

# Methods 3: Multilevel Statistical Modeling and Machine Learning

Week 10: *Organising and preprocessing messy data*

November 30, 2021

*by:* Lau Møller Andersen

These slides are distributed according to the CC  
BY 4.0 licence:

<https://creativecommons.org/licenses/by/4.0/>



**Attribution 4.0 International (CC BY 4.0)**

**REMEMBER:** Corona-passport is a requirement  
now

(I'm not going to check you)

# Santa instructions?



<https://bornibyen.dk/aarhus/articles/sjov-udendoers-kunst-i-aarhus>



December 21<sup>st</sup>  
(11-13)  
Too late?

Silkeborgvej 41E, baghuset 1.th, 8000 Aarhus C.

Sign-up list here:

<https://cryptpad.fr/pad/#/2/pad/edit/NHEOwwpF2nH8vTGDNPuQZENx>



# Follow-up (PCA): How to interpret $W$ ?

```
print('Weight matrix:\n', w)
```

```
Weight matrix:  
[[ 0.14669811  0.50417079]  
 [-0.24224554  0.24216889]  
 [-0.02993442  0.28698484]  
 [-0.25519002 -0.06468718]  
 [ 0.12079772  0.22995385]  
 [ 0.38934455  0.09363991]  
 [ 0.42326486  0.01088622]  
 [-0.30634956  0.01870216]  
 [ 0.30572219  0.03040352]  
 [-0.09869191  0.54527081]  
 [ 0.30032535 -0.27924322]  
 [ 0.36821154 -0.174365 ]  
 [ 0.29259713  0.36315461]]
```

$$\mathbf{Z} = \mathbf{XW}$$

$\mathbf{Z}_{n \times k}$ : Underlying generator of data

$\mathbf{X}_{n \times d}$ : Observed data (mix of the generators)

$\mathbf{W}_{d \times k}$ : (inverse) weighting matrix (bringing us from data to generator)  
Brings us from what we observe back the original generator

$n$ : number of observations rows

$d$ : dimensions in observed data (  $n$  predictor variables ) Text

$k$ : number of dimensions kept

$d - k$ : number of dimensions projected out dimensions not part of generator

Let us derive the **forward** weighting matrix,  $W_{d \times d}$ , which brings us from generator,  $Z$ , to observed data,  $X$

Text

$$X = ?$$

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

$$\mathbf{Z}^T = (\mathbf{X}\mathbf{W})^T = \mathbf{W}^T \mathbf{X}^T$$

$$(\mathbf{W}^T)^{-1} \mathbf{Z}^T = \mathbf{X}^T$$

$$\mathbf{W}\mathbf{Z}^T = \mathbf{X}^T$$

for orthonormal vectors:  $(\mathbf{W}^T)^{-1} = \mathbf{W}$

for orthonormal vector, the inverse of a vector is transposed.

Or expressed based on  $X$

$$X^T = WZ^T$$

$$X = X^{TT} = (WZ^T)^T$$

$$X = ZW^T$$

$W^T$ : forward weighting matrix: bringing <sup>Text</sup> us from generator to data

$$\mathbf{Z} = \mathbf{X}\mathbf{W} \quad \mathbf{X} = \mathbf{Z}\mathbf{W}^T$$

$\mathbf{W}^T$ : forward weighting matrix (or mixing matrix); from generator to data  
 $\mathbf{W}$ : inverse weighting matrix (or unmixing matrix); from data to generator

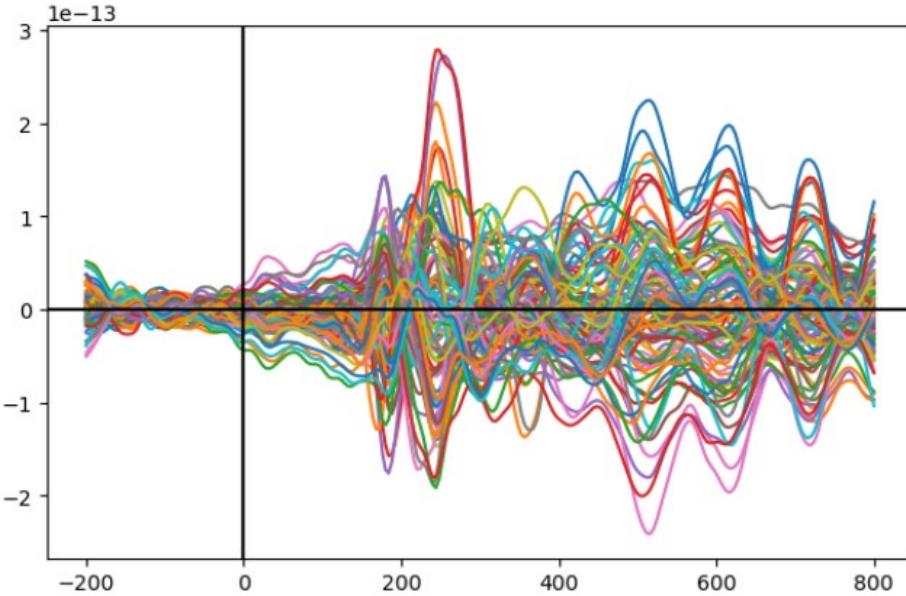
mixing, because it mixes the underlying sources

Text

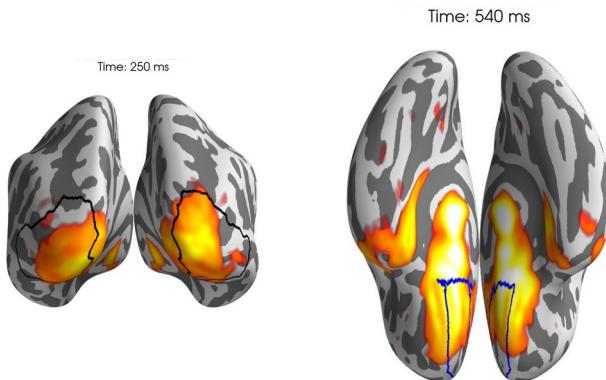
**W** matrices are involved in many cases where we make measurements, **X**, on something that is generated by a plethora of sources, **Z**

e.g magneto- and electroencephalography

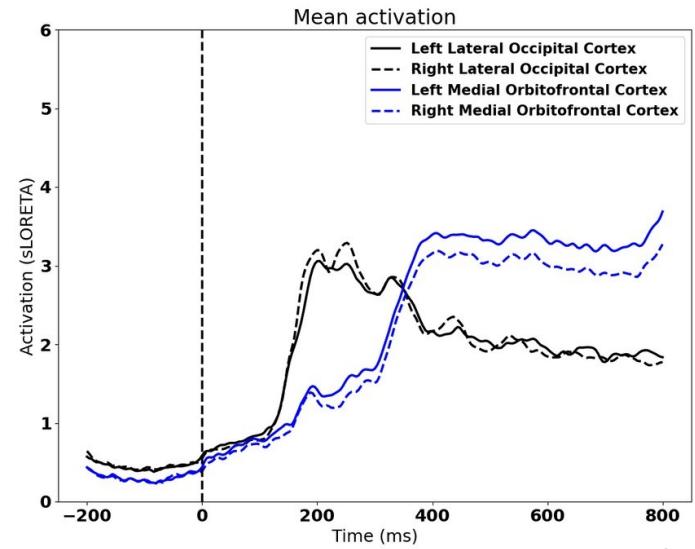
**X**



**Z**

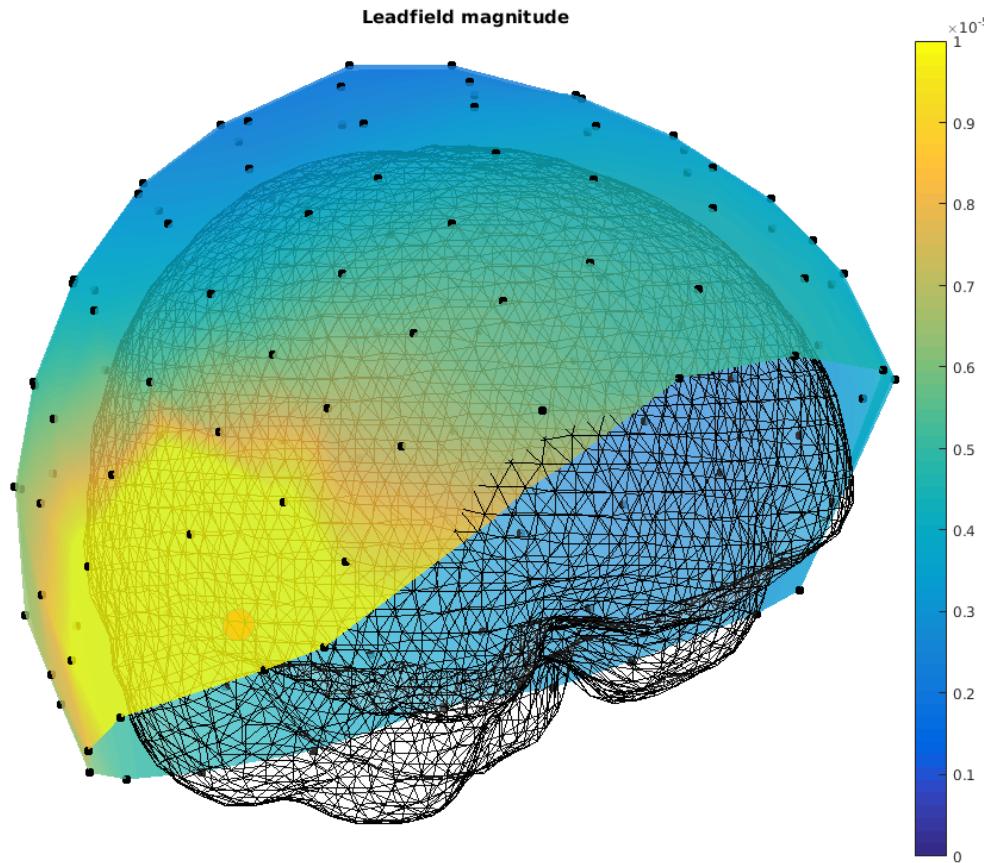


$$X = ZW^T$$



$$X = ZW^T$$

$$W^T$$



The so-called forward model,  $W^T$ , models how each source,  $Z$ , is seen by the sensors,  $X$ , when it is active

# **COURSE EVALUATION**

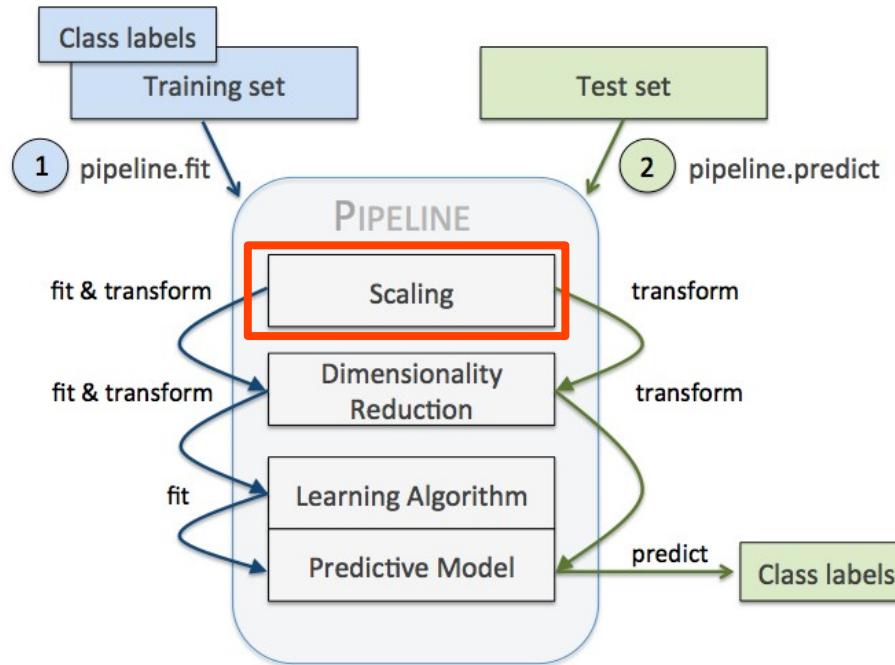
you should have received an email  
we'll use ~15 minutes on it

# Learning goals

*Organising and preprocessing messy data*

- 1) Learning why data should be scaled
- 2) Using feature selection to avoid overfitting
- 3) Understanding the basic steps of a machine learning pipeline

# Pipeline example



(p. 172: Raschka, 2015)

# Scaling

definitions of normalization and standardization is pretty interchangeable

- Normalisation

min-max normalization

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

$x_{max}$ : maximum value for the feature:  $x$

$x_{min}$ : minimum value for the feature:  $x$

Brings everything on to  
the scale: [0, 1]

- Standardisation

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

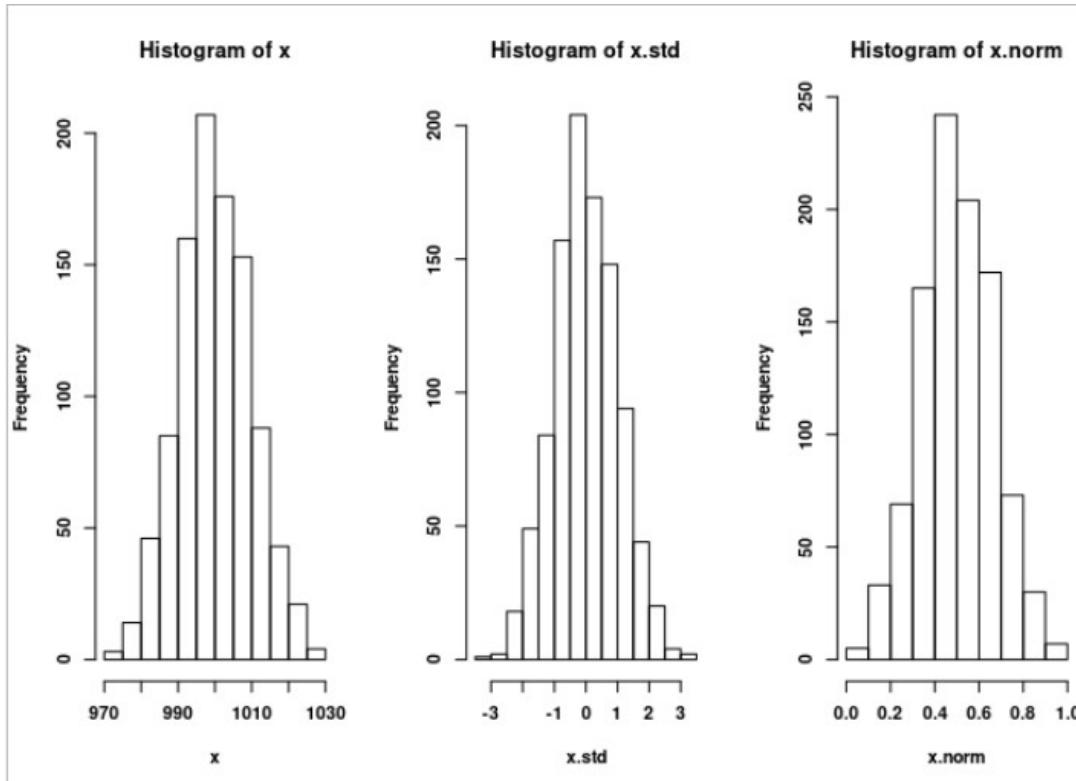
$\mu_x$ : sample mean for the feature:  $x$

$\sigma_x$ : sample standard deviation for the feature:  $x$

Text

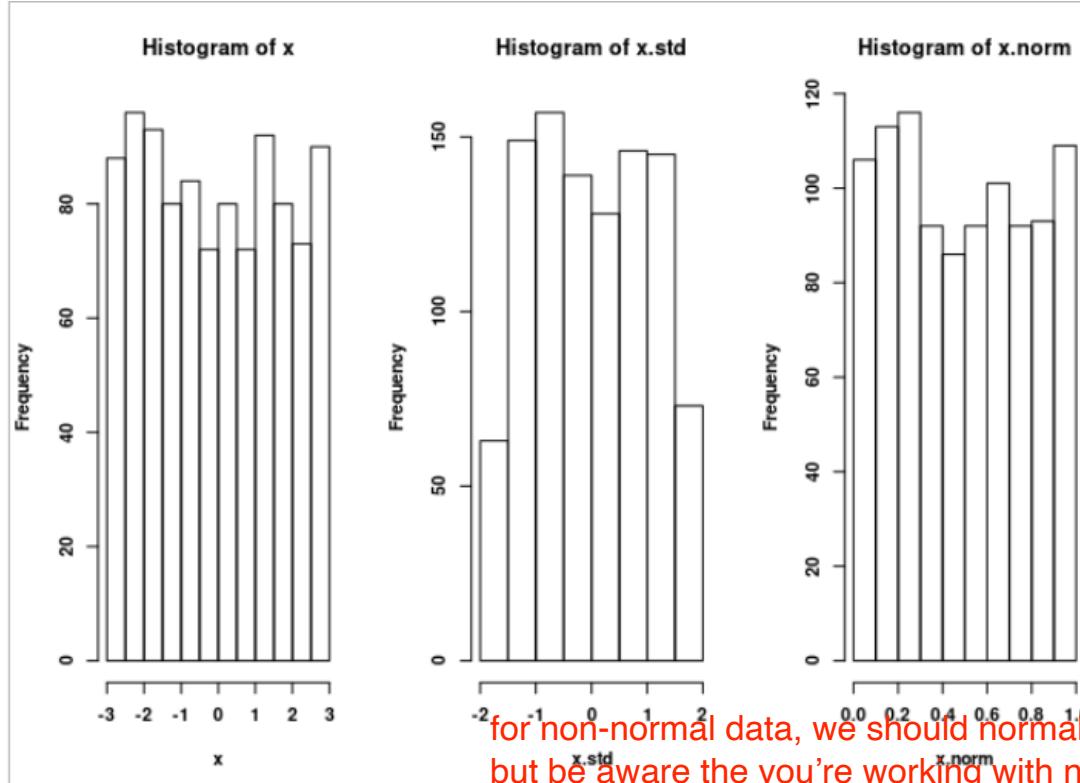
Brings data onto a  
normal distribution with  
with  $\mu=0$  and  $\sigma=1$

# Normal data



$$\mu = 0; \sigma = 1$$

# Non-Normal data



for non-normal data, we should normalize it instead of standardise, but be aware you're working with non-normal data.  
as a rule of thumb, use normalization.

$$\mu = 0; \sigma = 1$$

<b>input</b>	<b>standardized</b>	<b>normalized</b>
0.0	-1.336306	0.0
1.0	-0.801784	0.2
2.0	-0.267261	0.4
3.0	0.267261	0.6
4.0	0.801784	0.8
5.0	1.336306	1.0

(p. 111: Raschka, 2015)

min/max normalization

```
from sklearn.preprocessing import MinMaxScaler  
mms = MinMaxScaler()  
X_train_norm = mms.fit_transform(X_train)  
X_test_norm = mms.transform(X_test)
```

standardisation

```
from sklearn.preprocessing import StandardScaler  
stdsc = StandardScaler()  
X_train_std = stdsc.fit_transform(X_train)  
X_test_std = stdsc.transform(X_test)
```

# Wine dataset

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

(p. 109: Raschka, 2015)

Original data:

```
[[1.371e+01 1.860e+00 2.360e+00 ... 1.110e+00 4.000e+00 1.035e+03]
 [1.222e+01 1.290e+00 1.940e+00 ... 8.600e-01 3.020e+00 3.120e+02]
 [1.327e+01 4.280e+00 2.260e+00 ... 5.900e-01 1.560e+00 8.350e+02]
 ...
 [1.242e+01 1.610e+00 2.190e+00 ... 1.060e+00 2.960e+00 3.450e+02]
 [1.390e+01 1.680e+00 2.120e+00 ... 9.100e-01 3.330e+00 9.850e+02]
 [1.416e+01 2.510e+00 2.480e+00 ... 6.200e-01 1.710e+00 6.600e+02]]
```

```
stdsc.fit(X_train)
```

Fitted mean:

```
[1.29830645e+01 2.38370968e+00 2.36314516e+00 1.95258065e+01
 1.00088710e+02 2.25838710e+00 1.96951613e+00 3.64274194e-01
 1.61250000e+00 4.99991935e+00 9.55854839e-01 2.60193548e+00
 7.46766129e+02]
```

Fitted variance:

```
[6.36966415e-01 1.28165721e+00 7.57683338e-02 1.27099792e+01
 2.13161485e+02 3.57332882e-01 9.46920734e-01 1.53099571e-02
 3.55007460e-01 5.48412330e+00 5.50353660e-02 5.17446254e-01
 9.46240663e+04]
```

Nothing happens to data after fit....:

```
[[1.371e+01 1.860e+00 2.360e+00 ... 1.110e+00 4.000e+00 1.035e+03]
 [1.222e+01 1.290e+00 1.940e+00 ... 8.600e-01 3.020e+00 3.120e+02]
 [1.327e+01 4.280e+00 2.260e+00 ... 5.900e-01 1.560e+00 8.350e+02]
 ...
 [1.242e+01 1.610e+00 2.190e+00 ... 1.060e+00 2.960e+00 3.450e+02]
 [1.390e+01 1.680e+00 2.120e+00 ... 9.100e-01 3.330e+00 9.850e+02]
 [1.416e+01 2.510e+00 2.480e+00 ... 6.200e-01 1.710e+00 6.600e+02]]
```

stdsc.transform(X\_train)      transform = apply the fitted means and variances on data

... Only after transform ...:

```
[[ 0.91083058 -0.46259897 -0.01142613 ...  0.65706596  1.94354495
  0.93700997]
 [-0.95609928 -0.96608672 -1.53725357 ... -0.40859506  0.58118003
  -1.41336684]
 [ 0.35952243  1.67501572 -0.37471838 ... -1.55950896 -1.44846566
  0.28683658]
 ...
 [-0.70550467 -0.68342693 -0.62902295 ...  0.44393375  0.49776993
  -1.30608823]
 [ 1.14889546 -0.6215951  -0.88332752 ... -0.19546286  1.0121322
   0.77446662]
 [ 1.47466845  0.11155374  0.42452457 ... -1.43162964 -1.23994042
  -0.28206514]]
```

```
stdsc.fit_transform(X_train)

... Can be done in one step:
[[ 0.91083058 -0.46259897 -0.01142613 ...  0.65706596  1.94354495
  0.93700997]
 [-0.95609928 -0.96608672 -1.53725357 ... -0.40859506  0.58118003
 -1.41336684]
 [ 0.35952243  1.67501572 -0.37471838 ... -1.55950896 -1.44846566
  0.28683658]

...
[-0.70550467 -0.68342693 -0.62902295 ...  0.44393375  0.49776993
 -1.30608823]
 [ 1.14889546 -0.6215951  -0.88332752 ... -0.19546286  1.0121322
  0.77446662]
 [ 1.47466845  0.11155374  0.42452457 ... -1.43162964 -1.23994042
 -0.28206514]]
```

# Why scaling - convergence

Text

```
from sklearn.linear_model import LogisticRegression  
  
logr = LogisticRegression(penalty='none')  
logr.fit(X_train, y_train)
```

```
/home/lau/miniconda3/envs/methods3/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning:  
  lbgfs failed to converge (status=1):  
  STOP: TOTAL NO. OF ITERATIONS REACHED LIMIT.
```

Increase the number of iterations (max\_iter) or scale the data as shown in:

<https://scikit-learn.org/stable/modules/preprocessing.html>

Please also refer to the documentation for alternative solver options:

[https://scikit-learn.org/stable/modules/linear\\_model.html#logistic-regression](https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)

n\_iter\_i = \_check\_optimize\_result(

# Why scaling - accuracy

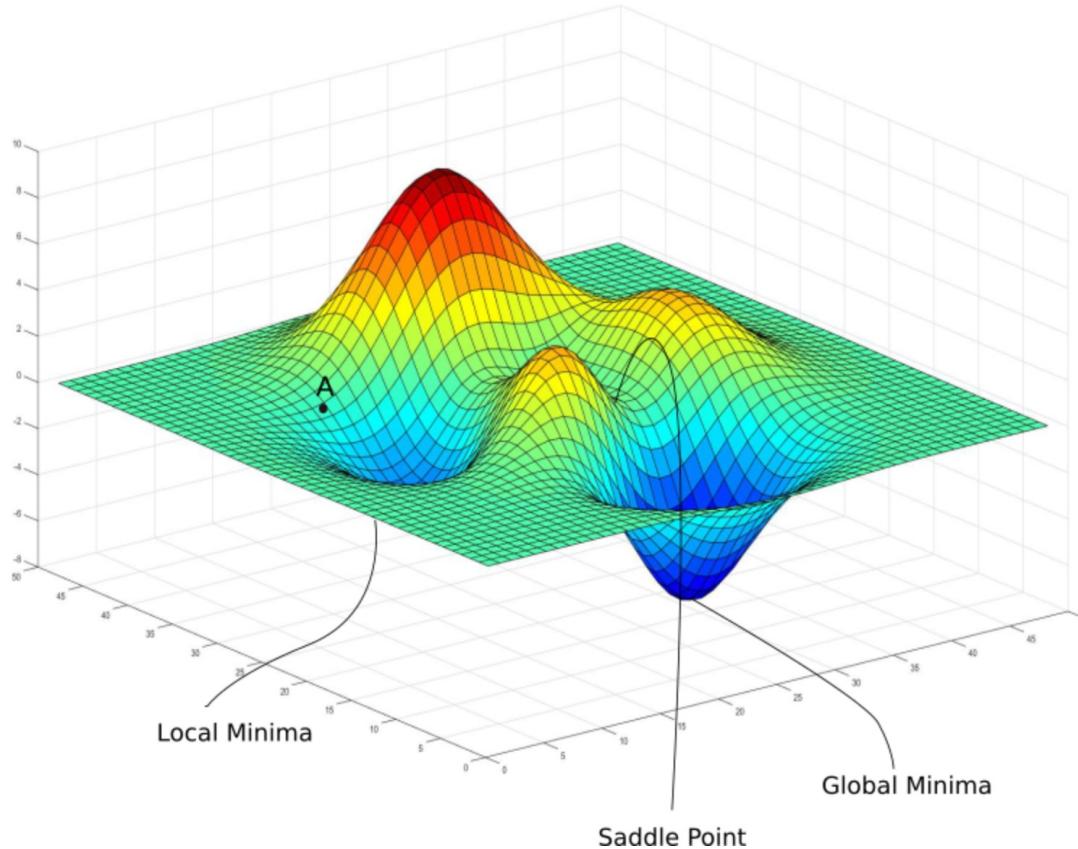
```
: logr = LogisticRegression(penalty='none', solver='newton-cg')
logr.fit(X_train, y_train)
score_org_scale = logr.score(X_test, y_test)
logr.fit(X_train_std, y_train)
score_std_scale = logr.score(X_test_std, y_test)

print('Score original scale: ' + str(score_org_scale))
print('Score standardized scale: ' + str(score_std_scale))
```

Score original scale: 0.9259259259259259

Score standardized scale: 1.0

# Why scaling - accuracy



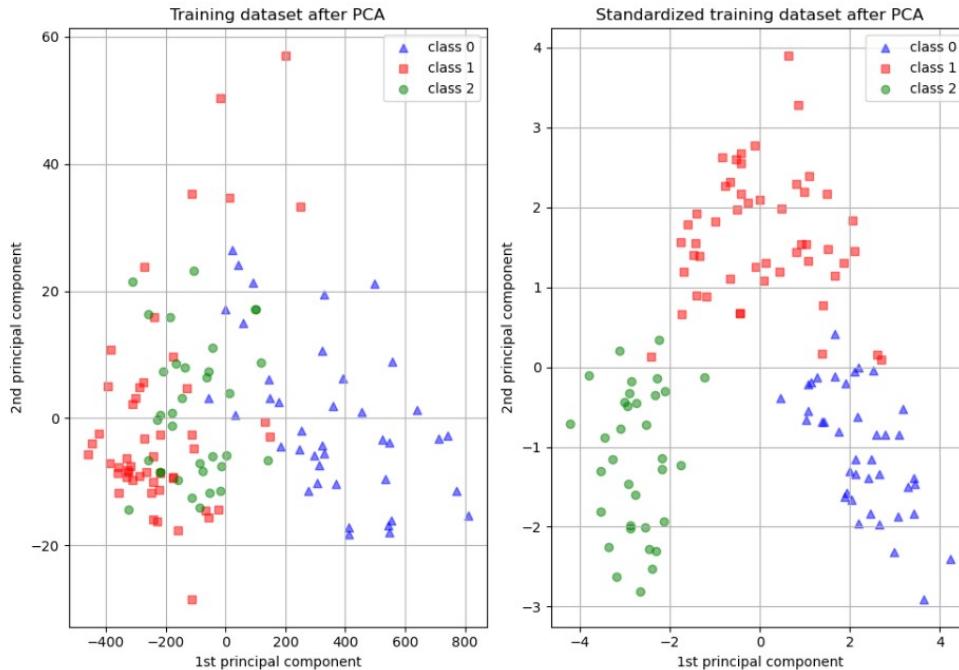
**if not scaled, variables with high values will dominate, and we might end in local minima**

<https://wngaw.github.io/linear-regression/>

# Why scaling – *lmer* and *glmer*

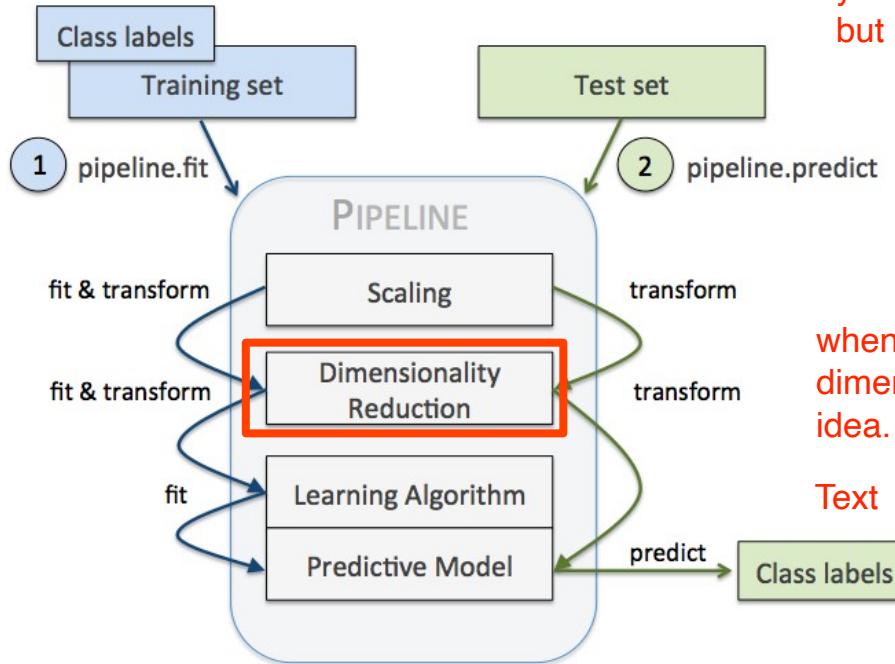
Can also help convergence (also fits via gradient descent)

# Also important for PCA



[https://scikit-learn.org/stable/auto\\_examples/preprocessing/plot\\_scaling\\_importance.html#sphx-glr-auto-examples-preprocessing-plot-scaling-importance-py](https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html#sphx-glr-auto-examples-preprocessing-plot-scaling-importance-py)

# Pipeline example



you can also scale linear variables,  
but Lau wouldn't normally do it.

when working with several dimension  
dimensionality reduction is often a good  
idea.

Text

# Dimensionality reduction

- Feature selection
  - Choose a subset of the original features
  - L1 regularisation is an example
- Feature extraction
  - Create a new feature subspace based on derived information
  - Principal component analysis is an example

Examples:

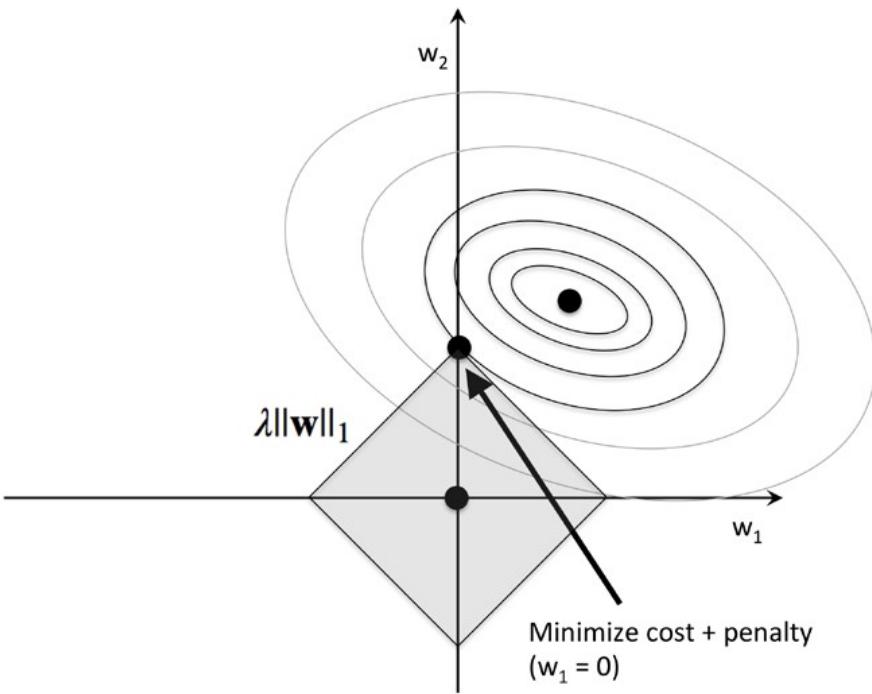
Feature selection:

Focus on subsets of original features.

E.g. the matrix Z is

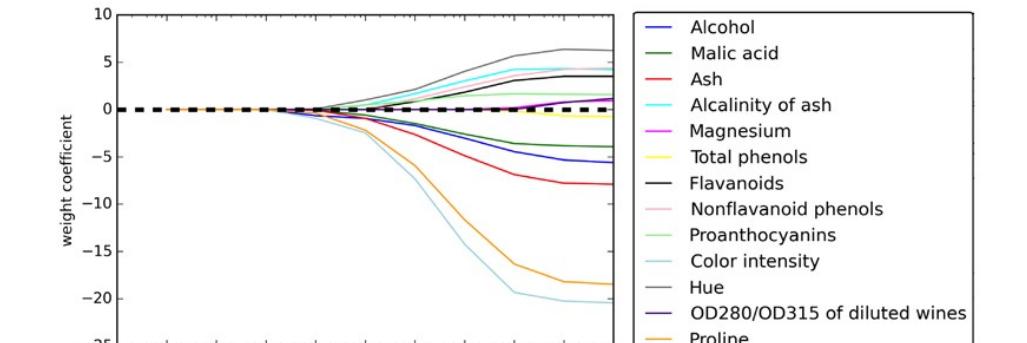
# Feature selection example

## L1 REGULARISATION



(p. 115: Raschka, 2015)

the less we regularize the more variance there is between the coefficients.

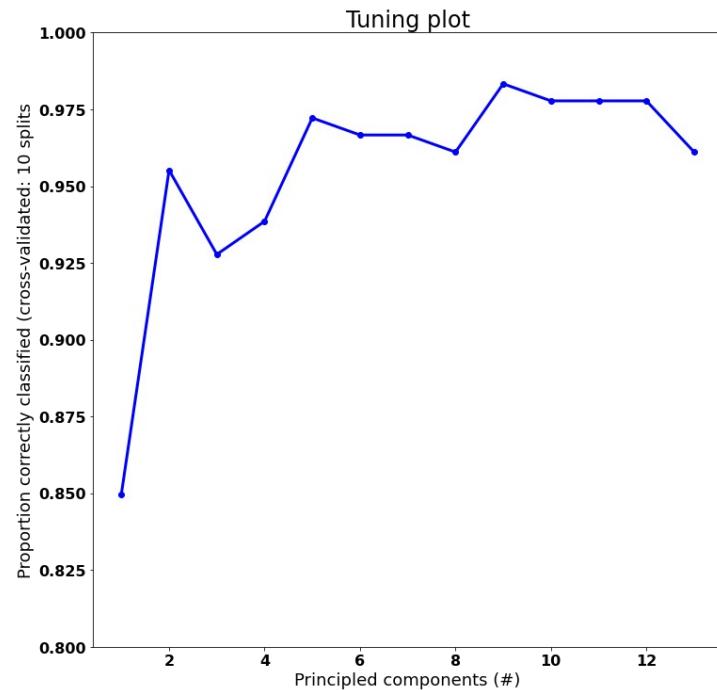
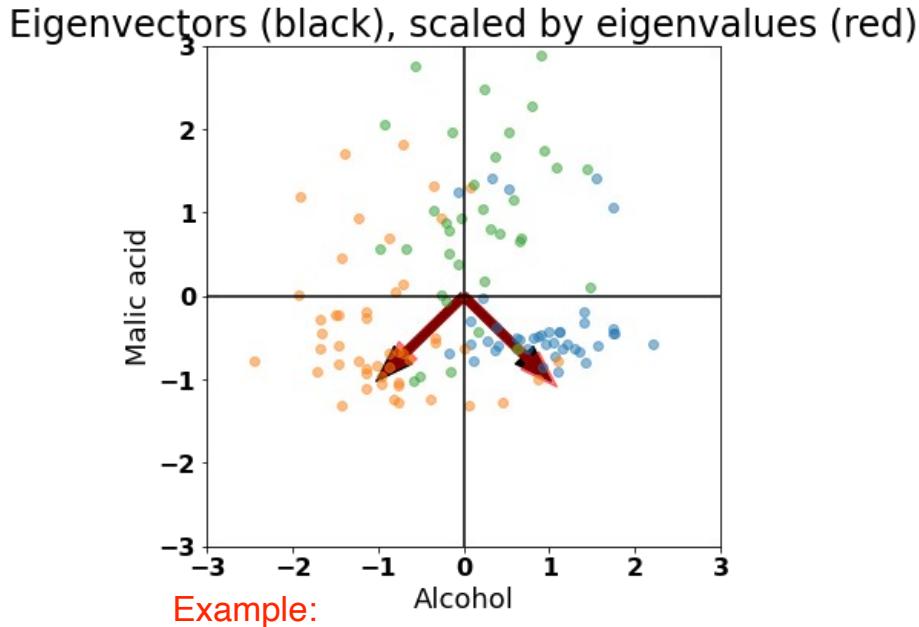


“selecting the non-zero features” is also a kind of feature selection and thus a method of dimensionality reduction.

(p. 118: Raschka, 2015)

# Feature extraction example

## PRINCIPLED COMPONENT ANALYSIS



## **Summary so far:**

**Scaling** makes sure that we get reliable fits and treat each variable equally

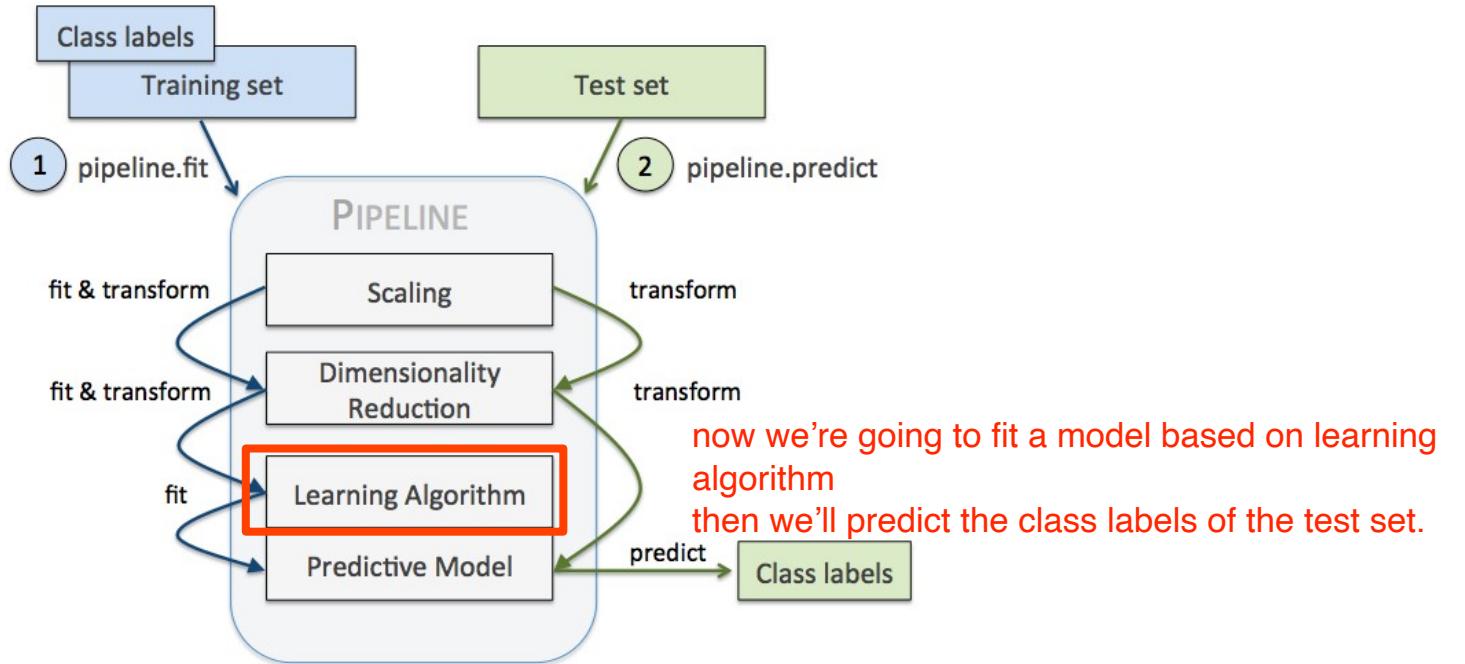
**Dimensionality reduction** makes sure that we are not overfitting

# Ways of preventing overfitting

- Collect more training data
- Introduce a penalty for complexity via regularization
- Choose a simpler model with fewer parameters Text
- Reduce the dimensionality of the data

(p. 112: Raschka, 2015)

# Pipeline example



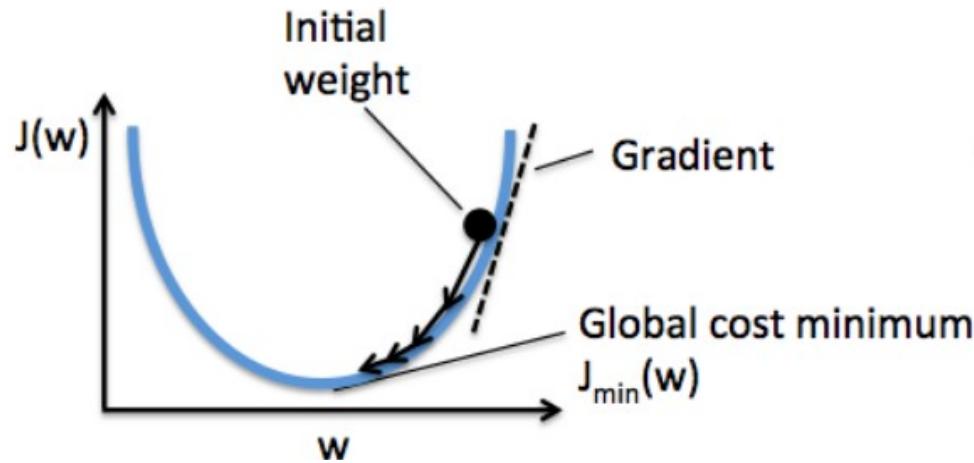
# Learning algorithm examples

## GRADIENT DESCENT

$$\Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

the gradient, is the rate of change.

A general formulation for linear models:  $\Delta w_j = \eta \sum_i^n (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$



(p. 40: Raschka, 2015)

# Learning algorithm examples

## SUPPORT VECTOR MACHINE

minimize half time the weights squared, and the slack variable introduces a few errors determined by the value of C

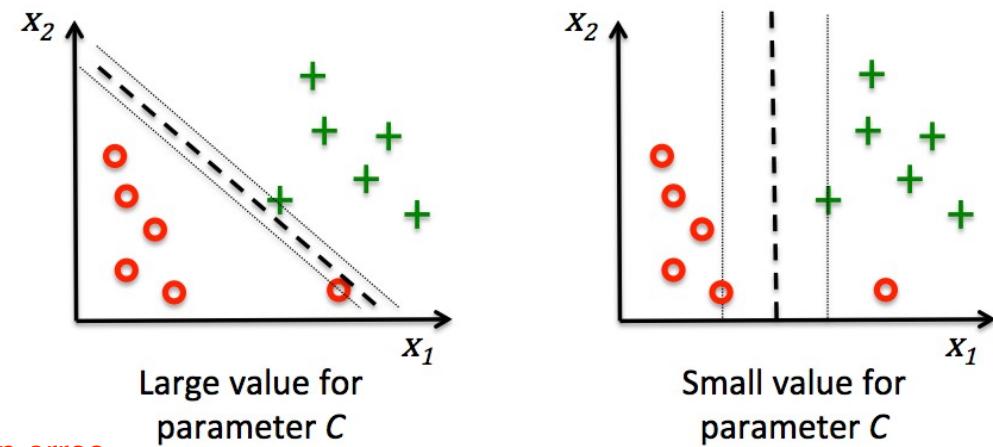
$$\text{minimize: } \frac{1}{2} \|\mathbf{w}\|^2 + C \left( \sum_i^n \xi^{(i)} \right)$$

under the constraints:

$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1 \text{ if } y^{(i)} = 1$$

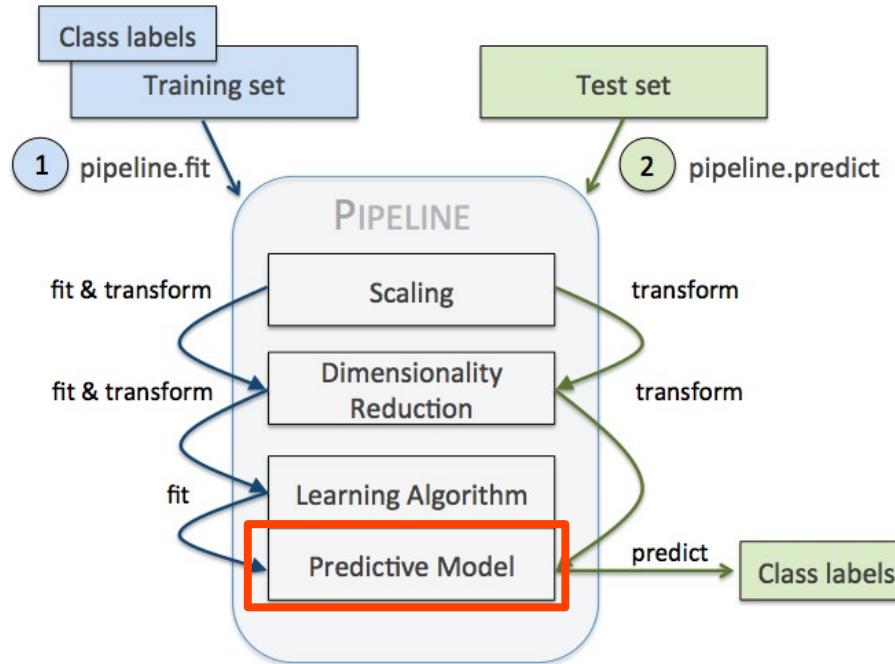
$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} < -1 \text{ if } y^{(i)} = -1$$

we try to find the w values (weights) that minimize the prediction errors



(p. 72: Raschka, 2015)

# Pipeline example



(p. 172: Raschka, 2015)

# Predictive model

## LOGISTIC REGRESSION

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

# Predictive model

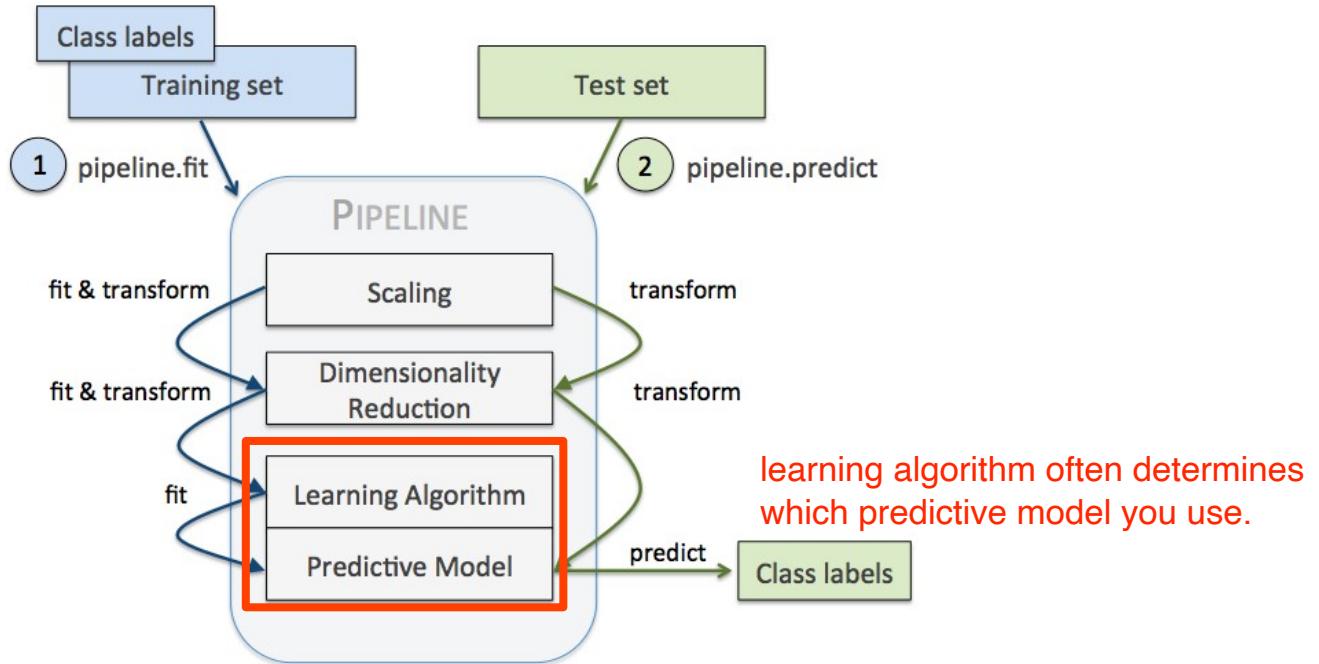
## LINEAR REGRESSION

in linReg we don't quantize, we just use the  $\hat{y}$  as our prediction.

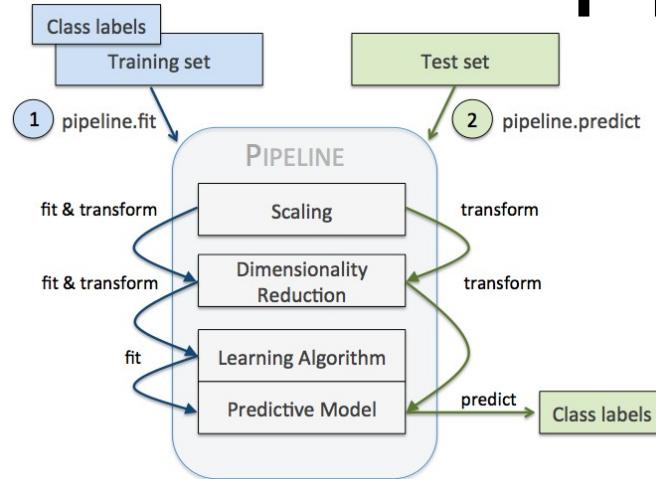
$$\hat{y} = \mathbf{w}^T \mathbf{x} = w_0 x_0 + w_1 x_1 + \dots + w_{m-1} x_{m-1} + w_m x_m$$

# Pipeline example

These in practice go together



# Pipeline example



```
>>> from sklearn.preprocessing import StandardScaler  
>>> from sklearn.decomposition import PCA  
>>> from sklearn.linear_model import LogisticRegression  
>>> from sklearn.pipeline import Pipeline  
>>> pipe_lr = Pipeline([('scl', StandardScaler()),  
...                     ('pca', PCA(n_components=2)),  
...                     ('clf', LogisticRegression(random_state=1))])  
>>> pipe_lr.fit(X_train, y_train)  
>>> print('Test Accuracy: {:.3f}' % pipe_lr.score(X_test, y_test))  
Test Accuracy: 0.947
```

(p. 172: Raschka, 2015)

(p. 171: Raschka, 2015)

Text

Read more in chapter 6 –  
you know all the  
ingredients – now it is just  
setting the recipe

# VALIDATING YOUR MODEL

## Dividing into training and test sets

# Effects of train-test sizes

```
## TRAIN-TEST SETS SIZES
# training proportion is 10%, 20% etc,
training_proportions = np.array([0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.90, 0.99])
test_proportions      = 1 - training_proportions

n_trainings = len(training_proportions)
scores = np.zeros(shape=n_trainings)

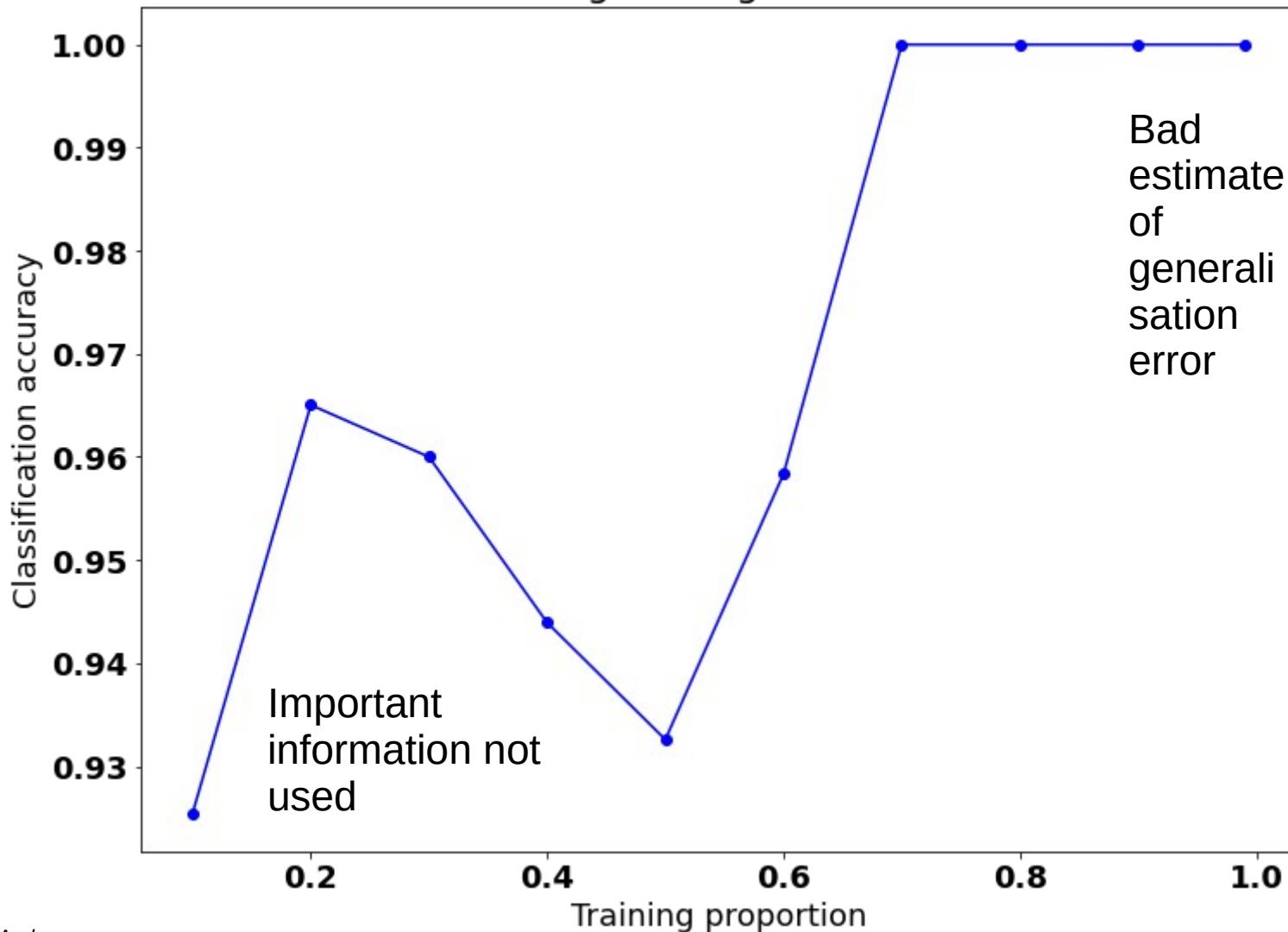
logr = LogisticRegression(penalty='none') no regularization in this case

for training_index in range(n_trainings):    it's expensive (data wise) to have validation sets
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=test_proportions[training_index], random_state=0)

    X_train_std = stdsc.fit_transform(X_train)
    X_test_std  = stdsc.transform(X_test)

    fit = logr.fit(X_train_std, y_train)
    scores[training_index] = logr.score(X_test_std, y_test)
```

## Logistic regression



With large enough datasets, small-proportioned  
test sets less of a problem

# Also less of a problem with $K$ -fold cross-validation



(p. 176: Raschka, 2015)

# Stratified $K$ -fold cross-validation

Makes sure that each fold contains an equal number of labels such that the classifier is not biased towards a set of labels

if u use cross validation, then remember to stradify it.

# Two types of parameters

- Parameters learnt from data
  - e.g. weights in regression types
- Tuning parameters
  - or hyperparameters -> determines how weights are set
  - e.g.  $C$  and  $\gamma$

C influences how much we regularize

lower  $c$ , higher lambda  
higher  $c$ , lower lambda

# Grid search

```
# Set the parameters by cross-validation
tuned_parameters = [
    {"kernel": ["rbf"], "gamma": [1e-3, 1e-4], "C": [1, 10, 100, 1000]},
    {"kernel": ["linear"], "C": [1, 10, 100, 1000]},
]
```

```
clf = GridSearchCV(svc(), tuned_parameters, scoring="%s_macro" % score)
clf.fit(x_train, y_train)
```

[https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_grid\\_search\\_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py](https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py)

# Grid search

Grid scores on development set:

```
0.986 (+/-0.016) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}  
0.959 (+/-0.028) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}  
0.988 (+/-0.017) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}  
0.982 (+/-0.026) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}  
0.988 (+/-0.017) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}  
0.983 (+/-0.026) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}  
0.988 (+/-0.017) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}  
0.983 (+/-0.026) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}  
0.974 (+/-0.012) for {'C': 1, 'kernel': 'linear'}
```

[https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_grid\\_search\\_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py](https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py)

# Grid search

Read more in chapter 6  
– you know all the  
ingredients – now it is  
just setting the recipe

# Did you learn?

*Organising and preprocessing messy data*

- 1) Learning why data should be scaled
- 2) Using feature selection to avoid overfitting
- 3) Understanding the basic steps of a machine learning pipeline

# **OPTIONAL: SUPPORT VECTOR MACHINES AND KERNELS**

Based on chapter 2:

Abe, S., 2010. Support Vector Machines for Pattern Classification. Springer, London.

# Kernels can be used to not explicitly go into higher-dimensional space

$x_{n \times m}$ : observations and predictor variables (features)

$\phi(x) = (\phi_1(x), \dots, \phi_l(x))^T$  : mapping from  $m$  dimensions to  $l$  dimensions ( $l > m$ )

$K(x, x') = \phi^T(x) \phi(x')$ : the Kernel function means that we need not explicitly go into higher-dimensional space

# Different kernels

Linear kernel

$$K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$$

Polynomial kernel

$$K(\mathbf{x}^T \mathbf{x}' + 1)^d$$

Radial basis function kernel

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$$

$K$  can be rewritten as:

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x}\|^2) \exp(-\gamma \|\mathbf{x}'\|^2) \exp(2\gamma \mathbf{x}^T \mathbf{x}')$$

$$\exp(2\gamma \mathbf{x}^T \mathbf{x}') = 1 + 2\gamma \mathbf{x}^T \mathbf{x}' + 2\gamma^2 (\mathbf{x}^T \mathbf{x}')^2 + \frac{(2\gamma)^3}{3!} (\mathbf{x}^T \mathbf{x}')^3 + \dots, \text{(an infinite sum)}$$

Thus the Kernel function contains all possible polynomials and can thus represent non-linear problems

# Decision functions

$$D(\mathbf{x}) = \sum_{i \in S} \alpha_i y_i \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}\|^2) + b$$

$\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_M)^T$  are non-negative Lagrange multipliers  
(Lagrange multipliers are used to find local extrema)  
 $b$  is a bias (a constant)

$$\mathbf{x} \in \begin{cases} \text{Class 1 if } D(\mathbf{x}) > 0 \\ \text{Class 2 if } D(\mathbf{x}) < 0 \end{cases}$$

# Minimisation problem

$$\text{minimise } Q(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2$$

$$\text{subject to } y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \text{ for } i=1, \dots, M$$

unconstrained version

$$Q(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^M \alpha_i \{ y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \}$$

$$\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_M)^T$$

$M$ :  $m$ -dimensional training inputs  $\mathbf{x}_i (i=1, \dots, M)$

# References

- Abe, S., 2010. Support Vector Machines for Pattern Classification. Springer, London.
- Raschka, S., 2015. Python Machine Learning. Packt Publishing Ltd.