A monad for Quantile Regression workflows

Version 1.0

Anton Antonov
MathematicaForPrediction at WordPress
MathematicaForPrediction at GitHub
MathematicaVsR at GitHub
June-July 2018

Introduction

In this document we describe the design and implementation of a (software programming) monad for Quantile Regression workflows specification and execution. The design and implementation are done with Mathematica / Wolfram Language (WL).

What is Quantile Regression? : Assume we have the points of a single variable distribution and we want to find a curve that splits the values from that distribution in such a way that 30% of them are above the curve. This is done with Quantile Regression, see [Wk2, CN1, AA2, AA3]. Quantile Regression is a method to estimate the variable relations for all parts of the distribution. (Not just, say, the mean of the relationships found with Least Squares Regression.)

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

The monad is named `QRegMon` and it is based on the State monad package “StateMonadCodeGenerator.m”, [AAp1, AA1] and the Quantile Regression package “QuantileRegression.m”, [AAp4, AA2].

The data for this document is read from WL’s repository or created ad-hoc.

The monadic programming design is used as a Software Design Pattern. The `QRegMon` 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 `QRegMon` monad over time series temperature data:

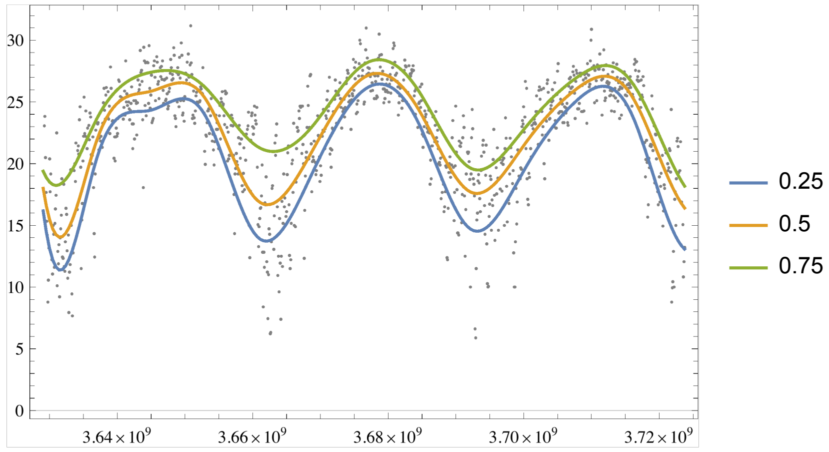
```
QRMonUnit[tsData] ⟹                      lift to the monad
  QRMonEchoDataSummary ⟹                 show the data summary
  QRMonQuantileRegression[12] ⟹          do quantile regression with 12 knots
  QRMonPlot ⟹                            plot data and regression quantiles
  QRMonOutliers ⟹                        find outliers
  QRMonEchoFunctionValue[Short] ⟹        show the found ouliers
  QRMonOutliersPlot                      plot data and outliers
```

» Data summary:

1 column 1	
Min	3 629 059 200
1st Qu	3 652 776 000
Mean	3.67643×10^9
Median	3 676 449 600
3rd Qu	3 700 123 200
Max	3 723 753 600

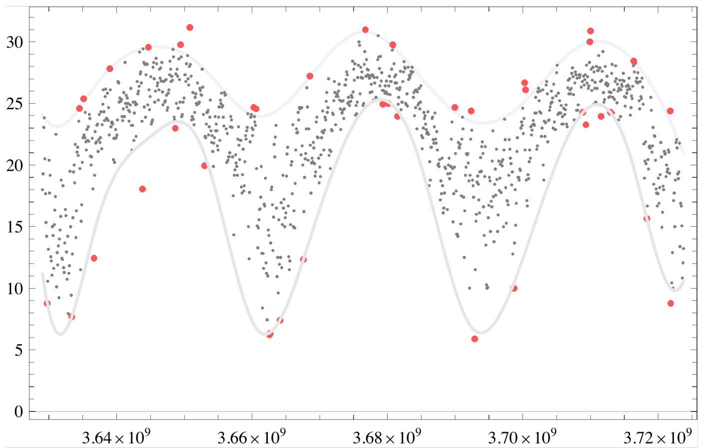
2 column 2	
Min	5.89
1st Qu	19.695
Mean	22.2982
Median	23.33
3rd Qu	25.89
Max	31.17

» Plot:




```
» ⟨|topOutliers → {{3 634 502 400, 24.61}, {3 635 107 200, 25.39}, ≪15≫, {3 716 409 600, 28.44}, {3 721 766 400, 24.39}}, bottomOutliers → {≪1≫}|⟩
```

» Outliers plot:



The table above is produced with the package “MonadicTracing.m”, [AAp2, AA1], and some of the explanations below also utilize that package.

As it was mentioned above the monad QRMon can be seen as a DSL. Because of this the monad pipelines made with QRMon are sometimes called “specifications”.

Remark: With “*regression quantile*” we mean “a curve or function that is computed with Quantile Regression”.

Contents description

The document has the following structure.

- The sections "Package load" and "Data load" obtain the needed code and data.
  - (Needed and put upfront from the "Reproducible research" point of view.)
- The sections "Design consideration" and "Monad design" provide motivation and design decisions rationale.
- The sections "QRMon overview" and "Monad elements" provide technical description of the QRMon monad needed to utilize it.
  - (Using a fair amount of examples.)
- The section "Unit test" describes the tests used in the development of the QRMon monad.

- (The random pipelines unit tests are especially interesting.)
- The section "Future plans" outlines future directions of development.
  - (The most interesting and important one is the "conversational agent" direction.)
- The section "Implementation notes" just says QRMon’s development process and this document follow the ones of the classifications workflows monad ClCon, [AA6].

Remark: One can read only the sections "Introduction", "Design consideration", "Monad design", and "QRMon overview". That set of sections provide a fairly good, programming language agnostic exposition of the substance and novel ideas of this document.

Package load

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

In[7]:=
```
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/MonadicQuantileRegression.m"]
Import["https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/MonadicProgramming/MonadicTracing.m"]
```

Data load

In this section we load data that is used in the rest of the document. The time series data is obtained through WL’s repository.

The data summarization and plots are done through QRMon, which in turn uses the function `RecordsSummary` from the package “MathematicaForPredictionUtilities.m”, [AAp6].

Distribution data

The following data is generated to have heteroscedasticity.

In[9]:=
```
distData = Table[{x, Exp[-x^2] + RandomVariate[NormalDistribution[0, .15 Sqrt[Abs[1.5 - x] / 1.5]]]}, {x, -3, 3, .01}];
Length[distData]
```

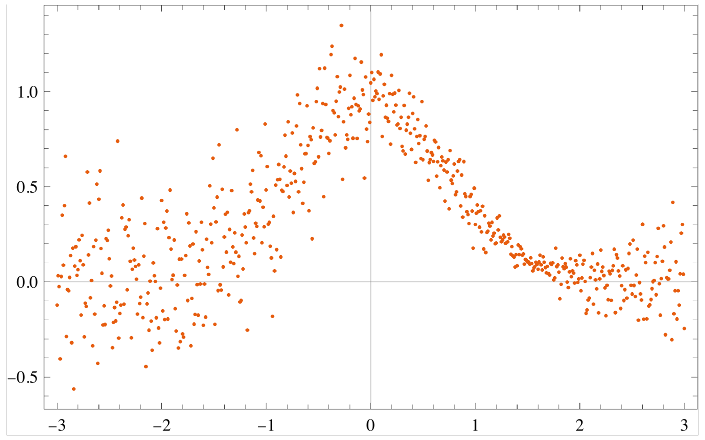
Out[10]= 601

In[11]:=
```
QRMonUnit[distData] ⟹ QRMonEchoDataSummary ⟹ QRMonPlot;
```

» Data summary:

1 column 1		2 column 2	
Min	-3.	Min	-0.562388
1st Qu	-1.5025	1st Qu	0.031954
Median	0.	Median	0.20604
Mean	8.86701×10^{-17}	Mean	0.300542
3rd Qu	1.5025	3rd Qu	0.605181
Max	3.	Max	1.34789

» Plot:



Temperature time series

In[12]:=
```
tsData = WeatherData[{"Orlando", "USA"}, "Temperature", {{2015, 1, 1}, {2018, 1, 1}, "Day"}]
```

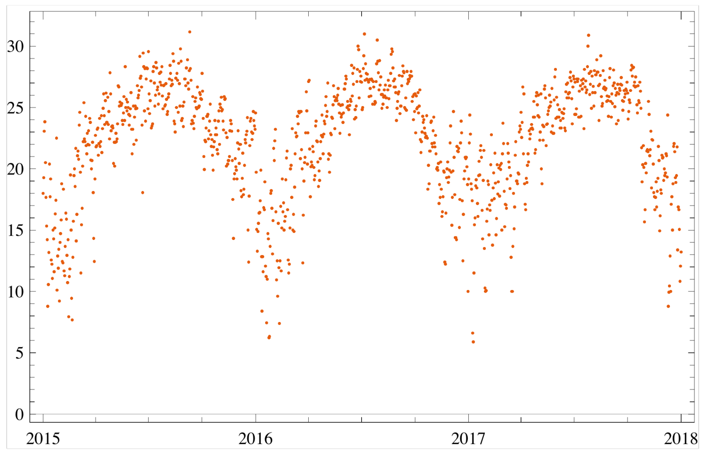
Out[12]=
```
TimeSeries[ Time: 01 Jan 2015 to 01 Jan 2018
            Data points: 1096 ]
```

In[13]:=
```
QRMonUnit[tsData] ⟹ QRMonEchoDataSummary ⟹ QRMonDateListPlot;
```

» Data summary:

1 column 1		2 column 2	
Min	3.62906×10^{9}	Min	5.89
1st Qu	3.65278×10^{9}	1st Qu	19.695
Mean	3.67643×10^{9}	Mean	22.2982
Median	3.67645×10^{9}	Median	23.33
3rd Qu	3.70012×10^{9}	3rd Qu	25.89
Max	3.72375×10^{9}	Max	31.17

» Plot:



Financial time series

The following data is typical for financial time series. (Note the differences with the temperature time series.)

In[14]:=
```
finData = TimeSeries[FinancialData["NYSE:GE", {{2014, 1, 1}, {2018, 1, 1}, "Day"}]];
```

In[15]:=
```
QRMonUnit[finData]⟹
  QRMonEchoDataSummary⟹
  QRMonDateListPlot;
```

» Data summary: {

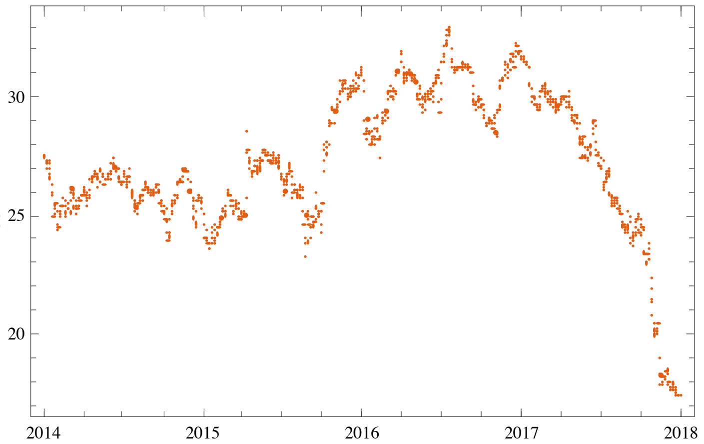
1 column 1	
Min	3.59761×10^9
1st Qu	3.62902×10^9
Median	3.66051×10^9
Mean	3.66062×10^9
3rd Qu	3.69202×10^9
Max	3.72349×10^9

,

2 column 2	
Min	17.36
1st Qu	25.5
Median	27.01
Mean	27.2791
3rd Qu	29.85
Max	32.93

}

» Plot:



Design considerations

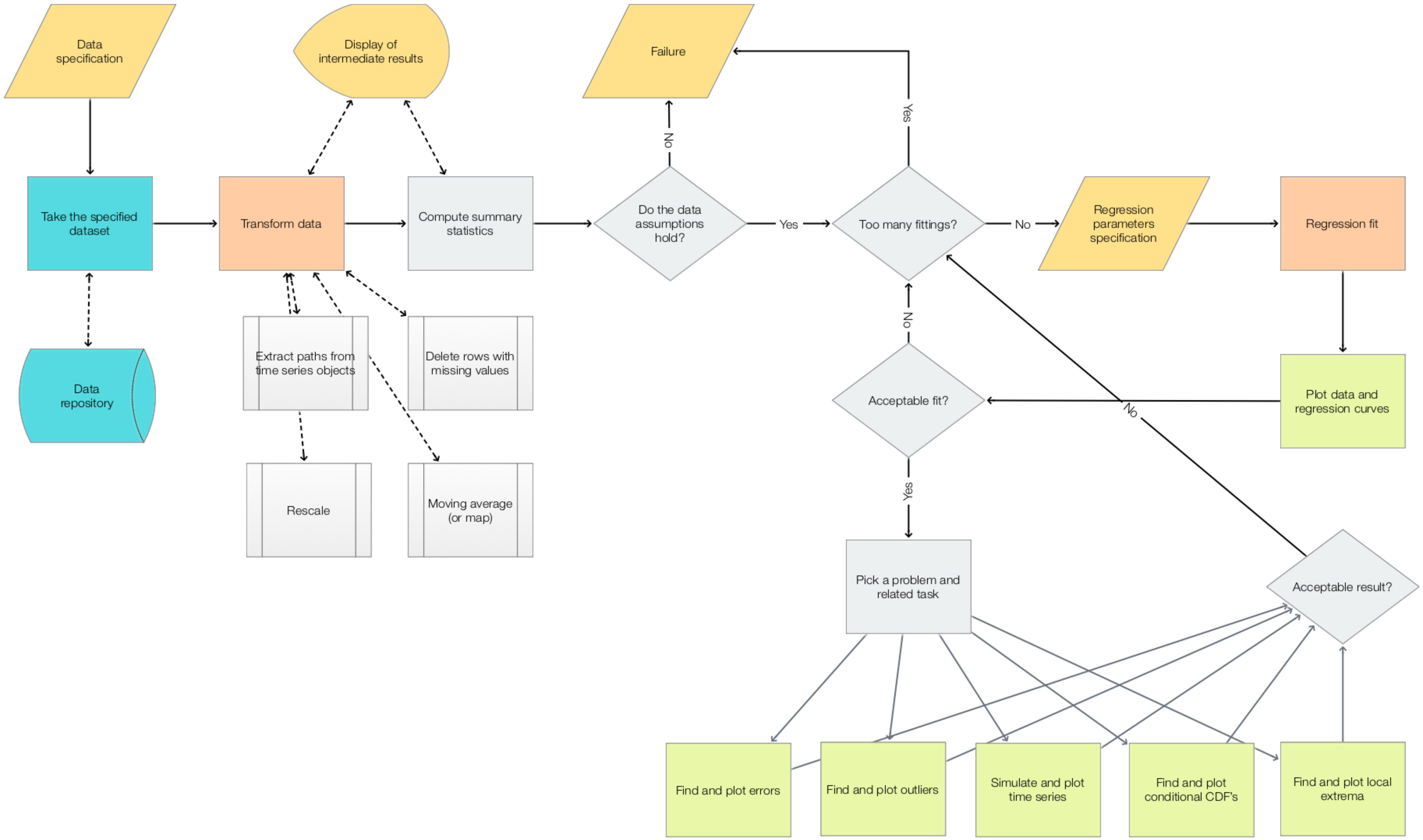
The steps of the main classification workflow addressed in this document follow.

1. Retrieving data from a data repository.
2. Optionally, transform the data.
   - 2.1. Delete rows with missing fields.
   - 2.2. Rescale data along one of the axes or both axes.
   - 2.3. Apply moving average.
3. Verify assumptions of the data
4. Run a regression algorithm with a certain basis of functions using:
   - 4.1. Quantile Regression, or
   - 4.2. Least Squares Regression.
5. Visualize the data and regression functions.
6. If the fit is not satisfactory go to step 4.

7. Utilize the found regression curves to compute:
   - 7.1. outliers,
   - 7.2. local extrema,
   - 7.3. approximation or fitting errors,
   - 7.4. conditional density distributions,
   - 7.5. time series simulations.

The following diagram gives a flow-chart interpretation of the list of steps above.

Out[16]=



In order to address:

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

it is beneficial to have a DSL for regression 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 we 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 QRMon for “**Q**uantile **R**egression **Mon**ad”.

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

$$\text{QRMon}[\_] \xrightarrow[\text{QRMonBind}[\text{QRMon}[\_],f\_]]{} f_1 \xrightarrow[\text{QRMonBind}[\text{QRMon}[\_],f\_]]{} f_2 \xrightarrow[\text{QRMonBind}[\text{QRMon}[\_],f\_]]{} \ldots \xrightarrow[\text{QRMonBind}[\text{QRMon}[\_],f\_]]{} f_k \tag{1}$$

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

$$\text{QRMon}[\text{pval\_},\ \text{context\_}] \xrightarrow[\text{QRMonBind}[m\_,f\_]]{} \ldots \left(\begin{cases} f_i[\$\text{QRMonFailure}] & m \equiv \$\text{QRMonFailure} \\ f_i[x\_,\ c\_\text{Association}] & m \text{ is } \text{QRMon}[x\_,\ c\_\text{Association}] \\ \$\text{QRMonFailure} & \text{otherwise} \end{cases} \right) \xrightarrow[\text{QRMonBind}[m\_,f\_]]{} \ldots \tag{2}$$

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

In the monad pipelines of QRMon we store different objects in the contexts for at least one of the following two reasons.

1. The object will be needed later on in the pipeline, or
2. The object is hard to compute.

Such objects are transformed data, regression functions, and outliers.

Let us list the desired properties of the monad.

- Rapid specification of non-trivial quantile regression workflows.
- The monad works with time series, numerical matrices, and numerical vectors.
- The pipeline values can be of different types. Most monad functions modify the pipeline value; some modify the context; some just echo results.
- The monad can do quantile regression with B-Splines, quantile regression fit and least squares fit with specified functions.
- The monad allows of cursory examination and summarization of the data.
- We can easily obtain the pipeline value, context, and different context objects for manipulation outside of the monad.
- We can easily plot different combinations of data, regression functions, outliers, approximation errors, etc.

The `QRMon` components and their interaction are fairly simple.

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

- (time series) data,
- regression functions,
- outliers and outlier regression functions.

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

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

QRMon overview

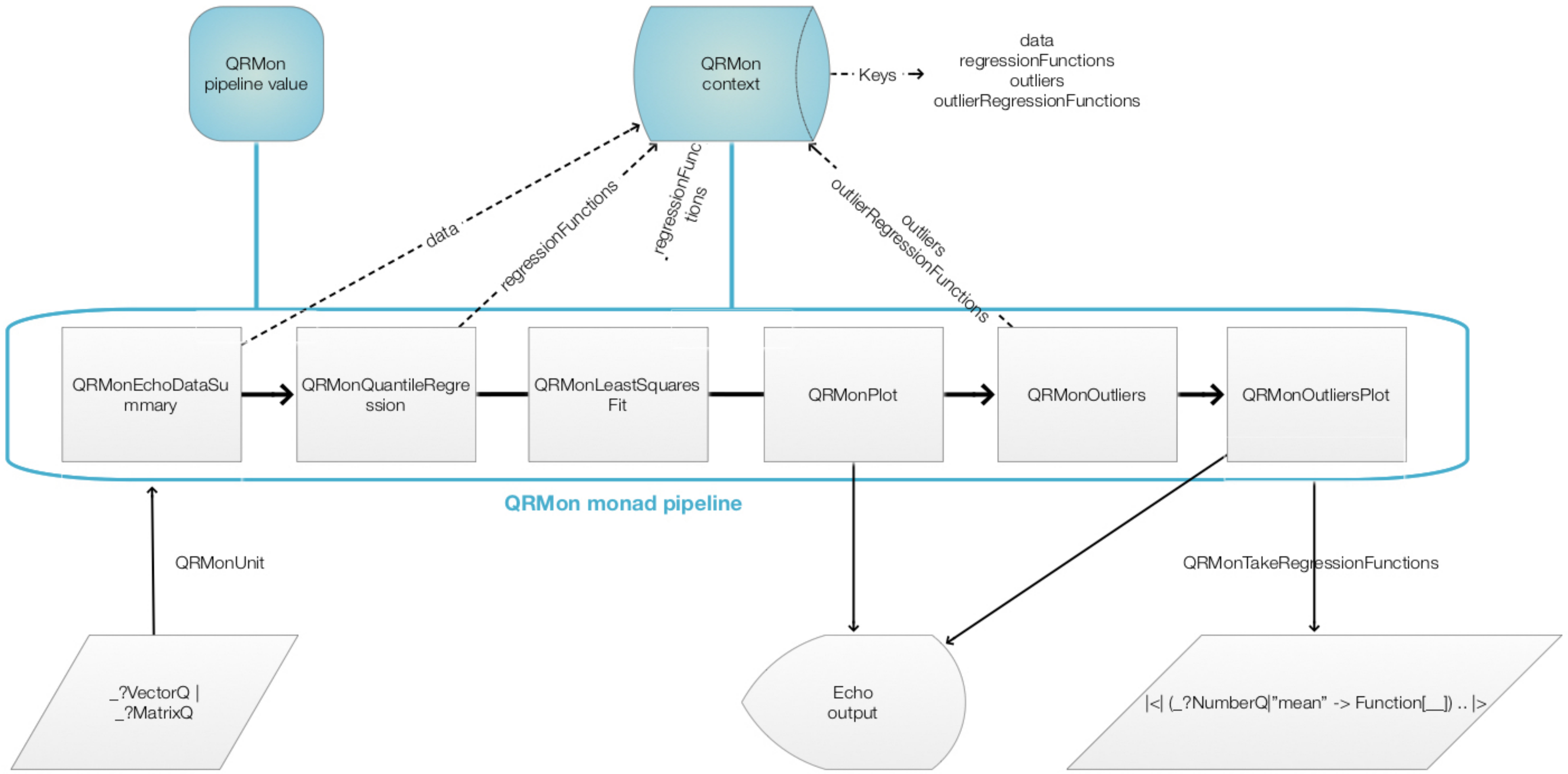
When using a monad we lift certain data into the “monad space”, using monad’s operations we navigate computations in that space, and at some point we take results from it.

With the approach taken in this document the “lifting” into the QRMon monad is done with the function QRMonUnit. Results from the monad can be obtained with the functions QRMonTakeValue, QRMonContext, or with the other QRMon functions with the prefix “QRMonTake” (see below.)

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

In[18]:=
```
imgPipeline = Import["~/Documents/Conceptual diagrams/Quantile regression workflows/QRMon-pipeline.jpg"];
imgPipeline
AutoCollapse[]
```

Out[19]=



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

Let us examine a concrete QRMon 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.

Out[•]=

operation	explanation	context keys
QRMonUnit[tsData] ⟹	lift to the monad	{}
QRMonEchoDataSummary ⟹	show the data summary	{}
QRMonQuantileRegression[12] ⟹	do quantile regression with 12 knots	{data, regressionFunctions}
QRMonLeastSquaresFit[{1, x, Sin[1.9953`*^-7 x]}] ⟹	do least squares fit	{data, regressionFunctions}
QRMonPlot ⟹	plot data and regression quantiles	{data, regressionFunctions}
QRMonOutliers ⟹	find outliers	{data, regressionFunctions, outliers, outlierRegressionFunctions}
QRMonOutliersPlot	plot data and outliers	{data, regressionFunctions, outliers, outlierRegressionFunctions}

Here is the output of the pipeline:

» Data summary: {

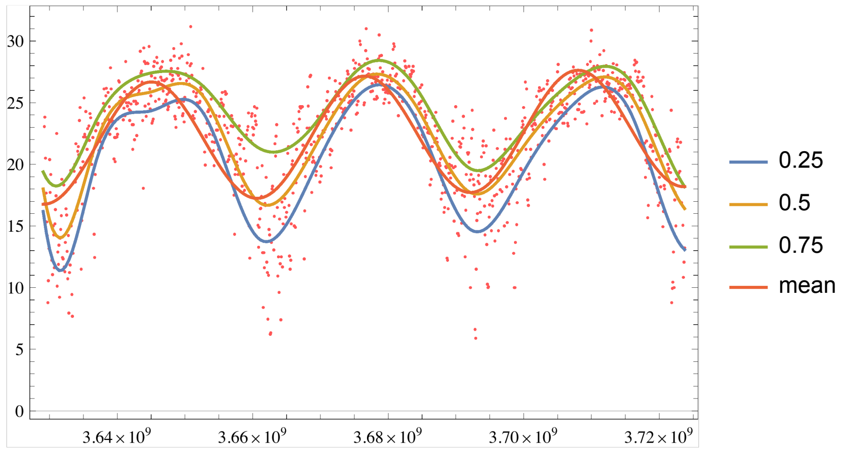
1 column 1	
Min	3 629 059 200
1st Qu	3 652 776 000
Mean	3.67643×10^9
Median	3 676 449 600
3rd Qu	3 700 123 200
Max	3 723 753 600

,

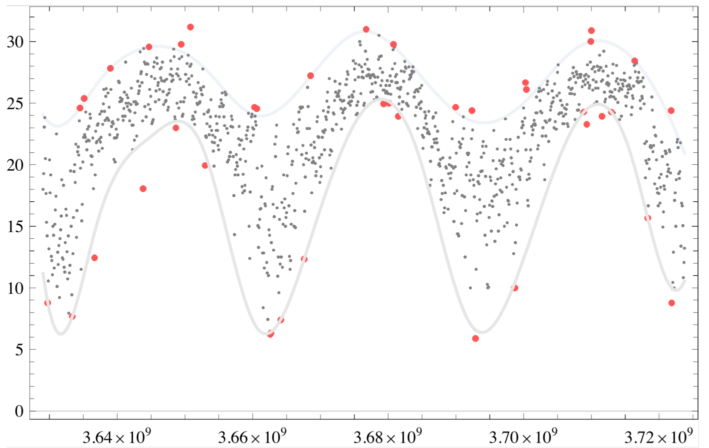
2 column 2	
Min	5.89
1st Qu	19.695
Mean	22.2982
Median	23.33
3rd Qu	25.89
Max	31.17

}

» Plot:



» Outliers plot:



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

The QRMon 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 QRMon operations.

Monad functions interaction with the pipeline value and context

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

Out[ ]=

#	name	echoes result	puts in context	uses from context	uses pipeline value
1	*operations*				
2	QRMonBandsSequence	no	none	{data, regressionFunctions}	no
3	QRMonConditionalCDF	no	none	{data, regressionFunctions}	no
4	QRMonConditionalCDFPlot	yes	none	{data, regressionFunctions}	no
5	QRMonDateListPlot	yes	data	data	via QRMonGetData
6	QRMonDateListPlot	yes	data	{data, regressionFunctions}	via QRMonGetData
7	QRMonDeleteMissing	no	data	data	via QRMonGetData
8	QRMonEchoDataSummary	yes	none	data	via QRMonGetData
9	QRMonErrorPlots	yes	none	{data, regressionFunctons}	no
10	QRMonEvaluate	no	none	{data, regressionFunctons}	no
11	QRMonFit	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonLeastSquaresFit
12	QRMonGetData	no	data	data	if "data" is not in the context
13	QRMonGridSequence	no	none	{data, regressionFunctions}	no
14	QRMonLeastSquaresFit	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonGetData
15	QRMonMovingAverage	no	none	data	no
16	QRMonMovingMap	no	none	data	no
17	QRMonMovingMedian	no	none	data	no
18	QRMonOutliers	no	{data, outliers}	data	via QRMonGetData
19	QRMonOutliersPlot	yes	{data, outliers}	data	via QRMonOutliers
20	QRMonOutliersPlot	yes	none	{data, outliers}	no
21	QRMonPlot	yes	data	data	via QRMonGetData
22	QRMonPlot	yes	data	{data, regressionFunctions}	no
23	QRMonQuantileRegression	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonGetData
24	QRMonQuantileRegressionFit	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonGetData
25	QRMonRegression	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonQuantileRegression
26	QRMonRegressionFit	no	{data, regressionFunctions}	{data, regressionFunctions}	via QRMonQuantileRegressionFit
27	QRMonRescale	no	data	data	via QRMonGetData
28	*setters*				
29	QRMonSetData	no	data	none	no
30	QRMonSetRegressionFunctions	no	regressionFunctions	none	no
31	*takers*				
32	QRMonTakeData	no	none	data	no
33	QRMonTakeOutlierRegressionFunctions	no	none	outlierRegressionFunctions	no
34	QRMonTakeOutliers	no	none	outliers	no
35	QRMonTakeRegressionFunctions	no	none	regressionFunctions	no
36	*droppers*				
37	QRMonDropData	no	none	data	no
38	QRMonDropOutlierRegressionFunctions	no	none	outlierRegressionFunctions	no
39	QRMonDropOutliers	no	none	outliers	no
40	QRMonDropRegressionFunctions	no	none	regressionFunctions	no

The following table shows the functions that are function synonyms or short-cuts.

Out[ ]=

#	name	same as
1	QRMonDateListPlot	QRMonPlot[DateListPlot → True]
2	QRMonFit	QRMonLeastSquaresFit
3	QRMonRegression	QRMonQuantileRegression
4	QRMonRegressionFit	QRMonQuantileRegressionFit

State monad functions

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

Out[ ]=

#	name	description
1	QRMon	monad head
2	QRMonAddToContext	adds the pipeline value into the context
3	QRMonBind	monad binding function
4	QRMonContexts	gives the contexts associated with a monad head
5	QRMonDropFromContext	drops from the context elements specified by their keys
6	QRMonEcho	echoes argument(s); a monad version of Echo
7	QRMonEchoContext	echoes the context
8	QRMonEchoFunctionContext	echoes the result of a function applied to the context
9	QRMonEchoFunctionValue	echoes the result of a function applied to the pipeline value
10	QRMonEchoValue	echoes the pipeline value
11	QRMonFail	gives the monad failure symbol
12	QRMonIfElse	choses between two functions based on condition
13	QRMonIterate	general interation function
14	QRMonModifyContext	modifies the context with the argument function
15	QRMonModule	allows faster pipeline function specifications
16	QRMonOption	ignores a result if it is failure
17	QRMonPutContext	replaces the context with the argument
18	QRMonPutValue	replaces the pipeline value with the argument
19	QRMonRetrieveFromContext	using a key retrieves into the pipleine a value from the context
20	QRMonSetContext	same as QRMonPutContext
21	QRMonSetValue	same as QRMonPutValue
22	QRMonSucceed	gives a success element of the form QRMon[___]
23	QRMonTakeContext	takes the context
24	QRMonTakeValue	takes the pipeline value
25	QRMonUnit	lifts to the monad
26	QRMonUnitQ	gives True if monad unit
27	QRMonWhen	executes a function based on a condition

Monad elements

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

The monad head

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

```
In[21]:= QRMonUnit[{{1, 223}, {2, 323}}, <||>] ⟹ QRMonEchoDataSummary;
```

```
                  1 column 1        2 column 2
                  1st Qu  1         1st Qu  223
                  Min     1         Min     223
» Data summary: { Mean    1.5   ,   Mean    273  }
                  Median  1.5       Median  273
                  3rd Qu  2         3rd Qu  323
                  Max     2         Max     323
```

Lifting data to the monad

The function lifting the data into the monad QRMon is `QRMonUnit`.

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

```
In[22]:= QRMonUnit[distData] ⟹ QRMonEchoDataSummary;
```

```
                  1 column 1                   2 column 2
                  Min     -3.                  Min     -0.562388
                  1st Qu  -1.5025              1st Qu  0.031954
» Data summary: { Median  0.                 , Median  0.20604    }
                  Mean    8.86701×10^-17       Mean    0.300542
                  3rd Qu  1.5025               3rd Qu  0.605181
                  Max     3.                   Max     1.34789
```

```
In[23]:= QRMonUnit[] ⟹ QRMonSetData[distData] ⟹ QRMonEchoDataSummary;
```

```
                  1 column 1                   2 column 2
                  Min     -3.                  Min     -0.562388
                  1st Qu  -1.5025              1st Qu  0.031954
» Data summary: { Median  0.                 , Median  0.20604    }
                  Mean    8.86701×10^-17       Mean    0.300542
                  3rd Qu  1.5025               3rd Qu  0.605181
                  Max     3.                   Max     1.34789
```

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

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

- time series,
- numerical vectors,
- numerical matrices of rank two.

When the data lifted to the monad is a numerical vector `vec` it is assumed that `vec` has to become the second column of “time series” matrix; the first column is derived with `Range[Length[vec]]` .

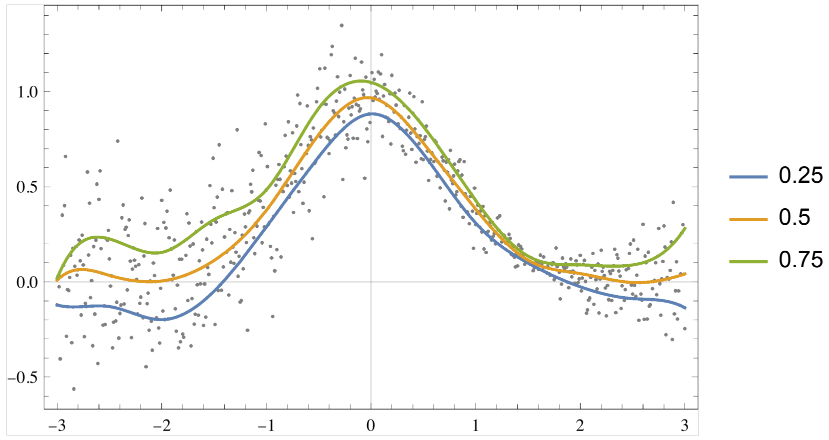
Generally, WL makes it easy to extract columns datasets and time series in order to obtain numerical matrices, so datasets and time series and not currently supported in `QRMon`.

Quantile regression with B-splines

This computes quantile regression with 12 knots. (Using Linear Programming algorithms; see [AA2] for details.)

In[24]:=
```
QRMonUnit[distData]⟹
  QRMonQuantileRegression[12]⟹
  QRMonPlot;
```

» Plot:



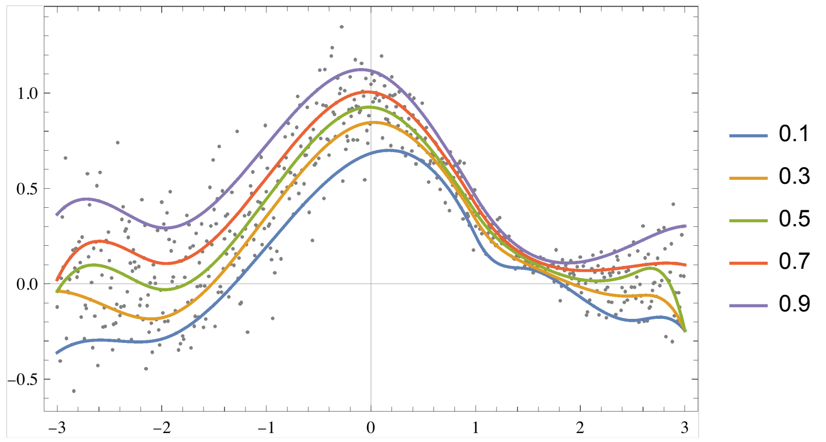
In[25]:=
```
Options[QRMonQuantileRegression]
```

Out[25]=
```
{InterpolationOrder → 3, Method → {LinearProgramming, Method → CLP}}
```

Lets compute regression using a list of particular knots, specified quantiles, and the Linear Programming library CLP:

In[26]:=
```
p =
  QRMonUnit[distData]⟹
   QRMonQuantileRegression[{-3, -2, 1, 0, 1, 1.5, 2.5, 3}, Range[0.1, 0.9, 0.2], Method → {LinearProgramming, Method → "CLP"}]⟹
   QRMonPlot;
```

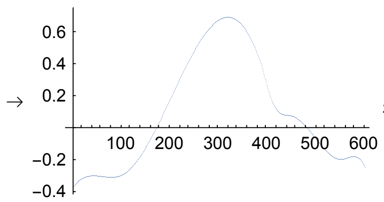
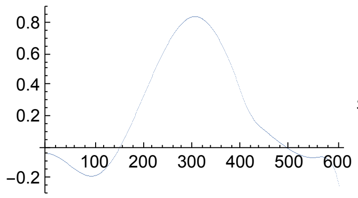
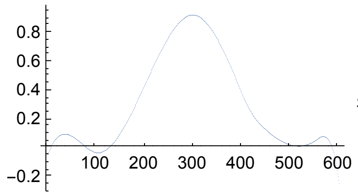
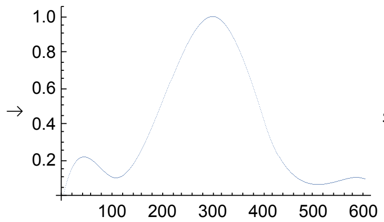
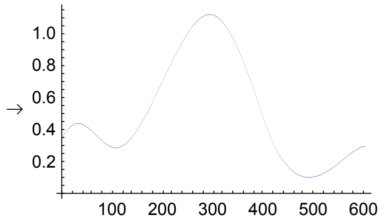
» Plot:



Remark: As it was mentioned above the function `QRMonRegression` is a synonym of `QRMonQuantileRegression`.

The fit functions can be extracted from the monad with `QRMonTakeRegressionFunctions`, which gives an association of quantiles and pure functions.

In[27]:=
```
ListPlot[# /@ distData⟦All, 1⟧] & /@ (p⟹QRMonTakeRegressionFunctions)
```

Out[27]= ⟨| 0.1 →  , 0.3 →  , 0.5 →  , 0.7 →  , 0.9 →  |⟩

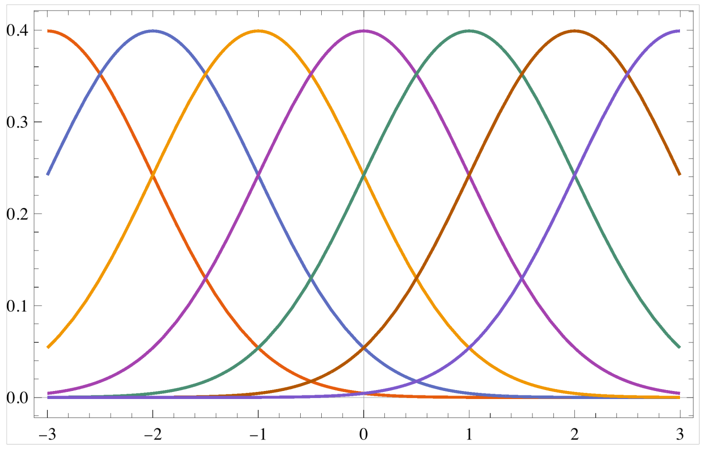
Quantile regression fit and Least squares fit

Instead of using B-spline basis of functions we can compute a fit with our own basis of functions.

Here is a basis functions:

In[28]:=
```
bFuncs = Table[PDF[NormalDistribution[m, 1], x], {m, Min[distData[[All, 1]]], Max[distData[[All, 1]]], 1}];
Plot[bFuncs, {x, Min[distData[[All, 1]]], Max[distData[[All, 1]]]}, PlotRange → All, PlotTheme -> "Scientific"]
```

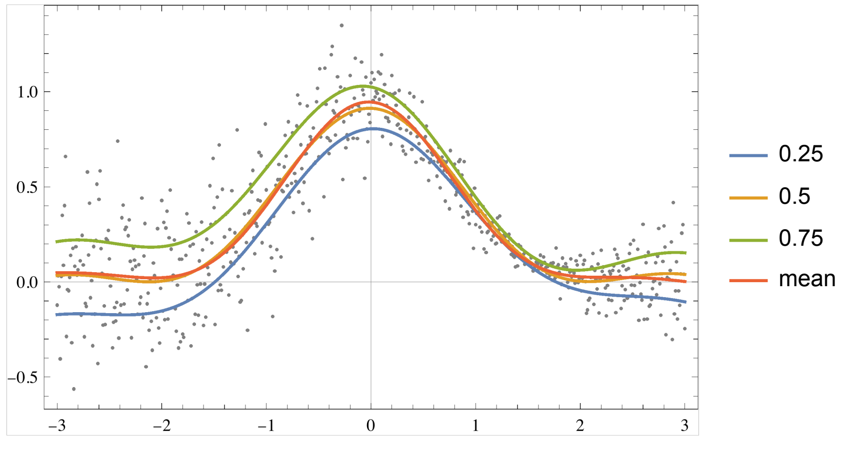
Out[29]=



Here we do a Quantile Regression fit, a Least Squares fit, and plot the results:

In[30]:=
```
p =
  QRMonUnit[distData] ⟹
   QRMonQuantileRegressionFit[bFuncs] ⟹
   QRMonLeastSquaresFit[bFuncs] ⟹
   QRMonPlot;
```

» Plot:

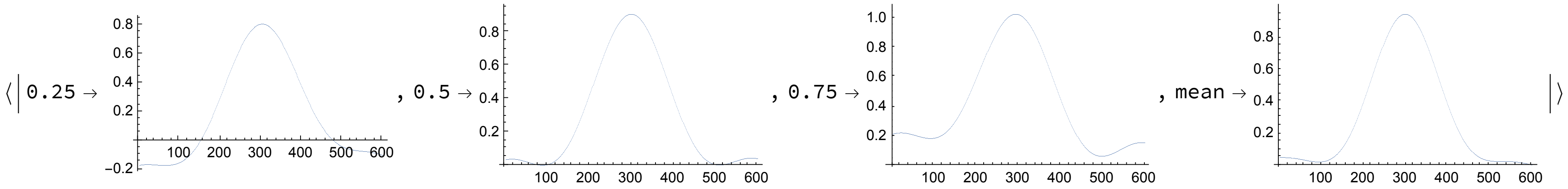


Remark: As it was mentioned above the function `QRMonRegressionFit` is a synonym of `QRMonQuantileRegressionFit` and `QRMonFit` is a synonym of `QRMonLeastSquaresFit`

As it was pointed out in the previous sub-section, the fit functions can be extracted from the monad with `QRMonTakeRegressionFunctions`. Here the keys of the returned/taken association consist of quantiles and "mean" since we applied both Quantile Regression and Least Squares Regression.

```
In[31]:= ListPlot[# /@ distData[[All, 1]]] & /@ (p ⟹ QRMonTakeRegressionFunctions)
```

Out[31]=



Default basis to fit (using Chebyshev polynomials)

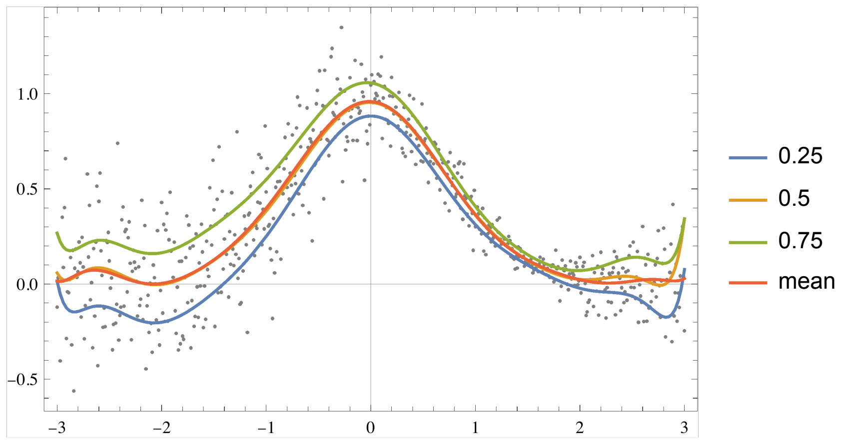
One of the main advantages of using the function `QuanileRegression` of the package [AAp4] is that the functions used to do the regression with are specified a few parameters that are numbers. (Most often only the number of knots is sufficient.) This achieved by using a basis of B-spline functions of certain interpolation order.

We want similar behaviour for Quantile Regression fitting we need to select a certain well known basis with certain desirable properties. Such basis is given by Chebyshev polynomials of first kind [Wk3]. Chebyshev polynomials bases can be easily generated in Mathematica with the functions `ChebyshevT` or `ChebyshevU`.

Here is an application of fitting with a basis of 12 Chebyshev polynomials of first kind:

```
In[32]:= QRMonUnit[distData] ⟹
           QRMonQuantileRegressionFit[12] ⟹
           QRMonLeastSquaresFit[12] ⟹
           QRMonPlot;
```

» Plot:

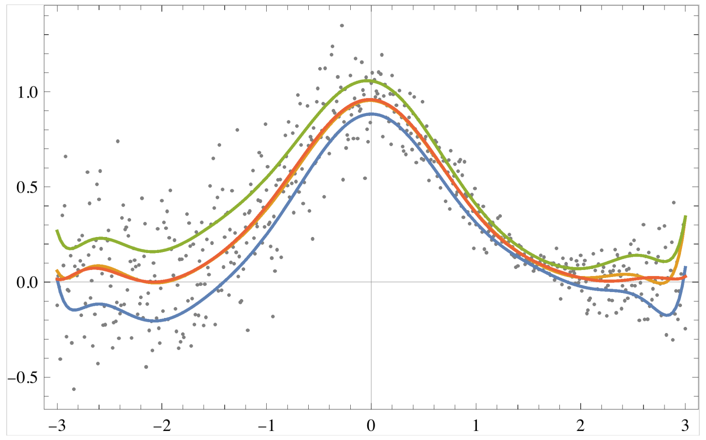


The code above is equivalent to the following code:

```
In[33]:= bfuncs = Table[ChebyshevT[i, Rescale[x, MinMax[distData[[All, 1]]], {-0.95, 0.95}]], {i, 0, 12}];
```

In[34]:=
```
p =
  QRMonUnit[distData]⟹
   QRMonQuantileRegressionFit[bfuncs]⟹
   QRMonLeastSquaresFit[bfuncs]⟹
   QRMonPlot;
```

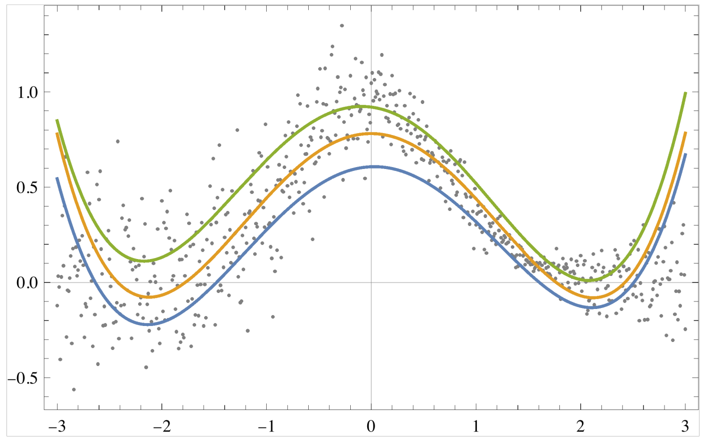
» Plot:



The shrinking of the ChebyshevT domain seen in the definitions of bfuncs is done in order to prevent approximation error effects at the ends of the data domain. The following code uses the ChebyshevT domain {–1, 1} instead of the domain {–0.95, 0.95} used above.

In[35]:=
```
QRMonUnit[distData]⟹
  QRMonQuantileRegressionFit[{4, {-1, 1}}]⟹
  QRMonPlot;
```

» Plot:



Regression functions evaluation

The computed quantile and least squares regression functions can be evaluated with the monad function QRMonEvaluate.

Evaluation for a given value of the independent variable:

In[36]:=
```
p⟹QRMonEvaluate[0.12]⟹QRMonTakeValue
```

Out[36]=
```
<|0.25 → 0.869342, 0.5 → 0.938764, 0.75 → 1.03505, mean → 0.942931|>
```

Evaluation for a vector of values:

In[37]:=
```
p⟹QRMonEvaluate[Range[-1, 1, 0.5]]⟹QRMonTakeValue
```

Out[37]=
```
<|0.25 → {0.250039, 0.65972, 0.88304, 0.671719, 0.312368}, 0.5 → {0.382968, 0.75689, 0.954513, 0.740732, 0.366504},
  0.75 → {0.548785, 0.887067, 1.05699, 0.814377, 0.408361}, mean → {0.398379, 0.771228, 0.958975, 0.745513, 0.362792}|>
```

Evaluation for complicated lists of numbers:

```
In[38]:= p⟹QRMonEvaluate[{0, 1, {1.5, 1.4}}]⟹QRMonTakeValue
```

```
Out[38]= <|0.25 → {0.88304, 0.312368, {0.0993299, 0.131628}}, 0.5 → {0.954513, 0.366504, {0.124344, 0.160315}},
 0.75 → {1.05699, 0.408361, {0.154165, 0.190211}}, mean → {0.958975, 0.362792, {0.125647, 0.158299}}|>
```

The obtained values can be used to compute estimates of the distributions of the predicted variable. See the sub-sections “Estimating conditional distributions” and “Predicted variable simulation”.

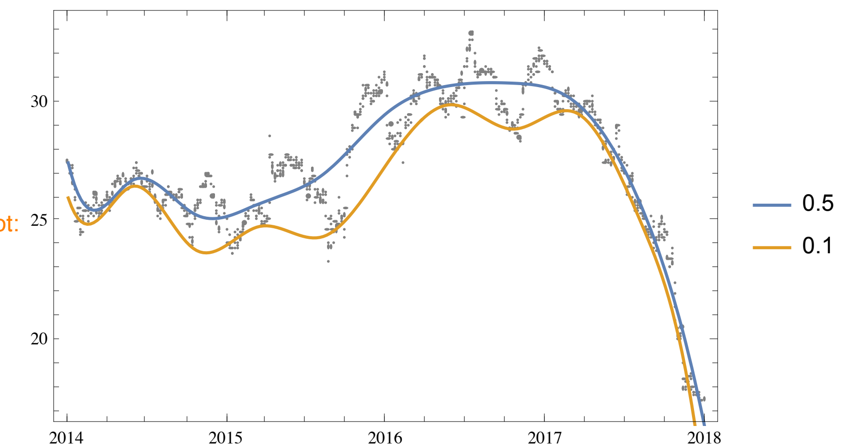
Errors and error plots

Here with “errors” we mean the differences between data’s dependent variable values and the corresponding values calculated with the fitted regression curves.

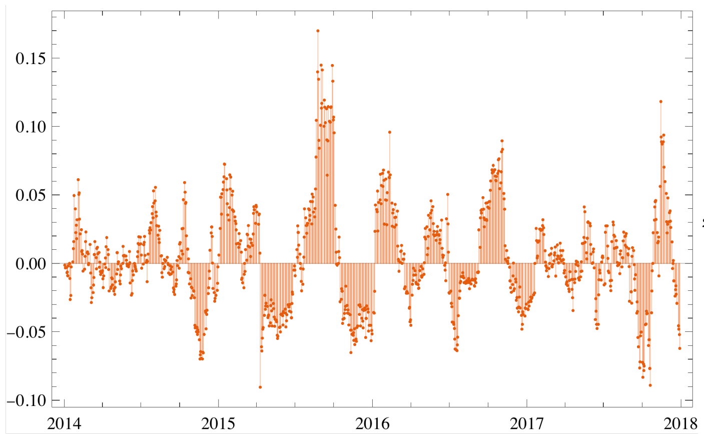
In the pipeline below we compute couple of regression quantiles, plot them together with the data, we plot the errors, compute the errors, and summarize them.

```
In[39]:= QRMonUnit[finData]⟹
  QRMonQuantileRegression[10, {0.5, 0.1}]⟹
  QRMonDateListPlot[Joined → False]⟹
  QRMonErrorPlots["DateListPlot" → True, Joined → False]⟹
  QRMonErrors⟹
  QRMonEchoFunctionValue["Errors summary:", RecordsSummary[#[[All, 2]]] & /@ # &];
```

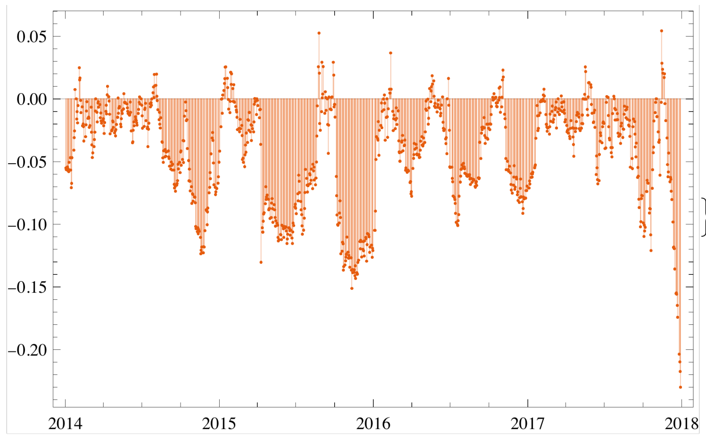
» Plot:



» Error plots: {0.5 →



, 0.1 →



}

» Errors summary: <|0.5 →

1 column 1	
Min	-0.0906067
1st Qu	-0.0201859
Median	-8.10259×10^{-11}
Mean	0.00349736
3rd Qu	0.02445
Max	0.16995

, 0.1 →

1 column 1	
Min	-0.230056
1st Qu	-0.0691127
Mean	-0.0433778
Median	-0.03276
3rd Qu	-0.0109969
Max	0.0544299

|>

Each of the functions QRMonErrors and QRMonErrorPlots compute the errors. (That computation is considered cheap.)

Finding outliers

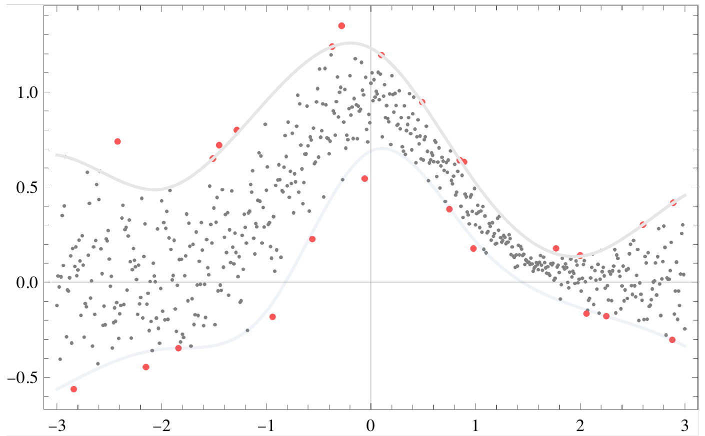
Finding outliers can be done with the function `QRMonOultiers`. The outliers found by `QRMonOutliers` are simply points that above or below certain regression quantile curves, for example, the ones corresponding to 0.98 and 0.02.

Here is an example:

In[40]:=
```
p =
  QRMonUnit[distData]⟹
   QRMonQuantileRegression[6, {0.02, 0.98}]⟹
   QRMonOutliers⟹
   QRMonEchoValue⟹
   QRMonOutliersPlot;
```

```
» value: ⟨|bottomOutliers → {{-2.84, -0.562388}, {-2.15, -0.445678}, {-1.84, -0.346993},
    {-0.94, -0.18139}, {-0.56, 0.226788}, {-0.06, 0.545139}, {0.75, 0.384333}, {0.98, 0.176936}, {2.06, -0.165037}, {2.25, -0.178139}, {2.88, -0.30321}},
  topOutliers → {{-2.42, 0.739855}, {-1.51, 0.64987}, {-1.45, 0.720544}, {-1.28, 0.799602}, {-0.37, 1.2381}, {-0.28, 1.34789}, {0.1, 1.19396}, {0.49, 0.946943},
    {0.85, 0.64259}, {0.89, 0.632092}, {1.77, 0.177765}, {2., 0.140101}, {2.6, 0.302376}, {2.89, 0.417835}}|⟩
```

» Outliers plot:



The function `QRMonOutliers` puts in the context values for the keys “outliers” and “outlierRegressionFunctions”. The former is for the found outliers, the latter is for the functions corresponding to the used regression quantiles.

In[41]:=
```
Keys[p⟹QRMonTakeContext]
```

Out[41]=
```
{data, regressionFunctions, outliers, outlierRegressionFunctions}
```

Here are the corresponding quantiles of the plot above:

In[42]:=
```
Keys[p⟹QRMonTakeOutlierRegressionFunctions]
```

Out[42]=
```
{0.02, 0.98}
```

The control of the computation of the outliers is done though the arguments and options of `QRMonQuantileRegression`.

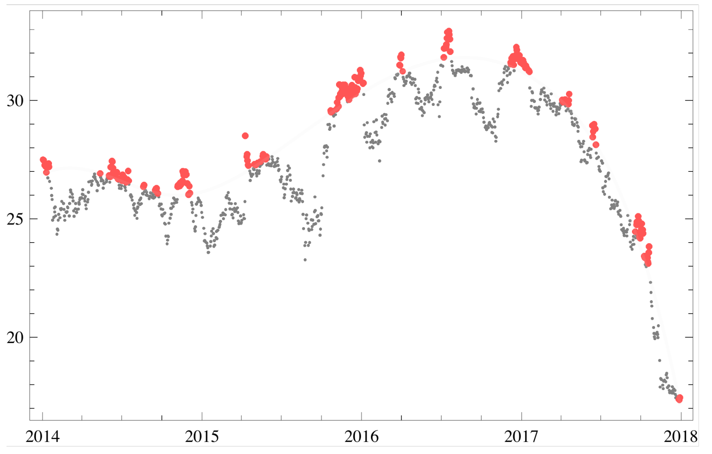
If only one regression quantile is found the and the corresponding quantile is less than 0.5 then `QRMonOutliers` finds only bottom outliers. If only one regression quantile is found the and the corresponding quantile is greater than 0.5 then `QRMonOutliers` finds only top outliers.

Here is an example for finding only the top outliers:

In[43]:=
```
QRMonUnit[finData] ⟹
  QRMonQuantileRegression[5, 0.8] ⟹
  QRMonOutliers ⟹
  QRMonEchoFunctionContext["outlier quantiles:", Keys[#outlierRegressionFunctions] &] ⟹
  QRMonOutliersPlot["DateListPlot" → True];
```

» outlier quantiles: {0.8}

» Outliers plot:



Plotting outliers

The function QRMonOutliersPlot makes an outliers plot. If the outliers are not in the context then QRMonOutliersPlot calls QRMonOutliers first.

Here are the options of QRMonOutliersPlot:

In[44]:= `Options[QRMonOutliersPlot]`

Out[44]= `{Echo → True, DateListPlot → False, ListPlot → {Joined → False}, Plot → {}}`

The default behavior is to echo the plot. That can be suppressed with the option “Echo”.

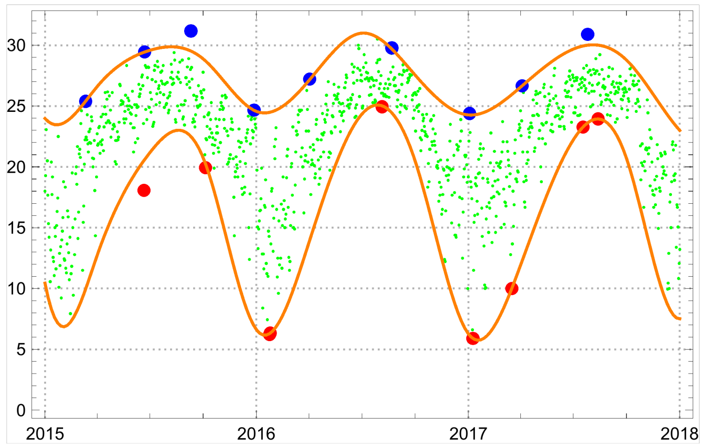
QRMonOutliersPlot utilizes combines with Show two plots:

- one with ListPlot (or DateListPlot) for the data and the outliers,
- the other with Plot for the regression quantiles used to find the outliers.

That is why separate lists of options can be given to manipulate those two plots. The option DateListPlot can be used make plots with date or time axes.

In[45]:=
```
QRMonUnit[tsData]⟹
 QRMonQuantileRegression[12, {0.01, 0.99}]⟹
 QRMonOutliersPlot[
  "Echo" → False,
  "DateListPlot" → True,
  ListPlot → {PlotStyle → {Green, {PointSize[0.02], Red}, {PointSize[0.02], Blue}}, Joined → False, PlotTheme → "Grid"},
  Plot -> {PlotStyle → Orange}]⟹
 QRMonTakeValue
```

Out[45]=



Estimating conditional distributions

Consider the following problem:

How to estimate the conditional density of the predicted variable given a value of the conditioning independent variable?

(In other words, find the distribution of the *y*-values for a given, fixed *x*-value.)

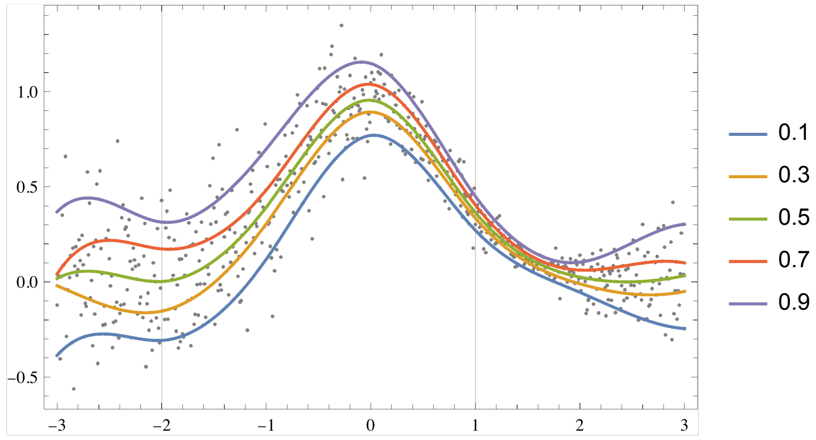
The solution of this problem using Quantile Regression is discussed in detail in [PG1] and [AA4].

Finding a solution of this problem can be seen as a primary motivation to develop Quantile Regression algorithms.

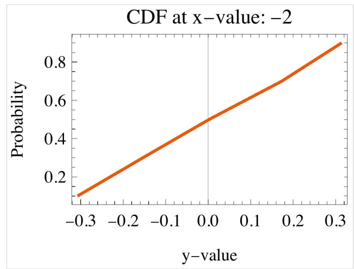
The following pipeline (1) computes and plots a set of (five) regression quantiles and (2) then using the found regression quantiles computes and plots the conditional distributions for two focus points (–2 and 1.)

In[46]:=
```
QRMonUnit[distData]⟹
  QRMonQuantileRegression[6, Range[0.1, 0.9, 0.2]]⟹
  QRMonPlot[GridLines → {{-2, 1}, None}]⟹
  QRMonConditionalCDF[{-2, 1}]⟹
  QRMonConditionalCDFPlot;
```

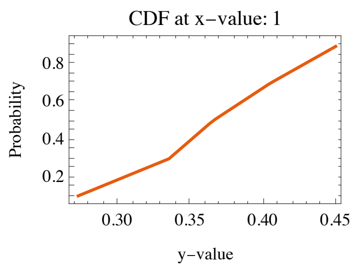
» Plot:



» Conditional CDF's: ⟨| -2 →



, 1 →



|⟩

Moving average, moving median, and moving map

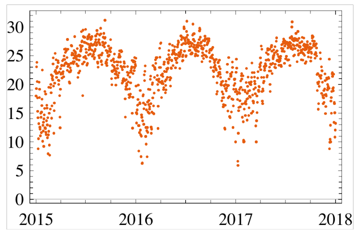
Fairly often it is a good idea for a given time series to apply the functions moving average, moving median.
We might want to:

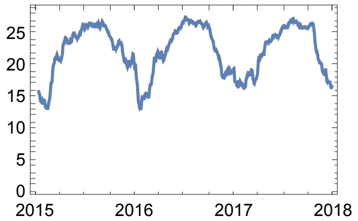
- visualize the obtained transformed data,
- do regression over the transformed data,
- compare with regression curves over the original data.

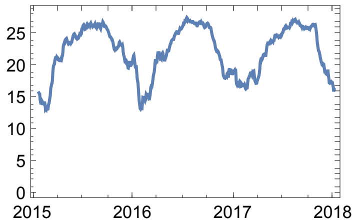
For these reasons QRMon has the functions `QRMonMovingAverage`, `QRMonMovingMedian`, and `QRMonMovingMap` that correspond to the built-in functions MovingAverage, MovingMedian, and MovingMap.

Examples follow.

In[47]:=
```
QRMonUnit[tsData] ⟹
  QRMonDateListPlot[ImageSize → Small] ⟹
  QRMonMovingAverage[20] ⟹
  QRMonEchoFunctionValue["Moving avg: ", DateListPlot[#, ImageSize → Small] &] ⟹
  QRMonMovingMap[Mean, Quantity[20, "Days"]] ⟹
  QRMonEchoFunctionValue["Moving map: ", DateListPlot[#, ImageSize → Small] &];
```

» Plot: 

» Moving avg: 

» Moving map: 

Time series simulation

Consider the problem of making a time series that is simulation of a process given with a known time series.

More formally,

- we are given a time-axis grid (regular or irregular),
- we consider each grid node to correspond to a random variable,
- we want to generate time series based on the empirical CDF’s of the grid node random variables.

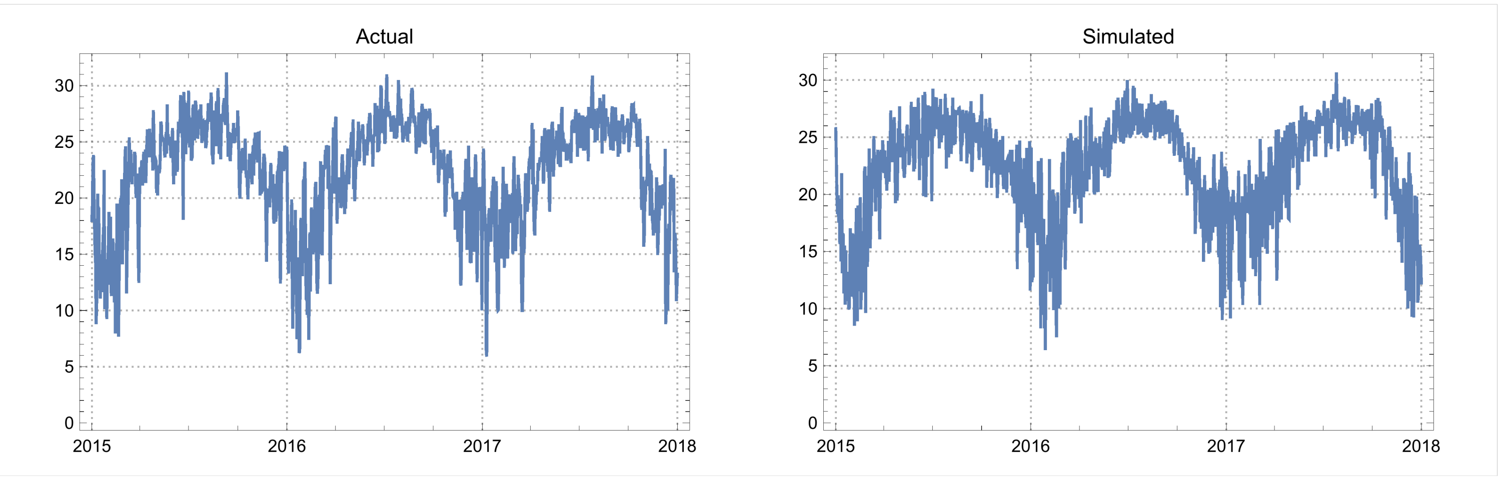
The formulation of the problem hints to an (almost) straightforward implementation using Quantile Regression.

In[48]:=
```
p = QRMonUnit[tsData] ⟹
   QRMonQuantileRegression[30, Join[{0.01}, Range[0.1, 0.9, 0.1], {0.99}]];
```

In[49]:=
```
tsNew =
  p⟹
   QRMonSimulate[1000] ⟹
   QRMonTakeValue;
```

In[50]:=
```
opts = {ImageSize → Medium, PlotTheme → "Detailed"};
GraphicsGrid[{{DateListPlot[tsData, PlotLabel → "Actual", opts], DateListPlot[tsNew, PlotLabel → "Simulated", opts]}}]
```

Out[51]=



Finding local extrema in noisy data

Using regression fitting -- and Quantile Regression in particular -- we can easily construct semi-symbolic algorithms for finding local extrema in noisy time series data; see [AA5]. The `QRMon` function with such an algorithm is `QRMonFindLocalExtrema`.

In brief, the algorithm steps are as follows. (For more details see [AA5].)

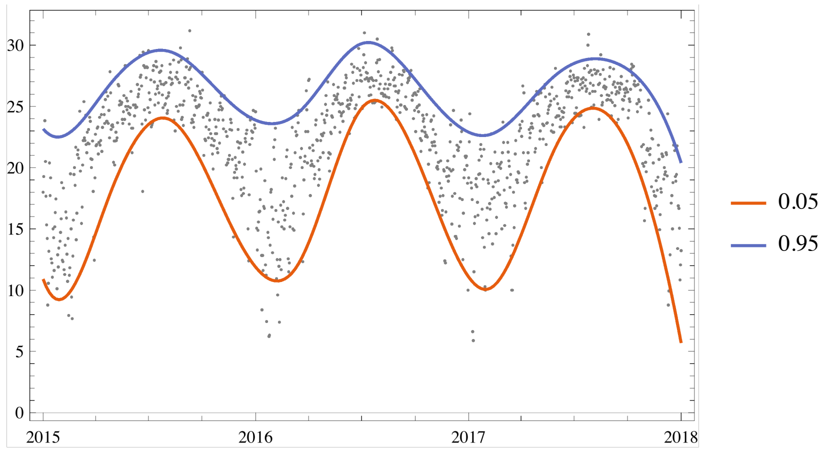
1. Fit a polynomial through the data.
2. Find the local extrema of the fitted polynomial. (We will call them fit estimated extrema.)
3. Around each of the fit estimated extrema find the most extreme point in the data by nearest neighbors search (by using `Nearest`).

The function `QRMonFindLocalExtrema` uses the regression quantiles previously found in the monad pipeline (and stored in the context.) The bottom regression quantile is used for finding local minima, the top regression quantile is used for finding the local maxima.

An example of finding local extrema follows.

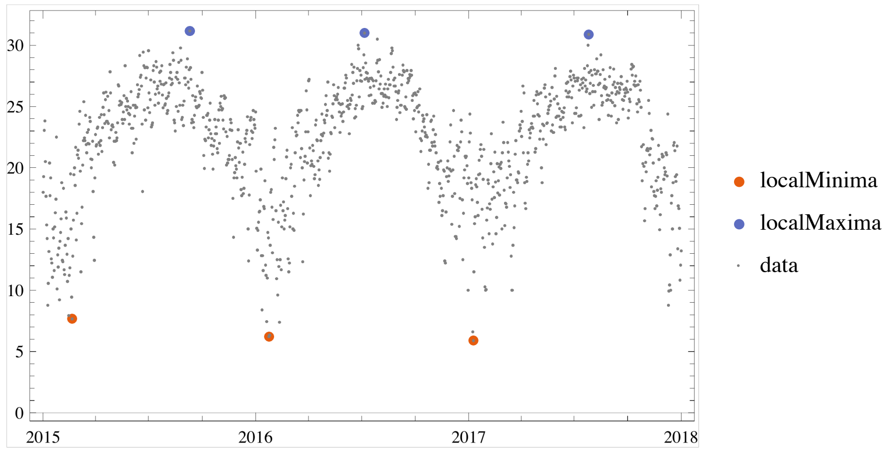
In[52]:=
```
QRMonUnit[TimeSeriesWindow[tsData, {{2015, 1, 1}, {2018, 12, 31}}]]⟹
  QRMonQuantileRegression[10, {0.05, 0.95}]⟹
  QRMonDateListPlot[Joined → False, PlotTheme -> "Scientific"]⟹
  QRMonFindLocalExtrema["NumberOfProximityPoints" → 100]⟹
  QRMonEchoValue⟹
  QRMonAddToContext⟹
  QRMonEchoFunctionContext[DateListPlot[{#localMinima, #localMaxima, #data},
     PlotStyle → {PointSize[0.015], PointSize[0.015], Gray}, Joined → False, PlotLegends → {"localMinima", "localMaxima", "data"}, PlotTheme -> "Scientific"] &];
```

» Plot:



» value: ⟨| localMinima → {{3.63338 × 10^9, 7.67}, {3.66258 × 10^9, 6.22}, {3.69291 × 10^9, 5.89}}, localMaxima → {{3.65083 × 10^9, 31.17}, {3.67675 × 10^9, 31.}, {3.71002 × 10^9, 30.89}} |⟩

»



Note that in the pipeline above in order to plot the data and local extrema together some additional steps are needed. The result of `QRMonFindLocalExtrema` becomes the pipeline value; that pipeline value is displayed with `QRMonEchoValue`, and stored in the context with `QRMonAddToContext`. If the pipeline value is an association -- which is the case here -- the monad function QRMonAddToContext joins that association with the context association. In this case this means that we will have key-value elements in the context for “localMinima” and “localMaxima”. The date list plot the at the end of the pipeline uses values of those context keys (together with the value for “data”.)

Setters, takers, and droppers

The values from the monad context can be set, obtained, or dropped with the corresponding “setter”, “taker”, and “dropper” functions as summarized in a previous section.

For example:

In[53]:=
```
p =
  QRMonUnit[distData] ⟹ QRMonQuantileRegressionFit[2];
```

In[54]:=
```
p ⟹ QRMonTakeRegressionFunctions
```

Out[54]=
```
⟨| 0.25 → (0.0260914 + 0.00552391 #1 + 4.947 × 10^-14 #1^2 &), 0.5 → (0.20604 + 2.75372 × 10^-14 #1 + 7.00728 × 10^-15 #1^2 &), 0.75 → (0.604829 + 6.92879 × 10^-13 #1 + 5.75419 × 10^-13 #1^2 &) |⟩
```

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

In[55]:=
```
p = QRMonUnit[RandomReal[1, {2, 2}]] ⟹ QRMonAddToContext["data"];
```

In[56]:=
```
(p ⟹ QRMonTakeContext) ["data"]
```

Out[56]=
```
{{0.507495, 0.754024}, {0.885643, 0.336476}}
```

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

Here is an example of the “data dropper” QRMonDropData:

In[57]:=
```
p⟹QRMonDropData⟹QRMonTakeContext
```

Out[57]=
```
QRMonDropData[{{0.507495, 0.754024}, {0.885643, 0.336476}}, ⟨|data → {{0.507495, 0.754024}, {0.885643, 0.336476}}|⟩]⟹QRMonTakeContext
```

(The “droppers” simply use the state monad function QRMonDropFromContext, [AAp1]. For example, QRMonDropData is equivalent to QRMonDropFromContext[“data”].)

Unit tests

The development of QRMon was done with two types of unit tests: (1) directly specified tests, [AAp7], and (2) tests based on randomly generated pipelines, [AA8].

The unit test package should be further extended in order to provide better coverage of the functionalities and illustrate -- and postulate -- pipeline behavior.

Directly specified tests

Here we run the unit tests file “MonadicQuantileRegression-Unit-Tests.wlt”, [AAp7]:

In[80]:=
```
AbsoluteTiming[
 testObject = TestReport["~/MathematicaForPrediction/UnitTests/MonadicQuantileRegression-Unit-Tests.wlt"]
]
```

Out[80]=
```
{1.00366, TestReportObject[
  Title: Test Report: MonadicQuantileRegression-Unit-Tests.wlt
  Success rate: 100%    Tests run: 20
  Succeeded: 20
  Failed: 0
  Failed with wrong results: 0
  Failed with messages: 0
  Failed with errors: 0
]}
```

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

In[81]:=
```
Values[Map[#["TestID"] &, testObject["TestResults"]]]
```

Out[81]=
```
{LoadPackage, GenerateData, QuantileRegression-1, QuantileRegression-2, QuantileRegression-3, QuantileRegression-and-Fit-1, Fit-and-QuantileRegression-1, QuantileRegressionFit-and-Fit-1,
 Fit-and-QuantileRegressionFit-1, Outliers-1, Outliers-2, GridSequence-1, BandsSequence-1, ConditionalCDF-1, Evaluate-1, Evaluate-2, Evaluate-3, Simulate-1, Simulate-2, Simulate-3}
```

Random pipelines tests

Since the monad QRMon is a DSL it is natural to test it with a large number of randomly generated “sentences” of that DSL. For the QRMon DSL the sentences are QRMon pipelines. The package “MonadicQuantileRegressionRandomPipelinesUnitTests.m”, [AAp8], has functions for generation of QRMon random pipelines and running them as verification tests. A short example follows.

Generate pipelines:

In[69]:=
```
SeedRandom[234]
pipelines = MakeQRMonRandomPipelines[100];
Length[pipelines]
```

Out[71]=
```
100
```

Here is a sample of the generated pipelines:

Out[72]=

#	pipeline
1	QRMon[tsData, <||>]⟹QRMonRescale[Axes → True]⟹QRMonEchoDataSummary⟹QRMonRescale[Axes → True]⟹QRMonQuantileRegressionFit[{1, x}, x]⟹QRMonGridSequence⟹QRMonTakeValue
2	QRMon[distData, <||>]⟹QRMonRescale[Axes → {True, False}]⟹QRMonRescale[Axes → True]⟹QRMonQuantileRegressionFit[{1, x}]⟹QRMonErrors[]⟹QRMonTakeRegressionFunctions
3	QRMon[tsData, <||>]⟹QRMonLeastSquaresFit[{1, x}, x]⟹QRMonGridSequence⟹QRMonTakeRegressionFunctions
4	QRMon[None, <||>]⟹QRMonDeleteMissing⟹QRMonQuantileRegressionFit[{1, x}]⟹QRMonErrors[]⟹QRMonTakeRegressionFunctions
5	QRMon[None, <||>]⟹QRMonEchoDataSummary⟹QRMonMovingMap[Mean, 0.0495906]⟹QRMonTakeValue
6	QRMon[tsData, <||>]⟹QRMonQuantileRegression[]⟹QRMonSimulate⟹QRMonTakeValue

Here we run the pipelines as unit tests:

In[74]:=
```
AbsoluteTiming[
 res = TestRunQRMonPipelines[pipelines, "Echo" → False];
]
```

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

In[75]:=
```
rpTRObj = TestReport[res]
```

Out[75]=
```
TestReportObject[ Title: Automatic
                  Success rate: 97%    Tests run: 100
                  Succeeded: 97
                  Failed: 3
                  Failed with wrong results: 0
                  Failed with messages: 3
                  Failed with errors: 0 ]
```

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

Future plans

Workflow operations

A list of possible, additional workflow operations and improvements follows.

- Certain improvements can be done over the specification of the different plot options.
- It will be useful to develop a function for automatic finding of over-fitting parameters.
- The time series simulation should be done by aggregation of similar time intervals.
  - For example, for time series with span several years, for each month name is made Quantile Regression simulation and the results are spliced to obtain a one year simulation.

Conversational agent

Using the packages [AAp10, AAp11] we can generate QRMon pipelines with natural commands. The plan is to develop and document those functionalities further.

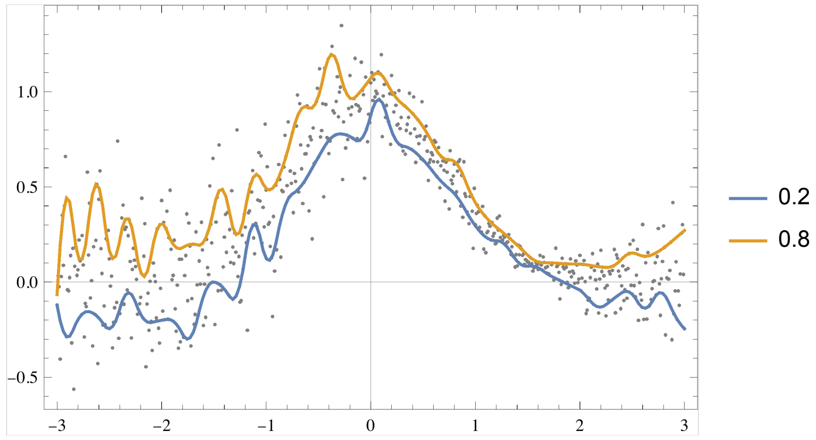
Here is an example of a pipeline constructed with natural language commands:

In[78]:=
```
QRMonUnit[distData]⟹
  ToQRMonPipelineFunction["show data summary"]⟹
  ToQRMonPipelineFunction["calculate quantile regression for quantiles 0.2, 0.8 and with 40 knots"]⟹
  ToQRMonPipelineFunction["plot"];
```

» Data summary:

1 column 1		2 column 2	
Min	-3.	Min	-0.562388
1st Qu	-1.5025	1st Qu	0.031954
Median	0.	Median	0.20604
Mean	8.86701×10^{-17}	Mean	0.300542
3rd Qu	1.5025	3rd Qu	0.605181
Max	3.	Max	1.34789

» Plot:



Implementation notes

The implementation methodology of the QRMon monad packages [AAp3, AAp8] followed the methodology created for the ClCon monad package [AAp9, AA6]. Similarly, this document closely follows the structure and exposition of the ClCon monad document “A monad for classification workflows”, [AA6].

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 Quantile Regression Mathematica package, (2018), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicQuantileRegression.m.

[AAp4] Anton Antonov, Quantile regression Mathematica package, (2014), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/QuantileRegression.m .

[AAp5] Anton Antonov, Monadic contextual classification Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicContextualClassification.m .

[AAp6] Anton Antonov, MathematicaForPrediction utilities, (2014), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MathematicaForPredictionUtilities.m .

[AAp7] Anton Antonov, Monadic Quantile Regression unit tests, (2018), MathematicaVsR at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/UnitTests/MonadicQuantileRegression-Unit-Tests.wlt .

[AAp8] Anton Antonov, Monadic Quantile Regression random pipelines Mathematica unit tests, (2018), MathematicaForPrediction at GitHub.

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

[AAp9] Anton Antonov, Monadic contextual classification Mathematica package, (2017), MathematicaForPrediction at GitHub.
URL: https://github.com/antononcube/MathematicaForPrediction/blob/master/MonadicProgramming/MonadicContextualClassification.m .

ConverationalAgents Packages

[AAp10] Anton Antonov, Time series workflows grammar in EBNF, (2018), ConversationalAgents at GitHub, https://github.com/antononcube/ConversationalAgents.

[AAp11] Anton Antonov, QRMon 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, "Quantile regression through linear programming", (2013), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2013/12/16/quantile-regression-through-linear-programming/ .

[AA3] Anton Antonov, "Quantile regression with B-splines", (2014), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2014/01/01/quantile-regression-with-b-splines/ .

[AA4] Anton Antonov, "Estimation of conditional density distributions", (2014), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2014/01/13/estimation-of-conditional-density-distributions/ .

[AA5] Anton Antonov, "Finding local extrema in noisy data using Quantile Regression", (2015), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2015/09/27/finding-local-extrema-in-noisy-data-using-quantile-regression/ .

[AA6] Anton Antonov, "A monad for classification workflows", (2018), MathematicaForPrediction at WordPress.
URL: https://mathematicaforprediction.wordpress.com/2018/05/15/a-monad-for-classification-workflows/ .

Other

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

[Wk2] Wikipedia entry, Quantile Regression,
URL: https://en.wikipedia.org/wiki/Quantile_regression .

[Wk3] Wikipedia entry, Chebyshev polynomials,
URL: https://en.wikipedia.org/wiki/Chebyshev_polynomials .

[CN1] Brian S. Code and Barry R. Noon, "A gentle introduction to quantile regression for ecologists", (2003). Frontiers in Ecology and the Environment. 1 (8): 412–420. doi:10.2307/3868138.
URL: http://www.econ.uiuc.edu/~roger/research/rq/QReco.pdf .

[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 .

[RG1] Roger Koenker, Quantile Regression, Cambridge University Press, 2005.