Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group

Critical Systems Integration Laboratory

Data Analysis Techniques for Benchmark Evaluation

Ágnes Salánki

October 2, 2016

Contents

1 Data Analysis Techniques for Benchmark Evaluation

Author: Ágnes Salánki

1.1 How to read this document

This short syllabus summarizes the basics of a typical data analysis workflow, provides a very high-level overview of some useful R packages and briefly introduces the concepts of layered graphics and linear regression.

These are useful (and, of course, interesting) readings, however, not everything will be covered in the pre-laboratory test. Then what will be covered exactly?

1.2 Theory

- Concept of long and wide data types: it is essential to know the difference between them and know how to use the R functions to transform one into another. (You can find some details e.g. here.)
- Linear regression: what is it good for, what are the input and output of the `lm` function, how to interpret the results? (You do not have to go deep, this level is fine.)
- Basics of visualization: what are boxplots, histograms and scatterplots? (These three should be enough to complete the laboratory, however, if you prefer any other visualization form, feel free to experiment.)

1.3 Practice

Before the laboratory, you have to perform a performance test on your implementation and upload the results in a well-formed .csv file here. At the beginning of the lab (while you are working on your pre-lab tests), a collector script will create one large data frame from your files, this will be your input data set to analyze.
Without your uploaded benchmark file, you will not be allowed to start the lab.

The input files (parts from two Jane Austen novel with varying size) are here.

You should run a comparison on word granularity for each `Sense` (Sense and Sensibility) and `Pride` (Pride and Prejudice) text pair, meaning 36 (6x6) comparisons at the end (`Pride1.txt` and `Sense1.txt`, `Pride1.txt` and `Sense2.txt`, etc.). Please run each comparison independently from each other to assure the computation times to be as precise as possible.

The format of the output is theoretically the same what you used for logging, besides presenting the name of the team as a fifth column. Explicitly:

- it has to contain 5 columns, namely `kind`, `name`, `id`, `time`, `team`;
- `kind` is either "Start" or "End";
- `name` is the name of the processing node and one of the strings "Tokenize", "Collect", "ComputeScalar", "ComputeCosine";
- `id` is the the name of the document (Sense1, Pride6, etc.) or document pair (Sense1_Pride6, Sense6_Pride1, etc.) the node is working on. (For the document pairs, please follow the syntax above.)
- `time` is the output of your `System.nanoTime()` call;
- `team` is the name of your team.

This is the script which will validate your .csv log file and merge each submission together into a single large data frame while you are working on the pre-lab test. If you feel uncertain about your submission, you should test it with the script above until it does not return with an error.

If this script fails on your submission then you will not be allowed to start the laboratory.

You can find the script generating the input texts here, the example .csv file lays here.

Based on our experiences, it is possible that your solution is not fast enough to produce the results for each pair of documents. (Note that Sense6 and Pride6 contains words in order of magnitude of 10.000.) In this case please leave the time column empty (note last row in the output example) but keep the configuration information in each other column.

If you have any questions or comments about the input format feel free to reach me.

1.4 If you would like to be prepared for the laboratory

The primary coding language of this laboratory is R. In order to spend the laboratory time with meaningful analysis tasks, obtaining a basic understanding of R syntax is highly recommended before the laboratory session. If you have never used the language, please consider to gain some experience. Many good R *getting started* tutorials are available, an excellent one is the Introduction to R by Datacamp. It contains six chapters (~4 hours altogether), 1-2 and 5 are about basic variable assignment and the most important data type called data frame. Covering these chapters should be enough to start some basic analysis. Datacamp offers an interactive framework for learning R, thus, no prior installation of R is required, you will only need a browser.

1.5 If you want to be the data rock star of the class

As you will see, visualization is a key component in analytics. Any type of visualization is allowed in the lab, however, the de facto visualization framework of R is ggplot2. It is not hard to get familiar with, but, if you want to be sure to finish the lab in time, some basic ggplot2 knowledge could help.

2 Running example

A certain three-dimensional data set, simulating the results of a document comparison performance test, will appear frequently in the code snippets below. The variables to be analyzed are the type of the *similarity metric* (whether the tokenization granularity is character-level or word-level), the length of the input text and the processing time. (As you can see, this data is an artificial data set.)

```
set.seed(25)
benchmarking.results <- data.frame(task.id = c(rep("VectorizationWords", times = 100),
                                               rep("VectorizationChars", times = 100)),
                                   input.length = rpois(n = 200, lambda = 100),
                                   processing.time = c(rnorm(n = 100), rnorm(n = 100, mean = 15)))
```

2.1 Data Analysis Basics

The goal of data analysis projects is extracting data-based information about an observed system or phenomenon. In an ideal case, at the end of the project we have a deeper insight of how our system works (what is happening?) and have some hypotheses about the cause-effect relationships (why is it happening this way?).

In our case, the observed system is the implemented workflow, our goal is to analyze its performance, find the possible bottlenecks and document suggestions for code improvement (both data structures and algorithms).

The analysis usually consists of two steps: exploratory and confirmatory phases.

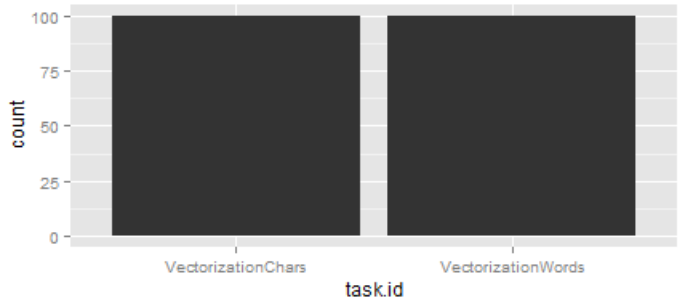
Exploratory data analysis (EDA) This phase consists of activities related to the exploration of the system. It answers the most basic questions related to the *marginal* and *joint* distributions of variables, e.g., the marginal distribution of each individual variable and basic relationships between them. Since one- and two-dimensional visualization techniques are excellent (i.e., intuitive, fast, reliable) tools for extracting this information, the exploratory analysis in practice means the plotting of, and inspection into, many graphical plots.

The most frequently used one- and two-dimensional visualization techniques are:

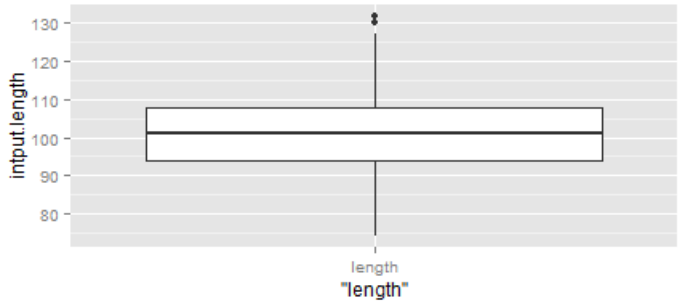
- Histograms and boxplots for one-dimensional numerical variables;
- Barcharts for one-dimensional categorical variables;
- Scatterplots for visualization of two numerical variables;
- Mosaic plots for visualization of two categorical variables;
- Colored/grouped one-dimensional plots (either histograms and boxplots) for visualization of the relationship between a categorical and a numerical variable.

For the visualization of our one- and two-dimensional distributions, we use the `qplot` (quick plot) function of the R software package `ggplot2`. For an introduction into more sophisticated visualizations, see the *Grammar of graphics* part.

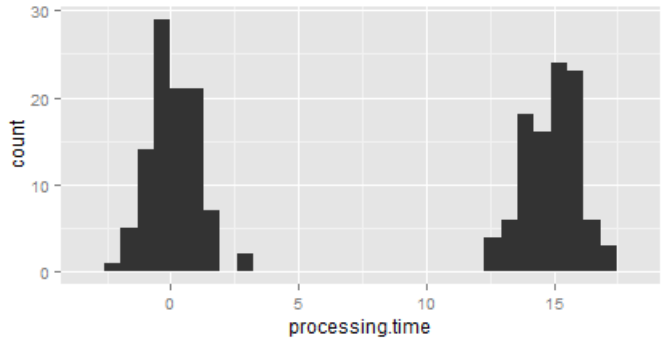
```
library(ggplot2)

benchmarking.results <- data.frame(task.id = c(rep("VectorizationWords", times = 100),
                                               rep("VectorizationChars", times = 100)),
                                   input.length = rpois(n = 200, lambda = 100),
                                   processing.time = c(rnorm(n = 100), rnorm(n = 100, mean = 15)))
## 1-dimensional distributions
qplot(data = benchmarking.results, task.id, geom = "bar")
```



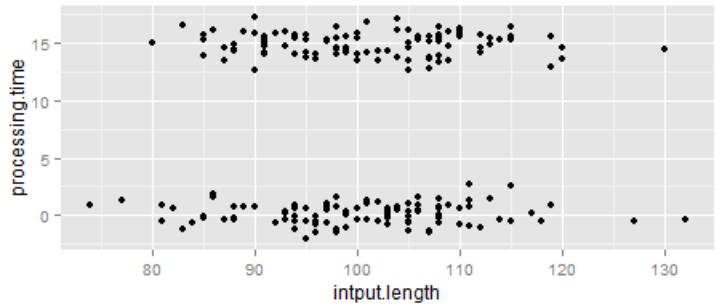

```
qplot(data = benchmarking.results, x = "length",
      y = input.length, geom = "boxplot")
```

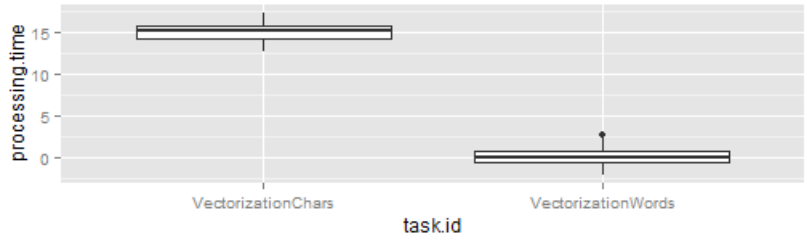



```
qplot(data = benchmarking.results, x = processing.time, geom = "histogram")
```




```
## 2D distributions: 2 numeric
qplot(data = benchmarking.results, x = input.length, y = processing.time)

## 2D distributions: 1 categorical, 1 numerical
qplot(data = benchmarking.results, x = task.id,
      y = processing.time, geom = "boxplot")
```





Confirmatory data analysis (CDA) At the end of the EDA phase, we already have some ideas, so-called hypotheses about the basic phenomenon in the system. E.g., *the distribution family of the processing time is a two-modal Gaussian.* However, to prove (or publish)

these, ad-hoc ideas are not enough, we need *statistically significant* results. This is the main task of the CDA phase. Primary tools here are the statistical tests (z-test, chi-square test, etc.).

This laboratory focuses on the exploratory phase, thus, concepts of statistical testing are not covered here.

2.2 The R Language

The R language is a popular framework, tailored to statistical analysis and visualization. With more than 7000 packages, everything can be found from the basic data manipulation to complex machine learning functions.

2.2.1 Basic Data Types

Similarly to MATLAB, R can handle vectors, lists and matrices efficiently. Its biggest advantage is its *data frame concept*, which allows collection of values with identical data types into a separate column and thus, to make it easier for developers to think in (actually Excel-like) tables.

Data frames are tables, where each column represents a vector of values of a single type, and all column vectors have the same length. For example, data frames are suitable for representing measurement results, with columns representing dimensions (statistical variables) and each row of the data frame corresponding to a data point.

In R, data frame columns can be referred by name using the `$` operator and not only with indices as we get used to it in other languages. Several packages are able to handle columns of data frames as separate objects, which can cause a much more readable and maintainable code.

E.g., the `ggplot2` code snippets above used the same concept: the first parameter of the `qplot` function is the name of the data frame and later only the column names are required:
`qplot(data = benchmarking.results, x = input.length, y = processing.time)` or
`qplot(x = benchmarking.results$input.length, y = benchmarking.results$processing.time)` and not
`qplot(data = benchmarking.results, x = benchmarking.results$input.length, y = benchmarking.results$processing.`

2.2.2 Advanced plotting: the `ggplot2` package

R has several built-in visualization functions (see `?hist()`, `?boxplot()`, `?plot()` and `?mosaicplot()` for help) and they are excellent tools for exploring the data set. However, there are packages for much more sophisticated, publication-ready charts. Packages `lattice` and `ggplot2` create static charts with many customizing functions for colors, sizes, etc. The short paragraphs below summarizes the main concepts of `ggplot2`.

The main trick of `ggplot2` charts is the layered visualization. That means that different layers are connected with a `+` sign, defining

- the underlying data
- the exact binding to it (so called geom layers) and
- the parameters responsible for the appearance (such background color, fonts, etc.).

This is the so called *Grammar of graphics* concept in information visualization.

Example: we would like to plot the relationship between the input size and processing time, separating the different tasks.

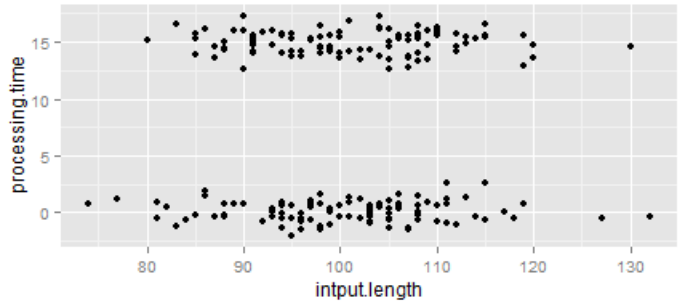
First, we declare which data we would like to use.

```
base <- ggplot(benchmarking.results)
base
> Error: No layers in plot
```

At this point, we cannot plot anything, since no layers are defined. To create a simple scatterplot, we add a point layer with the `geom_point()` function. Rule of thumb: everything inside the `aes()` function is data binding, everything else is a global setting.

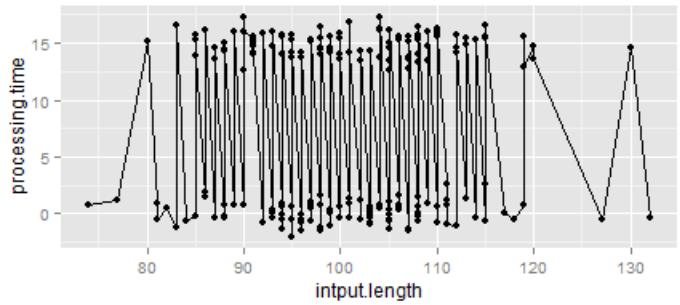
Note: several layers – e.g. points, lines, boxplot, histogram, area, density, line range – are available, for the full list, visit the ggplot2 documentation.

```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base
```



If we would like to extend our plot with another geom layer, we just concatenate them with a + sign, e.g.:

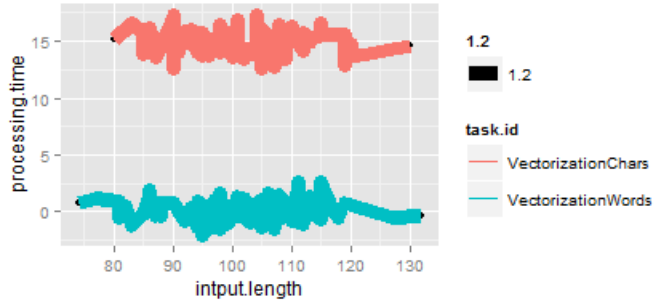
```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time))
base
```



Inside the `aes()` function we can set up the color, size, etc. of the points and lines if *it depends on the data itself*. Aesthetic features set up outside the function are valid for every represented data point.

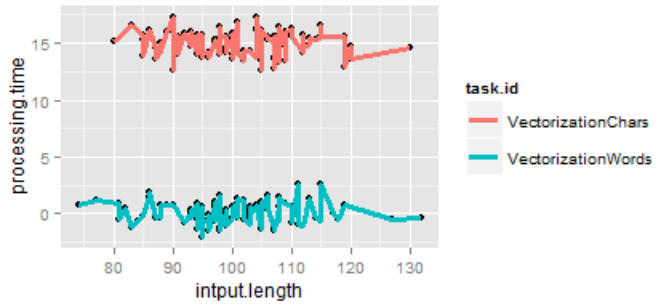
Compare the difference between

```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time,
                              col = task.id, size = 1.2))
base
```



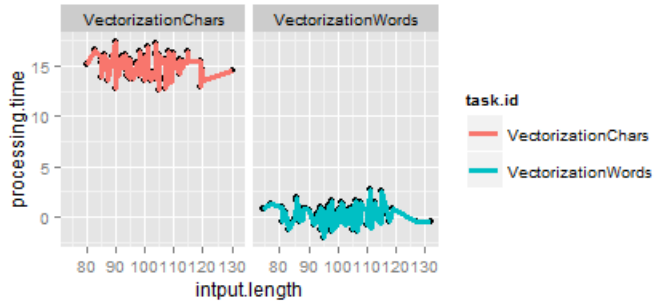
and

```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time,
                              col = task.id), size = 1.2)
base
```



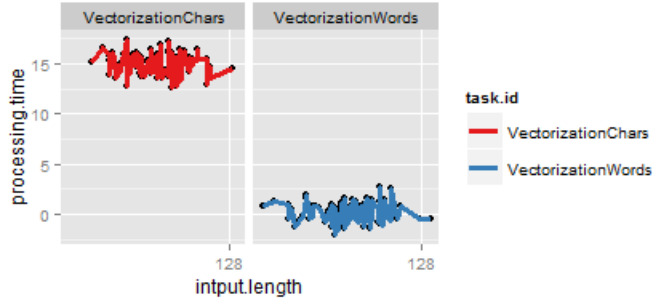
With the `facet_grid` functions, we can separate the plots along the values of a column, like this:

```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time,
                              col = task.id), size = 1.2) +
  facet_grid(. ~ task.id)
base
```



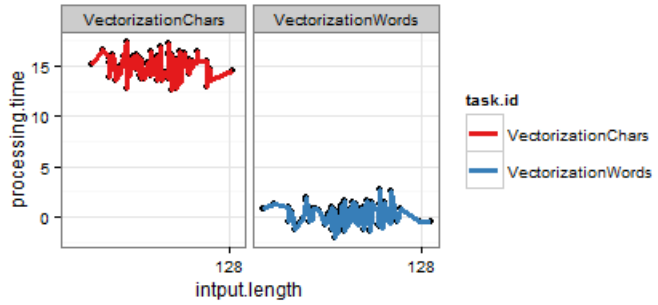
We use the `scale_*` functions to fine-tune everything inside the `aes` (including axes), e.g.

```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time,
                              col = task.id), size = 1.2) +
  facet_grid(. ~ task.id) +
  scale_color_brewer(palette = "Set1") +
  scale_x_continuous(trans = "log2")
base
```

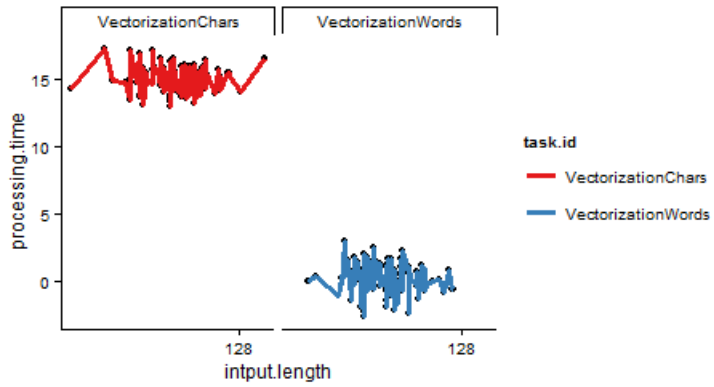


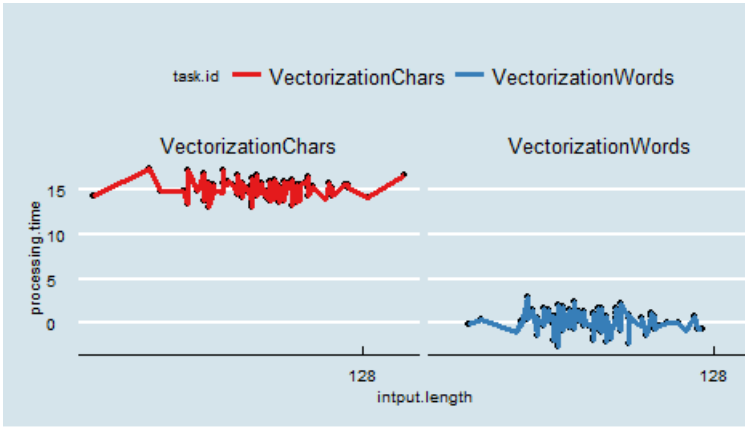
We use the `theme*` functions to fine-tune everything outside the `aes` scope, like background color, font size, etc.

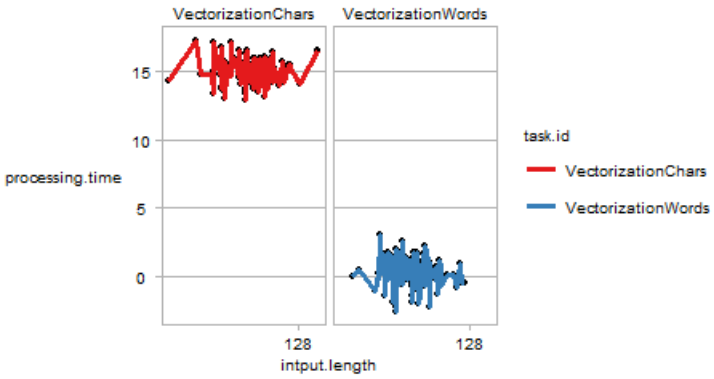
```
base <- ggplot(benchmarking.results)
base <- base + geom_point(aes(x = input.length, y = processing.time))
base <- base +  geom_line(aes(x = input.length, y = processing.time,
                              col = task.id), size = 1.2) +
  facet_grid(. ~ task.id) +
  scale_color_brewer(palette = "Set1") +
  scale_x_continuous(trans = "log2") +
  theme_bw()
base
```

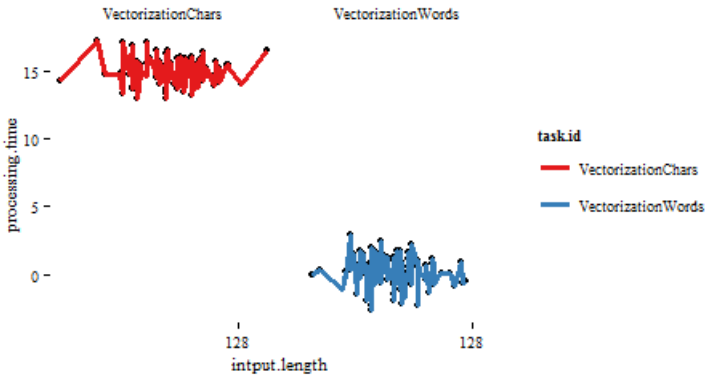


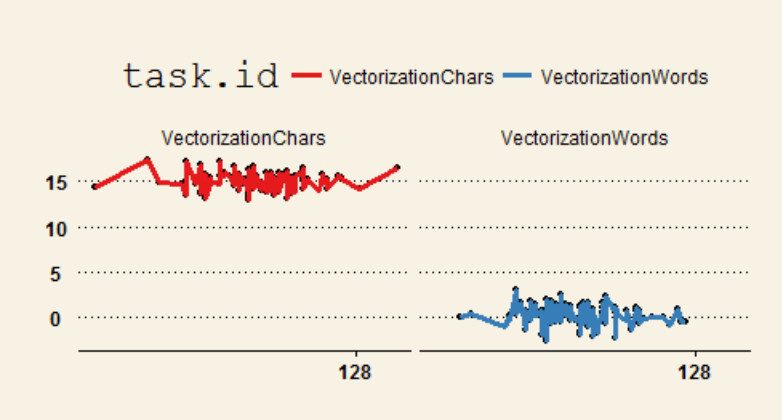
There are a couple of themes you can use.

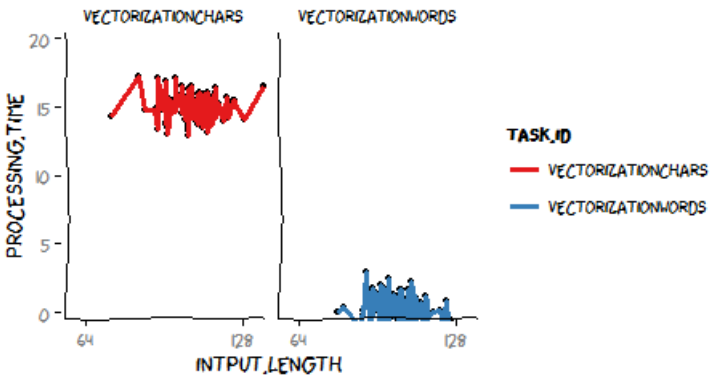












For the full list of parameters, visit the *ggplot2 documentation*.

2.2.3 Data Manipulation in R

Data representation can be *long* or *wide*.

- Long data frames have as few as possible columns, minimally three: the ID of the object, a feature name (what we have measured, observed, etc.) and the value itself.
- Wide data frames have one ID column and the features appear in separate columns.

Running example: we have run three experiments on two different documents, each of them producing two processing times (e.g., one for tokenization and one for single collection).

The results can be stored in both long and wide way:

```
> processing.times.experiments
  Document.ID Experiment Tokenize.Times ShingleCollecting.Times
1           A          1            0.5                     0.6
2           A          2            0.6                     0.5
3           A          3            0.7                     0.8
4           B          1            1.3                     1.6
5           B          2            1.2                     1.4
6           B          3            1.6                     1.2

> processing.times.long
   Document.ID Experiment                variable value
1            A          1          Tokenize.Times   0.5
2            A          2          Tokenize.Times   0.6
3            A          3          Tokenize.Times   0.7
4            B          1          Tokenize.Times   1.3
5            B          2          Tokenize.Times   1.2
6            B          3          Tokenize.Times   1.6
7            A          1 ShingleCollecting.Times   0.6
8            A          2 ShingleCollecting.Times   0.5
9            A          3 ShingleCollecting.Times   0.8
10           B          1 ShingleCollecting.Times   1.6
11           B          2 ShingleCollecting.Times   1.4
12           B          3 ShingleCollecting.Times   1.2

> processing.times.wide
  Document.ID 1_Tokenize.Times 1_ShingleCollecting.Times 2_Tokenize.Times
1           A              0.5                       0.6              0.6
2           B              1.3                       1.6              1.2
  2_ShingleCollecting.Times 3_Tokenize.Times 3_ShingleCollecting.Times
```

```
1                   0.5             0.7                   0.8
2                   1.4             1.6                   1.2
```

The long format allows the exploration of processing time values together, the wide format is ideal for computing an aggregate value over processing times (e.g., the median of times). Somewhere in between, the processing.times.experiment is good for calculating e.g. the sum of the times or for the analysis of only one attribute.

As a conclusion: there is no "ideal" data format, it should be adopted to the characteristics of the task and the variables to be emphasized.

Fortunately, there are two packages called `reshape2` and `tidyr` which allows comfortable transformations between the long and wide formats with functions: `reshape2::melt/tidyr::gather` and `reshape2::dcast/tidyr::spread`.

Their usage is quite obvious, please, check the parametrization with `?melt/?gather` and `?dcast/?spread` before the lab, since effective usage of these functions can extremely speed up the analysis process.

In our example, "melting" the wide(r) format into a long one can look like this:

```
processing.times.long <- melt(processing.times.experiment, id.vars = c("Document.ID", "Experiment"))
(processing.times.long)

processing.times.long <- processing.times.experiments %>%
  gather(variable, value, -Document.ID, -Experiment)
(processing.times.long)
```

The opposite direction could look like:

```
processing.times.wide <-
    dcast(processing.times.long, formula = Document.ID ~ Experiment + variable,
                            value.var = "value")
(processing.times.wide)

processing.times.wide <- processing.times.long %>%
  unite(Experiment_variable, Experiment, variable) %>%
  spread(Experiment_variable, value)
(processing.times.wide)
```

2.3 Regression

Regression is one of the most frequently used machine learning methods; the goal is to find a function that estimates a single numeric *target variable* we want to predict, based on given *input variables*, so that the estimate is as good as possible. Assuming that a function family (linear, exponential, etc.) has already been chosen, the main task of regression is to choose the parameters of the function (that identify a single member of the family) so that the function fits the data (estimates/predicts the target variable) as well as possible (minimizing some measure of error).

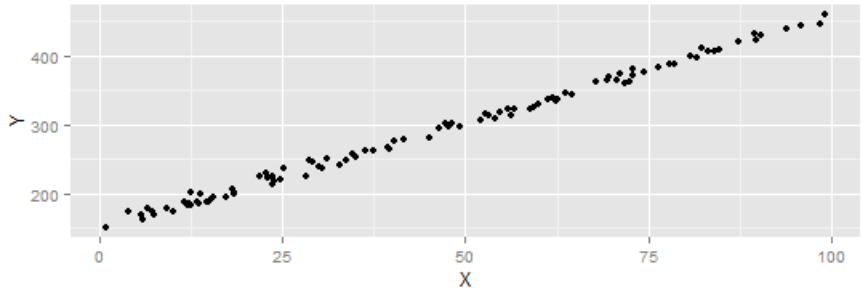
Note: choosing a function family is not in the scope of regression, it has to be performed in another way. In this example it is going to be a human task and detection will be made by visualization.

Then, what does regression do? It computes:

- the best parametrization minimizing estimation error,
- the residuals for given data points and
- the goodness of fitting – a numeric value indicating how well the function works.

Example: there are only two variables, `X` and `Y`, inspecting their scatterplot we see the following:

```
qplot(example.data$x, example.data$y, xlab = "X", ylab = "Y")
```



We suspect that there is a linear relationship between them. Considering an `y = ax + b` relationship, we can visually estimate even the parameters: `b ~ 150` (intersection of the curve and the axis `y`), `a ~ 3 ((450 - 150) / 100)`.

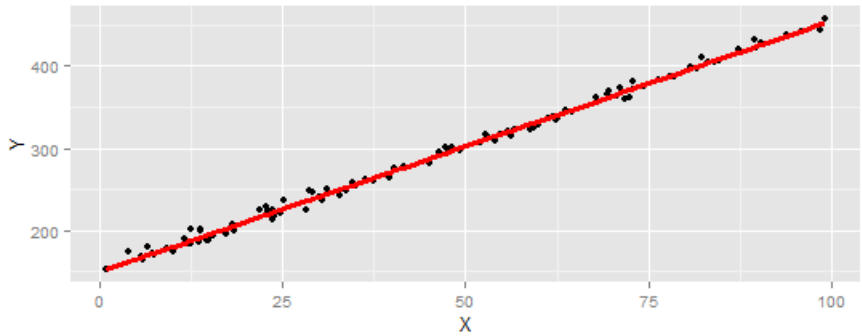
With the `lm` (*linear model*) function we can calculate the *most fitting* line (exact definition below).

```
> fit <- lm(formula = y ~ x, data = example.data)
> fit

Call:
lm(formula = y ~ x, data = example.data)

Coefficients:
(Intercept)            x
    149.173        3.063
```

The linear model object returns with the `a` and `b` coefficients: the slope of the line (in our formula `a`) is 3.063, while the intercept (in our formula `b`) is 149.173.

To plot the line with the points, we have to extract the parameters and add another layer to our plot:

```
a <- fit$coefficients[[2]]
b <- fit$coefficients[[1]]

example.data$fitted <- a  * example.data$x + b

qplot(example.data$x, example.data$y, xlab = "X", ylab = "Y") +
  geom_line(data = example.data, aes(x = x, y = fitted), size = 1.2, col = "red")
```



2.3.1 Details

The example above is simple enough to give an idea what we can expect while using the `lm` function. Below, there is a brief summary about the basic definitions of linear regression.

Assuming linear relationship between two numeric variables `X` and `Y`, we are searching for the best `(a, b)` parametrization of the line `Y = aX + b`. `X` and `Y` are variables with `(x_i, y_i)` data points in pairs. After we computed coefficients `a` and `b`, we can calculate a target `f(x_i)` value for each `x_i` data point.

- **Residual** – the difference between estimated `f(x)` and actual `y` value for a single `x_i` data point.
- **LSE** – Least Square Error is computed as the sum of residual squares: `LSE = sum(y - f(x))^2`
- **R-square values** – it is a good indicator how strong the established relationship is: it determines *how the variance of the target variable can be explained by the variance of input variables.* In case of linear regression, it is simply the (Pearson's) correlation coefficient between Y and the vector containing the predicted values.

Acceptance rules of thumb: the linear model is considered good, if

- the R-square value is high enough and
- the distribution of residuals is Gaussian.

Both criteria can be extracted from the output of the `lm` function: residuals are stored in `fit$residuals`, R-square value is presented in the `summary` (call `summary(fit)`).

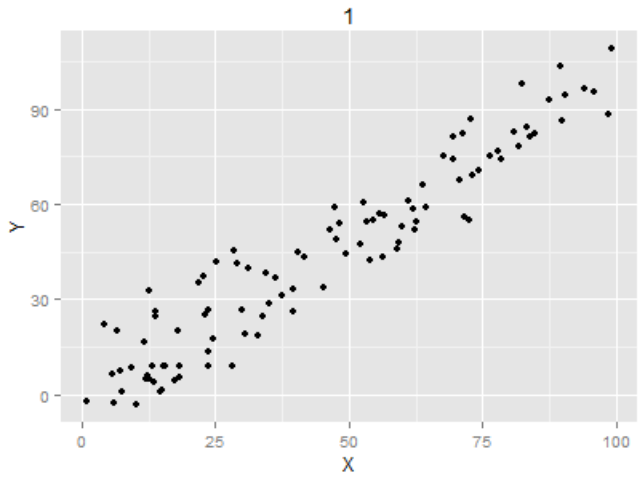
3 Questions

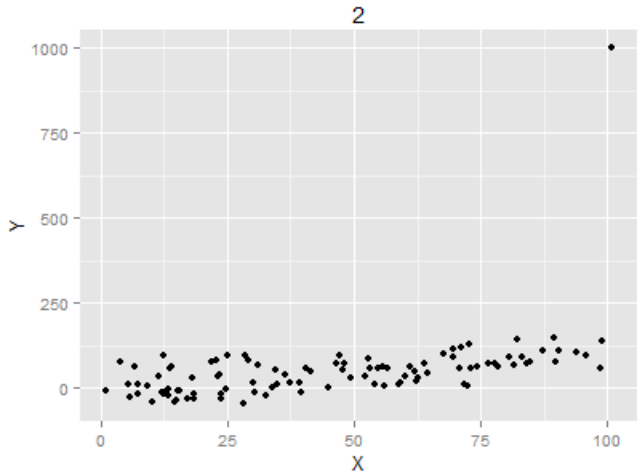
- We had three two-dimensional data sets, all of them with variables `x` and `y`. We have run a linear model fitting for all of them, these are the inputs:

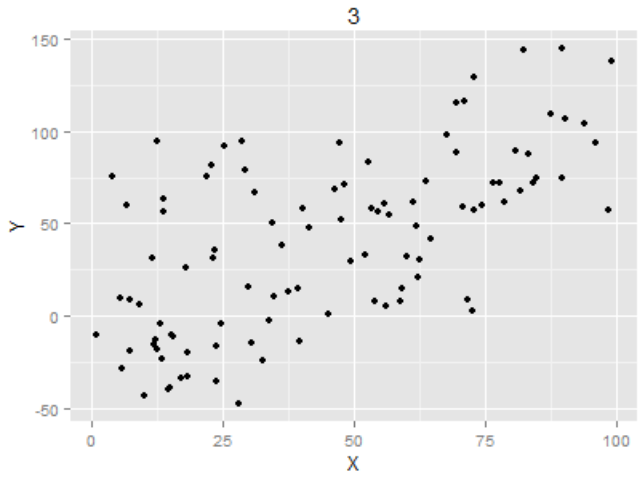
```
summary(fit.A)
Call:
lm(formula = y ~ x, data = example.data)

Residuals:
   Min     1Q Median     3Q    Max
-93.49 -36.90 -12.47  19.35 855.02

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -28.75      18.18  -1.582    0.117
x               1.72       0.34   5.060 1.94e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 95.45 on 99 degrees of freedom
Multiple R-squared:  0.2055,    Adjusted R-squared:  0.1974
F-statistic:  25.6 on 1 and 99 DF,  p-value: 1.937e-06


> summary(fit.B)

Call:
lm(formula = y ~ x, data = example.data)

Residuals:
    Min      1Q  Median      3Q     Max
-69.747 -24.894  -5.821  21.748  89.675

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -8.844      7.073  -1.250    0.214
x              1.088      0.134   8.119 1.41e-12 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 36.88 on 98 degrees of freedom
Multiple R-squared:  0.4022,    Adjusted R-squared:  0.3961
F-statistic: 65.92 on 1 and 98 DF,  p-value: 1.411e-12
```

```
> summary(fit.C)

Call:
lm(formula = y ~ x, data = example.data)

Residuals:
    Min      1Q  Median      3Q     Max
-17.437  -6.223  -1.455   5.437  22.419

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.21108    1.76835   -1.25    0.214
x            1.02204    0.03351   30.50   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9.22 on 98 degrees of freedom
Multiple R-squared:  0.9047,    Adjusted R-squared:  0.9037
F-statistic: 930.5 on 1 and 98 DF,  p-value: < 2.2e-16
```

Finally we found a scatterplot about the data sets, and we would like to match them with the outputs above.







- The data frame `cpu.data` consists of three measurements regarding CPU utilization of three hosts in a wide format:

```
> cpu.data
  Timestamp Host CPU.idle CPU.used
1         1    A       80       20
2         1    B       80       20
3         2    C       10       90
```

  How does it look in long format? Write an R code snippet for the transformation.

- The data frame `runtimes` contains runtime information for three problem types with two phases in a long format, some of the experiments were run more than once:

```
> runtimes
  Problem.Type Phase Runtime
1            A  Read       1
2            A  Read       2
3            B  Edit       3
4            B  Read       4
5            C  Edit       5
```

  We would like to transform it into a wide format indexed by problem type keeping only the median for each problem type-phase pair. How does the wide format look?

- We have a table containing information about benchmarking results: name of the team solving the problem, length of the input text in characters and the duration time in ms. How would you visualize the one-dimensional distributions for each variable?

3.1 Exercises

1. Your earned 2 points with generating an appropriate input for your analysis. Yay. (2p)
2. Write a script which will transform your .csv into a data frame containing only processing times of nodes in ms and not the output of the System.System.nanoTime() call. (1p)
3. Analyze *your own* processing times first. What is the bottleneck of your solution? How does the processing time of your solution scale? (Note that the size of the input texts increase exponentially.) How determined are the processing times among different rounds of running? (Note that you will run the Tokenize process e.g. on Sense1 6 times)(3p)
4. Read in the large .csv file containing the results of other teams and compare your times with those of other teams. (3p)
5. Document your results in an .Rmd file and upload it to your repository. (1p)

The team with the best visualization gets 1 bonus point.