CS5489 - Machine Learning

Lecture 6a - Unsupervised Learning: Dimensionality Reduction

Dr. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

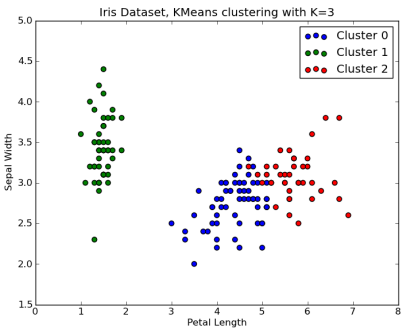
Supervised Learning

- *Supervised learning* considers input-output pairs $(\mathbf{x}, y)$
  - learn a mapping from input to output.
  - *classification*: output $y \in \pm 1$
  - *regression*: output $y \in \mathbb{R}$
- "Supervised" here means that the algorithm is learning the mapping that we want.

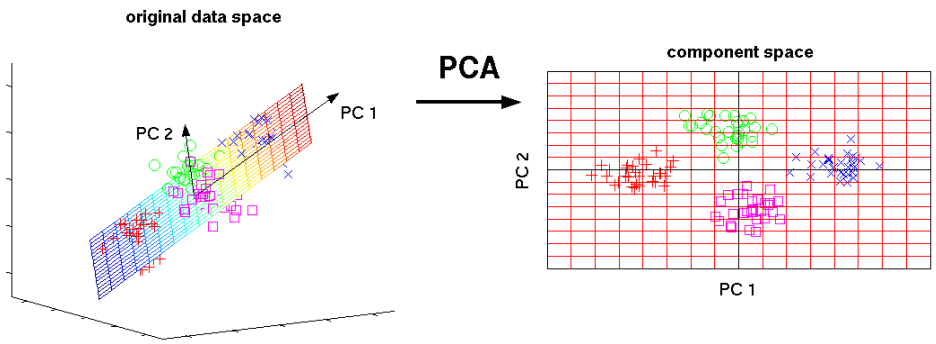
Unsupervised Learning

- Unsupervised learning only considers the input data $\mathbf{x}$.
  - There are no output values.
- **Goal:** Try to discover inherent properties in the data.

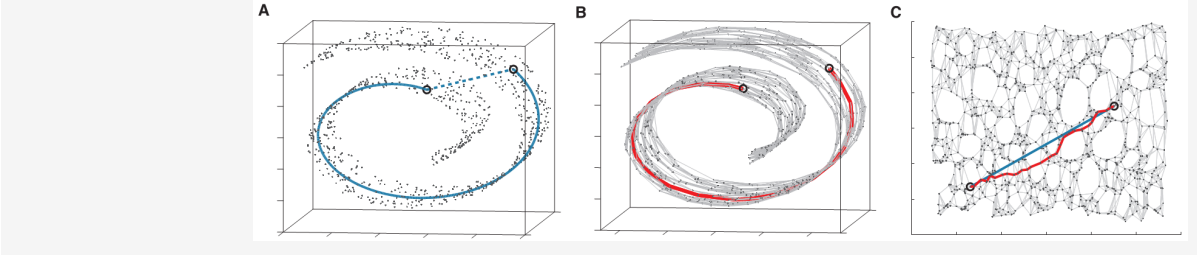
Clustering



Dimensionality Reduction



Manifold Embedding

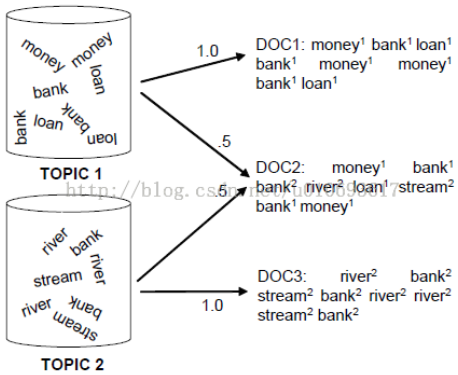


Outline

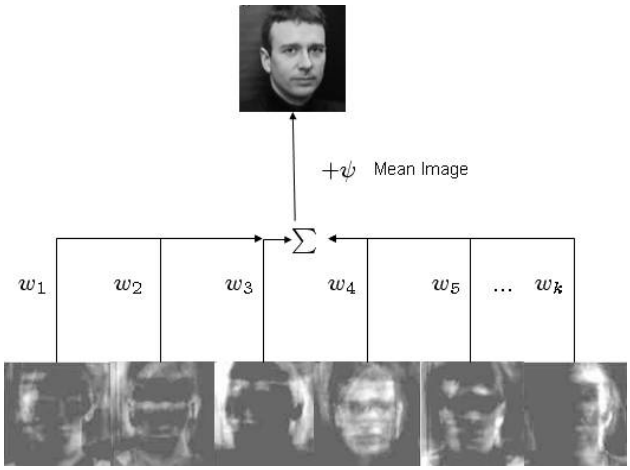
1. **Linear Dimensionality Reduction for Vectors**
2. Linear Dimensionality Reduction for Text
3. Non-linear Dimensionality Reduction
4. Manifold Embedding

Dimensionality Reduction

- **Goal:** Transform high-dimensional vectors into low-dimensional vectors.
  - Dimensions in the low-dim data represent co-occuring features in high-dim data.
  - Dimensions in the low-dim data may have semantic meaning.
- **For example:** document analysis
  - high-dim: bag-of-word vectors of documents
  - low-dim: each dimension represents similarity to a topic.



- **Example:** image analysis
  - approximate an image as a weighted combination of several basis images
  - represent the image as the weights.

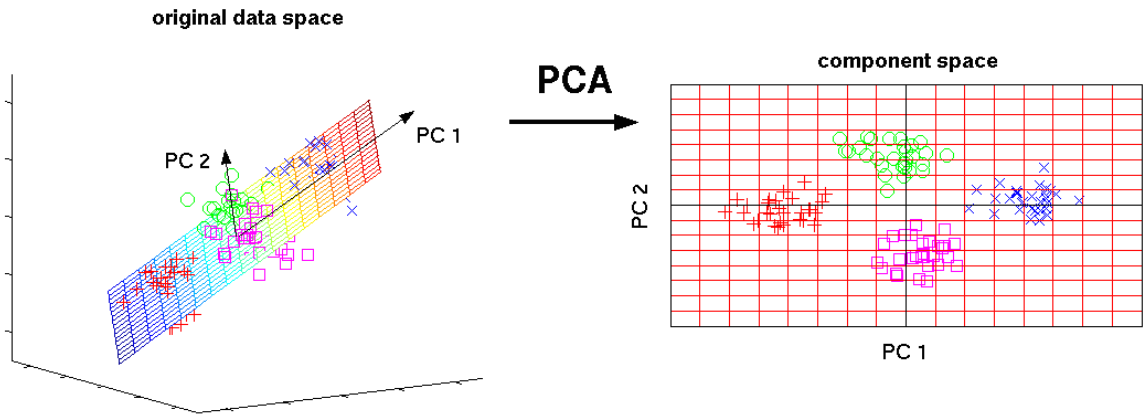


Reasons for Dimensionality Reduction

- Preprocessing - make the dataset easier to use
- Reduce computational cost of running machine learning algorithms
- Remove noise - convert to lower dimension, and then project back to high-dimension
- Make the results easier to understand (visualization)

Linear Dimensionality Reduction

- Project the original data onto a lower-dimensional hyperplane (e.g., line, plane).
  - I.e, Move and rotate the coordinate axis of the data
- Represent the data with coordinates in the new component space.



- Equivalently, approximate the data point $\mathbf{x}$ as a linear combination of basis vectors (components) in the original space.
  - original data point $\mathbf{x} \in \mathbb{R}^d$
  - approximation: $\hat{\mathbf{x}} = \sum_{j=1}^{p} w_j \mathbf{v}_j$
    - $\mathbf{v}_j \in \mathbb{R}^d$ is a basis vector and $w_j \in \mathbb{R}$ the corresponding weight.
  - the data point $\mathbf{x}$ is then represented its corresponding weights
    - $\mathbf{w} = [w_1, \cdots, w_P] \in \mathbb{R}^p$

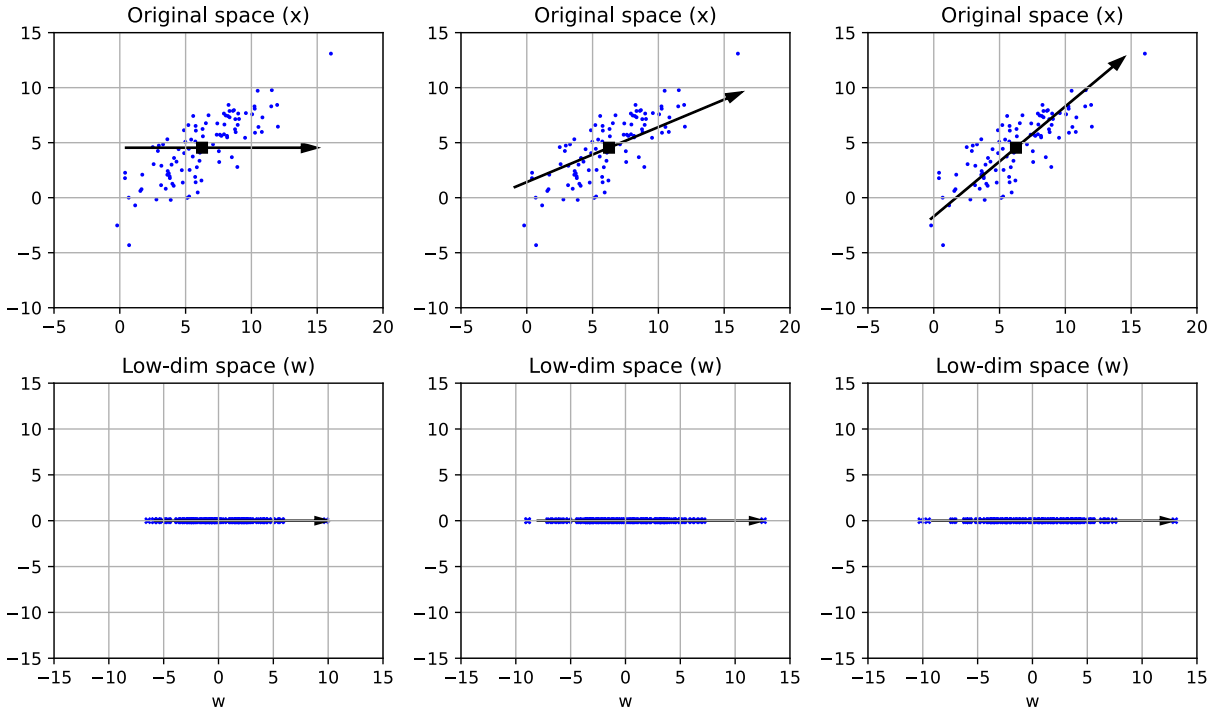
- Several methods for linear dimensionality reduction.
- **Differences:**
  - goal (reconstruction vs classification)
  - unsupervised vs. supervised
  - constraints on the basis vectors and the weights.
  - reconstruction error criteria

Principal Component Analysis (PCA)

- Unsupervised method
- **Goal:** preserve the variance of the data as much as possible
  - choose basis vectors along the maximum variance (longest extent) of the data.
  - the basis vectors are called *principal components* (PC).

In [5]:
```
vfig
```

Out[5]:



- **Goal:** Equivalently, minimize the reconstruction error over all the data points $\{\mathbf{x}_i\}_{i=1}^N$.
  - reconstruction: $\hat{\mathbf{x}}_i = \sum_{j=1}^p w_{i,j}\mathbf{v}_j$

$$\min_{w,\mathbf{v}} \sum_{i=1}^{N} ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2$$

  - *constraint:* principal components $\mathbf{v}_j$ are orthogonal (perpendicular) to each other.
    - $\mathbf{v}_j^T\mathbf{v}_i = \begin{cases} 1, i = j \\ 0, i \neq j \end{cases}$

PCA algorithm

- Iteratively select directions that explain the most variance:
  - 1) subtract the mean of the data
  - 2) the first PC $\mathbf{v}_1$ is the direction that explains the most variance of the data.
  - 3) the second PC $\mathbf{v}_2$ is the direction perpendicular to $\mathbf{v}_1$ that explains the most variance.
  - 4) the third PC $\mathbf{v}_3$ is the direction perpendicular to $\{\mathbf{v}_1, \mathbf{v}_2\}$ that explains the most variance.
  - 5) ...

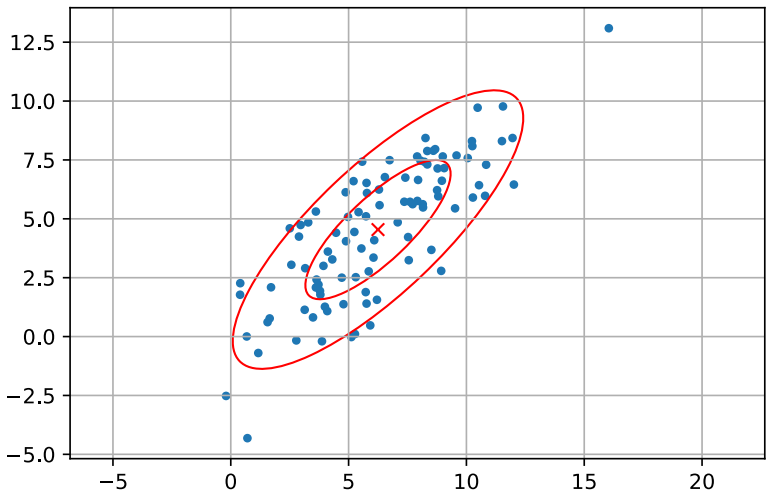
Solution

- Define the mean-subtracted data as $\bar{\mathbf{x}}_i = \mathbf{x}_i - \mu$.
  - matrix of data $\bar{\mathbf{X}} = [\bar{\mathbf{x}}_1 \cdots \bar{\mathbf{x}}_N]$.
- Consider the covariance matrix of the data: $\Sigma = \frac{1}{N}\sum_i \bar{\mathbf{x}}_i \bar{\mathbf{x}}_i^T = \frac{1}{N}\bar{\mathbf{X}}\bar{\mathbf{X}}^T$
  - the covariance matrix defines ellipses of equal-probability of the Gaussian

In [7]:
```
Sigma = cov(X, rowvar=False)
print(Sigma)
gfig
```

```
[[9.4987893  7.29218068]
 [7.29218068 8.74288433]]
```

Out[7]:



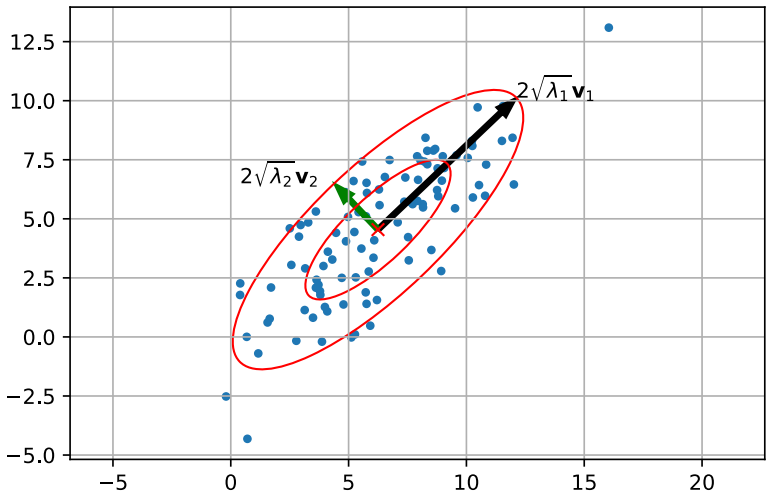
- Consider the eigenvectors and eigenvalues of Σ
  - i-th eigenvector/value pair: $\Sigma\mathbf{v}_i = \lambda_i \mathbf{v}_i$
  - all eigen-pairs: $\Sigma\mathbf{V} = \mathbf{V}\Lambda$
    - matrix of eigenvectors $\mathbf{V} = [\mathbf{v}_1 \cdots \mathbf{v}_D]$
    - eigenvectors are orthonormal: $\mathbf{V}^T\mathbf{V} = \mathbf{I}$
    - diagonal matrix of eigenvalues: $\Lambda = \text{diag}([\lambda_1, \cdots, \lambda_D])$
  - **Eigendecomposition** of covariance: $\Sigma = \mathbf{V}\Lambda\mathbf{V}^T$

- the eigenvector $\mathbf{v}_i$ is an axis of the ellipse.
- the extent of axis $\mathbf{v}_i$ is related to the eigenvalue: $\sqrt{\lambda_i}$

In [9]:
```
[L,V] = linalg.eig(Sigma)
print("lambda = ", L)
print("v1 = ", V[:,0])
print("v2 = ", V[:,1])
gfig
```

```
lambda =  [16.42280553  1.8188681 ]
v1 =  [0.72517596 0.68856359]
v2 =  [-0.68856359  0.72517596]
```

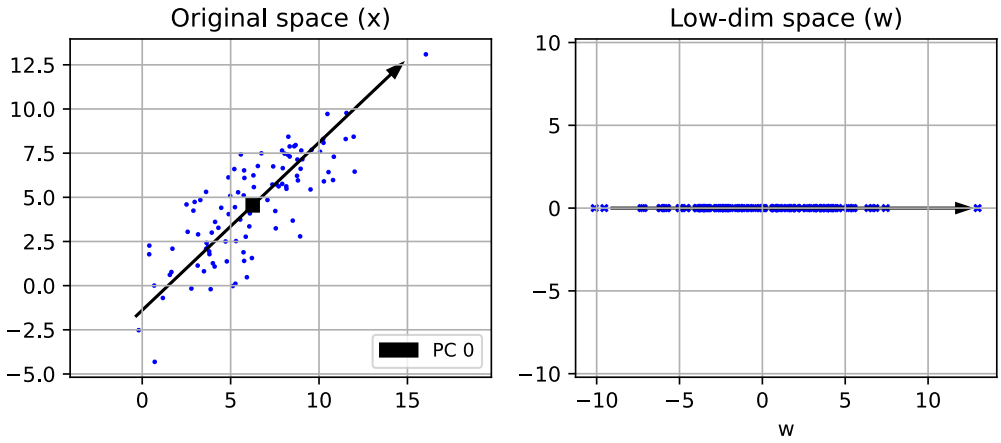
Out[9]:



- Thus the solution to PCA is to select the eigenvectors with largest eigenvalues first.
- To reduce to K dimensions
  - 1) subtract the mean of the data
  - 2) compute the covariance matrix Σ of the data.
  - 3) sort the eigenvector/values of Σ by $\lambda_1 > \lambda_2 > \cdots > \lambda_d$
  - 4) select the top K eigenvectors: $\mathbf{V} = [\mathbf{v}_1 \cdots \mathbf{v}_K]$
  - 5) project the data onto the PCA basis: $\mathbf{w}_i = \mathbf{V}^T\mathbf{x}_i \in \mathbb{R}^K$
- Reconstruction: $\hat{\mathbf{x}}_i = \mathbf{V}\mathbf{w}_i$

Example - 1 PC

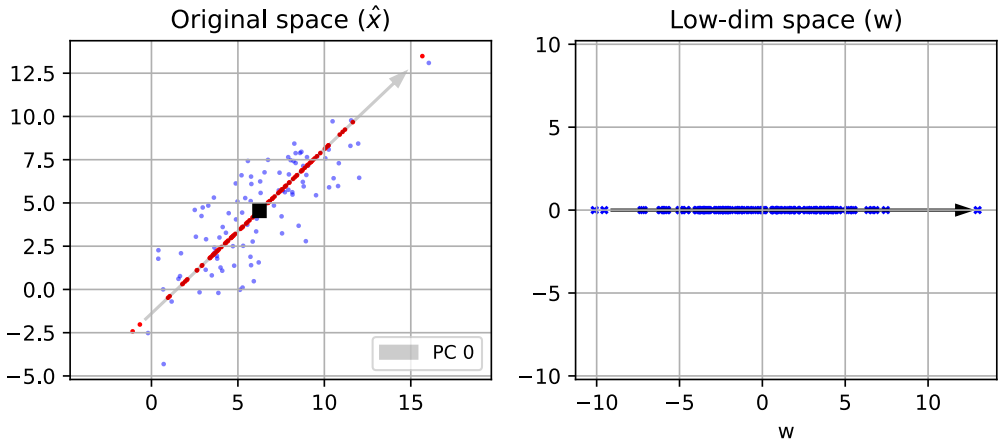
In [10]:
```
X = Xblob

# run PCA
pca = decomposition.PCA(n_components=1)
W   = pca.fit_transform(X)  # returns the coefficients

v = pca.components_  # the principal component vector
m = pca.mean_        # the data mean

plt.figure(figsize=(8,3))
plot_basis(X, v);
```



Example - 1 PC reconstruction

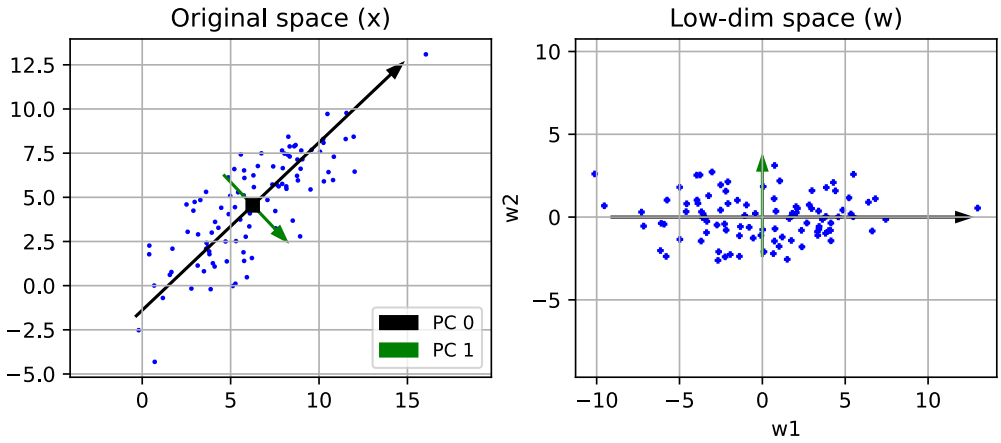
- reconstucted $\mathbf{x}_i$ are projected onto the PC

In [11]:
```
plt.figure(figsize=(8,3))
plot_basis(X, v, recon=True);
```



Example - 2 PC

In [12]:
```
# run PCA
pca = decomposition.PCA(n_components=2)
W   = pca.fit_transform(X)

v = pca.components_  # the principal component vector
m = pca.mean_        # the data mean

plt.figure(figsize=(8,3))
plot_basis(X, v);
```

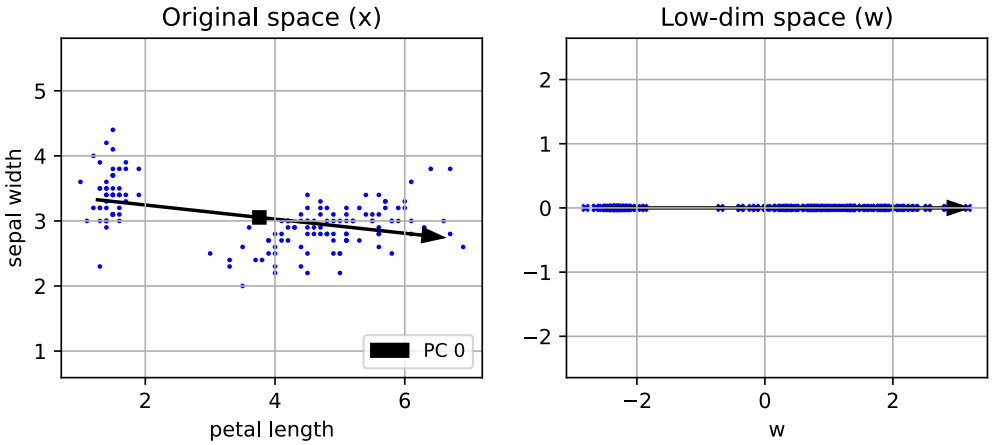


Example on Iris data

- 2D (petal length, sepal width) to 1D

In [14]:
```
ifig
```

Out[14]:



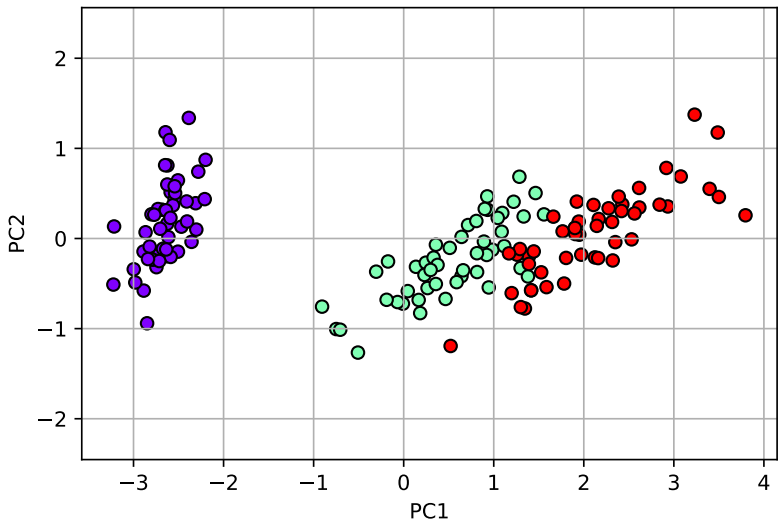
- 4D to 2D
- mostly preserves the structure of the classes.

In [15]:
```
# get data
iris = datasets.load_iris()
X = iris.data
Y = iris.target

# run PCA
pca = decomposition.PCA(n_components=2)
W = pca.fit_transform(X)
print(iris.feature_names)
print(pca.components_)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal w
idth (cm)']
[[ 0.36138659 -0.08452251  0.85667061  0.3582892 ]
 [ 0.65658877  0.73016143 -0.17337266 -0.07548102]]
```

In [17]:
```
i4fig
```

Out[17]:



How to choose the number of principal components?

- Two methods to set the number of components p:
  - preserve some percentage of the variance (e.g., 95%).
  - whatever works well for our final task (e.g., classification, regression).

Handwritten digits data

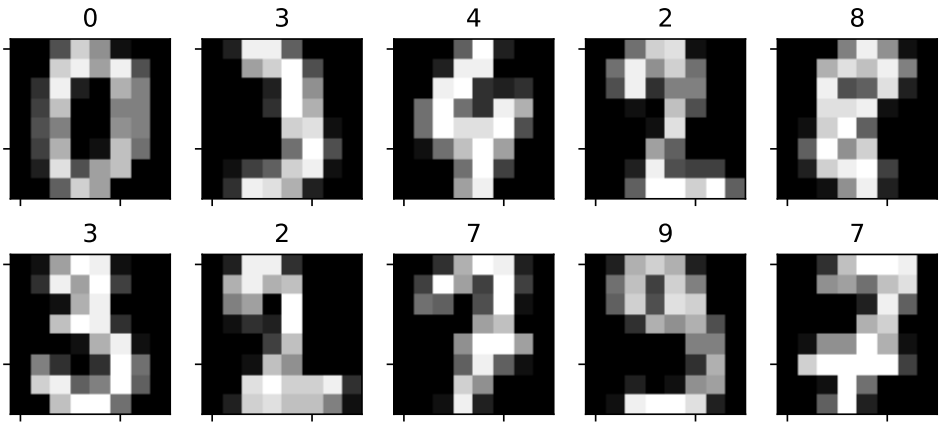
- 1797 images of handwritten digits 0-9
  - each image is 8x8
  - flattened into a 64 dimensional vector

In [18]:
```
# get digit data
digits = datasets.load_digits()
Xdigits = float64(digits.data)
Ydigits = digits.target

print(Xdigits.shape)
```

```
(1797, 64)
```

In [20]:
```
dfig
```

Out[20]:



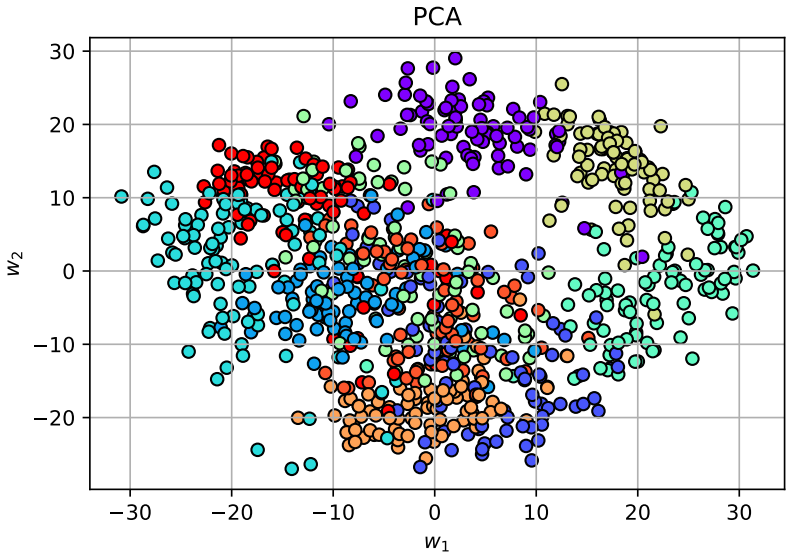
Run PCA on the data

- split data into training and testing sets.
- run PCA on training set, apply to test set

In [21]:
```
# randomly split data into 80% train and 20% test set
trainX, testX, trainY, testY = \
  model_selection.train_test_split(Xdigits, Ydigits,
  train_size=0.5, test_size=0.5, random_state=4487)
Xdim = Xdigits.shape[1]

# run PCA
pca = decomposition.PCA() # default: n_components=dimension
W   = pca.fit_transform(trainX)  # fit the training set
Wt = pca.transform(testX)  # use the pca model to transform the test set
```

- Visualize the coefficients for the first two PCs.
  - grouping of different digits is sometimes preserved

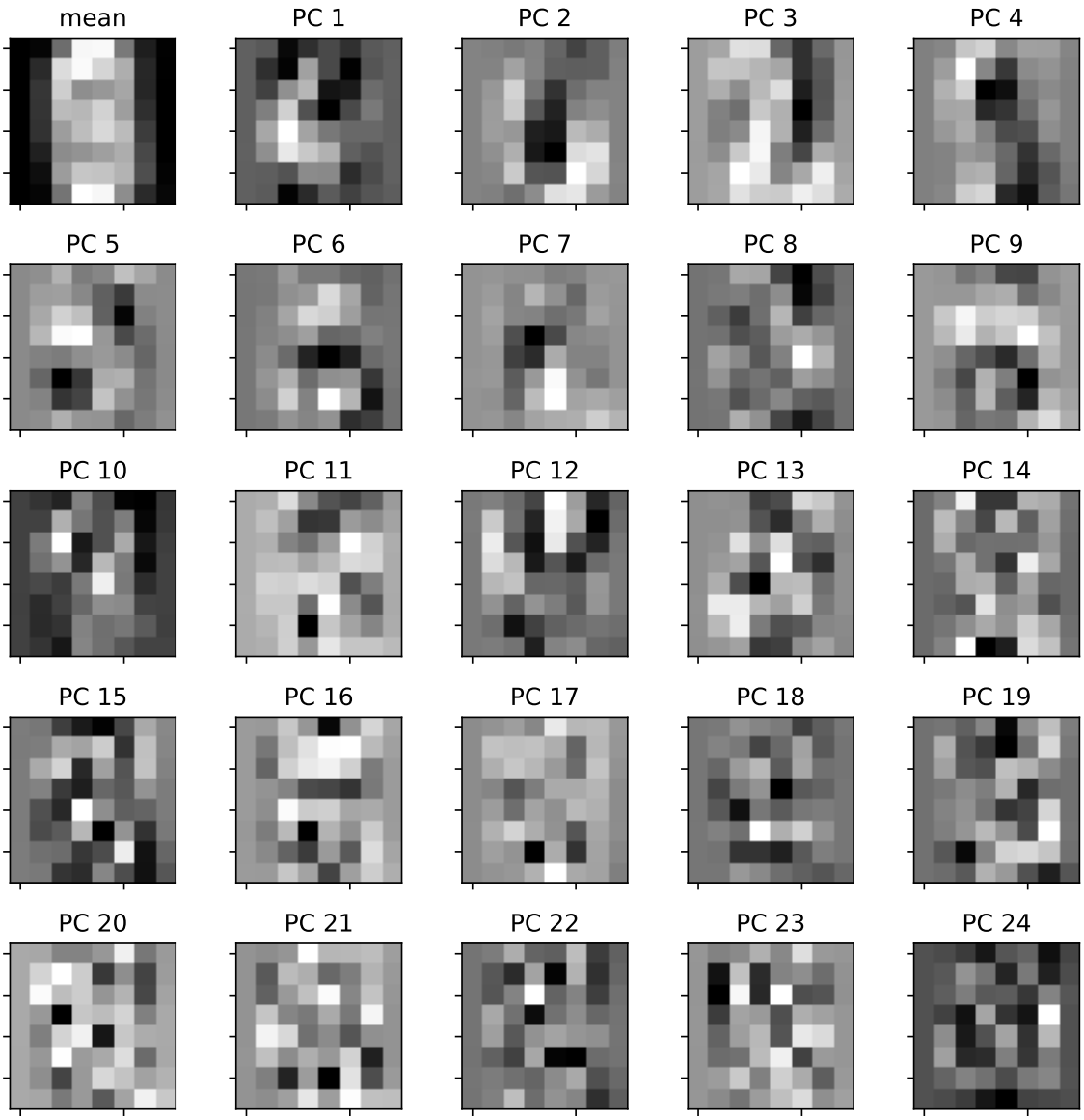
In [22]:
```
plt.figure()
plt.scatter(W[:,0], W[:,1], c=trainY, cmap=rbow, edgecolors='k')
plt.xlabel('$w_1$'); plt.ylabel('$w_2$')
plt.title('PCA'); plt.grid(True);
```



- Look at the mean and principal components

In [24]:
```
pcfig
```

Out[24]:

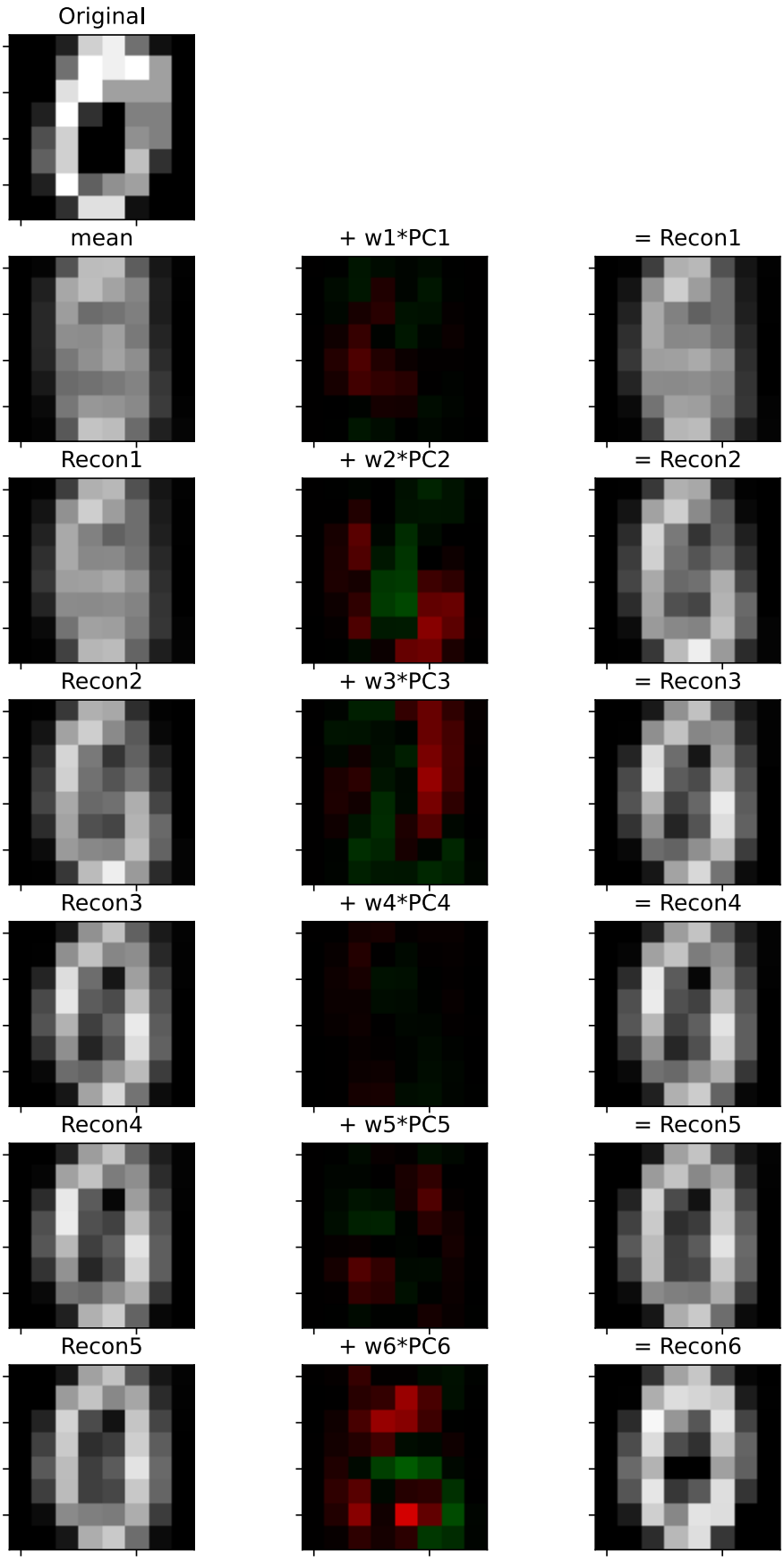


- Reconstruction of a digit image from PC coefficients
  - red/green corresponds to postive/negative values
  - using more PCs will make the reconstruction better

In [26]:
```
print(myW[0:6])
reconfig
```

```
[  6.3225555   14.02802981 -13.20374264   3.42192754  -6.55493652
  17.31869321]
```

Out[26]:

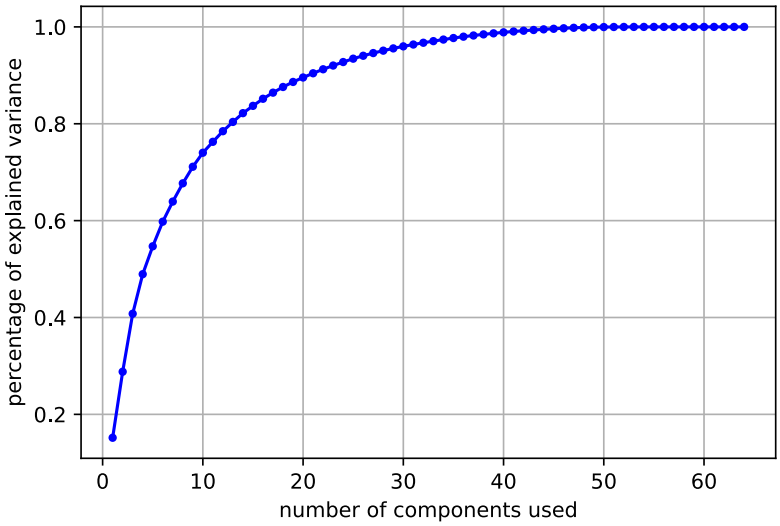


Explained variance

- each PC explains a percentage of the original data
  - this is called the *explained variance*.
  - PCs are already sorted by explained variance from highest to lowest

- pick the number of PCs to get a certain percentage of explained variance
  - typically 95%

In [27]:
```
ev    = pca.explained_variance_ratio_  # variance explained by each component
cumev = cumsum(ev)                     # cumulative explained variance

plt.plot(range(1,Xdim+1), cumev, 'b.-')
plt.grid(True)
plt.xlabel('number of components used')
plt.ylabel('percentage of explained variance');
```



Task-dependent Selection

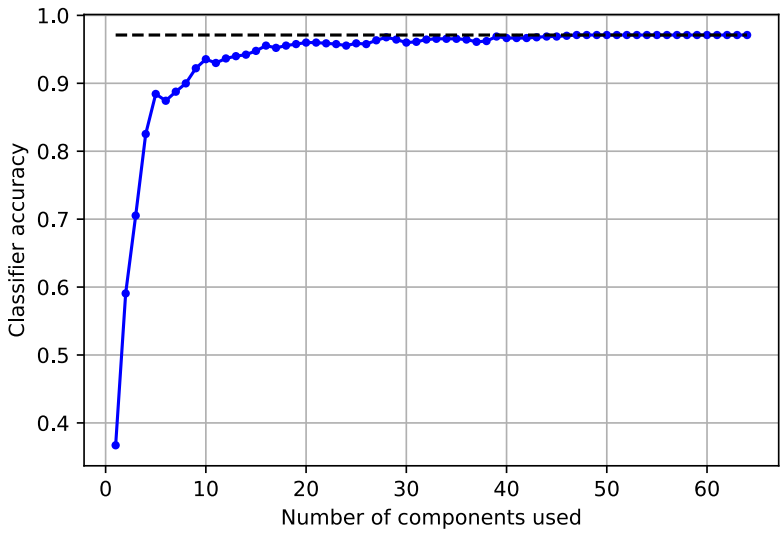
- use results on the final task (in this case classification) to select the best number of components
- Note: we don't need to rerun PCA for each number of components
  - just select the subset of PCs based on the number of components desired.

In [28]:
```
acc = zeros(Xdim)
for j in range(Xdim):
    # extract the subset of PC weights [0,j]
    Wnew     = W[:,0:(j+1)]
    Wnewtest = Wt[:,0:(j+1)]

    # train classifier
    clf = svm.SVC(kernel='linear', C=1)
    clf.fit(Wnew, trainY)

    # test classifier
    Ypred = clf.predict(Wnewtest)
    acc[j] = metrics.accuracy_score(testY, Ypred)
```

- classification accuracy is stable after using 20 PCs.
  - not much loss in performance if using only 20 PCs.

In [29]:
```
# make a plot
plt.plot(range(1,Xdim+1), acc, '.b-')
plt.plot([1,Xdim], [acc.max(), acc.max()], 'k--')
plt.grid(True)
```

```
plt.xlabel('Number of components used')
plt.ylabel('Classifier accuracy');
```



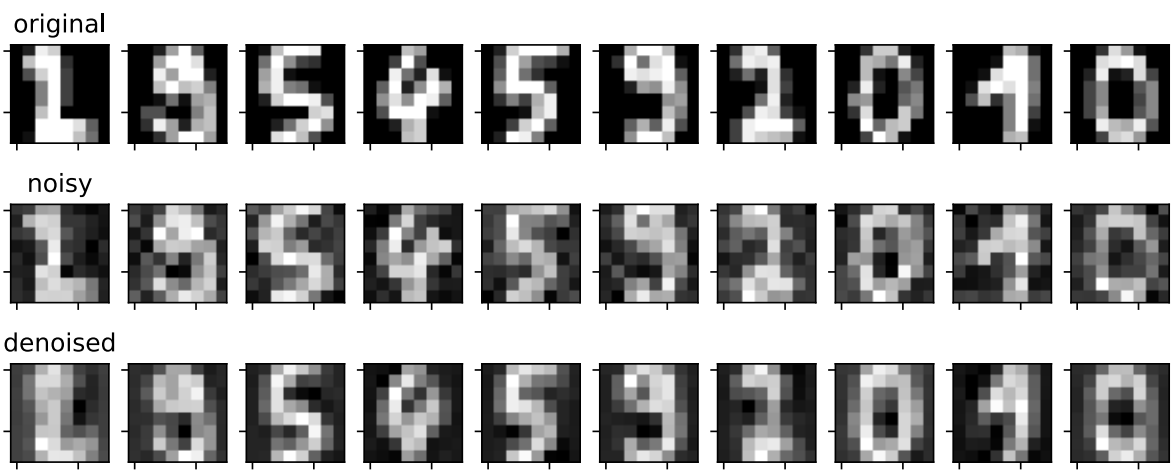
Denoising

- the low-dim PCA space summarizes the important variations of the data.
- the original image can be denoised by:
  - 1) project into the low-dimensional space to get PCA coefficients
    - (keep only important variations)
  - 2) reconstruct an image from the PCA coefficients

In [30]:
```
# add noise to data
noisyX = trainX + 2*random.normal(size=trainX.shape)
noisyXt = testX + 2*random.normal(size=testX.shape)

# learn PCA
pca = decomposition.PCA(n_components=10)
pca.fit(noisyX)  # fit the training set

# transform and reconstuct
testW = pca.transform(noisyXt)
testXr = pca.inverse_transform(testW)   # reconstruction
```

In [32]:
```
rfig
```

Out[32]:



Random Projections

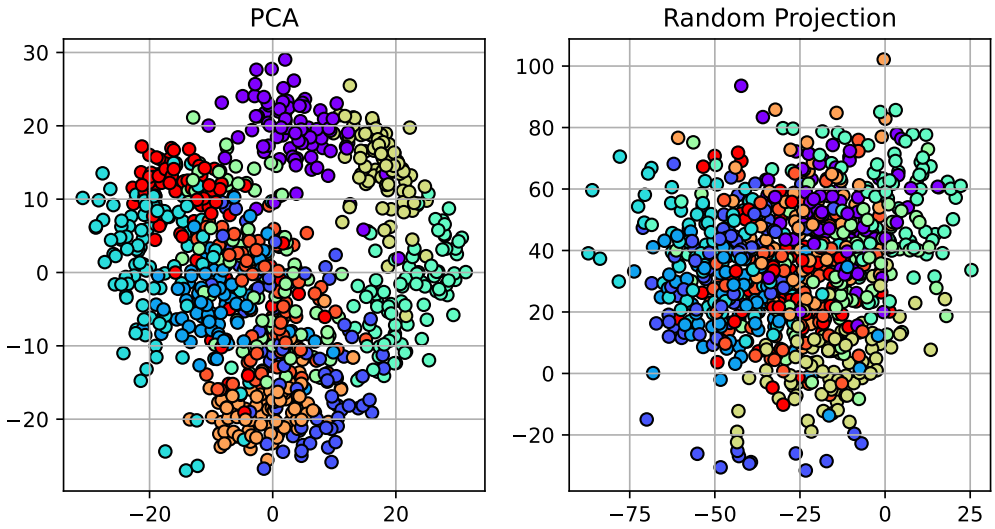
- If the data is very high-dimensional, then it might take too many calculations to do PCA.
  - Complexity: $O(dk^2)$, d is the dimension, k is the number of components
- Do we really need to estimate the principal components to reduce the dimension?

- **Solution:**
  - We can generate random basis vectors and use those.
    - Each entry of $\mathbf{v}_j$ sampled from a Gaussian.
  - This will save a lot of time.
  - Random Projections can reduce computation at the expense of losing some accuracy in the points (adding noise).

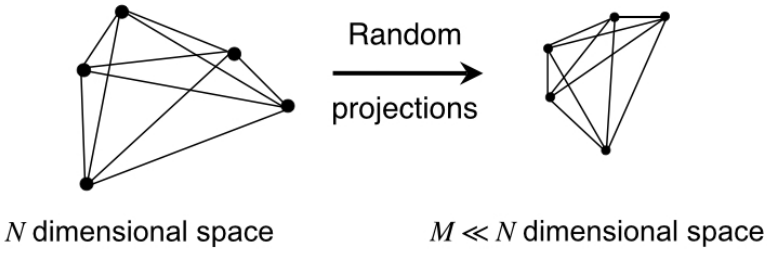
In [33]:
```
# project the digits data with Random Projection
rp = random_projection.GaussianRandomProjection(n_components=2, random_state=4487)
Wrp = rp.fit_transform(trainX)
```

In [35]:
```
rfig
```

Out[35]:



- Okay, but is it good?
  - One way to measure "goodness" is to see if the structure of the data is preserved.
  - In other words, are distances between points preserved in the transformed data?

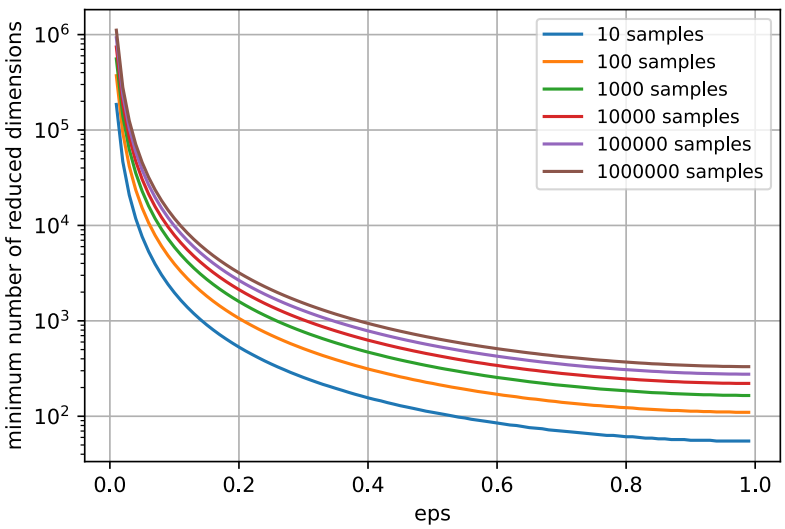


- **Answer:**
  - Yes!
  - According to the *Johnson-Lindenstrauss lemma*, carefully selecting the distribution of the random projection matrices will preserve the pairwise distances between any two samples of the dataset, within some error *epsilon*.
    - $(1-\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2 < \|\mathbf{w}_i - \mathbf{w}_j\|^2 < (1+\epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|^2$

- the minimum reduced dimension p to gaurantee an ϵ error depends on the number of samples.
  - (actually, this is fairly conservative)

In [37]:
```
jlfig
```

Out[37]:

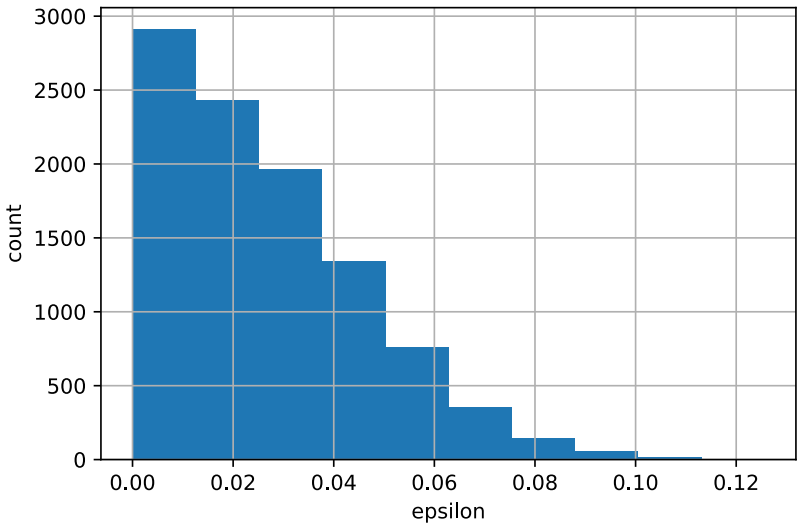


- Example

In [38]:
```
# generate random data
# (dimension=10000, samples=100)
X = random.rand(100,10000)

# fit to 500 components
rp = random_projection.GaussianRandomProjection(n_components=500, random_state=4487)
Wrp = rp.fit_transform(X)
```

In [40]:
```
efig
```

Out[40]:



Sparse Random Projection

- More computation can be saved by using a *sparse* random projection matrix
  - "sparse" means that many entries in the basis vector are zero, so we can ignore those entries when multiplying.

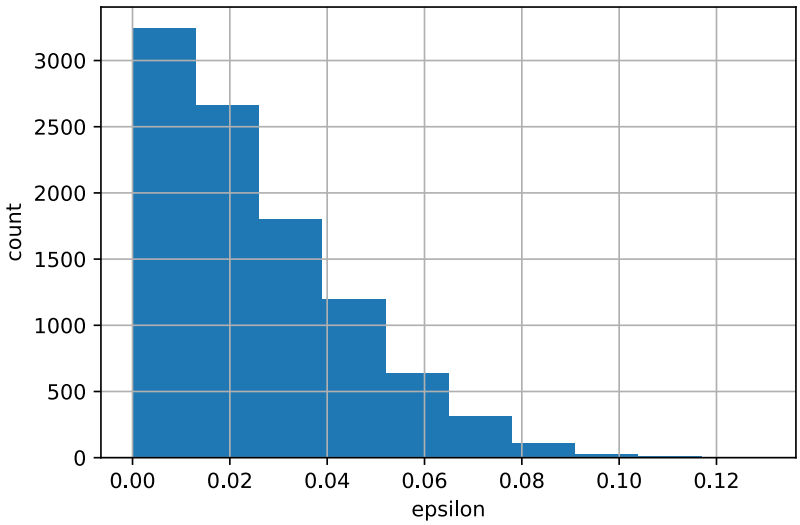
In [41]:

```
# project the digits data with Random Projection
srp = random_projection.SparseRandomProjection(n_components=500, random_state=4487)
Wsrp = srp.fit_transform(X)
```

In [45]:

```
efig
```

Out[45]:



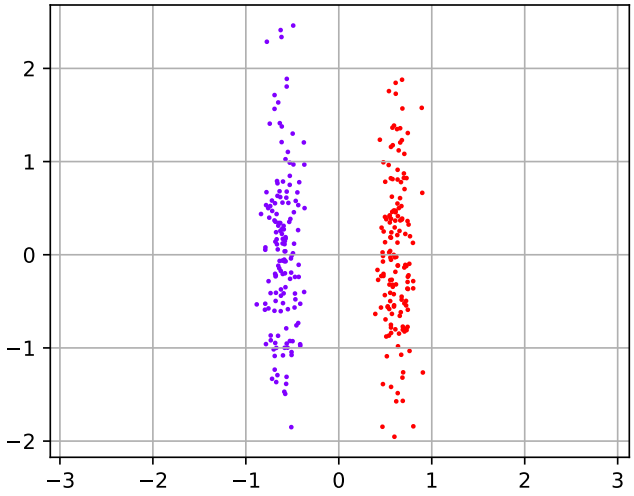
Question

- Suppose we have the below classification problem...
- We want to reduce the data to 1 dimension using PCA.
  - What is the first PC?

In [47]:

```
efig
```

Out[47]:

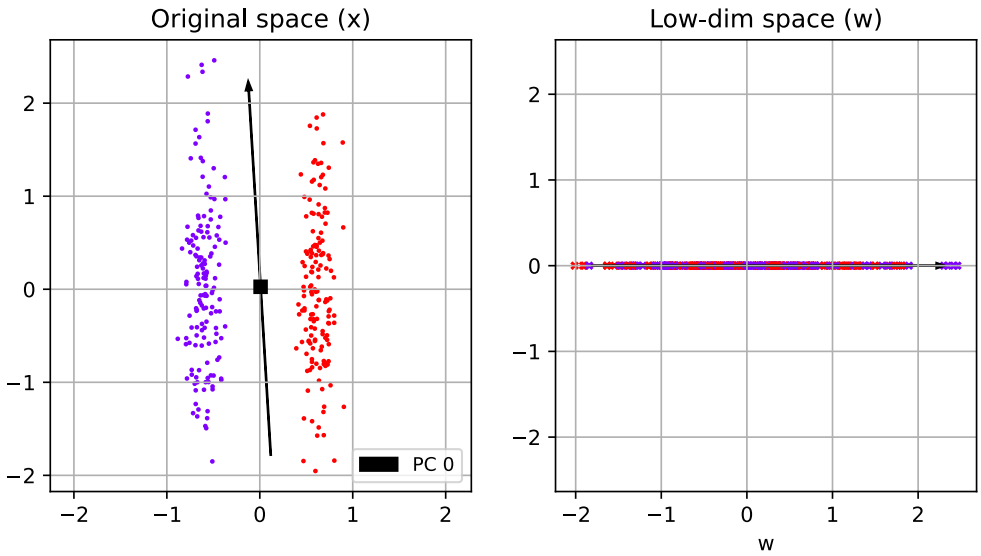


Answer

- first PC is along the direction of most variance.
  - collapses the two classes together!

In [48]:
```
e2fig
```

Out[48]:



Problem with Unsupervised Methods

- If our end goal is classification, preserving the variance sometimes won't help!
  - PCA doesn't consider which class the data belongs to.
  - When the "classification" signal is less than the "noise", PCA will make classification more difficult.

Fisher's Linear Discriminant (FLD)

- Supervised dimensionality reduction

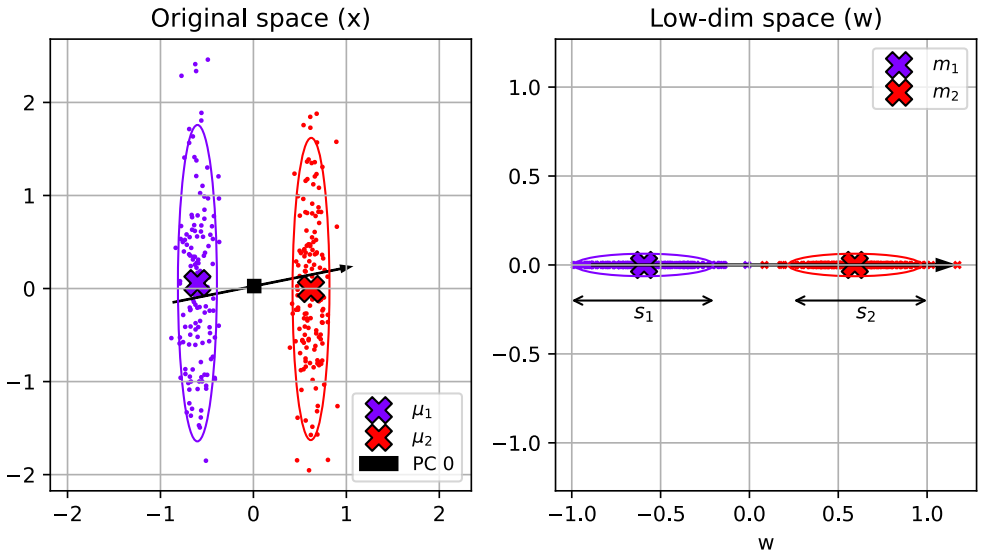
- Also called *"Linear Discriminant Analysis"* (LDA)
- **Goal:** find a lower-dim space so as to minimize the class overlap (or maximize the class separation).
  - data from each class is modeled as a Gaussian.
  - requires the class labels

Problem setup

- Input space: class means μ_j and covariance (scatter) matrices $\mathbf{S}_j$.
- Projected space: class means $m_j = \mathbf{w}^T \mu_j$, and scatter $s_j = \mathbf{w}^T \mathbf{S}_j \mathbf{w}$

In [50]:
```
lfig
```

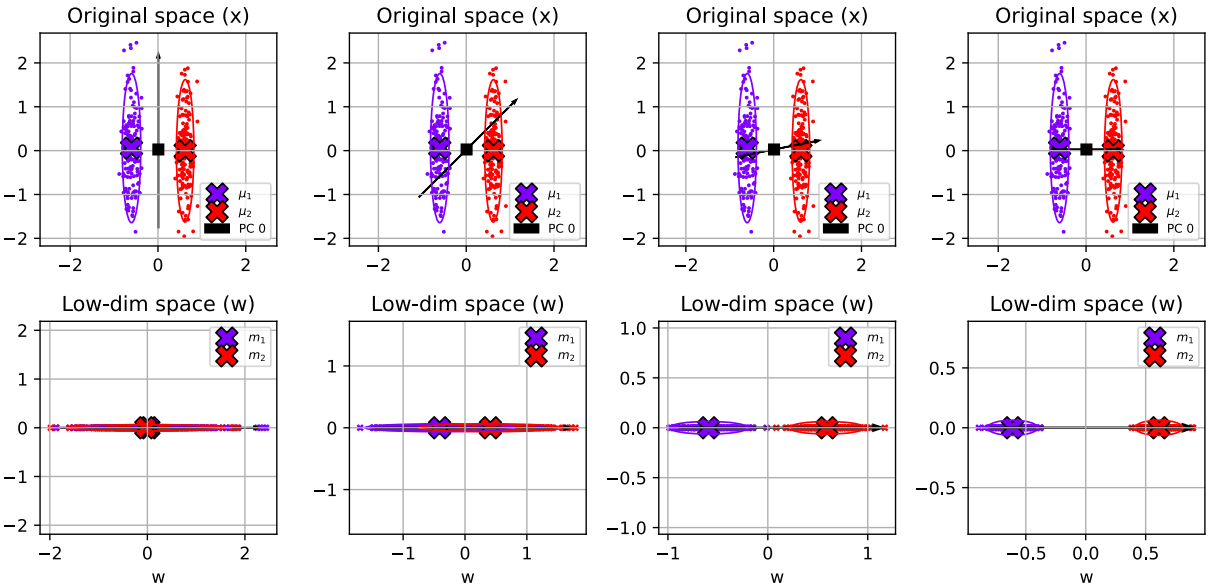
Out[50]:



- **Idea:** make the projected points in each class as compact as possible
  - maximize the distance between the projected means
  - minimize the projected variances

In [52]:
```
efig
```

Out[52]:



- Fisher's linear discriminant (FLD)
  - Problem:

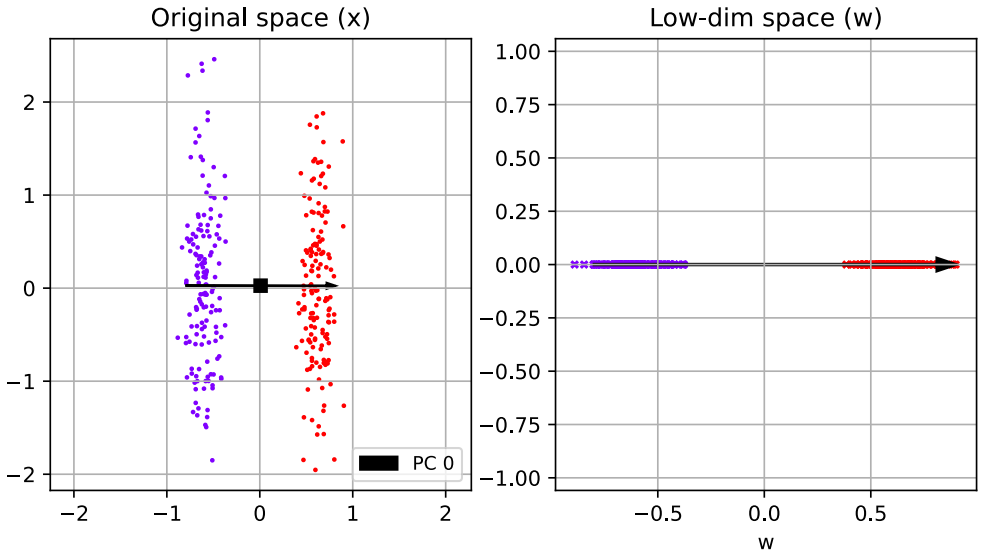
$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmax}} \frac{(m_1 - m_2)^2}{s_1 + s_2}$$

  - Solution:

$$\mathbf{w}^* = (\mathbf{S}_1 + \mathbf{S}_2)^{-1}(\mu_1 - \mu_2)$$

In [53]:

```
# example of FLD projection (using LDA name)
fld = discriminant_analysis.LinearDiscriminantAnalysis(n_components=1)
W  = fld.fit_transform(X, Y)

v = fld.coef_   # the basis vectors

plt.figure(figsize=(8,4))
plot_basis(X, v, Y=Y);
```



On Iris data

- 4D vector to 2D vector
- FLD forms more compact classes
- With FLD, classes have less overlap if only using 1st basis vector.

In [55]:
```
ifig
```

Out[55]: