Course Project 2024/2025

Denitsa Grigorova & Alexander Roiatchki

2024-12-28

Problem 1

Implement a function that simulates the Central Limit Theorem by following the instructions below:

1. The function should accept the arguments:
   - `i` - an integer which denotes the number of iterations of the simulation between 1000 and 10000
   - `n` - an even positive integer between 90 and 900 which represents the sample size being generated on every iteration
   - `p` - a real number in the range $[0, 1]$, representing probability of success and needed for the specification of the distributions in step 3. of the problem
2. Do the appropriate checks to ensure that the arguments are of the required types and in the required ranges.
3. On each iteration of the simulation:
   i) Generate a vector of random numbers from a normal distribution $\mathcal{N}(\mu = (1-p)/p, \sigma^2 = (1-p)/p^2)$ of length `n / 2`
   ii) Generate a vector of random numbers from a Geometric distribution $Ge(p)$ of length `n / 2`
   iii) Concatenate the two vectors together and shuffle them using the `sample()` function
   iv) Calculate $\frac{\bar{X}-\mu}{\sigma}\sqrt{n}$ where $\bar{X}$ is the mean of the shuffled vector. Store the value in a vector after each iteration
4. The function should return the vector of values of each iteration from iv.

Run the function with values of the arguments by your choice. What is the mean and the standard deviation of the values in the returned vector? Do a normal probability (also known as quantile-quantile or q-q) plot for the vector. Does it look like the values follow a normal distribution?

Problem 2

Supervised learning with `tidymodels`

Statistical learning deals with the statistical inference problem of finding a *predictive function* based on observed data, with the goal of better understanding the process that has generated the observations or applying the function to new observations. These predictive functions are also known as *models*, as they don't necessarily describe exactly the real process but try to approximate it in an optimal way. We call the process of finding the particular model *training* or *fitting* (since a model is 'trained' by the gathered data, or its graph is 'fit' to the data points).

Supervised learning is a specific type of statistical (and machine) learning where models are trained using labeled datasets. In its simplest form supervised learning consists of fitting a function to a training dataset

$\mathcal{D}$, that we can think of as a matrix $\mathcal{D} = [X, Y]$ which contains vectors of observations, X and Y, from two variables - one of the variables is called the *independent variable* and the other is called the *dependent variable*. In real life terms we can think of these two variables as *random variables* and denote the independent one as ξ and the dependent one as ψ, and assume the existence of a functional relationship between them denoted as $\psi = f(\xi)$. In statistical terms X is a sample of observations from ξ and Y is a sample of observations from ψ, and based on this limited data the aim is to find a model (predictive function) $\hat{f}$ that describes the real but unknown functional relationship f. Note that the existence of a functional relationship between the variables doesn't necessarily mean that there is also a causal relationship between them.

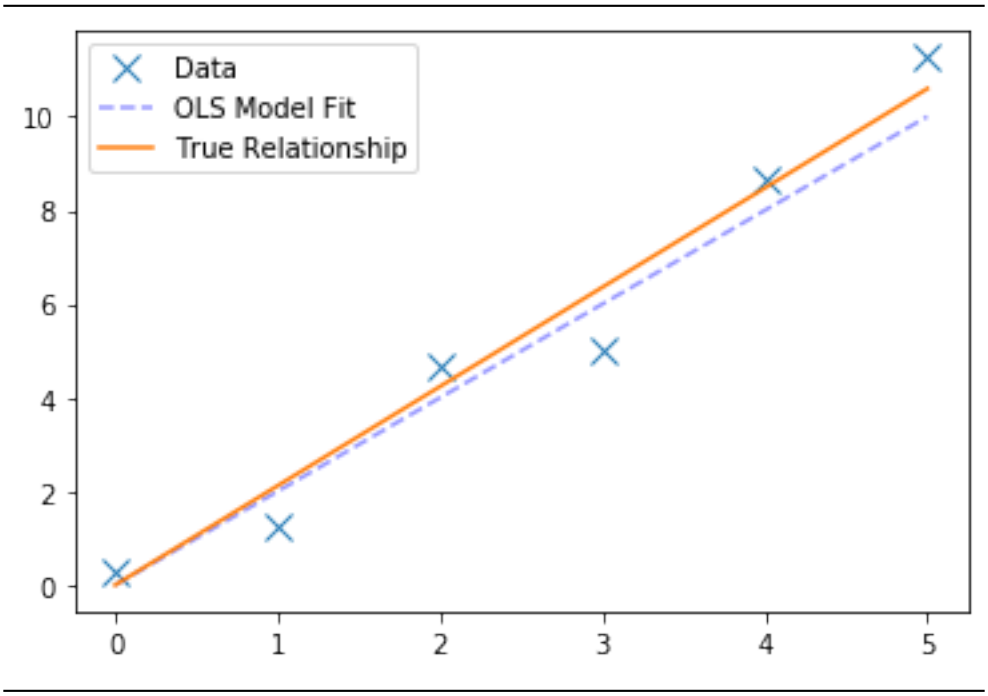
Thus, $\hat{f}$ is a model of the actual but unknown function f. We also account for the fact that measuring the observations introduces some random noise.

$$\underbrace{Y = \hat{f}(X) + \text{noise}}_{\text{statistical approximation}} \quad \sim \quad \underbrace{\psi = f(\xi)}_{\text{real life process}}$$

When the dependent variable is numeric, the model type which describes its relationship to the independent variables is called a *regression*. When X and Y are both numeric, a common way of finding the model $\hat{f}$ is by fitting a curve from a given family (such as a polynomial) to the data $\mathcal{D}$ using the *ordinary least squares (OLS)* estimator. This estimation selects the curve from the family which is 'as close as possible' to the points from the dataset by minimising the total sum of squares

$$\text{OLS} = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 = \sum_{i=1}^{n} (Y_i - \hat{f}(X_i))^2 \quad \longrightarrow \quad \min$$

where n is the sample size.

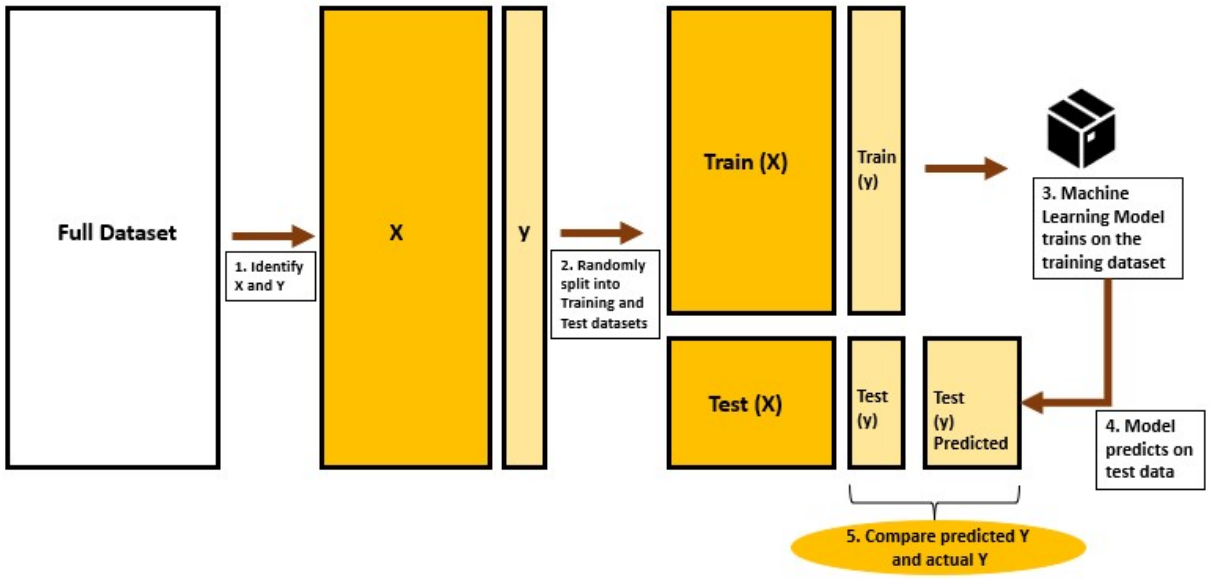


We call the vector $\hat{Y} = \hat{f}(X)$ the *fitted* (or *predicted*) values by the model $\hat{f}$ that has been estimated. The fitted values lie on the graph of the model, and the distance between them and the actual observations can be used to evaluate an *error function* that tells us how well the model approximates the actual functional

relationship. One example of an error function applied to continuous models is the root mean square error (RMSE)

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}$$

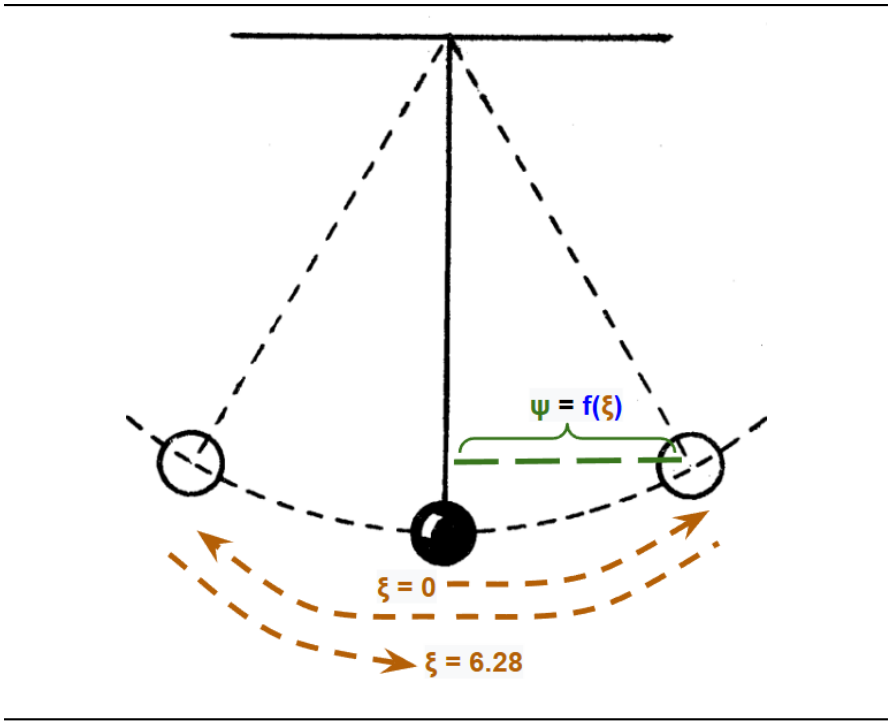
which indicates by how much on average the model $\hat{f}$ is off when used to predict values of the dependent variable based on newly observed values of the independent variable. It's important to note that model estimation (finding $\hat{f}$) is done on a portion of the dataset $\mathcal{D}$ that's commonly called the *training dataset* $\mathcal{D}_{\text{train}}$, and the model evaluation (calculating an error function) is done on another portion called the *testing dataset* $\mathcal{D}_{\text{test}}$. Most importantly, the two sets $\mathcal{D}_{\text{train}} \cap \mathcal{D}_{\text{test}} = \varnothing$ don't intersect, in order to simulate the application of the model to new 'unseen' data. A typical partitioning of the entire data $\mathcal{D}$ into training and testing sets uses the 75:25 randomised split, but this is just a rule of thumb.



Worked example:

Let's recap all of the above by considering the following synthetic example. We'll use the `tidymodels` package that provides a framework for modelling and machine learning.

1. We wish to model the relationship between a random variable ξ and a random variable ψ, assuming that there exists a function f such that $\psi = f(\xi)$, by finding an approximating function $\hat{f}$. The function f is unknown but for the sake of this example, let's imagine that $\psi = f(\xi) = \sin(\xi)$. Such a functional relationship describes, for instance, the displacement ψ of a pendulum from the axis of its lowest point as a function of time ξ. The time ξ is measured within the interval $[0, 2\pi]$ seconds.



2. We gather a sample of experimental units and for each unit record the measurements of ξ and ψ and store them in sample vectors X and Y respectively. This forms the dataset $\mathcal{D} = [X, Y]$. In this example, we can think of taking measurements of the displacement ψ at different points in time ξ and storing the data points (X_i, Y_i).

```
# x <- sample of observations from the time variable
# y <- sample of observations from the displacement variable
data <- data.frame(x, y)
```

3. We partition the full dataset $\mathcal{D}$ into training $\mathcal{D}_{\text{train}}$ and testing $\mathcal{D}_{\text{test}}$ sets. The training set will be used for estimating the model $\hat{f}$, while the testing set would be used to evaluate its performance on new data.

   Using `tidymodels`' `initial_split()` function, the random split of the full data set into training and testing sets is achieved.

```
library(tidymodels)

data_split <- initial_split(data)
train_data <- training(data_split)
test_data <- testing(data_split)
```

4. Based on a scatterplot of the data, it looks like a polynomial of degree 3 would be a good fit to model the functional dependence.

   Using `tidymodels`, since the dependent and the independent variables are both numeric, we instantiate a regression model using the `linear_reg()` function. Then we pass the model object to the `fit()` function by specifying the family of functions of which we want the model to be, in this case the 3rd degree polynomials $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$, and also provide the training data set as input. The

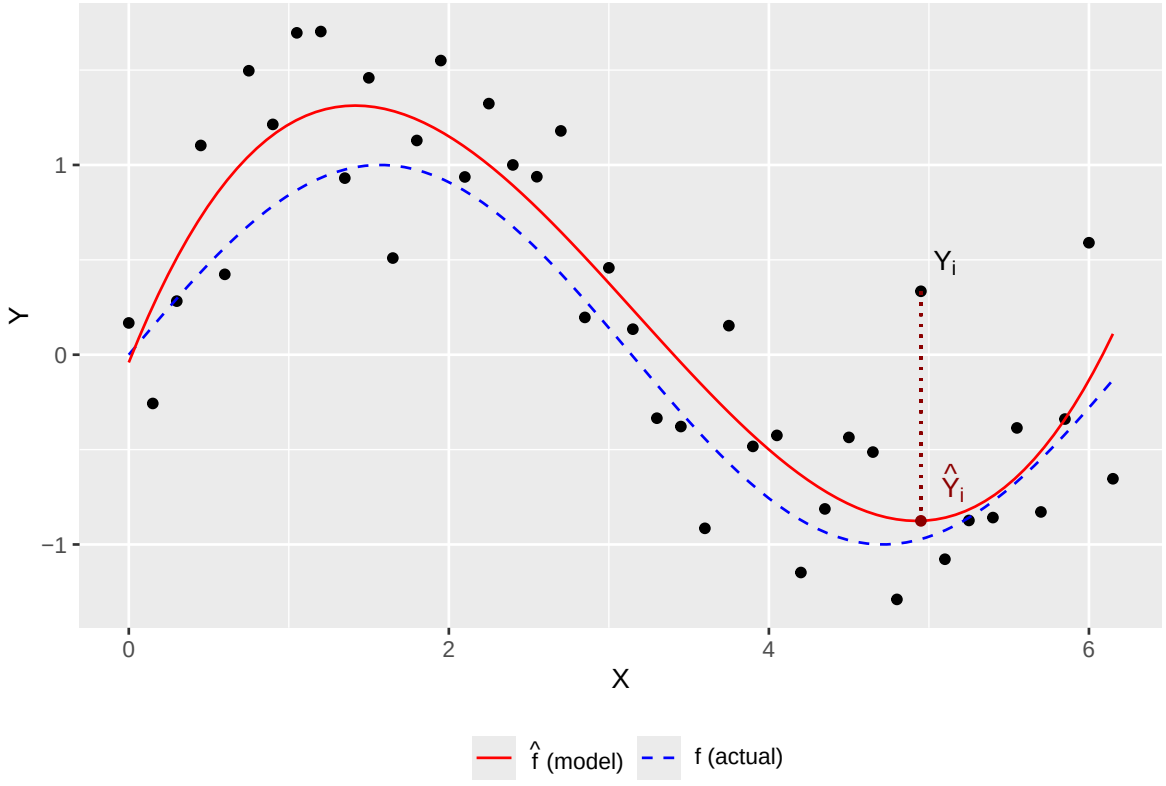
`fit()` function will estimate the coefficients $\beta_0, ..., \beta_3$ using the OLS estimator, which would mean that we've found our model $\hat{f}$.

```
lm_mod <- linear_reg()
lm_fit <- lm_mod %>%
  fit(y ~ poly(x, 3, raw=TRUE), data = train_data)
tidy(lm_fit)
```

```
## # A tibble: 4 x 5
##   term                    estimate std.error statistic      p.value
##   <chr>                      <dbl>     <dbl>     <dbl>        <dbl>
## 1 (Intercept)              -0.0408    0.249     -0.164 0.871
## 2 poly(x, 3, raw = TRUE)1   2.12      0.355      5.96  0.000000647
## 3 poly(x, 3, raw = TRUE)2  -0.963     0.135     -7.13  0.0000000167
## 4 poly(x, 3, raw = TRUE)3   0.101     0.0144     7.01  0.0000000235
```

The fitted model is the polynomial $\hat{f}(x) = -0.0408 + 2.12x - 0.963x^2 + 0.101x^3$ as seen in the `estimate` column of the table above. This approximation is valid within the range of the observations X.

The plot below depicts how the actual functional relationship f and its model $\hat{f}$ look in the context of the observed data.



5. We can see how the fitted values can sometimes differ quite a lot from the actual observations. For instance, we can use the `predict()` function on the fitted model and provide it with new data. At time $X = 4.95$ seconds, the fitted model predicts a displacement of $\hat{Y}_i = -0.88$, while the actual displacement in the data for this time is $Y_i = 0.33$.

```r
predict(lm_fit, new_data = list(x=4.95))
```

```
## # A tibble: 1 x 1
##    .pred
##    <dbl>
## 1 -0.876
```

6. Finally, we can evaluate the model's performance on the unseen data on the testing set by using the function `augment()` to create a new column `.pred` containing the predicted values $\hat{Y}_{\text{test}}$ for the vectors of new observations X_{test} and Y_{test}, and then calculating the RMSE on the testing set by using the `rmse()` function.

```r
augment(lm_fit, new_data = test_data)
```

```
## # A tibble: 42 x 4
##      .pred  .resid     x      y
##      <dbl>   <dbl> <dbl>  <dbl>
##  1 -0.0408  0.208   0     0.167
##  2  0.255  -0.512   0.15 -0.257
##  3  0.510  -0.228   0.3   0.282
##  4  0.726   0.377   0.45  1.10
##  5  0.904  -0.481   0.6   0.423
##  6  1.05    0.449   0.75  1.50
##  7  1.16    0.0559  0.9   1.21
##  8  1.24    0.459   1.05  1.70
##  9  1.29    0.416   1.2   1.70
## 10  1.31   -0.380   1.35  0.931
## # i 32 more rows
```

```r
augment(lm_fit, new_data = test_data) %>%
  rmse(y, .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.419
```

By following the process above, fit a model to the dataset `mtcars`:

1. Load the dataset `mtcars`. The aim is to fit a model $\hat{f}$ that estimates the relationship mpg $= f(\text{wt})$ between the car weight `wt` and the miles per gallon `mpg`. Which are the independent and dependent variables?
2. Read the documentation about the `initial_split()` function. Use the appropriate argument to create a split that would result in a training set of 30 samples and a testing set of 2 samples.
3. Instantiate a linear regression model and fit it on the training data. Use a simple linear relationship where `mpg ~ wt` as the formula for the `fit()` function. What family of functions would the fitted model be a member of?
4. Use the `tidy()` function on the fitted model to see the estimated coefficients for $\hat{f}$. Write down the explicit form of $\hat{f}(x)$.
5. Do a scatterplot of the training dataset and plot the estimated model on it.
6. Predict the values of `mpg` based on the unseen values from the testing set. Measure the RMSE on it.