**Feature Extraction**

So if we have a high dimensional dataset, we might want to simplify matters by looking for ways to reduce its dimensionality. There are a few ways to proceed. One is Feature Selection, which has to do with dropping features from our dataset. The other methods have to do with Feature Extraction, which more or less keeps all of the features, but looks for a new reference frame with axes along which the data varies the most. Essentially it reorganizes the N features into a set of N new features (which will be linear combinations of the old features). Ultimately, usually, we will drop the least variant of the features of this new reference frame. And so this kind of employs Feature Selection too, in the end.

Well, this kind of Feature Selection describes PCA and NNMF, but not t-SNE. One of the advantages of Feature Selection and Feature Extraction is that we eliminate less predictive features (or linear combinations thereof), and so make overfitting our models less likely.

**Feature Extraction**

The other way to reduce dimensionality is feature extraction. This is combining features into new, more relevant, features, we’ll call features´. And once we have features´, we may employ feature selection again to keep just the most important ones. A simple example would be replacing left and right leg lengths with their average (if we were using this information to predict, say, height, instead of say, running injury frequency). Or if we have a dataset for predicting weight, we might replace height and weight with BMI = weight/height2. Datacamp guy mentions that we can use these techniques for image compression. For instance, he reduced a 2800 pixel image to 300 or so by using PCA. So let’s take a look at some formal ways to do this. We’ll start with PCA.

**Principal Component Analysis (PCA)**

So say we have some feature data consisting of m rows of feature vectors **r**i, in some n-dimensional space.



We want to plot this data along a new set of axes in this n-space. And we want to construct these axes to point along the directions of the data’s highest variability. If we can find these axes where variability is highest, then we may be able to reduce the number of variables in our analysis.

Chart, scatter chart, bubble chart

Description automatically generated

Before we do, it is customary to shift the origin of our coordinate system to lie at the center (average) of all the data **r**i – **μ** → **r**i. Equivalently, we’re shifting the rij coordinates rij – μj → rij, where μj = Σi=1…m (rij/m)

Chart, scatter chart, bubble chart

Description automatically generated

And then it is also best practice to scale the axes by standard deviation, σj = √[(1/m)Σi=1…m(rij – μj)2], of the data along that axis. So rij/σj → rij. Kind of looks the same,

Chart, scatter chart, bubble chart

Description automatically generated

So basically we’re normalizing each column of our data, as is somewhat standard, and, critical for PCA. Okay, and then we want to find the orthogonal axes, along which the data has the most variability. For illustrations sake with our dim n = 2 data, we’re looking for a mathematical way to find **u**1, and **u**2 here.

Chart

Description automatically generated

Datacamp guy makes a point that while Math Score and Reading Score are clearly correlated, and so contain somewhat duplicate information, the u1 score and u2 score are orthogonal, and so don’t contain duplicate information. And if we did a correlation matrix of u1 and u2 we should get close to 0’s.

Okay, back to the details. So first we have to quantify variability along a particular axis. So say we have a unit vector . We’ll define the variability of the data along that axis as the sum of the (squared) projections of the data onto that axis. We’ll call this the sum of the squared distances SSDu. And mathematically, it would just be:



Note m is the number of data points/rows we have. Instead of requiring **u** have unit length from the beginning, we could add it on as a constraint. So we’d say,



and then maximize this. Okay so [first index on r labels vector; second index labels component],



And then,



Going to switch α and β to make it look better.



Let’s extract the matrix R,



where <α| and |β> are unit vectors along those directions, and **r**i2 is an outer product of **r**i with itself. R is a symmetric matrix, as can see. So it should have n eigenvalues and eigenvectors. Also, R is almost the moment of inertia tensor. So if the data is symmetrically distributed about the x-axis, and also about the y-axis, then R would already be diagonal, and the eigenvectors would be along the x and y axes. Well we can write R in terms of X:



So,



And then we can write our eigenvalue equation as:



Parenthetically, we know that the eigenvalues and eigenvectors of XTX show up in the singular value decomposition of X (see Linear Algebra Folder). And I think there are routines that efficiently find that decomposition. So I guess this is how it is done in practice?

But anyway, as discussed in Physics Notes/Quantum Mechanics/Variational Principle, we know that the largest eigenvalues λ1 will correspond to the principal component (eigenvector) **u**1. Then next largest eigenvalue λ2 will correspond to the 2nd principal component (eigenvector) **u**2, which is the eigenvector which has the largest distance and is yet perpendicular to **u**1. The next largest eigenvalue after that, λ3, will correspond to the eigenvector **u**3 with the largest distance which is perpendicular to **u**1 and **u**2, etc. It turns out we will want to rewrite the data points in terms of the coordinate system defined by the new eigenvectors. But before we get into it, let’s consider the following fact. Let the total squared distance be defined as the sum of the SSDu\_j’s.



Then it turns out this quantity is actually independent of **u**, given that the **u**’s form an orthonormal set. We have:



where ri2 is the squared magnitude of the vector **r**i. Now recalling,



It is evident we can also say,



where Tr denotes the Trace function. And furthermore since we can express R as R = UΛUT, where U is the matrix of eigenvector columns, and Λ the diagonal matrix of eigenvalues, we can use the cyclical property of traces to assert that SSD is just the sum of the eigenvalues:



So we have:



Note that all the λ’s are guaranteed to be non-negative since R = XTX is a positive semi-definite matrix. One more thing. Given that SSD is invariant, we can define the proportion of the variance explained by eigenvector **u** as SSDu/SSD. And furthermore, this can be simplified to:



So we can say,



Keeping this in mind, let’s re-express our coordinates in terms of the eigenvectors. So we’ll say,



The primed components are illustrated in blue for instance,

A diagram of math scores

Description automatically generated

So altogether, we have:



And we can call the matrix of new components, X´,



Might want to know what X´ is in terms of X. Well, we can introduce the resolution of identity in terms of the axis directions of the original data **1** = Σkkk, where k runs from 1 to n.



Now Xik is the matrix of row data in the original feature vector space, and Ukj is the matrix of eigenvectors,



And so we can write,



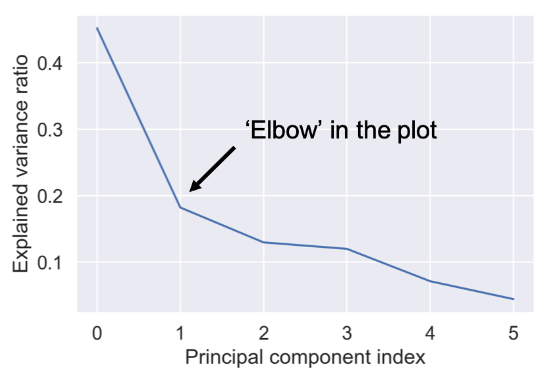
Now say we want a reduced representation. Say we only care about the first two principal components **u**1 and **u**2. Then we could approximate X´ by:



(and 0 is the value we’d use for r´i,j>2 because data should be centered at the origin even in the new coordinate system – it’s just rotated/permuted w/r to the original I think?) So effectively, we’d just have a matrix of m data rows with just two columns. This would make the analysis much faster, and easier to understand, if accurate enough. We could estimate how faithfull a representation this approximate X´ is to X by calculating the % variation accounted for. This would be:



And of course we could generalize to keeping 3, or 4, etc. components instead of just 2. Working backwards, we may in this fashion determine how many principal components to keep if we want to account for a certain percentage of the variation. To ascertain what the ‘natural' number of components to keep is, we can do an elbow plot of explained variance ratio vs. number of principal components kept. Something like this:



(# principal components = 1 + principal component index). In this case, looks like 2 components is most natural. But the merits of trading complexity for accuracy has to be decided on a case by case basis.

Should be emphasized that keeping k principal component columns in X´ is not the same as keeping some k columns in X. No column in X´ is equal to a column in X. The columns of X´ are linear combinations of the columns of X.

**Appendix 1**

There seems to be an equivalent formulation of this procedure. We will again center the data (placing the origin at the average), and scale it so that the standard deviations along each axis are 1 I suppose. Now we make the projection along the u-axis, and sum the distances between all data points.



Can simplify,



(Einstein summation over repeated indices) and we can extend the sum to include i = j since that makes the term just go to zero. If we minimize this, then do we get the same equation? Well let’s form F, relaxing the constraint in SSD and imposing it with a Lagrange Multiplier,



And minimizing,



and proceeding,



Now since the average is at the origin, the last term in the bracket is zero. And we’re left with:



And so we have:



which is the same equation as we had above, except the eigenvalue is scalled by 1/m. So these two methods are equivalent. We can use other metrics besides the Euclidean one though. One popular one is the log difference. So we say distance between two points is: d = ln|a|-ln|b|. If we use this metric, then we’d obviously get a different R.

**Appendix 2**

After scaling the data, we might want to know something about how ‘polar’ the data in a colum is. We could calculate something like <x4>. Then data which is located at the ends will have a large value of <x4>, while data that is evenly spread throughout its range will have a smaller <x4> value.

**Appendix 3**

We can relate this to the NMF formalism down below. So we have:



(U is a real unitary matrix, so its inverse is the same as its transpose). So we have:



Now say we approximate X´ by its first two columns – presuming they be the largest and the variation represented by the other columns is negligible.



But when multiply this X´ by UT, we can see that the n-2 columns of zeros in X´ cancel out the bottom n-2 rows of UT. And in fact we’d get the same thing as X´UT by replacing them with:



and with these definitions, we can say,



as below in NMF. This would generalize accordingly if we kept more than the first two principal components.

**Appendix 4**

We can’t really apply PCA to categorical features, like Gender (though maybe can get away with One-Hot-Encoding?). But at the very least, we can set aside the categorical features, run PCA on the numerical ones, get back the n desired principal components of each numerical row vector (i.e., W), and concatenate with the categorical features again.

Saw this nice graph on Datacamp. The ANSUR dataset has a large number of categorical and numerical features. Guy ran PCA on the numerical ones and then concatenated with the categorical ones. Here he plotted the first two PCA components in x, y directions, and then colored according to gender. Can see that gender is fairly highly correlated with pc1 and pc2 as we can almost linearly separate the genders with a plane. So in might make sense to drop Gender as a feature, since we can *almost* express it in terms of pc1 and pc2.



**Non Negative Matrix Factorization (NNMF)**

This procedure attempts to factorize the matrix X (which must be non-negative) into two other matrices:



Can think of H as a matrix of basis vectors, and W as the components of these basis vectors, required to reconstruct X. If X is m×n´, then it constitutes m row vectors of dimensionality n´. And say we want to reduce the *intrinsic* dimensionality of X (by which we mean, number of independent components required to specify each row vector of X) from n´ to n. Then W will be an m×n matrix. And H will be a n×n´ matrix. W’s rows will constitute the n components of the basis vectors given by H’s rows, which when multiplied together will give us X’s rows.

This is clearer if we multiply it out. Let’s say X constitutes m = 5, n´ = 4 row vectors. And we want to reduce the intrinsic dimensionality of this set to n = 2. Then W will be a matrix of m = 5, n = 2 vectors components. Then H will be a set of n = 2, n´ = 4 row vectors.



When worked out, we have:



and in this form it is plain that we have expressed the four row vectors in X in terms of linear combinations of the two row vectors in H. And so we will have in a sense replaced X with W.

I’m guessing that since this must in general be impossible to do, this is why we are trying to execute this process with a numerical algorithm that just minimizes the mean squared error (or some other loss function) as much as possible. The mean squared error is:



by which we mean the sum of the squares of the elements of X-WH (perhaps divided by total number of elements). We can expect NMF to be a more faithful dimension reduction scheme than PCA. PCA finds the n axes with the greatest variations, and discards the rest. But NMF tries to fit all the variation within the n axes. Evidentally, we can think of the basis vectors of H – let’s call them **h**1, **h**2 for the sake of discusssion – as representing meaningful building blocks of the entire dataset to a much greater extent than we can the basis vectors of the approximate H found under the PCA rubric (see Appendix 3).

Like with PCA, we can’t really apply NNMF to categorical features, like Gender (though maybe can get away with One-Hot-Encoding?). But at the very least, we can set aside the categorical features, run NNMF on the numerical ones, get back the n desired principal components of each numerical row vector (i.e., W), and concatenate with the categorical features again.

**t-distributed Stochastic Neighbor Embedding (t-SNE)**

This has the same end as PCA and NNMF, but works differently. Apparently tSNE it works better the greater number of features we have. Typically we start with some high dimensional data set and try to represent it in a two or three dimensional space. For simplicity, I’ll look at a 2d data set and try to represent it in a 1d space. So let’s say we start with these data points,

A grid with red green and blue dots

Description automatically generated

And we’d like to represent it in 1d. We start by finding the distance, rij, between every pair of points. For instance, the distances r0,j and r7,j are illustrated below.

A diagram of a graph

Description automatically generated

Then for every point i = 0, 2, 3, …, N=9 in our ensemble, we map the distance ri,0, ri,1, ri,2, …, ri,9 (in our illustration) to conditional probabilities with a normal distribution curve:



where σ2is the variance of the normal distribution (and don’t know how this is chosen – maybe doesn’t matter?), and ci is a normalization factor chosen so that Σjpi|j = 1. We can see the points, j, far away from point i, will be associated with lower probabilities than those close by. So we can interpret pi|j as the probability that j is ‘close’ to i. We’d like to interpret pi|j as a joint probability distribution, pi,j, which would give the probability particles i and j are close. As it stands, though, we’ll note that we can’t interpret pi|j as such because while pi|j ought to equal pj|I (because the probability that particle i and j are close must be the same as the probability that particle j and i are close), according to our formula pi|j ≠ pj|i (because while ri,j = rj,i, the normalization factors ci and cj for pi|j and pj|I respectively, would be different). We redress this by constructing a joint probability distribution out of the conditional one,



Can see that pi,j = pj,I now. And it’s also normalized, as:



I don’t know why we construct the joint probability distribution this way. Seems like we could’ve just said pi,j = c/√(2πσ2)·exp(-rij2/2σ2) or something – c being the overall normalization constant. Oh well whatever. Now we want to use this pi,j probability distribution that encodes the distances between particles to map the particles to a lower dimensional space. So apparently we just kind of randomly dump all the particles into our lower dimensional space, something like this,

A line of circles with numbers

Description automatically generated

And now we recalculate the ri,j and feed them into a different probability distribution, a Student’s t distribution, I think with ν = 1 d.o.f. Not sure.



where ci is a normalization factor again (different value than the previous one) which ensures that Σjfi|j = 1. Anyway, it kind of looks like a normal distribution, but with longer tails. And not really sure why we’re using this distribution instead of the normal distribution again, or any other distribution for that matter. And we construct a joint probability distribution same as before,



which we can interpret as the probability that particles i and j are ‘close’ in this lower dimensional space. Now we move the particles in our 1D space around until we get the fi,j to match the pi,j as well as possible. The definition of ‘as well as possible’ is this. We construct the cross entropy function, from the Multivariable Logistic Classification file (in that file, there was also the parameter xi in p and f, but that is irrelevant to our discussion),



and if we subtract off the entropy of p itself, S(p) = -Σijpijln(pij), we obtain the Kullback-Liebler divergence:



This quantifies the loss in information we get by substituting the approximate distribution f for the actual distribution p. Now that we have this, we would shuffle the locations of the points in our 1D space (thereby changing the rij, thereby changing the fij) so as to minimize DKL(p|f). Note if somehow we got f to match p, DKL would indeed go to zero. After shuffling the particles around – probably using Gradient Descent or something – we might come up with something like this,

A grid with green circles and black lines

Description automatically generated

**Example**

Let X follow an Exponential distribution with rate parameter λ = 1. Compute the Kullback-Leibler (KL) divergence between the distribution of X and a Normal distribution that has the same mean and variance as X.

Our exponential distribution is:



The mean and variance of this distribution is <x> = λ, <x>2 = λ2. The normal distribution with the same mean and variance is:



Filling these into our KL formula:



