What's New in *mlf* 2.0

Summary

- First of all, mlf was adapted to be ***compatible with Mathematica* 7.0**.
- Mlf was enhanced by new functions which allow ***parallel data mining*** tasks: the cross-validation (see ParallelCrossValidation) and the run of the "*Model Explorer*" (see ParallelRunModellingTasks). The "*Model Explorer*" allows to automatically run a lot of time-consuming tasks "overnight" and afterwards look at the performance of the different models that were created and tested.
- The range of available *methods/algorithms* was enlarged by ***kernel methods for classification and regression***: support vector machines (see CreateSVM) and gaussian process regression (see CreateGPR).
- A feature that allows the ***integration of user defined algorithms*** and performance measures into *mlf*.
- New meta-Learning algorithms for ***feature selection and boosting*** algorithms are now available (see CreateFeatureSelectionModel and CreateAdaBoostM1).
- Furthermore, ***building models for time series data*** is more directly supported.

Parallel computations

The new built-in parallel computing feature in *Mathematica* 7.0 made it possible to process typically time-consuming machine learning tasks within *mlf* in parallel; without making the handling much more complicated:

ParallelCrossValidation

When learning and testing a new model, in general, the most time-consuming part is the performing of the cross-validation. Giving the ability of computing the folds in parallel is a great help to reduce the overall computation time.

Here is a simple example of performing the cross-validation in parallel:

```
LaunchKernels[2]
```

```
{KernelObject[1, local], KernelObject[2, local]}
```

```
ParallelNeeds["mlf`"]
```

```
id3models =
  ParallelCrossValidation[irisData, Algorithm → CreateID3, RandomSeed → 1];
```

```
PrintMLFEvaluations[id3models]
```

	Total	Mean	Median	StdDev
NumberOfInstances	150	15	15	0
FractionCorrect	0.9467	0.9467	0.9333	0.05259
NormalizedMutualInformation	0.8308	0.8786	0.836	0.114
ChiSquared	255.8	26.18	25	3.65
PLevel	3.592×10^{-54}	0.00008975	0.00005031	0.0001377
MutualInformation	1.317	1.393	1.325	0.1808
RatioOfNullPredictions	0	0	0	0
ClassCondAccuracy	0.92 1 0.92	0.92 1 0.92	1 1 1	0.1398 0 0.1033
ChanceLevel	0.3333	0.3333	0.3333	0
MeanClassCondAccuracy	0.9467	0.9467	0.9333	0.05259
ModelSize	-	4.3	4	1.418

- **ParallelRunModellingTasks**

The "Model Explorer" of *mlf* makes working with machine learning models much more powerful. Instead of trying out one model, one dataset, and one input feature set and one goal feature at a time, using the model explorer a whole set of different algorithms, datasets, and/or features can be compared. This can be useful for architecture selection, for feature selection, or to compare different data sets.
As all permutations of combinations can be tried out, the number of modelling tasks to be carried out often increases very quickly. Giving the ability of computing the tasks in parallel is a great help to reduce the overall computation time.

Here is a simple example of performing the model training within the "Model Explorer" in parallel:

```
LaunchKernels[2]
```

```
{KernelObject[1, local], KernelObject[2, local]}
```

```
ParallelNeeds["mlf`"]
```

```
ParallelEvaluate[ Off[CreateNeuralNetwork::catCols] ];
ParallelEvaluate[ SetOptions[ $Output, PageWidth → Infinity ] ];
```

When the tasks are run, progress is indicated.

```
resList = ParallelRunModellingTasks[mtList];
```

```
Working on MT 5: {{Include["*"],
  Exclude[Last]} -> Last, TrainData[Name`AutoPriceTrainData1],
  TestData[Name`AutoPriceTestData1], RowSelection[1], LIRT[MaxLevel -> 2]}

Working on MT 1: {{Include["*"],
  Exclude[Last]} -> Last, TrainData[Name`HousingTrainData1],
  TestData[Name`HousingTestData1], RowSelection[1], LIRT[MaxLevel -> 2]}

Working on MT 6: {{Include["*"],
  Exclude[Last]} -> Last, TrainData[Name`AutoPriceTrainData1],
  TestData[Name`AutoPriceTestData1], RowSelection[1], LIRT[MaxLevel -> 8]}

Working on MT 7: {{Include["*"], Exclude[Last]} -> Last,
  TrainData[Name`AutoPriceTrainData1], TestData[Name`AutoPriceTestData1],
  RowSelection[1], RidgeRegression[AbsoluteDOF -> False, DOF -> Automatic]}

Working on MT 2: {{Include["*"],
  Exclude[Last]} -> Last, TrainData[Name`HousingTrainData1],
  TestData[Name`HousingTestData1], RowSelection[1], LIRT[MaxLevel -> 8]}

Working on MT 8: {{Include["*"], Exclude[Last]} -> Last,
  TrainData[Name`AutoPriceTrainData1], TestData[Name`AutoPriceTestData1],
  RowSelection[1], RidgeRegression[AbsoluteDOF -> False, DOF -> 0.8]}

Working on MT 3: {{Include["*"], Exclude[Last]} -> Last,
  TrainData[Name`HousingTrainData1], TestData[Name`HousingTestData1],
  RowSelection[1], RidgeRegression[AbsoluteDOF -> False, DOF -> Automatic]}

Working on MT 4: {{Include["*"], Exclude[Last]} -> Last,
  TrainData[Name`HousingTrainData1], TestData[Name`HousingTestData1],
  RowSelection[1], RidgeRegression[AbsoluteDOF -> False, DOF -> 0.8]}
```

Kernel methods

Support vector machines (CreateSVM)

Support Vector Machines (SVMs) are one of *kernel-based* methods in machine learning and are appropriate for both one-class and multi-class classification as well as regression.

Here is a simple example for multi-class classification:

```
{train, test} = LoadData["iris.txt", TrainPart → 0.65, RandomSeed → 1234];
goal = 5;
GetParamDS[ds, "Labels"]
```

```
{sepal_length, sepal_width, petal_length, petal_width, class}
```

```
m1 = TrainModel[train, {1, 2, 3, 4}, 5,
   Algorithm -> CreateSVM, AlgorithmOpts -> {Kernel -> RBFKernel}];
svm1 = Model /. (List @@ m1)[[1]];
```

```
PrintSVM[svm1]
```

```
svm_type c_svc
kernel_type rbf
gamma 1
nr_class 3
total_sv 48
rho -0.296534 -0.1564 0.131122
label 3 2 1
nr_sv 19 11 18
```

```
PrintMLFEvaluations[TestModel[m1, test, ReturnPredictions → False]]
```

NumberOfInstances	52
FractionCorrect	0.980769
NormalizedMutualInformation	0.930339
ChiSquared	98.3125
PLevel	2.24925×10^{-20}
MutualInformation	1.4711
RatioOfNullPredictions	0.
ClassCondAccuracy	1. 0.9375 1.
ChanceLevel	0.365385
MeanClassCondAccuracy	0.979167
ModelSize	0

- **Gaussian Precoess Regression (CreateGPR)**

Gaussian Process Regressions (GPRs) are one of *kernel-based* methods in machine learning.

Here is a simple example for gaussian process regression:

```
housingData = LoadData["housing.csv", MinDiffValues → 0];
```

```
headers = GetParamDS[housingData, Labels]
```

```
{CRIM, ZN, INDUS, CHAS, NOX, RM, AGE, DIS, RAD, TAX, PTRATIO, B, LSTAT, class}
```

```
gprModel = TrainModel[housingData, Range[13], 14, Algorithm -> CreateGPR];
```

```
PlotModel[gprModel]
```

Kernel	RBFKernel	
OptimalHyperParameters	1.36378	4.34833
MarginalLogLikelihood	-207.121	
NoiseLevel	2.27035	

```
PrintMLFEvaluations[TestModel[gprModel, housingData]]
```

```
NumberOfInstances       506
CorrelationCoefficient  0.978202
MSE                     3.66904
RSE                     0.0434619
MeanError               0.00229484
NormalizedMSE           0.00181187
MAE                     1.32976
RAE                     0.200047
NormalizedMAE           0.0295501
StdDev                  1.91737
TargetStdDev            9.1971
TargetMAD               6.64721
RatioOfNullPredictions  0.
RMSE                    1.91547
RRMSE                   0.208475
ModelSize               467.625
```

Integreation of user algorithms and performance measures

Registering new algorithms and/or performance measures to the *mlf* allows the user to work with the respective user-defined algorithms and performance measures within *mlf* like with standard mlf algorithms and performance measures respectively; e.g. in the case of algorithms the use of TrainModel, TestModel, the "Model Explorer" framework is possible.

RegisterAlgorithm

Here is a simple example of how a new algorithm can be registered by the user:

A very simple model, learning a scaling and offset of the first provided input column to predict the goal column (by least mean squares cost function):

```
CreateMyModel[ds_, inputs_, goal_, opts : OptionsPattern[]] :=
  Block[{a, b}, Module[{res, invals = GetData[ds, All, First@inputs],
     goalvals = GetData[ds, All, goal]},
    res  = Minimize[Total[(a * invals + b - goalvals) ^2], {a, b}];
    MyModel[Input → First@inputs, Sequence @@ Last@res]
   ]];
RecallMyModel[ds_, model_, opts : OptionsPattern[]] :=
  Module[{rules = List @@ model},
   (a /. rules) * GetData[ds, All, (Input /. rules)] + (b /. rules)
  ];
PrintMyModel[model_, inputvars_, goalvars_, goalcol_, opts : OptionsPattern[]] :=
  TableForm[(List @@ #) & /@ List @@ model];
```

Options need to be provided to specify call-back functions to use for training or plotting the model, for recall (prediction) using the model, etc. Not all of these must be set, but the functionality provided by these can not be used then, of course.

```
RegisterAlgorithm[
  MyModel,
  CreateFunction → CreateMyModel,
  NumRecallFunction → RecallMyModel,
  PlotFunction → PrintMyModel,
  NeedsColumnInfo → {CreateFunction},
  NeedsPredicates → {},
  HandlesCatFeatures → False,
  HandlesNumFeatures → True
 ];
```

Test the algorithm on some data :

```
dataset =
  CreateDataSet[{{0.5, 1}, {0, 0}, {1, 2}}, {"Input", "Goal"}, MinDiffValues → 1];
model = TrainModel[dataset, {1}, 2, Algorithm → MyModel];
Print@InputForm@model;
Print@PlotModel[model];
Print@Predict[model, dataset];
```

```
MLFModel[{Model -> MyModel[Input -> 1, a -> 2., b -> 0.], FuzzySetsOfModel -> {},
  Algorithm -> MyModel, InputVars -> {"Input"}, GoalVars -> {"Goal"},
  InputCols -> {1}, GoalCol -> 2, GoalType -> "N", DataSetInfo ->
   {"Dim" -> 2, "Labels" -> {"Input", "Goal"}, "AttrLabels" -> {{}, {}}},
  ModelSize -> ModelSize[MyModel[Input -> 1, a -> 2., b -> 0.]]}]
```

```
Input  1
a      2.
b      0.
```

```
{1., 0., 2.}
```

- **RegisterPerformanceMeasure**

Here is a simple example of how to register a new, user-defined performance measure, that then is automatically calculated when a model is tested:

Define the measure evaluation function:

```
Options@WeightedClassError = {Weights → {}};
WeightedClassError[orig_, pred_, opts : OptionsPattern[]] :=
  Module[{weights = OptionValue[Weights]},
   Total@Map[
     If[orig[[#]] === pred[[#]], 0.0,
       If[orig[[#]] > Length@weights, 1.0,
        weights[[orig[[#]]]]
       ]
      ] &, Range@Length@orig]
  ];
```

Define the measure for *mlf* :

```
RegisterPerformanceMeasure[
 WeightedClassError,
 EvaluationFunction →
   (WeightedClassError[#1, #2, Weights → {0.5, 0.8, 2.5, 3.0}, ##3] &),
 DefaultOrdering → Less,
 HandlesCatFeatures → True,
 Requires → {OriginalData, Predictions},
 MLFEvaluations → True
]
```

```
True
```

Assuming some hypothetical goal values and predictions for them, the computed performance values, including the newly defined one, are:

```
ComputePerformanceMeasures[{1, 1, 2, 2, 2, 3, 3, 4},
  {1, 2, 2, 3, 2, 3, 1, 4}, GoalType → "C"]
```

```
{NumberOfInstances → 8, FractionCorrect → 0.625,
 NormalizedMutualInformation → 0.556915,
 ConfusionMatrix → {{1, 1, 0, 0}, {0, 2, 1, 0}, {1, 0, 1, 0}, {0, 0, 0, 1}},
 ChiSquared → 12.2222, PLevel → 0.201069, MutualInformation → 1.06128,
 WeightedClassError → 3.8, RatioOfNullPredictions → 0.,
 ClassCondAccuracy → {0.5, 0.666667, 0.5, 1.},
 ChanceLevel → 0.375, MeanClassCondAccuracy → 0.666667}
```

■ New Meta-Learning algorithms

■ Feature selection (CreateFeatureSelectionModel)

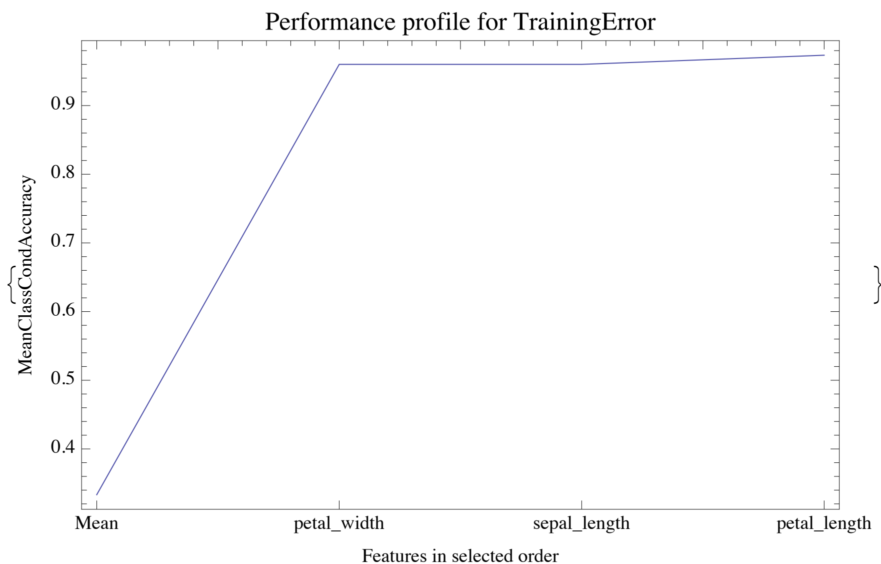
`CreateFeatureSelectionModel` creates a Meta-Model (ie. a model relying on another base model). It wraps a forward feature selection method around this base model, ie. it evaluates one feature in turn, selects the best of these, then evaluates each remaining feature in addition to the first selected feature, and so on.

Below, there is a simple example for building a feature selection model:

```
model = CreateFeatureSelectionModel[irisData,
   EvaluationCriterion → TrainingError[MeanClassCondAccuracy],
   BaseModel → {CreateNeuralNetwork,
     NetworkModel → MultiLayerPerceptron[HiddenUnits → {2}]},
   MinNumberFeatures → 3, MaxNumberFeatures → 4,
   Verbose → True];
```

```
Default performance: 0.333333

New feature: petal_width, new performance: 0.96`, improvement: 1.88`

New feature: sepal_length, new performance: 0.96`, improvement: 0.`

Not enough improvement. Best one was: 0.

New feature: petal_length, new performance:
  0.9733333333333333`, improvement: 0.01388888888888884`

Not enough improvement. Best one was: 0.0138889
```

```
PlotFeatureSelectionModel[model,
 PlotProfile → True, PlotModel → False, ImageSize → 400]
```



Above, we see which features are used by the final model, in which order they were chosen, and how much they did improve the prediction performance.

- **Boosting (CreateAdaBoostM1)**

CreateAdaBoostM1 creates a Meta-Model (ie. a model relying on nother base models). It wraps a boosting approach (in particular AdaBoostM1) which can be considered as an inkremental procedure for generating additive models for classification tasks. Boosting is considered as one of the most powerful out-of-the-box learning algorithms available.

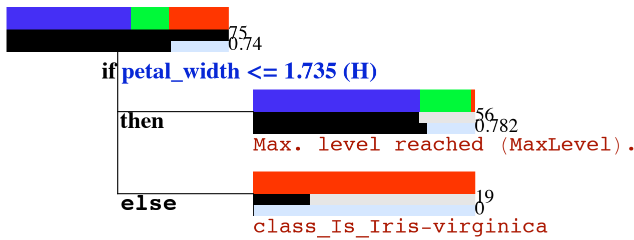
The following example shows how to train a boosting model with fuzzy decision trees (actually decision stumps):

```
m = TrainModel[ ds, Algorithm → CreateAdaBoostM1 , AlgorithmOpts →
    {MaximumNumberOfModels → 3, BasisFunctions → {{{CreateID3, MaxLevel → 1}}}}];
```

```
PlotModel[m, ImageSize → 300]
```

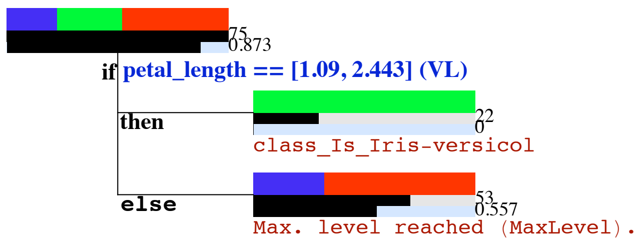
++++++++++++ Model 3 (weight=1.292,
importance=0.4167, FractionEqual={0.946667}) ++++++++++++

class_Is_Iris-setosa
class_Is_Iris-versicol
class_Is_Iris-virginica



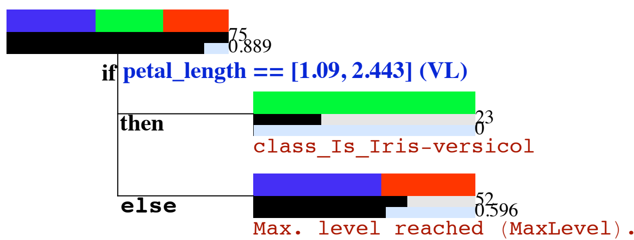
++++++++++++ Model 2 (weight=0.9295,
importance=0.2998, FractionEqual={0.6}) ++++++++++++

class_Is_Iris-setosa
class_Is_Iris-versicol
class_Is_Iris-virginica



++++++++++++ Model 1 (weight=0.8792,
importance=0.2836, FractionEqual={0.706667}) ++++++++++++

class_Is_Iris-setosa
class_Is_Iris-versicol
class_Is_Iris-virginica



■ Building models for time series

mlf provides a set of functions which allows the preprocessing of time series data in order to be processed by its learning algorithms.

Below we see the plot of a time series of Australian beer production:

In particular, the function **ApplyFilterBank** can be used to apply several filters to the time series and to add the respective columns to the data set. In this way a more informative training data set (e.g. with information about the past and/or the future) can be generated.

The data set originally contains the following colums:

```
H
```

```
{"Year", "Month", "Production"}
```

Now lets create a filterbank and apply it to the data:

```
FB = {
   Function[{X}, X],
   Function[{X}, Filter[ X[[All, 3]], FilterAverage, -12, 0]],
   Function[{X}, Filter[ X[[All, 3]], FilterPast,      -9   ]],
   Function[{X}, Filter[ X[[All, 3]], FilterFuture,     3   ]]
  };
```

```
{Hfilter, Xfilter} = ApplyFilterBank[H, X, FB, Verbose → True];
```

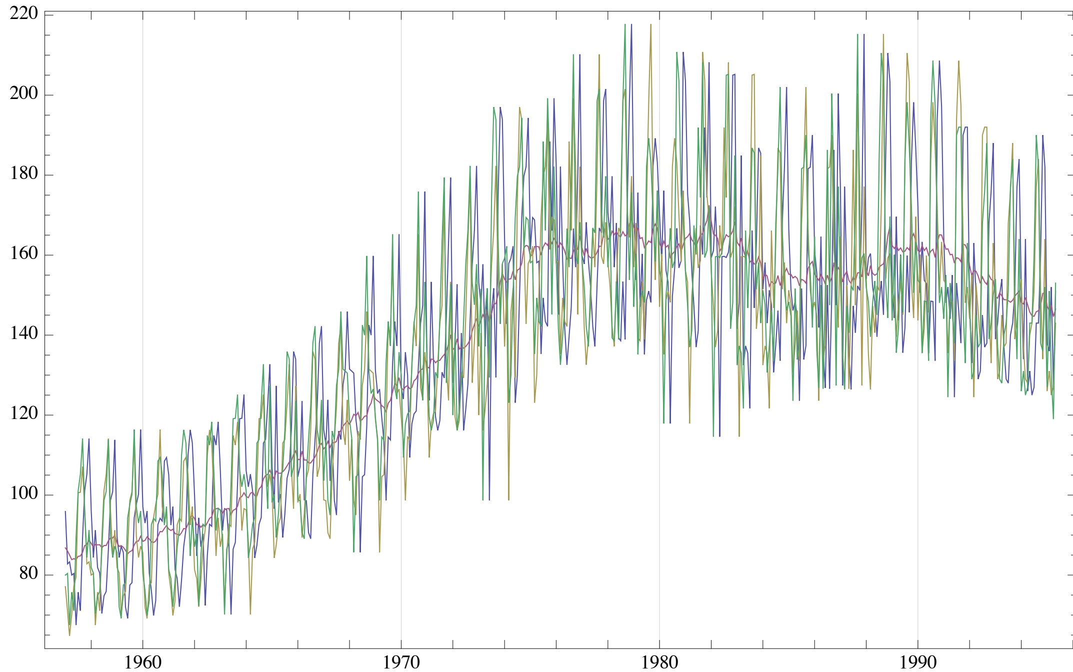
```
Filter 1 of 4 done.

Filter 2 of 4 done.

Filter 3 of 4 done.

Filter 4 of 4 done.
```

The result is a new data set with the following columns:

```
Hfilter
```

```
{Year, Month, Production,
 Mean[Production(t-12)...Production(t)], Production(t-9), Production(t+3)}
```

```
data = Map[Join[{{#[[1]], #[[2]]}}, #[[Range[3, Length@Hfilter]]]] &, Xfilter];
data = Table[data[[All, {1, j}]], {j, 2, Length@Hfilter - 1}];
DateListPlot[data, PlotLabel → Hfilter[[Range[3, Length@Hfilter]]],
 Joined -> True, ImageSize → 500]
```



Above, we see the time series plots of the original column "Production" as well as the three filtrations of it.

We created a new data set with additional information. Hence using the new data set for model training might result in better models by including the new features:

```
PlotModel[TrainModel[ds, Algorithm -> CreateRidgeRegression]]
```

```
DOF:      3.
Lambda:   0.
Noise:    11.6602
```

Weight	Attribute	Norm.Weight	Mean	Std.dev
4.371				
0.7469	Production(t-9)	0.7685	136.4	33.9
0.2014	Mean[Production(t-12)...Production(t)]	0.1712	136.9	28.02
0.656	Month	0.06887	6.462	3.459