A monad for classification workflows

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

Version 1.0

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, Wk2].

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, AAp12]:

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"]
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/UnitTests/MonadicContextualClassificationRandomPipelinesUnitTests.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[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 used for illustrative purposes.

In[12]:=
```
SeedRandom[212]
dsData = RandomInteger[{0, 1000}, {100}];
dsData = Dataset[Transpose[{dsData, Mod[dsData, 3], Last@*IntegerDigits /@ dsData, ToString[Mod[#, 3]] & /@ dsData}]];
dsData = Dataset[dsData[All, AssociationThread[{"number", "feature1", "feature2", "label"}, #] &]];
Dimensions[dsData]
```

Out[16]=
```
{100, 4}
```

Here is a sample of the data:

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

Out[17]=

number	feature1	feature2	label
199	1	9	1
288	0	8	0
96	0	6	0
990	0	0	0
705	0	5	0
565	1	5	1

Here is a summary of the data:

In[18]:=
```
ClConUnit[dsData] ⟹ ClConSummarizeData;
```

» summaries: {Anonymous → {

1 number		2 feature1		3 feature2	
Min	20	1st Qu	0	Min	0
1st Qu	278	Min	0	1st Qu	2
Mean	531.17	Mean	0.92	Mean	4.27
Median	531.5	Median	1	Median	4.5
3rd Qu	801.5	3rd Qu	2	3rd Qu	6
Max	998	Max	2	Max	9

4 label

0	40
2	32
1	28

}}

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

In[19]:=
```
mlrData = ClConToNormalClassifierData[dsData];
ClConUnit[mlrData] ⟹ ClConSummarizeData;
```

» summaries: {Anonymous → {

1 column 1		2 column 2		3 column 3	
Min	20	1st Qu	0	Min	0
1st Qu	278	Min	0	1st Qu	2
Mean	531.17	Mean	0.92	Mean	4.27
Median	531.5	Median	1	Median	4.5
3rd Qu	801.5	3rd Qu	2	3rd Qu	6
Max	998	Max	2	Max	9

} → {

1 column 1

0 40
2 32
1 28

}}

Finally, we make the array version of the dataset:

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

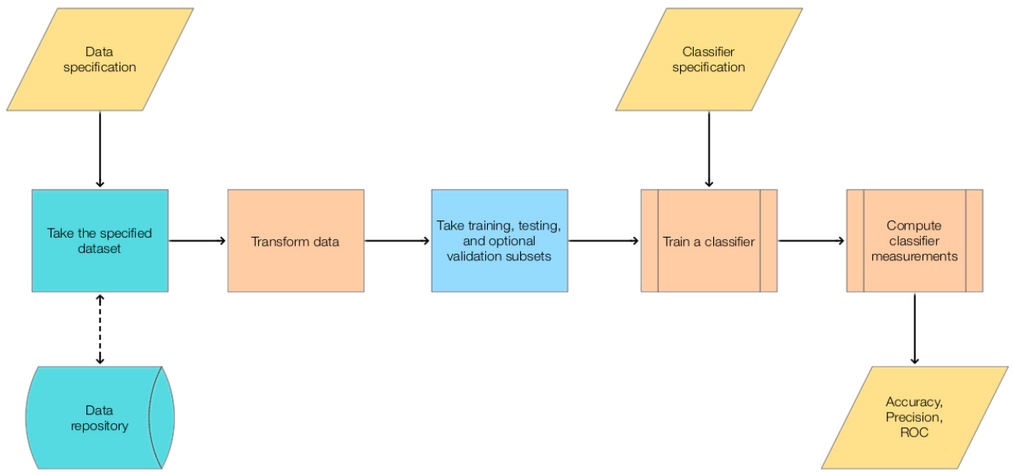
Design considerations

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.

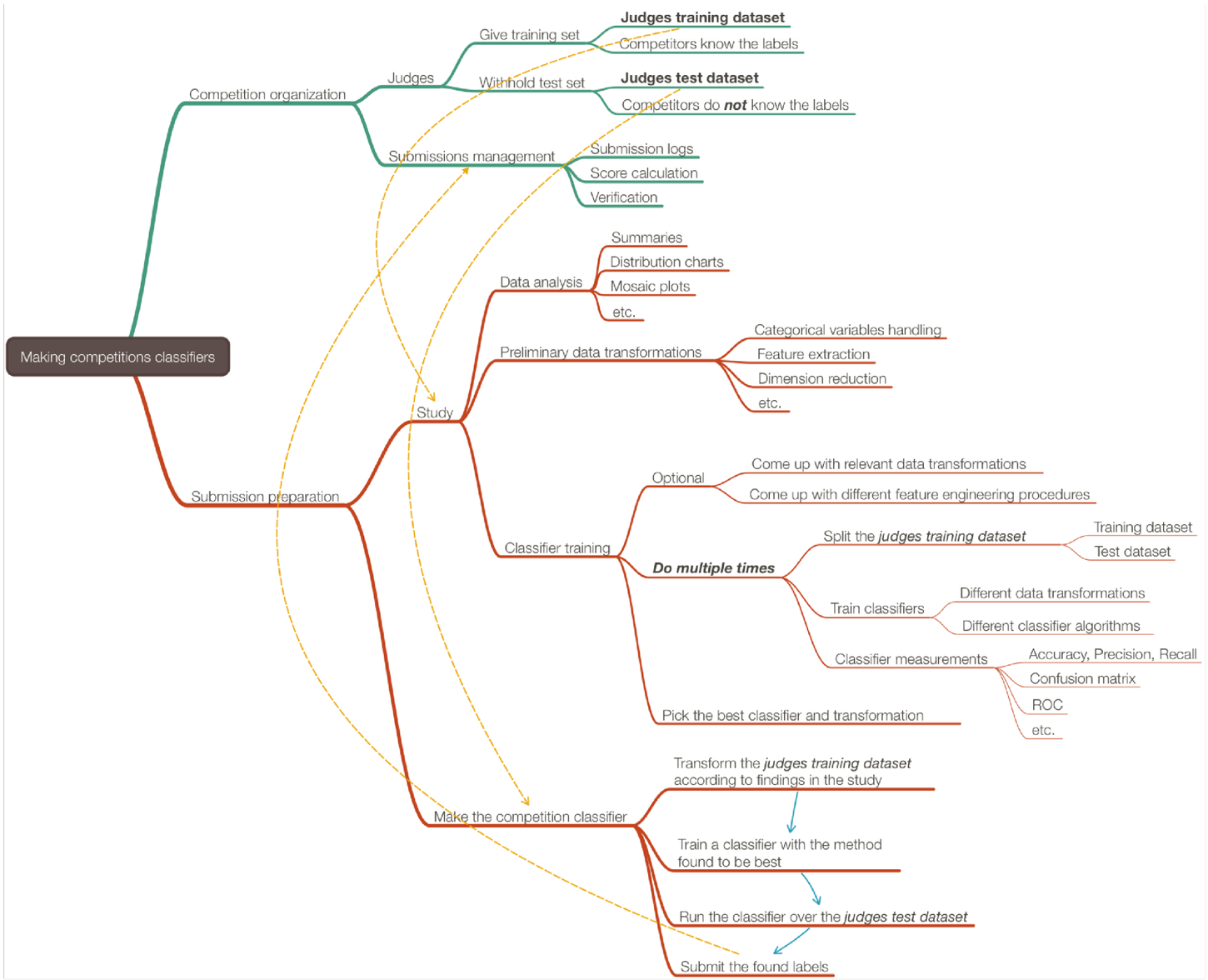
The following diagram shows the steps.

Out[23]=



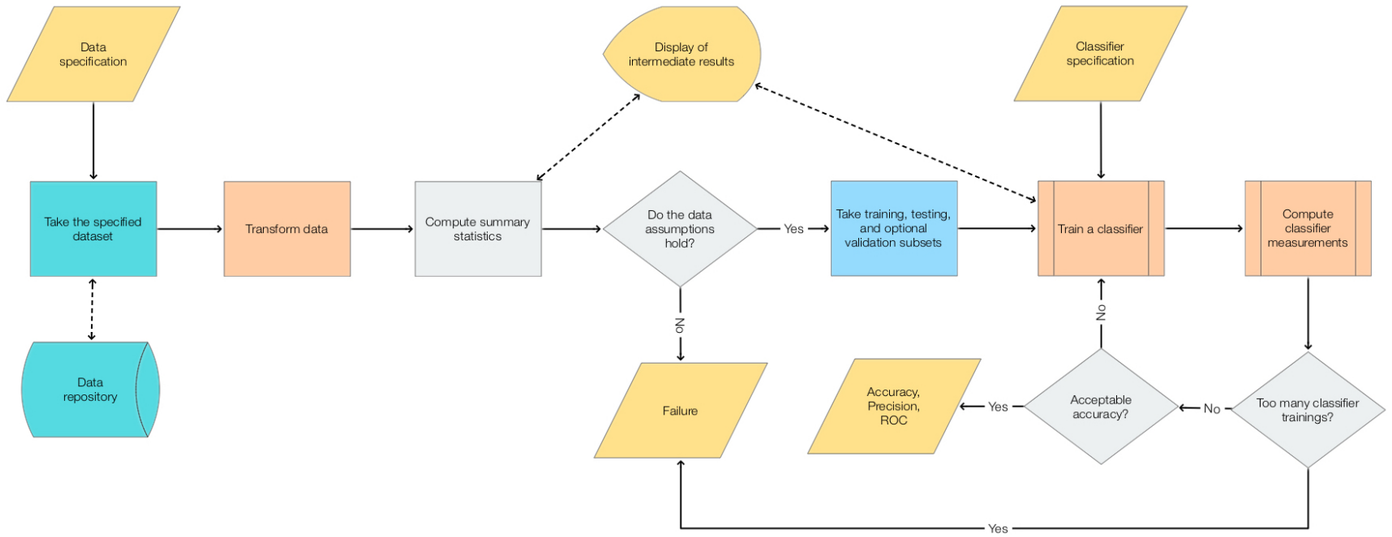
Very often the workflow above is too simple in real situations. Often when making “real world” classifiers we have to experiment with different transformations, different classifier algorithms, and parameters for both transformations and classifiers. Examine the following mind-map that outlines the activities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

Out[25]=



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[28]=



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:

$$\texttt{ClCon}[\_] \xRightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_1 \xRightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_2 \xRightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} \cdots \xRightarrow[\texttt{ClConBind}[\texttt{ClCon}[\_],f\_]]{} f_k \tag{1}$$

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

$$\texttt{ClCon}[\texttt{pval\_}, \texttt{context\_}] \xRightarrow[\texttt{ClConBind}[m\_,f\_]]{} \cdots \left(\begin{cases} f_i[\$\texttt{ClConFailure}] & m \equiv \$\texttt{ClConFailure} \\ f_i[\texttt{x\_}, \texttt{c\_Association}] & m \; \textit{is} \; \texttt{ClCon}[\texttt{x\_}, \texttt{c\_Association}] \\ \$\texttt{ClConFailure} & \text{otherwise} \end{cases} \right) \xRightarrow[\texttt{ClConBind}[m\_,f\_]]{} \cdots \tag{2}$$

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.

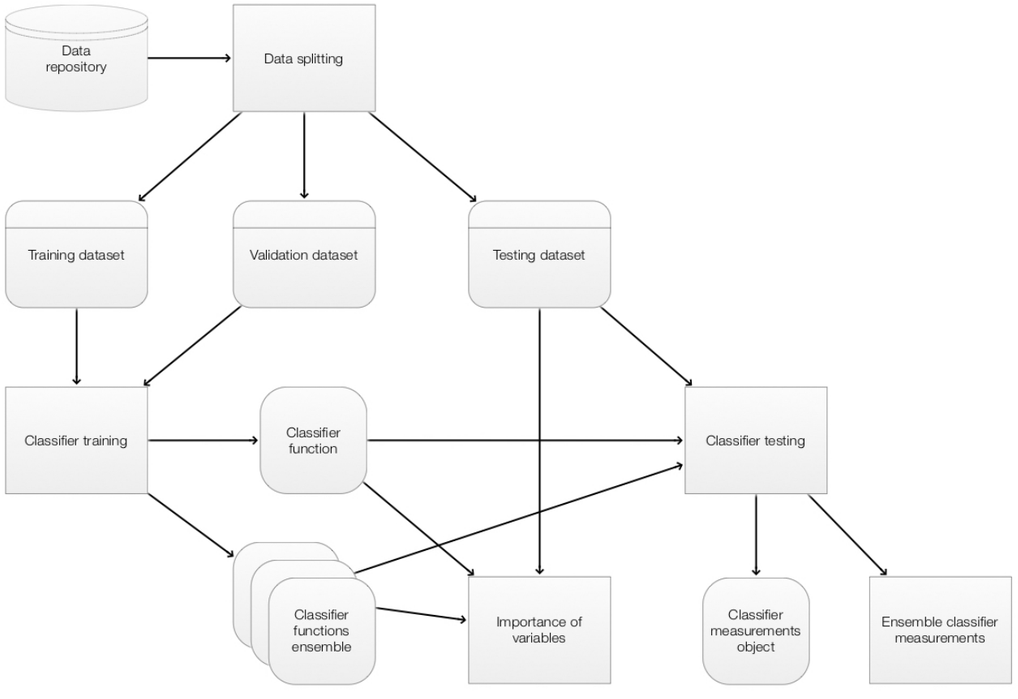
This means that some monad operations would not just change the pipeline value but they will also change the pipeline context.

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[31]=



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].

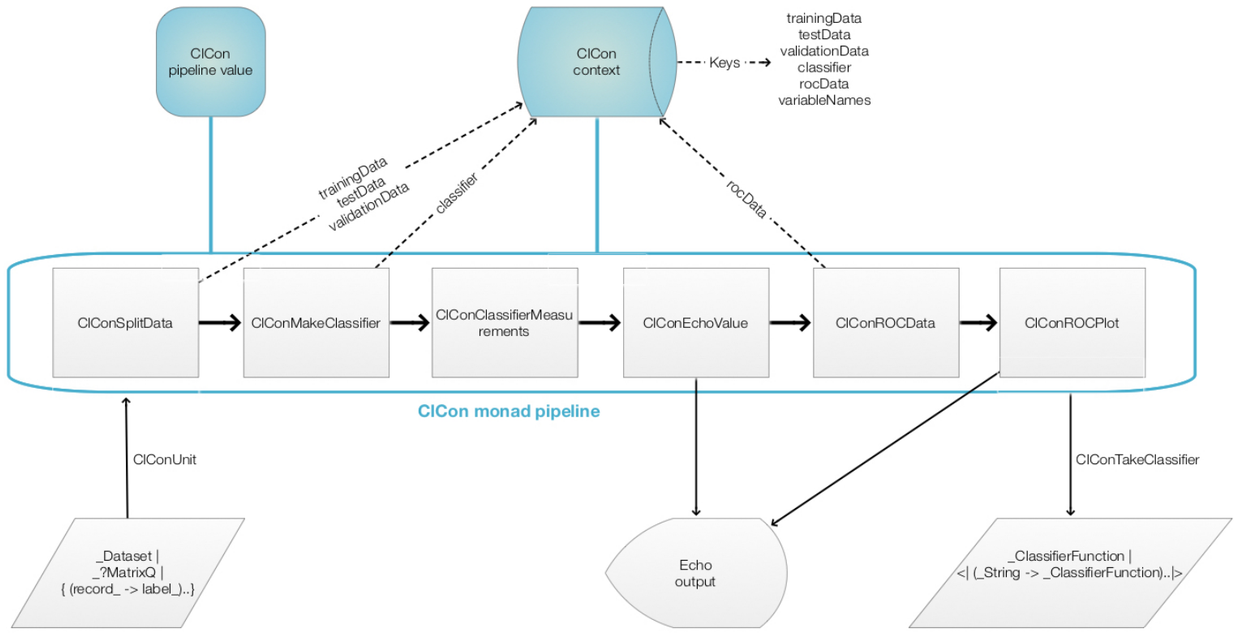
ClCon overview

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.)

Here is a corresponding diagram of a generic computation with the ClCon monad:

Out[34]=



Remark: It is a good idea to compare the diagram with formulas (1) and (2).

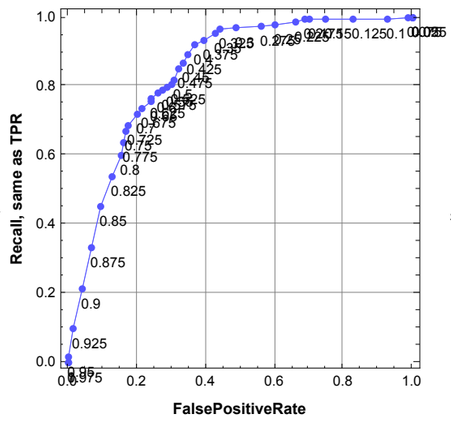
Let us examine a concrete ClCon pipeline that corresponds to the diagram above. In the following table each pipeline operation is combined together with a short explanation and the context keys after its execution.

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 classifer	{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	{}

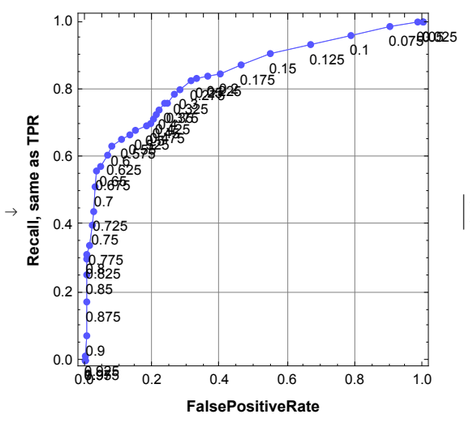
Here is the output of the pipeline:

» value: ⟨| Accuracy → 0.770992 |⟩

» ROC plot(s): ⟨| died →



, survived →



|⟩

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 the pipeline value and context.

Out[•]=

#	name	echoes result	puts in context	uses from context	uses pipeline value
1	*operations*				
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	ClConROCData	no	{rocData}	{classifier, testData}	no
12	ClConROCListLinePlot	yes	none	{rocData}	no
13	ClConROCListLinePlot	yes	{rocData}	{classifier, testData}	no
14	ClConROCPlot	yes	none	{rocData}	no
15	ClConROCPlot	yes	{rocData}	{classifier, testData}	no
16	ClConSplitData	no	{trainingData, testData}	none	yes
17	ClConSuggestROCThresholds	yes	none	{rocData}	no
18	ClConSuggestROCThresholds	yes	{rocData}	{classifier, testData}	no
19	ClConSummarizeData	yes	none	{trainingData, testData, validationData}	tries first
20	ClConSummarizeDataLongForm	yes	none	{trainingData, testData, validationData}	tries first
21	*setters*				
22	ClConSetClassifier	no	classifier	none	no
23	ClConSetTestData	no	testData	none	no
24	ClConSetTrainingData	no	trainingData	none	no
25	ClConSetValidationData	no	validationData	none	no
26	ClConSetVariableNames	no	variableNames	none	no
27	*takers*				
28	ClConTakeClassifier	no	none	classifier	no
29	ClConTakeData	no	none	data	no
30	ClConTakeROCData	no	none	rocData	no
31	ClConTakeTestData	no	none	testData	no
32	ClConTakeTrainingData	no	none	trainingData	no
33	ClConTakeValidationData	no	none	validationData	no
34	ClConTakeVariableNames	no	none	variableNames	no

Several functions that use ROC data have two rows in the table because they calculate the needed ROC data if it is not available in the monad context.

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	ClConEcho	echoes argument(s); a monad version of Echo
7	ClConEchoContext	echoes the context
8	ClConEchoFunctionContext	echoes the result of a function applied to the context
9	ClConEchoFunctionValue	echoes the result of a function applied to the pipeline value
10	ClConEchoValue	echoes the pipeline value
11	ClConFail	gives the monad failure symbol
12	ClConIfElse	choses between two functions based on condition
13	ClConIterate	general interation function
14	ClConModifyContext	modifies the context with the argument function
15	ClConModule	allows faster pipeline function specifications
16	ClConOption	ignores a result if it is failure
17	ClConPutContext	replaces the context with the argument
18	ClConRetrieveFromContext	using a key retrieves into the pipleine a value from the context
19	ClConSucceed	gives a success element of the form ClCon[___]
20	ClConTakeContext	takes the context
21	ClConTakeValue	takes the pipeline value
22	ClConUnit	lifts to the monad
23	ClConUnitQ	gives True if monad unit
24	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[36]:=
```
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[37]:=
```
ClConUnit[dsData] ⟹ ClConSummarizeData;
```

» summaries: {Anonymous → {
1 number		2 feature1		3 feature2	
Min	20	1st Qu	0	Min	0
1st Qu	278	Min	0	1st Qu	2
Mean	531.17	Mean	0.92	Mean	4.27
Median	531.5	Median	1	Median	4.5
3rd Qu	801.5	3rd Qu	2	3rd Qu	6
Max	998	Max	2	Max	9

4 label

0	40
2	32
1	28

}}

In[38]:=
```
ClConUnit[] ⟹ ClConSetTrainingData[dsData] ⟹ ClConSummarizeData;
```

» summaries: {trainingData → {
1 number		2 feature1		3 feature2	
Min	20	1st Qu	0	Min	0
1st Qu	278	Min	0	1st Qu	2
Mean	531.17	Mean	0.92	Mean	4.27
Median	531.5	Median	1	Median	4.5
3rd Qu	801.5	3rd Qu	2	3rd Qu	6
Max	998	Max	2	Max	9

4 label

0	40
2	32
1	28

}}

(See the sub-section “Setters and Takers” for more details of setting and taking values in ClCon contexts.)

Currently the monad can deal with data in the following forms:

- datasets,
- matrices,
- lists of example→label rules.

The ClCon monad also has the non-monadic function ClConToNormalClassifierData which can be used to convert datasets and matrices to lists of example→label rules. Here is an example:

In[39]:=
```
Short[ClConToNormalClassifierData[dsData], 3]
```

Out[39]//Short=
```
{{639, 0, 9} → 0, {121, 1, 1} → 1, {309, 0, 9} → 0, {648, 0, 8} → 0, {995, 2, 5} → 2, {127, 1, 7} → 1, {908, 2, 8} → 2, {564, 0, 4} → 0, {380, 2, 0} → 2, {860, 2, 0} → 2,
  <<80>>, {464, 2, 4} → 2, {449, 2, 9} → 2, {522, 0, 2} → 0, {288, 0, 8} → 0, {51, 0, 1} → 0, {108, 0, 8} → 0, {76, 1, 6} → 1, {706, 1, 6} → 1, {765, 0, 5} → 0, {195, 0, 5} → 0}
```

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. If the value of option Method is ”LabelsProportional”, then the splitting is done in correspondence of the class labels tallies.

Data splitting demonstration examples follow.

Here are the dimensions of the dataset dsData:

In[40]:=
```
Dimensions[dsData]
```

Out[40]=
```
{100, 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[42]:=
```
val = ClConUnit[dsData] ⟹ ClConSplitData[0.7] ⟹ ClConTakeValue;
Map[Dimensions, val]
Total[First /@ %]
```

Out[43]=
```
⟨|trainingData → {69, 4}, testData → {31, 4}|⟩
```

Out[44]=
```
100
```

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[46]:= val = ClConUnit[dsData] ⟹ ClConSplitData[0.7, 0.1] ⟹ ClConTakeValue;
         Map[Dimensions, val]
         Total[First /@ %]
Out[47]= ⟨| trainingData → {61, 4}, testData → {31, 4}, validationData → {8, 4} |⟩
Out[48]= 100
```

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

Here is information about the obtained classifier:

```
In[51]:= ClassifierInformation[cf, "TrainingTime"]
Out[51]= 3.66136 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[52]:= cf =
           ClConUnit[dsTitanic] ⟹
            ClConSplitData[0.75] ⟹
            ClConMakeClassifier[Method → {"RandomForest", "TreeNumber" → 400}] ⟹
            ClConTakeClassifier;

In[53]:= ClassifierInformation[cf, "TreeNumber"]
Out[53]= 400
```

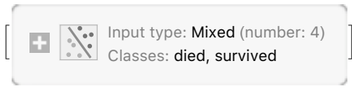
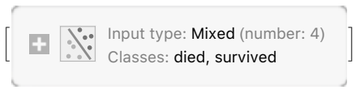
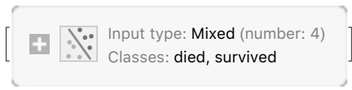
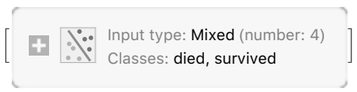
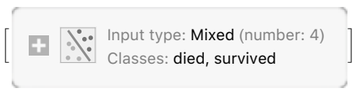
Classifier ensemble training

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[54]:= 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[55]:=
```
ensemble
```

Out[55]=
```
⟨|LogisticRegression[1,0.9] → ClassifierFunction[  ],
 LogisticRegression[2,0.9] → ClassifierFunction[  ], LogisticRegression[3,0.9] → ClassifierFunction[  ],
 NearestNeighbors[1,0.9] → ClassifierFunction[  ], NearestNeighbors[2,0.9] → ClassifierFunction[  ]|⟩
```


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

In[56]:=
```
ClassifierInformation[#, "TrainingTime"] & /@ ensemble
```

Out[56]=
```
⟨|LogisticRegression[1,0.9] → 5.30167 s , LogisticRegression[2,0.9] → 3.50852 s ,
 LogisticRegression[3,0.9] → 3.55011 s , NearestNeighbors[1,0.9] → 1.8541 s , NearestNeighbors[2,0.9] → 1.83438 s |⟩
```

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

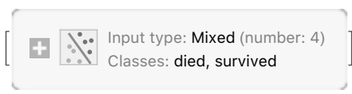
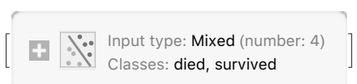
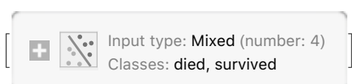
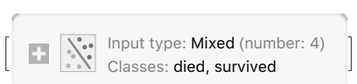
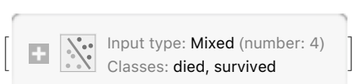
```
<|"method" → "LogisticRegression", "sampleFraction" → 0.9, "numberOfClassifiers" → 3, "samplingFunction" → RandomChoice|>
```

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[57]:=
```
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[58]:=
```
ensemble2
```

Out[58]=
```
⟨|LogisticRegression[1,0.9] → ClassifierFunction[  ],
 LogisticRegression[2,0.9] → ClassifierFunction[  ], LogisticRegression[3,0.9] → ClassifierFunction[  ],
 NearestNeighbors[1,0.9] → ClassifierFunction[  ], NearestNeighbors[2,0.9] → ClassifierFunction[  ]|⟩
```


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[59]:=
```
p =
  ClConUnit[dsTitanic] ⟹
   ClConSplitData[0.75] ⟹
   ClConMakeClassifier["DecisionTree"];
```

Here is how we compute selected classifier measures:

In[60]:=
```
p⟹
 ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall", "FalsePositiveRate"}]⟹
 ClConTakeValue
```

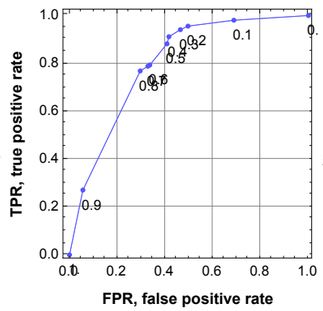
Out[60]=
```
⟨|Accuracy → 0.768293, Precision → ⟨|died → 0.782222, survived → 0.737864|⟩, Recall → ⟨|died → 0.866995, survived → 0.608|⟩, FalsePositiveRate → ⟨|died → 0.392, survived → 0.133005|⟩|⟩
```

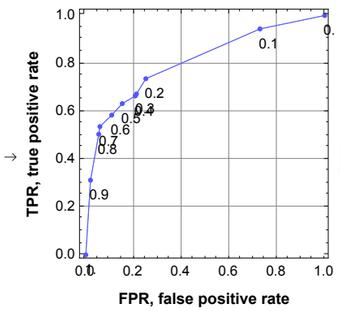
Here we show the confusion matrix plot:

In[61]:=
```
p⟹ClConClassifierMeasurements["ConfusionMatrixPlot"];
```

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

In[62]:=
```
p⟹
  ClConROCPlot["FPR", "TPR", "ROCRange" → Range[0, 1, 0.1], ImageSize → 200];
```

» ROC plot(s): ⟨|died → 

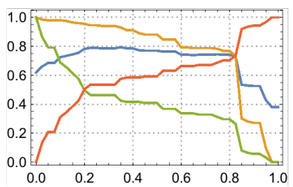
, survived → 

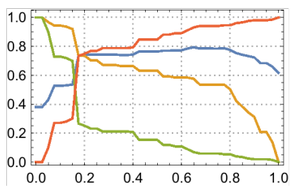
|⟩

Remark: ClCon uses the package ROCFunctions.m, [AAp5], which implements all functions defined in [Wk2].

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

In[63]:=
```
p⟹ClConROCListLinePlot[{"ACC", "TPR", "FPR", "SPC"}];
```

» ROC line plot(s): ⟨|died → 

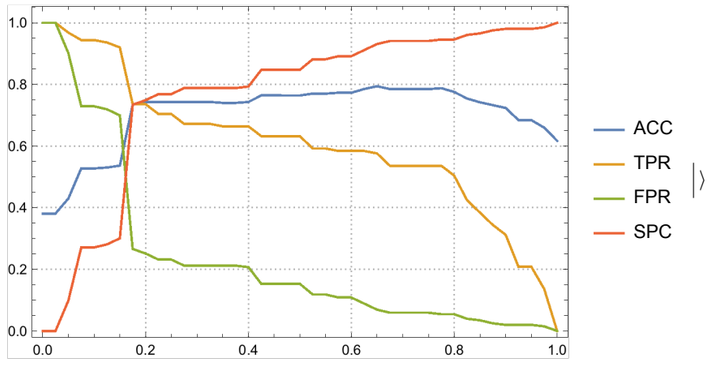
, survived → 

|⟩

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”→False the automatic echoing of plots can be prevented. With the option “TargetClasses” we can focus on specific class labels.

```
In[64]:= p⟹
          ClConROCListLinePlot[{"ACC", "TPR", "FPR", "SPC"}, "Echo" → False, "TargetClasses" → "survived", ImageSize → Medium]⟹
          ClConEchoValue;
```

» value: ⟨| survived →



|⟩

Variable importance finding

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

```
In[65]:= p⟹
          ClConAccuracyByVariableShuffling⟹
          ClConTakeValue
Out[65]= ⟨| None → 0.768293, id → 0.594512, passengerClass → 0.676829, passengerAge → 0.734756, passengerSex → 0.588415 |⟩
```

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.

For example:

```
In[66]:= p⟹ClConTakeClassifier
Out[66]= ClassifierFunction[ Input type: Mixed (number: 4)
                             Classes: died, survived
                             Method: DecisionTree
                             Number of training examples: 981 ]
```

```
In[67]:= Short[Normal[p⟹ClConTakeTrainingData]]
Out[67]//Short= {⟨| id → 215, passengerClass → 1st, passengerAge → 20, passengerSex → female, passengerSurvival → survived |⟩, <<979>>, ⟨| <<1>> |⟩}

In[68]:= Short[Normal[p⟹ClConTakeTestData]]
Out[68]//Short= {⟨| id → 459, passengerClass → 2nd, passengerAge → 20, passengerSex → female, passengerSurvival → survived |⟩, <<326>>, ⟨| id → 1235, <<4>> |⟩}

In[69]:= p⟹ClConTakeVariableNames
Out[69]= {id, passengerClass, passengerAge, passengerSex, passengerSurvival}
```

If other values are put in the context they can be obtained through the (generic) function `ClConTakeContext`, [AAp1]:

In[70]:=
```
p = ClConUnit[RandomReal[1, {2, 2}]] ⟹ ClConAddToContext["data"];
```

In[71]:=
```
(p ⟹ ClConTakeContext)["data"]
```

Out[71]=
```
{{0.241557, 0.869559}, {0.325535, 0.816288}}
```

Another generic function is from [AAp1] is `ClConTakeValue` (used many times above.)

Example use cases

Classification with MNIST data

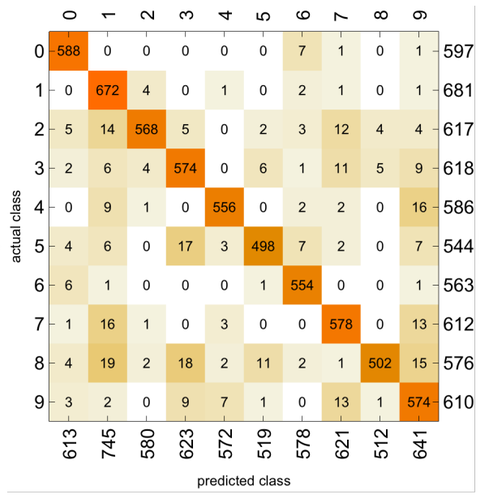
Here we show an example of using `ClCon` with the reasonably large dataset of images MNIST, [YL1].

In[72]:=
```
mnistData = ExampleData[{"MachineLearning", "MNIST"}, "Data"];
```

In[73]:=
```
SeedRandom[3423]
p =
  ClConUnit[RandomSample[mnistData, 20 000]] ⟹
   ClConSplitData[0.7] ⟹
   ClConSummarizeData ⟹
   ClConMakeClassifier["NearestNeighbors"] ⟹
   ClConClassifierMeasurements[{"Accuracy", "ConfusionMatrixPlot"}] ⟹
   ClConEchoValue;
```

» summaries: ⟨|trainingData → {1 column 1 | 1 | 1 | 1 | 1 | 1 | 1 | (Other) 13 990} → {1 column 1 | Min 0 | 1st Qu 2 | Median 4 | Mean 4.42948 | 3rd Qu 7 | Max 9}, testData → {1 column 1 | 1 | 1 | 1 | 1 | 1 | 1 | (Other) 5998} → {1 column 1 | Min 0 | 1st Qu 2 | Median 4 | Mean 4.42921 | 3rd Qu 7 | Max 9}|⟩

» value: ⟨|Accuracy → 0.943371, ConfusionMatrixPlot →

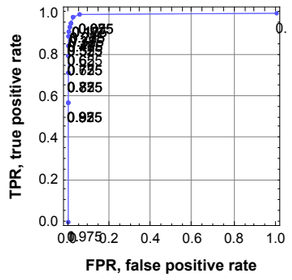


|⟩

Here we plot the ROC curve for a specified digit:

In[75]:=
```
p⟹ClConROCPlot["TargetClasses" → 5];
```

» ROC plot(s): ⟨|5 →



|⟩

Conditional continuation

In this sub-section we show how the computations in a `ClCon` pipeline can be stopped or continued based on a certain condition.

The pipeline below makes a simple classifier (“LogisticRegression”) for the WineQuality data, and if the recall for the important label (“high”) is not large enough makes a more complicated classifier (“RandomForest”.)

In[362]:=
```
SeedRandom[267]
res =
  ClConUnit[dsWineQuality[All, Join[#, <|"wineQuality" → If[#wineQuality ≥ 7, "high", "low"]|>] &]] ⟹
   ClConSplitData[0.75, 0.2] ⟹
   ClConSummarizeData ⟹
   ClConMakeClassifier[Method → "LogisticRegression"] ⟹
   ClConROCPlot["FPR", "TPR", "ROCPointCallouts" → False] ⟹
   ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall", "FalsePositiveRate"}] ⟹
   ClConEchoValue ⟹
   ClConIfElse[#["Recall", "high"] > 0.70 &,
    ClConEcho["Good recall for \"high\"!", "Success:"],
    ClConUnit[##] ⟹
      ClConEcho["Recall for \"high\" not good enough... making a large random forest.", "Info:"] ⟹
      ClConMakeClassifier[Method → {"RandomForest", "TreeNumber" → 400}] ⟹
      ClConROCPlot["FPR", "TPR", "ROCPointCallouts" → False] ⟹
      ClConClassifierMeasurements[{"Accuracy", "Precision", "Recall", "FalsePositiveRate"}] ⟹
      ClConEchoValue &];
```

» summaries: {trainingData →

1 RowID	
1	13
10	13
100	13
1000	13
1001	13
1002	13
(Other)	38 106

,

2 Variable	
alcohol	2938
chlorides	2938
density	2938
fixedAcidity	2938
freeSulfurDioxide	2938
id	2938
(Other)	20 556

,

3 Value	
low	2302
high	636
0.28	340
0.3	314
0.32	305
0.26	284
(Other)	34 003

},

testData →

1 RowID	
1	13
10	13
100	13
1000	13
1001	13
1002	13
(Other)	15 840

,

2 Variable	
alcohol	1225
chlorides	1225
density	1225
fixedAcidity	1225
freeSulfurDioxide	1225
id	1225
(Other)	8568

,

3 Value	
low	960
high	265
0.3	140
0.28	133
0.27	117
0.32	117
(Other)	14 186

}, validationData →

1 RowID	
1	13
10	13
100	13
101	13
102	13
103	13
(Other)	9475

,

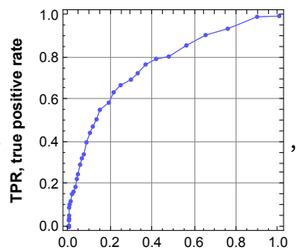
2 Variable	
alcohol	735
chlorides	735
density	735
fixedAcidity	735
freeSulfurDioxide	735
id	735
(Other)	5143

,

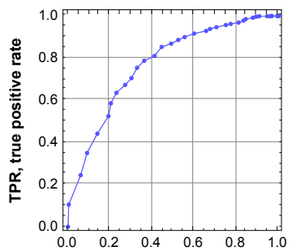
3 Value	
low	576
high	159
0.28	85
0.3	82
0.34	75
0.32	71
(Other)	8505

}}

» ROC plot(s): ⟨|high →



, low →

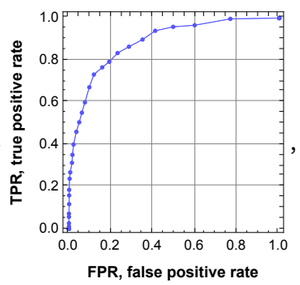


|⟩

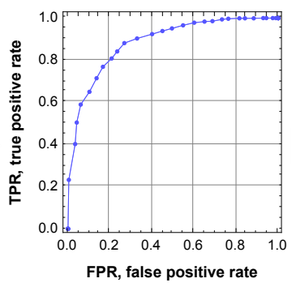
» value: ⟨|Accuracy → 0.805714, Precision → ⟨|high → 0.604651, low → 0.82938|⟩, Recall → ⟨|high → 0.29434, low → 0.946875|⟩, FalsePositiveRate → ⟨|high → 0.053125, low → 0.70566|⟩|⟩

» Info: Recall for "high" not good enough... making a large random forest.

» ROC plot(s): ⟨|high →



, low →



|⟩

» value: ⟨|Accuracy → 0.854694, Precision → ⟨|high → 0.784314, low → 0.864739|⟩, Recall → ⟨|high → 0.45283, low → 0.965625|⟩, FalsePositiveRate → ⟨|high → 0.034375, low → 0.54717|⟩|⟩

We can see that the recall with the more complicated is classifier is higher. Also the ROC plots of the second classifier are visibly closer to the ideal one. Still, the recall is not good enough so we have to find threshold better that the default one. (See the next sub-section.)

Classification with custom thresholds

In this sub-section we use the monad from the previous sub-section.

Here we compute classification measures using the threshold 0.3 for the important class label (“high”):

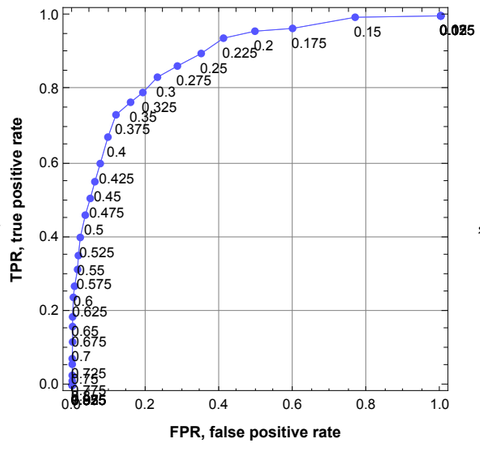
In[361]:=
```
res⟹
ClConClassifierMeasurementsByThreshold[{"Accuracy", "Precision", "Recall", "FalsePositiveRate"}, "high" → 0.3]⟹
ClConTakeValue
```

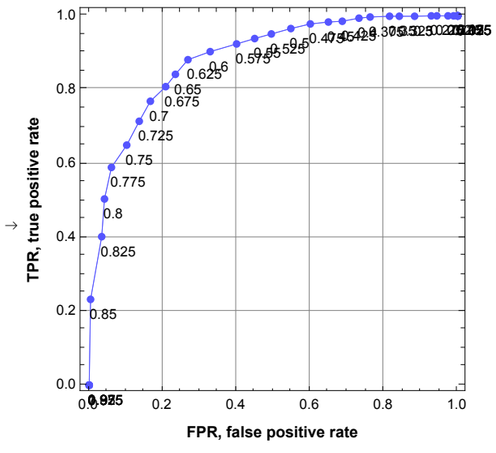
Out[361]= ⟨|Accuracy → 0.768163, Precision → ⟨|high → 0.479212, low → 0.940104|⟩, Recall → ⟨|high → 0.826415, low → 0.752083|⟩, FalsePositiveRate → ⟨|high → 0.247917, low → 0.173585|⟩|⟩

We can see that the recall for “high” is fairly high and the rest of the measures have satisfactory values. (The accuracy did not drop that much, and false positive rate is not that large.)

Here we compute suggestions for the best thresholds:

In[369]:=
```
res(* start with a previous monad *)⟹
  ClConROCPlot[ImageSize → 300](* make ROC plots *)⟹
  ClConSuggestROCThresholds[3](* find best 3 thresholds per class label *)⟹
  ClConEchoValue(* echo the result *);
```

» ROC plot(s): ⟨|high → 

, low → 

|⟩

» value: ⟨|high → {0.35, 0.325, 0.3}, low → {0.65, 0.675, 0.7}|⟩

The suggestions are the ROC points that closest to the point {0, 1} (which corresponds to the ideal classifier.)

Here is a way to use threshold suggestions within the monad pipeline:

In[364]:=
```
res⟹
  ClConSuggestROCThresholds⟹
  ClConEchoValue⟹
  (ClConUnit[##]⟹ClConClassifierMeasurementsByThreshold[{"Accuracy", "Precision", "Recall"}, "high" → First[#1["high"]]] &)⟹
  ClConEchoValue;
```

» value: ⟨|high → {0.35}, low → {0.65}|⟩

» value: ⟨|Accuracy → 0.825306, Precision → ⟨|high → 0.571831, low → 0.928736|⟩, Recall → ⟨|high → 0.766038, low → 0.841667|⟩|⟩

Unit tests

The development of ClCon was done with two types of unit tests: (1) directly specified, [AAp11], and (2) through randomly generated pipelines, [AAp12].

Both unit test packages should be further extended.

Directly specified tests

Here we run the unit tests file “MonadicContextualClassification-Unit-Tests.wlt”, [AAp11]:

```
In[81]:= AbsoluteTiming[
          testObject = TestReport["~/MathematicaForPrediction/UnitTests/MonadicContextualClassification-Unit-Tests.wlt"]
         ]
```

```
Out[81]= {64.1562, TestReportObject[ Title: Test Report: MonadicContextualClassification-Unit-Tests.wlt
                                     Success rate: 100%  Tests run: 27 ]}
```

The natural language derived test ID’s should give a fairly good idea of the functionalities covered in [AAp11].

```
In[82]:= Values[Map[#["TestID"] &, testObject["TestResults"]]]
```

```
Out[82]= {LoadPackage, EvenOddDataset, EvenOddDataMLRules, DataToContext-no-[], DataToContext-with-[], ClassifierMaking-with-Dataset-1,
          ClassifierMaking-with-MLRules-1, AccuracyByVariableShuffling-1, ROCData-1, ClassifierEnsemble-different-methods-1, ClassifierEnsemble-different-methods-2-cont,
          ClassifierEnsemble-different-methods-3-cont, ClassifierEnsemble-one-method-1, ClassifierEnsemble-one-method-2, ClassifierEnsemble-one-method-3-cont,
          ClassifierEnsemble-one-method-4-cont, AssignVariableNames-1, AssignVariableNames-2, AssignVariableNames-3, SplitData-1, Set-and-take-training-data,
          Set-and-take-test-data, Set-and-take-validation-data, Partial-data-summaries-1, Assign-variable-names-1, Split-data-100-pct, MakeClassifier-with-empty-unit-1}
```

Random pipelines tests

Since the monad `ClCon` is a DSL it is natural to test it with a large number of randomly generated “sentences” of that DSL. For the `ClCon` DSL the sentences are `ClCon` pipelines. The package “MonadicContextualClassificationRandomPipelinesUnitTests.m”, [AAp12], has functions for generation of `ClCon` random pipelines and running them as verification tests. A short example follows.

Generate pipelines:

```
In[•]:= SeedRandom[234]
        pipelines = MakeClConRandomPipelines[300];
        Length[pipelines]
```

```
Out[•]= 300
```

Here is sample of the generated pipelines:

Out[86]=

#	pipeline
1	`ClCon[ds, ⟨| |⟩] ⟹ ClConSetTrainingData[ds] ⟹ ClConSetTestData[mlrData] ⟹ ClConSetValidationData[ds] ⟹ ClConSetValidationData[ds] ⟹ ClConMakeClassifier[Method → {RandomForest, TreeNumber → 200}] ⟹ ClConClassifierMeasurements[{Accuracy, Precision, Recall}] ⟹ ClConTakeValue`
2	`ClCon[None, ⟨| |⟩] ⟹ ClConSetTestData[mlrData] ⟹ ClConMakeClassifier[{NearestNeighbors, 0.8, 5, RandomChoice}] ⟹ ClConROCData ⟹ ClConTakeValue`
3	`ClCon[ds, ⟨| |⟩] ⟹ ClConSetValidationData[mlrData] ⟹ ClConSplitData[1] ⟹ ClConMakeClassifier[NearestNeighbors] ⟹ ClConClassifierMeasurements[{Accuracy, Precision, Recall}] ⟹ ClConTakeValue`
4	`ClCon[ds, ⟨| |⟩] ⟹ ClConSplitData[0.7] ⟹ ClConSummarizeData ⟹ ClConTakeValue`
5	`ClCon[mlrData, ⟨| |⟩] ⟹ ClConSplitData[0.7] ⟹ ClConSetTestData[ds] ⟹ ClConSetValidationData[ds] ⟹ ClConMakeClassifier[LogisticRegression] ⟹ ClConClassifierMeasurements[{Accuracy, Precision, Recall}] ⟹ ClConTakeValue`
6	`ClCon[ds, ⟨| |⟩] ⟹ ClConSetTestData[mlrData] ⟹ ClConSplitData[0.7, 0.2] ⟹ ClConSetTrainingData[ds] ⟹ ClConSetValidationData[ds] ⟹ ClConMakeClassifier[NearestNeighbors] ⟹ ClConClassifierMeasurements[{Accuracy, Precision, Recall}] ⟹ ClConTakeValue`

Here we run the pipelines as unit tests:

```
In[•]:= AbsoluteTiming[
         res = TestRunClConPipelines[pipelines, "Echo" → True];
        ]
Out[•]= {350.083, Null}
```

From the test report results we see that a dozen tests failed with messages, all of the rest passed.

```
In[•]:= rpTRObj = TestReport[res]
```

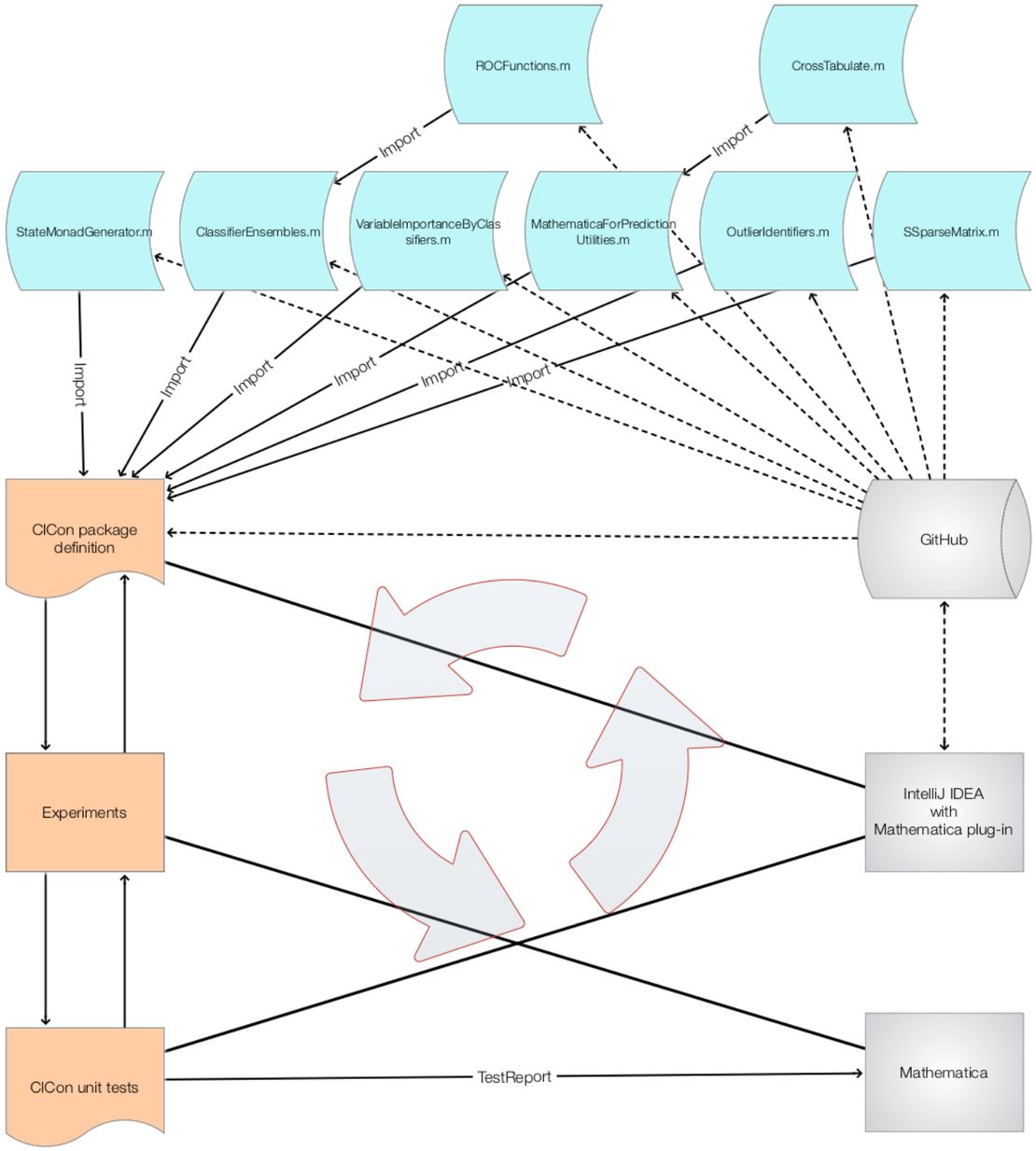
```
Out[•]= TestReportObject[
          Title: Automatic
          Success rate: 96%    Tests run: 300
          Succeeded: 288
          Failed: 12
          Failed with wrong results: 0
          Failed with messages: 12
          Failed with errors: 0
          Data not in notebook; Store now »
        ]
```

(The message failures, of course, have to be examined -- some bugs were found in that way. Currently the actual test messages are expected.)

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.

Out[92]=



Some observations and morals follow.

- Making the unit tests [AAp11] made the final stages of implementation much more comfortable.
  - Of course, in retrospect that is obvious.
- Initially MonadicContextualClassification.m was not real a package, just a collection of functions with prefix “ClCon” in the global context. This made some programming design decisions harder and more cumbersome. By making a proper package the development became much easier because of the “peace of mind“ brought by the context feature encapsulation.

- The explanation for this is that the initial versions of MonadicContextualClassification.m were made to illustrate the monad programing described in [AA1] using the package StateMonadCodeGenerator.m .
- The making of random pipeline tests, [AAp12], helped catch a fair amount of inconvenient "features" and bugs.
  - (Both tests sets [AAp11, AAp12] can be made to be more comprehensive.)
- The design of a conversational agent for producing `ClCon` pipelines with natural language commands brought very fruitful viewpoint on the overall functionalities and the determination and limits of the `ClCon` development goals. See [AAp13, AAp14, AAp15].
- “Eat your own dog food”, or in this case: “use `ClCon` functionalities to implement `ClCon` functionalities.”
  - Since we are developing a DSL it is natural to use that DSL for its own advancement.
  - Again, in retrospect that is obvious. Also probably should be seen as a consequence of practicing a certain code refactoring discipline.
  - The reason to list that moral is that often it is somewhat "easier" to implement functionalities thinking locally, ad-hoc, forgetting or not reviewing other, already made implemented functions.

Future plans

Workflow operations

Better outliers finding and manipulation incorporation in `ClCon`. Currently only outlier finding is surfaced in [AAp3]. (The package has internally other related functions.)

In[373]:=
```
ClConUnit[dsTitanic[Select[#passengerSex == "female" &]]] ⟹
 ClConOutlierPosition⟹
 ClConTakeValue
```

Out[373]=
```
{4, 17, 21, 22, 25, 29, 38, 39, 41, 59}
```

Support of dimension reduction application -- quick construction of pipelines that allow the applying different dimension reduction methods.

Conversational agent

Using the packages [AAp13, AAp15] we can generate ClCon pipelines with natural commands. The plan is to develop and document those functionalities further.

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.

[AAp11] Anton Antonov, Monadic contextual classification Mathematica unit tests, (2018), MathematicaVsR at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/UnitTests/MonadicContextualClassification-Unit-Tests.wlt .

[AAp12] Anton Antonov, Monadic contextual classification random pipelines Mathematica unit tests, (2018), MathematicaVsR at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/UnitTests/MonadicContextualClassificationRandomPipelinesUnitTests.m .

ConverationalAgents Packages

[AAp13] Anton Antonov, Classifier workflows grammar in EBNF, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

[AAp14] Anton Antonov, Classifier workflows grammar Mathematica unit tests, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

[AAp15] 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) .

[Wk2] Wikipedia entry, Receiver operating characteristic,
URL: https://en.wikipedia.org/wiki/Receiver_operating_characteristic .

[YL1] Yann LeCun et al., MNIST database site.
URL: http://yann.lecun.com/exdb/mnist/ .

[PS1] Patrick Scheibe, Mathematica (Wolfram Language) support for IntelliJ IDEA, (2013-2018), Mathematica-IntelliJ-Plugin at GitHub.
URL: https://github.com/halirutan/Mathematica-IntelliJ-Plugin .