CS4618: Artificial Intelligence I

Dimensionality Reduction

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [116]:

```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [117]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [118]:

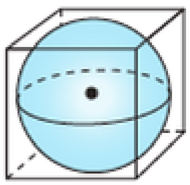
```
from math import pow, pi, sqrt
from numpy.random import rand
from scipy.special import gamma

from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.metrics.pairwise import euclidean_distances

from sklearn.decomposition import PCA

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d

# Class, for use in pipelines, to select certain columns from a DataFrame and co
nvert to a numpy array
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlow, O'Rei
lly, 2017
# Modified by Derek Bridge to allow for casting in the same ways as pandas.DataF
rame.astype
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names, dtype=None):
        self.attribute_names = attribute_names
        self.dtype = dtype
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X_selected = X[self.attribute_names]
        if self.dtype:
            return X_selected.astype(self.dtype).values
        return X_selected.values

# Class to draw 3D arrows
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlow, O'Rei
lly 2017
# Geron credits http://stackoverflow.com/questions/11140163
class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)
```

The Curse of Dimensionality

- In some datasets, examples have thousands or even millions of features
  - E.g. datasets from astronomy
- Is it better or worse to have more features?
  - Storage and processing costs increase
  - Apart from efficiency, intuitively, more features is better
    - E.g. describing houses more completely
  - But, counter-intuitively, that isn't true in general
    - As the number of features grows, algorithms that use distance and density, will find it harder to find good solutions
- To start our thinking about this:
  - Suppose there are two features, each with 10 different values. Then there are $10 \times 10 = 100$ different *possible* examples. Imagine that we've collected a dataset that contains 50 of these examples. Then the density is $50/100 = 0.5$
  - Suppose there are three features, each with 10 different values. That means $10 \times 10 \times 10 = 1000$ different *possible* examples. The density of our dataset (50 examples) is now $50/1000 = 0.05$
  - With four such features, density becomes $50/10000 = 0.005$
  - And so on

  To keep the original level of density would require an exponentially growing dataset
- The problems that arise as the number of features grows have been called **the curse of dimensionality**

The Curse of Dimensionality



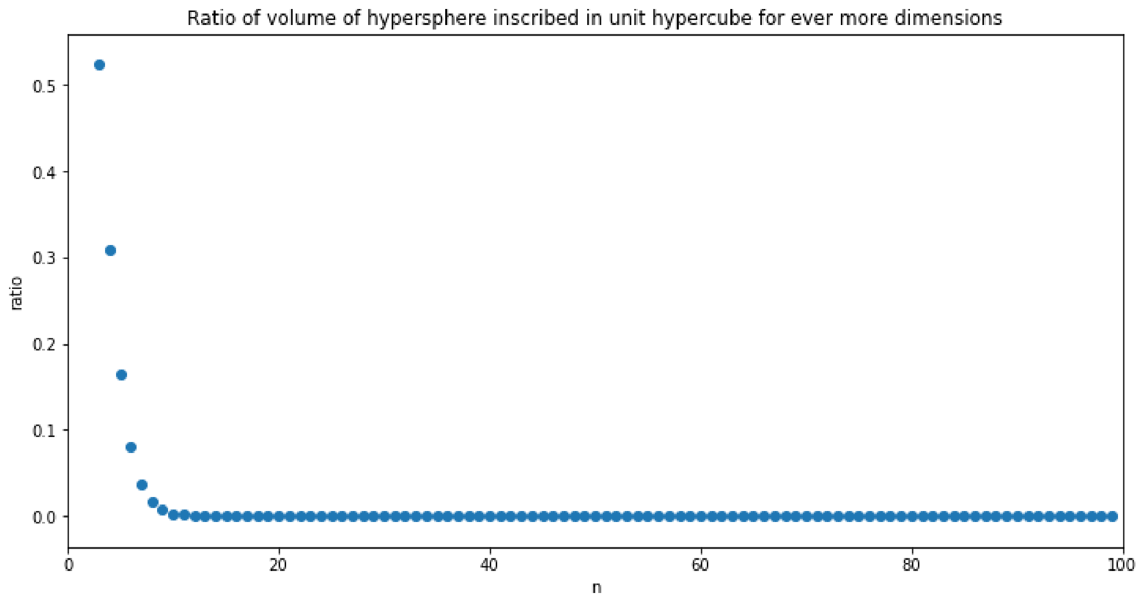
- Consider a sphere 'inscribed' inside a cube (i.e. it touches the sides):
- Suppose the sides of the cube are of length 1 (to simplify calculations)
- What is the ratio of the volume of the sphere to volume of the cube?
  - Volume of the sphere $(4/3)\pi r^3 = 0.52$ (the radius, $r = 0.5$)
  - Volume of the cube $1 \times 1 \times 1 = 1$
  - Ratio $0.52/1 = 0.52$

  So just over half the points fall within the sphere and less than half in the 'corners'
- But now increase the number of dimensions (features): a hypersphere inside a hypercube
  - Volume of hypersphere $\frac{\pi^{\frac{n}{2}}}{\Gamma\left(\frac{n}{2}+1\right)} r^n$
  - Volume of hypercube is still 1
  - As n increases, ratio tends to 0

  So it's as if most of the points are in the corners (relatively few in the sphere)!
- Intuition: as we consider ever more features, everyone becomes an extremist!

In [119]:

```
r = 0.5 # radius
n_range = range(3, 100)

# We don't need to divide the vol of the hypersphere by the vol of the hypercube
 because the latter is always 1
ratios = [pow(pi, n/2) * (r**n) / gamma(n/2 + 1) for n in n_range]

fig =plt.figure(figsize=(12,6))
plt.title("Ratio of volume of hypersphere inscribed in unit hypercube for ever m
ore dimensions")
plt.scatter(n_range, ratios)
plt.xlabel("n")
plt.xlim(0, 100)
plt.ylabel("ratio")
plt.show()
```

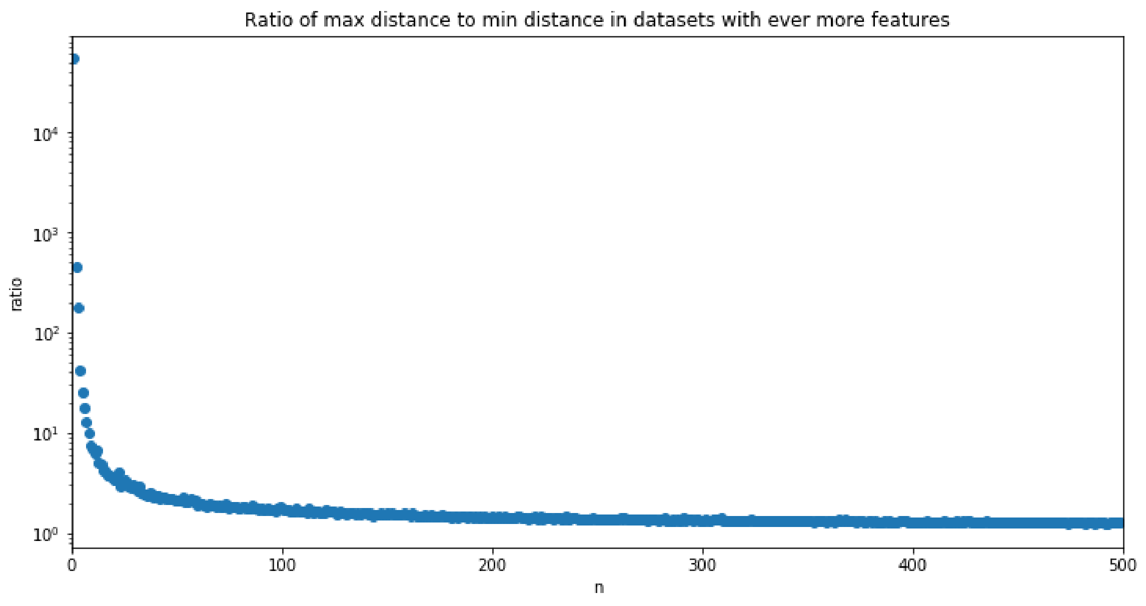


The Curse of Dimensionality

- Here's another problem, this one concerning distances
- The code that follows (which you don't need to study)
  - generates a random dataset where $m = 400$ and $n = 2$ and both features have values in $[0, 1)$
  - computes the Euclidean distance between all pairs of examples
  - finds d_{min}, the smallest of these distances
  - finds d_{max}, the largest of the distances
  - computes the ratio $\frac{d_{max}}{d_{min}}$
- It then does this all again but with $n = 3, 4, 5, \ldots, 500$
- Then it plots the ratios that it has computed (y-axis, but note its scale) against n (x-axis)

In [120]:

```
m = 400
n_range = range(1, 500)

ratios = []
for n in n_range:
    X = rand(m, n)
    dists = euclidean_distances(X)
    non_zero_dists = dists[dists > 0]
    ratios += [np.max(non_zero_dists) / (np.min(non_zero_dists))]

fig = plt.figure(figsize=(12,6))
plt.title("Ratio of max distance to min distance in datasets with ever more feat
ures")
plt.scatter(n_range, ratios)
plt.yscale('log')
plt.xlabel("n")
plt.xlim(0, 500)
plt.ylabel("ratio")
plt.show()
```



- As $n \rightarrow \infty$, $d_{max} \rightarrow d_{min}$, so their rato tends to 1

In [121]:

```
# Since it may not be clear from the graph, we'll show the last 5 of the ratios
 that it calculated
ratios[-5:]
```

Out[121]:

```
[1.2603422534040309,
 1.275390689650084,
 1.2678245843413802,
 1.268209993661171,
 1.2630744863826584]
```

- We conclude (counter-intutively) that examples become equi-distant!
- This obviously undermines methods that depend on finding objects that are similar to each other, as we were doing in the previous lecture — with more features, the most similar object becomes more arbitrary!
- The problem extends to other distance/similarity measures, e.g. cosine similarity

Reducing the Number of Features

- Lots of methods available
- We look at **Principal Components Analysis** (PCA), roughly:
  - Creates new features based on the existing ones (linear combinations of the existing ones), one per existing feature
  - Projects the dataset to a subset of the new features
- A very informal presentation of PCA (without the maths)…

Creating new features

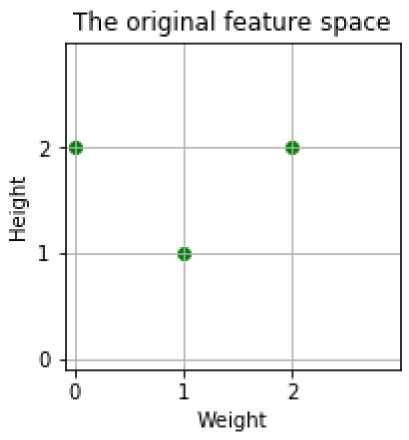
- Suppose a dataset describes objects using just two features, e.g. weight and height
- E.g.

Weight	**Height**
1	1
2	2
0	2

- Since there are only two features, we can plot — a 2D visualization:

In [122]:

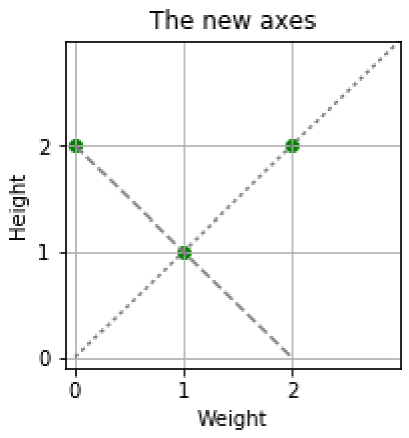
```
fig = plt.figure(figsize=(3,3))
plt.title("The original feature space")
ax = fig.gca()
ax.set_xticks(np.arange(0, 3, 1))
ax.set_yticks(np.arange(0, 3, 1))
plt.scatter([1, 2, 0], [1, 2, 2], color = 'g')
plt.xlabel("Weight")
plt.xlim(-0.1, 3)
plt.ylabel("Height")
plt.ylim(-0.1, 3)
plt.grid()
plt.show()
```



- But the coordinate system we use (the axes) are arbitrary
- We normally use a horizontal and a vertical axis
- But a different pair of axes would work just as well
  - E.g. we could draw one axis diagonally
  - By convention, the other will be **perpendicular** (or 'orthogonal') to the first — in other words, at right angles
  - But even then, we can decide where along the first axis to place it

```
In [123]:
```

```
fig = plt.figure(figsize=(3,3))
plt.title("The new axes")
ax = fig.gca()
ax.set_xticks(np.arange(0, 3, 1))
ax.set_yticks(np.arange(0, 3, 1))
plt.scatter([1, 2, 0], [1, 2, 2], color = 'g')
plt.plot([0, 4], [0, 4], linestyle = 'dotted', color = 'gray')
plt.plot([0, 2], [2, 0], linestyle = 'dashed', color = 'gray')
plt.xlabel("Weight")
plt.xlim(-0.1, 3)
plt.ylabel("Height")
plt.ylim(-0.1, 3)
plt.grid()
plt.show()
```



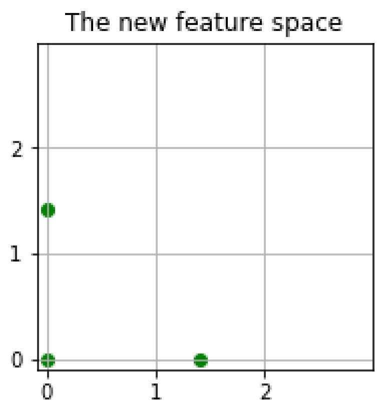
- But on this new 'graph paper', the coordinates of the points are now different
    - The first object used to be at $\langle 1, 1 \rangle$ but is now at $\langle 0, 0 \rangle$
    - The second object used to be at $\langle 2, 2 \rangle$ but is now at $\langle \sqrt{2}, 0 \rangle$

0	0
$\sqrt{2}$	0
0	$\sqrt{2}$

- You can see this if we rotate the graph paper, so that the diagonal is horizontal:

In [124]:

```
fig = plt.figure(figsize=(3,3))
plt.title("The new feature space")
ax = fig.gca()
ax.set_xticks(np.arange(0, 3, 1))
ax.set_yticks(np.arange(0, 3, 1))
plt.scatter([0, sqrt(2), 0], [0, 0, sqrt(2)], color = 'g')
plt.xlim(-0.1, 3)
plt.ylim(-0.1, 3)
plt.grid()
plt.show()
```



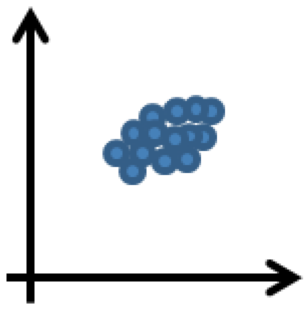
- These new feature values convey exactly the same information as the originals — they're just on a different coordinate system
- But there's an infinite choice of axes
  - For example, with just two features (in 2-dimensions),
    - the first axis can be horizontal (as it was originally), vertical or diagonal with all sorts of different slopes (steeper, shallower, etc.);
      then the usual convention is that the second axis is perpendicular to the first
  - With more features, we have more choices: which to choose next and how to orient it

Creating new features in PCA

- In PCA,
  - The feature we transform next is always the one with the next greatest **variance** (spread)
  - And the orientation of the axis is also based on covering the greatest amount of spread (with the constraint that axes must be perpendicular to one another)
- **Important:** Making these decisions does require that the features be comparable
  - Hence, their values should be *standardized* first
  - (Happily, scikit-learn's PCA class standardizes for us, so we don't even need the StandardScaler in our pipeline)

Class exercise

- The plot contains examples described by two features, which you can assume have been standardized
- Draw onto this plot the new axes that PCA would choose



Example

- Code adapted from A. Géron: *Hands-On Machine Learning with Scikit-Learn & TensorFlow*, O'Reilly, 2017
- (Warning: It generates a random dataset and, for some random datasets, there may be division-by-zero errors. Just run it again!)

In [125]:

```
# Generate dataset

m = 60
w1, w2 = 0.1, 0.3
noise = 0.1

angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * np.random.randn(m)

pca = PCA(n_components = 3)
pca.fit(X)
Xpc = pca.transform(X)

Xpc_inv = pca.inverse_transform(Xpc)
```

In [126]:

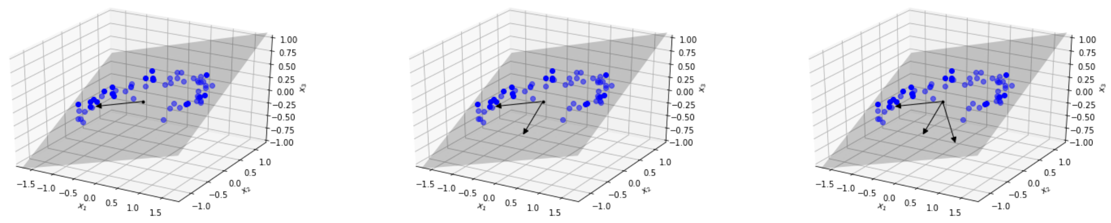
```
# Some variables

axes = [-1.8, 1.8, -1.3, 1.3, -1.0, 1.0]
x1s = np.linspace(axes[0], axes[1], 10)
x2s = np.linspace(axes[2], axes[3], 10)
x1, x2 = np.meshgrid(x1s, x2s)

C = pca.components_
R = C.T.dot(C)
z = (R[0, 2] * x1 + R[1, 2] * x2) / (1 - R[2, 2])

above = X[X[:, 2] > Xpc_inv[:, 2]]
below = X[X[:, 2] <= Xpc_inv[:, 2]]
```

In [127]:

```
def plot(ax):
    ax.plot(below[:, 0], below[:, 1], below[:, 2], "bo", alpha=0.5)
    ax.plot(above[:, 0], above[:, 1], above[:, 2], "bo")
    ax.plot_surface(x1, x2, z, alpha=0.2, color="k")
    ax.set_xlim(axes[0:2])
    ax.set_ylim(axes[2:4])
    ax.set_zlim(axes[4:6])
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    ax.set_zlabel("$x_3$")
```

In [128]:

```
# Plot the dataset showing the three new axes
fig = plt.figure(figsize=(25, 5))

ax1 = fig.add_subplot(131, projection='3d')
ax1.plot([0], [0], [0], "k.")
ax1.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
plot(ax1)

ax2 = fig.add_subplot(132, projection='3d')
ax2.plot([0], [0], [0], "k.")
ax2.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax2.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
plot(ax2)

ax3 = fig.add_subplot(133, projection='3d')
ax3.plot([0], [0], [0], "k.")
ax3.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax3.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax3.add_artist(Arrow3D([0, C[2, 0]],[0, C[2, 1]],[0, C[2, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
plot(ax3)

plt.show()
```



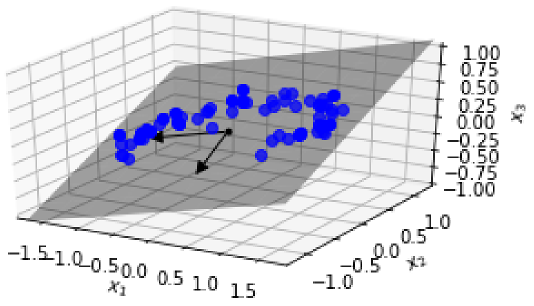
- The unit vector that defines the *i*th axis is called the *i*th **principal component**
- But how does PCA find these axes?
  - By maths!
  - (Look it up, if you're interested!)
- At this stage, we've gained very little: we had n features, we still have n features

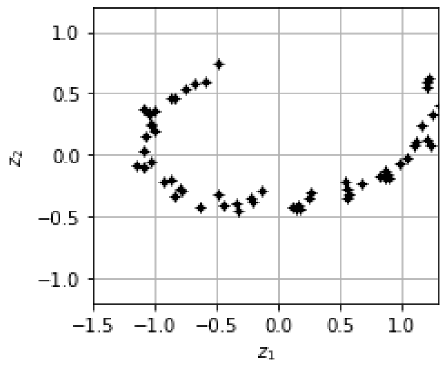
Project the dataset to a subset of the new features

- Examples in real-world datasets often lie close to a lower-dimensional subspace of the high-dimensional space
  - See the example above: the examples (which have $n = 3$ features) lie close to a lower-dimensional subspace, the 2D plane in grey
- So, we can re-express our data ('project it') to just the first k principal components, ignoring the others
  - We get a dataset with fewer features
  - But, hopefully, very little loss of information

In [129]:

```
fig = plt.figure(figsize=(12, 3))

ax1 = fig.add_subplot(121, projection='3d')
plot(ax1)
ax1.plot([0], [0], [0], "k.")
ax1.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax1.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]],
mutation_scale=15, lw=1, arrowstyle="-|>", color="k"))
plot(ax1)
ax1.set_xlim(axes[0:2])
ax1.set_ylim(axes[2:4])
ax1.set_zlim(axes[4:6])

ax2 = fig.add_subplot(122, aspect='equal')
ax2.plot(Xpc[:, 0], Xpc[:, 1], "k+")
ax2.plot(Xpc[:, 0], Xpc[:, 1], "k.")
ax2.set_xlabel("$z_1$")
ax2.set_ylabel("$z_2$")
ax2.axis([-1.5, 1.3, -1.2, 1.2])
ax2.grid(True)
plt.show()
```





PCA in scikit-learn

- Include it in your pipeline
- It automatically does standardization for you

In [130]:

```
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")

# The features we want to select
features = ["flarea", "bdrms", "bthrms"]

# Create the pipeline
pipeline = Pipeline([
        ("selector", DataFrameSelector(features)),
        ("pca", PCA(n_components=2))
    ])

# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [131]:

```
# Let's take a look at a few rows in X
X[:3]
```

Out[131]:

```
array([[  3.68894389e+02,  -3.83770303e+00],
       [ -4.45064940e+01,  -3.02931319e-01],
       [ -3.05968933e+01,   1.05386436e-01]])
```

Explained Variance Ratio

- The **explained variance ratio** of each principal component tells us how much of the dataset's variance lies along the axis of that principal component

In [132]:

```
pipeline.named_steps["pca"].explained_variance_ratio_
```

Out[132]:

```
array([  9.99742385e-01,   1.54567956e-04])
```

How many principle components should we keep?

- If our goal is to **visualize** our dataset, then obviously 2 or 3
- Otherwise, there is no hard-and-fast answer:
  - depends on the dataset and what we are using it for
- One option is to specify the proportion of variance that we wish to preserve
  - In scikit-learn, specify a number in $[0, 1]$ for the `n_components` parameter
  - But, we still have to guess that number!
- See the setting of hyperparameter values in CS4619

In [133]:

```
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")

# The features we want to select
features = ["flarea", "bdrms", "bthrms"]

# Create the pipeline
pipeline = Pipeline([
        ("selector", DataFrameSelector(features)),
        ("pca", PCA(n_components=0.9))
    ])

# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [134]:

```
# Let's take a look at a few rows in X
X[:3]
```

Out[134]:

```
array([[ 368.8943895 ],
       [ -44.50649404],
       [ -30.59689329]])
```

Concluding remarks

- PCA is limited:
  - We use it for numeric-valued features
  - Its new features are always linear combinations of the existing ones: in essence, we're fitting straight-line axes through the values
- There are many techniques that work in a similar way but with more flexibility
  - E.g. techniques for non-numeric data such as Canonical Correspondence Anlaysis (CCA) for nominal-valued data, and Singular Value Decomposition (SVD) (or Non-Negative Matrix Factorization, NMF) for text that uses the bag-of-words representation (next lecture)
  - E.g. Kernel PCA to allow non-linear combinations
- And 'manifold learning' methods, such as Isomap, Locally Linear Embedding, Multidimensional Scaling (MDS), which work somewhat differently, but work on some quite complex datasets

In [ ]:

```

```