

1. Classification
  - (a) K-Nearest Neighbor
  - (b) Support Vector Machines
  - (c) Adaboost
  - (d) Iterative Dichotomiser 3
  - (e) C4.5
  - (f) Naive Bayes
  - (g) Bagging
  - (h) Random Forest
  - (i) Logistic Regression
2. Neural Networks
  - (a) Perceptron
  - (b) Back-Propagation
  - (c) Learning Vector Quantization
  - (d) Self Organizing Map
3. Clustering Algorithms
  - (a) Hierarchial Agglomerative Clustering
  - (b) Hierarchial Division Clustering
  - (c) DBSCAN
4. Regression Algorithms
  - (a) Lasso Regression
5. Deep Learning Algorithms
  - (a) Deep Q-Learning
6. Other Methods
  - (a) Gradient Descent
  - (b) Gaussian Process
  - (c) PageRank

---

**Algorithm 1** k-Nearest Neighbor [Tay et al., 2014] link:16,31

---

**Input:** X: training data, Y: Class labels of X,  $x$ : unknown sample

**Output:** Class label of unknown sample

```
1: function CLASSIFY( $X, Y, x$ )
2:   for  $i = 1$  to  $m$  do
3:     Compute distance  $d(X_i, x)$ 
4:   end for
5:   Compute set  $I$  containing indices for the  $k$  smallest distances  $d(X_i, x)$ 
6:   Return majority label  $\{Y_i \text{ where } i \in I\}$ 
7: end function
```

---

---

**Algorithm 2** Adaboost [Schapire, 2014]

---

**Input:**

Training data  $\{(x_i, y_i)_{i=1}^N$  where  $x_i \in \mathbb{R}^k$  and  $y_i \in \{-1, 1\}\}$   
Large number of classifiers denoted by  $f_m(x) \in \{-1, 1\}$   
0-1 loss function  $I$  defined as

$$I(f_m(x, y)) = \begin{cases} 0, & \text{if } f_m(x_i) = y_i \\ 1, & \text{if } f_m(x_i) \neq y_i \end{cases} \quad (1)$$

**Output:** The final classifier

```
1: for  $i = 1$  to  $N$  do
2:   for  $i = 1$  to  $M$  do
3:     Fit weak classifier  $m$  to minimize the objective function:
4:      $\epsilon_m = \frac{\sum_{i=1}^N w_i^m I(f_m(x_i) \neq y_i)}{x^2 + 2x + 1}$ 
5:     where  $I(f_m(x_i) \neq y_i) = 1$  if  $f_m(x_i) \neq y_i$  and 0 otherwise
6:      $\alpha_m = \ln \frac{1 - \epsilon_m}{\epsilon_m}$ 
7:   end for
8:   for all  $i$  do
9:      $w_i^{m+1} = w_i^{(m)} e^{\alpha_m I(f_m(x_i) \neq y_i)}$ 
10:  end for
11: end for
```

---

---

**Algorithm 3** Adaboost [Hertzmann et al., 2015]

---

**Input:**

Training data  $\{(x_i, y_i)_{i=1}^N$  where  $x_i \in \mathbb{R}^k$  and  $y_i \in \{-1, 1\}\}$

**Output:** Weighted sum that represents the final output of the boosted classifier

```
1: Given Training data  $\{(x_i, y_i) \text{ where } y_i \in \{-1, 1\}\}$ 
2: initialize  $D_1$  = uniform distribution on training examples
3: for  $t = 1$  to  $T$  do
4:   Train weak classifier  $h_t$  on  $D_t$ 
5:   choose  $\alpha_t > 0$ 
6:   compute new distribution  $D_{t+1}$ :
7:   for all  $i$  do
8:     multiply  $D_t(x)$  by
```

$$\begin{cases} e^{-\alpha_t}, & (< 1) \text{ if } y_i = h_t(x_i) \\ e^{\alpha_t}, & (> 1) \text{ if } y_i \neq h_t(x_i) \end{cases} \quad (3)$$

```
9:   renormalize
10: end for
11: output final classifier  $H_{final}(x) = \text{sign}(\sum \alpha_t h_t(x))$ 
12: end for
```

---

---

**Algorithm 4** Adaboost4 [Hertzmann et al., 2015] Link:23,35,93,95

---

**Input:**

Training data  $\{(x_i, y_i)_{i=1}^N$  where  $x_i \in \mathbb{R}^k$  and  $y_i \in \{-1, 1\}\}$

**Output:** Weighted sum that represents the final output of the boosted classifier

```
1: Set uniform example weights.
2: for each base learner do
3:   Train base learner with weighted sample.
4:   Test base learner on all data.
5:   Set learner weight with weighted error.
6:   Set example weights based on ensemble predictions.
7: end for
```

---

---

**Algorithm 5** Random forest [Bernstein, 2016] Link:42

---

**Input:**  $S$ : training set,  $F$ : Features and number of trees in forest  $B$

**Output:** Constructed tree

```
1: function RANDOMFOREST( $S, F$ )
2:    $H \leftarrow \emptyset$ 
3:   for  $i \in 1, \dots, B$  do
4:      $S^{(i)} \leftarrow$  A bootstrap sample from  $S$ 
5:      $h_i \leftarrow \text{RANDOMIZEDTREELEARN}(S^{(i)}, F)$ 
6:      $H \leftarrow H \cup \{h_i\}$ 
7:   end for
8:   return  $H$ 
9: end function
10: function RANDOMIZEDTREELEARN( $S, F$ )
11:   At each node:
12:    $f \leftarrow$  a very small subset of  $F$ 
13:   Split on best feature in  $f$ 
14:   return The learned tree
15: end function
```

---

---

**Algorithm 6** Iterative Dichotomiser 3 [., 2015a] Link:14,61

---

**Input:**  $D$  : Training Data,  $X$  : Set of Input Attributes

**Output:** A decision tree

```
1: function ID3( $D, X$ )
2:   Let  $T$  be a new tree
3:   if all instances in  $D$  have the same class  $c$  then
4:     Label( $T$ ) =  $c$ ; Return  $T$ 
5:   end if
6:   if  $X = \emptyset$  or no attribute has positive information gain then
7:     Label( $T$ ) = most common class in  $D$ ; Return  $T$ 
8:   end if
9:    $X \leftarrow$  attribute with highest information gain
10:  Label( $T$ ) =  $X$ 
11:  for each value  $x$  of  $X$  do
12:     $D_x \leftarrow$  instances in  $D$  with  $X = x$ 
13:    if  $D_x$  is empty then
14:      Let  $T_x$  be a new tree
15:      Label( $T_x$ ) = most common class in  $D$ 
16:    else
17:       $T_x = \text{ID3}(D_x, X - \{x\})$ 
18:    end if
19:    Add a branch from  $T$  to  $T_x$  labeled by  $x$ 
20:  end for
21:  return  $T$ 
22: end function
```

---

---

**Algorithm 7** Perceptron [Brownlee, 2015d] Link:65

---

**Input:**  $ProblemSize, InputPatterns, iterations_{max}, learn_{rate}$

**Output:**  $Weights$

```
1: for  $i = 1$  to  $iterations_{max}$  do
2:    $Pattern_i \leftarrow \text{SelectInputPattern}(InputPatterns)$ 
3:    $Activation_i \leftarrow \text{ActivateNetwork}(Pattern_i, Weights)$ 
4:    $Output_i \leftarrow \text{TransferActivation}(Activation_i)$ 
5:    $UpdateWeights(Pattern_i, Output_i, learn_{rate})$ 
6: end for
7: Return  $Weights$ 
```

---

---

**Algorithm 8** Back-propagation [Brownlee, 2015a] Link:17,30,42

---

**Input:** *ProblemSize, InputPatterns, iterations<sub>max</sub>, learn<sub>rate</sub>*

**Output:** *Network*

- 1: *Network*  $\leftarrow$  *ConstructNetworkLayers*()
  - 2: *Network<sub>weights</sub>*  $\leftarrow$  *InitializeWeights*(*Network, ProblemSize*)
  - 3: **for**  $i = 1$  to *iterations<sub>max</sub>* **do**
  - 4:   *Pattern<sub>i</sub>*  $\leftarrow$  *SelectInputPattern*(*InputPatterns*)
  - 5:   *Output<sub>i</sub>*  $\leftarrow$  *ForwardPropagate*(*Pattern<sub>i</sub>, Network*)
  - 6:   *BackwardPropagateError*(*Pattern<sub>i</sub>, Output<sub>i</sub>, Network*)
  - 7:   *UpdateWeights*(*Pattern<sub>i</sub>, Output<sub>i</sub>, Network, learn<sub>rate</sub>*)
  - 8: **end for**
  - 9: **Return** *Network*
- 

---

**Algorithm 9** Back-propagation2 [Ng, ]

---

**Input:**

Training Set  $x^{(1)}, y^{(1)}, \dots, (x^{(m)}, y^{(m)})$

**Output:**

Gradient of the cost function

- 1:  $\Delta_{ij}^{(l)} = 0$  (for all  $l, i, j$ )
  - 2: **for**  $i = 1$  to  $m$  **do**
  - 3:   Set  $a^{(1)} = x^{(i)}$
  - 4:   Perform forward propagation to compute  $a^{(l)}$  for  $l = 2, 3, \dots, L$
  - 5:   Using  $y^{(i)}$ , compute  $\delta^{(L)} = a^{(L)} - y^{(i)}$
  - 6:   Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$
  - 7:    $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
  - 8: **end for**
  - 9:  $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij} + \lambda \theta_{ij}^{(l)}$  if  $j \neq 0$
  - 10:  $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}$  if  $j = 0$
  - 11:  $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$
- 

---

**Algorithm 10** Learning Vector Quantization [Brownlee, 2015c] Link : 50 and 58

---

**Input:** *ProblemSize, InputPatterns, iterations<sub>max</sub>, CodebookVectors<sub>num</sub>, learn<sub>rate</sub>*

**Output:** *CodebookVectors*

- 1: *CodebookVectors*  $\leftarrow$  *InitializeCodebookVectors*(*CodebookVectors<sub>num</sub>, ProblemSize*)
  - 2: **for**  $i = 1$  to *iterations<sub>max</sub>* **do**
  - 3:   *Pattern<sub>i</sub>*  $\leftarrow$  *SelectInputPattern*(*InputPatterns*)
  - 4:   *Bmu<sub>i</sub>*  $\leftarrow$  *SelectBestMatchingUnit*(*Pattern<sub>i</sub>, CodebookVectors*)
  - 5:   **for**  $Bmu_i^{attribute} \in Bmu_i$  **do**
  - 6:     **if**  $Bmu_i^{class} \equiv Pattern_i^{class}$  **then**
  - 7:        $Bmu_i^{attribute} \leftarrow Bmu_i^{attribute} + learn_{rate} \times (Pattern_i^{attribute} - Bmu_i^{attribute})$
  - 8:     **else**
  - 9:        $Bmu_i^{attribute} \leftarrow Bmu_i^{attribute} - learn_{rate} \times (Pattern_i^{attribute} - Bmu_i^{attribute})$
  - 10:   **end if**
  - 11: **end for**
  - end for**
  - Return** *CodebookVectors*
-

---

**Algorithm 11** Self Organizing Map [Brownlee, 2015b] Link:45

---

**Input:**  $InputPatterns, iterations_{max}, learn_{rate}, Grid_{width}, Grid_{height}$

**Output:**  $CodebookVectors$

```
1:  $CodebookVectors \leftarrow InitializeCodebookVectors(Grid_{width}, Grid_{height}, InputPatterns)$ 
2: for  $i = 1$  to  $iterations_{max}$  do
3:    $Learn_{rate}^i \leftarrow CalculateLearningRate(i, learn_{rate}^{init})$ 
4:    $neighborhood_{size}^i \leftarrow CalculateNeighborhoodSize(i, neighborhood_{init}^{size})$ 
5:    $Pattern_i \leftarrow SelectInputPattern(InputPatterns)$ 
6:    $Bmu_i \leftarrow SelectBestMatchingUnit(Pattern_i, CodebookVectors)$ 
7:    $Neighborhood \leftarrow Bmu_i$ 
8:    $Neighborhood \leftarrow SelectNeighbors(Bmu_i, CodebookVectors, neighborhood_{size}^i)$ 
9:   for  $Vector_i \in Neighborhood$  do
10:    for  $Vector_i^{attribute} \in Vector_i$  do
11:       $Vector_i^{attribute} \leftarrow Vector_i^{attribute} + learn_{rate} \times (Pattern_i^{attribute} - Vector_i^{attribute})$ 
12:    end for
13:  end for
14: end for
15: Return  $CodebookVectors$ 
```

---

---

**Algorithm 12** Hierarchial Agglomerative Algorithm [Stein, 2016a] Link:24,54

---

**Input:**

$\langle V, E, w \rangle$ .Weighted graph  
 $d_c$ .Distance measure for two clusters

**Output:**  $\langle V_T, E_T \rangle$ .Cluster hierarchy or dendogram

```
1:  $C = \{\{v \mid v \in V\}\}$  ▷ Initial Clustering
2:  $V_t = \{v_C \mid C \in C\}, E_T = \emptyset$  ▷ Initial Dendogram
3: while  $|C| > 1$  do
4:    $update\_distance\_matrix(C, G, d_c)$ 
5:    $\{C, C'\} = \underset{\{C_i, C_j\} \in C: C_i \neq C_j}{argmin} d_c(C_i, C_j)$ 
6:    $C = (C \setminus \{C, C'\}) \cup \{C \cup C'\}$  ▷ Merging
7:    $V_T = V_T \cup \{v_{C, C'}\}, E_T = E_T \cup \{\{v_{C, C'}, v_C\}, \{v_{C, C'}, v_{C'}\}\}$  ▷ Dendogram
8: end while
9: Return  $T$ 
```

---

---

**Algorithm 13** Hierarchial Agglomerative Algorithm 2 [., 2015c] Link:28,54

---

**Input:**

$\langle V, E, w \rangle$ .Weighted graph  
 $d_c$ .Distance measure for two clusters

**Output:**  $\langle V_T, E_T \rangle$ .Cluster hierarchy or dendogram

```
1: while More than one cluster remains do
2:   Compute the proximity graph if necessary
3:   repeat
4:     Merge the closest two clusters.
5:     Update the proximity matrix to reflect the proximity between the new cluster and the original clusters.
6:   end while
```

---

---

**Algorithm 14** Hierarchial Divisive Algorithm [Stein, 2016b]

---

**Input:**

$\langle V, E, w \rangle$ . Weighted graph  
 $d_c$ . Distance measure for two clusters

**Output:**  $\langle V_T, E_T \rangle$ . Cluster hierarchy or dendogram

```
1:  $C = \{V\}$  ▷ Initial Clustering
2:  $V_t = \{v_C \mid C \in C\}, E_T = \emptyset$  ▷ Initial Dendogram
3: while  $\exists C_x : (C_x \in C \wedge |C'| > 1)$  do
4:    $update\_distance\_matrix(C, G, d_c)$ 
5:    $\{C, C'\} = \underset{\{C_i, C_j\} : C_i \cup C_j = C_x \wedge C_i \cap C_j = \emptyset}{argmax} d_c(C_i, C_j)$ 
6:    $C = (C \setminus \{C, C'\}) \cup \{C \cup C'\}$  ▷ Merging
7:    $V_T = V_T \cup \{v_{C, C'}\}, E_T = E_T \cup \{\{v_{C, C'}, v_C\}, \{v_{C, C'}, v_{C'}\}\}$  ▷ Dendogram
8: end while
9: Return  $T$ 
```

---

---

**Algorithm 15** Hierarchial Divisive Algorithm [Stein, 2016b]

---

**Input:**

$\langle V, E, w \rangle$ . Weighted graph  
 $d_c$ . Distance measure for two clusters

**Output:**  $\langle V_T, E_T \rangle$ . Cluster hierarchy or dendogram

```
1: while More than one cluster remains do
2:   Create a new cluster by breaking the link corresponding to the largest distance (smallest similarity).
3: end while
```

---

---

**Algorithm 16** C4.5 [Dai and Ji, 2014] Link :22,55

---

**Input:**

$T$  : Training dataset  
 $S$  : Attributes

**Output:** decision tree  $Tree$ 

```
1: function C4.5( $T$ )
2:   if  $T$  is NULL then
3:     return failure
4:   end if
5:   if  $S$  is NULL then
6:     return  $Tree$  as a single node with most frequent class label in  $T$ 
7:   end if
8:   if  $|S| = 1$  then
9:     return  $Tree$  as a single node  $S$ 
10:  end if
11:  set  $Tree = \{\}$ 
12:  for  $a \in S$  do
13:    set  $Info(a, T) = 0$  and  $SplitInfo(a, T) = 0$ 
14:    compute  $Entropy(a)$ 
15:    for  $v \in values(a, T)$  do
16:      set  $T_{a,v}$  as the subset of  $T$  with attribute  $a = v$ 
17:       $Info(a, T) += \frac{|T_{a,v}|}{|T_a|} Entropy(a)$ 
18:       $SplitInfo(a, T) += -\frac{|T_{a,v}|}{|T_a|} \log \frac{|T_{a,v}|}{|T_a|}$ 
19:    end for
20:     $Gain(a, T) = Entropy(a) - Info(a, T)$ 
21:     $GainRatio(a, T) = \frac{Gain(a, T)}{SplitInfo(a, T)}$ 
22:  end for
23:  set  $a_{best} = argmax\{GainRatio(a, T)\}$ 
24:   $a_{best}$  into  $Tree$ 
25:  for  $v \in values(a_{best}, T)$  do call C4.5( $T_{a,v}$ )
26:  end for
27:  return  $Tree$ 
28: end function
```

---

---

**Algorithm 17** Gradient Descent [[., 2015b](#)] Link : 59

---

**Input:**

$f$   
starting value  $x_1$   
termination tolerances

**Output:**  $x_{maxIters}$

```
1: for  $i = 1$  to  $maxIters$  do
2:   Compute the search direction  $d_t = -\delta f(x_t)$ 
3:   if  $|d_T| < \epsilon_g$  then
4:     return "Converged to critical point", output  $x_t$ 
5:     Find  $\alpha_t$  so that  $f(x_t + \alpha_t d_t) < f(x_t)$ 
6:   end if
7:   if  $|\alpha_t d_T| < \epsilon_x$  then
8:     return "Converged in x", output  $x_t$ 
9:     Find  $\alpha_t$  so that  $f(x_t + \alpha_t d_t) < f(x_t)$ 
10:  end if
11:  Let  $x_{t+1} = x_t + \alpha_t d_t$ 
12: end for
13: Return "Max number of iterations reached", output  $x_{maxIters}$ 
```

---

---

**Algorithm 18** Naive Bayes [[., e](#)] Link: 9, 59

---

**Input:**

$C$  : A fixed set of classes  
 $D$  : Documents

**Output:** Category(Class) of the Documents

```
1: function TRAINMULTINOMIALNB( $C, D$ )
2:    $V \leftarrow EXTRACTVOCABULARY(D)$ 
3:    $N \leftarrow COUNTDOCS(D)$ 
4:   for each  $c \in C$  do
5:      $N_c \leftarrow COUNTDOCSINCLASS(D, c)$ 
6:      $prior|c| \leftarrow N_c / N$ 
7:      $text_c \leftarrow CONCATENATE TEXT OF ALL DOCS IN CLASS(D, C)$ 
8:     for each  $t \in V$  do
9:        $condprob|t||c| \leftarrow \frac{T_{ct}+1}{\sum_{t'} (T_{ct'}+1)}$ 
10:    end for
11:  end for
12:  return  $V, prior, condprob$ 
13: end function
14: function APPLYMULTINOMIALNB( $C, D, prior, condprob, d$ )
15:    $W \leftarrow EXTRACTTOKENS FROM DOC(V, d)$ 
16:   for each  $c \in C$  do
17:      $score|c| \leftarrow \log prior|c|$ 
18:     for each  $t \in W$  do
19:        $score|c|+ = \log condprob|t||c|$ 
20:     end for
21:   end for
22:   return  $\arg \max_{c \in C} score|c|$ 
23: end function
```

---

---

**Algorithm 19** Lasso Regression [[Wang et al.,](#)] Link:30,33,124

---

**Input:**

$ipy$  : Inner product vector,  $ipy_i = \langle y, X_{\cdot i} \rangle$   
 $ipx$  : Inner product matrix,  $ipx_{ij} = \langle X_{\cdot i}, X_{\cdot j} \rangle$   
 $\lambda$  : Penalty parameter  
 $N$  : Number of samples

**Output:**  $\beta$  : Regression parameter vector

```
1: function FASTLASSO( $ipy, ipx, \lambda, N$ )  
2:   stop_thr ▷ Threshold for stopping iteration  
3:    $p \leftarrow \text{length}(ipy)$   
4:    $\beta \leftarrow 0$  with length  $p$   
5:    $gc \leftarrow 0$  with length  $p$   
6:   while  $\text{difBeta}_{max} \geq \text{stop\_thr}$  do  
7:      $\text{difBeta}_{max} \leftarrow 0$   
8:     for  $j = 1 \leftarrow p$  do  
9:        $z \leftarrow (ipy[j] - gc[j])/N + \beta[j]$   
10:       $\text{beta\_tmp} \leftarrow \max(0, z - \lambda) - \max(0, -z - \lambda)$   
11:       $\text{difBeta} \leftarrow \text{beta\_tmp} - \beta[j]$   
12:       $\text{difabs} \leftarrow \text{abs}(\text{difBeta})$   
13:      if  $\text{difabs} > 0$  then  
14:         $\beta[j] \leftarrow \text{beta\_tmp}$   
15:         $gc \leftarrow gc + ipx[j] \times \text{difBeta}$   
16:         $\text{difBeta}_{max} = \max(\text{difBeta}_{max}, \text{difabs})$   
17:      end if  
18:    end for  
19:  end while  
20: end function
```

---

---

**Algorithm 20** Bagging [[., a](#)] Link:57

---

**Input:**

$B$ : the number of bags or base hypotheses  
 $L$ : Base Learning Algorithm

**Output:** New Training Sets

```
1: function BAGGING( $examples, B, L$ )  
2:   for  $i = 1$  to  $B$  do  
3:      $examples_i \leftarrow$  a bootstrap sample of  $examples$   
4:   end for  
5:   Compute set  $I$  containing indices for the  $k$  smallest distances  $d(X_i, x)$   
6:    $h_i \leftarrow \text{applyLtoexamples}_i$   
7:   Return  $h_1, h_2, \dots, h_B$   
8: end function
```

---



---

**Algorithm 21** Deep Q-Learning with Experience Replay [Mnih et al., 2013] Link: 91

---

**Input:**

D: data set

Q: Action-Value Function

**Output:** New Training Sets

```

1: for  $i = 1$  to  $M$  do
2:   Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi = \phi(s_1)$ 
3:   for  $i = 1$  to  $T$  do
4:     With probability  $\epsilon$  select a random action  $a_t$  otherwise select  $a_t = \max_a Q * (\phi(s_t).a : \theta)$ 
5:     Execute action  $a_t$  in emulator and observe reward  $r$  and image  $x_{t+1}$ 
6:     Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
7:     Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
8:     Set  $y_j =$ 

$$\begin{cases} r_j, & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for terminal } \phi_{j+1} \end{cases} \quad (5)$$


$$\quad (6)$$

9:     Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to the following equation
10:

$$\Delta_{\theta} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \epsilon[(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \Delta_{\theta_i} Q(s, a; \theta_i)]$$

11:   end for
12: end for

```

---



---

**Algorithm 22** PageRank [., f] Link:24,64,98,99

---

**Input:**

G: inlink file

iteration: Number of iteration

**Output:** PageRank

```

1: function PAGERANK( $G, iteration$ )
2:    $d \leftarrow 0.85$  ▷ damping factor: 0.85
3:    $oh \leftarrow G$  ▷ get outlink hash from G
4:    $ih \leftarrow G$  ▷ get inlink hash from G
5:    $N \leftarrow G$  ▷ get number of pages from G
6:   for all  $p$  in the graph do
7:      $opg[p] \leftarrow \frac{1}{N}$ 
8:   end for
9:   while  $iteration > 0$  do
10:     $dp \leftarrow 0$ 
11:    for all  $p$  that has no out-links do
12:       $dp \leftarrow dp + d * \frac{opg[p]}{N}$ 
13:    end for
14:    for all  $p$  in the graph do
15:       $npg[p] \leftarrow dp + \frac{[1-d]}{N}$ 
16:      for all  $ip$  in  $ih[p]$  do
17:         $npg[p] \leftarrow dp + \frac{d * opg[ip]}{oh[ip]}$ 
18:      end for
19:    end for
20:     $opg \leftarrow npg$ 
21:     $iteration \leftarrow iteration - 1$ 
22:  end while
23: end function

```

---

---

**Algorithm 23** K means [., b] Link:25,29,40,104

---

**Input:**

$E = \{e_1, e_2, \dots, e_k\}$

$k$ (number of clusters)

$MaxIters$ (limit of iterations)

**Output:**

$C = \{c_1, c_2, \dots, c_k\}$

$L = \{l(e) | e = 1, 2, \dots, n\}$  (set of cluster labels of E)

```
1: for each  $c_i \in C$  do
2:    $c_i \leftarrow e_j \in E$  (e.g random selection)
3: end for
4: for each  $e_i \in E$  do
5:    $l(e_i) \leftarrow \text{argminDistance}(e_i, c_j) j \in \{1..k\}$ 
6: end for
7:  $changed \leftarrow false$ 
8:  $iter \leftarrow 0$ 
9: while  $changed = true$  and  $iter \leq MaxIters$  do
10:   for  $c_i \in C$  do
11:      $UpdateCluster(c_i)$ 
12:   end for
13:   for  $c_i \in E$  do
14:      $minDist \leftarrow \text{argminDistance}(e_i, e_j) j \in \{1..k\}$ 
15:     if  $minDist \neq l(e_i)$  then
16:        $l(e_i) \leftarrow minDist$ 
17:        $changed \leftarrow true$ 
18:     end if
19:   end for
20:    $iter++$ 
21: end while
```

---

---

**Algorithm 24** DBSCAN link:13,76 [Ram et al., 2010]

---

**Input:** $D$ : Data $\epsilon$ : Threshold distance $MinPts$  : Minimum number of points required to form a cluster**Output:** Clustered Data

```
1: function DBSCAN( $D, \epsilon, minPts$ )
2:    $C = 0$ 
3:   for each point  $P$  in dataset  $D$  do
4:     if  $P$  is visited then
5:       continue next point
6:     end if
7:     mark  $P$  as visited
8:      $NeighborPts = regionQuery(P, \epsilon)$ 
9:     if  $sizeof(NeighborPts) < MinPts$  then
10:      mark  $P$  as NOISE
11:    else
12:       $C =$  next cluster
13:       $expandCluster(P, NeighborPts, C, \epsilon, MinPts)$ 
14:    end if
15:  end for
end function
function EXPANDCLUSTER( $P, NeighborPts, C, \epsilon, MinPts$ )
18: add  $P$  to Cluster  $C$ 
19: for each point  $P'$  in  $NeighborPts$  do
20:   if  $P'$  is not visited then
21:     mark  $P'$  as visited
22:      $NeighborPts' = regionQuery(P', \epsilon)$ 
23:     if  $sizeof(NeighborPts') \geq MinPts$  then
24:        $NeighborPts = NeighborPts$  joined with  $NeighborPts'$ 
25:     end if
26:   end if
27:   if  $P'$  is not yet member of any cluster then
28:     add  $P'$  to cluster  $C$ 
29:   end if
30: end for
end function
function REGIONQUERY( $P, \epsilon$ )
33: return all points within  $P$ 's  $\epsilon$  neighborhood
end function
```

---

---

**Algorithm 25** Principle Component Analysis [., 2014] Link:41,69,72

---

**Input:** $x_1, \dots, x_n$  d length vector  $k$ **Output:** Transform matrix  $R$ 

```
1:  $X \leftarrow n \times d$  data matrix with  $x_i$  in each row
2:  $\bar{x} \leftarrow \frac{1}{n} \sum_{i=1}^n x_i$ 
3:  $X \leftarrow$  subtract  $\bar{x}$  from each row  $x_i$  in  $X$ 
4:  $COV \leftarrow \frac{1}{n-1} X^T \times X$  Compute eigenvalue  $e_1, \dots, e_d$  of  $COV$ , and sort them
5: Compute matrix  $V$  which satisfy  $V^{-1} \times COV \times V = D$ ,  $D$  is the diagonal matrix of eigenvalue of  $COV$ 
6:  $R \leftarrow$  the first  $k$  column of  $V$ 
```

---

---

**Algorithm 26** Logistic Regression [., d] link:66,67,71

---

**Input:**

Training data of the form  $\{(x_1, 1), (x_2, 0), \dots\}$

$x$ : unknown sample

**Output:** The output is a probability that the given input point belongs to a certain class

1:  $0 \leftarrow \beta$

2: Compute  $y$  by setting its elements to

$$y = \begin{cases} 1, & \text{if } g_i = 1 \\ 0, & \text{if } g_i = 2 \end{cases} \quad (7)$$

$i = 1, 2, \dots, N$

3: Compute  $p$  by setting its elements to

$$p(x_i, \beta) = \frac{e^{\beta^T x_i}}{1 + e^{\beta^T x_i}} \quad (8)$$

$i = 1, 2, \dots, N$

4: Compute the diagonal matrix  $W$ . The  $i$ th diagonal element is  $p(x_i, \beta)(1 - p(x_i, \beta))$

5:  $z \leftarrow X\beta + W^{-1}(y - p)$

6:  $\beta \leftarrow (X^T W X)^{-1} X^T W z$

7: If the stopping criteria, stop; otherwise go back to step 3

---

---

**Algorithm 27** Gaussian Process [., c] Link:78, 122

---

**Input:**

$X = \begin{bmatrix} x_1^T \\ \dots \\ x_n^T \end{bmatrix} \in \mathbb{R}^{n \times D}$ ,  $m$  training inputs

$y = \begin{bmatrix} y_1^T \\ \dots \\ y_n^T \end{bmatrix} \in \mathbb{R}^n$

$k(\cdot, \cdot) : \mathbb{R}^{D \times D}$

$x_*$  test input

$\sigma^2$  noise level on the observations

$$[y(x) = f(x) + \epsilon, \epsilon \sim N(0, \sigma^2)]$$

**Output:**

$f_*$

$cov(f_*)$

1:  $K \in \mathbb{R}^{n \times n}$  Gram matrix.  $K_{ij} = k(x_i, x_j)$

$$k(x_*) = k_* = k(X, x_*) = \begin{bmatrix} k(x_1, x_*) \\ \dots \\ k(x_n, x_*) \end{bmatrix} \in \mathbb{R}$$

2:  $\alpha = (K + \sigma^2 \mathbb{I}_n)^{-1} y$

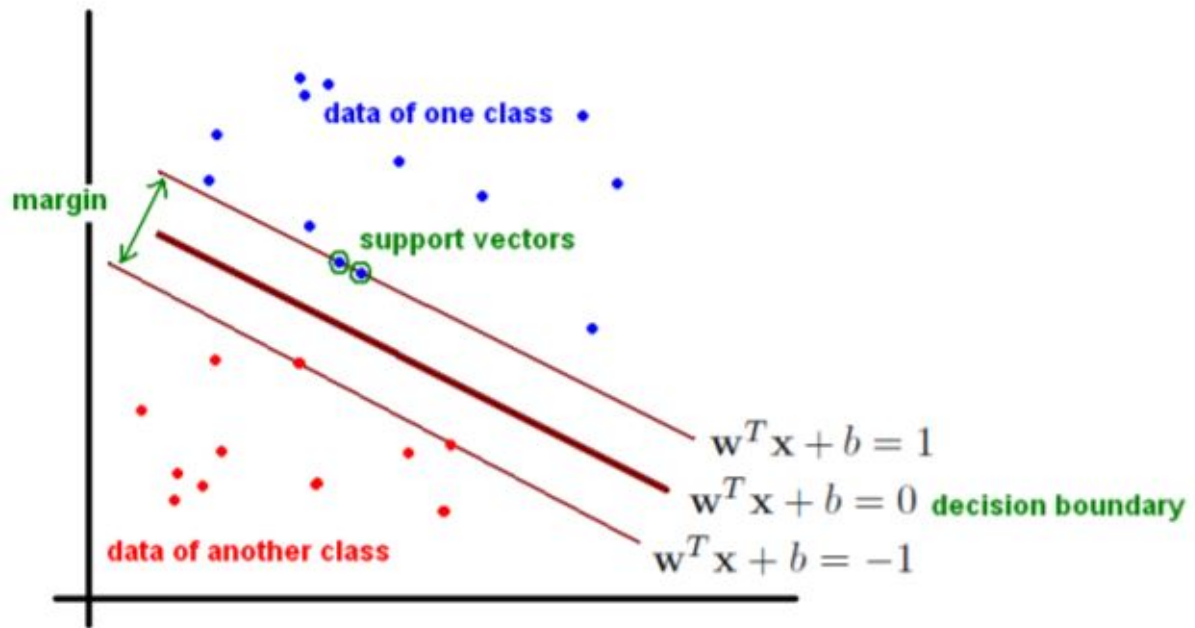
3:  $f_* = k_*^T \alpha \in \mathbb{R}$

4:  $cov(f_*) = k(x_*, x_*) - k_*^T [K + \sigma^2 \mathbb{I}_n]^{-1} k_*$

---

**Input:**

Set of  $N$  input-output pairs  $\{x, y\}^{N_1}$   $x$ : input vectors of the same dimension and  $y$  : set of output target labels  $y_i = \{0, 1\}$



## 1 Simple case : linearly-separable data, binary classification

Goal: we want to find the hyperplane (i.e. decision boundary) linearly separating our classes. Our boundary will have the equation:  $\mathbf{w}^T \mathbf{x} + \mathbf{b} = 0$

Anything above the decision boundary should have label 1 i.e.,  $\mathbf{w}^T \mathbf{x}_i + b > 0$  will have corresponding  $y_i = 1$

Similarly, anything below the decision boundary should have label -1 i.e.  $\mathbf{w}^T \mathbf{x}_i + b < 0$  will have corresponding  $y_i = -1$

The reason for this labeling scheme is that it lets us condense for the decision function to

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

since  $f(x) = +1$  for all  $x$  above the boundary, and  $f(x) = -1$  for all  $x$  below the boundary.

Thus, we can figure out if an instance has been classified properly by checking that  $y(\mathbf{w}^T \mathbf{x} + b) \geq 1$  (which will be the case as long as either both  $y, \mathbf{w}^T \mathbf{x} + b > 0$  or else  $y, \mathbf{w}^T \mathbf{x} + b < 0$ )

You'll notice that we will now have some space between our decision boundary and the nearest data points of either class. Thus, let's rescale the data such that anything on or above the boundary  $\mathbf{w}^T \mathbf{x} + b = 1$  is of one class (with label 1), and anything on or below the boundary  $\mathbf{w}^T \mathbf{x} + b = -1$  is of another class (with label -1)

What is the distance between these newly added boundaries?

First note that the two lines are parallel, and thus share their parameters  $w, b$ . Pick an arbitrary point  $x_1$  to lie on line  $\mathbf{w}^T \mathbf{x} + b = -1$ . Then the closest point on line  $\mathbf{w}^T \mathbf{x} + b = 1$  is the point  $\mathbf{x}_2 = \mathbf{x}_1 + \lambda \mathbf{w}$  (since the closest point will always lie on the perpendicular; recall that the vector  $\mathbf{w}$  is perpendicular to both lines). Using this formulation,  $\lambda \mathbf{w}$  will be the line segment connecting  $\mathbf{x}_1$  and  $\mathbf{x}_2$ , and thus,  $\lambda \|\mathbf{w}\|$ , the distance between  $\mathbf{x}_1$  and  $\mathbf{x}_2$  is the shortest distance between the two lines/boundaries.

## References

- [., a] . (.a). Bagging.
- [., b] . (.b). Data mining algorithms in r/clustering/k-means.
- [., c] . (.c). Gaussian process.
- [., d] . (.d). Logistic regression.
- [., e] . (.e). Naive bayes.
- [., f] . (.f). Page rank.
- [., 2014] . (2014). Principal component analysis.
- [., 2015a] . (2015a). Decision trees.
- [., 2015b] . (2015b). Gradient descent.
- [., 2015c] . (2015c). Hierarchical.
- [Bernstein, 2016] Bernstein, M. (2016). Random forests.
- [Brownlee, 2015a] Brownlee, J. (2015a). Back-propagation.
- [Brownlee, 2015b] Brownlee, J. (2015b). Clever algorithms: Nature-inspired programming recipes.
- [Brownlee, 2015c] Brownlee, J. (2015c). Learning vector quantization.
- [Brownlee, 2015d] Brownlee, J. (2015d). Perceptron.
- [Dai and Ji, 2014] Dai, W. and Ji, W. (2014). A mapreduce implementation of c4.5 decision tree algorithm.
- [Gavrilov, ] Gavrilov, Z. (.). Support vector machines.
- [Hertzmamm et al., 2015] Hertzmamm, A., Fleet, D., and Brubaker, M. (2015). Adaboost.
- [Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.
- [Ng, ] Ng, A. Coursera machine learning.
- [Ram et al., 2010] Ram, A., Jalal, S., Jalal, A. S., and Kumar, M. (2010). A density based algorithm for discovering density varied clusters in large spatial databases. *International Journal of Computer Applications*, 3(6):1–4.
- [Schapire, 2014] Schapire, R. (2014). Machine learning algorithms for classification.
- [Stein, 2016a] Stein, B. (2016a). Unit hierarchial cluster analysis.
- [Stein, 2016b] Stein, B. (2016b). Unit hierarchial cluster analysis.
- [Tay et al., 2014] Tay, B., Hyun, J., and Sejong, O. (2014). A machine learning approach for specification of spinal cord injuries using fractional anisotropy values obtained from diffusion tensor images.
- [Wang et al., ] Wang, T., Ren, Z., Ding, Y., Fang, Z., Sun, Z., MacDonald, M. L., Sweet, R. A., Wang, J., and Chen, W. Fastggm: An efficient algorithm for the inference of gaussian graphical model in biological networks.

---

**Algorithm 28** Support Vector Machines (continued)

---

Solving for  $\lambda : \mathbf{w}^T \mathbf{x}_2 + b = 1$  where  $\mathbf{x}_2 = \mathbf{x}_1 + \lambda \mathbf{w}$   
 $\mathbf{w}^T (\mathbf{x}_1 + \lambda \mathbf{w}) + b = 1 \Rightarrow \mathbf{w}^T \mathbf{x}_1 + \lambda \mathbf{w}^T \mathbf{w} + b = 1$   
 $\mathbf{w}^T \mathbf{x}_1 + b + \lambda \mathbf{w}^T \mathbf{w} = 1$  where  $\mathbf{w}^T \mathbf{x}_1 + b = -1$   
 $-1 + \lambda \mathbf{w}^T \mathbf{w} = 1$   
 $\lambda \mathbf{w}^T \mathbf{w} = 2$   
 $\lambda = \frac{2}{\mathbf{w}^T \mathbf{w}} = \frac{2}{\|\mathbf{w}\|^2}$   
 And, so the distance  $\lambda \|w\|$  is  $\frac{2}{\|\mathbf{w}\|^2} \|w\| = \frac{2}{\|\mathbf{w}\|} = \frac{2}{\sqrt{\mathbf{w}^T \mathbf{w}}}$

It's intuitive that we would want to maximize the distance between the two boundaries demarcating the classes (Why? We want to be as sure that we are not making classification mistakes and thus we want our data points from the two classes to lie as far away from each other as possible). This distance is called the margin, so we want to obtain the maximal margin.

Thus, we want to maximize  $\frac{2}{\sqrt{\mathbf{w}^T \mathbf{w}}}$ , which is equivalent to minimizing  $\frac{\sqrt{\mathbf{w}^T \mathbf{w}}}{2}$  which is in turn equivalent to minimizing  $\frac{\mathbf{w}^T \mathbf{w}}{2}$  (since square root is a monotonic function)

This quadratic programming problem is expressed as :

$\min_{w,b} \frac{\mathbf{w}^T \mathbf{w}}{2}$   
 subject to :  $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 (\forall \text{ data points } \mathbf{x}_i)$

## 2 Soft-margin extension

Consider the case that your data isn't linearly separable. For instance, maybe you aren't guaranteed that all your data points are correctly labelled, so you want to allow some data points of one class to appear on the other side of the boundary.

We can introduce *slack variables*  $\epsilon_i \geq 0$ . Our quadratic programming problem becomes:

$\min_{\mathbf{w}, b, \epsilon} \frac{\mathbf{w}^T \mathbf{w}}{2} + C \sum_i \epsilon_i$   
 subject to :  $y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \epsilon_i$

## 3 Nonlinear decision boundary

Mapping your data vectors,  $x_i$ , into a higher-dimension (even infinite) feature space may make them linearly separable in that space (whereas they may not be linearly separable in the original space). The formation of the quadratic programming problem is as above, but with all  $\mathbf{x}_i$  replaced with  $\phi(\mathbf{x}_i)$ , where  $\phi$  provides the higher-dimensional mapping. So we have the standard SVM formulation:

$\min_{\mathbf{w}, b, \epsilon} \frac{\mathbf{w}^T \mathbf{w}}{2}$   
 subject to :  $y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1 - \epsilon_i$  and  $\epsilon_i \geq 0 (\forall \text{ data points } \mathbf{x}_i)$

## 4 Reformulating as a Lagrangian

We can introduce Lagrange multipliers to represent the condition:

$y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b)$  must be as close to 1 as possible. This condition is captured by:  $\max_{\alpha_i \geq 0} \alpha_i [1 - y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b)]$ . This ensures that when  $y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1$ , the expression above is maximal when  $\alpha_i = 0$  (since  $[1 - y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b)]$  ends up being negative). Otherwise,  $y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) < 1$ , so  $[1 - y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b)]$  is a positive value, and the expression is maximal when  $\alpha_i \rightarrow \infty$ . This has the effect of penalizing any misclassified data points, while assigning 0 penalty to properly classified instances.

We thus have the following formulation:

$\min_{w,b} [\frac{\mathbf{w}^T \mathbf{w}}{2} + \sum_i \max_{\alpha_i \geq 0} \alpha_i [1 - \mathbf{w}^T \phi(\mathbf{x}_i) + b]]$

To allow for slack (soft-margin), preventing the  $\alpha$  variables from going to  $\infty$ , we can impose constraints on the Lagrange multipliers to lie within:  $0 \leq \alpha_i \leq C$ . We can define the dual problem by interchanging the max and min as follows (i.e minimize after fixing alpha):

$\max_{\alpha} [\min_{w,b} J(\mathbf{w}, b; \alpha)]$  where  $J(\mathbf{w}, b; \alpha) = \frac{\mathbf{w}^T \mathbf{w}}{2} + \sum_i \alpha_i [1 - y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b)]$

---

---

**Algorithm 28** Support Vector Machines (continued)

---

Since, we're solving an optimization problem, we set  $\frac{\partial J}{\partial \mathbf{w}} = 0$  and discover that the optimal setting of  $\mathbf{w}$  is  $\sum_i \alpha_i y_i \phi(\mathbf{x}_i)$ , while setting  $\frac{\partial J}{\partial b} = 0$  yields the constraint  $\sum_i \alpha_i y_i = 0$

Thus, after substituting and simplifying, we get:

$$\begin{aligned} \min_{w,b} J(\mathbf{w}, b, \alpha) &= \sum_i \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \text{ And thus our dual is:} \\ \max_{\alpha \geq 0} & [\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)] \\ \text{Subject to: } & \sum_i \alpha_i y_i = 0 \text{ and } 0 \leq \alpha_i \leq C \end{aligned}$$

## 5 Kernel trick

Because we're working in a higher-dimension space (and potentially even an infinite-dimensional space), calculating  $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$  may be intractable. However, it turns out there are special *kernel* functions that operate on the lower dimension vectors  $\mathbf{x}_i$  and  $\mathbf{x}_j$  to produce a value equivalent to the dot-product of the higher dimensional vectors. For instance, consider the function  $\phi: \mathbb{R}^3 \mapsto \mathbb{R}^{10}$ , where  $\phi(x) = (1, \sqrt{2}x^{(1)}, \sqrt{2}x^{(2)}, \sqrt{2}x^{(3)}, [\mathbf{x}^{(1)}]^2, [\mathbf{x}^{(2)}]^2, [\mathbf{x}^{(3)}]^2, \sqrt{2}x^{(1)(2)}, \sqrt{2}x^{(1)(3)}, \sqrt{2}x^{(2)(3)})$ . Instead, we have the kernel trick, which tells us that  $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^2 = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$  for the given  $\phi$ . Thus, we can simplify our calculations. Re-writing the dual in terms of the kernel yields:  $\max_{\alpha \geq 0} [\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)]$

## 6 Decision function

To classify a novel instance  $\mathbf{x}$  once you've learned the optimal  $\alpha_i$  parameters, all you have to do is calculate  $f(\mathbf{x}) = \text{sign}(\mathbf{w}^T \mathbf{x} + b) = \sum_i \alpha_i y_i \phi(\mathbf{x}_i)$  (and using the kernel trick). Note that  $\alpha_i$  is only non-zero for instances  $\phi(\mathbf{x}_i)$  on or near the boundary—those are called the *support vector* since they alone specify the decision boundary. We can toss out the other data points once training is complete. Thus, we only sum over the  $x_i$  which constitute the support vectors.

---