

A monad for classification workflows

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

Version 0.8

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:

»	<code>ClConUnit[dsTitanic] =></code>	lift the data to the monad
	<code>ClConSplitData[0.75] =></code>	split the data
	<code>ClConMakeClassifier["LogisticRegression"] =></code>	create a classifier
	<code>ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall"}] =></code>	compute classifier measurements
	<code>ClConEchoValue</code>	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]:

```
In[1]:= 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: MathematicaForPredictionUtilities.m
» Importing from GitHub: MosaicPlot.m
» Importing from GitHub: CrossTabulate.m
» Importing from GitHub: StateMonadCodeGenerator.m
» Importing from GitHub: ClassifierEnsembles.m
» Importing from GitHub: ROCFunctions.m
» Importing from GitHub: VariableImportanceByClassifiers.m
» Importing from GitHub: SSparseMatrix.m
» Importing from GitHub: OutlierIdentifiers.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.

Remark: 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[4]:= dsTitanic = GetMachineLearningDataset["Titanic"];
dsMushroom = GetMachineLearningDataset["Mushroom"];
dsWineQuality = GetMachineLearningDataset["WineQuality"];
```

Here is are the dimensions of the datasets:

```
In[7]:= Dataset[Dataset[Map[Prepend[Dimensions[ToExpression[#]], #] &, {"dsTitanic", "dsMushroom", "dsWineQuality"}]][All, AssociationThread[{"name", "rows", "columns"}, #] &]]
```

Out[7]=

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

Here is the summary of `dsTitanic`:

```
In[8]:= ClConUnit[dsTitanic] ==> ClConSummarizeData["MaxTallies" -> 12];
```

» summaries: {Anonymous -> {

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

}

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

```
In[9]:= ClConUnit[dsMushroom] ==> ClConSummarizeDataLongForm["MaxTallies" -> 12];
```

1 RowID	2 Variable	3 Value
1	bruises?	white 21 402
10	cap-Color	smooth 12 668
100	cap-Shape	partial 8124
1000	cap-Surface	free 7914
1001	edibility	one 7488
» summaries: {Anonymous -> {1002	gill-Attachment	close 6812 }}
1003	gill-Color	brown 6356
1004	gill-Size	broad 5612
1005	gill-Spacing	pink 5380
1006	habitat	False 4748
1007	id	silky 4676
(Other)	(Other)	103 796

Here is the summary of dsWineQuality in long form:

```
In[10]:= ClConUnit[dsWineQuality] ==> ClConSummarizeDataLongForm["MaxTallies" -> 12];
```

1 RowID	2 Variable	3 Value
1	alcohol	4898
10	chlorides	4898
100	density	4898
1000	fixedAcidity	Min 0.009
1001	freeSulfurDioxide	1st Qu 0.46
» summaries: {Anonymous -> {1002	id	Median 4.6 }}
1003	pH	3rd Qu 12.
1004	residualSugar	Mean 204.533
1005	sulphates	Max 4898
1006	totalSulfurDioxide	4898
1007	volatileAcidity	4898
(Other)	(Other)	9777

“Quick” data

In this subsection we make up some data that is used for illustrative purposes.

```
In[11]:= 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[15]:= {400, 4}
```

Here is a sample of the data:

```
In[16]:= RandomSample[dsData, 6]
```

Out[16]=

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[17]:= 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[18]:= 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	
» 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[20]:= arrData = Normal[dsData[All, Values]];
```

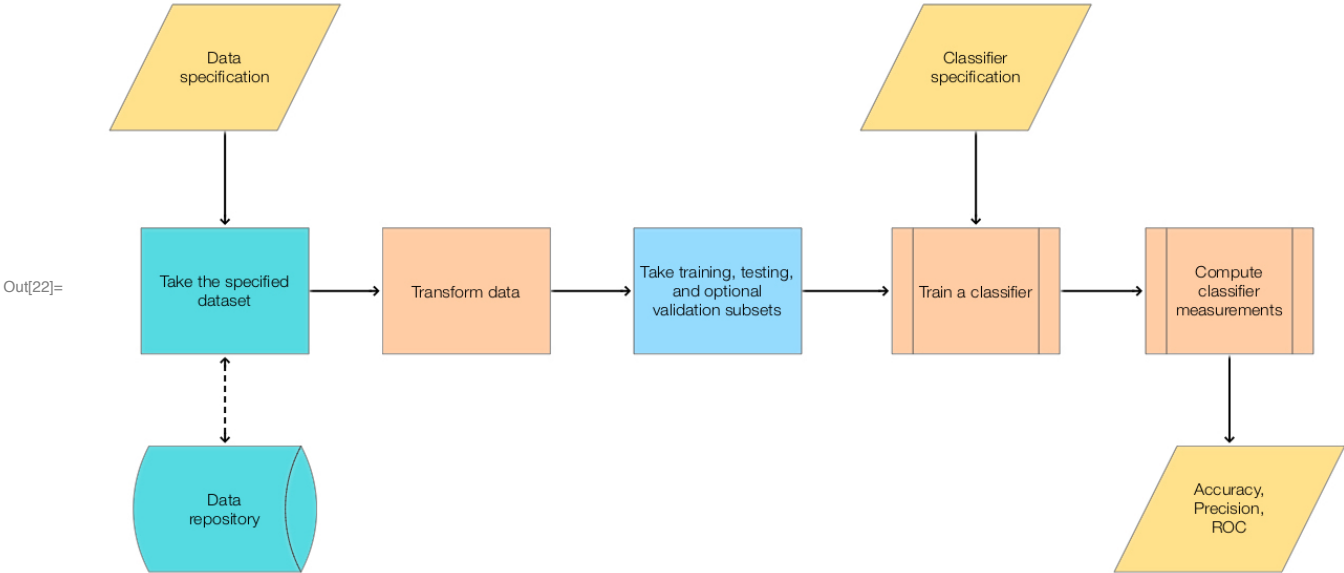
Design considerations

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.

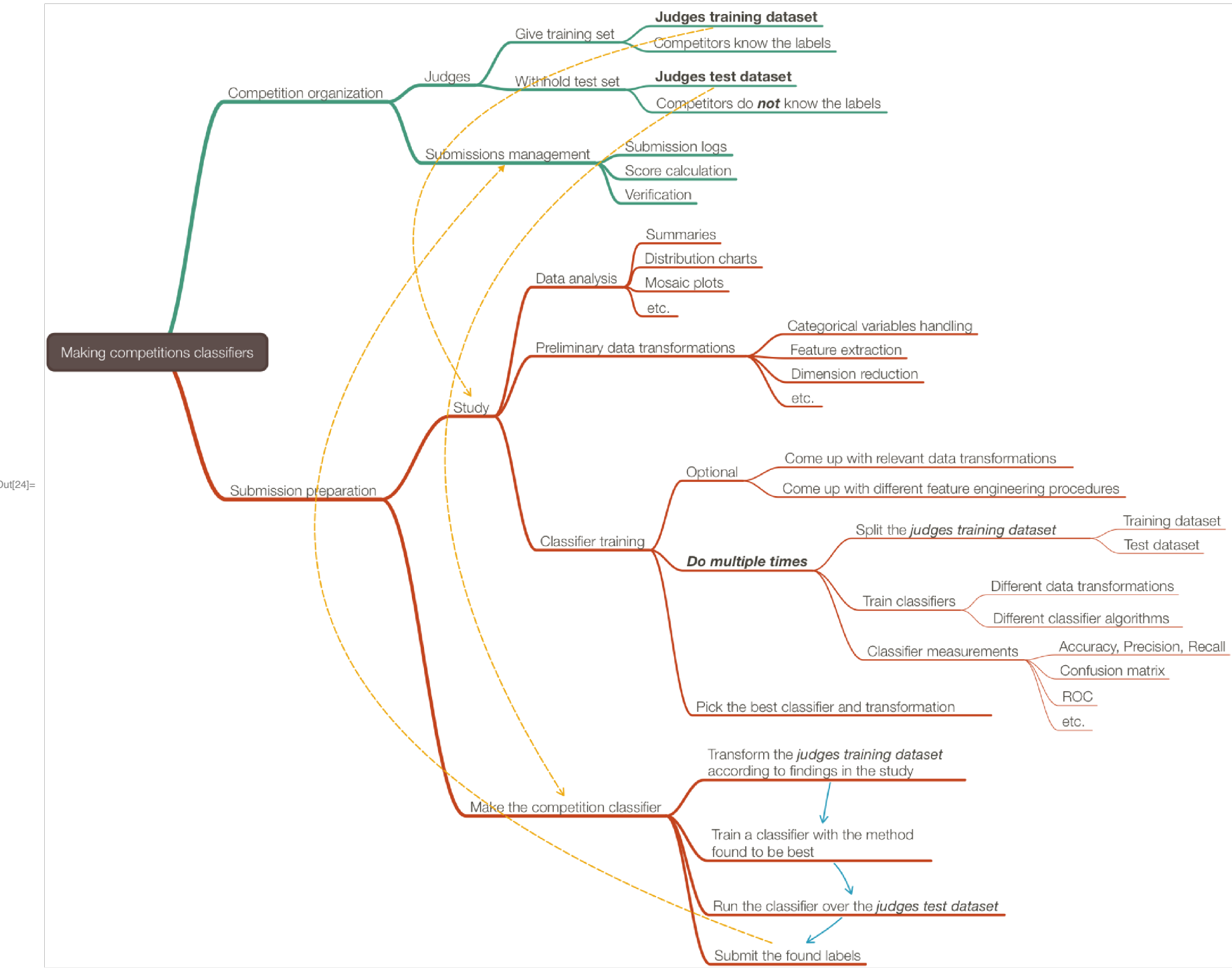
The following diagram shows the steps.

```
In[21]:= imgSimpleFlow = Import["~/Documents/Conceptual diagrams/Classification workflows/Classification-workflow-horizontal-layout.jpg"];
imgSimpleFlow
AutoCollapse[]
```



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.

```
In[24]:= imgMM = Import["~/Documents/Conceptual diagrams/Classification workflows/Making-competitions-classifiers-mind-map.png"]
AutoCollapse[]
```

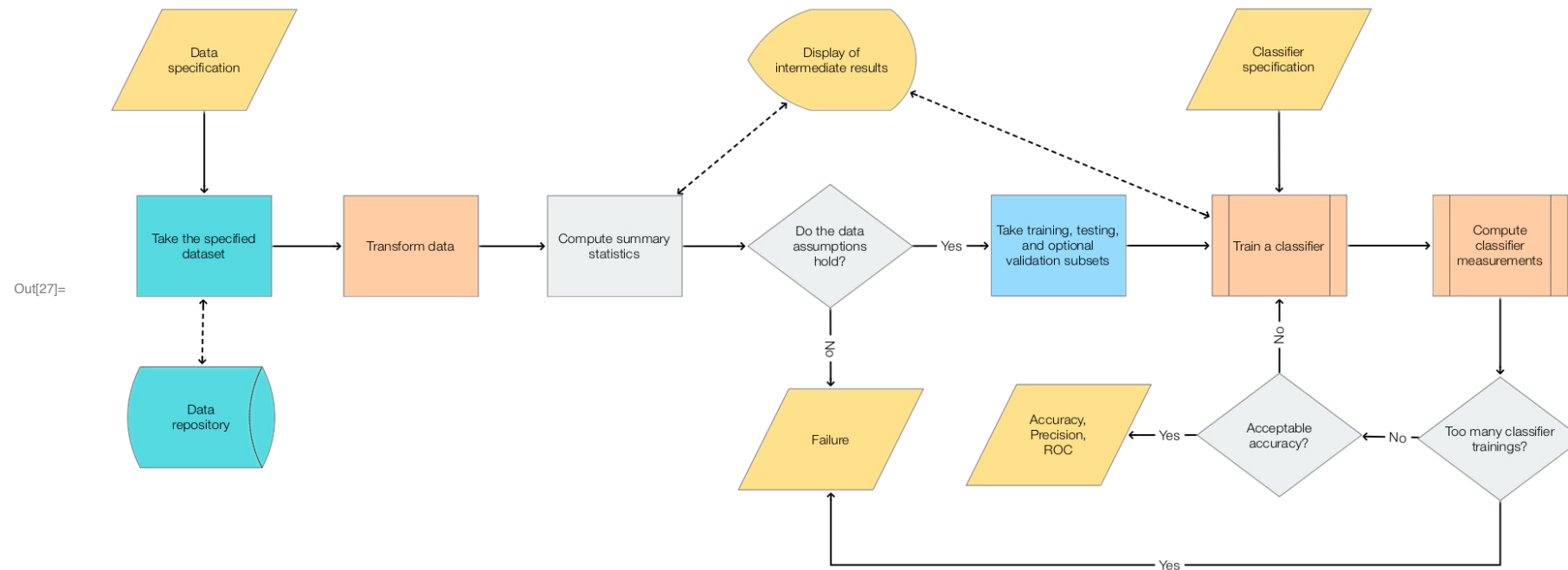


In view of the mind-map above we can come up with the following flow-chart that is an elaboration on the main, simple workflow flow-chart.

```

In[26]:= imgExtendedFlow = Import["~/Documents/Conceptual diagrams/Classification workflows/Classification-workflow-extended.jpg"];
imgExtendedFlow
AutoCollapse[]

```



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 “**C**lassification with **C**ontext”.

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

$$\text{ClCon}[_] \xrightarrow{\text{ClConBind}[\text{ClCon}[_], f_-]} f_1 \xrightarrow{\text{ClConBind}[\text{ClCon}[_], f_-]} f_2 \xrightarrow{\text{ClConBind}[\text{ClCon}[_], f_-]} \dots \xrightarrow{\text{ClConBind}[\text{ClCon}[_], f_-]} f_k \quad (1)$$

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

$$\text{ClCon}[pval_ , context_] \xrightarrow{\text{ClConBind}[m_ , f_]} \dots \left(\begin{cases} f_i[\text{\$ClConFailure}] & m \equiv \text{\$ClConFailure} \\ f_i[x_ , c_Association] & m \text{ is ClCon}[x_ , c_Association] \\ \text{\$ClConFailure} & \text{otherwise} \end{cases} \right) \xrightarrow{\text{ClConBind}[m_ , f_]} \dots \quad (2)$$

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.

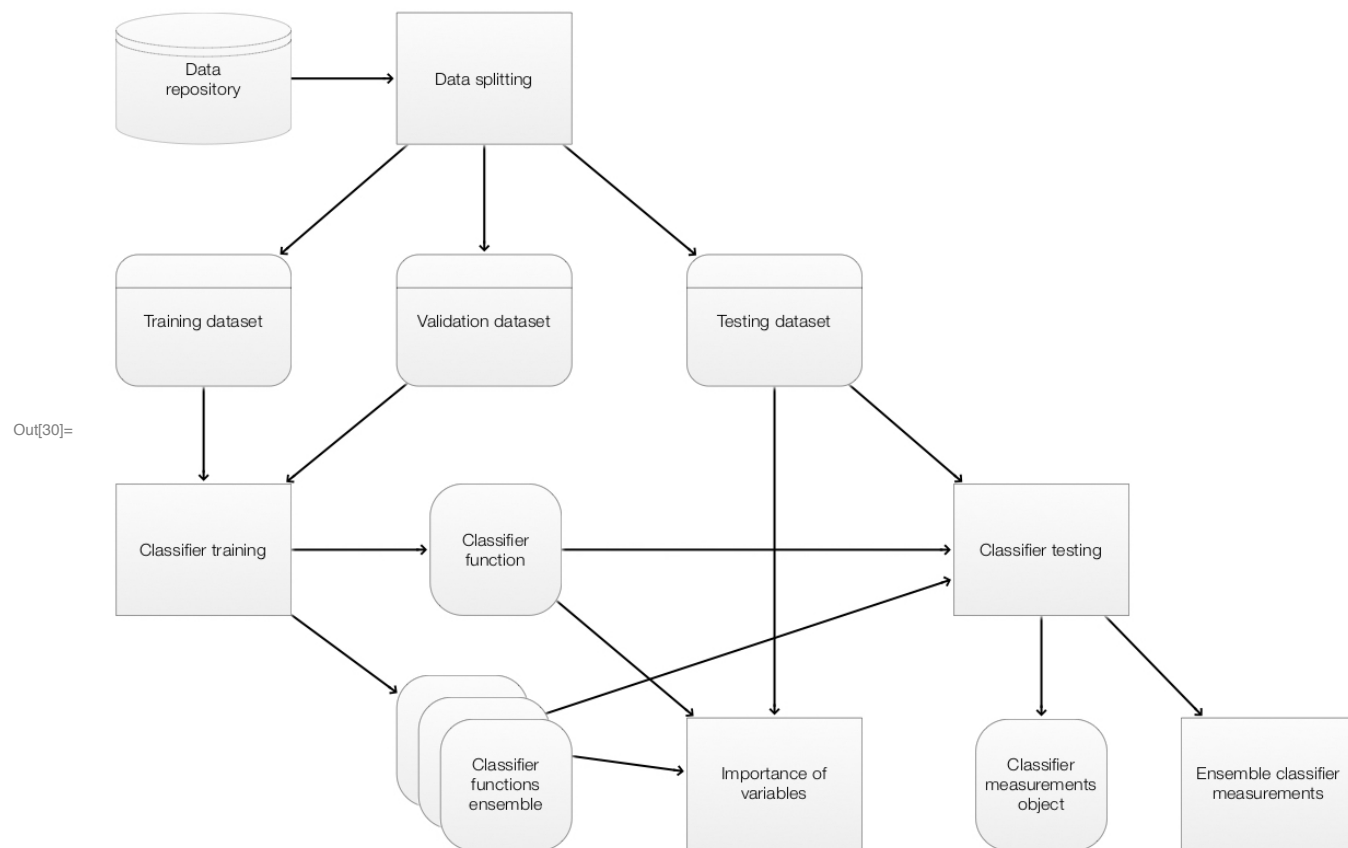
- Rapid specification of non-trivial classification workflows.
- The monad works with different data types: Dataset, lists of machine learning rules, full arrays.
- The pipeline values can be of different types. Generally, every monad function modifies the pipeline value; some modify the context.
- The 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.)


```

In[29]:= img = Import["~/Documents/Conceptual diagrams/Classification workflows/ClCon-components-interaction.jpg"];
img
AutoCollapse[]

```



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

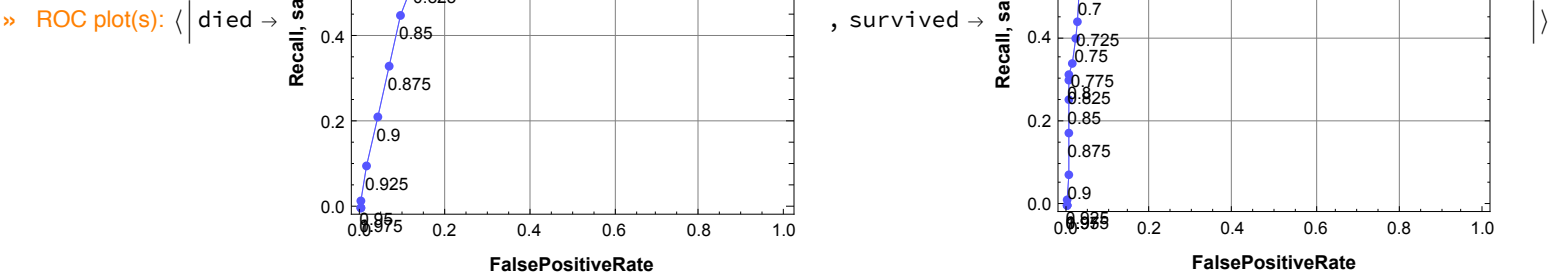
Obviously, we can put in the context any object through the generic operations of the State monad of the package StateMonadGenerator.m, [Aap1].

ClCon overview

When using a monad we lift certain data into the “monad space”, using monad’s operations we navigate computations in that space, and at some point we take a result from it.

With the approach taken in this document the “lifting” into the ClCon monad is done with the function `ClConUnit`. Results from the monad can be obtained with the functions `ClConTakeValue`, `ClConContext`, or with the other ClCon functions with the prefix “ClConTake” (see below.)

» value: <| Accuracy → 0.770992 |>



In the specified pipeline computation the last column of the dataset is assumed to be the one with the class labels.

The ClCon functions are separated into four groups:

- operations,
- setters,
- takers,
- State Monad generic functions.

An overview of the those functions is given in the tables in next two sub-sections. The next section, “Monad elements”, gives details and examples for the usage of the ClCon operations.

Monad functions interaction with the pipeline value and context

The following table gives an overview the interaction of the ClCon monad functions with pipeline value and context.

Out[*=]=

#	name	echoes result	puts in context	uses from context	uses pipeline value
1	<i>operations</i>				
2	ClConAccuracyByVariableShuffling	no	none	{classifier, testData}	no
3	ClConAssignVariableNames	no	{variableNames}	{trainingData}	no
4	ClConClassifierMeasurements	no	none	{classifier, testData}	no
5	ClConClassifierMeasurementsByThreshold	no	none	{classifier, testData}	no
6	ClConGetVariableNames	no	none	{trainingData, variableNames}	tries first
7	ClConEchoVariableNames	yes	none	{variableNames}	yes
8	ClConMakeClassifier	no	{classifier}	{trainingData, validationData}	tries first
9	ClConOutlierPosition	no	none	{trainingData, testData}	tries first
10	ClConRecoverData	no	none	{trainingData, testData}	tries first
11	ClConROCDData	no	{rocData}	{classifier, testData}	no
12	ClConROCListLinePlot	yes	{rocData}	{classifier, testData}	no
13	ClConROCListLinePlot	yes	none	{rocData}	no
14	ClConROCPlot	yes	{rocData}	{classifier, testData}	no
15	ClConROCPlot	yes	none	{rocData}	no
16	ClConSplitData	no	{trainingData, testData}	none	yes
17	ClConSummarizeData	yes	none	{trainingData, testData, validationData}	tries first
18	ClConSummarizeDataLongForm	yes	none	{trainingData, testData, validationData}	tries first
19	<i>setters</i>				
20	ClConSetClassifier	no	classifier	none	no
21	ClConSetTestData	no	testData	none	no
22	ClConSetTrainingData	no	trainingData	none	no
23	ClConSetValidationData	no	validationData	none	no
24	ClConSetVariableNames	no	variableNames	none	no
25	<i>takers</i>				
26	ClConTakeClassifier	no	none	classifier	no
27	ClConTakeData	no	none	data	no
28	ClConTakeROCDData	no	none	rocData	no
29	ClConTakeTestData	no	none	testData	no
30	ClConTakeTrainingData	no	none	trainingData	no
31	ClConTakeValidationData	no	none	validationData	no
32	ClConTakeVariableNames	no	none	variableNames	no

State monad functions

Here are the ClCon State Monad functions (generated using the prefix “ClCon”), [AAp1, AA1]:

Out[*]=

#	name	description
1	ClCon	monad head
2	ClConAddToContext	adds the pipeline value into the context
3	ClConBind	monad binding function
4	ClConContexts	gives the contexts associated with a monad head
5	ClConDropFromContext	drops from the context elements specified by their keys
6	ClConEchoContext	echoes the context
7	ClConEchoFunctionContext	echoes the result of a function applied to the context
8	ClConEchoFunctionValue	echoes the result of a function applied to the pipeline value
9	ClConEchoValue	echoes the pipeline value
10	ClConFail	gives the monad failure symbol
11	ClConIfElse	chooses between two functions based on condition
12	ClConIterate	general iteration function
13	ClConModifyContext	modifies the context with the argument function
14	ClConModule	allows faster pipeline function specifications
15	ClConOption	ignores a result if it is failure
16	ClConPutContext	replaces the context with the argument
17	ClConRetrieveFromContext	using a key retrieves into the pipeline a value from the context
18	ClConSucceed	gives a success element of the form ClCon[_____]
19	ClConTakeContext	takes the context
20	ClConTakeValue	takes the pipeline value
21	ClConUnit	lifts to the monad
22	ClConUnitQ	gives True if monad unit
23	ClConWhen	executes a function based on a condition

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 definition in [Wk1].)

```
In[35]:= ClCon[{ {1, "a"}, {2, "b"} }, <| |>] ==> ClConSummarizeData;
```

1 column 1

1st Qu 1

Min 1

Mean 1.5 , a 1

Median 1.5 b 1

3rd Qu 2

Max 2

2 column 2

```
» summaries: {Anonymous -> { { {1, "a"}, {2, "b"} }, { {1, "a"}, {2, "b"} } } }
```

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[36]:= ClConUnit[dsData] ==> ClConSummarizeData;
```

	1 number	2 feature1	3 feature2	4 label
Min	9	1st Qu 0	Min 0	
1st Qu	255	Min 0	1st Qu 2	
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	

```
In[37]:= ClConUnit[] ==> ClConSetTrainingData[dsData] ==> ClConSummarizeData;
```

	1 number	2 feature1	3 feature2	4 label
Min	9	1st Qu 0	Min 0	
1st Qu	255	Min 0	1st Qu 2	
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	

(See the sub-section “Setters and Takers” for more details of setting and taking 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[38]:= Dimensions[dsData]
```

```
Out[38]:= {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[39]:= Map[Dimensions, ClConUnit[dsData] ==> ClConSplitData[0.7] ==> ClConTakeValue]
```

```
Total[First /@ %]
```

```
Out[39]:= <| trainingData -> {279, 4}, testData -> {121, 4} |>
```

```
Out[40]:= 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[41]:= Map[Dimensions,
  ClConUnit[dsData] ==> ClConSplitData[0.7, 0.1] ==> ClConTakeValue]
Total[First /@ %]
```

```
Out[41]:= <| trainingData -> {250, 4}, testData -> {121, 4}, validationData -> {29, 4} |>
```

```
Out[42]:= 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[43]:= cf =
  ClConUnit[dsTitanic] =>
    ClConSplitData[0.75] =>
      ClConMakeClassifier["LogisticRegression"] =>
        ClConTakeClassifier;
```

Here is information about the obtained classifier:

```
In[44]:= ClassifierInformation[cf, "TrainingTime"]
```

```
Out[44]= 3.6596 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[45]:= cf =
  ClConUnit[dsTitanic] =>
    ClConSplitData[0.75] =>
      ClConMakeClassifier[Method -> {"RandomForest", "TreeNumber" -> 400}] =>
        ClConTakeClassifier;
```

```
In[46]:= ClassifierInformation[cf, "TreeNumber"]
```

```
Out[46]= 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[47]:= 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[48]:= ensemble
```

```
Out[48]= {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[49]:= ClassifierInformation[#, "TrainingTime"] & /@ ensemble
Out[49]= <| LogisticRegression[1,0.9] → 3.55873 s , LogisticRegression[2,0.9] → 3.56161 s ,
  LogisticRegression[3,0.9] → 3.45833 s , NearestNeighbors[1,0.9] → 1.75196 s , NearestNeighbors[2,0.9] → 1.80373 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[50]:= 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[51]:= ensemble2
```

```
Out[51]= <| 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

The classifier testing is done with the testing data in the context.

Here is a pipeline that takes the Titanic data, splits it, and trains a classifier:

```
In[52]:= p =
  ClConUnit[dsTitanic] ⇒
  ClConSplitData[0.75] ⇒
  ClConMakeClassifier["DecisionTree"];
```

Here is how we compute selected classifier measures:

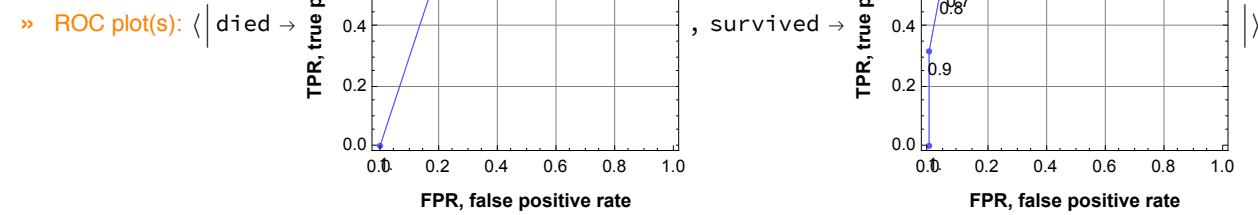
```
In[53]:= p ⇒
  ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall", "FalsePositiveRate"}] ⇒
  ClConTakeValue
Out[53]= <| Accuracy → 0.762195, Precision → <| died → 0.782805, survived → 0.719626 |>, Recall → <| died → 0.852217, survived → 0.616 |>, FalsePositiveRate → <| died → 0.384, survived → 0.147783 |> |>
```

Here we show the confusion matrix plot:

```
In[54]:= p ⇒ ClConClassifierMeasurements["ConfusionMatrixPlot"];
```

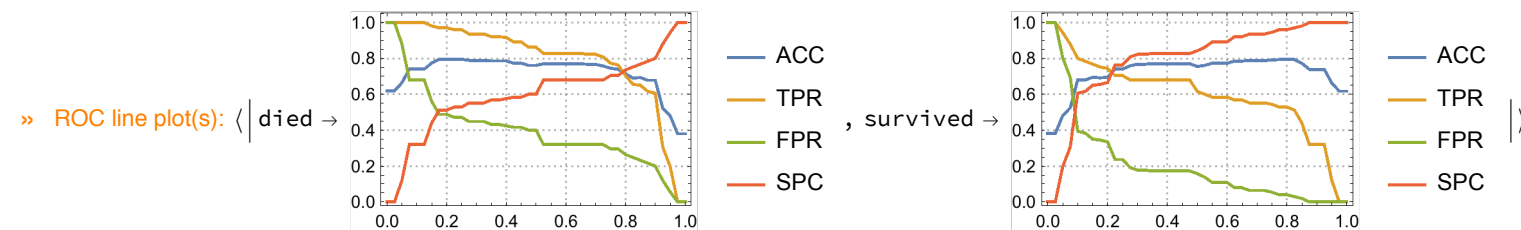
Here is how we plot ROC plots by specifying the ROC parameter range and the image size:


```
In[55]:= p =>
  ClConROCPlot["FPR", "TPR", "ROCRange" -> Range[0, 1, 0.1], ImageSize -> 200];
```



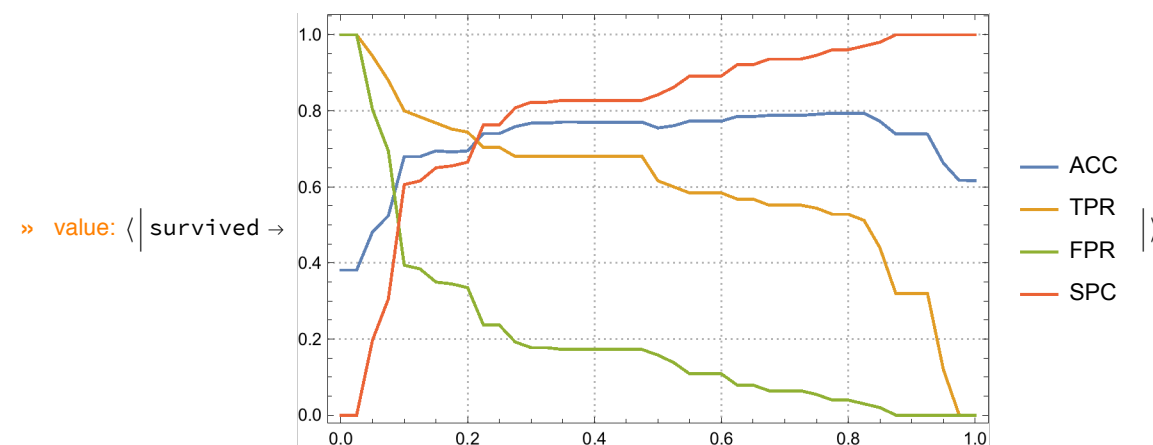
Here we plot ROC functions values (y-axis) over the ROC parameter (x-axis):

```
In[56]:= p => ClConROCListLinePlot[{"ACC", "TPR", "FPR", "SPC"}];
```



Note of the “ClConROC*Plot” functions automatically echo the plots. The plots are also made to be the pipeline value. Using the option specification “Echo” \rightarrow False the automatic echoing of plots can be prevented. With the option “TargetClasses” we can focus on specific class labels.

```
In[57]:= p =>
  ClConROCListLinePlot[{"ACC", "TPR", "FPR", "SPC"}, "Echo" -> False, "TargetClasses" -> "survived", ImageSize -> Medium] =>
  ClConEchoValue;
```



Variable importance finding

Using the pipeline constructed above let us find the most decisive variables using systematic random shuffling (as explained in [AA3]):

```
In[58]:= p ==>
  ClConAccuracyByVariableShuffling ==>
  ClConTakeValue
```

```
Out[58]:= <|None -> 0.762195, id -> 0.695122, passengerClass -> 0.734756, passengerAge -> 0.746951, passengerSex -> 0.554878|>
```

We deduce that “passengerSex” is the most decisive variable because its corresponding classification success rate is the smallest. (See [AA3] for more details.)

Setters and takers

The values from the monad context can be set or obtained with the corresponding “setters” and “takers” functions as summarized in previous section.

If other values are put in the context they can be obtained through the (generic) function `ClConTakeContext`:

```
In[59]:= p = ClConUnit[RandomReal[1, {2, 2}]] ==> ClConAddToContext["data"];
```

```
In[60]:= (p ==> ClConTakeContext) ["data"]
```

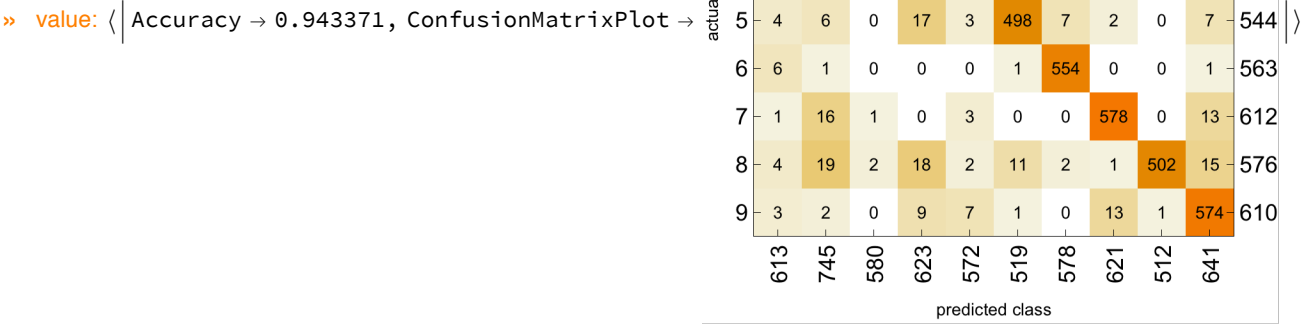
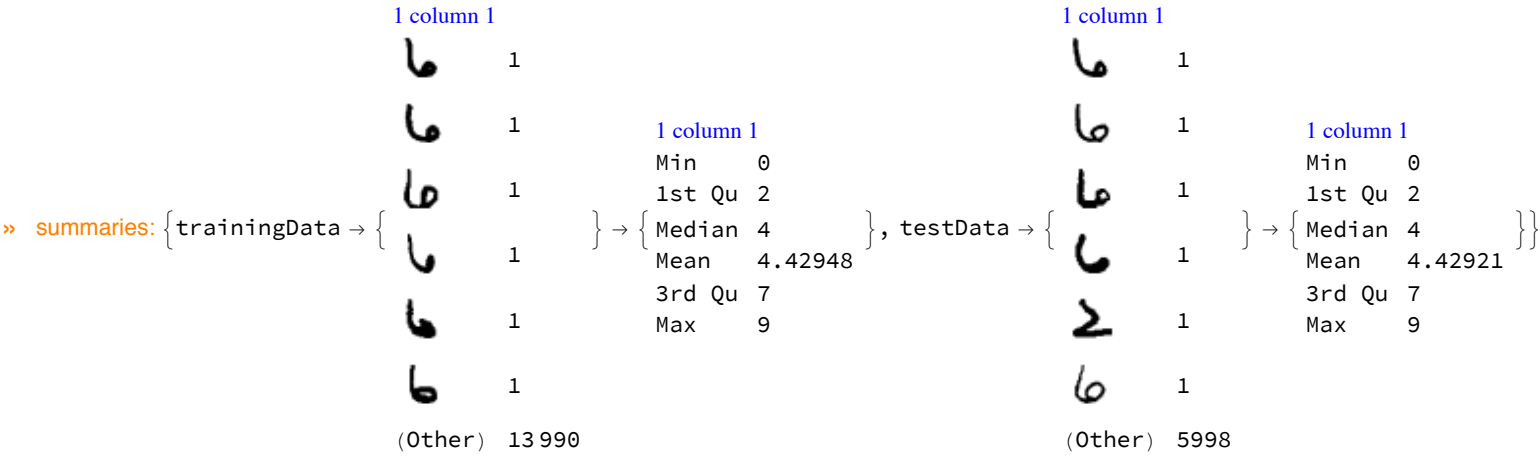
```
Out[60]:= {{0.862073, 0.267576}, {0.211994, 0.745763}}
```

Case study examples

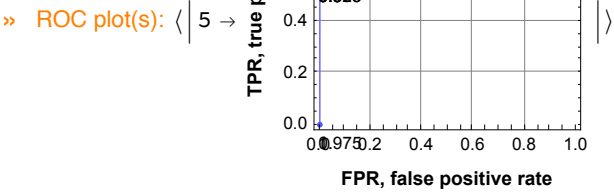
```
In[61]:= mnistData = ExampleData[{"MachineLearning", "MNIST"}, "Data"];
```

```
In[62]:= SeedRandom[3423]
```

```
p =
  ClConUnit[RandomSample[mnistData, 20 000]] ==>
  ClConSplitData[0.7] ==>
  ClConSummarizeData ==>
  ClConMakeClassifier["NearestNeighbors"] ==>
  ClConClassifierMeasurements[{"Accuracy", "ConfusionMatrixPlot"]} ==>
  ClConEchoValue;
```

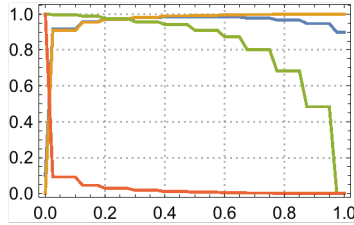
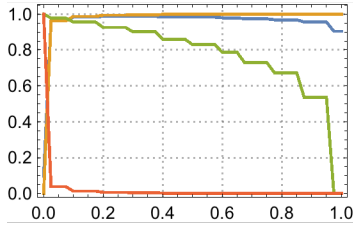
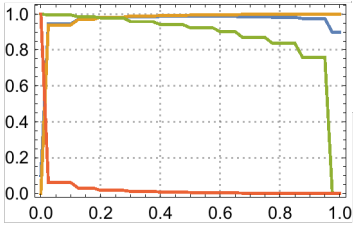
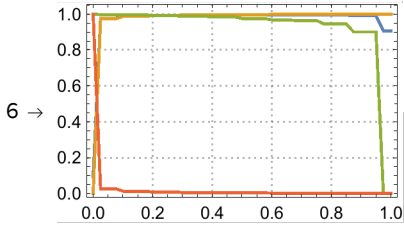
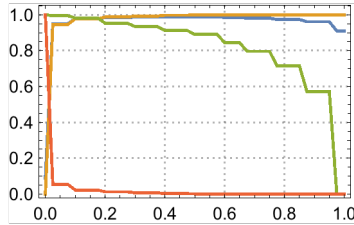
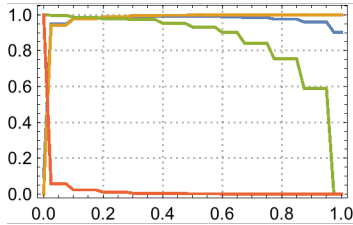
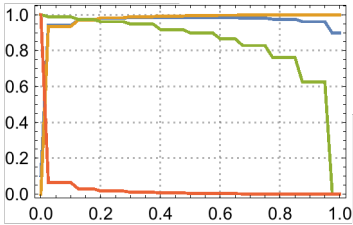
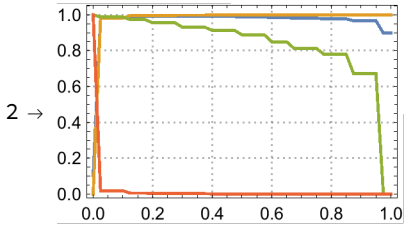
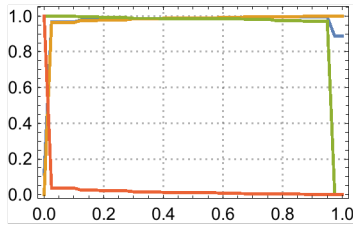
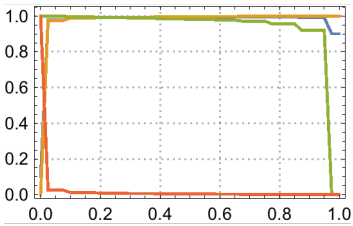


```
In[64]:= p=>ClConROCPlot["TargetClasses" -> 5];
```



```
In[65]:= p=>ClConROCListLinePlot[{"ACC", "SPC", "TPR", "FPR"}];
```

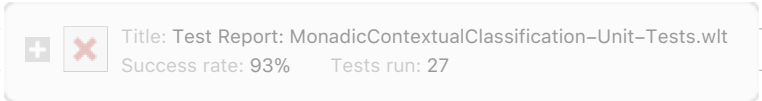
» ROC line plot(s):



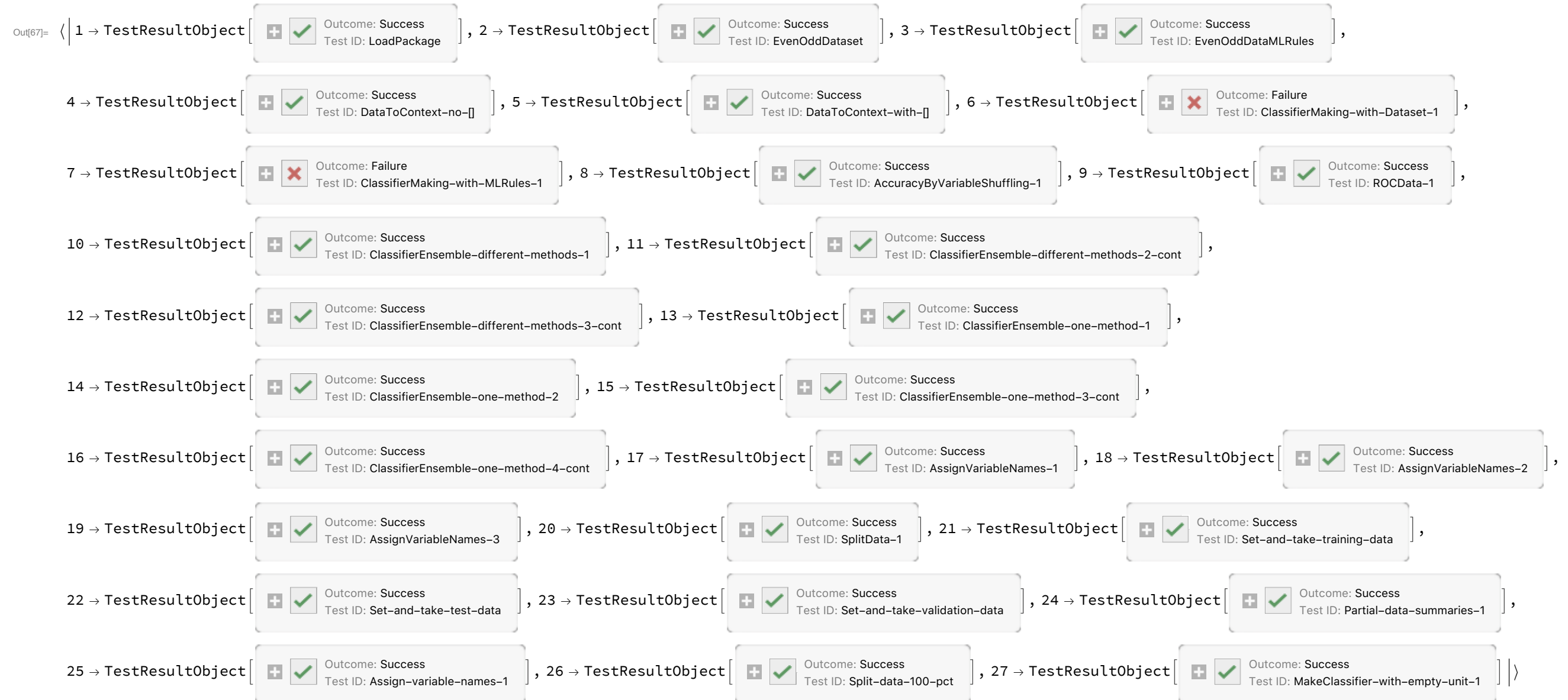
Unit tests

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

```
Out[66]= TestReportObject[
```

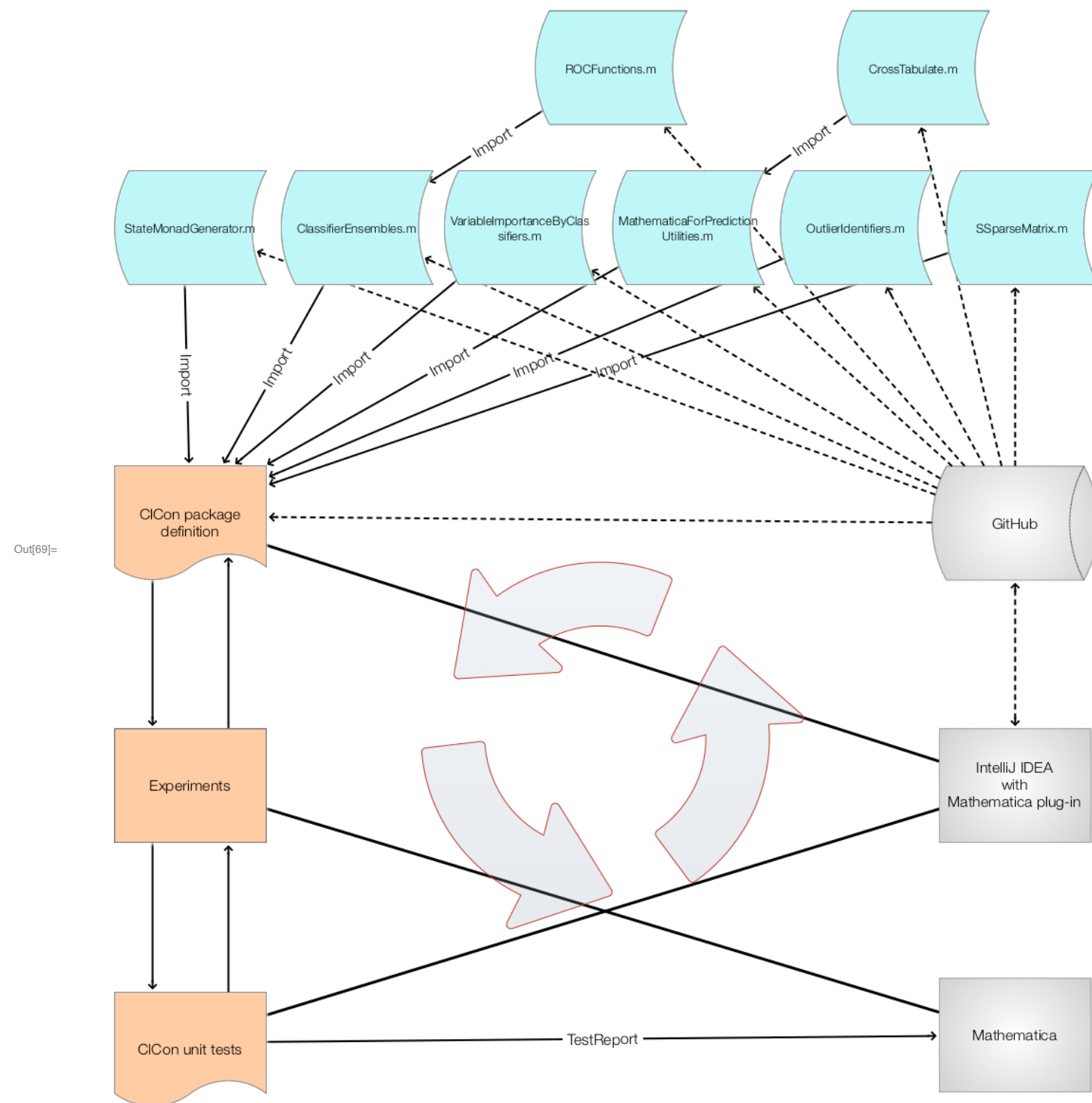


In[67]:= testObject["TestResults"]



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]. The following diagram shows the development workflow.



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/> .
- [AA3] Anton Antonov, “Importance of variables investigation guide”, (2016), MathematicaForPrediction at GitHub.
URL: <https://github.com/antononcube/MathematicaForPrediction/blob/master/MarkdownDocuments/Importance-of-variables-investigation-guide.md> .

Other

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