A. Agent Input-Ouput System (AIOS)

Contents

A.1 Machine Learning (ml)

All algorithms and models are accessed via a standard API.

The following API functions are available and supported by most models and algorithms:

- *ml.learn* - applies a learner to training data and returns learned model or creates a learner instance (all)
- *ml.classify* - applies data for prediction to an already learned model
- *ml.action* - returns next predicted action from reinforcement learner (RL)

- *ml.test* - evaluates test data
- *ml.print* - prints the model
- *ml.update* - updates an already learned model (only available with incremental and reinforcement learners)

All models returned by *ml.learn* are free of side effects and can be carried by mobile agents between platforms. The model/learner can be used on any other platform via the standard *ml* API.

Except for clusterer (unsupervised learning), a classifier learner expects input training data to learn a model that maps input feature variables on output target variables (labels or classes).

The following learners are available:

- Text similarity analysis `ml.ML.TXT`
- Decision Tree C45 for mixed numerical and categorical data variables `ml.ML.C45`
- Decision Tree ID3 for categorical data variables `ml.ML.ID3`
- Decision Tree ICE for numerical data variables (Interval arithmetic) `ml.ML.ICE`
- Decision Tree DTI for numerical data variables (Interval arithmetic and incremental) `ml.ML.DTI`
- k-nearest neighbour classifier `ml.ML.KNN`
- k-nearest neighbour classifier `ml.ML.KNN2`
- k-means clusterer `ml.ML.KMN`
- Single layer perception (ANN) `ml.ML.SLP`
- Multi layer perception (ANN) `ml.ML.MLP`
- Reinforcement Learning `ml.ML.RL`
- Support Vector Machine (unary, binary, n.ary) `ml.ML.SVM`

Most learners support different data formats (training, test, sample):

```
type data =
 'a [][] |
 'b [] |
 {$x:'a, $y:'b} [] |
 ('a|'b) [][]
```

Different data formats with 'a: Feature input variable type and 'b: Label output variable type.

A.1.1 Text

The text module supports currently only string similarity analysis.

ml.learn

```
function ({algorithm=ml.ML.TXT,
           data:string []})
  → model
```

The learner currently just stores the provided set of text strings (sentences) and returns a model object that can be used by other ML functions.

ml.classify

```
function (model,
          string|string[])
   → {similarity number,
      string:string} | {} []
```

The classification function computes the similarity of the given test sentence (or an array of sentences) with the sentences of the data base. It returns the best matching sentence of the data base.

ml.similarity

```
function (string, string) → number
```

Returns the similarity value [0..1] of two given strings.

Example

```
model = ml.learn({algorithm:ml.ML.TXT,
                 data:['Ein Satz','Noch ein Satz']})
```

```
result= ml.classify(model,["Mein Satz","Dein Satz"])
// [ { match: 0.88, string: 'Ein Satz' },
//   { match: 0.88, string: 'Ein Satz' } ]
test = ml.test({algorithm:ml.ML.TXT,
                string:"Dieser Satz"},"Mein Satz"))
// 0.54
```

ml.test

```
function ({algorithm:ml.ML.TXT,
           test1:string},
          test2:string) → number
```

Returns the similarity value [0..1] of two strings *test1* and *test2*.

A.1.2 Decision Tree (C4.5)

The classical C4.5 decision tree learner supporting mixed numerical and categorical data variables. Different training and test data formats are supported: Matrix (array of array) and array of records. Training data can be provided with data sets combining data and class labels in one row, or with separated *x* and *y* data (label) sets.

ml.learn

```
function ({algorithm=ML.C45,
           data:{}[]|[][],
           features?:string [],
           target?:string) → model
function ({algorithm=ML.C45,
           x:[][], y:[]) → model
```

Creates a decision tree from training data *data* containing the *features* and the *target* variable. The training data can either an array of records or an array of arrays of numerical data. If *features* and *target* is not specified, the last record attribute is assumed to be the target (label) and the first record attributes the feature variables. If the data set is a numerical array, the features variable names are index number, and the last array entry is assumed to be the target (label) variable.

ml.classify

```
function (model,
          data:{}|[]|{}[]|[][])
  → result:string|number|
            boolean|null|[]
```

Returns the classification (prediction) result for one or multiple data set(s).

A.1.3 Decision Tree (ID3)

This classical ID3 decision tree learner supporting categorical data variables based on entropy calculation and feature selection.

ml.learn

```
function ({algorithm=ML.ID3,
           data:{} [],
           target:string,
           features:string []})
   → model
```

Creates a decision tree from training data *data* containing the *features* and the *target* variable. The data consists of an array of records of the form `{x1:string|boolean|number, x2:*, .. , xn,y}`. The target variable *y* is the feature variable (classification result).

ml.classify

```
function (model,
          data:{}|{}[])
   → result:string|number|
            boolean|null|[]
```

Returns the classification (prediction) result for one or multiple data set(s).

A.1.4 Decision Tree with Interval Arithmetic (ICE)

This learner bases on the C45/ID3 algorithms supporting numerical data values only but uses 2-epsolon interval value instead of discrete value arithmetic for entropy calculation and feature selection. The classification bases on a nearest-neighbourhood look-up of best matching results. The prediction returns the best matching label and a matching probability (product of fit probabilities along a search path in the tree). The ε interval value can be specified for all data variables or individually for each data variable.

ml.learn

```
function ({algorithm=ML.ICE,
           data: {$x:number,$y:'a}[],
           target:string,
           features: string [],
           eps?:number | number[] |{$v:number} })
  → model
function ({algorithm=ML.ICE,
           x: number [][],
           y: 'a [],
           eps?:number|number[]|{$v:number})
  → model
```

ml.classify

```
function (model,
          data:{}|[]|{}[]|[][])
  → {val:'a, prob:number}| {}[]
```

A.1.5 Decision Tree with Interval Arithmetic and Incremental Learning (DTI)

This special learner bases on the ID3 algorithm supporting numerical values only but uses interval instead discrete value numeric for entropy calculation and feature selection. The classification bases on a nearest-neighbourhood look-up of best matching results. In contrast to the ICE algorithm, the DTI selects variables on a non-selective information entropy calculation of data columns.

Two different sub-algorithms are supported:

1. Static (using *learn*), the DTI learner using attribute selection based on entropy. The training data must be available in advance.
2. Dynamic (using *update*), the DTI learner using attribute selection based on significance. The training data is applied sequentially (stream learning) updating the model. Although in principle both algorithms can be mixed (first static learning, then dynamic updating), the resulting model will have poor classification quality. Either use static or only dynamic (stream) learning.

ml.learn

```
function ({algorithm=ML.DTI,
           data: {}[],
           target:string,
           features: string [],
           eps?:number,
           maxdepth?:number})
  → model
function ({algorithm=ML.DTI,
           x: number [][],
           y: 'a [],
           eps?:number,
           maxdepth?:number)
  → model
```

ml.classify

```
function (model,
          data:{}|[]|{}[]|[][])
  → 'a|'a []
```

ml.update

```
function (model,
          {data:{}[]})
  → model
function (undefined,
          {algorithm=ML.DTI,
           data: {}[],
           target:string,
           features: string [],
           eps?:number,
           maxdepth?:number})
  → model
```

Updates or creates a new decision tree model from (more or initial) input data.

A.1.6 k-Nearest Neighbour (KNN)

A k-nearest neighbour classifier with labelled nodes. The model is **portable**! The classifier returns discrete results (not interpolated).

ml.learn

```
function ({algorithm=ML.KNN,
           data:{}[],
           features:string [],
           target:string,
           k?:number,
           distance?:function})
  → model
function ({algorithm:ML.KNN,
           data:number [][],
           k?:number,
           distance?:function})
  → model
function ({algorithm:ML.KNN,
           x:number [][],
           y:number [],
           k?:number,
           distance?:function})
  → model
```

Creates a node graph model from input data (nodes with labels). There is no specific learning task here. The *k* and *distance* parameters are used by the classifier. Two data set formats are supported: Record arrays and matrix number arrays. In the latter case, either the last array element is the target (label) variable, or the feature variables (*x*) and target variable (*y*) are separated.

ml.classify

```
function (model,
          data:{}|number[]|{}[]|number[][])
  → *|*[]
```

Returns the best matching label or an array of best matching labels from an array of data nodes.

Example

```
var dataset = [
  {a: 1, b:2, y: 'yellow'},
  {a: 3, b:1, y: 'green'},
  {a: 5, b:2, y: 'yellow'},
  {a: 7, b:1, y: 'red'},
  {a: 9, b:2, y: 'blue'},
];
var model = ml.learn({
  algorithm:ml.ML.KNN,
  data:dataset,
  features : ["a","b"],
  target : 'y'
});
var nearest = ml.classify(model,[{ a: 7, b:1},
                                 { a:4, b:1}]);
log(nearest);
```

A.1.7 k-Nearest Neighbour 2 (KNN2)

A simplified k-nearest neighbour classifier with labelled nodes. Only numerical feature and target variables are supported. The model is **portable**! The classifier returns **interpolated results**.

ml.learn

```
function ({algorithm=ML.KNN2,
           data: {$:number}[],
           features:string [],
           target:string,
           k?:number,
           distance?:function})
  → model
function ({algorithm=ML.KNN2,
           data:number [][],
           k?:number,
           distance?:function})
  → model
function ({algorithm=ML.KNN2,
           x:number [][],
           y:number [],
           k?:number,
           distance?:function})
  → model
```

Creates a data model. There is no specific learning task here. The *k* and *distance* parameters are used by the classifier. Two data set formats are supported: Record arrays and matrix number arrays. In the latter case, either the last array element is the target (label) variable, or the feature variables (*x*) and target variable (*y*) are separated. Only numerical data is supported

ml.predict

```
function (model,
          data:{$:number} |
               number[] |
               {$:number}[] |
               number[][])
  → number|number[]
```

Returns interpolation of best matching numeric labels or an array of interpolated best matching labels from an array of data nodes.

Examples

```
var x = [[0, 0, 0], [0, 1, 1], [1, 1, 0],
         [2, 2, 2], [1, 2, 2], [2, 1, 2]];
var y = [0, 0.2, 0.4, 1, 1.2, 1.4];
model = ml.learn({
  algorithm:ml.ML.KNN2,
  x:x,
  y:y,
});
var test_data =[[0, 1, 0],
                [2, 2, 3]];
log(ml.classify(model,test_data))
```

A.1.8 K-means Clustering

This is an unsupervised learner returning group clustering of input data. The model (containing the clusters and data) is **portable**.

ml.learn

```
function ({algorithm:ML.KMN,
           data: number [][],
           k?:number,
           epochs?:numebr,
           distance?:function})
  → model
```

The model contains the clusters and means in the *clusters* and *means* attributes. The clusters array consists of data row index arrays.

ml.classify

```
function (model) → clusters: number [][]
```

The clusters array consists of data row index arrays.

Example

```
var data = [[1,0,1,0,1,1,1,0,0,0,0,0,1,0],
             [1,1,1,1,1,1,1,0,0,0,0,0,1,0],
             [1,1,1,0,1,1,1,0,1,0,0,0,1,0],
             [1,0,1,1,1,1,1,1,0,0,0,0,1,0],
             [1,1,1,1,1,1,1,0,0,0,0,0,1,1],
             [0,0,1,0,0,1,0,0,1,0,1,1,1,0],
             [0,0,0,0,0,0,1,1,1,0,1,1,1,0],
             [0,0,0,0,0,1,1,1,0,1,0,1,1,0],
             [0,0,1,0,1,0,1,1,1,1,0,1,1,1],
             [0,0,0,0,0,0,1,1,1,1,1,1,1,1],
             [1,0,1,0,0,1,1,1,1,1,0,0,1,0]
            ];
var model = ml.learn({
    algorithm: ml.ML.KMN,
    data : data,
    k : 4,
    epochs: 100,
    distance : {type : "pearson"}
});
log(ml.classify(model))
```

A.1.9 Multi-layer Perceptron (MLP)

An artificial neural network (ANN) model consists of an input layer, optional hidden layers, and an neuron output layer. The model is **portable**. The learner expects input data in the range [0,1]. If the data is not normalised, the option *normalize* has to be set, which transforms all data to the range [0,1]. The *bipolar* options transforms all data to the range [-1,1]. The *x* data set is input, the *y* set is the output data.

ml.learn

```
function ({algorithm=ML.MLP,
           x: number [][],
           y:number [],
           hidden_layers?:number [],
           normalize?,
           bipolar?:boolean})
  → model
function ({algorithm=ML.SLP,
           x: number [][],
           y:number [],
           normalize?,
           bipolar?:boolean})
  → model
```

A single layer ANN without hidden layers can be created by setting `algorithm=ML.SLP`.

ml.classify

```
function (model,
          data:number []|number [][])
  → number []|number [][]
```

Example

```
var x = [[1,1,1,0,0,0],
         [1,0,1,0,0,0],
         [1,1,1,0,0,0],
         [0,0,1,1,1,0],
         [0,0,1,1,0,0],
         [0,0,1,1,1,0]];
var y = [[1, 0],
         [1, 0],
         [1, 0],
         [0, 1],
         [0, 1],
         [0, 1]];
var model = ml.learn({
    algorithm : ml.ML.MLP,
    x : x,
    y : y,
    epochs : 20000,
    hidden_layers : [4,4,5]
});
a = [[1, 1, 0, 0, 0, 0],
     [0, 0, 0, 1, 1, 0],
     [1, 1, 1, 1, 1, 0]];
log(ml.classify(model,a));
```

A.1.10 Reinforcement Learning (RL)

The RL learner implements implements several common RL algorithms:

- Dynamic Programming methods (DPAgent)
- (Tabular) Temporal Difference (TDAgent) Learning (SARSA/Q-Learning)
- Deep Q-Learning (DQNAgent) for Q-Learning with function approximation with Neural Networks

ml.learn

```
function ({algorithm:ML.RL,
           kind:ML.DQNAgent|
                ML.TDAgent|
                ML.DPAgent,
           environment,
           specs})
  → model
```

This operation only creates an RL learner instance that can be used with the *ml.update* and *ml.action* operations.

ml.update

```
function (model,
          reward?:number)
```

Performs one training.

ml.action

```
function (model,
          state:number) → action
```

Returns the next action for the current state.

TDAgent

Temporal Difference learning algorithm. The learner is trained at run-time and requires repetitions of runs using *ml.action* and *ml.update* alternately The model is **portable**.

The environment consists of:

```
type environment = {
  getNumStates: function () → number,
  getMaxNumActions: function () → number,
  nextState: function (state:number,action)
             → state:number,
  allowedActions: function (state) → action []
}
```

The specification parameters are:

```
  type specs = {
   alpha   : number,
   beta    : number,
   epsilon : number,
   gamma : number,
   lambda : number,
   planN : number,
   replacing_traces : boolean,
   smooth_policy_update : boolean,
   update : 'qlearn' | 'sarsa'
  }
```

The *alpha* parameter is the value function learning rate, *beta* is the learning rate for smooth policy updates, *epsilon* is an initial epsilon for epsilon-greedy policy [0-1), the *gamma* number is the discount factor [0-1), *lambda* is the eligibility trace decay [0..1) and 0 means no eligibility traces, *planN* is the number of planning steps per iteration and 0 means no planning.

DPAgent

Dynamic Programming learning algorithm. The learner is trained in advance applying *ml.update* repeatedly. The internal policy evaluation and update of the learner will call the *reward* and *nextState* functions. After the learner is trained, a run sequence can be retrieved by calling *ml.action* repeatedly until a final state is reached. The model is **portable**.

The environment consists of:

```
type environment = {
  getNumStates: function () → number,
  getMaxNumActions: function () → number,
  nextState: function (state:number,action)
             → state:number,
  allowedActions: function (state) → action [],
  reward: function (state,action,nextstate) → number
}
```

The specification parameters are:

```
type $specs = {
 gamma : number is discount factor [0, 1)
}
```

DQNAgent

Deep Q-Learning learning algorithm using an artificial multi-layer neural network. The model is **portable**, but requires a large amount of data (> 10kB). The state vector is used to select an approproiate action from a given set of possible actions. The state vector is basically a sensor providing perception of the world state. After a suggested action was executed, the effect on the environment is back propagated by a reward.

The environment consists of:

```
type environment = {
  getNumStates: function () → number,
  getMaxNumActions: function () → number,
}
```

The specification parameters are:

```
  type $specs = {
   alpha  : number,
   epsilon : number,
   gamma : number is discount factor [0..1),
   experience_add_every : number,
   experience_size : number,
   learning_steps_per_iteration : number,
   tderror_clamp : number,
   num_hidden_units  : number,
   update : 'qlearn' | 'sarsa'
  }
```

The *alpha* parameter is the value for the function learning rate, *epsilon* is initial epsilon for epsilon-greedy policy [0-1), *experience_add_every* is number of time steps before another experience is added to replay, *experience_size* is size of experience, *tderror_clamp* is a robustness parameter, and *num_hidden_units* is number of neurons in hidden layer.

Example

```
var height=7,width=7,start=0;
// 0: free place, 1: start, 2: destination, -1: wall
var f=0,s=1,d=2,w=-1
var maze = [
[s,f,w,d,w,f,f],
[f,f,w,f,w,f,f],
[f,f,w,f,f,f,f],
[f,f,w,w,w,f,f],
[f,f,f,f,f,f,f],
[f,f,f,f,w,w,w],
[f,w,f,f,f,f,f],
]
var states = []
maze.forEach(function (row) {
  states=states.concat(row) })
var actions = ['left','right','up','down']
var env = {};
env.steps = 0;
env.iteration = 0;
// required by learner
env.getNumStates      = function() {
   return height*width; }
env.getMaxNumActions  = function() {
   return actions.length; }
env.nextState = function(state,action) {
  var nx, ny, nextstate;
  var x = env.stox(state);
  var y = env.stoy(state);
  switch (states[state]) {
    case f:
    case s:
      // free place to move around
      switch (action) {
        case 'left'  : nx=x-1; ny=y; break;
        case 'right' : nx=x+1; ny=y; break;
        case 'up'    : ny=y-1; nx=x; break;
        case 'down'  : ny=y+1; nx=x; break;
      }
      nextstate = ny*width+nx;
      env.steps++;
      break;
    case w:
      // cliff! oh no! Should not happend - see below
      nextstate=start;
      env.steps=0;
      env.iteration++;
      break;
    case d:
      // agent wins! teleport to start
      log('['+env.iteration+'] Found destination! steps='+
           env.steps)
      env.steps=0;
      nextstate=start;
      env.iteration++;
      break;
  }
  return nextstate;
}
env.reward = function (state,action,nextstate) {
  // reward of being in s, taking action a, and
  // ending up in ns
  var reward;
  // If the destination was found, weight the
  // reward with the number of steps
  // return best reward for shortest path
  if (states[state]==d) reward = 1.0-(env.steps/100)
  else if (states[state]==w) reward = -1;
  else reward = 0;
  return reward;
}
env.allowedActions    = function(state) {
  var x = env.stox(state), y = env.stoy(state);
  var actions=[];
  if (x>0) actions.push('left');
  if (y>0) actions.push('up');
  if (x<width-1) actions.push('right');
  if (y<height-1) actions.push('down');
  return actions
}
// utils
env.stox = function (s) { return s % width }
env.stoy = function (s) { return Math.floor(s / width) }
// create the DQN agent
var model = ml.learn({
  algorithm   : ml.ML.RL,
  kind        : ml.ML.TDAgent,
  actions     : actions,
  // specs
  alpha       : 0.1,
  beta        : 0.2,
  epsilon     : 0.2,
  gamma       : 0.5,
  lambda      : 0,
  planN       : 5,
  replacing_traces : true,
  smooth_policy_update : false,
  update : 'qlearn',  // 'qlearn' or 'sarsa
  environment : env
});
print(model)
var state = start;  // upper left corner
var timer = setInterval(function(){
  // start the learning loop
  // state is an integer
  var action = ml.action(model,state);
  // .. execute action in environment
  // and get the reward
  var ns = env.nextState(state,action);
  var reward = env.reward(ns)-0.01
  ml.update(model,reward)
  state = ns
}, 1);
```

Example: Temporal Difference Learning (TD) for moving in a maze world

```
var height=7,width=7,start,dest;
// 0: free place, 1: start, 2: destination, -1: wall
var f=0,s=1,d=2,w=-1
var maze = [
[s,f,w,d,w,f,f],
[f,f,w,f,w,f,f],
[f,f,w,f,f,f,f],
[f,f,w,w,w,f,f],
[f,f,f,f,f,f,f],
[f,f,f,f,w,w,w],
[f,w,f,f,f,f,f],
]
// world states
var states = []
maze.forEach(function (row,j) {
  states=states.concat(row)
  row.forEach(function (cell,i) {
    if (cell==s) start=i+j*width;
    if (cell==d) dest={x:i,y:j}
  })
})
```

```
var way = []
function reset (pr) {
  if (pr) print(way.join('\n'))
  way = maze.map(function (row) {
    return row.map(function (col) {
       return col==s?1:(col==w?'w':0) })})
  env.steps=0;
  env.good=0;
  env.error=0;
  env.iteration++;
}
var actions = ['left','right','up','down']
// Agent sensor states (perception)
// Distances {N,S,W,E} to boundaries and walls, distance
var sensors = [0,0,0,0,0]
var env = {};
env.steps = 0;
env.iteration = 0;
env.error = 0;
env.good = 0;
env.last = 0;
// required by learner
env.getNumStates      = function() {
    return sensors.length /*!!*/ }
env.getMaxNumActions  = function() {
    return actions.length; }
// internals
env.nextState = function(state,action) {
  var nx, ny, nextstate;
  var x = env.stox(state);
  var y = env.stoy(state);
  // free place to move around
  switch (action) {
    case 'left'  : nx=x-1; ny=y; break;
    case 'right' : nx=x+1; ny=y; break;
    case 'up'    : ny=y-1; nx=x; break;
    case 'down'  : ny=y+1; nx=x; break;
  }
  nextstate = env.xytos(nx,ny);
  if (nx<0 || ny<0 || nx >= width || ny >= height ||
      states[nextstate]==w) {
    nextstate=-1;
    return nextstate;
  }
  way[ny][nx]=1;
  env.steps++;
  return nextstate;
}
env.reward = function (state,action,ns) {
  // reward of being in s, taking action a,
  // and ending up in nextstate ns
  var reward;
  var dist1=Math.sqrt(Math.pow(dest.x-env.stox(ns),2)+
                      Math.pow(dest.y-env.stoy(ns),2))
  var dist2=Math.sqrt(Math.pow(dest.x-env.stox(state),2)+
                      Math.pow(dest.y-env.stoy(state),2))
  if (ns==env.laststate)
      reward = -10; // avoid ping-pong
  else if (ns==-1)
      reward = -100; // wall hit or outside world
  else if (dist1 < 1)
      reward = 100-env.steps/10;
      // destination found
  else reward = (dist1-dist2)<0?dist1/10:-dist1/10;
      // on the way
  env.laststate=ns;
  return reward;
}
// Update sensors
env.perception = function (state) {
  var i,
      dist=Math.sqrt(Math.pow(dest.x-env.stox(state),2)+
                     Math.pow(dest.y-env.stoy(state),2)),
      x = env.stox(state),
      y = env.stoy(state),
      sensors = [0,0,0,0,dist]; // N S W E
  // Distances to obstacles
  for(i=y;i>0;i--)
    { if (states[env.xytos(x,i)]==w) break }
  sensors[0]=y-i-1;
  for(i=y;i<height;i++)
    { if (states[env.xytos(x,i)]==w) break }
  sensors[1]=i-y-1;
  for(i=x;i>0;i--)
    { if (states[env.xytos(i,y)]==w) break }
  sensors[2]=x-i-1;
  for(i=x;i<width;i++)
    { if (states[env.xytos(i,y)]==w) break }
  sensors[3]=i-x-1;
  return sensors
}
// utils
env.stox = function (s) { return s % width }
env.stoy = function (s) { return Math.floor(s / width) }
env.xytos = function (x,y) { return x+y*width }
reset()
// create the DQN agent
var model = ml.learn({
  algorithm   : ml.ML.RL,
  kind        : ml.ML.DQNAgent,
  actions     : actions,
  // specs
  update : 'qlearn', // qlearn | sarsa
  gamma : 0.9,
  epsilon : 0.2,
  alpha : 0.005,
  experience_add_every : 5,
  experience_size : 10000,
  learning_steps_per_iteration : 5,
  tderror_clamp : 1.0,
  num_hidden_units : 100,
  environment : env
});
// world state, upper left corner
var state = start;
// The agent searches the destination with random walk
// If the the destination was found,
// it jumps back to the start
var timer = setInterval(function(){
  // start the learning loop
  sensors = env.perception(state);
  // sensors is a vector
  var action = ml.action(model,sensors);
  //... execute action in environment
  // and get the reward
  var ns = env.nextState(state,action);
  var reward = env.reward(state,action,ns)
  if (states[ns]==d) {
    // destination found
    print('iteration='+env.iteration,
          ', reward='+reward,' action: steps='+
          env.good,'error='+env.error)
    ns=start;
    reset(true);
  }
  if (ns==-1) env.error++;
  else env.good++;
  ml.update(model,reward)
  state = ns==-1?state:ns
}, 1);
```

Example: Deep-Q Learning (DQN) for movnng in a maze world

A.1.11 Support Vector Machine (SVM)

A Support Vector Machine learner for numerical data implementing a simplified SMO algorithm for binary classification. The model is **portable**! The value set of target variable *y* contains only values {-1,1}!

ml.learn

```
function({algorithm=ML.SVM,
          x:number [][],
          y:{-1,1} [],
          classes?:number|string [],
          threshold?:number|boolean,
          C?:numer,
          tol?:number,
          max_passes?:number,
          alpha_tol?:number,
          kernel?:kernel})
  → model
```

Creates an SVM model from input data *x* and *y*. The target variable *y* consists only of values {-1,1} (binary classification)! The trainer uses Monte Carlo simulation of the selection of input data rows. For n-ary classification, one SVM is used for each label. A higher tolerance parameter *tol* means higher precision (default 10^{-4}), a higher *max_passes* value means higher precision (default 20), and a higher *alpha_tol* parameter means higher precision, too (default 10^{-5}).

A SVM is a binary classifier (either predicting one ore two classes). A n-ary classifier can be created by multiple binary SVMs. If the *classes* set attribute is provided (a list of class symbols), a multi-SVM is created. In this case, the *y* vector consists of values from the *classes* set.

The optional *kernel* specifies the SVM kernel. Choices are:

- `{ type: "polynomial", c: number, d: number}`
- `{ type: "linear" }`
- `{ type: "rbf", sigma: number}`

ml.classify

```
function(model,
         data:number []|number [][])
  → result
with type result =
  number |
  string |
  number [] |
  string [] |
  number [] [] |
  string [] [] |
  {value:number|string,prob:number} |
  {value:number|string,prob:number} []
  {value:number|string,prob:number} [] []
```

In case of a binary SVM classifier only two values -1 and +1 are returned. If the *classes* attribute was provided, the class symbol is returned. In case of multi-SVM, the prediction can be ambitious and a list of symbols is returned. The SVM itself returns a linear value output value in the interval [-1,1]. Values greater a threshold value will output a value 1, else -1. If the *threshold* parameter is set to `false`, the linear output value is returned and in case of a multi-SVM the symbol with the largest value is returned.

Example

```
var x = [[0.4, 0.5, 0.5, 0.,  0.,  0.],
         [0.5, 0.3,  0.5, 0.,  0.,  0.01],
         [0.4, 0.8, 0.5, 0.,  0.1,  0.2],
         [1.4, 0.5, 0.5, 0.,  0.,  0.],
         [1.5, 0.3,  0.5, 0.,  0.,  0.],
         [0., 0.9, 1.5, 0.,  0.,  0.],
         [0., 0.7, 1.5, 0.,  0.,  0.],
         [0.5, 0.1,  0.9, 0.,  -1.8,  0.],
         [0.8, 0.8, 0.5, 0.,  0.,  0.],
         [0.,  0.9,  0.5, 0.3, 0.5, 0.2],
         [0.,  0.,  0.5, 0.4, 0.5, 0.],
         [0.,  0.,  0.5, 0.5, 0.5, 0.],
         [0.3, 0.6, 0.7, 1.7,  1.3, -0.7],
         [0.,  0.,  0.5, 0.3, 0.5, 0.2],
         [0.,  0.,  0.5, 0.4, 0.5, 0.1],
         [0.,  0.,  0.5, 0.5, 0.5, 0.01],
         [0.2, 0.01, 0.5, 0.,  0.,  0.9],
         [0.,  0.,  0.5, 0.3, 0.5, -2.3],
         [0.,  0.,  0.5, 0.4, 0.5, 4],
         [0.,  0.,  0.5, 0.5, 0.5, -2]];
var y1 =  [-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
          1,1,1,1,1,1,1,1,1,1];
var model1 = ml.learn({
    algorithm : ml.ML.SVM,
    x : x,
    y : y,
    C : 1.0, // default : 1.0. C in SVM.
    tol : 1e-5,
    max_passes : 200,
    alpha_tol : 1e-5,
    kernel : { type: "polynomial", c: 1, d: 5}
});
a = [
  [1.3,  1.7,  0.5, 0.5, 1.5, 0.4],
  [0.05,  0.1,  0.5, 1.5, -0.5, -1.4]
]
log(ml.classify(model1,a));
```

Examples of a binary classifier

```
var x = [[0, 0, 0], [0, 1, 1], [1, 1, 0],
         [2, 2, 2], [1, 2, 2], [2, 1, 2]];
var y = ['A', 'A', 'B', 'B', 'C', 'C'];
var model = ml.learn({
    algorithm:ml.ML.SVM,
    x:x,
    y:y,
    threshold:false,
    classes:['A','B','C'],
    C : 15.0,
    tol : 1e-5,
    max_passes : 200,
    alpha_tol : 1e-5,
    kernel : { type: 'rbf', sigma: 0.5 }
});
var test_data =[[0, 1.2, 0],
                [2.1, 2, 3],
                [2.1,1.1,2.0]
];
var xnoisy =
  x.map(function (row) {
   return row.map(function (col) {
    return col+random(-0.3,0.3,0.001) })}))
print(ml.classify(model,x))
print(ml.classify(model,xnoisy)
print(ml.classify(model,test_data))
print(ml.classify(model,[1,2,3]))
```

Examples of n-ary SVM classifier

A.1.12 Meta Data

```
Author: Stefan Bosse
```

Revision: 07/10/2019