Welcome to our notebook here

on dimensionality reduction. In this notebook, we're going to be using the Portuguese

wholesale distributor data set. That data sets going to

contain the annual spending on fresh products on milk products,

grocery products, and so on. And then the last two which we're actually

going to end up dropping are going to be channel and reach. And the reason we dropped those is

because we want to focus on the numeric values here. And these are technically going

to both be categorical values. And it's just as easy if we

wanted two to 100 code them. But for this, we're just going

to drop those two columns. We're then going to import our necessary

libraries as we do at the start of each one of our notebooks. And then here for part one,

we're going to want to import our data and check each of the data types. Or then as mentioned going to drop

the channel and the region columns. As we won't be focusing on these

throughout our examples here using PCA. We're then going to convert the remaining

columns, two floats, if that's necessary. And then we're going to copy a version

of the data that we just created using the dot copy method

to preserve it and we'll be using that later on and

we'll see how in a bit. So, first things first,

we import our data using pandas.readcsv. We look at the shape and we see that

we have 440 rows, and 8 columns. And recall the number of columns that

are going to be important as our goal here with PCA is to reduce that number

of columns that were working with when we create our models, or whatever

it is we want to do with our data. Maybe want to visualize, and

we want to reduce the two columns. So we see our first five rows. And we see here that we

still had that channel and region which we said we

don't want to include. So we're just going to call data drop,

and we drop the channel and region from access equals one. And we look at the data types,

we see that they're each integers. And we're just going to convert

those each to float calling dot s. Dot as type float for

each one of the different columns. Now we have them all as floats. And then as mentioned, we're going to want

to save this original data for later. So recall here we have our data, which is

our data frame that we've just created. And then data are which is

going to be a copy of that, which we're not going to touch for a bit. Here in part two, we need to again ensure

that our data is scaled and relatively normally distributed, it will be easier

to work with normally distributed data. And then as mentioned in the lecture, we saw how important it is to scale

our data to ensure that no feature has extra weight when trying to come up

with the different principal components. So we're going to examine

the correlation between each one of our different features. And recall this will be important

as when we are doing PCA. What we will be looking for is if two

features are very highly correlated, they're not adding any extra information

and we want to remove or reduce those or combine a few to end up

with less features overall. So if they're highly correlated, we can probably remove some without losing

much variance from the overall data set. We're then going to perform

any transformations and scale our data using whatever

scaling method you prefer. Whether it's min max scaler,

or the standard scaler. We're then going to view the pairwise

correlation plots, using our pair plot just to visualize all the relationships,

as well as now seeing if we have normally distributed data looking

across that diagonal of a pair plot. So the first thing that we

want to do is called data.core so we can see the correlation between

each one of the different features. So, this will give us for

each feature the correlation with all the other features in a square

matrix in a square data frame. And just to ensure that

we can get the highest correlation which feature is

the highest correlation and because of one feature with itself

will always have a correlation of one. We're going to replace that diagonal value

which are going to start off as all ones with all zeros. So we're saying for

x in the range of format.shape zero, it's a square matrix we could have

called shape zero or shape one. So that's going to be for every single

value in our matrix, for every single numeric value for the range of our matrix,

we're going to take the diagonal value. So zero, zero, one, one, two, two and

replace that one with one with a zero. And we can see now our correlation matrix

has the correlation between fresh and milk and grocery. And then for fresh and fresh, it's just a zero across each

one of the different diagonals. Now we're going to call the absolute

value on that full correlation. As we don't care if it's a positive or

negative, just the strength of that correlation, and

we're going to call it x max to see which feature is most highly correlated with G,

which of the other features? So we're saying what's

the max index value? So for fresh, it's frozen for milk,

it's grocery, so on and so forth. We're then going to examine the skew for

each one of our ten values. And then take the long transformation,

if necessary, for those that have higher skew. Recall that the skew's going to be

a value with zero being no skew, positive value being a right skew, and

a negative value being a left skew. The higher that value is,

the stronger the skew. So we call data.skew to see the skew

of each one of our different columns. We sort them from largest to smallest. And those are going to be our log columns. And that will now be a panda's series. And then we're just going to take those

log columns that are greater than 0.75. Those that have a higher skew and we see here we have these values that

have tend to have a higher skew. And for those, we're going to to take

the log transformation with each, hopefully creating more

normally distributed data. So for calling each one of

these log columns.index, so these are, this is our log columns that

we just defined is that pandas series for called the index we get each one of

these Delicatessen frozen milk and so on, which is going to also match up with

each one of our different data columns. So we're going to place those columns in

place with the log transformation of those columns. We can then also call the min, max scaler. So we import from scaler.pre processing,

the min max scaler. We want to ensure that all our

values are on the same scale. We call min max scaler, we initiate

the object, and then we say for column and each one of our columns are going to

fit and transform on that column. So we're going to replace it again in

place, to standardize that data, so all values are between zero and one by using

the midnight scalar, which were calls just subtracting the minimum value and

then dividing by the max minus the min. So that'll ensure all our values

are between zero and one. The next thing that we want to do is we're

going to visualize everything that we've just done. So we're going to see each of the

relationships and hopefully see those high correlations with each one of

the different scatter plots that we'll see with the pear plot, as well as saying,

hopefully, more normally distributed data, which we see for the most part throughout

each one of our different columns. And we see for example, milk, wood and

grocery have a pretty high correlation. If you look just three columns in and

two columns down, you see that high correlation. Now in part three, we want to introduce

how we can do this all in one step. NSV is especially useful if we want to

incorporate this into some supervised learning model later on, and be able to

pass in different parameters through out. So we're going to pass

enough pipeline function and we saw that during our course

on supervised learning. But what's important when

using the pipeline function. Is that, each one of the functions that

are passed in, each one of the different pieces of that pipeline, have to have

a fit and transform method to them. So, we want to take the log, and

then take the Min Max Scalar. But the log doesn't have that

fit-transform that's build in with each one of our different sklearn

objects that we've been working with. So min max scalar has a fit transform but

log transformer does not. So in order to ensure that we have a

version of taking that log transformation that has the fit and transform methods

that we can pass into our pipeline. We're going to call this

function transformer and this function will take whatever

function is that you want to pass in and convert it so that it has a fit and

transform method available to it. So now we have a log transformer object, which is going to be a log transformer

with a fit and transform method and once we do that we can

pass it into our pipeline. So first within our pipeline,

we need to pass in that list of tuples, where the first value of that tuple

is going to just be that name. If we want to pull it out later. And then the next value is going to be

the actual function that we want to call. So here we call that log transformer that

we just created, and then min max scaler. We pass in this list of

tuples into our pipeline. And then we can just call pipeline.fit

transform on our original data. If you recall, we made a copy, and

we didn't change our data at all for that copy of the data. And we can call fit, transform and

get the output down the line of both taken that log transformation and

that min max scaler. And we run this and then that data pipe should equal

that data that we just transformed. So we're going to check that using

NumPy.allclose, which is just going to check that each value within each

of our arrays are exactly the same, with a bit of possible rounding error,

many decimal points down the line. So we run this and we see that it's true

that all of our values are the same. And we see that our pipeline work just as

well as taking each one of these different steps separately. Now that closes out part 3. In part four, we're going to start working

with PCA on this transform data that we've been working with and

see how much of the variance can we explain with different numbers

of these principal components? All right, I'll see you there.

LAB

Welcome to our notebook here

on dimensionality reduction. In this notebook, we're going to be using the Portuguese

wholesale distributor data set. That data sets going to

contain the annual spending on fresh products on milk products,

grocery products, and so on. And then the last two which we're actually

going to end up dropping are going to be channel and reach. And the reason we dropped those is

because we want to focus on the numeric values here. And these are technically going

to both be categorical values. And it's just as easy if we

wanted two to 100 code them. But for this, we're just going

to drop those two columns. We're then going to import our necessary

libraries as we do at the start of each one of our notebooks. And then here for part one,

we're going to want to import our data and check each of the data types. Or then as mentioned going to drop

the channel and the region columns. As we won't be focusing on these

throughout our examples here using PCA. We're then going to convert the remaining

columns, two floats, if that's necessary. And then we're going to copy a version

of the data that we just created using the dot copy method

to preserve it and we'll be using that later on and

we'll see how in a bit. So, first things first,

we import our data using pandas.readcsv. We look at the shape and we see that

we have 440 rows, and 8 columns. And recall the number of columns that

are going to be important as our goal here with PCA is to reduce that number

of columns that were working with when we create our models, or whatever

it is we want to do with our data. Maybe want to visualize, and

we want to reduce the two columns. So we see our first five rows. And we see here that we

still had that channel and region which we said we

don't want to include. So we're just going to call data drop,

and we drop the channel and region from access equals one. And we look at the data types,

we see that they're each integers. And we're just going to convert

those each to float calling dot s. Dot as type float for

each one of the different columns. Now we have them all as floats. And then as mentioned, we're going to want

to save this original data for later. So recall here we have our data, which is

our data frame that we've just created. And then data are which is

going to be a copy of that, which we're not going to touch for a bit. Here in part two, we need to again ensure

that our data is scaled and relatively normally distributed, it will be easier

to work with normally distributed data. And then as mentioned in the lecture, we saw how important it is to scale

our data to ensure that no feature has extra weight when trying to come up

with the different principal components. So we're going to examine

the correlation between each one of our different features. And recall this will be important

as when we are doing PCA. What we will be looking for is if two

features are very highly correlated, they're not adding any extra information

and we want to remove or reduce those or combine a few to end up

with less features overall. So if they're highly correlated, we can probably remove some without losing

much variance from the overall data set. We're then going to perform

any transformations and scale our data using whatever

scaling method you prefer. Whether it's min max scaler,

or the standard scaler. We're then going to view the pairwise

correlation plots, using our pair plot just to visualize all the relationships,

as well as now seeing if we have normally distributed data looking

across that diagonal of a pair plot. So the first thing that we

want to do is called data.core so we can see the correlation between

each one of the different features. So, this will give us for

each feature the correlation with all the other features in a square

matrix in a square data frame. And just to ensure that

we can get the highest correlation which feature is

the highest correlation and because of one feature with itself

will always have a correlation of one. We're going to replace that diagonal value

which are going to start off as all ones with all zeros. So we're saying for

x in the range of format.shape zero, it's a square matrix we could have

called shape zero or shape one. So that's going to be for every single

value in our matrix, for every single numeric value for the range of our matrix,

we're going to take the diagonal value. So zero, zero, one, one, two, two and

replace that one with one with a zero. And we can see now our correlation matrix

has the correlation between fresh and milk and grocery. And then for fresh and fresh, it's just a zero across each

one of the different diagonals. Now we're going to call the absolute

value on that full correlation. As we don't care if it's a positive or

negative, just the strength of that correlation, and

we're going to call it x max to see which feature is most highly correlated with G,

which of the other features? So we're saying what's

the max index value? So for fresh, it's frozen for milk,

it's grocery, so on and so forth. We're then going to examine the skew for

each one of our ten values. And then take the long transformation,

if necessary, for those that have higher skew. Recall that the skew's going to be

a value with zero being no skew, positive value being a right skew, and

a negative value being a left skew. The higher that value is,

the stronger the skew. So we call data.skew to see the skew

of each one of our different columns. We sort them from largest to smallest. And those are going to be our log columns. And that will now be a panda's series. And then we're just going to take those

log columns that are greater than 0.75. Those that have a higher skew and we see here we have these values that

have tend to have a higher skew. And for those, we're going to to take

the log transformation with each, hopefully creating more

normally distributed data. So for calling each one of

these log columns.index, so these are, this is our log columns that

we just defined is that pandas series for called the index we get each one of

these Delicatessen frozen milk and so on, which is going to also match up with

each one of our different data columns. So we're going to place those columns in

place with the log transformation of those columns. We can then also call the min, max scaler. So we import from scaler.pre processing,

the min max scaler. We want to ensure that all our

values are on the same scale. We call min max scaler, we initiate

the object, and then we say for column and each one of our columns are going to

fit and transform on that column. So we're going to replace it again in

place, to standardize that data, so all values are between zero and one by using

the midnight scalar, which were calls just subtracting the minimum value and

then dividing by the max minus the min. So that'll ensure all our values

are between zero and one. The next thing that we want to do is we're

going to visualize everything that we've just done. So we're going to see each of the

relationships and hopefully see those high correlations with each one of

the different scatter plots that we'll see with the pear plot, as well as saying,

hopefully, more normally distributed data, which we see for the most part throughout

each one of our different columns. And we see for example, milk, wood and

grocery have a pretty high correlation. If you look just three columns in and

two columns down, you see that high correlation. Now in part three, we want to introduce

how we can do this all in one step. NSV is especially useful if we want to

incorporate this into some supervised learning model later on, and be able to

pass in different parameters through out. So we're going to pass

enough pipeline function and we saw that during our course

on supervised learning. But what's important when

using the pipeline function. Is that, each one of the functions that

are passed in, each one of the different pieces of that pipeline, have to have

a fit and transform method to them. So, we want to take the log, and

then take the Min Max Scalar. But the log doesn't have that

fit-transform that's build in with each one of our different sklearn

objects that we've been working with. So min max scalar has a fit transform but

log transformer does not. So in order to ensure that we have a

version of taking that log transformation that has the fit and transform methods

that we can pass into our pipeline. We're going to call this

function transformer and this function will take whatever

function is that you want to pass in and convert it so that it has a fit and

transform method available to it. So now we have a log transformer object, which is going to be a log transformer

with a fit and transform method and once we do that we can

pass it into our pipeline. So first within our pipeline,

we need to pass in that list of tuples, where the first value of that tuple

is going to just be that name. If we want to pull it out later. And then the next value is going to be

the actual function that we want to call. So here we call that log transformer that

we just created, and then min max scaler. We pass in this list of

tuples into our pipeline. And then we can just call pipeline.fit

transform on our original data. If you recall, we made a copy, and

we didn't change our data at all for that copy of the data. And we can call fit, transform and

get the output down the line of both taken that log transformation and

that min max scaler. And we run this and then that data pipe should equal

that data that we just transformed. So we're going to check that using

NumPy.allclose, which is just going to check that each value within each

of our arrays are exactly the same, with a bit of possible rounding error,

many decimal points down the line. So we run this and we see that it's true

that all of our values are the same. And we see that our pipeline work just as

well as taking each one of these different steps separately. Now that closes out part 3. In part four, we're going to start working

with PCA on this transform data that we've been working with and

see how much of the variance can we explain with different numbers

of these principal components? All right, I'll see you there.

Now for Part 4 as we will be working

through here in this video, we're going to perform PCA on that data that we worked

through in the last video, and we're going to perform PCA for the number of components

ranging from 1-5. We start off with six columns and no matter what

we're going to try and reduce the number of columns that we'll ultimately

be working with. We're then going to

store the amount of explained variance for each one on the different

numbers of dimensions. For one dimension, how much

variance was explained. For two, so on and so forth. If we were to do number of

components equal to six, then we would have explained

100 percent of the variance. We're saying how much of

the variance going to explain each one of

the different steps. We're also going to store the feature importances for each one of the number of dimensions. Something to know is that

PCA won't explicitly provide this feature importance, but the components properties which we'll show you how

to use in just a bit, will show you how each one of those principal components was composed as a combination of each one of the

original features, and the larger those values are given that we've

standardized our data, the more impact each

one of those features has had on that

principal component, and therefore we can assume that that is a more

important feature. Then we're going

to plot both that explained variance as well as

these feature importances. Now, I'm going to break

this down step by step. I'm going to actually

create a cell above, but before I do that just to show you where

we're starting off. We're going to import from

sklearn.decomposition. We're going to import PCA. We're going to initiate

an empty list of the PCA lists and the feature

weight list which we're going to use to store our explained variance and

the feature importances. Then for n in range

one through six, so one through five

if including five, so that's what we want

to range through. We're going to initiate a model, a PCA model with the number of components equal to wherever

we are within that range, and then we're going to

fit it to our data that we have now done

the transformations to ensure that it is on the same scale and

mostly normal data. We're then going to take

the explained variance of each and append that to PCA list, and then after a few steps which I'll walk you

through in just a bit, we're going to take each one of the feature importances and append it to the

feature weight list. Then after we do this for

the n range one through six, we have this for each one of our different numbers of

principal components. Let's start off by looking

at just this step here. We're going to create

a Pandas series. We actually are also

going to need of course to initiate our model. What I'm going to do since

I'm pulling this out here, is going to set n equal to two as we discuss all the steps, and you can imagine that

this is going to do it for n equals of course

one through five. We set n equals two, and then let's see

what this series is that we're going

to be outputting. It should be n which is the number of components

which we set to two, the actual model, as well as the explained variance

up to that point. For using two components, how much variance was explained

by using two components? I'll run this, and

we can see that it explained 72 percent of

the overall variance. Now, just to see how the explained variance

ratio actually looks, let's pull this out and

we can see that it says, if you set n equal to two, it shows you how much of the explained variance

ratio was covered with the first principle

component which was about 45 percent, and how much was done by the second component which

was about 27 percent, and the first one should

always have more than the second which always

had more than a third. Our first principal

component should be the component that

explains the most variance. We will have there for each

of our number components, the amount of variance

explained. That is covered. Our next step is going to be to find the feature importances. The first thing that

we're going to do here is we're going to, and let's add this on over here. Set some weights, and the

idea of the weights is that we have the breakdown of each of our principal components, but we want to add more weight to the more important

principle components. The first one should be more important than the second one, and so on and so forth. What I'm doing here is I'm taking this explained variance

ratio that we output here, and then we're just setting

it if we're working with two components or setting

it as a proportion of one, so we're saying 44 percent and 28 percent, so

we're adding those up, and we're saying, out of one, what proportion is 44 and

what proportion is 28? Just to look at what that means, you see we take that original

amounts with 45 and 27, and we just divide it by

the total of 45 plus 27. That we see that the

weights are 62 for that first component and 38

for that second component. We're going to weigh our

components according to how important these different

principal components are. This will become clear

in just a second. The next thing that

we're going to see is this pca.components. What was important here for the PCA components is

this is going to be the breakdown of how each one of the components

is actually comprised. Let's first strip away everything

besides pca.component. We can see here that we have

for the first components, how each one of the different

features that we had. We have six different features, how they each created a linear combination to come up with our first components, and t hen the linear combination that came up with our

second component. Again, the idea is the larger

these absolute values are, the more they contributed to each component and the more

important that feature is. What we had here before is

we took the absolute value, because we don't care about whether it's positive

or negative. We just care about how much it affected that

principal component. Then we're weighting it

according to these weights. If you recall, the

weights are going to be how important each one the

principal components are. This first one is going

to be multiplied by 0.62, and this second one is going

to be multiplied by 0.38, so that we don't put

on too much weight. We see here that we use 70 percent of whatever

feature this is; this is the fifth feature. Then we use 70 percent here

in the second feature. The second PCA for a

different feature, we want to ensure that these

do not get equal weights. This should get a higher

weight than this one since this is part of the

first principal component. That's why we multiply

it by the weights, and then we can see what the

overall contribution is. Let's just copy and paste that. We can see the

overall contribution for each one of the

different components, and then we're going to

take the sum axis equals 0. So that we can see now that we've weighted each one of them, how much each one of these

different features with their weightings were able to comprise these principal

components that we have. We see here that's

whatever feature it is. The fifth feature was

the most important in the first two components if you add up the weights of the

first two components. We're then going to divide

that value down here. We have the absolute

features values. We're going to divide that by the total sum of these values, to ensure that each one of these values as a

proportion up to one. So that we can see, again, these each represent how

much weight each one of our original features played in coming up with our two

principal components. We're going to

normalize that over one to see the proportion of one, of each one of these features; how much they comprise, how much did they contribute to coming up with these

principal components. That's going to be the

values that we have here. Then we are going to

have a data frame that has the number of components, and then it's going

to have each one of the different columns

so that we can line that up with each

one of these values. Then we're going to

have for each one of those different values, what is the aligned

column that I went with, and that's going to

be our values here. I'm going to run this. The first thing that

outputs is the number of explained variance for each one of our different

principal components. You see the first one

covered 45 percent, the first two covered 72, then 83, 92, and 98. We see once we get to five, we've covered 98 percent

of our overall variance. We're then going to

concatenate fewer call. Let's look actually at this feature waitlist

that we created. This is going to be a

bunch of data frames. Let's just look at the first one. We see this is going to be for a number of components

equals to 1. How much each one of these different

features contributed to that principle components? We set this equal to one. We can see for the

first two how much it contributed to each one of the difference

principle components. We're going to concatenate all these different data frames together so that we have

one long data frame. Then we're going to

pivot that and set the index equal to this n so that we don't

have multiple ones, twos, but we'll sum

up all of the ns. We are also going

to set our columns equal to the different features. Then we can just have our

values as the values. Now we have this data

frame that we have here, where we see when the number

of features is equal to 1, the contribution of each one

of these different features; one the number of features, one the number of

components is equal to 2, the contribution of

each of the features, and so on and so forth. Now we're going to plot the overall variance

just using a bar plot. This is plotting

what we had up here, that pca\_df, which is just

that overall variance. We just set our x-label, our y-label, and our title. We see that how much of

the overall variance was explained once we add on each one of these different

principal components. Then finally, we have

plotting the features\_df, and we're going to see

as we have each one of the different number of dimensions that

we're working with. How much does each one of

the different features contribute to all of our

principle components. We see here for detergents

paper, at first, it explained most of the variance that was the

most important feature and tends to balance out as we add on that number of components. Now that closes out our

section here on question four, showing you how to see use PCA, see the explained

overall variance, as well as getting a hint at the actual feature importances as we create each one of our different

principal components. In the next section, we will discuss how

we can actually use grid search to fine

tune our PCA model, especially when working with

kernels. I'll see you there.

Welcome back for

part five of our notebook here. Here we're going to introduce kernel PCA,

or PCA working with a kernel where we're going to use what we

discussed in the lecture. And that we can come up with a non linear

combination rather than the linear PCA to come up with a way to say where

the highest variance is by mapping up to higher dimensions to get that

curvature in those lower dimensions. Now, we want to know, choosing here

that our kernel is equal to RBF. We can also search through

different kernels and I suggest you looking at

the documentation as well. But we can also search through when we're

working with RBF using different gammas and that'll tell you essentially how

complex that boundary is going to be or how curvy your line that you can

project on to will actually be. So we're going to search

through different gammas. And we're going to use grid search. And when we use grid search, what we're

trying to do is find the best model and when we do this with supervised learning,

this is clear we can do this with using a scoring methods such as mean squared

error, or working with the accuracy or whatever other classification score you

want to use and optimize on that score. Now when we're using unsupervised

learning, it's not quite as clear how we can end up scoring which one of these

different models performs better. But we do need to come up with some type

of scoring option in order to decide which gamma or if we want to

search through different kernels, which kernel work the best. So what we're going to do here

is we're going to introduce a custom scoring method. So you'll see here that

we defined a score. And we'll walk through what that score is. But essentially what we're going

to do is take a model, fit a PCA, fit that PCA model to our data, and

then take the inverse of that, and then see how far away the inverse of that

PCA model is from our original data. And the lower that value is,

the better we did. So let's walk through that here. So first we're going to import

the kernelPCA rather than just PCA. We're going to import GridSearchCV

as we'll be using that in order to find the optimal hyper

parameters for our kernel PCA. And then you'll see in just a second how

we're going to incorporate mean squared error in regards to coming up with

the best version of our kernel PCA. So first thing that we're

going to do is define a score. So we're going to pass into

that score the PCA model, as well as our x and there's going to

be no y here just going to be that x, right we're using unsupervised data. There's no label that

we're attributing to this. All we're doing here with this try and

accept is just we want to ensure that we are working with a NumPy array rather

than working with a Pandas data frame. So if this x is equal

to a pandas data frame, we call that values and

we're working with the array. If it's already an array,

then they'll just say x val to that array. We're then going to call our PCA

model that we pass into the score. And we're going to call it on that X val. And we fit transform our data

to get our new version with however many components

we're passing through. One component, two components, so on,

as well as whatever kernel we're using and whatever gamma we're using. Specific to what this PCA model is. We're then going to take the output of

that and pass it into this PCA.inverse transform function to get the inverse,

which should undo what we did, but it can't perfectly undo because we

lost some information as we did that original transformation, as we did that

original dimensionality reduction. So we'll take the inverse, and

that will be our new data end. And then, what we're going to do is

take the original data that we had, and see how far off that is from our

inverse transform that we just did. And in order to do that,

we'll just take the mean squared error. Now when we do a square,

we want to get the highest value possible. When we do mean squared error, obviously we want to minimize

our mean squared error. So we're just going to

multiply it by -1 so that we can optimize by

getting the highest value. And that's going to be

our scoring function. From there it should be as simple as

any other grid search that we've worked with in the past. You're going to set your parameter grid,

which is going to be gamma and we'll loop through different gamma values. It's going to be this dictionary and

the number of components and it will loop through different

numbers of components. Now I'll let you know generally speaking,

the higher the number of components, the better this transform,

inverse transform will work, but this will allow us to hone in

on the right level of gamma. We're then going to do grid search CV. We're going to say that we want

to pass in the kernel PCA. And the things that we don't want to

search over but want to keep the same through every single loop is going to

be that the kernel is equal to RBF. And we want it to fit

the inverse transform. If we don't call this, when we call the

PCA, then we won't have the option to call this inverse transform that we have called

up here during our scoring function. So we say Fit\_inverse\_transform=True. We can then pass in our param

grid that we defined up here. And then we can pass in

the score that we just created. We say n\_jobs=-1 just to say we want

to paralyze as much as possible. And then using this kernelPCA

that we're defining here, we can call kernelPCA.fit on the data and get our best estimator to see which one

of these gammas performed the best. So run that, and

that will take just a second to run. So I'm going to pause the video,

there it is, never mind. And we see here that we have for

our gamma value 0.5 was the best option in regards to that

transform to inverse transform and we see that the number of

components is equal to 4, which is the max value,

which is what I said. Usually when you're working with looping

through the number of components, the max value will be the one chosen. But now we can see that we should

probably use that gamma equals 0.5 when choosing our gamma for our criminal PCA. Now for Part Six, we're going to

show you how you can use PCA built into your modeling pipeline

in order to perhaps use it to make your logistic regression work

better on the data that you have. So we're going to be loading

in this very large data set, which is the human activity

recognition using smartphones. We've seen this before,

it has tons of different columns. We can look at the shape here and

see that it is 10,299 rows and 562 different columns. So we're going to try and

reduce that number of columns. So what we're going to do is

we're going to first import the different libraries needed,

our pipeline, standard scaler, stratified shuffle split to keep that same ratio of

each one of our different outcome values. We're now using logistic regression and we can pull in our accuracy score since

we're doing a classification problem here. x is going to be all values except for

activity. y is going to be the activity. And then we're going to initiate

our stratified shuffle split and we'll call this in just a bit when

we want to get our average score. Now this get average score is going to

just be a function that does all the steps in the pipeline to standard scaling,

PCA and then logistic regression. And all we're going to change at each one

of the steps is the number of components. So we set this pipe equal to this list and we pass it pass it into our

pipeline as we've done before. We have our scores which are just blank. So we initiated our pipeline but

haven't fit anything yet. We have our scores equal to that blank. We're then using that SSS

that we initiated here, that stratified shuffle split, and

we're going to get five different splits, since we set the number

of splits equal to five. And for each of those,

we'll get a new x train and a new x test as well as the new

y train and the new y test. And we can call pipe,

that being the pipeline we created here, .fit on our x train and y train. And then once we do that five

different times throughout each time, we're also going to get

the accuracy score on the test set. So once it's fit on the training set, we

can see the actual score on the test set. We'll have five different scores, and then we'll output the average

of those five different scores. We're going to set the number

of ns from 10 up till 500. So we see our original data set was 562. We're going to see if we reduce

the number of dimensions, is there a point where perhaps we

don't need all of the data set or even perhaps some improvement

with lower dimensions. So we're going to get our score list

by running this get average score that we defined up here on each n in

this option of ns that we have here. So I'll run this, and

this one will actually take some time, so I'm going to pause the video here. And I'll see you in just a bit as we touch

on the results from running this function. All right, I'll see you there. All right, now that has finished running

and it may take a couple minutes. Let's see what the score list came out as. We run this and this should be in the same

order as our ns that we have here. And we see that after a certain point

once we get to the 450, 500 range, there doesn't seem to be any more

improvement in adding more variables, in adding more features. And we can see this with the plot as well, just plotting out ns versus

our different score lists. And we can see that it really plateaus and it's not even starting at zero here

on the y axis, it's starting at 0.84. So we see that adding on all these extra

dimensions doesn't really add that much extra value in regards to

the logistic regression. So you could probably shrink this down to

even 100 features here, or 200 features and still have a pretty high accuracy

depending on what you're trying to get at. And be able to speed up the process of how

long it will take to learn this model. That closes out our demo here

on dimensionality reduction, and I'll see you back at lecture. Thank you.

Now we introduce another way of

reducing the number of mentions mainly non negative matrix factorization. Now with non negative

matrix factorization, we're still going to be

decomposing our original matrix, but this time we're starting with

as input only positive value. So you can think word counts, or pixels image as examples of

matrices with only positive values. And then we decompose that original

matrix of positive values into two matrices W and H, with both

also having only positive values. So that's that non negative

matrix factorization. Now we can think of taking a term and

a document matrix. So to create a matrix of

the sort out of many documents, you can think of each one of your

different observations or each one of your different rows as being a specific

document and each column being a word. And the values for that documents for

rows and wrote words for columns. Each one the values will be the word count

or some other measure of the word for that document depending on how

you pre process your text data. We can then decompose this into how

the terms each makeups certain topics, and that's your W here. And that number of topics

will be of your choosing, similar to the choosing of

components when we're doing PCA. And then the H will be how

to combine these new topics together to recreate

our original documents. Now thinking of images if we think back

to PCA, PCA is highly recommended when you have to transform higher dimensions

into lower dimensions and you are okay to lose the original features in the

process as new ones are being introduced. So when we look at the breakdown

of the components, it's going to be difficult to gain

any insight into how they all combine to recreate that original image,

as each one of these new components are composed of a weird combination

of those original features. Now with non negative matrix factorization

since we're only working with positive values, and

we can only add those values together, we can't subtract since everything's

positive and both our W and H matrices. The different components tend to have

more of an intuitive feel as we'll be adding together, the shading of the eyes,

the eyebrows, the nose, etc, all together to recreate an image

of our face as we see here. Now non negative matrix factorization

has proven to be powerful for word and vocabulary recognition,

image processing problems, text mining, transcriptions processes, cryptic encoding

and decoding and it can also handle decomposition of non interpretable data

objects such as video, music or images. So why focus on a decomposition

of only positive values? For one, since nonnegative

matrix factorization only works with positive values, it can never undo

the application of a latent feature. There's no canceling out with negative

values, it's only going to be additive, and thus each included feature

must be important as again, we can't cancel it out down the line. Also, since it's only positive values,

this leads to features that may be interpretable as they must all add

together to recreate our original data. So as mentioned for something like

a dataset of different faces, you may have the nose, the ears, etc, and those

will add together to recreate the face. Something to note,

is that because non negative matrix factorization has the extra

constraint of positive values only. If we ended up in that original

decomposition with some negative values, the algorithm will automatically

truncate those to zero, and thus may not be able to maintain as

much of our original information. Something else know, is that unlike PCA

there's going to be no constrains of only orthogonal vectors,

one we'll looking only positive values. So the decomposition can thus

have portions pointing in similar directions in n dimensional space. So now let's briefly touch on how non

negative matrix factorization will work with something like natural

language processing. So as input to our non negative

matrix factorization for documents, you'd pass in some type of pre-process

version of each of your documents. Turning words into numeric values can

either use a count vectorizer for the count of words, or the TF-IDF,

which is term frequency inverse document frequency, which will give you a value

that gives less weight to more common words such as a or de or is within

the entire range of all of your documents. We can then have the possibilities of

tuning the number of topics that we ultimately want as well as the means

of pre processing our text. May want to remove certain stop words or

frequent terms altogether. And then our output will be how

the different terms relate to the different topics. And then another matrix telling us how

to use those topics to reconstruct our original document. Now in order to actually use NMF within

Python, this syntax will be very similar to what we've seen so far with

the different decomposition methods. So from sklearn decomposition,

we import NMF. We then create an instance of our class

passing in the appropriate arguments. So we say how many topics? How many different components

do we actually want? And then we say how do

we want to initialize. Most of you will initialize random but

what is important to note is that the method can be sensitive to

the type of initialization as we've seen with other models, and the results

will not necessarily be unique. So we initiate our class nmF

with a number of components. And then we can fit the instance and create a transformed version of

the data by calling NMF.fit as well as NMF.transform in order to

come up with our new data set. Now just to recap the different

approaches that we went through, dimensionality reduction is going to be

common across a wide range of application. And we have here some rules of thumb for

selecting what approach you'd like to use. For principal component analysis, this will be great if you have

a linear combination of features. You believe that you can create or maintain the amount of original variants

and that's your goal is to preserve variance by creating a linear

combination of those original features. Kernel PCA will be similar except for assuming there's more of

a nonlinear relationship. We still want to preserve the overall

variance within each one of our features. Multidimensional scaling, like PCA with

new transformed features are determined based on preserving distance rather than

maintaining variants as we did a PCA. So if maintaining the amount

of distance is more important, which may be something useful if you

want to visualize different clusters, this may be a better approach,

then you'd want to use MDS. And then finally as we just discussed,

is non negative matrix factorization, which is useful when you're working with

only positive values such as working with word matrices or working with images. Now let's recap what we

learned here in this section. In this section we discussed

dimensionality reduction, and how we can solve our problem of this

cursive of dimensionality by coming up with a lower dimensional representation

of our original data that maintains the majority of the information important

to us in that original data set. We then discussed principal

component analysis or PCA, and how we can use it to come up with new

features created as a linear combination of those original features. Or if we use kernel PCA, a nonlinear

combination of those original features to maintain as much of the variance

from that original data set as possible. And then finally, we discussed

non-negative matrix factorization and how working with only positive values can

lead to us being able to come up with more intuitive and powerful representations

of our original data in lower dimension. Now that close out our lecture

on dimensionality reduction, and from here we're going to move to

a demo of actually working with non-negative matrix

factorization using Python. All right, I'll see you there.

New LAB

Welcome to our notebook here are non

negative matrix factorization. In this notebook, we're going to be

covering the BBC data set on different articles across five different topics. The data has been pre processed so that we have a sparse matrix, we'll

see what that means in just a second. With that we have bbc.terms, which is just a list of the words

that are used, as well as bbc.docs, which is just going to be a list

of the articles listed by topic. So at a high level,

what we're going to do here is turn our bbc.matrix file

into an actual sparse matrix. So it's already in sparse matrix form,

as we'll see. But in general, working with a sparse

matrix just means rather than having a ton of zeros for many of your columns, we're

just going to have for each row or column we will specify whether or not there's

a value there and what that value is. Rather than when you have a larger not

sparse matrix with a lot of zeros, you can end up eating a lot of memory. We're then going to decompose that

sparse matrix using non-negative matrix factorization. And then use the resulting components of

that non-negative matrix factorizations to analyse the topics that

we end up coming up with. So the first thing that we

want to do is take that bbc.mtx, which is our sparse matrix, and

we're going to open that file. So now we have our flat

file available as f. And then once we have that f object,

we just call read lines, and the output of read lines

will be output into content. So now we have our contents, and

just to see what that looks like, I'm going to run this and

you see that it's going to be a list. And what we have beyond the first two values are just going to be

a sparse matrix representation. And we're going to go through in just

that part one below what each of these different lines mean. But first we want to remove each

of these first two values, so we're just going to call it content.pop. We're going to call 0 twice,

so we remove the 0 value and then we remove the 0 value again. So we see that the last one that I removed

was this value here from the list. And now we should only have from here and

below, if we call content. Now in part one, we're going to

turn this list, and currently, this is a list of strings

into a list of tuples. And that list of tuples will

represent a sparse matrix. So that sparse matrix is going to have

as that first column, the wordID. As the second column within that table, that second value,

we're going to have the articleID. And the third is going to be the number

of times that that particular word shows up in that particular article. So as an example here,

if word 1 appears in article 3 two times, then our element for a list,

that tuple will be word 1, article 3, showed up 2 times. Now in order to create this tuple, what we do is this somewhat complicated

looking list comprehension. I will break it down very quickly,

if we just do c for c and contents. Let's actually, that will just give

us the exact list that we saw before. So let's just first call that split. And we see that we've now split that

string that we originally had into the three separate values,

so we're on our way there. And then all we do from there

is map over a float, since it'll be difficult to get a integer just out

of that 1, but now we map over a float. And then we take the integer

of that float, so that we're only working with

integers each time mapping. So first we mapped to each one

the values in this tuple, the float, then we map over the integer, and

then we set that output as just a tuple. And if we look at the output for just those first eight values, we see

that we now have a list of tuples 1, 1, 1, 1, 7, 2, so on and so forth,

telling us the word, the article, and then the count of that

word within that article. Now we want to prepare the actual sparse

matrix that we're going to be parsing into our NMF, into our

non-negative matrix factorization. So we're going to import NumPy and

Pandas, and we're also going to import from

scipy.sparse the CO matrix. Which will give us a means of parsing

in the way we have our data currently constructed into a sparse matrix. So we're going to specify what

our rows are going to be. So since the start off if

you look back up here, it's going to actually start with word 1,

article 1. Just for

it to match up with Python syntax, We're going to make it word 0,

article 0, and so on and so forth. So we call that every single value x1,

these are going to be our rows. We want our rows to be each one

of our different documents. So we say [x[1]- 1, so it's going to

be whatever this ID is minus 1 for x within sparse matrix

that we have to find here. And then [x[0], recall that's the wordID

are going to subtract 1 from that 0 value, and that will be our different columns. So our different columns will be

each one of our different words. And then the actual values are going to be

the amount of times that that shows up. And when we call COO matrix,

we parse in the values. And then with that,

we have the related rows and columns for where those values should actually fall. So it'll plug that in,

if we have row 1, column 1, it'll plug in whatever that value is. So for this second 1, it'll say, row 1, or row 7, column 1, plug in the value 2. So we run this, and just to make this

perfectly clear, we're actually going to recreate from that sparse matrix and

the actual Panda's data frame. So we know what our actual matrix that

we're working with that we're doing non-negative matrix factorization

on actually is made up of. So we're going to pull in the actual

terms and these will relate, 0 will be the 0 term, first will be

the first term, so on and so forth. So we say from this flat file,

bbc.terms, we'll call f.readlines again. And then just to access that first value, which is going to be the actual word,

we call c.split on that string. That'll output strings as before,

and we only want the first value. And that will be our output for words, and I'll run this and

we see the different words that come out. And then we'll do the same thing for each

one of our document names, we can do that. All the codes the same except we're

working with a different flat file, and we can see all the different

document names. And then I'm going to take that COO,

which we initialize here, which is just going to be a sparse matrix. We're going to turn that into a NumPy

array, parse that into our data frame and we're going to set our column equal

to those words we pulled out, and our index equal to those columns. So this is going to be the actual original

data frame that we're working with. This is going to be article 1, business

001, and we see that the word ad showed up once, the word sale showed up five times,

profit 10 times, so on and so forth. And you see, the reason why we'd want a

sparse matrix is because we'd have all of these 0's for almost every single

one of these different articles. Because we need a separate column for every single word that showed up

in any single one of the articles. This is why we generally work with sparse

matrices when we're doing natural language processing. So now we have our data frame that

we want to work with, and the next step will be to decompose our matrix

using non-negative matrix factorization. And we'll save that for the next video and

I look forward to seeing you there.

Welcome back to our notebook. In this video, we're

going to actually conduct non-negative

matrix factorization. If you recall, just before we

created our DataFrame that had each one of our different

articles for each row, and then for each column we had each one of the

different words and the values were how

often those words showed up in each one of

the different articles. We're going to decompose

that into different topics, and we will end up

with two matrices. One will be each one of the words and how much they

relate to each topic, and then the other one will

be how to take those topics and recreate those

documents that we have. In order to do non-negative

matrix factorization, we're going to have to define how many components we want. We're going to set the number of components equal to five, which is the number of

topics that we actually had in the original documents, and this will allow us

later on compare to see how related the new topics are to the actual topics that we had within each one

of the different articles. We import NMF. We then call the NMF, we set the number of components. Our initialization is going to be just random with a

random set of 818. Recall that with non-negative

matrix factorization, you not guaranteed to get the same exact solution

every single time. We're going to pass

in our sparse matrix calling model that fit transform

on that sparse matrix, and that will output

this doc topic, which is just going to be a DataFrame that's going

to be of shape 2,225, which is going to be the number

of articles that we had, and then we have

reshaped that into just the five topics

that we now want. Rather than than having

2,025 by, if we recall, coup was about somewhere in the nine thousands in regards

to the number of words, we have reduced it

down to five topics. Now we want to look at the

components of this model, and when we look

at the components, all that is is going to be the different words and how they make up each one of the different topics

that we now have. We're going to create a

new DataFrame, which is, we're going to call

here topic word, and we're going to pass in that model.components that's going

to be this output here. Again, going to be

the waiting for each one of the words for

each particular topic. We want the index equal to just, we'll call it Topic 1-Topic 5, and then the columns

are going to be those words that we

pulled in earlier. When we look at this, we can see that we have, for each one of the

different topics, how much each one in

the different words contribute to that

particular topic. Now, just to make further

sense of how this relates to the topics and the words as well as the articles and the words, we recall that the original data had five topics, business, entertainment, politics,

sports and tech. Now I'm going to

do topics per doc, I will again pass in the actual values of that doc topic that

we pulled it earlier. We're then going to

set as our index. Rather than, if you recall what the docs

actually look like, this is going to be the

different articles. It's going to be each one

of these different values; business.001, business.002. That first word

before that dot is going to tell us which

topic we're working with. We're just going to call

i.split on that dot, and then we're just going

to take the first value. We'll have all business or

later on, all entertainment, so on and so forth, and then our columns will

be Topic 1, Topic 2, so on and so forth, and when we look at this, we can see that taking that each one of those original documents and saying which topic

they most relate to. You see business seems to

relate most to Topic 2 and we see that repeatedly for every one of the different

business topics, and then for tech, we see

that Topic 4, I believe. We'll see in just a second, what we're going to do here is, we're going to reset the index so that this index is

in it's own column. We're then going to

group by that index and get the average value for

each one of the topics, and when we do that, we can see that Topic 1, the max value is politics, Topic 2 was business, Topic 3 was sports, so on and so forth. Let's just quickly,

just to make clear, show you what that

matrix looks like. This is the matrix and

we're just seeing which one of these have

the highest values, and that's how we end up

with these different groups, and then to make this

perfectly clear, we see that Topic 1 should, for example, relate to politics. If we take our topic word

that we saw up here, we're going to transpose

that so that each one of the different topics

are going to be the columns, and then we will

sort by first Topic 1 with the highest values on top. Ascending equals false

and we'll see party, Labour government, elect, blair. These all tend to highly relate with politics,

which makes sense. If we did Topic 3,

which was sport, we can see that game, play, so on and so forth, and you can play

around with this with each one and the

different topics. We see that with this

unsupervised model, if perhaps you don't actually

had your topics available, you can come up

with this in a way, types of clusters, with the non-negative

matrix factorization. That closes out

our video here and our notebook on non-negative

matrix factorization, and I'll see you back

in lecture. Thank you.

In this video, we will use an example

to see how PCA can be used to reduce the feature space of

an actual image in practice. Now, the learning goals for

this section will be just to show how dimensionality reduction can be

used in a real world application. And with that, bring together an example

using dimensionality reduction to take one image and compress it down

to smaller amount of features and see what that compressed image would

actually comprise of when doing PCA. Now we're going to walk through how we can

use dimensionality reduction in real life practice. So frequently, we want to use

dimensionality reduction when we end up with a lot of different features,

when we had high dimensional data. And this can happen often with text data,

features are usually going to be the word existence flags or

the word counts per document. And as we saw with the non-negative matrix

factorization notebook we just went through, this can end up creating quite

a lot of features very, very fast, and thus a lot of dimensions. So you want to use it often

when we're working with NLP. Or, as we see here, if we're working with

images, especially if we're working, say, with colored images, the features can be

the brightness value for RG and B values. So the brightness of each one of

those different colors per pixel. So it means that we can end up

with quite a lot of features on the order of the number of pixels

that are present within our image. Here we're working with black and

white, so it'll just be the brightness per each one

of the pixels without the RGB values, but still can end up with

quite a lot of pixels. So in this example, we're going to

see how PCA is going to be used for image compression. We're going to reduce this

image's dimensions, but hopefully retain most of the image. So to see this image as a data set,

we put on a grid on top of this image, where each square is going to

be 12 by 12 pixel sections. So each one of these different squares

will have 144 pixels per square. And each one of those squares will

represent a single observation within the full data set of this image. Something to note is that this grid

is just for visual representation. But in our example, we would imagine that there are more

squares than what we see here. So each square, again,

is a single observation that is 12 by 12, so a total of 144 pixels. This is going to be a black and

white image, which means that every pixel contains only one numeric value

indicating the brightness of that pixel. And putting those 144 pixels side by size,

we can end up with just one row vector. So we see here we take that 12 by 12 and

we unravel it to have 144 different features for

each one of our different squares. And each row in our data set will be

each one of the individual squares in our original image. We can then perform PCA on

all of our data points. So we see here, again, that we have each one of our different

rows representing a single square. So we end up with a matrix that's

the size of the number of squares times 144, which is the number

of features we now have. We can apply PCA to this matrix to try

to reduce the current dimensionality so that we end up with a new matrix that

still has that same number of rows, which is going to match up

with the number of squares, times m where m is going to

be some value less than 144. And those new columns will be projections

of some special combination of those original features that will

create our principal components, that will describe the most

amount of variance. So to see this in action, we see here

reducing from 144 down to 60 dimensions. So each square rather than being

represented by those 144 different values are now represented by

these 60 different values. We can still see quite a clear

picture of our original image. Reduce down to 16 dimensions, and

we still don't lose much from the original image in regards to visually

looking at one next to the other. So after PCA,

you will get these top 16 components, and these will be the 16 most

important principal components. And every original 12 by 12 grid

in this image before is now some linear combination of these

16 components that we have here, once we reduce down to 16 in regards

to our dimensions using PCA. We can reduce this further,

down to just four dimensions. So here we're reducing the dimensionality

severely, but since we're keeping the four most important principal components,

the image is still somewhat recognizable. And here we have the L2 error

between that original image and the compressed image with various

levels of dimensionality. Where we're just seeing the distance

of what that original image looked like compared to the values that we're working

with now with the compressed version. And we can see that for quite some time,

we don't have that high of a relative error as we continue to reduce

that number of dimensions. Now we see here just the top

four principal components. And again, we are going to be able to,

from that original 144, create some combination of those two, come

up with these four principal components. And something to know, as we recall

when we were working with PCA in the PCA notebook, it's going to be the top

four of our original toxic scene or even of our top original 144 components. So reducing to 16 and then selecting from the top is the same

as just reducing down to our top four. So no matter what, we always have the

first most important principal component first, the second one second,

so on and so forth. And then here we can see what

that image actually looks like reduced down to one dimension. So you see that now we're only

working with one dimension, and each one of our different squares is

just going to be a different weight for each one of those different original

squares that we were working with. And we can still see somewhat

of a fuzzy image here. So we can see here how PCA is actually

compressing our original image and the amount of data that we have to

store In order to represent that image. Now, just to quickly recap, in this

section, we discussed the applications of dimensionality reduction

in the real world. Using the example of working

with that butterfly image, using PCA to reduce the number of

dimensions and show how we didn't lose much from that original image when

we reduced the number of features. And that closes out our section

here on unsupervised learning. And it was a pleasure teaching you. Thank you.

# Summary

## **Non Negative Matrix Decomposition**

Non Negative Matrix Decomposition is another way of reducing the number of dimensions. Similar to PCA, it is also a matrix decomposition method in the form V=WxH.

The main difference is that it can only be applied to matrices that have positive values as inputs, for example:

* pixels in a matrix
* positive attributes that can be zero or higher

In the case of word and vocabulary recognition, each row in the matrix can be considered a document, while each column can be considered a topic.

NMF has proven to be powerful for:

* word and vocabulary recognition
* image processing,
* text mining
* transcribing
* encoding and decoding
* decomposition of video, music, or images

There are advantages and disadvantages of only dealing with non negative values.

An advantage, is that NMF leads to features that tend to be more interpretable. For example, in facial recognition, the decomposed components match to something more interpretable like, for example, the nose, the eyebrows, or the mouth.

A disadvantage is that NMF truncates negative values by default to impose the added constraint of only positive values. This truncation tends to lose more information than other decomposition methods.

Unlike PCA, it does not have to use orthogonal latent vectors, and can end up using vectors that point in the same direction.

### **NMF for NLP**

In the case of Natural Language Processing, NMF works as below given these inputs, parameters to tune, and outputs:

**Inputs**

Given vectorized inputs, which are usually pre-processed using count vectorizer or vectorizers in the form Term Frequency - Inverse Document Frequency (TF-IDF).

**Parameters to tune**

The main two parameters are:

* Number of Topics
* Text Preprocessing (stop words, min/max document frequency, parts of speech, etc)

**Output**

The output of NMF will be two matrices:

1. W Matrix telling us how the terms relate to the different topics.
2. H Matrix telling us how to use those topics to reconstruct our original documents.

### **Syntax**

The syntax consists of importing the class containing the clustering method:

**from sklearn.decomposition import NMF**

creating the instance of the class:

**nmf=NMF(n\_components=3, init='random')**

and fit the instance and create a transformed version of the data:

**x\_nmf=NMF.fit(X)**