Case Study: Vorhersage von Versicherungsaufwendungen

Einführung

In dieser Case Study analysieren wir Versicherungsdaten, um mit Machine-Learning-Modellen die Versicherungsaufwendungen (**expenses**) vorherzusagen. Ziel ist es, die Teilnehmer schrittweise durch eine praxisorientierte Analyse zu führen und dabei die wichtigsten Konzepte der Datenvorverarbeitung und Modellierung zu vermitteln.

Hinweis: Diese Aufgaben umfassen den gesamten Machine-Learning-Prozess von der Datenaufbereitung über die Modellierung bis zur Evaluierung.

1. Laden und Untersuchen der Daten

Aufgabe 1: Dateneinsicht

- Laden Sie den Datensatz und verschaffen Sie sich einen Überblick.
- Welche Variablen könnten wichtige Prädiktoren für die Versicherungsaufwendungen sein?

Hinweis: Verwenden Sie skim() und plot_missing() aus dem Paket DataExplorer, um fehlende Werte und wichtige Datenmerkmale zu erkennen.

2. Umgang mit fehlenden Werten

Aufgabe 2: Imputation

- Führen Sie die Imputation durch und vergleichen Sie die Daten vor und nach der Imputation.
- Warum ist die Imputation ein wichtiger Schritt?

Hinweis: Nutzen Sie plot_missing() erneut, um den Erfolg der Imputation zu überprüfen.

3. Datensplitting

Wir teilen den Datensatz in Trainings- und Testdaten auf.

1 Aufgabe 3: Datensplitting

- Nutzen Sie Strata=… beim Splitting → Für Balancing
- Warum ist das Splitten in Trainings- und Testdaten notwendig?

4. Erstellen eines Rezepts

Aufgabe 4: Rezept-Erstellung

- Welche Schritte werden im Rezept ausgeführt?
- Experimentieren Sie mit dem Entfernen oder Hinzufügen von Schritten.

5. Lineares Modell mit Workflow

Wir erstellen einen Workflow und trainieren ein lineares Regressionsmodell.

Ein workflow() in R ist ein Werkzeug, das **Modellierungsschritte und Vorverarbeitungsrezepte** (z.B. durch das Paket {recipes}) elegant miteinander verbindet. Es stellt sicher, dass die Datenvorbereitung und das Training eines Modells konsistent und reproduzierbar sind.

Wichtigkeit:

 Modulare Struktur: Trennt Datenvorverarbeitung (recipes) und Modelltraining (models).
 Konsistenz: Fehler durch inkonsistente Datenaufbereitung werden vermieden.
 Wiederverwendbarkeit: Workflows lassen sich für unterschiedliche Modelle und Rezepte anpassen.

 Automatisierung: Komplexe Pipelines (z.B. Feature Engineering, Transformationen) werden automatisiert.

```
lm_workflow <- workflow() %>%
  add_recipe(insurance_recipe) %>%
  add_model(linear_reg() %>% set_engine("lm"))
```

 workflow(): Erstellt ein leeres Workflow-Objekt.
 add_recipe(insurance_recipe): Bindet das Datenvorbereitungsrezept (insurance_recipe) ein, z.B. Skalierung oder Transformationen der Versicherungsdaten.
 add_model(): Fügt das gewünschte Modell hinzu, in diesem Fall eine lineare Regression mit der Engine "lm".

```
lm_fit <- lm_workflow %>% fit(data = train_data)
```

 lm_workflow: Das bereits definierte Workflow-Objekt, das sowohl das Rezept (insurance_recipe) als auch das Modell (linear_reg) enthält.
 fit(): Diese Funktion trainiert den gesamten Workflow — einschließlich Datenvorverarbeitung und Modelltraining.
 data = train_data: Die Trainingsdaten (train_data), mit denen das Modell trainiert wird.

Aufgabe 5: Lineare Regression

- Erklären Sie die Bedeutung des Root Mean Squared Error (RMSE).
- Interpretieren Sie die Leistung des Modells anhand des RMSE.

6. SVM mit Workflow & Hyperparm tuning

6.1 Modell definition mir radialem Kernel

```
# Definiere das Modell: SVM mit radialem Kernel (mit kernlab als Engine)

svm_model <- svm_rbf(
  cost = tune(),# Hyperparameter, der optimiert werden soll
  rbf_sigma=tune()
  ) %>%
  set_engine("kernlab") %>%  # Verwende den "kernlab" Engine
  set_mode("regression")  # Regression, da es um die Vorhersage einer kontinuierlichen Variablen geht
```

- **`svm_rbf()`**: Erzeugt ein SVM-Modell mit einem radialen Basisfunktions-Kernel (RBF).
- **`cost = tune()`**: Steuert, wie streng das Modell Fehlklassifikationen bestraft.
    - Höhere Werte führen zu komplexeren Modellen mit geringerem Bias, aber höherem Risiko von Overfitting.
- **`rbf_sigma = tune()`**: Bestimmt die Flexibilität des Kernels.
- **`set_engine("kernlab")`**: Auswahl der `kernlab`-Engine zur Implementierung von SVM in R.
- **`set_mode("regression")`**: Da das Ziel eine kontinuierliche Variable ist, wird ein Regressionsproblem definiert.

6.2 Workflow -Erstellung

```
svm_workflow <- workflow() %>%
  add_recipe(insurance_recipe) %>%
  add_model(svm_model)
```

Kombiniert das Datenvorbereitungsrezept (`insurance_recipe`) und das SVM-Modell (`svm_model`) in einem Workflow.

6.3 Hyperparameter-Tuning-Grid

```
tune_grid <- expand.grid(
  cost = c(0.1, 1, 10, 100),
  rbf_sigma = c(0.01, 0.1, 1, 10)
)
```

- **Cost (C)**

Funktion: Bestimmt den Kompromiss zwischen Fehlern und der Komplexität des Modells.

Hoher C-Wert: Strengere Fehlerkontrolle, führt zu einem komplexeren Modell, kann **Overfitting**verursachen.

Niedriger C-Wert: Mehr Fehlertoleranz, führt zu einem allgemeineren Modell, kann **Underfitting**verursachen.

Typische Werte: 0.01 bis 1000 (je nach Daten und Modellkomplexität).

- **RBF-Sigma (γ)**

Funktion: Steuert den Einflussbereich eines einzelnen Trainingspunkts im RBF-Kernel.

Hoher Gamma-Wert (kleines Sigma): Der Einflussbereich ist klein, das Modell wird sehr flexibel, was zu **Overfitting** führen kann.

Niedriger Gamma-Wert (großes Sigma): Der Einflussbereich ist groß, das Modell wird weniger flexibel, was zu **Underfitting** führen kann.

Typische Werte: 0.001 bis 10.

6.4 Kontrollmechanismus für Cross-Validation

```
ctrl <- control_grid(
  save_pred = TRUE,  # Speichere die Vorhersagen während des Tuning-Prozesses (wichtig bei mehrere Modelle um zu vergleichen)
  verbose = TRUE  # Gib mehr Informationen aus- Fortschritte beim training
)
```

6.5 Durchführung der Hyperparameter-Tuning

```
tuned_results <- tune_grid(
  object = svm_workflow,
  resamples = vfold_cv(train_data, v = 5),  # 5-fache Kreuzvalidierung
  grid = tune_grid,
  control = ctrl
)
```

6.6 Auswahl der besten parameter

```
best_params <- tuned_results %>%
   select_best(metric = "rmse")
```

6.7 Finalizierung des Modells

```
final_model <- finalize_workflow(svm_workflow, best_params)
```

6.8 Trainieren des Final_model & Prognostizieren

Aufgabe 6: SVM

- Erklären Sie die Bedeutung des Root Mean Squared Error (RMSE).
- Interpretieren Sie die Leistung des Modells anhand des RMSE.

7. RF Modell

7.1 Modell Definition mit Ranger

```
rf_model <- rand_forest(
  mtry = tune(),        # Hyperparameter für mtry
  trees = tune()        # Hyperparameter für num.trees
) %>%
  set_engine("ranger") %>%
  set_mode("regression")  # Regression, da es um die Vorhersage einer kontinuierlichen
Variablen geht
```

7.2 Weitere Schritte wie 6. Mit folgenden Änderungen

```
# Erstelle den Tuning-Grid
tune_grid <- expand.grid(
  trees = c(100, 200, 300, 400, 500),
  mtry = c(2, 4, 6)
)
```

8. XGboost Modell

8.1 Modell Definieren

```
xgb_model <- boost_tree(
  trees = tune(),
  tree_depth = tune(),
  learn_rate = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

8.2 Weitere Schritte mit folgenden Änderungen

```
xgb_grid <- grid_regular(
  trees(range = c(100, 500)),
  tree_depth(range = c(3, 10)),
  learn_rate(range = c(0.01, 0.3)),
  levels = 5
)
```

`trees(range = c(100, 500))`: Anzahl der Entscheidungsbäume im Modell, Werte zwischen 100 und 500 werden getestet.

`tree_depth(range = c(3, 10))`: Maximale Tiefe der Bäume, Werte zwischen 3 und 10 werden getestet.

`learn_rate(range = c(0.01, 0.3))`: Lernrate, die angibt, wie stark jedes neue Modell das bestehende korrigiert. Werte zwischen 0.01 und 0.3 werden ausprobiert.
`levels = 5`: Jede Hyperparameter-Dimension wird in 5 gleichmäßig verteilte Punkte unterteilt.

Typische Werte für die Lernrate

1. **Sehr niedrige Lernrate (z.B. 0.0001 bis 0.001)**:
   - **Vorteil**: Sehr präzise Anpassungen an den Modellparametern, langsame, aber stabile Konvergenz.
   - **Nachteil**: Der Lernprozess kann extrem langsam sein, was bei großen Datensätzen oder komplexen Modellen sehr ressourcenintensiv wird.
2. **Niedrige Lernrate (z.B. 0.001 bis 0.01)**:
   - **Vorteil**: Das Modell lernt stabiler, ohne große Sprünge oder Instabilitäten.
   - **Nachteil**: Es kann viele Iterationen benötigen, um eine gute Lösung zu finden, was in längeren Trainingszeiten resultiert.
3. **Mittlere Lernrate (z.B. 0.01 bis 0.1)**:
   - **Vorteil**: Diese Werte bieten oft einen guten Kompromiss zwischen Geschwindigkeit und Stabilität. In vielen Fällen konvergieren Modelle in einem akzeptablen Zeitraum.
   - **Nachteil**: Bei zu großen Werten könnte das Modell überschiessen und die Lösung nicht stabil finden.
4. **Hohe Lernrate (z.B. 0.1 bis 0.3)**:
   - **Vorteil**: Der Lernprozess kann schnell sein und das Modell erreicht möglicherweise schnell eine grobe Lösung.
   - **Nachteil**: Das Modell kann in Instabilität geraten und die optimale Lösung überschreiten, insbesondere bei komplexen Daten oder tiefen Netzwerken. Es kann auch zu einem ungenauen Minimum führen.
5. **Sehr hohe Lernrate (z.B. 0.5 bis 1.0)**:
   - **Vorteil**: In sehr wenigen Iterationen könnte ein grobes, aber schnelles Lernen erreicht werden.
   - **Nachteil**: Das Modell wird wahrscheinlich nie zu einer stabilen oder optimalen Lösung konvergieren. Die Fehlerfunktion könnte schwingen oder sogar divergieren.

→ levels in unserem Bsp mit 5 und lernrate 0.01 und 0.3 :

0.01 0.0825 0.155 0.2275 0.3