Lecture 19: Dimensionality Reduction

Applied Machine Learning

Volodymyr Kuleshov
Cornell Tech

Part 1: What is Dimensionality Reduction?

Dimensionality reduction is another important unsupervised learning problem with many applications.

We will start by defining the problem and providing some examples.

Review: Unsupervised Learning

We have a dataset *without* labels. Our goal is to learn something interesting about the structure of the data:

- Clusters hidden in the dataset.
- Outliers: particularly unusual and/or interesting datapoints.
- Useful signal hidden in noise, e.g. human speech over a noisy phone.

Dimensionality Reduction: Examples

Consider a dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ of motorcylces, characterized by a set of attributes.

- Attributes include size, color, maximum speed, etc.
- Suppose that two attributes are closely correlated: e.g., $x_j^{(i)}$ is the speed in `mph` and $x_k^{(i)}$ is the speed in `km/h`.
- The real dimensionality of the data is $d - 1$!

We would like to automatically identify the right data dimensionality.

Another example can be obtained on the Iris flower dataset.

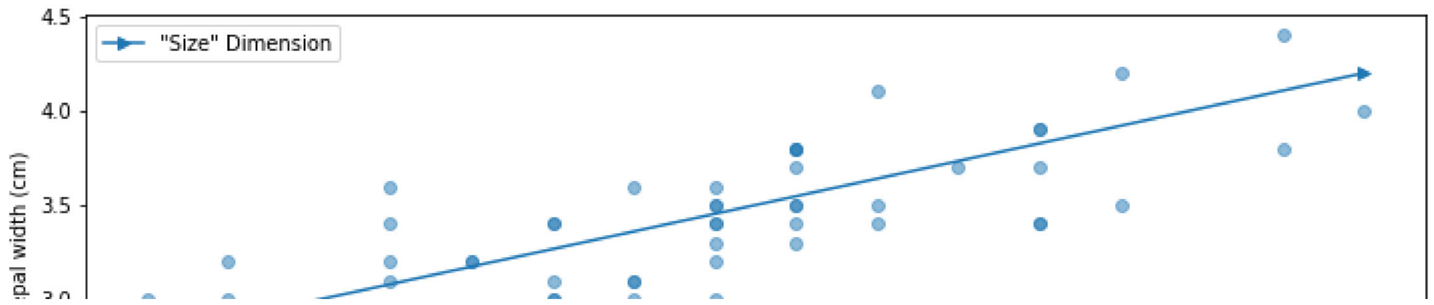
In [3]:
```
# import standard machine learning libraries
import numpy as np
import pandas as pd
from sklearn import datasets

# Load the Iris dataset
iris = datasets.load_iris()
```

Consider the petal length and the petal width of the flowers: they are closely correlated.

This suggests that we may reduce the dimensionality of the problem to one dimension: petal size.

In [4]:
```python
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# Visualize the Iris flower dataset
setosa_flowers = (iris.target == 0)
plt.scatter(iris.data[setosa_flowers,0], iris.data[setosa_flowers,1], alpha=0.5)
plt.plot([4.3, 5.8], [2.8, 4.2], '->')
plt.ylabel("Sepal width (cm)")
plt.xlabel("Sepal length (cm)")
plt.legend(['"Size" Dimension'])
```

Out[4]:
```
<matplotlib.legend.Legend at 0x12bdea4e0>
```



Dimensionality Reduction

More generally, a dimensionality reduction algorithm learns from data an unsupervised model

$$f_\theta : \mathcal{X} \to \mathcal{Z},$$

where $\mathcal{Z}$ is a low-dimensional representation of the data.

For each input $x^{(i)}$, f_θ computes a low-dimensional representation $z^{(i)}$.

Linear Dimensionality Reduction

Suppose $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Z} = \mathbb{R}^p$ for some $p < d$. The transformation

$$f_\theta : \mathcal{X} \to \mathcal{Z}$$

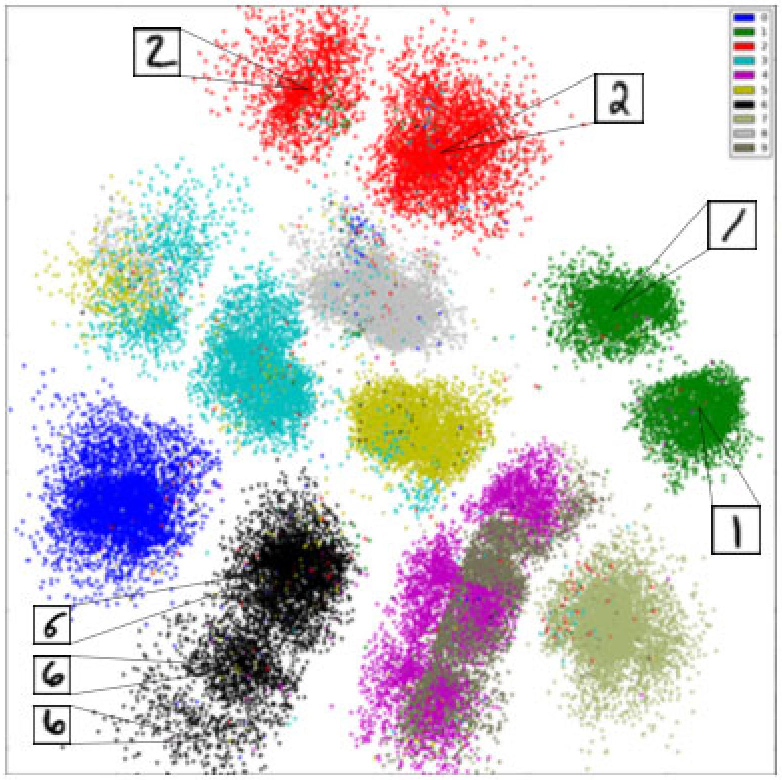
is a linear function with parameters $\theta = W \in \mathbb{R}^{d \times p}$ that is defined by

$$z = f_\theta(x) = W^\top \cdot x.$$

The latent dimension z is obtained from x via a matrix W.

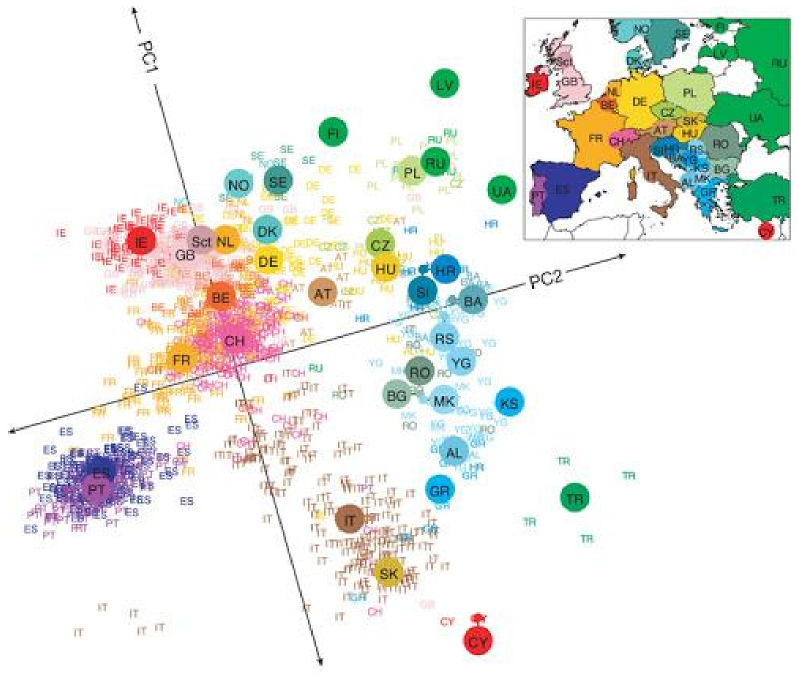
Example: Discovering Structure in Digits

Dimensionality reduction can reveal interesting structure in digits without using labels.



Example: DNA Analysis

Even linear dimensionality reduction is powerful. Here, in uncovers the geography of European countries from only DNA data



Other Kinds of Dimensionality Reduction

We will focus on linear dimensionality reduction this lecture, but there exist many other methods:

- Non-linear methods based on kernels (e.g., Kernel PCA)
- Non-linear methods based on deep learning (e.g., variational autoencoders)
- Non-linear methods based on maximizing signal independence (independent component analysis)
- Probabilistic versions of the above

See the `scikit-learn` guide (https://scikit-learn.org/stable/modules/unsupervised_reduction.html) for more!

Part 2: Principal Component Analysis

We will now describe principal component analysis (PCA), one of the most widely used algorithms for dimensionality reduction.

Components of an Unsupervised Learning Problem

At a high level, an unsupervised machine learning problem has the following structure:

$$\underbrace{\text{Dataset}}_{\text{Attributes}} + \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer}} \to \text{Unsupervised Model}$$

The dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ does not include any labels.

Review: Linear Dimensionality Reduction

Suppose $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Z} = \mathbb{R}^p$ for some $p < d$. The transformation

$$f_\theta : \mathcal{X} \to \mathcal{Z}$$

is a linear function with parameters $\theta = W \in \mathbb{R}^{d \times p}$ that is defined by

$$z = f_\theta(x) = W^\top x.$$

The latent dimension z is obtained from x via a matrix W.

Principal Components Model

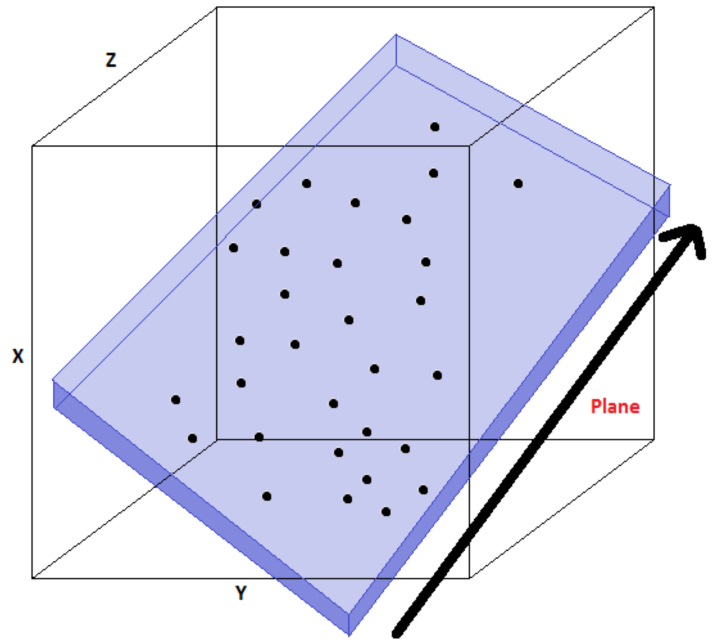
Principal component analysis (PCA) assumes that

- Datapoints $x \in \mathbb{R}^d$ live close to a low-dimensional subspace $\mathcal{Z} = \mathbb{R}^p$ of dimension $p < d$
- The subspace $\mathcal{Z} = \mathbb{R}^p$ is spanned by a set of orthonormal vectors $w^{(1)}, w^{(2)}, \ldots, w^{(p)}$
- The data x are approximated by a linear combination $\tilde{x}$ of the $w^{(k)}$

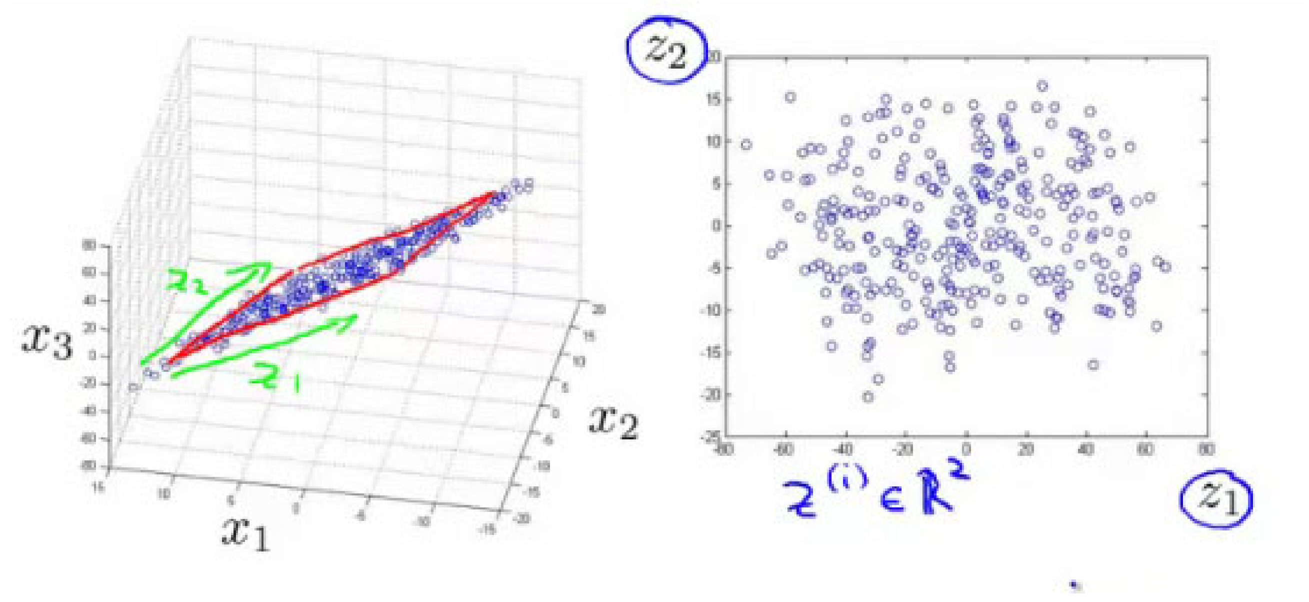
$$x \approx \tilde{x} = \sum_{k=1}^{p} w^{(k)} z_k = W z$$

for some $z \in \mathcal{X}$ that are the coordinates of $\tilde{x}$ in the basis W.

In this example, the data lives in a lower-dimensional 2D plane within a 3D space (image credit (https://doc.plob.org/machine_learning/14_Dimensionality_Reduction.html)).



We can choose a basis W for this plane. The coordinates in this basis are denoted by z (image credit (https://doc.plob.org/machine_learning/14_Dimensionality_Reduction.html)).



The model for PCA is a function f_θ of the form

$$z = f_\theta(x) = W^\top x,$$

where $\theta = W$ and W is a $d \times p$ matrix of p orthonormal column vectors denoted as $w^{(1)}, w^{(2)}, \ldots, w^{(p)}$.

This model enables performing two tasks:

- **Encoding**: $z = W^\top x$, finding the low-dimensional representation of input x
- **Decoding**: $\tilde{x} = Wz$, converting a low-dimensional z to a high-dimensional representation x

PCA Objective: Reconstruction

How do we find a good subpace $\mathcal{Z}$ as defined by a set of orthonormal vectors W?

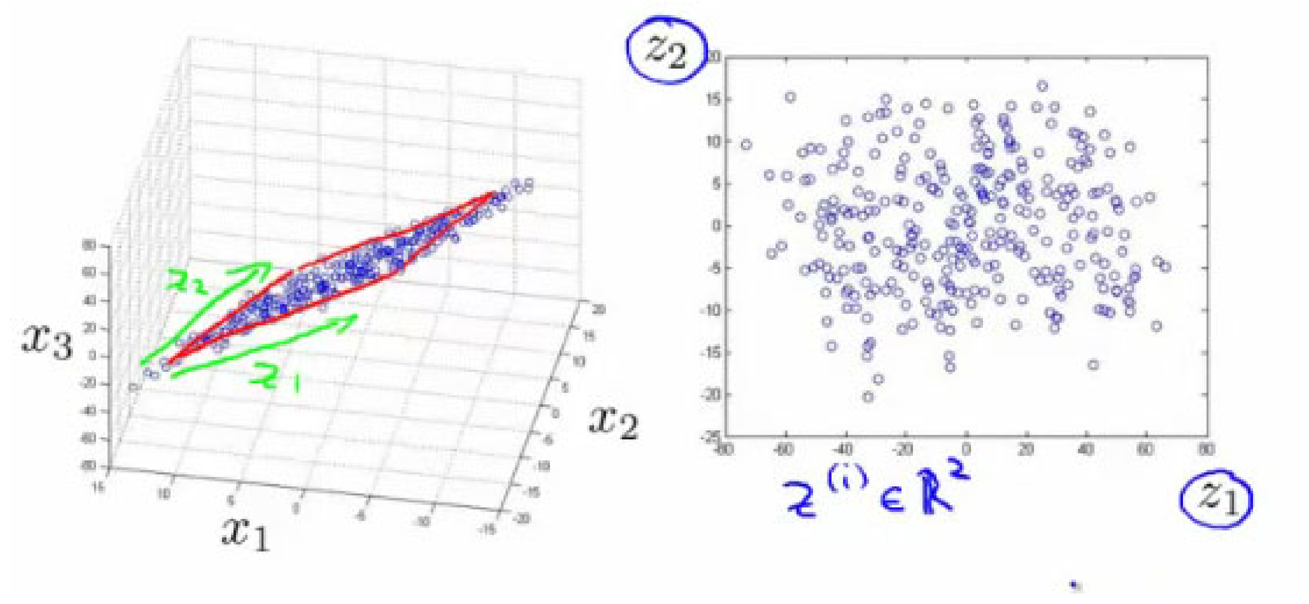
A natural objective is to minimize the reconstruction error

$$J_1(W) = \sum_{i=1}^{n} \|x^{(i)} - \tilde{x}^{(i)}\|_2^2 = \sum_{i=1}^{n} \|x^{(i)} - WW^\top x^{(i)}\|_2^2$$

between each input $x^{(i)}$ and its approximate reconstruction

$$\tilde{x}^{(i)} = W \cdot z^{(i)} = W \cdot W^\top \cdot x^{(i)}.$$

In this example, if the points don't lie perfectly on a plane, we choose the plane such that the points' distance to it is minimized (image [credit (https://doc.plob.org/machine_learning/14_Dimensionality_Reduction.html)](https://doc.plob.org/machine_learning/14_Dimensionality_Reduction.html)).



PCA Objective: Maximizing Variance

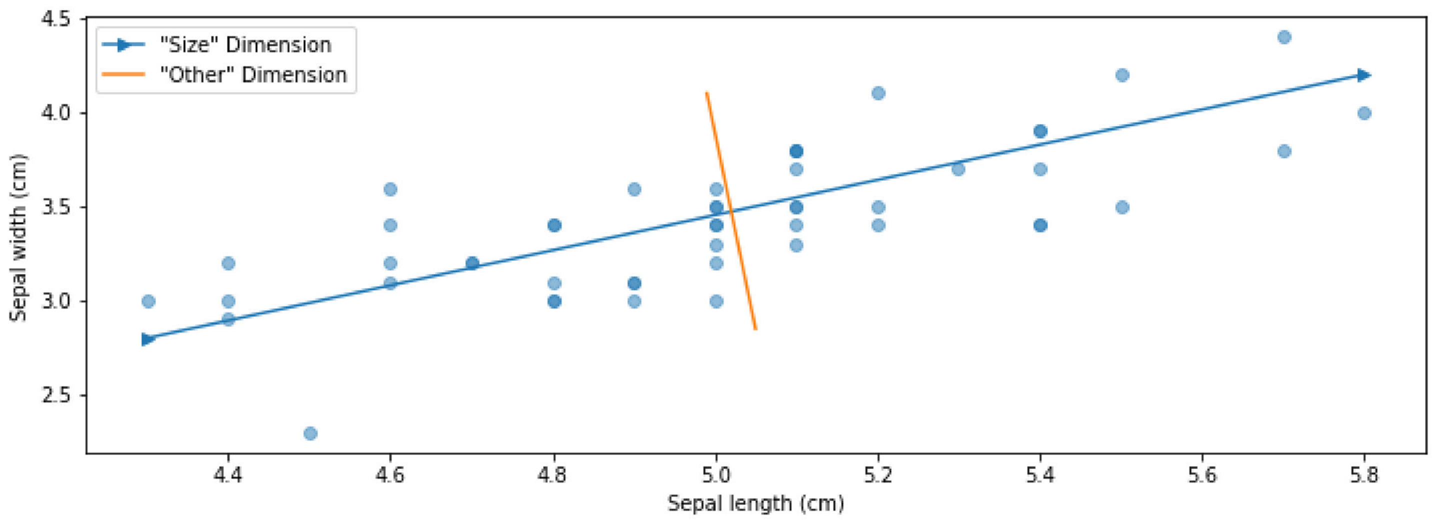
An alternative objective for learning a PCA model is maximizing variance.

We start with some intuition. Consider the Iris flower we have seen earlier.

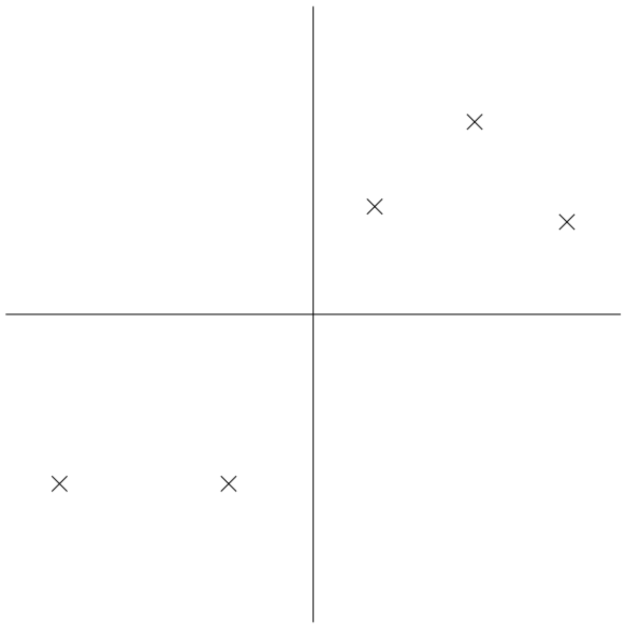
Below, we can project the data along the blue line or the orange line.

The blue line is better because it captures the shape of the data and can be naturally interpreted as "sepal size".

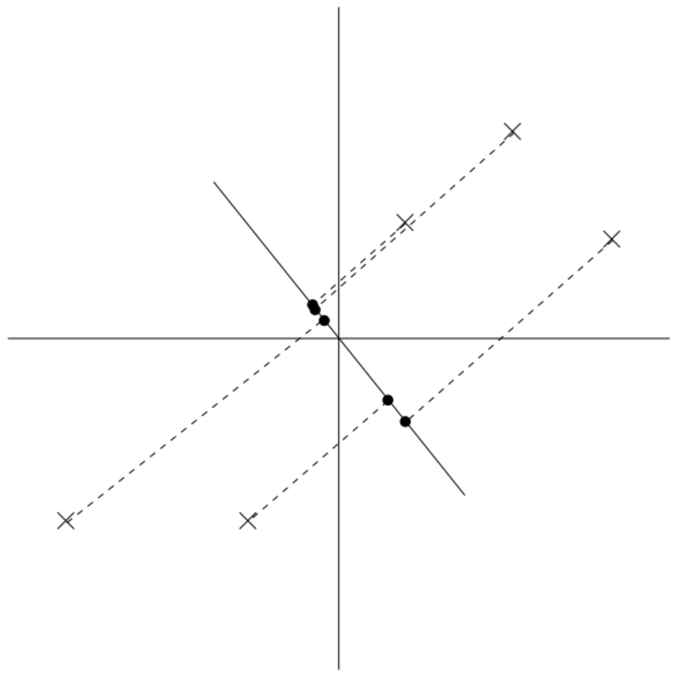
```
In [5]: from matplotlib import pyplot as plt
        plt.rcParams['figure.figsize'] = [12, 4]

        # Visualize the Iris flower dataset
        setosa_flowers = (iris.target == 0)
        plt.scatter(iris.data[setosa_flowers,0], iris.data[setosa_flowers,1], alpha=0.5)
        plt.plot([4.3, 5.8], [2.8, 4.2], '->')
        plt.plot([5.05, 4.99], [2.85, 4.1])
        plt.ylabel("Sepal width (cm)")
        plt.xlabel("Sepal length (cm)")
        plt.legend(['"Size" Dimension', '"Other" Dimension'])

Out[5]: <matplotlib.legend.Legend at 0x12c073d68>
```



How do we automatically identify such natural directions of variation in the data? Consider the following dataset (image by [Andrew Ng (http://cs229.stanford.edu/)](http://cs229.stanford.edu/)).

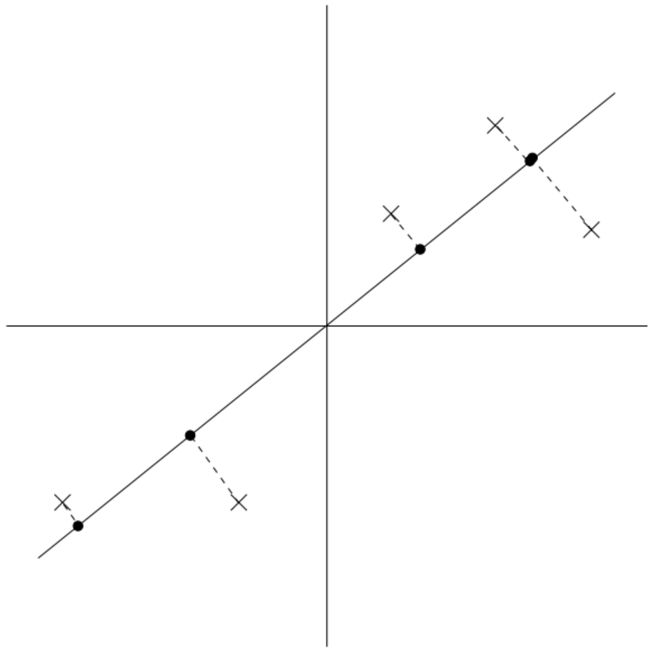


One way to reduce the dimensionality of this dataset from is to project it along the following line.



Projected data is tightly clustered around its mean. It has *low variance.*

An alternative projection is along the following line. Data is much more spread out: it has *high variance* around its mean.



Our goal is to identify this direction automatically.

We may formalize this as follows.

- Let $\hat{\mathbb{E}}[f(x)]$ denote empirical expectation for any f:
$$\hat{\mathbb{E}}[f(x)] = \frac{1}{n}\sum_{i=1}^{n} f(x^{(i)}).$$

- Assume that we have centered the data, i.e. $\hat{\mathbb{E}}[x] = 0 \text{ and thus } \hat{\mathbb{E}}[W^\top x] = W^\top \hat{\mathbb{E}}[x] = 0.$

- The the variance of the projected data is $\hat{\mathbb{E}}\left[\|z - \hat{\mathbb{E}}[z]\|^2\right] = \hat{\mathbb{E}}\left[\|W^\top x - \hat{\mathbb{E}}[W^\top x]\|^2\right] = \hat{\mathbb{E}}\left[\|W^\top x\|^2\right]$

Thus, the variance objective is simply

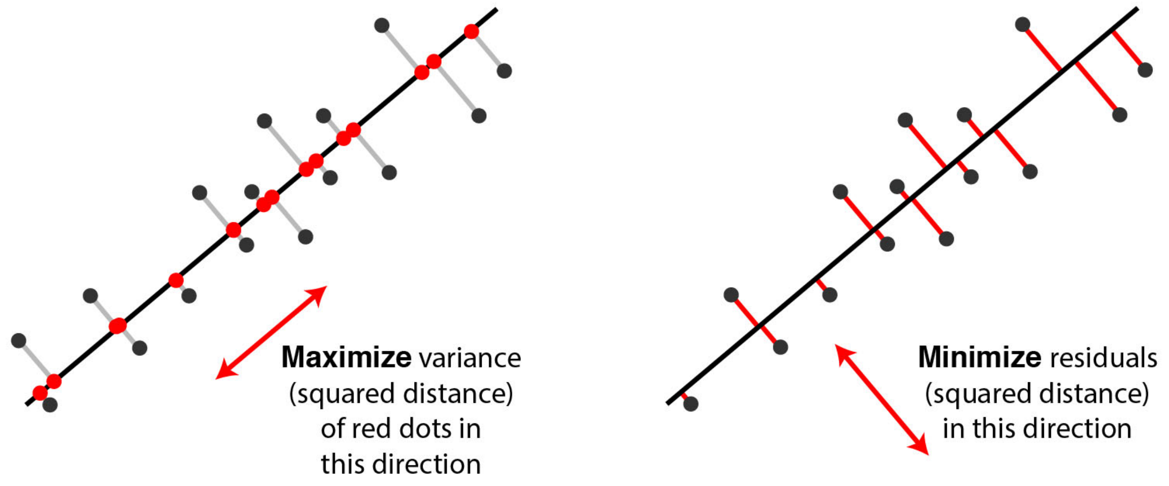
$$J_2(W) = \hat{\mathbb{E}}\left[\|W^\top x\|^2\right] = \frac{1}{n}\sum_{i=1}^{n}\|W^\top x^{(i)}\|_2^2.$$

Equivalence Between PCA Objectives

It turns out that minimizing reconstruction error and maximizing variance are equivalent.

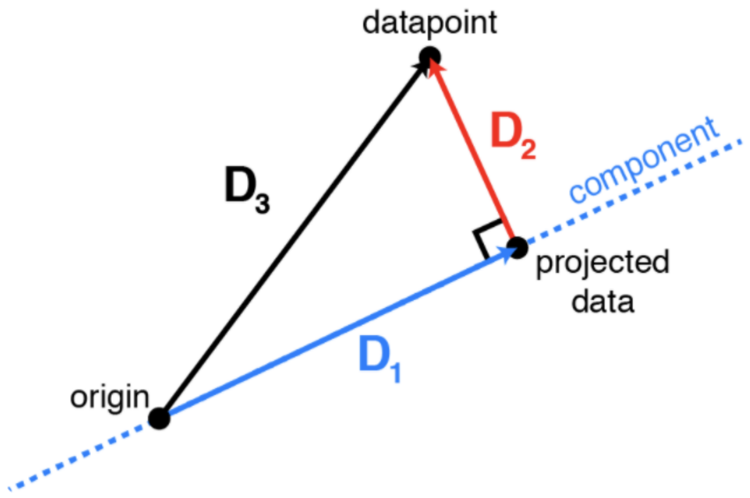
$$\arg\min_{W} J_1(W) = \arg\max_{W} J_2(W).$$

This image by [Alex Williams (http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/)](http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/) provides intuition.


Maximize variance (squared distance) of red dots in this direction
Minimize residuals (squared distance) in this direction

Consider the operator $WW^\top x$. We can decompose any x into a sum of two orthoginal vectors:

$$\begin{aligned} x &= x + WW^\top x - WW^\top x \\ &= \underbrace{WW^\top x}_{\text{projected data } \tilde{x} \text{ (D1)}} + \underbrace{(I - WW^\top)x}_{\text{difference between datapoint } x \text{ and } \tilde{x} \text{ (D2)}} \end{aligned}$$



We can compute the norm of both sides to obtain

$$\begin{aligned}\|x\|_2^2 &= \|WW^\top x + (I - WW^\top)x\|_2^2 \\ &= \|WW^\top x\|_2^2 + \|(I - WW^\top)x\|_2^2 \\ &= \|W^\top x\|_2^2 + \|(I - WW^\top)x\|_2^2\end{aligned}$$

- In the second line we used the fact that $WW^\top x$ and $(I - WW^\top)x$ are orthogonal (easy to check)
- In the third line we used that $\|Wa\| = \|a\|$ for any vector a and orthogonal matrix W.

Thus we find that

$$\begin{aligned} J_1(W) &= \sum_{i=1}^{n} \|(I - WW^\top)x^{(i)}\|_2^2 \\ &= \sum_{i=1}^{n} \left(\|x^{(i)}\|_2^2 - \|W^\top x^{(i)}\|_2^2 \right) \\ &= -n \cdot J_2(W) + \text{const.} \end{aligned}$$

and minimizing the reconstruction objective J_1 is the same as maximizing the variance objective J_2.

Finding Principal Components

Next, how do we optmimize either of these objectives? Let's look at the variance objective J_2, which we can write as:

$$J_2(W) = \frac{1}{n}\sum_{i=1}^{n} \|W^\top x^{(i)}\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{p}((w^{(j)})^\top x^{(i)})^2$$

where $w^{(j)}$ is the j-th column of W and $\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}\left(x^{(i)}(x^{(i)})^\top\right)$ is the empirical covariance matrix of the data.

We can further write this as:

$$\begin{aligned} J_2(W) &= \frac{1}{n}\sum_{i=1}^{n}\|W^\top x^{(i)}\|_2^2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{p}((w^{(j)})^\top x^{(i)})^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{p}\left((w^{(j)})^\top x^{(i)}\right)\cdot\left((x^{(i)})^\top w^{(j)}\right) \\ &= \sum_{j=1}^{p}(w^{(j)})^\top\cdot\left(\frac{1}{n}\sum_{i=1}^{n}x^{(i)}(x^{(i)})^\top\right)\cdot w^{(j)} \\ &= \sum_{j=1}^{p}(w^{(j)})^\top\cdot\hat{\Sigma}\cdot w^{(j)}, \end{aligned}$$

where $\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}\left(x^{(i)}(x^{(i)})^\top\right)$ is the empirical covariance matrix of $\mathcal{D}$.

Recall that the positive semidefinite matrix $\hat{\Sigma}$ has an *eigendecomposition*

$$\hat{\Sigma} = Q \Lambda Q^\top = \sum_{j=1}^{d} \lambda_j q^{(j)} (q^{(j)})^\top .$$

- Q is a matrix whose columns are orthonormal eigenvectors $q^{(j)}$ for $j = 1, 2, \ldots, d$.
- Λ is a diagonal matrix of positive eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$.

Consider our optimization problem for $p = 1$:

$$J(w) = w^\top \cdot \hat{\Sigma} \cdot w.$$

How do we find the best projection vector w?

Using the eigendecomposition, we can write this as:

$$J(w) = w^\top \cdot Q\Lambda Q^\top \cdot w = \sum_{j=1}^{d} \lambda_j (w^\top q^{(j)})^2.$$

The optimal solution to

$$\max_w J(w) = \max_w \sum_{j=1}^{d} \lambda_j (w^\top q^{(j)})^2$$

is attained by the top eigenvector $w = q^{(1)}$. The optimum is $J(q^{(1)}) = \lambda_1$.

- Let $a_j = (w^\top q^{(j)})^2$ and note that $\sum_j a_j^2 = ||Qw||_2^2 = 1$ because all vectors are orthonormal.
- Our objective $J(w) = \sum_{j=1}^{d} \lambda_j a_j^2$ is a weighted average of λ_j
- The weighted average $\sum_{j=1}^{d} \lambda_j a_j^2$ attains a maximum of λ_1 when all "weight" goes to $a_1 = 1$ and $w = q^{(1)}$.

More generally when $p > 1$, our objective is

$$J(W) = \sum_{k=1}^{p} \sum_{j=1}^{d} \lambda_j ((w^{(k)})^\top q^{(j)})^2$$

where W is a matrix of orthonormal columns $w^{(1)}, w^{(2)}, \ldots, w^{(p)}$.

By analogy with the previous example,

- $J(W)$ is maximized when $w^{(1)} = q^{(1)}, w^{(2)} = q^{(2)}, ..., w^{(p)} = q^{(p)}$
- The maximum value attained is $\lambda_1 + \lambda_2 + \ldots + \lambda_p$
- We refer to
  $$\frac{\lambda_1 + \lambda_2 + \ldots + \lambda_p}{\lambda_1 + \lambda_2 + \ldots + \lambda_d}$$
  as the proportion of *variance explained* by the lower-dimensional projection $z = W^\top x$. When $d = p$ it is one.

Algorithm: Principal Component Analysis

- **Type**: Unsupervised learning (dimensionality reduction)
- **Model family**: Linear projection $W^\top z$ of low-dimensional z
- **Objective function**: Reconstruction error or variance maximization
- **Optimizer**: Matrix eigendecomposition

Practical Considerations

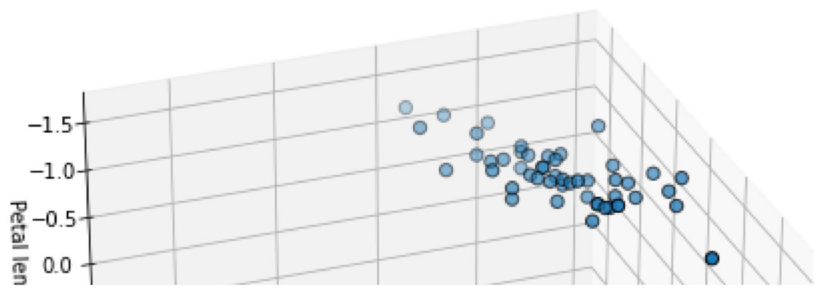
When applying PCA, the following tricks are useful.

- Before applying PCA, it is important to normalize the data to have zero mean and unit variance.
$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \text{for all } i, j,$$
where μ_j, σ_j are the mean and variance along the j-th dimension.
- This address scaling issues due to choice of units (`km/h` vs `cm/h`).
- In order to choose the optimal number of components, we can apply the Elbow method.

An Example: Iris Flowers

Let's look at an example over the Iris flower dataset. In its entirety, it has four dimensions; let's visualize it in 3D by looking at the first 3 dimensions.

In [46]:
```
from mpl_toolkits.mplot3d import Axes3D

# form the design matrix and target vector
X, y = iris.data, iris.target
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# display data in 3D
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
ax.set_xlabel("Sepal length")
ax.set_ylabel("Sepal width")
ax.set_zlabel("Petal length")
p1 = ax.scatter(X[:, 0], X[:, 1], X[:, 2], edgecolor='k', s=40)
```

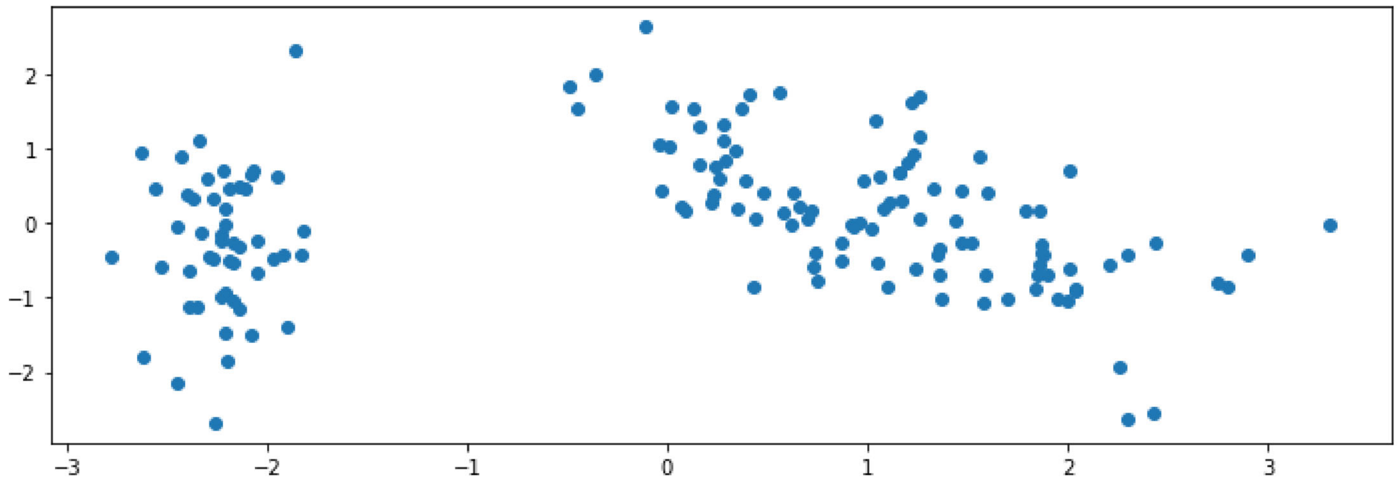


We can implement PCA using a small number of `numpy` operations.

In [36]:
```python
def pca_project(X, p=2):
    Sigma = X.T.dot(X) / X.shape[0] # form covariance matrix
    L, Q = np.linalg.eig(Sigma) # perform eigendecomposition
    W = Q[:,:p] # get top p eigenvectors
    Z = X.dot(W) # project on these eigenvectors
    return Z
```

Visualizing the data, we obtain the following structure.

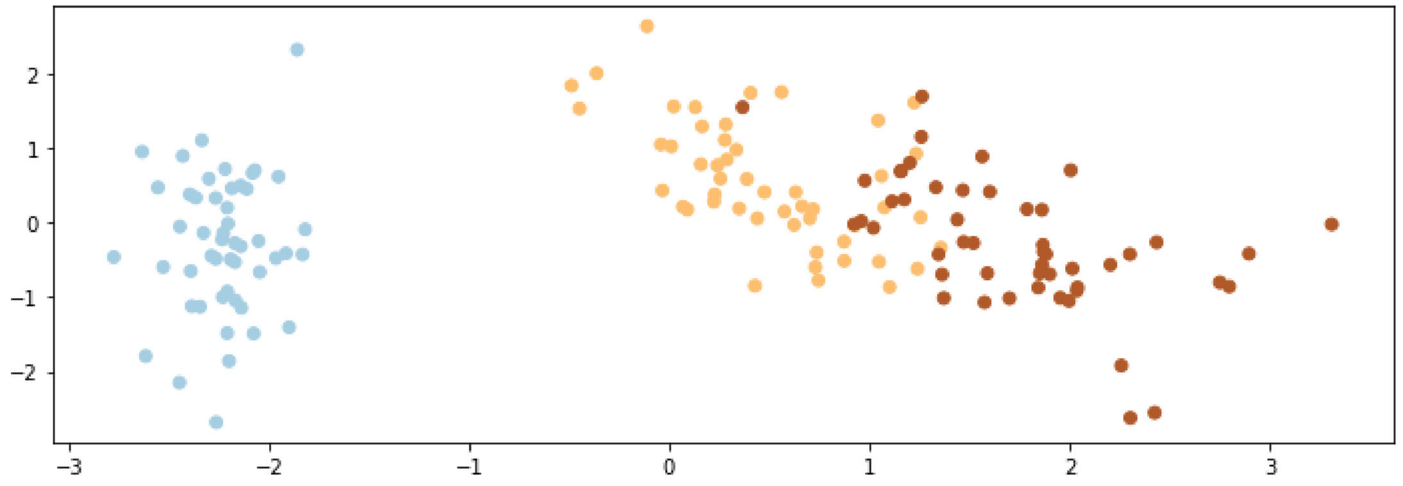
```
In [41]: Z = pca_project(X, p=2)
         plt.scatter(Z[:,0], Z[:,1])

Out[41]: <matplotlib.collections.PathCollection at 0x12d072828>
```



We can also add labels. The classes are well-separated.

In [42]:
```
plt.scatter(Z[:,0], Z[:,1], c=y, cmap=plt.cm.Paired)
```

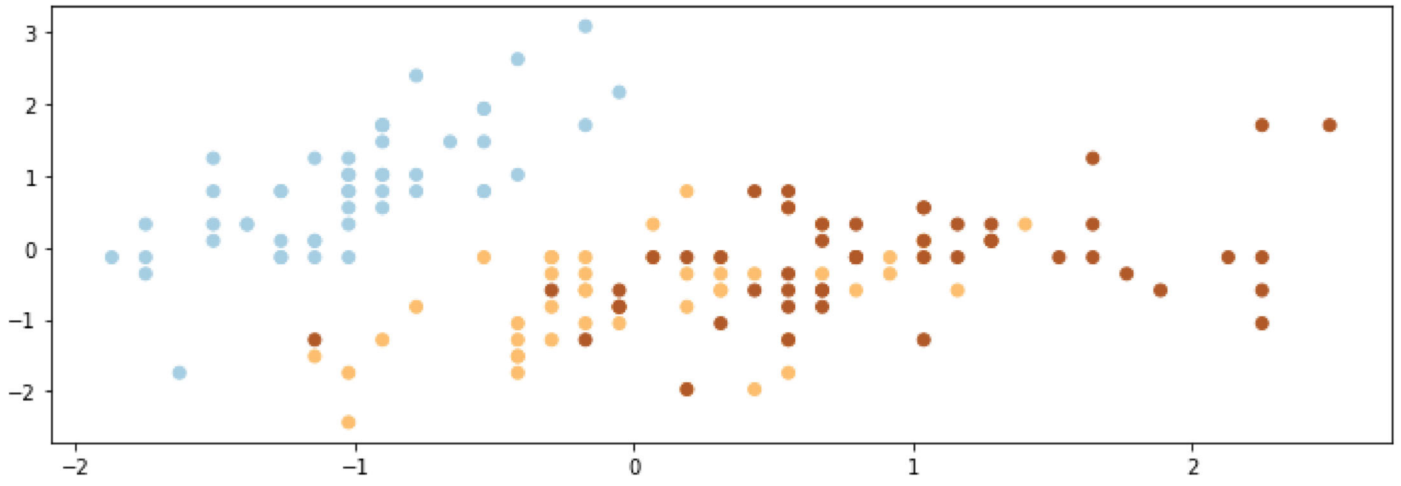
Out[42]:
```
<matplotlib.collections.PathCollection at 0x12cbc9f60>
```



The separation is better than if we just chose the first two dimensions.

```
In [43]: plt.scatter(X[:,0], X[:,1], c=y, cmap=plt.cm.Paired)
```

```
Out[43]: <matplotlib.collections.PathCollection at 0x12c8c2cf8>
```



We can train two classifiers on this data and compare their accuracy.

PCA dimensions result in better accuracy that just choosing the first two dimensions.

In [50]:
```
from sklearn.linear_model import LogisticRegression

# train softmax on non-PCA data
logreg1 = LogisticRegression(C=1e5, multi_class='multinomial')
logreg1.fit(X[:,:2], y)
print('Accuracy on first two dimensions: %.2f' % logreg1.score(X[:,:2],y))

# train softmax on PCA data
logreg2 = LogisticRegression(C=1e5, multi_class='multinomial')
logreg2.fit(Z, y)
print('Accuracy on two PCA dimensions: %.2f' % logreg2.score(Z,y))
```

```
Accuracy on first two dimensions: 0.83
Accuracy on two PCA dimensions: 0.92
```

Pros and Cons of PCA

PCA is perhaps the most widely used dimensionality reduction algorithm.

- It is both highly intuitive and effective
- It is also fast and easy to implement

Its limitations include:

- Linear projections may be too limited in some applications
- Choosing the right dimension p can be somewhat of an art

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Review: Learning GMMs

Gaussian mixtures are latent variable models, and we can learn them using maximum marginal log-likelihood:

$$\max_\theta \sum_{x\in\mathcal{D}} \log P_\theta(x) = \max_\theta \sum_{x\in\mathcal{D}} \log \left(\sum_{z\in\mathcal{Z}} P_\theta(x,z) \right)$$

- Unlike in GMMs for supervised learning, cluster assignments are latent.
- Hence, there is not a closed form solution for θ.
- We will see specialized algorithm for this task.

Expectation Maximization: Intuition

Expecation maximization (EM) is an algorithm for maximizing marginal log-likelihood

$$\max_\theta \sum_{x^{(i)} \in \mathcal{D}} \log \left(\sum_{z \in \mathcal{Z}} P_\theta(x^{(i)}, z) \right)$$

that can also be used to learn Gaussian mixtures.

We want to optimize the marginal log-likelihood

$$\max_\theta \sum_{x^{(i)} \in \mathcal{D}} \log \left(\sum_{z \in \mathcal{Z}} P_\theta(x^{(i)}, z) \right).$$

- If we know the true $z^{(i)}$ for each $x^{(i)}$, we maximize

  $$\max_\theta \sum_{x^{(i)}, z^{(i)} \in \mathcal{D}} \log \Big(P_\theta(x^{(i)}, z^{(i)}) \Big).$$

  and it's easy to find the best θ (use solution for supervised learning).
- If we know θ, we can estimate the cluster assignments $z^{(i)}$ for each i by computing $P_\theta(z|x^{(i)})$.

Expectation maximization alternates between these two steps.

1. (**E-Step**) Given an estimate θ_t of the weights, compute $P_\theta(z|x^{(i)})$. and use it to "hallucinate" expected cluster assignments $z^{(i)}$.
2. (**M-Step**) Find a new θ_{t+1} that maximizes the marginal log-likelihood by optimizing $P_\theta(x^{(i)}, z^{(i)})$ given the $z^{(i)}$ from step 1.

This process increases the marginal likelihood at each step and eventually converges.

Expectation Maximization: Definition

Formally, EM learns the parameters θ of a latent-variable model $P_\theta(x, z)$ over a dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ as follows.

For $t = 0, 1, 2, \ldots$, repeat until convergence:

1. (**E-Step**) For each $x^{(i)} \in \mathcal{D}$ compute $P_{\theta_t}(z|x^{(i)})$
2. (**M-Step**) Compute new weights θ_{t+1} as

$$\theta_{t+1} = \arg\max_\theta \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim P_{\theta_t}(z|x^{(i)})} \log P_\theta(x^{(i)}, z^{(i)})$$

Since assignments $P_{\theta_t}(z|x^{(i)})$ are "soft", M-step involves an expectation.

Understanding the E-Step

Intuitively, we hallucinate $z^{(i)}$ in the E-Step.

In practice, the $P_{\theta_t}(z|x^{(i)})$ define "soft" assignments, and we compute a vector of class probabilities for each $x^{(i)}$. <!-- * The $P_{\theta_t}(z|x^{(i)})$ define "soft" assignments, and we compute a vector of class probabilities for each $x^{(i)}$.

- We compute an expected values over $z^{(i)}$ instead of hallucinating one value. -->

Understanding the M-Step

Since class assignments from E-step are probabilistic, we maximize an expectation:

$$\begin{aligned}\theta_{t+1} &= \arg\max_{\theta} \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim P_{\theta_t}(z|x^{(i)})} \log P_\theta(x^{(i)}, z^{(i)}) \\ &= \arg\max_{\theta} \sum_{i=1}^{n} \sum_{k=1}^{K} P_{\theta_t}(z = k|x^{(i)}) \log P_\theta(x^{(i)}, z = k)\end{aligned}$$

For many interesting models, this is tractable.

Pros and Cons of EM

EM is a very important optimization algorithm in machine learning.

- It is easy to implement and is guaranteed to converge.
- It works in a lot of imporant ML models.

Its limitations include:

- It can get stuck in local optima.
- We may not be able to compute $P_{\theta_t}(z|x^{(i)})$ in every model.

Part 3: Expectation Maximization in Gaussian Mixture Models

Next, let's work through how Expectation Maximization works in Gaussian Mixture Models.

Review: Gaussian Mixture Models

Gaussian mixtures are latent-variable probabilistic models that are useful for clustering. They define a model

$$P_\theta(x, z) = P_\theta(x|z)P_\theta(z)$$

- $z \in \{1, 2, \ldots, K\}$ is discrete and follows a categorical distribution $P_\theta(z = k) = \phi_k$.
- $x \in \mathbb{R}$ is continuous; conditioned on $z = k$, it follows a Normal distribution $P_\theta(x|z = k) = \mathcal{N}(\mu_k, \Sigma_k)$.

The parameters θ are the μ_k, Σ_k, ϕ_k for all $k = 1, 2, \ldots, K$.

Review: Expectation Maximization

Formally, EM learns the parameters θ of a latent-variable model $P_\theta(x, z)$ over a dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ as follows.

For $t = 0, 1, 2, \ldots$, repeat until convergence:

1. (**E-Step**) For each $x^{(i)} \in \mathcal{D}$ compute $P_{\theta_t}(z|x^{(i)})$
2. (**M-Step**) Compute new weights θ_{t+1} as

$$\theta_{t+1} = \arg\max_\theta \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim P_{\theta_t}(z|x^{(i)})} \log P_\theta(x^{(i)}, z^{(i)})$$

Since assignments $P_{\theta_t}(z|x^{(i)})$ are "soft", M-step involves an expectation.

Deriving the E-Step

In the E-step, we compute the posterior for each data point x as follows

$$P_\theta(z = k \mid x) = \frac{P_\theta(z = k, x)}{P_\theta(x)} = \frac{P_\theta(x|z = k)P_\theta(z = k)}{\sum_{l=1}^{K} P_\theta(x|z = l)P_\theta(z = l)}$$

$P_\theta(z \mid x)$ defines a vector of probabilities that x originates from component k given the current set of parameters θ

Deriving the M-Step

At the M-step, we optimize the expected log-likelihood of our model.

$$\max_\theta \sum_{x \in D} \mathbb{E}_{z \sim P_{\theta_t}(z|x)} \log P_\theta(x, z) =$$

$$\max_\theta \left(\sum_{k=1}^K \sum_{x \in D} P_{\theta_t}(z_k|x) \log P_\theta(x|z_k) + \sum_{k=1}^K \sum_{x \in D} P_{\theta_t}(z_k|x) \log P_\theta(z_k) \right)$$

As in supervised learning, we can optimize the two terms above separately.

We will start with $P_\theta(x \mid z = k) = \mathcal{N}(x; \mu_k, \Sigma_k)$. We have to find μ_k, Σ_k that optimize

$$\max_\theta \sum_{x^{(i)} \in D} P(z = k | x^{(i)}) \log P_\theta(x^{(i)} | z = k)$$

Note that this corresponds to fitting a Gaussian to a dataset whose elements $x^{(i)}$ each have a weight $P(z = k | x^{(i)})$.

Similarly to how we did this the supervised regime, we compute the derivative, set it to zero, and obtain closed form solutions:

$$\begin{aligned}
\mu_k &= \frac{\sum_{i=1}^n P(z=k|x^{(i)})x^{(i)}}{n_k} \\
\Sigma_k &= \frac{\sum_{i=1}^n P(z=k|x^{(i)})(x^{(i)}-\mu_k)(x^{(i)}-\mu_k)^\top}{n_k} \\
n_k &= \sum_{i=1}^n P(z=k|x^{(i)})
\end{aligned}$$

Intuitively, the optimal mean and covariance are the emprical mean and convaraince of the dataset $\mathcal{D}$ when each element $x^{(i)}$ has a weight $P(z=k|x^{(i)})$.

Similarly, we can show that the class priors are

$$\phi_k = \frac{n_k}{n}$$
$$n_k = \sum_{i=1}^{n} P(z = k | x^{(i)})$$

EM in Gaussian Mixture Models

EM learns the parameters θ of a Gaussian mixture model $P_\theta(x, z)$ over a dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ as follows.

For $t = 0, 1, 2, \ldots$, repeat until convergence:

1. (**E-Step**) For each $x^{(i)} \in \mathcal{D}$ compute $P_{\theta_t}(z|x^{(i)})$
2. (**M-Step**) Compute parameters μ_k, Σ_k, ϕ_k using the above formulas

Part 4: Generalization in Probabilistic Models

Let's now revisit the concepts of overfitting and underfitting in GMMs.

Review: Data Distribution

We will assume that the dataset is sampled from a probability distribution $\mathbb{P}$, which we will call the *data distribution*. We will denote this as

$$x \sim \mathbb{P}.$$

The dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, \ldots, n\}$ consists of *independent and identicaly distributed* (IID) samples from $\mathbb{P}$.

Review: Gaussian Mixture Models

Gaussian mixtures are latent-variable probabilistic models that are useful for clustering. They define a model

$$P_\theta(x, z) = P_\theta(x|z)P_\theta(z)$$

- $z \in \{1, 2, \ldots, K\}$ is discrete and follows a categorical distribution $P_\theta(z = k) = \phi_k$.
- $x \in \mathbb{R}$ is continuous; conditioned on $z = k$, it follows a Normal distribution $P_\theta(x|z = k) = \mathcal{N}(\mu_k, \Sigma_k)$.

The parameters θ are the μ_k, Σ_k, ϕ_k for all $k = 1, 2, \ldots, K$.

Review: Generalization

In machine learning, **generalization** is the property of predictive models to achieve good performance on new, heldout data that is distinct from the training set.

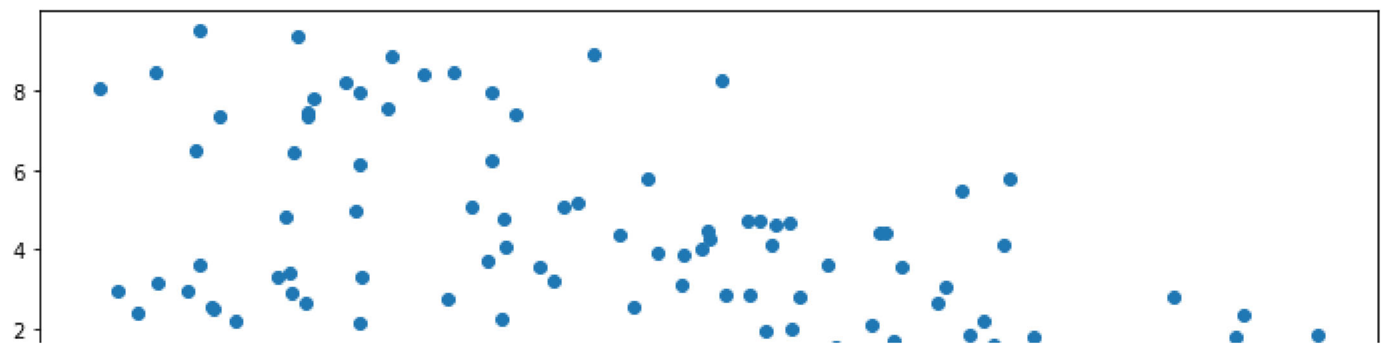
How does generalization apply to probabilistic unsupervised models like GMMs?

An Unsupervised Learning Dataset

Consider the following dataset, consisting of a mixture of Gaussians.

In [49]:
```python
import numpy as np
from sklearn import datasets
from matplotlib import pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# generate 150 random points
np.random.seed(0)
X_all, y_all = datasets.make_blobs(150, centers=4)

# use the first 100 points as the main dataset
X, y = X_all[:100], y_all[:100]
plt.scatter(X[:,0], X[:,1])
```

Out[49]:
```
<matplotlib.collections.PathCollection at 0x12b583780>
```



We know the true labels of these clusers, and we can visualize them.

We will also keep 50 points as a holdout set.

Underfitting in Unsupervised Learning

Underfitting happens when we are not able to fully learn the signal hidden in the data.

In the context of GMMs, this means not capturing all the clusters in the data.

Let's fit a GMM on our toy dataset.

In [52]:
```
# fit a GMM
from sklearn import mixture
model = mixture.GaussianMixture(n_components=2)
model.fit(X)
```

Out[52]:
```
GaussianMixture(n_components=2)
```

The model finds two distinct components in the data, but they fail to capture the true structure.

We can also measure the value of our objective (the log-likelihood) on the training and holdout sets.

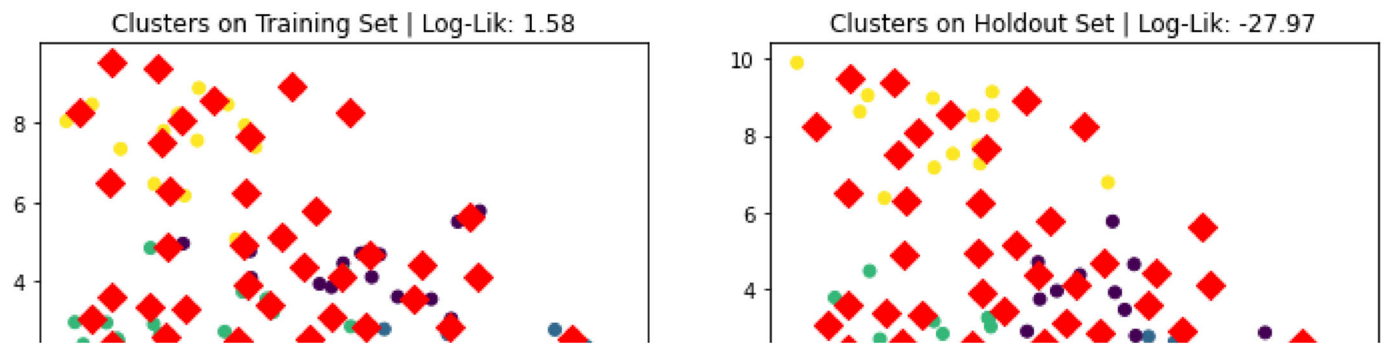
Consider now what happens if we further increase the number of clusters.

Overfitting in Unsupervised Learning

Overfitting happens when we fit the noise, but not the signal.

In our example, this means fitting small, local noise clusters rather than the true global clusters.

```
In [55]: model = mixture.GaussianMixture(n_components=50)
         model.fit(X)

         plt.subplot(121)
         plt.title('Clusters on Training Set | Log-Lik: %.2f' % model.score(X))
         plt.scatter(X[:,0], X[:,1], c=y)
         plt.scatter(model.means_[:,0], model.means_[:,1], marker='D', c='r', s=100)

         plt.subplot(122)
         plt.title('Clusters on Holdout Set | Log-Lik: %.2f' % model.score(X_holdout))
         plt.scatter(X_holdout[:,0], X_holdout[:,1], c=y_holdout)
         plt.scatter(model.means_[:,0], model.means_[:,1], marker='D', c='r', s=100)

Out[55]: <matplotlib.collections.PathCollection at 0x12aeb95c0>
```



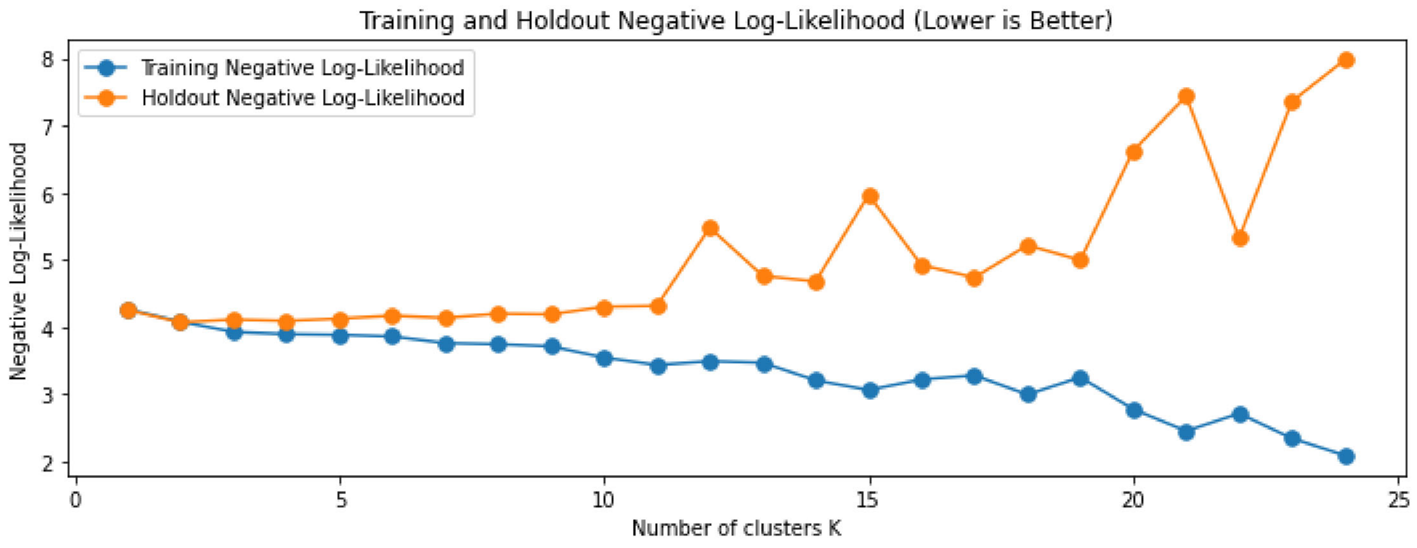
Measuring Generalization Using Log-Likelihood

Probabilistic unsupervised models optimize an objective that can be used to detect overfitting and underfitting by comparing performance between training and holdout sets.

Below, we visualize the performance (measured via negative log-likelihood) on training and holdout sets as K increases.

```
In [56]: Ks, training_objs, holdout_objs = range(1,25), [], []
         for k in Ks:
             model = mixture.GaussianMixture(n_components=k)
             model.fit(X)
             training_objs.append(-model.score(X))
             holdout_objs.append(-model.score(X_holdout))

         plt.plot(Ks, training_objs, '.-', markersize=15)
         plt.plot(Ks, holdout_objs, '.-', markersize=15)
         plt.xlabel("Number of clusters K")
         plt.ylabel("Negative Log-Likelihood")
         plt.title("Training and Holdout Negative Log-Likelihood (Lower is Better)")
         plt.legend(['Training Negative Log-Likelihood', 'Holdout Negative Log-Likelihood'
         ])
```

```
Out[56]: <matplotlib.legend.Legend at 0x12c463320>
```



Warning: This process doesn't work as well as in supervised learning

For example, detecting overfitting with larger datasets will be paradoxically harder (try it!)

Summary

- Generalization is important for supervised and unsupervised learning.
- A probabilistic model can detect overfitting by comparing the likelihood of training data vs. that of holdout data.
- We can reduce overfitting by making the model less expressive.