Table of Contents

1. Abstract 1.1
2. Introduction 1.2
3. Data sets 1.3
4. Variables, frames and data manipulation 1.4
5. Reading and writing data 1.5
6. Statistical tools 1.6
    1. Distributions 1.6.1
    2. Core statistical tools 1.6.2
    3. Hypothesis testing 1.6.3
    4. Correlations 1.6.4
    5. Samples and sampling techniques 1.6.5
7. Visualization 1.7
    1. Box plot 1.7.1
    2. Points (scatter plot) 1.7.2
    3. Histogram 1.7.3
    4. Density line 1.7.4
8. Getting started: Kaggle's Titanic Competition 1.8
    1. Get the data 1.8.1
    2. Simple models 1.8.2
    3. Tree model 1.8.3
    4. Random forest model 1.8.4
    5. Feature engineering 1.8.5
    6. SVM model 1.8.6
    7. Stacking classifiers 1.8.7

Abstract

Rapaio Manual of Usage

This book presents features of rapaio library. This library is a statistics, data mining and machine learning library written in Java. The presented material contains intuitions behind the features and explained and commented usage examples.

This book is a work in progress.

The project uses github and it's address is https://github.com/padreati/rapaio.

Please give some feedback by star or follow if you like the manual or you find interesting the manual. We would also love to drop some comments or suggestions on text.

Introduction

Introduction

Why another library for statistics and machine learning?

There are a lot of software stacks out there which provides plenty of nicely crafted tools for statistics, machine learning, data mining or pattern recognition. Many of them are available as open source, quality is high and they are full of reach features.

We have *R* which is a standard even for companies these days. *R* provides a language, an interactive data analysis workbench, solid visualization. *R* also incorporates the experience of many sound scientists which contributed to it.

There are Python stacks which benefits from the fact that Python is a simple and elegant language and works like a glue language like no other. We have *numpy*, *scipy*, *scikit-learn* and plenty of other things. There is also *matplotlib* which is beautiful and flexible. There is also *IPython* which gives to the interactive analysis a new dimension.

There are also some other tools with different purposes or principles, like *Weka*, *Shogun*, *Spark* with *MLib* and so on. Some of them are have a lot of implemented algorithms. Some of them leverage distributed computation. Some of them exploits GPUs or other types of grids. And so on.

That being said, it appears like a legitimate question "*Why another library for statistics and machine learning?*". Socrates said that "*understanding a question is half of an answer*". In our case the question would be complete if we append it with context. It now becomes: "*Why another library for statistics and machine learning, when there are many available already?*". A new question arise: why so many?. My answer is: because none of them covers the taste and needs of everybody.

Some reasons which provides motivation for this library are:

- I love *R*. I like functional paradigms, but I do not believe that the *R* language like it is now is something which I really want to program in for along time. Take for example the error handling.
- I love Python for it's simplicity and elegance. However there are two main things which I really do not like using libraries. If you want something to work with you have to have an almost complete operation system. That is only because Python is *really* good as glue language. This advantage comes to a price. But more than that I really hate things like *making a graph shown in a window of a given size can be done in too many ways, many of them undocumented and hackish*. And I really want a language with full and intelligent auto-complete, at any time. I prefer to memorize ideas that syntax.
- I like *Weka* for it's plenty of implementations. But I really do not like the standard Java way of doing things. There are no short methods. And by the way, not all implementations are complete (and the same thing can be said for Python stack).
- The last but not least, I would really love to have an environment, a box for with plenty of tools, which can be extended, which allows be to experiment, study and learn.

Prerequisites

Rapaio library is written in Java. Since this is a continuous work in progress the best way to use the

library is to have the source code and use it locally. Of course, this is not the only possibility. One can get published releases and use only the compiled artifacts. A list of published releases is on github.

As far as I understand one can commit code for a github project in two ways. The first one is to clone a project you like and the second one is to commit directly into a project which either you own or you collaborate on.

Either way you have to get a link from github to the project. In case of collaboration the direct link to rapaio project is `https://github.com/padreati/rapaio.git`. For a cloned repository you would get a link like `https://github.com/your_user_name/rapaio`, where *your_user_name* is your used name (sic!).

Install Intellij Idea

If you don't already have it, grab the "community edition" of IDEA from http://www.jetbrains.com/idea/download/. Extract the IntelliJ distribution somewhere convenient (e.g. under your home directory), and run `bin/idea.sh` to start the IDE. You should then see a series of step-by-step configuration dialog boxes where you can enable or disable plugins. The less plugins you have activated, the more resource-efficient IDEA will be. The following steps will give you a minimal IDEA setup. You can always enable more plugins later if you need them.

1. From the version control plugins you need only Git.
2. From "Other plugins" you need GitHub and JUnit.

Super tip: In recent versions of Ubuntu, it is not straightforward in the default GUI to manually add application launcher icons. IDEA has a menu item to automatically create one at `Tools -> Create desktop entry`.

Install JDK

This library requires jdk 1.8 or newer. For testing the project needs junit. Other dependencies are not needed. The main reason is that the first of the only two principles which governs this library is **Write yourself any feature you need, do not rely on external libraries**. As a consequence, you need only JDK at runtime, and for testing only junit. Any other thing is either a leverage of jdk (logging, image manipulation, graphics, etc) or written in library itself.

Setup rapaio from GitHub

1. Create a directory which will hold `rapaio` modules and perhaps other modules. Suppose this directory is `/local/workspace/` (but you can name it how you like it).

2. IDEA allows you to check out Git repositories from GitHub in two ways. You have already open idea with a project and you setup a new project for `rapaio` from there (in which case use menu `VCS` -> `Checkout from Version Control` -> `GitHub`). Otherwise you did not yet setup and opened a project yet (in which case from `Welcome to Intellij IDEA` dialog box, you use `Checkout from Version Control` -> `GitHub`).

3. In both cases you the `Login to GitHub` dialog box will appear and you'll have to complete `Host`, `Auth type`: Password, `Login` (often the email) and `Password` (often the password for github).

4. If you checked `Save password` than you are asked for a master password. With this password the IDE encrypts its own repository of passwords.

5. After successful log in, the `Clone repository` dialog box allows you to select a repository. The required one is `https://github.com/padreati/rapaio.git`. Select a `Parent directory` to be something like `/local/workspace`, select `Directory Name` as `rapaio` and push `Clone` button.

6. The IDE asks you if you want to create a project from sources. In my setup I select `No` since I want `rapaio` as a module. After selecting `No`, you will be back in the `Welcome to Intellij IDEA` screen.

7. Now it's time to create the project. Select `Create new project`. In the `New project` modal dialog select `Empty Project` from the right panel and click `Next` and from the new screen select as `Project location`: `/home/workspace`, and perhaps `Project Name`: `rapaio`, and push `Finish`.

8. IDE will create the new empty project for you (please remember that there are inside sources for rapaio) and open it. Because it's empty two things happens: it will show some warnings `Unregistered VCS root detected` which you should ignore it for now and it will show modal dialog `Project Structure` since the project is empty. In the `Project Structure` add a new module via `+` -> `New module`. In the `New module` screen a `Module SDK`, it has to be JDK 8 or greater (if none exists, than selects `New` -> `JDK` -> select JDK root -> `OK`). Then select `Next`. On the new modal dialog select a location for `Content root` to be `/local/workspaces/rapaio`, the other fields could remain the same, and click `Finish`.

9. Now it's time to solve `VCS root configuration problems`. For that you will have to click `Configure` link from notifications. From the modal dialog you have to solve 2 things: remove project from VCS (select project and click `-` button) and register `rapaio` under VCS (select `rapaio` module and click `+` button).

10. After all of this is done you can start to work. Final setups might need to change markings with right click and `Mark directory as` for src, tst or whatever. In order to experiment, you can create your own module, add `rapaio` as a module dependency and work with data.

Final note: please check if the sources for the project use JDK 8 or greater for language level. In case it is not properly set (compilation errors or similar), you would have to go to `Project Struncture` -> `Modules` -> `rapaio` -> `Sources` and set the field `Language level` to `8 - Lambdas, types annotations, etc`

Data sets

Data sets

For the purpose of this book some built-in data sets are used. Most of the built in data sets are available via `rapaio.datasets.Datasets` class. This is an utility class which provides some standard data sets used in many statistical and machine learning text books. There are a few which we will use for the purpose of this tutorial.

Iris data set

The **Iris flower data set** or **Fisher's Iris data set** is a multivariate data set introduced by Ronald Fisher in his 1936 paper *The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis*. It is sometimes called **Anderson's Iris data** set because Edgar Anderson collected the data to quantify the morphological variation of Iris flowers of three related species.

Two of the three species were collected in the Gaspé Peninsula *all from the same pasture, and picked on the same day and measured at the same time by the same person with the same apparatus.*

The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). There are for measures for each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

```
Frame iris = Datasets.loadIrisDataset();
iris.printSummary();
```

This is the summary of this data set:

```
Frame Summary
=============
* rowCount: 150
* complete: 150/150
* varCount: 5
* varNames:

 0. sepal-length : num | 2. petal-length : num | 4. class : nom |
 1.  sepal-width : num | 3.  petal-width : num |

   sepal-length      sepal-width     petal-length      petal-width
   Min. : 4.300     Min. : 2.000     Min. : 1.000     Min. : 0.100
1st Qu. : 5.100  1st Qu. : 2.800  1st Qu. : 1.600  1st Qu. : 0.300  
 Median : 5.800   Median : 3.000   Median : 4.350   Median : 1.300
   Mean : 5.843     Mean : 3.057     Mean : 3.758     Mean : 1.199
2nd Qu. : 6.400  2nd Qu. : 3.300  2nd Qu. : 5.100  2nd Qu. : 1.800
   Max. : 7.900     Max. : 4.400     Max. : 6.900     Max. : 2.500
```

Pearson's Height Data

This simple data set comes from a famous experiment by Karl Pearson around 1903. The number of cases is 1078. The original data values were rounded to produce heights to the nearest 0.1 inch.

```
Frame fs = Datasets.loadPearsonHeightDataset();
fs.printSummary();
```

The summary of the data set shows the two numeric variables with heights for fathers and sons.

```
Frame Summary
=============
* rowCount: 1078
* complete: 1078/1078
* varCount: 2
* varNames:

 0. Father : num | 1. Son : num |

          Father                Son
   Min. : 59.000     Min. : 58.500
1st Qu. : 65.800  1st Qu. : 66.900
 Median : 67.800   Median : 68.600
   Mean : 67.687     Mean : 68.684
2nd Qu. : 69.600  2nd Qu. : 70.500
   Max. : 75.400     Max. : 78.400
```

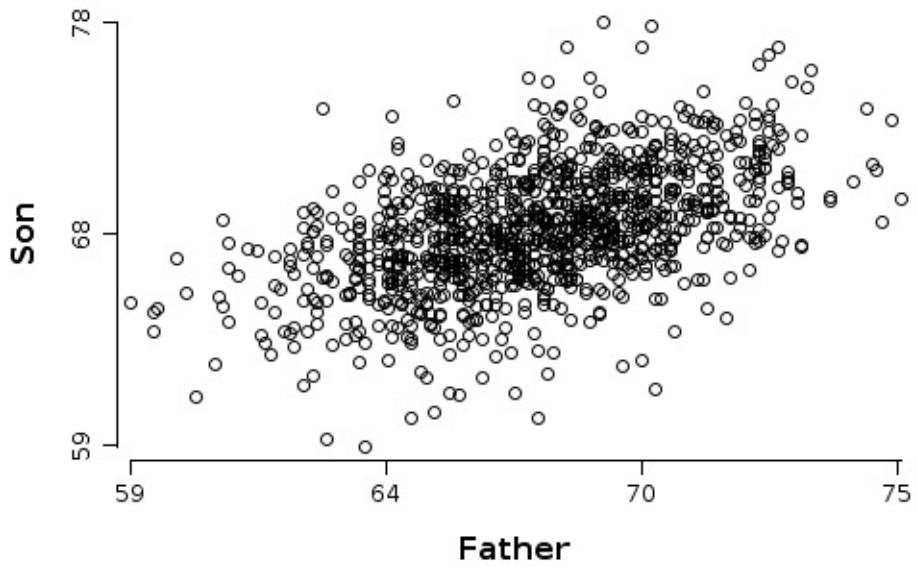


Figure 1.3.1 Classical father and son heights based on Pearson's original experiment from 1907

Variables, frames and data manipulation

Variables, frames and data manipulation

There are two main data structures used all over the place: variables and frames. A variable is a list of values. You can think of a variable as a column in a tabular data format. A set of variables is a data frame.

Let's take a simple example. We will load the iris data set, which is already contained in the library.

```
Frame df = Datasets.loadIrisDataset();
df.printSummary();
```

```
Frame Summary
=============
* rowCount: 150
* complete: 150/150
* varCount: 5
* varNames:

 0. sepal-length : num | 2. petal-length : num | 4. class : nom |
 1.  sepal-width : num | 3.  petal-width : num |

   sepal-length      sepal-width     petal-length      petal-width
   Min. : 4.300     Min. : 2.000     Min. : 1.000     Min. : 0.100
1st Qu. : 5.100  1st Qu. : 2.800  1st Qu. : 1.600  1st Qu. : 0.300
 Median : 5.800   Median : 3.000   Median : 4.350   Median : 1.300
   Mean : 5.843     Mean : 3.057     Mean : 3.758     Mean : 1.199
2nd Qu. : 6.400  2nd Qu. : 3.300  2nd Qu. : 5.100  2nd Qu. : 1.800
   Max. : 7.900     Max. : 4.400     Max. : 6.900     Max. : 2.500
```

Frame summary is a simple way to see some general information about a data frame. We see the data frame contains 150 observations. The data set contains five variables.

The listing continues with enumerating the name and type of the variables contained in a data frame. Notice that there are four numeric variables and one nominal variable, named *class*.

The summary listing ends with a section which describes each variable. For numeric variables the summary contains 6 number summary. These are some sample order statistics and sample mean. We have then the minimum and maximum value, the median value, the first and third quartile value and the mean value. For nominal values with have an enumeration of the first most frequent levels and the associated counts. For our *class* variable we see that there are three levels, each with 50 instances.

Reading and writing data

Reading and writing data

CSV - reading and writing

Reading and writing CSV files is an important feature for any system which works with data. The reason for its importance is the simplicity of the file format and its popularity.

Rapaio library offers support for both reading and writing operations. It has a lot of features and allows flexibility. However we read a file only into a data frame, and we write a csv file only from a data frame. This might look like a constraint in the beginning, but it comes natural since both are tabular data. The only difference is the fact that one operates in the memory of a program and the other one is persisted on disk.

Simple read/write data frames from/into Csv files

We can read a file with the default options simply by instantiating a `rapaio.io.Csv` object and calls one of `read` methods.

```
Frame iris = new Csv().read(Datasets.class, "iris-r.csv");
```

We select few rows and inspect what it is inside:

```
// use only few rows
iris = iris.mapRows(0, 1, 50, 51, 100, 101);
iris.printLines();
```

.. with the following output:

```
 sepal-length  sepal-width  petal-length  petal-width       class
     5.100000     3.500000      1.400000     0.200000      setosa
     4.900000     3.000000      1.400000     0.200000      setosa
     7.000000     3.200000      4.700000     1.400000  versicolor
     6.400000     3.200000      4.500000     1.500000  versicolor
     6.300000     3.300000      6.000000     2.500000   virginica
     5.800000     2.700000      5.100000     1.900000   virginica
```

Persisting a data frame into csv file format is also simple. We instantiate a `rapaio.io.Csv` object and call one of implementation of `write` methods:

```
new Csv().write(iris, "/tmp/iris.csv");
```

If we open the `/tmp/iris.csv` file with an editor, we can discover that it will have the following content:

```
sepal-length,sepal-width,petal-length,petal-width,class
5.1,3.5,1.4,0.2,setosa
4.9,3,1.4,0.2,setosa
7,3.2,4.7,1.4,versicolor
```

```
6.4,3.2,4.5,1.5,versicolor
6.3,3.3,6,2.5,virginica
5.8,2.7,5.1,1.9,virginica
```

Various read and write methods for Csv

Java has a nice abstraction over data named input and output streams. This is enough to make any tool to read or write data from anywhere. We followed that line of thinking by having

```
public Frame read(InputStream inputStream) throws IOException
public void write(Frame df, OutputStream os) throws IOException
```

Implemented on Csv class. With these two methods we basically can read from anywhere and can write to anywhere.

To simplify some common tasks there are some specialized forms of read and write:

- Read from a file giving a `File` instance
- Read from a file giving a `String` for path name
- Read from a gz archive `File` instance

- Read from a resource giving `Class` and `String` for class and name of the resource (this is useful when loading data from a loaded jar or for test)

- Write ...

Statistical tools

Core statistical tools

Rapaio library builds it's machine learning and data mining approach from a statistical perspective. As such, the library provides the following set of features:

- generate samples from a set of common distributions
- evaluate pdf, cdf, quantile for a common set of distributions
- standard hypothesis testing
- core statistical procedures on samples like mean and variance
- correlation coefficients computations

Distributions

Distributions

Statistics work with the concept or random variable. We can imagine a random variable like a process which generates numbers but we do not know precisely the generated value. This analogy works fine, even when we think at observing something. Let work that by an example.

Imagine that we measure the heights of some men from a given place. We notice and collect some values. We might ask which is the process that generated the heights? Since we measured something, but the values are out of our control, the question is legitimate. The answer is the total number of factors which influenced the height of a men, including genes, culture, climate, longitude and latitude, and so on. All those imaginable factors together can be seen as a process which determines the heights.

Now we identified which is the process. The next legitimate question is why random? Since there are a lot of factors inside the process, we usually are in the situation where we don't understand the underlying mechanics of the process. This can happen because we did not identified all the determinant factors. This can also happen simply because it is too complex. And there are also cases when we simply do not want or can't understand. Because of that we simply can't determine precisely the height of an individual men starting from known factors.

We can't do that for individual, but we can say something more when more individuals are analyzed. Statistics is about populations (the total set of observations) and samples (a part of the existent samples), not about individuals. Thus when we talk about an individual we talk through the lenses of population. And because it's not precise we name it random.

A random variable can be described in many ways, but all those formulations maps a potential interval of observable values with a probability. What we know when we have a random variable is a set of potential observable values and probability assigned with those values.

Normal distribution

Normal or Gaussian distribution is perhaps the most known distribution. It is the distribution of the *bell shape*, even thought it is not unique among bell shaped distributions.

The implementation of normal distribution lies in:`rapaio.core.distributions.Normal`.

```
// build a normal distribution with mean = 2, and standard deviation
Normal normal = new Normal(2,3);
```

Normal distribution defined by two independent parameters: mean and standard deviation. Mean tells us when is the center of the distribution and standard deviation is a measure of variability.

We can generate a random sample from this distribution.

```
RandomSource.setSeed(1);
Numeric x = normal.sample(100).withName("x");
x.printSummary();
```

```
> printSummary(var: x)
name: x
```

```
type: NUMERIC
rows: 100, complete: 100, missing: 0
   Min. : -5.722
1st Qu. : -0.441
 Median :  2.040
   Mean :  1.927
2nd Qu. :  3.680
   Max. : 11.254
```

We can notice that the sample mean is `1.927`, which is close to the population mean, but not equal. This is as expected. Since we generated a set of random values we can't expect to have a sample mean identical with population mean. If we expect that than we can suspect the generated numbers are not random.

```
WS.draw(funLine(normal::pdf).xLim(-8, 12).yLim(0, 0.15));
```

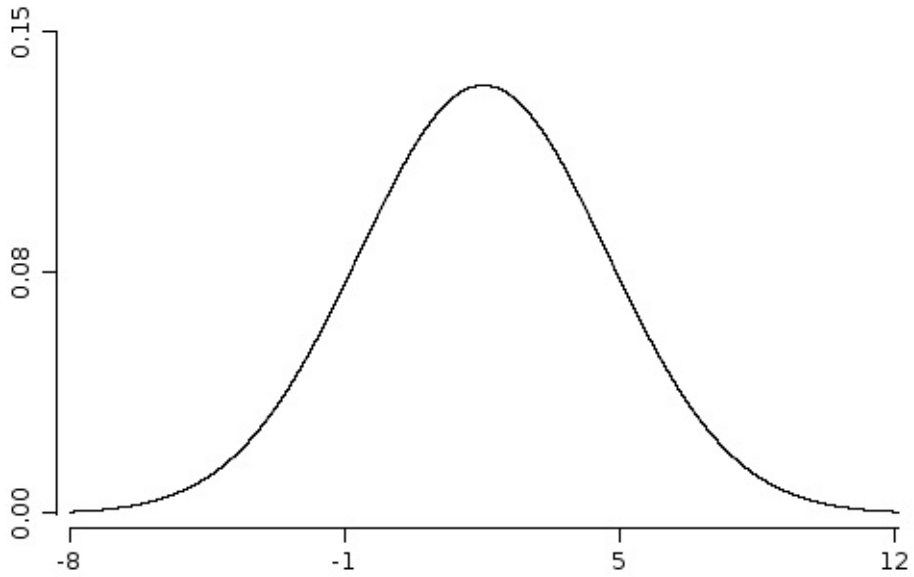


Figure 1.6.1.1 pdf of normal nistribution with mean=2 and sd=3

We draw now a histogram built from the generated sample, and the distribution which was used to generate values:

```
WS.draw(hist(x, prob(true), bins(20)).funLine(normal::pdf));
```

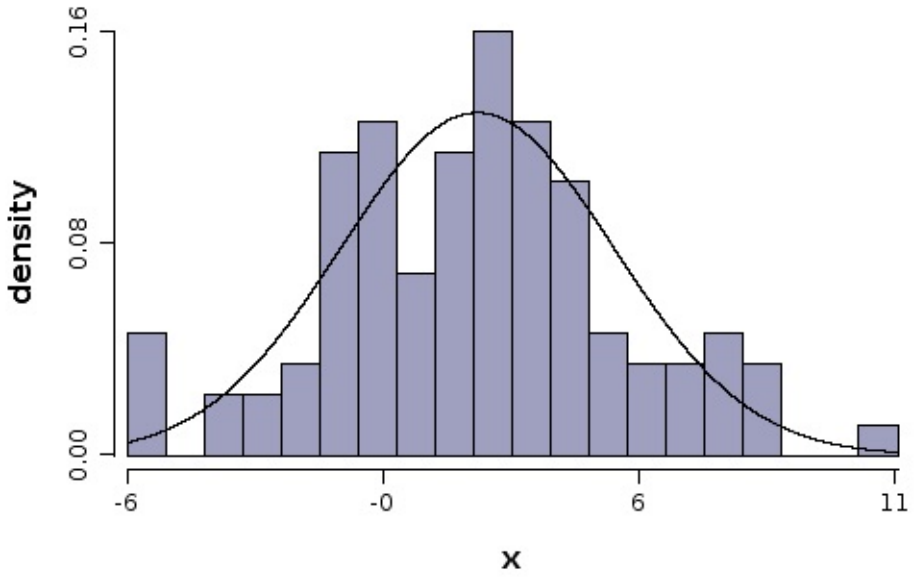


Figure 1.6.1.2 pdf of normal nistribution with mean=2 and sd=3

Core statistical tools

Hypothesis testing

Hypothesis testing

Rapaio library aims to contain an extensive set of alternatives for hypothesis testing. Right now there are available just some of them. Hypothesis testing an invaluable tool to answer questions when we are dealing with uncertainty.

We deal with presenting hypothesis testing by following some examples.

Z tests

Any hypothesis test which uses normal distribution for the computed statistic is named a z test. We should note that z tests needs to know standard deviations for the involved populations. It is accustomed that when the sample is large enough the value of the population standard deviation can be estimated from data. This is not implemented in library. For the case when one does not know the involved standard deviations one can use t tests.

Note than because of this requirement, the z tests are rarely used. This is so because we rarely know the population parameters.

Example 1: One sample z-test

Sue is in charge of Quality Control at a bottling facility[1]*. She is checking the operation of a machine that should deliver 355 mL of liquid into an aluminum can. If the machine delivers too little, then the local Regulatory Agency may fine the company. If the machine delivers too much, then the company may lose money. For these reasons, Sue is looking for any evidence that the amount delivered by the machine is different from 355 mL.*

During her investigation, Sue obtains a random sample of 10 cans. She measures the following volumes:

```
355.02 355.47 353.01 355.93 356.66 355.98 353.74 354.96 353.81 355.7
```

The machine's specifications claim that the amount of liquid delivered varies according to a normal distribution, with mean μ = 355 mL and standard deviation σ = 0.05 mL.

Do the data suggest that the machine is operating correctly?

The null hypothesis is that the machine is operating according to its specifications; thus

$H_0 : \mu = 355$ (μ is the mean volume delivered by the machine)

Sue is looking for evidence of any difference; thus, the alternate hypothesis is

$H_1 : \mu \neq 355$

Since the hypothesis concerns a single population mean and the population follows a normal distribution with known standard deviation, a z-test is appropriate.

What we can do is to use *HTTools* facility which offers shortcut methods to all implemented

hypothesis tests. One of them is one sample z test which enables one to test if the sample mean is far from the expected sample mean.

```
// build the sample
Var cans = Numeric.copy(355.02, 355.47, 353.01, 355.93, 356.66, 355.9
// run the test and print results
HTTools.zTestOneSample(cans, 355, 0.05).printSummary();
```

```
> HTTools.zTestOneSample

 One Sample z-test

mean: 355
sd: 0.05
significance level: 0.05
alternative hypothesis: two tails P > |z|

sample size: 10
sample mean: 355.037
z score: 2.3400855
p-value: 0.019279327322640594
conf int: [355.0060102,355.0679898]
```

The interpretation of the results is the following:

- the z-score is 2.34, which means that the computed sample mean is greater with more than 2 standard deviations
- for critical level being 0.05 and p-value 0.019, we reject the null hypothesis that the mean volume delivered by the machine is equal with 355

Note: even if we know that the sample mean is greater than the proposed mean, we cannot propose this conclusion. The proper conclusion would be that *is different than* 355.

What if we ask if the machine produces more than standard specification?

We deal with this question by changing the null hypothesis. Our hypotheses become:

$$H_0 : \mu \leq 355$$

$$H_1 \geq 355$$

Our code looks like:

```
HTTools.zTestOneSample(cans,
  355, \\ mean
  0.05, \\ sd
  0.05, \\ significance level
  HTTools.Alternative.GREATER_THAN \\ alternative
).printSummary();
```

```
> HTTools.zTestOneSample

 One Sample z-test

mean: 355
```

```
sd: 0.05
significance level: 0.05
alternative hypothesis: one tail P > z

sample size: 10
sample mean: 355.037
z score: 2.3400855
p-value: 0.009639663661320297
conf int: [355.0060102,355.0679898]
```

As expected the statistical power of this test is increased. As a consequence the p value was smaller and we still reject the null hypothesis. In this case we had an obvious case, when testing one side gave the same result as testing with two sides. I gave example just to help the user to pay attention to those kind of details.

1: This example is adapted from here.

Example 2: One sample z-test

A herd of 1500 *steer was fed a special high-protein grain for a month. A random sample of* 29 *were weighed and had gained an average of* 6.7 *pounds. If the standard deviation of weight gain for the entire herd is* 7.1*, test the hypothesis that the average weight gain per steer for the month was more than* 5 *pounds.*[2]

We have the following null and alternative hypothesis:

$$H_0 : \mu = 5 \; H_1 : \mu > 5$$

```
ZTestOneSample ztest = HTTools.zTestOneSample(
  6.7, // sample mean
  29, // sample size
  5, // tested mean
  7.1, // population standard deviation
  0.05, // significance level
  HTTools.Alternative.GREATER_THAN // alternative
);
ztest.printSummary();

> HTTools.zTestOneSample

 One Sample z-test

mean: 5
sd: 7.1
significance level: 0.05
alternative hypothesis: one tail P > z

sample size: 29
sample mean: 6.7
z score: 1.2894057
p-value: 0.0986285477062051
conf int: [4.1159112,9.2840888]
```

P-value is greater than significance level which means that we cannot reject the null hypothesis. We don't have enough evidence.

2: This example is taken from here.

Example 3: Two samples z test

The amount of a certain trace element in blood is known to vary with a standard deviation of 14.1 *ppm (parts per million) for male blood donors and* 9.5 *ppm for female donors. Random samples of* 75 *male and* 50 *female donors yield concentration means of* 28 *and* 33 *ppm, respectively. What is the likelihood that the population means of concentrations of the element are the same for men and women?*[3]

According with central limit theorem we can assume that the distribution of the sample mean is a normal distribution. More than that, since we have random samples, than the sample mean difference has a normal distribution. And because we know the standard deviation for each population, we can use a two sample z test for testing the difference of the sample means.

$$H_0 : \mu_1 - \mu_2 = 0 \; H_1 : \mu_1 - \mu_2 \neq 0$$

```
HTTools.zTestTwoSamples(
                28, 75, // male sample mean and size
                33, 50, // female sample mean and size
                0, // difference of means
                14.1, 9.5, // standard deviations
                ).printSummary();
```

```
> HTTools.zTestTwoSamples

 Two Samples z-test

x sample mean: 28
x sample size: 75
y sample mean: 33
y sample size: 50
mean: 0
x sd: 14.1
y sd: 9.5
significance level: 0.05
alternative hypothesis: two tails P > |z|

sample mean: -5
z score: -2.3686842
p-value: 0.017851489594360337
conf int: [-9.1372421,-0.8627579]
```

The test run with 0.05 significance level (because it was a default value). The alternative is two tails since we test for difference in means not equal with zero. The resulted p-value is lower than the significance value which means that we reject the hypothesis that the two populations have the same mean. We can see that also from confidence interval, since it does not include 0.

Note: If we would considered a significance level of 0.01 than we would not be able to reject the null hypothesis.

3: This example is taken from here.

T tests

T tests are similar with z tests. Both tests uses a statistic which has a normal distribution. A z test is used when one knows standard deviations. A t test is used when the standard deviation is estimated from data. Note that when the sample size is large enough, both tests gives virtually identical results, since the t distributions converges to a normal distribution when the sample increases.

Correlations

Samples and sampling techniques

Visualization

Data visualization

The purpose of this document is not to exhaust the graphics features of the library and graphics components which are already built in. There are good reasons why such a goal would be inappropriate:

- the library it's still in working phase, even there are many components ready for production use; any careful enumeration would be incomplete as soon as another graphical features would be added
- stressing the whole features even for a single component would be time consuming since working with all the combinations of graphical aspect options would take too much space to time for everybody

The goal of this document is to illustrate the core ideas behind the graphic system and to provide enough examples to have a fast and productive feedback.

The design of the graphical components of this library is influenced by many ideas from existing popular graphical systems. Among the main inspirations there are some which deserves appropriate consideration: `R` standard graphical library, `ggplot2` package and `matplotlib` from Python stack.

Brief overview

Drawing of graphical figures or text output is handled by classes which implements interface `rapaio.printer.Printer`. There are some built-in implementations of this interface. The default one is `StandardPrinter`, which allows printing text at console and drawing graphics in a Swing modal window. Another used implementation is `IdeaPrinter`, which is similar with standard printer for text output, but for graphics drawing implements a TCP/IP protocol with serialization which allows plug-ins like `rapaio-studio` to handle graphics drawing.

A printer can be instantiated by itself by a handy way of using it is by the means of the utility class called `rapaio.sys.WS` (which comes from **W**ork **W**pace). This is an utility class which offers static shortcut methods yo allow one to simulate a working session. This class has an instance of a printer implementation (which can be changes by `WS.setPrinter` method) and shortcuts for text output and drawing.

We are interested here in methods for drawing, so the workspace contains methods for this purpose. All of those methods works with a construct which contains all the indications regarding what kind of graphic should be built. This construct implements interface `rapaio.graphics.base.Figure`. Thus a figure is a construct which allows the printer system to obtain images which can be later drawn over graphical surfaces. Also, there are methods which allows one to obtain an image directly from a figure, or store an image directly on a disk file.

Here are the shortcut methods implemented in `WS` utility:

- *void draw(Figure figure)* - draws a figure in a printing system; the size of drawing is either adaptive of has the default size of a graphical image contained in the printer implementation.

- *void draw(Figure figure, int width, int height)* - draws a figure with dimensions specified by width and height
- *BufferedImage buildImage(Figure figure)* - builds an image from a figure with dimensions specified as default width and default height from printer system
- *BufferedImage buildImage(Figure figure, int width, int height)* - builds an image from a figure with specified dimensions
- *void saveImage(Figure figure, int width, int height, String fileName)* - builds an image with specified dimensions by width and height from a given figure, and save the image into a *png* file with the specified file name
- *void saveImage(Figure figure, int width, int height, OutputStream os)* - builds an image with specified dimensions by width and height, and send the image in *png* format to the specified output stream (could be a file or could be sent over network)

Now in order to build figures one can instantiate and compose figures directly in a pure Java way. Here is an example:

```
Plot plot = new Plot();
plot.add(new Histogram(iris.var("sepal-length"), 0, 10,
    bins(40), color(10), prob(true)));
plot.add(new DensityLine(iris.var("sepal-length"), lwd(2), color(2))
WS.draw(plot);
```

It is simple enough and structured. However graphical tools offers also a class named `rapaio.graphics.Plotter` which has a lot of shortcut methods to help one to simplify the above code. Here is an example:

```
WS.draw(hist(iris.var("sepal-length"), 0, 10, bins(40), color(10), pr
        .densityLine(iris.var("sepal-length"), lwd(2), color(2)));
```

Box plot

Box plots

A box plot is a standard way of displaying information about the distribution of a continuous data variable based on five data summary. The five data summary consists of: *minimum*, *first quartile*, *median* (*second quartile*), *third quartile* and *maximum* number summaries.

Box plots might be constructed in different manners by different authors. The common characteristics for all types of box-plots is *the box*.

Which means that in all cases the bottom margin of the box lies on first quartile, the top margin of the box lies on third quartile and the line inside the box lies on median values.

The extensions which comes from the box might differ and `rapaio` system implements the version which it's usually named: **Tuckey's box plot**. What is specific to this box plot is that the whiskers lies at the datum still within 1.5 IQR (*Interquartile range*).

Any other points above or below whiskers are outliers. Outliers are of two different types:

- *extreme outliers* - outliers which are at a distance greater or equal than 3*IQR
- *outliers* - outliers which are at a distance greater or equal than 1.5*IQR

Example 1

Scope: *Draw one box plot for each numerical variable from iris data set. We want each box plot to have a different color and we want some de-saturated colors.*

Solution:

```
WS.draw(boxPlot(iris.mapVars("0~3"), color(1, 2, 3, 4), alpha(0.5f))
```

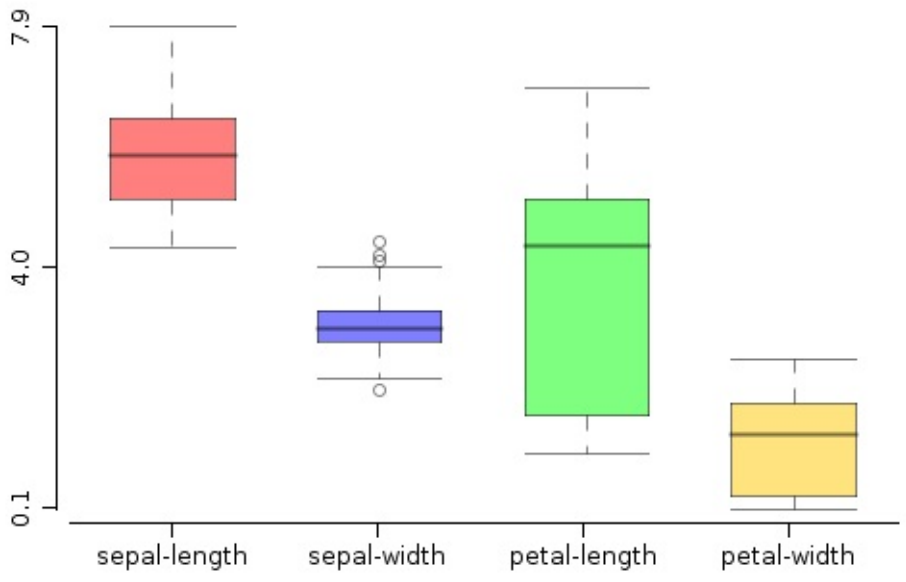


Figure 1.7.1.1 One box plot for each numerical variable

- `iris.mapVars("0~3")` - obtain a data set from `iris` data set, by keeping only the first 4 variables. We do that using the range notation (index of the start variable, concatenation symbol ~, index of the last variable inclusive). Pay attention that variable indexes are 0 based.
- `color(1, 2, 3, 4)` - we use colors from the current color palette, indexed with the specified integer values.
- `alpha(0.3f)` - we de-saturate the drawing keeping only 0.3 of the actual color.

Example 2

Scope: *In order to identify the overlap between values of `sepal-length` variable from iris data set, we draw one box plot for each segment of the nominal `class` variable, and add a title*

Solution:

```
WS.draw(boxPlot(iris.var("sepal-length"), iris.var("class"))
    .title("sepal-length separation"));
```

- `iris.var("sepal-length")` - is the variable named `sepal-length` from
- `iris` - data set, which is the numerical variable to be segmented and later box-plotted
- `iris.var("class")` - specifies the segment discriminator, depending on the levels of this variable, the same number of levels will be created
- `.title("..")` - adds a title to the box plot

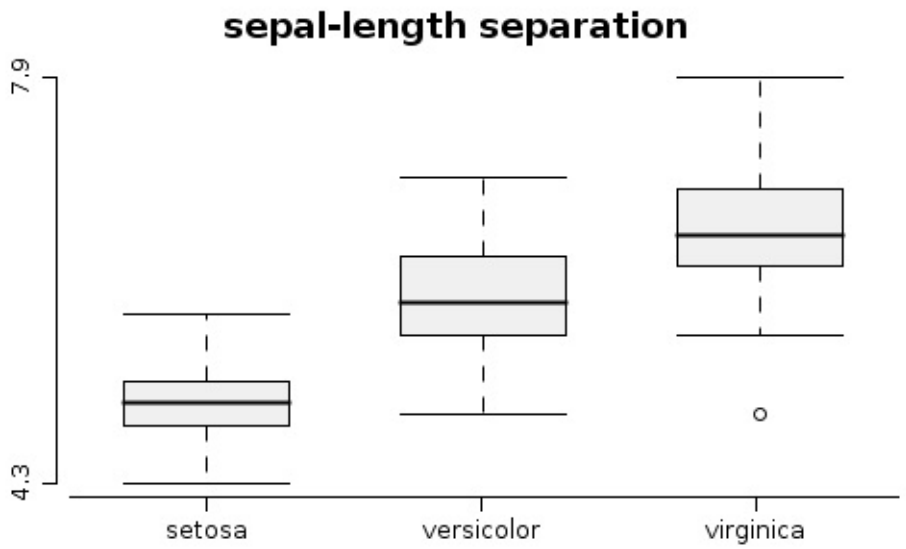


Figure 1.7.1.2 Box plots for the "sepal-length" variable, but discriminated by class

Box plot

Points (scatter plot)

Points (XY Scatter plot)

We can study the relation between two numerical variables by drawing one point for each instance.

Example 1

Scope: *Study which is the relation between* `petal-length` *and* `sepal-length` *from* `iris` *data set.*

Solution:

```
WS.draw(points(iris.var("petal-length"), iris.var("petal-width")));
```

- `iris.var("petal-length")` - variable used to define horizontal axis
- `iris.var("petal-width")` - variable used to define vertical axis

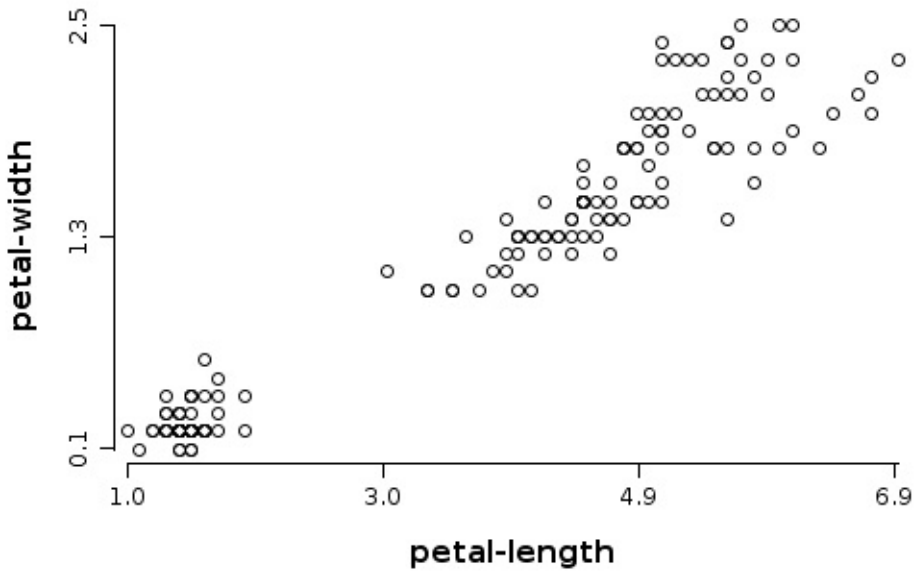


Figure 1.7.2.1 Scatter XY points for `petal-length` and `petal-width` variables

Example 2

Scope: *Study which is the relation between* `petal-length` *and* `sepal-length` *from* `iris` *data set. Color each point with a different color corresponding with value from variable* `class` *and add a legend for colors.*

Solution:

```
WS.draw(points(iris.var("petal-length"), iris.var("petal-width"),
    color(iris.var("class")), pch(2))
```

```
.legend(1.5, 2.2, labels("setosa", "versicolor", "virginica"
```

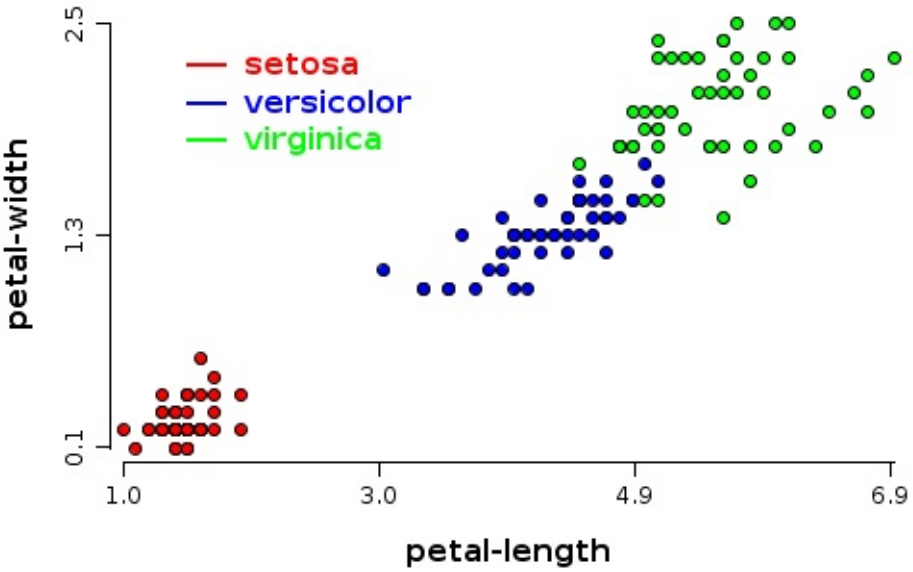


Figure 1.7.2.2 Scatter XY points for `petal-length` and `petal-width` variables with colors and legend

- `iris.var("petal-length")` - variable used to define horizontal axis
- `iris.var("petal-width")` - variable used to define vertical axis
- `color(iris.var("class"))` - nominal variable which provides indexes to select colors from current palette
- `pch(2)` - select the type of figure used to draw points (in this case is a circle filled with solid color and a black border)
- `legend(1.5, 2.2, labels("setosa", "versicolor", "virginica"))` - adds a legend at the specific position specified in the data range; labels are specified by parameter, colors are taken as default from current palette starting with 1

Histogram

Histogram

A histogram is a graphical representation of the distribution of a continuous variable.

The histogram is only an estimation of the distribution. To construct a histogram you have to *bin* the range of values from the variable in a sequence of equal length intervals, and later on counting the values from each bin. Histograms can display counts, or can display proportions which are counts divided by the total number of values.

Because the histogram uses bins that the main parameter of a histogram is the bin width. The bin's width is computed. To compute the width of a bin we need the number of bins and the minimum and maximum from the range of values. The range of values can be computed automatically from data or it can be specified when the histogram is built.

We can omit the number of bins, in which case its value is estimated also from data. For estimation is used the Freedman-Diaconis rule. See Freedman-Diaconis wikipedia page for more details.

Example 1

Scope: *Build a histogram with default values to estimate the pdf of* `sepal-length` *variable from* `iris` *data set.*

Solution:

```
WS.draw(hist(iris.var("sepal-length")));
```

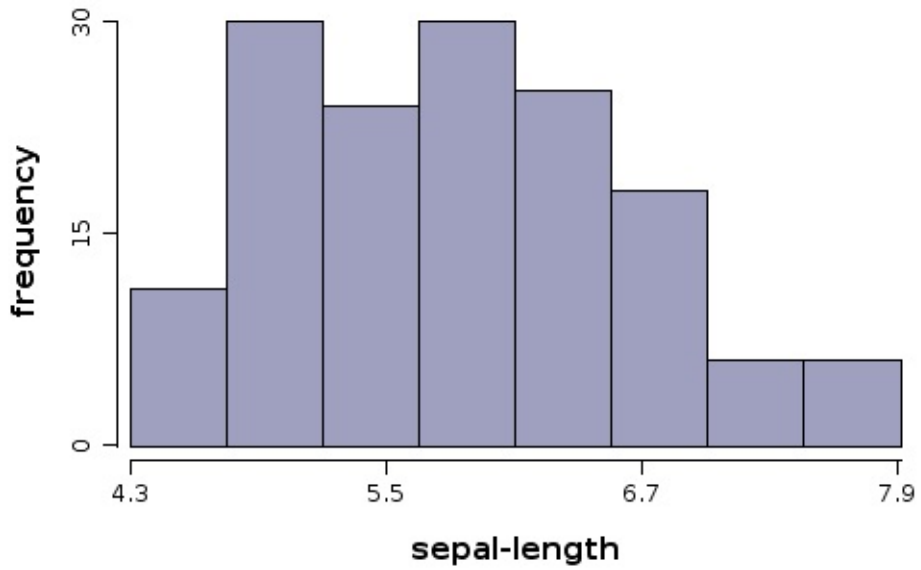


Figure 1.7.3.1 Histogram of `sepal-length` variable from iris data set

Example 2

Scope: *Build two overlapped histograms with default values to estimate the pdf of* `sepal-length` *and* `petal-length` *variables from* `iris` *data set. We want to get bins in range (0-10) of width 0.25, colored with red, and blue, with a big transparency for visibility*

Solution:

```
WS.draw(plot(alpha(0.3f))
    .hist(iris.var("sepal-length"), 0, 10, bins(40), color(1))
    .hist(iris.var("petal-length"), 0, 10, bins(40), color(2))
    .legend(7, 20, labels("sepal-length", "petal-length"), color(1, 2
    .xLab("variable"));
```

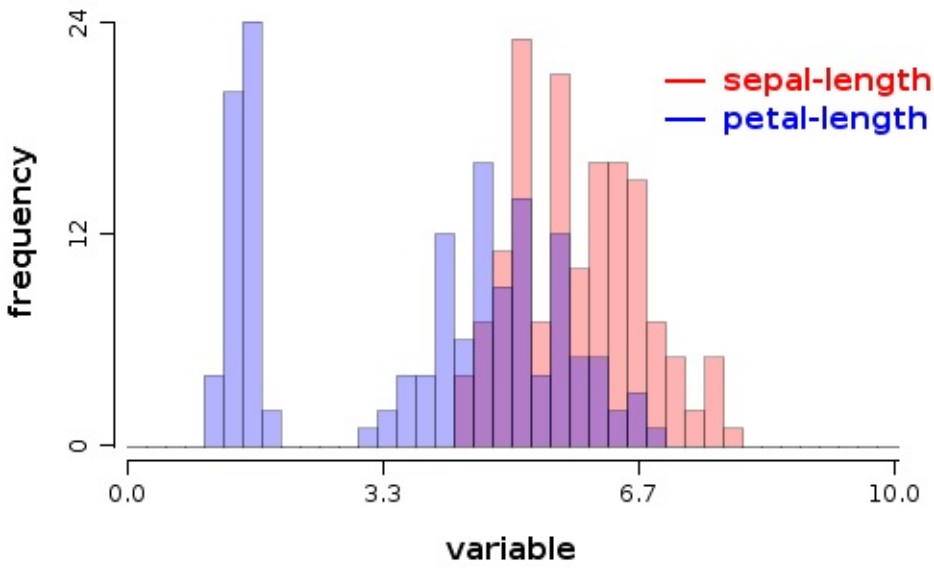


Figure 1.7.3.2 Histogram of `sepal-length`, `petal-length` variable from iris data set

- `plot(alpha(0.3f))` - builds an empty plot; this is used only to pass default values for alpha for all plot components, otherwise the plot construct would not be needed
- `hist` - adds a histogram to the current plot
- `iris.var("sepal-length")` - variable used to build histogram
- `0, 10` - specifies the range used to compute bins
- `bins(40)` - specifies the number of bins for histogram
- `color(1)` - specifies the color to draw the histogram, which is the color indexed with 1 in color palette (in this case is red)
- `legend(7, 20, ...)` - draws a legend at the specified coordinates, values are in the units specified by data
- `labels(..)` - specifies labels for legend
- `color(1, 2)` - specifies color for legend

- `xLab` = specifies label text for horizontal axis

Density line

Density line

A density line is a graphical representation of the distribution of a continuous variable. The density line is similar with a histogram in purpose but it has a different strategy to build the estimate. The density line plot component implements a *kernel density estimator* which is basically a non-parametric smoothing method, named also Parzen-Rosenblatt window method.

There are two main parameters for a density line: bandwidth and base density kernel function. The default bandwidth is computed according with Silverman's rule of thumb (more details on Wikipedia kernel density page). The default kernel function is the Gaussian pdf.

Kernel function estimators can be constructed using various kernel functions like Gaussian, uniform, triangular, Epanechnikov, cosine, tricube, triweight, biweight. All of them are available in `rapaio` library and also some custom can be built.

Example 1

Scope: *Illustrate the process of building the KDE estimation*

Solution:

```
// this is our sample
Numeric x = Numeric.wrap(-2.1, -1.3, -0.4, 1.9, 5.1, 6.2);

// declare a bandwidth for smoothing
double bw = 1.25;

// build a density line
Plot p = densityLine(x, bw);

// for each point draw a normal distribution
x.stream().forEach(s -> p.funLine(xi ->
    new Normal(s.value(), bw).pdf(xi) / x.rowCount(),
    color(1)));
WS.draw(p);
```

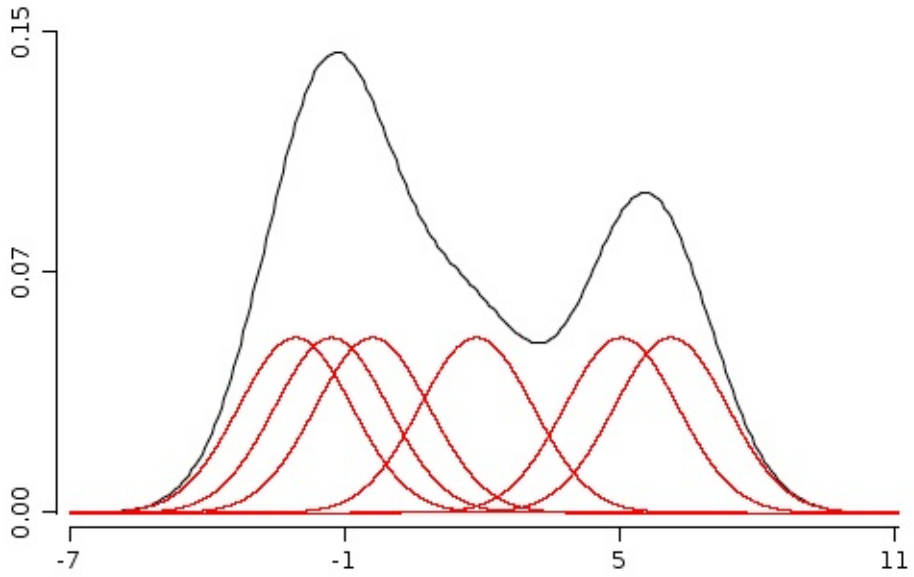


Figure 1.7.4.1 KDE construction from a sample of points

With red are depicted the kernel functions used to spread probability around sample points. With black is depicted the kernel density estimation which is the sum of all individual kernel functions.

Example 2

Scope: *Estimate density of iris* `sepal-length` *variable by histogram and density function*

Solution:

```
WS.draw(hist(iris.var("sepal-length"), prob(true))
    .densityLine(iris.var("sepal-length")));
```

- `hist(..)` - builds a histogram
- `iris.var("sepal-length")` - variable used to build histogram and also to build the density line
- `prob(true)` - parameter which specifies to a histogram to use probabilities (approximated by frequency ratios)
- `densityLine(..)` - builds a density line

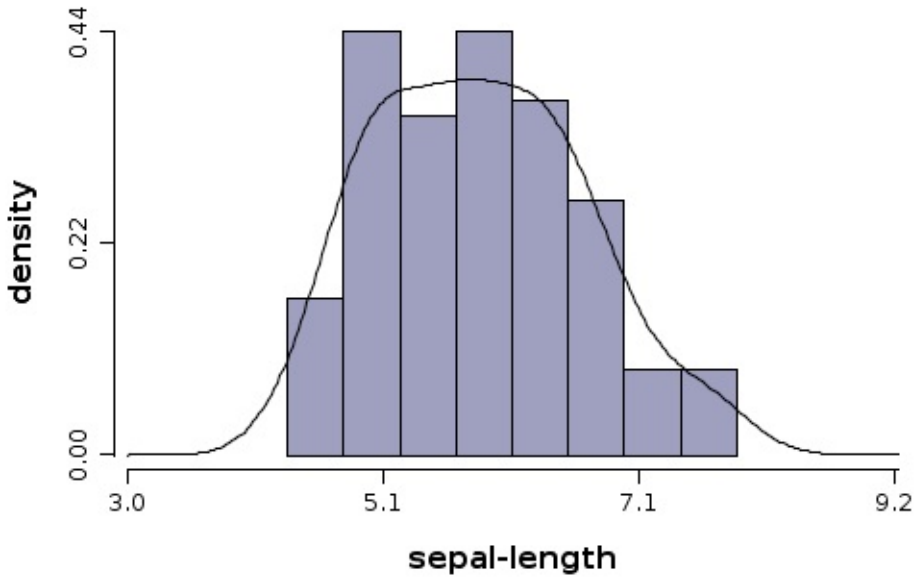


Figure 1.7.4.2 Density function and histogram estimation

Example 3

Scope: *Build multiple kernel density estimates for various bandwidth values*

Solution:

```
Plot p2 = plot();
DoubleStream.iterate(0.05, xi -> xi + 0.02)
    .limit(20)
    .forEach(v -> p2.densityLine(iris.var("sepal-length"), v));
WS.draw(p2);
```

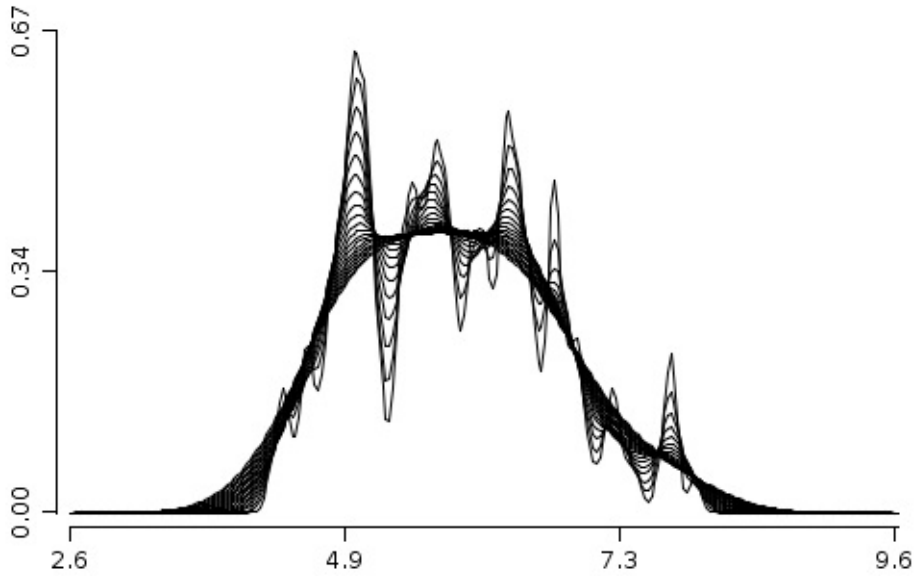


Figure 1.7.4.3 Multiple kernel density estimates with various bandwidth parameter values

Density line

Getting started: Kaggle's Titanic Competition

Getting started: Kaggle's Titanic Competition

Kaggle is already established as the best place which hosts machine learning competitions. If you do not know it already, then it's time to do it.

Titanic Competition is perhaps the first competition which one should try. Of course, if you are already an experienced data scientist, than you can skip to an advanced competition.

The purpose of the competition is to learn if a passenger has survived or not. We illustrate some steps and ideas one can apply to compete in this learning competition using the available tools one can find in rapaio library.

Get the data

Getting started: Kaggle's Titanic Competition

Get the data

The purpose of the competition is to predict which passengers have survived or not. The available data has two parts. The first part consists in a data set which contains what happened with some passengers and some related information like sex, cabin, age, class, etc. This data set contains information regarding their survival. The purpose why this data set contains survival data is because it will be used to train a model which learns how to decide if a passenger survives or not. This is the `train.csv`. The other file is a data set which contains data about another set of passenger, this time without knowing if they survived or not. They contain, however an identification number. This data set is `test.csv` and this is used to make predictions. Those predictions should be similar with the provided `gendermodel.csv`.

We also have to take a look of the data description provided on contest dedicated page:

```
VARIABLE DESCRIPTIONS:
survival        Survival
                (0 = No; 1 = Yes)
pclass          Passenger Class
                (1 = 1st; 2 = 2nd; 3 = 3rd)
name            Name
sex             Sex
age             Age
sibsp           Number of Siblings/Spouses Aboard
parch           Number of Parents/Children Aboard
ticket          Ticket Number
fare            Passenger Fare
cabin           Cabin
embarked        Port of Embarkation
                (C = Cherbourg; Q = Queenstown; S = Southampton)

SPECIAL NOTES:
Pclass is a proxy for socio-economic status (SES)
 1st ~ Upper; 2nd ~ Middle; 3rd ~ Lower

Age is in Years; Fractional if Age less than One (1)
 If the Age is Estimated, it is in the form xx.5

With respect to the family relation variables (i.e. sibsp and parch)
some relations were ignored.  The following are the definitions used
for sibsp and parch.

Sibling:  Brother, Sister, Stepbrother, or Stepsister of Passenger Al
Spouse:   Husband or Wife of Passenger Aboard Titanic (Mistresses an
Parent:   Mother or Father of Passenger Aboard Titanic
Child:    Son, Daughter, Stepson, or Stepdaughter of Passenger Aboar
```

```
Other family relatives excluded from this study include cousins,
nephews/nieces, aunts/uncles, and in-laws.  Some children travelled
only with a nanny, therefore parch=0 for them.  As well, some
travelled with very close friends or neighbors in a village, however,
the definitions do not support such relations.
```

The first step in our adventure is to download those 3 data file in *csv* format. You can do it from data section of the competition. Let's suppose you downloaded somewhere in a local folder. We will name this folder `data` folder, and actually it can have any name you would like.

Read train data from csv file

Because the data is small we can load the whole data in memory with no problems.

Let's see how we can load the data into memory. In rapaio the sets of data are loaded into the form of *frames* (`rapaio.data.Frame`). A frame is basically a tabular data, with columns for each variable (feature) and rows for each instance (in our case for each passenger).

A first try of loading the train data set and see what has happened is the following:

```
new Csv().read(root + "train.csv").printSummary();
```

What happened is that the csv reader was instantiated, a frame was loaded from the `train.csv` file, an instance of data frame was created and a method was called to see a print summary of the loaded frame.

```
Frame Summary
=============
* rowCount: 891
* complete: 889/891
* varCount: 12
* varNames:

 0. PassengerId : idx | 4.   Sex : nom |  8.   Ticket : nom |
 1.    Survived : bin | 5.   Age : nom |  9.     Fare : num |
 2.      Pclass : idx | 6. SibSp : idx | 10.    Cabin : nom |
 3.        Name : nom | 7. Parch : idx | 11. Embarked : nom |

      PassengerId    Survived          Pclass
   Min. :   1.000     0 : 549     Min. : 1.000
1st Qu. : 223.500     1 : 342  1st Qu. : 2.000
 Median : 446.000  NA's :   0   Median : 3.000
   Mean : 446.000                Mean : 2.309
2nd Qu. : 668.500             2nd Qu. : 3.000
   Max. : 891.000                Max. : 3.000

                                                 Name
                            "Braund, Mr. Owen Harris" :   1    male
"Cumings, Mrs. John Bradley (Florence Briggs Thayer)" :   1  female
                             "Heikkinen, Miss. Laina" :   1
       "Futrelle, Mrs. Jacques Heath (Lily May Peel)" :   1
                         "Allen, Mr. William Henry" :   1
```

```
                                  "Moran, Mr. James" :   1
                                            (Other) : 885
          SibSp            Parch           Ticket                Fare
   Min.   :0.000     Min.   :0.000    347082 :   7     Min.   :  0.000
1st Qu.   :0.000  1st Qu.   :0.000      1601 :   7  1st Qu.   :  7.910
 Median   :0.000   Median   :0.000  CA. 2343 :   7   Median   : 14.454
   Mean   :0.523     Mean   :0.382   3101295 :   6     Mean   : 32.204
2nd Qu.   :1.000  2nd Qu.   :0.000   CA 2144 :   6  2nd Qu.   : 31.000
   Max.   :8.000     Max.   :6.000    347088 :   6     Max.   :512.329
                                     (Other) : 852
  Embarked
   S : 644
   C : 168
   Q :  77
NA's :   2
```

How can we interpret the output of the frame's summary?

- We loaded a frame which has 891 rows and 12 columns (variables)
- From all the rows, 889 are complete (non missing data)
- The name of the variables are listed, together with their types
- It follows a data summary for the frame: 6 number summary for numeric variables, most frequent levels for nominal variables

Let's inspect each variable and see how it fits our needs.

PassengedId

The type for this variable is index (integer values). This field looks like an identifier for the passenger, so from our point of view the sorting is not required. What we can do, but is not required, is to change the field type to nominal. Anyway, we do not need this field for learning since it should be unique for each instance, thus the predictive power is null. We will ignore it for now since we will not consider it for learning

Survived

This is our target variable. It is parsed as binary, but since we do classification, we will change it's type to nominal. We do that directly from the csv parsing, by indicating that we want Survived parsed as nominal variable:

```
new Csv()
        .withTypes(VarType.NOMINAL, "Survived")
        .read(root + "train.csv")
        .printSummary();
```

The output becomes

```
Frame Summary
=============
* rowCount: 891
* complete: 889/891
* varCount: 12
* varNames:

 0. PassengerId : idx | 4.   Sex : nom |  8.   Ticket : nom |
```

```
 1.    Survived : nom | 5.   Age : nom |  9.     Fare : num |
 2.      Pclass : idx | 6. SibSp : idx | 10.    Cabin : nom |
 3.        Name : nom | 7. Parch : idx | 11. Embarked : nom |

      PassengerId  Survived           Pclass
   Min. :   1.000   0 : 549     Min. : 1.000
1st Qu. : 223.500   1 : 342  1st Qu. : 2.000
 Median : 446.000             Median : 3.000
   Mean : 446.000               Mean : 2.309
2nd Qu. : 668.500            2nd Qu. : 3.000
   Max. : 891.000               Max. : 3.000

                                                 Name
                            "Braund, Mr. Owen Harris" :   1    male
"Cumings, Mrs. John Bradley (Florence Briggs Thayer)" :   1  female
                             "Heikkinen, Miss. Laina" :   1
       "Futrelle, Mrs. Jacques Heath (Lily May Peel)" :   1
                           "Allen, Mr. William Henry" :   1
                                  "Moran, Mr. James" :   1
                                              (Other) : 885
          SibSp            Parch           Ticket                Fare
   Min. : 0.000     Min. : 0.000    347082 :   7     Min. :   0.000
1st Qu. : 0.000  1st Qu. : 0.000      1601 :   7  1st Qu. :   7.910
 Median : 0.000   Median : 0.000  CA. 2343 :   7   Median :  14.454
   Mean : 0.523     Mean : 0.382   3101295 :   6     Mean :  32.204
2nd Qu. : 1.000  2nd Qu. : 0.000   CA 2144 :   6  2nd Qu. :  31.000
   Max. : 8.000     Max. : 6.000    347088 :   6     Max. : 512.329
                                    (Other) : 852

  Embarked
   S : 644
   C : 168
   Q :  77
NA's :   2
```

And notice how type of the `Survived` variable changed to nominal.

Pclass

This variable has index type. We can keep it like it is or we can change it to nominal. Both ways can be useful. For example parsed as index could give an interpretation to the order. We can say that somehow, because of ordering class 1 is lower than class 2, and class 2 is between classes 1 and 3. At the same time we can keep it as nominal if we do not want to use the ordering. Let's choose nominal for now, considering that 1,2 and 3 are just labels for type of tickets, with no other meaning attached. We proceed in the same way:

```
new Csv()
        .withTypes(VarType.NOMINAL, "Survived", "Pclass")
        .read(root + "train.csv")
        .printSummary();
```

Notice that we append the variable name after `Survived`. This is possible since the `withTypes` method specify a type, and follows a dynamic array of strings, for the names of variables.

Name

This is the passenger names and the values are unique. As it is, the predictive power of this field is null. We keep it as it is. Note that it contains valuable information, but not in this direct form.

Sex

This field specifies the gender of the passenger. We have 577 males and 314 females.

Age

This field specifies the age of an passenger. We would expect that to parse this variable as numeric or at leas index, but is nominal. Why that happened? Notice that the values looks like numbers. But the first value (the most frequent one, 117 instances) has nothing specified. Well, the variable is nominal has to do with how *Csv* parsing handles missing values. By default, the *csv* parsing considers as missing values only the string "?". But the most frequent value in this field is empty string "". This means that empty string is not considered a missing value. Because empty string can't produce numbers from parsing, the variable is *promoted* to nominal.

We can customize the missing value handling by specifying the valid strings for that purpose. We use `.useNAValues(String...naValues)` to tell the parser all the valid strings which are missing values. In our case we want just the empty string to be a missing value. When the parser will found an empty string it will set the variable value as missing value. It will *not promote* variable to nominal, since a missing value is a legal value.

```
new Csv()
        .withNAValues("")
        .withTypes(VarType.NOMINAL, "Survived", "Pclass")
        .read(root + "train.csv")
        .printSummary();
```

```
Frame Summary
=============
* rowCount: 891
* complete: 183/891
* varCount: 12
* varNames:

 0. PassengerId : idx | 4.   Sex : nom |  8.   Ticket : nom |
 1.    Survived : nom | 5.   Age : num |  9.     Fare : num |
 2.      Pclass : nom | 6. SibSp : idx | 10.    Cabin : nom |
 3.        Name : nom | 7. Parch : idx | 11. Embarked : nom |

      PassengerId  Survived   Pclass
   Min. :   1.000   0 : 549  3 : 491
1st Qu. : 223.500   1 : 342  1 : 216
 Median : 446.000            2 : 184
   Mean : 446.000
2nd Qu. : 668.500
   Max. : 891.000

                                                    Name
                              "Braund, Mr. Owen Harris" :   1    male
"Cumings, Mrs. John Bradley (Florence Briggs Thayer)" :   1  female
                               "Heikkinen, Miss. Laina" :   1
       "Futrelle, Mrs. Jacques Heath (Lily May Peel)" :   1
```

```
                          "Allen, Mr. William Henry" :   1
                                 "Moran, Mr. James" :   1
                                           (Other) : 885
             Age              SibSp             Parch           Ticket
   Min.   :  0.420    Min.   : 0.000    Min.   : 0.000    347082 :   7
1st Qu.   : 20.125    1st Qu. : 0.000    1st Qu. : 0.000      1601 :   7
 Median   : 28.000    Median : 0.000    Median : 0.000    CA. 2343 :   7
   Mean   : 29.699    Mean   : 0.523    Mean   : 0.382    3101295 :   6
2nd Qu.   : 38.000    2nd Qu. : 1.000    2nd Qu. : 0.000    CA 2144 :   6
   Max.   : 80.000    Max.   : 8.000    Max.   : 6.000    347088 :   6
   NA's   :    177                                       (Other) : 852
            Cabin     Embarked
        G6      :   4     S : 644
C23 C25 C27     :   4     C : 168
    B96 B98     :   4     Q :  77
        F33     :   3   NA's :   2
       E101     :   3
    (Other)     : 183
       NA's     : 687
```

Notice what happened: *Age* field is now numeric and it contains 177 missing values.

SibSp

It's meaning is "siblings/spouses". It's parsed as index, which is natural. In pathological cases with sick imagination we can consider a "quarter of a wife" for example.

Parch

It's meaning is "parents/children". It is naturally parsed as index.

Ticket

This is the code of the ticket. Probably a family can have the same ticket, thus must be the reason why the frequencies have values up to 7. This field is nominal. It has low predictive power used directly. Perhaps contains valuable information, but used directly in row format would not help much.

Fare

This is the price for passenger fare and should be numeric, like it is.

Cabin

Code of the passenger's cabin, parsed as nominal. Same notes as for `Ticket` variable.

Embarked

Code for the embarking city, which could be: C = Cherbourg, Q = Queenstown, S = Southampton. It's parsed as nominal and has 2 missing values.

If we are content with our parsing, we load data into a data frame for later use:

```
Frame train = new Csv()
        .withNAValues("")
        .withTypes(VarType.NOMINAL, "Survived", "Pclass")
```

```
        .read(root + "train.csv");
train.printSummary();
```

Read test data from *csv* file

Once we have a training frame we can load also the test data. We do that to take a look at the frame and because data is small and there is no memory or time problem cost associated with it. To avoid adding again the *csv* options and to get identical levels nominal variables, we use a different way to parse the data set. We specify variable types by frame templates:

```
Frame test = new Csv()
        .withNAValues("")
        .withTemplate(train)
        .read(root + "test.csv");
```

Instead to specify again the preferred types for variables, we use train frame as a template for variable types. This has also the side effect that the encoding of categorical variables is identical.

```
Frame Summary
=============
* rowCount: 891
* complete: 183/891
* varCount: 12
* varNames:

 0. PassengerId : idx | 4.   Sex : nom |  8.   Ticket : nom |
 1.    Survived : nom | 5.   Age : num |  9.     Fare : num |
 2.      Pclass : nom | 6. SibSp : idx | 10.    Cabin : nom |
 3.        Name : nom | 7. Parch : idx | 11. Embarked : nom |

      PassengerId  Survived   Pclass
   Min. :   1.000   0 : 549  3 : 491
1st Qu. : 223.500   1 : 342  1 : 216
 Median : 446.000            2 : 184
   Mean : 446.000
2nd Qu. : 668.500
   Max. : 891.000

                                                 Name
                           "Braund, Mr. Owen Harris" :   1    male
"Cumings, Mrs. John Bradley (Florence Briggs Thayer)" :   1  female
                            "Heikkinen, Miss. Laina" :   1
       "Futrelle, Mrs. Jacques Heath (Lily May Peel)" :   1
                          "Allen, Mr. William Henry" :   1
                                  "Moran, Mr. James" :   1
                                             (Other) : 885
             Age            SibSp            Parch          Ticket
   Min. :  0.420     Min. : 0.000     Min. : 0.000    347082 :   7
1st Qu. : 20.125  1st Qu. : 0.000  1st Qu. : 0.000      1601 :   7  1
 Median : 28.000   Median : 0.000   Median : 0.000  CA. 2343 :   7
   Mean : 29.699     Mean : 0.523     Mean : 0.382   3101295 :   6
2nd Qu. : 38.000  2nd Qu. : 1.000  2nd Qu. : 0.000   CA 2144 :   6  2
   Max. : 80.000     Max. : 8.000     Max. : 6.000    347088 :   6
```

```
   NA's :    177                                          (Other) : 852
            Cabin    Embarked
         G6 :   4     S : 644
C23 C25 C27 :   4     C : 168
    B96 B98 :   4     Q :  77
        F33 :   3  NA's :   2
       E101 :   3
    (Other) : 183
       NA's : 687
```

We can note that we don't have *Survived* variable anymore. This is correct since this is what we have to predict. Note also that the types for the remaining variables are the same with training data set.

Simple models

Build a majority model

To make a first submission we will build a very simple model, which classifies with a single value all instances. This value is the majority label.

Let's inspect at how target variable look like.

```
DVector.newFromCount(false, train.var("Survived")).printSummary();
```

```
        0       1
        -       -
  549.000 342.000
```

As we already new from the summary, the number of passengers who didn't survived is lower than those who did. Let's see percentages:

```
DVector.newFromCount(false, train.var("Survived")).normalize().print
```

```
      0     1
      -     -
  0.616 0.384
```

We note that there are about 61% of passengers who did not survived. We will create a submit data set, which we will save for later submission. How we can do that?

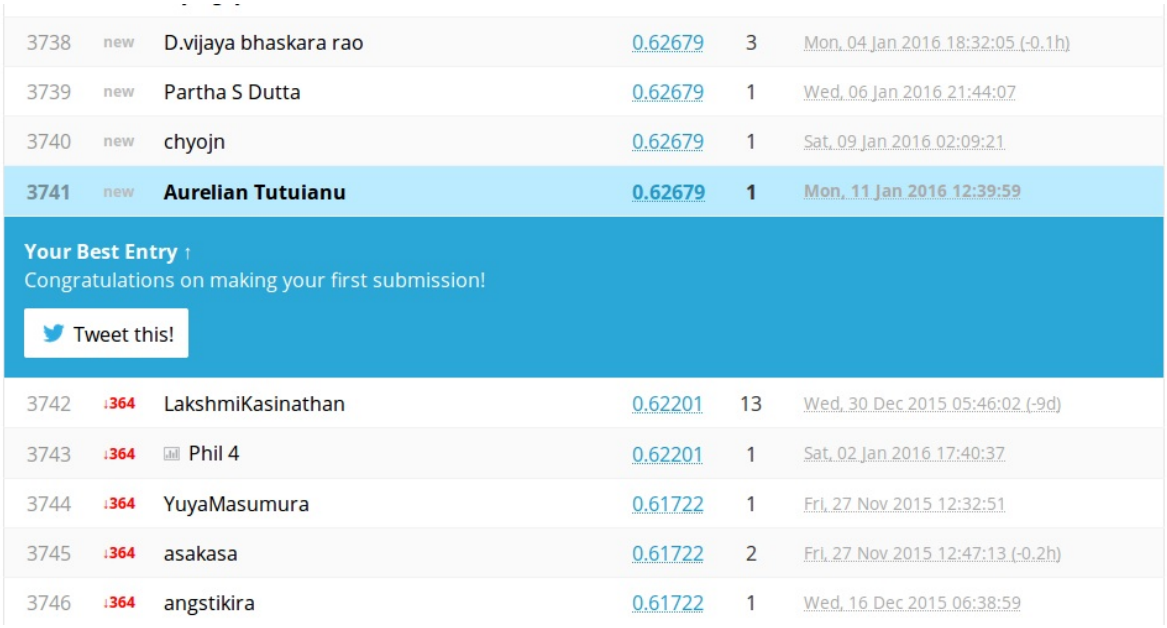
```
Nominal prediction = Nominal.from(test.rowCount(), row -> "0").withNa
Frame submit = SolidFrame.wrapOf(test.var("PassengerId"), prediction

new Csv().withQuotes(false).write(submit, root + "majority_submit.csv
```

In the first line we created a new nominal variable. The size of the new variable is the number of rows from the test frame. For each row we produce the same label `"0"`. We name this variable `Survived`.

In the second line we created a new frame taking the variable named `PassengerId` from the test data set and the new prediction variable.

In the last line we wrote a new csv file with the csv parsing utility, taking care to not write quotes. We can submit this file and see which are the results.



3738	new	D.vijaya bhaskara rao	0.62679	3	Mon, 04 Jan 2016 18:32:05 (-0.1h)
3739	new	Partha S Dutta	0.62679	1	Wed, 06 Jan 2016 21:44:07
3740	new	chyojn	0.62679	1	Sat, 09 Jan 2016 02:09:21
3741	new	**Aurelian Tutuianu**	0.62679	1	Mon, 11 Jan 2016 12:39:59

Your Best Entry ↑
Congratulations on making your first submission!
Tweet this!

3742	↓364	LakshmiKasinathan	0.62201	13	Wed, 30 Dec 2015 05:46:02 (-9d)
3743	↓364	Phil 4	0.62201	1	Sat, 02 Jan 2016 17:40:37
3744	↓364	YuyaMasumura	0.61722	1	Fri, 27 Nov 2015 12:32:51
3745	↓364	asakasa	0.61722	2	Fri, 27 Nov 2015 12:47:13 (-0.2h)
3746	↓364	angstikira	0.61722	1	Wed, 16 Dec 2015 06:38:59

Figure 1.8.2.1 Submission result with majority classifier

Build a simple gender model

It has been said that "women and children first" really happened during Titanic tragedy. If this was true or not, we do not know. But we can use data to see if we are hearing the same story. For now we will take the gender and see if it had an influence. We will build a contingency table for variables `Sex` and `Survived`.

```
DTable.newFromCounts(train.var("Sex"), train.var("Survived"), false)
```

```
              0       1   total
   male 468.000 109.000 577.000
 female  81.000 233.000 314.000
  total 549.000 342.000 891.000
```

On rows we have levels of `Sex` variable. On columns we have levels of `Sex` variable. Cells are computed as counts. What we see is that there are a lot of men who did not survived and a lot of women who does. We will normalize on rows to take a closer look.

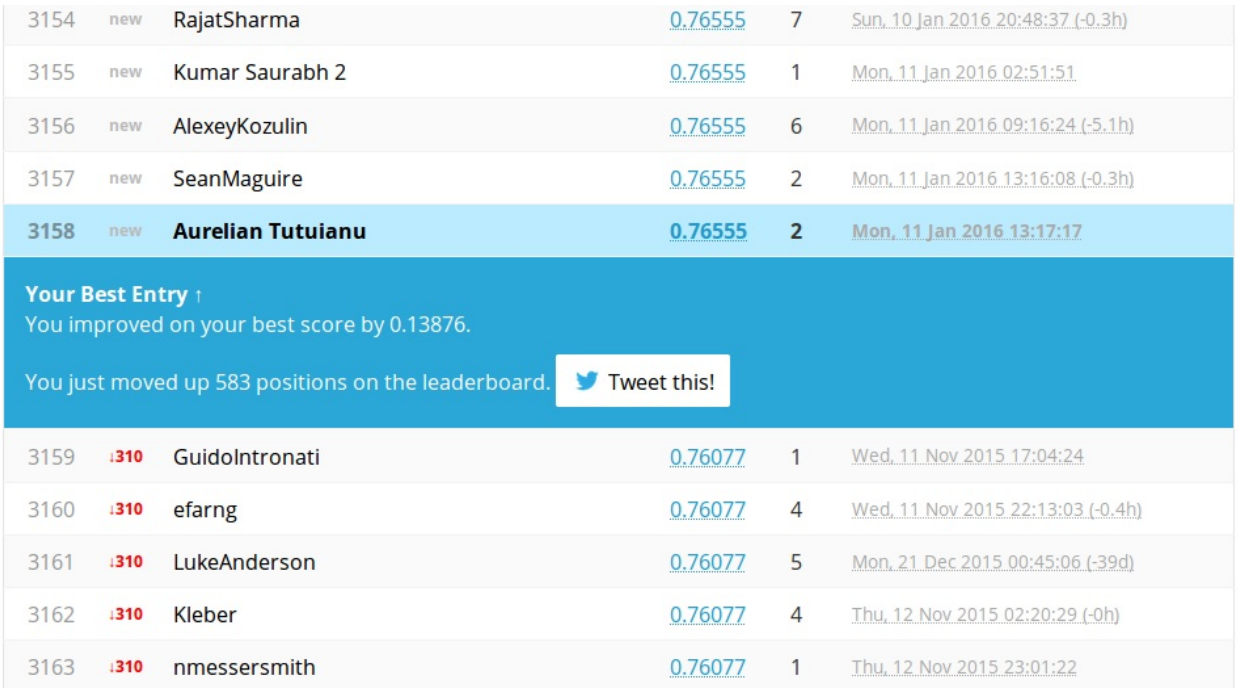
```
DTable.newFromCounts(train.var("Sex"), train.var("Survived"), false)
  .normalizeOnRows().printSummary();
```

```
            0     1 total
   male 0.811 0.189 1.000
 female 0.258 0.742 1.000
  total 1.069 0.931 2.000
```

It seems that men survived with a rate of 0.19 and women with 0.74. The values are so obvious, we need no hypothesis testing to check that this variable is significant for classification. We will build a simple model where we predict as survived all the women and not survived all the men.

```
Var prediction = Nominal.from(test.rowCount(),
```

```
        row -> test.label(row, "Sex").equals("male") ? "0" : "1")
        .withName("Survived");
Frame submit = SolidFrame.wrapOf(test.var("PassengerId"), prediction
new Csv().withQuotes(false).write(submit, root + "gender_submit.csv"
```



3154	new	RajatSharma	0.76555	7	Sun, 10 Jan 2016 20:48:37 (-0.3h)
3155	new	Kumar Saurabh 2	0.76555	1	Mon, 11 Jan 2016 02:51:51
3156	new	AlexeyKozulin	0.76555	6	Mon, 11 Jan 2016 09:16:24 (-5.1h)
3157	new	SeanMaguire	0.76555	2	Mon, 11 Jan 2016 13:16:08 (-0.3h)
3158	new	**Aurelian Tutuianu**	**0.76555**	**2**	**Mon, 11 Jan 2016 13:17:17**

Your Best Entry ↑
You improved on your best score by 0.13876.

You just moved up 583 positions on the leaderboard. Tweet this!

3159	↓310	GuidoIntronati	0.76077	1	Wed, 11 Nov 2015 17:04:24
3160	↓310	efarng	0.76077	4	Wed, 11 Nov 2015 22:13:03 (-0.4h)
3161	↓310	LukeAnderson	0.76077	5	Mon, 21 Dec 2015 00:45:06 (-39d)
3162	↓310	Kleber	0.76077	4	Thu, 12 Nov 2015 02:20:29 (-0h)
3163	↓310	nmessersmith	0.76077	1	Thu, 12 Nov 2015 23:01:22

Figure 1.8.2.2 Submission result with gender classifier

Tree model

Tree model

Building models in the manual way is often not the way to go. This process is tedious and time consuming. There are already built automated procedures, which incorporate miscellaneous approaches to learn a classifier. One of the often used models is the decision tree. Decision trees are greedy local approximations build in a recursive greedy fashion. Often the split decision at node level uses a single feature. At leave nodes a simple majority classifier creates the classification rule.

Gender model with decision tree

Initially we will build a CART decision tree using as input feature the *Sex* variable. We do this to exemplify how a manual rule can be created in an automated fashion.

```
Frame tr = train.mapVars("Survived,Sex");
CTree tree = CTree.newCART();
tree.train(tr, "Survived");
tree.printSummary();

CTree model
===============

Description:
CTree {varSelector=VarSelector[ALL];
minCount=1;
maxDepth=-1;
tests=ORDINAL:NumericBinary,INDEX:NumericBinary,BINARY:BinaryBinary,I
func=GiniGain;
split=ToAllWeighted;
}

Capabilities:
types inputs/targets: BINARY,INDEX,NOMINAL,NUMERIC/NOMINAL
counts inputs/targets: [1,1000000] / [1,1]
missing inputs/targets: true/false

Learned model:
input vars:

 0. Sex : NOMINAL  |

target vars:
> Survived : NOMINAL [?,0,1]

total number of nodes: 3
total number of leaves: 2
description:
split, n/err, classes (densities) [* if is leaf / purity if not]
```

```
|- 1. root    891/342 0 (0.616 0.384 ) [0.139648]
|   |- 2. Sex == female    314/81 1 (0.258 0.742 ) *
|   |- 3. Sex != female    577/109 0 (0.811 0.189 ) *
```

Taking a closer look at the last three rows from the output, one can identify our manual rule. Basically the interpretation is: *"all the females survived, all the males did not"*. For exemplification purposes we build also the submit file.

```
// fit the tree to test data frame
CFit fit = tree.fit(test);
// build teh submission
Frame submit = SolidFrame.wrapOf(
        // use original ids
        test.var("PassengerId"),
        // use class prediction from fit
        fit.firstClasses().solidCopy().withName("Survived")
);
// write to a submit file
new Csv().withQuotes(false).write(submit, root + "tree1-model.csv");
```

Enrich tree by using other features

Our training data set has more than a single input feature. Thus We can state we didn't use all the information available. We will add now the class and embarking port and see how it behaves.

```
Frame tr = train.mapVars("Survived,Sex,Pclass,Embarked");

CTree tree = CTree.newCART();
tree.train(tr, "Survived");
tree.printSummary();

CFit fit = tree.fit(test);
Frame submit = SolidFrame.wrapOf(
        test.var("PassengerId"),
        fit.firstClasses().withName("Survived")
);
new Csv().withQuotes(false).write(submit, root + "tree2-model.csv");
```

This is the resulted tree:

```
Capabilities:
types inputs/targets: BINARY,INDEX,NOMINAL,NUMERIC/NOMINAL
counts inputs/targets: [1,1000000] / [1,1]
missing inputs/targets: true/false

Learned model:
input vars:

 0. Sex : NOMINAL  | 1. Pclass : NOMINAL  | 2. Embarked : NOMINAL  |

target vars:
> Survived : NOMINAL [?,0,1]
```

```
total number of nodes: 35
total number of leaves: 18
description:
split, n/err, classes (densities) [* if is leaf / purity if not]

|- 1. root    891/342 0 (0.616 0.384 ) [0.139648]
|   |- 2. Sex == female    314/81 1 (0.258 0.742 ) [0.0204665]
|   |   |- 3. Pclass == 2    76/6 1 (0.079 0.921 ) [0.0003369]
|   |   |   |- 4. Embarked == Q    2/0 1 (0 1 ) *
|   |   |   |- 5. Embarked != Q    74/6 1 (0.081 0.919 ) [0.0013737]
|   |   |   |   |- 6. Embarked == C    7/0 1 (0 1 ) *
|   |   |   |   |- 7. Embarked != C    67/6 1 (0.09 0.91 ) *
|   |   |- 8. Pclass != 2    238/75 1 (0.315 0.685 ) [0.1047167]
|   |   |   |- 9. Pclass == 1    94/3 1 (0.032 0.968 ) [0.0000234]
|   |   |   |   |- 10. Embarked == Q    1/0 1 (0 1 ) *
|   |   |   |   |- 11. Embarked != Q    93/3 1 (0.032 0.968 ) [0.000
|   |   |   |   |   |- 12. Embarked == C    43/1 1 (0.023 0.977 ) *
|   |   |   |   |   |- 13. Embarked != C    50/2 1 (0.04 0.96 ) *
|   |   |   |- 14. Pclass != 1    144/72 1 (0.5 0.5 ) [0.0307125]
|   |   |   |   |- 15. Embarked == Q    33/9 1 (0.273 0.727 ) *
|   |   |   |   |- 16. Embarked != Q    111/48 0 (0.568 0.432 ) [0.0
|   |   |   |   |   |- 17. Embarked == C    23/8 1 (0.348 0.652 ) *
|   |   |   |   |   |- 18. Embarked != C    88/33 0 (0.625 0.375 ) *
|   |- 19. Sex != female    577/109 0 (0.811 0.189 ) [0.0020493]
|   |   |- 20. Embarked == Q    41/3 0 (0.927 0.073 ) [0.0002677]
|   |   |   |- 21. Pclass == 2    1/0 0 (1 0 ) *
|   |   |   |- 22. Pclass != 2    40/3 0 (0.925 0.075 ) [0.0002885]
|   |   |   |   |- 23. Pclass == 1    1/0 0 (1 0 ) *
|   |   |   |   |- 24. Pclass != 1    39/3 0 (0.923 0.077 ) *
|   |   |- 25. Embarked != Q    536/106 0 (0.802 0.198 ) [0.0049791]
|   |   |   |- 26. Embarked == C    95/29 0 (0.695 0.305 ) [0.002607
|   |   |   |   |- 27. Pclass == 2    10/2 0 (0.8 0.2 ) *
|   |   |   |   |- 28. Pclass != 2    85/27 0 (0.682 0.318 ) [0.0148
|   |   |   |   |   |- 29. Pclass == 1    42/17 0 (0.595 0.405 ) *
|   |   |   |   |   |- 30. Pclass != 1    43/10 0 (0.767 0.233 ) *
|   |   |   |- 31. Embarked != C    441/77 0 (0.825 0.175 ) [0.00022
|   |   |   |   |- 32. Pclass == 2    97/15 0 (0.845 0.155 ) *
|   |   |   |   |- 33. Pclass != 2    344/62 0 (0.82 0.18 ) [0.01809
|   |   |   |   |   |- 34. Pclass == 1    79/28 0 (0.646 0.354 ) *
|   |   |   |   |   |- 35. Pclass != 1    265/34 0 (0.872 0.128 ) *
```

The tree is much richer and there are more chances to be better. This is what happened after submission.

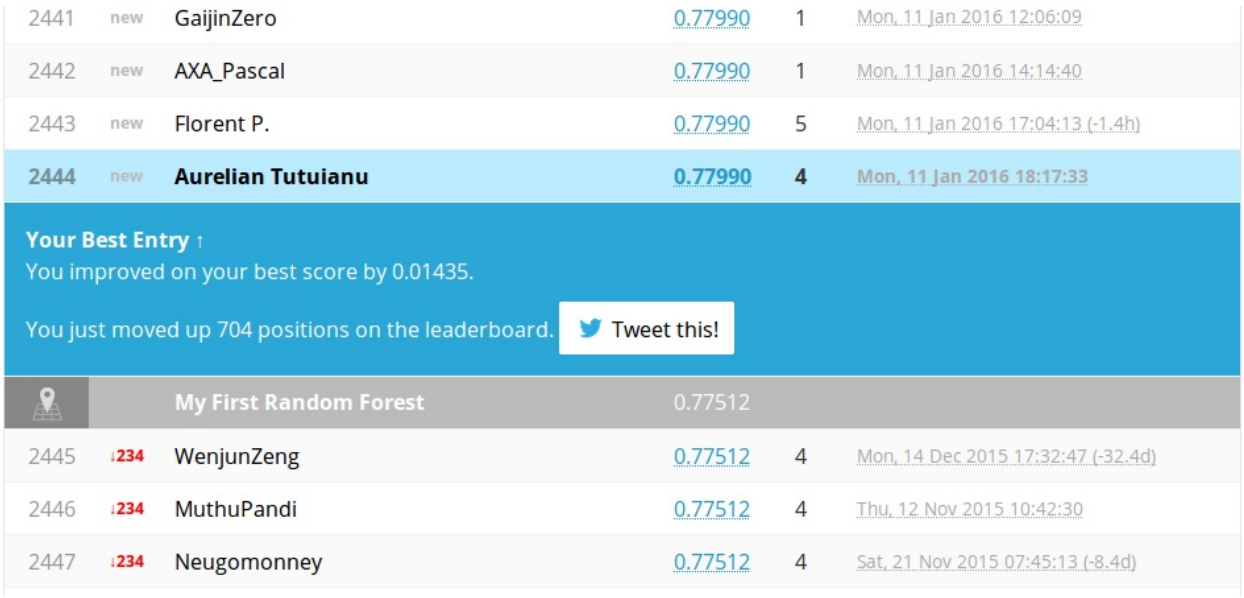


Figure 1.8.3.1 Results after submission of enriched tree

Nice!. We advanced 704 positions and improved our score with 0.01435. On public leader board we have a nice 0.77990 accuracy score.

Overfitting with trees

What about using other input features to improve our prediction accuracy? There are some of them which we can include directly, with no changes: *Age*,*Fare*,*SibSp* and *Parch*.

We can change the script slightly, to include those new input features. But we can do better, we can use cross-validation to estimate what will happen.

```
CTree tree = CTree.newCART();
CEvaluation.cv(train.mapVars("Survived,Sex,Pclass,Embarked"),
        "Survived", tree, 10);
CEvaluation.cv(train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,S
        "Survived", tree, 10);

CrossValidation with 10 folds
...
Mean accuracy:0.811473
SE: 0.034062     (Standard error)

CrossValidation with 10 folds
...
Mean accuracy:0.774370
SE: 0.046494     (Standard error)
```

Notice that the 10-crossfold estimator of the accuracy has dropped with a large quantity. What happens? We can have an idea if we take a look at the learned tree:

```
tree.train(train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,SibSp
tree.printSummary();
```

```
total number of nodes: 1197
total number of leaves: 599
description:
split, n/err, classes (densities) [* if is leaf / purity if not]

|- 1. root    891/342 0 (0.616 0.384 ) [0.139648]
|   |- 2. Sex == female    314/81 1 (0.258 0.742 ) [0.0282131]
|   |   |- 3. Fare <= 48.2    225/76 1 (0.338 0.662 ) [0.0645841]
|   |   |   |- 4. Pclass == 2    74/6 1 (0.081 0.919 ) [0.0099206]
|   |   |   |   |- 5. Age <= 56    73/5 1 (0.069 0.931 ) [0.0036241]
|   |   |   |   |   |- 6. Age <= 23.5    22/0 1 (0 1 ) *
|   |   |   |   |   |- 7. Age > 23.5    53/5 1 (0.095 0.905 ) [0.006
|   |   |   |   |   |   |- 8. Age <= 27.5    14/3 1 (0.243 0.757 ) [
|   |   |   |   |   |   |   |- 9. Age <= 25.5    10/1 1 (0.122 0.878
|   |   |   |   |   |   |   |   |- 10. Fare <= 13.75    3/1 1 (0.474
|   |   |   |   |   |   |   |   |   |- 11. Embarked == Q    1/0 1 (0
|   |   |   |   |   |   |   |   |   |- 12. Embarked != Q    2/1 1 (0
|   |   |   |   |   |   |   |   |- 13. Fare > 13.75    7/0 1 (0 1 )
|   |   |   |   |   |   |   |- 14. Age > 25.5    6/2 1 (0.486 0.514
|   |   |   |   |   |   |   |   |- 15. Fare <= 17.42915    3/0 1 (0
|   |   |   |   |   |   |   |   |- 16. Fare > 17.42915    3/1 0 (0.9
|   |   |   |   |   |   |   |   |   |- 17. Fare <= 29.5    2/0 0 (1 (
|   |   |   |   |   |   |   |   |   |- 18. Fare > 29.5    1/0 1 (0 1
|   |   |   |   |   |   |- 19. Age > 27.5    41/2 1 (0.05 0.95 ) [0.(
|   |   |   |   |   |   |   |- 20. Age <= 37    26/0 1 (0 1 ) *

...

|   |   |   |   |   |   |   |   |   |   |- 1189. Fare > 28.275    3/(
|   |   |   |   |   |   |   |   |   |- 1190. Fare > 30.5979    13/3 0 (0
|   |   |   |   |   |   |   |   |   |- 1191. Pclass == 1    10/1 0 ((
|   |   |   |   |   |   |   |   |   |   |- 1192. Fare <= 37.55    4/1
|   |   |   |   |   |   |   |   |   |   |   |- 1193. Fare <= 35.25
|   |   |   |   |   |   |   |   |   |   |   |- 1194. Fare > 35.25
|   |   |   |   |   |   |   |   |   |   |- 1195. Fare > 37.55    6/0
|   |   |   |   |   |   |   |   |   |- 1196. Pclass != 1    3/1 1 (0
|   |   |   |   |- 1197. SibSp > 2.5    15/0 0 (1 0 ) *
```

Notice how large is the tree. Basically the tree was full grown and overfit the training data set too much. We can ask ourselves why that happens? Why it happens now, and did not happened when we had fewer inputs? The answer is that it happened also before. But it's consequences were not so drastic.

The first tree used for training just 3 input nominal features. Notice that all three features are nominal. The maximum number of groups which one can form is given by the product of the number of levels for each feature. This total maximal number is $2 * 3 * 3 = 18$. It practically exhausted the discrimination potential of those features. It did overfit in that reduced space of features. When we apply the model to the whole data set, the effect of exhaustion is not seen anymore.

The second tree does the same thing, but this time in a richer space, with added input dimensions. Compared with the full feature space, we see the effect.

There are two approaches to avoid overfit for a decision tree. The first approach is to stop learning up to the moment when we exhaust the data. The name for this approach is *early stop*. We can do that by

specifying some parameters of the tree model:

- Set a minimum number of instances for leaf node
- Set a maximal depth for the tree
- Not implemented yet, but easy to do: complexity threshold, maximal number of nodes in a tree

The second approach is to prune the tree. Pruning procedure consists of growing the full tree and later on removing some nodes if they do not provide some type of gain. Currently we implemented only *reduced error pruning strategy*.

We will test with 10-fold cross validation an early-stopping strategy to see how it works.

```
CEvaluation.cv(train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,S
        "Survived", tree.withMaxDepth(12).withMinCount(4), 10);

Mean accuracy:0.796854
SE: 0.034929     (Standard error)
```

I tried some values, just to show that we can do something about it, but the progress did not appear. We should try a different approach, and that is an ensemble. Next session contains directions on how to build such an ensemble.

Random forest model

CForest model

Random forests are well-known to work well when the irreducible error from the training data is high. This is probably the case of this Titanic data set. We have reasons to believe that this is the situation since it was a tragedy. A lot of random or not-so-expected things happened. That happened despite of the bravery and the sacrifice of the crew and others.

Random forests are the invention of Leo Breiman. The first design was a joint effort together with Adele Cutler. The base of random forests is bagging (or **b**ootstrapp **ag**gregation). On top of that, selecting just a random limited number of variables at each node is the core of the algorithm.

We will work with random forests for now. This ensemble is mode robust and is capable of obtaining much better results than a single tree. At the same time we will introduce 10-fold cross validation to check our progress and estimate the error produced.

In the beginning we will use 10-fold cross validation for estimating the accuracy on public leader board. We will build a static method which will to cross validation for a single classifier. Note that there are similar construct in the library, but is instructive to build one tailored for our needs.

Build a 10-fold cross validation

```
public static void cv(Frame df, Classifier c) {
    String target = "Survived"; // this is our class
    int folds = 10; // number of folds

    // take 10 samples from our data set
    Mapping[] mappings = buildFolds(df, target, folds);
    Numeric acc = Numeric.empty(folds); // variable to store accurac

    WS.printf("Cross validation 10-fold\n");
    for (int fold = 0; fold < folds; fold++) {
        Frame train = df.removeRows(mappings[fold]); // train used i
        Frame test = df.mapRows(mappings[fold]); // remaining instan

        Classifier cc = c.newInstance(); // builds a new instance of
        cc.train(train, target); // train it on old data se

        // build a confusion matrix to compute accuracy
        double foldAcc = new Confusion(
            test.var(target), cc.fit(test).firstClasses()).accuracy(
        acc.setValue(fold, foldAcc); // collect accuracy

        WS.printf("CV fold:%2d, acc: %.6f, mean: %.6f, se: %.6f\n",
                fold + 1,
                foldAcc,
                CoreTools.mean(acc).value(),
                CoreTools.var(acc).sdValue());
    }
```

```
    WS.printf("=================\n");
    WS.printf("mean: %.6f, se: %.6f\n\n",
            CoreTools.mean(acc).value(),
            CoreTools.var(acc).sdValue());
}

// builds almost equals folds with samples stratified by target fiel
// the idea is to get folds with proportion of strata as close as pos
public static Mapping[] buildFolds(Frame df, String target, int folds
    Var rows = IntStream.range(0, df.rowCount()).boxed().collect(Ind
    rows = Filters.shuffle(rows); // shuffle all rows
    rows = Filters.refSort(rows, df.var(target).refComparator()); //

    // build strata
    Mapping[] strata = new Mapping[folds];
    for (int i = 0; i < folds; i++) {
        strata[i] = Mapping.empty();
    }
    for (int i = 0; i < df.rowCount(); i++) {
        strata[i % folds].add(rows.index(i));
    }
    return strata;
}
```

We can use this 10-fold cross validation procedure to test our old tree classifier in the following way:

```
cv(train.mapVars("Survived,Sex,Pclass,Embarked"), CTree.newCART());
```

This will give us the following results:

```
Cross validation 10-fold
CV fold: 1, acc: 0.811111, mean: 0.811111, se: NaN
CV fold: 2, acc: 0.820225, mean: 0.815668, se: 0.006444
CV fold: 3, acc: 0.786517, mean: 0.805951, se: 0.017436
CV fold: 4, acc: 0.797753, mean: 0.803901, se: 0.014815
CV fold: 5, acc: 0.752809, mean: 0.793683, se: 0.026205
CV fold: 6, acc: 0.887640, mean: 0.809342, se: 0.044952
CV fold: 7, acc: 0.707865, mean: 0.794846, se: 0.056169
CV fold: 8, acc: 0.853933, mean: 0.802232, se: 0.056042
CV fold: 9, acc: 0.910112, mean: 0.814218, se: 0.063571
CV fold:10, acc: 0.786517, mean: 0.811448, se: 0.060572
=================
mean: 0.811448, se: 0.060572
```

Our first random forest

The name of the random forest implementation is `CForest`. To build a new ensemble of trees, one have to instantiate it in the following way:

```
Classifier rf = CForest.newRF();
```

There are a lot of things which can be customized for a random forest. Among them one can change:

- Number of trees for classification

- Which kind of weak classifier to use (you can customize this customized accordingly, like any other classifier)
- Number of threads in pool (if you want to use parallelism)
- What to do after each running step

Let's build one and use ore new cross validation procedure to estimate it's error.

```
RandomSource.setSeed(123);
Frame tr = train.mapVars("Survived,Sex,Pclass,Embarked");
CForest rf = CForest.newRF().withRuns(100);
cv(tr, rf);

Cross validation 10-fold
CV fold: 1, acc: 0.833333, mean: 0.833333, se: NaN
CV fold: 2, acc: 0.820225, mean: 0.826779, se: 0.009269
CV fold: 3, acc: 0.808989, mean: 0.820849, se: 0.012184
CV fold: 4, acc: 0.808989, mean: 0.817884, se: 0.011582
CV fold: 5, acc: 0.764045, mean: 0.807116, se: 0.026083
CV fold: 6, acc: 0.797753, mean: 0.805556, se: 0.023641
CV fold: 7, acc: 0.876404, mean: 0.815677, se: 0.034392
CV fold: 8, acc: 0.820225, mean: 0.816245, se: 0.031881
CV fold: 9, acc: 0.797753, mean: 0.814191, se: 0.030453
CV fold:10, acc: 0.786517, mean: 0.811423, se: 0.030015
=================
mean: 0.811423, se: 0.030015
```

Well, an identical output. This is due to the fact that our variables are already exhausted by the tree. It looks like an underfit. If one consider bias variance trade off, one can see this as high bias. We need to enrich our feature space to improve our performance.

Let's be direct and test what would happen if we would use all our directly usable features? This time we will fit also the training data set, to see the distribution of the training error.

```
RandomSource.setSeed(123);
Frame tr = train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,SibS
CForest rf = CForest.newRF().withRuns(100);
cv(tr, rf);

rf.train(tr, "Survived");
CFit fit = rf.fit(test);
new Confusion(tr.var("Survived"), rf.fit(tr).firstClasses()).printSu

Cross validation 10-fold
CV fold: 1, acc: 0.844444, mean: 0.844444, se: NaN
CV fold: 2, acc: 0.820225, mean: 0.832335, se: 0.017126
CV fold: 3, acc: 0.808989, mean: 0.824553, se: 0.018120
CV fold: 4, acc: 0.786517, mean: 0.815044, se: 0.024095
CV fold: 5, acc: 0.764045, mean: 0.804844, se: 0.030913
CV fold: 6, acc: 0.808989, mean: 0.805535, se: 0.027701
CV fold: 7, acc: 0.820225, mean: 0.807633, se: 0.025890
CV fold: 8, acc: 0.797753, mean: 0.806398, se: 0.024222
CV fold: 9, acc: 0.853933, mean: 0.811680, se: 0.027649
CV fold:10, acc: 0.808989, mean: 0.811411, se: 0.026081
=================
```

```
mean: 0.811411, se: 0.026081

> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >540    9 |   549
     1 |   15 >327 |   342
 ----- |    -    - | -----
 total |  555  336 |   891


Complete cases 891 from 891
Acc: 0.973064         (Accuracy )
F1:  0.9782609         (F1 score / F-measure)
MCC: 0.9429616         (Matthew correlation coefficient)
Pre: 0.972973         (Precision)
Rec: 0.9836066         (Recall)
G:   0.9782753         (G-measure)
```

This time we have a good example of overfit. Why is that? Look at the confusion matrix on the training set. We fit too well the training data. This data set is well known for its high irreducible error. And there is an explanation for that. During the tragic event a lot of exceptional things happened. For example I read somewhere that an old lady which had a dog was not allowed to embark with her pet due to regulations. As a consequence she decided to not leave it and she chose to die with him. It's close to impossible to learn those kind of things, even if the information would be available.

We should reduce the error somehow. We can try to decrease the overfit by adding more learners. Let's see if that would be enough for our purpose.

```
RandomSource.setSeed(123);
Frame tr = train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,SibS
CForest rf = CForest.newRF().withRuns(500);
cv(tr, rf);

rf.train(tr, "Survived");
CFit fit = rf.fit(test);
new Confusion(tr.var("Survived"), rf.fit(tr).firstClasses()).printSu

Cross validation 10-fold
CV fold: 1, acc: 0.844444, mean: 0.844444, se: NaN
CV fold: 2, acc: 0.820225, mean: 0.832335, se: 0.017126
CV fold: 3, acc: 0.797753, mean: 0.820807, se: 0.023351
CV fold: 4, acc: 0.786517, mean: 0.812235, se: 0.025641
CV fold: 5, acc: 0.775281, mean: 0.804844, se: 0.027681
CV fold: 6, acc: 0.808989, mean: 0.805535, se: 0.024816
CV fold: 7, acc: 0.820225, mean: 0.807633, se: 0.023324
CV fold: 8, acc: 0.808989, mean: 0.807803, se: 0.021600
CV fold: 9, acc: 0.865169, mean: 0.814177, se: 0.027819
CV fold:10, acc: 0.797753, mean: 0.812534, se: 0.026737
=================
mean: 0.812534, se: 0.026737

> Confusion
```

```
 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >540    9 |   549
     1 |   16 >326 |   342
 ----- |    -    - | -----
 total |  556  335 |   891


Complete cases 891 from 891
Acc: 0.9719416         (Accuracy )
F1:  0.9773756         (F1 score / F-measure)
MCC: 0.9405826         (Matthew correlation coefficient)
Pre: 0.971223         (Precision)
Rec: 0.9836066         (Recall)
G:   0.9773952         (G-measure)
```

This is slightly better than before. But the difference does not look significantly better than previous. We will use a simple pre-pruning strategy is to limit the number instances in leaf nodes. We set the minimum count to 3.

```
RandomSource.setSeed(123);
Frame tr = train.mapVars("Survived,Sex,Pclass,Embarked,Age,Fare,SibS
CForest rf = CForest.newRF()
        .withClassifier(CTree.newCART().withMinCount(3))
        .withRuns(100);
cv(tr, rf);

rf.train(tr, "Survived");
CFit fit = rf.fit(test);
new Confusion(tr.var("Survived"), rf.fit(tr).firstClasses()).printSu
```

Notice that we changed the classifier used by `CForest`. This is the same classifier used by default by random forest. We do this because we customized the classifier by changing the min count parameter.

```
Cross validation 10-fold
CV fold: 1, acc: 0.855556, mean: 0.855556, se: NaN
CV fold: 2, acc: 0.853933, mean: 0.854744, se: 0.001148
CV fold: 3, acc: 0.808989, mean: 0.839492, se: 0.026429
CV fold: 4, acc: 0.808989, mean: 0.831866, se: 0.026425
CV fold: 5, acc: 0.820225, mean: 0.829538, se: 0.023470
CV fold: 6, acc: 0.820225, mean: 0.827986, se: 0.021333
CV fold: 7, acc: 0.865169, mean: 0.833298, se: 0.024016
CV fold: 8, acc: 0.808989, mean: 0.830259, se: 0.023838
CV fold: 9, acc: 0.842697, mean: 0.831641, se: 0.022680
CV fold:10, acc: 0.808989, mean: 0.829376, se: 0.022551
================
mean: 0.829376, se: 0.022551

> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
```

```
     0 | >525   24 |   549
     1 |   49 >293 |   342
 ----- |    -    - | -----
 total |  574  317 |   891


Complete cases 891 from 891
Acc: 0.9180696         (Accuracy )
F1:  0.9349955         (F1 score / F-measure)
MCC: 0.8258652         (Matthew correlation coefficient)
Pre: 0.9146341         (Precision)
Rec: 0.9562842         (Recall)
G:   0.9352273         (G-measure)
```

That had indeed some effect. However after submitting to competition we did not saw any improvement. We should look forward to engineer a little bit our features for further improvements.

Feature engineering

Feature engineering

Title feature

It is clear that we can't use directly the `"Name"` variable. This is due to the fact that names are almost unique, and that leads to a tiny generalization power. To understand that we should see that even if we learned that a passenger with a given name survived or not. We can't decide if another passenger survived, using only the name of the new passenger.

Lets inspect some of the values from `"Name"` variable.

```
SolidFrame.wrapOf(train.var("Name")).printLines(20);
```

```
                                Name
  [0]                                    "Braund, Mr. Owen Harris"
  [1]     "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
  [2]                                     "Heikkinen, Miss. Laina"
  [3]            "Futrelle, Mrs. Jacques Heath (Lily May Peel)"
  [4]                                   "Allen, Mr. William Henry"
  [5]                                           "Moran, Mr. James"
  [6]                                    "McCarthy, Mr. Timothy J"
  [7]                            "Palsson, Master. Gosta Leonard"
  [8]       "Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)"
  [9]                     "Nasser, Mrs. Nicholas (Adele Achem)"
 [10]                         "Sandstrom, Miss. Marguerite Rut"
 [11]                                "Bonnell, Miss. Elizabeth"
 [12]                          "Saundercock, Mr. William Henry"
 [13]                             "Andersson, Mr. Anders Johan"
 [14]                    "Vestrom, Miss. Hulda Amanda Adolfina"
 [15]                        "Hewlett, Mrs. (Mary D Kingcome) "
 [16]                                    "Rice, Master. Eugene"
 [17]                               "Williams, Mr. Charles Eugene"
 [18] "Vander Planke, Mrs. Julius (Emelia Maria Vandemoortele)"
 [19]                                    "Masselmani, Mrs. Fatima"
```

We notice that the names contains title information of the individual. This is valuable, but how can we benefit from that? First of all see that the format of that string is clear: space + title + dot + space. We can try to model a regular expression or we can take a simpler, but manual path. Intuition tells us that there should not be too many keys.

We build a set with known keys. After that we filter out names with known titles, and print first twenty of them. We see that we have already `"Mrs"` and `"Mr"`. Let's find others.

```
// build incrementally a set with known keys
HashSet<String> keys = new HashSet<>();
keys.add("Mrs");
keys.add("Mr");
```

```
// filter out names with known keys
// print first twenty to inspect and see other keys
train.var("Name").stream()
        .mapToString()
        .filter(txt -> {
            for(String key : keys)
                if(txt.contains(" " + key + ". "))
                    return false;
            return true;
        })
        .limit(20)
        .forEach(WS::println);
```

```
"Heikkinen, Miss. Laina"
"Palsson, Master. Gosta Leonard"
"Sandstrom, Miss. Marguerite Rut"
"Bonnell, Miss. Elizabeth"
"Vestrom, Miss. Hulda Amanda Adolfina"
"Rice, Master. Eugene"
"McGowan, Miss. Anna 'Annie'"
"Palsson, Miss. Torborg Danira"
"O'Dwyer, Miss. Ellen 'Nellie'"
"Uruchurtu, Don. Manuel E"
"Glynn, Miss. Mary Agatha"
"Vander Planke, Miss. Augusta Maria"
"Nicola-Yarred, Miss. Jamila"
"Laroche, Miss. Simonne Marie Anne Andree"
"Devaney, Miss. Margaret Delia"
"O'Driscoll, Miss. Bridget"
"Panula, Master. Juha Niilo"
"Rugg, Miss. Emily"
"West, Miss. Constance Mirium"
"Goodwin, Master. William Frederick"
```

We reduced our search and found other titles like `"Miss"`, `"Master"`. We arrive at the following set of keys:

```
HashSet<String> keys = new HashSet<>();
keys.addAll(Arrays.asList(
        "Mrs", "Mme", "Lady", "Countess", "Mr", "Sir",
        "Don", "Ms", "Miss", "Mlle", "Master", "Dr",
        "Col", "Major", "Jonkheer", "Capt", "Rev"));

Nominal title = train.var("Name").stream()
        .mapToString()
        .map(txt -> {
            for(String key : keys)
                if(txt.contains(" " + key + ". "))
                    return key;
            return "?";
        })
        .collect(Nominal.collector());
DVector.fromCount(true, title).printSummary();
```

```
     ?      Mr     Mrs    Miss Master   Don   Rev    Dr   Mme    Ms
     -      --     ---    ---- ------   ---   ---    --   ---    --
 0.000 517.000 125.000 182.000 40.000 1.000 6.000 7.000 1.000 1.000
```

We note that we exhausted training data. This is enough. It is possible that in test data to appear new titles. We will consider them missing values. That is why we return `"?"` when no matching is found. Another thing to notice is that some of the labels have few number of appearances. We will merge them in a greater category.

Another useful feature built in `rapaio` is filters. There are two types of filters: variable filters and frame filters. The nice part of frame filters is that learning algorithms are able to use frame filters naturally, in order to make feature transformations on data. This kind of filters are called *input filters* from the learning algorithm perspective. It is important that you know that input filters transforms features before train phase and also on fit phase.

We will build a learning filter to create a new feature.

```
/**
 * Frame filter which adds a title variable based on name variable
 */
class TitleFilter implements FFilter {

    private static final long serialVersionUID = -3496753631972757415

    private HashMap<String, String[]> replaceMap = new HashMap<>();
    private Function<String, String> titleFun = txt -> {
        for (Map.Entry<String, String[]> e : replaceMap.entrySet()) {
            for (int i = 0; i < e.getValue().length; i++) {
                if (txt.contains(" " + e.getValue()[i] + ". "))
                    return e.getKey();
            }
        }
        return "?";
    };

    @Override
    public void fit(Frame df) {
        replaceMap.put("Mrs", new String[]{"Mrs", "Mme", "Lady", "Cou
        replaceMap.put("Mr", new String[]{"Mr", "Sir", "Don", "Ms"});
        replaceMap.put("Miss", new String[]{"Miss", "Mlle"});
        replaceMap.put("Master", new String[]{"Master"});
        replaceMap.put("Dr", new String[]{"Dr"});
        replaceMap.put("Military", new String[]{"Col", "Major", "Jonk
        replaceMap.put("Rev", new String[]{"Rev"});
    }

    @Override
    public Frame apply(Frame df) {
        Nominal title = Nominal.empty(0, new ArrayList<>(replaceMap.k
        df.var("Name").stream().mapToString().forEach(name -> title.a
        return df.bindVars(title);
    }
}
```

Now let's try a new random forest on the reduced data set and also on title.

```
RandomSource.setSeed(123);
CForest rf = CForest.newRF()
        .withInputFilters(
                new TitleFilter(),
                new FFMapVars("Survived,Sex,Pclass,Embarked,Title")
        )
        .withClassifier(CTree.newCART().withMinCount(3))
        .withRuns(100);
cv(train, rf);

rf.train(train, "Survived");
rf.printSummary();
CFit fit = rf.fit(test);
new Confusion(train.var("Survived"), rf.fit(train).firstClasses()).pr
new Csv().withQuotes(false).write(SolidFrame.wrapOf(
        test.var("PassengerId"),
        fit.firstClasses().withName("Survived")
), root + "rf2-submit.csv");
```

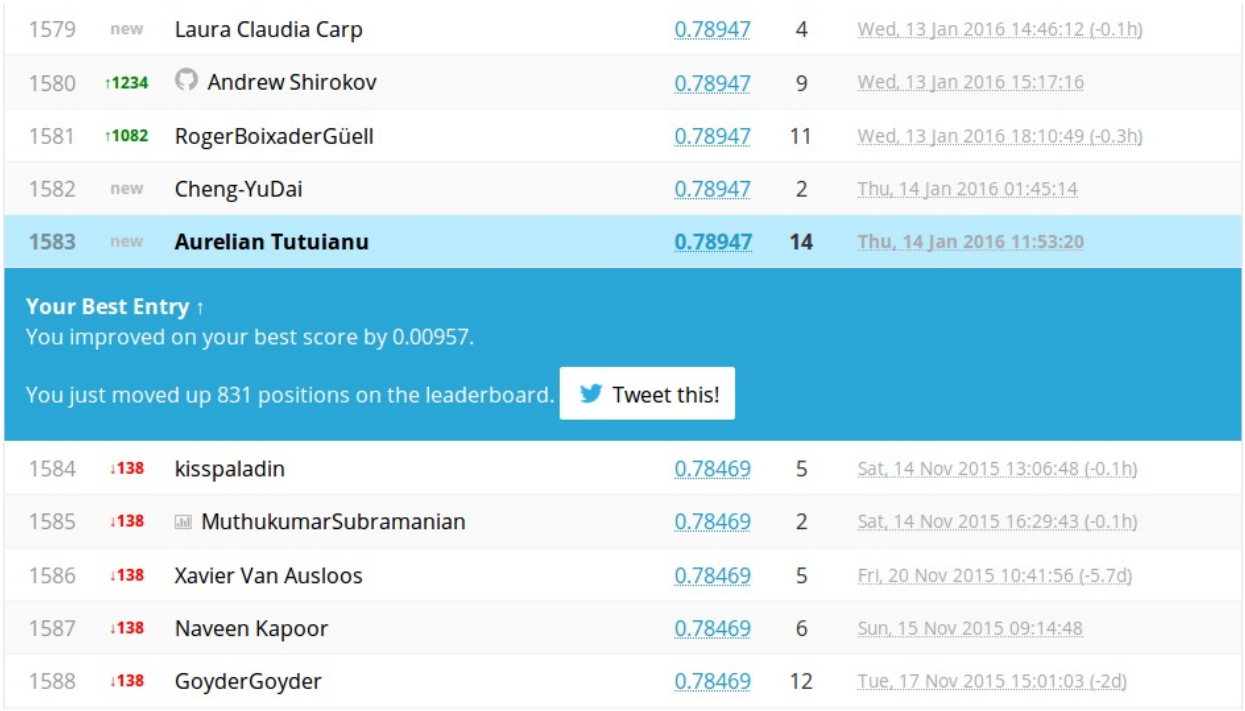
```
Cross validation 10-fold
CV fold: 1, acc: 0.833333, mean: 0.833333, se: NaN
CV fold: 2, acc: 0.831461, mean: 0.832397, se: 0.001324
CV fold: 3, acc: 0.797753, mean: 0.820849, se: 0.020024
CV fold: 4, acc: 0.820225, mean: 0.820693, se: 0.016352
CV fold: 5, acc: 0.764045, mean: 0.809363, se: 0.029023
CV fold: 6, acc: 0.820225, mean: 0.811174, se: 0.026335
CV fold: 7, acc: 0.887640, mean: 0.822097, se: 0.037593
CV fold: 8, acc: 0.820225, mean: 0.821863, se: 0.034811
CV fold: 9, acc: 0.820225, mean: 0.821681, se: 0.032567
CV fold:10, acc: 0.831461, mean: 0.822659, se: 0.030860
==================
mean: 0.822659, se: 0.030860

> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >520   29 |   549
     1 |  123 >219 |   342
 ----- |    -    - | -----
 total |  643  248 |   891


Complete cases 891 from 891
Acc: 0.8294052         (Accuracy )
F1:  0.8724832         (F1 score / F-measure)
MCC: 0.6375234         (Matthew correlation coefficient)
Pre: 0.8087092         (Precision)
Rec: 0.9471767         (Recall)
G:   0.8752088         (G-measure)
```

Now that looks definitely better than our best classifier. We submit that to kaggle to see the improvement.



1579	new	Laura Claudia Carp	0.78947	4	Wed, 13 Jan 2016 14:46:12 (-0.1h)
1580	↑1234	Andrew Shirokov	0.78947	9	Wed, 13 Jan 2016 15:17:16
1581	↑1082	RogerBoixaderGüell	0.78947	11	Wed, 13 Jan 2016 18:10:49 (-0.3h)
1582	new	Cheng-YuDai	0.78947	2	Thu, 14 Jan 2016 01:45:14
1583	new	**Aurelian Tutuianu**	0.78947	**14**	Thu, 14 Jan 2016 11:53:20

Your Best Entry ↑
You improved on your best score by 0.00957.
You just moved up 831 positions on the leaderboard. Tweet this!

1584	↓138	kisspaladin	0.78469	5	Sat, 14 Nov 2015 13:06:48 (-0.1h)
1585	↓138	MuthukumarSubramanian	0.78469	2	Sat, 14 Nov 2015 16:29:43 (-0.1h)
1586	↓138	Xavier Van Ausloos	0.78469	5	Fri, 20 Nov 2015 10:41:56 (-5.7d)
1587	↓138	Naveen Kapoor	0.78469	6	Sun, 15 Nov 2015 09:14:48
1588	↓138	GoyderGoyder	0.78469	12	Tue, 17 Nov 2015 15:01:03 (-2d)

Figure 1.8.5.1 Progress after incorporating title name into input features

Other features

There are various authors which published their work on solving this kaggle competition. Most interesting part of their work is the feature engineering section. I developed here some ideas in order to show how one can do this with the library.

Family size

Using directly `"SibSp"` and `"Parch"` fields yields no value for a random forest classifier. Studying this two features it looks like those values can be both combined into a single one by summation. This would give us a family size estimator.

In order to have an idea of the performance of this new estimator I used a ChiSquare independence test. The idea is to study if those features taken separately worth less than combined.

```
// convert sibsp and parch to nominal types to be able to use a chi-s
Nominal sibsp = Nominal.from(train.rowCount(), row -> train.label(rou
Nominal parch = Nominal.from(train.rowCount(), row -> train.label(rou

// test individually each feature
ChiSquareTest.independence(train.var("Survived"), sibsp).printSummary
ChiSquareTest.independence(train.var("Survived"), parch).printSummary

// build a combined feature by summation, as nominal
Nominal familySize = Nominal.from(train.rowCount(),
```

```
    row -> "" + (1 + train.index(row, "SibSp") + train.index(row, "Pa

// run the chi-square test on sumation
ChiSquareTest.independence(train.var("Survived"), familySize).printSt

> ChiSquareTest.independence

        Pearson’s Chi-squared test

data:
           1.0     0.0    3.0    4.0    2.0   5.0   8.0   total
     0  97.000 398.000 12.000 15.000 15.000 5.000 7.000 549.000
     1 112.000 210.000  4.000  3.000 13.000 0.000 0.000 342.000
 total 209.000 608.000 16.000 18.000 28.000 5.000 7.000 891.000


X-squared = 37.2717929, df = 6, p-value = 1.5585810465568173E-6

> ChiSquareTest.independence

        Pearson’s Chi-squared test

data:
           0.0     1.0    2.0   5.0   3.0   4.0   6.0   total
     0 445.000  53.000 40.000 4.000 2.000 4.000 1.000 549.000
     1 233.000  65.000 40.000 1.000 3.000 0.000 0.000 342.000
 total 678.000 118.000 80.000 5.000 5.000 4.000 1.000 891.000


X-squared = 27.9257841, df = 6, p-value = 9.703526421045439E-5

> ChiSquareTest.independence

        Pearson’s Chi-squared test

data:
             2       1      5       3      7      6      4     8     1
     0  72.000 374.000 12.000  43.000  8.000 19.000  8.000 6.000 7.00
     1  89.000 163.000  3.000  59.000  4.000  3.000 21.000 0.000 0.00
 total 161.000 537.000 15.000 102.000 12.000 22.000 29.000 6.000 7.00


X-squared = 80.6723134, df = 8, p-value = 3.574918139293004E-14
```

How we can interpret the result? The test says that each feature brings value separately. The *p-value* associated with both test could be considered significant. That means there are string evidence that those features are not independent of target class. As a conclusion, those features are useful. The last test is made for their summation. It looks like he test is more significant than the previous two. As a consequence we can use the summation instead of those two values taken independently.

Cabin and Ticket

It seems that cabin and ticket denominations are not useful as they are. There are some various

reasons why not to do so. First of all they have many missing values. But a stringer reason is that both have too many levels to contain solid generalization base for learning.

If we take only the first letter from each of those two fields, more generalization can happen. This is probably due to the fact that perhaps there is some localization information encoded in those. Perhaps information about the deck, the comfort level, auxiliary functions is encoded in there. As a conclusion it worth a try so we proceed with this thing.

To combine all those things we can do it in a single filter or into many filters applied on data. I chose to do a single custom filter to solve all those problems.

```
class CustomFilter implements FFilter {

    private static final long serialVersionUID = -349675363197275741

    private HashMap<String, String[]> replaceMap = new HashMap<>();
    private Function<String, String> titleFun = txt -> {
        for (Map.Entry<String, String[]> e : replaceMap.entrySet()) 
            for (int i = 0; i < e.getValue().length; i++) {
                if (txt.contains(" " + e.getValue()[i] + ". "))
                    return e.getKey();
            }
        }
        return "?";
    };

    @Override
    public void fit(Frame df) {
        replaceMap.put("Mrs", new String[]{"Mrs", "Mme", "Lady", "Cou
        replaceMap.put("Mr", new String[]{"Mr", "Sir", "Don", "Ms"})
        replaceMap.put("Miss", new String[]{"Miss", "Mlle"});
        replaceMap.put("Master", new String[]{"Master"});
        replaceMap.put("Dr", new String[]{"Dr"});
        replaceMap.put("Military", new String[]{"Col", "Major", "Jonk
        replaceMap.put("Rev", new String[]{"Rev"});
    }

    @Override
    public Frame apply(Frame df) {

        Nominal title = Nominal.empty(0, new ArrayList<>(replaceMap.k
        df.var("Name").stream().mapToString().forEach(name -> title.a

        Var famSize = Numeric.from(df.rowCount(), row ->
                1.0 + df.index(row, "SibSp") + df.index(row, "Parch"
        ).withName("FamilySize");

        Var ticket = Nominal.from(df.rowCount(), row ->
                df.missing(row, "Ticket") ? "?" : df.label(row, "Tick
                    .substring(0, 1).toUpperCase()
        ).withName("Ticket");

        Var cabin = Nominal.from(df.rowCount(), row ->
```

```
                df.missing(row, "Cabin") ? "?" : (df.label(row, "Cab
                    .substring(0, 1).toUpperCase())
        ).withName("Cabin");

        return df.removeVars("Ticket,Cabin").bindVars(famSize, ticke
    }
}
```

Another try with random forests

So we have some new features and we look to learn from them. We can use a previous classifier like random forests to test it before submit.

But we know that we are in danger to overfit is rf. One idea is to transform numeric features into nominal ones by a process named discretization. For this purpose we use a filter from the library called `FFQuantileDiscrete`. This filter computes a given number of quantile intervals and put labels according with those intervals on numerical values. Let's see how we proceed and how the data looks like:

```
FFilter[] inputFilters = new FFilter[]{
        new CustomFilter(),
        new FFQuantileDiscrete(10, "Age"),
        new FFQuantileDiscrete(10, "Fare"),
        new FFQuantileDiscrete(3, "SibSp"),
        new FFQuantileDiscrete(3, "Parch"),
        new FFQuantileDiscrete(8, "FamilySize"),
        new FFMapVars("Survived,Sex,Pclass,Embarked,Title,Age,Fare,Fa
};

// print a summary of the transformed data
train.applyFilters(inputFilters).printSummary();
```

```
> printSummary(frame, [Survived, Sex, Pclass, Embarked, Title, Age, F
Ticket, Cabin])
rowCount: 891
complete: 183/891
varCount: 10
varNames:

 0. Survived : NOMINAL |  4.      Title : NOMINAL |  8. Ticket : NOM
 1.      Sex : NOMINAL |  5.        Age : NOMINAL |  9.  Cabin : NOM
 2.   Pclass : NUMERIC |  6.       Fare : NOMINAL |
 3. Embarked : NOMINAL |  7. FamilySize : NOMINAL |

Survived           Sex           Pclass    Embarked          Title
 0 : 549    male : 577     Min. : 1.000     S : 644       Mr : 520
 1 : 342  female : 314  1st Qu. : 2.000     C : 168     Miss : 184
                         Median : 3.000     Q :  77      Mrs : 128
                           Mean : 2.309  NA's :   2   Master :  40
                        2nd Qu. : 3.000                  Dr :   7
                           Max. : 3.000                 Rev :   6
                                                    (Other) :   6
```

```
               Fare    FamilySize         Ticket          Cabin
   7.854~8.05 : 106  -Inf~1 : 537        3 : 301        C :  59
    -Inf~7.55 :  92     1~2 : 161        2 : 183        B :  47
    27~39.688 :  91     2~3 : 102        1 : 146        D :  33
    21.679~27 :  89   3~Inf :  91        P :  65        E :  32
39.688~77.958 :  89                      S :  65        A :  15
14.454~21.679 :  88                      C :  47  (Other) :   5
      (Other) : 336                 (Other) :  84     NA's : 687
```

We can see that `"Age"` values are now intervals and still 177 missing values.

The values chosen for quantile numbers is more or less arbitrary. There is no *good* numbers in general, only for some specific purposes.

As promised, we will give a try to another random forest to see if it can better generalize.

```
RandomSource.setSeed(123);

CForest model = CForest.newRF()
        .withInputFilters(inputFilters)
        .withMCols(4)
        .withBootstrap(0.7)
        .withClassifier(CTree.newCART()
            .withFunction(CTreePurityFunction.GainRatio).withMinGain
        .withRuns(200);
model.train(train, "Survived");
CFit fit = model.fit(test);
new Confusion(train.var("Survived"), model.fit(train).firstClasses()
cv(train, model);
```

I tried some ideas to make the forest to generalize better.

- Smaller bootstrap percentage - this could lead to increased independence of trees
- Use `GainRatio` as purity function because sometimes is more conservative
- Use `MinGain` to avoid growing trees to have many leaves with a single instance
- Use `mCols=4`, number of variables used for testing - more than default value, to improve the quality of each tree

```
> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >532   17 |   549
     1 |   33 >309 |   342
 ----- |    -    - | -----
 total |  565  326 |   891


Complete cases 891 from 891
Acc: 0.9438833         (Accuracy )
F1:  0.9551167         (F1 score / F-measure)
MCC: 0.880954         (Matthew correlation coefficient)
Pre: 0.9415929         (Precision)
Rec: 0.9690346         (Recall)
```

```
G:   0.9552152         (G-measure)
Cross validation 10-fold
CV fold: 1, acc: 0.833333, mean: 0.833333, se: NaN
CV fold: 2, acc: 0.786517, mean: 0.809925, se: 0.033104
CV fold: 3, acc: 0.842697, mean: 0.820849, se: 0.030099
CV fold: 4, acc: 0.898876, mean: 0.840356, se: 0.046109
CV fold: 5, acc: 0.831461, mean: 0.838577, se: 0.040129
CV fold: 6, acc: 0.842697, mean: 0.839263, se: 0.035932
CV fold: 7, acc: 0.853933, mean: 0.841359, se: 0.033267
CV fold: 8, acc: 0.764045, mean: 0.831695, se: 0.041179
CV fold: 9, acc: 0.842697, mean: 0.832917, se: 0.038694
CV fold:10, acc: 0.842697, mean: 0.833895, se: 0.036612
=================
mean: 0.833895, se: 0.036612
```

These are the results. At a first look might seem like an astonishing result. But we know that the irreducible error for this data set is high and is close to 0.2. It seems obvious that we failed to reduce the variance and we still overfit a lot using this construct. Since this is a tutorial I will not insist on improving this model, but I think that even if it would be improved, the gain would be very small. Perhaps another approach would be better.

SVM model

SVM model

SVM (Support Vector Machines) is a nice framework to test new ideas for various types of problems. The power of SMVs comes from their kernels. A kernel is basically a transformation of the original space generated by the input features into another space, often with more dimensions. It's like a feature engineering in a singe function.

But SVMs have a practical problem. The features needs to be numeric and does not allows missing values. This is not a constrain on the algorithm itself. At any moment one can build a kernel for nominal features. But the implemented ones allows only numeric non missing values and is much simpler to shape our data into this format.

How can we do that?

Data preparation

We can use a filter to impute data for missing values. The filter we use is an imputation with a classifier of imputation with a regression. The logic is the following: train a classifier from a specified set of input features to predict the field with missing values. The data set inside the filter is filtered to contain only instances with non-missing target values.

After we impute the missing values we encode nominal features into numeric features. We can accomplish this task using, again, another filter for this purpose. The name of this filter is `FFOneHotEncoding`. What it does is to create a number of numeric feature for level of the nominal variable. Than the values on these numeric variables receives the value of the indicator function. We have 1 if the level equals the numeric variable's name, 0 otherwise.

After we have numerical variables, is better to make all the variables to be in the same range. This is not a requirement for SMVs in general. The meaning is to give same weight to all the involved variables. As a side effect it makes the algorithm to run faster. This is due to the fact that the convex optimization problem has smaller chances to have a close-to-flat big surfaces.

Finally, we will remove the not used variables from the frame in order to be prepared for learning.

```
FFilter[] inputFilters = new FFilter[]{
        new CustomFilter(),

        new FFImputeWithRegression(RForest.newRF().withRuns(100),
            new VarRange("Age,Pclass,Embarked,Sex,Fare,Title"), "Age"

        new FFImputeByClassifier(CForest.newRF().withRuns(10),
            new VarRange("Embarked,Age,Pclass,Sex,Title"), "Embarked"

        new FFImputeByClassifier(CForest.newRF().withRuns(100),
            new VarRange("Age,Pclass,Embarked,Sex,Fare,Ticket"), "Ti

        new FFImputeByClassifier(CForest.newRF().withRuns(100),
            new VarRange("Age,Pclass,Embarked,Sex,Fare,Cabin"), "Cab
```

```
        new FFOneHotEncoding("Sex,Embarked,Title,Cabin"),
        new FFOneHotEncoding("Ticket"),

        new FFStandardize("all"),

        new FFRemoveVars("PassengerId,Name,SibSp,Parch")
};
train.applyFilters(inputFilters).printSummary()

Frame Summary
=============
* rowCount: 891
* complete: 891/891
* varCount: 41
* varNames:

  0.   Survived : nom | 14. Ticket.3 : num | 28.        Cabin.G : nur
  1.     Pclass : num | 15. Ticket.2 : num | 29.        Cabin.D : nur
  2.   Sex.male : num | 16. Ticket.C : num | 30.        Cabin.A : nur
  3. Sex.female : num | 17. Ticket.7 : num | 31.        Cabin.B : nur
  4.        Age : num | 18. Ticket.W : num | 32.        Cabin.F : nur
  5.       Fare : num | 19. Ticket.4 : num | 33.        Cabin.T : nur
  6. Embarked.S : num | 20. Ticket.F : num | 34.   Title.Master : nur
  7. Embarked.C : num | 21. Ticket.L : num | 35.      Title.Rev : nur
  8. Embarked.Q : num | 22. Ticket.9 : num | 36.       Title.Mr : nur
  9. FamilySize : num | 23. Ticket.6 : num | 37.     Title.Miss : nur
 10.   Ticket.A : num | 24. Ticket.5 : num | 38.       Title.Dr : nur
 11.   Ticket.P : num | 25. Ticket.8 : num | 39.      Title.Mrs : nur
 12.   Ticket.S : num | 26.  Cabin.C : num | 40. Title.Military : nur
 13.   Ticket.1 : num | 27.  Cabin.E : num |

Survived            Pclass          Sex.male        Sex.female
 0 : 549     Min. : -1.565     Min. : -1.355     Min. : -0.737     M
 1 : 342  1st Qu. : -0.369  1st Qu. : -1.355  1st Qu. : -0.737  1st (
           Median :  0.827   Median :  0.737   Median : -0.737   Medi
             Mean : -0.000     Mean : -0.000     Mean :  0.000     Me
          2nd Qu. :  0.827  2nd Qu. :  0.737  2nd Qu. :  1.355  2nd (
             Max. :  0.827     Max. :  0.737     Max. :  1.355     Ma

             Fare        Embarked.S        Embarked.C        Embarked
   Min. : -0.648     Min. : -1.632     Min. : -0.482     Min. : -0.3(
1st Qu. : -0.489  1st Qu. : -1.632  1st Qu. : -0.482  1st Qu. : -0.3(
 Median : -0.357   Median :  0.612   Median : -0.482   Median : -0.3(
   Mean :  0.000     Mean :  0.000     Mean :  0.000     Mean :  0.0(
2nd Qu. : -0.024  2nd Qu. :  0.612  2nd Qu. : -0.482  2nd Qu. : -0.3(
   Max. :  9.662     Max. :  0.612     Max. :  2.073     Max. :  3.2

         Ticket.A          Ticket.P          Ticket.S          Ticket
   Min. : -0.139     Min. : -0.280     Min. : -0.251     Min. : -0.4
1st Qu. : -0.139  1st Qu. : -0.280  1st Qu. : -0.251  1st Qu. : -0.4
 Median : -0.139   Median : -0.280   Median : -0.251   Median : -0.4
   Mean : -0.000     Mean : -0.000     Mean : -0.000     Mean : -0.0(
```

```
2nd Qu. : -0.139  2nd Qu. : -0.280  2nd Qu. : -0.251  2nd Qu. : -0.4
   Max. :  7.166     Max. :  3.563     Max. :  3.974     Max. :  2.3

       Ticket.2          Ticket.C          Ticket.7          Ticket
   Min. : -0.508     Min. : -0.261     Min. : -0.101     Min. : -0.1
1st Qu. : -0.508  1st Qu. : -0.261  1st Qu. : -0.101  1st Qu. : -0.1
 Median : -0.508   Median : -0.261   Median : -0.101   Median : -0.1
   Mean :  0.000     Mean : -0.000     Mean : -0.000     Mean :  0.0
2nd Qu. : -0.508  2nd Qu. : -0.261  2nd Qu. : -0.101  2nd Qu. : -0.1
   Max. :  1.966     Max. :  3.823     Max. :  9.894     Max. :  9.8

       Ticket.F          Ticket.L         Ticket.9          Ticket.
   Min. : -0.082     Min. : -0.067     Min. : 0.000     Min. : -0.06
1st Qu. : -0.082  1st Qu. : -0.067  1st Qu. : 0.000  1st Qu. : -0.06
 Median : -0.082   Median : -0.067   Median : 0.000   Median : -0.06
   Mean : -0.000     Mean : -0.000     Mean : 0.000     Mean : -0.00
2nd Qu. : -0.082  2nd Qu. : -0.067  2nd Qu. : 0.000  2nd Qu. : -0.06
   Max. : 12.138     Max. : 14.883     Max. : 0.000     Max. : 14.88

       Ticket.8           Cabin.C           Cabin.E           Cabin
   Min. : -0.047     Min. : -0.379     Min. : -0.615     Min. : -0.3
1st Qu. : -0.047  1st Qu. : -0.379  1st Qu. : -0.615  1st Qu. : -0.3
 Median : -0.047   Median : -0.379   Median : -0.615   Median : -0.3
   Mean : -0.000     Mean : -0.000     Mean :  0.000     Mean : -0.0
2nd Qu. : -0.047  2nd Qu. : -0.379  2nd Qu. :  1.623  2nd Qu. : -0.3
   Max. : 21.071     Max. :  2.636     Max. :  1.623     Max. :  3.3

        Cabin.A           Cabin.B           Cabin.F          Cabin.
   Min. : -0.159     Min. : -0.269     Min. : -0.538     Min. : 0.00
1st Qu. : -0.159  1st Qu. : -0.269  1st Qu. : -0.538  1st Qu. : 0.00
 Median : -0.159   Median : -0.269   Median : -0.538   Median : 0.00
   Mean :  0.000     Mean : -0.000     Mean :  0.000     Mean : 0.00
2nd Qu. : -0.159  2nd Qu. : -0.269  2nd Qu. : -0.538  2nd Qu. : 0.00
   Max. :  6.281     Max. :  3.719     Max. :  1.858     Max. : 0.00

      Title.Rev          Title.Mr        Title.Miss          Title.
   Min. : -0.082     Min. : -1.183     Min. : -0.510     Min. : -0.0
1st Qu. : -0.082  1st Qu. : -1.183  1st Qu. : -0.510  1st Qu. : -0.0
 Median : -0.082   Median :  0.844   Median : -0.510   Median : -0.0
   Mean : -0.000     Mean : -0.000     Mean : -0.000     Mean :  0.0
2nd Qu. : -0.082  2nd Qu. :  0.844  2nd Qu. : -0.510  2nd Qu. : -0.0
   Max. : 12.138     Max. :  0.844     Max. :  1.959     Max. : 11.2

  Title.Military
   Min. : -0.082
1st Qu. : -0.082
 Median : -0.082
   Mean :  0.000
2nd Qu. : -0.082
   Max. : 12.138
```

There is a lot of content. Notice that we have numerical variables for each ticket first letter, title, cabin first letter, etc.

Train a polynomial SVM

A linear kernel is a polynomial kernel with degree 1. We let the C parameter to the default value which is 1.

```
Classifier model = new BinarySMO()
        .withInputFilters(inputFilters)
        .withC(0.0001)
        .withKernel(new PolyKernel(1));
                model.train(train, "Survived");
CFit fit = model.fit(test);
new Confusion(train.var("Survived"), model.fit(train).firstClasses()
new Csv().withQuotes(false).write(SolidFrame.wrapOf(
        test.var("PassengerId"),
        fit.firstClasses().withName("Survived")
), root + "svm1-submit.csv");
cv(train, model);
```

```
> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >292  257 |   549
     1 |   41 >301 |   342
 ----- |    -    - | -----
 total |  333  558 |   891


Complete cases 891 from 891
Acc: 0.6655443         (Accuracy )
F1:  0.6621315         (F1 score / F-measure)
MCC: 0.4141426         (Matthew correlation coefficient)
Pre: 0.8768769         (Precision)
Rec: 0.5318761         (Recall)
G:   0.6829274         (G-measure)
Cross validation 10-fold
CV fold: 1, acc: 0.666667, mean: 0.666667, se: NaN
CV fold: 2, acc: 0.764045, mean: 0.715356, se: 0.068857
CV fold: 3, acc: 0.775281, mean: 0.735331, se: 0.059730
CV fold: 4, acc: 0.707865, mean: 0.728464, se: 0.050666
CV fold: 5, acc: 0.719101, mean: 0.726592, se: 0.044077
CV fold: 6, acc: 0.696629, mean: 0.721598, se: 0.041278
CV fold: 7, acc: 0.842697, mean: 0.738898, se: 0.059286
CV fold: 8, acc: 0.730337, mean: 0.737828, se: 0.054972
CV fold: 9, acc: 0.752809, mean: 0.739492, se: 0.051663
CV fold:10, acc: 0.808989, mean: 0.746442, se: 0.053437
================
mean: 0.746442, se: 0.053437
```

The results are not promising. This is better than random but it is not enough for our purpose. There are some explanations for this result. First one could be that if the space would be linear, than the original feature space would be the same as transformed. This means that a classifier as random forest would work well if the linear svm would have worked. This might not be true in general, but in this

case it looks like a good explanation. We need to be more flexible.

To increase the flexibility of the model and to allow features to interact with one another we change the degree of the polynomial kernel. This time we will use `degree=3`. Also, we use $C = 0.0001$ to allow for some errors. This parameter is the factor of the slack regularization constraints of the SVM optimization problem. The bigger the value the more is the penalty for wrong decisions. If the space would be linear separable than one can theoretically set this value as high as possible. But we know it is not. Also we know that we have plenty of irreducible error. As a consequence, it looks like we should decrease the value of this parameter.

```
Classifier model = new BinarySMO()
        .withInputFilters(inputFilters)
        .withC(0.0001)
        .withKernel(new PolyKernel(1));
                model.train(train, "Survived");
CFit fit = model.fit(test);
new Confusion(train.var("Survived"), model.fit(train).firstClasses()
new Csv().withQuotes(false).write(SolidFrame.wrapOf(
        test.var("PassengerId"),
        fit.firstClasses().withName("Survived")
), root + "svm1-submit.csv");
cv(train, model);
```

```
> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >472   77 |   549
     1 |   48 >294 |   342
 ----- |    -    - | -----
 total |  520  371 |   891


Complete cases 891 from 891
Acc: 0.8597082         (Accuracy )
F1:  0.8830683         (F1 score / F-measure)
MCC: 0.7097044         (Matthew correlation coefficient)
Pre: 0.9076923         (Precision)
Rec: 0.859745         (Recall)
G:   0.8833934         (G-measure)
Cross validation 10-fold
CV fold: 1, acc: 0.822222, mean: 0.822222, se: NaN
CV fold: 2, acc: 0.786517, mean: 0.804370, se: 0.025248
CV fold: 3, acc: 0.853933, mean: 0.820891, se: 0.033728
CV fold: 4, acc: 0.808989, mean: 0.817915, se: 0.028174
CV fold: 5, acc: 0.797753, mean: 0.813883, se: 0.026012
CV fold: 6, acc: 0.786517, mean: 0.809322, se: 0.025809
CV fold: 7, acc: 0.831461, mean: 0.812484, se: 0.025002
CV fold: 8, acc: 0.876404, mean: 0.820474, se: 0.032350
CV fold: 9, acc: 0.764045, mean: 0.814204, se: 0.035631
CV fold:10, acc: 0.820225, mean: 0.814806, se: 0.033647
================
mean: 0.814806, se: 0.033647
```

This time the results are promising. We achieved a training error which is not close to zero and the cross validation errors are close to our desired results. We definitely should try this classifier.

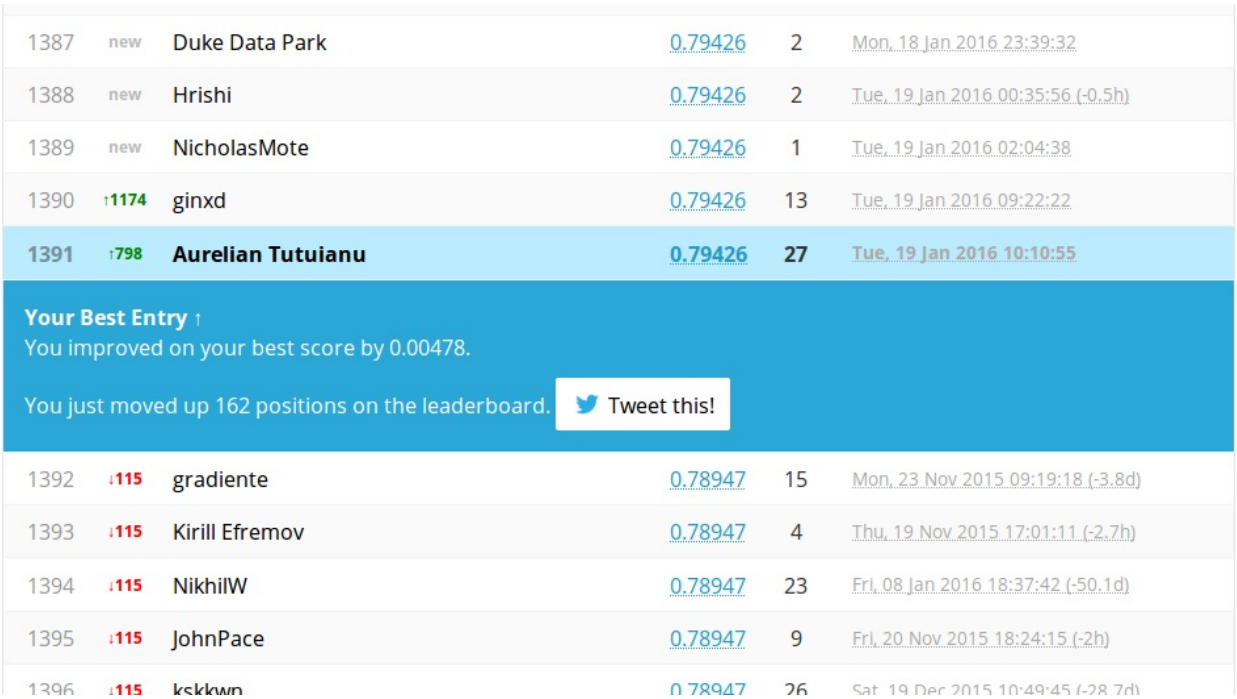


Figure 1.8.6.1 SVM1

We have a better score also on public leader board. Which is very fine. Usually in this competition a score in $0.75 - 0.78$ is fine and one in $0.78 - 0.81$ is excellent.

Tuning manually the SVM

We can work more on SVMs. One thing which deserves a try is the radial basis kernel. This is similar with working in an infinite dimensional space! We tried some `RBFKernel` approaches, but much better results gave the `CauchyKernel`. The `CauchyKernel` works in a similar way like a RBF kernel. The difference which sometimes is important is that it is a distribution with tails fatter than Gaussian distribution. This produces an effect of long distance influence. This is reasonable to use in this problem because we know we have noise. We can think that a kernel which acts on wider ranges is better if it is combined with a small value for C.

After some manual tuning we arrived at the following classifier.

```
Classifier model = new BinarySMO()
        .withInputFilters(inputFilters)
        .withC(1)
        .withTol(1e-10)
        .withKernel(new CauchyKernel(25));
```

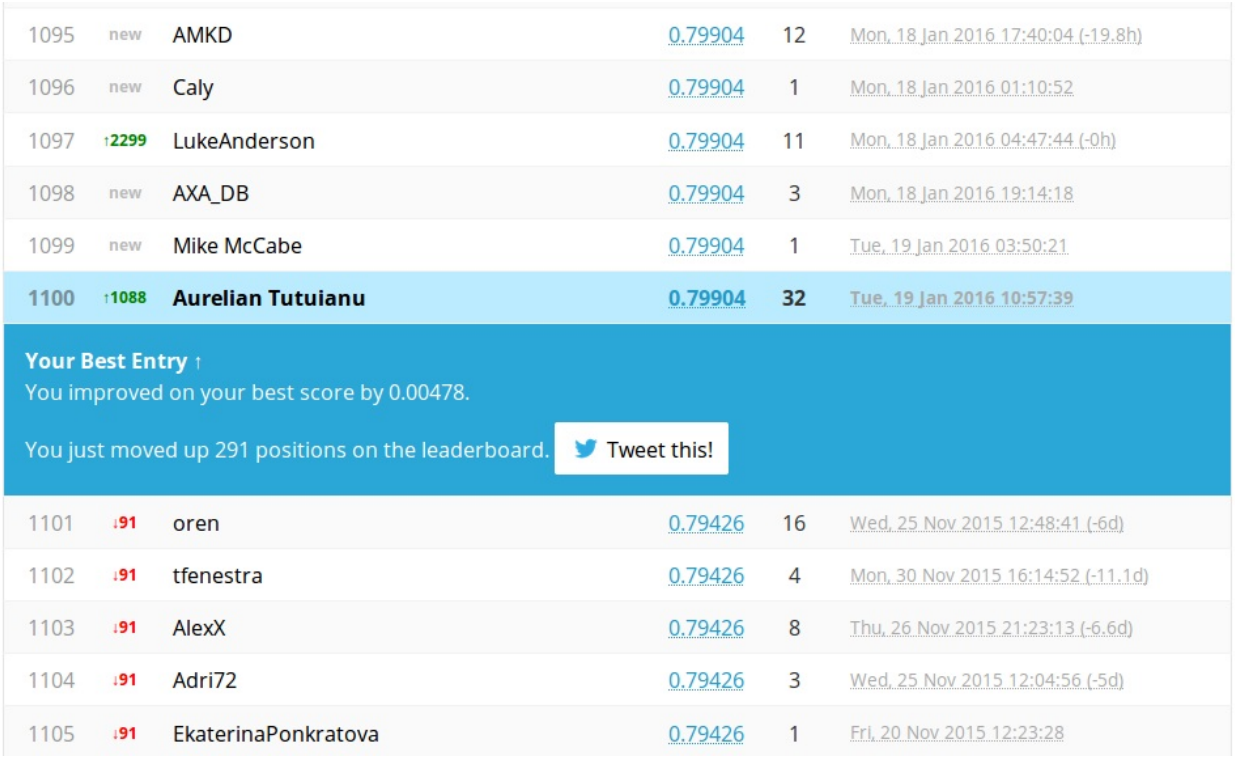
```
> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >520   29 |   549
```

```
     1 |  104 >238 |   342
 ----- |    -    - | -----
 total |  624  267 |   891


Complete cases 891 from 891
Acc: 0.8507295          (Accuracy )
F1:  0.8866155          (F1 score / F-measure)
MCC: 0.6826819          (Matthew correlation coefficient)
Pre: 0.8333333          (Precision)
Rec: 0.9471767          (Recall)
G:   0.8884334          (G-measure)
Cross validation 10-fold
CV fold: 1, acc: 0.811111, mean: 0.811111, se: NaN
CV fold: 2, acc: 0.786517, mean: 0.798814, se: 0.017391
CV fold: 3, acc: 0.865169, mean: 0.820932, se: 0.040235
CV fold: 4, acc: 0.797753, mean: 0.815137, se: 0.034836
CV fold: 5, acc: 0.831461, mean: 0.818402, se: 0.031040
CV fold: 6, acc: 0.842697, mean: 0.822451, se: 0.029481
CV fold: 7, acc: 0.820225, mean: 0.822133, se: 0.026926
CV fold: 8, acc: 0.820225, mean: 0.821895, se: 0.024937
CV fold: 9, acc: 0.797753, mean: 0.819212, se: 0.024676
CV fold:10, acc: 0.831461, mean: 0.820437, se: 0.023585
================
mean: 0.820437, se: 0.023585
```

This classifier has similar results, but there are reasons to believe that it is slightly betten than previous. The training error smaller. But we know that training error is not a good estimator. The 10 fold cv is greater. This is a good sign. A better interpretation would be that the gap between those two has shrunken and this is a good thing. A new submit on kaggle follows.



1095	new	AMKD	0.79904	12	Mon, 18 Jan 2016 17:40:04 (-19.8h)
1096	new	Caly	0.79904	1	Mon, 18 Jan 2016 01:10:52
1097	↑2299	LukeAnderson	0.79904	11	Mon, 18 Jan 2016 04:47:44 (-0h)
1098	new	AXA_DB	0.79904	3	Mon, 18 Jan 2016 19:14:18
1099	new	Mike McCabe	0.79904	1	Tue, 19 Jan 2016 03:50:21
1100	**↑1088**	**Aurelian Tutuianu**	**0.79904**	**32**	**Tue, 19 Jan 2016 10:57:39**

Your Best Entry ↑
You improved on your best score by 0.00478.

You just moved up 291 positions on the leaderboard. Tweet this!

1101	↓91	oren	0.79426	16	Wed, 25 Nov 2015 12:48:41 (-6d)
1102	↓91	tfenestra	0.79426	4	Mon, 30 Nov 2015 16:14:52 (-11.1d)
1103	↓91	AlexX	0.79426	8	Thu, 26 Nov 2015 21:23:13 (-6.6d)
1104	↓91	Adri72	0.79426	3	Wed, 25 Nov 2015 12:04:56 (-5d)
1105	↓91	EkaterinaPonkratova	0.79426	1	Fri, 20 Nov 2015 12:23:28

Figure 1.8.6.2 SVM2

Well we are really, really close to our psychological milestone of 0.8. Perhaps some tuning will give more results. This is true in general. However, next section provides you with a better approach which usually provides some gain in accuracy: stacking.

Stacking classifiers

Stacking classifiers

Using random forests or SVMs did not provided us with a result over 0.8. We have two very different types of models which performed well. For the sole purpose of prediction we use a nice ensemble technique which often provides good prediction performance gain. This technique is called stacking.

The idea behind stacking is that one can explore the space of the solutions with different approaches. Each approach (or statistical model) is basically an interpretation of the solution. But often times a proper interpretation is really hard to find. Each interpretation of the solution can have good points and weak points. The idea is to blend those interpretations into a single one in a way that we try somehow to keep what is string from each individual classifier.

A stacking classifier take some base learners and train them on training data. The results of the base learners are used as input for a stacking learner. This stacking learner is trained on the output of base learners and target variable and is finally used for prediction.

```
Classifier model = new CStacking()
        .withLearners(
                new BinarySMO()
                        .withInputFilters(inputFilters)
                        .withC(1)
                        .withTol(1e-10)
                        .withKernel(new CauchyKernel(20)),
                CForest.newRF()
                        .withInputFilters(inputFilters)
                        .withMCols(4)
                        .withBootstrap(0.07)
                        .withClassifier(CTree.newCART()
                            .withFunction(CTreePurityFunction.GainRa
                            .withMinGain(0.001))
                        .withRuns(200)
        ).withStacker(CForest.newRF().withBootstrap(0.3)
            .withRuns(200)
            .withClassifier(CTree.newCART()
                .withFunction(CTreePurityFunction.GainRatio)
                .withMinGain(0.05)));
```

Usually one uses a binary logistic regression model but it provided weak results. What looked much better is another random forrest classifier. However the stacking model uses a big value for minimum gain parameter because we want to act as an draft average over the results.

```
> Confusion

 Ac\Pr |    0    1 | total
 ----- |    -    - | -----
     0 | >522   27 |   549
     1 |   87 >255 |   342
```

```
 ----- |     -     - | -----
 total |   609   282 |   891


Complete cases 891 from 891
Acc: 0.8720539          (Accuracy )
F1:  0.9015544          (F1 score / F-measure)
MCC: 0.7281918          (Matthew correlation coefficient)
Pre: 0.8571429          (Precision)
Rec: 0.9508197          (Recall)
G:   0.902767          (G-measure)
Cross validation 10-fold
CV fold: 1, acc: 0.888889, mean: 0.888889, se: NaN
CV fold: 2, acc: 0.786517, mean: 0.837703, se: 0.072388
CV fold: 3, acc: 0.842697, mean: 0.839367, se: 0.051267
CV fold: 4, acc: 0.820225, mean: 0.834582, se: 0.042940
CV fold: 5, acc: 0.808989, mean: 0.829463, se: 0.038908
CV fold: 6, acc: 0.853933, mean: 0.833541, se: 0.036206
CV fold: 7, acc: 0.786517, mean: 0.826824, se: 0.037527
CV fold: 8, acc: 0.898876, mean: 0.835830, se: 0.043082
CV fold: 9, acc: 0.831461, mean: 0.835345, se: 0.040326
CV fold:10, acc: 0.775281, mean: 0.829338, se: 0.042500
=================
mean: 0.829338, se: 0.042500
```

Again, the results are promising. The space between training error and cross validation error is smaller and our expectations grows.

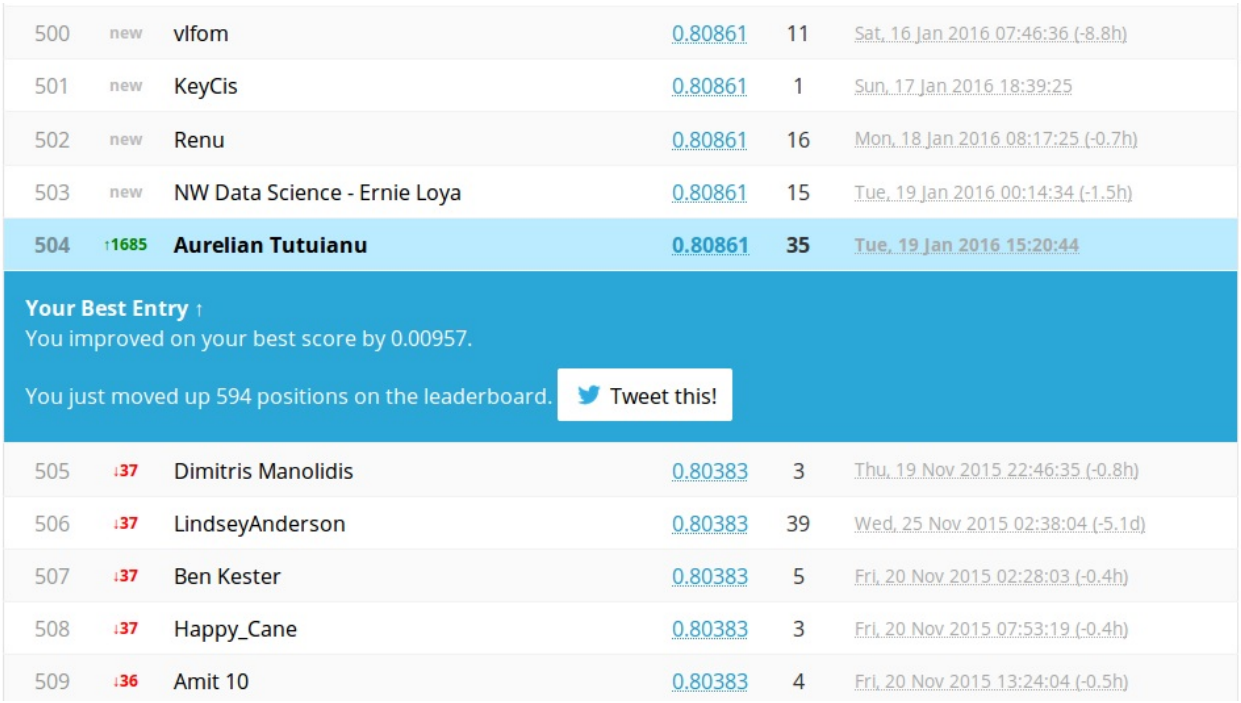


Figure 1.8.7.1 Stacking with a random forest

Finally our target performance was achieved!!

Note:

The general advice in real life is to not fight for each piece of performance measure. It really depends on the question one wants to answer. Often measures like ROC or partial ROC are much better than error frequency. We fixed this milestone because we know it is possible and because it looks like a psychological difficulty. (that 0.799 is outrageous).