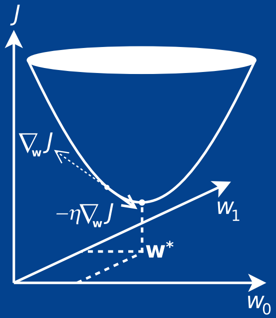
AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

MACHINE LEARNING

LESSON 5: Training II

CARSTEN EIE FRIGAARD

SPRING 2019



L05: Training II

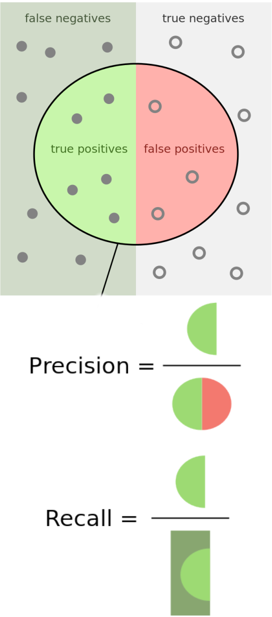
Agenda

- Important: your feedback (positive as well as negative)...
- Course modification i): *more OPTIONAL* exercices
- Course modification ii): revision to 'Microlearning'
  (only one lecure session, beginning of class)
- Installing `keras` is slow/buggy, see WIKI..
- L04 Training I: Training a linear regression model
  OPTIONAL: Exercise: `L04/linear_regression_1.ipynb`
  OPTIONAL: Exercise: `L04/linear_regression_2.ipynb`
- L05 Training II: **Training and model concepts**
  Exercise: `L05/gradient_descent.ipynb`
  Exercise: `L05/capacity_under_overfitting.ipynb`
  Exercise: `L05/generalization_error.ipynb`
  OPTIONAL: Exercise: `L05/train_test_split.ipynb`

RESUMÉ: Metrics

Precision, recall and accuracy, F_1-score, and confusion matrix

$$
\begin{aligned}
\text{precision,} \quad & p &= \frac{TP}{TP+FP} \\
\text{recall (or sensitivity),} \quad & r &= \frac{TP}{TP+FN} \\
\text{accuracy,} \quad & a &= \frac{TP+TN}{TP+TN+FP+FN} \\
F_1\text{-score,} \quad & F_1 &= \frac{2pr}{p+r}
\end{aligned}
$$



Confusion Matrix, binary-class data,

$\mathbf{M}_{\text{confusion}} =$

	actual true	actual false
predicted true	TP	FP
predicted false	FN	TN

RESUMÉ: Covariance Matrix

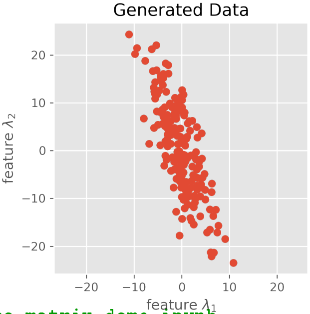
Data matrix for a two-dimensional feature space

$$\mathbf{X} = \begin{array}{c} \begin{array}{cc} \color{green}{\lambda_1} & \color{green}{\lambda_2} \end{array} \\ \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \\ x_1^{(n)} & x_2^{(n)} \end{bmatrix} \end{array} = \begin{bmatrix} 0.35 & -7.62 \\ -4.99 & 13.79 \\ \vdots & \vdots \\ 9.54 & -25.64 \\ 4.21 & -2.25 \end{bmatrix}$$

Covariance matrix, for the two-dimensional feature space

$$\mathbf{\Sigma}(\mathbf{X}) = \begin{bmatrix} \sigma(\lambda_1,\lambda_1) & \sigma(\lambda_1,\lambda_2) \\ \sigma(\lambda_2,\lambda_1) & \sigma(\lambda_2,\lambda_2) \end{bmatrix} = \begin{bmatrix} \sigma(\lambda_1)^2 & \sigma(\lambda_1,\lambda_2) \\ \sigma(\lambda_2,\lambda_1) & \sigma(\lambda_2)^2 \end{bmatrix}$$

with $$\sigma(\lambda_1,\lambda_2) = \frac{1}{n}\sum_{i=1}^{n}(x_1^{(i)} - \mu_{\lambda_1})(x_2^{(i)} - \mu_{\lambda_2})$$



Example: $\mathbf{X}$; a 100 x 2 matrix, see fig..

$$\mathbf{\Sigma}(\mathbf{X}) = \begin{bmatrix} 13.2 & -28.8 \\ -28.8 & 93.3 \end{bmatrix}$$

- $\mathbf{\Sigma}$ is real and symmetric,
- diagonal: the (auto)-variance of a feature, $\sigma(\lambda_j)^2$
- Pearson's r: cross-correlation via cross-covar,
- ~~similar dimension as Confusion matrix~~,
- python implementation: see `L02/Extra/covariance_matrix_demo.ipynb`.

RESUMÉ: Covariance Matrix, Take II

For a dataset, **X**; features cat and dog; classifying cat/non-cat

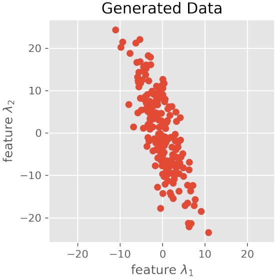
Covariance matrix (dim = features x features)

$$\boldsymbol{\Sigma}(\mathbf{X}) = \begin{bmatrix} \sigma_{\text{cat,cat}} & \sigma_{\text{cat,dog}} \\ \sigma_{\text{dog,cat}} & \sigma_{\text{dog,dog}} \end{bmatrix} = \begin{array}{c|cc} & \text{cat} & \text{dog} \\ \hline \text{cat} & \sigma^2_{\text{cat}} & \sigma_{\text{cat,dog}} \\ \text{dog} & \sigma_{\text{dog,cat}} & \sigma^2_{\text{dog}} \end{array}$$

co-variance

$$\sigma_{\text{cat,dog}} = \frac{1}{n}\sum_{i=1}^{n}(x^{(i)}_{\text{cat}} - \mu_{\text{cat}})(x^{(i)}_{\text{dog}} - \mu_{\text{dog}})$$

Example $\boldsymbol{\Sigma}(\mathbf{X}) = \begin{bmatrix} 13.2 & -28.8 \\ -28.8 & 93.3 \end{bmatrix}$



Confusion matrix, cat/non-cat

(dim = classes x classes)

$$\mathbf{M} = \begin{bmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{bmatrix} = \begin{array}{c|cc} & \multicolumn{2}{c}{\textit{actual}} \\ & \text{cat} & \text{non-cat} \\ \hline \text{cat} & \text{T, cat} & \text{F, cat} \\ \text{dog} & \text{F, dog} & \text{T, non-cat} \end{array}$$

RESUMÉ: Covariance Matrix

- python implementation: see L02/Extra/covariance_matrix_demo.ipynb.

not so scarry, afterall...

```python
# Covariance
def cov(x, y, bias=True):
    assert len(x)==len(y)
    xbar, ybar = x.mean(), y.mean()
    if bias:
        n=len(x)
    else:
        n=len(x) - 1
    assert n>0
    return np.sum((x - xbar)*(y - ybar))/n

# Covariance matrix
def cov_mat(X, bias=True):
    return np.array([ \
      [cov(X[0], X[0], bias), cov(X[0], X[1], bias)], \
      [cov(X[1], X[0], bias), cov(X[1], X[1], bias)]  \
    ])
```

RESUMÉ: Training a Linear Regressor

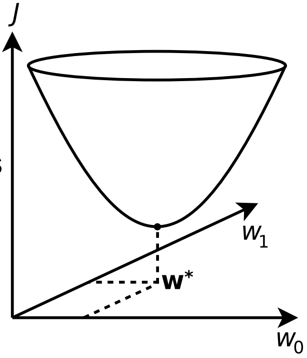
Minimizing the Linear Regression: The `argmin` concept

Our linear regression cost function was

$$J(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \frac{1}{2} \, ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2$$

and training amounts to finding a value of $\mathbf{w}$, that minimizes J. This is denoted as

$$\begin{aligned} \mathbf{w}^* &= \mathrm{argmin}_{\mathbf{w}} \, J(\mathbf{X}, \mathbf{y}; \mathbf{w}) \\ &= \mathrm{argmin}_{\mathbf{w}} \, \tfrac{1}{2} \, ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \end{aligned}$$



and by minima, we naturally hope for

- the global minumum

thought for non-linear models this cannot be guarantied, hitting some

- local minimum

RESUMÉ: `L04/linear_regression_1.ipynb`

Training: The Closed-form Linear-Least-Squares Solution

To solve for $\mathbf{w}^*$ in closed form, we find the gradient of J with respect to $\mathbf{w}$

$$\nabla_{\mathbf{w}} J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_m}\right]^\top$$

Taking the partial deriverty $\partial/\partial_{\mathbf{w}}$ of the J via the gradient (nabla) operator

$$\begin{aligned} \nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{y}; \mathbf{w}) &= \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \ = \ 0 \\ 0 &= \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y} \end{aligned}$$

with a small amount of matrix algebra, this gives the *normal equation*

$$\begin{aligned} \mathbf{w}^* &= \mathrm{argmin}_{\mathbf{w}} \ \tfrac{1}{2} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \\ &= \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y} \end{aligned}$$

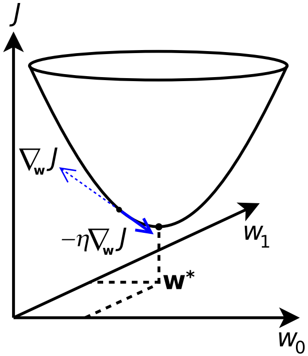
Numerical Solution: Gradient Descent (GD)

The GD Algorithm

First, find the deriverty of J, via $\nabla_{\mathbf{w}} J$, that after some matrix algebra gives

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{2}{m}\mathbf{X}^{\top}\left(\mathbf{X}\mathbf{w} - \mathbf{y}\right)$$

Then move along in the *opposite* direction of this gradient, taking a step of size η



$$\mathbf{w}^{(step\ \ N+1)} = \mathbf{w}^{(step\ \ N)} - \eta\nabla_{\mathbf{w}} J(\mathbf{w})$$

The SG-algo in python code

```python
for iteration in range(n_iterations):
    gradients=2/m*X_b.T.dot(X_b.dot(theta)-y)
    theta=theta - eta * gradients
```

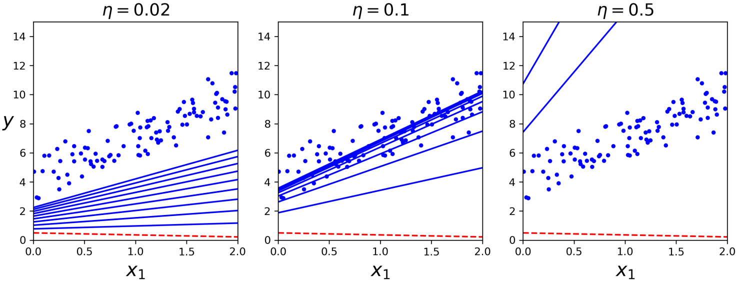
NOTE: The `X_b` is a `X` with a all-1 column prepended, and using the contant factor $2/m$ instead of just $1/2$ and with $\mathbf{w} = \boldsymbol{\theta}$

Learning rate, η

And understanding Scikit-learn's `SGDRegressor`

Scikit-learn η updating schemes:

constant, adaptive, invscaling, optimal



The `SGDRegressor` in Scikit-learn has a *hyperparameter* for this:

```
learning_rate:  string, optional
'invscaling':[default], eta=eta0/pow(t,power_t)
```

Learning rate, η

And understanding Scikit-learn's `SGDRegressor`

The `SGDRegressor` constructor in Scikit-learn

```
class sklearn.linear_model.SGDRegressor(
  loss    ='squared_loss', penalty      ='l2',
  alpha   =0.0001,         l1_ratio     =0.15,
  tol     =None,           shuffle      =True,
  verbose =0,              epsilon      =0.1,
  eta0    =0.01,           power_t      =0.25,
  n_iter_no_change=5,      warm_start   =False,
  fit_intercept   =True,   max_iter     =None,
  average         =False,  n_iter       =None
  random_state    =None,   learning_rate='invscaling',
  early_stopping =False,   validation_fraction=0.1
)
```

Important for now...(hyperparam search in L07)

- `loss`, `penalty` (our MSE and $\mathcal{L}_2$ norm),
- `eta0`, `learning_rate`,
- `shuffle`, `early_stopping`,
- and perhaps `random_state`.

Stochastic Gradient Descent (SGD) Method

Exercise: `gradient_descent.ipynb`

The problem with the GD Algorithm, it takes $\mathbf{X}$ as input, the complete dataset

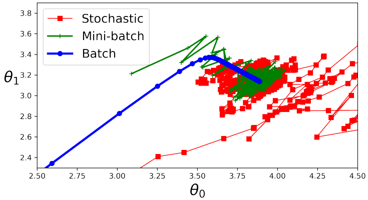
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{2}{m}\mathbf{X}^\top \left(\mathbf{X}\mathbf{w} - \mathbf{y}\right)$$

That's a big mouthful to matrix transpose and multiply!

Introducing a per-data-sample, or stochastic, method

$$\nabla_{\mathbf{w}} J(\mathbf{w})^{(i)} = \frac{2}{m}\mathbf{x}^{(i)\top} \left(\mathbf{x}^{(i)}\mathbf{w} - \mathbf{y}\right)$$

```python
for epoch in range(n_epochs):
  for i in range(m):
    ...
    r = np.random.randint(m)
    xi = X_b[r:r+1]
    yi = y[r:r+1]
    grads = 2*xi.T.dot(xi.dot(theta)-yi)
    eta   = ...
    theta = ...
```



NOTE: Notice the use of `epoch` in SGD, that is different from `iteration` in GD.

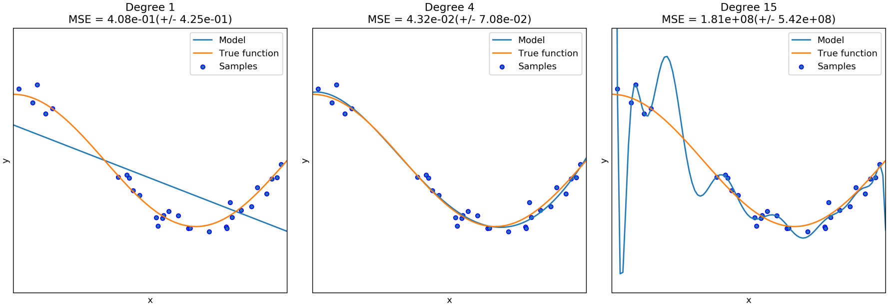
Model capacity and under/overfitting

Exercise: `capacity_under_overfitting.ipynb`

Dummy and Paradox classifier:
capacity fixed ~ 0, cannot generalize at all!

Linear regression for a polynomial model:
capacity $\sim$ degree of polynomial, x^n

Polynomial linear reg. fit for underlying model: `cos(x)`

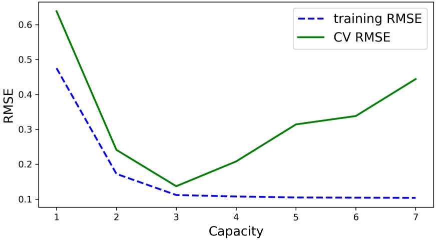


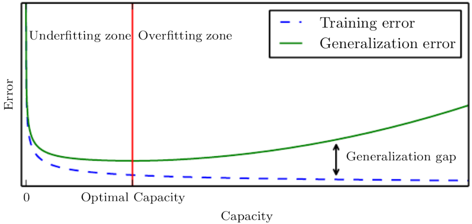
- underfitting: capacity of model too low,
- overfitting: capacity to high.

Generalization Error

Exercise: `generalization_error.ipynb`

RMSE-capacity plot for lin. reg. with polynomial features
(capacity = degree of poly)





Inspecting the plots from [HOML] and [DL], extracting the concepts

- training/generalization error,
- generalization gab,
- underfit/overfit zone,
- optimal capacity (best-model, early stop).