A monad for classification workflows

Anton Antonov
MathematicaForPrediction at WordPress
MathematicaForPrediction at GitHub
MathematicaVsR at GitHub
March-May 2018

Version 0.6

Introduction

In this document I am going to describe the design and implementation of a (software programming) monad for classification workflows specification and execution. The design and implementation are done with Mathematica / Wolfram Language (WL).

The goal of the monad design is to make the specification of classification workflows (relatively) easy, straightforward, by following a certain main scenario and specifying variations over that scenario.

The monad is named `ClCon` and it is based on the State monad package "StateMonadCodeGenerator.m", [AAp1, AA1], the classifier ensembles package "ClassifierEnsembles.m", [AAp4, AA2], and the package for Receiver Operating Characteristic (ROC) functions calculation and plotting "ROCFunctions.m", [AAp5, AA2].

The data for this document is read from WL's repository using the package "GetMachineLearningDataset.m", [AAp10].

The monadic programming design is used as a Software Design Pattern. The `ClCon` monad can be also seen as a Domain Specific Language (DSL) for the specification and programming of machine learning classification workflows.

Here is an example of using the `ClCon` monad over the Titanic data:

`ClConUnit[dsTitanic] ⟹`	lift the data to the monad
`ClConSplitData[0.75`] ⟹`	split the data
`ClConMakeClassifier["LogisticRegression"] ⟹`	create a classifier
`ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall"}] ⟹`	compute classifier measurements
`ClConEchoValue`	display the current pipeline value

» value: <|Accuracy → 0.75841, Precision → <|died → 0.818653, survived → 0.671642|>, Recall → <|died → 0.782178, survived → 0.72|>|>

The table above is produced with the package "MonadicTracing.m", [AAp2], and some of the explanations below also utilize that package.

As it was mentioned above the monad `ClCon` can be seen as a DSL. Because of this the monadic pipelines made with `ClCon` are sometimes called "specifications".

Package load

The following commands load the packages [AAp1, AAp10]:

```
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/MonadicContextualClassification.m"]
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/MonadicTracing.m"]
Import["https://raw.githubusercontent.com/antononcube/MathematicaVsR/master/Projects/ProgressiveMachineLearning/Mathematica/GetMachineLearningDataset.m"]
```

» Importing from GitHub: `StateMonadCodeGenerator.m`

» Importing from GitHub: `ClassifierEnsembles.m`

» Importing from GitHub: `VariableImportanceByClassifiers.m`

» Importing from GitHub: `SSparseMatrix.m`

Data load

In this section we load data that is used in the rest of the document. The “quick” data is created in order to specify quick, illustrative computations. In all datasets the classification labels are in the last column.

The summarization of the data is done through `ClCon`, which in turn uses the function `RecordsSummary` from the package “MathematicaForPredictionUtilities.m”, [AAp7].

WL resources data

The following commands produce datasets using the package [AAp10] (that utilizes `ExampleData`):

In[5]:=
```
dsTitanic = GetMachineLearningDataset["Titanic"];
dsMushroom = GetMachineLearningDataset["Mushroom"];
dsWineQuality = GetMachineLearningDataset["WineQuality"];
```

Here is are the dimensions of the datasets:

In[8]:=
```
Dataset[Dataset[Map[Prepend[Dimensions[ToExpression[#]], #] &, {"dsTitanic", "dsMushroom", "dsWineQuality"}]][All, AssociationThread[{"name", "rows", "columns"}, #] &]]
```

Out[8]=

name	rows	columns
dsTitanic	1309	5
dsMushroom	8124	24
dsWineQuality	4898	13

Here is the summary of `dsTitanic`:

In[9]:=
```
ClConUnit[dsTitanic] ⟹ ClConSummarizeData["MaxTallies" → 12];
```

```
» summaries: {Anonymous → {
1 id
Min     1
1st Qu  327.75
Mean    655
Median  655
3rd Qu  982.25
Max     1309
,
2 passengerClass
3rd  709
1st  323
2nd  277
,
3 passengerAge
Min     -1
1st Qu  10
Median  20
Mean    23.55
3rd Qu  40
Max     80
,
4 passengerSex
male    843
female  466
,
5 passengerSurvival
died      809
survived  500
}}
```

Here is the summary of `dsMushroom` in long form:

In[10]:=
```
ClConUnit[dsMushroom] ⟹ ClConSummarizeDataLongForm["MaxTallies" → 12];
```

» summaries: {Anonymous → {

1 RowID	
1	24
10	24
100	24
1000	24
1001	24
1002	24
1003	24
1004	24
1005	24
1006	24
1007	24
(Other)	194 712

,

2 Variable	
bruises?	8124
cap-Color	8124
cap-Shape	8124
cap-Surface	8124
edibility	8124
gill-Attachment	8124
gill-Color	8124
gill-Size	8124
gill-Spacing	8124
habitat	8124
id	8124
(Other)	105 612

,

3 Value	
white	21 402
smooth	12 668
partial	8124
free	7914
one	7488
close	6812
brown	6356
broad	5612
pink	5380
False	4748
silky	4676
(Other)	103 796

}}

Here is the summary of dsWineQuality in long form:

In[11]:=
```
ClConUnit[dsWineQuality] ⟹ ClConSummarizeDataLongForm["MaxTallies" → 12];
```

» summaries: {Anonymous → {

1 RowID	
1	13
10	13
100	13
1000	13
1001	13
1002	13
1003	13
1004	13
1005	13
1006	13
1007	13
(Other)	63 512

,

2 Variable	
alcohol	4898
chlorides	4898
density	4898
fixedAcidity	4898
freeSulfurDioxide	4898
id	4898
pH	4898
residualSugar	4898
sulphates	4898
totalSulfurDioxide	4898
volatileAcidity	4898
(Other)	9777

,

3 Value	
Min	0.009
1st Qu	0.46
Median	4.6
3rd Qu	12.
Mean	204.533
Max	4898

}}

“Quick” data

In this subsection we make up some data that is to going to be used illustrative purposes.

In[12]:=
```
SeedRandom[212]
dsData = RandomInteger[{0, 1000}, {400}];
dsData = Dataset[Transpose[{dsData, Mod[dsData, 3], Last@*IntegerDigits /@ dsData, OddQ /@ dsData}]];
dsData = Dataset[dsData[All, AssociationThread[{"number", "feature1", "feature2", "label"}, #] &]];
Dimensions[dsData]
```

Out[16]=
```
{400, 4}
```

Here is a sample of the data:

In[17]:=
```
RandomSample[dsData, 6]
```

Out[17]=

number	feature1	feature2	label
358	1	8	False
57	0	7	True
49	1	9	True
833	2	3	True
267	0	7	True
306	0	6	False

Here is a summary of the data:

In[23]:=
```
ClConUnit[dsData] ⟹ ClConSummarizeData;
```

```
                                1 number          2 feature1       3 feature2
                                Min     9         1st Qu 0         Min     0
                                1st Qu  255       Min    0         1st Qu  2           4 label
» summaries: {Anonymous → {     Mean    491.663 , Median 1     ,   Mean    4.5375 ,    False 207 }}
                                Median  501       Mean   1.02      Median  5           True  193
                                3rd Qu  733       3rd Qu 2         3rd Qu  7
                                Max     998       Max    2         Max     9
```

Here we convert the data into a list of record-label rules (and show the summary):

In[26]:=
```
mlrData = ClConToNormalClassifierData[dsData];
ClConUnit[mlrData] ⟹ ClConSummarizeData;
```

```
                                1 column 1        2 column 2       3 column 3
                                Min     9         1st Qu 0         Min     0
                                1st Qu  255       Min    0         1st Qu  2             1 column 1
» summaries: {Anonymous → {     Mean    491.663 , Median 1     ,   Mean    4.5375 } → {  False 207 }}
                                Median  501       Mean   1.02      Median  5             True  193
                                3rd Qu  733       3rd Qu 2         3rd Qu  7
                                Max     998       Max    2         Max     9
```

Finally, we make an array version of the dataset:

In[21]:=
```
arrData = Normal[dsData[All, Values]];
```

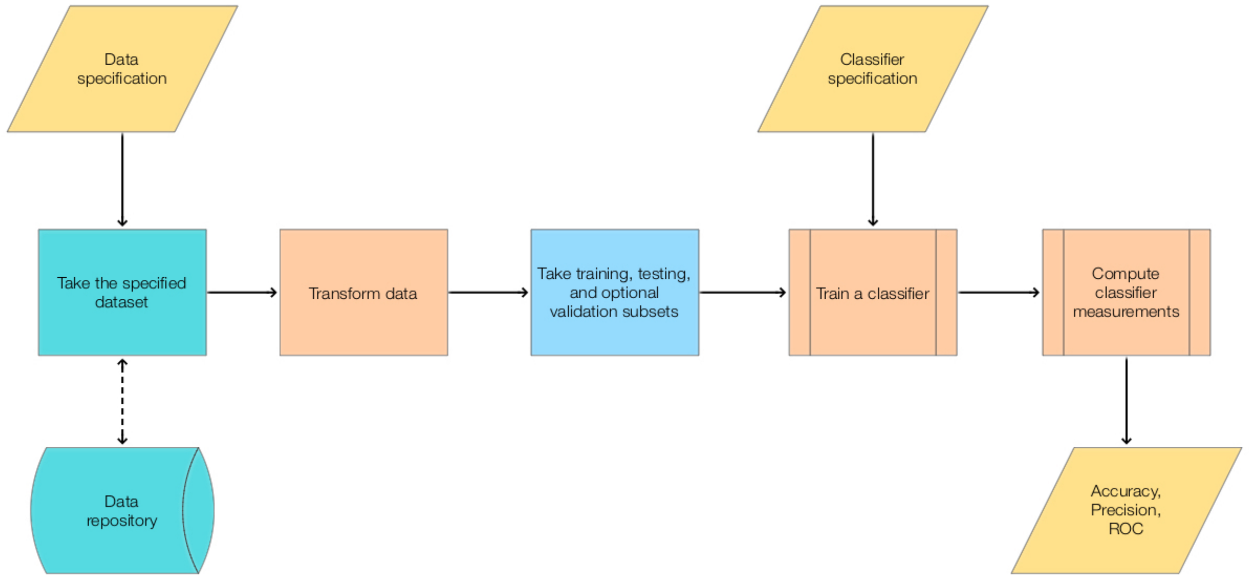
Design considerations

The steps of the main classification workflow addressed in this document follow.

1. Retrieving data from a data repository.
2. Optionally, transform the data.
3. Split data into training and test parts.
   - 3.1. Optionally, split training data into training and validation parts.
4. Make a classifier with the training data.
5. Test the classifier over the test data.
   - 5.1. Computation of different measures including ROC.

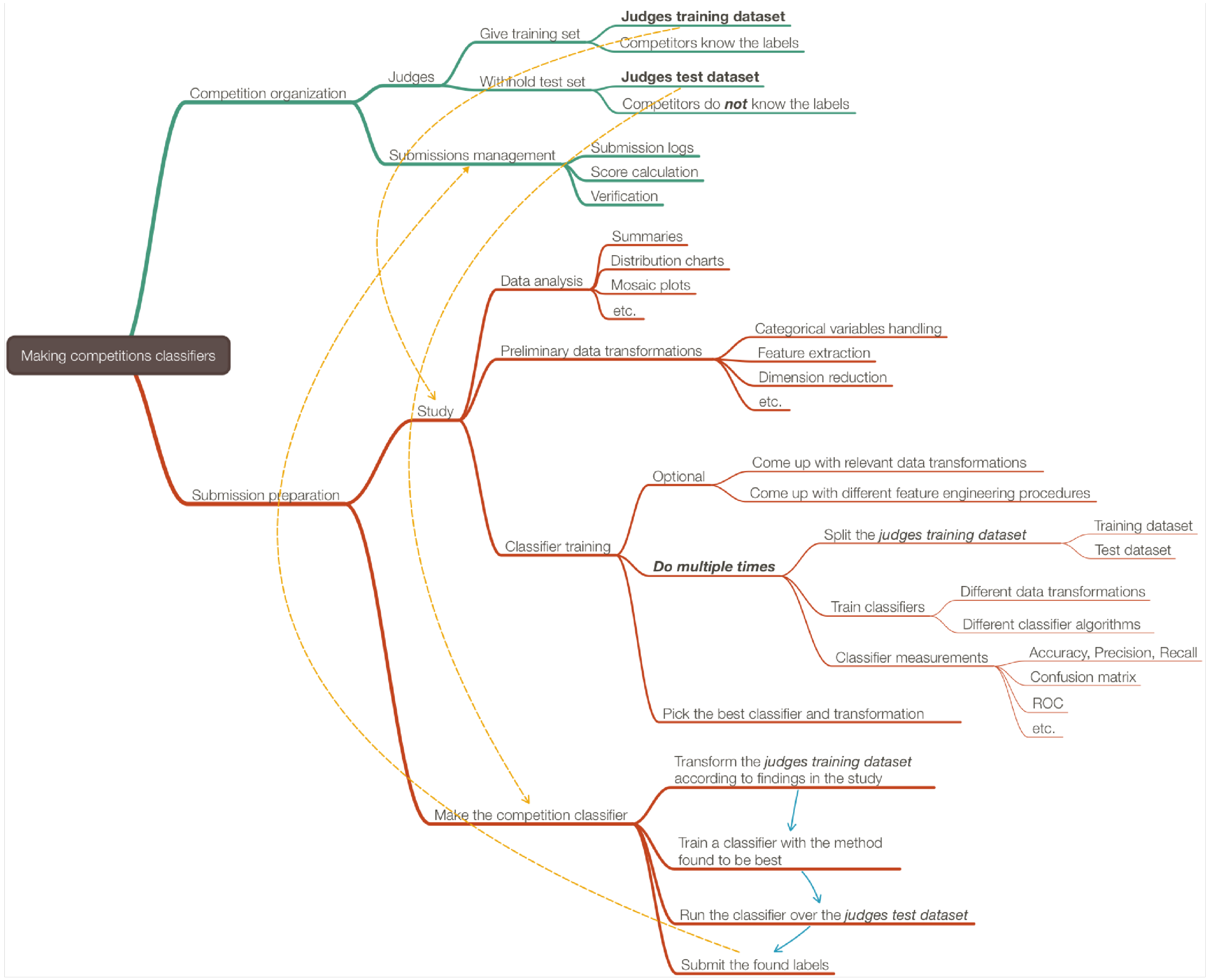
The following diagram shows the steps.

Out[ ]=



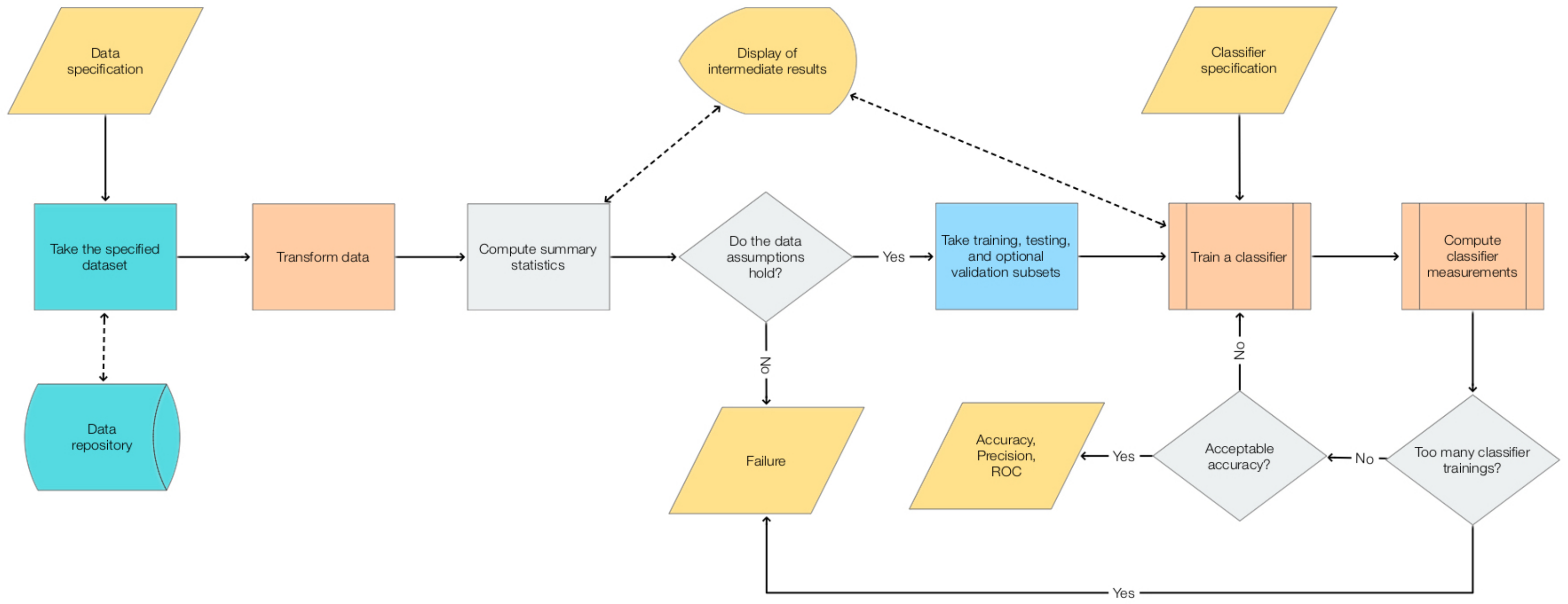
Very often the workflow above is too simple in real situations. Often when making “real world” classifiers we have to experiment with different transformations, different classifier algorithms, and parameters for both transformations and classifiers. Examine the following mind-map that outlines the activities in making competition classifiers.

Out[39]=



In view of the mind-map above we can come up with the following flow-chart that is an elaboration on the main, simple workflow flow-chart.

Out[•]=



In order to address:

- the introduction of new elements in classification workflows,
- workflows elements variability, and
- workflows iterative changes and refining,

it is beneficial to have a DSL for classification workflows. We choose to make such a DSL through a (functional programming) monad, [Wk1, AA1].

Here is a quote from [Wk1] that fairly well describes why choose to make a classification workflow monad and hints on the desired properties of such a monad.

> [...] The monad represents computations with a sequential structure: a monad defines what it means to chain operations together. This enables the programmer to build pipelines that process data in a series of steps (i.e. a series of actions applied to the data), in which each action is decorated with the additional processing rules provided by the monad. [...]
> Monads allow a programming style where programs are written by putting together highly composable parts, combining in flexible ways the possible actions that can work on a particular type of data. [...]

Remark: Note that quote from [Wk1] refers to chained monadic operations as “pipelines”. We use the terms “monad pipeline” and “pipeline” below.

Monad design

The monad we consider is designed to speed-up the programming of classification workflows outlined in the previous section. The monad is named **ClCon** for “**Cl**assification with **Con**text”.

We want to be able to construct monad pipelines of the general form:

$$\texttt{ClCon}[\_] \xrightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_1 \xrightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_2 \xrightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} \ldots \xrightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_k \qquad (1)$$

ClCon is based on the State monad, [Wk1, AA1], so the monad pipeline form (1) has the following more specific form:

$$\texttt{ClCon}[\texttt{pval\_},\ \texttt{context\_}] \xrightarrow[\texttt{ClConBind}[m\_,f\_]]{} \ldots \left(\begin{cases} f_i[\texttt{\$ClConFailure}] & m \equiv \texttt{\$ClConFailure} \\ f_i[x\_,\ c\_\texttt{Association}] & m \textit{ is } \texttt{ClCon}[x\_,\ c\_\texttt{Association}] \\ \texttt{\$ClConFailure} & \text{otherwise} \end{cases} \right) \xrightarrow[\texttt{ClConBind}[m\_,f\_]]{} \ldots \qquad (2)$$

In the monad pipelines of `ClCon` we are going to store different objects in the context for at least one of the following two reasons.

1. The object would be needed later on in the pipeline, or
2. The object is hard to compute. Such objects are training data and classifiers.

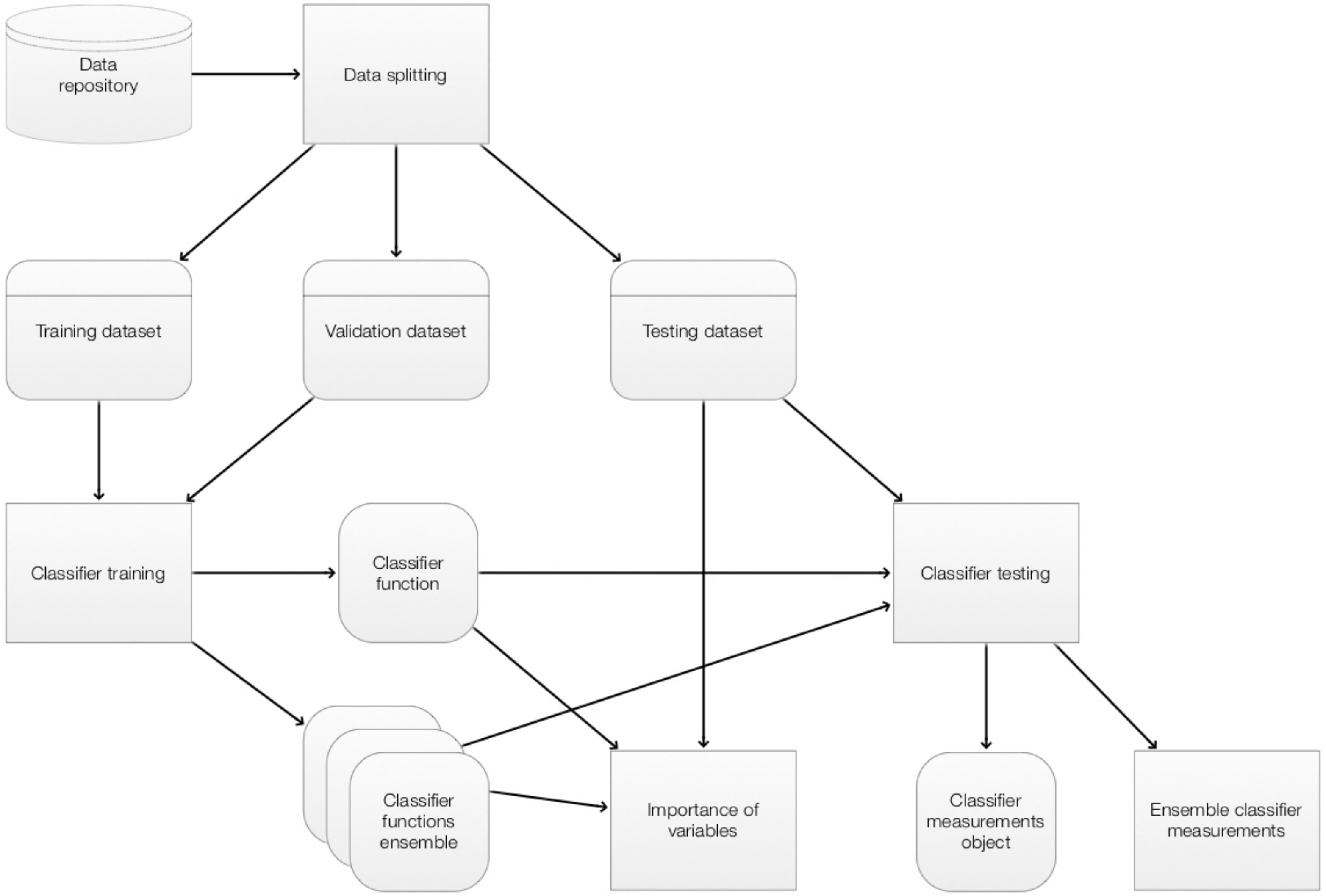
This means that some monad operations would not just change the pipeline value but they will also change the pipeline context.

Let us list the desired properties of the monad.

- Rapid specification of non-trivial classification workflows.
- The monad works with different data types: Dataset, lists of machine learning rules, full arrays.
- The pipeline values can be of different types. Generally, every monad function modifies the pipeline value; some modify the context.
- The monad works with single classifier objects and classifier ensembles.
  - This means support of different classifier measures and ROC plots for both single classifiers and classifier ensembles.
- The monad allows of cursory examination and summarization of the data.
  - For insight and in order to verify assumptions.
- The monad gives means to compute importance of variables.
- We can easily obtain the pipeline value, context, and different context objects for manipulation outside of the monad.
- We can calculate classification measures using a specified ROC parameter and class label.
- We can easily plot different combinations of ROC functions.

The `ClCon` components and their interaction are given in the following diagram. (The components correspond to the main workflow given in the previous section.)

Out[ ]=



In the diagram above the operations are given in rectangles. Data objects are given in round corner rectangles and classifier objects are given in round corner squares.

The main ClCon operations implicitly put in the context or utilize from the context the following objects:

- training data,
- test data,
- validation data,
- classifier (a classifier function or an association of classifier functions),
- ROC data,
- variable names list.

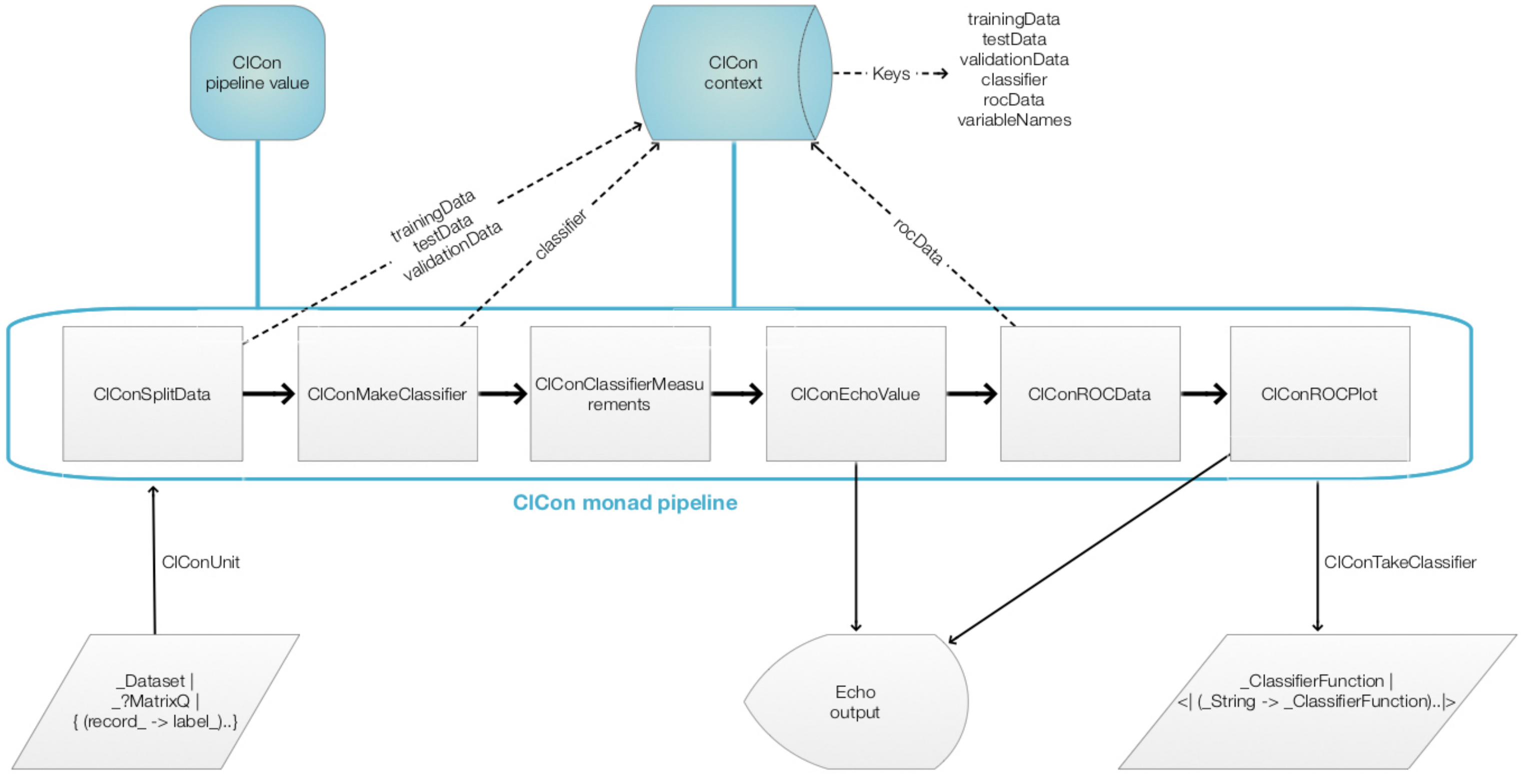
Note the that the monadic set of types of ClCon pipeline values is fairly heterogenous and certain awareness of “the current pipeline value” is assumed when writing ClCon pipelines.

Obviously, we can put in the context any object through the generic operations of the State monad of the package StateMonadGenerator.m, [AAp1].

ClCon structure by examples

The following diagram shows the structure of the ClCon monad.

Out[45]=



Let us examine a ClCon pipeline that corresponds to the diagram above.

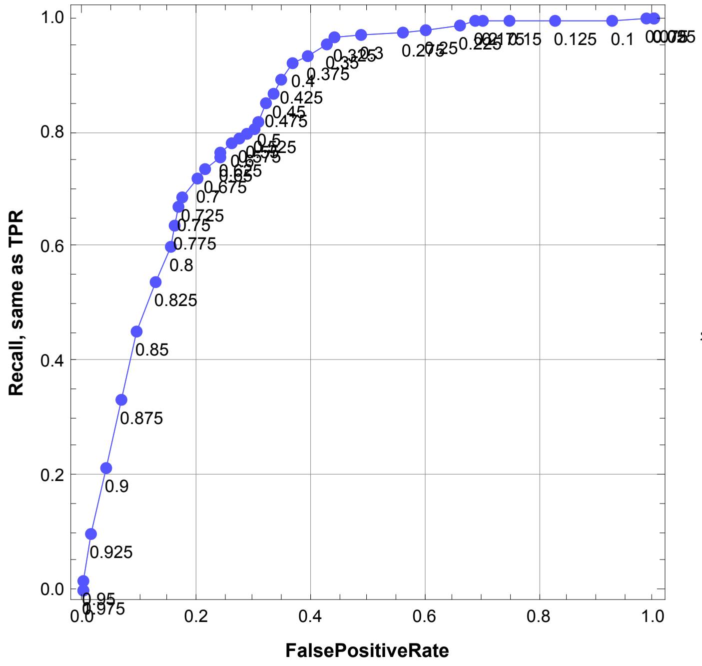
In the following table each pipeline operation is combined together with a short explanation and the context keys after its execution.

operation	explanation	context keys
`ClConUnit[dsTitanic] ⟹`	lift data to the monad	{}
`ClConSplitData[0.7`, 0.1`] ⟹`	split the data 0.7 for training, 0.07 for validation	{trainingData, testData, validationData}
`ClConMakeClassifier ⟹`	make a classifier	{trainingData, testData, validationData, classifier}
`ClConClassifierMeasurements["Accuracy"] ⟹`	compute classifier accuracy (over the test data)	{trainingData, testData, validationData, classifier}
`ClConEchoValue ⟹`	echo the pipeline value	{trainingData, testData, validationData, classifier}
`ClConROCData ⟹`	compute ROC data	{trainingData, testData, validationData, classifier, rocData}
`ClConROCPlot["FalsePositiveRate", "Recall"] ⟹`	plot ROC curve	{trainingData, testData, validationData, classifier, rocData}
`ClConTakeClassifier`	take the classifer from the context	{}

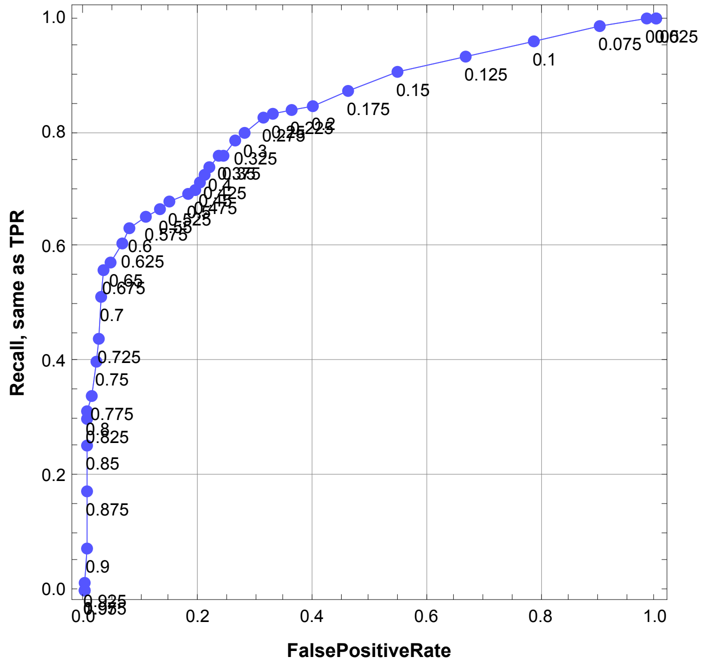
Here is the output of the pipeline:

» value: <|Accuracy → 0.770992|>

» ROC plot(s): ⟨| died →



, survived →



|⟩

The last column of the dataset is assumed to be the one with the class labels.

Monad elements

In this section we show that `ClCon` has all of the properties listed in the previous section.

The monad head

The monad head is `ClCon`. Anything wrapped in ClCon can serve as monad’s pipeline value. It is better though to use the constructor ClConUnit. (Which adheres to the monad defintion in [Wk1].)

```
In[101]:= ClCon[{{1, "a"}, {2, "b"}}, <||>] ⟹ ClConSummarizeData;
```

» summaries: {Anonymous → {

1 column 1	
1st Qu	1
Min	1
Mean	1.5
Median	1.5
3rd Qu	2
Max	2

,

2 column 2	
a	1
b	1

}}

Lifting data to the monad

The function lifting the data into the monad `ClCon` is `ClConUnit`.

The lifting to the monad marks the beginning of the monadic pipeline. It can be with done data or without data. Examples follow.

```
In[91]:= ClConUnit[dsData] ⟹ ClConSummarizeData;
```

» summaries: {Anonymous → {...}}

1 number		2 feature1		3 feature2		4 label	
Min	9	1st Qu	0	Min	0	False	207
1st Qu	255	Min	0	1st Qu	2	True	193
Mean	491.663	Median	1	Mean	4.5375		
Median	501	Mean	1.02	Median	5		
3rd Qu	733	3rd Qu	2	3rd Qu	7		
Max	998	Max	2	Max	9		

```
In[90]:= ClConUnit[] ⟹ ClConSetTrainingData[dsData] ⟹ ClConSummarizeData;
```

» summaries: {trainingData → {...}}

1 number		2 feature1		3 feature2		4 label	
Min	9	1st Qu	0	Min	0	False	207
1st Qu	255	Min	0	1st Qu	2	True	193
Mean	491.663	Median	1	Mean	4.5375		
Median	501	Mean	1.02	Median	5		
3rd Qu	733	3rd Qu	2	3rd Qu	7		
Max	998	Max	2	Max	9		

See the section “Getters and Setters” for more details of getting and setting values in `ClCon` contexts.

Data splitting

The splitting is made with ClConSplitData, which takes up to two arguments and options.
The first argument specifies the fraction of training data. The second argument -- if given -- specifies the fraction of the validation part of the training data.

Data splitting demonstration examples follow.

Here are the dimensions of the dataset `dsData`:

```
In[111]:= Dimensions[dsData]
Out[111]= {400, 4}
```

Here we split the data into 70% for training and 30% for testing and then we verify that the corresponding number of rows add to the number of rows of `dsData`:

```
In[126]:= Map[Dimensions, ClConUnit[dsData] ⟹ ClConSplitData[0.7] ⟹ ClConTakeValue]
          Total[First /@ %]
Out[126]= <|trainingData → {279, 4}, testData → {121, 4}|>
Out[127]= 400
```

In the following we split the data into 70% for training and 30% for testing, then the training data is further split into 90% for training and 10% for classifier training validation; then we verify that the number of rows add up.

```
In[128]:= Map[Dimensions,
           ClConUnit[dsData] ⟹ ClConSplitData[0.7, 0.1] ⟹ ClConTakeValue]
          Total[First /@ %]
Out[128]= <|trainingData → {250, 4}, testData → {121, 4}, validationData → {29, 4}|>
Out[129]= 400
```

Classifier training

The monad ClCon supports both single classifiers obtained with Classify and classifier ensembles obtained with Classify and managed with the package “ClassifierEnsembles.m”, [AAp4].

Single classifier training

With the following pipeline we take the Titanic data, split it into 75/25 % parts, train a Logistic Regression classifier, and finally pull that classifier from the monad.

In[130]:=
```
cf =
  ClConUnit[dsTitanic] ⟹
   ClConSplitData[0.75] ⟹
   ClConMakeClassifier["LogisticRegression"] ⟹
   ClConTakeClassifier;
```

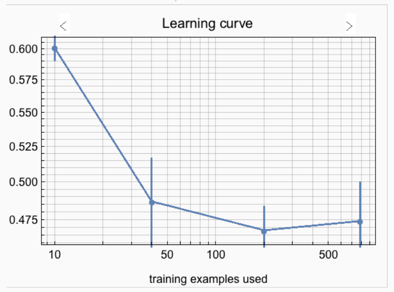
Here is information of the obtained classifier:

In[131]:=
```
Magnify[ClassifierInformation[cf], 0.6]
```

Out[131]=

Classifier information

Input type	Mixed (number: 4)
Classes	died, survived
Method	LogisticRegression
Accuracy	78.1% ± 1.1%
Loss	0.468 ± 0.015
Single evaluation time	2.05 ms/example
Batch evaluation speed	183. examples/ms
Classifier memory	317. kB
Training examples used	981 examples
Training time	3.43 s



If we want to pass parameters to the classifier training we can use the Method option. Here we train a Random Forest classifier with 400 trees:

In[134]:=
```
cf =
  ClConUnit[dsTitanic] ⟹
   ClConSplitData[0.75] ⟹
   ClConMakeClassifier[Method → {"RandomForest", "TreeNumber" → 400}] ⟹
   ClConTakeClassifier;
```

In[136]:=
```
ClassifierInformation[cf, "TreeNumber"]
```

Out[136]=
```
400
```

Classifier ensemble training

With the following pipeline we take the Titanic data, split it into 75/25 % parts, train a classifier ensemble of three Logistic Regression classifiers and two Nearest Neighbors classifier using random sampling of 90% of the training data, and finally pull that classifier ensemble from the monad.

In[132]:=
```
ensemble =
  ClConUnit[dsTitanic] ⟹
   ClConSplitData[0.75] ⟹
   ClConMakeClassifier[{{"LogisticRegression", 0.9, 3}, {"NearestNeighbors", 0.9, 2}}] ⟹
   ClConTakeClassifier;
```

The classifier ensemble is simply an Association with keys that are automatically derived names and corresponding values that are classifiers.

In[133]:=
```
ensemble
```

Out[133]=
```
⟨|LogisticRegression[1,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ],
 LogisticRegression[2,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ], LogisticRegression[3,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ],
 NearestNeighbors[1,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ], NearestNeighbors[2,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ]|⟩
```

Here are the training times of the classifiers in the obtained ensemble:

In[ ]:=
```
ClassifierInformation[#, "TrainingTime"] & /@ ensemble
```

Out[ ]=
```
⟨|LogisticRegression[1,0.9] → 3.4199 s , LogisticRegression[2,0.9] → 3.41272 s ,
 LogisticRegression[3,0.9] → 3.4329 s , NearestNeighbors[1,0.9] → 2.00509 s , NearestNeighbors[2,0.9] → 2.01433 s |⟩
```

A more precise specification can be given using associations. The specification

```
<|"method" → "LogisticRegression", "sampleFraction" → 0.9, "numberOfClassifiers" → 3, "samplingFunction" → RandomChoice|>
```

says: make three Logistic regression classifiers for each taking 90% of the training data using the function `RandomChoice`.

Here is a pipeline specification equivalent to the pipeline specification above:

In[138]:=
```
ensemble2 =
  ClConUnit[dsTitanic] ⟹
   ClConSplitData[0.75] ⟹
   ClConMakeClassifier[{<|"method" → "LogisticRegression", "sampleFraction" -> 0.9, "numberOfClassifiers" -> 3,
      "samplingFunction" → RandomSample|>, <|"method" → "NearestNeighbors", "sampleFraction" -> 0.9, "numberOfClassifiers" → 2, "samplingFunction" → RandomSample|>}] ⟹
   ClConTakeClassifier;
```

In[139]:=
```
ensemble2
```

Out[139]=
```
⟨|LogisticRegression[1,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ],
 LogisticRegression[2,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ], LogisticRegression[3,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ],
 NearestNeighbors[1,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ], NearestNeighbors[2,0.9] → ClassifierFunction[ Input type: Mixed (number: 4) Classes: died, survived ]|⟩
```

Classifier testing

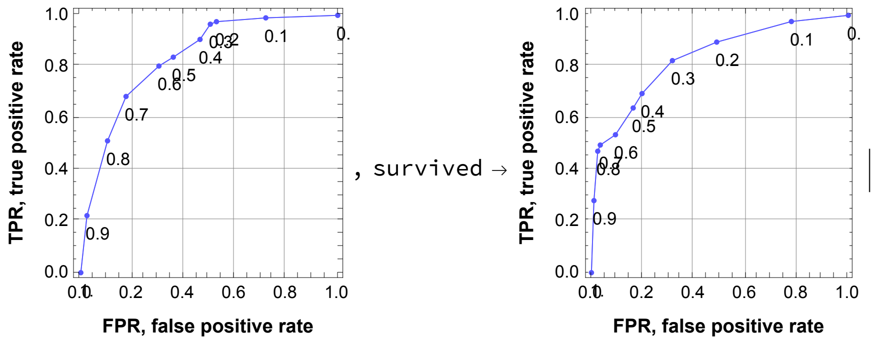
In[∘]:=
```
mObj =
  ClConUnit[dsTitanic]⟹
   ClConSplitData[0.75]⟹
   ClConMakeClassifier["NearestNeighbors"];
```

In[∘]:=
```
mObj⟹
 ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall"}]⟹
 ClConTakeValue
```

Out[∘]=
```
<|Accuracy → 0.762195, Precision → <|died → 0.790698, survived → 0.707965|>, Recall → <|died → 0.837438, survived → 0.64|>|>
```

In[∘]:=
```
mObj⟹
  ClConROCPlot["FPR", "TPR", ImageSize → Small, "ROCRange" → Range[0, 1, 0.1]];
```

» ROC plot(s): <|died → , survived → |>



Variable importance finding

In[∘]:=
```
mObj⟹
 ClConAccuracyByVariableShuffling⟹
 ClConTakeValue
```

Out[∘]=
```
<|None → 0.762195, id → 0.740854, passengerClass → 0.737805, passengerAge → 0.728659, passengerSex → 0.640244|>
```

Setters and getters

The monad can be seen as a OOP class that with certain setter and getter methods.

In[∘]:=
```
RecordsSummary[mObj⟹ClConTakeTrainingData]
```

Out[∘]=

1 id

Min	1
1st Qu	324.75
Mean	653.492
Median	656
3rd Qu	980.25
Max	1309

2 passengerClass

3rd	530
1st	244
2nd	207

3 passengerAge

Min	-1
1st Qu	10
Median	20
Mean	23.2008
3rd Qu	32.5
Max	80

4 passengerSex

male	642
female	339

5 passengerSurvival

died	606
survived	375

In[◦]:=
```
RecordsSummary[mObj⟹ClConTakeTestData]
```

Out[◦]=

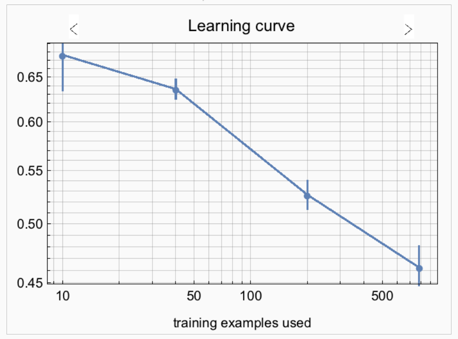
1 id		2 passengerClass		3 passengerAge		4 passengerSex		5 passengerSurvival	
Min	5	3rd	179	Min	-1	male	201	died	203
1st Qu	342	1st	79	1st Qu	10	female	127	survived	125
Median	645.5	2nd	70	Median	20				
Mean	659.509			Mean	24.5945				
3rd Qu	990			3rd Qu	40				
Max	1303			Max	70				

In[◦]:=
```
ClassifierInformation[mObj⟹ClConTakeClassifier] // Magnify[#, 0.7] &
```

Out[◦]=

Classifier information

Input type	Mixed (number: 4)
Classes	died, survived
Method	NearestNeighbors
Accuracy	78.4% ± 1.1%
Loss	0.462 ± 0.018
Single evaluation time	2.14 ms/example
Batch evaluation speed	81.3 examples/ms
Classifier memory	223. kB
Training examples used	981 examples
Training time	1.96 s



Implementation notes

The ClCon package, MonadicContextualClassification.m, [AAp3], is based on the packages [AAp1, AAp4-AAp9]. It was developed using Mathematica and the Mathematica plug-in for IntelliJ IDEA, by Patrick Scheibe , [PS1].

Case study examples

Unit tests

```
testObject = TestReport["/Users/aantonov/MathematicaForPrediction/UnitTests/MonadicContextualClassification-Unit-Tests.wlt"]
```

TestReportObject[Title: Test Report: MonadicContextualClassification-Unit-Tests.wlt Success rate: 100% Tests run: 16]

```
testObject["TestResults"]
```

⟨|1 → TestResultObject[Outcome: Success Test ID: LoadPackage], 2 → TestResultObject[Outcome: Success Test ID: EvenOddDataset],
3 → TestResultObject[Outcome: Success Test ID: EvenOddDataMLRules], 4 → TestResultObject[Outcome: Success Test ID: DataToContext–no–[]], 5 → TestResultObject[Outcome: Success Test ID: DataToContext–with–[]],
6 → TestResultObject[Outcome: Success Test ID: ClassifierMaking–with–Dataset–1], 7 → TestResultObject[Outcome: Success Test ID: ClassifierMaking–with–MLRules–1],
8 → TestResultObject[Outcome: Success Test ID: AccuracyByVariableShuffling–1], 9 → TestResultObject[Outcome: Success Test ID: ROCData–1], 10 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–different–methods–1],
11 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–different–methods–2–cont], 12 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–different–methods–3–cont],
13 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–one–method–1], 14 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–one–method–2],
15 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–one–method–3–cont], 16 → TestResultObject[Outcome: Success Test ID: ClassifierEnsemble–one–method–4–cont]|⟩

Future plans

Workflows generation with natural language commands

References

Packages

[AAp1] Anton Antonov, State monad code generator Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/StateMonadCodeGenerator.m .

[AAp2] Anton Antonov, Monadic tracing Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicTracing.m .

[AAp3] Anton Antonov, Monadic contextual classification Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicContextualClassification.m .

[AAp4] Anton Antonov, Classifier ensembles functions Mathematica package, (2016), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/ClassifierEnsembles.m .

[AAp5] Anton Antonov, Receiver operating characteristic functions Mathematica package, (2016), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/ROCFunctions.m .

[AAp6] Anton Antonov, Variable importance determination by classifiers implementation in Mathematica, (2015), MathematicaForPrediction at GitHub.

URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/VariableImportanceByClassifiers.m .

[AAp7] Anton Antonov, MathematicaForPrediction utilities, (2014), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MathematicaForPredictionUtilities.m .

[AAp8] Anton Antonov, Cross tabulation implementation in Mathematica, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/CrossTabulate.m .

[AAp9] Anton Antonov, SSparseMatrix Mathematica package, SSparseMatrix.m, (2018), MathematicaForPrediction at GitHub.

[AAp10] Anton Antonov, Obtain and transform Mathematica machine learning data-sets, GetMachineLearningDataset.m, (2018), MathematicaVsR at GitHub.

ConverationalAgents Packages

[AAp7] Anton Antonov, Classifier workflows grammar in EBNF, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

[AAp8] Anton Antonov, Classifier workflows grammar Mathematica unit tests, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

[AAp9] Anton Antonov, ClCon translator Mathematica package, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

MathematicaForPrediction articles

[AA1] Anton Antonov, Monad code generation and extension, (2017), MathematicaForPrediction at GitHub, https://github.com/antononcube/MathematicaForPrediction.

[AA2] Anton Antonov, “ROC for classifier ensembles, bootstrapping, damaging, and interpolation”, (2016), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2016/10/15/roc-for-classifier-ensembles-bootstrapping-damaging-and-interpolation/ .

Other

[Wk1] Wikipedia entry, Monad, URL: https://en.wikipedia.org/wiki/Monad_(functional_programming) .

[PS1] Patrick Scheibe, Mathematica (Wolfram Language) support for IntelliJ IDEA, (2013-2018), Mathematica-IntelliJ-Plugin at GitHub.