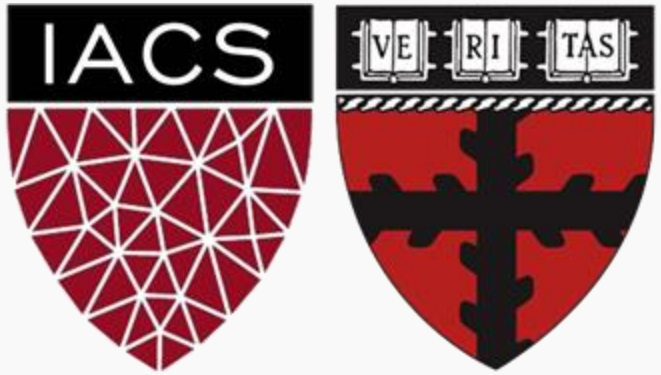
Principal Component Analysis

CS1009A Introduction to Data Science

Pavlos Protopapas, Natesh Pillai, Chris Gumb



Lecture Outline: High Dimensionality and PCA

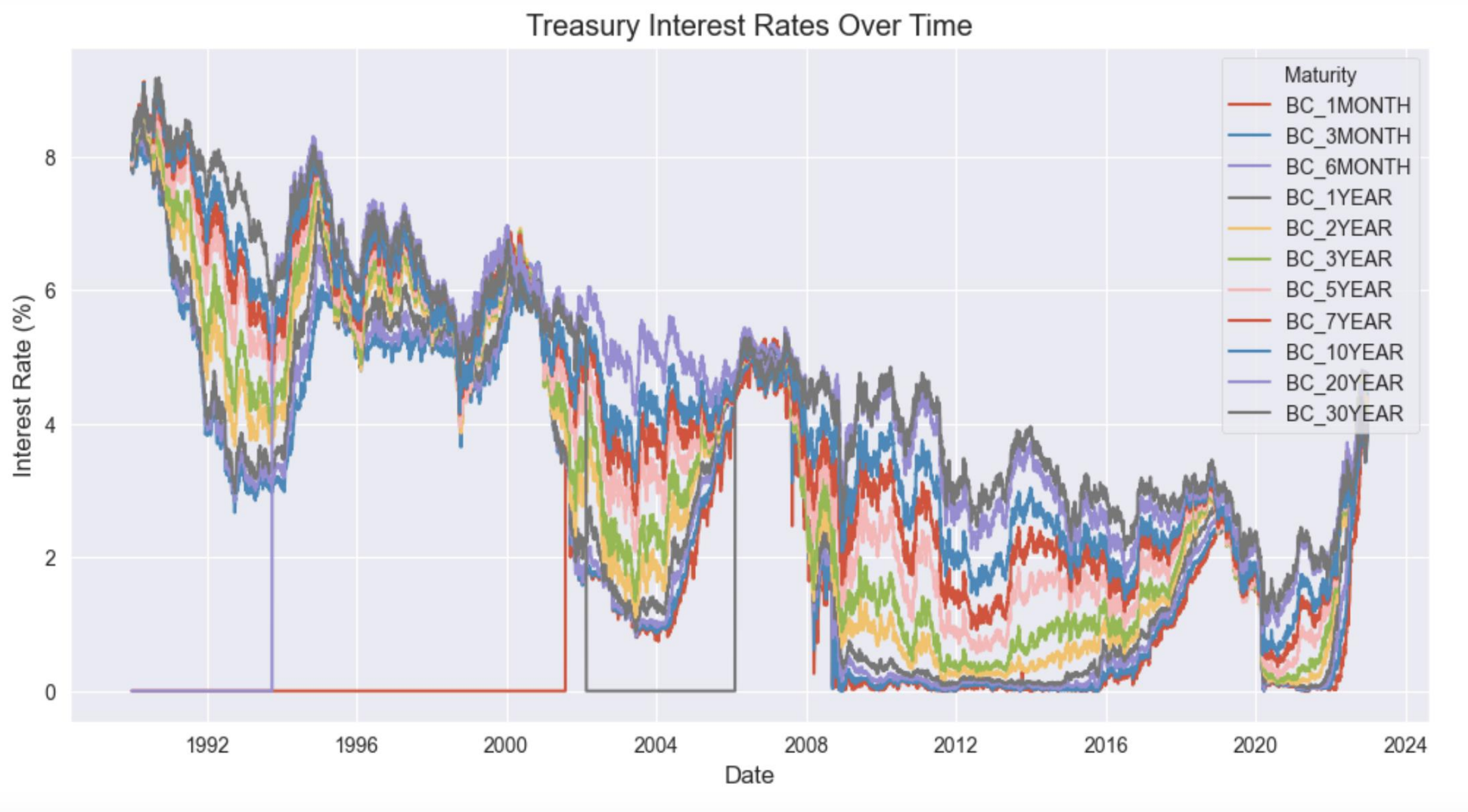
- Motivating Example -- High Dimensionality
- Principal Components Analysis (PCA)
- PCA for Visualization
- PCA for Regression (PCR)
- Non-linear Dimensionality Reduction (t-SNE)

Interest Rates Data from the US Treasury

date	BC_1MONTH	BC_3MONTH	BC_6MONTH	BC_1YEAR	BC_2YEAR	BC_3YEAR	BC_5YEAR	BC_7YEAR	BC_10YEAR	BC_20YEAR	BC_30YEAR
1990-01-02	0.00	7.83	7.89	7.81	7.87	7.90	7.87	7.98	7.94	0.00	8.00
1990-01-03	0.00	7.89	7.94	7.85	7.94	7.96	7.92	8.04	7.99	0.00	8.04
1990-01-04	0.00	7.84	7.90	7.82	7.92	7.93	7.91	8.02	7.98	0.00	8.04
1990-01-05	0.00	7.79	7.85	7.79	7.90	7.94	7.92	8.03	7.99	0.00	8.06
1990-01-08	0.00	7.79	7.88	7.81	7.90	7.95	7.92	8.05	8.02	0.00	8.09
...	...	...	...	...	...	...	...	...	...	...	...
2022-12-23	3.80	4.34	4.67	4.66	4.31	4.09	3.86	3.83	3.75	3.99	3.82
2022-12-27	3.87	4.46	4.76	4.75	4.32	4.17	3.94	3.93	3.84	4.10	3.93
2022-12-28	3.86	4.46	4.75	4.71	4.31	4.18	3.97	3.97	3.88	4.13	3.98
2022-12-29	4.04	4.45	4.73	4.71	4.34	4.16	3.94	3.91	3.83	4.09	3.92
2022-12-30	4.12	4.42	4.76	4.73	4.41	4.22	3.99	3.96	3.88	4.14	3.97

8256 rows × 11 columns

Time Series Plot of the Interest Rates Data



"Co-movement" of the time series.....

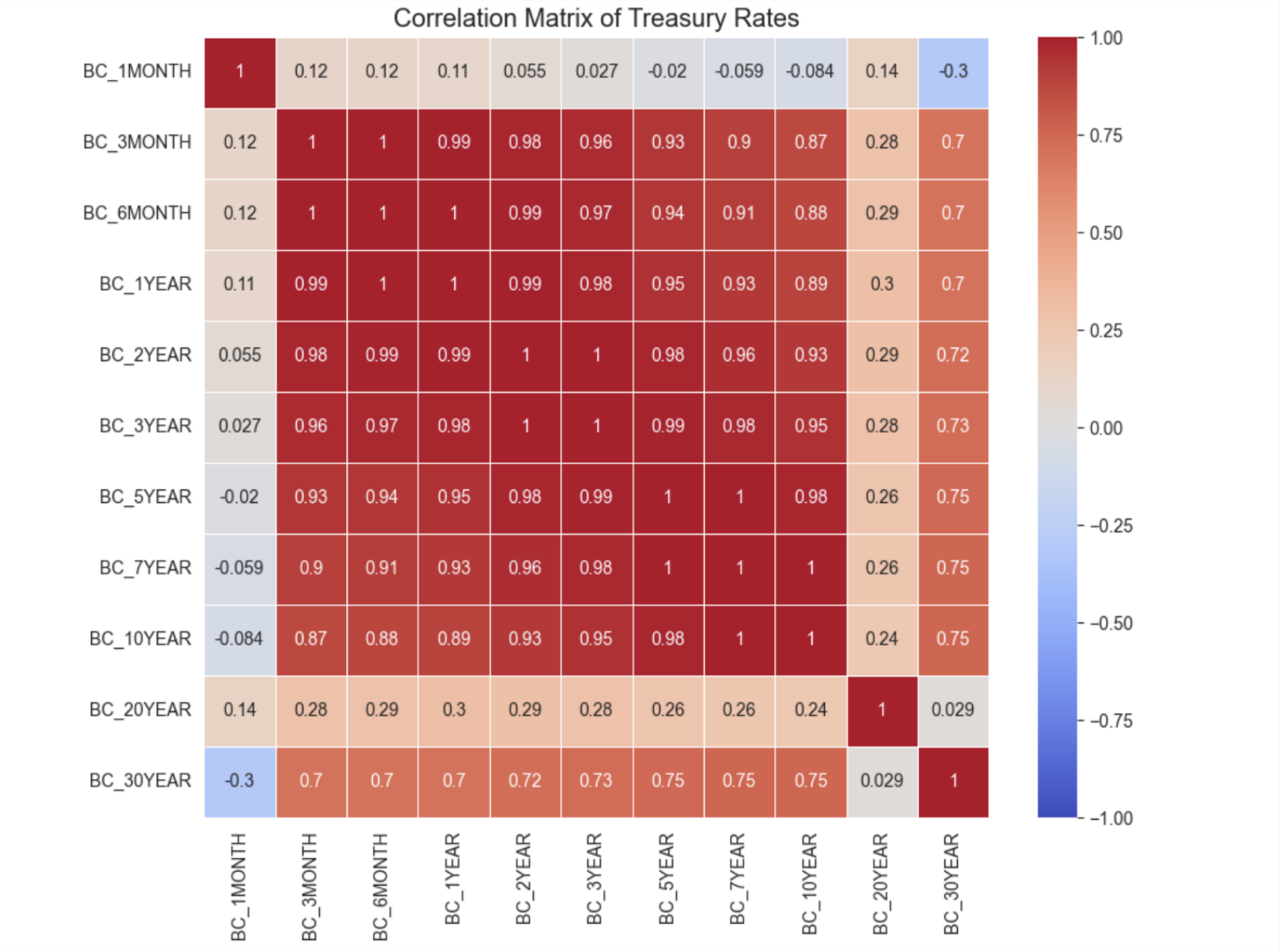
Visualizing high dimensional data

Visualization is often the quickest ways to gain insight into the relationships and patterns within a dataset.

But how can we possibly visualize data beyond 2 or 3 dimensions?

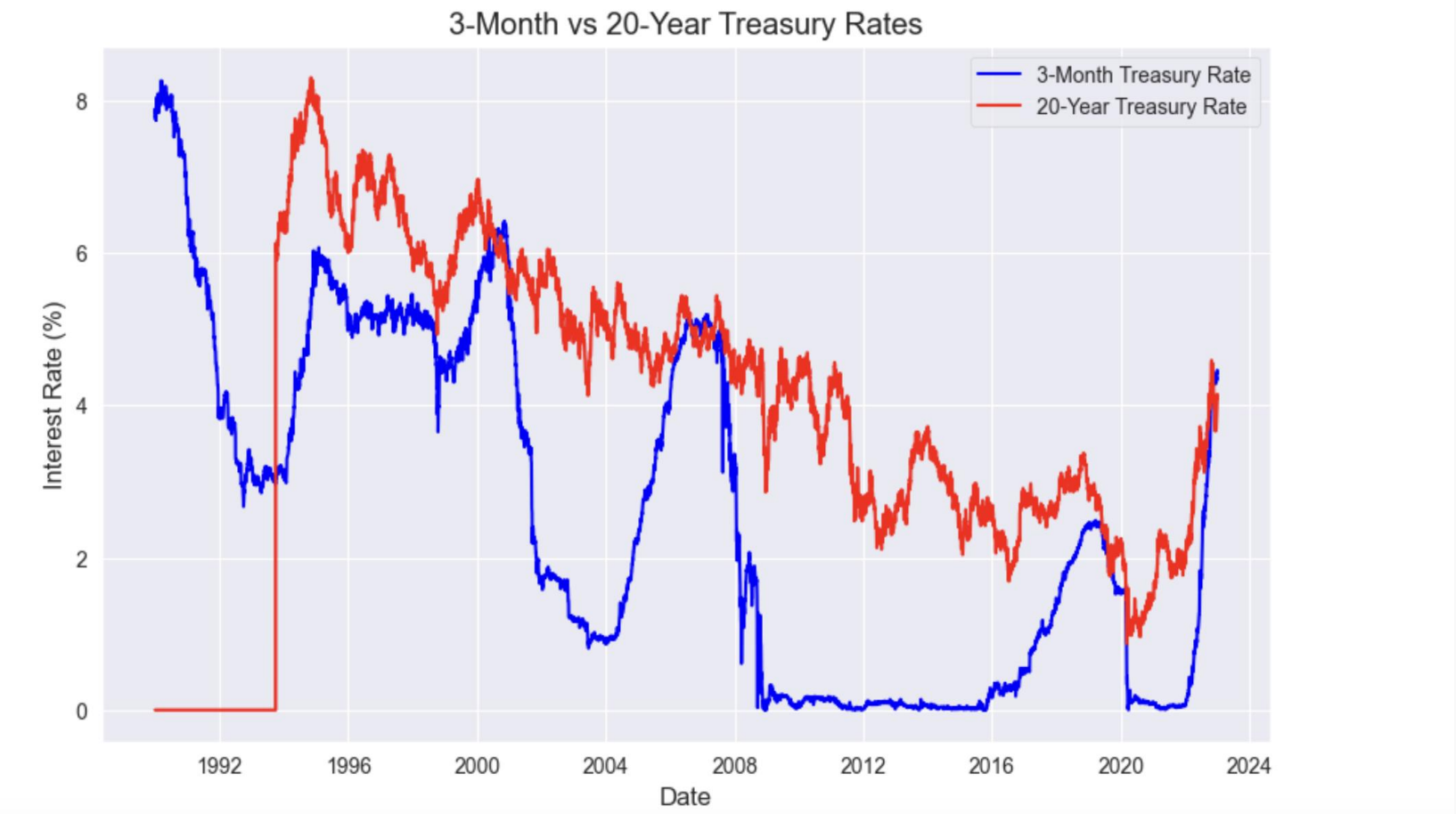
One option is to compromise and look only at small ‘slices’ of the predictor space at time.

For example, we can create a series of pair-wise 2-D scatter plots between all the predictors.

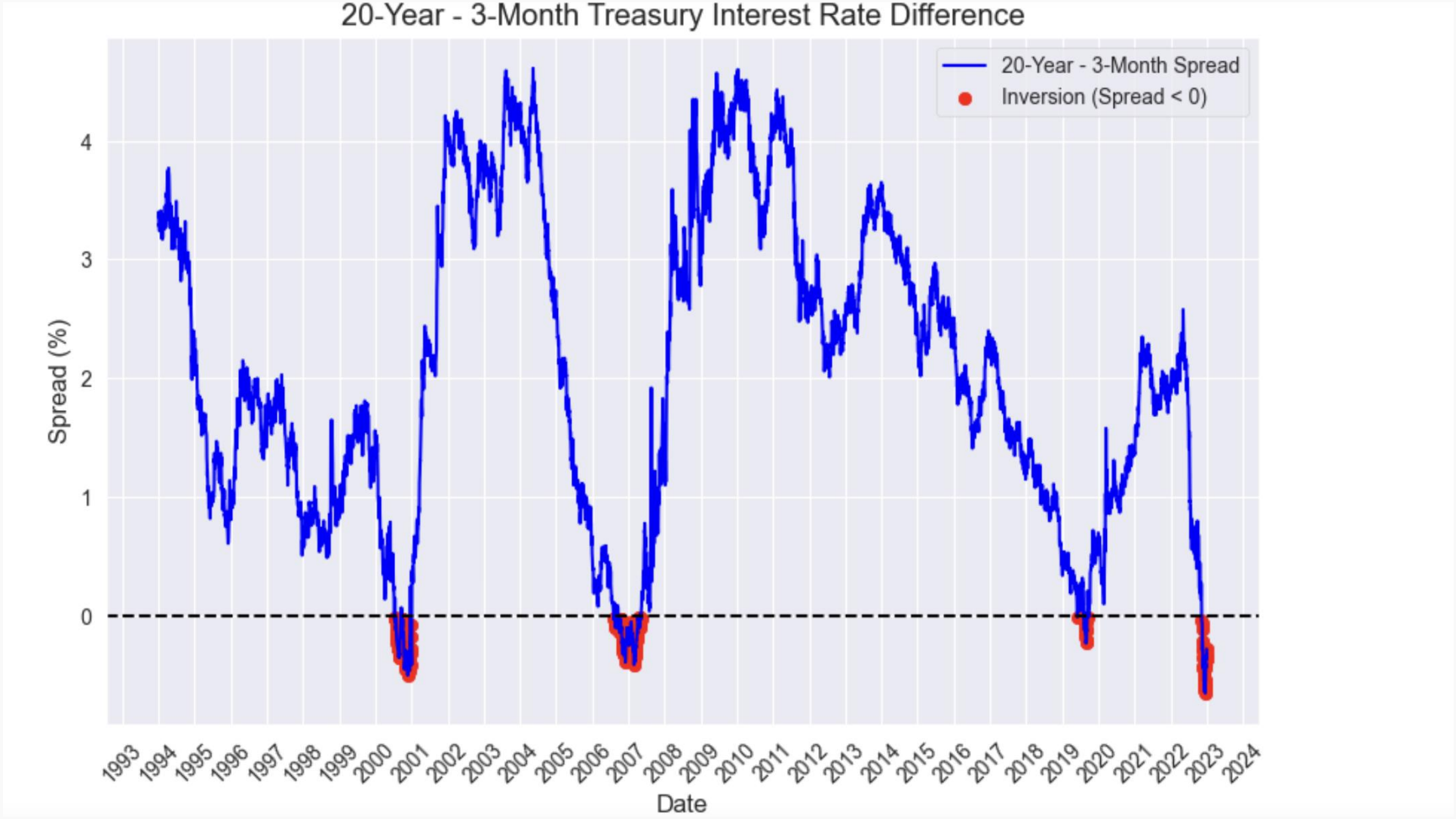


Correlation Matrix

Long Term vs. Short Term Rates.. Any patterns?



What happened the Market during Each of the Inversion Event?

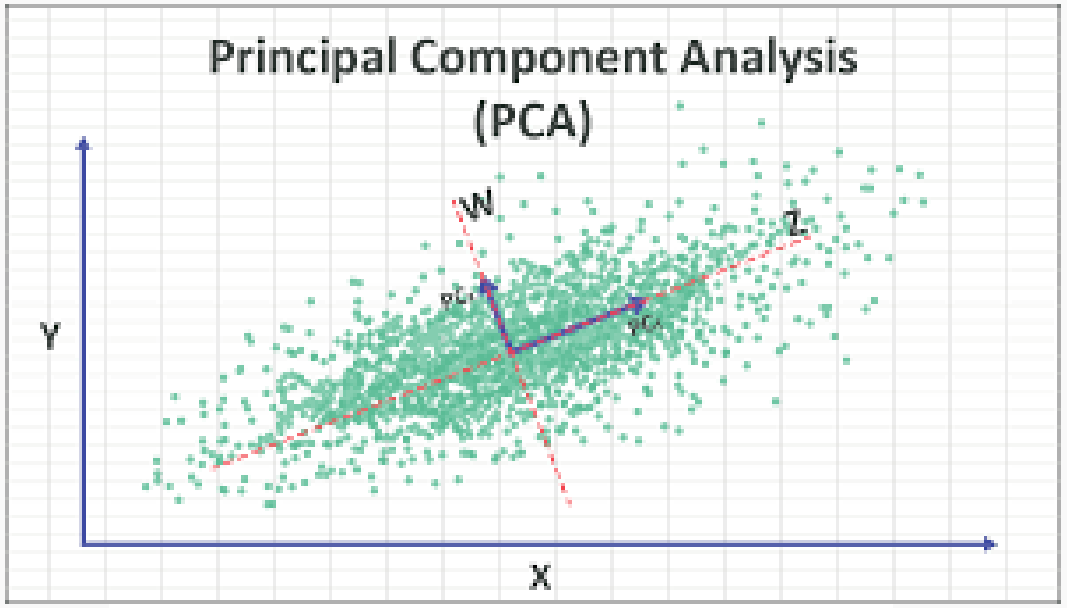


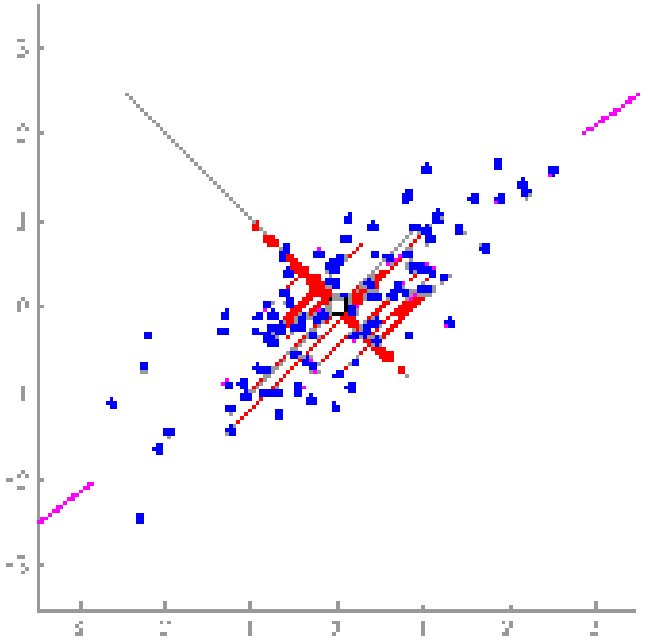
Why is this data/analysis different from Regression?

- In Regression, we have X (features) and Y (outcome). Thus we can relate Y to X. Labelled data.
- Called “Supervised” Learning.
- In our example, no “Y”; just X.
- X contains “interesting” patterns; how to extract these hidden patterns?
- Called “Unsupervised” Learning.

Principal Component Analysis (PCA)

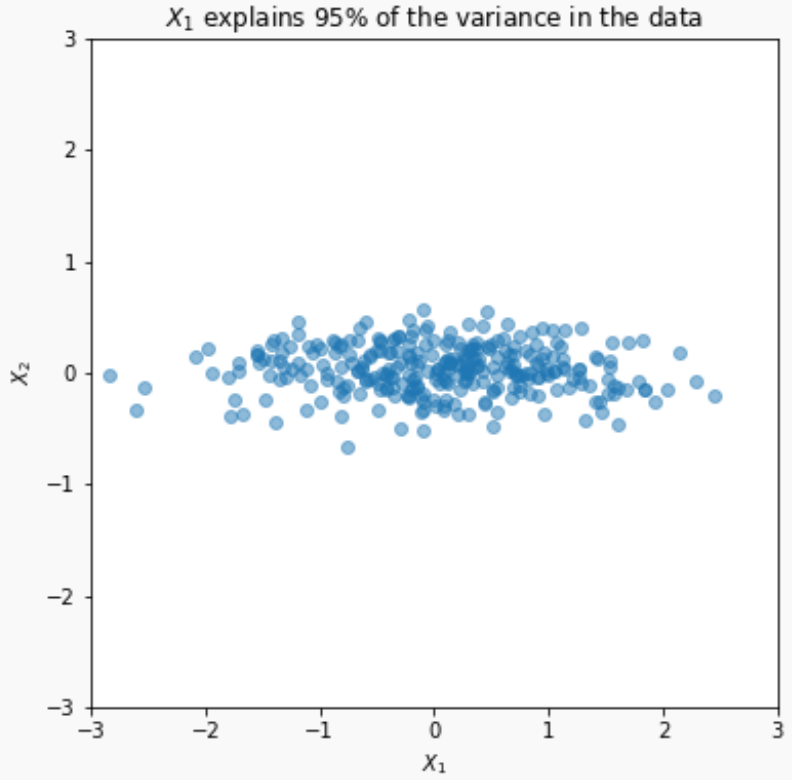
- PCA provides a “geometric” way looking at variation in data
- Find “directions” in which the variation is **maximum**
- These directions reveal hidden patterns in the data.
- Most importantly, it is a “dimension-reduction” technique. We only need to look at a few directions (even for large p)





A geometric view of looking at data

One strategy is to remove those predictors which explain the least amount of variance in the data, or, equivalently, retain those predictors that explain the most variance.

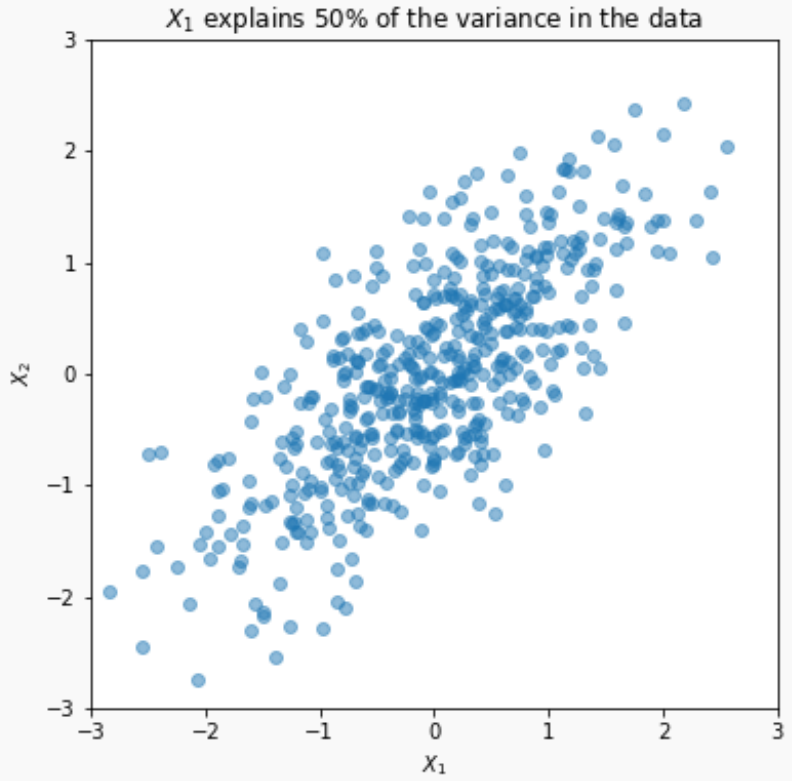


In this example, almost all the variance is explained by X_1.

Keeping only this predictor would retain most of the information while cutting the dimensionality in half.

An example with high correlation

A slightly different data set



We can clearly see there are *some directions in the predictor space* that have more spread than others. They just don't lie along the axes (i.e., the basis vectors)

Directions of maximum variance

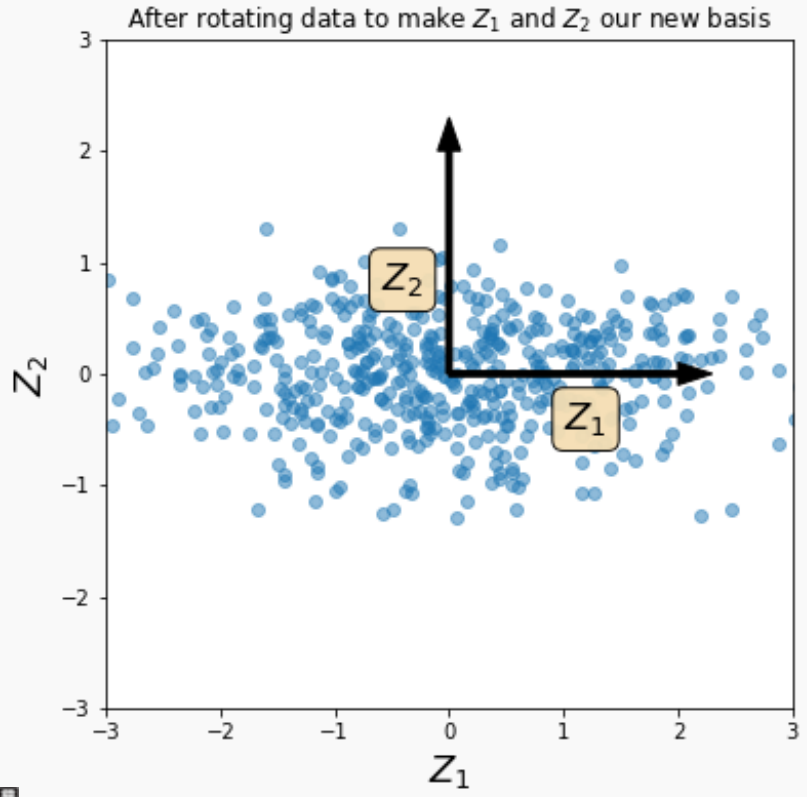
The vector Z_1 represents the direction of maximum variance in the predictor space. Z_2 is orthogonal and captures the remaining unexplained variance.

If our data were represented in terms of Z_1 and Z_2 then we could decide to keep only the Z_1 component.

This would again cut the dimensionality in half while retaining the most information.

Rotation for a change of basis

We can rotate our data so that the orthogonal Z_1 and Z_2 vectors now lie along the axes. This is equivalent to a change of basis from X_1 and X_2 to Z_1 and Z_2

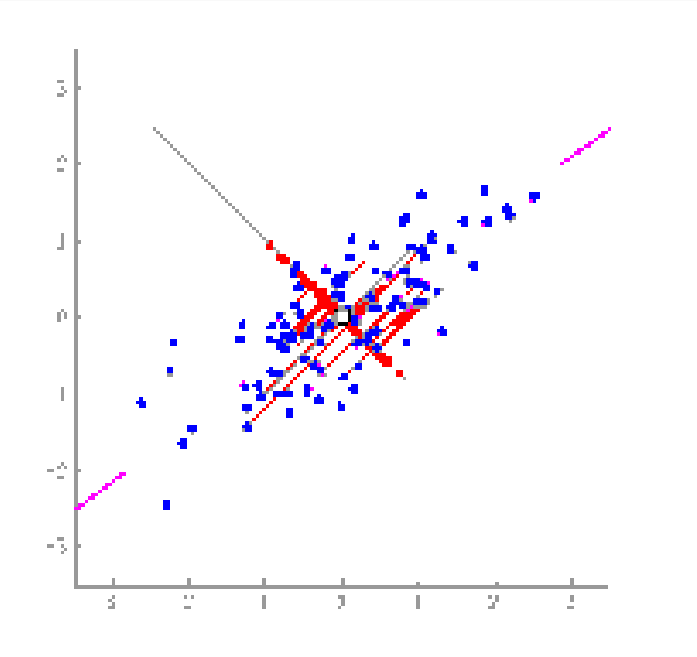


We can think of Z_1 and Z_2 as *a new set of predictors*. The decision of which predictor to keep to retain the most variance is once again clear.

Great! But how did we discover the optimal vectors Z_1 and Z_2!?

The Intuition Behind PCA

Transforming our observed data means **projecting** our dataset onto the space defined by the top *m* PCA components. These components become our new predictors.

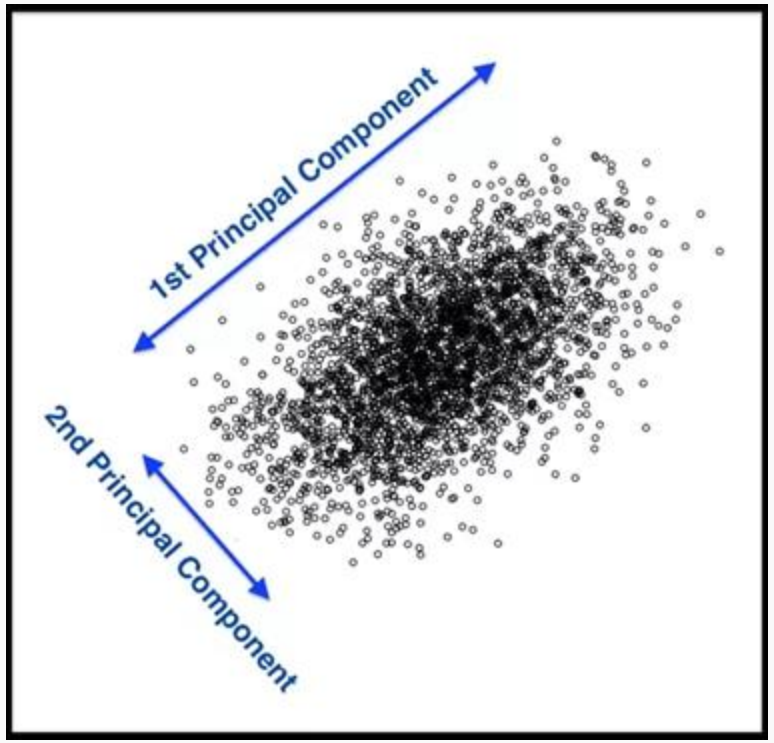


The animation shows projections onto several candidate vectors in red.

You can see the points retain the most spread when projected onto what turns out to be the first PCA component.

Principal Components Analysis (PCA)

Principal Components Analysis (PCA) is a method to identify a new set of predictors, as linear combinations of the original ones, that captures the maximum amount of variance in the observed data.



How are the PC components calculated?

Data(X)

X1	X2	X3
2.0	4.0	1.0
1.0	3.0	5.0
4.0	2.0	3.0
3.0	5.0	2.0
5.0	1.0	4.0

Loadings (l)

Loading 1	Loading 2	Loading 3
0.5	-0.3	0.2

$PC1 = +0.50 \times X1$

(PC1)

PC1
1.0
0.5
2.0
1.5
2.5

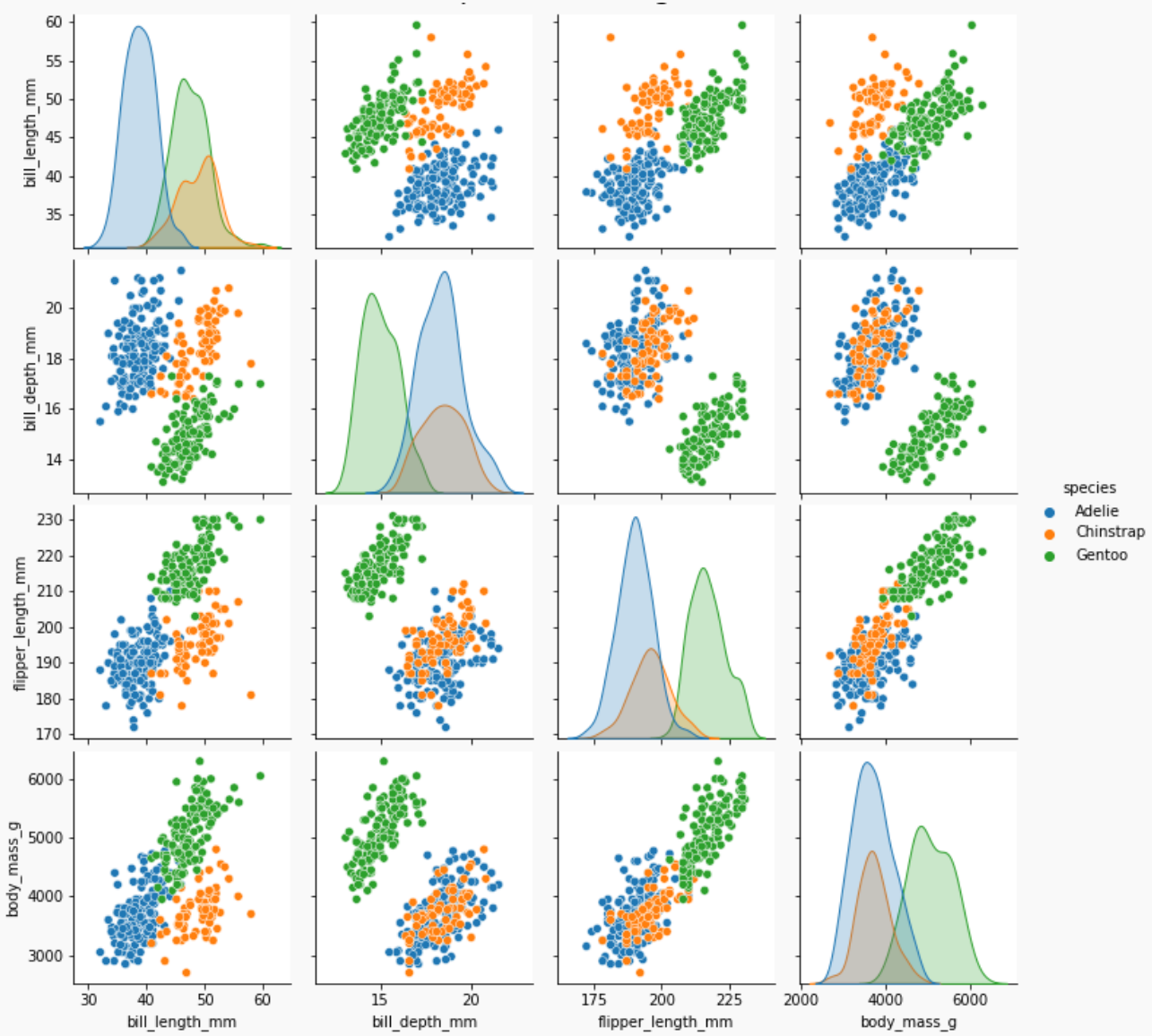
A simpler dataset: 3 Penguin Species

```
In [12]: penguins = sns.load_dataset('penguins')

In [13]: penguins.head()
```

Out[13]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	MALE
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	FEMALE
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	FEMALE
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	FEMALE

3 Penguin Species: Pair-plot of features



We have 4 measurements (4 *X*s) from 3 species of penguins: body mass, flipper length, bill length, and bill depth.

We can see some separation of the species in a few of the scatter plots. But what about relationships involving more than just 2 predictors?

This approach also becomes unwieldy with more than just a few predictors.

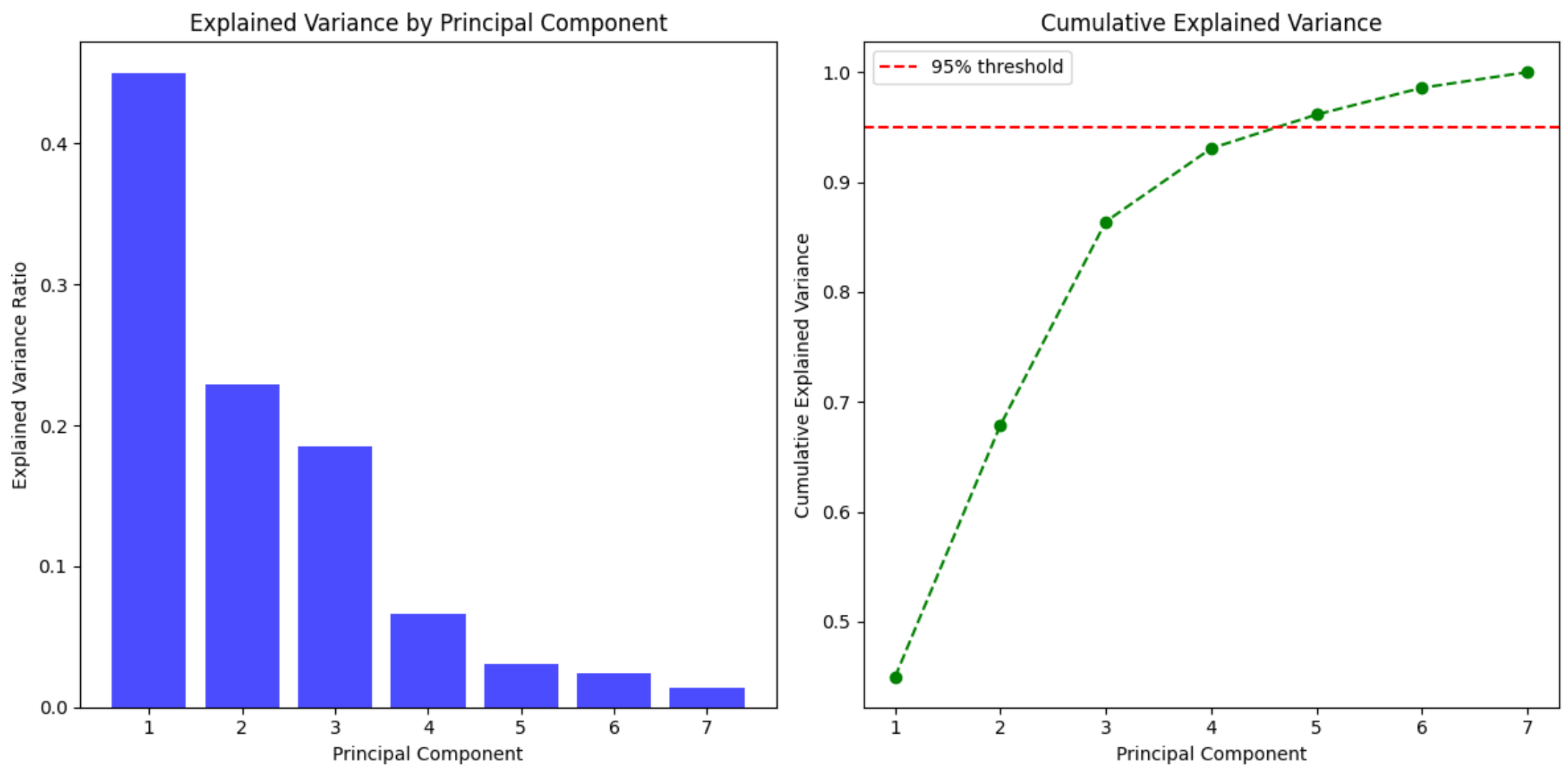
PCA fitting

```
X = pd.get_dummies(penguins, columns=['island', 'sex'], drop_first=True)
X = X.dropna()
```

```
X_std = StandardScaler().fit_transform(X.drop('species', axis=1))
```

```
# Perform PCA (keeping all components for now)
pca = PCA(n_components=None)
X_pca = pca.fit_transform(X_std)

# Explained variance ratio for each principal component
explained_variance = pca.explained_variance_ratio_

# Cumulative explained variance
cumulative_explained_variance = np.cumsum(explained_variance)
```

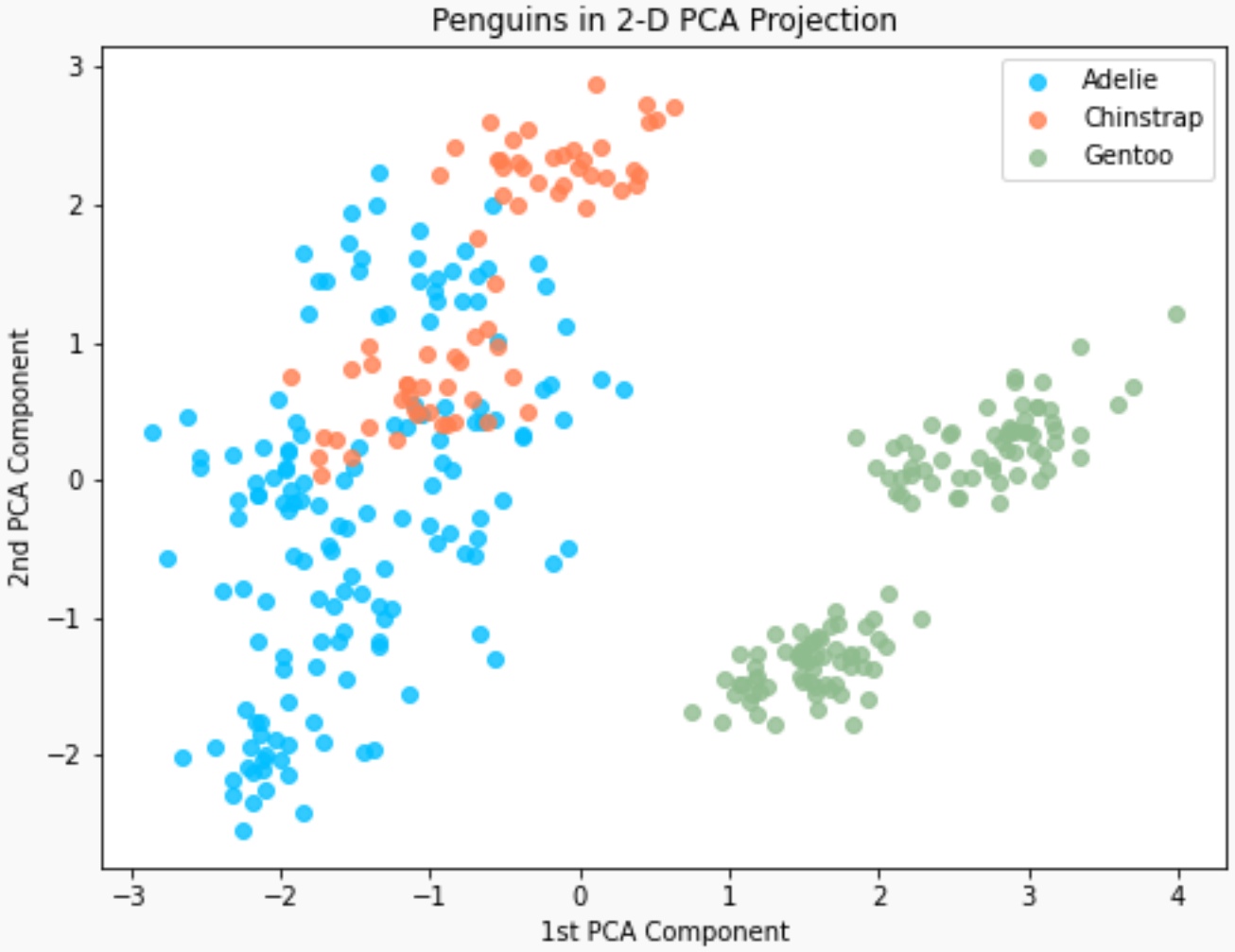
Explained Variance



Loadings: Unique only up to signs!!

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
bill_length_mm	0.410902	0.349188	-0.082807	0.678847	-0.441822	-0.187035	0.106528
bill_depth_mm	-0.357584	0.453812	0.364238	0.014070	-0.307439	0.661835	-0.027885
flipper_length_mm	0.535030	-0.031667	0.027397	0.104100	0.294947	0.402166	-0.672622
body_mass_g	0.524285	0.026440	0.175142	-0.162403	0.259477	0.322410	0.704344
island_Dream	-0.253579	0.546865	-0.393841	0.234672	0.648157	0.004212	0.079561
island_Torgersen	-0.219095	-0.327456	0.618131	0.573507	0.351530	-0.080348	0.061185
sex_Male	0.172792	0.513937	0.540233	-0.343184	0.110781	-0.504810	-0.171112

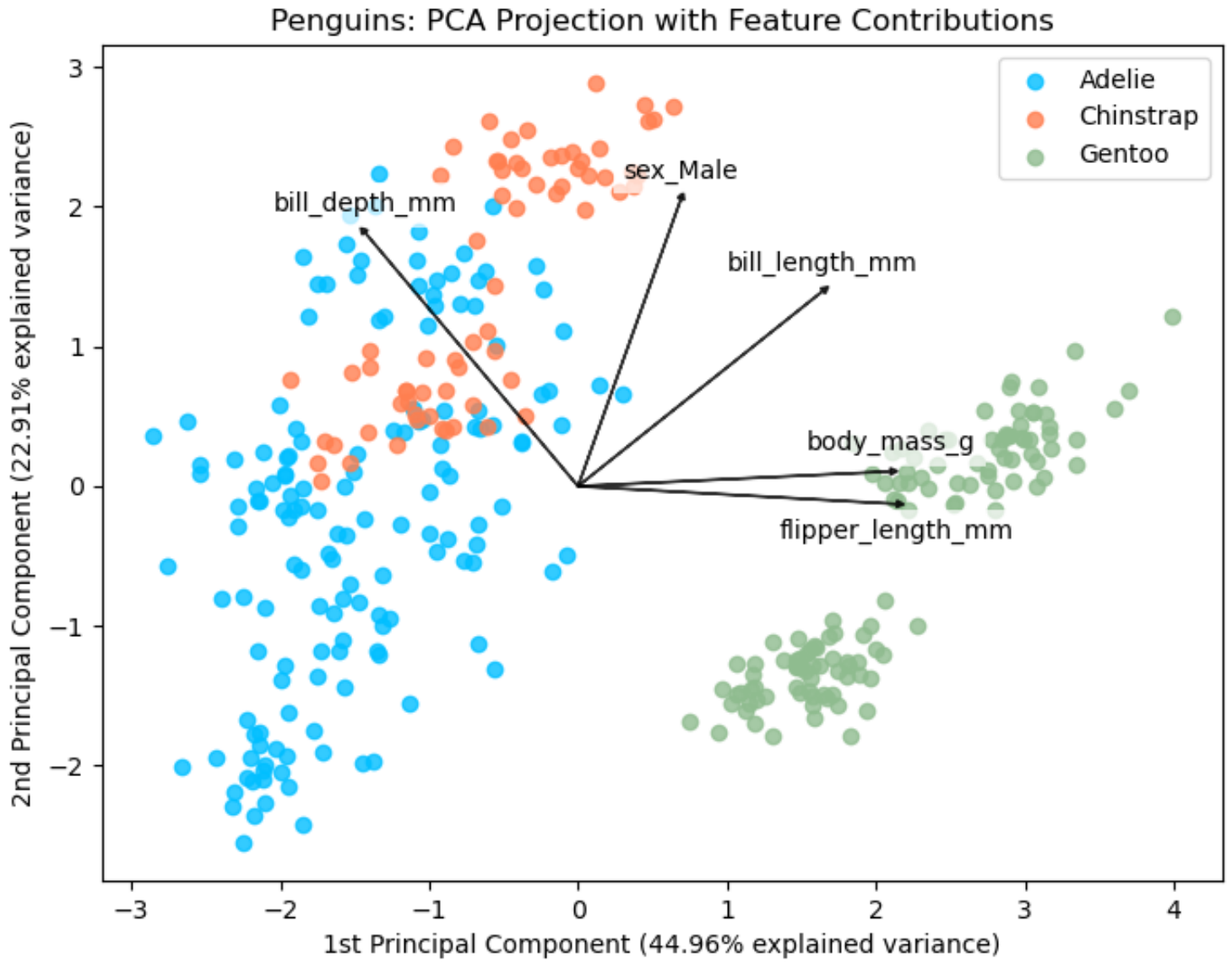
3 Penguin Species: Visualized with PCA



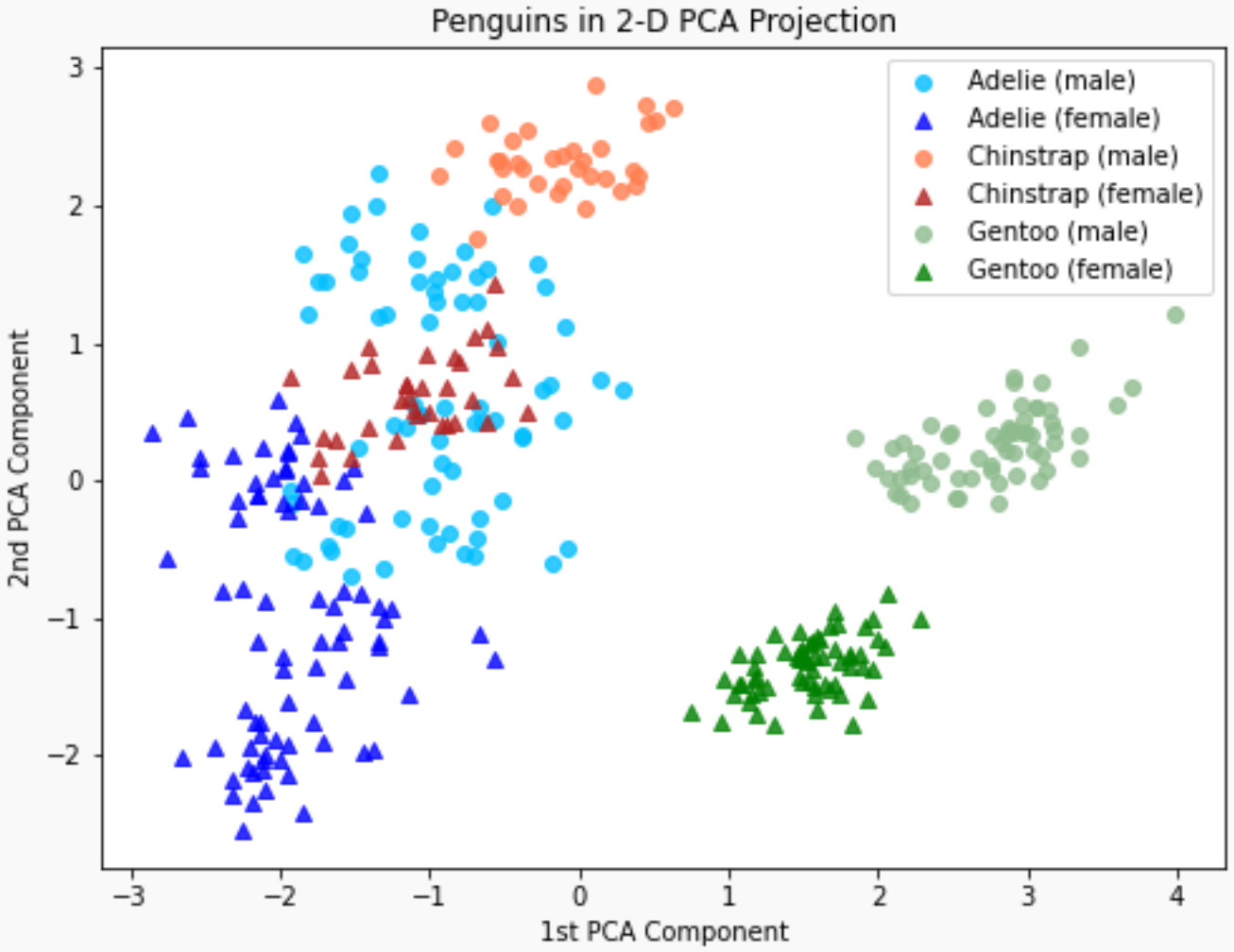
Here is the data projected onto the first 2 principal components.

Important point: we have massively reduced dimensions!!!

3 Penguin Species: Interpretability



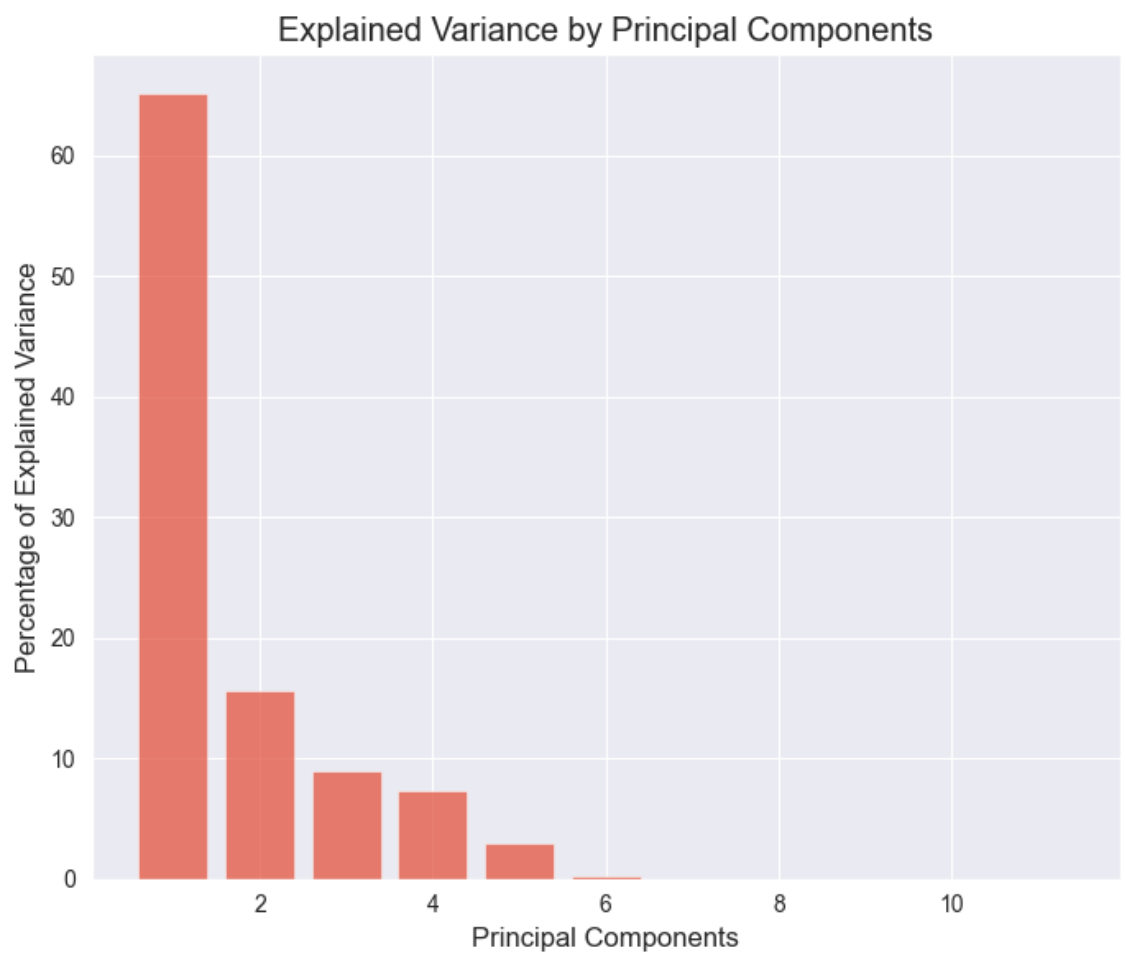
3 Penguin Species: Visualized with PCA

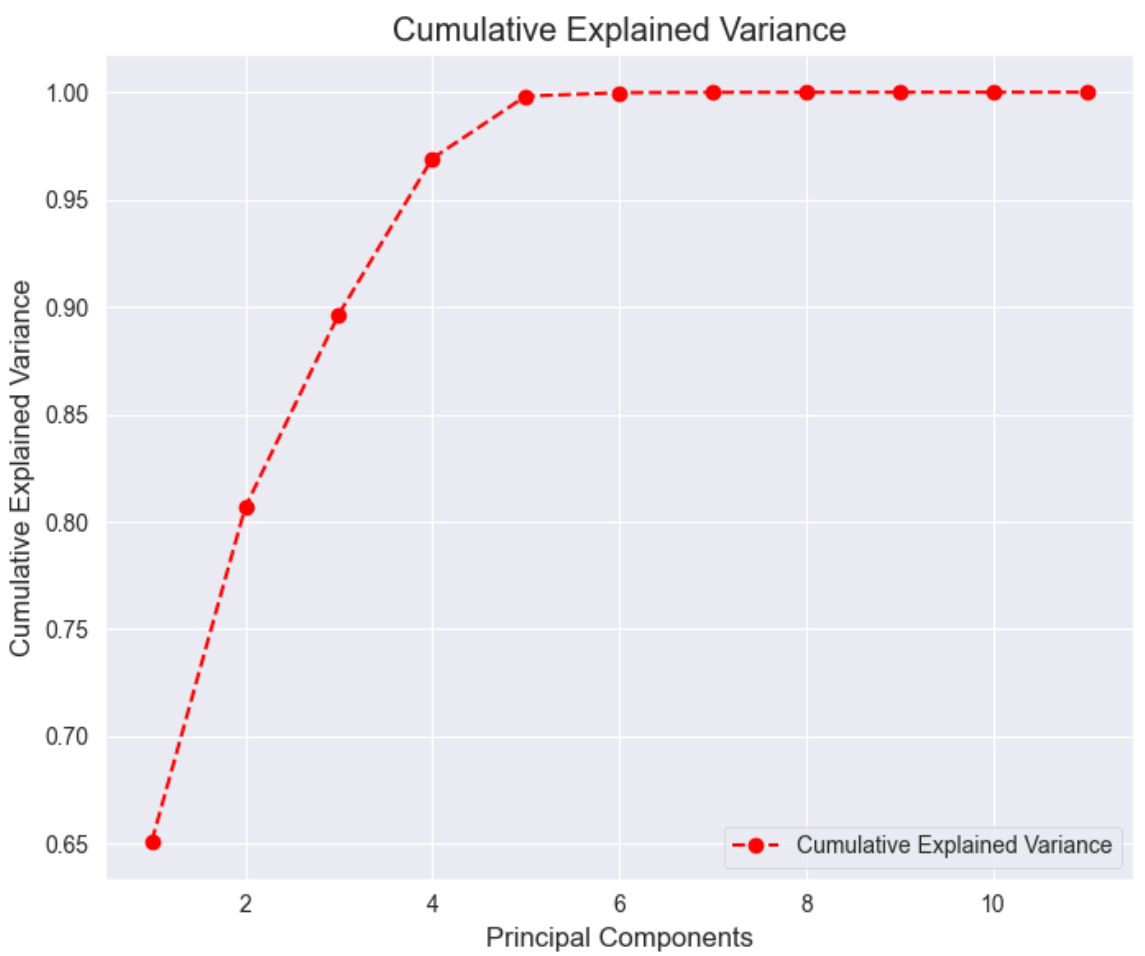


It turns out that these within species clusters correspond to male and female penguins.

Now let's try another example with a much higher dimensional dataset...

PCA for the interest rates data. First three Principal Components Dominate



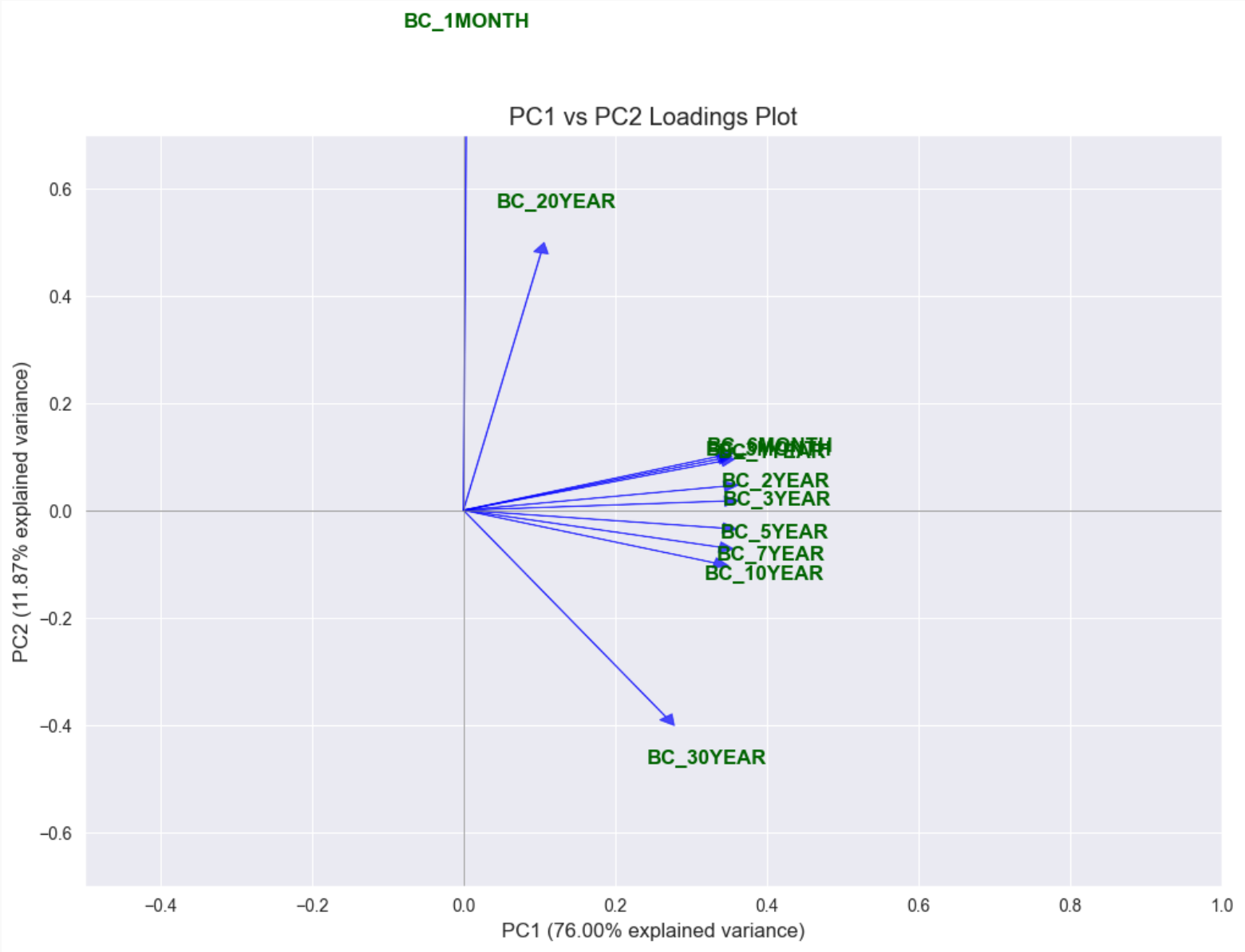


Understanding the Loadings

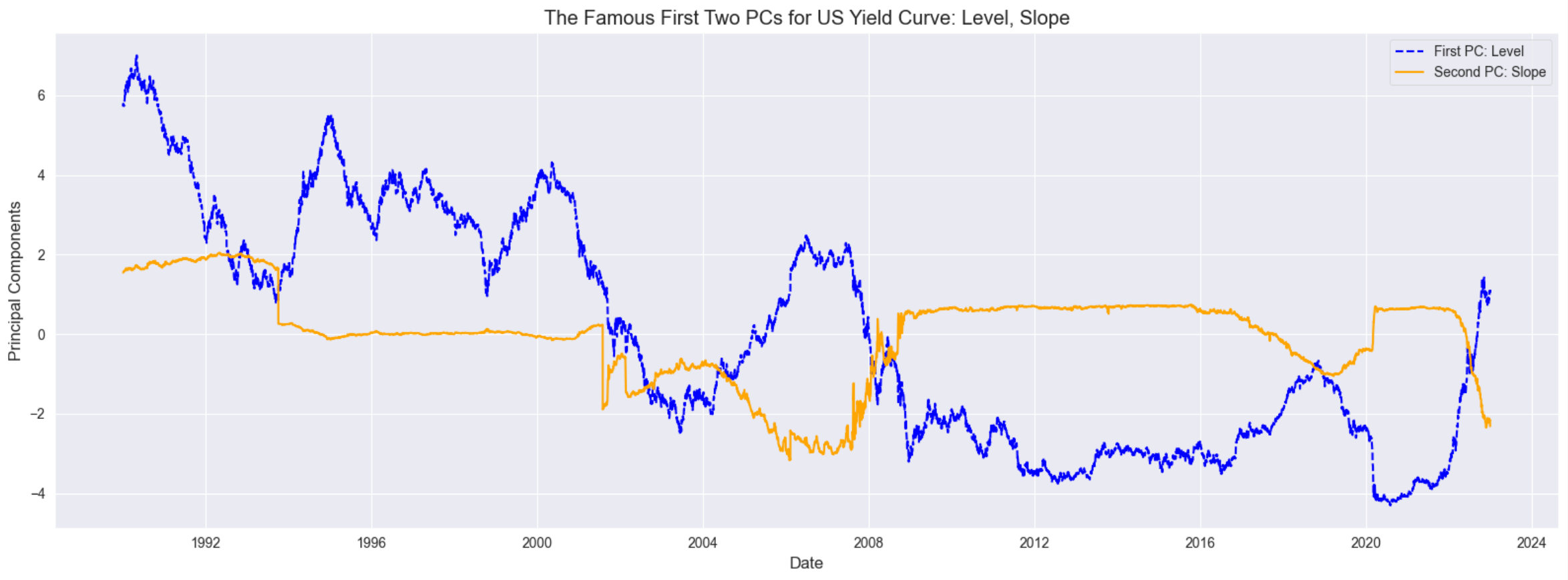
```
df_loadings
```

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
BC_1MONTH	0.003884	0.759330	-0.497909	0.261360	0.326251	-0.022919	-0.007349	-0.008473	-0.009024	0.002632	0.001284
BC_3MONTH	0.335358	0.095067	-0.098885	0.055223	-0.401221	0.537819	-0.483162	-0.370692	0.117841	0.174460	-0.016144
BC_6MONTH	0.336874	0.100567	-0.088690	0.043383	-0.377082	0.226047	0.196436	0.491260	-0.235412	-0.580388	0.034264
BC_1YEAR	0.339706	0.091051	-0.064455	0.006561	-0.308415	-0.129561	0.511945	0.203964	0.307951	0.597991	0.063948
BC_2YEAR	0.343529	0.044519	-0.037764	-0.079538	-0.149961	-0.432133	0.123669	-0.444379	-0.423842	-0.056834	-0.517096
BC_3YEAR	0.344336	0.017043	-0.021784	-0.127764	-0.014820	-0.456816	-0.175505	-0.214892	0.095228	-0.208806	0.725117
BC_5YEAR	0.341843	-0.033323	0.000477	-0.186855	0.228424	-0.204045	-0.347311	0.303758	0.568896	-0.166145	-0.434932
BC_7YEAR	0.337698	-0.068740	0.025266	-0.214769	0.357009	0.088821	-0.276198	0.365566	-0.554810	0.413768	0.100922
BC_10YEAR	0.330845	-0.098031	0.036843	-0.230143	0.502487	0.441122	0.471754	-0.331655	0.124917	-0.171576	0.044006
BC_20YEAR	0.102257	0.479515	0.852282	0.176357	0.042625	0.010165	-0.012244	-0.003277	0.001569	-0.006008	-0.001747
BC_30YEAR	0.266917	-0.385129	-0.002598	0.859627	0.196519	-0.052028	-0.010878	-0.003775	-0.002192	-0.001016	0.001171

Interests rate -- PC components overlay



The evolution of principal components overtime



The Math behind PCA

Let $\boldsymbol{X}$ be the n x p matrix with columns X_1,..., X_p (our original predictors), each standardized to have mean zero and variance one, and without the intercept)

Let let $\boldsymbol{W}$ be the p x p matrix whose columns are the eigenvectors of the **correlation matrix**, $\boldsymbol{X}^T\boldsymbol{X}$

Let $\boldsymbol{Z}$ be the n x p matrix with columns Z_1,..., Z_p (the principal components)

$$\mathbf{Z}_{n\times p} = \mathbf{X}_{n\times p}\mathbf{W}_{p\times p}$$

Implementation of PCA using linear algebra

To implement PCA yourself you can perform the following steps:

I. Standardize your predictors (so they each have mean = 0, var = 1).

II. Calculate the eigenvectors of $\boldsymbol{X^T X}$ and and arrange them as columns in order of descending eigenvalues in a new matrix, $\boldsymbol{W}$.

III. Use matrix multiplication to project $\boldsymbol{X}$ onto the eigenvectors to create the new feature matrix, $\boldsymbol{Z = XW}$

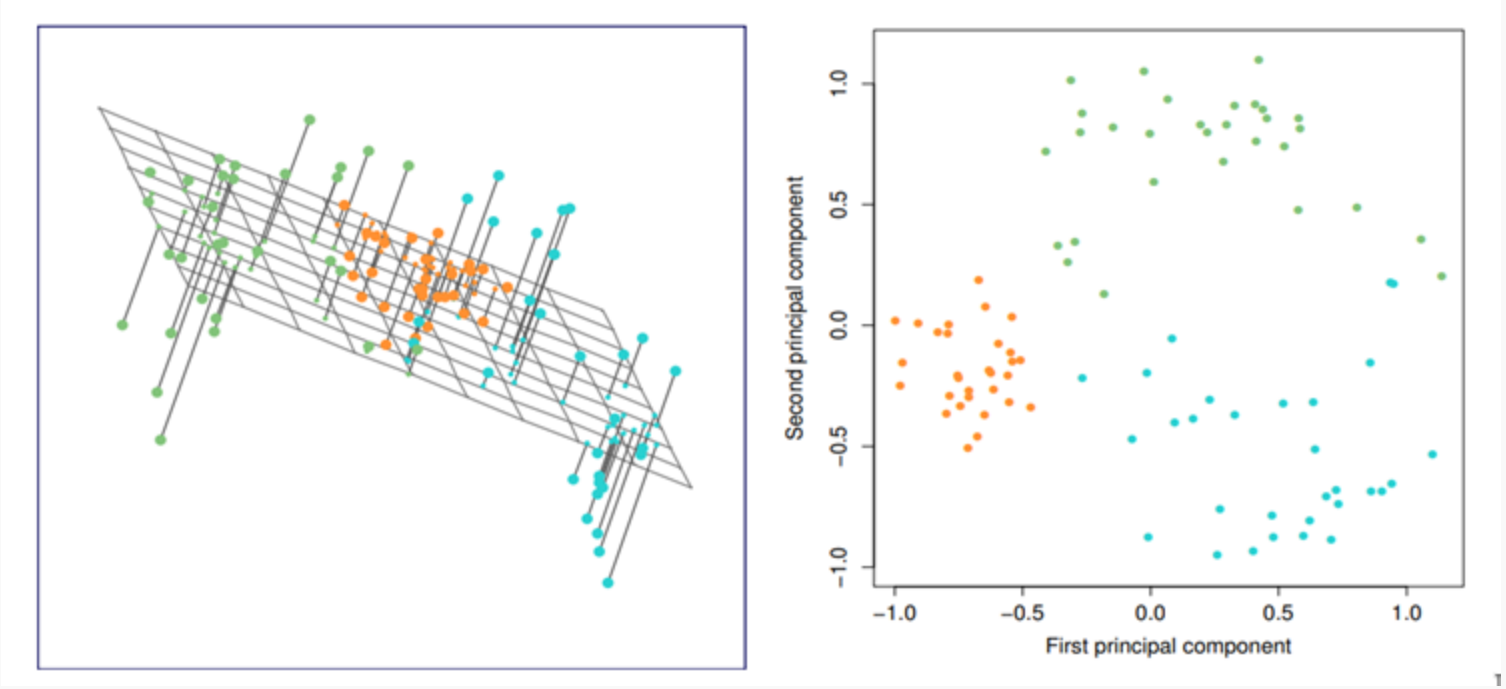
Note: this is not efficient from a computational perspective. This can be sped up using Cholesky decomposition.

PCA is easy to perform in Python using sklearn's decomposition.PCA class.

An Alternative Interpretation of PCA

We've seen an interpretation of PCA as finding the directions in the predictor space along which the data varies the most.

An alternative interpretation is that PCA finds a low-dimensional linear surface which is *closest* to the data points.



PCA example in sklearn

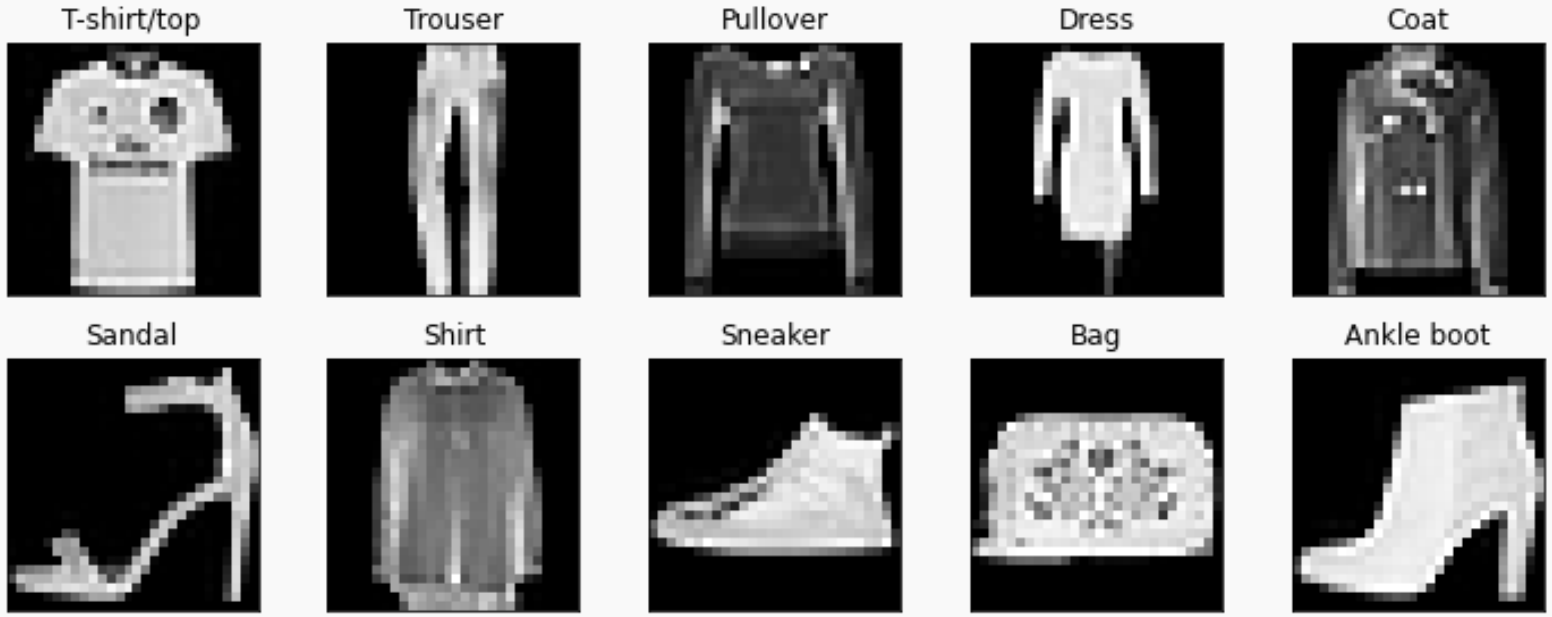
```
X = homes[['sqft','beds','baths','lotsize','dist']]

pca = PCA().fit(X)

pcaX = pca.transform(X)
W = pca.components_.T


print("First PCA Component vector (w1):",W[0,:].round(7))
print("Second PCA Component vector (w2):",W[1,:].round(7))

print("Variance explained by each component:",pca.explained_variance_ratio_.round(7))
```

```
First PCA Component vector (w1): [ 3.237999e-01  9.461243e-01 -7.237000e-04  9.225000e-04  1.009800e-03]
Second PCA Component vector (w2): [ 4.902000e-04  1.144900e-03  6.274349e-01 -7.753376e-01 -7.194070e-02]
Variance explained by each component: [9.097255e-01 9.027410e-02 2.000000e-07 1.000000e-07 1.000000e-07]
```

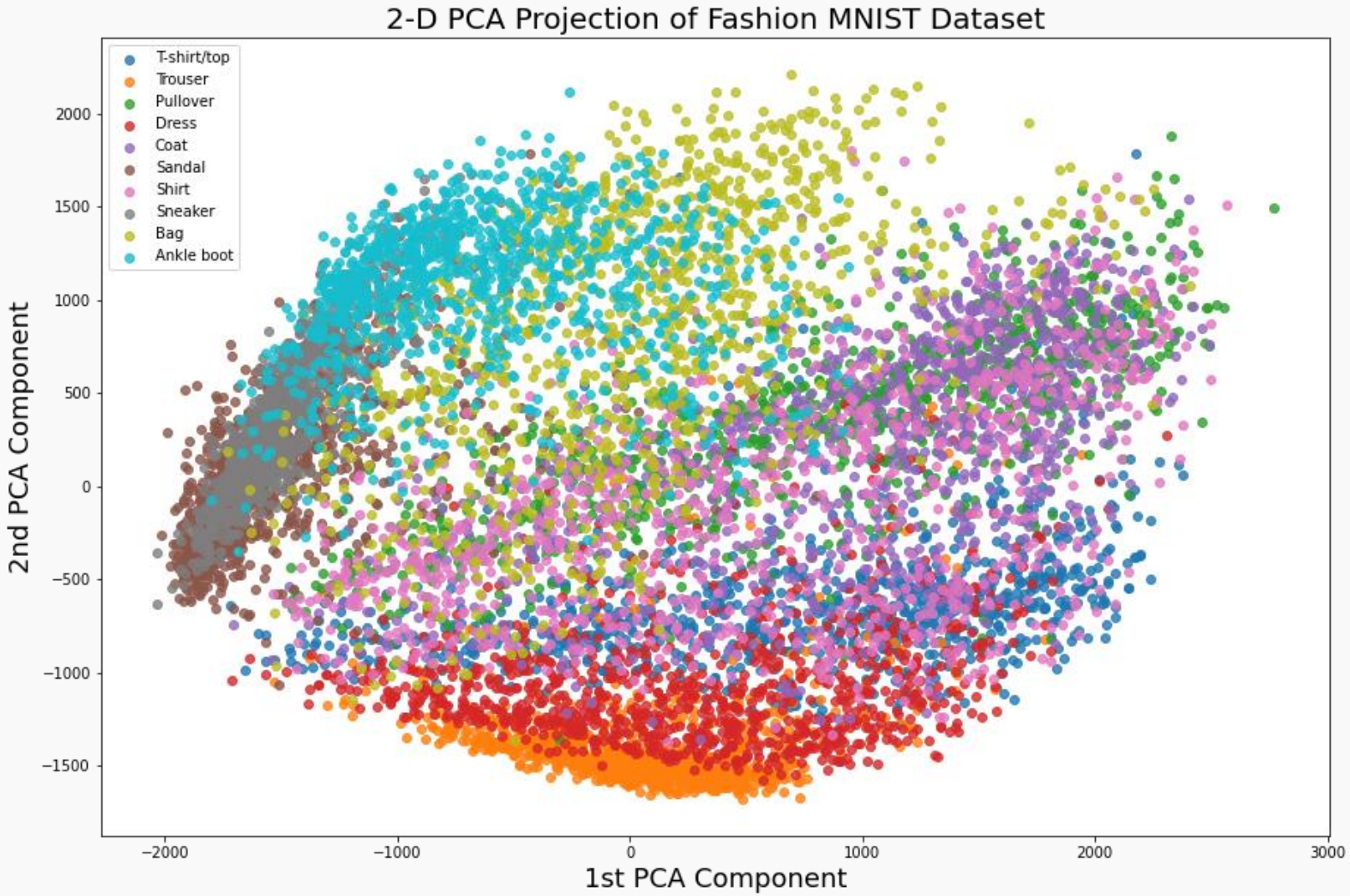
Images as high dimensional data

The **Fashion MNIST** data set consists of thousands of 28x28 grayscale images of articles of clothing from 10 different categories. Each pixel is essentially a feature. 28 x 28 = 784 features!

And obviously plotting pair-wise scatter plots of pixel values wouldn't be insightful.



PCA for visualizing image data

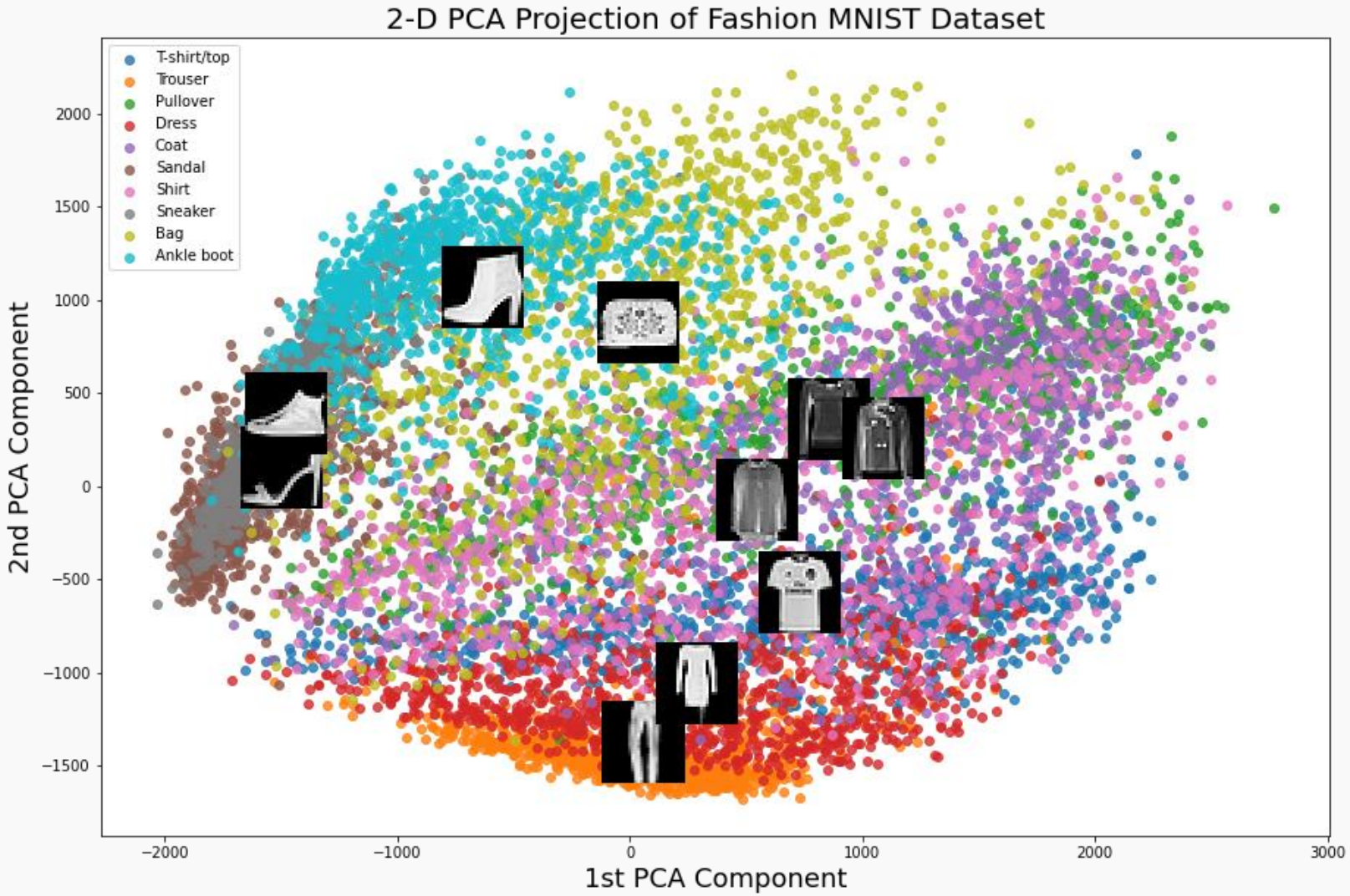


Projecting onto the top 2 principal components allows us to visualize the dataset.

While there exists considerable overlap, we see that most categories are grouped together.

And remember, PCA never actually saw the category labels!

PCA for visualizing image data



Displaying examples at the center of their respective cluster we can appreciate that similar categories are close to one another in the projected space.

All this was done by finding linear combinations of pixels that explained the most variance in the data!

Brief Interlude: What is 'Big Data'?

In the world of Data Science, the term *Big Data* gets thrown around a lot. What does *Big Data* mean?

A rectangular data set has two dimensions: number of observations (n) and the number of predictors (p). Both can play a part in defining a problem as a *Big Data* problem.

What are some issues when:

- n is big (and p is small to moderate)?
- p is big (and n is small to moderate)?
- n and p are both big?

When *n* is big

A very large sample size can pose computational a challenge.

- Algorithms can take forever to finish. Estimating the coefficients of a regression model, especially one that does not have closed form (like LASSO), can take a while.
- Model selection and hyperparameters tuning, especially when using cross-validation, exacerbates the problem.

What can we do to fix this computational issue?

- Perform 'preliminary' steps (model selection, tuning, etc.) on a subset of the training data set. 10% or less can be justified.

Note: statistical inference is usually ok when *n* is VERY big.

Keep in mind, big *n* doesn't solve everything

The era of Big Data (aka, large *n*) can help us answer lots of interesting scientific and application-based questions, but it does not fix everything.

Remember the old adage: "**garbage in; garbage out**". If the data are not representative of the population, then modeling results can be terrible. Random sampling ensures representative data.

Xiao-Li Meng does a wonderful job describing the subtleties involved (WARNING: it's a little technical, but digestible): https://www.youtube.com/watch?v=8YLdIDOMEZs

When p is big

When the number of predictors is large (many interactions, polynomial terms, etc.), lots of issues can occur.

What are some issues that may arise?

- Matrices may not be invertible (issue in OLS).
- Multicollinearity is likely to be present
- Models are susceptible to overfitting

This situation is called *High Dimensionality* and needs to be accounted for when performing data analysis and modeling.

When Does High Dimensionality Occur?

The problem of high dimensionality can occur when the number of parameters exceeds or is close to the number of observations. This can happen when we consider lots of interaction terms, like in our previous example. But this can also happen when the number of main effects is high.

For example:

- When we are performing polynomial regression with a high degree and a large number of predictors.
- When the predictors are genomic markers (and possible interactions) in a computational biology problem.
- When the predictors are the counts of all English words appearing in a text.

How Does sklearn handle unidentifiability?

In a parametric approach: if we have an over-specified model ($p > n$), the parameters are unidentifiable: we only need $n - 1$ predictors to perfectly predict every observation ($n - 1$ because of the intercept).

So, what happens to the 'extra' parameter estimates (the extra $\beta's$)?

The remaining $p - (n - 1)$ predictors' coefficients can be estimated to be *anything!* Thus, there are an infinite number of sets of estimates that will give us identical predictions. There is not one unique set of $\beta's$!

Perfect Multicollinearity

The $p > n$ situation leads to perfect collinearity of the predictor set. But this can also occur with redundant predictors.

Let's see what sklearn does in this situation:

```
regress_sqft = sk.linear_model.LinearRegression().fit(X = homes[['sqft']], y = homes['price'])
print("Intercept =",regress_sqft.intercept_.round(2),", Slope =",regress_sqft.coef_[0].round(4))

regress_sqft2 = sk.linear_model.LinearRegression().fit(X = homes[['sqft','sqft']], y = homes['price'])
print("Intercept =",regress_sqft2.intercept_.round(2),", Slopes =",regress_sqft2.coef_.round(4))
```

```
Intercept = 247438.24 , Slope = 589.7823
Intercept = 247438.24 , Slopes = [294.8911 294.8911]
```

Multicollinearity: A Challenge for Regression

In datasets with many features, especially when they are highly correlated, traditional linear regression struggles.

Correlated predictors can lead to unstable coefficient estimates, making the model less reliable and harder to interpret.

This motivates **Principal Component Regression (PCR)**

PCR is a two-step technique designed to address multicollinearity and reduce dimensionality of the predictors for a regression problem.

PCA for Regression (PCR)

PCA Step: First, can we use the components derived from PCA to transform the original predictors, X, into a set of new uncorrelated predictors, Z.

Regression Step: We then perform linear regression using a subset of these Z predictors.

By using a subset, PCR can simplify the model, reduce overfitting, and improve performance on unseen data.

PCA for Regression (PCR)

PCA is easy in Python, so how can we use it for regression modeling in a real-life problem?

If we use all p of the new Z_i, then we have *not* reduced the dimensionality. To do so, we must select only the first m PCA variables, $Z_1,\ldots,Z_m$, to use as predictors for some $m < p$.

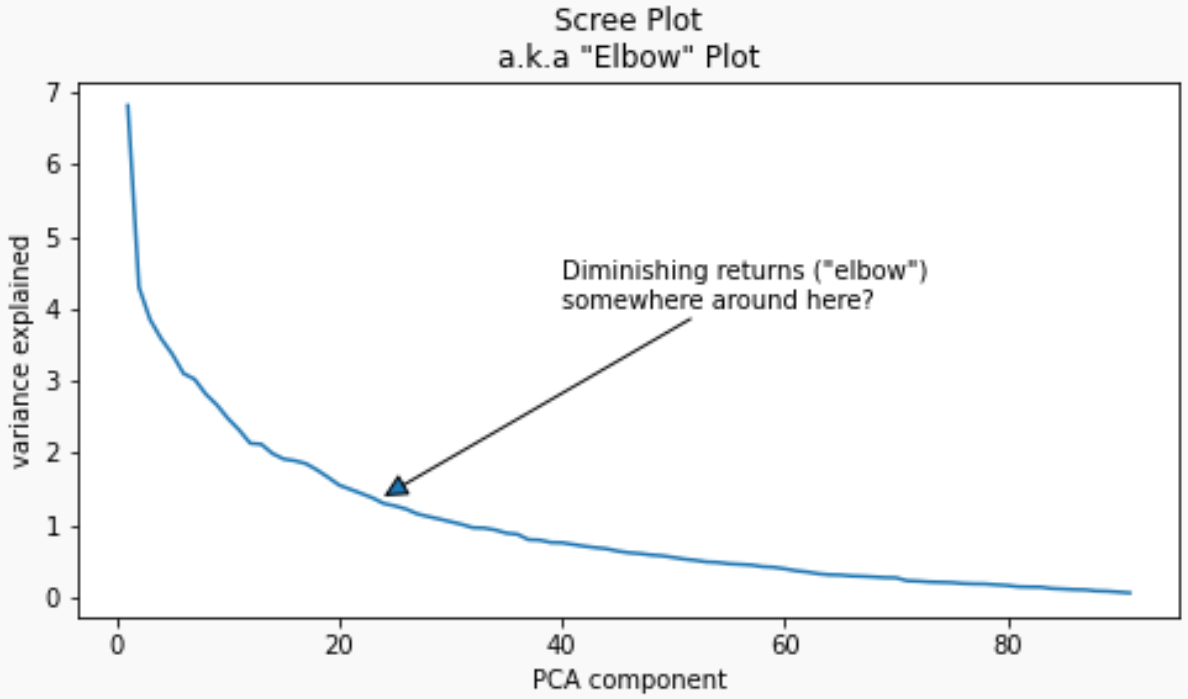
The choice of m is important and can vary from application to application. It depends on various things, like how collinear the predictors are, how truly related they are to the response, etc.

So how should we decide on a value of m?

How many components to keep?

One approach for deciding on the number of components to keep is to just “eyeball” it.

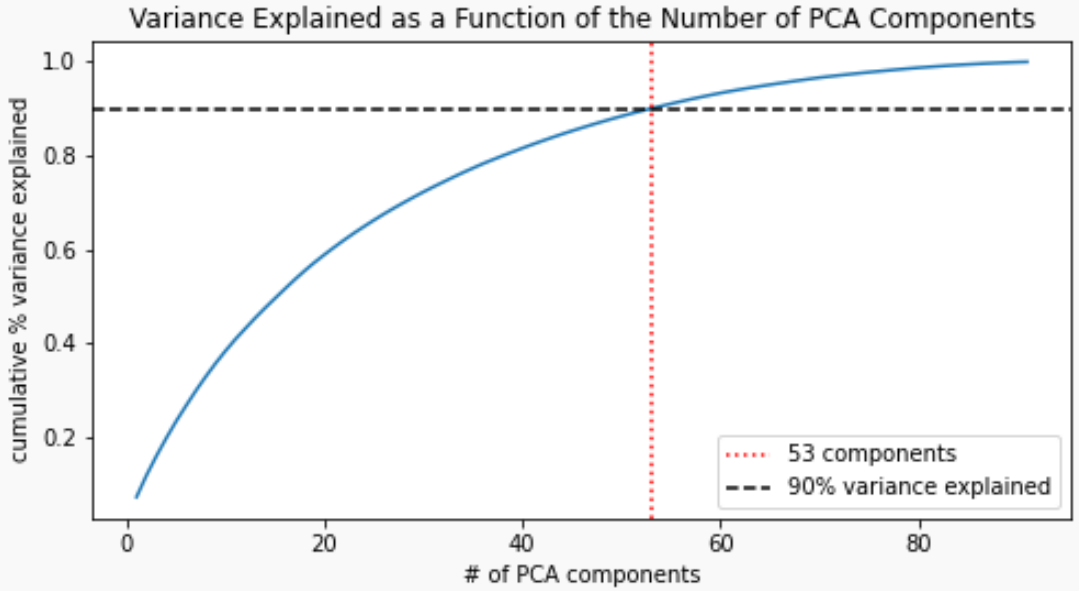
We look for a point of diminishing returns, or “elbow”, in a plot showing the variance explained by each component.



How many components to keep? (cont.)

Another approach is to specify how much of the total variance we want explained by our *m* components. For example, 90% of the total variance.

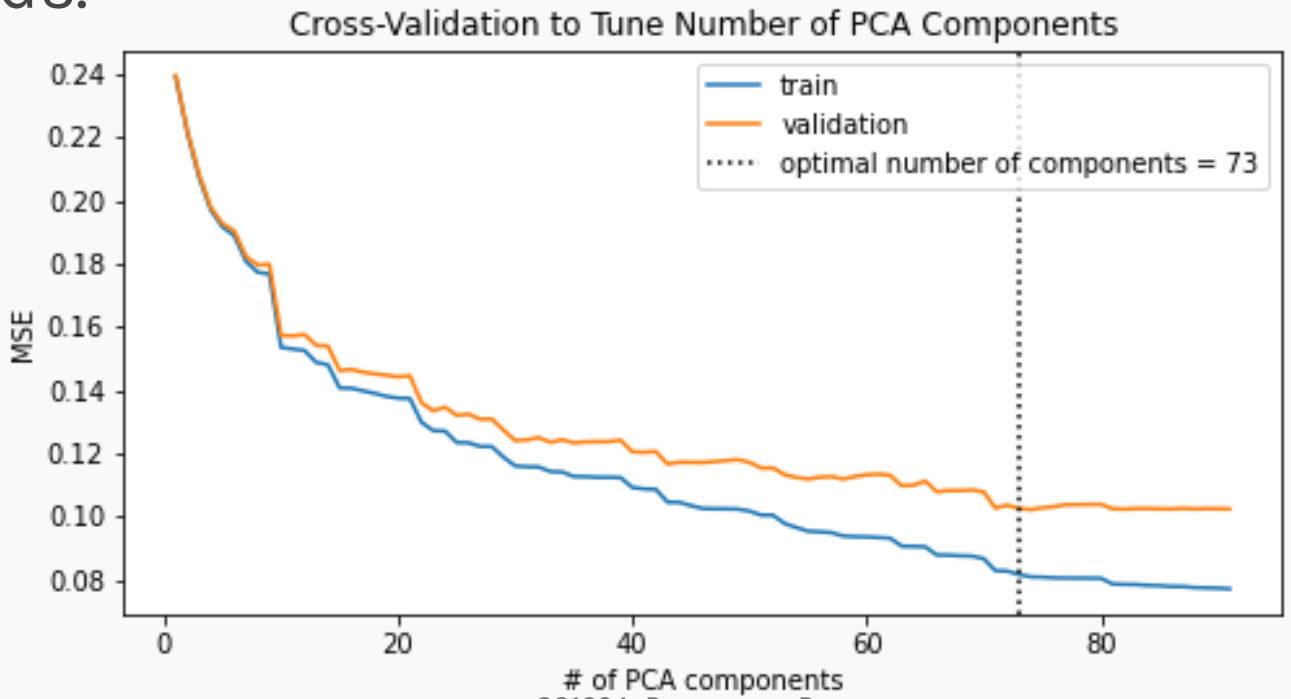
We then inspect the cumulative variance explained by each additional component and stop when we meet our goal.



Cross-validation to tune # of components in PCR

The two previous strategies **did not consider model performance!**

If modeling is your end goal, then you can use cross-validation to *tune* the number of components like any other hyperparameter, selecting the number that received the best mean validation error across all folds.



Interpreting the Results of PCR

A PCR can be interpreted in terms of original predictors…very carefully.

Each estimated β coefficient in the PCR can be *distributed* across the predictors via the associated component vector, *w*.

An example is worth a thousand words:

```
pcaX_df = pd.DataFrame(pcaX, columns=[f'PCA{i}' for i in range(1,X.shape[1]+1)])

PCR_simple = sk.linear_model.LinearRegression().fit(X = pcaX_df[['PCA1']], y = homes['price'])
print("PCA Intercept =",PCR_simple.intercept_.round(2),", PCA Slope =",PCR_simple.coef_[0].round(4))

print("First PCA Component vector (w1):",W[0,:])
```

```
PCA Intercept = 1244.2 , PCA Slope = 0.2049
First PCA Component vector (w1): [ 3.23799939e-01  9.46124307e-01 -7.23671683e-04  9.22493237e-04
  1.00975436e-03]
```

So how can this be transformed back to the original variables?

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 Z_1 = \hat{\beta}_0 + \hat{\beta}_1(\vec{w}_1^T X) = \hat{\beta}_0 + \left(\hat{\beta}_1 \vec{w}_1^T\right) X$$

$$= 1244.2 + 0.2049 \cdot (0.3238X_1 + 0.9461X_2 - 0.00072\, X_3 + 0.00092X_4 + 0.00101X_5)$$

$$= 1244.2 + 0.0663X_1 + 0.194X_2 - 0.00015\, X_3 + 0.00019X_4 + 0.00021X_5$$

Interpreting the Results of PCR

You can always put the PCR coefficients in terms of the original predictors.

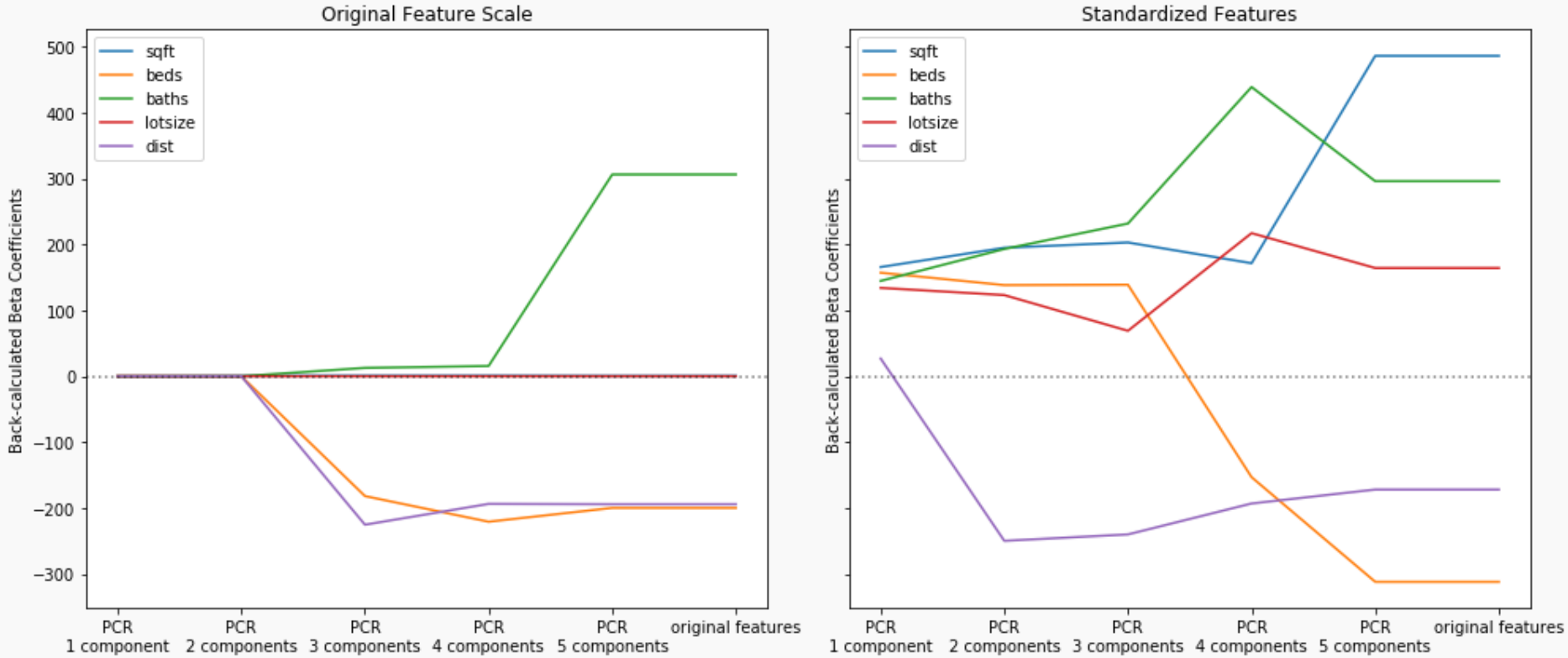
$$\hat{y} = \boldsymbol{Z}\beta_Z = (\boldsymbol{X}\boldsymbol{W}_m)\beta_Z = \boldsymbol{X}(\boldsymbol{W}_m\beta_Z) = \boldsymbol{X}\beta_X$$

$\boldsymbol{X}$ is our original dataset, centered to have a mean of zero
$\boldsymbol{W}_m$ is the matrix whose columns are the first m eigenvectors of $\boldsymbol{X}^\top\boldsymbol{X}$
$\boldsymbol{Z}_m$ is the transformed dataset created by projecting $\boldsymbol{X}$ onto the principal components contained in $\boldsymbol{W}_m$
β_Z are the coefficients for the PCR model
β_X are coefficients in terms of the original predictors

When $m = p$ then the recovered coefficients are identical to those from a model fit on the original predictors.

Coefficient 'paths' as more components enter the PCR

Coefficients for the PCR model using all components are identical to coefficients found when fitting on the original predictors.



Notice the effect standardizing has on the coefficients.

A few notes on using PCA

- PCA cons:
    1. PCA *knows nothing about the response variable*. Component vectors as predictors might not be ordered from best to worst!
    2. *Direct* interpretation of coefficients in PCR is lost, so if easy interpretation is important to you, perhaps steer clear.
    3. PCA can often fail to improve the predictive power of a model.
- PCA pros:
    1. Can help avoid the curse of dimensionality and overfitting.
    2. Easy to visualize how predictive your features are of the response variable, especially in the classification setting.
    3. Reduces multicollinearity, and so can improve the computational time when fitting models.

Matrix Completion (a brief aside)

Matrix Completion is another application of PCA and is suitable for imputing data which are missing at random. This approach is commonly used in *recommender systems* which deal in very large sparse matrices.

Consider an n x p matrix of movie ratings by Netflix customers where n is the number of customers and p is the number of movies. Such a matrix will certainly contain many missing entries.

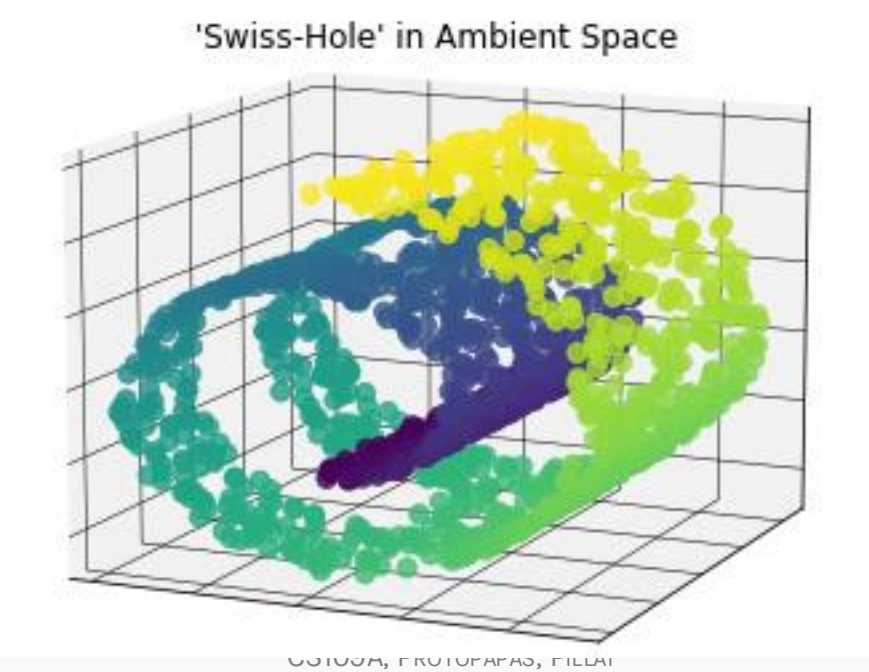
Imputing these missing values well is equivalent to predicting what customers will think of movies they haven't seen yet.

You can read more about the matrix completion algorithm in your textbook.
(section 12.3 pg. 510)

PCA: Limitations of Linearity

We've seen that PCA projects your data onto the hyperplane (i.e., a linear subspace) that minimizes the distance between the original data and the projected data…

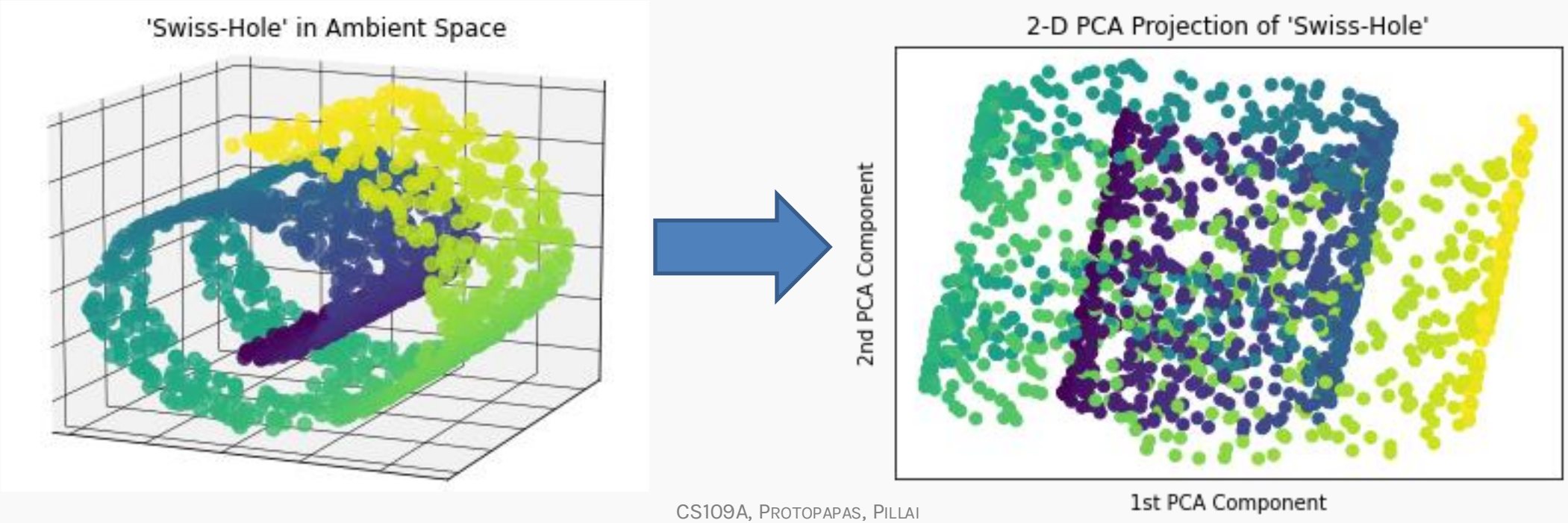
…but what if your data clearly do *not* lie on a hyperplane within the ambient space!



PCA ‘squashes’ the Swiss-Hole

What are the results of trying PCA here? It isn’t pretty.

PCA doesn’t respect the coiled, 2-D manifold embedded in the 3-D space. Points that were far apart on the original manifold end up close together in the PCA projection.



PCA: Preserving Global Pairwise Distances

You can think of PCA as striving to preserve the structure of the original data. PCA interprets 'structure' as 'pairwise distances.'

As a result, points which are (relatively) close in the original space should remain close in the projected space. Likewise, points which were distant in the original space should remain distance once projected.

A different strategy would be to relax this demand a bit. What if we focus on preserving only *'local'* structure? That is, keep nearby point close, but don't worry too much about preserving the pairwise distances of more distant points.

t-SNE: t-distributed Stochastic Neighbor Embeddings

The t-SNE algorithm attempts to minimize the following loss function:

$$\mathcal{L} = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

p_{ij} = affinity between points i and j in the **original** space

q_{ij} = affinity between points i and j in the **target** space

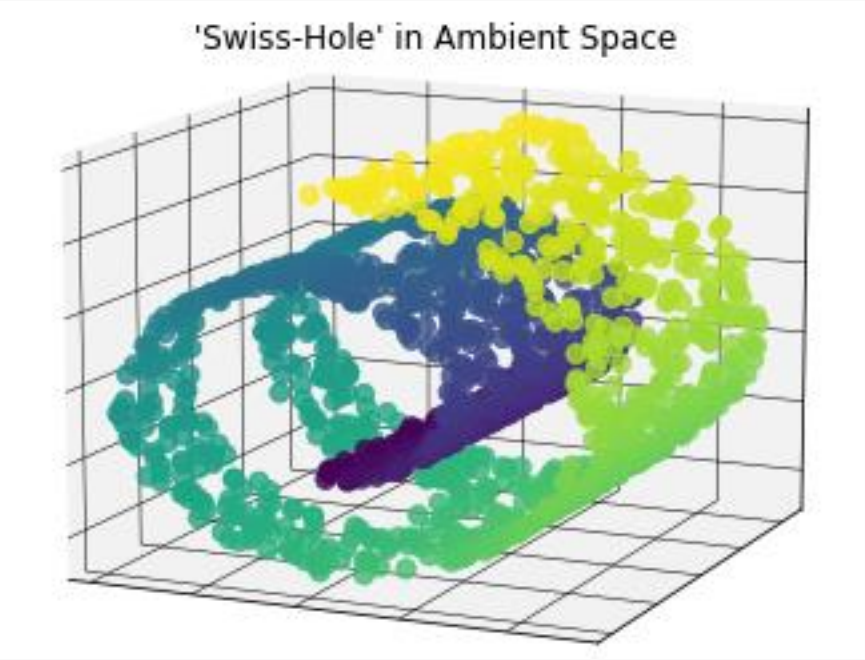
You can think of 'affinity' as roughly the inverse of distance. The p affinity is calculated using a Gaussian kernel normalized to sum to 1 across all points. For the q affinity, a t-distribution kernel is used.

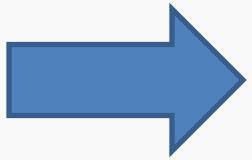
You can see that the loss suffers most when we map points that were originally close to points far away in the target space.

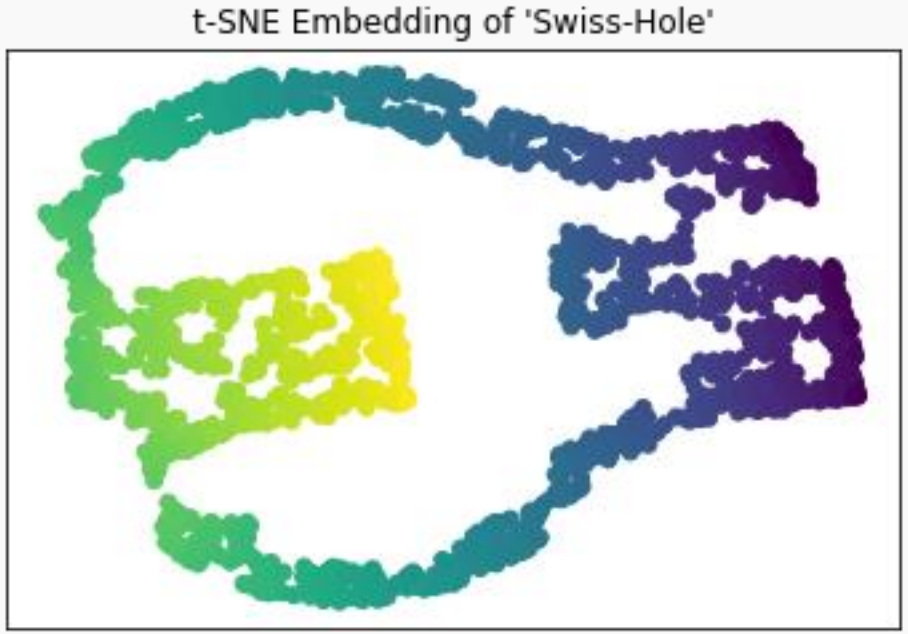
t-SNE ‘unrolls’ the Swiss-Hole

With t-SNE we’ve managed to reduce the dimensionality while preserving the *local* structure of the original data.

t-SNE is just one example of an algorithm that learns a low dimensional manifold that captures the *intrinsic* dimensionality of the original data.







Revisiting Fashion MNIST with t-SNE

The ‘swiss-role’ had an intrinsic dimensionality of 2 while being embedded in a 3-D ambient space.

Fashion MNIST data may have 784 predictors, but it is perhaps best to also think of this as the ambient space.

As an experiment, generate random vectors of 784 pixel values. How long until you happen to generate an image of a pair of pants? We’ll wait…

Images of clothes live on some lower dimensional manifold embedded within the space of all 28x28 grayscale images.

Assuming the manifold is linear is asking too much

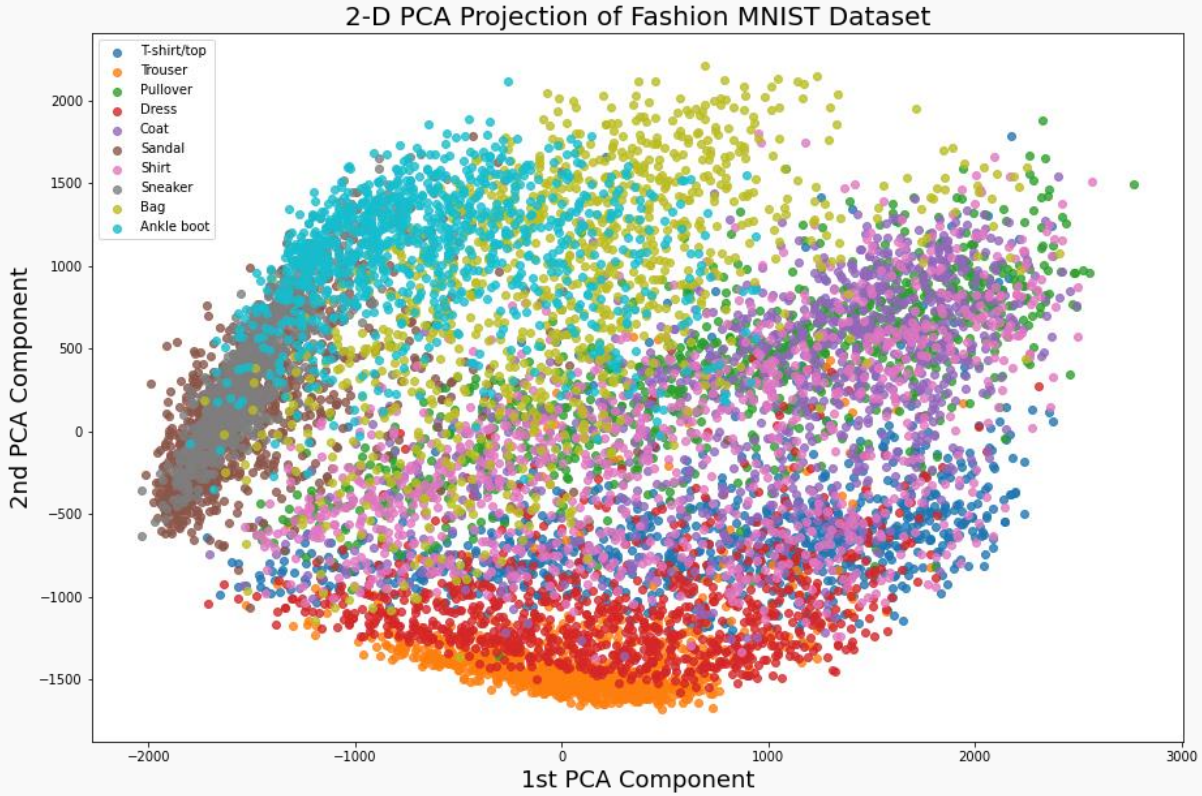
The space of 28x28 grayscale images include not just images of clothes, but all possible 28x28 images! Not to mention the countless ‘noise’ images depicting nothing at all.

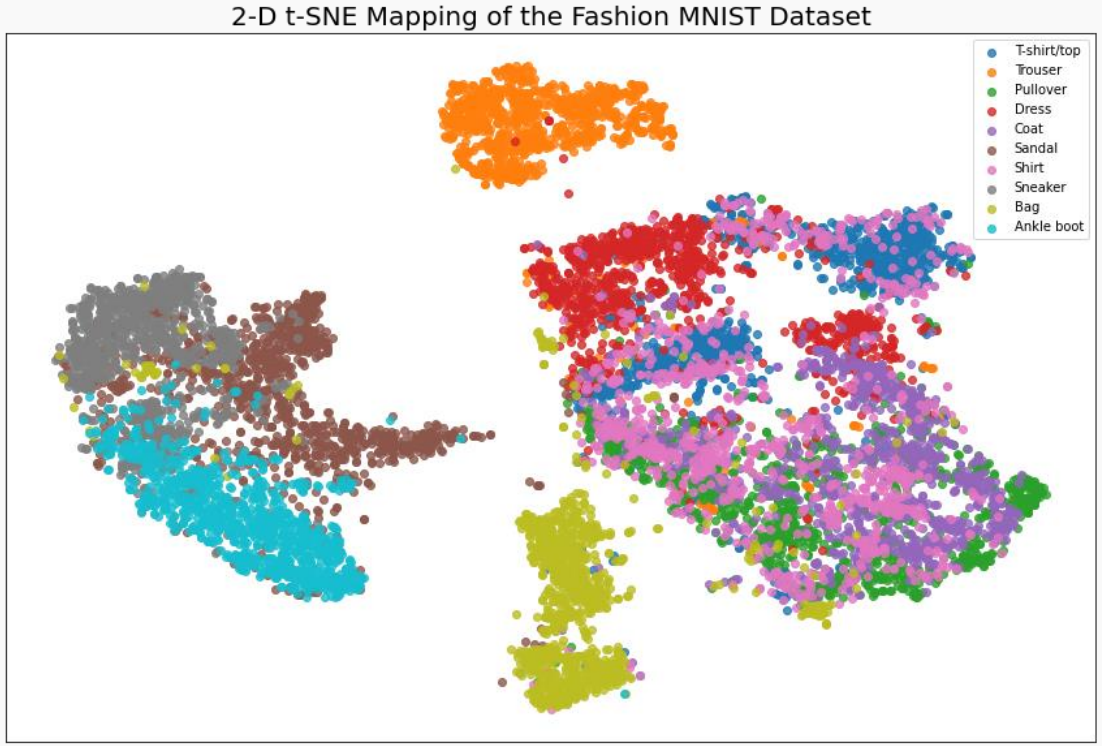
You can think of images of clothes as living on some lower dimensional manifold embedded within the ambient 784-D space.

PCA limits us to finding hyperplanes in the *ambient* linear space. We don’t have any reason to believe that the actual manifold is a linear subspace. *t*-SNE is more flexible.

Fashion MNIST: PCA vs t-SNE

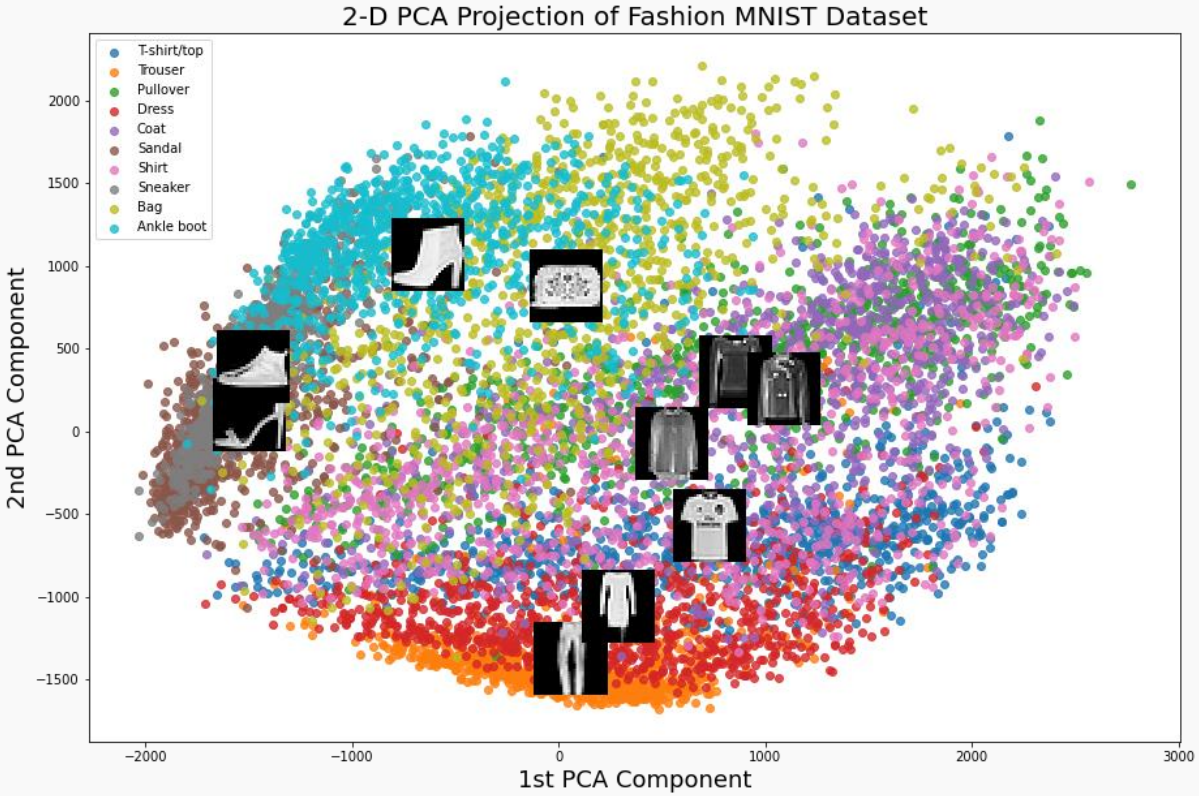
In persevering local structure, t-SNE ends up providing more separation.

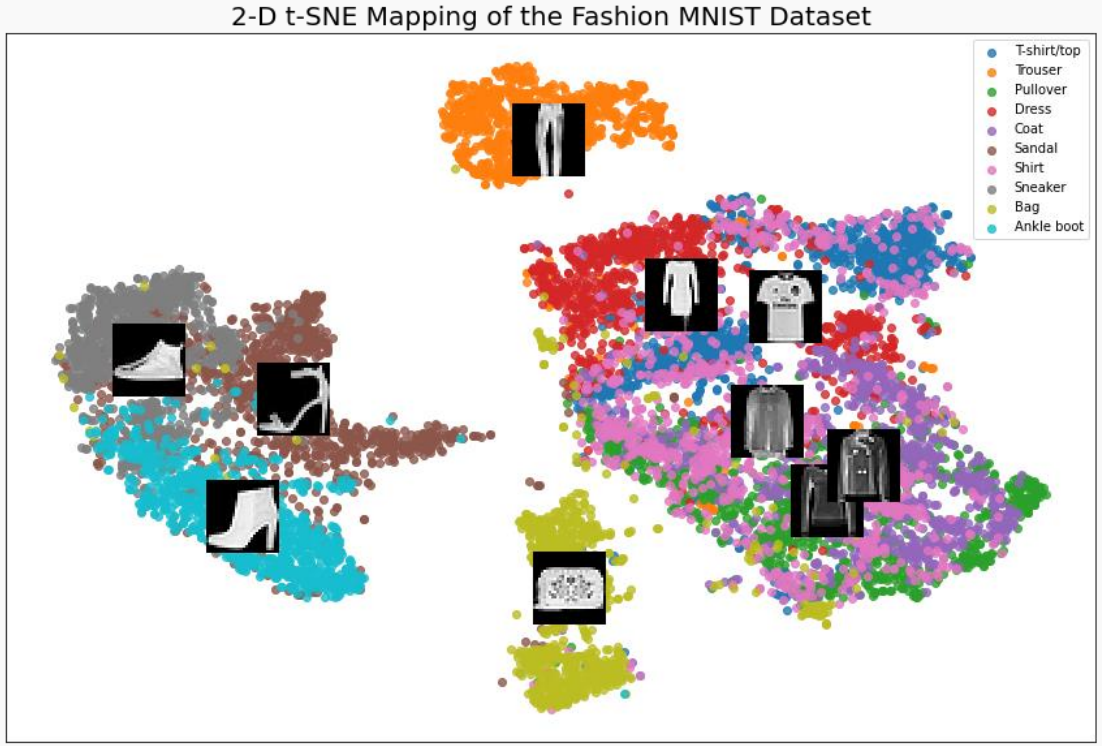




Fashion MNIST: PCA vs t-SNE (with images)

Unique categories, like pants and bags, end up as little islands.





Challenges of t-SNE

- There is randomness in the algorithm so your results will vary (the 'S' is for 'stochastic').
- It is an iterative algorithm and can be very slow. Some preliminary dimensionality reduction is often used to speed things up. You can use PCA for this!
- It has hyperparameters to be tuned: number of iterations, learning rate, number of neighboring points to consider ('perplexity'), etc.
- Doesn't have anything like PCA's components and so is hard to interpret. This makes it primarily a tool for visualization.