\|\mathbf{c}_k - \mathbf{x}_p\right\|_2$
    end for
    # Update centroid locations
    for $k = 1, \ldots, K$
        denote $\mathcal{S}_k$ the index set of points $\mathbf{x}_p$ currently assigned to the k^{th} cluster
        update $\mathbf{c}_k$ via $\mathbf{c}_k = \frac{1}{|\mathcal{S}_k|} \sum_{p \in \mathcal{S}_k} \mathbf{x}_p$
    end for
end for
# Update cluster assignments using final centroids
for $p = 1, \ldots, P$
    $a_p = \underset{k=1,...,K}{\text{argmin}} \left\|\mathbf{c}_k - \mathbf{x}_p\right\|_2$
end for
output: optimal centroids and assignments

toy dataset with K=2 clusters to find. With the initial centroid positions shown in the top panel, the K-means algorithm gets stuck in a local minimum and consequently fails to cluster the data properly. A different initialization for one of the centroids however leads to a successful clustering of the data, as shown in the bottom panel of the Figure. To overcome this issue in practice we often run the algorithm multiple times with different initializations, with the best solution being one that results in the smallest value of some objective value of cluster quality.

For example, one metric for determining the best clustering from set of runs is the *average distance of each point to its cluster centroid* - called the *average intra-cluster distance*. Denoting $\mathbf{c}_{k_p}$ the final cluster centroid of the p^{th} point $\mathbf{x}_p$, then the average distance from each point to its respective centroid can be written as

$$\text{average intra-cluster distance} = \frac{1}{P}\sum_{p=1}^{P}\left\|\mathbf{x}_p - \mathbf{c}_{k_p}\right\|_2. \tag{8.28}$$

Computing this for each run of K-means we choose the final clustering that achieves the *smallest* such value as the best clustering arrangement.

Example 8.7 **Choosing the ideal number of clusters K**

To determine the optimal setting of the parameter K - i.e., the number of clusters in which to cluster the data - we typically must try a range of different values for K, run the K-means algorithm in each case, and compare the results using an appropriate metric like the average intra-cluster distance in Equation (8.28). Of course if we achieve an optimal clustering for each value of K (perhaps running the algorithm multiple times for each value of K) then the intra-cluster distance should *always go down monotonically as we increase the value of* K since we are partitioning the dataset into smaller and smaller chunks.

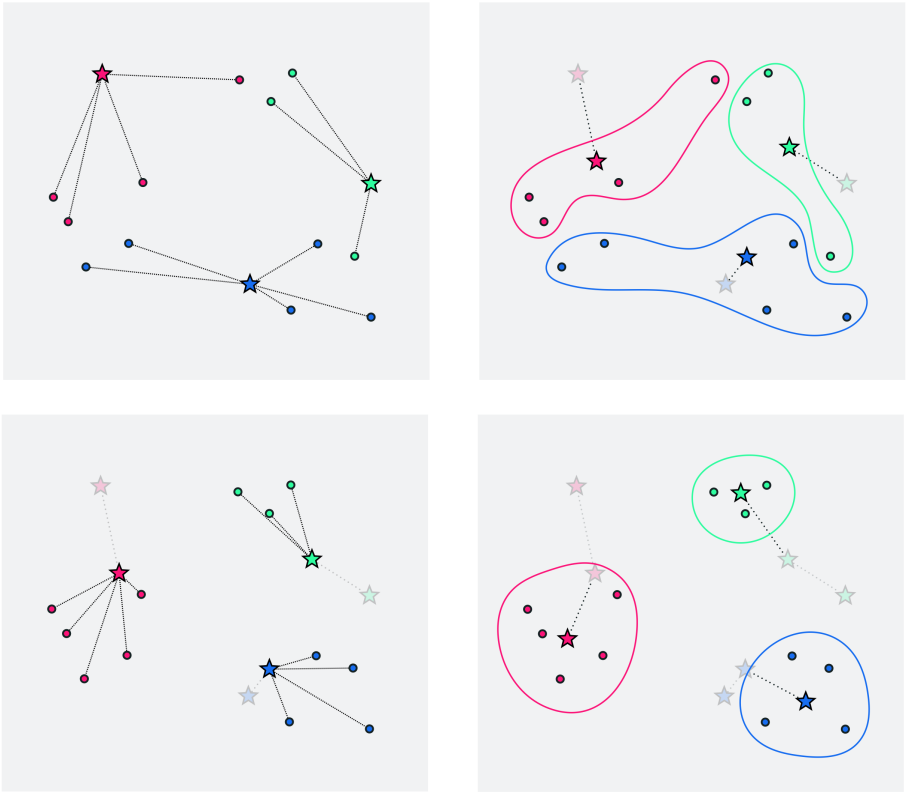

Figure 8.11 The first two iterations of the K-means algorithm illustrated using the dataset first shown in Figure 8.10. (top row) (left panel) A set of data points with random centroid initializations and assignments. (right panel) Centroid locations updated as average of points assigned to each cluster. (bottom row) (left panel) Assigning points based on updated centroid locations. (right panel) Updated location of centroids given by cluster averages.

For example, in Figure 8.13 we show the results of running 10 runs of K-means using ranging the value of K from $K = 1, \ldots, 10$ and keeping the clustering that provided the lowest intra-cluster distance for each value of K for the dataset shown in the left panel of the Figure. In the right panel we plot the best distance value attained for each value of K tried, a plot often referred to in the jargon of machine learning as a *scree plot*.

As one should expect, the intra-cluster distance decreases monotonically as we increase K. Notice, however that the scree plot above has an *elbow* at $K = 3$, meaning that increasing the number of clusters from 3 to 4 and on-wards reduces the distance value by very little. Because of this we can argue that $K = 3$ is a good choice for the number of clusters for this particular dataset (which also makes sense in this instance, since we can visualize the dataset and clearly see

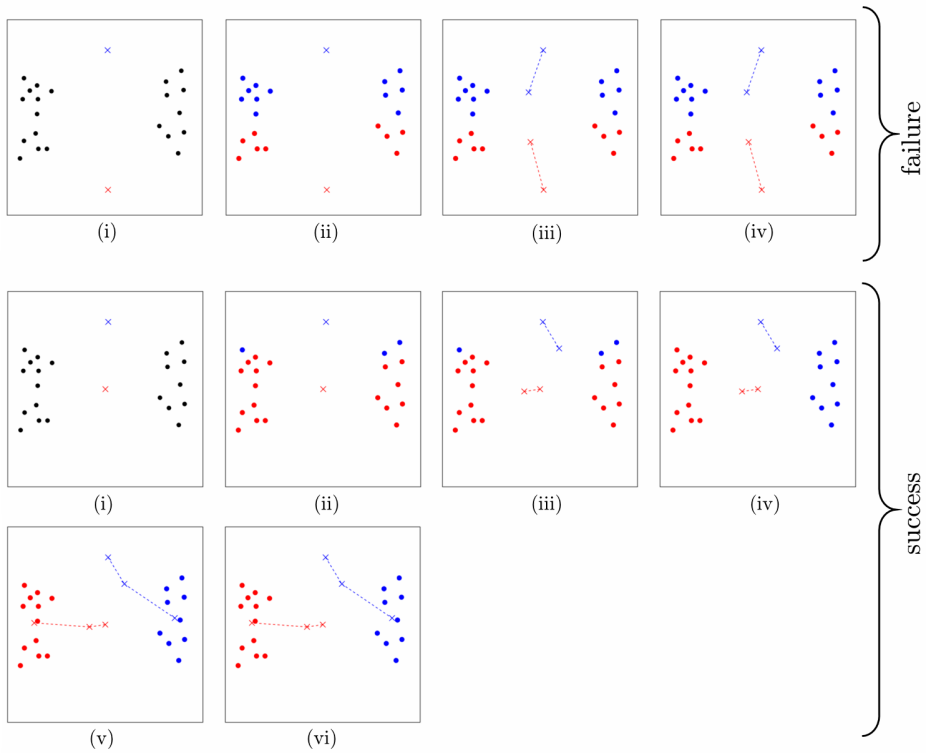

Figure 8.12 Success or failure of K-means depends on the centroids' initialization. (top) (i) two centroids are initialized, (ii) cluster assignment is updated, (iii) centroid locations are updated, (iv) no change in the cluster assignment of the data points leads to stopping of the algorithm. (bottom) (i) two centroids are initialized with the red one being initialized differently, (ii) cluster assignment is updated, (iii) centroid locations are updated, (iv) cluster assignment is updated, (v) centroid locations are updated, (vi) no change in the cluster assignment of the data points leads to stopping of the algorithm.

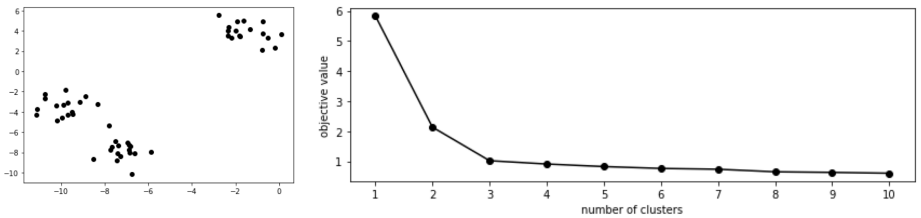

Figure 8.13 Figure associated with Example 8.7. (left panel) A dataset to cluster. (right panel) A *skree* plot. See text for further details.

it has three clusters) since any fewer clusters and the intra-cluster distance is comparatively large, while adding additional clusters does not decrease the total intra-cluster distance too much.

This illustrates the typical usage of the scree plot for deciding on an ideal number of clusters K for K-means. We compute and then plot the intra-cluster distance over a range of values for K, and pick the value at the 'elbow' of the plot. In practice this value is often chosen subjectively (by visual analysis of the skree plot) as was done here.

8.5 General Matrix Factorization techniques

In this Section we tie together the unsupervised learning methods described in this Chapter by describing them in all through the singular lens of *matrix factorization*.

8.5.1 Unsupervised learning and matrix factorization problems

If we compactly represent our P input data points by stacking them column-wise into the data matrix $\mathbf{X}$ as in Equation (8.18), we can write the Least Squares cost function in Equation (8.16) that is the basis for Principal Component Analysis (Section 8.2) compactly as

$$g\left(\mathbf{W}, \mathbf{C}\right) = \frac{1}{P}\left\|\mathbf{C}\mathbf{W} - \mathbf{X}\right\|_F^2. \tag{8.29}$$

Here $\|\mathbf{A}\|_F^2 = \sum_{n=1}^{N}\sum_{p=1}^{P} A_{n,p}^2$ is the 'Frobenius' norm, which is the analog of the squared ℓ_2 norm for matrices (see Section 8.12.3).

We can likewise express our set of desired approximations that motivate PCA - given in Equation (8.15) - and that the minimization of this cost function force to hold as closely as possible quite compactly as

$$\mathbf{C}\mathbf{W} \approx \mathbf{X}. \tag{8.30}$$

This set of desired approximations is often referred to as a *matrix factorization* since we desire to *factorize* the matrix $\mathbf{X}$ into a product of two matrices $\mathbf{C}$ and $\mathbf{W}$. This is the matrix analog of factorizing a single digit into two 'simpler' ones, e.g., as $5 \times 2 = 10$. Thus, in other words, the PCA problem can be interpreted as a basic exemplar of a *matrix factorization problem*.

PCA is not the only unsupervised learning method we can re-cast in this way. Recommender Systems, as we saw in Section 8.3, results in a cost function that closely mirrors PCA's - and so likewise closely mirrors its compact form given above. Here the only difference is that many entries of the data matrix are unknown, thus the factorization is restricted to only those values of $\mathbf{X}$. Denoting by Ω the set of indices of *known* values of $\mathbf{X}$, the matrix factorization involved in Recommender Systems takes the form

$$\{\mathbf{CW} \approx \mathbf{X}\}|_{\Omega} \tag{8.31}$$

where the symbol $\{\mathbf{V}\}|_{\Omega}$ is used to denote that we only care about entries of an input matrix $\mathbf{V}$ the index set Ω, which is a slight deviation of the PCA factorization in Equation (8.30). The corresponding Least Squares cost is similarly a slight twist on the compact PCA Least Squares cost in Equation (8.29) and is given as

$$g(\mathbf{W}, \mathbf{C}) = \frac{1}{P} \| \{\mathbf{CW} - \mathbf{X}\}|_{\Omega} \|_F^2 . \tag{8.32}$$

Note how this is simply the matrix form of the Recommender Systems cost function given earlier in Equation (8.22).

Finally, we can also easily see that K-Means (Section 8.4) falls into the same category, and can too be interpreted as a *matrix factorization problem*. We can do this by first re-interpreting our initial desire with K-Means clustering, which is our that points in the k^{th} cluster should lie close to its centroid, which may be written mathematically as

$$\mathbf{c}_k \approx \mathbf{x}_p \quad \text{for all } p \in \mathcal{S}_k \qquad k = 1, ..., K \tag{8.33}$$

where $\mathbf{c}_k$ is the centroid of the k^{th} cluster and $\mathcal{S}_k$ the set of indices of the subset of those P data points belonging to this cluster. These desired relations can be written more conveniently in matrix notation for the centroids - denoting by $\mathbf{e}_k$ the k^{th} standard basis vector (that is a $K \times 1$ vector with a 1 in the k^{th} slot and zeros elsewhere) - likewise as

$$\mathbf{C}\,\mathbf{e}_k \approx \mathbf{x}_p \quad \text{for all } p \in \mathcal{S}_k \qquad k = 1, ..., K. \tag{8.34}$$

Then, introducing matrix notation for the weights (here constrained to be standard basis vectors) and the data we can likewise write the above relations as

$$\mathbf{CW} \approx \mathbf{X} \quad \text{for all } p \in \mathcal{S}_k \qquad k = 1, ..., K. \tag{8.35}$$

where

$$\mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^{K}, \;\; p = 1, \ldots, P \tag{8.36}$$

Figure 8.14 pictorially illustrates the compactly written desired K-means relationship above for a small prototypical dataset. Note that the location of the only nonzero entry in each column of the assignment matrix $\mathbf{W}$ determines the cluster membership of its corresponding data point in $\mathbf{X}$. So, in other words, K-Means too is a matrix factorization problem (with a very particular set of constraints on the matrix $\mathbf{W}$).

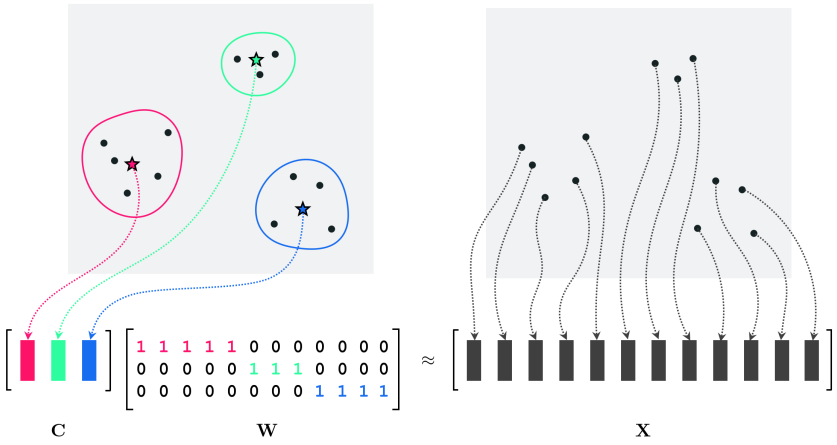

Figure 8.14 K-means clustering relations described in a compact matrix form. Cluster centroids in $\mathbf{C}$ lie close to their corresponding cluster points in $\mathbf{X}$. The p^{th} column of the assignment matrix $\mathbf{W}$ contains the standard basis vector corresponding to the data point's cluster centroid.

Having framed the desired outcome - when parameters are set optimally - the associated cost function for K-Means can then likewise be written compactly as

$$g\left(\mathbf{W},\mathbf{C}\right) = \frac{1}{P}\left\|\mathbf{CW}-\mathbf{X}\right\|_F^2. \tag{8.37}$$

subject to the constraint that $\mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^K$, $p = 1,\ldots,P$. In other words, we can interpret the K-Means algorithm described in Section 8.4 as a way of solving the constrained optimization problem

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{minimize}} \;\; \left\|\mathbf{CW}-\mathbf{X}\right\|_F^2 \\ &\text{subject to} \;\; \mathbf{w}_p \in \{\mathbf{e}_k\}_{k=1}^K, \;\; p = 1,\ldots,P. \end{aligned} \tag{8.38}$$

One can easily show that the K-Means algorithm we derived in the previous Section is also the set of updates resulting from the application of the block-coordinate descent method for solving the above K-Means optimization problem. This perspective on K-Means is particularly helpful, since in the natural derivation of K-Means shown in the previous Section K-Means is a somewhat heuristic algorithm (i.e., it is not tied to the minimization of a cost function, like every other method we discuss is). One practical consequence of this is that - previously - we had no framework in which to judge how a single run of the algorithm was progressing. Now we do. Now we know that we can treat the K-Means algorithm precisely as we do every other optimization method we discuss - as a way of minimizing a particular cost function - and can use the cost function to understand how the algorithm is functioning.

8.5.2 Further variations

We saw in Equation (8.38) how K-means can be re-cast as a constrained matrix factorization problem, one where each column $\mathbf{w}_p$ of the assignment matrix $\mathbf{W}$ is constrained to be a standard basis vector. This is done to guarantee every data point $\mathbf{x}_p$ ends up getting assigned to one (and only one) cluster centroid $\mathbf{c}_k$. There are many other popular matrix factorization problems that - from a modeling perspective - simply employ different constraints than the one given for K-Means. For example, a natural generalization of K-Means called *Sparse Coding* is a clustering-like algorithm that differs from K-Means only in that it allows assignment of data points to *multiple clusters*. Sparse coding is a constrained matrix factorization problem often written as

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{minimize}} \;\; \|\mathbf{C}\mathbf{W} - \mathbf{X}\|_F^2 \\ &\text{subject to} \quad \left\|\mathbf{w}_p\right\|_0 \leq S, \;\; p = 1, \ldots, P. \end{aligned} \tag{8.39}$$

where the K-Means constraints are replaced with constraints of the form $\left\|\mathbf{w}_p\right\|_0 \leq S$, making it possible for each $\mathbf{x}_p$ to be assigned to at most S clusters simultaneously. Recall, $\left\|\mathbf{w}_p\right\|_0$ indicates the number of nonzero entries in the vector $\mathbf{w}_p$ (see Section 8.12.1).

Besides sparsity, seeking a non-negative factorization of the input matrix $\mathbf{X}$ is another constraint sometimes put on matrices $\mathbf{C}$ and $\mathbf{W}$, giving the so-called *non-negative matrix factorization problem*

$$\begin{aligned} &\underset{\mathbf{C},\mathbf{W}}{\text{minimize}} \;\; \|\mathbf{C}\mathbf{W} - \mathbf{X}\|_F^2 \\ &\text{subject to} \quad \mathbf{C}, \mathbf{W} \geq 0 \end{aligned} \tag{8.40}$$

Non-negative matrix factorization is used predominantly in situations where data is naturally non-negative (e.g., Bag-of-Word representation of text data, pixel intensity representation of image data, etc.) where presence of negative entries hinders interpretability of learned solutions.

Table 1 shows a list of common matrix factorization problems subject to possible constraints on $\mathbf{C}$ and $\mathbf{W}$.

8.6 Exercises

Section 8.1 exercises

1. **The standard basis**

A simple example of an orthonormal spanning set is the set of N *standard*

Table 8.1 Common matrix factorization problems $\mathbf{C}\mathbf{W} \approx \mathbf{X}$ subject to possible constraints on $\mathbf{C}$ and $\mathbf{W}$

Matrix factorization problem	Constraints on C and W
Principle component analysis (PCA)	$\mathbf{C}$ is orthonormal
Recommender systems	No constraint on $\mathbf{C}$ and $\mathbf{W}$, but $\mathbf{X}$ is only partially known
K-means clustering	Each column of $\mathbf{W}$ is a standard basis vector
Sparse dictionary learning	Each column of $\mathbf{W}$ is sparse, i.e., has at most S nonzero entries
Non-negative matrix factorization	Both $\mathbf{C}$ and $\mathbf{W}$ are non-negative

basis vectors. The n^{th} element of a standard basis is a vector that consist entirely of zeros, except for a 1 in its n^{th} slot

$$(n^{th} \text{ element of the standard basis}) \quad \mathbf{c}_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{8.41}$$

This is also what we referred to as a *one-hot-encoded vector* in Section 7.5.

Simplify the formula for the optimal weight vector / encoding in Equation 8.11 when using a standard basis.

2. **Encoding data**

Repeat the experiment described in Example 8.1, reproducing the illustrations shown in Figure 8.2.

3. **Orthonormal matrices and eigenvalues**

Show that the $N \times K$ matrix $\mathbf{C}$ is orthonormal *if and only if* the non-zero eigenvalues of $\mathbf{C}\,\mathbf{C}^T$ all equal +1.

Section 8.2 exercises

4. **Non-convexity of the Linear Autoencoder**

   With $K = 1$ make a contour plot on the range $[-5, 5] \times [-5, 5]$ of the Linear Autoencoder in Equation (8.17) over the dataset shown in the left panel of Figure 8.5. How many global minima does this contour plot appear to have? Given the concept of the Linear Autoencoder and the result described in Example 8.3 describe the optimal spanning vector(s) represented by these minima and how they compare to the one shown in the left panel of Figure 8.5.

5. **Minimizing the Linear Autoencoder over a toy dataset**

   Repeat the experiment described in Example 8.3, reproducing the illustrations shown in Figure 8.5. Implement your code so that you can easily compute the gradient of the Linear Autoencoder using the `autograd`.

6. **Producing a PCA basis**

   Repeat the experiment described in Example 8.4, reproducing the illustrations shown in Figure 8.7. You may use the implementation given in Section 8.2.4 as a basis for your work.

Section 8.4 exercises

7. **Perform K-Means**

   Implement the K-Means algorithm detailed in Section 2 and apply to properly cluster the dataset shown in the left panel of Figure 8.13 using $K = 3$ cluster centroids. Visualize your results by plotting the dataset, coloring each cluster a unique color.

8. **Making a Skree-plot**

   Repeat the experiment described in Example 8.7, reproducing the illustration shown in the right panel of Figure 8.13.

8.7 Endnotes

8.7.1 Formal proof that minima of the Autoencoder are all orthonormal matrices

To show that the minima of the Linear Autoencoder in Equation 8.17 are all orthonormal matrices, we first substitute the eigenvalue decomposition (see Section **??**) $\mathbf{C}\mathbf{C}^T$ as $\mathbf{C}\mathbf{C}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T$, where $\mathbf{V}$ is an $N \times K$ orthogonal matrix of eigenvectors and $\mathbf{D}$ is a $K \times K$ diagonal matrix with all *non-negative* eigenvalues along its diagonal (since $\mathbf{C}\,\mathbf{C}^T$ is an *outer-product matrix* - see Exercise 2), for the matrix $\mathbf{C}\,\mathbf{C}^T$ in the p^{th} summand of the Linear Autoencoder

$$\left\|\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \left\|\mathbf{V}\mathbf{D}\mathbf{V}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \mathbf{x}_p^T\mathbf{V}\mathbf{D}\mathbf{D}\mathbf{V}^T\mathbf{x}_p - 2\mathbf{x}_p^T\mathbf{V}\mathbf{D}\mathbf{V}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p. \quad (8.42)$$

Introducing $\mathbf{I}_{N\times N} = \mathbf{V}^T\mathbf{V}$ in-between the inner product $\mathbf{x}_p^T\mathbf{x}_p = \mathbf{x}_p^T\mathbf{V}^T\mathbf{V}\mathbf{x}_p$, denoting $\mathbf{q}_p = \mathbf{V}\mathbf{x}_p$, and denoting $\mathbf{A}^2 = \mathbf{A}\mathbf{A}$ for any square matrix $\mathbf{A}$ we may re-write the right hand side above equivalently as

$$= \mathbf{q}_p^T\mathbf{D}\mathbf{D}\mathbf{q}_p - 2\mathbf{q}_p^T\mathbf{D}\mathbf{q}_p + \mathbf{q}_p^T\mathbf{q}_p = \mathbf{q}_p^T\left(\mathbf{D}^2 - 2\mathbf{D} + \mathbf{I}_{K\times K}\right)\mathbf{q}_p = \mathbf{q}_p^T\left(\mathbf{D} - \mathbf{I}_{K\times K}\right)^2\mathbf{q}_p \quad (8.43)$$

where the last equality follows from completing the square.

Performing this for each summand the Linear Autoencoder in Equation 8.17 can be written equivalently as

$$g\left(\mathbf{C}\right) = \sum_{p=1}^{P}\mathbf{q}_p^T\left(\mathbf{D} - \mathbf{I}_{K\times K}\right)^2\mathbf{q}_p. \quad (8.44)$$

Since the eigenvalues of $\mathbf{D}$ are non-negative it is easy to see that this quantity is minimized for $\mathbf{C}$ such that $g\left(\mathbf{C}\right) = 0$, i.e., where $\mathbf{D}_{K\times K}$ (the upper $K \times K$ portion of $\mathbf{D}$) is precisely the identity i.e., $\mathbf{D}_{K\times K} = \mathbf{I}_{K\times K}$. In other words, the Linear Autoencoder is minimized over matrices that have all non-zero eigenvalues equal to +1, and the only such matrices that have this property are *orthonoral* (see Exercise 3).

8.7.2 Formal derivation principal components

To begin the derivation of the classic *Principal Components Analysis* solution to the Linear Autoencoder in Equation 8.17 all we must do is examine one summand of the cost, under the assumption that $\mathbf{C}$ is orthonormal. Expanding the of the p^{th} summand we have

$$\left\|\mathbf{C}\,\mathbf{C}^T\mathbf{x}_p - \mathbf{x}_p\right\|_2^2 = \mathbf{x}_p^T\mathbf{C}^T\mathbf{C}\mathbf{C}\mathbf{C}^T\mathbf{x}_p - 2\mathbf{x}_p^T\mathbf{C}\mathbf{C}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p \quad (8.45)$$

and then using our assumption that $\mathbf{C}^T\mathbf{C} = \mathbf{I}_{N\times N}$ we can see that it may be re-written equivalently as

$$= -\mathbf{x}_p^T\mathbf{C}\mathbf{C}^T\mathbf{x}_p + \mathbf{x}_p^T\mathbf{x}_p = -\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2 + \left\|\mathbf{x}_p\right\|_2^2. \tag{8.46}$$

Since our aim is to *minimize* the summation of terms taking the form of the above, and the data point $\mathbf{x}_p$ is fixed and does not include the variable $\mathbf{C}$ we are minimizing with respect to, minimizing the original summand on the left is equivalent to minimizing *only the first term* $-\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2$ on the right hand side. Summing up these terms, the p^{th} of which can be written decomposed over each individual basis element we aim to learn as

$$-\left\|\mathbf{C}^T\mathbf{x}_p\right\|_2^2 = -\sum_{n=1}^{N}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2, \tag{8.47}$$

gives us the following equivalent cost function to minimize for our ideal orthonormal basis

$$g\left(\mathbf{C}\right) = -\frac{1}{P}\sum_{p=1}^{P}\sum_{n=1}^{K}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2. \tag{8.48}$$

Studying this reduced form of our Linear Autoencoder cost function we can see that it *decomposes completely over the basis vectors* $\mathbf{c}_n$, i.e., there are no terms where $\mathbf{c}_i$ and $\mathbf{c}_j$ interact when $i \neq j$. This means - practically speaking - that we can optimize our orthonormal basis *one element at a time*. Reversing the order of the summands above we can isolate each individual basis element over the entire dataset, writing above equivalently as

$$g\left(\mathbf{C}\right) = -\frac{1}{P}\sum_{n=1}^{K}\sum_{p=1}^{P}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2. \tag{8.49}$$

Now we can think about minimizing our cost function one basis element at a time. Beginning with $\mathbf{c}_1$ we first isolate only those relevant terms above, which consists of: $-\frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_1^T\mathbf{x}_p\right)^2$. Since there is a minus sign out front of this summation, this is the same as *maximizing* its negation which we denote as

$$h\left(\mathbf{c}_1\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_1^T\mathbf{x}_p\right)^2. \tag{8.50}$$

Since our basis is constrained to be orthonormal the basis element $\mathbf{c}_1$ in particular is constrained to have unit-length. Statistically speaking, the above measures the *variance of the dataset in the direction defined by* $\mathbf{c}_1$. Note: this quantity is precisely the variance because our data is assumed to have been *mean-centered*. Since we aim to maximize this quantity we can phrase our optimization in

purely sample statistical terms as well: we aim to recover the a form of the basis vector $\mathbf{c}_1$ that points in the maximum direction of variance in the dataset.

To determine the maximum value of the function above / determine the direction of maximum variance in the data we can rewrite the formula above by stacking the data points $\mathbf{x}_p$ column-wise - forming the $N \times P$ *data matrix* $\mathbf{X}$ (as shown in Equation 8.18) - giving the equivalent formula in

$$h(\mathbf{c}_1) = \frac{1}{P}\mathbf{c}_1^T\mathbf{X}\mathbf{X}^T\mathbf{c}_1 = \mathbf{c}_1^T\left(\frac{1}{P}\mathbf{X}\mathbf{X}^T\right)\mathbf{c}_1. \tag{8.51}$$

Written in this form the above takes the form of a so-called *Rayleigh Quotient* whose maximum can is expressible algebraically in closed form based on the eigenvalue / eigenvector decomposition of the matrix $\mathbf{X}\mathbf{X}^T$ (in the middle term) or likewise the matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ (in the term on the right). Because the matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ can be interpreted statistically as the *covariance matrix* of the data it is more common to use the particular algebraic arrangement on the right.

So, denoting $\mathbf{v}_1$ and d_1 the eigenvector and largest eigenvalue of $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ the maximum of the above occurs when $\mathbf{c}_1 = \mathbf{v}_1$, where $h(\mathbf{v}_1) = d_1$ - which is also the variance in this direction. In the jargon of machine learning $\mathbf{v}_1$ is referred to as the first *principal component* of the data.

With our first basis vector in hand, we can move on to determine the second element of our ideal orthonormal spanning set. Plucking out the relevant terms from above and following the same thought process we went through above results in the familiar looking function

$$h(\mathbf{c}_2) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_2^T\mathbf{x}_p\right)^2 \tag{8.52}$$

that we aim to maximize in order to recover our second basis vector. This formula has the same sort of statistical interpretation as the analogous version of the first basis vector had above - here again it calculates the variance of the data in the direction of $\mathbf{c}_2$. Since our aim here is to maximize - given that $\mathbf{c}_1$ has already been resolved and that $\mathbf{c}_1^T\mathbf{c}_2 = 0$ due to our orthonormal assumption - the statistical interpretation here is that we are aiming to find the *second* largest orthogonal direction of variance in the data.

This formula can also be written in compact vector-matrix form as $h(\mathbf{c}_2) = \mathbf{c}_2^T\left(\frac{1}{P}\mathbf{X}\mathbf{X}^T\right)\mathbf{c}_2$, and its maximum (given our restriction to an orthonormal basis implies that we must have $\mathbf{c}_1^T\mathbf{c}_2 = 0$) is again expressible in closed form in terms of the eigenvalue / eigenvector decomposition of the covariance matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T$. Here the same analysis leading to the proper form of $\mathbf{c}_1$ shows that the the maximum of the above occurs when $\mathbf{c}_2 = \mathbf{v}_2$ the eigenvector of $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ associated with its second largest eigenvalue d_2, and the variance in this direction is then $h(\mathbf{v}_2) = d_2$. This ideal basis element / direction is referred to as the *second principal component of the data*.

More generally - following the same analysis for the n^{th} member of our ideal orthonormal basis we look to maximize the familiar looking formula

$$h\left(\mathbf{c}_n\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\mathbf{c}_n^T\mathbf{x}_p\right)^2 \tag{8.53}$$

As with the first two cases above, the desire to maximize this quantity can be interpreted as the quest to uncover the n^{th} orthonormal direction of variance in the data. And following the same arguments, writing the above more compactly as $h\left(\mathbf{c}_n\right) = \mathbf{c}_n^T\left(\frac{1}{P}\mathbf{X}\mathbf{X}^T\right)\mathbf{c}_n$ etc.,, we can show that it takes the form $\mathbf{c}_n = \mathbf{v}_n$, where $\mathbf{v}_n$ is the n^{th} eigenvector of $\frac{1}{P}\mathbf{X}\mathbf{X}^T$ associated with its n^{th} largest eigenvalue d_n, and here the sample variance is expressible in terms of this eigenvalue $h\left(\mathbf{c}_n\right) = d_n$. This is learned element / direction is referred to as the n^{th} principal component of the data.

8.8 16.1 Vectors and vector operations

8.8.1 The vector

A vector is another word for an ordered listing of numbers. For example, the following

$$\left[-3,\ 4,\ 1\right] \tag{8.54}$$

is a vector of three *elements* or *entries*, also referred to as a vector of *length* or *dimension* three. In general a vector can have an arbitrary number of elements, and can contain numbers, variables, or both. For example,

$$\left[x_1,\ x_2,\ x_3,\ x_4\right] \tag{8.55}$$

is a vector of four variables. When numbers or variables are listed out horizontally (or in a row), we call the resulting vector a *row* vector. Notice, we can also list them just as well vertically (or in a column) in which case we refer to the resulting vector as a *column* vector. For example, the following

$$\begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix} \tag{8.56}$$

is a column vector of length three. We can swap back and forth between a row and column version of a vector by *transposing* it. Transposition is usually denoted by a superscript ^{T} placed just to the right and above a vector, and simply turns a row vector into an equivalent column vector and vice-versa. For example, we can write