Lecture 09

Machine Learning 3:

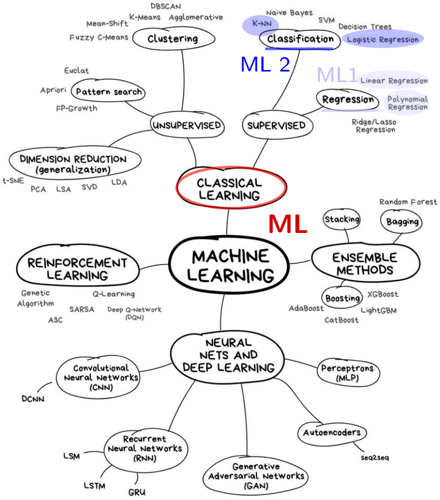
classification (part 2)

2024-10-16

Sébastien Valade

Introduction

Last week: classification *part-1* (ML2)

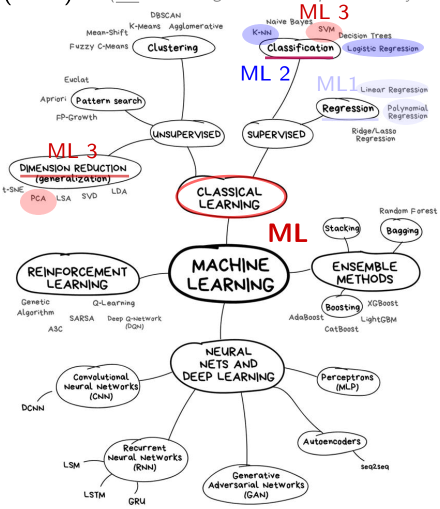


source

Introduction

Last week: classification *part-1* (ML2)
This week: classification *part-2* (ML3)

(NB: several figures were adapted courtesy of A. Ley & R. Hänsch from TU-Berlin)

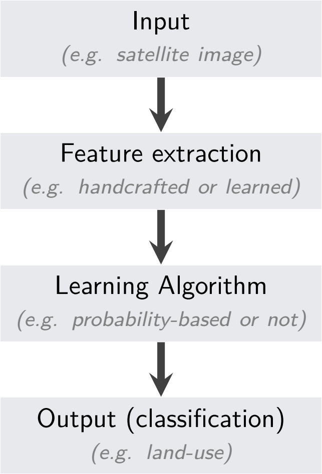


source

Introduction

Classification task

- Goal:
  Learn the mapping between low level **features**, and **high level information** *(e.g. semantic classes)*
- Steps:
  1. features extraction *(e.g. handcrafted | learned)*
  2. learning algorithm *(e.g. probablity-based | not)*
- Strategies:
  ⇒ last week:
  handcrafted features + probability-based learning
  ⇒ this week:
  learned features (PCA) + SVM learning



Input
(e.g. satellite image)

Feature extraction
(e.g. handcrafted or learned)

Learning Algorithm
(e.g. probability-based or not)

Output (classification)
(e.g. land-use)

1. Introduction
2. Principal Component Analysis (PCA)
    1. introduction
    2. how it works
    3. implementation steps
3. Classification algorithms (perceptron + SVM)
4. Exercise

Principal Component Analysis (PCA)

⇒ PCA is an **unsupervised learning** technique

- → in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, ..., X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

- → it is used to reduce data dimensionality, while preserving as much of the variance as possible
- → it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?

Principal Component Analysis (PCA)

⇒ PCA is an **unsupervised learning** technique

- → in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, ..., X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

- → it is used to reduce data dimensionality, while preserving as much of the variance as possible
- → it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?

Principal Component Analysis (PCA)

⇒ PCA is an **unsupervised learning** technique

→ in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, ..., X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

→ it is used to reduce data dimensionality, while preserving as much of the variance as possible

→ it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?

Principal Component Analysis (PCA)

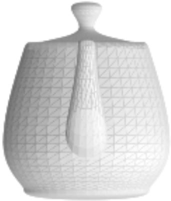
⇒ PCA is an **unsupervised learning** technique

- → in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, ..., X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

- → it is used to reduce data dimensionality, while preserving as much of the variance as possible
- → it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?



NB: run animation with PDF reader having built-in JS engine

Principal Component Analysis (PCA)

⇒ PCA is an **unsupervised learning** technique

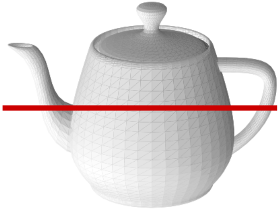
→ in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, ..., X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

→ it is used to reduce data dimensionality, while preserving as much of the variance as possible

→ it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?



1st principal component
= 1st "eigen vector"
(longest axis)

Principal Component Analysis (PCA)

⇒ PCA is an **unsupervised learning** technique

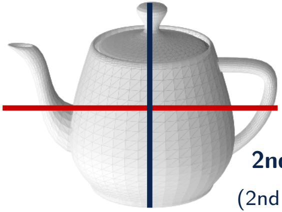
→ in contrast to *supervised learning*, *unsupervised learning* algorithms operate on unlabeled data (we only have a set of k features $X_1, X_2, \ldots, X_k$ measured on n observations, without any associated target variable Y, thus we are not interested in any prediction task)

⇒ PCA is the most popular **dimensionality reduction algorithm**:

→ it is used to reduce data dimensionality, while preserving as much of the variance as possible

→ it is often used as data pre-processing technique before supervised techniques are applied (e.g. feature extraction to reduce computational load of the classifier)

⇒ Intuitive explanation: which angle captures the most information about the teapot?

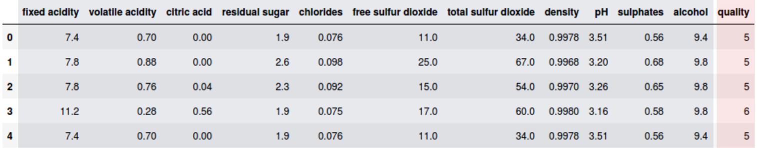


1st principal component
= 1st "eigen vector"
(longest axis)

2nd principal component
= 2nd "eigen vector"
(2nd longest axis ⊥ to 1st axis)

PCA toy example

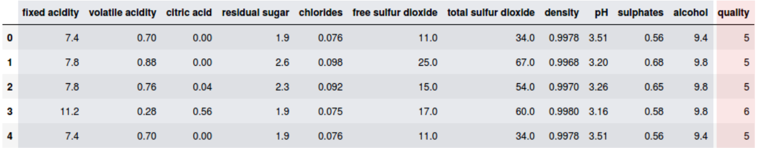
*We have several wine bottles in our cellar, 11 **features** (alcohol, acidity, etc.) describe its **quality**. Which features best define it, are there related features (i.e. covariant) which are redundent?*



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

PCA toy example

We have several wine bottles in our cellar, 11 ***features*** *(alcohol, acidity, etc.) describe its* ***quality****. Which features best define it, are there related features (i.e. covariant) which are redundent?*

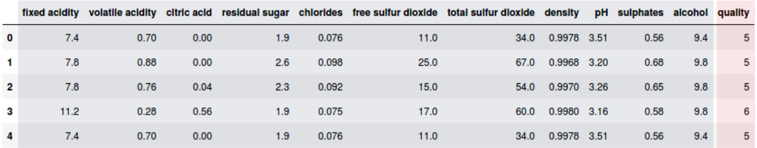


	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

⇒ PCA allows to summarize each wine with fewer characteristics
⇒ **reduce data dimensions**

PCA toy example

We have several wine bottles in our cellar, 11 ***features*** *(alcohol, acidity, etc.) describe its* ***quality****.*
Which features best define it, are there related features (i.e. covariant) which are redundent?



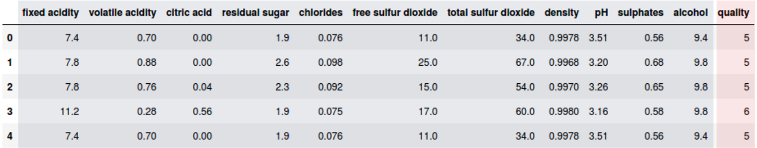
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

⇒ PCA allows to summarize each wine with fewer characteristics
⇒ **reduce data dimensions**

⇒ PCA does not select some features and discards others,
instead it **defines new features** (using linear combinations of available features)
which will best represent wine variability

PCA toy example

We have several wine bottles in our cellar, 11 ***features*** *(alcohol, acidity, etc.) describe its* ***quality****.*
Which features best define it, are there related features (i.e. covariant) which are redundent?



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

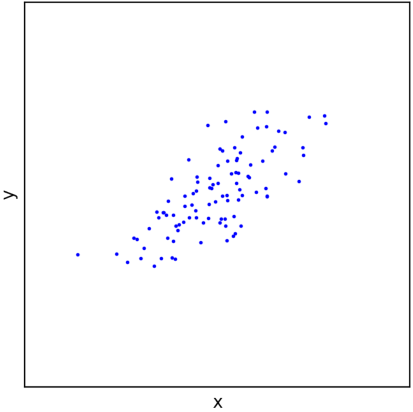
⇒ PCA allows to summarize each wine with fewer characteristics
⇒ **reduce data dimensions**

⇒ PCA does not select some features and discards others,
instead it **defines new features** (using linear combinations of available features)
which will best represent wine variability

How?

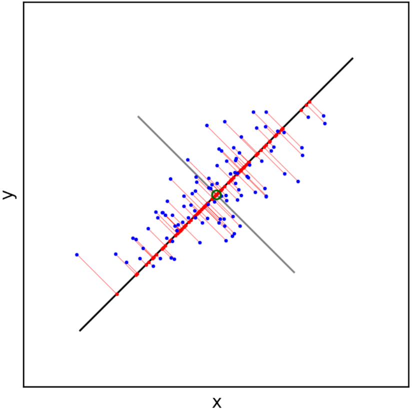
2.2. how it works

Consider 2 correlated features x and y:



Consider 2 correlated features x and y:

⇒ a **new "feature"** (red dots ●) can be constructed by drawing a line through the cloud and projecting all points onto it

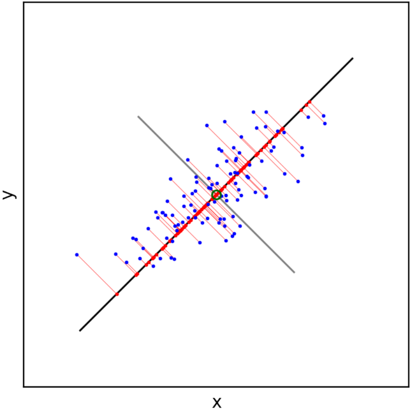


2.2. how it works

Consider 2 correlated features x and y:

$\Rightarrow$ a **new "feature"** (red dots ●) can be constructed by drawing a line through the cloud and projecting all points onto it

$\Rightarrow$ **linear combination** $w_1 x + w_2 y$



2.2. how it works

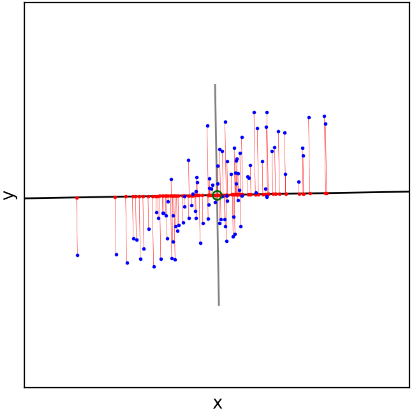
Consider 2 correlated features x and y:

⇒ a **new "feature"** (red dots ●) can be constructed by drawing a line through the cloud and projecting all points onto it

⇒ **linear combination** $w_1 x + w_2 y$

⇒ PCA will find the "best" line according to 2 criteria:

- maximum **variance** of the red dots (i.e., spread along black line)
- minimum **distance** to black line (i.e., length of red lines)



NB: run animation with PDF readers having built-in JavaScript engine

Consider 2 correlated features x and y:

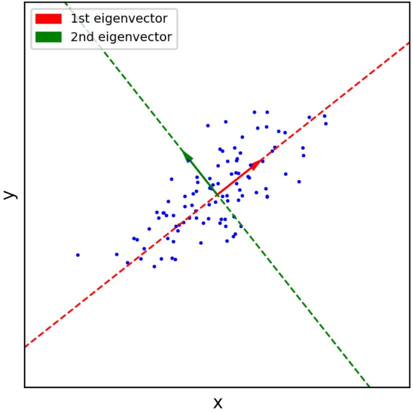
⇒ a **new "feature"** (red dots ●) can be constructed by drawing a line through the cloud and projecting all points onto it

⇒ **linear combination** $w_1 x + w_2 y$

⇒ PCA will find the "best" line according to 2 criteria:

- maximum **variance** of the red dots (i.e., spread along black line)
- minimum **distance** to black line (i.e., length of red lines)

⇒ "best" line = **1st eigenvector** = **1st principal component**



Consider 2 correlated features x and y:

⇒ a **new "feature"** (red dots ●) can be constructed by drawing a line through the cloud and projecting all points onto it

⇒ **linear combination** $w_1x + w_2y$

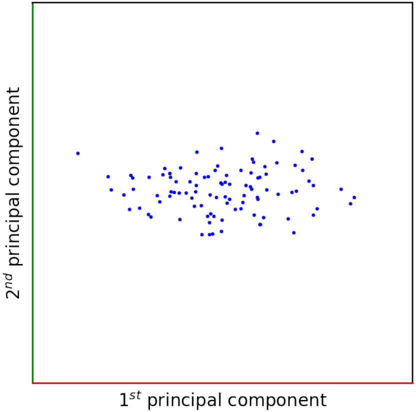
⇒ PCA will find the "best" line according to 2 criteria:

- maximum **variance** of the red dots (i.e., spread along black line)
- minimum **distance** to black line (i.e., length of red lines)

⇒ "best" line = **1st eigenvector** = **1st principal component**

⇒ we can project the data on the principal components, and thereby reduce dimensionality

NB: if only one eigenvector was kept, the transformed data would have only one dimension



Implementation steps

Math reminders

$$\begin{aligned}\textbf{variance } \sigma^2 &= \text{measure of the ``spread'' or ``extent'' of the data about some particular axis} \\ &= \text{average of the squared differences from the mean} \\ &= \text{square of standard deviation } (\sigma) \\ var_x &= \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N} \\ var_y &= \frac{\sum_{i=1}^{N}(y_i - \bar{y})^2}{N}\end{aligned}$$

$$\begin{aligned}\textbf{covariance} &= \text{measure the level to which two variables vary together} \\ cov_{x,y} &= \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N - 1} \\ \textbf{covariance matrix} &= \begin{bmatrix} var_x & cov_{x,y} \\ cov_{y,x} & var_y \end{bmatrix}\end{aligned}$$

Math reminders (continued)

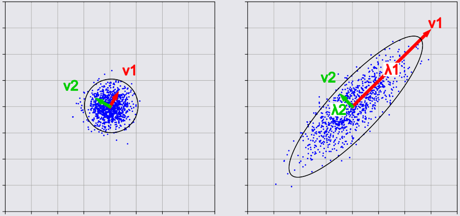
Covariance matrix $= \begin{bmatrix} var_x & cov_{x,y} \\ cov_{y,x} & var_y \end{bmatrix}$

Eigenvalue analysis of covariance matrix $\Rightarrow$ find directions with maximal variance

- **eigenvectors** $(\vec{v}_1, \vec{v}_2)$: represent the directions of the largest variance of the data
- **eigenvalues** (λ_1, λ_2): represent the magnitude of this variance in those directions

Determinant and trace of covariance matrix

- **determinant** $det(covmat) = \lambda_1 \lambda_2$: measures the "spread" of the data captured by the covariance matrix
- **trace** $trace(covmat) = \lambda_1 + \lambda_2$: measures the "total variance" captured by the covariance matrix



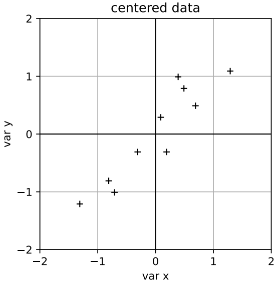
$covmat = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

uncorrelated variables

$covmat = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$

highly correlated variables

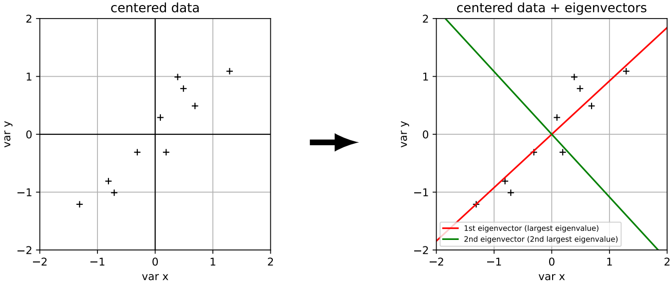
Implementation steps (example with 2 variables)



```
x = x - x.mean()
y = y - y.mean()
```

1. center points around origin $(0, 0)$
2. compute **covariance matrix** → get **eigenvalues** & **eigenvectors** (= Principal Components) → sort by eigenvalue
   - ⇒ eigenvectors represent the directions of the largest variance of the data, eigenvalues represent the magnitude of this variance in those directions
   - ⇒ highest eigenvalue = direction with most variance (data dispersion) = 1^{st} principal component
3. project the data onto the **principal components** (PCs)
   - ⇒ if only 1 eigenvector was kept, the 2 original features (var x, var y) could be reduced to 1 dimension

Implementation steps (example with 2 variables)

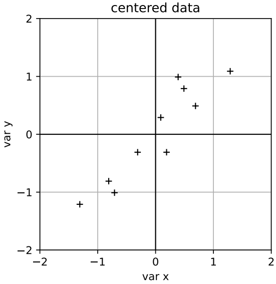


```
x = x - x.mean()
y = y - y.mean()
```

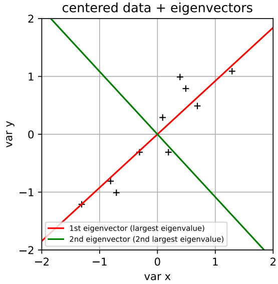
```
cov = np.cov(x, y)
eig_val, eig_vec = np.linalg.eig(cov)
idx = np.argsort(-eig_val)
eig_val, eig_vec = eig_val[idx], eig_vec[:,idx]
```

1. center points around origin $(0, 0)$
2. compute **covariance matrix** → get **eigenvalues** & **eigenvectors** (= Principal Components) → sort by eigenvalue
   - ⇒ eigenvectors represent the directions of the largest variance of the data, eigenvalues represent the magnitude of this variance in those directions
   - ⇒ highest eigenvalue = direction with most variance (data dispersion) = 1^{st} principal component
3. project the data onto the **principal components** (PCs)
   - ⇒ if only 1 eigenvector was kept, the 2 original features (var x, var y) could be reduced to 1 dimension

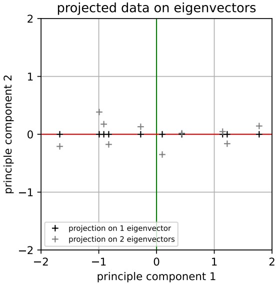
Implementation steps (example with 2 variables)



```
x = x - x.mean()
y = y - y.mean()
```



```
cov = np.cov(x, y)
eig_val, eig_vec = np.linalg.eig(cov)
idx = np.argsort(-eig_val)
eig_val, eig_vec = eig_val[idx], eig_vec[:,idx]
```



```
XY = np.array([x, y]).T
XY_proj = np.dot(XY, eig_vec[:2].T)
```

1. center points around origin $(0, 0)$
2. compute **covariance matrix** → get **eigenvalues** & **eigenvectors** (= Principal Components) → sort by eigenvalue
   - ⇒ eigenvectors represent the directions of the largest variance of the data, eigenvalues represent the magnitude of this variance in those directions
   - ⇒ highest eigenvalue = direction with most variance (data dispersion) = 1^{st} principal component
3. project the data onto the **principal components** (PCs)
   - ⇒ if only 1 eigenvector was kept, the 2 original features (var x, var y) could be reduced to 1 dimension

2.3. implementation steps

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

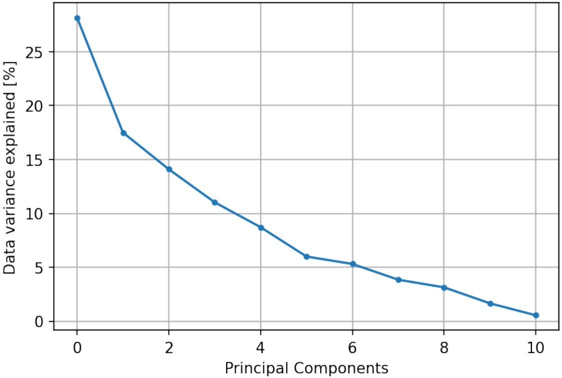
2.3. implementation steps

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?



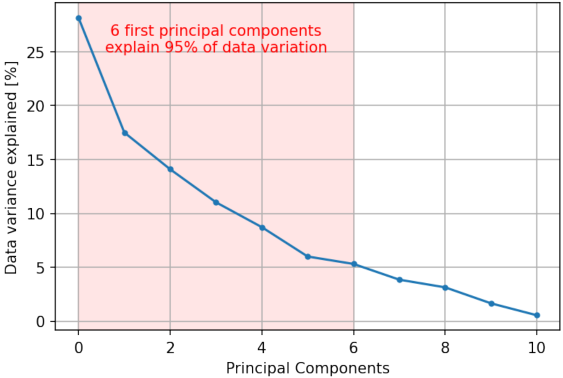
2.3. implementation steps

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?



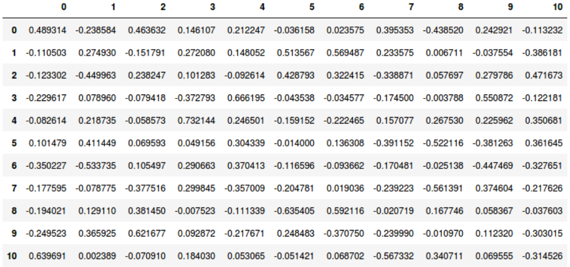
2.3. implementation steps

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?



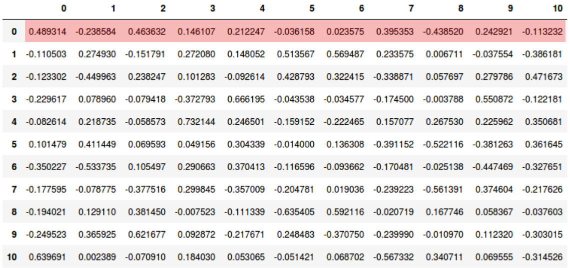
	0	1	2	3	4	5	6	7	8	9	10
0	0.489314	-0.238584	0.463632	0.146107	0.212247	-0.036158	0.023575	0.395353	-0.438520	0.242921	-0.113232
1	-0.110503	0.274930	-0.151791	0.272080	0.148052	0.513567	0.569487	0.233575	0.006711	-0.037554	-0.386181
2	-0.123302	-0.449963	0.238247	0.101283	-0.092614	0.428793	0.322415	-0.338871	0.057697	0.279786	0.471673
3	-0.229617	0.078960	-0.079418	-0.372793	0.666195	-0.043538	-0.034577	-0.174500	-0.003788	0.550872	-0.122181
4	-0.082614	0.218735	-0.058573	0.732144	0.246501	-0.159152	-0.222465	0.157077	0.267530	0.225962	0.350681
5	0.101479	0.411449	0.069593	0.049156	0.304339	-0.014000	0.136308	-0.391152	-0.522116	-0.381263	0.361645
6	-0.350227	-0.533735	0.105497	0.290663	0.370413	-0.116596	-0.093662	-0.170481	-0.025138	-0.447469	-0.327651
7	-0.177595	-0.078775	-0.377516	0.299845	-0.357009	-0.204781	0.019036	-0.239223	-0.561391	0.374604	-0.217626
8	-0.194021	0.129110	0.381450	-0.007523	-0.111339	-0.635405	0.592116	-0.020719	0.167746	0.058367	-0.037603
9	-0.249523	0.365925	0.621677	0.092872	-0.217671	0.248483	-0.370750	-0.239990	-0.010970	0.112320	-0.303015
10	0.639691	0.002389	-0.070910	0.184030	0.053065	-0.051421	0.068702	-0.567332	0.340711	0.069555	-0.314526

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?



	0	1	2	3	4	5	6	7	8	9	10
0	0.489314	-0.238584	0.463632	0.146107	0.212247	-0.036158	0.023575	0.395353	-0.438520	0.242921	-0.113232
1	-0.110503	0.274930	-0.151791	0.272080	0.148052	0.513567	0.569487	0.233575	0.006711	-0.037554	-0.386181
2	-0.123302	-0.449963	0.238247	0.101283	-0.092614	0.428793	0.322415	-0.338871	0.057697	0.279786	0.471673
3	-0.229617	0.078960	-0.079418	-0.372793	0.666195	-0.043538	-0.034577	-0.174500	-0.003788	0.550872	-0.122181
4	-0.082614	0.218735	-0.058573	0.732144	0.246501	-0.159152	-0.222465	0.157077	0.267530	0.225962	0.350681
5	0.101479	0.411449	0.069593	0.049156	0.304339	-0.014000	0.136308	-0.391152	-0.522116	-0.381263	0.361645
6	-0.350227	-0.533735	0.105497	0.290663	0.370413	-0.116596	-0.093662	-0.170481	-0.025138	-0.447469	-0.327651
7	-0.177595	-0.078775	-0.377516	0.299845	-0.357009	-0.204781	0.019036	-0.239223	-0.561391	0.374604	-0.217626
8	-0.194021	0.129110	0.381450	-0.007523	-0.111339	-0.635405	0.592116	-0.020719	0.167746	0.058367	-0.037603
9	-0.249523	0.365925	0.621677	0.092872	-0.217671	0.248483	-0.370750	-0.239990	-0.010970	0.112320	-0.303015
10	0.639691	0.002389	-0.070910	0.184030	0.053065	-0.051421	0.068702	-0.567332	0.340711	0.069555	-0.314526

Principal Component 1

PC 1 *= 0.49*feature0 + -0.24*feature1 + 0.46*feature2 + 0.15*feature3 + 0.21*feature4 + -0.04*feature5 + 0.02*feature6 + 0.40*feature7 + -0.44*feature8 + 0.24*feature9 + -0.11*feature10*

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?
Q3: How accurate is the prediction using all original 11 features, versus using only the e.g. 6 first principal components?

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?
Q3: How accurate is the prediction using all original 11 features, versus using only the e.g. 6 first principal components?

Prediction accuracy of wine quality (categorical variable ⇒ classification task using kNN):

- using 11 original features ⇒ accuracy $= 0.79$
- using 6 first principal components ⇒ accuracy $= 0.78$

2.3. implementation steps

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?
Q3: How accurate is the prediction using all original 11 features, versus using only the e.g. 6 first principal components?

Prediction accuracy of wine quality (categorical variable ⇒ classification task using kNN):

- using 11 original features ⇒ accuracy = 0.79
- using 6 first principal components ⇒ accuracy = 0.78

⇒ PCA can successfully reduce data dimensionality,
and achieve (almost) the same prediction accuracy with fewer features

Implementation steps (back to our toy-example on wine quality)

⇒ do the same with the 11 features: search for the principal components in a 11-dimensional space

NB: the maximum number of components is restricted by the number of features

Q1: How much data variance is explained by each principal component (eigenvector)?
Q2: How do the 11 eigenvectors (PCs) relate to the original feature space?
Q3: How accurate is the prediction using all original 11 features, versus using only the e.g. 6 first principal components?

Prediction accuracy of wine quality (categorical variable ⇒ classification task using kNN):

- using 11 original features ⇒ accuracy = 0.79
- using 6 first principal components ⇒ accuracy = 0.78

⇒ PCA can successfully reduce data dimensionality,
and achieve (almost) the same prediction accuracy with fewer features

⇒ how about using PCA on images?
→ Sentinel-2 exercise: a crop of 900 pixels (4-bands 15×15 pixels) can be reduced fairly accurately to 32 points! (i.e., projection in a 32-dimensional space, first 32 pcs)

1. Introduction
2. Principal Component Analysis (PCA)
3. Classification algorithms (perceptron + SVM)
    1. Perceptron
    2. Support Vector Machine (SVM)
4. Exercise

Classification algorithms (perceptron + SVM)

Once features have been extracted, we can feed to the classifier! (recall last week lecture)

⇒ the classification algorithm needs to learn the **decision boundary** *(i.e. surface separating the different classes)* in an N-dimensional **feature space**:

- *probabilistic approaches*:
  - **Logistic Regression** ⇒ *last week*
  - **Softmax Regression** ⇒ *last week*
  - **Naive Bayes**
- *non-probabilistic approaches*:
  - **k-Nearest Neighbors (kNN)** ⇒ *last week*
  - **Perceptron** ⇒ today!
    → algorithm finding a hyperplane to separate classes, adjusting weights based on misclassified points
  - **Support Vector Machines (SVM)** ⇒ today!
    → algorithm finding the optimal hyperplane that maximizes the margin between classes
  - **Random Forest** ⇒ *next lectures*
  - **Convolutional Neural Networks (CNNs)** ⇒ *next lectures*

Classification algorithms (perceptron + SVM)

Choosing a classifier

- **Logistic regression (& Softmax)** *(last week lecture)*

  ⇒ probability-based linear classification method
  ⇒ *advantage*: simple, fast, interpretable
  ⇒ *disadvantage*: limited to linear decision boundaries

- **k-Nearest Neighbor (kNN)** *(last week lecture)*

  ⇒ label images by comparing them to (annotated) images from the training set
  ⇒ *advantage*: non-linear decision boundaries
  ⇒ *disadvantage*: classifier needs to keep all training data for future comparisons with the test data *(classifying test images is expensive as it requires comparison to all training images, inefficient with v. large datasets* $\geq$*GB)*

- **Support Vector Machines** *(this week lecture)*

  ⇒ parametric linear classification method
  ⇒ *advantage*: once the parameters are learnt, training data can be discarded *(classification of new images is fast: simple matrix multiplication with learned weights, not an exhaustive comparison to every single training data)*

- **Convolutional Neural Networks** *(coming weeks)*

  ⇒ CNNs map image pixels to classes, but the mapping is more complex and will contain more parameters
  ⇒ *advantage*: very powerful
  ⇒ *disadvantage*: needs LOTS of data!

Classification algorithms (perceptron + SVM)

Choosing a classifier

- **Logistic regression (& Softmax)** *(last week lecture)*
  - ⇒ probability-based linear classification method
  - ⇒ *advantage*: simple, fast, interpretable
  - ⇒ *disadvantage*: limited to linear decision boundaries
- **k-Nearest Neighbor (kNN)** *(last week lecture)*
  - ⇒ label images by comparing them to (annotated) images from the training set
  - ⇒ *advantage*: non-linear decision boundaries
  - ⇒ *disadvantage*: classifier needs to keep all training data for future comparisons with the test data *(classifying test images is expensive as it requires comparison to all training images, inefficient with v. large datasets $\geq$GB)*
- **Support Vector Machines** *(this week lecture)*
  - ⇒ parametric linear classification method
  - ⇒ *advantage*: once the parameters are learnt, training data can be discarded *(classification of new images is fast: simple matrix multiplication with learned weights, not an exhaustive comparison to every single training data)*
- **Convolutional Neural Networks** *(coming weeks)*
  - ⇒ CNNs map image pixels to classes, but the mapping is more complex and will contain more parameters
  - ⇒ *advantage*: very powerful
  - ⇒ *disadvantage*: needs LOTS of data!

Classification algorithms (perceptron + SVM)

Choosing a classifier

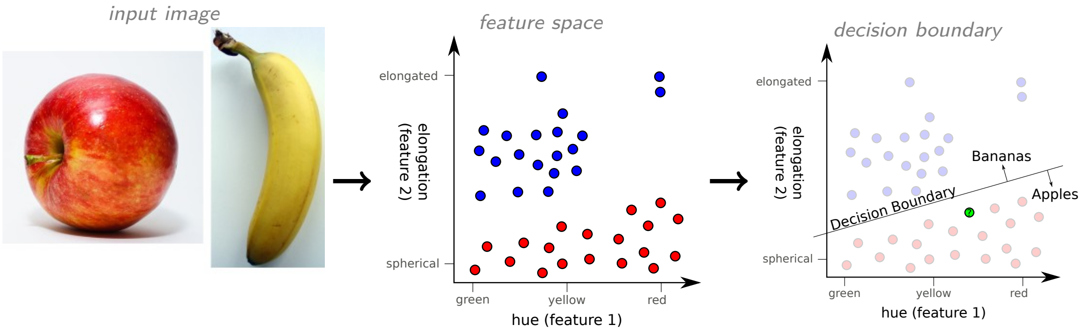
- **Logistic regression (& Softmax)** *(last week lecture)*
  - ⇒ probability-based linear classification method
  - ⇒ *advantage*: simple, fast, interpretable
  - ⇒ *disadvantage*: limited to linear decision boundaries
- **k-Nearest Neighbor (kNN)** *(last week lecture)*
  - ⇒ label images by comparing them to (annotated) images from the training set
  - ⇒ *advantage*: non-linear decision boundaries
  - ⇒ *disadvantage*: classifier needs to keep all training data for future comparisons with the test data *(classifying test images is expensive as it requires comparison to all training images, inefficient with v. large datasets $\geq$GB)*
- **Support Vector Machines** *(this week lecture)*
  - ⇒ parametric linear classification method
  - ⇒ *advantage*: once the parameters are learnt, training data can be discarded *(classification of new images is fast: simple matrix multiplication with learned weights, not an exhaustive comparison to every single training data)*
- **Convolutional Neural Networks** *(coming weeks)*
  - ⇒ CNNs map image pixels to classes, but the mapping is more complex and will contain more parameters
  - ⇒ *advantage*: very powerful
  - ⇒ *disadvantage*: needs LOTS of data!

Classification algorithms (perceptron + SVM)

Choosing a classifier

- **Logistic regression (& Softmax)** *(last week lecture)*
  - ⇒ probability-based linear classification method
  - ⇒ *advantage*: simple, fast, interpretable
  - ⇒ *disadvantage*: limited to linear decision boundaries
- **k-Nearest Neighbor (kNN)** *(last week lecture)*
  - ⇒ label images by comparing them to (annotated) images from the training set
  - ⇒ *advantage*: non-linear decision boundaries
  - ⇒ *disadvantage*: classifier needs to keep all training data for future comparisons with the test data *(classifying test images is expensive as it requires comparison to all training images, inefficient with v. large datasets ≥GB)*
- **Support Vector Machines** *(this week lecture)*
  - ⇒ parametric linear classification method
  - ⇒ *advantage*: once the parameters are learnt, training data can be discarded *(classification of new images is fast: simple matrix multiplication with learned weights, not an exhaustive comparison to every single training data)*
- **Convolutional Neural Networks** *(coming weeks)*
  - ⇒ CNNs map image pixels to classes, but the mapping is more complex and will contain more parameters
  - ⇒ *advantage*: very powerful
  - ⇒ *disadvantage*: needs LOTS of data!

Recall our toy example from last week: *classify fruit images into either bananas or apples*



⇒ *how is the decision boundary learned?*

Perceptron classifier

⇒ algorithm which classifies data based on linear decision boundary

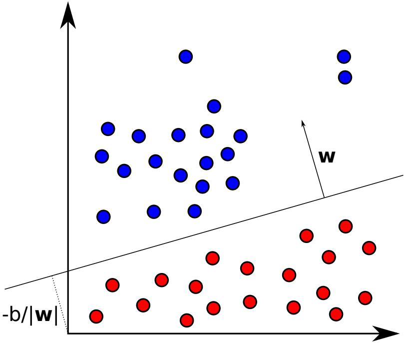
NB: the original perceptron algorithm is a binary classifier (similar to logistic regression but non-probabilistic)

NB: in an N-dimensional feature space, the decision boundary is a ***hyperplane***

⇒ **perceptron**:

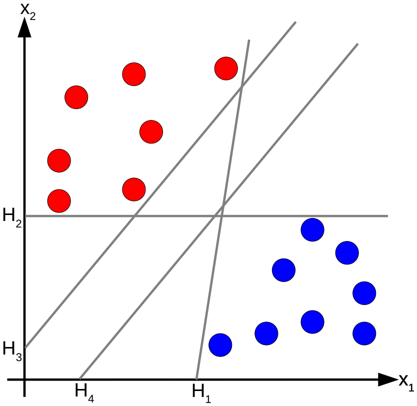
$$\boxed{\hat{y} = \text{sign}(\mathbf{w}^T x + \mathbf{b})}$$

- $\hat{y} \in \{-1, 1\}$: predicted class → *banana or apple*
- $x \in \mathbb{R}^2$: feature vector → *hue, elongation*
- $\mathbf{w} \in \mathbb{R}^2$: **weight vector** → needs to be learned
- $\mathbf{b} \in \mathbb{R}$: **bias** → needs to be learned
- *sign*: sign function returning the sign of a real number



Perceptron classifier

- ⇒ **perceptron**: $\boxed{\hat{y} = \text{sign}(\mathbf{w}^T x + \mathbf{b})}$
- ⇒ problem: multiple "good" boundaries can be found
- ⇒ need to find the *optimal hyperplane*
  - = boundary with maximal margins
  - = perceptron of maximal stability to new inputs

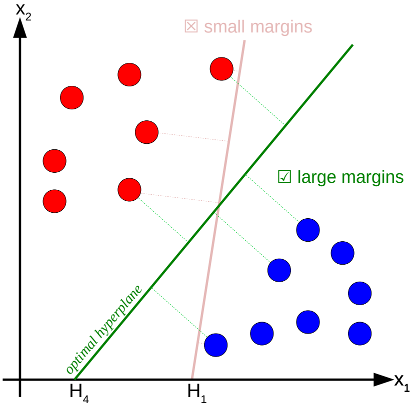


Perceptron classifier

⇒ **perceptron**: $\boxed{\hat{y} = \mathrm{sign}(\mathbf{w}^T x + \mathbf{b})}$

⇒ problem: multiple “good” boundaries can be found

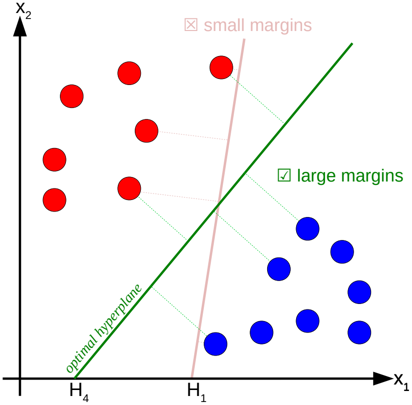
⇒ need to find the *optimal hyperplane*
= boundary with maximal margins
= perceptron of maximal stability to new inputs



Perceptron classifier

- ⇒ **perceptron**: $\boxed{\hat{y} = \mathrm{sign}(\mathbf{w}^T x + \mathbf{b})}$
- ⇒ problem: multiple "good" boundaries can be found
- ⇒ need to find the *optimal hyperplane*
  - = boundary with maximal margins
  - = perceptron of maximal stability to new inputs

> ⇒ the **Support Vector Machine** (SVM) algorithm will find the *optimal hyperplane* and learn the best *weights* $\mathbf{w}$ and *bias* $\mathbf{b}$ to classify the data

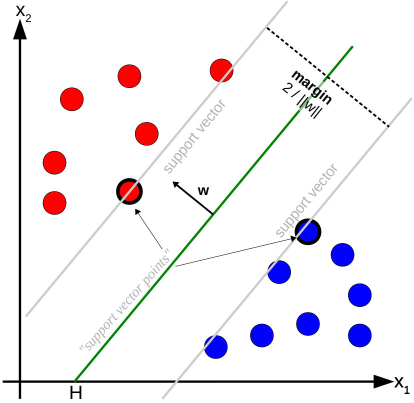


Support Vector Machine (SVM)

⇒ **perceptron**: $\boxed{\hat{y} = \text{sign}(\mathbf{w}^T x + \mathbf{b})}$

⇒ definitions:

- support vector points = points closest to the hyperplane
  (only these points are contributing to the result, other points are not)
- margin = distance between hyperplane & support vector points
  $= \frac{2}{||w||}$



⇒ maximize margin:

$$\max_{w} \frac{2}{||w||}, \text{ subject to } \begin{cases} w^T x_i + b \geq 1 & \text{if } y_i = +1 \\ w^T x_i + b \leq 1 & \text{if } y_i = -1 \end{cases}$$

which is equivalent to:

$$\min_{w} ||w||^2, \text{ subject to } y_i(w^T x_i - b) \geq 1$$

Support Vector Machine (SVM)

⇒ **perceptron**: $\boxed{\hat{y} = \text{sign}(\mathbf{w}^T x + \mathbf{b})}$

⇒ definitions:

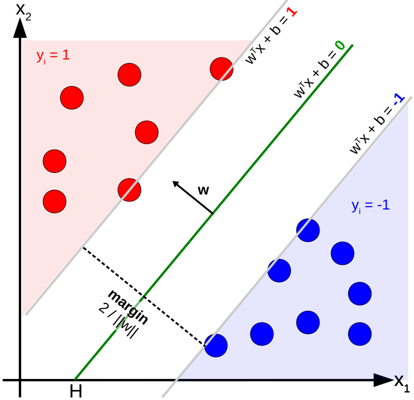
- support vector points = points closest to the hyperplane
  (only these points are contributing to the result, other points are not)
- margin = distance between hyperplane & support vector points
  $= \frac{2}{||w||}$

⇒ maximize margin:

$$\max_{w} \frac{2}{||w||}, \text{ subject to } \begin{cases} w^T x_i + b \geq 1 & \text{if } y_i = +1 \\ w^T x_i + b \leq 1 & \text{if } y_i = -1 \end{cases}$$

which is equivalent to:

$$\min_{w} ||w||^2, \text{ subject to } y_i(w^T x_i - b) \geq 1$$



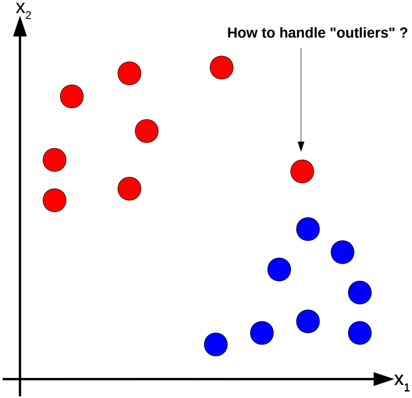
Support Vector Machine (SVM)

⇒ **How can outliers be handled?**

- ⇒ is a hard-margin with 100% accuracy good?
- ⇒ no, allow small errors (**soft-margin**) to favour overall better model
- ⇒ tolerate margin violation & favour large margin boundaries
- ⇒ optimization becomes:

$$\min_{w,\xi_i} ||w||^2 + C\sum_{i}^{N}\xi_i, \text{ subject to } y_i(w^Tx_i - b) \geq 1 - \xi_i$$

$$\text{where: } \begin{cases} C & \text{regularization parameter} \\ & \text{- small C} \Rightarrow \text{constraints easily ignored, large margin} \\ & \text{- large C} \Rightarrow \text{towards hard-margin SVM} \\ \xi_i & \text{slack variable for each data point} \end{cases}$$



Support Vector Machine (SVM)

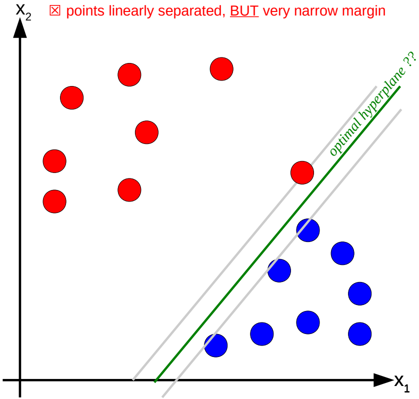
⇒ **How can outliers be handled?**

- ⇒ is a hard-margin with 100% accuracy good?
- ⇒ no, allow small errors (**soft-margin**) to favour overall better model
- ⇒ tolerate margin violation & favour large margin boundaries
- ⇒ optimization becomes:

$$\min_{w,\xi_i} ||w||^2 + C\sum_{i}^{N} \xi_i, \text{ subject to } y_i(w^T x_i - b) \geq 1 - \xi_i$$

where:
C regularization parameter
- small C ⇒ constraints easily ignored, large margin
- large C ⇒ towards hard-margin SVM

ξ_i slack variable for each data point


☒ points linearly separated, BUT very narrow margin

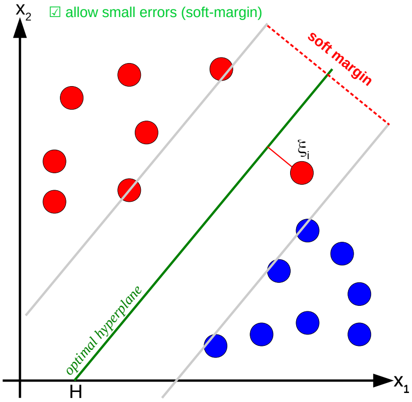
Support Vector Machine (SVM)

⇒ **How can outliers be handled?**

- ⇒ is a hard-margin with 100% accuracy good?
- ⇒ no, allow small errors (**soft-margin**) to favour overall better model
- ⇒ tolerate margin violation & favour large margin boundaries
- ⇒ optimization becomes:

$$\min_{w,\xi_i} ||w||^2 + C\sum_i^N \xi_i, \text{ subject to } y_i(w^T x_i - b) \geq 1 - \xi_i$$

$$\text{where: } \begin{cases} C & \text{regularization parameter} \\ & \text{- small C} \Rightarrow \text{constraints easily ignored, large margin} \\ & \text{- large C} \Rightarrow \text{towards hard-margin SVM} \\ \xi_i & \text{slack variable for each data point} \end{cases}$$



Support Vector Machine (SVM)

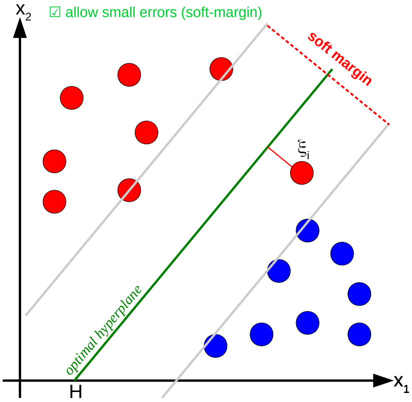
⇒ **How can outliers be handled?**

- ⇒ is a hard-margin with 100% accuracy good?
- ⇒ no, allow small errors (**soft-margin**) to favour overall better model
- ⇒ tolerate margin violation & favour large margin boundaries
- ⇒ optimization becomes:

$$\min_{w,\xi_i} ||w||^2 + C\sum_{i}^{N}\xi_i, \text{ subject to } y_i(w^T x_i - b) \geq 1 - \xi_i$$

where:
C regularization parameter
- small C ⇒ constraints easily ignored, large margin
- large C ⇒ towards hard-margin SVM

ξ_i slack variable for each data point



Support Vector Machine (SVM)

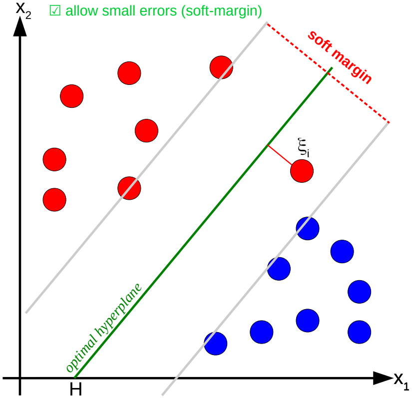
⇒ **How can outliers be handled?**

- ⇒ is a hard-margin with 100% accuracy good?
- ⇒ no, allow small errors (**soft-margin**) to favour overall better model
- ⇒ tolerate margin violation & favour large margin boundaries
- ⇒ optimization becomes:

$$\min_{w,\xi_i} ||w||^2 + C \sum_{i}^{N} \xi_i, \text{ subject to } y_i(w^T x_i - b) \geq 1 - \xi_i$$

where: C regularization parameter
- small C ⇒ constraints easily ignored, large margin
- large C ⇒ towards hard-margin SVM

ξ_i slack variable for each data point



3.2. Support Vector Machine (SVM)

Side note: *reformulating optimization in terms of* ***regularization*** *and* ***loss function*** *(anticipating DL lectures)*

Learning an SVM has been formulated as a constrained optimization problem over w and ξ:

$$\min_{w,\xi_i} ||w||^2 + C \sum_i^N \xi_i \qquad \text{subject to:} \quad y_i(w^T x_i - b) \geq 1 - \xi_i$$

The constraint $y_i(w^T x_i - b) \geq 1 - \xi_i$ can be written more concisely as: $y_i f(x_i) \geq 1 - \xi_i$

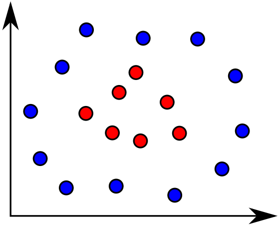
Together with $\xi_i > 0$, it is equivalent to: $\xi_i = max(0, 1 - y_i f(x_i))$

Hence the learning problem is equivalent to the unconstrained optimization problem over w:

$$\min_w \underbrace{||w||^2}_{regularization} + C \sum_i^N \underbrace{max(0,\ 1 - y_i f(x_i))}_{loss\ function\ (Hinge\ loss)}$$

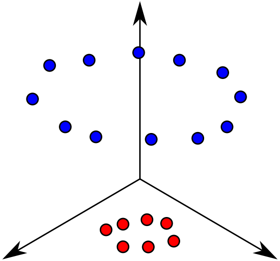
Support Vector Machine (SVM)

- **What if the features x_i are not linearly separable**?



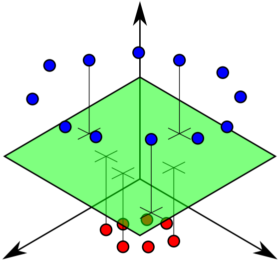
Support Vector Machine (SVM)

- **What if the features x_i are not linearly separable**?
  $\Rightarrow$ compute new features $x_i \mapsto \phi(x)$
  $\phi(x)$ is a **feature map**, mapping x to $\phi(x)$ where data is separable



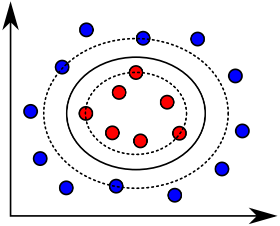
Support Vector Machine (SVM)

- **What if the features x_i are not linearly separable**?
  $\Rightarrow$ compute new features $x_i \mapsto \phi(x)$
  $\phi(x)$ is a **feature map**, mapping x to $\phi(x)$ where data is separable
  $\Rightarrow$ solve for $\mathbf{w}$ in high dimensional feature space



Support Vector Machine (SVM)

- **What if the features x_i are not linearly separable**?
  $\Rightarrow$ compute new features $x_i \mapsto \phi(x)$
  $\phi(x)$ is a **feature map**, mapping x to $\phi(x)$ where data is separable
  $\Rightarrow$ solve for $\mathbf{w}$ in high dimensional feature space
  $\Rightarrow$ data not lineary-seperable in original feature space become separable



3.2. Support Vector Machine (SVM)

Kernel trick

The Representer Theorem states that the solution **w** can be written as a linear combination of the training data:

$$w = \sum_{j=1}^{N} \alpha_j y_j x$$

3.2. Support Vector Machine (SVM)

Kernel trick

The Representer Theorem states that the solution **w** can be written as a linear combination of the training data:

$$w = \sum_{j=1}^{N} \alpha_j y_j x$$

The linear classifier can therefore be reformulated as:

$$\begin{aligned} f(x) &= w^T x + b \\ &= \sum_{i}^{N} \alpha_i y_i (x_i^T x) + b \end{aligned}$$

3.2. Support Vector Machine (SVM)

Kernel trick

The Representer Theorem states that the solution **w** can be written as a linear combination of the training data:

$$w = \sum_{j=1}^{N} \alpha_j y_j x$$

The linear classifier can therefore be reformulated as:

$$\begin{aligned} f(x) &= w^T x + b \\ &= \sum_{i}^{N} \alpha_i y_i (x_i^T x) + b \end{aligned}$$

NB: this reformulation seems to have the disadvantage of a kNN classifier, i.e. requires the training data points x_i.

However, many of the $\alpha_i = 0$: the ones that are non-zero define the support vector points x_i

3.2. Support Vector Machine (SVM)

Kernel trick

The Representer Theorem states that the solution **w** can be written as a linear combination of the training data:

$$w = \sum_{j=1}^{N} \alpha_j y_j x$$

The linear classifier can therefore be reformulated as:

$$\begin{aligned} f(x) &= w^T x + b \\ &= \sum_{i}^{N} \alpha_i y_i (x_i^T x) + b \end{aligned}$$

NB: this reformulation seems to have the disadvantage of a kNN classifier, i.e. requires the training data points x_i. However, many of the $\alpha_i = 0$: the ones that are non-zero define the support vector points x_i

Using the feature map $\phi(x)$, it can be reformulated as:

$$\begin{aligned} f(x) &= \sum_{i}^{N} \alpha_i y_i (\phi(x_i)^T \phi(x)) + b \\ &= \sum_{i}^{N} \alpha_i y_i k(x_i, x) + b \end{aligned}$$

where $k(x_i, x)$ is known as a **Kernel**

Kernel trick

- Classifier can be learnt and applied without explicitly computing $\phi(x)$
- All that is required is the kernel $k(x, x')$
- Multiple kernels exist:
  - linear kernels: $k(x, x') = x^T x'$
    → *very fast and easy to train, but very simple*
  - polynomial kernels: $k(x, x') = (1 + x^T x')^d$
    → *contains all polynomial terms up to degree d*
  - gaussian kernels: $k(x, x') = exp(-||x - x'||^2 / 2\sigma^2)$ *(RBF kernel)*
    → *kernel very powerful and most often used*

3.2. Support Vector Machine (SVM)

Kernel trick

- Classifier can be learnt and applied without explicitly computing $\phi(x)$
- All that is required is the kernel $k(x, x')$
- Multiple kernels exist:
  - linear kernels: $k(x, x') = x^T x'$
    → very fast and easy to train, but very simple
  - polynomial kernels: $k(x, x') = (1 + x^T x')^d$
    → contains all polynomial terms up to degree d
  - gaussian kernels: $k(x, x') = exp(-||x - x'||^2/2\sigma^2)$ *(RBF kernel)*
    → kernel very powerful and most often used

3.2. Support Vector Machine (SVM)

Kernel trick

- Classifier can be learnt and applied without explicitly computing $\phi(x)$
- All that is required is the kernel $k(x, x')$
- Multiple kernels exist:
  - linear kernels: $k(x, x') = x^T x'$
    → *very fast and easy to train, but very simple*
  - polynomial kernels: $k(x, x') = (1 + x^T x')^d$
    → *contains all polynomial terms up to degree d*
  - gaussian kernels: $k(x, x') = exp(-||x - x'||^2 / 2\sigma^2)$ *(RBF kernel)*
    → *kernel very powerful and most often used*

1. Introduction
2. Principal Component Analysis (PCA)
3. Classification algorithms (perceptron + SVM)
4. Exercise

Exercise

EXERCISE:

classify land-use in satellite images (Sentinel-2) using PCA and SVM

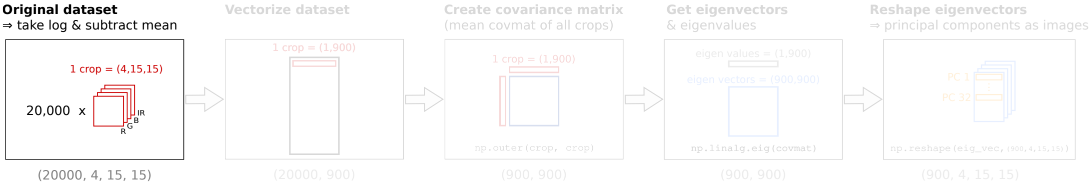
PCA dimensionality reduction

train SVM & apply

land-use classification

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

(20000, 4, 15, 15)

Vectorize dataset

(20000, 900)

Create covariance matrix
(mean covmat of all crops)

`np.outer(crop, crop)`

(900, 900)

Get eigenvectors
& eigenvalues

`np.linalg.eig(covmat)`

(900, 900)

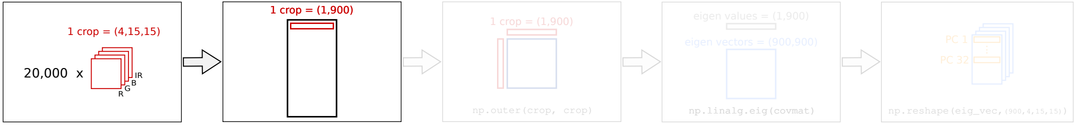
Reshape eigenvectors
⇒ principal components as images

`np.reshape(eig_vec,(900,4,15,15))`

(900, 4, 15, 15)

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean
(20000, 4, 15, 15)

Vectorize dataset
(20000, 900)

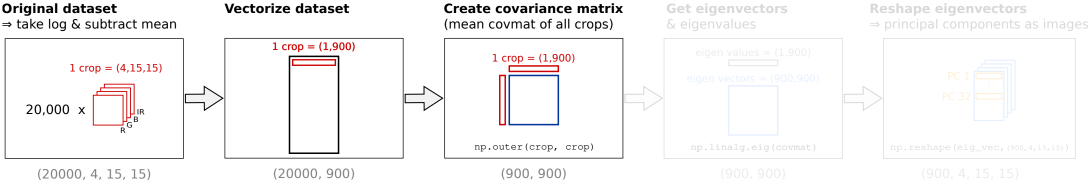
Create covariance matrix
(mean covmat of all crops)
(900, 900)

Get eigenvectors
& eigenvalues
(900, 900)

Reshape eigenvectors
⇒ principal components as images
(900, 4, 15, 15)

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

(20000, 4, 15, 15)

Vectorize dataset

(20000, 900)

Create covariance matrix
(mean covmat of all crops)

`np.outer(crop, crop)`

(900, 900)

Get eigenvectors
& eigenvalues

`np.linalg.eig(covmat)`

(900, 900)

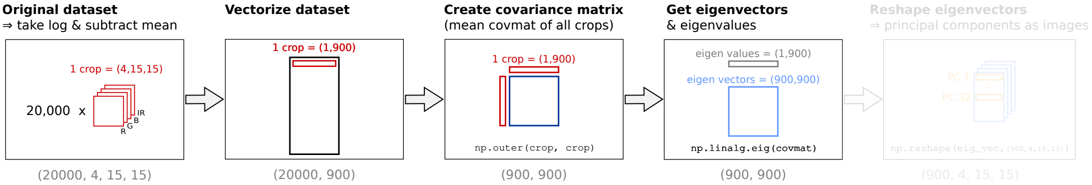
Reshape eigenvectors
⇒ principal components as images

`np.reshape(eig_vec, (900,4,15,15))`

(900, 4, 15, 15)

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

1 crop = (4,15,15)

20,000 x

IR

B

G

R

`(20000, 4, 15, 15)`

Vectorize dataset

1 crop = (1,900)

`(20000, 900)`

Create covariance matrix
(mean covmat of all crops)

1 crop = (1,900)

`np.outer(crop, crop)`

`(900, 900)`

Get eigenvectors
& eigenvalues

eigen values = (1,900)

eigen vectors = (900,900)

`np.linalg.eig(covmat)`

`(900, 900)`

Reshape eigenvectors
⇒ principal components as images

PC 1

⋮

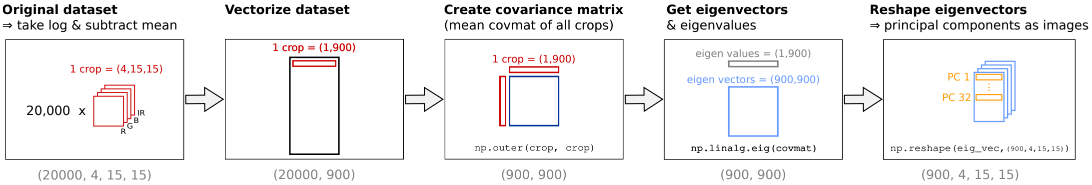
PC 32

`np.reshape(eig_vec,(900,4,15,15))`

`(900, 4, 15, 15)`

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

20,000 x 1 crop = (4,15,15)

(20000, 4, 15, 15)

Vectorize dataset

1 crop = (1,900)

(20000, 900)

Create covariance matrix
(mean covmat of all crops)

1 crop = (1,900)

`np.outer(crop, crop)`

(900, 900)

Get eigenvectors
& eigenvalues

eigen values = (1,900)

eigen vectors = (900,900)

`np.linalg.eig(covmat)`

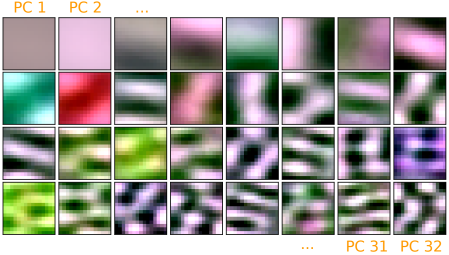
(900, 900)

Reshape eigenvectors
⇒ principal components as images

PC 1
⋮
PC 32

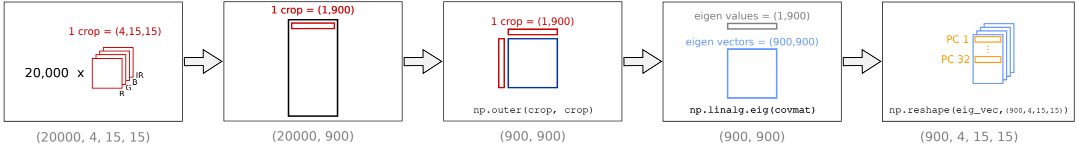
`np.reshape(eig_vec, (900,4,15,15))`

(900, 4, 15, 15)



Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

(20000, 4, 15, 15)

Vectorize dataset

(20000, 900)

Create covariance matrix
(mean covmat of all crops)

`np.outer(crop, crop)`

(900, 900)

Get eigenvectors
& eigenvalues

`np.linalg.eig(covmat)`

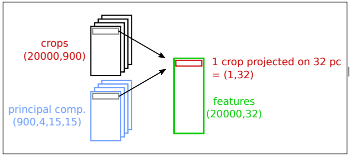
(900, 900)

Reshape eigenvectors
⇒ principal components as images

`np.reshape(eig_vec, (900,4,15,15))`

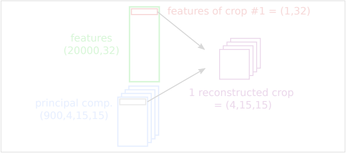
(900, 4, 15, 15)

Compute features
⇒ project each crop on first 32 pc



```
for i in range(20000):   #loop crops
   for j in range(32):   #loop pcs
      features[i,j] = np.dot(crop[i,:::], pc[j,:::])
                    = (900,1) @ (1,900)
                    = (scalar)
```

Reconstruct crops
⇒ reconstruct crop from its 32 features & 32 first pcs

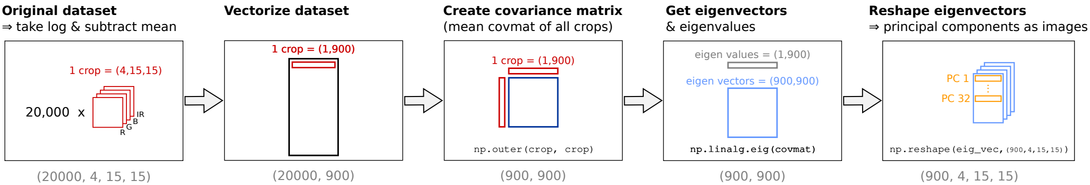


Reconstruction crop #1:

```
reconstruction = mean_crops   #(4, 15, 15)
for i in range(32):           #loop crop features/pcs
   reconstruction += features[0,i] * pc[i,:::]
                   = (scalar) * (4,15,15)
                   = (4,5,15)
```

Exercise

Part 1: apply PCA on satellite image crops



Original dataset
⇒ take log & subtract mean

1 crop = (4,15,15)

20,000 x

IR B G R

(20000, 4, 15, 15)

Vectorize dataset

1 crop = (1,900)

(20000, 900)

Create covariance matrix
(mean covmat of all crops)

1 crop = (1,900)

`np.outer(crop, crop)`

(900, 900)

Get eigenvectors
& eigenvalues

eigen values = (1,900)

eigen vectors = (900,900)

`np.linalg.eig(covmat)`

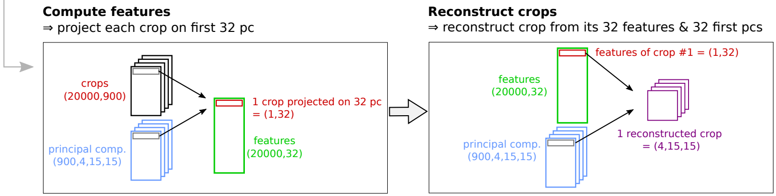
(900, 900)

Reshape eigenvectors
⇒ principal components as images

PC 1
⋮
PC 32

`np.reshape(eig_vec,(900,4,15,15))`

(900, 4, 15, 15)



Compute features
⇒ project each crop on first 32 pc

crops
(20000,900)

principal comp.
(900,4,15,15)

1 crop projected on 32 pc
= (1,32)

features
(20000,32)

```
for i in range(20000):    #loop crops
   for j in range(32):    #loop pcs
      features[i,j] = np.dot(crop[i,:::], pc[j,:::])
                    = (900,1) @ (1,900)
                    = (scalar)
```

Reconstruct crops
⇒ reconstruct crop from its 32 features & 32 first pcs

features of crop #1 = (1,32)

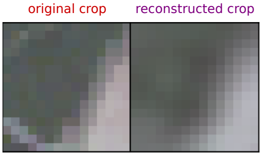
features
(20000,32)

principal comp.
(900,4,15,15)

1 reconstructed crop
= (4,15,15)

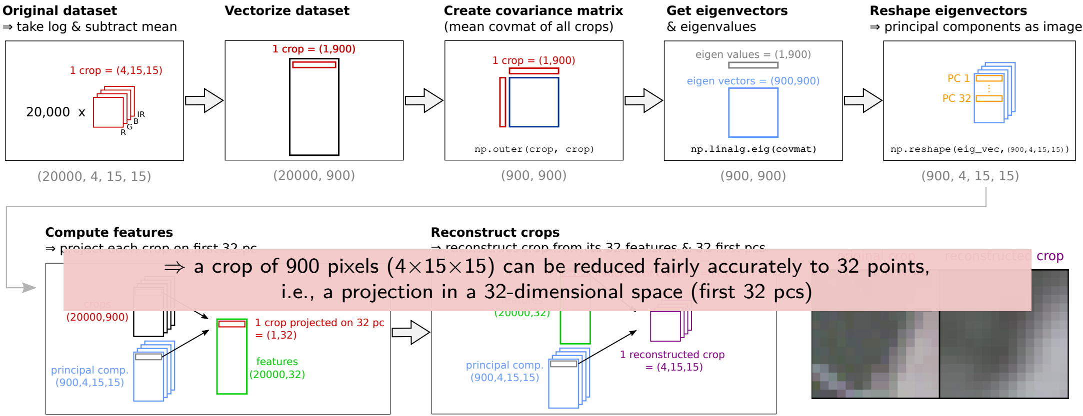
Reconstruction crop #1:

```
reconstruction = mean_crops   #(4, 15, 15)
for i in range(32):           #loop crop features/pcs
   reconstruction += features[0,i] * pc[i,:::]
                   = (scalar) * (4,15,15)
                   = (4,5,15)
```



Exercise

Part 1: apply PCA on satellite image crops



⇒ a crop of 900 pixels (4×15×15) can be reduced fairly accurately to 32 points,
i.e., a projection in a 32-dimensional space (first 32 pcs)

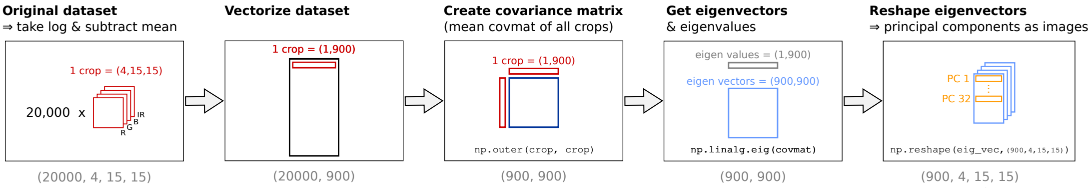
```
for i in range(20000):   #loop crops
   for j in range(32):   #loop pcs
      features[i,j] = np.dot(crop[i,:::], pc[j,:::])
                    = (900,1) @ (1,900)
                    = (scalar)
```

Reconstruction crop #1:

```
reconstruction = mean_crops    #(4, 15, 15)
for i in range(32):            #loop crop features/pcs
   reconstruction += features[0,i] * pc[i,:::]
                   = (scalar) * (4,15,15)
                   = (4,5,15)
```

Exercise

Part 1: apply PCA on satellite image crops



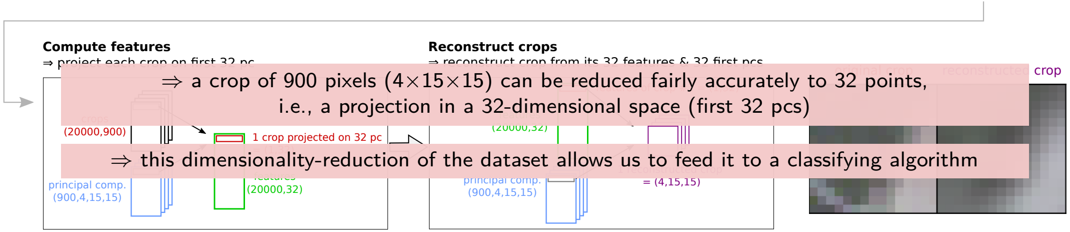
Original dataset
⇒ take log & subtract mean

Vectorize dataset

Create covariance matrix
(mean covmat of all crops)

Get eigenvectors
& eigenvalues

Reshape eigenvectors
⇒ principal components as images



Compute features
⇒ project each crop on first 32 pc

Reconstruct crops
⇒ reconstruct crop from its 32 features & 32 first pcs

⇒ a crop of 900 pixels (4×15×15) can be reduced fairly accurately to 32 points,
i.e., a projection in a 32-dimensional space (first 32 pcs)

⇒ this dimensionality-reduction of the dataset allows us to feed it to a classifying algorithm

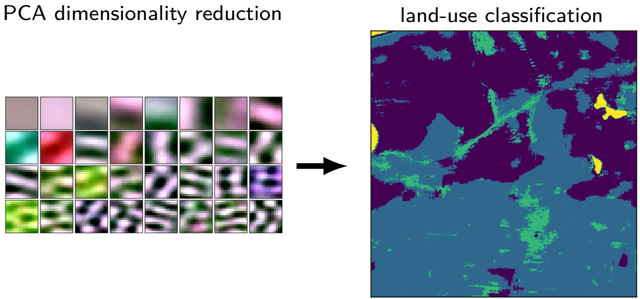
```
for i in range(20000):   #loop crops
   for j in range(32):   #loop pcs
      features[i,j] = np.dot(crop[i,:::], pc[j,:::])
                    = (900,1) @ (1,900)
                    = (scalar)
```

Reconstruction crop #1:

```
reconstruction = mean_crops   #(4, 15, 15)
for i in range(32):           #loop crop features/pcs
   reconstruction += features[0,i] * pc[i,:::]
                   = (scalar) * (4,15,15)
                   = (4,5,15)
```

Exercise

Part 2: train SVM on principal components and apply to classify full image



PCA dimensionality reduction

land-use classification