ml4ec: Machine Learning for Eddy Covariance data

Benjamin Stocker

2021-08-28

Contents

Chapter 1

Set up

1.1 Apps

For this workshop, you need R and RStudio. Follow the links for downloading and installing these apps.

1.2 Libraries

Install missing packages for this tutorial.

```
list_pkgs <- c("caret", "recipes", "rsample", "tidyverse", "conflicted", "modelr", "forcats", "ya
new_pkgs <- list_pkgs[!(list_pkgs %in% installed.packages()[, "Package"])]
if (length(new_pkgs) > 0) install.packages(new_pkgs)
```

Chapter 2

Motivation

2.1 Learning objectives

After this workshop unit, you will be able to …

- Understand how overfitting models can happen and how it can be avoided.
- Implement a typical workflow using a machine learning model for a supervised regression problem.
- Evaluate the power of the model

2.2 Important points

Machine learning refers to a class of algorithms that generate statistical models of data. There are two main types of machine learning:

Unsupervised machine learning: A class of algorithms that automatically detect patterns in data without using labels or 'learning without a teacher'. Examples are: PCAs, k-means clustering, auto-encoders, self-organizing maps, etc.

Supervised machine learning: A class of algorithms that automatically learn an input-output relationships based on example input-output pairs. Examples include: support vector machines, random forests, decision trees, neural networks, etc. Supervised machine learning requires three ingredients: (1) Input data (2) Output data (3) A measure of model performance (a.k.a. "loss"). Supervised machine learning be used for regression (predict a continuous label) or classification (predict a categorical label).

Loss is a concept central to many supervised machine learning algorithms. It measures how well our predicted model values fit the actual observed model

values, a higher value indicates a higher loss, essentially meaning the model fits less well. Ideally, loss should be minimised. Loss can be used to update our model parameters in the next iteration of model training, this is called learning.

2.3 Overfitting

This example is based on this example from scikit-learn.

Machine learning (ML) may appear magical. The ability of ML algorithms to detect patterns and make predictions is fascinating. However, several challenges have to be met in the process of formulating, training, and evaluating the models. In this practical we will discuss some basics of supervised ML and how to achieve best predictive results.

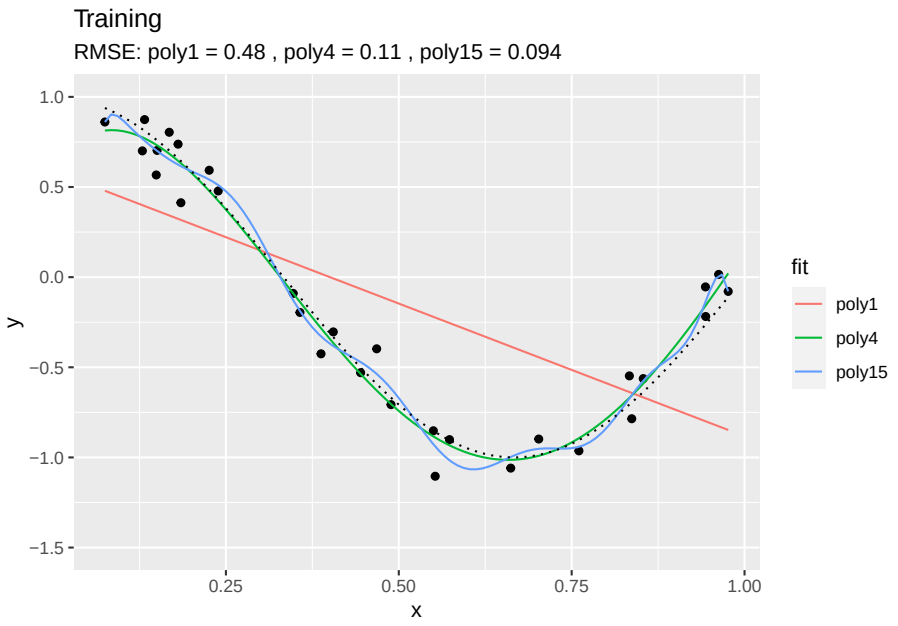
In general, the aim of supervised ML is to find a model $\hat{Y} = f(X)$ that is *trained* (calibrated) using observed relationships between a set of *features* (also known as *predictors*, or *labels*, or *independent variables*) X and the *target* variable Y. Note, that Y is observed. The hat on $\hat{Y}$ denotes an estimate. Some algorithms can even handle predictions of multiple target variables simultaneously (e.g., neural networks). ML algorithms consist of (more or less) flexible mathematical models with a certain structure and set of parameters. At the simple extreme end of the model spectrum is the univariate linear regression. You may not want to call this a ML algorithm because there is no iterative learning involved. Nevertheless, also univariate linear regression provides a prediction $\hat{Y} = f(X)$, just like other (proper) ML algorithms do. The functional form of a linear regression is not particularly flexible (just a straight line for the best fit between predictors and targets) and it has only two parameters (slope and intercept). At the other extreme end are, for example, deep neural networks. They are extremely flexible, can learn highly non-linear relationships and deal with interactions between a large number of predictors. They also contain very large numbers of parameters, typically on the order of thousands. You can imagine that this allows these types of algorithms to very effectively learn from the data, but also bears the risk of *overfitting*.

What is overfitting? The following example illustrates it. Let's assume that there is some true underlying relationship between a predictor x and the target variable y. We don't know this relationship (in the code below, this is `true_fun()`) and the observations contain a (normally distributed) error (`y = true_fun(x) + 0.1 * rnorm(n_samples)`). Based on our training data (`df_train`), we fit three polynomial models of degree 1, 4, and 15 to the observations. A polynomial of degree N is given by:

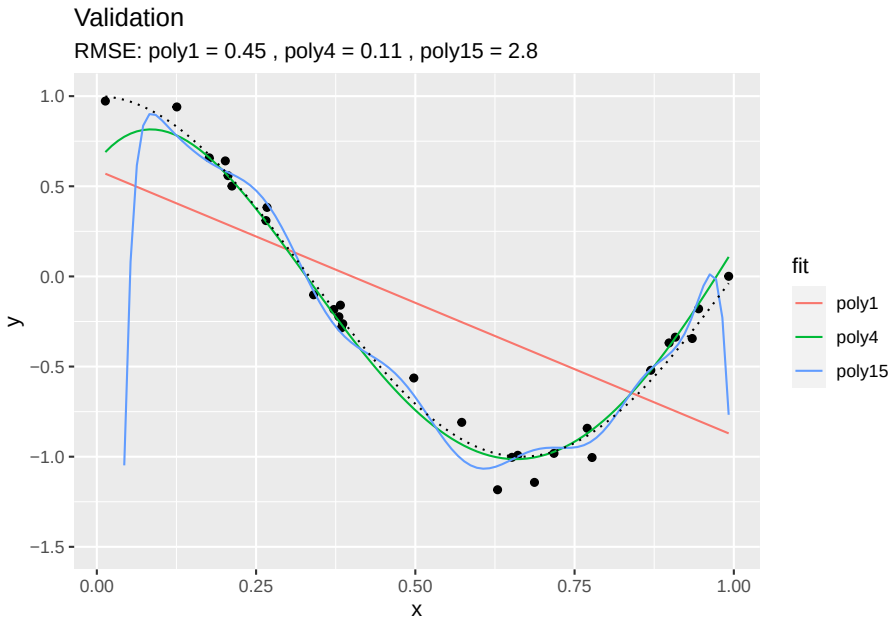
$$y = \sum_{n=0}^{N} a_n x^n$$

a_n are the coefficients, i.e., model parameters. The goal of the training is to get the coefficients a_n. From the above definition, the polynomial of degree 15

has 16 parameters, while the polynomial of degree 1 has two parameters (and corresponds to a simple linear regression). You can imagine that the polynomial of degree 15 is much more flexible and should thus yield the closest fit to the training data. This is indeed the case.

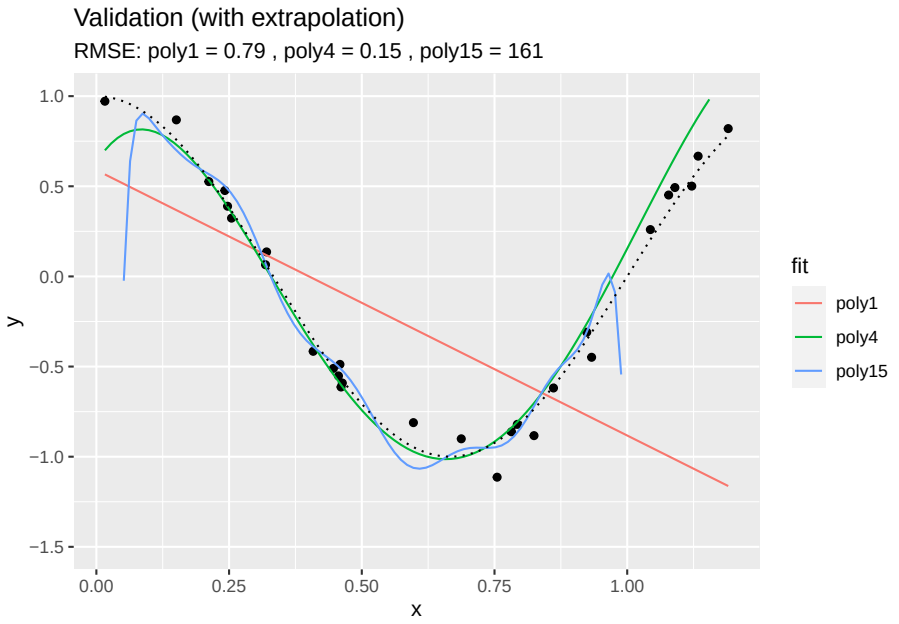


We can use the same fitted models on unseen data - the *validation data*. This is what's done below. Again, the same true underlying relationship is used, but we sample a new set of data points in x and add a new sample of errors on top of the true relationship.



You see that, using the validation set, we find that "poly4" actually performs the best - it has a much lower RMSE that "poly15". Apparently, "poly15" was overfitted. Apparently, it indeed used its flexibility to fit not only the shape of the true underlying relationship, but also the observation errors on top of it. This has obviously the implication that, when this model is used to make predictions for data that was not used for training (calibration), it will yield misguided predictions that are affected by the errors in the training set. In the above pictures we can also conclude that "poly1" was underfitted.

It gets even worse when applying the fitted polynomial models to data that extends beyond the range in x that was used for model training. Here, we're extending just 20% to the right.



You see that the RMSE for "poly15" literally explodes. The model is hopelessly overfitted and completely useless for prediction, although it looked like it fit the data best when we considered at the training results. This is a fundamental challenge in ML - finding the model with the best *generalisability*. That is, a model that not only fits the training data well, but also performs well on unseen data.

The phenomenon of fitting/overfitting as a function of the model "flexibility" is also referred to as *bias vs. variance trade-off*. The bias describes how well a model matches the training set (average error). A model with low bias will match the data set closely and vice versa. The variance describes how much a model changes when you train it using different portions of your data set. "poly15" has a high variance, but much of its variance is the result of misled training on observation errors. On the other extreme, "poly1" has a high bias. It's not affected by the noise in observations, but its predictions are also far off the observations. In ML, we are challenged to balance this trade-off. In Figure **??** you can see a schematic illustration of the bias–variance trade-off.

This chapter introduces the methods to achieve the best model generalisability and find the sweet spot between high bias and high variance. The steps to get there include the preprocessing of data, splitting the data into training and testing sets, and model training that "steers" the model towards what is considered a good model fit in terms of its generalisation power.

You have learned in video 6a about the basic setup of supervised ML, with input data containing the features (or predictors) X, predicted ($\hat{Y}$) and observed target values (Y, also known as *labels*). In video 6b (title 6c: loss and it's min-

imization), you learned about the loss function which quantifies the agreement between Y and $\hat{Y}$ and defines the objective of the model training. Here, you'll learn how all of this can be implemented in R. Depending on your application or research question, it may also be of interest to evaluate the relationships embodied in $f(X)$ or to quantify the importance of different predictors in our model. This is referred to as *model interpretation* and is introduced in the respectively named subsection. Finally, we'll get into *feature selection* in the next Application session.

The topic of supervised machine learning methods covers enough material to fill two sessions. Therefore, we split this part in two. Model training, implementing the an entire modelling workflow, model evaluation and interpretation will be covered in the next session's tutorial (Supervised Machine Learning Methods II).

Of course, a plethora of algorithms exist that do the job of $Y = f(X)$. Each of them has its own strengths and limitations. It is beyond the scope of this course to introduce a larger number of ML algorithms. Subsequent sessions will focus primarily on Artificial Neural Networks (ANN) - a type of ML algorithm that has gained popularity for its capacity to efficiently learn patterns in large data sets. For illustration purposes in this and the next chapter, we will briefly introduce two simple alternative "ML" methods, linear regression and K-nearest-neighbors. They have quite different characteristics and are therefore great for illustration purposes in this chapter.

2.4 Our modelling task

xxxx

Building a ML model is an iterative process and it is typically not possible to know which ML algorithm will perform best when you start with the process. It is also essential that you understand your data in order to apply appropriate pre-processing steps, splitting your data wisely into training and testing sets, building a robust model, and achieving an informative model evaluation. This workshop introduces essential methods of the modelling process that allow you to walk a safe (and hopefully still spectacular) path. This workshop is inspired by the fantastic tutorial *Hands On Machine Learning in R* by Boehmke & Gladwell.

xxx

2.5 Motivation

Ecosystem-atmosphere exchange fluxes of water vapour and CO2 are continuously measured at several hundred of sites, distributed across the globe. The

oldest running sites have been recording data since over twenty years. Thanks to the international FLUXNET initiative, these time series data are made openly accessible from over hundred sites and provided in a standardized format and complemented with measurements of several meteorological variables, plus soil temperature and moisture, measured in parallel. These data provide an opportunity for understanding ecosystem fluxes and how they are affected by environmental covariates. The challenge is to build models that are sufficiently generalisable in space. That is, temporally resolved relationships learned from one subset of sites should be used effectively to predict time series, given environmental covariates, at new sites (spatial upscaling). This is a challenge as previous research has shown that relatively powerful site-specific models can be trained, but predictions to new sites have been found wanting. This may be due to site-specific characteristics (e.g. vegetation type) that have thus far not been satisfactorily encoded in models. In other words, factors that would typically be regarded as random factors in mixed effects modelling, continue to undermine effective learning in machine learning models.

2.6 Data

Data is provided here at daily resolution from a selection of sites ($N = 71$, 265,177 data points, see `prepare_data.Rmd`), and paired with satellite data of the fraction of absorbed photosynthetically active radiation (fAPAR, product MODIS FPAR). This provides crucial complimentary information about vegetation structure and seasonally varying green foliage cover, responsible for photosynthesis and transpiration. The target variable is ecosystem photosynthesis, referred to as gross primary production (variable `GPP_NT_VUT_REF`). Available covariates are briefly described below. For more information, see FLUXNET 2015 website, and Pastorello et al., 2020, and the document `variable_codes_FULLSET_20160711.pdf` provided in this repository.

The data is provided through this repository (`data/ddf_combined_mlflx.csv`).

2.6.1 Available variables

- `sitename`: FLUXNET standard site name
- `date`
- `fpar_loess`: fraction of absorbed photosynthetically active radiation, interpolated to daily values using LOESS.
- `fpar_linear`: fraction of absorbed photosynthetically active radiation, linearly interpolated to daily values
- `TA_F`: Air temperature. The meaning of suffix `_F` is described in Pastorello et al., 2020.
- `SW_IN_F`: Shortwave incoming radiation
- `LW_IN_F`: Longwave incoming radiation

- `VPD_F`: Vapour pressure deficit (relates to the humidity of the air)
- `PA_F`: Atmospheric pressure
- `P_F`: Precipitation
- `WS_F`: Wind speed
- `USTAR`: A measure for atmospheric stability
- `CO2_F_MDS`: CO2 concentration
- `GPP_NT_VUT_REF`: Gross primary production - **the target variable**
- `NEE_VUT_REF_QC`: Quality control information for `GPP_NT_VUT_REF`. Specifies the fraction of high-quality underlying high-frequency data from which the daily data is derived. 0.8 = 80% underlying high-quality data, remaining 20% of the high-frequency data is gap-filled.
- `SWC_F_MDS_*`: Soil water content. The number provides information about the soil layer, counting from the top
- `wscal` : Water scalar. 0 = dry, at 100% water holding capacity of the soil; 1 = wet, at 100% water holding capacity of the soil

The following variables are provided in the dataset as well and are site-specific meta data, taken from Falge et al.

- `lon`, `lat`: Longitude and latitude (decimal degrees)
- `elv` : Elevation (m above sea level)
- `classid` : Vegetation type. Has the following codes:
  - `ENF` : Evergreen needle-leaved forest
  - `WSA` : Woody savannah
  - `GRA` : Grassland
  - `SAV` : Savannah
  - `WET`: Wetland
  - `EBF`: Evergreen broadleaved forest
  - `MF`: Mixed forest (consisting of deciduous and evergreen broadleaved and needle-leaved trees)
  - `CRO` : Cropland
  - `DBF` : Deciduous broadleaved forest
  - `CSH`: Closed shrubland
- `c4` : Whether vegetation is present that follows the C4 photosynthetic pathway (as opposed to the C3 pathway). This should have effects of the functional relationships.
- `whc` : Total rooting zone water storage capacity. IGNORE THIS.
- `koeppen_code` : Classification of the climate at the site, following Koeppen-Geiger (see e.g. here for an explanation, and Beck et al., 2018 for a global map)
- `plant_functional_type`: is missing. IGNORE THIS.

2.7 The modelling challenge

In this workshop, we formulate a model for predicting ecosystem gross primary production (photosynthesis) from environmental covariates. `GPP_NT_VUT_REF ~`, selecting predictors as suitable. Conceive the model training such that validation and test sets are split along sites (data from a given site must not be both in the test and training sets). Demonstrate the spatial generalisability of the trained model.

This is to say that `GPP_NT_VUT_REF` is the target variable, and other available variables can be used as predictors.

This challenge is on the one hand to model and predict time series across space with the aim of "**spatial upscaling**". On the other hand, we're interested in the functional relationships of GPP to the different environmental covariates separately (**partial dependence**).

2.7.1 Some peculiarities of the data

- Hierarchical structure: site → date (multiple dates for each site). Th
- Commonalities between sites exist and may be useful for modeling. Site meta data is available in a separate data frame (`data/df_sites.csv`).
- The data points may not be *iid.* That is: there may be an explicit temporal component to the data. The underlying reason is that the photosynthetic machinery "acclimates" to changes in the environment over the course of a year (high radiation in summer, low temperatures in winter). The effect is that the functional relationships between GPP and the different environmental covariates may not be stationary, i.e., the sensitivity may change over the course of a year.

Chapter 3

Data splitting

3.1 Reading and wrangling data

Let's start by reading our data and apply few processing steps (*wrangling*).

There is a difference between data wrangling and pre-processing as part of the modelling workflow, which we will learn about in Chapter 4. Data wrangling can be considered to encompass the steps to prepare the data set prior to modelling, including, combining variables from different sources, removal of bad or missing data, and aggregating to the desired resolution or granularity (e.g., averaging over all time steps in a day, or over all replicates in a sample). See the Quartz Guide to Bad Data for an overview of how to deal with different types of bad data.

In contrast, *pre-processing* refers to the additional steps that are either required by the ML algorithm (examples will be given below) or the transformation of variables guided by the resulting improvement of the predictive power of the ML model. In other words, pre-processing is part of the modelling workflow and includes all steps that apply transformations that use parameters from the data.

We are provided with a data file in the format of comma-separated-values (CSV), obtained through FLUXNET2015. It contains data from one site (CH-Lae) at a daily time step, and includes quality control information for each variable. Variables and how they are derived is explained in detail in Pastorello et al. (2020).

Let's read the data, select relevant variables, convert the time stamp column to a time object and interpret missing values (encoded -9999 in the file).

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v tibble  3.1.2     v purrr   0.3.4
## v readr   2.0.0     v stringr 1.4.0
```

```
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
## x yardstick::mae()  masks modelr::mae()
## x yardstick::mape() masks modelr::mape()
## x yardstick::rmse() masks modelr::rmse()
## x readr::spec()     masks yardstick::spec()
```

```
ddf <- read_csv("./data/FLX_CH-Dav_FLUXNET2015_FULLSET_DD_1997-2014_1-3.csv") %>%

  ## select only the variables we are interested in
  select(starts_with("TIMESTAMP"),
         ends_with("_F"),   # all meteorological variables
         GPP_NT_VUT_REF,
         NEE_VUT_REF_QC,
         -contains("JSB")) %>%

  ## convert to a nice date object
  mutate(TIMESTAMP = lubridate::ymd(TIMESTAMP)) %>%

  ## set all -9999 to NA
  na_if(-9999) %>%

  ## drop QC variables (no longer needed), except NEE_VUT_REF_QC
  select(-ends_with("_QC"), NEE_VUT_REF_QC)
```

```
## Rows: 6574 Columns: 334
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (334): TIMESTAMP, TA_F_MDS, TA_F_MDS_QC, TA_F_MDS_NIGHT, TA_F_MDS_NIGHT_...
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

If the style of the code above looks unfamiliar - this is the **tidyverse**. The aim of the tidyverse is to have a collection of R functions and packages that share a common syntax style and structure of arguments and function return values. This enables ease with typical data wrangling and visualization tasks (**ggplot2**

is part of it). This tutorial is generally written using tidyverse packages and code syntax.

The column `NEE_VUT_REF_QC` provides information about the fraction of gap-filled half-hourly data used to calculate daily aggregates. Let's remove `GPP_NT_VUT_REF` data, where more than 80% of the half-hourly data was gap-filled. Make sure to not actually remove the respective rows, but rather replace values with NA.

```
ddf <- ddf %>%
  mutate(GPP_NT_VUT_REF = ifelse(NEE_VUT_REF_QC < 0.8, NA, GPP_NT_VUT_REF))
```

At this stage, we won't use `NEE_VUT_REF_QC` any longer. So we can drop it.

```
ddf <- ddf %>%
  select(-NEE_VUT_REF_QC)
```

3.2 Splitting into testing and training sets

The introductory example impressively demonstrated the importance of validating the fitted model with data that was not used for training. Thus, we can test the model's generalisability. The essential step that enables us to assess the model's *generalization error* is to hold out part of the data from training, and set it aside (leaving it absolutely untouched) for *testing*.

There is no fixed rule for how much data are to be used for training and testing, respectively. We have to balance the trade-off between:

- Spending too much data for training will leave us with too little data for testing and the test results may not be robust. In this case, the sample size for getting robust validation statistics is not sufficiently large and we don't know for sure whether we are safe from an over-fit model.
- Spending too much data for validation will leave us with too little data for training. In this case, the ML algorithm may not be successful at finding real relationships due to insufficient amounts of training data.

Typical splits are between 60-80% for training. However, in cases where the number of data points is very large, the gains from having more training data are marginal, but come at the cost of adding to the already high computational burden of model training.

In environmental sciences, the number of predictors is often smaller than the sample size ($p < n$), because its typically easier to collect repeated observations of a particular variable than to expand the set of variables being observed.

Nevertheless, in cases where the number p gets large, it is important, and for some algorithms mandatory, to maintain $p < n$ for model training.

An important aspect to consider when splitting our data is to make sure that all "states" of the system for which we have data are approximately equally represented in training and testing sets. This is to make sure that the algorithm learns relationships $f(X)$ also under rare conditions X, for example meteorological extreme events.

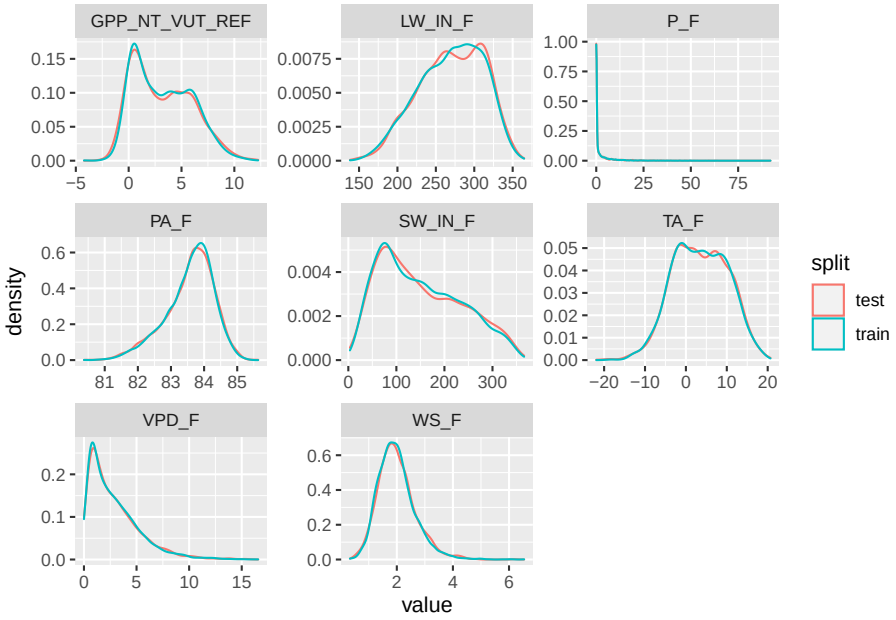
Several alternative functions for the data splitting step are available from different packages in R. We will use the the **rsample** package as it allows to additionally make sure that data from the full range of a given variable's values are well covered in both training and testing sets.

```
library(rsample)
set.seed(123)  # for reproducibility
split <- initial_split(ddf, prop = 0.7, strata = "VPD_F")
ddf_train <- training(split)
ddf_test <- testing(split)
```

Plot the distribution of values in the training and testing sets.

```
ddf_train %>%
  mutate(split = "train") %>%
  bind_rows(ddf_test %>%
    mutate(split = "test")) %>%
  pivot_longer(cols = 2:9, names_to = "variable", values_to = "value") %>%
  ggplot(aes(x = value, y = ..density.., color = split)) +
  geom_density() +
  facet_wrap(~variable, scales = "free")
```

```
## Warning: Removed 395 rows containing non-finite values (stat_density).
```


GPP_NT_VUT_REF
LW_IN_F
P_F
PA_F
SW_IN_F
TA_F
VPD_F
WS_F
density
value
split
test
train

Chapter 4

Pre-processing

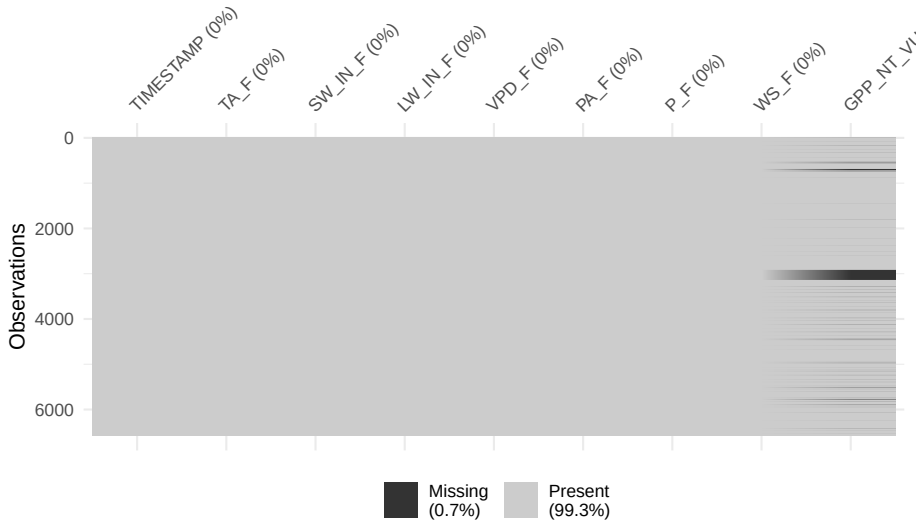
Skewed data, outliers, and values covering multiple orders of magnitude can create difficulties for certain ML algorithms, e.g., or K-nearest neighbours, or neural networks. Other algorithms, like tree-based methods (e.g., Random Forest), are more robust against such issues.

4.1 Dealing with missingness and bad data

Several ML algorithms require missing values to be removed. That is, if any of the cells in one row has a missing value, the entire cell gets removed. Data may be missing for several reasons. Some yield random patterns of missing data, others not. In the latter case, we can speak of *informative missingness* (Kuhn & Johnson, 2003) and its information can be used for predictions. For categorical data, we may replace such data with `"none"` (instead of `NA`), while randomly missing data may be dropped altogether. Some ML algorithms (mainly tree-based methods, e.g., random forest) can handle missing values. However, when comparing the performance of alternative ML algorithms, they should be tested with the same data and removing missing data should be done beforehand.

Visualising missing data is essential for making decisions about dropping rows with missing data versus removing predictors from the model (which would imply too much data removal). The cells with missing data in a data frame can be eaily visualised e.g. with `vis_miss()` from the *visdat* package.

```
library(visdat)
vis_miss(
  ddf,
  cluster = FALSE,
  warn_large_data = FALSE
  )
```



The question about what is "bad data" and whether or when it should be removed is often critical. Such decisions are important to keep track of and should be reported as transparently as possible in publications. In reality, where the data generation process may start in the field with actual human beings writing notes in a lab book, and where the human collecting the data is often not the same as the human writing the paper, it's often more difficult to keep track of such decisions. As a general principle, it is advisable to design data records such that decisions made during its process remain transparent throughout all stages of the workflow and that sufficient information be collected to enable later revisions of particularly critical decisions.

4.2 Standardization

Several algorithms explicitly require data to be standardized. That is, values of all predictors vary within a comparable range. The necessity of this step becomes obvious when considering neural networks, the activation functions of each node have to deal with standardized inputs. In other words, inputs have to vary over the same range, expecting a mean of zero and standard deviation of one.)

To get a quick overview of the distribution of all variables (columns) in our data frame, we can use the *skimr* package.

```
library(skimr)
skim(ddf)
```

We see for example, that typical values of `LW_IN_F` are by a factor 100 larger than values of `VPD_F`. The data must be standardized before using it for a neural network model. Standardization is done, for example, by dividing each variable, that is all values in one column, by the standard deviation of that variable, and then subtracting its mean. This way, the resulting standardized values are centered around 0, and scaled such that a value of 1 means that the data point is one standard deviation above the mean of the respective variable (column). When applied to all predictors individually, the absolute values of their variations can be directly compared and only then it can be meaningfully used for determining the distance.

Standardization can be done not only by centering and scaling (as described above), but also by *scaling to within range*, where values are scaled such that the minimum value within each variable (column) is 0 and the maximum is 1.

In order to avoid *data leakage*, centering and scaling has to be done separately for each split into training and validation data (more on that later). In other words, don't center and scale the entire data frame with the mean and standard deviation derived from the entire data frame, but instead center and scale with mean and standard deviation derived from the training portion of the data, and apply that also to the validation portion, when evaluating.

The *caret* package takes care of this. The R package **caret** provides a unified interface for using different ML algorithms implemented in separate packages. All you have to do is specify the required pre-processing steps as:

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
## The following objects are masked from 'package:yardstick':
##
##     precision, recall, sensitivity, specificity
```

```
pp <- preProcess(ddf_train, method = c("center", "scale"))
```

As seen above for the feature engineering example, this does not return a standardized version of the data frame `ddf`. Rather, it returns the information that allows us to apply the same standardization also to other data sets. In other words, we use the distribution of values in the data set to which we applied the function (here `ddf`) to determine the standardization (here: mean and standard deviation). As you will learn below, this is essential and critical to avoid *data leakage* (where information from the testing data set leaks into the training data set).

4.3 More pre-processing

Depending on the algorithm and the data, additional pre-processing steps may be required. You can find more information about this in the great and freely available online tutorial Hands-On Machine Learning in R.

One such additional pre-processing step is *imputation*, where missing values are imputed (gap-filled), for example by the mean of each variable respectively. Also imputation is prone to cause data leakage and must therefore be implemented as part of the resampling and training workflow. The **recipes** package offers a great way to deal with imputation (and also all other pre-processing steps). Here is a link to learn more about it.

Chapter 5

Model formulation

Remember that the aim of supervised ML is to find a model $\hat{Y} = f(X)$ so that $\hat{Y}$ agrees well with observations Y. We typically start with a research question where Y is given - naturally - by the problem we are addressing and we have a data set at hand where one or multiple predictors (or features) X are recorded along with Y. From our data, we have information about how GPP (ecosystem-level photosynthesis) depends on set of abiotic factors, mostly meteorological measurements. In R, it is common to use the *formula* notation to specify the target and predictor variables. You have probably encountered formulas before, e.g., for a linear regression using the `lm()` function. To specify a linear regression model for `GPP_NT_VUT_REF` with three predictors `SW_F_IN`, `VPD_F`, and `TA_F`, we write:

```
lm(GPP_NT_VUT_REF ~ SW_F_IN + VPD_F + TA_F, data = ddf)
```

Actually, the way we formulate a model is independent of the algorithm, or *engine* that takes care of fitting $f(X)$. As mentioned in Chapter 4 the R package **caret** provides a unified interface for using different ML algorithms implemented in separate packages. In other words, it acts as a *wrapper* for multiple different model fitting, or ML algorithms. This has the advantage that it unifies the interface (the way arguments are provided). caret also provides implementations for a set of commonly used tools for data processing, model training, and evaluation. We'll use caret for model training with the function `train()` (more on model training in Chapter 6). Note however, that using a specific algorithm, which is implemented in a specific package outside caret, also requires that the respective package be installed and loaded. Using caret for specifying the same linear regression model as above, using the base-R `lm()` function, can be done with caret in a generalized form as:

```
library(caret)
train(
  form = GPP_NT_VUT_REF ~ SW_F_IN + VPD_F + TA_F,
  data = ddf,
  method = "lm"
)
```

Of course, this is an overkill compared to just writing `lm(...)`. But the advantage of the unified interface is that we can simply replace the `method` argument to use a different ML algorithm. For example, to use a shallow neural network model using the **nnet** package, we can write:

```
train(
  form = GPP_NT_VUT_REF ~ SW_F_IN + VPD_F + TA_F,
  data = ddf,
  method = "nnet"
)
```

Chapter 6

Model training

Model training in supervised ML is guided by the match (or mismatch) between the predicted and observed target variable(s), that is, between $\hat{Y}$ and Y. The *loss* function quantifies this mismatch ($L(\hat{Y}, Y)$), and the algorithm takes care of progressively reducing the loss during model training. Let's say the ML model contains two parameters and predictions can be considered a function of the two ($\hat{Y}(w_1, w_2)$). Y is actually constant. Thus, the loss function is effectively a function $L(w_1, w_2)$. Therefore, we can consider the model training as a search of the parameter space of the machine learning model (w_1, w_2) to find the minimum of the loss. Common loss functions are the root mean square error (RMSE), or the mean square error, or the mean absolute error.

Loss minimization is a general feature of ML model training. Practically all ML algorithms have some "knobs" to turn in order to achieve efficient model training and predictive performance. What these knobs are, depends on the ML algorithm.

In Video 6B you learned how the loss minimization, at least for some ML methods (e.g., artificial neural networks), is guided by *gradient descent*. It offers a principle for determining in what direction to jump and search the parameter space. *Gradient descent* changes parameters relative to a reference such that for a given "distance" of the "jump", the loss is reduced by as much as possible. In other words, it descends along the steepest gradient of the loss hyperplane (the yellow-red plane in the figure above). You can imagine this as the trajectory of a ball rolling into the loss "depression".

Model training is implemented in R for different algorithms in different packages. Some algorithms are even implemented by multiple packages (e.g., `nnet` and `neuralnet` for artificial neural networks). As described in Chapter **??**, the **caret** package provides "wrappers" that handle a large selection of different ML model implementations in different packages with a unified interface (see here for an overview of available models). The **caret** function `train()` is the

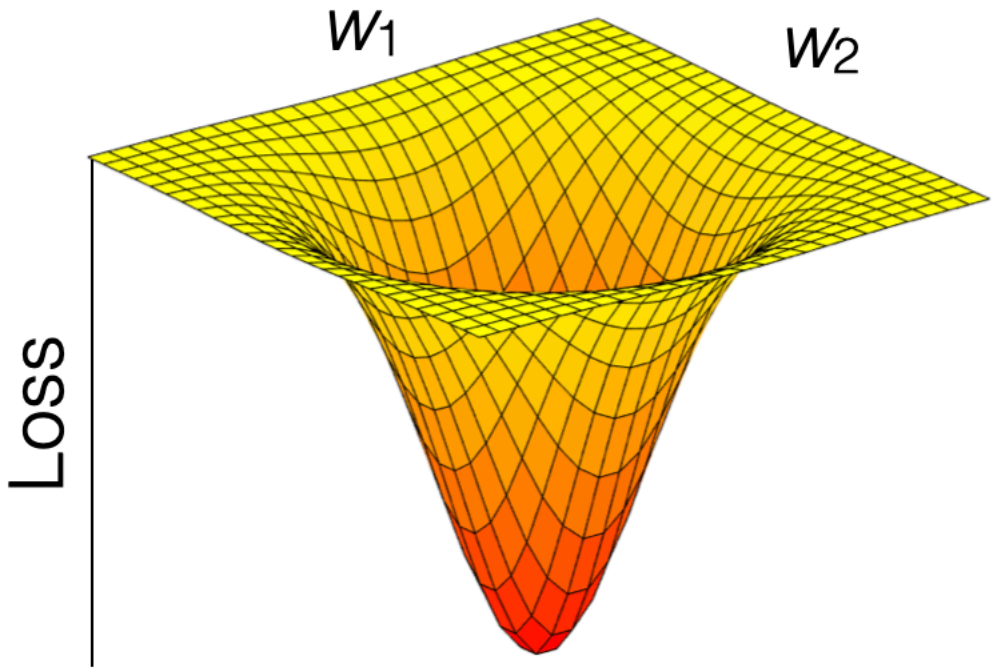


Figure 6.1: Visualization of a loss function as a plane spanned by the two parameters w_1 and w_2.

centre piece. Its argument `metric` specifies the loss function and defaults to RMSE for regression models and accuracy for classification (see sub-section on metrics below). A complete implementation of model training with caret is demonstrated further below.

6.1 Hyperparameter tuning

As for practically all ML algorithms, there are free parameters that determine their characteristics and how the learning operates - *hyperparameters*. Turning back to the analogue of the ball and the loss depression, imagine the effect of how fast the ball mentioned before is rolling. If it rolls too fast (that is if the "jump" of the parameter search is too large), it descends into the depression fast but may shoot beyond the minimum and it has to do a "180-degrees turn" and continue the search. That's not very efficient. On the other extreme, a very slowly rolling ball (the parameter space is searched with small jumps) will take much longer to arrive at the bottom of the depression. Hence, there is an optimum in between. This "size of the jump" is called the *learning rate*, and it usually has to be tuned for optimal performance of model training of artificial neural networks. Other ML algorithms have other types of hyperparameters. While some hyperparameters determine the performance of the model training, other hyperparameters are decisive for the model's predictive skills. Hyperparameters are not to be confused with the model coefficients. For example, in linear regression, the number of predictors can be considered a hyperparameter, while the values of β are the coefficients. In general, hyperparameters determine the *structure* of the algorithm.

Let's turn to the K-nearest neighbour (KNN) algorithm as an example. In KNN, the hyperparameter is k. That is, the number of neighbours to consider taking their mean. With KNN, there is always an optimum k. Obviously, if $k = n$, we consider all observations as neighbours and each prediction is simply the mean of all observed target values Y, irrespective of the predictor values. This cannot be optimal and such a model will likely underfit. On the other extreme, with $k = 1$, the model will be strongly affected by the noise in the single nearest neighbour and its generalisability will suffer. This should be reflected in a poor performance on the validation data. Indeed, it is, as the Figure 6.2 illustrates.

You will encounter different hyperparameters for neural networks in later chapters.

In **caret**, hyperparameter tuning is implemented as part of the `train()` function. Values of hyperparameters to consider are to be specified by the argument `tuneGrid`, which takes a data frame with column(s) named according to the name(s) of the hyperparameter(s) and rows for each combination of hyperparameters to consider. More explicit examples follow below.

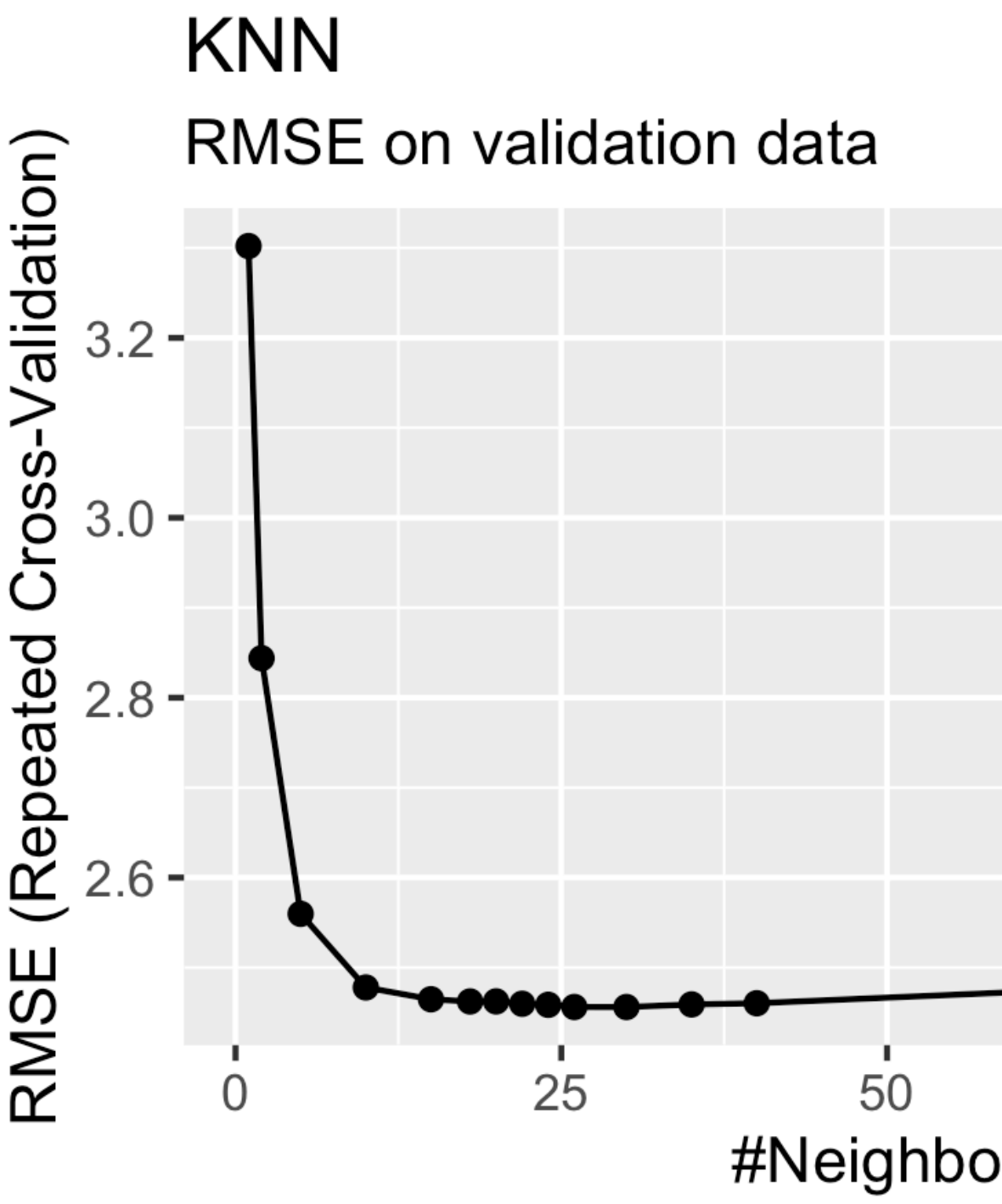


Figure 6.2: Improvement in RMSE on validation data with increasing number of neighbours.

6.2 Resampling

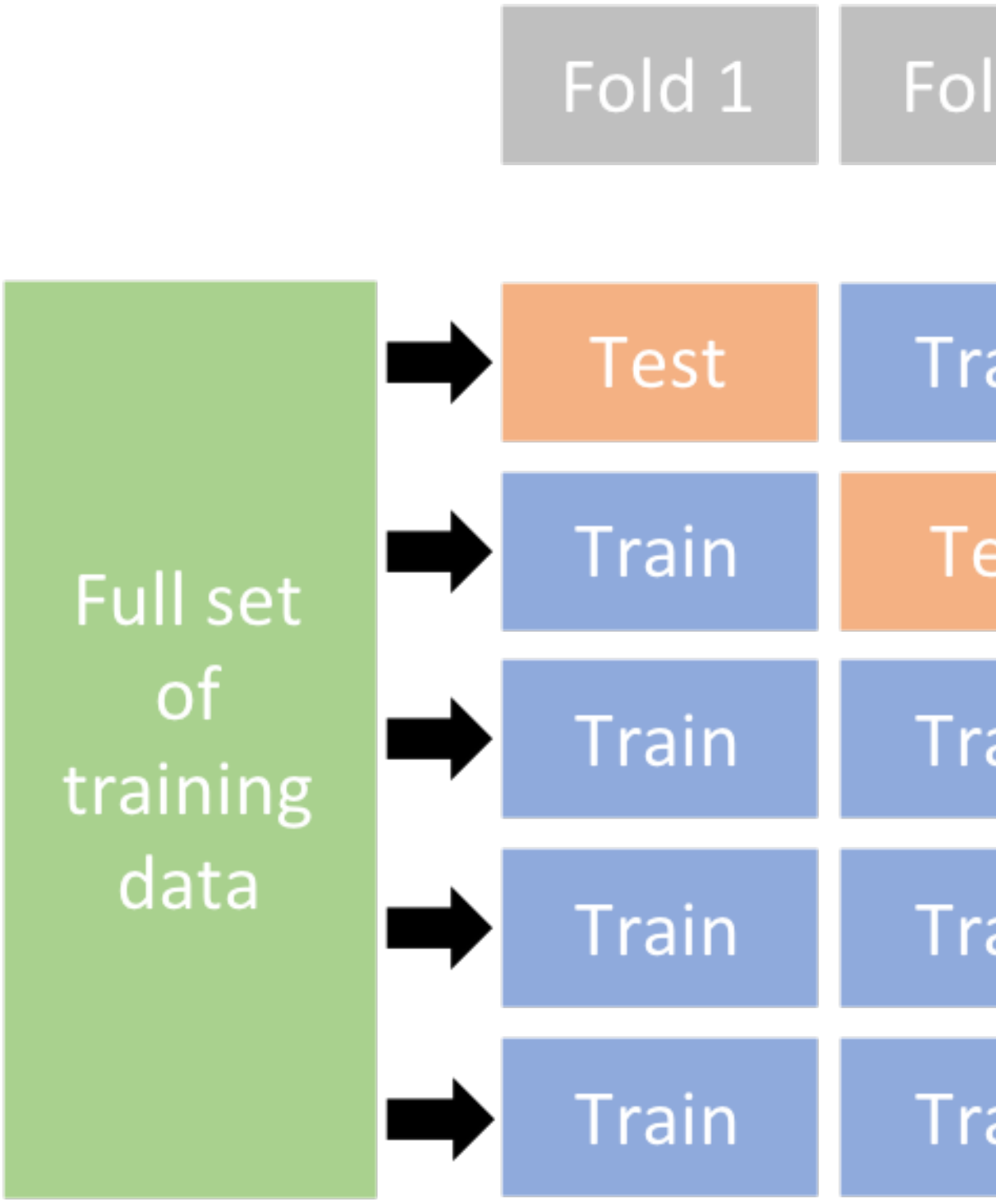
At the beginning of this tutorial, we demonstrated a case of overfitting. In the example with KNN above, the increasing error on the validation data (measured by RMSE) is an indication of poor *generalisability*. The goal of model training is to achieve the best possible generalisation performance. That is, the lowest validation error measured by applying the trained model on the testing data set from the initial split. Note the distinction: validation data is what is used during model training, testing data is held out at the initial split and not used during model training.

But how can the generalisability be assessed when the testing set is held out completely during the training step? To measure the model's generalisability and "direct" the ML algorithm to minimize the validation error during the training step, we can further split the training set into one or more training and validation sub-set. This is called *resampling*. The model performance determined on this resampled validation set is then a good estimate of the generalisation error we get when evaluated against the testing data that was set aside from the initial split.

The challenge to be met here is that we further reduce the number of data points in the resampled training and validation sets which may lead to evaluation statistics not being sufficiently robust and bears the potential that we're training to some peculiarities in the (relatively small) validation set. In order to control for this, common practice is to do multiple resamples (multiple *folds* of training-validation splits) which is the *repeat* part of repeated CV.

By repeating cross-validation we are picking different folds for training and validating. For each repetition, we can determine the validation error, i.e., how well the trained model performed on the validation fold. Taking the mean performance across all repetitions gives us the average performance we would expect for the model to have on unseen data. This provides a general assessment of how good the model may perform. To actually check for its performance, we can use the training data set that we excluded from the beginning and was not used in the CV process.

Taking the mean validation error across all repetitions, we can This is called *k-fold cross validation*.


Fold 1
Fol
Full set of training data
Test
Tra
Train
Te
Train
Tra
Train
Tra
Train
Tra

There is no formal rule about the number of folds and hence the number of data points in each training and testing fold. *Leave-one-out* cross validation is an extreme variant of k-fold cross validation, where k equals the number of data points in the full set of training data.

To do a hyperparameter tuning and k-fold cross validation during model training in R, we don't have to implement the loops ourselves. The resampling procedure can be specified in the **caret** function `train()` with the argument `trControl`. The object that this argument takes is the output of a function call to `trainControl()`. This can be implemented in two steps. For example, to do a 10-fold cross-validation, repeated five times, we can write:

```
my_cv <- trainControl(
  method = "repeatedcv",
  number = 10,
  repeats = 5
)

train(..., trControl = my_cv)
```

Chapter 7

Exercises

Now that you are familiar with the basic steps for supervised machine learning, you can get your hands on the data yourself and implement code for addressing the modelling task outlined in Chapter 2.

7.1 Reading and cleaning

Read the CSV file `"./data/FLX_CH-Dav_FLUXNET2015_FULLSET_DD_1997-2014_1-3.csv"`, select all variables with name ending with `"_F"`, the variables `"TIMESTAMP"`, `"GPP_NT_VUT_REF"`, and `"NEE_VUT_REF_QC"`, and drop all variables that contain `"JSB"` in their name. Then convert the variable `"TIMESTAMP"` to a date-time object with the function `ymd()` from the *lubridate* package, and interpret all values `-9999` as missing values. Then, set all values of `"GPP_NT_VUT_REF"` to missing if the corresponding quality control variable indicates that less than 90% are measured data points. Finally, drop the variable `"NEE_VUT_REF_QC"` - we won't use it anymore.

```
library(tidyverse)

ddf <- read_csv("./data/FLX_CH-Dav_FLUXNET2015_FULLSET_DD_1997-2014_1-3.csv") %>%

  ## select only the variables we are interested in
  select(starts_with("TIMESTAMP"),
         ends_with("_F"),   # all meteorological variables
         GPP_NT_VUT_REF,
         NEE_VUT_REF_QC,
         -contains("JSB")) %>%

  ## convert to a nice date object
```

```
  mutate(TIMESTAMP = lubridate::ymd(TIMESTAMP)) %>%

  ## set all -9999 to NA
  na_if(-9999) %>%

  ## drop QC variables (no longer needed), except NEE_VUT_REF_QC
  select(-ends_with("_QC"), NEE_VUT_REF_QC) %>%

  mutate(GPP_NT_VUT_REF = ifelse(NEE_VUT_REF_QC < 0.9, NA, GPP_NT_VUT_REF)) %>%
  select(-NEE_VUT_REF_QC)
```

```
## Rows: 6574 Columns: 334

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (334): TIMESTAMP, TA_F_MDS, TA_F_MDS_QC, TA_F_MDS_NIGHT, TA_F_MDS_NIGHT_...

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

7.2 Data splitting

Split the data a training and testing set, that contain 70% and 30% of the total available data, respectively.

```
library(rsample)
set.seed(1982)  # for reproducibility
split <- initial_split(ddf, prop = 0.7)
ddf_train <- training(split)
ddf_test <- testing(split)
```

7.3 Linear model

7.3.1 Training

Fit a linear regression model using the base-R function `lm()` and the training set. The target variable is `"GPP_NT_VUT_REF"`, and predictor variables are all available meterological variables in the dataset. Answer the following questions:

- What is the R^2 of predicted vs. observed `"GPP_NT_VUT_REF"`?

- Is the linear regression slope significantly different from zero for all predictors?
- Is a linear regression model with "poor" predictors removed better supported by the data than a model including all predictors?

```
## fit linear regression model
linmod_baser <- lm(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP)
)

## show summary information of the model
summary(linmod_baser)
```

```
##
## Call:
## lm(formula = GPP_NT_VUT_REF ~ ., data = ddf_train %>% drop_na() %>%
##     select(-TIMESTAMP))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1941 -1.0393 -0.1299  0.8859  7.7205
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.9570704  3.4840107  -0.275 0.783558
## TA_F         0.1633211  0.0108181  15.097  < 2e-16 ***
## SW_IN_F      0.0137226  0.0003914  35.064  < 2e-16 ***
## LW_IN_F      0.0207311  0.0012040  17.219  < 2e-16 ***
## VPD_F       -0.1420886  0.0195509  -7.268 4.35e-13 ***
## PA_F        -0.0402727  0.0404675  -0.995 0.319704
## P_F         -0.0214314  0.0046177  -4.641 3.57e-06 ***
## WS_F        -0.1418356  0.0398616  -3.558 0.000378 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4135 degrees of freedom
## Multiple R-squared:  0.6904, Adjusted R-squared:  0.6899
## F-statistic:  1317 on 7 and 4135 DF,  p-value: < 2.2e-16
```

```
## Models can be compared, for example by the Bayesian Information Criterion (BIC).
## It penalizes complex models and is optimal (lowest) where the trade-off between
## explanatory power on the training set vs. number of predictors is best. The BIC
```

```
## tends to be more conservative than the AIC.
BIC(linmod_baser)
```

```
## [1] 15450.89
```

```
## Fit an lm model on the same data, but with PA_F removed.
linmod_baser_nopaf <- lm(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP, -PA_F)
)

## The R-squared is slightly lower here (0.6903) than in the model with all predictors
summary(linmod_baser_nopaf)
```

```
##
## Call:
## lm(formula = GPP_NT_VUT_REF ~ ., data = ddf_train %>% drop_na() %>%
##     select(-TIMESTAMP, -PA_F))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1849 -1.0390 -0.1306  0.8867  7.7594
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.4102425  0.3135013 -14.068  < 2e-16 ***
## TA_F         0.1595814  0.0101445  15.731  < 2e-16 ***
## SW_IN_F      0.0136923  0.0003902  35.093  < 2e-16 ***
## LW_IN_F      0.0210424  0.0011626  18.099  < 2e-16 ***
## VPD_F       -0.1400240  0.0194405  -7.203 6.98e-13 ***
## P_F         -0.0212661  0.0046147  -4.608 4.18e-06 ***
## WS_F        -0.1343263  0.0391409  -3.432 0.000605 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4136 degrees of freedom
## Multiple R-squared:  0.6903, Adjusted R-squared:  0.6899
## F-statistic:  1537 on 6 and 4136 DF,  p-value: < 2.2e-16
```

```
## ... but the BIC is clearly lower, indicating that the model with PA_F removed is be
BIC(linmod_baser_nopaf)
```

```
## [1] 15443.55
```

Use caret and the function `train()` for fitting the same linear regression model (with all predictors) on the same data. Does it yield identical results as using `lm()` directly? You will have to set the argument `trControl` accordingly to avoid resampling, and instead fit the model on the all data in `ddf_train`. You can use `summary()` also on the object returned by the function `train()`.

```
library(caret)

linmod_caret <- train(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP),
  method = "lm",
  trControl = trainControl(method = "none")
)

summary(linmod_caret)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1941 -1.0393 -0.1299  0.8859  7.7205
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.9570704  3.4840107  -0.275 0.783558
## TA_F         0.1633211  0.0108181  15.097  < 2e-16 ***
## SW_IN_F      0.0137226  0.0003914  35.064  < 2e-16 ***
## LW_IN_F      0.0207311  0.0012040  17.219  < 2e-16 ***
## VPD_F       -0.1420886  0.0195509  -7.268 4.35e-13 ***
## PA_F        -0.0402727  0.0404675  -0.995 0.319704
## P_F         -0.0214314  0.0046177  -4.641 3.57e-06 ***
## WS_F        -0.1418356  0.0398616  -3.558 0.000378 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4135 degrees of freedom
## Multiple R-squared:  0.6904, Adjusted R-squared:  0.6899
## F-statistic:  1317 on 7 and 4135 DF,  p-value: < 2.2e-16
```

7.3.2 Prediction

With the model containing all predictors and fitted on `ddf_train`, make predictions using first `ddf_train` and then `ddf_test`. Compute the R^2 and the root-mean-square error, and visualise modelled vs. observed values to evaluate both predictions.

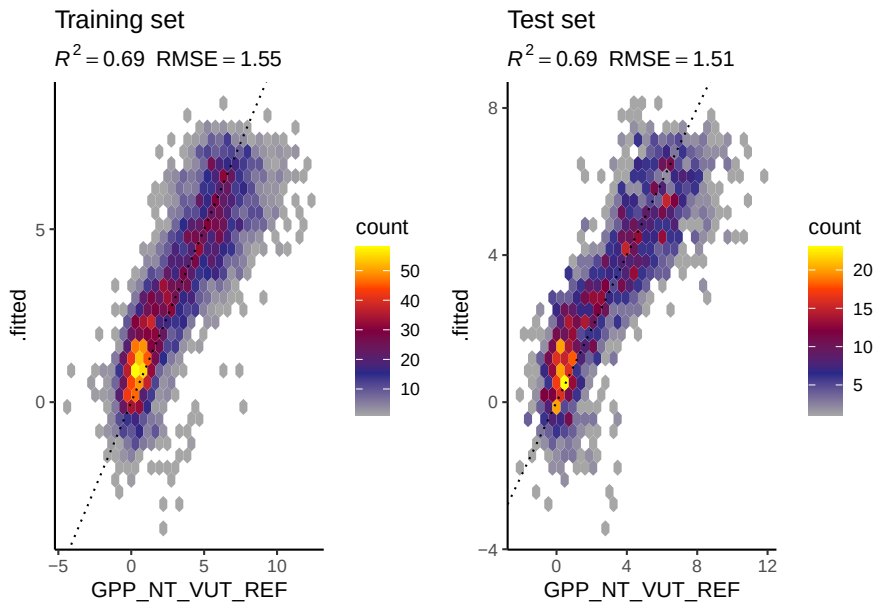
Do you expect the linear regression model trained on `ddf_train` to predict substantially better on `ddf_train` than on `ddf_test`? Why (not)?

Hints:

- To calculate predictions, use the generic function `predict()` with the argument `newdata = ....`
- The R^2 can be extracted from the model object as `summary(model_object)$r.squared`, or is (as the RMSE) given in the metrics data frame returned by `metrics()` from the *yardstick* library.
- For visualisation the model performance, consider a scatterplot, or (better) a plot that reveals the density of overlapping points. (We're plotting information from over 4000 data points here!)

```
library(patchwork)
library(yardstick)

## made into a function to reuse code below
eval_model <- function(mod, df_train, df_test){

  ## add predictions to the data frames
  df_train <- df_train %>%
    drop_na() %>%
    mutate(.fitted = predict(mod, newdata = .))

  df_test <- df_test %>%
    drop_na() %>%
    mutate(.fitted = predict(mod, newdata = .))

  ## get metrics tables
  metrics_train <- df_train %>%
    yardstick::metrics(GPP_NT_VUT_REF, .fitted)

  metrics_test <- df_test %>%
    yardstick::metrics(GPP_NT_VUT_REF, .fitted)

  ## extract values from metrics tables
  rmse_train <- metrics_train %>%
    filter(.metric == "rmse") %>%
```

```
    pull(.estimate)
  rsq_train <- metrics_train %>%
    filter(.metric == "rsq") %>%
    pull(.estimate)

  rmse_test <- metrics_test %>%
    filter(.metric == "rmse") %>%
    pull(.estimate)
  rsq_test <- metrics_test %>%
    filter(.metric == "rsq") %>%
    pull(.estimate)

  ## visualise with a hexagon binning and a customised color scale,
  ## adding information of metrics as sub-titles
  gg1 <- df_train %>%
    ggplot(aes(GPP_NT_VUT_REF, .fitted)) +
    geom_hex() +
    scale_fill_gradientn(
      colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
    geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
    labs(subtitle = bquote( italic(R)^2 == .(format(rsq_train, digits = 2)) ~~
                            RMSE == .(format(rmse_train, digits = 3))),
         title = "Training set") +
    theme_classic()

  gg2 <- df_test %>%
    ggplot(aes(GPP_NT_VUT_REF, .fitted)) +
    geom_hex() +
    scale_fill_gradientn(
      colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
    geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
    labs(subtitle = bquote( italic(R)^2 == .(format(rsq_test, digits = 2)) ~~
                            RMSE == .(format(rmse_test, digits = 3))),
         title = "Test set") +
    theme_classic()

  return(gg1 + gg2)
}

eval_model(mod = linmod_baser, df_train = ddf_train, df_test = ddf_test)
```

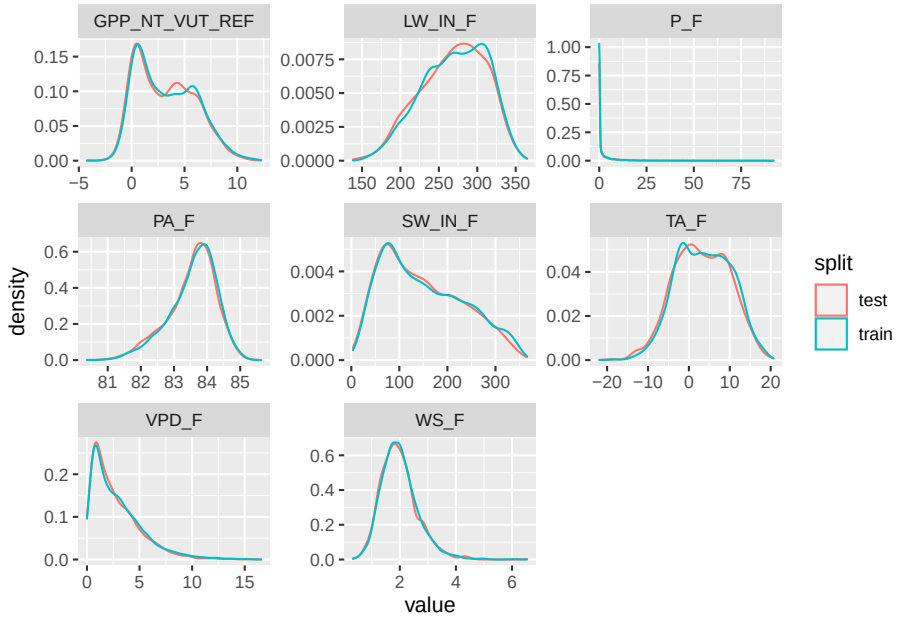


7.4 KNN

7.4.1 Check data

```
ddf_train %>%
  mutate(split = "train") %>%
  bind_rows(ddf_test %>%
    mutate(split = "test")) %>%
  pivot_longer(cols = 2:9, names_to = "variable", values_to = "value") %>%
  ggplot(aes(x = value, y = ..density.., color = split)) +
  geom_density() +
  facet_wrap(~variable, scales = "free")
```

```
## Warning: Removed 681 rows containing non-finite values (stat_density).
```



The variable `PA_F` looks weird and was not significant in the linear model. Therefore, we won't use it for the models below.

7.4.2 Training

Fit two KNN models on `ddf_train` (excluding `"PA_F"`), one with $k = 2$ and one with $k = 30$, both without resampling. Use the RMSE as the loss function. Center and scale data as part of the pre-processing and model formulation specification using the function `recipe()`.

```
library(recipes)
```

```
##
## Attaching package: 'recipes'
```

```
## The following object is masked from 'package:stringr':
##
##     fixed
```

```
## The following object is masked from 'package:stats':
##
##     step
```

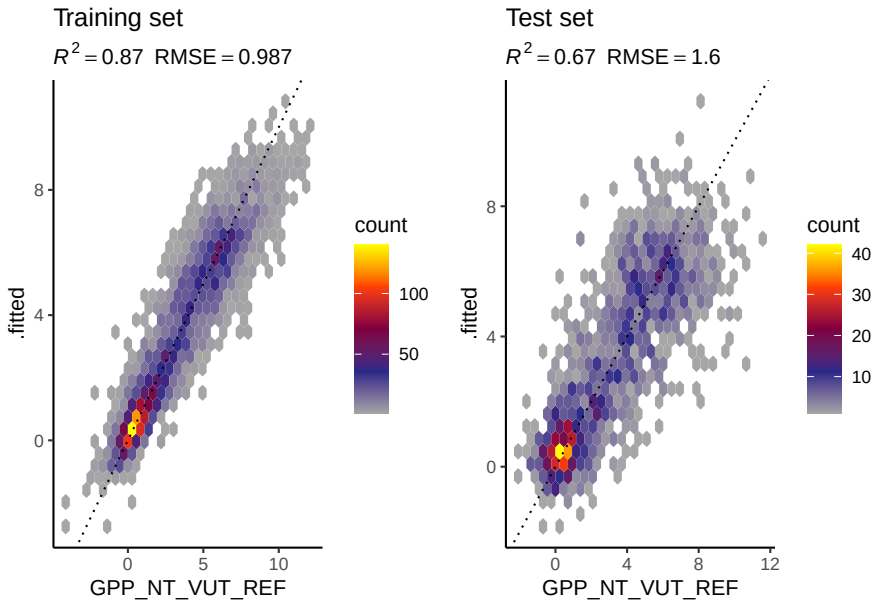
```
## model formulation and preprocessing specification
myrecipe <- recipe(
  GPP_NT_VUT_REF ~ TA_F + SW_IN_F + LW_IN_F + VPD_F + P_F + WS_F,
  data = ddf_train %>% drop_na()) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes())

## fit model with k=2
mod_knn_k2 <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "knn",
  trControl = trainControl("none"),
  tuneGrid = data.frame(k = c(2)),
  metric = "RMSE"
  )

## fit model with k=30
mod_knn_k30 <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "knn",
  trControl = trainControl("none"),
  tuneGrid = data.frame(k = c(30)),
  metric = "RMSE"
  )
```

7.4.3 Prediction

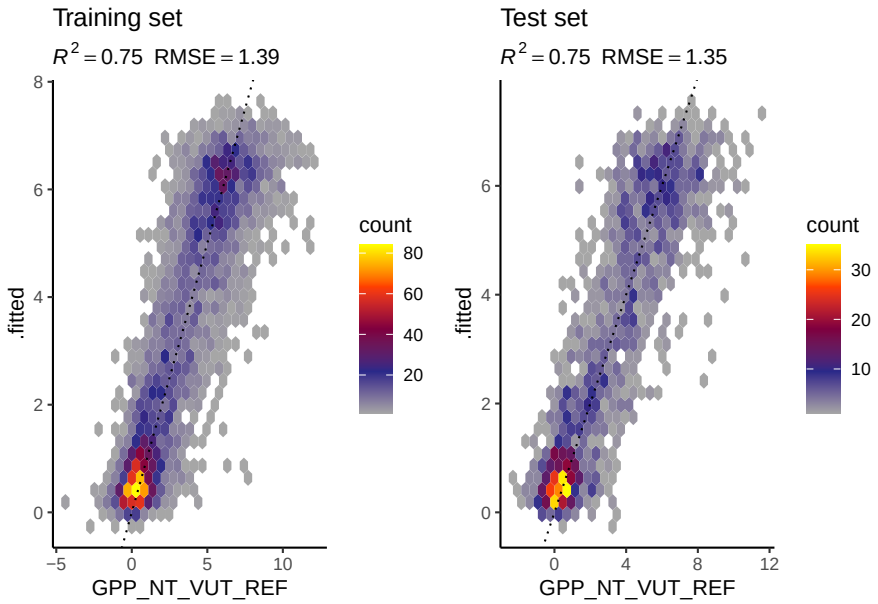
With the two models fitted above, predict `"GPP_NT_VUT_REF"` for both and training and the testing sets, and evaluate them as above (metrics and visualisation).

Which model do you expect to perform better on the training set and which to perform better on the testing set? Why? Do you find evidence for overfitting in any of the models?

```
## with k = 2
eval_model(
  mod_knn_k2,
  df_train = ddf_train,
  df_test = ddf_test
  )
```




```
## with k = 30
eval_model(
  mod_knn_k30,
  df_train = ddf_train,
  df_test = ddf_test
  )
```



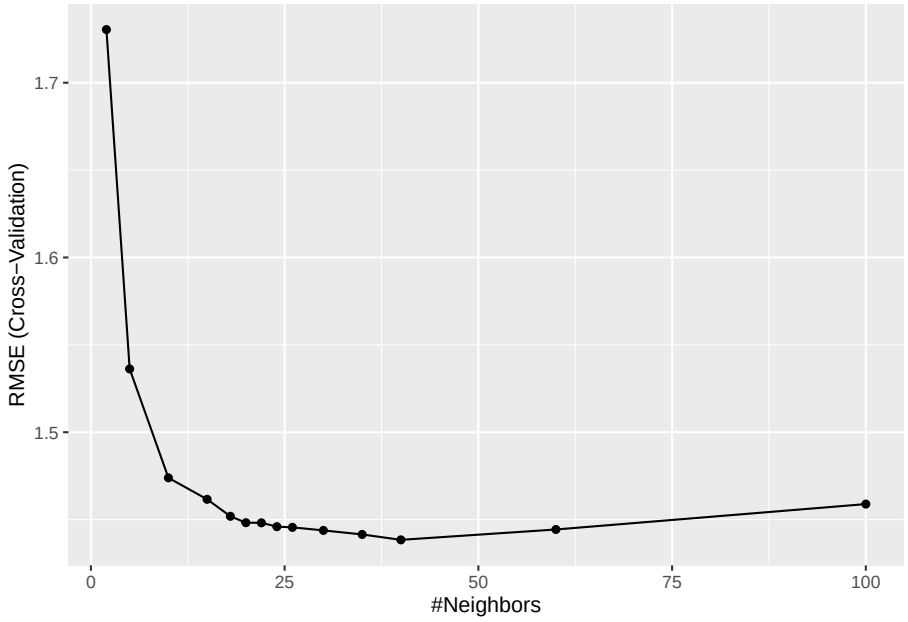
7.4.4 Sample hyperparameters

Train a KNN model with hyperparameter (k) tuned, and with five-fold cross validation, using the training set. As the loss function, use RMSE. Sample the following values for k: 2, 5, 10, 15, 18, 20, 22, 24, 26, 30, 35, 40, 60, 100. Visualise the RMSE as a function of k.

Hint:

- The visualisation of cross-validation results can be visualised with the `plot(model_object)` of `ggplot(model_object)`.

```
mod_knn <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  tuneGrid = data.frame(k = c(2, 5, 10, 15, 18, 20, 22, 24, 26, 30, 35, 40, 60, 100)),
  metric = "RMSE"
  )
ggplot(mod_knn)
```



7.5 Random forest

7.5.1 Training

Fit a random forest model with `ddf_train` and all predictors excluding `"PA_F"` and five-fold cross validation. Use RMSE as the loss function.

Hints:

- Use the package *ranger* which implements the random forest algorithm.
- See here for information about hyperparameters available for tuning with caret.
- Set the argument `savePredictions = "final"` of function `trainControl()`.

```
library(ranger)

## no pre-processing necessary
myrecipe <- recipe(
  GPP_NT_VUT_REF ~ TA_F + SW_IN_F + LW_IN_F + VPD_F + P_F + WS_F,
  data = ddf_train %>%
    drop_na())

rf <- train(
```

```
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = floor(6 / 3),
                          .min.node.size = 5,
                          .splitrule = "variance"),
  metric = "RMSE",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 2000,          # high number ok since no hperparam tuning
  seed = 1982                # for reproducibility
)
```

```
## Loading required namespace: e1071
```

```
##
## Attaching package: 'e1071'
```

```
## The following object is masked from 'package:rsample':
##
##     permutations
```

```
rf
```

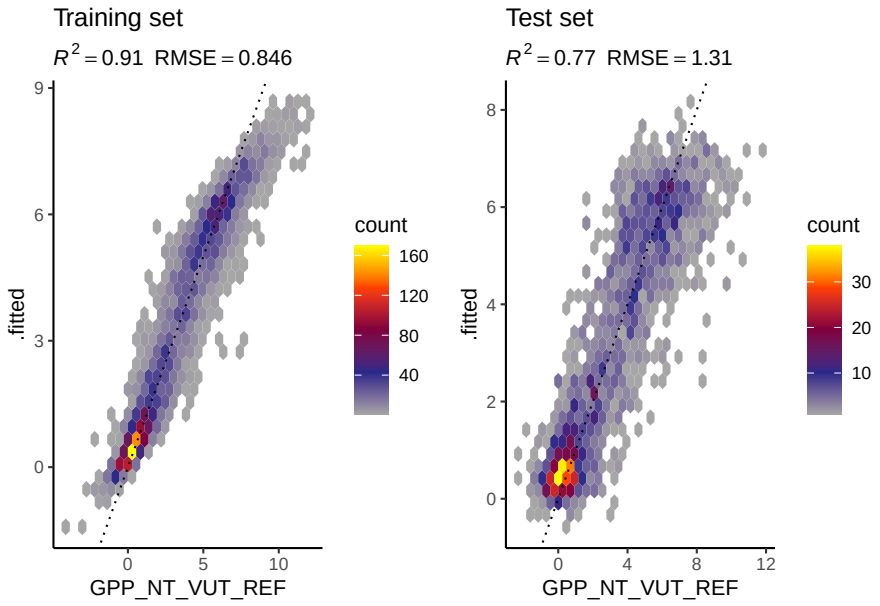
```
## Random Forest
##
## 4143 samples
##    8 predictor
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 3313, 3315, 3315, 3314, 3315
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   1.401107  0.746519  1.056488
##
## Tuning parameter 'mtry' was held constant at a value of 2
## Tuning
##  parameter 'splitrule' was held constant at a value of variance
##
## Tuning parameter 'min.node.size' was held constant at a value of 5
```

7.5.2 Prediction

Evaluate the trained model on the training and on the test set, giving metrics and a visualisation as above.

How are differences in performance to be interpreted? Compare the performances of linear regression, KNN, and random forest, considering the evaluation on the test set.

```
eval_model(rf, df_train = ddf_train, df_test = ddf_test)
```

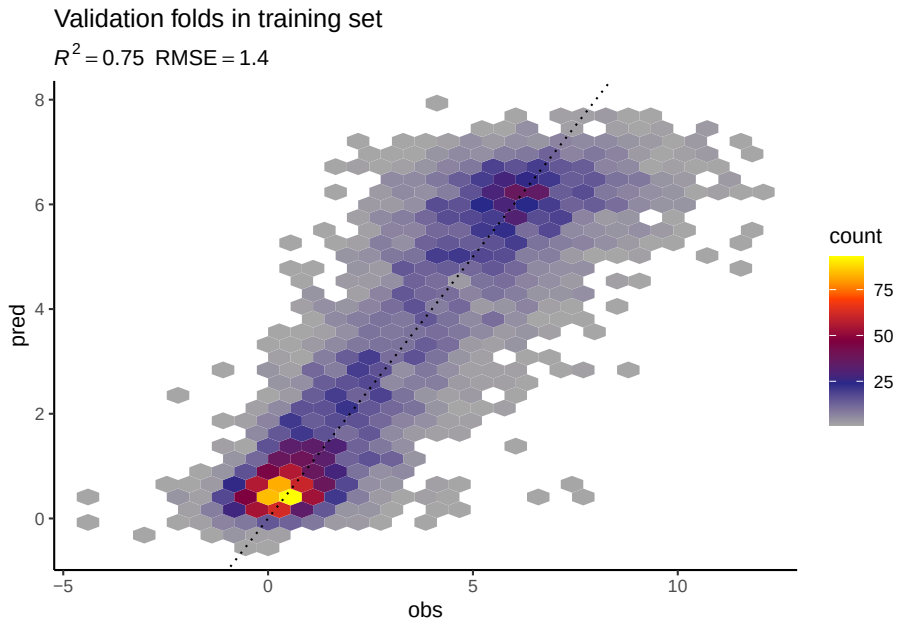


Show the model performance (metrics and visualisation) on the validation sets all cross validation folds combined.

Do you expect it to be more similar to the model performance on the training set or the testing set in the evaluation above? Why?

```
metrics_train <- rf$pred %>%
  yardstick::metrics(obs, pred)

rmse_train <- metrics_train %>%
  filter(.metric == "rmse") %>%
  pull(.estimate)
rsq_train <- metrics_train %>%
  filter(.metric == "rsq") %>%
  pull(.estimate)
```

```
rf$pred %>%
  ggplot(aes(obs, pred)) +
  geom_hex() +
  scale_fill_gradientn(
    colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
  geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
  labs(subtitle = bquote( italic(R)^2 == .(format(rsq_train, digits = 2)) ~~
                          RMSE == .(format(rmse_train, digits = 3))),
       title = "Validation folds in training set") +
  theme_classic()
```



Chapter 8

Solutions

Now that you are familiar with the basic steps for supervised machine learning, you can get your hands on the data yourself and implement code for addressing the modelling task outlined in Chapter 2.

8.1 Reading and cleaning

Read the CSV file `"./data/FLX_CH-Dav_FLUXNET2015_FULLSET_DD_1997-2014_1-3.csv"`, select all variables with name ending with `"_F"`, the variables `"TIMESTAMP"`, `"GPP_NT_VUT_REF"`, and `"NEE_VUT_REF_QC"`, and drop all variables that contain `"JSB"` in their name. Then convert the variable `"TIMESTAMP"` to a date-time object with the function `ymd()` from the *lubridate* package, and interpret all values `-9999` as missing values. Then, set all values of `"GPP_NT_VUT_REF"` to missing if the corresponding quality control variable indicates that less than 90% are measured data points. Finally, drop the variable `"NEE_VUT_REF_QC"` - we won't use it anymore.

```
library(tidyverse)

ddf <- read_csv("./data/FLX_CH-Dav_FLUXNET2015_FULLSET_DD_1997-2014_1-3.csv") %>%

  ## select only the variables we are interested in
  select(starts_with("TIMESTAMP"),
         ends_with("_F"),   # all meteorological variables
         GPP_NT_VUT_REF,
         NEE_VUT_REF_QC,
         -contains("JSB")) %>%

  ## convert to a nice date object
```

```
  mutate(TIMESTAMP = lubridate::ymd(TIMESTAMP)) %>%

  ## set all -9999 to NA
  na_if(-9999) %>%

  ## drop QC variables (no longer needed), except NEE_VUT_REF_QC
  select(-ends_with("_QC"), NEE_VUT_REF_QC) %>%

  mutate(GPP_NT_VUT_REF = ifelse(NEE_VUT_REF_QC < 0.9, NA, GPP_NT_VUT_REF)) %>%
  select(-NEE_VUT_REF_QC)
```

```
## Rows: 6574 Columns: 334

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (334): TIMESTAMP, TA_F_MDS, TA_F_MDS_QC, TA_F_MDS_NIGHT, TA_F_MDS_NIGHT_...

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

8.2 Data splitting

Split the data a training and testing set, that contain 70% and 30% of the total available data, respectively.

```
library(rsample)
set.seed(1982)  # for reproducibility
split <- initial_split(ddf, prop = 0.7)
ddf_train <- training(split)
ddf_test <- testing(split)
```

8.3 Linear model

8.3.1 Training

Fit a linear regression model using the base-R function `lm()` and the training set. The target variable is `"GPP_NT_VUT_REF"`, and predictor variables are all available meterological variables in the dataset. Answer the following questions:

- What is the R^2 of predicted vs. observed `"GPP_NT_VUT_REF"`?

- Is the linear regression slope significantly different from zero for all predictors?
- Is a linear regression model with "poor" predictors removed better supported by the data than a model including all predictors?

```
## fit linear regression model
linmod_baser <- lm(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP)
)

## show summary information of the model
summary(linmod_baser)
```

```
##
## Call:
## lm(formula = GPP_NT_VUT_REF ~ ., data = ddf_train %>% drop_na() %>%
##     select(-TIMESTAMP))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1941 -1.0393 -0.1299  0.8859  7.7205
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.9570704  3.4840107  -0.275 0.783558
## TA_F         0.1633211  0.0108181  15.097  < 2e-16 ***
## SW_IN_F      0.0137226  0.0003914  35.064  < 2e-16 ***
## LW_IN_F      0.0207311  0.0012040  17.219  < 2e-16 ***
## VPD_F       -0.1420886  0.0195509  -7.268 4.35e-13 ***
## PA_F        -0.0402727  0.0404675  -0.995 0.319704
## P_F         -0.0214314  0.0046177  -4.641 3.57e-06 ***
## WS_F        -0.1418356  0.0398616  -3.558 0.000378 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4135 degrees of freedom
## Multiple R-squared:  0.6904, Adjusted R-squared:  0.6899
## F-statistic:  1317 on 7 and 4135 DF,  p-value: < 2.2e-16
```

```
## Models can be compared, for example by the Bayesian Information Criterion (BIC).
## It penalizes complex models and is optimal (lowest) where the trade-off between
## explanatory power on the training set vs. number of predictors is best. The BIC
```

```
## tends to be more conservative than the AIC.
BIC(linmod_baser)
```

```
## [1] 15450.89
```

```
## Fit an lm model on the same data, but with PA_F removed.
linmod_baser_nopaf <- lm(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP, -PA_F)
)

## The R-squared is slightly lower here (0.6903) than in the model with all predictors
summary(linmod_baser_nopaf)
```

```
##
## Call:
## lm(formula = GPP_NT_VUT_REF ~ ., data = ddf_train %>% drop_na() %>%
##     select(-TIMESTAMP, -PA_F))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1849 -1.0390 -0.1306  0.8867  7.7594
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.4102425  0.3135013 -14.068  < 2e-16 ***
## TA_F         0.1595814  0.0101445  15.731  < 2e-16 ***
## SW_IN_F      0.0136923  0.0003902  35.093  < 2e-16 ***
## LW_IN_F      0.0210424  0.0011626  18.099  < 2e-16 ***
## VPD_F       -0.1400240  0.0194405  -7.203 6.98e-13 ***
## P_F         -0.0212661  0.0046147  -4.608 4.18e-06 ***
## WS_F        -0.1343263  0.0391409  -3.432 0.000605 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4136 degrees of freedom
## Multiple R-squared:  0.6903, Adjusted R-squared:  0.6899
## F-statistic:  1537 on 6 and 4136 DF,  p-value: < 2.2e-16
```

```
## ... but the BIC is clearly lower, indicating that the model with PA_F removed is be
BIC(linmod_baser_nopaf)
```

```
## [1] 15443.55
```

Use caret and the function `train()` for fitting the same linear regression model (with all predictors) on the same data. Does it yield identical results as using `lm()` directly? You will have to set the argument `trControl` accordingly to avoid resampling, and instead fit the model on the all data in `ddf_train`. You can use `summary()` also on the object returned by the function `train()`.

```
library(caret)

linmod_caret <- train(
  form = GPP_NT_VUT_REF ~ .,
  data = ddf_train %>%
    drop_na() %>%
    select(-TIMESTAMP),
  method = "lm",
  trControl = trainControl(method = "none")
)

summary(linmod_caret)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.1941 -1.0393 -0.1299  0.8859  7.7205
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.9570704  3.4840107  -0.275 0.783558
## TA_F         0.1633211  0.0108181  15.097  < 2e-16 ***
## SW_IN_F      0.0137226  0.0003914  35.064  < 2e-16 ***
## LW_IN_F      0.0207311  0.0012040  17.219  < 2e-16 ***
## VPD_F       -0.1420886  0.0195509  -7.268 4.35e-13 ***
## PA_F        -0.0402727  0.0404675  -0.995 0.319704
## P_F         -0.0214314  0.0046177  -4.641 3.57e-06 ***
## WS_F        -0.1418356  0.0398616  -3.558 0.000378 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.549 on 4135 degrees of freedom
## Multiple R-squared:  0.6904, Adjusted R-squared:  0.6899
## F-statistic:  1317 on 7 and 4135 DF,  p-value: < 2.2e-16
```

8.3.2 Prediction

With the model containing all predictors and fitted on `ddf_train`, make predictions using first `ddf_train` and then `ddf_test`. Compute the R^2 and the root-mean-square error, and visualise modelled vs. observed values to evaluate both predictions.

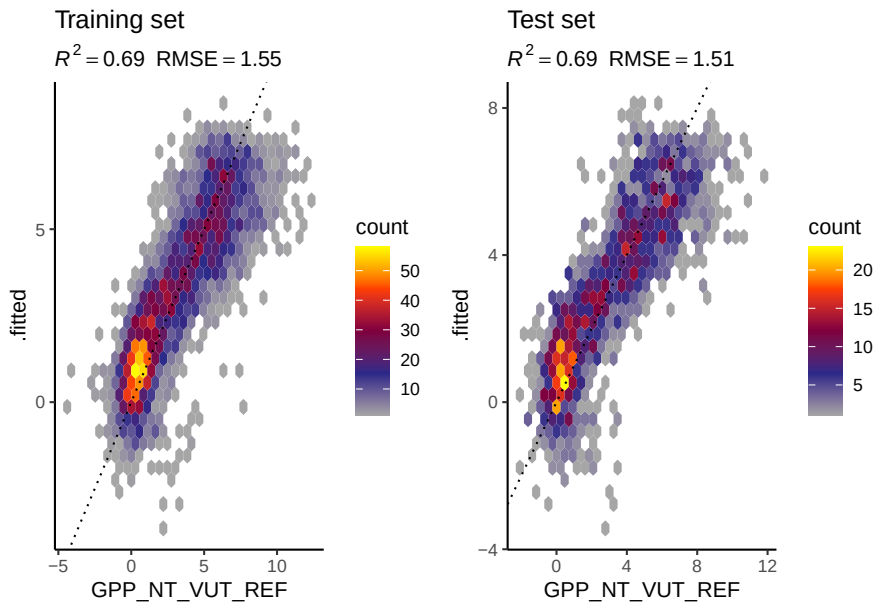
Do you expect the linear regression model trained on `ddf_train` to predict substantially better on `ddf_train` than on `ddf_test`? Why (not)?

Hints:

- To calculate predictions, use the generic function `predict()` with the argument `newdata = ...`.
- The R^2 can be extracted from the model object as `summary(model_object)$r.squared`, or is (as the RMSE) given in the metrics data frame returned by `metrics()` from the *yardstick* library.
- For visualisation the model performance, consider a scatterplot, or (better) a plot that reveals the density of overlapping points. (We're plotting information from over 4000 data points here!)

```
library(patchwork)
library(yardstick)

## made into a function to reuse code below
eval_model <- function(mod, df_train, df_test){

  ## add predictions to the data frames
  df_train <- df_train %>%
    drop_na() %>%
    mutate(.fitted = predict(mod, newdata = .))

  df_test <- df_test %>%
    drop_na() %>%
    mutate(.fitted = predict(mod, newdata = .))

  ## get metrics tables
  metrics_train <- df_train %>%
    yardstick::metrics(GPP_NT_VUT_REF, .fitted)

  metrics_test <- df_test %>%
    yardstick::metrics(GPP_NT_VUT_REF, .fitted)

  ## extract values from metrics tables
  rmse_train <- metrics_train %>%
    filter(.metric == "rmse") %>%
```

```
    pull(.estimate)
  rsq_train <- metrics_train %>%
    filter(.metric == "rsq") %>%
    pull(.estimate)

  rmse_test <- metrics_test %>%
    filter(.metric == "rmse") %>%
    pull(.estimate)
  rsq_test <- metrics_test %>%
    filter(.metric == "rsq") %>%
    pull(.estimate)

  ## visualise with a hexagon binning and a customised color scale,
  ## adding information of metrics as sub-titles
  gg1 <- df_train %>%
    ggplot(aes(GPP_NT_VUT_REF, .fitted)) +
    geom_hex() +
    scale_fill_gradientn(
      colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
    geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
    labs(subtitle = bquote( italic(R)^2 == .(format(rsq_train, digits = 2)) ~~
                            RMSE == .(format(rmse_train, digits = 3))),
         title = "Training set") +
    theme_classic()

  gg2 <- df_test %>%
    ggplot(aes(GPP_NT_VUT_REF, .fitted)) +
    geom_hex() +
    scale_fill_gradientn(
      colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
    geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
    labs(subtitle = bquote( italic(R)^2 == .(format(rsq_test, digits = 2)) ~~
                            RMSE == .(format(rmse_test, digits = 3))),
         title = "Test set") +
    theme_classic()

  return(gg1 + gg2)
}

eval_model(mod = linmod_baser, df_train = ddf_train, df_test = ddf_test)
```

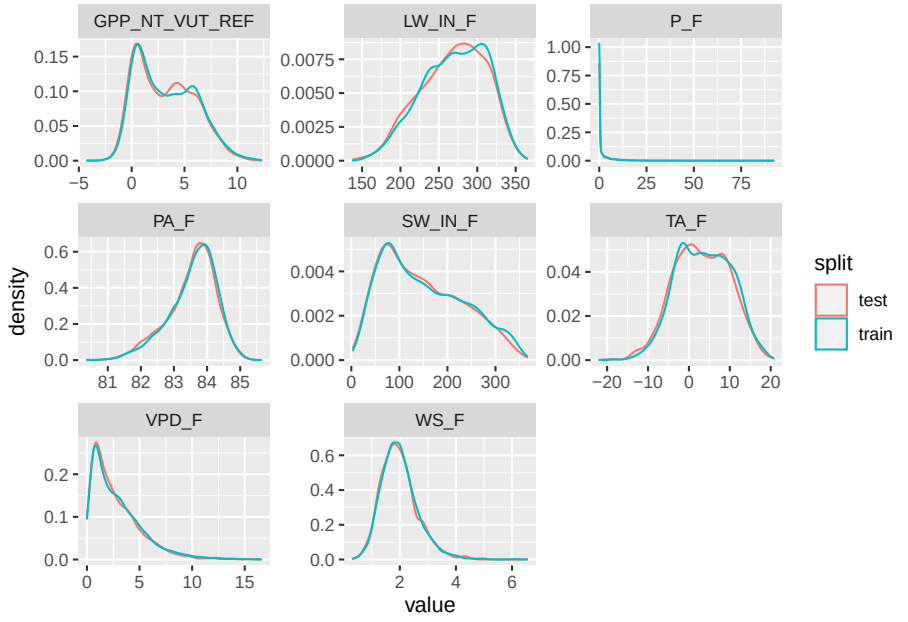


8.4 KNN

8.4.1 Check data

```
ddf_train %>%
  mutate(split = "train") %>%
  bind_rows(ddf_test %>%
    mutate(split = "test")) %>%
  pivot_longer(cols = 2:9, names_to = "variable", values_to = "value") %>%
  ggplot(aes(x = value, y = ..density.., color = split)) +
  geom_density() +
  facet_wrap(~variable, scales = "free")
```

```
## Warning: Removed 681 rows containing non-finite values (stat_density).
```



The variable `PA_F` looks weird and was not significant in the linear model. Therefore, we won't use it for the models below.

8.4.2 Training

Fit two KNN models on `ddf_train` (excluding `"PA_F"`), one with $k = 2$ and one with $k = 30$, both without resampling. Use the RMSE as the loss function. Center and scale data as part of the pre-processing and model formulation specification using the function `recipe()`.

```
library(recipes)

## model formulation and preprocessing specification
myrecipe <- recipe(
  GPP_NT_VUT_REF ~ TA_F + SW_IN_F + LW_IN_F + VPD_F + P_F + WS_F,
  data = ddf_train %>% drop_na()) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes())

## fit model with k=2
mod_knn_k2 <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
```

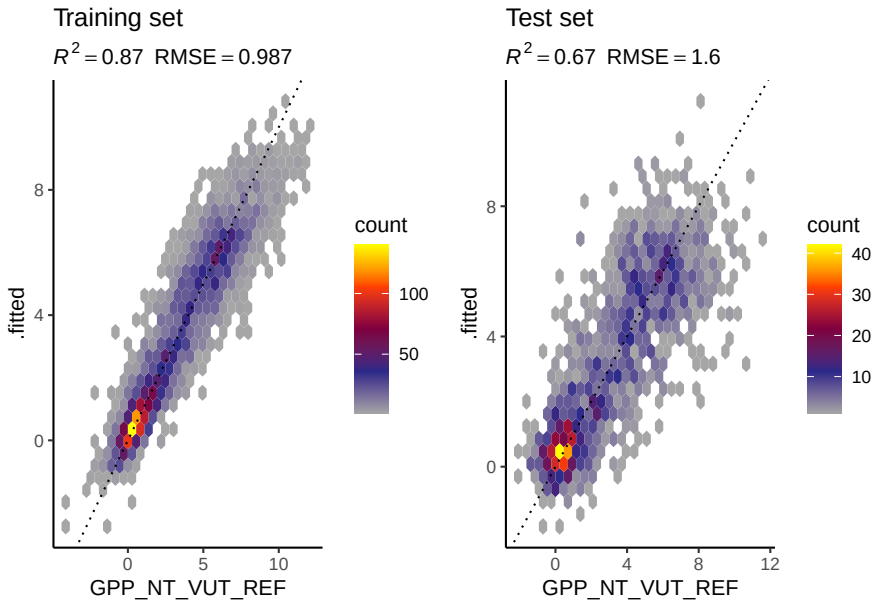
```
  method = "knn",
  trControl = trainControl("none"),
  tuneGrid = data.frame(k = c(2)),
  metric = "RMSE"
  )

## fit model with k=30
mod_knn_k30 <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "knn",
  trControl = trainControl("none"),
  tuneGrid = data.frame(k = c(30)),
  metric = "RMSE"
  )
```

8.4.3 Prediction

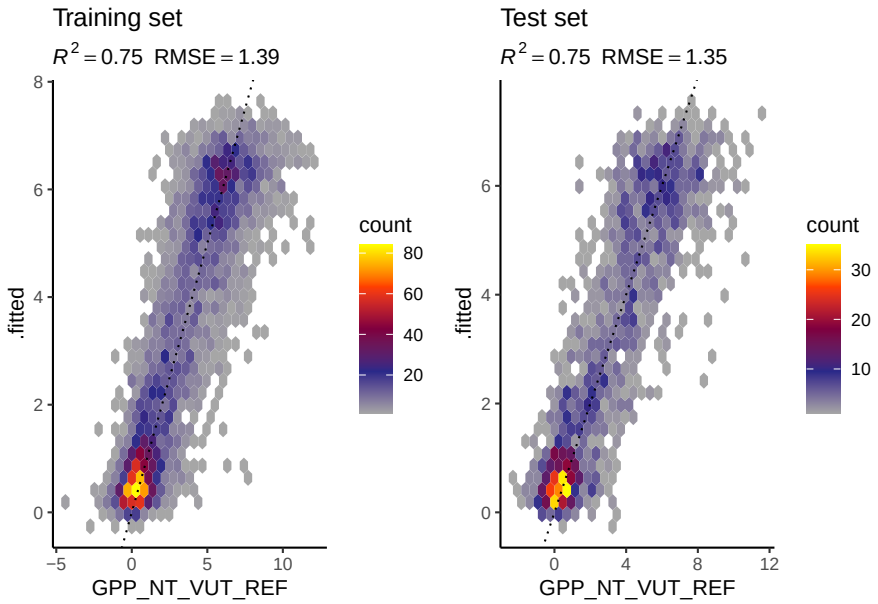
With the two models fitted above, predict `"GPP_NT_VUT_REF"` for both and training and the testing sets, and evaluate them as above (metrics and visualisation).

Which model do you expect to perform better better on the training set and which to perform better on the testing set? Why?

```
## with k = 2
eval_model(
  mod_knn_k2,
  df_train = ddf_train,
  df_test = ddf_test
  )
```




```
## with k = 30
eval_model(
  mod_knn_k30,
  df_train = ddf_train,
  df_test = ddf_test
  )
```



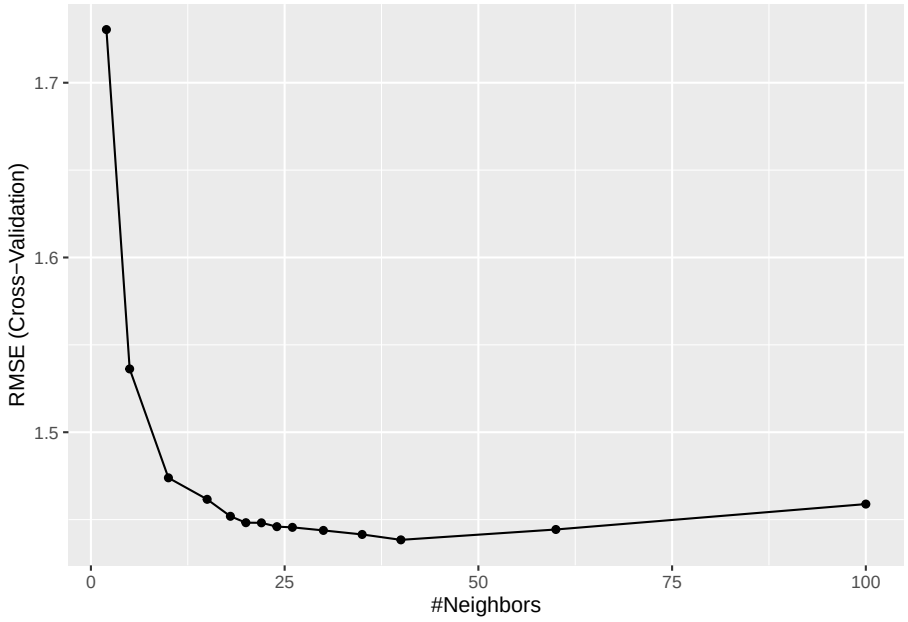
8.4.4 Sample hyperparameters

Train a KNN model with hyperparameter (k) tuned, and with five-fold cross validation, using the training set. As the loss function, use RMSE. Sample the following values for k: 2, 5, 10, 15, 18, 20, 22, 24, 26, 30, 35, 40, 60, 100. Visualise the RMSE as a function of k.

Hint:

- The visualisation of cross-validation results can be visualised with the `plot(model_object)` of `ggplot(model_object)`.

```
mod_knn <- train(
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "knn",
  trControl = trainControl(method = "cv", number = 5),
  tuneGrid = data.frame(k = c(2, 5, 10, 15, 18, 20, 22, 24, 26, 30, 35, 40, 60, 100)),
  metric = "RMSE"
  )
ggplot(mod_knn)
```



8.5 Random forest

8.5.1 Training

Fit a random forest model with `ddf_train` and all predictors excluding `"PA_F"` and five-fold cross validation. Use RMSE as the loss function.

Hints:

- Use the package *ranger* which implements the random forest algorithm.
- See here for information about hyperparameters available for tuning with caret.
- Set the argument `savePredictions = "final"` of function `trainControl()`.

```
library(ranger)

## no pre-processing necessary
myrecipe <- recipe(
  GPP_NT_VUT_REF ~ TA_F + SW_IN_F + LW_IN_F + VPD_F + P_F + WS_F,
  data = ddf_train %>%
    drop_na())

rf <- train(
```

```
  myrecipe,
  data = ddf_train %>%
    drop_na(),
  method = "ranger",
  trControl = trainControl(method = "cv", number = 5, savePredictions = "final"),
  tuneGrid = expand.grid( .mtry = floor(6 / 3),
                          .min.node.size = 5,
                          .splitrule = "variance"),
  metric = "RMSE",
  replace = FALSE,
  sample.fraction = 0.5,
  num.trees = 2000,          # high number ok since no hperparam tuning
  seed = 1982                # for reproducibility
)
rf
```

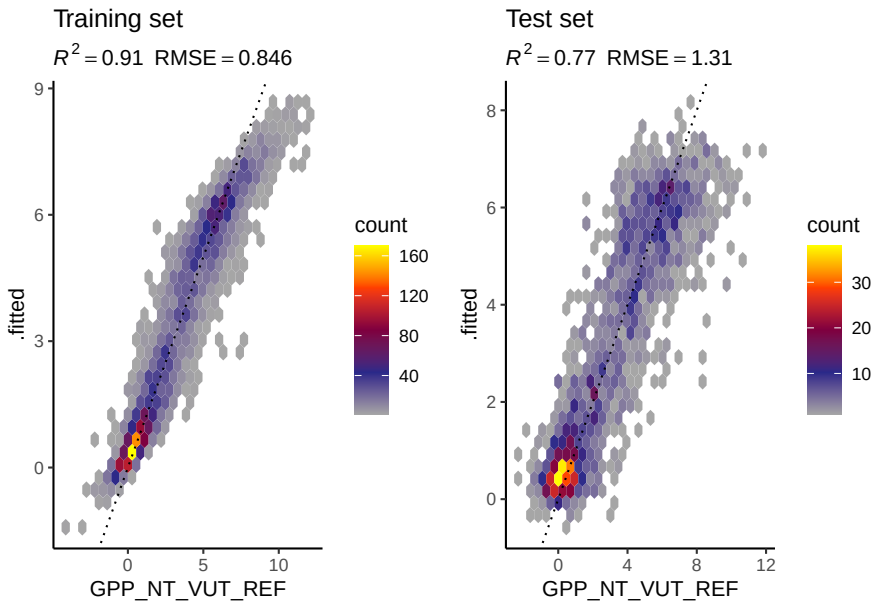
```
## Random Forest
##
## 4143 samples
##    8 predictor
##
## Recipe steps:
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 3313, 3315, 3315, 3314, 3315
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   1.401107  0.746519  1.056488
##
## Tuning parameter 'mtry' was held constant at a value of 2
## Tuning
##  parameter 'splitrule' was held constant at a value of variance
##
## Tuning parameter 'min.node.size' was held constant at a value of 5
```

8.5.2 Prediction

Evaluate the trained model on the training and on the test set, giving metrics and a visualisation as above.

How are differences in performance to be interpreted? Compare the performances of linear regression, KNN, and random forest, considering the evaluation on the test set.

```
eval_model(rf, df_train = ddf_train, df_test = ddf_test)
```



Show the model performance (metrics and visualisation) on the validation sets all cross validation folds combined.

Do you expect it to be more similar to the model performance on the training set or the testing set in the evaluation above? Why?

```
metrics_train <- rf$pred %>%
  yardstick::metrics(obs, pred)

rmse_train <- metrics_train %>%
  filter(.metric == "rmse") %>%
  pull(.estimate)
rsq_train <- metrics_train %>%
  filter(.metric == "rsq") %>%
  pull(.estimate)

rf$pred %>%
  ggplot(aes(obs, pred)) +
  geom_hex() +
  scale_fill_gradientn(
    colours = colorRampPalette( c("gray65", "navy", "red", "yellow"))(5)) +
  geom_abline(slope = 1, intercept = 0, linetype = "dotted") +
  labs(subtitle = bquote( italic(R)^2 == .(format(rsq_train, digits = 2)) ~~
```

```
                          RMSE == .(format(rmse_train, digits = 3))),
      title = "Validation folds in training set") +
theme_classic()
```