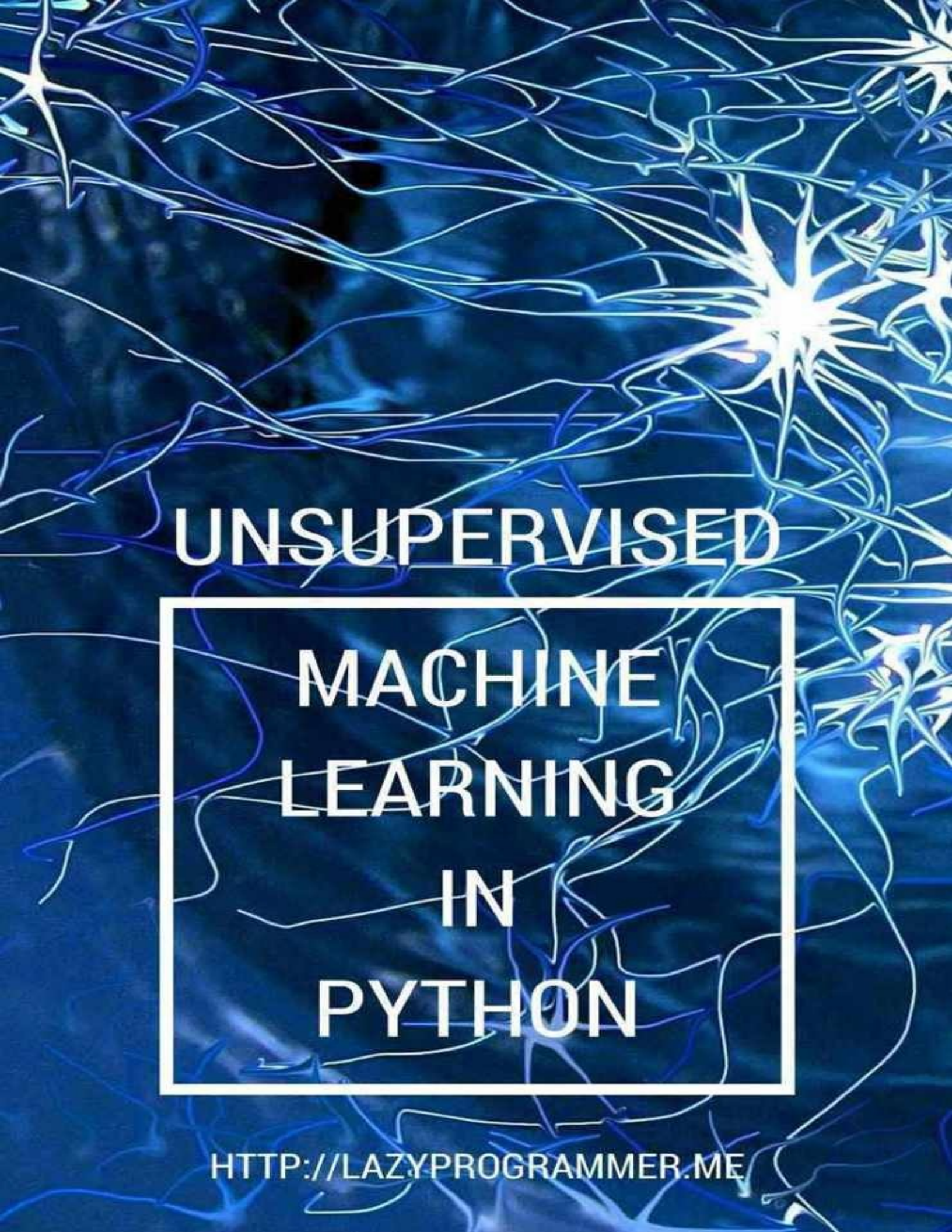

UNSUPERVISED
MACHINE
LEARNING
IN
PYTHON
HTTP://LAZYPROGRAMMER.ME

Unsupervised Machine Learning in Python

Master Data Science and Machine Learning with Cluster Analysis, Gaussian Mixture Models, and Principal Components Analysis

By: The LazyProgrammer (http://lazyprogrammer.me)

Singular Covariance Problem

Chapter 5: Principal Components Analysis

Conclusion

Introduction

In a real-world environment, you can imagine that a robot or an artificial intelligence won't always have access to the optimal answer, or maybe there isn't an optimal correct answer. You'd want that robot to be able to explore the world on its own, and learn things just by looking for patterns.

Think about the large amounts of data being collected today, by the likes of the NSA, Google, and other organizations. No human could possibly sift through all that data manually. It was reported recently in the Washington Post and Wall Street Journal that the National Security Agency collects so much surveillance data, it is no longer effective.

Could automated pattern discovery solve this problem?

Do you ever wonder how we get the data that we use in our supervised machine learning algorithms?

Kaggle always seems to provide us with a nice CSV, complete with Xs and corresponding Ys.

If you haven’t been involved in acquiring data yourself, you might not have thought about this, but *someone* has to make this data!

A lot of the time this involves manual labor. Sometimes, you don’t have access to the correct information or it is infeasible or costly to acquire.

You still want to have some idea of the structure of the data.

This is where unsupervised machine learning comes into play.

In this book we are first going to talk about clustering. This is where instead of training on labels, we try to create our own labels. We’ll do this by grouping together data that looks alike.

The 2 methods of clustering we’ll talk about: k-means clustering and hierarchical clustering.

Next, because in machine learning we like to talk about probability distributions,

we'll go into Gaussian mixture models and kernel density estimation, where we talk about how to learn the probability distribution of a set of data.

One interesting fact is that under certain conditions, Gaussian mixture models and k-means clustering are exactly the same! We'll prove how this is the case.

Lastly, we'll look at the theory behind principal components analysis or PCA. PCA has many useful applications: visualization, dimensionality reduction, denoising, and de-correlation. You will see how it allows us to take a different perspective on latent variables, which first appear when we talk about k-means clustering and GMMs.

All the algorithms we'll talk about in this course are *staples* in machine learning and data science, so if you want to know how to *automatically* find patterns in your data with **data mining** and **pattern extraction**, without needing someone to put in manual work to label that data, then this book is for you.

All of the materials required to follow along in this book are free: You just need to able to download and install Python, Numpy, Scipy, Matplotlib, and Sci-kit Learn.

Chapter 1: What is unsupervised learning used for?

In general: unsupervised learning is for learning the structure or the probability distribution of the data. What does this mean specifically?

In this chapter we'll talk about some specific examples of how you can use unsupervised learning in your data pipeline.

Density Estimation:

You already know that we use the PDF, or probability density function, to tell us the probability of a random variable. Density estimation is the process of taking samples of data of the random variable, and figuring out the probability density function.

Once you learn the distribution of a variable, you can generate your own samples of the variable using that distribution.

At a high level, for example, you could learn the distribution of a Shakespeare

play, and then generate text that looks like Shakespeare.

Latent Variables:

A lot of the time, we want to know about the hidden or underlying causes of the data we're seeing.

These can be thought of as latent, missing, or hidden variables.

As an example, suppose you're given a set of documents, but you aren't told what they are.

You could do clustering on them and discover that there are a few very distinct groups in that set of documents.

Then, when you actually read some of the documents in your dataset, you see that one set of documents is romance novels, this other one is children's books, and so on.

Some examples of clustering algorithms are: k-means clustering (covered in this book), hierarchical clustering (covered in this book), and affinity propagation. Gaussian mixture models can be thought of as a “soft” or “fuzzy” clustering algorithm.

Hierarchical clustering and the dendrogram (the visualization we use on hierarchical clustering output) has been used in biology for constructing phylogenetic trees.

A lot of the time, your data is just so big it’s infeasible to look at the entire thing yourself, so you need some way of summarizing the data like this.

One view of this is “topic modeling”, where the latent variable is the “topic”, and the observed variable is the words.

Dimensionality reduction:

Another way to think of unsupervised machine learning is dimensionality

reduction. Some examples of algorithms that do dimensionality reduction are principal components analysis (PCA), singular value decomposition (SVD), t-SNE (t-distributed stochastic neighbor embedding), LLE (locally linear embedding), and more.

A lot of data can be hundred or thousands of dimensions wide. Think of a 28x28 image. That's 784 dimensions. Humans can't visualize anything past 3 dimensions.

28x28 is a small image. What if we increase the size a little bit, to 32x32, and add color? Color (RGB) has 3 different channels. So that's 3x32x32 = 3072 dimensionality data. 32x32 is still a tiny image! 1080p is 1920 pixels in width and 1080 pixels in height. How many dimensions is that?

If your algorithm run time is dependent on the dimensionality of your input, this could be a problem. A lot of your data is correlated (redundant), so you probably don't need all the dimensions anyway. So how do we reduce dimensionality, while retaining information? We will answer this question later in the book.

Visualization:

Another useful reason to do unsupervised learning is visualization. Sometimes you just need a summary picture of your data to give you a sense of the structure. Dimensionality reduction is useful here because it allows you to first reduce your data to 2 dimensions, at which point you can create a scatter plot.

As you'll see in this book, other types of algorithms can help us generate useful pictures too, like the dendrogram in hierarchical clustering.

A surprising but useful application of visualization is that it can tell us when an algorithm is *not* working, as we'll see when we do k-means.

Chapter 2: K-Means Clustering

Basic idea: Take a bunch of unlabeled data (data as in a set of vectors) and group them into K clusters.

The input into K-Means is just a matrix X. We usually organize it so that each row is a different sample, and each column is a different feature, or factor, in statistics terminology.

We usually say there are N samples and D features. So X is an N x D matrix.

There are 2 main steps in the K-Means algorithm.

First, we choose K different cluster centers. Usually we just assign these to random points in the dataset.

Next, we go into our main loop. The main loop is where the 2 main steps take place.

1) The first step is to decide which cluster each point belongs to. We do that by looking at every sample, and choosing the closest cluster center. Remember, we just assigned these randomly to begin with.

2) The second step is to recalculate each cluster center based on the points that were assigned to it. We do this by just taking all the samples and calculating the mean. Now you know where the name K-Means comes from.

We do this until the algorithm converges, *i.e.* there are no more changes in the cluster assignments or centers. Usually this happens very fast, in less than 5 steps.

This is very different from gradient descent in deep learning, where we might have thousands of iterations before convergence.

Let’s go through a visual example of how K-Means works.

' ' ' '

--- ---

| |

| |

| |

-*- ---

|3| |4|

--- ---

This is our initialization point. We have 4 vectors, labeled 1, 2, 3, and 4. The two cluster centers (K=2) have been randomly assigned to points 2 and 3. I've denoted them as * and ^. We are now ready to begin the "main loop".

Step number 1 is to decide which cluster each point belongs to. We see that points 1 and 3 belong to the cluster center on the left, because they are both closer to that cluster center (in fact, the distance from vector 3 to the center is 0 at the current moment).

Similarly, points 2 and 4 belong to the cluster center on the right.

Step number 2 is to recalculate the cluster centers based on the points that belong to the cluster.

The * cluster center moves in between 1 and 3, because that is the mean of those 2 points. Similarly, the ^ cluster center moves between 2 and 4, because that is the mean of those 2 points.

--- ---

|1| |2|

|1| |2|

--- ---

| |

* ∧

| |

--- ---

|3| |4|

--- ---

After this step, the algorithm converges, because vectors 1 and 3 will continue to be assigned to cluster *, and vectors 2 and 4 will continue to be assigned to cluster ^.

No more changes will occur in subsequent iterations, so we are done.

Soft K-Means / Fuzzy K-Means

One thing we find when we do K-Means is that it's highly sensitive to its initialization.

One strategy often employed is just restarting K-Means multiple times and using whichever result gives us the best final cost. We'll see how we calculate this cost in the next section.

What does this tell us? It tells us that the cost function is susceptible to local minima.

One way we can overcome this challenge is by having "fuzzy" membership in each class.

This means each data point doesn't fully belong to one class or another, but rather, there is an "amount" of membership. For example, it may be 60% part of cluster 1, and 40% part of cluster 2.

You'll see that we can get soft k-means with just a small adjustment to the

regular k-means algorithm.

The first part of this is the same, we initialize the k cluster centers to random points in the dataset.

But it's the inside of our main loop that changes.

Step 1 becomes a calculation of cluster responsibilities.

r(k,n) = exp[-b *d(m(k), x(n))] / sum[j=1..K] { exp[-b* d(m(j), x(n))] }

You can see here that r(k,n) will now always be a fraction, a number between 0 and 1, whereas you can interpret hard k-means or regular k-means to be the case where r(k,n) is always exactly equal to 0 or 1.

d(*,*) can be any valid distance metric, but the Euclidean or squared Euclidean distance are used most often.

Step 2 is very similar to hard k-means - we just recalculate the means based on the responsibilities.

$$m(k) = \sum_{n=1..N} \{ r(k,n) * x(n) \} / \sum_{n=1..N} \{ r(k,n) \}$$

You can see that this is like sort of a weighted mean. If r(k,n) is higher, that means this x(n) matters more to this cluster k, which means it has more influence on the calculation of its mean.

K-Means Objective Function

As with supervised learning, it's important to talk about the objective functions that we are maximizing.

Assuming we're using the Euclidean distance as our distance measure, our objective function is:

J = sum[n=1..N] { sum[k=1..K] { r(k,n) || m(k) - x(n) ||^2 } }

Which is just the squared distance weighted by the responsibilities.

So if x(n) is far away from the mean of cluster k, hopefully that responsibility has been set very low.

If you've taken any of my deep learning classes, you know that the way we optimize an objective function is with gradient descent.

Yet here we did not do any gradient descent!

In fact, what we do here is called **coordinate descent**. It means we are moving in the direction of a smaller J with respect to only one variable at a time.

You can see that this is true because we only update one variable at a time (either r(k,n) or m(k)).

There is a mathematical guarantee that each iteration will result in the objective function decreasing, and thus, it will always converge, *however*, as we previously discussed, there is no guarantee that it will converge to a global minimum.

Soft K-Means in Code

In this section we will look at how to implement soft k-means in Python code. Note that you can grab this code, and all the other code in this book, from https://github.com/lazyprogrammer/machine_learning_examples/tree/master/unsu

We first begin with standard imports and utility functions: the squared Euclidean distance and cost function.

```
import numpy as np

import matplotlib.pyplot as plt


def d(u, v):

diff = u - v

return diff.dot(diff)


def cost(X, R, M):

cost = 0
```

for k in xrange(len(M)): for n in xrange(len(X)): cost += R[n,k]*d(M[k], X[n]) return cost

Next, we define a function to run the k-means algorithm and plot the result (i.e. a scatterplot where the color represents the amount of membership in a cluster).

def plot_k_means(X, K, max_iter=20, beta=1.0): N, D = X.shape

M = np.zeros((K, D)) R = np.zeros((N, K))

initialize M to random for k in xrange(K): M[k] = X[np.random.choice(N)]

costs = np.zeros(max_iter) for i in xrange(max_iter): # step 1: determine assignments / resposibilities # is this inefficient?

for k in xrange(K): for n in xrange(N): R[n,k] = np.exp(-beta*d(M[k], X[n])) / np.sum(np.exp(-beta*d(M[j], X[n])) for j in xrange(K))

step 2: recalculate means for k in xrange(K): M[k] = R[:,k].dot(X) / R[:,k].sum()

costs[i] = cost(X, R, M) if i > 0:

if np.abs(costs[i] - costs[i-1]) < 0.01: break

```
plt.plot(costs)

plt.title("Costs")

plt.show()
```

```
random_colors = np.random.random((K, 3)) colors = R.dot(random_colors)
plt.scatter(X[:,0], X[:,1], c=colors) plt.show()
```

Notice that both M and R are matrices. R is naturally a matrix because it contains 2 indices: k and n. M is also a matrix, because it contains K individual D-dimensional vectors.

The beta variable controls how “spread out” or “fuzzy” the cluster memberships are. This is called a “hyperparameter”.

The random_colors matrix is Kx3 because there are 3 color channels (RGB) and we need K of them to represent the K cluster centers.

Now let us create a main function that will create random clusters and then call

the function we defined above.

def main():

3 means

D = 2 # so we can visualize it more easily s = 4 # separation to control how far apart the means are mu1 = np.array([0, 0]) mu2 = np.array([s, s]) mu3 = np.array([0, s])

N = 900 # number of samples X = np.zeros((N, D)) X[:300, :] = np.random.randn(300, D) + mu1

X[300:600, :] = np.random.randn(300, D) + mu2

X[600:, :] = np.random.randn(300, D) + mu3

what does it look like without clustering?

plt.scatter(X[:,0], X[:,1]) plt.show()

K = 3 # luckily, we already know this plot_k_means(X, K)

K = 5 # what happens if we choose a "bad" K?

```
plot_k_means(X, K, max_iter=30)

K = 5 # what happens if we change beta?

plot_k_means(X, K, max_iter=30, beta=0.3)

if __name__ == '__main__': main()
```

Try this yourself and see what it looks like!

Example of Where K-Means can Fail

As an exercise, I would try some different, non-Gaussian cloud datasets and see how k-means performs.

It is easy to understand why k-means fails in these cases once we understand Gaussian mixture models.

In the following code we have 3 examples:

1) The donut problem. One circle inside another circle - intuitively we know that the inner circle is one “cluster” or class and the outer circle is another cluster / class.

2) Elongated clusters. K-means works on circular / spherical clusters, but not elongated ones. Why?

3) Clusters of different density.

Note that the code below assumes we’ve saved the above code in a file called

kmeans.py.

import numpy as np from kmeans import plot_k_means

def donut():

N = 1000

D = 2

R_inner = 5

R_outer = 10

distance from origin is radius + random normal # angle theta is uniformly distributed between (0, 2pi) R1 = np.random.randn(N/2) + R_inner theta = 2*np.pi*np.random.random(N/2) X_inner = np.concatenate([[R1 *np.cos(theta)]*, *[R1* np.sin(theta)]]).T

R2 = np.random.randn(N/2) + R_outer theta = 2*np.pi*np.random.random(N/2) X_outer = np.concatenate([[R2 *np.cos(theta)]*, *[R2* np.sin(theta)]]).T

```
X = np.concatenate([ X_inner, X_outer ]) return X




def main():

# donut

X = donut()

plot_k_means(X, 2)

# elongated clusters X = np.zeros((1000, 2)) X[:500,:] =
np.random.multivariate_normal([0, 0], [[1, 0], [0, 20]], 500) X[500:,:] =
np.random.multivariate_normal([5, 0], [[1, 0], [0, 20]], 500) plot_k_means(X, 2)

# different density X = np.zeros((1000, 2)) X[:950,:] = np.array([0,0]) +
np.random.randn(950, 2) X[950:,:] = np.array([3,0]) + np.random.randn(50, 2)
plot_k_means(X, 2)




if __name__ == '__main__': main()
```

Disadvantages of K-Means

We've already discussed a few of the disadvantages of k-means. In this section we are going list them out explicitly and talk about how you can maybe overcome them.

The first glaring issue is that you have to choose K! I was nice and showed you that our data came from 3 Gaussians, so we knew to choose K=3. But what if you didn't know what K was, or your dimensionality was too high so you couldn't plot it on a scatterplot? Then you have a high chance of your clusters looking really weird.

The second disadvantage, which we've mentioned is that k-means only converges to local minima. Well this is the same thing we have with deep learning, so it's not *necessarily* always a bad thing. However, with k-means, it is a bad thing, as we have seen.

One way to remedy this is to restart k-means several times, and to choose the clustering that gives you the best value for the objective. Unfortunately, this bring us to another disadvantage of k-means, which is that it's very sensitive to the initial configuration.

At the very least, each run of k-means can be run independently, so there's some

opportunity for parallelization.

Another disadvantage we just saw is that k-means can't solve problems like the donut problem. In fact, k-means can't even solve regular shapes like ellipses either. As you've seen, k-means only takes into account squared distance, which means it tends to look for spherical clusters.

If you don't understand how this works now, don't worry too much, since I will cover it more in-depth when we talk about Gaussian mixture models.

Another problem with k-means is that it doesn't take into account the density of the data in the cluster. This one is more subtle. Think about the fact that it takes the straight mean. If there's a really dense part where a lot of the data resides, the mean is going to be closer to that dense area.

Chapter 3: Hierarchical Clustering

In this chapter we are going to talk about a different way of building clusters called hierarchical clustering.

Specifically we are going to talk about the **agglomerative clustering** algorithm.

If you've ever studied algorithms, you'll recognize this as a "greedy" algorithm.

We are going to purposely be short-sighted and make what appears to be the best decision at the time.

In the following sections I will show you the basic agglomerative clustering algorithm, and after you get a good feel for how it works, we are going to focus on using libraries that show you how to use hierarchical clustering, rather than trying to code it ourselves.

The reason is, there are many variations, and the last step is a visual interpretation, which we would need a library to make anyway.

Let’s first look at the algorithm visually. So we have some data points, a,b,c,d,e and f.

a

b d f

c e

We’ll group these points based on a greedy strategy. We can see that b+c, and d+e are closest together, so we merge those.

(a) (b) (c) (d) (e) (f)

\ \

(b,c) (d,e)

Next, we see that f is closest to the cluster d+e, so we merge those into d/e/f.

(a) (b) (c) (d) (e) (f)

\ \ /

(b,c) (d,e) /

\ /

(d,e,f)

Next, the closest thing to the d/e/f cluster is the b/c cluster, so we merge those into b/c/d/e/f.

(a) (b) (c) (d) (e) (f)

\ \ /

(b,c) (d,e) /

\ \ /

\ (d,e,f)

\ /

(b,c,d,e,f)

Finally, we merge the big cluster with the “a” which is still by itself, to one final large cluster.

(a) (b) (c) (d) (e) (f)

\ \ \ /

\ (b,c) (d,e) /

\ \ \ /

\ \ (d,e,f)

\ \ /

\ (b,c,d,e,f)

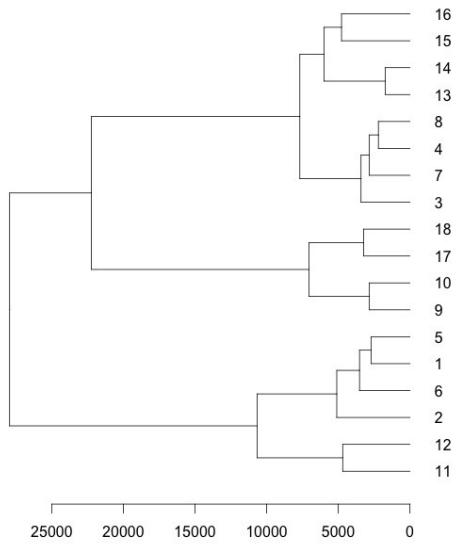
\ /

(a,b,c,d,e,f)

So at this point, you might ask, well, where are the actual clusters? Which ones do I choose?

To answer this question, we have to make a plot of the clustering process, and this is what's called a **dendrogram**.

Here is an example of a dendrogram:



Basically the height of each node in the tree is proportional to the distance of the 2 clusters that it joins together.

Height of cluster 1-2 = D(cluster1, cluster2)

So if you see a large gap, you know that those 2 clusters were far apart, and hence you could consider them separate.

Of course at some point you need to make a judgment based on what you see visually, or maybe use a numerical criterion like a threshold.

Hierarchical Clustering Options

At this point we have breezed by k-means and hierarchical clustering, and to look at them more in-depth we need to look at what kind of assumptions those algorithms make and what other options we have in place of those assumptions.

The first thing we'll look at is distance metrics. We've assumed so far that we were using Euclidean distance, but this doesn't necessarily have to be the case.

There are other distance metrics that may work better. It's up to you to experiment and see which one gives you the best results.

Here are some popular distance metrics:

Euclidean distance

d(u,v) = sqrt((u1-v1)^2 + (u2-v2)^2)

Squared Euclidean distance

$d(u,v) = (u1-v1)^2 + (u2-v2)^2$

Manhattan distance

$d(u,v) = |u1-v1| + |u2-v2|$

Maximum distance

$d(u,v) = \max\{i\} |u(i) - v(i)|$

Mahalanobis distance

$d(u,v) = \text{sqrt}((u - v)^T S^{-1} (u - v))$

(where S is the Covariance matrix)

You are also free to invent your own distance metrics, but there are some criteria that define what a valid metric is.

1.non-negativity or separation axiom

$d(u,v) >= 0$

2. identity of indiscernibles

$d(u,v) = 0$ implies $u = v$

3. symmetry

$d(u,v) = d(v,u)$

4. subadditivity or triangle inequality

$d(u,w) <= d(u,v) + d(v,w)$

Now we're going to talk about the different ways of joining clusters together.

Since we didn't go too in-depth in the last section, we still don't actually know *how* to decide on the distance between 2 clusters. Do we take the mean? The median? The 2 closest points? Or the 2 furthest points? In fact, these options all have names, which we will discuss.

The first is called single linkage. This is where we look at each point in cluster 1, and find the closest point in cluster 2. The minimum of all those points is the distance between the clusters. In code it might look like this.

```
min_dist = infinity

for p1 in cluster1:

for p2 in cluster2:

min_dist = min( d(p1, p2), min_dist )
```

One downside to this method is that you might get something called the "chaining effect", where we just keep choosing the next thing that is beside our current cluster, but we end up with something where in total all the things are far apart.

The opposite of this is “complete-linkage” clustering. This is where we find the maximum distance between all possible points in the 2 clusters, and assign that as the distance. Here is how you might do that in code

```
max_dist = 0

for p1 in cluster1:

for p2 in cluster2:

max_dist = max( d(p1, p2), max_dist )
```

The third type of cluster distance measuring is probably the most intuitive one, which is just to take the mean distance. This is also called UPGMA. This is what it might look like in code:

```
dist = 0

for p1 in cluster1:

for p2 in cluster2:

dist += d(p1, p2)
```

dist = dist / (len(cluster1) * len(cluster2))

The last type we'll talk about is called Ward's criterion.

This is where we look at each pair of clusters and see how much the variance would increase if we joined them together.

You can imagine that if clusters are very far apart, after joining them together, the variance will be very large.

So we'll avoid that by joining clusters that are closer together.

Hierarchical Clustering in Code

In the following code, we take the same data that we had in our k-means example, and use Ward linkage, single linkage, and complete linkage to compute the hierarchical clusters and plot the dendrograms.

import numpy as np import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import dendrogram, linkage

def main():

D = 2 # so we can visualize it more easily s = 4 # separation to control how far apart the means are mu1 = np.array([0, 0]) mu2 = np.array([s, s]) mu3 = np.array([0, s])

N = 900 # number of samples X = np.zeros((N, D)) X[:300, :] = np.random.randn(300, D) + mu1

X[300:600, :] = np.random.randn(300, D) + mu2

X[600:, :] = np.random.randn(300, D) + mu3

Z = linkage(X, 'ward') print "Z.shape:", Z.shape # Z has the format [idx1, idx2, dist, sample_count]

therefore, its size will be (N-1, 4) plt.title("Ward")

dendrogram(Z)

plt.show()

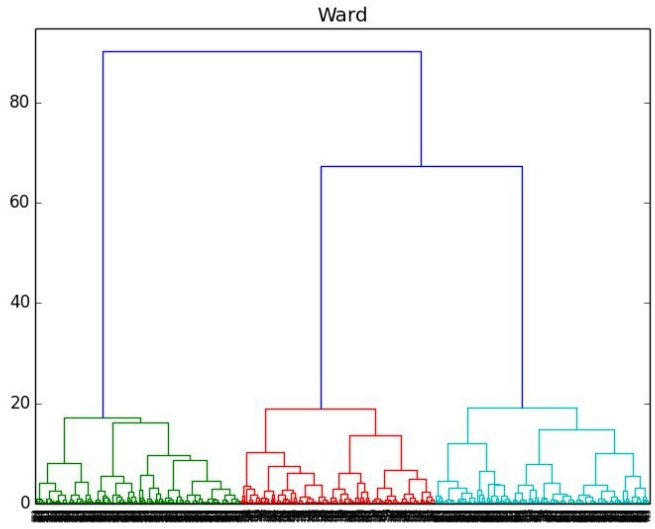
Z = linkage(X, 'single') plt.title("Single") dendrogram(Z)

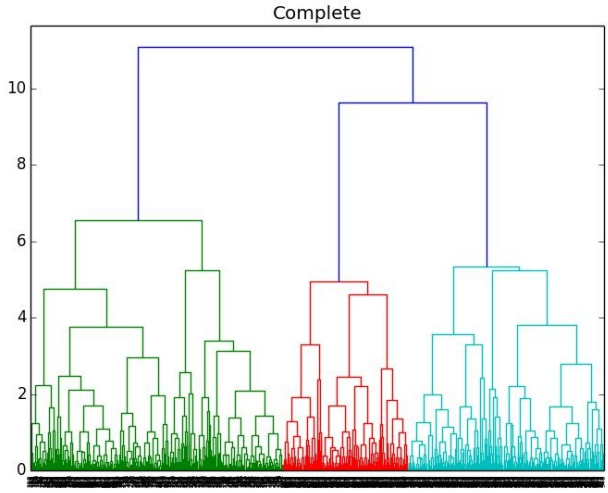
plt.show()

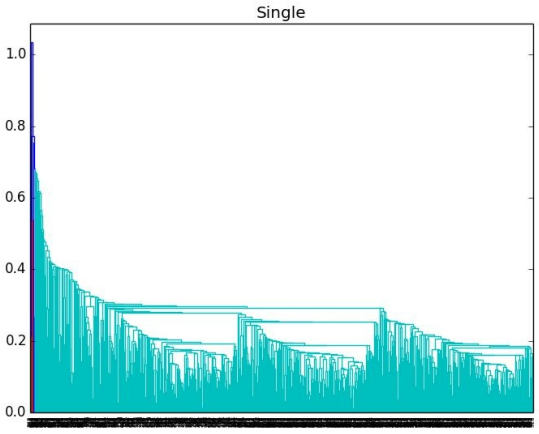
Z = linkage(X, 'complete') plt.title("Complete") dendrogram(Z)

plt.show()

if __name__ == '__main__': main()


Ward
80
60
40
20
0


Complete
10
8
6
4
2
0


Single
1.0
0.8
0.6
0.4
0.2
0.0

We see that both Ward linkage and complete linkage correctly identify the 3 clusters that are far apart, however Ward does a little better. Single linkage completely fails and exhibits the chaining effect.

Chapter 4: Gaussian Mixture Models

The next topic we'll talk about is Gaussian mixture models. Gaussian mixture models are a form of density estimation. They give us an approximation of the probability distribution of our data.

We use Gaussian mixture models when we notice our data is multi-modal, meaning, there are multiple modes or bumps in the histogram. If you remember from probability, the mode is the most common value.

A Gaussian mixture is just the sum of weighted Gaussians.

To represent these weights we'll introduce a new symbol called pi. We'll say pi(k) is the probability that x belongs to the kth Gaussian.

Since pi(k) is a probability, there is a constraint that all the pi's have to sum to 1.

Another way of thinking of this is that we introduced a new latent variable called "z". "Z" represents which Gaussian the data came from. So we can say pi(k) = P(z = k).

It’s like saying there’s some hidden cause called “Z” that we don’t know about and can’t measure, but that each of these “Z”s is causing a Gaussian to be generated, and all we can see in our data is the combined effects of those individual “Z”s.

Training a GMM is very much like the K-Means algorithm. There are 2 steps that mirror what we saw with k-means.

Step number 1 is to calculate the responsibilities.

r(k,n) is the responsibility of the kth Gaussian for generating the nth point. So it’s just the proportion of that Gaussian, divided by all the Gaussians. You can see that if pi(k) is large here, then it will overtake the other Gaussians, and this will be approximately equal to 1.

r(k,n) = pi(k)N(x(n), mu(k), C(k)) / sum[j=1..K]{ pi(j)N(x(n), mu(j), C(j)) }

Where I’m using C(k) to mean the covariance of the kth Gaussian. N(x, mu, C) means the PDF (probability density function) of the Gaussian of the data point x and the mean mu and covariance C.

Step number 2 is to recalculate all the parameters of the Gaussians - meaning the means and covariances, and the pi's. The way we do this is also similar to k-means where we weight each sample's influence on the parameter by the responsibility. If the responsibility is small, then that "x" matters less in the total calculation.

Define N(k) as $N(k) = \text{sum}[n=1..N]\{ r(k,n) \}$

Then each parameter update is:

$$\mu(k) = \text{sum}[n=1..N]\{ r(k,n)x(n) \} / N(k)$$

$$C(k) = \text{sum}[n=1..N]\{ r(k,n)(x(n) - \mu(k))(x(n) - \mu(k))^T \} / N(k)$$

$$\pi(k) = N(k) / N$$

Comparison with Soft K-Means

In this sectiom we'll compare Gaussian Mixture Models and Soft K-Means side by side.

This will help you not only appreciate the similarities between the two, but also to understand the limitations of K-means and why they exist.

So let's compare the 2 steps of each training algorithm. You see that the first step in both is to calculate the responsibilities, and the second step in both is to calculate the model parameters.

K-Means:

r(k,n) = exp[-b *d(m(k), x(n))] / sum[j=1..K] { exp[-b* d(m(j), x(n))] }

GMM:

r(k,n) = pi(k)N(x(n), mu(k), C(k)) / sum[j=1..K]{ pi(j)N(x(n), mu(j), C(j)) }

The first thing is we now understand why K-Means looks for clusters of equal weight. It's because it has no "pi" variable, which is equivalent to saying pi is uniform, or equal to 1/K.

Note that this means GMMs are limited in the same way as K-Means in that you still have to choose K.

The second thing to notice is that K-Means has this beta variable where GMMs have the full covariance matrix. This allows GMMs to have a lot more flexibility in the shape of its distribution. With K-Means, because you only have this one beta, it means all your clusters have to be spherical. With a full covariance matrix, you can have any type of elliptical shape in any orientation.

(Beta is like the inverse of the variance)

Now let's look at the parameter updates:

K-Means:

mu(k) = sum[n=1..N] { r(k,n)x(n) } / sum[n=1..N] { r(k,n) }

GMM:

$$\mu(k) = \sum_{n=1..N}\{ r(k,n)x(n) \} / N(k)$$

Notice that the equation for the mean is exactly the same.

Only the GMM has these updates:

$$C(k) = \sum_{n=1..N}\{ r(k,n)(x(n) - \mu(k))(x(n) - \mu(k))^T \} / N(k)$$

$$\pi(k) = N(k) / N$$

So by allowing a full covariance, we can have non-spherical clusters, meaning they can be elliptical or elongated, where we observed k-means can fail.

The pi(k) variable allows each cluster to have a different density / number of points.

In conclusion, you can think of soft k-means as a GMM where each cluster has the same weight, and each cluster is spherical with the same radius.

Coding a GMM

Now we know enough to jump straight into coding a GMM. We're going to use the same dataset we used on k-means, except that we will change the density and variance of each cluster.

import numpy as np import matplotlib.pyplot as plt

from scipy.stats import multivariate_normal

def gmm(X, K, max_iter=20): N, D = X.shape

M = np.zeros((K, D)) R = np.zeros((N, K)) C = np.zeros((K, D, D)) pi = np.ones(K) / K # uniform

initialize M to random, initialize C to spherical with variance 1

for k in xrange(K): M[k] = X[np.random.choice(N)]

C[k] = np.eye(D)

costs = np.zeros(max_iter) weighted_pdfs = np.zeros((N, K)) # we'll use these to store the PDF value of sample n and Gaussian k for i in xrange(max_iter): # step 1: determine assignments / resposibilities for k in xrange(K): for n in xrange(N):

weighted_pdfs[n,k] = pi[k]*multivariate_normal.pdf(X[n], M[k], C[k])

for k in xrange(K): for n in xrange(N): R[n,k] = weighted_pdfs[n,k] / weighted_pdfs[n,:].sum()

step 2: recalculate params for k in xrange(K): Nk = R[:,k].sum()

pi[k] = Nk / N

M[k] = R[:,k].dot(X) / Nk C[k] = np.sum(R[n,k]*np.outer(X[n] - M[k], X[n] - M[k]) for n in xrange(N)) / Nk + np.eye(D)*0.001

costs[i] = np.log(weighted_pdfs.sum(axis=1)).sum() if i > 0:

if np.abs(costs[i] - costs[i-1]) < 0.1: break

plt.plot(costs)

plt.title("Costs") plt.show()

random_colors = np.random.random((K, 3)) colors = R.dot(random_colors) plt.scatter(X[:,0], X[:,1], c=colors) plt.show()

```
print "pi:", pi

print "means:", M

print "covariances:", C




def main():

# assume 3 means

D = 2 # so we can visualize it more easily s = 4 # separation to control how far apart the means are mu1 = np.array([0, 0]) mu2 = np.array([s, s]) mu3 = np.array([0, s])

N = 2000 # number of samples X = np.zeros((N, D)) X[:1200, :] = np.random.randn(1200, D)*2 + mu1

X[1200:1800, :] = np.random.randn(600, D) + mu2

X[1800:, :] = np.random.randn(200, D)*0.5 + mu3



# what does it look like without clustering?

plt.scatter(X[:,0], X[:,1]) plt.show()
```

plt.scatter(X[:,0], X[:,1]) plt.show()

K = 3

gmm(X, K)

if __name__ == '__main__': main()

Singular Covariance Problem

One of the biggest problems with the GMM is the singular covariance problem.

To get a better intuition for why this is a problem, think of a 1-D Gaussian for a second.

Imagine that all the points are very close together, so that the variance of the Gaussian is almost 0.

Well, in the Gaussian formula, we actually divide by the variance. So if the variance gets too close to 0, what ends up happening is we divide by 0, which gives us a singularity.

You can think of inverting a matrix as taking “1 over” - since something times 1 over itself will be 1, or identity.

In multiple dimensions it’s the same problem - if your covariance is too small, the inverse will approach infinity.

This is yet another disadvantage of falling into local minima.

So what can we do about this?

One solution is to use what we call diagonal covariances.

Diagonal covariances are very useful not only for avoiding the singular covariance problem, but also for speeding up computation.

When you have a diagonal covariance, it’s really easy to take the inverse.

You just take every element that’s not 0, which are all along the diagonal, and you invert it, so you take 1 over that element.

You can try this yourself and verify that multiplying these two would give you the identity matrix.

You also now only need to store a D x 1 vector to represent the covariance instead of a DxD matrix.

The assumption you're making when you use a diagonal covariance is that each of your dimensions is independent. To see why this is, let's consider the definition of covariance.

$$C(i,j) = E[(x(i) - m(i))(x(j) - m(j))]$$

Since x(i) and x(j) are independent when i != j, we can split the expectations.

$$C(i,j) = E[(x(i) - m(i))] E[(x(j) - m(j))] = [m(i) - m(i)] [m(j) - m(j)] = 0$$

You can also think of diagonal covariance as a sort of regularization. You're making your model simpler by using less parameters.

Sometimes, even when you use diagonal covariance you still get singularities.

In that case, you might want to use a spherical Gaussian. This is where we use the same variance along every dimension.

This sounds like a pretty ridiculous assumption but in reality it can work well. As a bonus it’s also less computationally expensive.

Now your covariance is represented by a scalar instead of a Dx1 vector. Even more savings!

The last option is shared covariance. This is when you assume each Gaussian just has the same covariance. You can calculate it by just taking the full covariance over the entire dataset.

Chapter 5: Principal Components Analysis

In this chapter we are going to talk about PCA or principal components analysis. There are 2 components to their description - first describing what PCA does and how it is used, and then second is the math behind PCA.

So what does PCA do? Firstly, it is a linear transformation.

If you think about what happens when you multiply a vector by a scalar, you'll see that it never changes direction. It only becomes a vector of different length. Of course, you could multiply it by negative 1 and it would face the opposite direction, but it can't be rotated arbitrarily.

If you multiply a vector by a matrix - it *can* change direction and be rotated arbitrarily.

So that is what PCA does - it takes an input data matrix X, which is NxD, multiplies it by a transformation matrix Q, which is DxD, and you can back some transformed data Z which is also NxD.

In matrix form: $Z = XQ$

In vector form: $z = Qx$

If you want to transform an individual vector x to a corresponding individual vector z, that would be $z = Qx$. It's in a different order because when x is in a data matrix it's a 1xD row vector, but when we talk about individual vectors they are Dx1 column vectors.

What makes PCA an interesting algorithm is how it chooses the Q matrix.

Notice that because this is unsupervised learning, we have an X but no Y, or no targets.

Changing coordinate systems

Another view of what happens when you multiply by a matrix is not that you are are rotating the vectors, but you instead are rotating the coordinate system in which the vectors live.

Which view we take will depend on which problem we are trying to solve.

Dimensionality reduction

One use of PCA is dimensionality reduction. When you're looking at the MNIST dataset, which is 28x28 images, or vectors of size 784, that's a lot of dimensions, and it's definitely not something you can visualize.

Note that 28x28 is a very tiny image, and most images these days are much larger than that - so we either need to have the resources to handle data that can have millions of dimensions, or we could reduce the data dimensionality using techniques like PCA.

We of course can't just take arbitrary dimensions from X - we want to reduce the data size but at the same time capture as much information as possible.

So if we want to go from 784 to 2 dimensions - so that we can visualize it - we want those 2 dimensions to have as much information from X as possible.

How do we measure this information? In traditional PCA and many other traditional statistical methods we use variance. If something varies more, it carries more information.

You can imagine the opposite situation, where a variable is completely deterministic, as in, it has no variance. Then measuring this variable would not give us any new information, because we already knew what it was going to be.

So when we get our transformed data Z, what we want is for the first column to have the most information, the second column to have the second most information, and so on.

i.e.:

var(Z[:,0]) >= var(Z[:,1]) > = var(Z[:,2]) >= ...

So when we take the 2 columns which the most information, that would mean taking the first 2 columns.

De-correlation

Another thing PCA does is de-correlation.

You can imagine that correlations mean some of the data is redundant, because we can predict one column from another column.

Imagine a dataset on which you could use linear regression - predicting the x2 point from the x1 point, along with Gaussian noise.

If you take the coordinate system rotation view (i.e. (x1, x2) -> (x1', x2')), you can imagine that if we rotated our coordinate system to be aligned with the spread of these points, then the data would become uncorrelated - we should no longer to be able to predict x2' from x1'.

So again the question is - how do we find Q so that we rotate the data in exactly this way?

Visualization

Once we have the information of our data sorted in descending order in each dimension, we can just take the top 2 and make a scatter plot. This will then give us an idea of the separation of the data points, and it allows us to create a visual representation of high-dimensional data.

i.e. plt.scatter(Z[:,0], Z[:,1])

Data pre-processing and denoising

The last application of PCA we'll talk about is pre-processing and overfitting.

You can imagine that our data is often noisy. Hopefully, the noise is small compared to the true pattern. In that case, the variance of the noise, which should be small, would go into the last columns of our transformed data Z, at which point we could just discard it.

We could then feed this new data into a supervised machine learning model such as logistic regression - in fact we did this in my previous courses.

So by getting rid of the noise we are preventing overfitting by making sure we

are not fitting to the noise.

You'll see that this idea of unsupervised pre-training will come into play again when we study autoencoders and RBMs.

Latent variables

Another view of PCA is that the transformed variables Z are the "latent variables", as in they are some sort of underlying cause of the data X.

Then it makes sense that they should be uncorrelated, because they are just independent hidden causes.

It also makes sense that some of the data in X *is* correlated because they are just measurements you're taking of some data that is produced by a combination of those hidden causes.

What we are assuming when we do PCA is that the data is a linear combination of those hidden causes.

In fact with PCA the linearity goes both ways - the latent variable Z is a linear combination of the observed variable X, but if you were to do a reverse transformation, the observed data X is also a linear combination of the latent variable Z.

$$Z = XQ$$

$$X = ZQ^{-1}$$

You'll recall that we first encountered the idea of latent variables in clustering and Gaussian mixture models - because those models assumed that the identities of the clusters were the latent variable.

PCA derivation

In this section we are going to look at the math behind PCA.

A lot of the steps here may seem arbitrary at first, but you'll see how it all fits together in the end and results in all the properties that I talked about in the previous section.

So the first step is to calculate the covariance of X.

Remember that the definition of covariance is just the expected value of (x_i - mu_i) x (x_j - mu_j). If i = j then it's just the regular variance.

This gives us a DxD matrix.

$C_X(i,j) = E[(x(i) - mu(i)) (x(j) - mu(j))]$

Sample covariance in matrix form:

$C_X = (X - mu)^T(X - mu) / N$

Next you know that a DxD matrix has D eigenvalues and D eigenvectors. Now if you don't know what eigenvalues and eigenvectors are, I'm going to give you a short introduction.

Remember that matrices in general change the direction of, or rotate, a vector. Eigenvectors of a matrix are special vectors which are NOT rotated by the matrix, but just change in length. The change in length is called the eigenvalue.

So we can relate the covariance matrix, its eigenvector, and its eigenvalue, using this equation, where lambda is the eigenvalue and v is the eigenvector.

C_X v = lambda v

There are some theorems which we won't prove, but basically there will be D eigenvectors and D corresponding eigenvalues, and the eigenvalues will be greater than or equal to 0.

Finding eigenvectors and eigenvalues is itself not a trivial task, and there are many algorithms that can do this, including gradient descent. Since numpy already has a function to do this, we're not going to worry about it - just the theory is important to give you the right perspective.

So now we have these D eigenvalues, what do we do with them?

Again an arbitrary step, but let's sort the eigenvalues in descending order. This means that the corresponding eigenvectors have to be sorted in the same way.

i.e. lambda(1) >= lambda(2) >= lambda(3) >= ...

Once we've done this, we can put them into matrices of size DxD. So the eigenvalues will go in a diagonal matrix of size DxD we'll call "big lambda", and the eigenvectors will be lined up beside each other in a matrix we'll call V.

i.e. V = [v1, v2, ..., vD]

Lambda(i, i) = lambda(i), and Lambda(i, j) = 0 for i != j

It's easy to prove to yourself that $C_X V = V$ Lambda using a 2-D example, so I will not do this. I mention it because it looks backwards, given that when we talk about a single eigenvector and eigenvalue the eigenvector is always on the right.

$C_X V = V$ Lambda

One last ingredient is that the matrix V is orthonormal. This means that any eigenvector dotted with itself is 1, and any eigenvector dotted with another different eigenvector is 0.

Because of this, $V^{-1}V = I = V^{T}V$, so $V^{-1} = V^{T}$

Finally, in the last step, we'll now look at the transformed data, Z. Remember that we still don't know what Q is. But let's solve for its covariance. Notice how we can express the covariance of Z in terms of the covariance of X.

$C_Z = Q^{T}(X-mu)^{T}(X-mu)Q / N = Q^{T}C_XQ$

Next, look what happens if we choose Q = V.

$C_Z = V^{T}C_XV = V^{T}V$ Lambda = Lambda

We get that the covariance of Z is just equal to "big lambda", which is the diagonal matrix of eigenvalues.

So what does this all mean?

Since all the off-diagonal elements of Lambda are 0, that means any dimension i is not correlated with any other dimension j, which means there are no correlations in the transformed data.

So by choosing Q = V, we've de-correlated Z.

Next, because we sorted big lambda by the eigenvalues in descending order, that means the first dimension of Z has the most variance, the second dimension of Z has the second most variance, and so on.

To make this more explicit, the variance of any dimension of Z is equal to the corresponding eigenvalue.

$C_Z(i,i) = \text{lambda}(i)$

Natural Language Processing

One application of PCA is visualizing a term-document matrix. A term-document matrix contains terms (words) along one axis and documents along the other axis.

So for example, one row could be a document, and each column represents a different word.

You can imagine that two documents about cats would be similar to each other, but very different compared to a document about physics.

Another way of thinking about this is that the word cats appears in many of the same documents, but doesn't appear in many documents containing the word physics.

So there are some correlations to be found.

Exercise: Try creating a term-document matrix from a dataset like Wikipedia, and perform PCA on it. By plotting the first two principal components (what we've been calling Z), you should find that similar words appear on the same

axes / near each other.

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

A lot of the material in this book is covered in this course, but you get to see me derive the formulas and write the code live:

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to this book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

When you've got the basics of deep learning down, you're ready to explore alternative architectures. One very popular alternative is the convolutional neural network, created specifically for image classification. These have promising applications in medical imaging, self-driving vehicles, and more. In this course, I show you how to build convolutional nets in Theano and TensorFlow.

Deep Learning: Convolutional Neural Networks in Python

https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods. We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don’t forget to hit “like”!)