**Inhalt**

[1 Ressourcen 2](#_Toc165445689)

[2 Grund-Prozedur 2](#_Toc165445690)

[2.1 Split the data 2](#_Toc165445691)

[2.2 Create the recipe: Set up the Data and Model Formula 6](#_Toc165445692)

[2.3 Specify the model 11](#_Toc165445693)

[2.4 Fit the data 12](#_Toc165445694)

[2.5 The Final Test: Anwendung des Modells auf die Testdaten 18](#_Toc165445695)

[2.6 Hinweis: Parallel processing 19](#_Toc165445696)

[3 Das Workflow Package 21](#_Toc165445697)

[3.1 Import der Daten 21](#_Toc165445698)

[3.2 Split der Data 21](#_Toc165445699)

[3.3 Create the Recipe 21](#_Toc165445700)

[3.4 Specify the Model 21](#_Toc165445701)

[3.5 NEU: Add to Workflow 21](#_Toc165445702)

[3.6 Fit the Data 22](#_Toc165445703)

[3.7 Evaluate the Model 23](#_Toc165445704)

[3.8 The Final Test 23](#_Toc165445705)

[4 Klassifikation 24](#_Toc165445706)

[4.1 Import der Daten 24](#_Toc165445707)

[4.2 Split the data 24](#_Toc165445708)

[4.3 Create the recipe 25](#_Toc165445709)

[4.4 Specify the model 25](#_Toc165445710)

[4.5 Add to workflow 25](#_Toc165445711)

[4.6 Fit the Data 26](#_Toc165445712)

[4.7 NEU: Evaluieren der Klassifikationsgüte 26](#_Toc165445713)

[4.8 The Final Test 29](#_Toc165445714)

[5 Hyperparameter (HP) tuning 31](#_Toc165445715)

[5.1 Background 31](#_Toc165445716)

[5.2 NEU: Tuning-Grid generieren 32](#_Toc165445717)

[5.3 NEU: Train the Model mit tune\_grid() 34](#_Toc165445718)

[5.4 NEU: Finalisieren (updaten) des workflows für das finale Modell 34](#_Toc165445719)

[5.5 Vollständiges Beispiel mit KNN 35](#_Toc165445720)

[5.6 Hinweis zum housekeeping und Bennennung der Objekte 42](#_Toc165445721)

[6 Variable Importance 43](#_Toc165445722)

[6.1 Einfacher Fall: Koeffizienten und deren Signifikanz 43](#_Toc165445723)

[6.2 Variable importance plots bei nicht-parametrischen Modellen 43](#_Toc165445724)

[6.3 Besonderheit 1: VIP bei Regularisierung 44](#_Toc165445725)

[6.4 Bsonderheit 2: VIP bei random forests 44](#_Toc165445726)

[7 Out-of-Sample Prediction 45](#_Toc165445727)

[8 Time Series Forecasting 47](#_Toc165445728)

[9 Zusammenfassung 48](#_Toc165445729)

# Ressourcen

* **Website:** [www.tidymodels.org](http://www.tidymodels.org)
* **Tutorial:** <https://www.tidymodels.org/start/models/> **!!!**
* 4 Videos, die die Grundkonzepte zeigen. <https://www.youtube.com/watch?v=BNC-gcKm0kI>
* <https://emilhvitfeldt.github.io/ISLR-tidymodels-labs/index.html> Tidy models-Ansatz zum statistical learning Buch von James et al.
* <https://www.rebeccabarter.com/blog/2020-03-25_machine_learning/> Blog von Rebecca Barter
* <https://www.tidymodels.org/> Tidymodels Website mit tutorial
* <https://www.tmwr.org/> Zentrale Ressource: Buch von Kuhn und Silge
* <https://towardsdatascience.com/comparative-study-on-classic-machine-learning-algorithms-24f9ff6ab222> Überblick über alle Modelltypen; ihre Vor – und Nachteile, Annahmen und Vergleiche mit anderen zum selben Zweck.
* Companion zum tidymodels Buch: <https://r4ds.github.io/bookclub-tmwr/>

# Grund-Prozedur

Der workflow besteht aus folgenden Schritten (die in den nächsten Kapiteln noch etwas erweitert werden).

* **Schritt 1: Split the data** (🡪 Trainings- und Testdaten, bzw. diverse folds)
* **Schritt 2: Create a recipe** (data preprocessing und model formula)
* **Schritt 3: Specify the model**
  + **Pick the model:** Z.B. lineare Regression oder Regression tree etc.?
  + **Set the engine:** Wahl aus einer zum Modell passenden Collection von Paketen
* **Schritt 4: Fit the data** **/ train the model** (in den Trainingsdaten oder k-1 folds der Trainingsdaten)
* **Schritt 5: Evalute the model** zuerst mit den Trainingsdaten, dann mit den Testdaten

## Split the data

* Für Hintergrund-Infos siehe resampling-[Kapitel](https://www.tidymodels.org/start/resampling/) im Tidymodels Buch
* Zuerst werden die Daten in Training Set und Test Set geteilt. Dabei gibt es zwei Methoden:
  + **Bei größeren Datensätzen** wird der Trainingsdatensatz noch mal in k folds geteilt (kfold cross-validation).
  + **Bei kleinen Datensätzen** gibt es 2 Möglichkeiten:
  + Es bleibt beim simplen Split in Trainings- und Test-Daten (klassischer hold-out Ansatz, auch Train-Test-Only, oder kurz **TT only**). Dabei im gesamten Training Set das Modell gefitted und mit dem Test Set evaluiert. Von allen Varianten der nachteiligste, weil die Performance Metrik durch die Zufälligkeit des Splits in Training und Test Set beeinflusst wird (Details, sie Resampling Kapitel in James et al.)
  + Man zieht aus dem Training Set eine Hand voll Bootrap samples. In denen wird das Modell gefittet, vorzeitig evaluiert („Training performance“) und dann final am Test Set evaluiert. Dieser Ansatz versucht, das kfold CV zu approximieren (mehr zum Unterschied später).
* Das folgende Beispiel wird mit den **Advertising-Daten aus Kapitel 3 in James et al.** durchgeführt

library(ISLR)

advertising\_data <-

read\_csv("https://www.statlearning.com/s/Advertising.csv") %>%

select(-1)

### Kleine Datensätze: „TT only“ und Bootstrapping

* Häufig gibt die Größe der Daten CV nicht her. Die simple Lösung ist daher, das es bei der initialen Teilung in Trainings- und Testdaten zu belassen; das Modell zu fitten und im Test Set zu evalulieren.
* Ein Problem davon ist im James et al. Kapitel über resampling beschrieben: Der initiale Split ist random. Würde man ihn mehrfach durchführen, gäbe es eine Varianz in der performance.
* Es kann daher sein, dass der performance Test durch Glück zu gut oder durch Pech zu schlecht ausieht.
* (Eine Alternative zum simplen TT only-Ansatz ist daher Bootstraping. Dies kommt nach der Beschreibung von CV)

#### Vorgehen beim TT only

* **Initialer Split:** Die Daten werden in zwei Hälften geteilt (Training Set vs. Test Set)

set.seed(123)

advertising\_split = initial\_split(advertising\_data, prop=.5)

* + set.seed(123). Wie jede „Zufallsziehung“ in R ist der Split pseudozufällig—d.h. eine Zufallsziehung wird simuliert. Dies hat den Vorteil, dass man das Zufallsereignis durch sog. „seeds“ wiederholen kann. Da im gesamten workflow mehrere solcher „zufälligen“ Ziehungen durchgeführt werden und diese das Ergebnis (v.a. in kleineren Datensätzen) beeinflussen kann, muss man zumindest die Wiederholbarkeit bestimmen. Die set.seed-Funktion macht das. Die Zahlen sind Startwerte für den Prozess und inhaltlich unbedeutend.
  + prop: Relative Größe der Teildaten
  + Mittels optionalem strata -Argument lässt sich für sicherstellen, dass sie gleiche Verhältnisse in Trainings und Testdaten hat (also z.B. strata = sex)
  + Die Funktion führt nicht wirklich einen split aus—stattdessen speichert sie lediglich die split-Information:

advertising\_split

<Training/Testing/Total>

<100/100/200>

* + Wie man sieht, enthält das Objekt split-Infos—also das N in Trainings- und Testdaten (jeweils N=100) und das Gesamt-N.
* **Aufteilung in TT:** Diese wird mit den Funktionen training() und testing() auf Basis des split-Objekts vorgenommen.

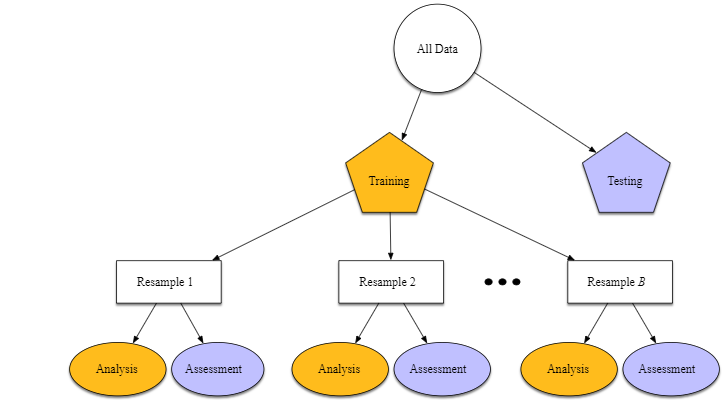
advertising\_train = training(advertising\_split)

advertising\_test = testing(advertising\_split)

### Große Datensätze: kfold crossvalidation

#### Grundlegendes zu Cross-Validation

* Bzgl. CV gibt es ein Grundkonzept, wie es in James et al. oder [Wikipedia](https://en.wikipedia.org/wiki/Cross-validation_(statistics)) beschrieben ist—die Tidymodels-Version weicht etwas davon ab. Die Grundidee ist, die gesamten Daten in k folds zu teilen und anschließend das Modell dann in k-1 folds zu trainieren und im kten validieren. Anschließend wird dies wiederholt mit einem anderen set von folds als Training und einem anderen als validation fold, siehe auch das [Video](https://www.youtube.com/watch?v=rSGzUy13F_0&list=PL5-da3qGB5IA6E6ZNXu7dp89_uv8yocmf&index=2) von Hastie und Tibshirani und die Abb. auf der Wikipedia-Seite.
* Im Tidymodels framework gibt es 2 Abweichungen. Die Abb. im resampling-Kapitel zeigt das:



* 1. Hier werden die gesamten Daten zuerst in ein Training Set und ein Test Set geteilt; anschließend wird nur das **Training Set** in k folds geteilt (nicht die gesamten Daten). Das Ziel hierbei ist, einen Teil der Daten vom fitting-procedure völlig abzukoppeln und final zu testen. Dies ist v.a. dann sinnvoll, wenn man mehrere *Modelle* *vergleichen* oder *Hyperparameter tunen* möchte. Die „Metriken“ (MSE, accuracy, ROC etc.), die man in jedem der Assessment-Samples erhebt (und mittelt) dienen bei der Entscheidung. Am Ende wird das Gewinnermodell final an den Testdaten evaluiert. Das ist die Stunde der Wahrheit.
  2. Damit geht auch eine etwas andere (aber präzisere) Begrifflichkeit gewählt (siehe Abb.). An diese werde ich mich im Skript halten. Wie die Trennung in die folds passiert, hängt davon ab, ob man **kfold CV** benutzt, **Monte-Carlo CV** oder **bootstrapping**. Dies erklär ich weiter unten.

#### Durchführung

* **Initialer Split:** Die Daten werden wieder in zwei Hälften geteilt (Training Set vs. Test Set)

set.seed(123)

advertising\_split = initial\_split(advertising\_data, prop=.5)

* + prop: Relative Größe der Teildaten
  + mittels strata-Argument lässt sich für manche Variablen sicherstellen, dass sie gleiche Verhältnisse in Trainings und Testdaten hat (also z.B. strata = sex)
* **Bilden der Trainings- und Testdaten:**

mtcars\_train = training(mtcars\_split)

mtcars\_test = testing(mtcars\_split)

* **kfold Crossvalidation: 5 folds aus dem Trainingsteil machen**

advertising\_cv <- vfold\_cv(advertising\_train, v=5)

# 5-fold cross-validation

# A tibble: 5 × 2

splits id

<list> <chr>

1 <split [80/20]> Fold1

2 <split [80/20]> Fold2

3 <split [80/20]> Fold3

4 <split [80/20]> Fold4

5 <split [80/20]> Fold5

* + Die Analysis vs. Assessment sets addieren sich *in jedem fold* auf das N=100 (durch Zufall ist im ersten fold eine leicht andere Aufteilung)
  + Außerdem sieht man dass die Quelle die Trainingsdaten sind (advertising\_train)
  + Das Ergebnis ist ein tibble, in dem die Sub-Datensatzes genested sind
  + Die jeweils 20 Fälle sind nicht-überlappende Teile des Trainings-Sets. Hier eine Visualisierung: Jede Zeile ist der gesamte Trainings-Datensatz—die weißen Teile (N=80) sind das Analysis Set in dem das Modell trainiert wird während der graue Teil das Assessment Set (N=20)ist, in dem das Modell verwendet wird, um die Performance Metrik zu erhalten.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Fold1 |
|  |  |  |  |  | Fold2 |
|  |  |  |  |  | Fold3 |
|  |  |  |  |  | Fold4 |
|  |  |  |  |  | Fold5 |

### Bootstrapping als Approximation von CV in kleinen samples

* Wie anfangs beschrieben ist der TT only-Ansatz nicht optimal wegen der randomness dieses einen splits. Daher ist Bootstrapping eine Alternative.
* Dabei werden aus dem Training Set zufällige Ziehungen *mit Zurücklegen* vorgenommen, bis das N des Training Sets erreicht ist. D.h. ein Fall kann mehrfach in einem Analysis Set sein. In das Assessment set kommen dann jeweils immer die Fälle, die es nicht in das Analysis Set geschafft haben („the remainder“).
* Die Folge ist, dass das N in sowohl Analysis als auch Assessment Set höher als beim CV ist. Im Assessment Set werden daher mehr als 1/k (CV).

advertising\_boot = bootstraps(advertising\_train, times = 5)

advertising\_boot

# Bootstrap sampling

# A tibble: 5 × 2

splits id

<list> <chr>

1 <split [100/37]> Bootstrap1

2 <split [100/36]> Bootstrap2

3 <split [100/35]> Bootstrap3

4 <split [100/36]> Bootstrap4

5 <split [100/35]> Bootstrap5

Die erste Zahl sind die gebootstrappten Fälle (mit Zurücklegen)—die zweite die „remainder“—diese bilden dass Assessment Set und werden zur Evaluation herangezogen.

## Create the recipe: Set up the Data and Model Formula

### Zweck des Recipes

* Das Recipe hat zwei Inhalte:

1. **Definieren der model formula**, d.h. des „targets“ (Outcomes) und der Prädiktoren
2. **Data preprocessing**: Standardisieren, Transformieren (z.B. Logarithmieren), Dummy-Bildung etc.

* Dies macht man **nach** dem split in Trainings- und Testdaten, weil sonst die splits unterschiedliche SDs hätten, was beim Standardiseren die Variablen verändert
* Das Recipe stellt somit eine Art „Steuerungsobjekt“ dar, in dem Metadaten gespeichert sind. Auf diese wird dann später beim Training des Modells zurückgegriffen.
* **Beispiel:**

advertising\_rec <- recipe(sales ~. , data = advertising\_train)%>%

step\_normalize(all\_predictors())

* + Erste Zeile ist die formula—hier wird mpg einfach auf alles andere in den Daten regressiert. Alternativ könnte man die Prädiktoren wie gewöhnlich aufzählen (x1+x2+x3)
  + Zweite Zeile ist ein *beispielhafter preprocessing*-Schritt (hier: alle Prädiktoren standardisieren). Neben all\_predictors gibt es weitere handliche Argumente (all\_outcomes, all\_numeric\_predictors, all\_nominal\_predictors etc.). Die Tabelle in Abschnitt 2.2.3 enthält die wichtigsten step-artigen Preprocessing-Möglichkeiten. Meine Standard steps sind immer step\_normalize(), step\_zv() und step\_dummy().
* Aufrufen des Recipes zeigt den Inhalt (Definition der Modellvariablen und preprocessing-Informationen)

advertising\_rec

Recipe

Inputs:

role #variables

outcome 1

predictor 4

Operations:

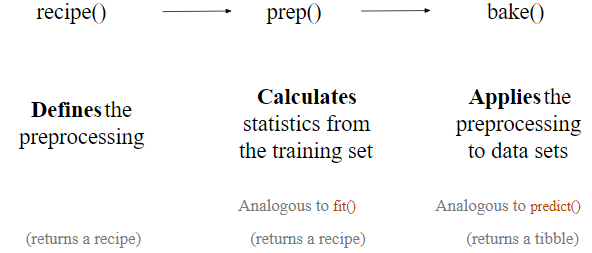
Scaling for all\_predictors()

* **Das Recipe hat 2 Funktionen** (alternativ oder parallel):
  + **Es wird später im Training benutzt.** Dort sind ja alle Informationen enthalten auf die das Modell dann zurückgreift.

Wenn man Variablen im Datensatz hat, die man nicht als Prädiktor oder Outcome möchte, sondern als ID geht das mit update\_role(x, new\_role = "id") (siehe Tabelle)

* + Es ist Grundlage für einen **Check des Preprocessings**. Dieser Aspekt ist erst mal seltsam—allerdings sei betont, dass das Recipe ja nur ein Rezept ist. D.h. es ist eine Blaupause für die Schritte, die später in den k folds jedesmal aufs Neue durchgeführt werden.
  + Um es vorab an den tatsächlichen (oder anderen) Daten tatsächlich anzuwenden, können die Funktionen prep() und bake() oder juice() verwendet werden (siehe nächster Abschnitt). Ich mache das jedesmal, um zu checken, ob auch alles so läuft, wie beabsichtigt.

### Prep(), bake() und juice()



* Siehe kurzer [post](https://stackoverflow.com/questions/62189885/what-is-the-difference-among-prep-bake-juice-in-the-r-package-recipes) von Julia
* Das recipe selbst (wenn man es aufruft) ist nicht informativ und enthält nur Meta-Daten. Es ist ja auch nur ein Rezept (kein Kuchen).
* Wie oben beschrieben erlauben prep(), bake() und juice(), das recipe auf Daten anzuwenden und diese anzuschauen. Mit anderen Worten: Den Kuchen zu backen und zu essen (Quelle: Barter-[Blog](https://www.rebeccabarter.com/blog/2020-03-25_machine_learning/)):
  + Mittels prep() wird das Recipe auf die Daten dahingehend angewendet, dass es die allgemeinen steps mit den Variablen verbindet. Noch immer wird aber nichts tatsächlich gemacht. Die Zutaten für den Kuchen werden nur bereitgelegt.
  + bake() schließlich wendet das „gepreppte“ recipe auf die Daten an (backt den Kuchen). Dabei muss mittels new\_data spezifiziert werden, worauf es angewendet werden soll (Gesamtdaten, Training Set oder Test Set). Im Grunde ist das egal, da es ja nur verschiedene Versionen derselben Daten sind. Hier am Beispiel der Anwendung auf das Training Set.

Wie man sieht, kann man jede Teilfunktion „pipen“

advertising\_rec %>%

prep() %>%

bake(new\_data= advertising\_train)

# A tibble: 100 × 4

TV radio newspaper sales

<dbl> <dbl> <dbl> <dbl>

1 1.20 0.246 -0.320 18.9

2 1.36 -1.53 -0.484 12

3 -0.000849 0.411 -0.818 15

4 -0.968 0.674 -0.291 11.8

5 0.545 0.366 -0.547 17.3

6 0.321 1.19 0.396 19

7 0.674 -0.115 0.101 16.6

8 0.870 0.727 0.754 19.6

9 -1.49 0.983 0.759 7.3

10 0.846 0.727 1.43 19.4

* + juice() ist ein netter shortcut zu bake() und wendet automtisch das gepreppte recipe auf diejenigen Daten an, die im recipe verwendet wurden. Wenn man das recipe auf alle Daten anwenden möchte, sollte man das eher explizit in der bake-Funktion machen.
* Beispiel (gesamt):

advertising\_rec <- recipe(~., data = advertising\_train)%>% #Recipe bilden

step\_normalize(all\_predictors())

mtcars\_rec %>% #prep() & juice

prep() %>%

juice()

Wie gesagt: Man kann auch recipe an Datensatz X bilden und in Datensatz Y juicen—Hauptsache die Variablen sind gleich benannt. Ist das nicht der Fall, bekommt man eine genaue Fehlermeldung, welche Variablen nicht identisch sind.

### Preprocessing-Varianten

* Alle verfügbaren preprocessing-Funktionen findet man auf [hier](https://recipes.tidymodels.org/reference/index.html)-- darunter auch logarithmieren, missings imputieren, dummy Bildung.
* Julia Silge hat noch andere Funktionen, wenn Variablen aus Text (!) bestehen (siehe [Code](https://juliasilge.com/blog/lasso-the-office/))
* Die wichtigsten sind in der nachfolgenden Tabelle

|  |  |  |
| --- | --- | --- |
| **Funktion** | **Code** | **Kommentar** |
| **Prädiktoren mit zero variance entfernen** | step\_zv() |  |
| **Dummy-Variablen anlegen** | step\_dummy() | Wichtig:   * Binäre *Outcomes* müssen Faktoren bleiben * ! Der Schritt generiert 0/1 Variablen, die aber von der Klasse „dbl“ sind! * 🡪 Muss NACH normalize kommen (ansonsten wird über die dummies standardisiert). (effect coding) * Kommt anschließend step\_pca, werden die dummies einbezogen! |
| **Zentrieren** | step\_center() | Zentriert die Variablen, womit sie M=0 haben (die SD und Metrik bleibt gleich) |
| **Skalieren** | step\_scale() | SD =1 (M ist unverändert) |
| **Standardisieren** | step\_normalize() | M= 0, SD=1. Routine Operation bei allen quantitativen Prädiktoren |
| **Missing data imputation** (mittels KNN) | step\_impute\_knn() | step­\_impute hat zig Imputationsmethoden. Sieht man in Rstudio bei Autoausfüllen, allem voran mean oder mittels linear model (dies macht aber wohl nur Sinn bei metrischen Variablen). Achtung: k = 5 per default! |
| **Logarithmieren** | step\_log(size, base=10) | Variablen logarithmieren |
| **Polynome verwenden** | step\_poly |  |
| **Downsampling**  bei unbalancierten Outcomes | themis::step\_downsample(y) | Hier werden von den häufigen Fällen weniger ausgewählt. Es gibt auch step\_upsample() und step\_smote(), siehe Himalaya-[Video](https://www.youtube.com/watch?v=9f6t5vaNyEM) (31:30). |
| **Upsampling** | themis::step\_rose(y) #binär  themis::step\_smote(y) #multi | Random oversampling oder downsampling  y = Name der AV |
| **Datetimes in Monate und Tage zerlegen** | step\_date(  date,  features = c(“year, "month"),  role ="dates") | Wandelt eine date-Variable (z.B. „2022-10-04“) in beliebige features (hier: Jahr und Monat)  Die original-Variable wird anschließend gelöscht mit step\_rm(date) und die Monats oder Tagesvariablen wird mit step\_dummy(has\_roles("dates") in dummies zerlegt (ist bei Year nicht nötig—die wird numeric) |
| **Variable löschen** | step\_rm() | Ist sinnvoll, wenn man die Variable in einem vorherigen step genutzt hat und sie anschließend entfernt (rm = remove) |
| **Faktorenlevel poolen (**in eine „other“-Kategorie= | step\_other(  threshold = 0.03) | Der threshold ist die Granuliertheit, je höher um so mehr werden zusammengefasst  Anstelle all\_nominal\_predictors() kann man einfach mehrere Variablen nennen. Das führt in einem Schlag dazu, dass man bei Faktoren nicht so viele level hat [das mit dem threshold ist doof, allerdings kann man so eben alle auf einen Schlag machen] |
| **Variablen eine non-model-Rolle zuweisen** | update\_role(x,  new\_role="id") | Das kann man machen, wenn man eine Variable im Datensatz haben möchte, die aber nicht im fitting benutzt werden soll. |
| **PCA vorschalten** | step\_pca() |  |
| **Hoch korrelierende Präditkoren eliminieren** | step\_corr(., threshold = 0.8) | . = all\_predictors o.ä. |
| **Produktterm einfügen** | step\_interact( ~ x:y) |  |
| **Kategoriale Variable in numerische umwandeln** | step\_lencode\_glm(x, outcome = vars(y) | Wenn die kategoriale *sehr* viele levels hat (high cardinality), kann sie in eine numerische Variable umgewandelt werden |
| **Text mining** |  |  |
| **Text-Korpus in tokens zerlegen** | step\_tokenize(text) | Hinweise: 1) Alle Text-preprep-Funktionen sind im textrecipes package  2) Alle Werte lassen sich tunen |
| **ngrams berücksichtigen** | step\_ngram(text,  num\_tokens = 2,  min\_num\_tokens = 2) |  |
| **Anzahl der benutzten Tokens bestimmen** | step\_tokenfilter(  text,  max\_tokens =1000) |  |
| **Tokens gewichten** | step\_tfidf(text) | Gewichtet die tokens gemäß ihrer “uniqueness” (siehe Bigfoot video). Dies bewirkt, dass die DTM aus den tf-idf-Werten besteht und nich aus dem N des Worts |
| **Stopwords entfernen** | step\_stopwords(text) |  |
| **Textfeautures extrahieren** | step\_textfeature(text) | Extrahiert 20 features (z.B. Anzahl der Worte bis hin zu 4 verschiedenen Sentiment scores). Im Gegensatz zur textfeature |
| **Word embedding Dimensionen extrahieren** | step\_word2vec(text) | Mit size kann die Anzahl der Dimensionen festgelegt werden. Default ist 10. Eine alternative Anzahl kann mit num\_topics gehält werden.  Achung: Man kann das Modell nur entweder nach einer word-count-Methode trainieren (inkl. TF-IDF-Gewichtung) oder nach den Dimensionen. In dem Moment, wo man die Dimensionen extrahiert, gibt es in den Daten keinen Text mehr. Allerdings kann man vorher tokenizing, stopword-removal |
| **Stemming** | step\_stem(text) |  |

* Variablen kann man in Gruppen ansprechen
  + all\_predictors()
  + all\_outcomes()
  + all\_nomin
  + all\_predictors()
  + all\_numeric\_predictors()
  + all\_nominal()
  + all\_numeric()
* ACHTUNG. Man kann auch bestimmte Variablennamen in einer Klammer auflisten und somit gezielt Variablen verarbieten (z.B. step\_dummy(gender, education) )

## Specify the model

* Dies ist der zweite Schritt (nach dem recipe). Hier geht’s jetzt darum,
  + - * 1. den **Modelltyp** zu wählen (Regression, random forest, LASSO)
        2. den **Modus** festzulegen (d.h. regression vs. Classification)
* Geht über das parsnip package (das man über tidymodels automatisch lädt)

### Wahl des Modelltyps

* **Pick the model:** Wahl des Modells nach der zu lösenden Aufgabe, z.B. Logistische Regression oder random forests.
* **Set the engine:** Je nach Modell gibt es verschiedene Pakete, die das jeweilige Modell rechnen können. Dies erlaubt, eine Vielzahl unterschiedlicher Modelle und den dazu passenden Paketen in einem konsistenten framework nutzen zu können
  + Beispiel: simple Regression

lm\_spec <- linear\_reg() %>%

set\_engine("lm")

* + Als engine soll hier die normale lm-Funktion, was dazu führt, dass ein OLS Modell geschätzt wird. Will man dagegen ein LASSO rechnen (was auch ein lineares Modell ist), geht das nur mit dem glmnet package (weil die lm-Funktion keine shrinkage-Option hat). Auch eine Bayes-Regression lässt sich mit der engine „stan“ rechnen.
  + Hinweis: Die engine ist meist keine problematische Wahl, das die gesemate „model spec“ sowiso copy&paste ist und sich die Frage nicht stellt.

### Wahl des Modus‘

* *Wenn*ein Modell sowohl regression als auch classification kann, muss man noch den Modus spezifizieren. Das geht, in dem man set\_mode("regression") anhängt.
* Beispiel:

tree\_spec <- decision\_tree() %>%

set\_engine("rpart") %>%

set\_mode("regression")

→ Fittet ein decision tree model mithilfe der rpart engine und modus regression (→ quantitatives outcome).

* Hier mal eine Übersicht über gängige Ansätze (beachtet die Groß-/Kleinschreibung bei den engines!)
* Auf <https://www.tidymodels.org/find/parsnip/> findet man eine große Liste von Models und engine-Möglichkeiten. Im TITLE feld kann man nach keywords suchen (→ Modellklasse)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modellklasse** | **Model** | **Engine** | **Mode** | |
| **Regression** | **Classification** |
| Lineare Regression | linear\_reg() | lm  glm  glmnet | **×** |  |
| Logistische Regression | logistic\_reg() | glm  glmnet  keras, LiblineaR, spark  brulee stan |  | **×** |
| multinom\_reg() | nnet |  | **×** |
| KNN | nearest\_neighbor() | kknn | **×** | **×** |
| Naive Bayes | naive\_Bayes() | Naivebayes | **×** | **×** |
| LASSO | linear\_reg() | glmnet | **×** | **×** |
| Decision trees | decision\_tree() | rpart  ranger | **×** | **×** |
| XGBoost | boost\_tree | C5.0  spark  xgboost | **×** | **×** |
| Random forests | rand\_forest() | ranger | **×** | **×** |
| Support vector machines | svm\_linear() | kernlab LiblinearR | **×** | **×** |

* Welches engine (=Paket) man wählt, ist meist egal—hängt im Einzelfall aber von eventuellen Spezifika ab, die u.U. Paket-spezifisch sind und die verfügbaren Argumente betreffen. Diese beziehen sich meist auf Hyperparameter (siehe Kapitel 5). Welche engine welche Argumente liefert, zeigt die 2. Tabelle auf der oben verlinkten Seite!

## Fit the data

* Nun geht’s los. Das Modell soll „trainiert“ werden—oder in traditionellerer Sprache—„gefittet“ werden. Dazu gibt im Tidymodels-framework verschiedene Möglichkeiten, je nach gewünschter Sophistiziertheit. Diese werden anschließend demonstriert:

1. **Einfaches Fitten über einen Datensatz** (sei es nur das Training Set oder sogar der gesamte Datensatz). Letzteres ist ja der gängige Ansatz in der wissenschaftlichen Anwendung von theoretisch-basierten Regressionsmodellen, die (siehe James et al.) die geringstmögliche Varianz haben und bei denen das Ausmaß von Unsicherheit über die SE adressiert wird. Dies passiert über eine einfache fit()-Funktion. Dies ist dann sehr nett, wenn man a) Koeffiizienten eines parametrischen Modell extrahieren sehen möchte, oder b) das Modell auf völlig neue Daten anwenden möchte und dazu noch mal das Modell auf die gesamten Daten fitten möchte (z.B. nach erfolgreicher Evaluation mittels CV).
2. **TT only:** Dies bedeutet, dass ein Training Set und ein Test Set vorliegt: Dabei wird das Modell auf die Trainingdaten gefittet und anhand der Testdaten evaluiert. Relevante Funktion ist last\_fit(). Natürlich kann parallel mittels fit() das Modell nur auf die Trainingsdaten gefittet werden, um die performance in den Trainingsdaten zu evalueiren (mehr darüber später). Hinweis: Diese Funktion wird auch im Falle des Crossvalidations am Ende des Prozesses verwendet, um das Modell final noch mal über den gesamten Datensatzu z fitten und am Testset zu evaluieren. Im Skript ist das immer unter dem Abscnitt „the final test“ zu finden.
3. **Crossvalditation oder Bootstrapping:** hier wird das Modell wird in jedem der folds/bootstraps mittels fit\_resamples() trainiert.
4. **Hyperparameter tuning:** Wie in Kapitel 5 noch behandelt wird, erlauben es Modelle, an Stellschrauben zu drehen und die Auswirkungen zu analysieren. Dies wird in der Regel mit Crossvalidation / Bootstrapping verbunden. Hier ersetzt man dann die o.g. Funktion durch tune\_grid()

### Einfaches Fitten auf einem einzigen Datensatz

* Hier nimmt man die Model-Spezfikation als Grundlage. Das recipe ist nicht nötig

lm\_spec %>%

fit(sales ~ ., data=advertising\_data) %>%

broom::tidy()

# A tibble: 4 × 5

term estimate std.error statistic p.value

<chr> <dbl> <dbl> <dbl> <dbl>

1 (Intercept) 2.94 0.312 9.42 1.27e-17

2 TV 0.0458 0.00139 32.8 1.51e-81

3 radio 0.189 0.00861 21.9 1.51e-54

4 newspaper -0.00104 0.00587 -0.177 8.60e- 1

Hinweis: Für diejenigen mit Erfahrung in R wird das seltsam erscheinen, gibt es doch die traditionelle Vorgehensweise mit lm(y ~x, data=dataset) und summary(). Es ist natürlich genauso möglich, diese zu verwenden.

### Bei trainings- vs. testdata only (“TT only”)

#### Training des Modells

* Die last\_fit-Funktion ist eine sehr komfortable Funktion, die das Model an den Trainingsdaten (gesamt) trainiert und automatisch an den Testdaten evaluiert (d.h. den prediction error anhand der performance Metriken im Test Set abschätzt):

lm\_fit\_tt <- last\_fit(lm\_spec,

advertising\_rec,

advertising\_split)

* Ihr input sind
  + Das **recipe** mit den Daten (inkl. preprocessing) und dem Modell—hier advertising\_rec
  + **das spec-Objekt** (mit Modelltype und engine)—hier lm\_spec
  + **das split-Objekt**, dass die Trainings- und Testdaten festlegt—hier advertising\_split. Darauf beruht das Wissen, wo trainiert und wo evaluiert wird.
* Ergebnis ist ein genestetes tibble, in dem eine Menge Informationen stecken und die extrahiert werden können

lm\_fit\_tt

# Resampling results

# Manual resampling

# A tibble: 1 × 6

splits id .metrics .notes .predi…¹ .workflow

<list> <chr> <list> <list> <list> <list>

1 <split [100/100]> train/tes… <tibble> <tibble> <tibble> <workflow>

* + Die **.metrics**-Spalte enthält die performance Metriken (z.B. rsquare oder rmse). Diese stammen aus der Anwendung des gefitteten Models auf dem Test Set.
  + Die **.predictions**-Spalte enthält die predictions der Y-Werte durch das modell im Test Set (dieses wrid später die Grundlage für genauere Analysen möglicher prediction errors sein)
  + Die **.workflow**-Spalte enhält die Parameter des Models aus dem Training Set

#### Extraktion interessanter Informationen

* **Extraktion der Performance-Metriken** (evaluiert am Test Set)

lm\_fit\_tt %>%

collect\_metrics()

# A tibble: 2 × 4

.metric .estimator .estimate .config

<chr> <chr> <dbl> <chr>

1 rmse standard 1.52 Preprocessor1\_Model1

2 rsq standard 0.922 Preprocessor1\_Model1

→ R-square ist .922 und RMSE ist 1.52 (bitte bedenken, dass es simulierte Daten sind!).

* **Extraktion der predictions**

lm\_fit\_tt %>%

collect\_predictions()

# A tibble: 60 × 5

id **.pred** .row **sales** .config

<chr> <dbl> <int> <dbl> <chr>

1 train/test split 12.3 2 10.4 Preprocessor1\_Model1

2 train/test split 12.3 3 9.3 Preprocessor1\_Model1

3 train/test split 12.4 10 10.6 Preprocessor1\_Model1

4 train/test split 6.98 11 8.6 Preprocessor1\_Model1

5 train/test split 18.3 15 19 Preprocessor1\_Model1

6 train/test split 23.1 18 24.4 Preprocessor1\_Model1

7 train/test split 9.96 19 11.3 Preprocessor1\_Model1

8 train/test split 14.2 20 14.6 Preprocessor1\_Model1

9 train/test split 17.0 28 15.9 Preprocessor1\_Model1

10 train/test split 19.4 29 18.9 Preprocessor1\_Model1

# … with 50 more rows

* + Relevant sind hier nur die Spalten ".pred" und "sales"
  + Das ganze lässt sich dann auch schön plotten, um zu sehen, wo das Modell u.U. Probleme hat

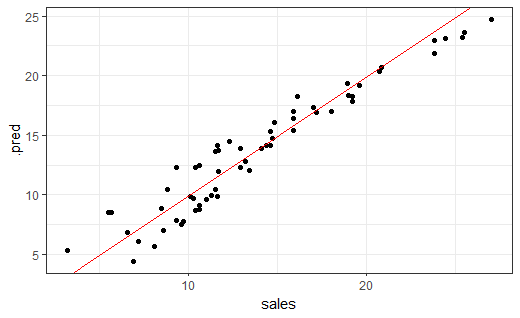
lm\_fit\_tt %>%

collect\_predictions() %>%

ggplot(aes(sales, .pred))+

geom\_point()+

geom\_abline(color="red")



* + Wie man sieht, werden höhere Sales-Werte systematisch unterschätzt (die Lösung ist ja dann in James et al. enthalten^)
* **Extraktion des Models** (Hinweis: Das wird später noch relevanter werden, wenn es um komplexere Modelle geht, bei denen es keine Parameter gibt)

lm\_fit\_tt %>%

extract\_fit\_parsnip() %>%

tidy()

term estimate std.error statistic p.value

<chr> <dbl> <dbl> <dbl> <dbl>

1 (Intercept) 14.1 0.148 95.2 2.46e-126

2 TV 3.89 0.149 26.1 7.95e- 55

3 radio 2.84 0.157 18.0 6.72e- 38

4 newspaper -0.0914 0.158 -0.580 5.63e- 1

(Hinweis: Wenn man die Koeffizienten mit denen aus dem simple-fit-Ansatz vergleicht, unterscheiden sie sich, wei diese hier auf dem Recipe basieren, in dem alle Variablen standardisiert waren. Die z-Werte („statistic“) und p-Werte sind identisch)

### Crossvalidation und Bootstrapping

#### Training des Modells

* Hier kommt jetzt anstelle der last\_fit() Funktion die fit\_resamples()-Funktion zum Tragen, weil—wie der Name schon sagt, das Modell über die ge-re-sampelten Daten gefittet werden soll.

lm\_cv\_rs <- #rs = results

fit\_resamples(

lm\_spec,

advertising\_rec,

resamples = advertising\_cv,

control = control\_resamples(save\_pred = TRUE, verbose = TRUE)

)

* Wie man sieht, müssen in die Funktion als essentielle Argumente genannt werden:
  + Die Modell-Spezifikation (lm\_spec) (Achtung: Dies muss vor dem Recipe kommen)
  + Das Recipe (advertising\_rec)
  + Das tibble mit den folds oder bootstrapped samples (advertising\_cv)
  + Die letzte Zeile ist optional und enthält die Aufforderung,

a) die predictions in den Daten zu speichern und

b) den Verlauf des fitting-prozesses anzeigen zu lassen

* Wie die pipeline zeigt, ist das Recipe Teil des gesamten resampling workflows. Dies bedeutet, dass alle preprocessing Schritte in jedem der folds durchgeführt werden. Bei den meisten (z.B. Standardisieren) ist das ziemlich unerheblich—nicht aber bei Prozesse, die eine zufällige Komponente haben und somit eine Varianz über die folds (z.B. Up-/downsampling oder Missing Data imputation).
* Wenn der Fehler *„The first argument to [fit\_resamples()] should be either a model or workflow“* kommt, dann liegt das daran, dass das model (lm\_spec) vor dem recipe kommen muss.
* Das durch fit\_resamples erzeugte Objekt enhält jetzt die Modell-Ergebnisse (Performance und Predictions für jeden der 5 folds (warum hier jetzt keine workflow-Spalte ist, weiß ich nicht)

lm\_cv\_rs

# Resampling results

# 5-fold cross-validation

# A tibble: 5 × 5

splits id .metrics .notes .predictions

<list> <chr> <list> <list> <list>

1 <split [80/20]> Fold1 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [20 × 4]>

2 <split [80/20]> Fold2 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [20 × 4]>

3 <split [80/20]> Fold3 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [20 × 4]>

4 <split [80/20]> Fold4 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [20 × 4]>

5 <split [80/20]> Fold5 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [20 × 4]>

#### Extraktion relevanter Informationen

* **Extraktion der Performance Metriken**
  + **Extraktion der Metriken für die folds getrennt** (hier filter ich mal nur den rsquare raus, was schön zeigt, dass all diese output-Tabellen tibbles sind, die man weiterverarbeiten kann)

lm\_cv\_rs %>%

unnest(.metrics) %>%

filter(.metric=="rsq") select(id, .metric, .estimate)

# A tibble: 5 × 3

id .metric .estimate

<chr> <chr> <dbl>

1 Fold1 rsq 0.935

2 Fold2 rsq 0.934

3 Fold3 rsq 0.914

4 Fold4 rsq 0.899

5 Fold5 rsq 0.858

Kann interessant sein um die Varianz zu analysieren oder zu plotten

* + **Extraktion der Mittelwerte der Metriken über die folds**

lm\_cv\_rs %>%

collect\_metrics()

.metric .estimator mean n std\_err .config

<chr> <chr> <dbl> <int> <dbl> <chr>

1 rmse standard 1.69 5 0.0828 Preprocessor1\_Model1

2 rsq standard 0.908 5 0.0142 Preprocessor1\_Model1

* + Wie man sieht, sind das Mittelwerte aus den n = 5 Läufen
  + Dies ist die Standard-Funktion. Am Mittelwert ist man primär erst mal interessiert
  + Nur noch mal als Erinnerung: Dies sind die Metriken aus dem Assessment Set der Trainingsdaten
* **Extraktion der predictions**

lm\_cv\_rs %>%

collect\_predictions()

id .pred .row sales .config

<chr> <dbl> <int> <dbl> <chr>

1 Fold1 9.28 1 9.5 Preprocessor1\_Model1

2 Fold1 21.9 2 23.2 Preprocessor1\_Model1

3 Fold1 15.0 4 13.4 Preprocessor1\_Model1

4 Fold1 6.53 5 5.6 Preprocessor1\_Model1

5 Fold1 15.2 13 12.5 Preprocessor1\_Model1

6 Fold1 7.82 14 7 Preprocessor1\_Model1

7 Fold1 8.08 31 9.5 Preprocessor1\_Model1

8 Fold1 15.7 34 13.4 Preprocessor1\_Model1

9 Fold1 14.5 36 15.5 Preprocessor1\_Model1

10 Fold1 16.3 38 16.9 Preprocessor1\_Model1

Hinweis: Dieses tibble enthält der Gesamt der Prediction aus den 5 × N=20 Assessment Sets. Wie anfangs betont, sind diese 5 sets exklusiv und enthalten nicht-überlappende Fälle. Der tibble hier aggregiert die Vorhersagen für diese Personen. Dies ist leicht anschaulich durhc die id-Spalte, die von 1-5 läuft und jeweils 20 Fälle umfasst.

* **Liste der Metriken**

<https://yardstick.tidymodels.org/reference/index.html>

## The Final Test: Anwendung des Modells auf die Testdaten

* Alle bisherigen Prozesse waren auf die Trainingsdaten bezogen. Die Metriken waren ebenfalls (nur) Metriken der Trainingsdaten (wenn auch im nicht-trainierten Teil des Assessmen Sets). Nun kommt aber die finale Frage: Kann mit dem Modell und seinen Parametern auch die Daten im Test Set vorhergesagt werden?
* Dies macht man dann schlicht über das Nutzen der last\_fit() Funktion, die bereits im TT-Only-Fall zur Anwendung kam.

Dabei wird das Modell noch einmal auf das gesamte Training Set gefittet, um das gesamte N (und nicht nur das der folds) zu nutzen. Später (beim Tuning von Hyperparametern wird dabei das „Gewinnermodell“ gewählt.

### Fitting des Modells

lm\_fit\_tt <- last\_fit(lm\_spec,

advertising\_rec,

advertising\_split)

### Extraktion der Metriken aus dem Test Set

lm\_fit\_tt %>%

collect\_metrics()

* + Im Gegensatz zum collect\_metrics im vorherigen Abschnitt sind dies die Performance-Metriken aus dem Test Set, dass niemals zuvor beim Fitten des Modells eine Rolle gespielt hatte.
  + Diese Moment dient dazu, den späteren eigentlichen Use-Case vorwegzunehmen oder zu simulieren, in dem das Model „in production“ geht (also für neue Daten genutzt wrid.)

### Extraktion der Predictions im Test Set

* Wie vorher kann es interessant (oder bei einem misfit) notwendig sein, die tatsächlichen predictions zu extrahieren.
* Dies ist im Grunde nur die Wiederholung aus Abschnitt 2.4.2.2

lm\_fit\_tt %>%

collect\_predictions()

# A tibble: 100 × 5

id .pred .row sales .config

<chr> <dbl> <int> <dbl> <chr>

1 train/test split 20.4 1 22.1 Preprocessor1\_Model1

2 train/test split 12.3 2 10.4 Preprocessor1\_Model1

3 train/test split 17.5 4 18.5 Preprocessor1\_Model1

4 train/test split 11.8 7 11.8 Preprocessor1\_Model1

5 train/test split 10.4 13 9.2 Preprocessor1\_Model1

6 train/test split 9.00 14 9.7 Preprocessor1\_Model1

* Es gibt noch einen weiteren Weg, wie man die predicted values in den **kompletten** Datensatz (mit den Prädiktoren) bekommt. Darauf wird in einem späteren Kapitel noch mal eingegangen, weil hier die enorm hilfreiche predict() Funktion benutzt wird:

lm\_spec %>%

fit(sales ~ ., data = advertising\_train) %>%

predict(., new\_data = advertising\_test) %>%

bind\_cols(advertising\_test)

* + Hier wird last\_fit() quasi in seine Einzelbestandteile zerlegt:

1. fit() fittet das Modell auf die Trainingsdaten
2. predict() nimmt dieses Modell und sagt damit die Y-Werte in den Testdaten voraus—auf Basis der dort befindlichen X-Werte
3. Abschließend werden die predicted values mittels bind\_cols mit den Variablen aus dem Test Set integriert

# A tibble: 100 × 5

**.pred** TV radio newspaper sales

<dbl> <dbl> <dbl> <dbl> <dbl>

1 20.4 230. 37.8 69.2 22.1

2 12.3 44.5 39.3 45.1 10.4

3 17.5 152. 41.3 58.5 18.5

4 11.8 57.5 32.8 23.5 11.8

5 10.4 23.8 35.1 65.9 9.2

6 9.00 97.5 7.6 7.2 9.7

7 18.4 204. 32.9 46 19

8 12.4 67.8 36.6 114 12.5

9 10.0 69.2 20.5 18.3 11.3

10 14.3 147. 23.9 19.1 14.6

## Hinweis: Parallel processing

Die Modelle brauchen je nach Komplexität sehr sehr lang. Mit parallel processing geht das schneller (in einem Video um den Faktor 2.7x)

Die einfachste Version ist, die folgende Zeile dem Modell-Training (z.B. duch fit\_resamples() ) vorzuschalten

doParallel::registerDoParallel()

Hilfreich ist, die Zeit zu messen durch

start\_time <- Sys.time()

<modell lauf>

end\_time <- Sys.time()

end\_time - start\_time

# Das Workflow Package

* Anstatt das recipe und die Model-Spezfikation getrennt anzusprechen, kann man sie in einem workflow bündeln (Julia Silge: "like lego blocks")
* Geht mit dem gleichnamigen package “workflow”. Dabei wrid das recipe und die Model-spezfikation in einem eigenen workflow-Objekt zusammengefasst. Ähnlich zu andere Objekten zuvor haben wir damit eine Art Steuuerungsobjekt
* Beim Modelfitting brauchen dann das recipe und das model nicht explizit genannt zu werden—stattdessen wird das workflow-Objekt in der jeweiligen fit-Funktion (z.B. fit\_resamples angesprochen.
* Wie im ersten Kapitel beschrieben basiert das folgende auf dem advertising-Datensatz aus James et al.‘s Regressionskapitel
* Bis zu Punkt 3.5 ist alles eine Wiederholung

## Import der Daten

library(ISLR)

advertising\_data <-

read\_csv("https://www.statlearning.com/s/Advertising.csv") %>%

select(-1)

## Split der Data

set.seed(123)

advertising\_split = initial\_split(advertising\_data, prop=.5)

advertising\_train = training(advertising\_split)

advertising\_test = testing(advertising\_split)

advertising\_cv <- vfold\_cv(advertising\_train, v=5)

## Create the Recipe

advertising\_rec <- recipe(sales~. , data = advertising\_train) %>%

step\_normalize(all\_numeric\_predictors())

## Specify the Model

lm\_spec <- linear\_reg() %>%

set\_engine("lm")

## NEU: Add to Workflow

* Nun wird das workflow-Objekt gebildet in dem
  + Die worfklow()-Funktion angewendet wrid—gefolgt von
  + add\_recipe()
  + add\_model()

advertising\_wf <- workflow() %>%

add\_recipe(advertising\_rec) %>%

add\_model(lm\_spec)

* Ergebnis ist das workflow-Objekt, dass alle Dinge enthält:

══ Workflow ════════════════════════════════════════════════════════════════

Preprocessor: Recipe #Check 1: ist vorhanden

Model: linear\_reg() #Check 2: ist vorhanden

── Preprocessor ────────────────────────────────────────────────────────────

1 Recipe Step

• step\_normalize() #Check3: Alle gewünschten operationen drin

── Model ───────────────────────────────────────────────────────────────────

Linear Regression Model Specification (regression)

Computational engine: lm

* Hinweise:
  + Hat man keine preprocessing-Schritte, kann man sich das recipe sparen und die model formula direkt mit add\_formula(y ~ .) dem workflow hinzufügen. Ich persönlich halte mich allerdings aus Gründen der Vereinfachung an den allgemeinen Prozess und würde auch hier ein Recipe erstellen.
  + Workflows erlauben auch, dass man effizient mehrere model specs vergleichen kann. Dabei wird ein generischeer workflow erstellt, der allerdings **nur das recipe** enthält (keine model spec)—während dann die beiden (oder mehrere) Modelle flexibel durch add\_model() im Fitting-Prozess hinzugefügt werden.
  + Eine Erweiterung des einfachen worfklows (das ich hier aber nur aus Vollständigkeitsgründen erwähne) ist das **workflow set** (siehe dieses [Kapitel](https://www.tmwr.org/workflow-sets.html) im Tidymodels Buch). Dies erlaubt, eine größere Anzahl von Recipe’s und Modell-Spezfikationen systematisch durchlaufen zu lassen ohne für jede denkbare Kombinationen einen eigenen workflow gefolgt von spezifischen fit-commands und Extraktionen vornehmen zu müssen. Stattdessen wird die jeweilige fitting funktion auf das set angewendet und alle Kombinationen durchprobiert und in einem einzigen komplexen tibble integriert. Ergebnis sind informationen darüber nicht nur über die den besten Modelltyp, sondern auch das sinnvollste preprocessing.

## Fit the Data

* Beim Trainieren des Modells an den Trainingsdaten muss nun nicht mehr das Recipe und Modell erwähnt werden—der Bezug auf den WF reicht:

advertising\_rs <- fit\_resamples(

advertising\_wf #workflow (Recipe + Model spec)

advertising\_cv, #Daten

control = control\_resamples(save\_pred = TRUE, verbose=TRUE))

* Einzig die Daten müssen erwähnt werden—hier die CV folds

## Evaluate the Model

advertising\_rs %>%

collect\_metrics()

Nichts Neues gegenüber dem vorherigen Kapitel.

## The Final Test

* Genau wir im vorherigen Kapitel wird hier die last\_fit()-Funktion verwendet um das Modell auf den gesamten Trainingsdaten zu fitten aber anhand der Testdaten zu evaluieren.
* Allerdings wird last\_fit() nun auf das workflow-Objekt angewendet (und nicht mehr wie vorhrer getrennt auf das recipe und model spec)

final\_advertising\_rs <- last\_fit(

advertising\_wf,

advertising\_split)

final\_advertising\_rs %>%

collect\_metrics()

# A tibble: 2 × 4

.metric .estimator .estimate .config

<chr> <chr> <dbl> <chr>

1 rmse standard 1.75 Preprocessor1\_Model1

2 rsq standard 0.875 Preprocessor1\_Model1

# Klassifikation

* Die bisherigen Themen wurden an hand der linearen Regression mit einem quantitativen Outcome durchgeführt. Dementsprechend waren die Performance-Metriken Rsquare und RMSE
* Jetzt kommt eine Klassifikation—am Beispiel der **logistischen Regression.**
* Genau wie in den ersten Kapiteln wird das am Beispiel der Daten im entsprechenden James et al. Kapitel illustriert (die default-Daten)
* Nachdem wir mit dem Rsquare und RMSE zentrale Performance-Metriken für eine Regressions-Aufgabe kennengelernt haben, soll es in diesem Kapitel um die Performence Metrik bei Klassifikatiosnaufgaben gehen. Hier ist das die
  + **Akkuratheit bzw. die Anzahl der korrekt klassifizierten Fälle**
  + die **ROC Kurve.**

Diese sind inPunkt 4.7 behandelt—bis dahin ist alles eine weitere Wiederholung.

* Der ganze R-workflow kann man [hier](https://raw.githubusercontent.com/IcarusAE/BusinessAnalytics/main/Rscripts/Default_Beispiel.r) herunterladen (wobei nicht ganz: Der code öffnet sich im Browser—kann aber von dort in R kopiert werden)

## Import der Daten

library(ISLR)

data("Default")

default <- as\_tibble(Default) #Mit kleinem Anfangbuchstaben und als tibble

# A tibble: 10,000 × 4

default student balance income

<fct> <fct> <dbl> <dbl>

1 No No 730. 44362.

2 No Yes 817. 12106.

3 No No 1074. 31767.

4 No No 529. 35704.

5 No No 786. 38463.

6 No Yes 920. 7492.

7 No No 826. 24905.

8 No Yes 809. 17600.

9 No No 1161. 37469.

10 No No 0 29275.

## Split the data

set.seed(123)

default\_split <- initial\_split(default, strata = default)

default\_train <- training(default\_split)

default\_test <- testing(default\_split)

default\_cv <- vfold\_cv(default\_train)

* Diesmal als Demonstration das strata-Argument, damit die AV im Trainings und Test Set gleich verteilt ist (wegen des sehr großen Ns nicht wirklich notwendig)
* Die default-Einstellung der volfd\_cv()-Funktion führt zu 10 folds

## Create the recipe

* Hier ist jetzt die Besonderheit, dass wir numerische Prädiktoren (balance und income) und kategoriale (student) haben. Daher werden zwei in solchen Fällen routinemäßig verwendeten steps angewandt:

default\_rec <- recipe(default ~ ., data = default\_train) %>%

step\_normalize(all\_numeric\_predictors()) %>%

step\_dummy(all\_nominal\_predictors())

Wichtig ist hierbei dass—wenn man wirkliche dummies möchte (also 0/1 kodierte Variablen)—dass step\_dummy nach der Standardisierung durch step\_normalize kommt, ansonsten wird über die 0en und 1en hinweg standardisiert und man bekommt negative und positive Abweichungen vom (unsinnigen) Mittelwert.

Wie immer sinnvoll: Ein check des presprocessings:

default\_rec %>%

prep() %>%

juice()

A tibble: 7,500 × 4

balance income default student\_Yes

<dbl> <dbl> <fct> <int>

1 -0.627 0.153 No 0

2 -0.0923 0.360 No 0

3 -0.00916 -0.657 No 0

4 -0.0443 -1.21 No 1

5 0.691 0.286 No 0

6 -1.73 -0.329 No 0

7 -1.73 -0.885 No 1

8 -1.24 -0.406 No 0

9 -0.465 0.850 No 0

10 0.590 -0.739 No 0

## Specify the model

logreg\_spec <- logistic\_reg() %>%

set\_engine("glm")

## Add to workflow

default\_wf <- workflow() %>%

add\_recipe(default\_rec) %>%

add\_model(logreg\_spec)

## Fit the Data

default\_rs <- fit\_resamples(

default\_wf,

default\_cv,

control = control\_resamples(save\_pred = TRUE, verbose=TRUE))

## NEU: Evaluieren der Klassifikationsgüte

* Im Fall der linearen Regression wurden mittels collect\_metrics der RMSE und R-Quadrat ausgegeben. Wendet man die Funktion auf ein Model mit dem Modus „classification“, werden automatisch
  + Der Accuracy-Wert (Prozent der korrekt klassifizierten Fälle)
  + Und der ROC (receiver-Operator-Characteristic ausgeben.

default\_rs %>%

collect\_metrics()

A tibble: 2 × 6

.metric .estimator mean n std\_err .config

<chr> <chr> <dbl> <int> <dbl> <chr>

1 accuracy binary 0.974 10 0.00220 Preprocessor1\_Model1

2 roc\_auc binary 0.953 10 0.00438 Preprocessor1\_Model1

* Besonders der accuacy-Wert täuscht hier schnell—v.a. dann, wenn das target ungleich verteilt war (was hier der Fall ist, da es von den 10,000 Fällen nur 333 Personen gab, die ihre Kreditkarte überzogen haben).
* In der Regel will man daher eine sog. **Confusion matrix** erstellen um zu analysieren ob man eher false positives oder false negatives erzeugt hat. Ein weiteres Evaluationsmaß ist die ROC-Kurve
* Beide bassieren auf den predicted classes

### Extraktion der Klassifikationen

* Sowohl die Confusion Matrix als auch die ROC-Curve basiert auf einem tibble, in dem die predicted classes mit den tatsächlichen integriert wurden. Wie das geht, wurde in Kapitel 2.4.3.2 gezeigt—nämlich über die Extraktion der predictions, die in jedem der folds auf Basis des Trainings des Modells im Analysis Set für die Daten im Assessment Set gebildet wurden:

default\_rs %>%

collect\_predictions()

# A tibble: 7,500 × 7

id .pred\_No .pred\_Yes .row .pred\_class default .config

<chr> <dbl> <dbl> <int> <fct> <fct> <chr>

1 Fold01 0.998 0.00203 6 No No Preprocessor1\_Model1

2 Fold01 1.00 0.0000194 16 No No Preprocessor1\_Model1

3 Fold01 1.00 0.000205 17 No No Preprocessor1\_Model1

4 Fold01 1.00 0.0000727 19 No No Preprocessor1\_Model1

5 Fold01 0.998 0.00241 22 No No Preprocessor1\_Model1

6 Fold01 0.998 0.00178 39 No No Preprocessor1\_Model1

7 Fold01 1.00 0.000345 71 No No Preprocessor1\_Model1

8 Fold01 0.985 0.0150 77 No No Preprocessor1\_Model1

9 Fold01 0.986 0.0141 116 No No Preprocessor1\_Model1

10 Fold01 0.998 0.00154 122 No No Preprocessor1\_Model1

* Das N von 7,500 ist die Summe der einzelnen **Assessments sets** für die Predictions gemacht worden waren (und in der Summe daher für den gesamten Trainingsdatensatz)

### Confusion matrix

default\_rs %>%

collect\_predictions() %>%

conf\_mat(default, .pred\_class)

Truth

Prediction No Yes

No 7222 164

Yes 33 81

* Wie man sieht, ist die sehr hohe Akkuratheit der Gesamt-Klassifikation im Großen und Ganzen durch die sehr gute Vorhersage der „No default“-Fälle geschehen, während die defaults immerhin mit 2/3 vorhergesgt werden. Im nächsten Kapitel wird es ein deutlich schlechteres Ergebnis geben. Bei so einem großen Datensatz wie diesem könnte man daher ein downsampling der no-default-Fälle erwägen.
* Ohne unnötig verwirren zu wollen, sieht man für die Confusion matrix oft diesen Befehl hier:

default\_rs %>%

unnest(.predictions) %>%

conf\_mat(default, .pred\_class)

Der ist aber im Grunde idenisch, weil collect\_predictions nichts Weiteres ist, als aus dem results-tibble die Spalte mit den predictions (".predictions") a) zu selekieren und b) zu „un-nesten“)

* Hinweis: Für Modelle, die zwar resampling (CV oder boostrapping) verwenden, aber kein HP tuning (siehe Kapitel 5) gibt es noch die etwas Version conf\_mat\_resampled(), die die Mittelwerte der Anzahl der Klassifikationen über die folds liefert. M.E. hat das keinen Mehrwert.

default\_rs %>%

conf\_mat\_resampled()%>%

spread(Truth, Freq)

### Plot der ROC curve

default\_rs %>%

collect\_predictions() %>%

group\_by(id) %>%

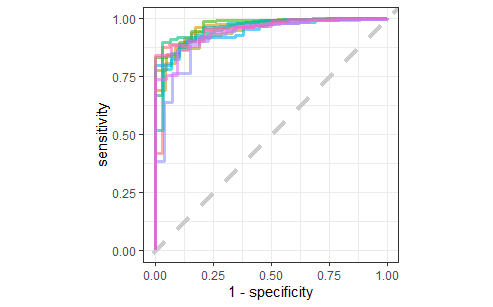
roc\_curve(default, .pred\_No) %>%

ggplot(aes(1- specificity, sensitivity, color=id))+

geom\_abline(lty = 2, color="gray80", size=1.5)+

geom\_path(show.legend = FALSE, alpha = 0.6, size = 1.2)+

coord\_equal()



* **Interpretation**
  + Dashed line: Guessing
  + Ergo sind Modelle in der linken oberen Hälfte besser als guessing.
  + Die „area under the curve“ ist ein Maß für die Nützlichkeit des Tests/Models
  + Zentral sind die beiden Begriffe **sensivity** und **specificity**
  + **Sensititivity** ist der Prozentsatz der Kranken, die entdeckt werden: true positives / alle Kranken bzw. true positives / (true positives + false negatives)
  + **Specificity** ist der Prozentsatz der Gesunden, für die nicht Alarm geschlagen wird: true negatives / alle Gesunden oder true negatives / (true negatives + false positives)
  + In der Regel berechnet man die "false positive rate" durch "1-specificity"
  + Beide sind aber nicht starr, sondern können mit einem **Threshold** variieren, den man für die Entscheidung jemand als „default“ zu kennzeichnen wählen muss. Im James etal. Kapitel über Klassifikation war dieser threshold p = .5. Je höher er ist, um so sicherer wird man false positives (=0) vermeiden—man übersieht aber immer mehr true positives (die dann false negatives werden)
  + Damit zeigt die ROC, wie stark bei einer (erwünschten) Steigerung der sensitivität auch die false positive rate steigt. Daher ist eine Kurve erwünscht, die stark entlang der links-oberen Eckeverläuft, weil dies bedeutet, dass die Sensitivity dynamischer ansteigt also die Fehlerrate

### Weitere Metriken

* Es gibt noch weitere Metriken, die besser als accuracy sind (basieren alle auf der Confusion matrix)
* **Precision:** Wie genau sagt man die True Positives (TP)—z.B. die tatsächlich Kranken--voraus → Anteil der TP an den als positiv klassifizierten Fällen
  + **Recall:** Anzahl der TP an allen Kranken, zu denen auch die FN gehören (also diejenigen die krank sind aber als nicht-krank klassifiziert wurden
  + **F1 score:** Precision und Recall haben eine inverse Beziehung: Wenn die Precision steigt, sinkt der Recall und umgekehrt. Der F1-Score betrifft das optimale Verhältnis beider und ist im Idealfall 1.0
* Welche Metriken man nimmt, hängt vom konkreten Fall ab und, ob man eher interessiert ist die FP oder die FN zu reduzieren. Im Fall „Diagnose einer Krankheit“ sind es die FN (lieber eine gesunde Person als krank diagnostiziert als eine kranke Person zu übersehen). D.h.
* Wenn die Priorität die Reduktino der FN sind, dann ist Recall das Maß
  + Wenn die Priorität der Reduktion der Fehlalarme (FP) ist, dann ist es Precision
  + Will man eigentlich beides und strebt einen Kompromiss an, dann ist es der F1-score
* Mehr Infos unter
  + <https://medium.com/data-reply-it-datatech/imbalanced-classification-in-fraud-detection-8f63474ff8c7>
  + <https://medium.com/mlearning-ai/evaluating-classification-models-simplified-b5929146504e>

## The Final Test

* Wie auch in den vorherigen Kapiteln wird der gesamte workflow beendet mit einem
  + Training des Modells über alle Trainingsdaten und
  + der Evaluation an den Testdaten

### Training des Modells

default\_final <- default\_wf %>%

last\_fit(default\_split)

### Extraktion der Performance Metriken

default\_final %>% collect\_metrics()

# A tibble: 2 x 4

.metric .estimator .estimate .config

<chr> <chr> <dbl> <chr>

1 accuracy binary 0.971 Preprocessor1\_Model1

2 roc\_auc binary 0.940 Preprocessor1\_Model1

### Test confusion matrix

default\_final %>%

collect\_predictions() %>%

conf\_mat(default, .pred\_class)

Truth

Prediction No Yes

No 2405 65

Yes 7 23

### ROC curve

default\_final %>%

collect\_predictions() %>%

roc\_curve(default, .pred\_No) %>%

ggplot(aes(x = 1 - specificity, y = sensitivity)) +

geom\_line(size = 1.5) +

geom\_abline(

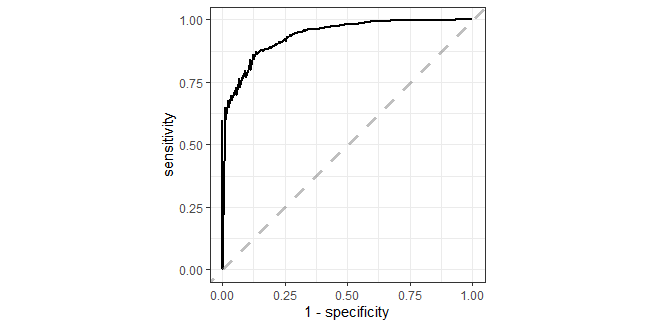
lty = 2, alpha = 0.5,

color = "gray50",

size = 1.2 ) +

coord\_equal()+

theme\_bw()



# Hyperparameter (HP) tuning

* Grundlage: [Kapitel](https://www.tidymodels.org/start/tuning/) 4 über Tune model parameters im Tidymodels-Buch

## Background

* Hyperparameter (HP) sind Parameter von Modelltypen, die man variieren kann, um die Vorhersagbarkeit zu erhöhen. [Hier](https://towardsdatascience.com/comparative-study-on-classic-machine-learning-algorithms-24f9ff6ab222) ist ein netter Überblick über die gängigen Typen, ihre Vor- und Nachteile und welche HP sie haben.
* Die HP bekommt man über ?<model> z.B. ?nearest\_neighbor
* HP haben default Werte in den models (model spec). Sie können aber auch auf einen bestimmten Wert fixiert werden oder es kann ein range durchgetestet werden. Dies ist das **Tuning**
* **Beispiele** (in Klammern die tidymodels-Argumente der model specifications und der default-Wert). Wie man sieht ist das noch in progress (TDB)

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Default** | **Erläuterung** |
| **KNN** |  |  |
| neighbors | 5 |  |
| weight\_func |  | Sollen die Nachbarn nach ihrer Nähe gleich gewichtet werden (→ Mittelwert) oder sollen nahe Nachbarn stärker gewichtet werden |
| dist\_power |  | Welches Distanzmaß --v.a. Euclidean vs. Manhatten (Minkowski distance order) |
| **Naïve Bayes** |  |  |
| smoothness |  | Smootness addiert Noise, um Wahrscheinlichkeiten von 0 zu vermeiden. |
| Laplace |  | Laplace correction |
| **LASSO** |  |  |
| penalty |  | ; Ausmaß in dem durch einen additiven Zusatz von || ein bias induziert wird |
| mixture | 1 | ; Mixing Komponente, die zwischen 0 (Ridge regression) und 1 (reines LASSO model) variiert. |
| **Decision tree** |  |  |
| cost\_complexity |  | Komplexitätsparameter Cp. Betrifft das Mindesmaß an Verbesserung durch einen Knoten-Split (siehe James et al., S. 331) |
| tree\_depth |  | Tiefe der Bäume (=Flexibiltät/overfitting). Definiert als „Länge des längsten Pfads vom root node zu einem terminal node“. |
| min\_n |  | Mindestanzahl der Fälle in einem Knoten, um ihn weiter zu splitten |
| **Random forest** |  |  |
| mtry | √p | Maximale Anzahl *m* von features, die per random sample aus dem Gesamtset *p* gezogen werden und die damit bei einem split berücksichtigt werden können. Der default (m = √p) hier ist in James et al. quasi die Definition von RF |
| trees |  | Anzahl der zu trees im Ensemble. Erhöht Stabilität. Regel ~ 10\*p. Tuning macht hier nicht so viel Sinn. Julia Silge nimmt meist 1000 („make sure that you have enough) |
| min\_n |  | Mindestanzahl der Fälle in einem Knoten, um ihn weiter zu splitten. Relevantester Aspekt für Overfitting |
| **XGBoost** |  |  |
| mtry |  | Sieh Random forest. Anzahl der bei jedem Tree zufällig gezogenen subsets an Prädiktoren, die zum Splitten herangezogen werden |
| trees |  | Anzahl der Trees entlang der gesamten Sequenz. |
| min\_n |  | Mindestanzahl der Fälle in einem Knoten, um ihn weiter zu splitten. Relevantester Aspekt für Overfitting |
| tree\_depth |  | Länge des längsten Pfades vom root node zu einem terminal node. In einem XGBoost model ist das meist nur 1 („stumps“) oder 2. D.h. es werden zwar mehrere Trees hintereinander geschaltet, aber jeder von ihnen ist sehr flach |
| learn\_rate |  | Shrinkage parameter . Geht von 0-1; typische Werte sind .01 oder .001 (James et al.). Speed vs. accuracy: l bestimmt die Schritte, über die das Modell seine Gewichte anpasst. Kleine Werte führen zu akkurateren Trees; können aber eine größere Anzahl von Iterationen/Trees nötig machen, was die Dauer erhöht. |
| loss\_reduction |  | „A number for the reduction in the loss function required to split further“ (nehme an, das betrifft Klassifikation, weil es dort wohl eine loss function gibt |
| sample\_size |  |  |
| **SVM (linear)** |  |  |
| cost | 1 | Budget für die Anzahl der tolerierten Punkte innerhalb der margin („costs of constraint violation“)   * + Tuning-Range : 0.0000000001 – 10,000,000,000 (der auch 1e-10 bis 1e10) |
|  |  |  |
| **SVM (nonlinear)** |  |  |
| cost |  | Budget für die Anzahl der tolerierten Punkte innerhalb der margin („costs of constraint violation“) |
| rbf\_sigma |  |  |
|  |  |  |

* In der Regel weiß man aber die optimalen Werte nicht. Die Lösung ist tuning. Dabei wird ein range von Werten durchprobiert und das beste Modell (im Rahmen der Performance Metriken im Training Set für den finalen Test mit den Test-Daten ausgewählt. Zusätzlich plottet man den Verlauf der prediction entlang der HP-Werte (siehe die vielen Beispiele in James et al.)
* Anstatt in der model specification nichts zu benennen (=Nutzen des default HP) oder einen bestimmen HP zu bestimmen, muss klargemacht werden, dass der HP getuned werden muss. Hier noch mal die Optionen am Beispiel KNN:
  + Default-Wert: knn\_spec <- nearest\_neighbors()
  + Fester Wert: knn\_spec <- nearest\_neighbors(neighbors = 10 )
  + HP Tuning: knn\_spec <- nearest\_neighbors(neighbors = tune() )
* Tuning kann auch auf bestimmte Prozesse im Recipe angewendet werden—z.B. wenn man eine PCA vorschaltet (als Teil des Precprocessing) und die Anzahl der PCs tuned. Damit wird der Preprocessing-Teil selbst Teil des ganzen—empirisch bestimmten—Auswahl-Prozesses.

## NEU: Tuning-Grid generieren

* Ein Grid ist ein möglicher Wertebereich für die HP (siehe entsprechendes Kapitel über grid-search im TM Buch: <https://www.tmwr.org/grid-search.html> )
* Dazu gibt es mehrere Funktionen. Eine davon ist grid\_regular
* Die simplelste Version ist, einfach nur die HPs zu nennen; dadurch wird ein plausibler Bereich angelegt, der spezifisch für den jeweiligen HP ist. Das levels-Argument steuert, wie viele Werte man haben möchte.
* Damit würde ich empfehlen, immer zu beginnen. In der Regel wird ein für den HP sinnvoller Bereich gewählt. Ein späterer Plot zeigt, ob es am Anfang oder am Ende einen Trend gibt, der es möglich erscheinen lässt, dass der Optimum außerhalb des Bereichs liegt. Wenn das der Fall ist, kann man über die Wahl des Bereichs (s.u.) in einen alternativen „rein-zoomen“.

Beispiel

grid\_regular(

trees(),

min\_n(),

levels=5 )

trees min\_n

<int> <int>

1 1 2

2 500 2

3 1000 2

4 1500 2

5 2000 2

6 1 11

7 500 11

8 1000 11

9 1500 11

10 2000 11

**Achtung:** Wenn man ein tree-model rechnet und den **mtry** HP tunen möchte—der hat hier eine unangenehme Sonderrolle: Da sein Wertebereich von 1-p geht und die grid-funktion nicht weiß, was p ist, muss man entweder einen Bereich wählen oder das Argument finalize(mtry(), train\_data) in die o.g. grid-Funktion einbauen (wobei train\_data die Traingsdaten sind). Das ist einfach um der Funktion eine Info über p zu geben. Macht man das nicht bekommt man einen Fehler (der das o.G. ja sagt)

Error in `grid\_regular()`:

! These arguments contains unknowns: `mtry`. See the `finalize()` function.

* Wenn man einen bestimmten Bereich **manuell bestimmen** möchte, geht das über das range-Argument

grid\_regular(

trees(range = c(1000, 5000)),

min\_n(range = c(2, 100)),

levels=5)

Hier wird bei den treees 5 tree-Anzahlen zwischen 1000 und 5000 angelegt (also in 1000er Schritten)

trees min\_n

<int> <int>

1 1000 2

2 2000 2

3 3000 2

4 4000 2

5 5000 2

6 1000 26

7 2000 26

8 3000 26

9 4000 26

10 5000 26

Wie man sieht ist die erste Version schneller/einfacher. Die zweite bietet mehr Kontrolle (d.h. eventuell erst mal testen und einen zweiten Lauf mit sinnvolleren Werten machen)

* Achtung: Gerade die Kombination zweier HP-Parameter (über Kombinationen von Werte-Bereichen) und Cross-Validation erzeugt schnell eine Masse von Modellen, was extrem lang dauern kann. Daher erst mal ohne HP-tuning anfangen und dann mal mit einem kleinen Bereich anfangen. Eine Alternative ist die Funktion grid\_latin\_hypercube() bei der Bildung des Skripts, die aus dem multidimensionalen HP Raum sinnvolle Kombinationen zusammenstellt. Eine Illustration sieht man in dem Volley-ball-[Video](https://www.youtube.com/watch?v=hpudxAmxHSM&ab_channel=JuliaSilge), in dem Julia dies bei 29:30 zeigt. Hier gibt’s den code des Beispiels als R-Skript (TBD: Verlinken mit meiner github-Version)
* Hinweis: Alternativ zu grid\_regular wirden auch die expand.grid()-Funktion verwendet. Diese ist etwas flexibler. Beispiel: expand.grid(neighbors = seq(1,10, by=1))

## NEU: Train the Model mit tune\_grid()

* Bislang war fit\_resamples() die zentrale Funktion, die das Modell über die k folds der Trainingsdaten hat laufen lassen. Im Fall von HP tuning kommt jetzt **stattdessen** tune\_grid() zum Einsatz.

model\_rs <- tune\_grid(

knn\_wf, #workflow Objekt (hier mal mit einem KNN model)

grid = model\_grid, #angelegtes Grid (Wertebereich der HP)

resamples= mtcars\_cv #Daten

)

* Wenn man nun mittels collect\_metrics() die Metriken abfragt, bekommt man für jeden der getuned HPs die Metrik (und kann diese auch schön plottten). Daraus wählt man das Gewinnermodell.

# Metriken komfortabel plotten

model\_rs %>% autoplot(

# Metriken extrahieren

model\_rs %>% collect\_metrics()

# Die besten 5 Modelle anzeigen

model\_rs %>% show\_best(n = 5)

## NEU: Finalisieren (updaten) des workflows für das finale Modell

* Der workflow enthält bislang die Information, dass ein oder mehrere HP getuned werden sollen. Da das Rennen gelaufen ist und ein Sieger feststeht, muss—bevor das Modell final gefittet werden kann, dieser tune()-Befehlt durch den tatsächlcih besten HP-Wert ersetzt werden. Das mach man nicht manuell sondern mittels der select\_best()-Funktion

best\_knn <- model\_rs %>%

select\_best(metric="accuracy")

* Mit diesem wird der Workflow nun finalisiert

final\_lm\_wf <- finalize\_workflow(

knn\_wf, #Alten Workflow wählen (in dem noch "tune()" steht)

best\_knn) #Objekt mit dem besten HP angeben

* Dieses Objekt wird dann mittels last\_fit() final am Testdatensatz evaluiert.

knn\_testfit <- final\_jsat\_wf %>%

last\_fit(jsat\_split)

knn\_testfit %>%

collect\_metrics()

## Vollständiges Beispiel mit KNN

Beispiel soll hier ein KNN mit einem Datensatz sein, der mögliche Prädiktoren von Arbeitszufriedenheit enthält.

Daten einlesen mit

jsat\_data<- read\_rds("https://gmudatamining.com/data/employee\_data.rds")

* Die Daten stammen aus einem data science Lehrprojekt. Variablenbeschreibungen finden sich [hier](https://www.gmudatamining.com/data-analysis-project.html). Das zentrale outcome job satisfaction ist ein factor mit vier leveln (von low bis very high). Ergo ist die Aufgabe eine Klassifikationsaufgabe.
* Zu den Vor-Checks gehört hier
  + Check der Formate (dbl vs. fct), z.B. mittels str(jsat\_data)
  + Check der Balance der AV mittels count(job\_satisfaction). Hier könnte man tatsächlich ein upsampling erwägen

jsat\_data %>% count(job\_satisfaction)

job\_satisfaction n

<fct> <int>

1 Low 289

2 Medium 280

3 High 442

4 Very High 459

### Split the Data

* **Splitten**

set.seed(123)

jsat\_split = initial\_split(jsat\_data, prop=.7)

jsat\_train = training(jsat\_split)

jsat\_test = testing(jsat\_split)

* **CV-Folds bilden**

set.seed(345)

jsat\_cv = vfold\_cv(jsat\_train, v = 10)

### Create the Recipe

* **Recipe generieren**

jsat\_rec = recipe(job\_satisfaction ~ ., data= jsat\_train) %>%

step\_normalize(all\_numeric\_predictors()) %>%

step\_dummy(all\_nominal\_predictors()) %>%

themis::step\_smote(job\_satisfaction)

* + step\_dummy() kommt hier nach normalize, weil sonst über die dummies normalisiert würde (was ihnen einen mix aus negativen und positiven Werten verschafft. Dies ändert zwar nichts an deren Funktionalität, wiederspricht aber dem dummy Konzept
  + Schließlich wird mit step\_smote() ein upsampling vorgenommen, da es doppelt so viele Zufriedene wie Unzufriedene gibt (smote ist die multiclass-Version von step\_rose)
* **Testweise mal anschauen** (mittels prep und juice)

jsat\_rec %>%

prep() %>%

juice()

### NEU: Specify the model

* Hier kommt jetzt die Spezfikation, dass HP getuned werden sollen

knn\_spec <- nearest\_neighbor(

neighbors = tune()

) %>%

set\_mode("classification") %>% #wichtig, da KNN beides kann

set\_engine("kknn")

* Anstelle neighbors = 10 etc. wird hier mit tune() klargemacht, dass ein Bereich getestet werden soll.
* Ruft man knn\_spec auf bekommt man die Übersicht

K-Nearest Neighbor Model Specification (classification)

Main Arguments:

neighbors = tune()

Computational engine: kknn

### Add to Workflow

* Nun wird das recipe und die model spec dem worflow hinzugeüfhrt
* Teaser für das nächste Kapitel: Wenn man mehrere Modelle vergleichen will, kann an die model spec hier weglassen und die model-specs der zu-vergleichenden Modelle später mittels pipe operators flexible ansprechen. Alternativ müsste man model #1 entfernen (remove\_model()) und das andere hinzufügen.

knn\_wf <- workflow()%>%

add\_recipe(jsat\_rec) %>%

add\_model(knn\_spec)

### NEU: Generate Tuning-Grid

* Nun generiert man ein sog. „grid“ von HP-Kandidaten. Da in dem Beispiel hier nur der K-Wert getuned werden soll ist das eine simple Liste.
* Hinweis: Als eine Daumenregel für die adequate K-Anzahl dient sqrt(N). In dem Fall hier ist das N = 9,000 = 94.

Ich hatte im ersten Lauf eine 10er Sequenz von 1 bis 100, musste die aber schnell anpassen, weil die Akkuratheitswerte immer mehr anstiegen (um es vorweg zu nehmen: Kein gutes Zeichen)

knn\_exp\_grid = expand.grid(neighbors = seq(1,400, by=20))

### NEU: Fit the Data and Tune HPs

knn\_rs <- tune\_grid(

knn\_wf, # workflow mit recipe und preproc-Anweisungen

resamples = jsat\_cv, # Daten (10 CV folds)

grid = knn\_exp\_grid, # tuning grid

control = control\_resamples(save\_pred = TRUE, verbose=TRUE)

)

### NEU: Evaluate the Model in the Assessment Set

* Jetzt kommt der große Unterschied zu den bisherigen non-tuning-Beispielen: Da das Modell ja nicht nur über 9 der 10 folds gelaufen und am 10. evaluiert wurde, sondern dies alles auch noch Dutzende Male für jeden der 20 HP-Kandidaten (von K=1 bis K = 381), bekommt man jetzt die über die folds gemittelten Metriken für jeden der HP Kandiaten.
* Da das knn\_res-Objekt für mehrere der folgenden Prozesse zentral ist, hier eine Erläuterung seiner Inhalte:

knn\_rs %>% select(-.notes) #Ich schließe mal die irrelante .notes-Spalte aus

# A tibble: 10 × 4

splits id .metrics .predictions

<list> <chr> <list> <list>

1 <split [926/103]> Fold01 <tibble [40 × 5]> <tibble [2,060 × 9]>

2 <split [926/103]> Fold02 <tibble [40 × 5]> <tibble [2,060 × 9]>

3 <split [926/103]> Fold03 <tibble [40 × 5]> <tibble [2,060 × 9]>

4 <split [926/103]> Fold04 <tibble [40 × 5]> <tibble [2,060 × 9]>

5 <split [926/103]> Fold05 <tibble [40 × 5]> <tibble [2,060 × 9]>

6 <split [926/103]> Fold06 <tibble [40 × 5]> <tibble [2,060 × 9]>

7 <split [926/103]> Fold07 <tibble [40 × 5]> <tibble [2,060 × 9]>

8 <split [926/103]> Fold08 <tibble [40 × 5]> <tibble [2,060 × 9]>

9 <split [926/103]> Fold09 <tibble [40 × 5]> <tibble [2,060 × 9]>

10 <split [927/102]> Fold10 <tibble [40 × 5]> <tibble [2,040 × 9]>

* + Jede Zeile ist eines der 10 folds—in jedem wurden 20 K-Werte getestet
  + In jedem fold wurden 2 Metriken berechnet (accuracy und roc\_auc)—daher 2x20 = 40 (genestete) Zeilen in der .metrics-Spalte.
  + Die .prediction-Spalte enthält schließlich für jeden der 103 Personen im Assesments Set (siehe Spalte „splits“) ihren tatsächlichen und predicted job satisfaction Wert—und das für jeden der 20 HP (→ 20 HP x 103 = 2,060). Diese Teil ist die Grundlage der confusion matrix unten.
* **Auflistung der Metriken**

knn\_rs %>% collect\_metrics() %>%

select(neighbors, .metric, mean) #ich wähl nur die relvanten Variablen

neighbors .metric mean

<dbl> <chr> <dbl>

1 accuracy 0.249

1 roc\_auc 0.489

21 accuracy 0.234

21 roc\_auc 0.477

41 accuracy 0.229

41 roc\_auc 0.480

61 accuracy 0.230

61 roc\_auc 0.487

81 accuracy 0.223

81 roc\_auc 0.490

* + Für jedes K gibt es accuracy und roc\_auc-Mittelwerte (über die Assessment-Sektionen der 10 folds).
  + Da multi-kategoriale Klassifikationen ist roc\_auc nicht sinnvoll ist (Quelle: [Video](https://www.youtube.com/watch?app=desktop&v=864iNqFNzc0&ab_channel=AndrewCouch) von Andrew Couch); daher kann man die ignorieren / rausfiltern
* Die Entscheidung kann man mittels plot am ehesten treffen

Die einfachste Version ist autoplot()

autoplot(knn\_rs)

Oder auch aufwändiger

knn\_rs %>%

collect\_metrics() %>%

filter(.metric == "accuracy") %>%

ggplot(aes(neighbors, mean)) +

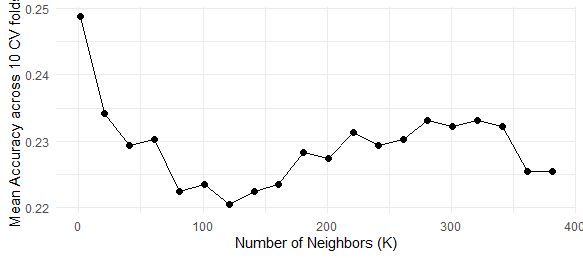
geom\_line() +

geom\_point(size = 2)+

labs(x = "Number of Neighbors (K)", y = "Mean Accuracy across 10 CV

folds")+

theme\_minimal()



* **Was ist das beste Modell?**

knn\_rs %>%

select\_best(metric="accuracy")

→ K = 1

* Hinweis: Eine ROC Kurve gibt es bei multi-class-Modellen nicht
* **Confusion matrix** (mit den Trainingsdaten)

Für die confusion matrix wählen wir aus dem gesamten Set der getunten Model (insgesamt 20 Modelle gefittet über

knn\_rs %>%

unnest(.predictions) %>%

filter(neighbors=="1") %>% #Auswahl des besten Modells

conf\_mat(job\_satisfaction, .pred\_class)

Truth

Prediction Low Medium High Very High

Low 36 42 63 69

Medium 41 30 70 53

High 59 66 84 97

Very High 68 48 97 106

* Die Matrix zeigt:
  + Generell sehr schlechte Vorhersage, da das Verhältnis der Werte auf der Diagonale zu den Spaltensummen (Missklassifikationen) sehr schlecht ist. Das ist am dramatischsten, wenn Unzufriedene als hochzufrieden klassifiziert wurden oder vice versa. Dies zeigt einfach, dass das Modell nicht prädiktiv ist
  + Daher auch der K=1-Wert als bestes Modell, weil das Modell alles tut (Maximierung der Flexibilität) um die Prädiktion zu erhöhen.
  + Gegenüber einem (nicht hier gezeigten) Modell ohne Upsampling ist der Anteil der korrekt klassifizierten Unzufriedenen höher als ohne, was die Wichtigkeit von Upsampling (oder downsampling) zeigt.
* **Training confusion matrix bei CV + HP tuning:** Anstatt die Auswahl des besten Modells wie oben über filter = zu regeln, gibt es in die conf\_mat\_resampled() Funktion und dort das Argument „parameters=“. Dort kann man einfach das Objekt aus der select\_best() Funktion nehmen! Das ist gerade dann sinnvoll, wenn das beste Modell eine Kombination aus mehreren HPs war. Hier ein Beispiel aus einem penalized regression model

log\_rs %>%

conf\_mat\_resampled(parameters = best\_auc)%>%

spread(Truth, Freq)

* + Hier war log\_rs der gefittete workflow:

# Tuning results

# 5-fold cross-validation

# A tibble: 5 × 5

splits id .metrics .notes .predictions

<list> <chr> <list> <list> <list>

1 <split [935/234]> Fold1 <tibble [100 × 5]> <tibble [0 × 3]> <tibble [11,700 × 7]>

2 <split [935/234]> Fold2 <tibble [100 × 5]> <tibble [0 × 3]> <tibble [11,700 × 7]>

3 <split [935/234]> Fold3 <tibble [100 × 5]> <tibble [0 × 3]> <tibble [11,700 × 7]>

4 <split [935/234]> Fold4 <tibble [100 × 5]> <tibble [0 × 3]> <tibble [11,700 × 7]>

5 <split [936/233]> Fold5 <tibble [100 × 5]> <tibble [0 × 3]> <tibble [11,650 × 7]>

* + best\_auc war ein bestimmter penality-Wert. Der war vorher mit select\_best generiert/ausgewählt und kann hier angesprochen werden.
  + Ergebnis eine confusion matrix, die den Mittlewetrt der 5 folds wiedergibt (daher die Dezimalzahlen.

# A tibble: 2 × 3

Prediction `0` `1`

<fct> <dbl> <dbl>

1 0 146. 6

2 1 71.6 9.8

### NEU: Finalize workflow

* Das bisherige workflow-Objekt jsat\_wf enthielt ja noch die Info, dass K getuned werden soll. Diese info wird jetzt ersetzt durch das best model.
* Das beste Modell (K = 1) kann man sich mit der bereits oben benutzten select\_best()-Funktion ausgeben lassen:

best\_knn <- knn\_rs %>%

select\_best(metric="accuracy")

neighbors .config

<dbl> <chr>

1 1 Preprocessor1\_Model11

* **Aktualisieren des Workflows**

final\_knn\_wf <- finalize\_workflow(

jsat\_wf,

best\_knn

)

* **Vergleich**

**jsat\_wf**

══ Workflow ═══════════════════════

Preprocessor: Recipe

Model: nearest\_neighbor()

── Preprocessor ───────────────────

2 Recipe Steps

• step\_normalize()

• step\_dummy()

── Model ──────────────────────────

K-Nearest Neighbor Model Specification (classification)

**Main Arguments:**

**neighbors = tune()**

Computational engine: kknn

**final\_jsat\_wf**

══ Workflow ═══════════════════════

Preprocessor: Recipe

Model: nearest\_neighbor()

── Preprocessor ───────────────────

2 Recipe Steps

• step\_normalize()

• step\_dummy()

── Model ──────────────────────────

K-Nearest Neighbor Model Specification (classification)

**Main Arguments:**

**neighbors = 1**

Computational engine: kknn

### Fit the Best Model on the Test Set

* Nun wird der aktualisierte workflow (mit dem besten Modell) über last\_fit() über das split-Objekt laufen lassen

knn\_testfit <- final\_jsat\_wf %>%

last\_fit(jsat\_split)

### Evaluate Model on Test Set

knn\_testfit %>%

collect\_metrics()

.metric .estimator .estimate .config

<chr> <chr> <dbl> <chr>

1 accuracy multiclass 0.270 Preprocessor1\_Model1

2 roc\_auc hand\_till 0.499 Preprocessor1\_Model1

→ Das Modell ist stabil....schlecht

* Confusion matrix test set

knn\_testfit %>%

unnest(.predictions) %>%

conf\_mat(job\_satisfaction, .pred\_class)

Truth

Prediction Low Medium High Very High

Low 12 14 18 23

Medium 16 12 21 28

High 26 31 47 35

Very High 31 37 42 48

## Hinweis zum housekeeping und Bennennung der Objekte

* Es ist sehr empfehlenswert, bei der Benennung der Objekte eine bestimmte Logik anzuwenden. Auf keinen Fall generische Begriffe verwenden, die über ggfls. Verschiedene Modelle zu ähnlich sind (model1, model2) oder sogar dieselben Begriffe zu wiederholen
* Meine persönliche Strategie, die Datenobjekte nach dem tibble zu benennen und die Modell-Objekte nach der model spec, z.B.
  + **mtcars\_split:** Split-Objekt
  + **mtcars\_train:** Trainingsset
  + **mtcars\_test:** Testset
  + **mtcars\_cv / mtcars\_boot**
  + **mtcars**\_**rec:** Recipe
  + **tree***\_***spec:** Model spec
  + **tree\_wf:** workflow
  + **tree\_grid:** HP grid
  + **tree\_rs:** Ergebnis tibble mit den getunten Ergebnissen aus dem Trainingset
  + **final\_tree\_wf:** Finalisierter workflow
  + **final\_tree\_rs:** „Gefitteter workflow“ mit predictions und metrics
* Allerdings hängt dies letztlich vom Fall ab. Es kann durchaus sein, dass es mehrere Versionen von Datensätzen, Recipes und Modelltypen gibt. Dann nutze ich Excel um alle Versionen vollständig durchzudeklinieren. Die Objektnamen können dann schnell auch mal länger werden z.B.

logR\_trm\_winner\_spec: Anwendung einer Logistischen Regression (logR) auf ein data file mit einer getrimmten AV (trm), die den Namen „winner“ hatte.

# Variable Importance

* Jenseits des prediction / classification errors will man ja meist auch wissen, welche Prädiktoren für die prediction / classification wichtig waren. Dieser Aspekt ist in den letzten Jahren insbesondere im Rahmen des Konzepts „explainable AI“ immer wichtiger geworden und wird in der Praxis v.a. bei Entscheidungen, die für Personen nachteilig sein können (z.B. Verweigerung einer Versicherung) immer wichtiger. D.h. Analysten sind gefordert, die Black box etwas zu öffnen.
* Bei einem Regressionsmodel (LS oder logistisch) ist die Einschätzung, welche Relevanz welche Prädiktoren haben, relativ simpel. Bei komplexeren (wie random forest) nicht. Dafür gibt’s das vip package (variable importance). Für einige Modelltypen (z.B. KNN) geht das allerdings (noch) nicht.
* In jedem Fall muss man sich hüten, den Prädiktoren jenseits ihrer Vorhersage-Fähigkeit irgendeine (kausale) Rolle zu geben.

## Einfacher Fall: Koeffizienten und deren Signifikanz

* Die Koeffizienten findet man im fitted workflow (gibt die Koeffizienten und p-Wert)—diesen kann man mit extract\_fit\_parsnip extrahieren (Hinweis: In älteren Videos von Julia Silge findet man die Form fit$workflow[[1]] oder pull\_workflow\_fit())

penguin\_final %>%

extract\_fit\_parsnip()%>%

tidy()

# A tibble: 7 x 5

term estimate std.error statistic p.value

<chr> <dbl> <dbl> <dbl> <dbl>

1 (Intercept) -79.5 13.5 -5.90 0.00000000369

2 speciesChinstrap -6.61 1.70 -3.89 0.000101

3 speciesGentoo -9.13 2.89 -3.16 0.00159

4 bill\_length\_mm 0.576 0.137 4.20 0.0000268

5 bill\_depth\_mm 1.36 0.373 3.64 0.000273

6 flipper\_length\_mm 0.0707 0.0538 1.31 0.189

7 body\_mass\_g 0.00509 0.00108 4.70 0.00000260

* Hinweise
  + Der **workflow**, ausdem die Koeffizienten extrahiert werden, ist eine Spalte aus dem gefitteten workflow, welches das Ergebnis der last\_fit()-Operation ist, bei der das Modell final auf die Trainingsdaten gefittet und an den Testdaten evaluiert wurde. D.h. dort sind die geschätzen Parameter drin, die sich manchmal direkt anschauen lassen (z.B. bei Regressionsmodellen) und die mittels predict() Funktion genutzt werden, um den Fit in den Testdaten zu evaluieren.
  + Für die Extraktion der Koeffizienten ist es egal, ob man das auf Basis der Trainings- oder Testdaten macht, weil sie sowieso aus dem Training im Training set resultuieren (!). Nur dort gibt es ja eine Parameter-Generierung.

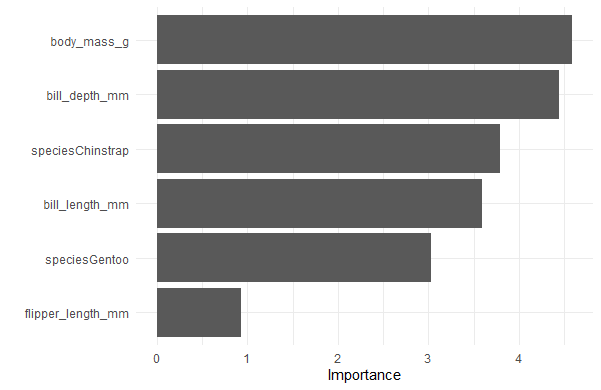
## Variable importance plots bei nicht-parametrischen Modellen

* Bei nicht-parameterischen Modellen gibt es eben keine Parameter. Dafür liefert die vip()-Funktion einen plot, der die Wichtigkeit des prädiktiven Beitrags eines Prädiktors anzeigt.
* Hier mal angewendet auf das Beispiel der logistischen Regression oben.
* Hier wieder angewandt auf den finalen workflow (Achtung: Da das Modell nur im Training Set gefittet/trainiert wurde, betrifft er nur diesen)

penguin\_final %>%

extract\_fit\_parsnip() %>%

vip::vip(num\_features = 10)



## Besonderheit 1: VIP bei Regularisierung

* Wenn man einen VIP anfordert für ein **LASSO model**, kann es durchaus passieren, dass plötzlich Prädiktor eine besondere Wichtigkeit hat, der aber im Modell auf 0 geshrinkt worden war! Ich hatte sogar einen Fall, wo der wichtigste Prädiktor im VIP ein 0-geshrinkter Parameter war.
* Analog können die Regressionseffekte der nicht-geshrinkten Prädiktoren ein anderes Ranking haben als das Ranking im VIP!
* Das liegt daran, dass die VIP Funktion anders vorgeht: So werden zur Analyse der Wichtigkeit eines Prädiktors die Reihenfolge der Werte der Fälle „durcheinandergebracht“ und die Auswirkung dieses kleinen Experiments auf den prediction error beobachtet. Ist sie gering, ist der Prädiktor anscheinende nicht wichtig (siehe kurzes Video [hier](https://www.youtube.com/watch?v=_Mv9kIHiTkc&ab_channel=CynthiaRudin)).

## Bsonderheit 2: VIP bei random forests

* Wenn man ein random forest model hat (siehe im entsprechenen applications-Kapitel), muss man bei set\_engine(ranger, importance = TRUE) nennen. Nachteil ist aber, dass dieses zusätzliche Argument das ganze Training extrem verlangsamt (v.a. beim tuning, siehe Kapitel 5). Wenn man das tun möchte, kann eine Alternative sein, es erst mal ohne das importance Argument zu machen und am Ende die model spec mit dieser importance-Anforderung nur für das Gewinner-Modell zu machen. Dies macht Julia Silge hier in einem [Video](https://www.youtube.com/watch?v=BgWCuyrwD1s&t=971s&ab_channel=JuliaSilge) bei 30:00.
* Im oben verlinkten kurze Erklär-Video über die VIP wird auch die Besonderheit der VIP-Funktion bei random forests erklärt

# Out-of-Sample Prediction

* Wenn die gesamte Fitting-Prozedur inkl. CV und Test-Evaluation erfolgreich durchlaufen wurde, fragt sich, wie man das Modell auf völlig neue Daten anwenden kann („model deployment“). Es gab schon einige Beispiele im Skript, in denen die predict-Funktion verwendet wurde. Hier wird sie final noch mal auf den gesamten Datensatz angewendet.
* Der [Blog](https://www.rebeccabarter.com/blog/2020-03-25_machine_learning/)-Post von Rebecca Barker enthält am Ende einen [Teil](https://www.rebeccabarter.com/blog/2020-03-25_machine_learning/#fitting-and-using-your-final-model), wo sie dies nach einem random forest model macht. Hier der [code](https://github.com/IcarusAE/BusinessAnalytics/blob/main/Rscripts/RebeccaBarker.R), den ich etwas angepasst habe.
* Dazu wird am Ende die fit()-Funktion nochmal auf die **gesamten Daten** (d.h. Training Set + Test Set) angewendet
* Dies resultiert in einem neuen workflow-Objekt

final\_model <- fit(final\_rf\_wf, diabetes\_clean)

Hier ist final\_rf\_wf der finalisierte workflow und diabetes\_clean die gesamten Daten

══ Workflow [trained] ══════════════════════════════════════════════════════

Preprocessor: Recipe

Model: rand\_forest()

── Preprocessor ────────────────────────────────────────────────────────────

2 Recipe Steps

• step\_normalize()

• step\_impute\_knn()

── Model ───────────────────────────────────────────────────────────────────

Ranger result

Call:

ranger::ranger(x = maybe\_data\_frame(x), y = y, mtry = min\_cols(~4, x), importance = ~"impurity", num.threads = 1, verbose = FALSE, seed = sample.int(10^5, 1), probability = TRUE)

Type: Probability estimation

Number of trees: 500

Sample size: 768

Number of independent variables: 8

Mtry: 4

Target node size: 10

Variable importance mode: impurity

Splitrule: gini

OOB prediction error (Brier s.): 0.1595594

* Nun können mittels der üblichen predict() Funktion neue Daten (oder simulierte Szenarien) vorhergsagt werden—Grundlage ist der final-model worfklow.
* Hier mal ein simulierter einzelner Fall

new\_woman <- tribble(~pregnant, ~glucose, ~pressure, ~triceps, ~insulin,

~mass, ~pedigree,~age,

2, 95, 70, 31, 102, 28.2, 0.67, 25,

2, 195, 60, 21, 92, 42.2, 0.90, 54)

# A tibble: 2 × 8

pregnant glucose pressure triceps insulin mass pedigree age

<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

1 2 95 70 31 102 28.2 0.67 25

2 2 195 60 21 92 42.2 0.9 54

Hier mal 2 exemplarische Fälle. Eine junge Frau mit niedrigem Glucose-Level und eine ältere mit hohem level.

* Nun wird die wahrschleinlichste Klass durch die neuen X-Variablen vorhergesagt und die predicted class mittels bind\_cols() an die Daten anflanschen (geht natürlich genauso bei Tausenden von neuen Daten)

predict(final\_model, new\_data = new\_woman) %>%

bind\_cols(new\_woman)

# A tibble: 2 × 9

**.pred\_class** pregnant glucose pressure triceps insulin mass pedigree age

<fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>

**neg** 2 95 70 31 102 28.2 0.67 25

**pos** 2 195 60 21 92 42.2 0.9 54

Ergebnis: Für die erste negative Vorhersage, für die zweite positive

* Wichtig: Wichtig ist, dass das Spaltenformat der neuen Daten und die Variablennamen identisch zu dem Trainingsdaten sind. Wenn also bei neuen Datensätzen nicht alle Variablen vorkommen/erhoben wurden, macht das nichts, solange es eine Variable in diesem Datensatz gibt, die so heißt wie die ursprüngliche. Deren Einträge können ruhig NA sein.

Allerdings: Wenn zentrale, hoch-prädiktive Variablen fehlen, kann daraus eine falsche Vorhersage resulitieren. Dann sollte man in jedem Fall imputieren.

# Time Series Forecasting

* Forecasting funktioniert nich direkt innerhalb des tidymodels framework, sein entsprechendes „tidy“ framework heißt modeltime und entspricht in der Philosophie daher dem tidymodels-framework. Auch wenn viele der Prozesschritte daher anders sind, sind andere und die generelle Logik die Gleiche
* **Ressourcen**
  + Modeltime Website: <https://business-science.github.io/modeltime/>
  + Video vom Paket-Herrsteller: <https://www.youtube.com/watch?v=-bCelif-ENY>
  + <https://www.youtube.com/watch?v=iS2MKDUuVKY>
  + <https://www.youtube.com/watch?v=6RjYIOCnRMk>
  + <https://www.youtube.com/watch?v=1pNA0GzILRw>
* <https://www.r-bloggers.com/2020/06/introducing-modeltime-tidy-time-series-forecasting-using-tidymodels/>

# Zusammenfassung

**(1) Split the data**

set.seed(123)

jsat\_split = initial\_split(jsat\_data, prop=.7)

jsat\_train = training(jsat\_split)

jsat\_test = testing(jsat\_split)

set.seed(345)

jsat\_cv = vfold\_cv(jsat\_train, v = 10)

svm\_spec <- **#Nonlinear SVM**

svm\_rbf(cost = tune(), rbf\_sigma = tune()) %>%

set\_mode("classification") %>%

set\_engine("kernlab")

xgb\_spec <- **#XG Boost**

boost\_tree() %>%

set\_engine("xgboost") %>%

set\_mode("classification")

**(3) Specify the Model**

lm\_spec <- **#Linear Regression**

linear\_reg() %>%

set\_engine("lm")

log\_spec <- **#Logistic Regression**

logistic\_reg() %>%

set\_engine(engine = "glm") %>%

set\_mode("classification")

rf\_spec <- **#Random Forest**

rand\_forest() %>%

set\_engine("ranger", importance = "impurity") %>%

set\_mode("classification")

knn\_spec <- **#K nearest neighbor**

nearest\_neighbor() %>%

set\_engine("kknn") %>%

set\_mode("classification")

nb\_spec <- **#Naive Bayes**

naive\_Bayes() %>%

set\_engine("naivebayes")%>%

set\_mode("classification")

svm\_spec <- **#Linear SVM**

svm\_linear() %>%

set\_mode("classification") %>%

set\_engine("LiblineaR")

**(10) Evaluate model (Test Set)**

knn\_testfit %>% collect\_metrics()

**(9) Fit the Best Model on the Test Set**

xxx\_testfit <- final\_jsat\_wf %>%

last\_fit(jsat\_split)

**(8) Finalize Workflow**

final\_jsat\_wf <- jsat\_wf %>%

finalize\_workflow(select\_best(xxx\_rs, metric="accuracy"))

**(7) Evaluate model (Assessment Set)**

xxx\_rs %>% collect\_metrics()

**(6) Fit the Data / tune HPs**

xxx\_rs <- tune\_grid(

jsat\_wf,

resamples = jsat\_cv,

grid = xxx\_exp\_grid,

control = control\_resamples(save\_pred = TRUE, verbose=TRUE)

)

**(5) Generate Tuning Grid**

xxx\_grid = expand.grid(<HP name> = <Bereich an Werten)

**(4) Add to Workflow**

jsat\_wf <- workflow()%>%

add\_recipe(jsat\_rec) %>%

add\_model(xxx\_spec)

**(2) Create the Recipe**

jsat\_rec = recipe(job\_satisfaction ~ ., data= jsat\_train) %>%

step\_normalize(all\_numeric\_predictors()) %>%

step\_dummy(all\_nominal\_predictors()) %>%

themis::step\_smote(job\_satisfaction)