Hyperparameter-Tuning mit Grid Search

by Woche 24

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,
RepeatedStratifiedKFold, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
import warnings
warnings.filterwarnings('ignore')
np.random.seed(42)
```

In den vorangegangenen Kapiteln haben wir verschiedene Klassifikationsalgorithmen kennengelernt: Logistische Regression, Decision Trees, Random Forest, k-Nearest Neighbors und Support Vector Machines. Dabei haben wir immer wieder gesehen, dass jeder Algorithmus verschiedene **Hyperparameter** besitzt, die seine Performance beeinflussen können.

Bisher haben wir diese Parameter meist manuell gesetzt oder die Standardwerte verwendet. Aber wie findet man systematisch die **besten** Hyperparameter für ein gegebenes Problem? Wie vermeidet man, dass man stundenlang verschiedene Kombinationen ausprobiert, ohne zu wissen, ob man die optimale Lösung gefunden hat?

Hier kommt **Grid Search** ins Spiel – eine systematische Methode zur Hyperparameter-Optimierung, die alle relevanten Kombinationen durchprobiert und dabei gleichzeitig Data Leakage verhindert.

Kurzer Rückblick: Hyperparameter der Algorithmen

Bevor wir uns Grid Search anschauen, erinnern wir uns an die wichtigsten Hyperparameter unserer Algorithmen und was sie bewirken:

Logistische Regression:

- `C`: Inverse Regularisierungsstärke. Kleine Werte führen zu stärkerer Regularisierung und einfacheren Modellen.
- `penalty`: Art der Regularisierung. `'l1'` kann Features automatisch auswählen, `'l2'` verkleinert Koeffizienten gleichmäßig.

Decision Tree:

- `max_depth`: Maximale Tiefe des Baums. Kleine Werte (z.B. 3-5) führen zu einfachen Modellen mit weniger Overfitting, große Werte oder `None` zu komplexeren Bäumen, die sich stärker an die Trainingsdaten anpassen.

- **`min_samples_split`**: Mindestanzahl Samples, um einen Knoten zu teilen. Höhere Werte (z.B. 10-20) machen den Baum konservativer und reduzieren Overfitting.
- **`min_samples_leaf`**: Mindestanzahl Samples in einem Blatt. Verhindert zu kleine Endknoten und macht das Modell stabiler.

Random Forest:

- **`n_estimators`**: Anzahl der Bäume im Forest. Mehr Bäume führen meist zu besserer Performance, aber längerer Trainingszeit.
- **`max_depth`, `min_samples_split`, `min_samples_leaf`**: Gleiche Bedeutung wie bei Decision Trees, aber angewendet auf jeden einzelnen Baum im Forest.

k-Nearest Neighbors:

- **`n_neighbors`**: Anzahl der berücksichtigten Nachbarn. Kleine Werte (z.B. 3-5) sind empfindlicher für Ausreißer, große Werte (z.B. 15-20) führen zu glatteren Entscheidungsgrenzen.
- **`weights`**: Wie Nachbarn gewichtet werden. `'uniform'` behandelt alle gleich, `'distance'` gewichtet nähere Nachbarn stärker.

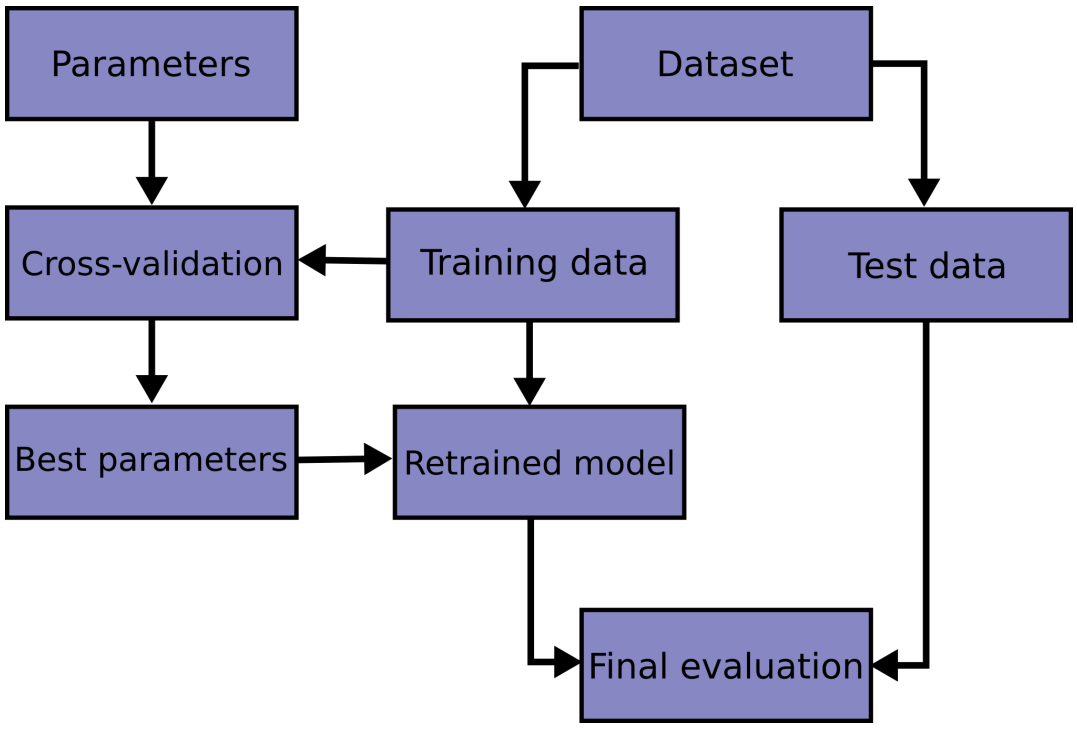
Support Vector Machine:

- **`C`**: Fehlertoleranz-Parameter. Kleine Werte (z.B. 0.1) erlauben mehr Fehlklassifikationen für größere Margin, große Werte (z.B. 100) streben perfekte Trennung an.
- **`gamma`**: Reichweite des Einflusses einzelner Support Vectors (nur bei RBF-Kernel). Kleine Werte führen zu glatteren Entscheidungsgrenzen.
- **`kernel`**: Art der Transformation. `'linear'` für gerade Trennlinien, `'rbf'` für komplexe gekrümmte Grenzen.

Das Problem: Diese Parameter interagieren oft miteinander, und die optimale Kombination ist nicht vorhersagbar. Was bei einem Datensatz funktioniert, muss bei einem anderen nicht optimal sein.

Der korrekte Workflow: Doppelter Split für ehrliche Evaluation

Hyperparameter-Tuning wird in der Regel quasi mit einen **doppelten Split** der Daten durchgeführt, um ehrliche Performance-Evaluation zu gewährleisten.



Der vollständige Workflow:

1. **Erster Split:** Ein einmaliger Train-Test Split → z.B. 20% der Daten werden als **echtes Holdout-Test-Set** beiseitegelegt
2. **Zweiter Split:** Grid Search mit Cross-Validation **nur** auf den 80% Trainingsdaten. Diese 80% Daten erfahren also während der CV wiederum mehrfach Train-Test-Splits. So finden wir dann die beste Kombination von Hyperparametern. Sobald die gefunden wurde trainieren wir das finale Modell auf allen 80% Trainingsdaten.
3. **Finale Evaluation:** Dieses retrained Modell wird **einmalig** auf dem echten Holdout-Test-Set evaluiert

Warum der doppelte Split?

- **Problem ohne Holdout-Set:** Grid Search mit CV alleine würde bedeuten, dass wir indirekt alle Daten für die Hyperparameter-Optimierung verwenden. Selbst mit CV-Folds "sieht" Grid Search durch die verschiedenen Iterationen letztendlich alle Daten.
- **Lösung mit Holdout-Set:** Das 20% Test-Set war **niemals** Teil der Hyperparameter-Optimierung und gibt uns eine unvoreingenommene finale Performance-Schätzung.

GridSearchCV macht das Retraining automatisch!

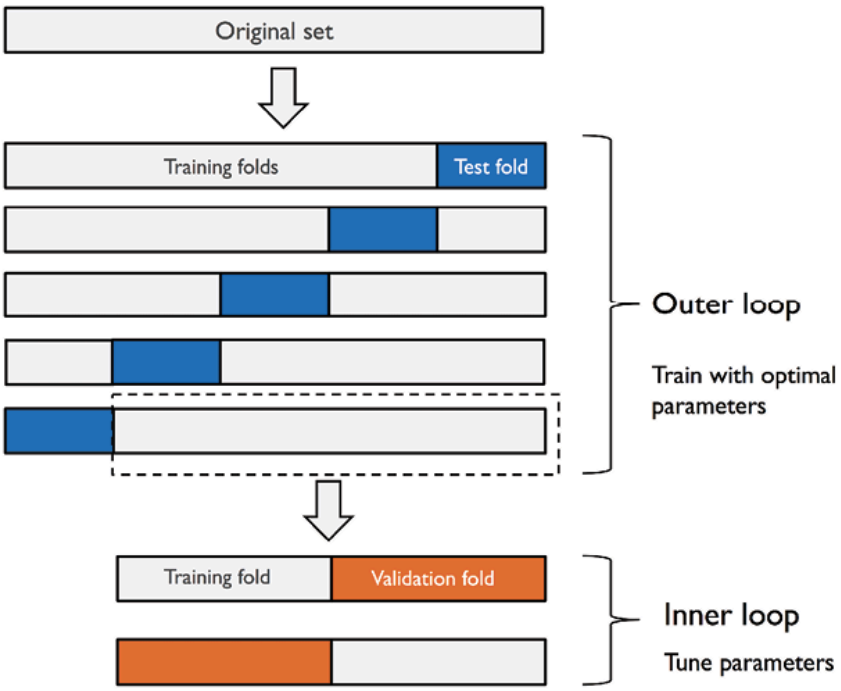
Schauen wir uns das Diagramm nochmal genau an: Nach "Best parameters" kommt "Retrained model". Dieser Schritt wird von `GridSearchCV` **automatisch** für uns erledigt!

- `GridSearchCV.fit(X_train, y_train)` führt intern Cross-Validation durch, findet die besten Parameter **und** trainiert anschließend automatisch ein finales Modell mit diesen Parametern auf **allen** `X_train` Daten
- Das finale Modell ist über `GridSearchCV.best_estimator_` verfügbar
- Wir müssen nichts manuell retrainieren!

Aus der sklearn Dokumentation: *"The best estimator is fit with the complete training set."*

Methodische Einordnung: Pragmatisch vs. Optimal

Dieser Workflow ist ein weit verbreiteter pragmatischer Ansatz, aber die Wahl der richtigen Methode hängt vom Kontext ab. Tatsächlich kann man nämlich auch noch einen Schritt weiter gehen und auch diesen vorgelagerten Train-Test Split durch eine Kreuzvalidierung ersetzen und erhält dann **Nested Cross-Validation**



Quelle: Analytics Yogi

Methodisch optimal: Nested Cross-Validation

- Äußere CV für finale Performance-Schätzung
- Innere CV für Hyperparameter-Tuning
- Eliminiert Data Leakage komplett und ist laut sklearn die “preferred way to evaluate tuned models”
- Wichtiger bei kleinen Datensätzen (<1000 Samples), wo einzelne Splits großen Einfluss haben

Pragmatisch häufig: Train-Test Split + GridSearchCV

- Einfacher umzusetzen und weniger rechenaufwändig
- Bei großen Datensätzen oder Deep Learning oft bevorzugt, da Nested CV unpraktikabel wird
- Der Bias durch einen ungünstigen Split wird bei großen Datensätzen weniger problematisch

Fazit: Nested CV ist methodisch überlegen, aber der Train-Test + GridSearchCV Ansatz ist oft ein akzeptabler Kompromiss zwischen Korrektheit und Praktikabilität. Für Lernzwecke und kleinere Datensätze wie unseren ist er völlig angemessen.

Datenvorbereitung

Für unser Hyperparameter-Tuning verwenden wir ein anspruchsvolleres Klassifikationsproblem als bisher: **Adelie vs. Chinstrap** Pinguine mit **allen verfügbaren Features**. Diese beiden Arten sind schwerer zu unterscheiden als Adelie vs. Gentoo, wodurch Hyperparameter-Tuning relevanter wird.

Wir werden außerdem **alle Features standardisieren**, obwohl das streng genommen nur bei k-Nearest Neighbors und SVM erforderlich ist. Für die anderen Algorithmen (Logistische Regression, Decision Tree, Random Forest) hat Standardisierung keinen Nachteil und sorgt für einheitliche Preprocessing-Schritte bei allen Modellen.

```
# Palmer Penguins Datensatz laden
csv_url = 'https://raw.githubusercontent.com/SchmidtPaul/ExampleData/refs/heads/main/palmer_penguins/palmer_penguins.csv'
penguins = pd.read_csv(csv_url)
```

```
# Adelie vs. Chinstrap (schwerer zu trennen)
penguins_binary = penguins[penguins['species'].isin(['Adelie', 'Chinstrap'])].copy()

# Alle verfügbaren Features verwenden
numeric_cols = ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']
categorical_cols = ['island', 'sex']

# Datensatz bereinigen und dummy-codieren
penguins_clean = penguins_binary[numeric_cols + categorical_cols +
['species']].dropna()
penguins_encoded = pd.get_dummies(penguins_clean, columns=categorical_cols,
drop_first=True)

# Features und Zielvariable
feature_cols = [col for col in penguins_encoded.columns if col != 'species']
X = penguins_encoded[feature_cols]
y = (penguins_encoded['species'] == 'Chinstrap').astype(int)

print(f"Datensatz: {len(X)} Pinguine mit {len(feature_cols)} Features")
print(f"Features: {feature_cols}")
print(f"Klassen-Verteilung: Adelie={sum(y==0)}, Chinstrap={sum(y==1)}")
```

```
Datensatz: 214 Pinguine mit 7 Features
Features: ['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g',
'island_Dream', 'island_Torgersen', 'sex_male']
Klassen-Verteilung: Adelie=146, Chinstrap=68
```

Erster Split: Echter Holdout-Testdatensatz

Zunächst legen wir also einmalig zufällig (aber stratifiziert) 20% der Daten beiseite und nutzen diese erst wieder bei der finalen Modellprüfung.

```
# Schritt 1: Train-Test Split (80% Training, 20% echtes Holdout-Test-Set)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Standardisierung auf Trainingsdaten fitten und auf beide Sets anwenden
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print(f"Training Set: {len(X_train)} Samples")
print(f"Test Set: {len(X_test)} Samples")
print(f"Training - Adelie: {sum(y_train==0)}, Chinstrap: {sum(y_train==1)}")
print(f"Test - Adelie: {sum(y_test==0)}, Chinstrap: {sum(y_test==1)}")
```

```
Training Set: 171 Samples
Test Set: 43 Samples
Training - Adelie: 117, Chinstrap: 54
Test - Adelie: 29, Chinstrap: 14
```

Grid Search: Systematische Parametersuche

Grid Search ist eine erschöpfende Suche über einen vordefinierten Parameterraum. Man definiert für jeden Hyperparameter eine Liste möglicher Werte, und Grid Search probiert **alle Kombinationen** dieser Werte aus.

Beispiel: Decision Tree

Beginnen wir mit einem einfachen Beispiel für Decision Trees:

```
# Parameter-Grid für Decision Tree
dt_param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 4, 8]
}

print(f"Parameter-Grid: {dt_param_grid}")
total_combinations = len(dt_param_grid['max_depth']) *
len(dt_param_grid['min_samples_split']) * len(dt_param_grid['min_samples_leaf'])
print(f"Anzahl Kombinationen: {total_combinations}")
```

```
Parameter-Grid: {'max_depth': [3, 5, 7, 10, None], 'min_samples_split': [2, 5, 10, 15,
20], 'min_samples_leaf': [1, 2, 4, 8]}
Anzahl Kombinationen: 100
```

Schritt 1: Grid Search auf Trainingsdaten

Grid Search führt Cross-Validation **nur** auf den Trainingsdaten durch und retrained automatisch das beste Modell:

```
# Grid Search mit Cross-Validation - NUR auf Trainingsdaten!
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=42)

dt_grid = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    dt_param_grid,
    cv=cv,
    scoring='accuracy',
    return_train_score=True
)

# WICHTIG: Grid Search NUR auf Trainingsdaten durchführen
print("Führe Grid Search auf Trainingsdaten durch...")
dt_grid.fit(X_train_scaled, y_train)

print(f"Beste Parameter: {dt_grid.best_params_}")
print(f"Beste CV-Score (auf Trainingsdaten): {dt_grid.best_score_:.4f}")
```

```
Führe Grid Search auf Trainingsdaten durch...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=DecisionTreeClassifier(random_state=42),
             param_grid={'max_depth': [3, 5, 7, 10, None],
                         'min_samples_leaf': [1, 2, 4, 8],
                         'min_samples_split': [2, 5, 10, 15, 20]},
             return_train_score=True, scoring='accuracy')
Beste Parameter: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5}
Beste CV-Score (auf Trainingsdaten): 0.9748
```

Schritt 2: Finale Evaluation auf Holdout-Test-Set

Jetzt nehmen wir das beste Modell aus dem Grid Search und testen es **einmalig** auf dem echten Test-Set:

```
# Das beste Modell aus Grid Search auf echtem Test-Set evaluieren
test_accuracy = dt_grid.score(X_test_scaled, y_test)

print(f"Finale Test-Accuracy auf Holdout-Set: {test_accuracy:.4f}")
print(f"Verwendetes Modell: {dt_grid.best_estimator_}")
```

```
Finale Test-Accuracy auf Holdout-Set: 0.9302
Verwendetes Modell: DecisionTreeClassifier(max_depth=5, min_samples_split=5,
random_state=42)
```

GridSearchCV macht es uns einfach!

Sehr praktisch: `dt_grid.score(X_test_scaled, y_test)` verwendet automatisch das beste retrained Modell (`best_estimator_`) - wir müssen nichts manuell extrahieren oder neu trainieren! GridSearchCV übernimmt das komplette Workflow-Management für uns:

- ✅ Cross-Validation auf Trainingsdaten
- ✅ Beste Parameter finden
- ✅ Automatisches Retraining mit besten Parametern auf allen Trainingsdaten
- ✅ Bereitstellung für finale Evaluation

Alternativ könnten wir auch schreiben:

```
# Äquivalent, aber expliziter:
test_accuracy = dt_grid.best_estimator_.score(X_test_scaled, y_test)
# oder:
predictions = dt_grid.predict(X_test_scaled)
test_accuracy = accuracy_score(y_test, predictions)
```

Was passiert hier genau?

1. `dt_grid.fit(X_train_scaled, y_train)` führt Cross-Validation auf den Trainingsdaten durch, findet die beste Parameter-Kombination und retrained automatisch ein Modell mit diesen Parametern auf **allen** Trainingsdaten
2. `dt_grid.score(X_test_scaled, y_test)` verwendet automatisch `dt_grid.best_estimator_` (das finale retrained Modell) und testet es auf dem unabhängigen Test-Set
3. Diese Test-Accuracy ist unsere ehrliche Performance-Schätzung!

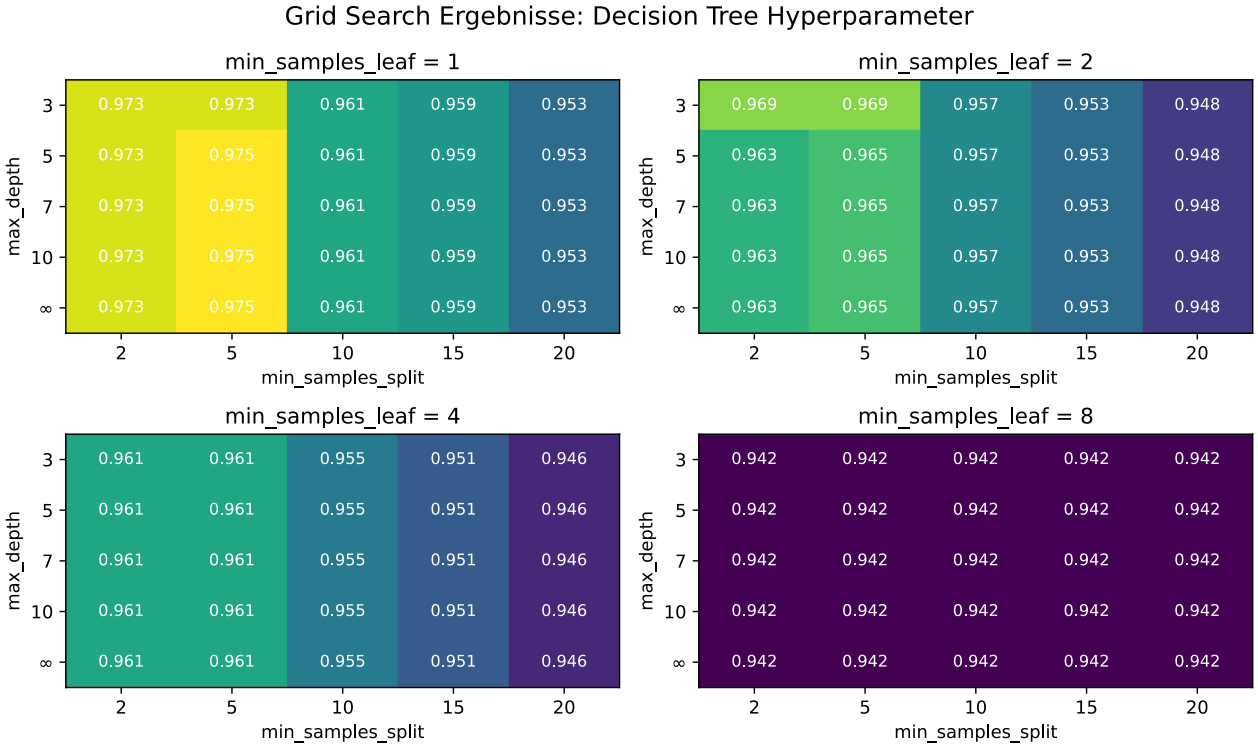
Grid Search Ergebnisse analysieren

Grid Search speichert alle Ergebnisse, sodass wir diese weiter analysieren können:

```
# Alle Ergebnisse als DataFrame
results_df = pd.DataFrame(dt_grid.cv_results_)

# Top 10 Parameter-Kombinationen
top_results = results_df.nlargest(10, 'mean_test_score')[
    ['mean_test_score', 'std_test_score', 'param_max_depth',
     'param_min_samples_split', 'param_min_samples_leaf']
]
```

```
print("Top 10 Parameter-Kombinationen (basierend auf CV):")
for i, row in top_results.iterrows():
    print(f"{row['mean_test_score']:.4f} (±{row['std_test_score']:.4f}): "
          f"depth={row['param_max_depth']}, split={row['param_min_samples_split']},
leaf={row['param_min_samples_leaf']}")

# Visualisierung der Grid Search Ergebnisse als Heatmaps
# 2x2 Layout für vier min_samples_leaf Werte
fig, axes = plt.subplots(2, 2, figsize=(10, 6), layout='tight')

# Bereite Daten für Heatmaps vor
max_depth_values = dt_param_grid['max_depth']
min_samples_split_values = dt_param_grid['min_samples_split']
min_samples_leaf_values = dt_param_grid['min_samples_leaf']
vmin = results_df['mean_test_score'].min()
vmax = results_df['mean_test_score'].max()

# Erstelle Labels für max_depth (None als "∞" darstellen)
max_depth_labels = [str(d) if d is not None else "∞" for d in max_depth_values]

# Farbverlauf von 0 (hellste) bis 1 (dunkelste) - ohne Legende
for i, leaf_val in enumerate(min_samples_leaf_values):
    row = i // 2
    col = i % 2
    ax = axes[row, col]

    # Filtere Ergebnisse für diesen min_samples_leaf Wert
    subset = results_df[results_df['param_min_samples_leaf'] == leaf_val]

    # Erstelle Heatmap-Matrix
    heatmap_data = np.zeros((len(max_depth_values), len(min_samples_split_values)))

    for _, row_data in subset.iterrows():
        depth_idx = max_depth_values.index(row_data['param_max_depth'])
        split_idx = min_samples_split_values.index(row_data['param_min_samples_split'])
        heatmap_data[depth_idx, split_idx] = row_data['mean_test_score']

    # Zeichne Heatmap mit Farbverlauf von 0 bis 1
    im = ax.imshow(heatmap_data, cmap='viridis', vmin = vmin, vmax = vmax,
aspect='auto');

    # Füge Werte als Text hinzu
    for depth_idx in range(len(max_depth_values)):
        for split_idx in range(len(min_samples_split_values)):
            text = ax.text(split_idx, depth_idx, f'{heatmap_data[depth_idx,
split_idx]:.3f}',
                           ha="center", va="center", color="white", fontsize=9);

    # Achsenbeschriftung
    ax.set_xticks(range(len(min_samples_split_values)));
    ax.set_xticklabels(min_samples_split_values);
    ax.set_yticks(range(len(max_depth_values)));
    ax.set_yticklabels(max_depth_labels);
    ax.set_xlabel('min_samples_split');
    ax.set_ylabel('max_depth');
    ax.set_title(f'min_samples_leaf = {leaf_val}');

plt.suptitle('Grid Search Ergebnisse: Decision Tree Hyperparameter', fontsize=14);
plt.show()
```

```
Top 10 Parameter-Kombinationen (basierend auf CV):
0.9748 (±0.0277): depth=5, split=5, leaf=1
0.9748 (±0.0277): depth=7, split=5, leaf=1
0.9748 (±0.0277): depth=10, split=5, leaf=1
0.9748 (±0.0277): depth=None, split=5, leaf=1
0.9729 (±0.0290): depth=3, split=2, leaf=1
0.9729 (±0.0290): depth=3, split=5, leaf=1
0.9729 (±0.0269): depth=5, split=2, leaf=1
0.9729 (±0.0269): depth=7, split=2, leaf=1
0.9729 (±0.0269): depth=10, split=2, leaf=1
0.9729 (±0.0269): depth=None, split=2, leaf=1
```



ℹ Parallelisierung mit n_jobs

GridSearchCV kann die Berechnungen **parallelisieren**, um die Laufzeit zu reduzieren:

- **`n_jobs=None`** (Default): Sequentielle Verarbeitung, 1 Prozessorkern
- **`n_jobs=-1`**: Nutzt alle verfügbaren Prozessorkerne
- **`n_jobs=4`**: Nutzt 4 Prozessorkerne (oder eine andere spezifische Anzahl)

Beispiel:

```
dt_grid = GridSearchCV(
    DecisionTreeClassifier(random_state=42),
    dt_param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1  # Alle verfügbaren Kerne nutzen
)
```

Die Parallelisierung lohnt sich (i) bei vielen Parameter-Kombinationen (>50), (ii) bei langsamen Algorithmen (SVM, Random Forest mit vielen Bäumen) und (iii) bei größeren Datensätzen.

Achtung: Mehr Parallelisierung = höherer Speicherverbrauch. Bei sehr großen Datensätzen kann n_jobs=-1 zu Memory-Problemen führen.

Systematischer Vergleich aller Algorithmen

Nun führen wir Grid Search für alle unsere Klassifikationsalgorithmen durch. Der **doppelte Split** Workflow gilt für jeden Algorithmus:

Für jeden Algorithmus: 1. **Schritt 1:** Grid Search mit CV **nur** auf Trainingsdaten (`X_train_scaled`, `y_train`) 2. **Schritt 2:** Finale Evaluation des besten Modells auf Holdout-Test-Set (`X_test_scaled`, `y_test`)

```
# Entsprechende Modelle definieren
base_models = {
    'Logistic Regression': LogisticRegression(random_state=42),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'k-Nearest Neighbors': KNeighborsClassifier(),
    'SVM': SVC(random_state=42)
}

# Parameter-Grids für alle Algorithmen
param_grids = {
    'Logistic Regression': {
        'C': [0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear']  # Unterstützt sowohl l1 als auch l2
    },

    'Decision Tree': {
        'max_depth': [3, 5, 7, 10, None],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    },

    'Random Forest': {
```

```
        'n_estimators': [50, 100, 200],
        'max_depth': [3, 5, 7, None],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2]
    },

    'k-Nearest Neighbors': {
        'n_neighbors': [3, 5, 7, 11, 15],
        'weights': ['uniform', 'distance'],
        'metric': ['euclidean', 'manhattan']
    },

    'SVM': {
        'C': [0.1, 1, 10, 100],
        'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
        'kernel': ['rbf', 'linear']
    }
}

# Anzahl Kombinationen pro Algorithmus berechnen
for name, grid in param_grids.items():
    combinations = 1
    for param_values in grid.values():
        combinations *= len(param_values)
    print(f"{name}: {combinations} Kombinationen")
```

```
Logistic Regression: 10 Kombinationen
Decision Tree: 45 Kombinationen
Random Forest: 48 Kombinationen
k-Nearest Neighbors: 20 Kombinationen
SVM: 40 Kombinationen
```

Grid Search für alle Algorithmen:

```
# Grid Search für alle Algorithmen durchführen
grid_results = {}
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=42)

print("Führe Grid Search für alle Algorithmen durch...")
print("=" * 50)

for name in base_models.keys():
    print(f"\n{name}:")

    # SCHRITT 1: Grid Search NUR auf Trainingsdaten
    grid_search = GridSearchCV(
        base_models[name],
        param_grids[name],
        cv=cv,
        scoring='accuracy',
        return_train_score=True
    )

    print(f"  Grid Search auf Trainingsdaten...")
    grid_search.fit(X_train_scaled, y_train)

    print(f"  Beste Parameter: {grid_search.best_params_}")
    print(f"  Beste CV-Score: {grid_search.best_score_:.4f}")
```

```
    # SCHRITT 2: Finale Evaluation auf echtem Holdout-Test-Set
    print(f"  Finale Evaluation auf Test-Set...")
    test_score = grid_search.score(X_test_scaled, y_test)
    print(f"  --> Test-Accuracy: {test_score:.4f}")

    # Ergebnisse speichern
    grid_results[name] = {
        'grid_search': grid_search,
        'best_params': grid_search.best_params_,
        'best_cv_score': grid_search.best_score_,
        'test_score': test_score  # Das ist die wichtige finale Performance!
    }

print("\n" + "=" * 50)
print("Grid Search für alle Algorithmen abgeschlossen!")
```

```
Führe Grid Search für alle Algorithmen durch...
==================================================

Logistic Regression:
  Grid Search auf Trainingsdaten...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=LogisticRegression(random_state=42),
             param_grid={'C': [0.01, 0.1, 1, 10, 100], 'penalty': ['l1', 'l2'],
                         'solver': ['liblinear']},
             return_train_score=True, scoring='accuracy')
  Beste Parameter: {'C': 1, 'penalty': 'l1', 'solver': 'liblinear'}
  Beste CV-Score: 0.9942
  Finale Evaluation auf Test-Set...
  --> Test-Accuracy: 0.9767

Decision Tree:
  Grid Search auf Trainingsdaten...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=DecisionTreeClassifier(random_state=42),
             param_grid={'max_depth': [3, 5, 7, 10, None],
                         'min_samples_leaf': [1, 2, 4],
                         'min_samples_split': [2, 5, 10]},
             return_train_score=True, scoring='accuracy')
  Beste Parameter: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 5}
  Beste CV-Score: 0.9748
  Finale Evaluation auf Test-Set...
  --> Test-Accuracy: 0.9302

Random Forest:
  Grid Search auf Trainingsdaten...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=RandomForestClassifier(random_state=42),
             param_grid={'max_depth': [3, 5, 7, None],
                         'min_samples_leaf': [1, 2],
                         'min_samples_split': [2, 5],
                         'n_estimators': [50, 100, 200]},
             return_train_score=True, scoring='accuracy')
  Beste Parameter: {'max_depth': 7, 'min_samples_leaf': 2, 'min_samples_split': 2,
'n_estimators': 100}
  Beste CV-Score: 0.9865
  Finale Evaluation auf Test-Set...
  --> Test-Accuracy: 1.0000
```

```
k-Nearest Neighbors:
  Grid Search auf Trainingsdaten...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=KNeighborsClassifier(),
             param_grid={'metric': ['euclidean', 'manhattan'],
                         'n_neighbors': [3, 5, 7, 11, 15],
                         'weights': ['uniform', 'distance']},
             return_train_score=True, scoring='accuracy')
  Beste Parameter: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
  Beste CV-Score: 0.9923
  Finale Evaluation auf Test-Set...
  --> Test-Accuracy: 0.9767

SVM:
  Grid Search auf Trainingsdaten...
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=3, n_splits=5, random_state=42),
             estimator=SVC(random_state=42),
             param_grid={'C': [0.1, 1, 10, 100],
                         'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
                         'kernel': ['rbf', 'linear']},
             return_train_score=True, scoring='accuracy')
  Beste Parameter: {'C': 1, 'gamma': 'scale', 'kernel': 'rbf'}
  Beste CV-Score: 0.9903
  Finale Evaluation auf Test-Set...
  --> Test-Accuracy: 0.9767

================================================
Grid Search für alle Algorithmen abgeschlossen!
```

Übersichtstabelle der finalen Ergebnisse:

```python
# Ergebnisse in Tabelle darstellen - NUR Test-Performance!
summary_data = []
for name, results in grid_results.items():
    summary_data.append({
        'Algorithmus': name,
        'Test Accuracy': f"{results['test_score']:.4f}",
        'Beste Parameter': str(results['best_params'])
    })

summary_df = pd.DataFrame(summary_data)
summary_df = summary_df.sort_values('Test Accuracy', ascending=False)

print("Finale Ergebnisse (sortiert nach Test Performance):")
print(summary_df.to_string(index=False))
```

```
Finale Ergebnisse (sortiert nach Test Performance):
        Algorithmus Test Accuracy
Beste Parameter
      Random Forest        1.0000 {'max_depth': 7, 'min_samples_leaf': 2,
'min_samples_split': 2, 'n_estimators': 100}
Logistic Regression        0.9767                                    {'C': 1,
'penalty': 'l1', 'solver': 'liblinear'}
k-Nearest Neighbors        0.9767                    {'metric': 'euclidean',
'n_neighbors': 3, 'weights': 'distance'}
                SVM        0.9767                                         {'C': 1,
'gamma': 'scale', 'kernel': 'rbf'}
```

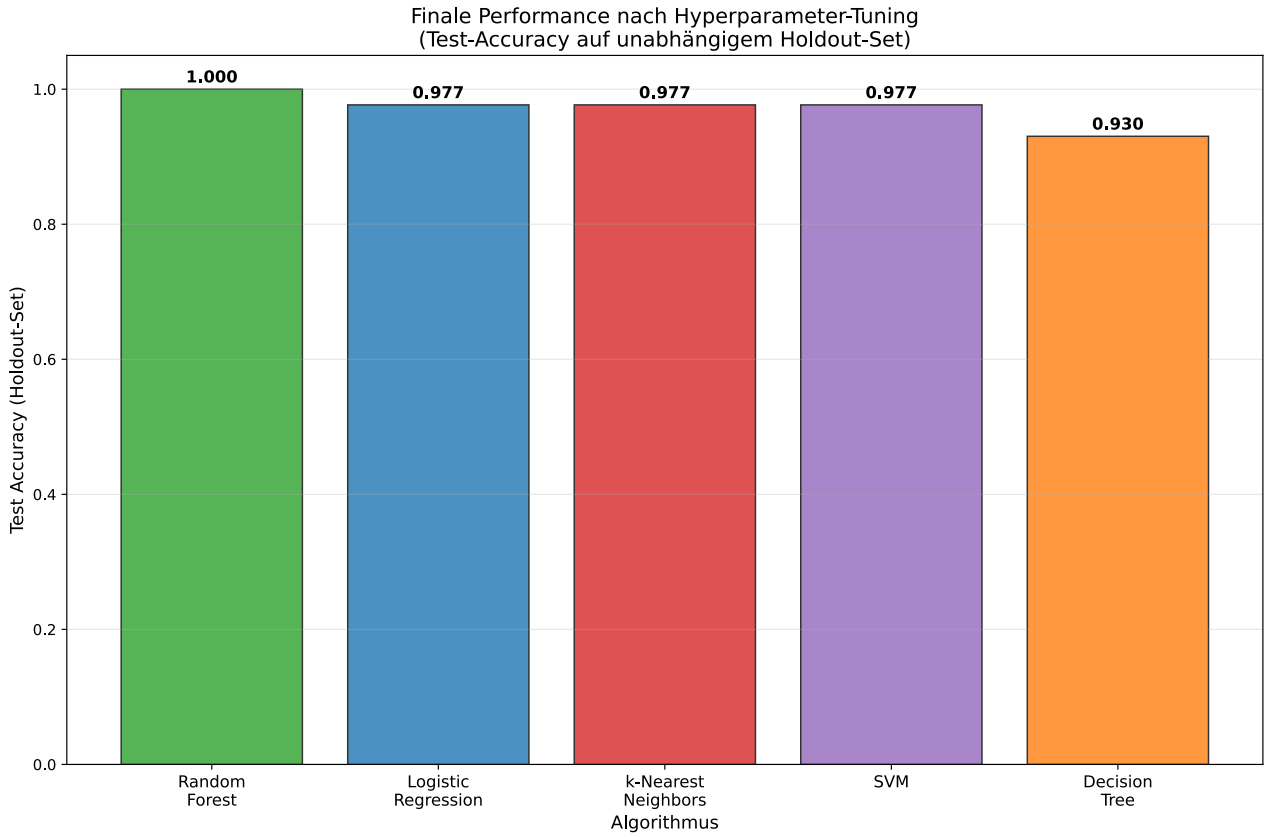
```
      Decision Tree        0.9302                     {'max_depth': 5,
'min_samples_leaf': 1, 'min_samples_split': 5}
```

Finale Test-Performance visualisieren

Wichtig: Wir zeigen nur die finale Test-Performance auf dem unabhängigen Holdout-Set - das ist die einzige ehrliche Performance-Schätzung!

```
# Finale Test-Performance visualisieren
fig, ax = plt.subplots(figsize=(12, 8), layout='tight')

# Daten für Plot vorbereiten
algorithms = list(grid_results.keys())
test_scores = [grid_results[name]['test_score'] for name in algorithms]

# Farben definieren
algorithm_colors = {
    'Logistic Regression': '#1f77b4',
    'Decision Tree': '#ff7f0e',
    'Random Forest': '#2ca02c',
    'k-Nearest Neighbors': '#d62728',
    'SVM': '#9467bd'
}

# Nach Test-Score sortieren
sorted_data = sorted(zip(algorithms, test_scores), key=lambda x: x[1], reverse=True)
sorted_algorithms, sorted_scores = zip(*sorted_data)

# Farben entsprechend sortieren
colors = [algorithm_colors[name] for name in sorted_algorithms]

# Balkendiagramm erstellen
bars = ax.bar(range(len(sorted_algorithms)), sorted_scores,
              color=colors, alpha=0.8, edgecolor='black', linewidth=1)

# Beschriftung
ax.set_xlabel('Algorithmus', fontsize=12);
ax.set_ylabel('Test Accuracy (Holdout-Set)', fontsize=12);
ax.set_title('Finale Performance nach Hyperparameter-Tuning\n(Test-Accuracy auf
unabhängigem Holdout-Set)', fontsize=14);
ax.set_xticks(range(len(sorted_algorithms)));
ax.set_xticklabels([name.replace(' ', '\n') for name in sorted_algorithms],
fontsize=11);
ax.grid(True, alpha=0.3, axis='y');

# Werte auf Balken anzeigen
for i, (bar, score) in enumerate(zip(bars, sorted_scores)):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.005,
            f'{score:.3f}', ha='center', va='bottom', fontsize=11, fontweight='bold');

# Y-Achse optimieren
ax.set_ylim(0, max(sorted_scores) + 0.05);

plt.show()

# Beste Performance hervorheben
best_algorithm = sorted_algorithms[0]
best_score = sorted_scores[0]
```

```
print(f"\n🏆 Beste Performance: {best_algorithm} mit {best_score:.4f} Test-Accuracy")
print(f"Beste Parameter: {grid_results[best_algorithm]['best_params']}")
```




```
🏆 Beste Performance: Random Forest mit 1.0000 Test-Accuracy
Beste Parameter: {'max_depth': 7, 'min_samples_leaf': 2, 'min_samples_split': 2,
'n_estimators': 100}
```

Abschluss: Hyperparameter-Tuning im Kontext

Hyperparameter-Tuning lohnt sich besonders:

- Bei **komplexen Datensätzen** mit vielen Features oder schwierigen Klassifikationsproblemen
- Wenn **kleine Verbesserungen** der Performance kritisch sind (z.B. in Produktionssystemen)
- Bei **Wettbewerben** oder wenn die bestmögliche Performance gefordert ist
- Wenn **interpretierbare Modelle** gebraucht werden (z.B. kleinere, verständlichere Decision Trees)

Grid Search mit dem korrekten **doppelten Split** Workflow gibt uns die Werkzeuge für systematisches Hyperparameter-Tuning aller Klassifikationsalgorithmen. Dabei übernimmt `GridSearchCV` das komplexe Retraining automatisch für uns - eine große Erleichterung!

Mit **Logistischer Regression, Decision Trees, Random Forest, k-Nearest Neighbors, Support Vector Machines** und dem systematischen **Hyperparameter-Tuning** haben wir alle wichtigen Klassifikationsmethoden kennengelernt, die in der modernen Datenanalyse und im Machine Learning verwendet werden. Dies bildet eine solide Grundlage für anspruchsvolle Klassifikationsprojekte in der Praxis.


Launching a 6-hour long Grid Search with many params
Best params are the default params