Introduction to Machine Learning

Linear Classification

Nils M. Kriege

WS 2023

Data Mining and Machine Learning
Faculty of Computer Science
University of Vienna

Credit: Slides based on the IML Lectures by Sebastian Tschiatschek and Andreas Krause

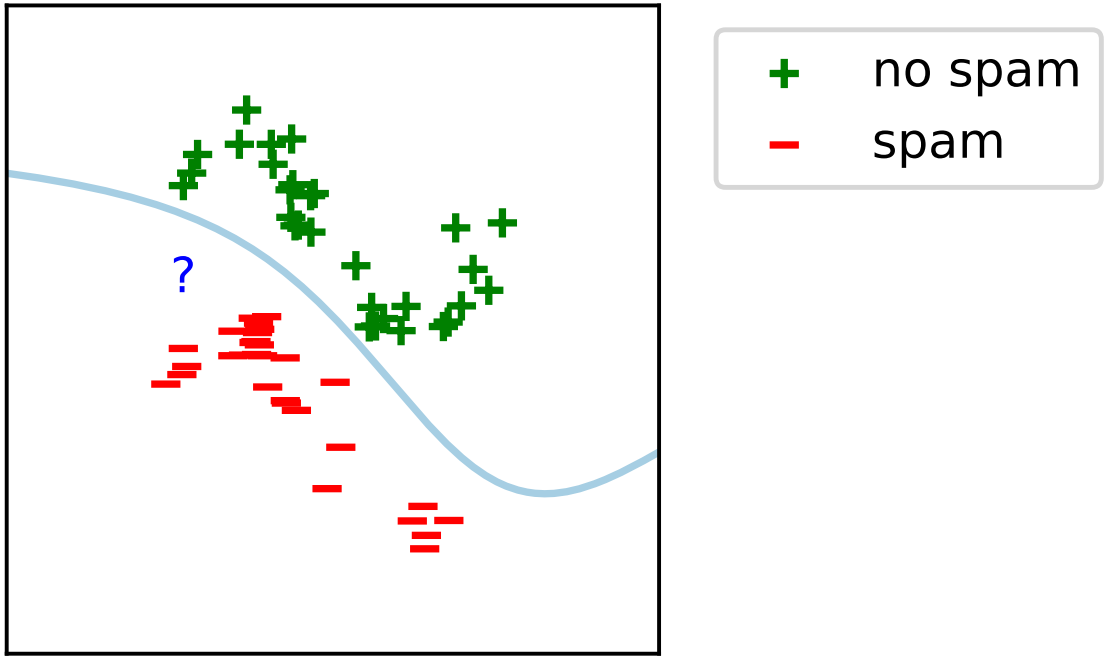
Recap: Classification

- Instance of supervised learning where $\mathcal{Y}$ is discrete (categorical)
- Want to assign a data point in $\mathcal{X}$, e.g.,
  - Documents
  - Queries
  - Images
  - User visits
  - ...

  a label in $\mathcal{Y}$ (spam/non-spam; topics such as sports, politics, entertainment; click/no-click, ...)

For now, focus on binary classification.

Illustration of binary classification

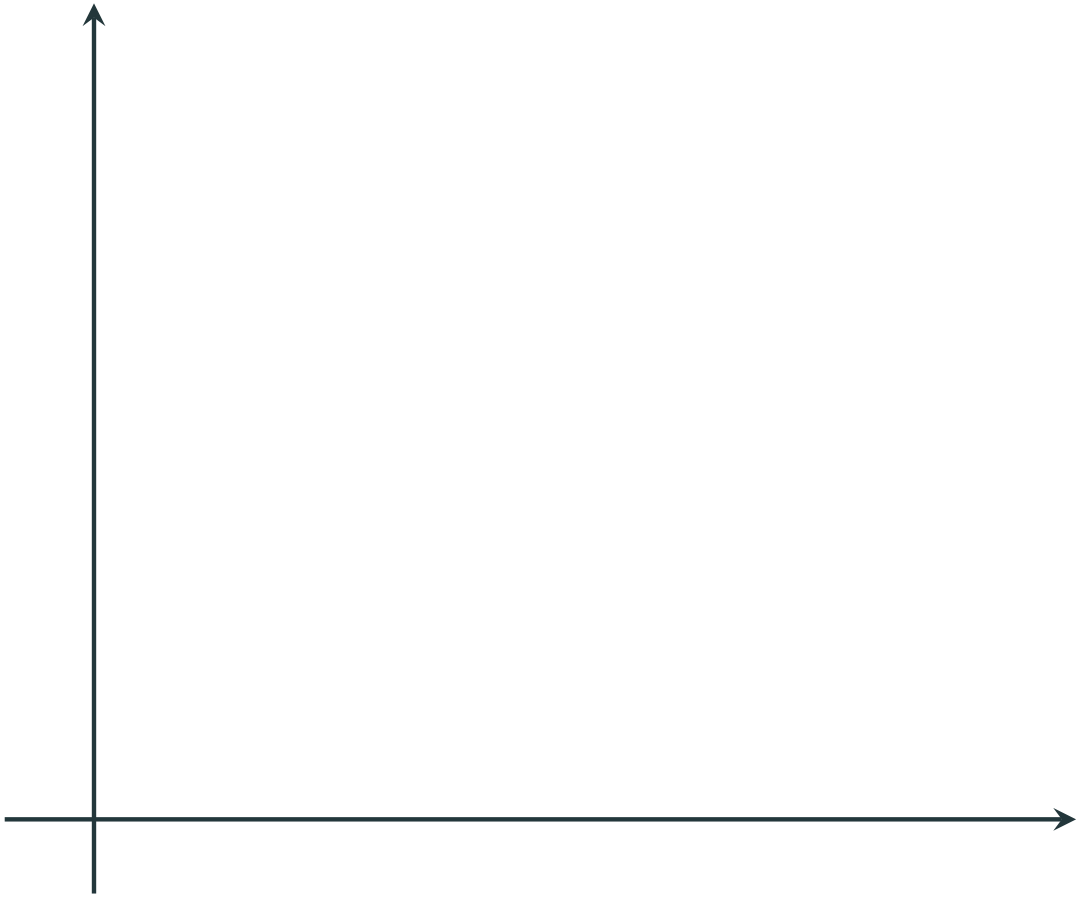


- **Input**: Labeled data set (e.g., represented as bag-of-words) with positive (+) and negative (-) examples
- **Output**: Decision rule (hypothesis)

[classification/classification.py]

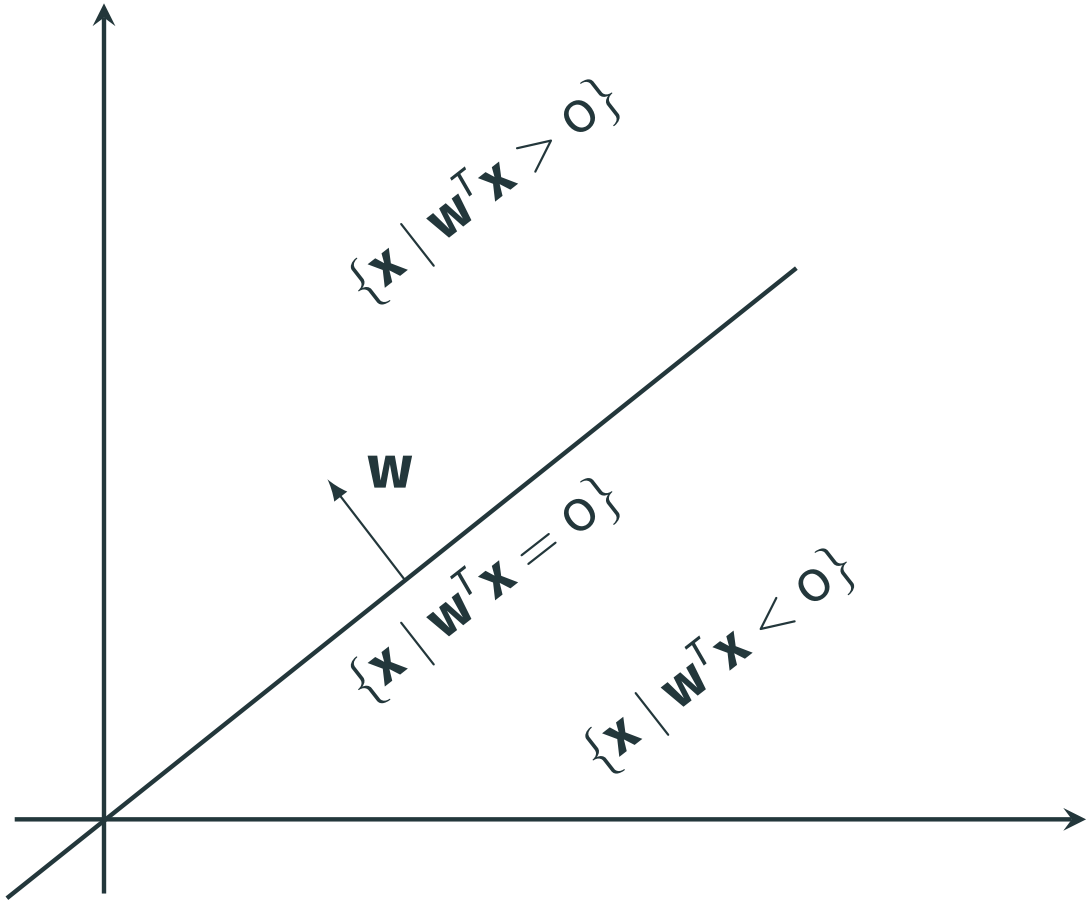
Linear classifiers

- Data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
- $h(x) = \text{sign}(\mathbf{w}^T\mathbf{x})$



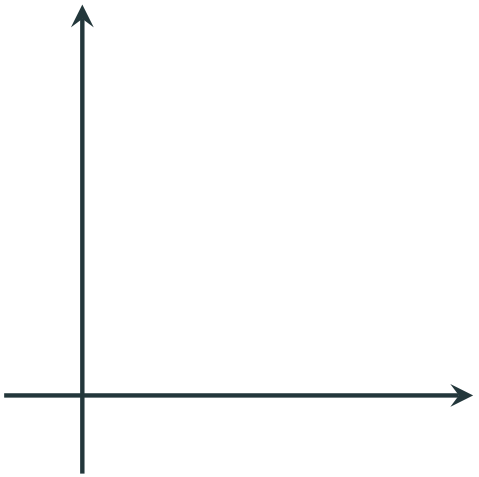
Linear classifiers

- Data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
- $h(x) = \text{sign}(\mathbf{w}^T\mathbf{x})$



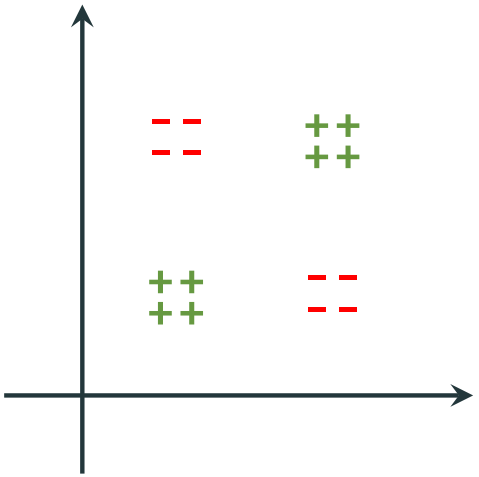
Why linear classification?

- ⚠ Linear classification seems restrictive



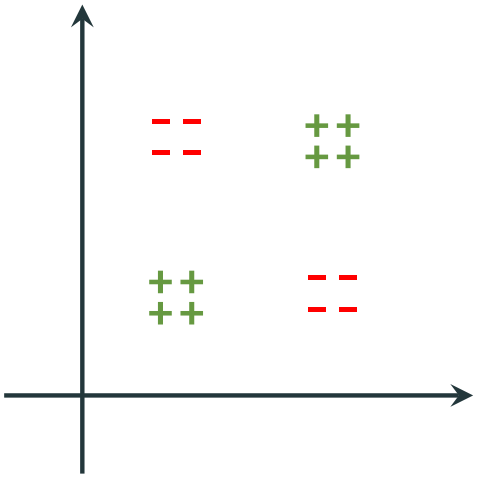
Why linear classification?

- ⚠ Linear classification seems restrictive



Why linear classification?

- ⚠ Linear classification seems restrictive



- Especially in high-dimensional settings / when using the right features, often works quite well
- Prediction is typically very efficient

Finding linear separators

- Want to write the search for a classifier as an optimization problem
- What should we optimize?

Finding linear separators

- Want to write the search for a classifier as an optimization problem
- What should we optimize?
- **First idea**: Seek $\mathbf{w}$ that minimzes # mistakes

Optimization problem

- Data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \{+1, -1\}$
- **Goal**:

$$\begin{aligned}\hat{\mathbf{w}} &= \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} [y_i \neq \operatorname{sign}(\mathbf{w}^T \mathbf{x}_i)] \\ &= \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l_{0/1}(\mathbf{w}; \mathbf{x}_i, y_i)\end{aligned}$$

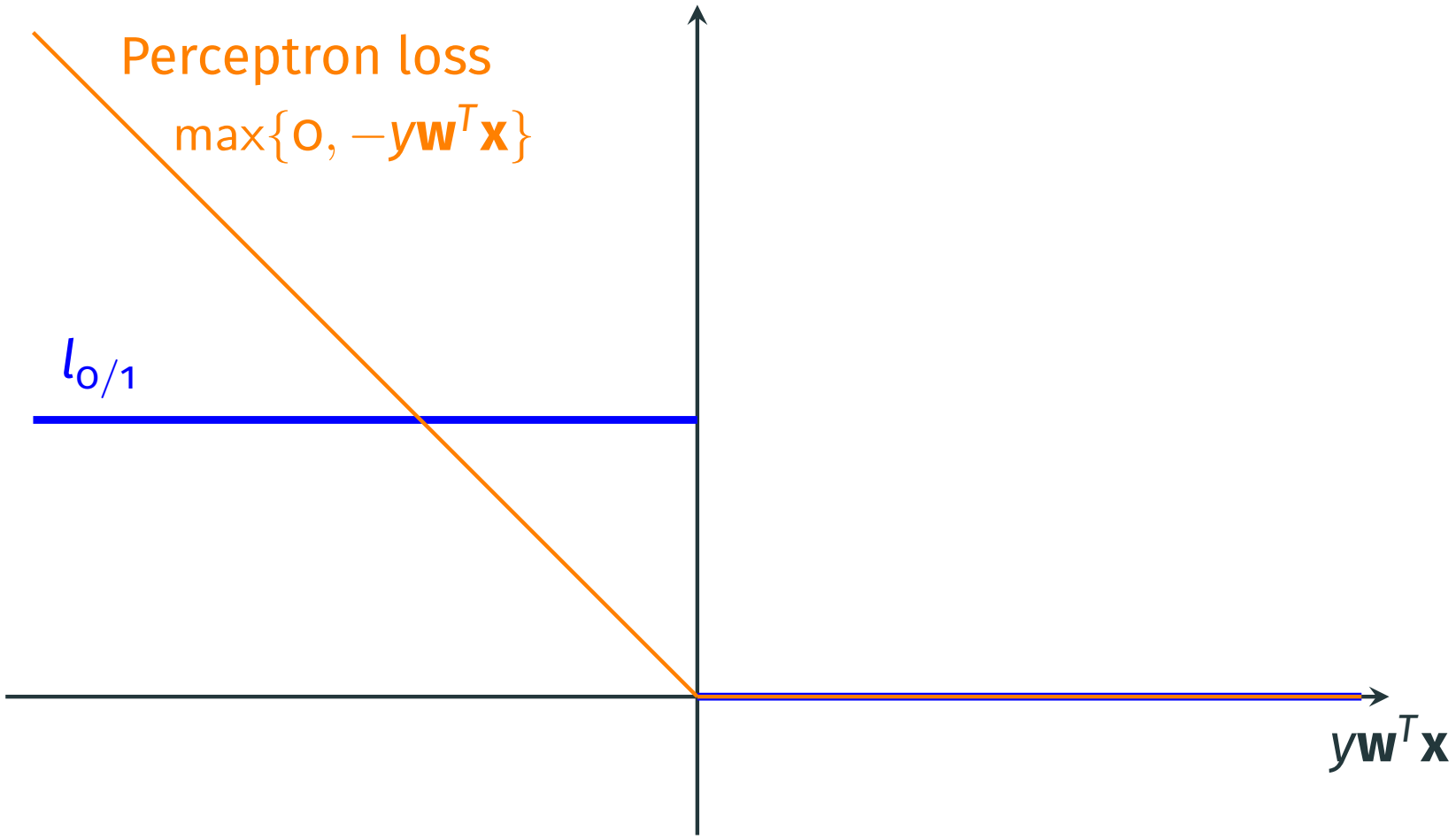
Optimization problem

- Data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \{+1, -1\}$
- **Goal**:

$$\begin{aligned}\hat{\mathbf{w}} &= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} [y_i \neq \operatorname{sign}(\mathbf{w}^T \mathbf{x}_i)] \\ &= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} l_{0/1}(\mathbf{w}; \mathbf{x}_i, y_i)\end{aligned}$$

- ⚠ **Challenge**: in contrast to squared loss, the 0/1 loss is not convex (not even differentiable)

A surrogate loss function

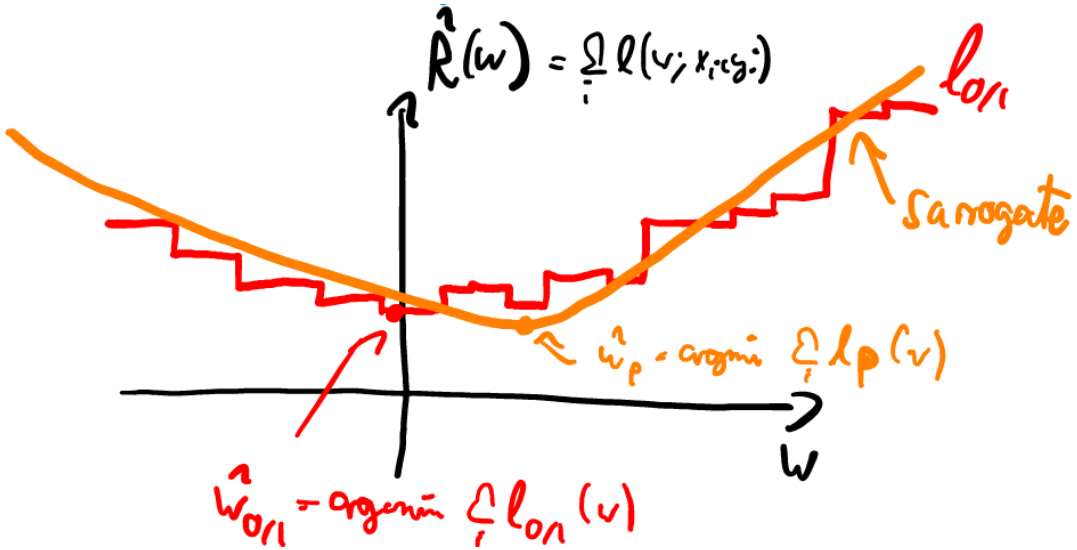


Key concept: Surrogate losses

- Replace intractable cost function that we care about (e.g., 0/1 loss) by tractable loss function (e.g., Perceptron loss) for sake of optimization / model fitting
- When evaluating a model (e.g., via cross-validation), use original cost / performance function

Key concept: Surrogate losses

- Replace intractable cost function that we care about (e.g., 0/1 loss) by tractable loss function (e.g., Perceptron loss) for sake of optimization / model fitting
- When evaluating a model (e.g., via cross-validation), use original cost / performance function



Surrogate optimization problem

Instead of

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l_{0/1}(\mathbf{w}; \mathbf{x}_i, y_i)$$

solve

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} l_P(\mathbf{w}; \mathbf{x}_i, y_i),$$

where

$$l_P = \max\{0, -y_i \mathbf{w}^T \mathbf{x}_i\}$$

is the Perceptron loss.

Gradient descent

Compute gradient of the perceptron loss function:

$$\hat{R}(\mathbf{w}) = \sum_{i=1}^{n} l_P(\mathbf{w}; \mathbf{x}_i, y_i) = \sum_{i=1}^{n} \max\{0, -y_i \mathbf{w}^T \mathbf{x}_i\}$$

$$\nabla \hat{R}(\mathbf{w}) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} l_P(\mathbf{w}; \mathbf{x}_i, y_i) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} \max\{0, -y_i \mathbf{w}^T \mathbf{x}_i\}$$

$$\nabla_{\mathbf{w}} l_P(\mathbf{w}; \mathbf{x}, y) = \begin{cases} 0 & \text{if } y\mathbf{w}^T\mathbf{x} \geq 0 \quad \text{(classified correctly)} \\ -y\mathbf{x} & \text{otherwise} \quad \text{(missclassified)} \end{cases}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \sum_{i \text{ missclassified by } \mathbf{w}_t} (-y_i \mathbf{x}_i)$$

Stochastic gradient descent

- Computing the gradient requires summing over all data
- ⚠ For large data sets, this is inefficient
- We can get a good (unbiased) gradient estimate by evaluating the gradient on few points

Stochastic gradient descent

- Computing the gradient requires summing over all data
- ⚠ For large data sets, this is inefficient
- We can get a good (unbiased) gradient estimate by evaluating the gradient on few points
- Extreme case: Evaluate on only one randomly chosen point

Stochastic gradient descent

- Computing the gradient requires summing over all data
- ⚠ For large data sets, this is inefficient
- We can get a good (unbiased) gradient estimate by evaluating the gradient on few points
- Extreme case: Evaluate on only one randomly chosen point

$\Rightarrow$ Stochastic gradient descent

Stochastic gradient descent

Stochastic gradient descent

- Start at an arbitrary $\mathbf{w}_0 \in \mathbb{R}^d$
- For $t = 0, 1, 2 \ldots$ do
  - Pick data point $(\mathbf{x}', y') \in \mathcal{D}$ from training set uniformly at random (with replacement), and set

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla l(\mathbf{w}_t; \mathbf{x}', y')$$

- Hereby, η_t is called learning rate
- Guaranteed to converge under mild conditions if

$$\sum_t \eta_t = \infty \qquad \sum_t \eta_t^2 < \infty \qquad \text{e.g. } \eta_t = \frac{1}{t} \text{ or } \eta_t = \min\left\{c, \frac{c'}{t}\right\}$$

The Perceptron algorithm

- Is just stochastic gradient descent (SGD) on the Perceptron loss function l_P with learning rate 1

The Perceptron algorithm

- Is just stochastic gradient descent (SGD) on the Perceptron loss function l_P with learning rate 1
- Theorem: If the data is linearly separable, the perceptron will obtain a linear separator

Demo: Perceptron

[classification/perceptron.ipynb]

Mini-batch SGD

- Using single points might have large variance in the gradient estimate, and hence lead to slow convergence
- Can reduce variance by averaging over the gradients w.r.t. multiple randomly selected samples (mini-batches)

Demo: SGD for regression

[classification/monitoring-sgd.ipynb]

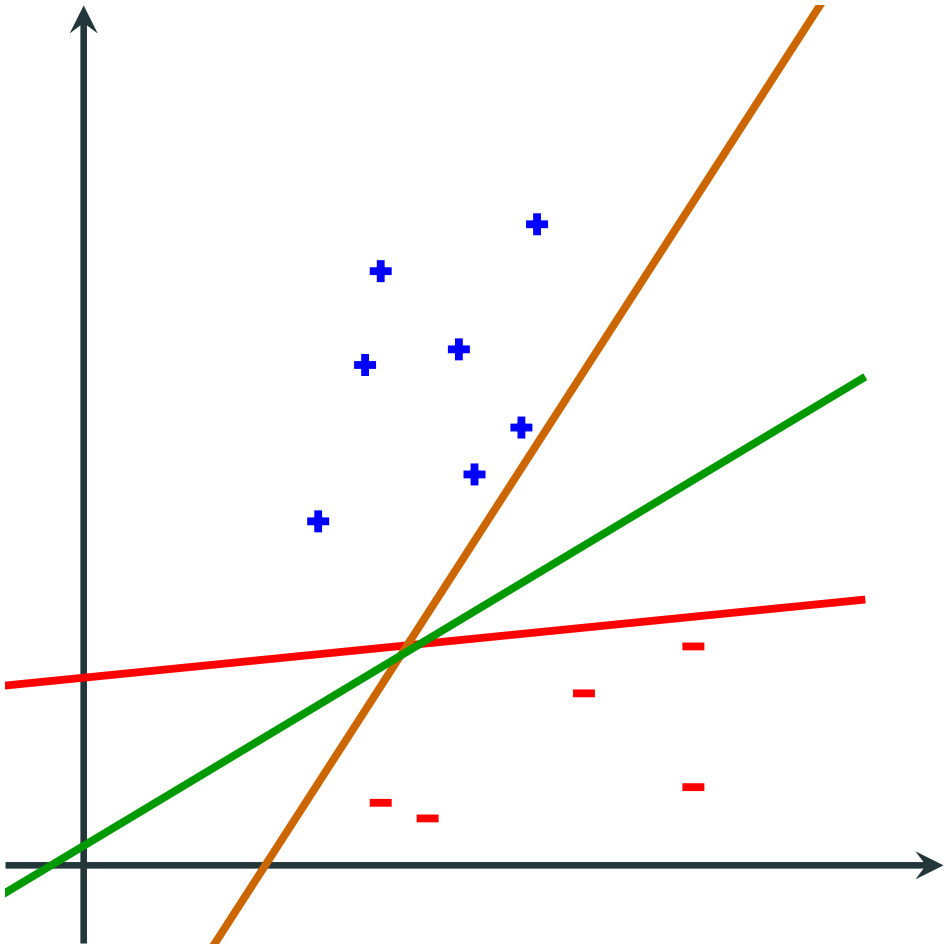
Adaptive learning rates

- Similar as in gradient descent, the learning rate is very important
- There exist various approaches for adaptively tuning the learning rate. Often times, these even use a different learning rate per features
- Examples: AdaGrad, RMSProp, Adam, . . .

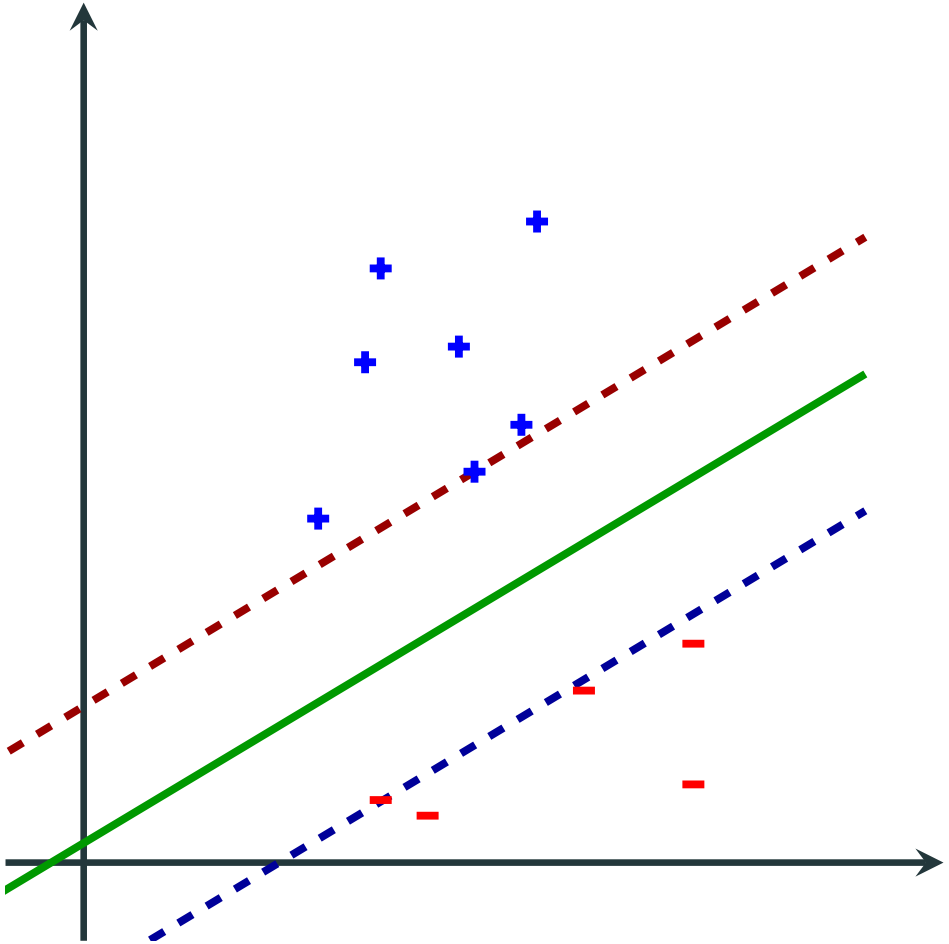
Supervised learning summary so far

Representation/ features	Linear hypotheses, non-linear hypotheses through feature transformations
Model/ objective	Loss-function (squared loss, ℓ_p loss, 0/1 loss, Perceptron loss) + Regularization (ℓ_2 norm)
Method	Exact solution, Gradient Descent, (mini-batch) SGD
Evaluation metric	Empirical risk = (mean) squared error
Model selection	k-fold cross-validation, Monte Carlo cross-validation

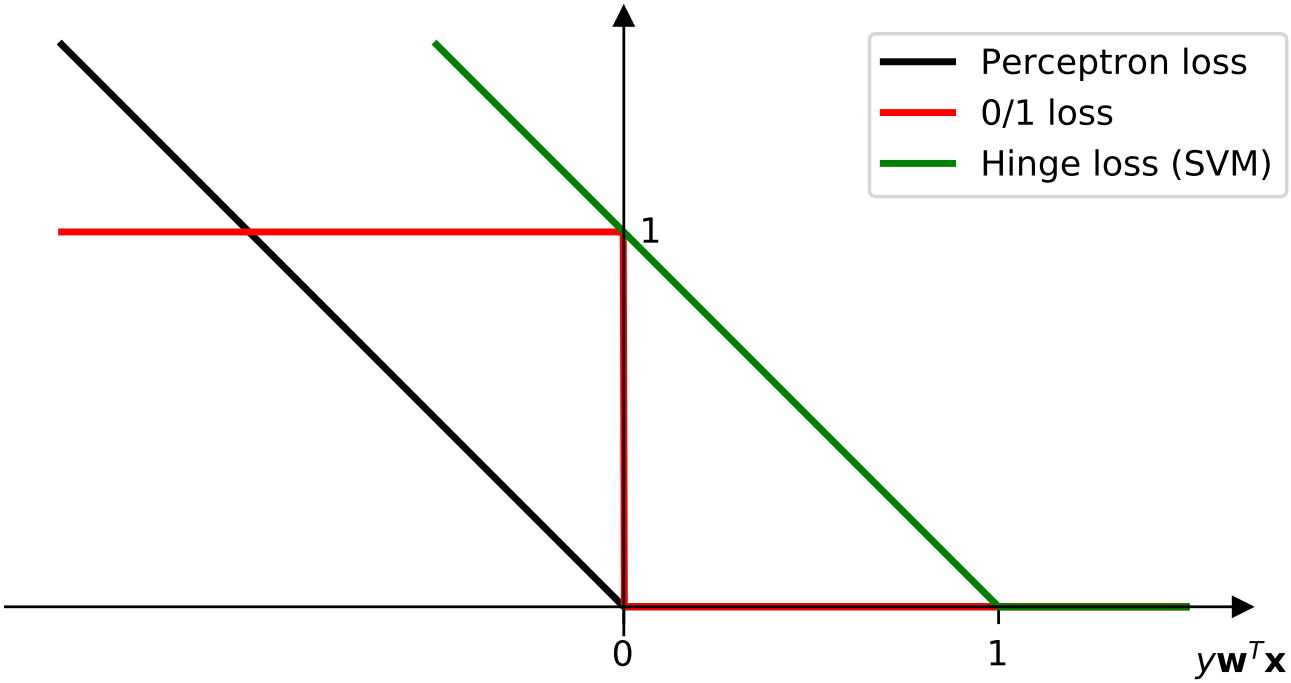
Which of these separators will the Perceptron “prefer”?



Support Vector Machines (SVMs): “max. margin” linear classification



Hinge vs. Perceptron loss



Hinge loss upper bounds #mistakes; encourages "margin":

$$l_H(\mathbf{w}; \mathbf{x}, y) = \max\{0, 1 - y\mathbf{w}^T\mathbf{x}\}$$

[classification/losses.py]

SVM vs. Perceptron

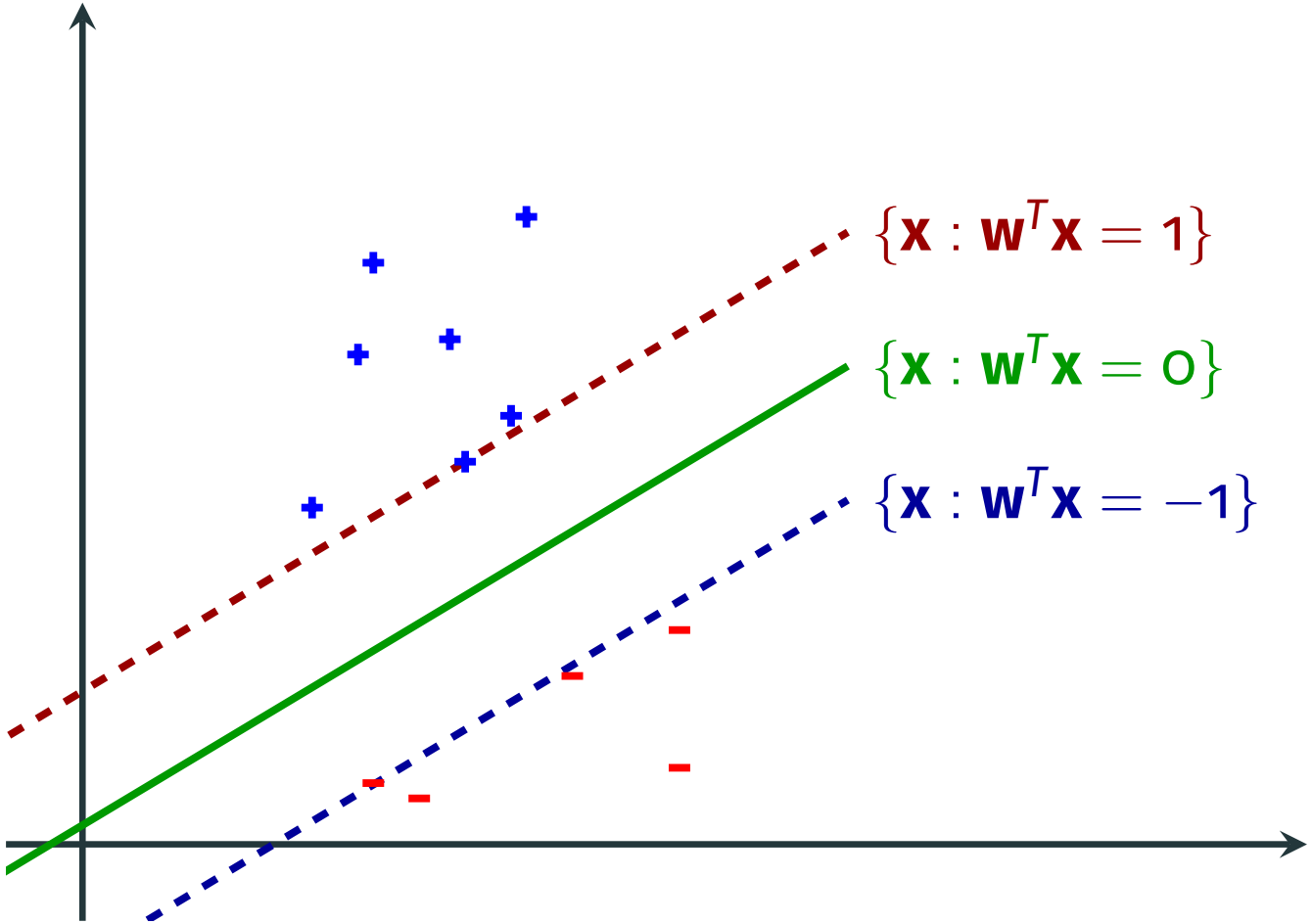
- Perceptron:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, -y_i \mathbf{w}^T \mathbf{x}_i\}$$

- Support vector machine (SVM):

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \mathbf{w}^T \mathbf{x}_i\} + \lambda \|\mathbf{w}\|_2^2$$

Support Vector Machines (SVMs): “max. margin” linear classification



Support vector machines

- Widely used, very effective linear classifier
- Almost like Perceptron. Only differences:
  - Optimize slightly different, shifted loss (hinge loss)
  - Regularize weights (like ridge regression)
- Can optimize via stochastic gradient descent
- Safe choice for learning rate: $\eta_t = \frac{1}{\lambda t}$
- More details in advanced machine learning literature

SGD for SVM

$$
\begin{aligned}
\hat{R}(\mathbf{w}) &= \sum_{i=1}^{n} l_H(\mathbf{w}; \mathbf{x}_i, y_i) + \lambda \|\mathbf{w}\|_2^2 \\
\nabla \lambda \|\mathbf{w}\|_2^2 &= 2\lambda \mathbf{w} \\
\nabla l_H(\mathbf{w}; \mathbf{x}_i, y_i) &= \nabla \max\{0, 1 - y_i \mathbf{w}^T \mathbf{x}_i\} \\
&= \begin{cases} 0 & \text{if } y_i \mathbf{w}^T \mathbf{x}_i \geq 1 \\ -y_i \mathbf{x}_i & \text{otherwise} \end{cases}
\end{aligned}
$$

Demo: Monitoring SGD

[classification/svm.ipynb]

Choosing the regularization parameter

- Can pick regularization parameter via cross-validation just like in linear regression!
- Note that instead of using the hinge-loss for validation, would use the target performance metric (e.g., accuracy)

Preview: Non-linear classification

- How can we find nonlinear classification boundaries?
- Similar as in regression, can use non-linear transformations of the inputs as feature vectors

Recall: Linear regression for polynomials

- We can fit non-linear classifiers via linear methods, using nonlinear features of our data (basis functions):

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) = \sum_{i=1}^{d} w_i \phi_i(\mathbf{x})$$

- For example: polynomials (in 1-D):

$$f(x) = \sum_{i=0}^{m} w_i x^i$$

- Higher dimensions $\rightarrow$ Monomials

Example

SVM in scikit-learn

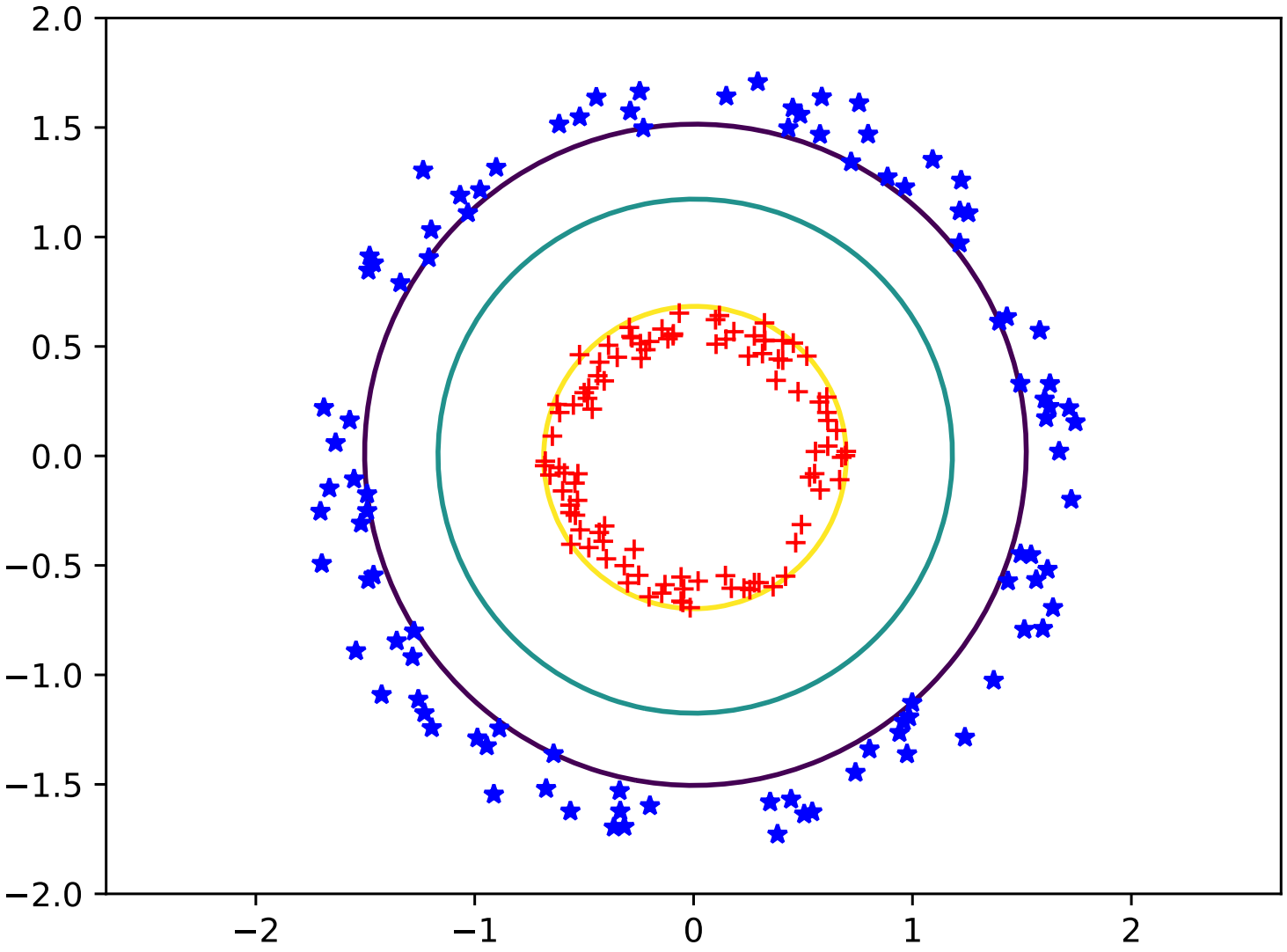
```
from sklearn.svm import LinearSVC

linearsvm = LinearSVC(C=1.0)
linearsvm.fit(X_train, y_train)
y_predict = linearsvm.predict(X_test)
```

Demo: Non-linear Classification with SVM

[classification/non-linear-classification.ipynb]

Demo: Non-linear Classification with SVM

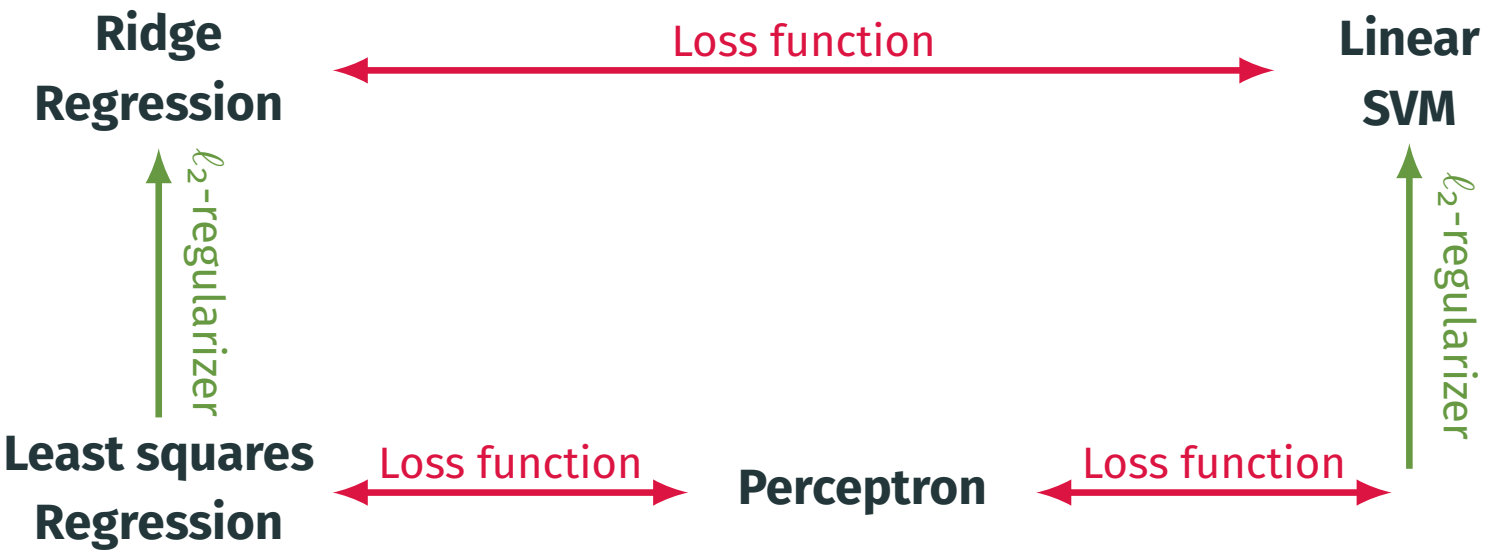


[classification/non-linear-classification.py]

What you need to know

- The Perceptron is an algorithm for linear classification
- It applies Stochastic Gradient Descent (SGD) on the Perceptron loss
- Mini-batches exploit parallelism, reduce variance
- The Perceptron loss is a convex surrogate function for the 0-1 (misclassification) loss
- It is guaranteed to produce a feasible solution (a linear separator) if the data is separable
- SGD is much more generally applicable
- Support Vector Machines (SVMs) are closely related to Perceptron; use hinge loss and regularization

Supervised learning big picture so far



Supervised learning summary so far

Representation/ features	Linear hypotheses, non-linear hypotheses through feature transformations
Model/ objective	Loss-function (squared loss, ℓ_p loss, 0/1 loss, Perceptron loss, Hinge loss) + Regularization (ℓ_2 norm)
Method	Exact solution, Gradient Descent, (mini-batch) SGD
Evaluation metric	Empirical risk = (mean) squared error, Accuracy
Model selection	k-fold cross-validation, Monte Carlo cross-validation