Part II

Unsupervised Learning

6
Clustering

So far, we have considered ML models which require labeled data in order to learn. However, there is a large class of models which can learn from *unlabeled* data. From this chapter, we will begin to introduce models from this modeling paradigm, called *unsupervised learning*. In this chapter, we focus on one application of unsupervised learning, called *clustering algorithm*.

6.1 Unsupervised Learning

Unsupervised learning is a branch of machine learning which only uses unlabeled data. Examples of unlabeled data include a text corpus containing the works of William Shakespeare (Chapter 8) or a set of unlabeled images (Chapter 7). Some key goals in this setting include:

- *Learn the structure of data:* It is possible to learn if the data consists of clusters, or if it can be represented in a lower dimension.
- *Learn the probability distribution of data:* By learning the probability distribution where the training data came from, it is possible to generate synthetic data which is "similar" to real data.
- *Learn a representation for data:* We can learn a representation that is useful in solving other tasks later. With this new representation, for example, we can reduce the need for labeled examples for classification.

6.2 Clustering

Clustering is one of the main tasks in unsupervised learning. It is the process of detecting *clusters* in the dataset. Often the membership of a cluster can replace the role of a label in the training dataset. In general, clusters reveal a lot of information about the underlying structure of the data.

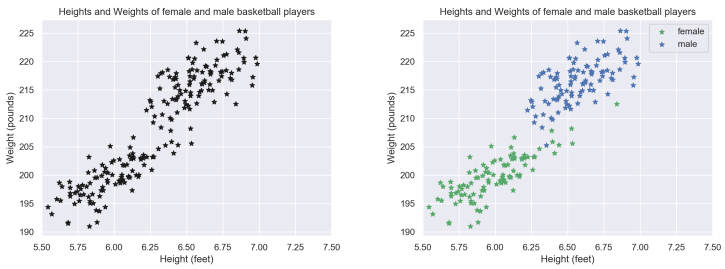

Figure 6.1: Height vs weight scatter plot of basketball players. In the plot on the right, the points in green and blue respectively correspond to female and male players.

In Figure 6.1, we see a scatter plot of measurements of height and weight of basketball players. If you look at the plot on the left, it is easy to conclude that there is a usual linear relationship between the height and the weight of the athletes. However, upon further inspection, it seems like there are two clusters of the data points, separated around the middle of the plot. In fact, this is indeed the case! If we label the dataset with the additional information of whether the data point is from a male or female athlete, the plot on the right shows something more than just the linear relationship. In practice, however, we do not always have access to this additional label. Instead, one uses clustering algorithms to find natural clusterings of the data. This raises the question of what a "clustering" is, in the first place.

Technically, any partition of the dataset $\mathcal{D}$ into k subsets $C_1, C_2, \ldots, C_k$ can be called a clustering. [1] That is,

$$\bigcup_{i=1}^{k} C_i = \mathcal{D} \quad \text{and} \quad \bigcap_{i=1}^{k} C_i = \emptyset$$

But we intuitively understand that not all partitions are a natural clustering of the dataset; our goal therefore will be to define what a "good" clustering is.

[1] Here k, the number of clusters may be given as part of the problem, or k may have to be decided upon after looking at the dataset. We'll revisit this soon.

6.2.1 *Some Attempts to Define a "Good" Cluster*

The sample data in Figure 6.1 suggests that our vision system has evolved to spot natural clusterings in two or three dimensional data. To do machine learning, however, we need a more precise definition in $\mathbb{R}^d$: specifically, for any partition of the dataset into clusters, we try to quantify the "goodness" of the clusters.

Definition 6.2.1 (Cluster: Attempt 1)**.** *A "good" **cluster** is a subset of points which are closer to each other than to all other points in the dataset.*

But this definition does not apply to the clusters in Figure 6.1. The points in the middle of the plot are far away from the points on the top right corner or the bottom left corner. So whichever cluster we assign the middle points to, they will be farther away from some

points in their assigned cluster than to some of the points on the other cluster. Ok, so that did not work. Consider the following definition.

Definition 6.2.2 (Cluster: Attempt 2)**.** *A "good" **cluster** is a subset of points which are closer to the **mean** of their own cluster than to the mean of other clusters.*

Here Mean and Variance are defined as follows:

Definition 6.2.3 (Mean and Variance of Clusters)**.** *Let C_i be one of the clusters for a dataset $\mathcal{D}$. Let $m_i = |C_i|$ denote the **cluster size**. The **mean** of the cluster C_i is*

$$\vec{\mathbf{y}}_i = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \vec{\mathbf{x}}$$

*and the **variance** within the cluster C_i is*

$$\sigma_i^2 = \frac{1}{m_i} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$$

You may notice that Definition 6.2.2 appears to be using circular reasoning: it defines clusters using the mean of the clusters, but the mean can only be calculated once the clusters have been defined. [2]

[2] Such circular reasoning occurs in most natural formulations of clustering. Look at the Wikipedia page on clustering for some other formulations.

6.3 k-Means Clustering

In this section, we present a particular partition of the dataset called the *k-means clustering*. Given k, the desired number of clusters, the k-means clustering partitions $\mathcal{D}$ into k clusters $C_1, C_2, \ldots, C_k$ so as to minimize the cost function:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2 \tag{6.1}$$

This can be seen as minimizing the average of the individual *cost* of the k clusters, where cost of C_i is $\sum_{\vec{\mathbf{x}} \in C_i} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2^2$. [3] This idea is similar in spirit to our earlier attempt in Definition 6.2.2 — we want the distance of each data point to the mean of the cluster to be small. But this method is able to circumvent the problem of circular reasoning.

[3] Notice that each cluster cost is the cluster size times the variance.

The process of finding the optimal solution for (6.1) is called the *k-means clustering problem*.

6.3.1 k-Means Algorithm

Somewhat confusingly, the most famous algorithm that is used to solve the *k-means clustering problem* is also called *k-means*. It is technically a *heuristic*, meaning it makes intuitive sense but it is not

guaranteed to find the optimum solution.[4] The following is the k-means algorithm. It is given some *initial* clustering (we discuss some choices for initialization below) and we repeat the following iteration until we can no longer improve the cost function:

[4] There is extensive research on finding near-optimal solutions to k-means. The problem is known to be NP-complete, so we believe that an algorithm that is guaranteed to produce the optimum solution on *all instances* must require exponential time.

Maintain clusters $C_1, C_2, \ldots, C_k$
For each cluster C_i, find the mean $\vec{\mathbf{y}}_i$
Initialize new clusters $C'_i \leftarrow \emptyset$
for $\vec{\mathbf{x}} \in \mathcal{D}$ **do**
  $i_x = \arg\min_i \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\|_2$
  $C'_{i_x} \leftarrow C'_{i_x} \cup \{\vec{\mathbf{x}}\}$
end for
Update clusters $C_i \leftarrow C'_i$

At each iteration, we find the mean of each current cluster. Then for each data point, we assign it to the cluster whose mean is the closest to the point, without updating the mean of the clusters. In case there are multiple cluster means that the point is closest to, we apply the tie-breaker rule that the point gets assigned to the current cluster if it is among the closest ones; otherwise, it will be randomly assigned to one of them. Once we have assigned all points to the new clusters, we update the current set of clusters, thereby updating the mean of the clusters as well. We repeat this process until there is no point that is mis-assigned.

6.3.2 *Why Does k-Means Algorithm Terminate in Finite time?*

The k-means algorithm is actually quite akin to Gradient Descent, in the sense that the iterations are trying to improve the cost.

Lemma 6.3.1. *Given a set of points $\vec{\mathbf{x}}_1, \vec{\mathbf{x}}_2, \ldots, \vec{\mathbf{x}}_m$, their mean $\vec{\mathbf{y}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$ is the point that minimizes the average squared distance to the points.*

Proof. For any vector $\vec{\mathbf{z}}$, let $C(\vec{\mathbf{z}})$ denote the sum of squared distance to the set of points. That is,

$$\begin{aligned} C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} \|\vec{\mathbf{z}} - \vec{\mathbf{x}}_i\|_2^2 &= \sum_{i=1}^{m} ((\vec{\mathbf{z}} - \vec{\mathbf{x}}_i) \cdot (\vec{\mathbf{z}} - \vec{\mathbf{x}}_i)) \\ &= \sum_{i=1}^{m} (\vec{\mathbf{z}} \cdot \vec{\mathbf{z}} - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \vec{\mathbf{x}}_i \cdot \vec{\mathbf{x}}_i) \\ &= \sum_{i=1}^{m} (\|\vec{\mathbf{z}}\|_2^2 - 2\vec{\mathbf{z}} \cdot \vec{\mathbf{x}}_i + \|\vec{\mathbf{x}}_i\|_2^2) \end{aligned}$$

To find the optimal $\vec{\mathbf{z}}$, we set the gradient ∇C to 0

$$\nabla C(\vec{\mathbf{z}}) = \sum_{i=1}^{m} (2\vec{\mathbf{z}} - 2\vec{\mathbf{x}}_i) = 0$$

which yields the solution

$$\vec{\mathbf{z}} = \frac{1}{m}\sum_{i=1}^{m} \vec{\mathbf{x}}_i$$

□

We are ready to prove the main result.

Theorem 6.3.2. *Each iteration of the k-means Algorithm 6.3.1, possibly except for the last iteration before termination, strictly decreases the total cluster cost (6.1).*

Proof. We follow the same notation as in Algorithm 6.3.1. The total cost at the end of one iteration is:

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i'\right\|_2^2$$

where $\vec{\mathbf{y}}_i'$ is the mean of the newly defined cluster C_i'. Notice that each of the cluster cost $\sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i'\right\|_2^2$ is the sum of the squared distance between a set of points $\vec{\mathbf{x}} \in C_i'$ and their mean. By Lemma 6.3.1, this sum is smaller than the sum of squared distance between the same set of points to any other point. In particular, we can compare with $\vec{\mathbf{y}}_i$, the mean of C_i before the update. That is,

$$\sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i'\right\|_2^2 \leq \sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\right\|_2^2$$

for any $1 \leq i \leq k$. If we sum over all clusters, we see that

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i'\right\|_2^2 \leq \sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\right\|_2^2$$

Now notice that the summand $\left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\right\|_2^2$ in the right hand side of the inequality is the squared distance between the point $\vec{\mathbf{x}}$ and the mean $\vec{\mathbf{y}}_i$ (before update) of the cluster C_i' that $\vec{\mathbf{x}}$ is newly assigned to. In other words, we can rewrite this term as $\left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\right\|_2^2$ and instead sum over all points $\vec{\mathbf{x}}$ in the dataset. That is,

$$\sum_{i=1}^{k} \sum_{\vec{\mathbf{x}} \in C_i'} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\right\|_2^2 = \sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\right\|_2^2$$

Finally, recall that the index i_x was defined as $i_x = \arg\min_i \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_i\right\|_2$. In particular, if j was the index of the cluster that a data point $\vec{\mathbf{x}}$ originally belonged to, then $\left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\right\|_2^2 \leq \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\right\|_2^2$. Therefore, we have the following inequality,

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\right\|_2^2 \leq \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \left\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\right\|_2^2$$

The equality holds in the inequality above if and only if when $\|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 = \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$ for each point $\vec{\mathbf{x}}$, which means that the original cluster C_j was one of the closest clusters to $\vec{\mathbf{x}}$. By the tie-breaker rule, i_x would have been set to j. This is exactly the case when the algorithm terminates immediately after this iteration since no point is reassigned to a different cluster. In all other cases, we have a strict inequality:

$$\sum_{\vec{\mathbf{x}} \in \mathcal{D}} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_{i_x}\|_2^2 < \sum_{j=1}^{k} \sum_{\vec{\mathbf{x}} \in C_j} \|\vec{\mathbf{x}} - \vec{\mathbf{y}}_j\|_2^2$$

Notice that the right hand side of the inequality is the total cost at the beginning of the iteration. □

Now we are ready to prove that the k-means algorithm is guaranteed to terminate in finite time. Since each iteration strictly reduces the cost, we conclude that the current clustering (*i.e.*, partition) will never be considered again, except at the last iteration when the algorithm terminates. Since there is only a finite number of possible partitions of the dataset $\mathcal{D}$, the algorithm must terminate in finite time.

6.3.3 *k-Means Algorithm and Digit Classification*

You might be familiar with the MNIST hand-written digits dataset. Here, each image, which depicts some digit between 0 and 9, is represented as a an 8×8 matrix of pixels and each pixel can take on a different luminosity value from 0 to 15.

We can apply k-means clustering to differentiate between images depicting the digit "1" and the digit "0." After running the model with $k = 2$ on 360 images of the two digits, we achieve the clusters in Figure 6.2. [5] Note the presence of two colored regions: a point is colored red if a hypothetical held-out data point at that location would get assigned a "0;" otherwise it is colored blue. This assignment is based on which cluster center is closer.

[5] This 2D visualization of the clusters is achieved through a technique called low dimensional representation, which is covered in Chapter 7.

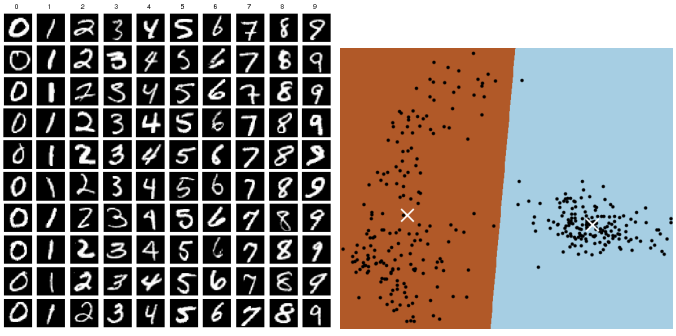

Figure 6.2: Sample images from the MNIST dataset (left) and 2D visualization of the k-means clusters differentiating between the digits "1" and "0" (right). Only two images were misclassified!

This example also shows that clustering into two clusters can be turned into a technique for binary classification — use training data to come up with two clusters; at test time, compute a ± 1 label for each data point according to which of the two cluster centers it is closer to.

6.3.4 *Implementation Detail: How to Pick the Initial Clustering*

The choice of initial clusters greatly influences the quality of the solution found by the k-means algorithm. The most naive method is to pick k data points randomly to serve as the initial cluster centers and create k clusters by assigning each data point to the closest cluster center. However, this approach can be problematic. Suppose there exists some "ground truth" clustering of the dataset. By picking the initial clusters randomly, we may end up splitting one of these ground truth clusters (*e.g.*, two initial centers are drawn from within the same ground truth cluster), and the final clustering ends up being very sub-optimal. Thus one tries to select the initial clustering more intelligently. For instance the popular k-means++ initialization procedure [6] is the following:

[6] It was invented by Arthur and Vassilvitskii in 2007.

1. Choose one center uniformly at random among all data points.
2. For each data point $\vec{\mathbf{x}}$ compute $D(\vec{\mathbf{x}})$, the distance between $\vec{\mathbf{x}}$ and the nearest center which has already been chosen.
3. Choose a new data point at random as a new center, where a point $\vec{\mathbf{x}}$ is chosen with probability proportional to $D(\vec{\mathbf{x}})^2$.
4. Repeat steps 2 and 3 until k centers have been chosen.

In COS 324, we will not expect you to understand why this is a good initialization procedure, but you may be expected to be able to implement this or similar procedures in code.

6.3.5 *Implementation Detail: Choice of k*

Above we assumed that the number of clusters k is given, but in practice you have to choose the appropriate number of clusters k.

Example 6.3.3. *Is there a value of k that guarantees an optimum cost of 0? Yes! Just set $k = n$ (i.e., each point is its own cluster). Of course, this is useless from a modeling standpoint!*

Problem 6.3.4. *Argue that the optimum cost for $k + 1$ clusters is no more than the optimum cost for k clusters.*

Note that Problem 6.3.4 only concerns the optimum cost, which as we discussed may not be attained by the k-means algorithm. Nevertheless, it does suggest that we can try various values of k and see when the cost is low enough to be acceptable.

A frequent heuristic is the *elbow method*: create a plot of the number of clusters vs. the final value of the cost as in Figure 6.3 and look for an "elbow" where the objective tapers off. Note that if the dataset is too complicated for a simple Euclidean distance cost, the data might not be easy to cluster "nicely" meaning there is no "elbow" shown on the plot.

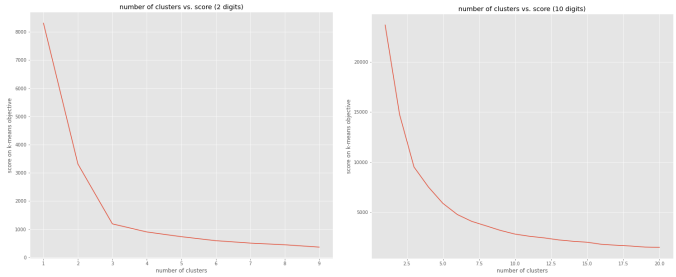

Figure 6.3: Two graphs of number of clusters vs. final value of cost. There is a distinct elbow on the left, but not on the right.

6.4 *Clustering in Programming*

In this section, we briefly discuss how to implement k-means algorithm for digit classification in Python. As usual, we use the *numpy* package to speed up computation and the *matplotlib* package for visualization. Additionally, we use the *sklearn* package to help perform the clustering.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA

# prepare dataset
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...

# define functions
def initialize_cluster_mean(X, k):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # returns Y: array of shape (k, d), each row is the center of a cluster

def assign_cluster(X, Y)
    # X: array of shape (n, d), each row is a d-dimensional data point
    # Y: array of shape (k, d), each row is the center of a cluster
    # returns loss, the sum of squared distance from each point to its
                                        assigned cluster
```

```
    # returns C: array of shape (n), each value is the index of the closest
                                        cluster

def update_cluster_mean(X, k, C):
    # X: array of shape (n, d), each row is a d-dimensional data point
    # k: number of clusters
    # C: array of shape (n), each value is the index of the closest cluster
    # returns Y: array of shape (k, d), each row is the center of a cluster

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y

def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()

# run k-means algorithm and plot the result
loss, C, Y = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

We start by importing outside packages.

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
```

The *load_digits()* method loads the MNIST digits dataset, with around 180 data points per digit. The *scale()* method linearly scales each of the data points such that the mean is 0 and variance is 1. The *PCA()* method helps visualize the MNIST digits data points, which are 64-dimensional, in the Cartesian plane (*i.e.*, $\mathbb{R}^2$). See the next Chapter 7 for details on this process.

Next we prepare the dataset by calling the *load_digits()* method.

```
X, y = load_digits(n_class=2, return_X_y=True)
X = scale(X)
X_train, X_test = ...
```

Notice that we discard the target array y because we are performing clustering, a type of unsupervised learning. If we were to perform supervised learning instead, we would need to make use of y.

Then we define the functions necessary for the k-means algorithm.

```
def initialize_cluster_mean(X, k):
    return Y

def assign_cluster(X, Y)
    return loss, C

def update_cluster_mean(X, k, C):
    return Y

def k_means(X, k, max_iters=50, eps=1e-5):
    Y = initialize_cluster_mean(X, k)

    for i in range(max_iters):
        loss, C = assign_cluster(X, Y)
        Y = update_cluster_mean(X, k, Y)

        if loss_change < eps:
            break

    return loss, C, Y
```

In practice, it is common to limit the number of cluster update iterations (*i.e.*, the parameter *max_iters*) and specify the smallest amount of loss change allowed for one iteration (*i.e.*, the constant ϵ). By terminating the algorithm once either one of the conditions is reached, we can get an approximate solution within a reasonable amount of time.

Next, take a look at the helper function used to plot the result of the k-means algorithm.

```
def scatter_plot(X, C):
    plt.figure(figsize=(12, 10))

    k = int(C.max()) + 1
    from itertools import cycle
    colors = cycle('bgrcmk')

    for i in range(k):
        idx = (C == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=next(colors))
    plt.show()
```

The *cycle()* method from the *itertools* package lets you iterate through an array indefinitely, with the index wrapping around back to the start, once it reaches the end of the array.

Now, consider the *for* loop section in the helper function above. We first use *Boolean* conditions to concisely generate a new array.

```
idx = (C == i)
```

This generates a *Boolean* array with the same length as C, where each entry is either *True/False* based on whether the corresponding entry in C is equal to i. The following code is equivalent.

```
idx = np.zeros(C.size)
for j in range(C.size):
```

```
    idx[j] = (C[j] == i)
```

We then use a technique called *Boolean masking* to extract a particular subset of rows of X.

```
X[idx, 0]
```

Notice that in place of a list of indices of rows to extract, we are indexing with the *Boolean* array we just defined. The code will extract only the rows where the *Boolean* value is *True*. For example, if the value of *idx* is *[True, False, True]*, then the code above is equivalent to

```
X[[0, 2], 0]
```

Finally, we make use of the helper functions we defined earlier to run the k-means algorithm and plot results.

```
_, C, _ = k_means(X_train, 2)
low_dim = PCA(n_components=2).fit_transform(X_train)
scatter_plot(low_dim, C)
```

The first line of this code snippet shows how we can use the _ symbol to selectively disregard individual return values of a function call. The second line of code uses the *PCA()* method to transform the 64-dimensional data *X_train* into 2-dimensional data so that we can visualize it with the *scatter_plot()* method. We will learn the details of this process in the next Chapter 7.

7
Low-Dimensional Representation

High-dimensional datasets arise in quite a few settings. This chapter concerns a phenomenon that arises frequently: the data points (*i.e.*, vectors) collectively turn out to be "approximately low rank." A running theme in this chapter is that arrays and matrices, which in introductory courses like COS 126 and COS 226 were thought of as data structures (*i.e.*, an abstraction from programming languages), are treated now as objects that we can pass through some remarkable (but simple) mathematical procedures.

If a large dataset of N vectors in $\mathbb{R}^d$ has rank k, then we can think of a natural compression method. Let U be the k-dimensional subspace spanned by the vectors, and identify k basis vectors for U. For each of the N vectors, find the k coefficients of their representation in terms of the basis vectors. Following this method, instead of specifying the N vectors using Nd real numbers, we can represent them using $k(N+d)$ real numbers, which is a big win if d is much larger than k.

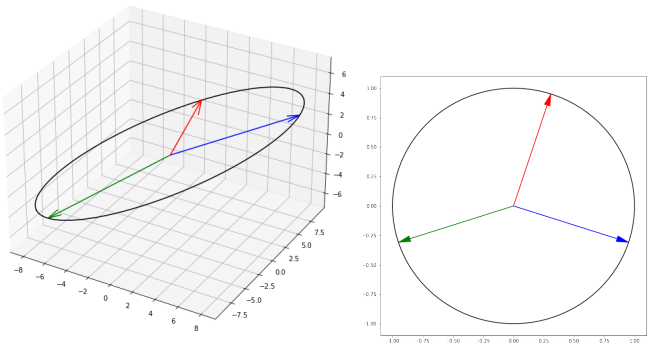

Figure 7.1: $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3 \in \mathbb{R}^3$ (left) and their 2-dimensional representations $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \widehat{\vec{\mathbf{v}}}_3 \in \mathbb{R}^2$.

Example 7.0.1. *Figure 7.1 shows three vectors* $\vec{\mathbf{v}}_1 = (3.42, -1.33, 6.94), \vec{\mathbf{v}}_2 = (7.30, 8.84, 1.95), \vec{\mathbf{v}}_3 = (-7.92, -6.37, -5.66)$ *in* $\mathbb{R}^3$. *The three vectors have rank* 2 *— they are all in the* 2*-dimensional linear subspace generated*

by $\vec{\mathbf{u}}_1 = (8, 8, 4)$ *and* $\vec{\mathbf{u}}_2 = (1, -4, 6)$. *Specifically,*

$$\vec{\mathbf{v}}_1 = 0.31\vec{\mathbf{u}}_1 + 0.95\vec{\mathbf{u}}_2$$
$$\vec{\mathbf{v}}_2 = 0.95\vec{\mathbf{u}}_1 - 0.31\vec{\mathbf{u}}_2$$
$$\vec{\mathbf{v}}_3 = -0.95\vec{\mathbf{u}}_1 - 0.31\vec{\mathbf{u}}_2$$

Therefore, we can represent these vectors in a 2*-dimensional plane, as* $\widehat{\vec{\mathbf{v}}}_1 = (0.31, 0.95), \widehat{\vec{\mathbf{v}}}_2 = (0.95, -0.31), \widehat{\vec{\mathbf{v}}}_3 = (-0.95, -0.31)$

7.1 Low-Dimensional Representation with Error

Of course, in general, high dimensional datasets are not exactly low rank. We're interested in datasets which have low-dimension representations *once we allow some error*.

Definition 7.1.1 (Low-dimensional Representation with Error)**.** *We say a set of vectors* $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_N \in \mathbb{R}^d$ *has* ***rank*** k ***with mean-squared error*** ϵ *if there exist some basis vectors* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k \in \mathbb{R}^d$ *and* N *vectors* $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \ldots, \widehat{\vec{\mathbf{v}}}_N \in \mathit{span}(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k)$ *such that*

$$\frac{1}{N}\sum_i \left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2 \leq \epsilon \tag{7.1}$$

We say that $\widehat{\vec{\mathbf{v}}}_1, \ldots, \widehat{\vec{\mathbf{v}}}_N$ *are the* ***low-rank*** *or* ***low-dimensional*** *approximation of* $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_N$*. Typically we will assume without loss of generality that the basis vectors are orthonormal (i.e., have* ℓ_2 *norm equal to* 1 *and are pairwise orthogonal).*

Definition 7.1.1 can be thought of as a *lossy* compression of the dataset of vectors since the low-dimensional representation of vectors is roughly correct, but with a bound of ϵ on the MSE. This compression view will be used in Section 7.3.

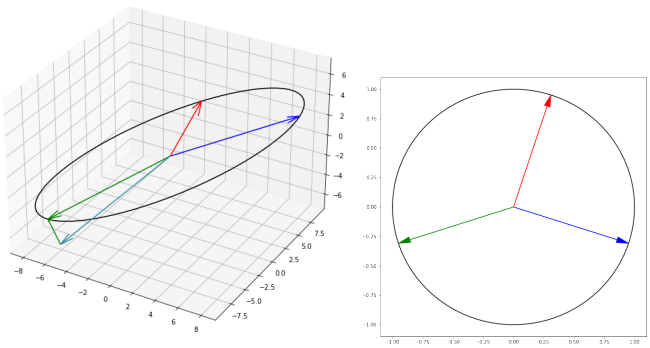

Figure 7.2: $\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \vec{\mathbf{v}}_3 \in \mathbb{R}^3$ (left) and their 2-dimensional approximations $\widehat{\vec{\mathbf{v}}}_1, \widehat{\vec{\mathbf{v}}}_2, \widehat{\vec{\mathbf{v}}}_3$ represented in the 2-dimensional subspace spanned by $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2$.

Example 7.1.2. *Figure 7.2 shows three vectors* $\vec{\mathbf{v}}_1 = (3.42, -1.33, 6.94), \vec{\mathbf{v}}_2 = (7.30, 8.84, 1.95), \vec{\mathbf{v}}_3 = (-6.00, -7.69, -6.86)$ *in* $\mathbb{R}^3$*. The three vectors have rank* 2 *with mean-squared error 2.5. If you choose the basis vectors* $\vec{\mathbf{u}}_1 = (8, 8, 4), \vec{\mathbf{u}}_2 = (1, -4, 6)$ *and the low-rank approximations* $\widehat{\vec{\mathbf{v}}}_1 = \vec{\mathbf{v}}_1, \widehat{\vec{\mathbf{v}}}_2 = \vec{\mathbf{v}}_2, \widehat{\vec{\mathbf{v}}}_3 = (-7.92, -6.37, -5.66) \in span(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2)$ *then,*

$$\frac{1}{3}\sum_i \left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2 \simeq 2.28 \leq 2.5$$

Note that the basis vectors in this example are only orthogonal and not orthonormal, but it is easy to set them as orthonormal by normalizing them.

Problem 7.1.3. *Show that if* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k \in \mathbb{R}^d$ *is any set of orthonormal vectors and* $\vec{\mathbf{v}} \in \mathbb{R}^d$ *then the vector* $\widehat{\vec{\mathbf{v}}}$ *in* $span(\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k)$ *that minimizes* $\left\|\vec{\mathbf{v}} - \widehat{\vec{\mathbf{v}}}\right\|_2^2$ *is*

$$\sum_{j=1}^{k} (\vec{\mathbf{v}} \cdot \vec{\mathbf{u}}_j)\vec{\mathbf{u}}_j \tag{7.2}$$

(Hint: If $\alpha_1, \alpha_2, \ldots, \alpha_k$ *minimize* $\left\|\vec{\mathbf{v}} - \sum_j \alpha_j \vec{\mathbf{u}}_j\right\|_2^2$ *then the gradient of this expression with respect to* $\alpha_1, \alpha_2, \ldots, \alpha_k$ *must be zero.)*

Problem 7.1.3 illustrates how to find the low-dimensional representation of the vectors, once we specify the k basis vectors. Notice that (7.2) is the *vector projection* of $\vec{\mathbf{v}}$ onto the subspace U spanned by the vectors $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k$. Therefore, the approximation error $\left\|\vec{\mathbf{v}} - \widehat{\vec{\mathbf{v}}}\right\|_2^2$ is the squared norm of the component of $\vec{\mathbf{v}}$ that is orthogonal to U. [1]

[1] Also known as the *vector rejection* of $\vec{\mathbf{v}}$ from U.

Problem 7.1.4 is only for more advanced students but all students should read its statement to understand the main point. It highlights how miraculous it is that real-life datasets have low-rank representations. It shows that generically one would expect ϵ in (7.1) to be $1 - k/n$, which is almost 1 when $k \ll n$. And yet in real life ϵ is small for fairly tiny k.

Problem 7.1.4. *Suppose the* $\vec{\mathbf{v}}_i$*'s are unit vectors* [2] *and the vectors* $\vec{\mathbf{u}}_1, \vec{\mathbf{u}}_2, \ldots, \vec{\mathbf{u}}_k$ *were the basis vectors of a random k-dimensional subspace in* $\mathbb{R}^d$*. (That is, chosen without regard to the* $\vec{\mathbf{v}}_i$*'s.) Heuristically argue that the* ϵ *one would need in* (7.1) *would be* $1 - k/n$*.*

[2] Note: the maximum possible value of ϵ when $\vec{\mathbf{v}}_i$'s are unit vectors is 1. Convince yourself!

7.1.1 Computing the Low-Dimensional Representation with Error

In Problem 7.1.3, we have already seen how to find the low-dimension representation with error, once we are given the basis vectors. All there remains is to identify a suitable value of k and find the corresponding basis vectors that will minimize the error.

There is a simple linear algebraic procedure, the *Singular Value Decomposition (SVD)*. Given a set of vectors $\vec{\mathbf{v}}_i$ and a positive integer

k, SVD can output the best orthonormal basis in sense of Definition 7.1.1 that has the lowest possible value of ϵ. In practice, we treat k as a hyperparameter and use trial and error to find the most suitable k. Problem 7.1.5 shows that the accuracy of the low-dimensional representation will decrease when we choose a smaller number of dimensions. So we are making a choice between the accuracy of the representations against how condensed our compression is.

Problem 7.1.5. *Show that as we decrease k in Definition 7.1.1, the corresponding ϵ can only increase (i.e., cannot decrease).*

Formally, SVD takes a matrix as its input; the rows of this matrix are the vector $\vec{\mathbf{v}}_i$'s. The procedure operates on this matrix to output a low-rank approximation. We discuss details in Section 20.3. To follow the rest of this chapter, you do not need to understand details of the procedure. You just need to remember the fact that the best k-dimensional representation is computable for each k. In practice, programming languages have packages that will do the calculations for you. Below is a Python code snippet that will calcuate the SVD.

```
import sklearn.decomposition.TruncatedSVD

# n * n matrix
data = ...
# prepare transform on dataset matrix "data"
svd = TruncatedSVD(n_components=k)
svd.fit(data)
# apply transform to dataset and output an n * k matrix
transformed = svd.transform(data)
```

Now we see some fun applications.

7.2 *Application 1: Stylometry*

In many cases in old literature, the identity of the author is disputed. For instance, the King James Bible (*i.e.*, the canonical English bible from the 17th century) was written by a team whose identities and work divisions are not completely known. Similarly the *Federalist Papers*, an important series of papers explicating finer points of the US government and constitution, were published in the early days of the republic with the team of authors listed as Alexander Hamilton, James Madison, and John Jay. But it was not revealed which paper was written by whom. In such cases, can machine learning help identify who wrote what?

Here we present a fun example about the books in the Wizard of Oz series. [3] L. Frank Baum was the author of the original *Wonderful Wizard of Oz*, which was a best-seller in its day and remains highly popular to this day. The publisher saw a money-making opportunity

[3] Original paper at http://dh.obdurodon.org/Binongo-Chance.pdf. A survey paper by Erica Klarreich in Science News Dec 2003: *Statistical tests are unraveling knotty literary mysteries* at http://web.mit.edu/allanmc/www/stylometrics.pdf

and convinced Baum to also write 15 follow-up books. After Baum's death the publisher managed to pass on the franchise to Ruth Plumly Thompson, who wrote many other books.

However, the last of the Baum books, *Royal Book of Oz* (RBOO), always seemed to Oz readers closer in style to Thompson's books than to Baum's. But with all the principals in the story now dead, there seemed to be no way to confirm the suspicion. Now we describe how simple machine learning showed pretty definitively that this book was indeed written by Ruth Plumly Thompson. The main idea is to represent the books vectors in some way and then find their low-rank representations.

Figure 7.3: *Royal Book of Oz* (1921). Cover image from https://en.wikipedia.org/wiki/The_Royal_Book_of_Oz

The key idea is that different authors use English words at different frequencies. Surprisingly, the greatest difference lies in frequencies of *function words* such as WITH, HOWEVER, UPON, rather than fancy vocabulary words (the ones found on your SAT exam).

Example 7.2.1. *Turns out Alexander Hamilton used* UPON *about* 10 *times more frequently than James Madison. We know this from analyzing their individual writing outside their collaboration on the Federalist Papers. Using these kinds of statistics, it has been determined that Hamilton was the principal author or even the sole author of almost all of the Federalist Papers.*

The statistical analysis of the Oz books consisted of looking at the frequencies of 50 function words. All Oz books except RBOO were divided into text blocks of 5000 words each. For each text block, the frequency (*i.e.*, number of occurrences) of each function word was computed, which allows us to represent the block as a vector in $\mathbb{R}^{50}$. There were 223 text blocks total, so we obtain 223 vectors in $\mathbb{R}^{50}$.

the (6.7%)	with (0.7%)	up (0.3%)	into (0.2%)	just (0.2%)
and (3.7%)	but (0.7%)	no (0.3%)	now (0.2%)	very (0.2%)
to (2.6%)	for (0.7%)	out (0.3%)	down (0.2%)	where (0.2%)
a/an (2.3%)	at (0.6%)	what (0.3%)	over (0.2%)	before (0.2%)
of (2.1%)	this/these (0.5%)	then (0.3%)	back (0.2%)	upon (0.1%)
in (1.3%)	so (0.5%)	if (0.3%)	or (0.2%)	about (0.1%)
that/those (1.0%)	all (0.5%)	there (0.3%)	well (0.2%)	after (0.1%)
it (1.0%)	on (0.5%)	by (0.3%)	which (0.2%)	more (0.1%)
not (0.9%)	from (0.4%)	who (0.3%)	how (0.2%)	why (0.1%)
as (0.7%)	one/ones (0.3%)	when (0.2%)	here (0.2%)	some (0.1%)

Figure 7.4: The top 50 most frequently used function words in the Wizard of Oz series. Their occurrences were counted in 223 text blocks. Figure from Binongo's paper.

Then we compute a rank 2 approximation of these 223 vectors.

Figure 7.5 shows the scatter plot in the 2-dimensional visualization. The two axes correspond to the two basis vectors we found for the rank 2 approximation. It becomes quickly clear that the vectors from the Baum books are in a different part of the space than those from the Thompson books. It is also clear that RBOO vectors fall in the

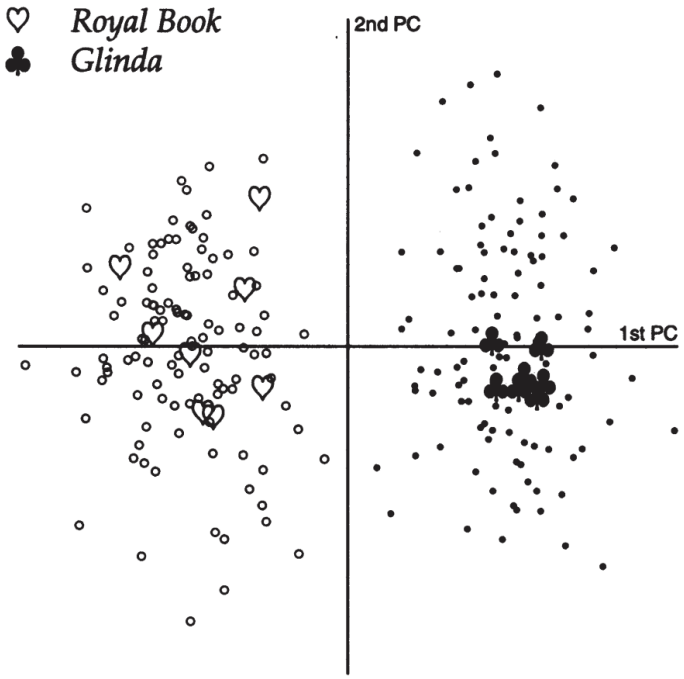

Figure 7.5: Rank-2 visualization of the 223 text block vectors from books of Oz. The dots on the left correspond to vectors from Oz books known to be written by Ruth Plumly Thompson. The hearts on the left correspond to vectors from RBOO. The ones on the right correspond to ones written by L. Frank Baum. Figure from Binongo's paper.

same place as those from other Thompson books. Conclusion: *Ruth Plumly Thompson was the true author of Royal Book of Oz!*

By the way, if one takes the non-Oz writings of Baum and Thompson and plot their vectors in the 2D-visualization in Figure 7.6, they also fall on the appropriate side. So the difference in writing style came across clearly even in non-Oz books!

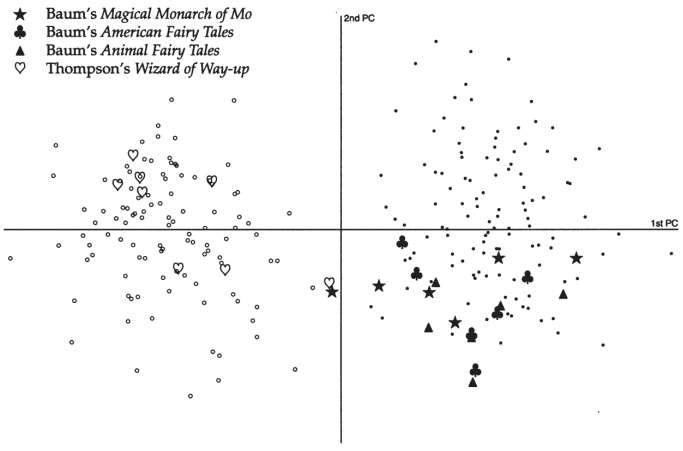

Figure 7.6: Rank-2 visualization of text block vectors from books written by Baum and Thompson outside of the Oz series. Figure from Binongo's paper.

7.3 *Application 2: Eigenfaces*

This section uses the *lossy compression* viewpoint of low-rank representations. As you may remember from earlier computer science courses (*e.g.*, Seam Carver from COS 226), images are vectors of pixel values. In this section, let us only consider grayscale (*i.e.*, B&W) images where each pixel has an integer value in $[-127, 127]$. -127 corresponds to the pixel being pitch black; 0 corresponds to middle gray; and 127 corresponds to total white. We can also reorganize the entries to form a single vector:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \rightarrow (a_{11}, a_{12}, \cdots, a_{1n}, a_{21}, \cdots, a_{2n}, \cdots, a_{mn})$$

Once we have this vectorized form of an image, it is possible to perform linear algebraic operations on the vectors. For example, we can take 0.3 times the first image and add it to -0.8 times the second image, etc. See Figure 7.7 for some of these examples.

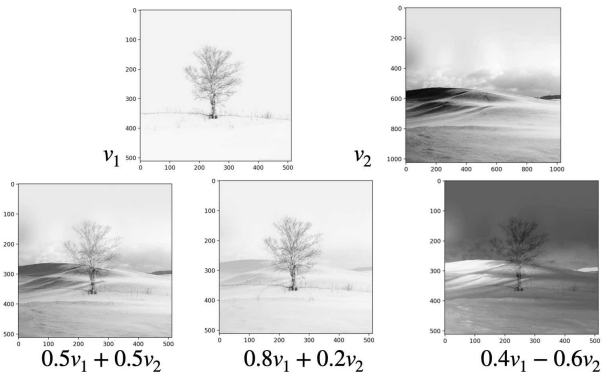

Figure 7.7: Example of linear algebra on images.

Eigenfaces was an idea for face recognition [4]. The dataset in this lecture is from a classic Olivetti dataset from 1990s. Researchers took images of people facing the camera in good light, downsized to 64×64 pixels. This makes them vectors in $\mathbb{R}^{4096}$. Now we can find a 64-rank approximation of the vectors using procedures we will explore in more detail in Section 20.3.

Figure 7.8 shows four basis vectors in the low-rank approximation of the images. The first image looks like a generic human with a ill-defined nose and lips; the second image looks like having glasses and a wider nose; the third image potentially looks like a female face; the fourth image looks like having glasses, a moustache, and a beard. All images in the dataset can be approximated as a linear

[4] L. Sirovich; M. Kirby (1987). *Low-dimensional procedure for the characterization of human faces*. Journal of the Optical Society of America.

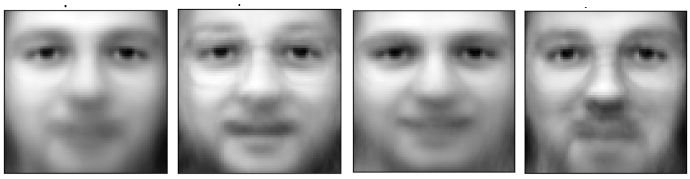

Figure 7.8: Some basis vectors (which turn out to be face-like) in the low-rank approximation of the Olivetti dataset.

combination of 128 of these basis images, and the approximations are surprisingly accurate. Figure 7.9 shows four original images of the dataset, compared with their 64-rank approximations and 128-rank approximations.

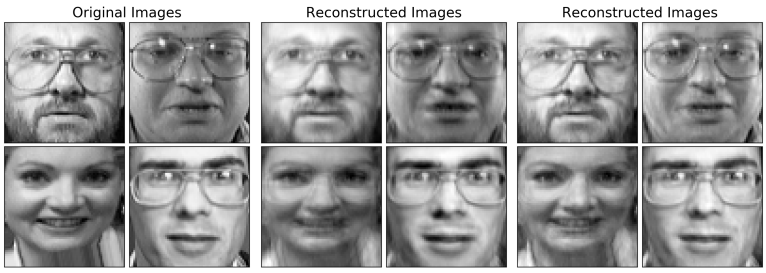

Figure 7.9: 4 original images in the Olivetti dataset (left), compared with their 64-rank approximations (middle) and 128-rank approximations (right).

From Figure 7.9, we also see that the approximations are more accurate when the corresponding value of k is larger. In fact, Figure 7.10 shows the average value of $\frac{\left\|\vec{\mathbf{v}}_i - \widehat{\vec{\mathbf{v}}}_i\right\|_2^2}{\left\|\vec{\mathbf{v}}_i\right\|_2^2}$ as a function of the rank of the approximation. Note that this value roughly represents the *fraction of* $\vec{\mathbf{v}}$ *lost in the approximation*. It can be seen that the error is a decreasing function in terms of k. [5] However, note that doing machine learning — specifically face recognition — on low-rank representations is computationally more efficient particularly because the images are compressed to a lower dimension. With a smaller value of k, we can improve the speed of the learning.

[5] This was also explored in Problem 7.1.5

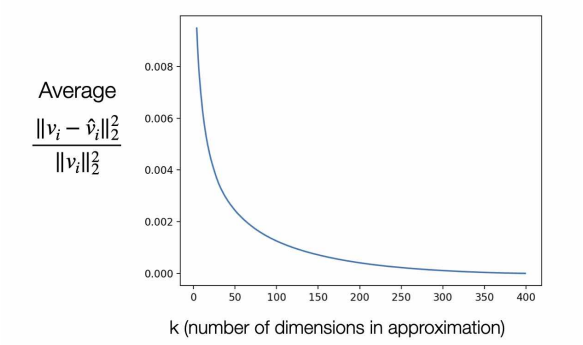

Figure 7.10: What fraction of norm of the image is not captured in the low-dimensional representation, plotted versus the rank k.

8
n-Gram Language Models

In this chapter, we continue our investigation into unsupervised learning techniques and now turn our attention to language models. You may have heard of natural language processing (NLP) and models such as GPT-3 in the news lately. The latter is quite impressive, being able to write and publish its own opinion article on a reputable news website! [1] While most of these models are trained using state-of-the-art deep learning techniques which we will discuss later on in this text, this chapter explores a key idea, which is to view language as the output of a probabilistic process, which leads to an interesting measure of the "goodness" of the model. Specifically, we will investigate the so-called *n-gram language model*.

[1] The full piece can be found at https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3

8.1 Probabilistic Model of Language

Classical linguistics focused on the syntax or the formal grammar of languages. The linguists believed that a language can be modeled by a set of sentences, constructed from a finite set of vocabularies and a finite set of grammatical rules. But this approach in language modeling had limited success in machine learning.

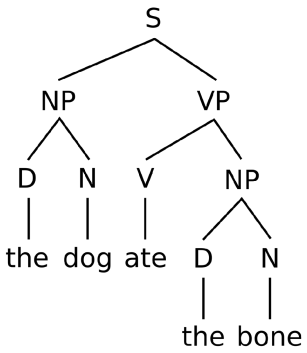

Figure 8.1: An example of a syntax tree of an English sentence.

Instead, the approach of machine learning in language models,

pioneered by Claude Shannon, has been to learn the *distribution* of pieces of text.

In other words, the model assigns a probability to all conceivable finite pieces of English text (even those that have not yet been spoken or written). For example, the sentence "how can I help you" will be assigned some probability, most likely larger than the probability assigned to the sentence "can I how you help." Note that we don't expect to find a "correct" model; all models found to date are approximations. But even an approximate probabilistic model can have interesting uses, such as the following:

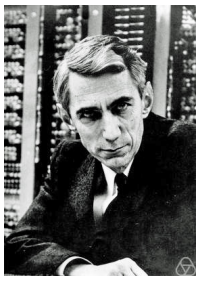

Figure 8.2: Claude Shannon, inventor of the n-gram language model in https://languagelog.ldc.upenn.edu/myl/Shannon1950.pdf. Picture from https://en.wikipedia.org/wiki/Claude_Shannon.

1. *Speech recognition:* A machine processes a recording of a human speech that sounds somewhere between "I ate a cherry" and "eye eight a Jerry." If the model assigns a higher probability score to the former, speech recognition can still work in this instance.

2. *Machine translation:* "High winds tonight" should be considered a better translation than "large winds tonight."

3. *Context sensitive spelling correction:* We can compare the probabilities of sentences that are similar to the following sentence — "Their are problems wit this sentence." — and output the corrected version of the sentence.

4. *Sentence completion:* We can compare the probabilities of sentences that will complete the following phrase — "Please turn off your ..." — and output the one with the highest probability.

8.2 *n-Gram Models*

Say we are in the middle of the process of assigning a probability distribution over all English sentences of length 5. We want to find the probability of the sentence "I love you so much." If we let X_i be the random variable that represents the value of the i-th word, the probability we are looking for is the joint probability

$$\Pr[X_1 = "I", X_2 = "love", X_3 = "you", X_4 = "so", X_5 = "much"] \quad (8.1)$$

By the Chain Rule, we can split this joint probability into the product of a marginal probability and four conditional probabilities:

$$\begin{aligned}(8.1) = &\Pr[X_1 = "I"] \\ &\times \Pr[X_2 = "love" \mid X_1 = "I"] \\ &\times \Pr[X_3 = "you" \mid X_1 = "I", X_2 = "love"] \\ &\times \Pr[X_4 = "so" \mid X_1 = "I", X_2 = "love", X_3 = "you"] \\ &\times \Pr[X_5 = "much" \mid X_1 = "I", X_2 = "love", X_3 = "you", X_4 = "so"]\end{aligned} \quad (8.2)$$

If we estimate all components of the product in (8.2), we will be able to estimate the joint probability (8.1).

Now consider the *bigram model*, which has the following two assumptions:

1. The probability of a word is only dependent on the immediately previous word.
2. That probability does not depend on the position of the word in the sentence.

The first assumption says that, for example, the conditional probability

$$\Pr[X_3 = \text{"}you\text{"} \mid X_1 = \text{"}I\text{"}, X_2 = \text{"}love\text{"}]$$

can be simplified as

$$\Pr[X_3 = \text{"}you\text{"} \mid X_2 = \text{"}love\text{"}]$$

The second assumption says that

$$\Pr[X_3 = \text{"}you\text{"} \mid X_2 = \text{"}love\text{"}] = \Pr[X_{i+1} = \text{"}you\text{"} \mid X_i = \text{"}love\text{"}]$$

for any $1 \leq i \leq 4$. We abuse notation and denote any of these probabilities as $\Pr[\text{"}you\text{"} \mid \text{"}love\text{"}]$.

Applying these assumptions to (8.2), we can simplify it as

$$\begin{aligned}(8.1) = \Pr[\text{"}I\text{"}] &\times \Pr[\text{"}love\text{"} \mid \text{"}I\text{"}] \times \Pr[\text{"}you\text{"} \mid \text{"}love\text{"}] \\ &\times \Pr[\text{"}so\text{"} \mid \text{"}you\text{"}] \times \Pr[\text{"}much\text{"} \mid \text{"}so\text{"}] \end{aligned} \tag{8.3}$$

Now we are going to estimate each component of (8.3) from a large corpus of text. The estimation for the marginal probability of the word "I" is given as

$$\Pr[\text{"}I\text{"}] \approx \frac{\text{Count("I")}}{\text{total number of words}} \tag{8.4}$$

where Count refers to the number of occurrences of the word in the text. In other words, this is the proportion of the occurrence of the word "I" in the entire corpus. Similarly, we can estimate the conditional probability of the word "love" given its previous word is "I" as

$$\Pr[\text{"}love\text{"} \mid \text{"}I\text{"}] \approx \frac{\text{Count("I love")}}{\sum_w \text{Count("I" + } w\text{)}} \tag{8.5}$$

where in the denominator, we sum over all possible words in the dictionary. This is the proportion of the word "love" occurring immediately after the word "I" out of every time some word w in the dictionary occurring immediately after the word "I." [2] In general, we

[2] Notice that there is no word occurring immediately after the word "I" when "I" is at the end of the sentence in the training corpus. Therefore, the denominator in (8.5) is equal to the Count of "I" minus the Count of "I" at the end of a sentence. This is not necessarily the case when we introduce the sentence stop tokens in Section 8.3.

can estimate the following conditional probability as

$$\Pr[w_{i+1} \mid w_i] \approx \frac{\text{Count}(w_i w_{i+1})}{\sum_w \text{Count}(w_i w)} \tag{8.6}$$

where w_i is the i-th word of the sentence. Once we calculate these estimates from the corpus, we are able to define the probability of the sentence "I love you so much."

8.2.1 Defining n-Gram Probabilities

We can extend the example above to a more general setting. Now we want to define the probability distribution over all sentences of length k (grammatical or not). Say we want to find the joint probability of the sentence $w_1 w_2 \dots w_k$ where w_i is the i-th word of the sentence. We will employ an n-gram model which has two assumptions:

1. The probability of a word is only dependent on the immediately previous $n-1$ words. [3]
2. That probability does not depend on the position of the word in the sentence.

[3] If $n = 1$, the model is called a *unigram model*, and the probability is not dependent on any previous word. When $n = 2$ and $n = 3$, the model is respectively called a *bigram* and a *trigram model*.

By a similar logic from the earlier example, we abuse notation and denote the joint probability of the sentence $w_1 w_2 \cdots w_k$ as $\Pr[w_1 w_2 \dots w_k]$; the marginal probability of the first word being w_1 as $\Pr[w_1]$; and so on. We can apply the Chain Rule again to define the n-gram model.

Definition 8.2.1 (n-Gram Model)**.** *An n-gram model assigns the following probability to the sentence $w_1 w_2 \dots w_k$ if $n > 1$:* [4]

$$\begin{aligned}\Pr[w_1 w_2 \dots w_k] &= \Pr[w_1] \Pr[w_2 \mid w_1] \cdots \Pr[w_k \mid w_1 w_2 \dots w_{k-1}] \\ &= \Pr[w_1] \times \prod_{i=2}^{k} \Pr[w_i \mid w_1 \dots w_{i-1}] \\ &= \Pr[w_1] \times \prod_{i=2}^{k} \Pr[w_i \mid w_{\max(1,i-n+1)} \dots w_{i-1}] \end{aligned} \tag{8.7}$$

[4] $\max(1, i-n+1)$ in the third line is to ensure that we access the correct indices for the first $n-1$ words, where there are less than $n-1$ previous words to look at.

and the following probability if $n = 1$:

$$\Pr[w_1 w_2 \dots w_k] = \prod_{i=1}^{k} \Pr[w_i] \tag{8.8}$$

where the n-gram probabilities are estimated from a training corpus as the following

$$\Pr[w_i] \approx \frac{\textit{Count}(w_i)}{\textit{total number of words}}$$

$$\Pr[w_j \mid w_i \dots w_{j-1}] \approx \frac{\textit{Count}(w_i \dots w_{j-1} w_j)}{\sum_w \textit{Count}(w_i \dots w_{j-1} w)}$$

This defines the "best" possible probabilistic model in terms of the Maximum Likelihood Principle from Subsection 4.2.1. [5] We now turn to the following example.

[5] We will prove this for $n = 1$ later.

Example 8.2.2. *We investigate a cowperson language which has two words in the dictionary: $\{Yee, Haw\}$. Suppose the training corpus is given as "Yee Haw Haw Yee Yee Yee Haw Yee." Then the unigram probabilities can be estimated as*

$$\Pr[\text{"}Yee\text{"}] = \frac{5}{8} \qquad \Pr[\text{"}Haw\text{"}] = \frac{3}{8}$$

We can also create the bigram frequency table as in Table 8.1 and we normalize the rows of the bigram frequency table to get the bigram probability table in Table 8.2.

previous \ next	"Yee"	"Haw"	Total
"Yee"	2	2	4
"Haw"	2	1	3

Table 8.1: Bigram frequency table of the cowperson language.

previous \ next	"Yee"	"Haw"	Total
"Yee"	2/4	2/4	1
"Haw"	2/3	1/3	1

Table 8.2: Bigram probabilty table of the cowperson language.

From Table 8.2, we get the following bigram probabilities:

$$\Pr[\text{"}Yee\text{"} \mid \text{"}Yee\text{"}] = \frac{2}{4} \qquad \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] = \frac{2}{4}$$
$$\Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] = \frac{2}{3} \qquad \Pr[\text{"}Haw\text{"} \mid \text{"}Haw\text{"}] = \frac{1}{3}$$

Then by the bigram model, the probability that we see the sentence "Yee Haw Yee" out of all sentences of length 3 can be calculated as

$$\Pr[\text{"}Yee\text{"}] \times \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] \times \Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] = \frac{5}{8} \times \frac{2}{4} \times \frac{2}{3} \simeq 0.21$$

8.2.2 *Maximum Likelihood Principle*

Recall the Maximum Likelihood Principle introduced in Subsection 4.2.1. It gave a way to measure the "goodness" of a model with probabilistic outputs.

Now we formally prove that the estimation methods given in Definition 8.2.1 satisfy the Maximum Likelihood Principle for the $n = 1$ case. A probabilistic model is "better" than another if it assigns more probability to the actual outcome. Here, the actual outcome is the training corpus, which also consists of words. So let us denote

the training corpus as a string of words $w_1 w_2 \dots w_T$. By definition, a unigram model will assign the probability

$$\Pr[w_1 w_2 \dots w_T] = \prod_{i=1}^{T} \Pr[w_i] \tag{8.9}$$

to this string. Remember that each of the w_i's are a member of a finite set of dictionary words. If we let V be the size of the dictionary, then the model is defined by the choice of V values, the probabilities we assign to each of the dictionary words. Let p_i be the probability that we assign to the i-th dictionary word, and let n_i be the number of times that the i-th dictionary word appears in the training corpus. Then (8.9) can be rewritten as

$$\Pr[w_1 w_2 \dots w_T] = \prod_{i=1}^{V} p_i^{n_i} \tag{8.10}$$

We want to maximize this value under the constraint $\sum_{i=1}^{V} p_i = 1$. A solution to this type of a problem can be found via the Lagrange multiplier method. We will illustrate with an example.

Example 8.2.3. *We revisit the cowperson language from Example 8.2.2. Here $V = 2$ and $T = 8$. Let $p_1 = \Pr[\text{"Yee"}]$ and $p_2 = \Pr[\text{"Haw"}]$. Then the probability assigned to the training corpus by the unigram model is*

$$\Pr[\text{"Yee Haw Haw Yee Yee Yee Haw Yee"}] = p_1^5 p_2^3$$

We want to maximize this value under the constraint $p_1 + p_2 = 1$. Therefore, we want to find the point where the gradient of the following is zero.

$$f(p_1, p_2) = p_1^5 p_2^3 + \lambda(p_1 + p_2 - 1)$$

for some λ. The gradients are given as

$$\frac{\partial f}{\partial p_1} = 5p_1^4 p_2^3 + \lambda \qquad \frac{\partial f}{\partial p_2} = 3p_1^5 p_2^2 + \lambda$$

From $5p_1^4 p_2^3 + \lambda = 3p_1^5 p_2^2 + \lambda = 0$, we get $\frac{p_1}{p_2} = \frac{5}{3}$. Combined with the fact that $p_1 + p_2 = 1$, we get the optimal solution $p_1 = \frac{5}{8}$ and $p_2 = \frac{3}{8}$.

Problem 8.2.4. *Following the same Lagrange multiplier method as in Example 8.2.3, verify that the heuristic solution $p_i = \frac{n_i}{T}$ (the empirical frequency) is the optimal solution that maximizes (8.10) under the constraint $\sum_{i=1}^{V} p_i = 1$.*

8.3 Start and Stop Tokens

In this section, we present a convention that is often useful: start token $\langle s \rangle$ and stop token $\langle /s \rangle$. They signify the start and the end

of each sentence in the training corpus. They are a special type of vocabulary item that will be augmented to the dictionary, so you will want to pay close attention to the way they contribute to the vocabulary size, number of words, and the n-gram probabilities. Also, by introducing these tokens, we are able to define a probability distribution over all sentences of finite length, not just a given length of k. For the sake of exposition, we will only consider the bigram model for most parts of this section.

8.3.1 Re-estimating Bigram Probabilities

Consider the cowperson language again.

Example 8.3.1. *The training corpus "Yee Haw Haw Yee Yee Yee Haw Yee" actually consists of three different sentences: (1) "Yee Haw," (2) "Haw Yee Yee," and (3) "Yee Haw Yee." We can append the start and stop tokens to the corpus and transform it into*

$$\begin{aligned} &\langle s \rangle \ \textit{Yee Haw} \ \langle /s \rangle \\ &\langle s \rangle \ \textit{Haw Yee Yee} \ \langle /s \rangle \\ &\langle s \rangle \ \textit{Yee Haw Yee} \ \langle /s \rangle \end{aligned}$$

With these start and stop tokens in mind, we slightly relax the Assumption 2 of the n-gram model and investigate the probability of a word w being the first or the last word of a sentence, separately from other probabilities. We will denote these probabilities respectively as $\Pr[w \mid \langle s \rangle]$ and $\Pr[\langle /s \rangle \mid w]$. The former probability will be estimated as

$$\Pr[w \mid \langle s \rangle] \approx \frac{\text{Count}(\langle s \rangle \ w)}{\text{total number of sentences}} \tag{8.11}$$

which is the proportion of sentences that start with the word w in the corpus. The latter probability is estimated as

$$\Pr[\langle /s \rangle \mid w] \approx \frac{\text{Count}(w \ \langle /s \rangle)}{\text{Count}(w)} \tag{8.12}$$

which is the proportion of the occurrence of w that is at the end of a sentence in the corpus.

Also, notice that other bigram probabilities are also affected when introducing the stop tokens. In (8.6), the denominator originally did not include the occurrence of the substring at the end of the sentence because there was no word to follow that substring. However, if we consider $\langle /s \rangle$ as a word in the dictionary, the denominator can now include the case where the substring is at the end of the sentence. Therefore, the denominator is just equivalent to the Count of the substring in the corpus. Therefore, the bigram probabilities after introducing start, stop tokens can be estimated instead as [6]

[6] If we consider $\langle s \rangle$, $\langle /s \rangle$ as vocabularies of the dictionary, (8.13) can also include (8.11), (8.12).

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)}{\text{Count}(w_{j-1})} \tag{8.13}$$

Example 8.3.2. *We revisit Example 8.2.2. The bigram frequency table and the bigram probability table can be recalculated as in Table 8.3 and Table 8.4.* [7]

[7] Note that the values in the *Total* column now correspond to the unigram count of that word.

previous \ next	"Yee"	"Haw"	$\langle /s \rangle$	Total
$\langle s \rangle$	2	1	0	3
"Yee"	1	2	2	5
"Haw"	2	0	1	3

Table 8.3: Bigram frequency table of the cowperson language with start and stop tokens.

previous \ next	"Yee"	"Haw"	$\langle /s \rangle$	Total
$\langle s \rangle$	2/3	1/3	0/3	1
"Yee"	1/5	2/5	2/5	1
"Haw"	2/3	0/3	1/3	1

Table 8.4: Bigram probabilty table of the cowperson language with start and stop tokens.

Therefore, the bigram probabilities of the cowperson language, once we introduce the start and stop tokens, are given as

$$\Pr[\text{"Yee"} \mid \langle s \rangle] = \frac{2}{3} \qquad \Pr[\text{"Haw"} \mid \langle s \rangle] = \frac{1}{3}$$
$$\Pr[\text{"Yee"} \mid \text{"Yee"}] = \frac{1}{5} \quad \Pr[\text{"Haw"} \mid \text{"Yee"}] = \frac{2}{5} \quad \Pr[\langle /s \rangle \mid \text{"Yee"}] = \frac{2}{5}$$
$$\Pr[\text{"Yee"} \mid \text{"Haw"}] = \frac{2}{3} \quad \Pr[\text{"Haw"} \mid \text{"Haw"}] = \frac{0}{3} \quad \Pr[\langle /s \rangle \mid \text{"Haw"}] = \frac{1}{3}$$

8.3.2 *Redefining the Probability of a Sentence*

The biggest advantage of introducing stop tokens is that now we can assign a probability distribution over all sentences of finite length, not just a given length k. Say we want to assign a probability to the sentence $w_1 w_2 \dots w_k$ (without the start and stop tokens). By introducing start and stop tokens, we can interpret this as the probability of $w_0 w_1 \dots w_{k+1}$ where $w_0 = \langle s \rangle$ and $w_{k+1} = \langle /s \rangle$. Following the similar logic from (8.2), we can define this probability by the Chain Rule.

Definition 8.3.3 (Bigram Model with Start, Stop Tokens)**.** *A bigram model, once augmented with start, stop tokens, assigns the following probability to a sentence $w_1 w_2 \dots w_k$* [8]

[8] Notice that we do not have the term $\Pr[w_0]$ in the expansion. A sentence always starts with a start token, so the marginal probability that the first word is $\langle s \rangle$ can be understood to be 1.

$$\Pr[w_1 w_2 \dots w_k] = \prod_{i=1}^{k+1} \Pr[w_i \mid w_{i-1}] \tag{8.14}$$

where the bigram probabilities are estimated as in (8.13).

Example 8.3.4. *The probability that we see the sentence "Yee Haw Yee" in the cowperson language can be calculated as*

$$\begin{aligned}&\Pr[\text{"}Yee\text{"} \mid \langle s\rangle] \times \Pr[\text{"}Haw\text{"} \mid \text{"}Yee\text{"}] \times \Pr[\text{"}Yee\text{"} \mid \text{"}Haw\text{"}] \times \Pr[\langle /s\rangle \mid \text{"}Yee\text{"}]\\ &= \frac{2}{3} \times \frac{2}{5} \times \frac{2}{3} \times \frac{2}{5} \simeq 0.07\end{aligned}$$

Note that this probability is taken over all sentences of finite length.

Problem 8.3.5. *Verify that (8.14) defines a probability distribution over all sentences of finite length.*

8.3.3 Beyond Bigram Models

In general, if we have an n-gram model, then we may need to introduce more than 1 start or stop tokens. For example, in a trigram model, we will need to define the probability that the word is the first word of the sentence as $\Pr[w \mid \langle s\rangle \langle s\rangle]$. Based on the number of start and stop tokens introduced, the n-gram probabilities will need to be adjusted accordingly.

8.4 Testing a Language Model

So far, we discussed how to define an n-gram language model given a corpus. This is analogous to training a model given a training dataset. Naturally, the next step is to test the model on a newly seen held-out data to ensure that the model generalizes well. In this section, we discuss how to test a language model.

8.4.1 Shakespeare Text Production

First consider a bigram text generator — an application of the bigram model. The algorithm initiates with the start token $\langle s\rangle$. It then outputs a random word w_1 from the dictionary, according to the probability $\Pr[w_1 \mid \langle s\rangle]$. It then outputs the second random word w_2 from the dictionary, according to the probability $\Pr[w_2 \mid w_1]$. It repeats this process until the newly generated word is the stop token $\langle /s\rangle$. The final output of the algorithm will be the concatenated string of all outputted words.

It is possible to define a text generator for any n-gram model in general. Figure 8.4 shows the output of the unigram, bigram, trigram, quadrigram text generators when the models were trained on all Shakespeare texts.

Notice the sentence "I will go seek the traitor Gloucester." in the output of the quadrigram text generator. This exact line appears in *King Lear*, Act 3 Scene 7. This is not a coincidence. Figure 8.5 presents

```
<s> I
    I want
      want to
           to eat
              eat tasty
                  tasty food
                       food  </s>
I want to eat tasty food
```

Figure 8.3: An example run of the bigram text generator.

Unigram

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
Every enter now severally so, let
Hill he late speaks; or! a more to leg less first you enter
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram

What means, sir. I confess she? then all sorts, he is trim, captain.
Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Trigram

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
This shall forbid it should be branded, if renown made it empty.
Indeed the duke; and had a very good friend.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

Quadrigram

King Henry.What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
Will you not tell me who I am?
It cannot be but so.
Indeed the short and the long. Marry, 'tis a noble Lepidus.

Figure 8.4: The outputs of unigram, bigram, trigram, quadrigram text generators trained on Shakespeare texts.

the snapshot of the bigram, trigram, and quadrigram text generators once they have outputted the phrase "go seek the." You can see that bigram models and trigram models assign very small probabilities to the word "traitor" because there are many more instances of phrases "the" or "seek the" in the corpus than "go seek the." On the other hand, the quadrigram model assigns a very large probability to the word "traitor" because there is only a limited number of times that the phrase "go seek the" appears in the corpus.

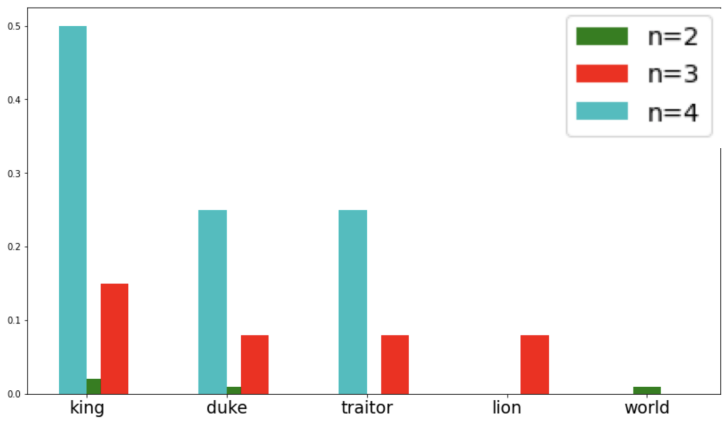

Figure 8.5: The probability of the next word given the previous three words are "go seek the" in the n-gram model, where $n = 2, 3, 4$.

Once the quadrigram model outputs the word "traitor" after the

phrase "go seek the," the problem is even worse. As can be seen in Figure 8.6, the quadrigram model assigns probability of 1 to the word "Gloucester" meaning that the phrase "seek the traitor" only appears before the word "Gloucester." So the model has *memorized* one completion of the phrase from the training text. From this example, we can see that text production based on n-grams is sampling and remixing text fragments seen in the training corpus.

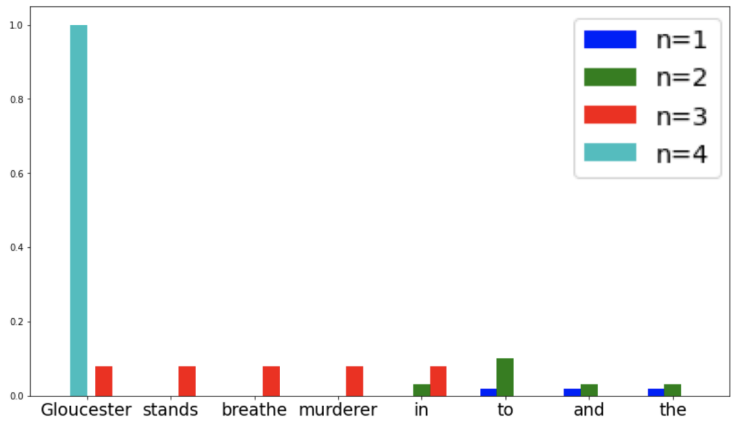

Figure 8.6: The probability of the next word given the previous three words are "seek the traitor."

The Shakespeare corpus consists of $N = 884,647$ words and $V = 29,066$ distinct words from the dictionary. There are about $V^2 \approx 845$ million possible combinations of bigrams, but Shakespeare only used around $300,000$ of them in his text. So 99.96% of the possible bigrams were never seen. The percentage is much higher for quadrigrams! Furthermore, for the quadrigrams that do appear in the corpus, most do not even repeat. Thus what comes out of the quadrigram model looks like Shakespeare because it is a *memorized* fragment of Shakespeare.[9]

[9] Do this remind you of overfitting?

8.4.2 *Perplexity*

Having described a way to train a simple language model, we now turn our attention to a formal way of testing [10] a language model.

[10] This method is used even for testing state of the art models.

Just like any other model in ML, a language model will be given a corpus $w_1 w_2 \dots w_T$. Then we can assess the performance of the model by its *perplexity* on the corpus.

Definition 8.4.1 (Perplexity)**.** *The **perplexity** of a language model on the corpus $w_1 w_2 \dots w_T$ is defined as*

$$\Pr[w_1 w_2 \dots w_T]^{-\frac{1}{T}} = \sqrt[T]{\frac{1}{\Pr[w_1 w_2 \dots w_T]}} \tag{8.15}$$

Note that, perplexity is defined for any probabilistic language model: the Chain Rule of joint probability applies to every model,

and does not require the n-gram assumptions. That is, [11]

$$\Pr[w_1 w_2 \dots w_T] = \Pr[w_1] \times \prod_{i=2}^{T} \Pr[w_i \mid w_1 \dots w_{i-1}]$$

Then the perplexity of the model can be rewritten as

$$\sqrt[T]{\frac{1}{\Pr[w_1]} \times \prod_{i=2}^{T} \frac{1}{\Pr[w_i \mid w_1 \dots w_{i-1}]}} \tag{8.16}$$

Example 8.4.2. *Consider the uniform ("clueless") model which assumes that the probability of all words are equal in any given situation. That is, if V is the vocabulary size (i.e., size of the dictionary),*

$$\Pr[w_i] = \Pr[w_i \mid w_1 \dots w_{i-1}] = \frac{1}{V}$$

*for any given $w_1, \dots, w_i \in V$. This model assigns $\left(\frac{1}{V}\right)^T$ to **every** sequence of T words, including the corpus. Therefore, the perplexity of the model is*

$$\left(\left(\frac{1}{V}\right)^T\right)^{-\frac{1}{T}} = V$$

Now we try to understand perplexity at an intuitive level. (8.16) is the geometric mean [12] of the following T values:

$$\frac{1}{\Pr[w_1]}, \frac{1}{\Pr[w_2 \mid w_1]}, \dots, \frac{1}{\Pr[w_T \mid w_1 \dots w_{T-1}]}$$

Now note that a probabilistic model splits the total probability of 1 to fractions and distributes them to the potential options for the next word. So the inverse of an assigned probability for a word can be thought roughly as the *number of choices the model considered for the next word*. With this viewpoint, perplexity as written in (8.16) means: *how much has the model narrowed down the number of choices for the next word* **on average?** The clueless model had not narrowed down the possibilities at all and had the worst-possible perplexity equal to the number of vocabulary words.

Example 8.4.3. *Consider a well-trained language model. At any given place of text, it can identify a set of* 20 *words and assigns probability $\frac{1}{20}$ to each of them to be the next word. It happens that the next word is **always** one of the* 20 *words that the model identifies. The perplexity of the model is*

$$\left(\left(\frac{1}{20}\right)^T\right)^{-\frac{1}{T}} = 20$$

Interestingly enough, the true perplexity of English is believed to be between 15 and 20. That is, if at an "average" place in text, you ask humans to predict the next word, then they are able to narrow down the list of potential next words to around 15 to 20 words. [13]

[11] Assume for now that start and stop tokens do not exist in the corpus.

[12] The *geometric mean* of T numbers $a_1, a_2, \dots, a_T$ is defined as $(\prod_i a_i)^{1/T}$

[13] The perplexity of state of the art language models is under 20 as well.

8.4.3 Perplexity on Test Corpus

The *perplexity* of a language model is analogous to a *loss* of an ML model. [14] Similar to ML models we have been studying so far, it is possible to define a *train perplexity* and a *test perplexity*. The "goodness" of the model will be defined by how low the perplexity was on a previously unseen, held-out data.

[14] It is customary to use the logarithm of the perplexity, as we also did for logistic loss in Chapter 4.

For example, when n-gram models are trained on 38 million words and tested on 1.5 million words from Wall Street Journal articles, they show the following test perplexities in Table 8.5. [15] Note that the state-of-the-art deep learning models achieve a test perplexity of around 20 on the same corpus.

[15] To be more exact, the models were augmented with smoothing, which will be introduced shortly.

Unigram	Bigram	Trigram
962	170	109

Table 8.5: Test perplexities of n-gram models on WSJ corpus.

8.4.4 Perplexity With Start and Stop Tokens

When start and stop tokens are introduced to a corpus, we also need to redefine how to calculate the perplexity of the model. Again, we will only focus on a bigram model for the sake of exposition.

Say the corpus consists of t sentences:

$$\begin{aligned}
&\langle s \rangle\, w_{1,1} w_{1,2}, \ldots, w_{1,T_1}\, \langle /s \rangle \\
&\langle s \rangle\, w_{2,1} w_{2,2}, \ldots, w_{2,T_2}\, \langle /s \rangle \\
&\qquad\qquad \vdots \\
&\langle s \rangle\, w_{t,1} w_{t,2}, \ldots, w_{t,T_t}\, \langle /s \rangle
\end{aligned}$$

The probability of the corpus $w_{1,1} w_{1,2} \ldots w_{t,T_t}$ is redefined as the product of the probability of each of the sentences:

$$\begin{aligned}
\Pr[w_{1,1} w_{1,2} \ldots w_{t,T_t}] &= \prod_{i=1}^{t} \Pr[w_{i,1} w_{i,2} \ldots w_{i,T_i}] \\
&= \prod_{i=1}^{t} \prod_{j=1}^{T_i+1} \Pr[w_{i,j} \mid w_{i,j-1}] \qquad (8.17)
\end{aligned}$$

Now we apply the interpretation of the perplexity that it is the geometric mean of probabilities of each word. Notice that we multiplied $\sum_{i=1}^{t} (T_i + 1)$ probabilities to calculate the probability of the corpus. If we let $T = \sum_{i=1}^{t} T_i$ denote the total number of words (excluding start and stop tokens) of the corpus, the number of probabilities we multiplied can be written as $T^* = T + t$. [16]

[16] This can also be thought as adding the number of stop tokens to the number of words in the corpus.

Definition 8.4.4 (Perplexity with Start, Stop Tokens)**.** *The perplexity of a bigram model with start, stop tokens can be redefined as*

$$\sqrt[T^*]{\frac{1}{\prod_{i=1}^{t}\prod_{j=1}^{T_i+1}\Pr[w_{i,j} \mid w_{i,j-1}]}} \tag{8.18}$$

8.4.5 Smoothing

One big problem with our naive definition of the perplexity of a model is that it does not account for a zero denominator. That is, if the model assigns probability exactly 0 to the corpus, then the perplexity of the model will be ∞! [17]

[17] Mathematically, it is undefined, but here assume that the result is a positive infinity that is larger than any real number.

Example 8.4.5. *Suppose the phrase "green cream" never appeared in the training corpus, but the test corpus contains the sentence "You like green cream." Then a bigram model will have a perplexity of ∞ because it assigns probability 0 to the bigram "green cream."*

To address this issue, we generally apply *smoothing* techniques, which never allow the model to output a zero probability. By *reducing* the naive estimate of *seen* events and *increasing* the naive estimate of *unseen* events, we can always assign nonzero probabilities to previously unseen events.

The most commonly used smoothing technique is the 1-add smoothing (a.k.a, Laplace smoothing). We describe how the smoothing works for a bigram model. The main idea of the 1-add smoothing can be summarized as "add 1 to all bigram counts in the bigram frequency table." Then the bigram probability as defined in Definition 8.2.1 can be redefined as

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)+1}{\sum_w (\text{Count}(w_{j-1}w)+1)} = \frac{\text{Count}(w_{j-1}w_j)+1}{\sum_w (\text{Count}(w_{j-1}w)) + V} \tag{8.19}$$

where V is the size of the dictionary. If we had augmented the corpus with the start and the stop tokens, the denominator in (8.19) is just equal to $\text{Count}(w_{j-1}) + V^*$ [18] and so the bigram probability can be written as

$$\Pr[w_j \mid w_{j-1}] \approx \frac{\text{Count}(w_{j-1}w_j)+1}{\text{Count}(w_{j-1}) + V^*} \tag{8.20}$$

[18] $V^* = V + 1$ is the size of the dictionary after adding the start and the stop tokens. It is customary to add only one to the vocabulary count. It may help to look at the number of rows and columns in the bigram frequency table 8.3.

Notice that the denominator is just V^*, the new vocabulary size, added to the unigram count of w_{j-1}.

Example 8.4.6. *Recall the cowperson language with the start and stop tokens from Example 8.3.2. Upon further research, it turns out the language actually consists of three words: $\{Yee, Haw, Moo\}$, but the training corpus*

"Yee Haw Haw Yee Yee Yee Haw Yee" left out one of the vocabularies in the dictionary. By applying add-1 smoothing to the bigram model, we can recalculate the bigram frequency and the bigram probability table as in Table 8.6 and Table 8.7

previous \ next	"Yee"	"Haw"	"Moo"	$\langle /s \rangle$	Total
$\langle s \rangle$	3	2	1	1	7
"Yee"	2	3	1	3	9
"Haw"	3	1	1	2	7
"Moo"	1	1	1	1	4

Table 8.6: Bigram frequency table of the cowperson language with start and stop tokens with smoothing.

previous \ next	"Yee"	"Haw"	"Moo"	$\langle /s \rangle$	Total
$\langle s \rangle$	3/7	2/7	1/7	1/7	1
"Yee"	2/9	3/9	1/9	3/9	1
"Haw"	3/7	1/7	1/7	2/7	1
"Moo"	1/4	1/4	1/4	1/4	1

Table 8.7: Bigram probability table of the cowperson language with start and stop tokens with smoothing.

The probability that we see the sentence "Moo Moo" in the cowperson language, which would have been 0 before smoothing, is now assigned a non-zero value:

$$\begin{aligned} &\Pr[\text{"Moo"} \mid \langle s \rangle] \times \Pr[\text{"Moo"} \mid \text{"Moo"}] \times \Pr[\langle /s \rangle \mid \text{"Moo"}] \\ &= \frac{1}{7} \times \frac{1}{4} \times \frac{1}{4} \simeq 0.01 \end{aligned}$$

Problem 8.4.7. *Verify that (8.19) defines a proper probability distribution over the conditioned event. That is, show that*

$$\sum_{w} \Pr[w \mid w'] = 1$$

for any w in the dictionary.

Another smoothing technique is called *backoff* smoothing. The intuition is that n-gram probabilities are less likely to be zero if n is smaller. So when we run into an n-gram probability that is zero, we replace it with a linear combination of n-gram probabilities of lower values of n.

Example 8.4.8. *Recall Example 8.4.5. The bigram probability of "green cream" can be approximated instead as*

$$\Pr[\text{"cream"} \mid \text{"green"}] \approx \Pr[\text{"cream"}]$$

Also, say we want to calculate the trigram probability of "like green cream," which is also zero in the naive trigram model. We can approximate it instead as

$$\Pr[\text{"cream"} \mid \text{"like green"}] \approx \alpha \Pr[\text{"cream"}] + (1-\alpha) \Pr[\text{"cream"} \mid \text{"green"}]$$

where α is a hyperparameter for the model.

There are other variants of the backoff smoothing, [19] with some theory for what the "best" choice is, but we will not cover it in these notes.

[19] For instance, Good-Turing and Kneser-Ney smoothing.

9
Matrix Factorization and Recommender Systems

9.1 Recommender Systems

Cataloging and recommender systems have always been an essential asset for consumers who find it difficult to choose from the vast scale of available goods. As early as 1876, the Dewey decimal system was invented to organize libraries. In 1892, Sears released their famed catalog to keep subscribers up to date with the latest products and trends, which amounted to 322 pages. Shopping assistants at department stores or radio disc jockeys in the 1940s are also examples of recommndations via human curation. In more contemporary times, bestseller lists at bookstores, or Billboard Hits list aim to capture what is popular among people. The modern recommender system paradigm now focuses on recommending products based on what is liked by people "similar" to you. In this long history of recommender systems, the common theme is that *people like to follow trends*, and recommender systems can help catalyze this process.

9.1.1 Movie Recommendation via Human Curation

Suppose we want to design a recommender system for movies. A human curator identifies r binary attributes that they think are important for a movie (*e.g.*, is a romance movie, is directed by Steven Spielberg, etc.) Then they assign each movie an r-dimensional *attribute vector*, where each element represents whether the movie has the corresponding attribute (*e.g.*, coordinate 2 will have value 1 if a movie is a "thriller" and 0 otherwise).

Now, using a list of movies that a particular user likes, the curator assigns an r-dimensional *taste vector* to a given user in a similar manner (*e.g.*, coordinate w will have value 1 if a user likes "thrillers" and 0 otherwise). With these concepts in mind, we can start with defining the affinity of a user for a particular movie:

Definition 9.1.1 (User Affinity). *Given a taste vector* $\mathbf{A}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,r})$ *for user* i *and the genre vector* $\mathbf{B}_j = (b_{1,j}, b_{2,j}, \ldots, b_{r,j})$ *for movie* j*, we define the* ***affinity of user*** i ***for movie*** j *as*

$$\mathbf{A}_i \cdot \mathbf{B}_j = \sum_{k=1}^{r} a_{i,k} b_{k,j} \tag{9.1}$$

Intuitively, this metric counts the number of attributes which are 1 in both vectors, or equivalently how many of the user's boxes are "checked off" by the movie. Mathematically, the affinity is defined as a dot product, which can be extended to matrix multiplication. Thus if we have a matrix $\mathbf{A} \in \mathbb{R}^{m \times r}$ where each of m rows is a taste vector for a user and a matrix $\mathbf{B} \in \mathbb{R}^{r \times n}$ where each of n columns is a genre vector for a movie, the (i, j) entry of the matrix product $\mathbf{M} = \mathbf{AB}$ represents the affinity score of user i for movie j.

We can also define an additional similarity metric:

Definition 9.1.2 (Similarity Metric). *Given taste vector* A_i *for user* i *and taste vector* $\mathbf{A}_j$ *for user* j*, we define the* ***similarity of user*** i ***and user*** j *as*

$$\sum_{k=1}^{r} a_{i,k} a_{j,k} \tag{9.2}$$

Similarly, the ***similarity of movie*** i ***and movie*** j *is defined as*

$$\sum_{k=1}^{r} b_{k,i} b_{k,j} \tag{9.3}$$

Finally, in practice, each individual is unique and has a different average level of affinity for movies (for example, some users like everything while others are very critical). This means that directly comparing the affinity of one user to another might not be helpful. One way to circumvent this problem is to augment (9.1) in Definition 9.1.1 as

$$\sum_{k=1}^{r} a_{i,k} b_{k,j} + a_{i,0} \tag{9.4}$$

with a bias term $a_{i,0}$.

Based on the affinity scores or similarity scores, the human curator will be able to recommend movies to users. This model design seems like it does the job as a recommender system. In practice, developing such models through human curation comes with a set of pros and cons:

- *Pros:* Using human curation allows domain expertise to be leveraged and this intuition can be critical in the development of a good model (*i.e.*, which attribute is important). In addition, a human curated model will naturally be interpretable.

- *Cons:* The process is tedious and expensive; thus it is difficult to scale. In addition, it can be difficult to account for niche demographics and genres and this becomes a problem for companies with global reach.

We conclude that while human curated models can certainly be useful, the associated effort is often too great.

9.2 *Recommender Systems via Matrix Factorization*

In this section, we provide another technique that can be used for recommender systems — matrix factorization. This method started to become popular since 2005.

9.2.1 *Matrix Factorization*

Matrix factorizations are a common theme throughout linear algebra. Some common techniques include LU and QR decomposition, Rank Factorization, Cholesky Decomposition, and Singular Value Decomposition.

Definition 9.2.1 (Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$*. A* ***matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *such that* $\mathbf{M} = \mathbf{AB}$ *for some* $r < m, n$*.*

Unfortunately, these techniques become less directly applicable once we consider the case where most of the entries of $\mathbf{M}$ are missing (*i.e.*, a missing-data setting). As we saw in Section 9.1.1, this is very common in real-world applications — for example, if the (m, n) entry of M represents the rating of user m for movie n, most entries in M are missing because not everyone has seen every movie. What can we do in such a case?

In turns out, if we assume that M is a low-rank matrix (which is true for many high-dimensional datasets, as noted in Chapter 7), then we can consider an approximate factorization $\mathbf{M} \approx \mathbf{AB}$ on the known entries. We express this as the following optimization problem:

Definition 9.2.2 (Approximate Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$ *where* $\Omega \subset [m] \times [n]$ *is the subset of* (i, j) *where* M_{ij} *is known. An* ***approximate matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *for some* $r < m, n$ *that minimize the loss function:*

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \tag{9.5}$$

We denote the approximation as $\mathbf{M} \approx \mathbf{AB}$*.*

Notice this form is familiar: we are effectively trying to find optimal matrices $\mathbf{A}, \mathbf{B}$ which will minimize the MSE between known entries of M and corresponding entries in the matrix product $\mathbf{AB}$! One thing to note is that by calculating the matrix product $\mathbf{AB}$, we can "predict" entries of $\mathbf{M}$ that are unknown.

You can take the following result from linear algebra as granted.

Theorem 9.2.3. *Given $\mathbf{M} \in \mathbb{R}^{m \times n}$, we can find the matrix factorization $\mathbf{M} = \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if M has rank at most r. Also, we can find the approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}, \mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if $\mathbf{M}$ is "close to" rank r.*

9.2.2 *Matrix Factorization as Semantic Embeddings*

Recall the setup in Section 9.1.1. But instead of calculating the affinity matrix $\mathbf{M}$ as the product of the matrices $\mathbf{A}, \mathbf{B}$, we will approach from the opposite direction. We will start with an affinity matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ (which is only partially known) and find its approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$. We can understand that $\mathbf{A} \in \mathbb{R}^{m \times r}$ represents a set of users and that $\mathbf{B} \in \mathbb{R}^{r \times n}$ represents a set of movies.

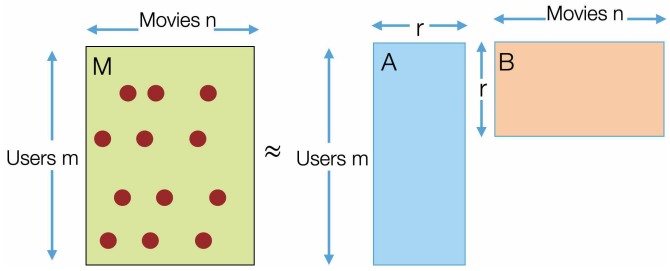

Figure 9.1: Matrix factorization on movie recommendations. Usually the inner dimension r would be much smaller than m, n.

Specifically, if we let A_{i*} denote the i-th row of $\mathbf{A}$ and B_{*j} denote the j-th column of $\mathbf{B}$, then A_{i*} can be understood as the *taste vector* of user i and B_{*j} can be understood as the *attribute vector* of movie j. One difference to note is that the output of a matrix factorization is real-valued, unlike the the $0/1$ valued matrices $\mathbf{A}, \mathbf{B}$ from Section 9.1.1. We can then use the vectors A_{i*} and B_{*j} to find similar users or movies and make recommendations.

Example 9.2.4. *Assume all columns of $\mathbf{B}$ have ℓ_2 norm 1. That is, $\|B_{*j}\|_2 = 1$ for all j. When the inner product $B_{*j} \cdot B_{*j'}$ of two movie vectors is actually 1, the two vectors are exactly the same! They have the same inner product with every user vector A_{i*} — in other words these movies have the same appeal to all users. Now suppose $B_{*j} \cdot B_{*j'}$ is not quite 1 but close to 1, say 0.9. This means the movie vectors are quite close but not the same. Still, their inner product with typical user vectors will not be too different. We*

conclude that two movies j, j' with inner product $B_{*j} \cdot B_{*j'}$ close to 1 tend to get recommended together to users. One can similarly conclude that high value of inner product between two user vectors is suggestive that the users have similar tastes.

9.2.3 *Netflix Prize Competition: A Case Study*

During 2006-09, DVDs were all the rage. Companies like Netflix were quite interested in recommending movies as accurately as possible in order to retain clients. At the time, Netflix was using an algorithm which had stagnated around $RMSE = 0.95$. [1] Seeking fresh ideas, Netflix curated an anonymized database of $100M$ ratings (each rating was on a $1-5$ scale) of $0.5M$ users for $18K$ movies. Adding a cash incentive of $\$1,000,000$, Netflix challenged the world to come up with a model that could achieve a much lower RMSE! [2] It turned out that matrix factorization would be the key to achieving lower scores. In this example, $m = 0.5M$, $n = 18k$, and Ω corresponds to the $100M$ ratings out of $m \cdot n = 10B$ affinities. [3]

After a lengthy competition, [4] the power of matrix factorization is on full display when we consider the final numbers:

- Netflix's algorithm: $RMSE = 0.95$
- Plain matrix factorization: $RMSE = 0.905$
- Matrix factorization and bias: $RMSE = 0.9$
- Final winner (an ensemble of many methods) : $RMSE = 0.856$

[1] $RMSE$ is shorthand for $\sqrt{MSE}$.

[2] This was an influential competition, and is an inspiration for today's hackathons, Kaggle, etc.

[3] Less than 1% of possible elements are accounted for by Ω.

[4] Amazingly, a group of Princeton undergraduates managed to achieve the second place!

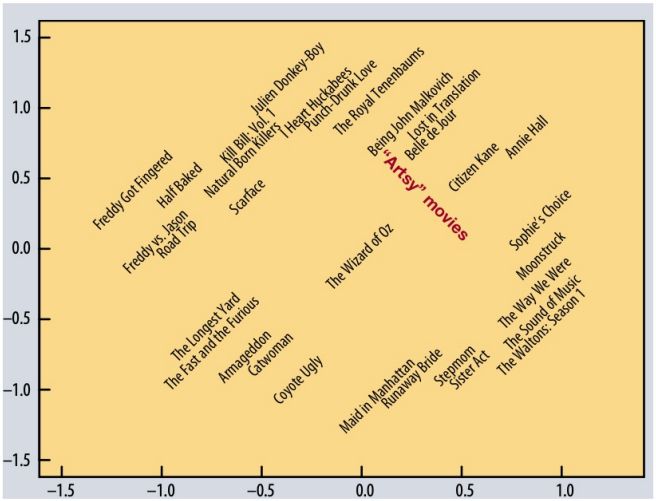

Figure 9.2: 2D visualization of embeddings of film vectors. Note that you see clusters of "artsy" films on top right, and romantic films on the bottom. *Credit: Koren et al.,* Matrix Factorization Techniques for Recommender Systems, *IEEE Computer* 2009.

9.2.4 *Why Does Matrix Factorization Work?*

In general, we need mn entries to completely describe a $m \times n$ matrix $\mathbf{M}$. However, if we find factor $\mathbf{M}$ into the product $\mathbf{M} = \mathbf{AB}$ of $m \times r$ matrix $\mathbf{A}$ and $r \times n$ matrix $\mathbf{B}$, then we can describe $\mathbf{M}$ with essentially only $(m+n)r$ entries. When r is small enough such that $(m+n)r \ll mn$, some entries of $\mathbf{M}$ (including the missing entries) are not truly "required" to understand $\mathbf{M}$.

Example 9.2.5. *Consider the matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & 2 \\ 1 & 1 & * & * \\ 4 & * & 8 & * \\ 4 & * & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? Since* $r = 1$*, it means that all the rows/columns of* $\mathbf{M}$ *are the same up to scaling. By observing the known entries, the second row should be equal to the first row, and the third and the fourth row should be equal to the the first row multiplied by* 4*. Therefore, we can fill in the missing entries as*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 4 & 4 & 8 & 8 \\ 4 & 4 & 8 & 8 \end{bmatrix}$$

It is not hard to infer that $\mathbf{M} = \mathbf{AB}$ *where* $\mathbf{A} = (1, 1, 4, 4)^T$ *and* $\mathbf{B} = (1, 1, 2, 2)$

Example 9.2.6. *Consider another matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & * \\ 1 & 7 & * & * \\ 4 & * & * & 2 \\ * & 4 & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? This time, the answer is no. Following a similar logic from Example 9.2.5, the second row should be equal to the first row multiplied by a constant. This is not feasible since* $M_{2,1}/M_{1,1} = 1$ *and* $M_{2,2}/M_{1,2} = 7$*.*

9.3 *Implementation of Matrix Factorization*

In this section, we look more deeply into implementing matrix factorization in an ML setting. As suggested in Definition 9.2.2, we can consider the process of approximating a matrix factorization to be an optimization problem. Therefore, we can use gradient descent.

9.3.1 Calculating the Gradient of Full Loss

Recall that for an approximate matrix factorization of a matrix $\mathbf{M}$, we want to find matrices $\mathbf{A}, \mathbf{B}$ that minimize the following loss:

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \qquad \text{(9.5 revisited)}$$

Here $(AB)_{ij} = A_{i*} \cdot B_{*j}$. Now we find the gradient of the loss $L(\mathbf{A}, \mathbf{B})$ by first finding the derivatives of L with respect to elements of $\mathbf{A}$ (a total of mr derivatives), then finding the derivatives of L with respect to elements of $\mathbf{B}$ (a total of nr derivatives).

First, consider an arbitrary element $A_{i'k'}$:

$$\begin{aligned}
\frac{\partial}{\partial A_{i'k'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{i'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \cdot (-B_{k'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \qquad (9.6)
\end{aligned}$$

Note that the second step is derived because $(AB)_{ij} = \sum_k A_{ik}B_{kj}$ and if $i \neq i'$, then $\frac{\partial (AB)_{ij}}{\partial A_{i'k'}} = 0$. Enumerating $(i,j) \in \Omega$ can be changed to only enumerating $(i',j) \in \Omega$. Similarly, we can consider an arbitrary element $B_{k'j'}$:

$$\begin{aligned}
\frac{\partial}{\partial B_{k'j'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \cdot (-A_{ik'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \qquad (9.7)
\end{aligned}$$

Whew! That's a lot of derivatives, but we now have $\nabla L(\mathbf{A}, \mathbf{B})$ at our disposal.

9.3.2 Stochastic Gradient Descent for Matrix Factorization

Of course, we could use $\nabla L(\mathbf{A}, \mathbf{B})$ for a plain gradient descent as shown in Chapter 3. However, given that each derivative in the gradient involves a sum over a large number of indices, it would be

worthwhile to use stochastic gradient descent in order to estimate the overall gradient via a small random sample (as shown in Section 3.2).

If we take a sample $S \subset \Omega$ of the known entries at each iteration, the loss becomes

$$\widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i,j \in S} (M_{ij} - (AB)_{ij})^2 \tag{9.8}$$

and the gradient becomes

$$\frac{\partial}{\partial A_{i'k'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{j:\ (i',j) \in S} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \tag{9.9}$$

$$\frac{\partial}{\partial B_{k'j'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i:\ (i,j') \in S} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \tag{9.10}$$

However, if we take a uniform sample S of Ω, the computation will not become much cheaper, since $(i, j) \in S$ can spread into many different rows and columns. One clever (and common) way to do so is to select a set of columns C by sampling k out of the overall n columns. This method is called *column sampling*. We then only need to consider entries $(i, j) \in \Omega$ where $j \in C$ and compute gradients only for the entries $B_{k,j}$ where $j \in C$. We can also perform row sampling in a very similar manner. In practice, whether we should use column sampling or row sampling, or gradient descent of full loss, depends on the actual sizes of m and n.