Solution 1: HRO in `mlr3`

a) Model classes representing a certain **hypothesis** are stored in `learner` objects. Before training them on actual data, they just contain information on the functional form of f. Once a learner has been trained we can examine the parameters of the resulting model. The empirical **risk** can be assessed after training by several performance measures (e.g., based on $L2$ loss). **Optimization** happens rather implicitly as `mlr3` only acts as a wrapper for existing implementations and calls package-specific optimization procedures.

b)
```
library(mlr3)
mlr3::tsk("iris")

## <TaskClassif:iris> (150 x 5): Iris Flowers
## * Target: Species
## * Properties: multiclass
## * Features (4):
##   - dbl (4): Petal.Length, Petal.Width, Sepal.Length, Sepal.Width
```

We obtain the following information:

- `iris` is a classification task.
- It has 150 observations of 5 variables, one of which is the target.
- The target `Species` contains more than 2 classes.
- We have 4 features, all of them floating numbers (`dbl`).

c) Let's have a look at the available learners (in case you are wondering why this list is so short: there is a dedicated extension package, `mlr3learners`, that holds other learners besides these most basic ones, and there is even `mlr3extralearners`):

```
head(mlr_learners$keys())

## [1] "classif.debug"       "classif.featureless" "classif.rpart"
## [4] "regr.debug"          "regr.featureless"    "regr.rpart"
```

Let's check out the **regression tree** learner. Roughly speaking, regression trees create small, homogeneous subsets ("nodes") by repeatedly splitting the data at some cut-off (e.g., for `iris`: partition into observations with `Sepal.Width` ≤ 3 and > 3), and predict the mean target value within each final group.

```
# Inspect regression tree learner
lrn("regr.rpart")

## <LearnerRegrRpart:regr.rpart>: Regression Tree
## * Model: -
## * Parameters: xval=0
## * Packages: mlr3, rpart
## * Predict Types:  [response]
## * Feature Types: logical, integer, numeric, factor, ordered
## * Properties: importance, missings, selected_features, weights

# List configurable hyperparameters
as.data.table(lrn("regr.rpart")$param_set)
```

```
##                id    class lower upper      levels nlevels is_bounded
##  1:            cp ParamDbl     0     1                 Inf       TRUE
##  2:    keep_model ParamLgl    NA    NA  TRUE,FALSE       2       TRUE
##  3:    maxcompete ParamInt     0   Inf                 Inf      FALSE
##  4:      maxdepth ParamInt     1    30                  30       TRUE
##  5:  maxsurrogate ParamInt     0   Inf                 Inf      FALSE
##  6:     minbucket ParamInt     1   Inf                 Inf      FALSE
##  7:      minsplit ParamInt     1   Inf                 Inf      FALSE
##  8: surrogatestyle ParamInt     0     1                   2       TRUE
##  9:  usesurrogate ParamInt     0     2                   3       TRUE
## 10:          xval ParamInt     0   Inf                 Inf      FALSE
##    special_vals        default storage_type  tags
##  1:    <list[0]>           0.01      numeric train
##  2:    <list[0]>          FALSE      logical train
##  3:    <list[0]>              4      integer train
##  4:    <list[0]>             30      integer train
##  5:    <list[0]>              5      integer train
##  6:    <list[0]> <NoDefault[3]>      integer train
##  7:    <list[0]>             20      integer train
##  8:    <list[0]>              0      integer train
##  9:    <list[0]>              2      integer train
## 10:    <list[0]>             10      integer train
```

We obtain the following information:

- `regr.rpart` is a *regression* learner
- It has not been trained yet, so no model is stored.
- The underlying package is `rpart`.
- `regr.rpart` predicts *response* (unsurprisingly, but classification learners might also predict *probabilities*).
- It supports boolean, numerical and categorical features (but no date variables, for instance).
- Special properties include the ability to handle missing values and compute feature importance.
- Regarding hyperparameters, we see that some `xval` has been set (the function reference can be found at `https://cran.r-project.org/web/packages/rpart/rpart.pdf`). However, there is typically a whole bunch of configurable hyperparameters:

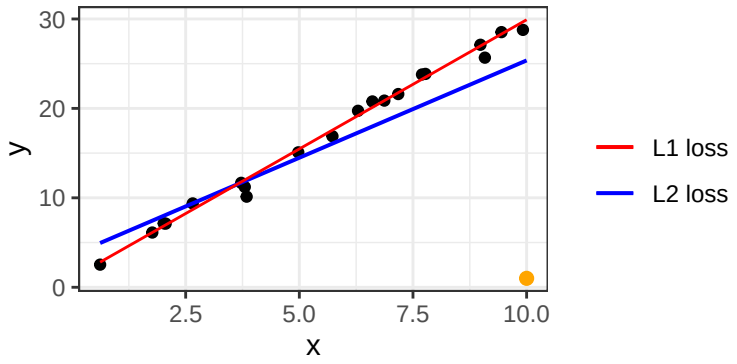
```
lrn("regr.rpart")$param_set
```

```
## <ParamSet>
##                id    class lower upper nlevels        default value
##  1:            cp ParamDbl     0     1     Inf           0.01
##  2:    keep_model ParamLgl    NA    NA       2          FALSE
##  3:    maxcompete ParamInt     0   Inf     Inf              4
##  4:      maxdepth ParamInt     1    30      30             30
##  5:  maxsurrogate ParamInt     0   Inf     Inf              5
##  6:     minbucket ParamInt     1   Inf     Inf <NoDefault[3]>
##  7:      minsplit ParamInt     1   Inf     Inf             20
##  8: surrogatestyle ParamInt     0     1       2              0
##  9:  usesurrogate ParamInt     0     2       3              2
## 10:          xval ParamInt     0   Inf     Inf             10     0
```

We might, for example, override the default of `minsplit`, which states the minimum number of observations a node must contain to be split further.

Solution 2: Loss Functions for Regression Tasks

a) $L2$ loss penalizes vertical distances to the regression line *quadratically*, while $L1$ only considers the *absolute* distance. As the outlier point lies pretty far from the remaining training data, it will have a large loss with $L2$, and the regression line will pivot to the bottom right to minimize the resulting empirical risk. A model trained with $L1$ loss is less susceptible to the outlier and will adjust only slightly to the new data.



b) The Huber loss combines the respective advantages of $L1$ and $L2$ loss: it is smooth and (once) differentiable like $L2$ but does not punish larger residuals as severely, leading to more robustness. It is simply a (weighted) piecewise combination of both losses, where ϵ marks where $L2$ transits to $L1$ loss. The exact definition is:

$$L\left(y, f(\mathbf{x})\right) = \begin{cases} \frac{1}{2}(y - f(\mathbf{x}))^2 & \text{if } |y - f(\mathbf{x})| \leq \epsilon \\ \epsilon|y - f(\mathbf{x})| - \frac{1}{2}\epsilon^2 & \text{otherwise} \end{cases}, \quad \epsilon > 0$$

In the plot we can see how the parabolic shape of the loss around 0 evolves into an absolute-value function at $|y - f(\mathbf{x})| > \epsilon = 5$.

Solution 3: Polynomial Regression

a) *Cubic* means degree 3, so our hypothesis space will look as follows:

$$\mathcal{H} = \{f(\mathbf{x} \mid \boldsymbol{\theta}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 \mid (\theta_0, \theta_1, \theta_2, \theta_3)^\top \in \mathbb{R}^4\}$$

b) The empirical risk is:

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \sum_{i=1}^{50} \left(y^{(i)} - \left[\theta_0 + \theta_1 x^{(i)} + \theta_2 \left(x^{(i)}\right)^2 + \theta_3 \left(x^{(i)}\right)^3 \right] \right)^2$$

c) We can find the gradient just as we did for an intermediate result when we derived the least-squares estimator:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) &= \frac{\partial}{\partial \boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 \\ &= \frac{\partial}{\partial \boldsymbol{\theta}} \left((\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) \right) \\ &= -2\mathbf{y}^\top \mathbf{X} + 2\boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X} \\ &= 2 \cdot \left(-\mathbf{y}^\top \mathbf{X} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X} \right) \end{aligned}$$

d) Recall that the idea of gradient descent (*descent*!) is to traverse the risk surface in the direction of the *negative* gradient as we are in search for the minimum. Therefore, we will update our current parameter set $\boldsymbol{\theta}^{[t]}$ with the negative gradient of the current empirical risk w.r.t. $\boldsymbol{\theta}$, scaled by learning rate (or step size) α:

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[t]}).$$

What actually happens here: we update each component of our current parameter vector $\boldsymbol{\theta}^{[t]}$ in the *direction* of the negative gradient, i.e., following the steepest downward slope, and also by an *amount* that depends on the value of the gradient.

In order to see what that means it is helpful to recall that the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ tells us about the effect (infinitesimally small) changes in $\boldsymbol{\theta}$ have on $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$. Therefore, gradient updates focus on influential components, and we proceed more quickly along the important dimensions.

e) We see that, for example, the first model in exercise b) fits the data fairly well but not perfectly. Choosing a more flexible function (a polynomial of higher degree or a function from an entirely different, more complex, model class) might be advantageous:

- We would be able to trace the observations more closely if our function were less smooth, and thus reduce empirical risk.

On the other hand, flexibility also has drawbacks:

- Flexible model classes often have more parameters, making training harder.
- We might run into a phenomenon called **overfitting**. Recall that our ultimate goal is to make predictions on *new* observations. However, fitting every quirk of the training observations – possibly caused by imprecise measurement or other factors of randomness/error – will not generalize so well to new data.

In the end, we need to balance model fit and generalization. We will discuss the choice of hypotheses quite a lot since it is one of the most crucial design decisions in machine learning.