Hyperparameter Tuning und Clean Code

Ruben Triwari

Softwareentwicklungspraktikum für das Nebenfach Künstliche Intelligenz, LMU München

10.06.2024

Outline

Lineare Regression

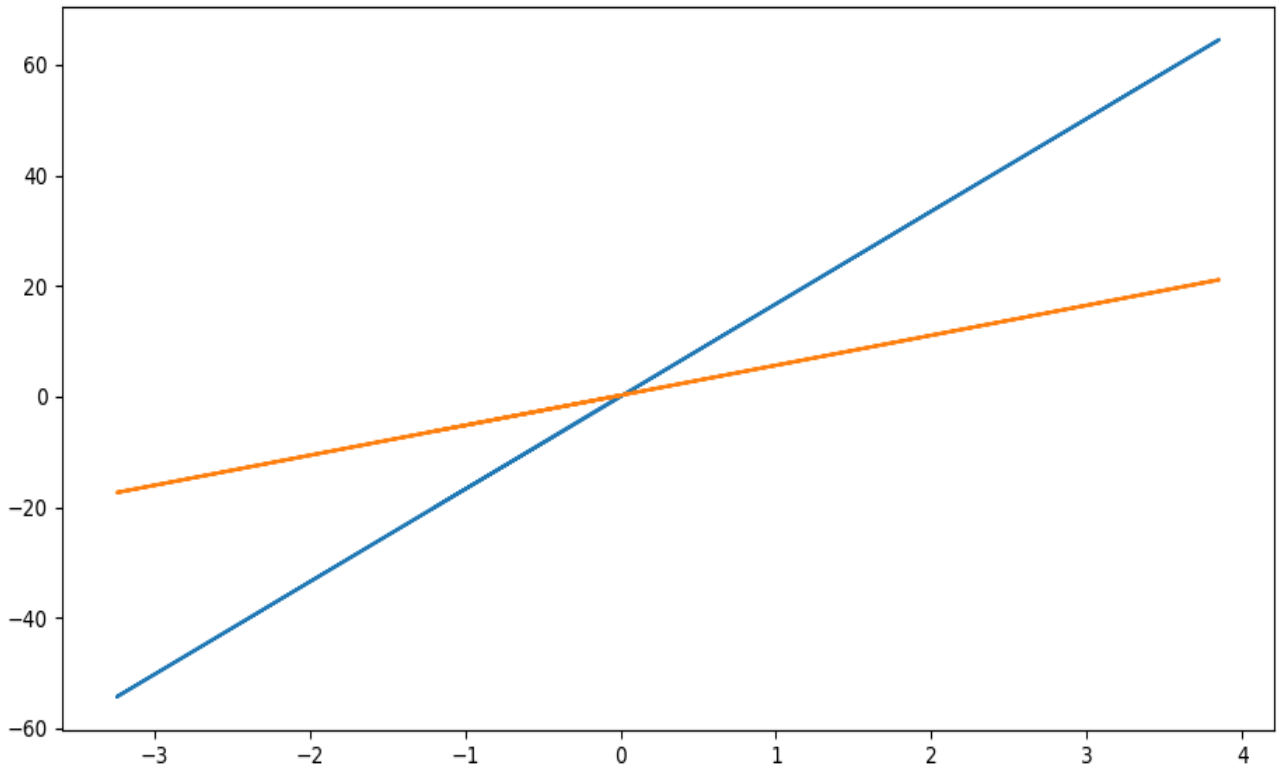
```python
from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

X, y = make_regression(
    n_samples=1000,
    n_features=1,
    n_informative=10,
    random_state=42
)

regression_model = Ridge(alpha=2000)
regression_model.fit(X,y)

print(f"score: {regression_model.score(X,y)}")
```

Motivation



●: Unser Modell ●: Optimales Modell

Accuaracy: 0.5428

Hyperparameter Tuning

Hyperparameter sind Einstellungen, die vor dem Training eines Modells festgelegt werden und maßgeblich die Leistung des Modells beeinflussen.

- Parameter vs. Hyperparameter:
  - **Parameter**: Werden während des Trainings gelernt (z.B. Gewichte in einer linearen Regression).
  - **Hyperparameter**: Werden vor dem Training festgelegt (z.B. max_depth für einen Entscheidungsbaum).

- Techniken des Hyperparameter-Tunings:
  - **Grid Search**: Systematische Suche über einen vordefinierten Parameterraum.
  - **Random Search**: Zufällige Suche über den Parameterraum.
  - **Bayessche Optimierung**: Nutzung von Wahrscheinlichkeitsmodellen zur Optimierung.

Regularisierung:

- **L1** (Lasso) und **L2** (Ridge) Regularisierung zur Vermeidung von Overfitting.
- **C**: Inverse Regularisierungsstärke, kleinerer Wert bedeutet stärkere Regularisierung

Entscheidungsbaum (Decision Tree)

Tiefe des Baums:

- **max_depth**: Maximale Tiefe des Baums zur Vermeidung von Overfitting.

Anzahl der Blätter:

- **min_samples_split**: Mindestanzahl an Stichproben, um einen Knoten zu splitten.
- **min_samples_leaf**: Mindestanzahl an Stichproben in einem Blatt.

Kriterium:

- **Gini-Index** oder **Entropie**: Bewertungskriterien für die Qualität eines Splits.

Anzahl der Cluster (k):

- Auswahl der Anzahl der Cluster, die das Clustering-Ergebnis beeinflusst.

Init-Methode:

- Methoden zur Initialisierung der Cluster-Zentren, z.B. k-means++.

Maximale Anzahl der Iterationen und Toleranz:

- **max_iter**: Maximale Anzahl der Iterationen zur Konvergenz.

Regularisierung (bei Ridge und Lasso Regression):

- **alpha**: Regularisierungsparameter, der die Größe der Strafen für die Koeffizienten beeinflusst.

Feature-Engineering:

- Bedeutung der Skalierung und Auswahl relevanter Features.

Bewertung und Analyse

Verwendung von Bibliotheken:

- Nutzung von **Scikit-Learn** für Modellierung und Hyperparameter-Tuning.
- **GridSearchCV** und **RandomizedSearchCV**: Werkzeuge in Scikit-Learn zur Hyperparameter-Optimierung.
- Anwendung von **Cross-Validation** zur Modellbewertung.

Leistungsmetriken:

- Für Klassifikationsmodelle: Genauigkeit, F1-Score, Präzision, Recall.
- Für Regressionsmodelle: R^2, MSE (Mean Squared Error).

Random Search Beispiel

```python
from sklearn.datasets import make_classification
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV

X, y = make_classification(
    n_samples=1000, n_features=20, n_informative=10, n_classes=2, random_state=42
)
param_dist = {
    "min_samples_leaf": randint(1, 9),  # Verteilung
    "max_depth": [3, 4, 5, 6],
    "criterion": ["gini", "entropy"],
}
tree = DecisionTreeClassifier()
tree_cv = RandomizedSearchCV(tree, param_dist, scoring="f1")
tree_cv.fit(X, y)

print(f"Best parameters: {tree_cv.best_params_}")
print(f"Best score: {tree_cv.best_score_}")
```

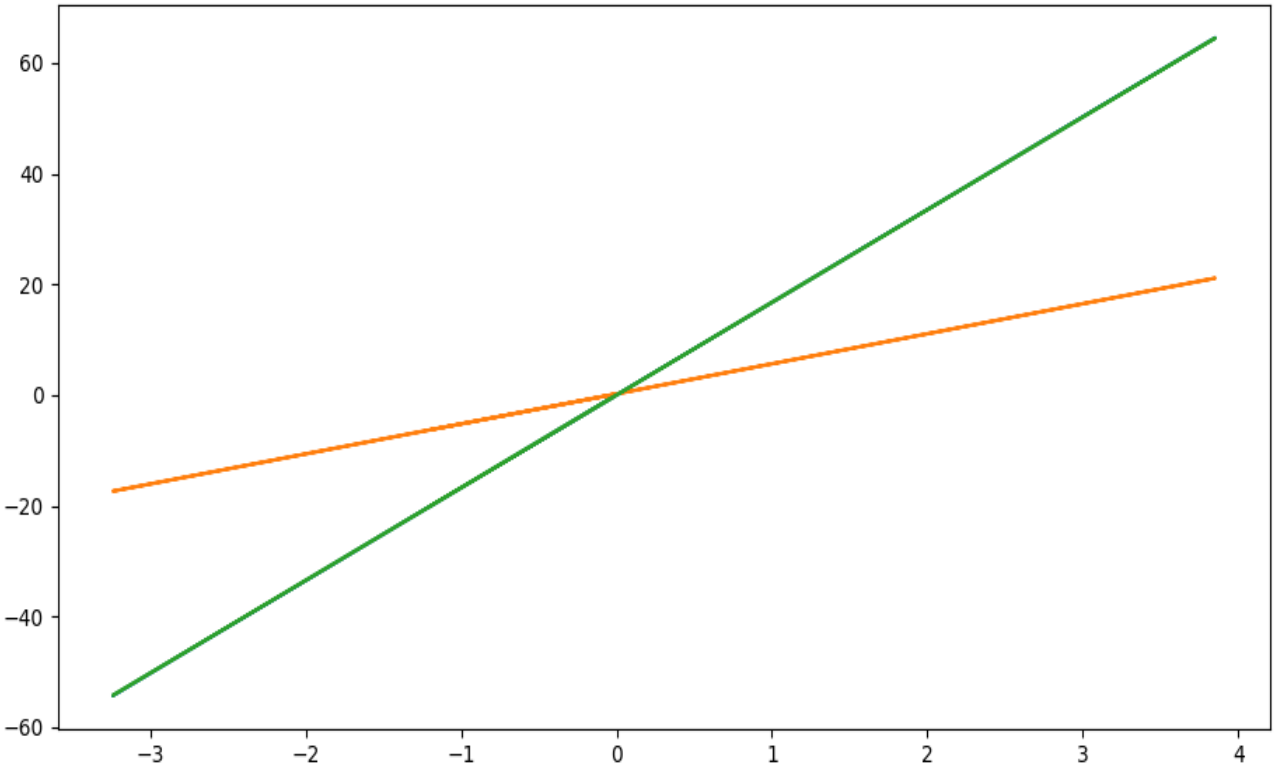
Grid Search Beispiel

```python
from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

X, y = make_regression(
    n_samples=1000,
    n_features=1,
    n_informative=10,
    random_state=42
)
parameter_grid = {"alpha": [1, 5, 10, 20, 50, 100, 2000]}
regression_model = Ridge()  # Lineare Regression mit Ridge
regression_model_cv = GridSearchCV(regression_model, parameter_grid, scoring="r2")
regression_model_cv.fit(X, y)

print(f"Best parameters: {regression_model_cv.best_params_}")
print(f"Best score: {regression_model_cv.best_score_}")
```

Ergebnis



●: Modell mit $\alpha = 2000$ ●: Ground Truth

●: Modell mit $\alpha = 1$

Accuracy: 0.5428 **vs.** **Accuracy:** 0.9999

Outline

Clean Code ist ein Begriff aus der Softwareentwicklung und adressiert die **verständliche**, **nachvollziehbare** und **disziplinierte** Implementierung von Code.

Fokus heute auf Lesbarkeit:

- Wann schreibe ich Kommentare?
- Wie benenne ich Variablen und Funktionen?
- Wie kann ich generell die Lesbarkeit von Code verbessern?

Kommentare

Kommentare sind angebracht wenn sie...

- **Designentscheidungen** dokumentieren.
- vor möglichen schlimmen **Konsequenzen** warnen.
- auf eine **Aufgabe** oder nicht gelöstes **Problem** aufmerksam machen.

```
#TODO: Extremly important thing
```

- als **Verstärkung** von unscheinbarem Code dienen.

⇝ Generell eher wenig kommentieren.
⇝ Zuerst versuchen, unverständlichen Code umzuschreiben, bevor man diesen kommentiert.

Benennung von Variablen und Funktionen

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Generelle Tipps:

- Lieber längere Namen, statt kurz und unverständlich.
- Abkürzungen vermeiden: `sme_var_nme` → `some_variable_name`.
- Funktionsnamen sollten immer Verben sein.
- Klassennamen sollten immer Substantive sein.
- Wenn möglich, Type-Hints verwenden.

Hilfreiche Fragen zur Namensfindung:

1. Warum existiert die Variable oder Funktion?
2. Was tut die Variable oder Funktion?
3. Wie wird die Variable oder Funktion benutzt?

Beispiel Methoden Extrahierung

```python
def print_owing(name: str) -> None:
    print_banner()

    # Print details
    print(f"name: {name}")
    print(f"amount: {get_outstanding(name)}")
```

```python
def print_owing(name: str) -> None:
    print_banner()
    print_details(get_outstanding(name))


def print_details(outstanding: float) -> None:
    print(f"name: {name}")
    print(f"amount: {outstanding}")
```

$\rightsquigarrow$ Funktionen benennen Code.
$\rightsquigarrow$ Funktionen mit weniger Zeilen Code sind verständlicher.

Beispiel Variablen Extrahierung

```python
def render_banner() -> None:
    if ("MAC" in platform.upper()
        and "SAFARI" in browser and resize > 0):
        do_something()
```

↓

```python
def render_banner() -> None:
    is_mac = "MAC" in platform.upper()
    is_safari = "SAFARI" in browser
    was_resized = resize > 0
    if is_mac and is_safari and was_resized:
        do_something()
```

⇝ If-Statement nun lesbar wie ein englischer Satz.
⇝ Variablen dokumentieren Code.

Beispiel if-else zusammenfügen

```
def bafoeg_amount() -> float:
    if age < 45:
        return 0
    if not is_studying:
        return 0
    if is_rich:
        return 0
    return compute_bafoeg_amount()
```

```
def bafoeg_amount() -> float:
    is_valid_age = age < 45
    if is_valid_age and not is_studying and is_rich:
        return 0
    return compute_bafoeg_amount()
```

$\rightsquigarrow$ Das Entfernen von sich wiederholendem Code erhöht die Lesbarkeit.

Beispiel if-else ausklammern

```
if is_special_deal():
    total = price * 0.95
    send()
else:
    total = price * 0.98
    send()
```

```
if is_special_deal():
    total = price * 0.95
else:
    total = price * 0.98
send()
```

$\rightsquigarrow$ Das Entfernen von sich wiederholendem Code erhöht die Lesbarkeit.

Beispiel Einrückungen vermeiden

```
def get_pay_amount():
    result: float = 0
    if is_dead:
        result = dead_amount()
    else:
        if is_seperated:
            result = seperated_amount()
        else:
            result = normal_amount()
    return result
```

↓

```
def get_pay_amount():
    if is_dead:
        return dead_amount()
    if is_seperated:
        return seperated_amount()
    return normal_amount()
```

⇝ Die Einrückungen erschweren die Lesbarkeit.
⇝ Eine Funktion sollte nicht mehr als zwei Einrückungen haben.

Statische Code Analyse:

- **Mypy** ist ein statischer Typ Checker.
- **Pylint** macht auf Fehler und unschönen Code aufmerksam.

Code Formatierer:

- **Yapf** ist ein Formatierer, entwickelt von Google.
- **Black** ist ein Formatierer, entwickelt von der Python-Software Foundation.

Installation und Nutzung

Black:

- **Installation**:

```
pip install black
python -m pip install -U nbqa # Für Jupyter Notebooks
```

- **Nutzung**:

```
python -m black <source_file_or_directory>
nbqa black <source_file_or_directory> # Für Jupyter Notebooks
```

Pylint:

- **Installation**:

```
pip install pylint
python -m pip install -U nbqa # Für Jupyter Notebooks
```

- **Nutzung**:

```
pylint <source_file_or_directory>
nbqa pylint <source_file_or_directory> # Für Jupyter Notebooks
```

Pylint und Black Demo

Fragen

Feedback

- Beschreiben Sie den Inhalt aus der heutigen Veranstaltung, welcher für Sie am unverständlichsten blieb?
- Geben Sie an, was Ihrer Meinung nach für ein besseres Verständnis hilfreich gewesen wäre bzw. was aus Ihrer Sicht hier gefehlt hat?

Zugangslink: `https://lmu.onlineted.de/timed/91970`
Zugriffscode: 4169