Loss Function For Regression

Recap: Linear Regression

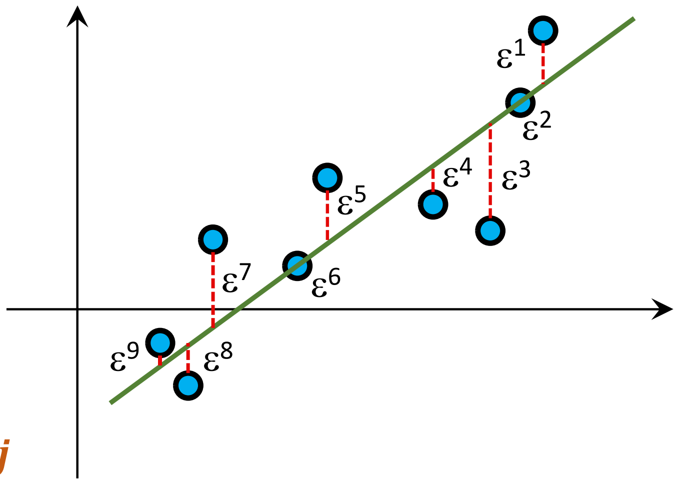
- For the n samples (x^j, y^j), to be fitted to a line

$$y^j = \alpha_0 + \alpha_1 x_1^j + \alpha_2 x_2^j + \cdots + \alpha_r x_r^j + \varepsilon^j,$$

where ε^j is the residual for the jth sample and r is the number of predictors.

- We measure goodness of fit by a loss function, defined on the residuals: $\varepsilon^j = y^j - f(x^j) = y^j - \hat{y}^j$

- A popular fit uses the least square approach that minimizes the sum of squared residuals
  - e.g., $L_{MSE} = \sum_j \varepsilon_j^2$



Loss Functions

- A loss function, $\boldsymbol{L}$, is defined as a mapping of $\boldsymbol{f}(\boldsymbol{x_i})$ with it's corresponding $\boldsymbol{y_i}$ to a real number $\boldsymbol{l} \in \mathbb{R}$, which captures the similarity between $\boldsymbol{f}(\boldsymbol{x_i})$ and $\boldsymbol{y_i}$

$$\boldsymbol{\varepsilon_i = y_i - f(x_i)}$$

❑ Mean Bias Error: $L_{MBE} = \frac{1}{N}\sum_{i=1}^{N} \varepsilon_i$

✓ Loss is continuous and differentiable.

✓ Helps to identify the direction of model bias

✓ Errors may cancel out, leading to zero loss

❑ Mean Absolute Error: $L_{MAE} = \frac{1}{N}\sum_i |\varepsilon_i|$

✓ Also called Laplace or l_1 loss.

✓ Least affected by outliers

✓ Not differentiable: Can create trouble

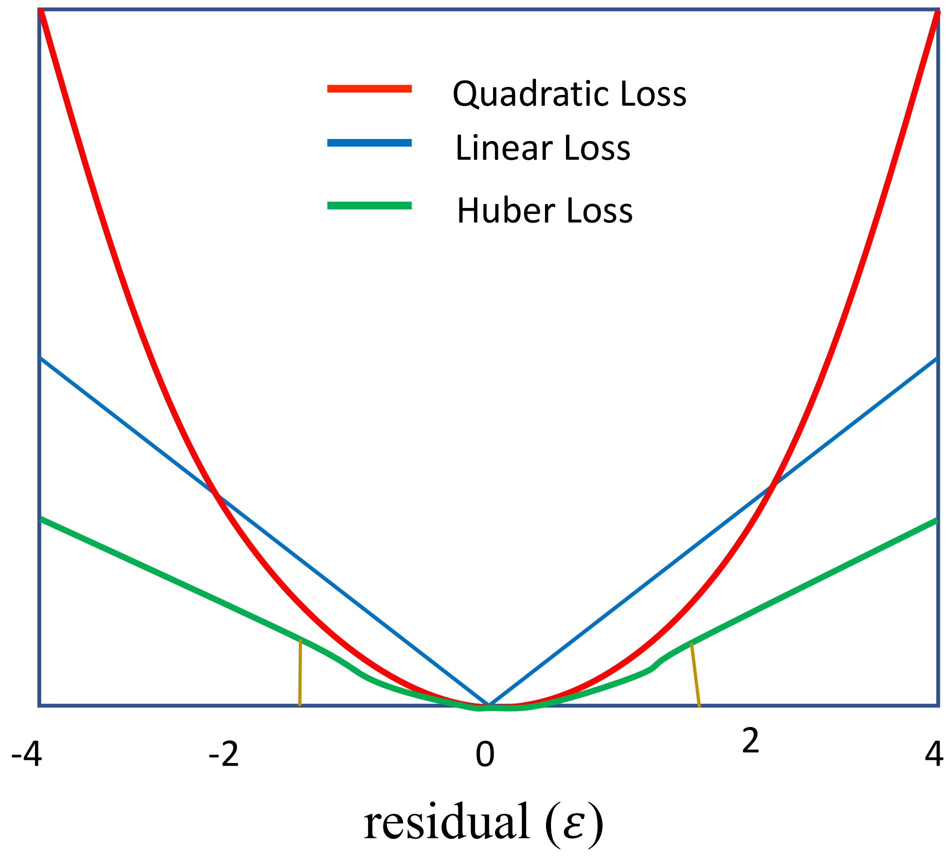
NOTE: the loss function is to capture the difference between the actual and predicted values for a single record whereas cost functions aggregate the difference for the entire training dataset.

Loss Functions

- ❑ Mean Squared Error: $L_{MSE} = \frac{1}{N}\sum_i \varepsilon_i^2$
  - ✓ Also called Quadratic or l_2 loss.
  - ✓ Significantly affected by outliers
  - ✓ Mathematically elegant optimization

- ❑ Root Mean Squared: $L_{RMSE} = \sqrt{\frac{1}{N}\sum_i \varepsilon_i^2}$
  - ✓ Less affected by outliers
  - ✓ More expensive to compute
  - ✓ Continuous and differentiable

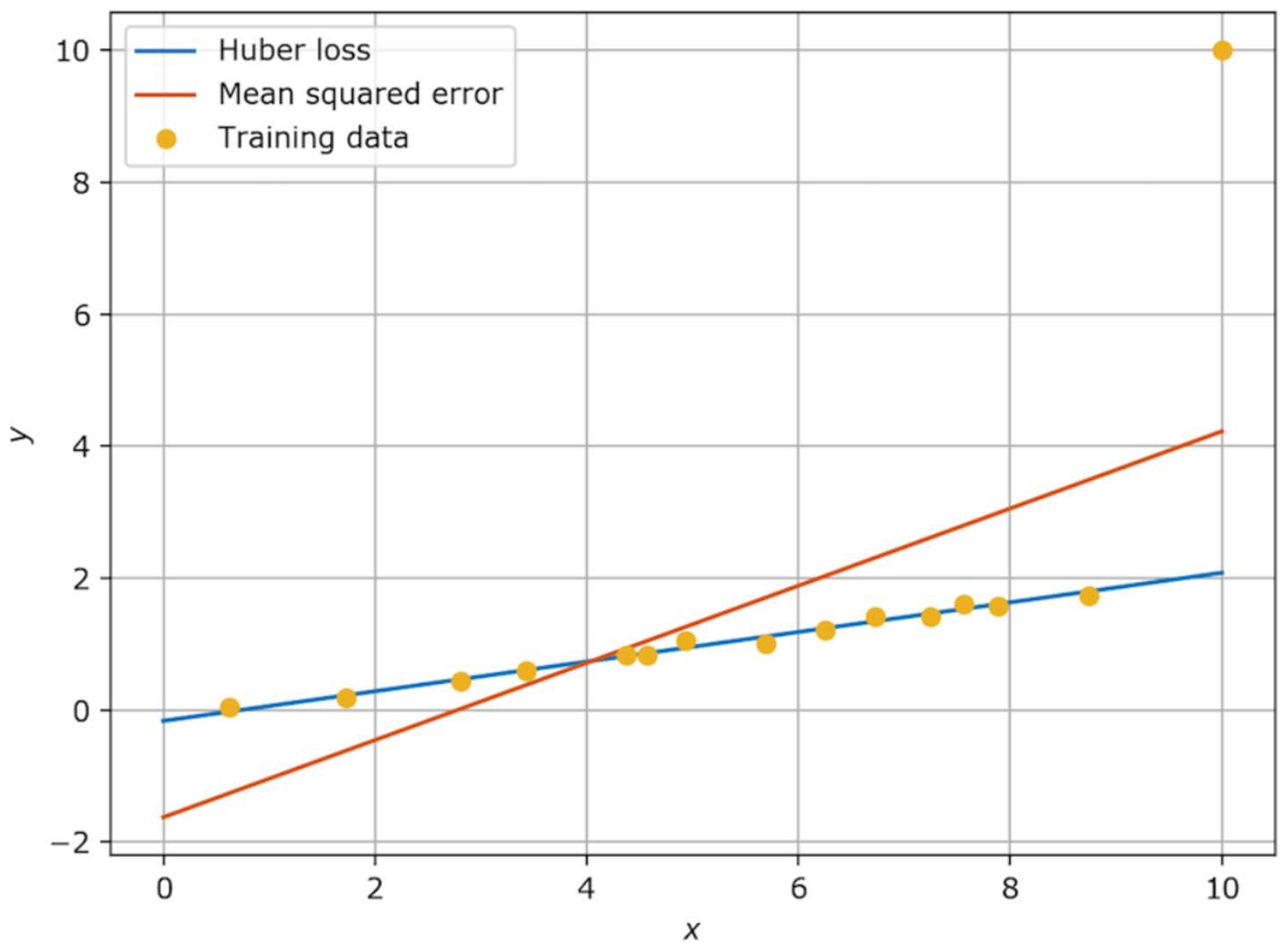
Huber loss: Robust Regression



- Combines the advantages of MSE and MAE.
- Mostly proportional to the absolute value, except for small errors, where it is proportional to the square of the error

$$L_\delta(\varepsilon_i) = \begin{cases} \frac{1}{2}\varepsilon_i^2, & |\varepsilon_i| \le \delta \\ \delta.\left(|\varepsilon_i| - \frac{1}{2}\delta\right), & |\varepsilon_i| > \delta \end{cases}$$

- Robustness of l_1 loss
- Differentiable
- δ is adjusted during training based on what is considered an outlier

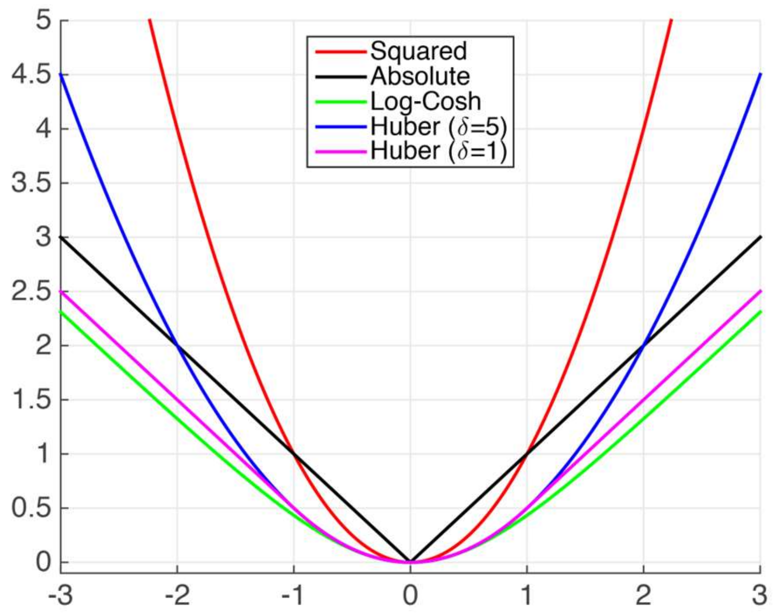


Model prediction with outlier in training data. Comparison of Huber loss and MSE.

Log-cosh loss

- Log-cosh loss is very close to the Huber loss in behavior, without the requirement of learning δ
  - ➢ linear behavior for very large values of error as $\log(\cosh x) \rightarrow |x| - \log 2$
  - ➢ quadratic behavior for small loss values, as $\log(\cosh x) \rightarrow \frac{x^2}{2}$.

$$L_{logcosh} = \frac{1}{N}\sum_{i=1}^{N} \log(\cosh(f(x_i) - y_i))$$



- More computationally expensive and less customizable compared to Huber Loss

RMSLE Loss

- The Root Mean Squared Logarithmic Error (RMSLE) loss is the RMSE of the log-transformed observed value y and log-transformed predicted value $\hat{y}$

$$\boldsymbol{L_{RMSLE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(y_i + 1) - \log(f(x_i) + 1))^2}$$

✓ Due to the properties of the logarithm, the error between the predicted and the actual values is relative, making the RMSLE more robust to outliers

y	55	64	72	450
$\hat{y}$	61	59	74	102

$RMSE = 4.655$ (174.047)

$RMSLE = 0.076$ (0.741)

RMSLE Loss

$$L_{RMSLE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(y_i+1)-\log(f(x_i)+1))^2} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\log(y_i+1)}{\log(f(x_i)+1)}\right)^2}$$

- ✓ Only considers the relative error between the Predicted and the actual value and the scale of the error is not significant
- ✓ RMSLE incurs a larger penalty for the underestimation of the Actual variable than the Overestimation.

y	100	10000
$\hat{y}$	80	8000

$RMSE = 20 \quad (2000)$

$RMSLE = 0.220 \quad (0.223)$

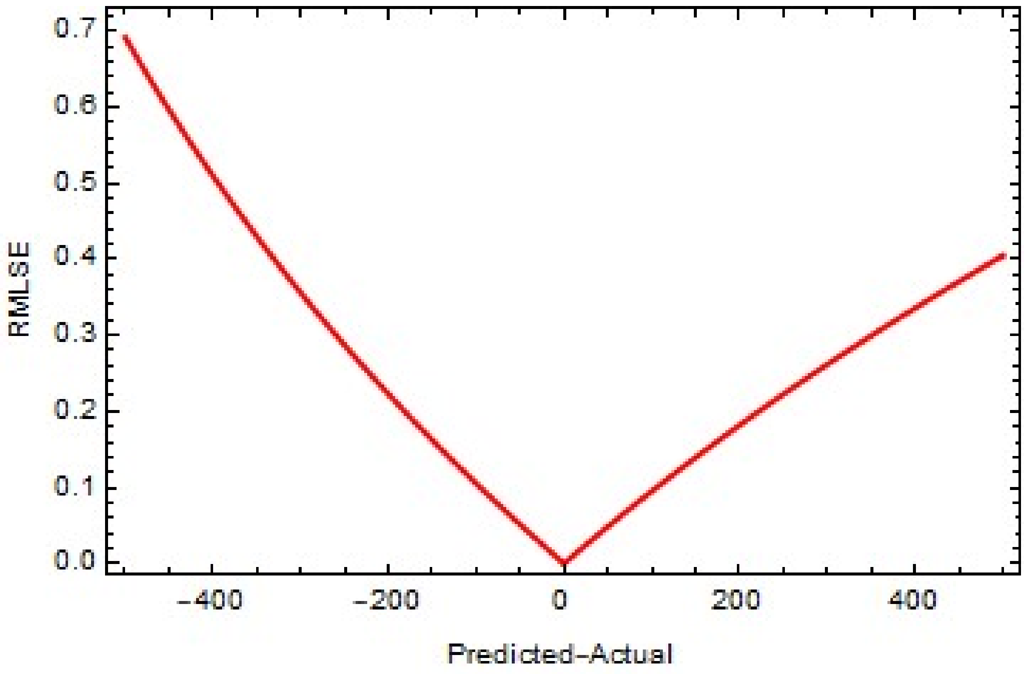
y	1000	1000
$\hat{y}$	600	1400

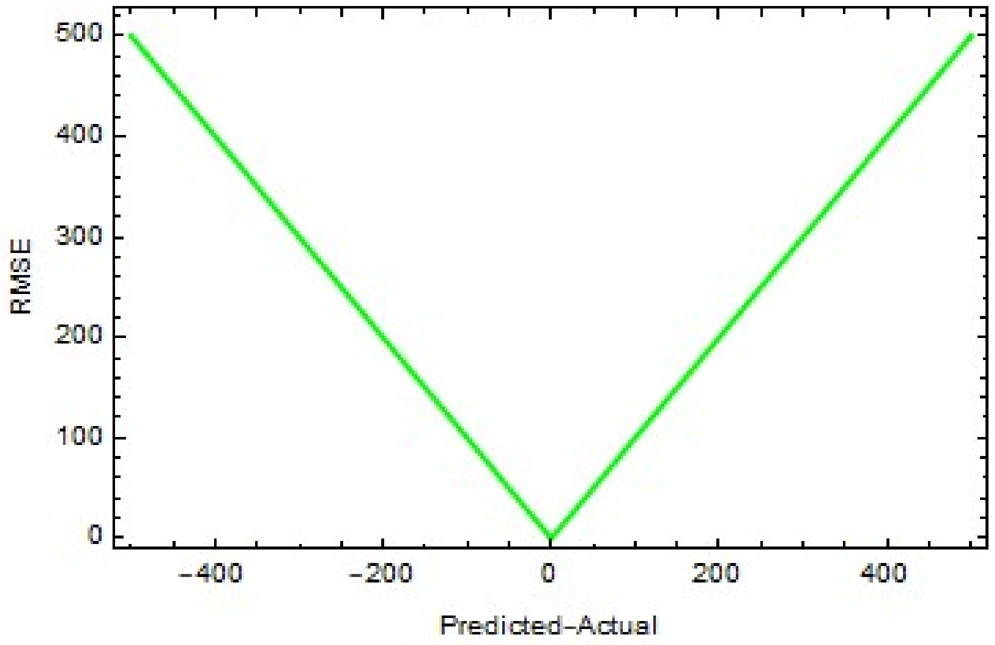
$RMSE = 400 \quad (400)$

$RMSLE = 0.510 \quad (0.33)$

RMSLE Loss

- Specially useful for business cases where the underestimation of the target variable is not acceptable but overestimation can be tolerated.





Regression Analysis

SVM, KNN, Decision Trees

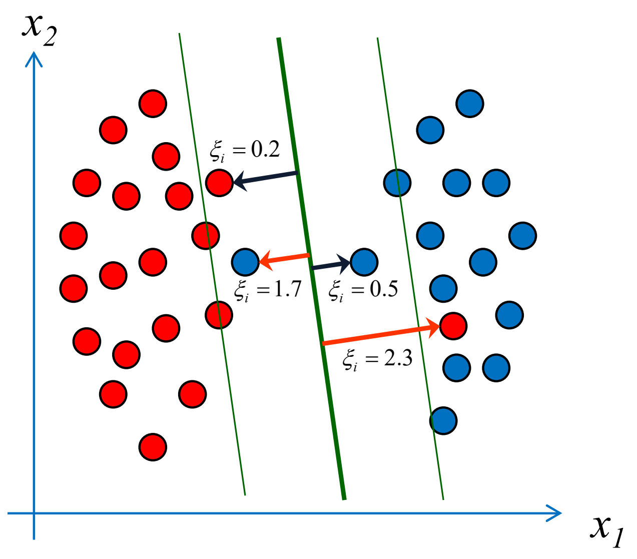
Recap: SVM Formulation

Minimize:

$$J(\mathbf{w}) = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{\boldsymbol{i}=1}^{\boldsymbol{N}} \xi_{\boldsymbol{i}}$$

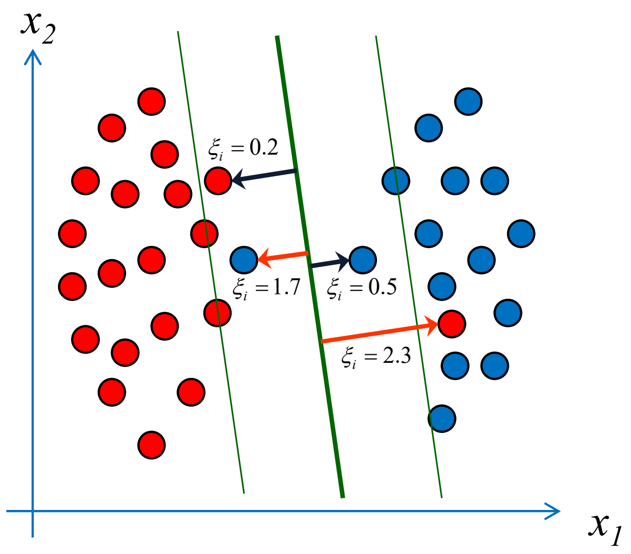
Subject to:

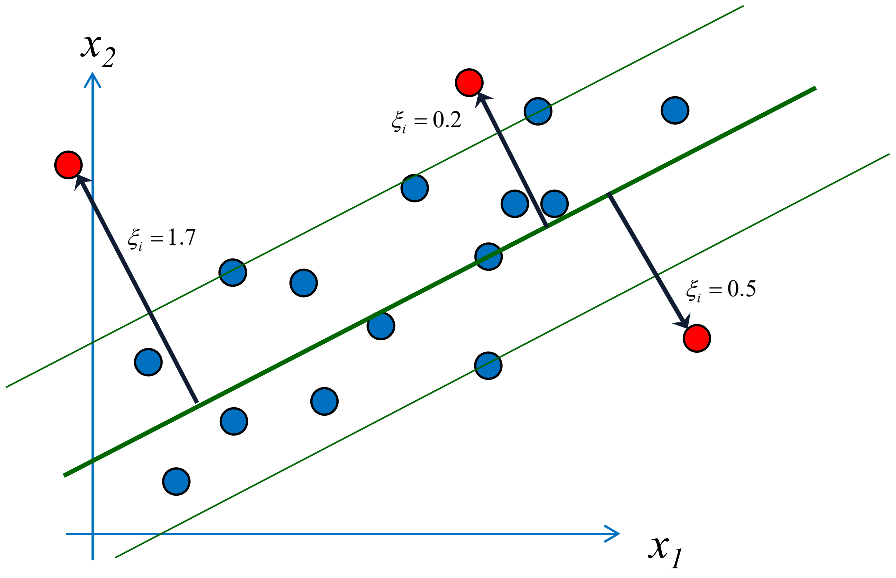
$$\forall_i, \qquad y_i\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i + b\right) > 1 - \xi_{\boldsymbol{i}}$$



- Solution is using QP solver
- **C** controls the relative importance of margin vs. training error

SVM Classification vs. Regression





For Regression:

- All samples should be within margin
- We want to minimize the margin
- Slack variables can be used to handle outliers

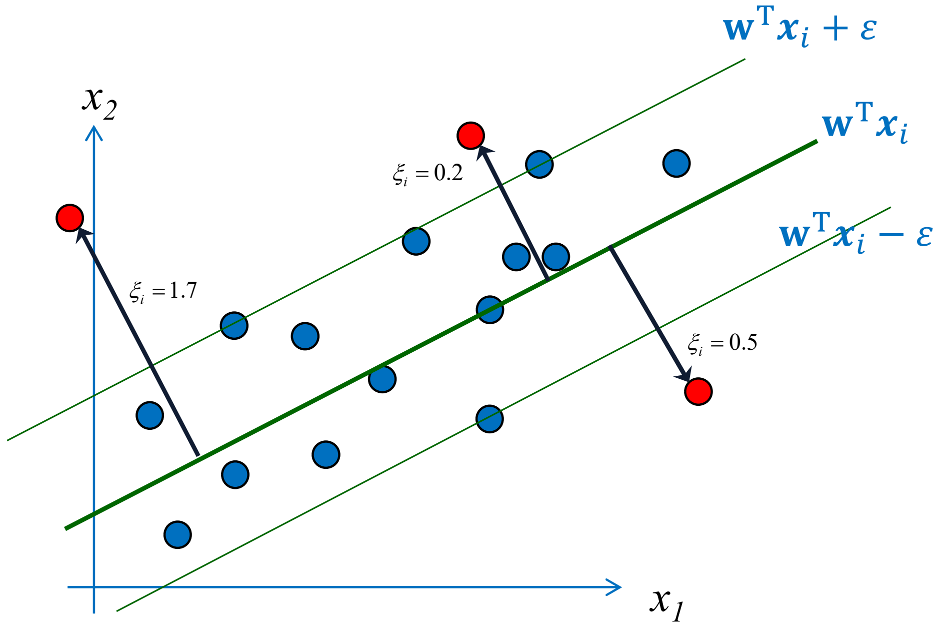
SVR Formulation

Minimize:

$$J(\mathbf{w}) = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

Subject to:

$$\forall_i, \left|y_i - \mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i\right| \leq \varepsilon + |\xi_i|$$



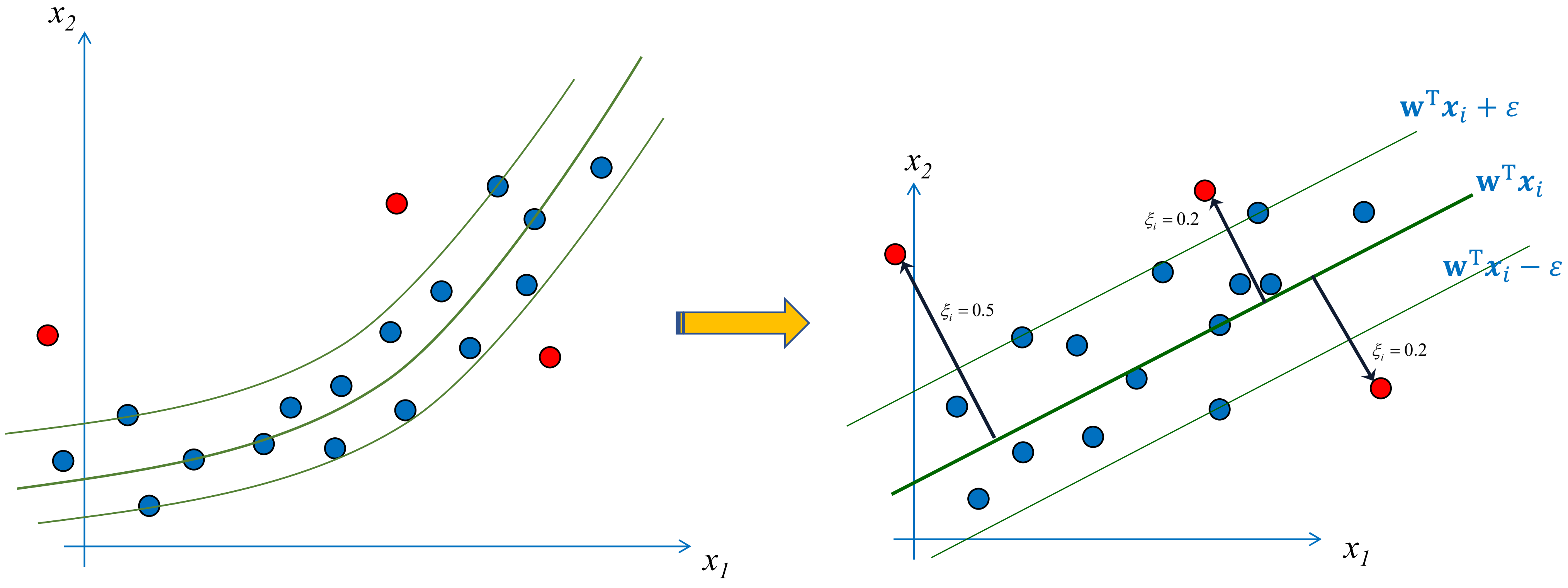
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i + \varepsilon$
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i$
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i - \varepsilon$
x_2
x_1
$\xi_i = 0.2$
$\xi_i = 1.7$
$\xi_i = 0.5$

- FurtherReading:

  Alex Smola and Bernhard Schölkopf, “A Tutorial on Support Vector Regression”, Statistics and Computing 14: 199–222, 2004.

Non-linear SVR

- We can use the kernel trick for regression as well



x_2
x_1
x_2
x_1
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i + \varepsilon$
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i$
$\mathbf{w}^{\mathrm{T}}\boldsymbol{x}_i - \varepsilon$
$\xi_i = 0.5$
$\xi_i = 0.2$
$\xi_i = 0.2$

sklearn.svm.SVR

from sklearn.svm import SVR

class SVR(, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=- 1)*

- **kernel:** {‘linear’, ‘poly’, ‘**rbf**’, ‘sigmoid’} or callable
- **degree:** int, default=**3** // degree of polynomial kernel
- **gamma:** {‘**scale**’, ‘auto’} or float // γ of poly, rbf or sigmoid kernels
- **C:** regularization parameter

For large datasets try using LinearSVR or SGDRegressor instead of SVR

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

k-NN Regression

- Key Idea: Similar inputs have similar predictions

Algorithm:

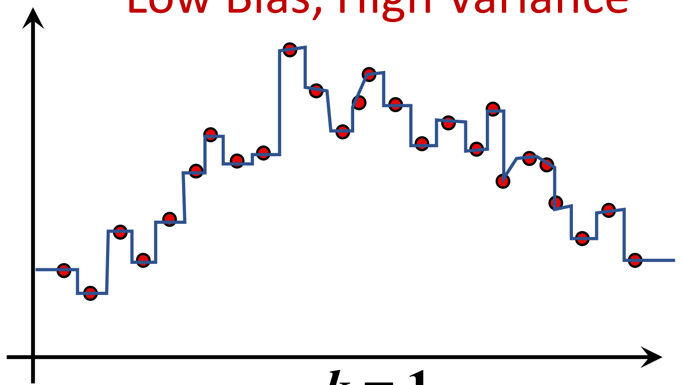
1. Find k-Nearest Neighbors of input $\boldsymbol{x}$: $\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_\mathrm{k}$.
2. Prediction for $\boldsymbol{x}$ is a function of the y-values of these points:

$$\hat{y} = f(p_{1y}, p_{2y}, \ldots, p_{ky})$$

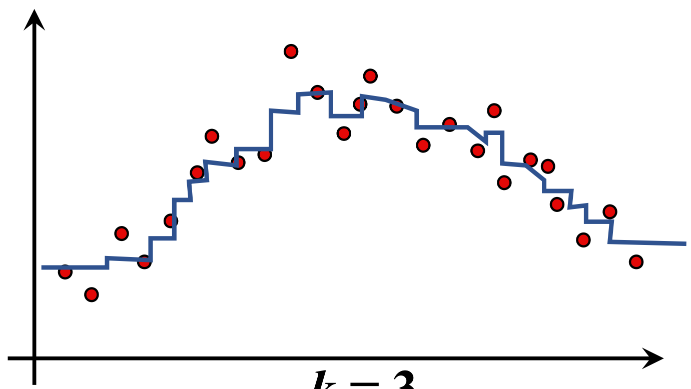
- Example 1: (simplest)
  - $\mathrm{k} = 1;\ \hat{y} = p_{1y}$
- Example 2:
  - $\mathrm{k} = 2;\ \hat{y} = (p_{1y} + p_{2y})/2$

Effect of k on prediction

Low Bias, High Variance

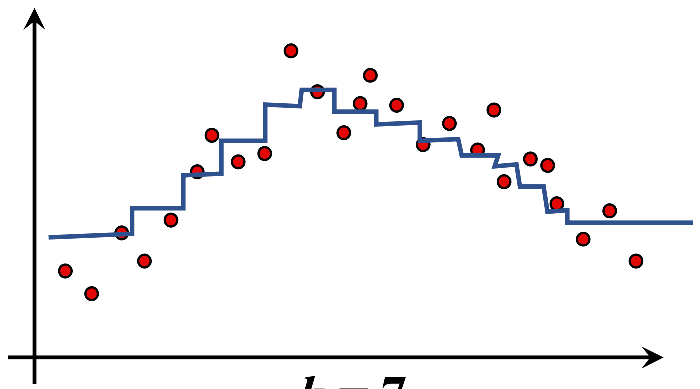


$k = 1$

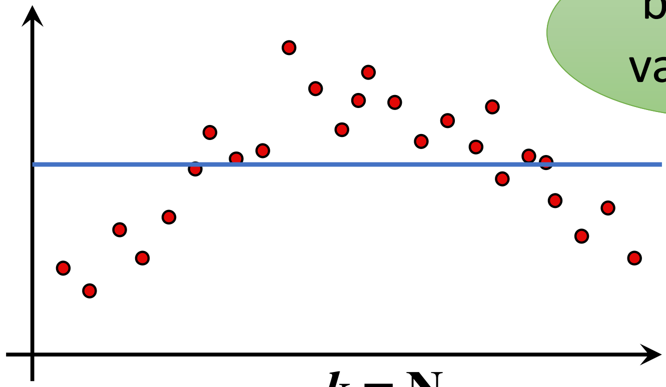


$k = 3$

bias vs. variance



$k = 7$



$k = \mathrm{N}$

High Bias, Low Variance

kNN Regression: Comments

- Weighted Regression
  - $\hat{y} = f(p_{1y}, p_{2y}, \dots, p_{ky})$ is a weighted combination of $\mathbf{p_{1y}}$ **..** $\mathbf{p_{ky}}$
  - Uniform weights vs Weights based on distance
    - One could use different distance metrics
    - Weights can be inverse of distance or Gaussian
- Computing nearest Neighbors
  - Exhaustive search
  - KD-Tree, Ball-Tree
- Non-parametric Method
- k controls the bias-variance tradeoff

sklearn.neighbors.KNeighborsRegressor

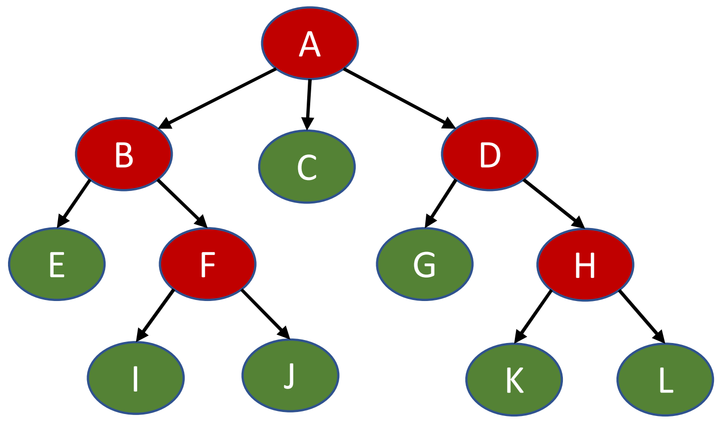
class sklearn.neighbors.KNeighborsRegressor (*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*)

- **n_neighbors:** Number of neighbors to use by default for k neighbors queries
- **weights:** {'uniform', 'distance'} or callable, default='uniform'
- **p:** int (for Minkowski), default=2
- **metric:** str or callable, default='minkowski'

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html

Recap: Decision Trees

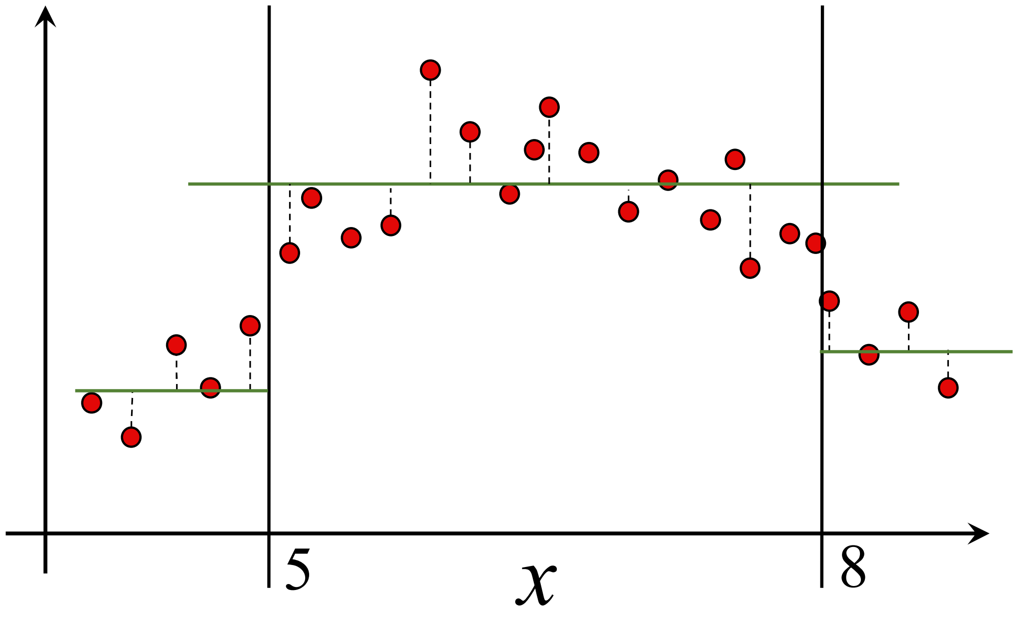
1. Find the best feature (and threshold) to split the training data
   - Use an objective metric like Entropy to decide
2. Partition the training data as per the selected feature and threshold
3. For each partition, if the entropy is low, stop.
   - else, Repeat the first two steps for that partition

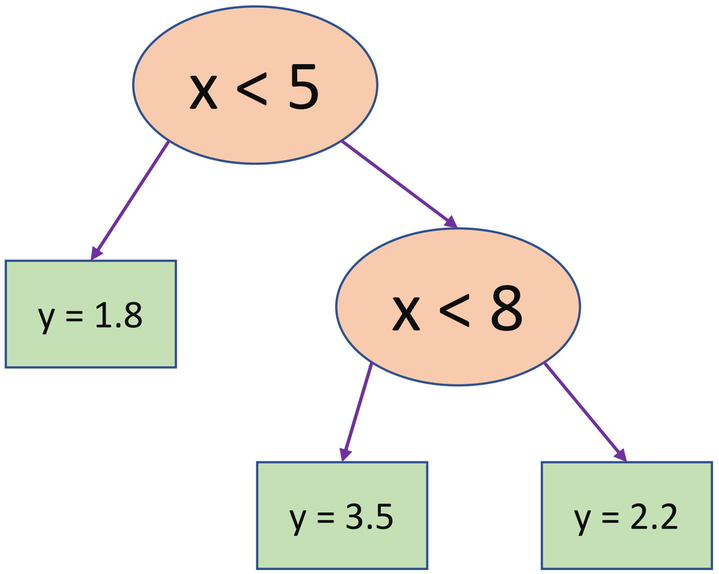


🟥 $Entropy > \tau$

🟩 $Entropy \leq \tau$

Example





- Compute sum of squared residuals for each split
- Choose the minimal one

Prediction and Split Criterion

- Value Predictor:
    - Common: Use the average value of training samples in a split
    - Minimum sum of squared residuals among all predictions
- Split Criterion
    - Find the variance of each potential split
    - Choose the split that minimizes total variance
        - Alternatively, maximize the reduction in variance

sklearn.tree.DecisionTreeRegressor

class sklearn.tree.DecisionTreeRegressor(, criterion='squared_error', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, ccp_alpha=0.0)*

- **criterion:** {“squared_error”, “friedman_mse”, “absolute_error”, “poisson”}
- **max_depth:** int, default=None
- **min_samples_split:** int or float, default=2
- **min_samples_leaf:** int or float, default=1

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html