Data Mining und Maschinelles Lernen

Prof. Kristian Kersting
Steven Lang
Johannes Czech

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2021
26. Juli 2021
Übungsblatt 6

Diese Übung wird am 10.06.2021 um 13:30 Uhr besprochen und nicht bewertet.

Benötigte Dateien
Alle benötigten Datensätze und Skriptvorlagen finden Sie in unserem Moodle-Kurs:
https://moodle.informatik.tu-darmstadt.de/course/view.php?id=1058

6.1 Adaboost - Entscheidungsgrenzen

Das Kernprinzip von AdaBoost besteht darin, eine Sequenz schwacher Lerner (d. h. Modelle, die nur geringfügig besser sind als zufällige Vermutungen, wie z. B. flache Entscheidungsbäume) auf wiederholt modifizierte Versionen der Daten anzupassen. Die von allen schwachen Lernenden erhaltenen Vorhersagen werden dann durch eine gewichtete Mehrheitsentscheidung (oder Summe) kombiniert, um die endgültige Vorhersage zu erstellen. Die Datenmodifikationen bei jeder sogenannten Boosting-Iteration bestehen aus der Anwendung von Gewichten $w_1, w_2, \ldots, w_N$ zu jeder der N Trainingsproben. Zu Beginn werden diese Gewichte alle auf $w_i = 1/N$ gesetzt, so dass im ersten Schritt lediglich ein schwacher Lerner auf den Originaldaten trainiert wird. Für jede aufeinanderfolgende Iteration werden die Stichprobengewichte individuell modifiziert, und der Lernalgorithmus wird erneut auf die neu gewichteten Daten angewendet. In einem Schritt werden die Gewichte der Trainingsbeispiele, die durch das im vorherigen Schritt induzierte verstärkte Modell falsch vorhergesagt wurden, erhöht, während die Gewichte für diejenigen, die korrekt vorhergesagt wurden, verringert werden. Im Laufe der Iterationen erhalten schwer vorhersagbare Beispiele einen immer stärkeren Einfluss. Jeder nachfolgende schwache Lerner ist dadurch gezwungen, sich auf die Beispiele zu konzentrieren, die von den vorhergehenden in der Sequenz übersehen werden [1].

a) **Datensatzvisualisierung**

Beginnen wir mit einem künstlichen Datensatz, der aus zwei konzentrischen Kreisen besteht, die zwei Klassen repräsentieren:

Visualisieren Sie den Datensatz in der Methode `plot_dataset`. Die Punkt der ersten und zweiten Klasse sollen dabei farblich unterschieden werden und Trainings- und Testpunkte verschieden markiert werden.

```
def plot_dataset(X_train: np.ndarray, X_test: np.ndarray, y_train: np.ndarray, y_test: np.ndarray) -> None:
    """Creates as scatterplot for the given dataset. The points for the first class are blue and the points for the
     second class are red. Training points are displayed as dots and testpoints are described by an X.
    """
    plt.figure(figsize=(4, 4))
    plt.scatter(*X_train.T, c=y_train, cmap=ListedColormap(["#FF0000", "#0000FF"]), label="Training Points", marker="o")
    plt.scatter(*X_test.T, c=y_test, cmap=ListedColormap(["#FF0000", "#0000FF"]), label="Test Points", marker="x")

    plt.xlabel("$X_0$")
    plt.ylabel("$X_1$")
    plt.legend()
    plt.show()
```

b) **Basislerner**

Als Basislerner für das Boosting-Modell werden wir einen einfachen Entscheidungsbaum der Tiefe 1 (wegen seiner visuellen Flachheit auch Entscheidungsstumpf genannt) wählen. Um ein Gefühl für dessen Einfachheit zu bekommen, können wir den Stumpf auf die Daten selbst anpassen und die Entscheidungsgrenze visualisieren.

Visualisieren Sie die Entscheidungsgrenze unseres Basislerners in der Methode `plot_decision_boundary_stump`.

```python
def plot_decision_boundary_stump(stump: DecisionTreeClassifier, X: np.ndarray, y: np.ndarray, N=1000)
     -> None:
    """Plot the decision boundary for a tree stump and scatters plot of the training data"""
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, N), np.linspace(y_min, y_max, N))

    plt.figure(figsize=(4, 4))
    plt.title(f"Decision Boundary of a Decision Stump")
    zz = stump.predict_proba(np.c_[xx.ravel(), yy.ravel()])
    Z = zz[:, 1].reshape(xx.shape)
    cm_bright = ListedColormap(["#FF0000", "#0000FF"])
    # Get current axis and plot
    ax = plt.gca()
    ax.contourf(xx, yy, Z, 2, cmap="RdBu", alpha=0.5)
    ax.contour(xx, yy, Z, 2, cmap="RdBu")
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cm_bright)
    ax.set_xlabel("$X_0$")
    ax.set_ylabel("$X_1$")
    plt.tight_layout()
```

c) **Algorithmus Beschreibung**

Es wird deutlich, dass ein Entscheidungsstumpf allein nicht in der Lage ist, die beiden Klassen voneinander zu trennen. Da der Stumpf die Tiefe 1 hat, kann er die Daten nur auf der Basis eines **einzigen** Merkmals trennen, das in diesem Fall etwa $X_1 == 0.5$ beträgt. Jeder Datenpunkt mit $X_1 \leq 0.5$ wird als blaue Klasse vorhergesagt, während alles mit $X_2 > 0.55$ als rote Klasse vorhergesagt wird.

Daher werden wir nun den AdaBoost-Algorithmus verwenden, um diesen einfachen Basislerner wie oben beschrieben zu verbessern.

Beschreiben Sie in eigenen Worten das Verfahren des Adaboost-Algorithmuses anhand Abbildung 1.

Wir haben nun einen AdaBoost-Klassifikator trainiert, wobei wir einen Entscheidungsstumpf der Tiefe 1 als Basis für vier Iterationen verwendet haben. Das heißt, jeder aufeinanderfolgende Entscheidungsstumpf konzentriert sich darauf, was das Ensemble der vorhergehenden Stümpfe falsch gemacht hat.

Für jede verstärkende Iteration: - Links: Darstellung der Entscheidungsstumpf-Entscheidungsgrenze sowie der Stichprobengewichte als Größe des Datenpunkts (mit Angabe, auf welche Stichproben sich der aktuelle Stumpf konzentriert) - Rechts: Ensemble-Entscheidungsgrenze, die die Kombination aller Stumpfentscheidungen bis zu den aktuellen Iterationen darstellt

Wir können sehen, dass das Ensemble bei jeder Iteration den Proben, die es zuvor falsch klassifiziert hat, ein größeres Gewicht beimisst. Entsprechend den gewichteten Proben konzentriert sich der Entscheidungsstumpf der nächsten Iteration dann darauf, diese Proben mit größerem Gewicht korrekt zu erhalten. Die Verstärkung konstruiert daher Experten für eine Teilmenge von Stichproben, die das vorherige Ensemble falsch klassifiziert hat.

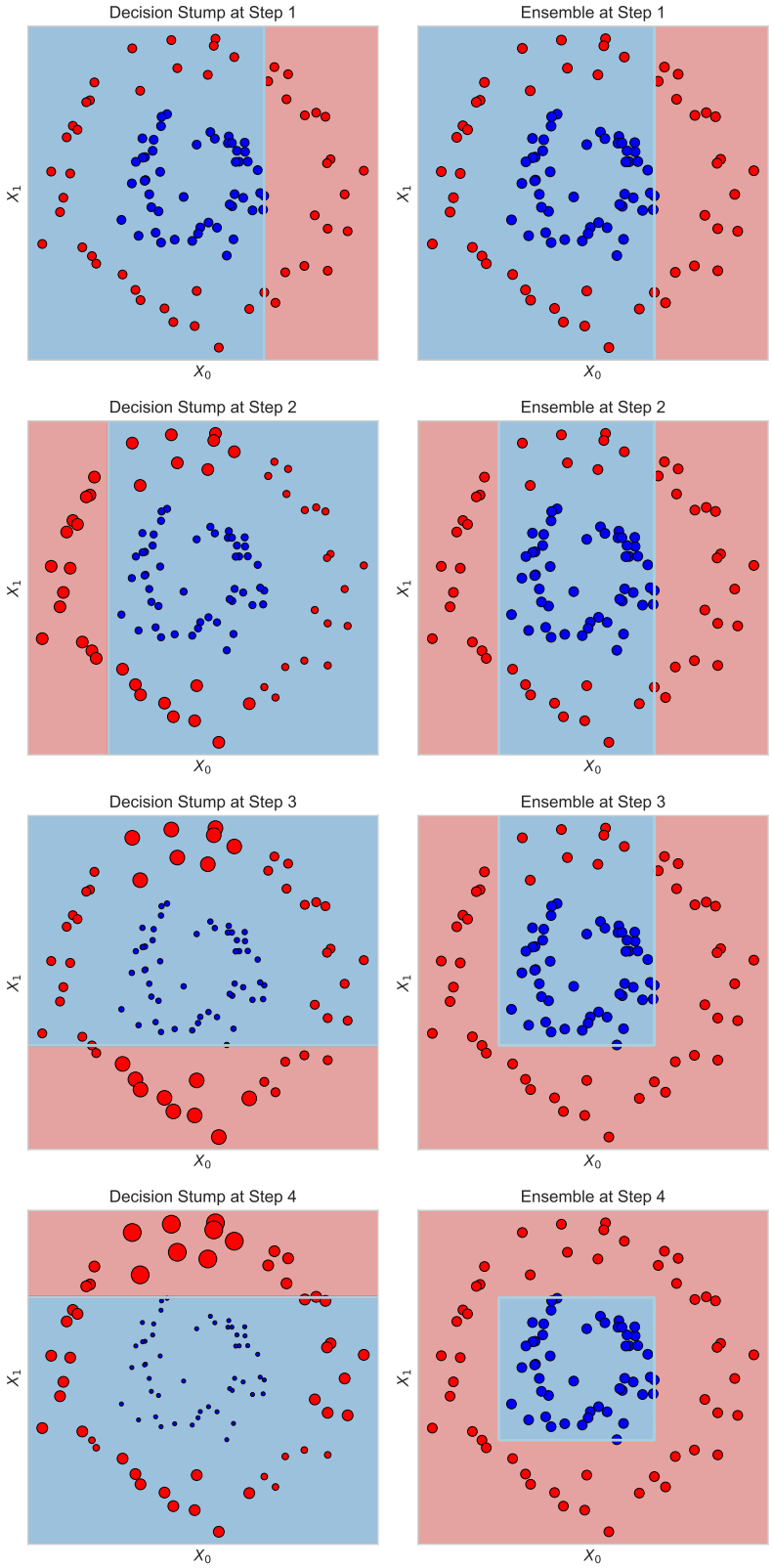

Abbildung 1: Iteration von Adaboost mit Entscheidungsstümpfen.

6.2 Adaboost - Klassifikation

Schauen wir uns ein etwas komplexeres Beispiel an: Das Klassifizieren von Ziffern. Der Datensatz `sklearn.datasets.load_digits` enthält 1797 verschiedene 8×8 Pixel-Bilder von Ziffern.

a) **Datensatzvisualisierung**

Visualisieren Sie die ersten fünf Trainingsbeispiele mit ihrem zugehörigen Label in der Methode `plot_digit_dataset`.

```python
def plot_digit_dataset(X_train: np.ndarray, y_train: np.ndarray) -> None:
    """Plots the first 5 images of the digit dataset."""
    plt.figure()
    for i, (x_i, y_i) in enumerate(zip(X_train[:5], y_train[:5]), start=1):
        plt.subplot(150 + i)
        plt.imshow(x_i.reshape(8, 8), cmap="gray")
        plt.title("label = " + str(y_i))
        plt.axis("off")
    plt.show()
```

b) **Iterationen / Anzahl Klassifier**

Wir werden das AdaBoost-Modell erneut mit einem Entscheidungsstumpf als Basis-Lerner für 300 Iterationen lernen. Für jede Iteration wird ein zusätzlicher Entscheidungsstumpf eingesetzt.

Sklearn ermöglicht es uns über wiederholte Anwendung der Methode `AdaBoostClassifier.staged_predict` stufenweise Vorhersagen zu treffen. Dies erzeugt Vorhersagen nach jeder Boosting-Iteration und erlaubt es uns, die Genauigkeitskurve über die Trainingsiterationen zu visualisieren.

Fitten Sie ein `AdaBoostClassifier` über die Methode `get_acc_for_estimators` und geben Sie das Modell und die zugehörigen Trainings- und Testgenauigkeiten für jede Iteration zurück.

Wie verhält sich die Performanz mit ansteigender Anzahl von Iterationen?

```python
def get_acc_for_estimators(base_estimator: DecisionTreeClassifier, X_train: np.ndarray,
                           X_test: np.ndarray, y_train: np.ndarray, y_test: np.ndarray,
                           n_estimators: int, learning_rate: float):
    """Fits an AdaBoostClassifier on the given training data and returns the training and test
    accuracy for all
     iterations."""
    # Create AdaBoost model
    model = AdaBoostClassifier(base_estimator=base_estimator, n_estimators=n_estimators,
    learning_rate=learning_rate)
    # Fit model
    model.fit(X_train, y_train)
    # Evaluate model on train/test set
    accuracy_train = [
        accuracy_score(y_pred, y_train) for y_pred in model.staged_predict(X_train)
    ]
    accuracy_test = [
        accuracy_score(y_pred, y_test) for y_pred in model.staged_predict(X_test)
    ]
    return model, accuracy_train, accuracy_test
```

Wir können beobachten, dass mit einer wachsenden Zahl von Entscheidungs-Stumpfen, das AdaBoost-Modell im Allgemeinen besser abschneidet.

c) **Lernrate**

Es ist wichtig zu beachten, dass die Steilheit und die tatsächliche Verbesserung auch von der Lernrate abhängt, d.h. der Rate, mit der jeder zusätzliche Basislernende zur Vorhersage beiträgt. Um zu sehen, wie viel Einfluss die Lernrate hat, können wir das Diagramm aus b) für verschiedene Lernraten visualisieren: Fitten Sie `AdaBoostClassifier` über Ihre zuvor implementierte Funktion `get_acc_for_estimators` für die Lernraten 2^i wobei $i \in \{-3, -2, -1, 0, 1\}$ und geben Sie die Trainingsgenauigkeiten für jede Iteration zurück.

Welche Schlussfolgerung können Sie aus dem Plot zum Vergleich der Lernraten ziehen?

Dies zeigt, dass die Wahl der Lernrate von AdaBoosts wichtig ist. Um einen guten Wert für die Lernrate zu finden, ist es üblich, eine Kreuzvalidierung über eine Reihe von möglichen Lernratenwerten durchzuführen:

6.3 Gradient Boosting

Gradient Tree Boosting oder gradientenverstärkte Regressions Bäume (GBRT) ist eine Verallgemeinerung des Boosting zu willkürlich differenzierbaren Verlust-Funktionen. GBRT ist ein präzises und effektives Verfahren, das sowohl für Regressions- als auch für Klassifikationsprobleme verwendet werden kann. GBRT Modelle werden in einer Vielzahl von Bereichen eingesetzt, darunter Web-Suchrangfolge und Ökologie.

Die Vorteile des GBRT sind:

- Natürliche Handhabung von Daten gemischten Typs (= heterogene Merkmale)
- Gute Vorhersagekraft
- Robustheit gegenüber Ausreißern im Zielvariablenraum (durch robuste Verlustfunktionen)

Die Nachteile des GBRT sind:

- Skalierbarkeit, aufgrund der sequentiellen Natur des Boostings kann es kaum parallelisiert werden.

Algorithmus

1. Trainieren Sie einen Entscheidungsstumpf (Entscheidungsbaum mit einer Tiefe von 1) T_1 auf den Daten. Dies führt zu unserem ersten Modell $M_1 = T_1$. Die Schwachstellen sind die Residuen $r = y_{true} - y_{pred}$.
2. Trainieren Sie einen weiteren Entscheidungsstumpf T_2 auf den Residuen. Dies führt zu unserem zweiten Modell $M_2 = M_1 + \gamma_2 T_2$, wobei γ die Schrittweite ist. Als Regularisierung können wir eine Lernrate $\nu \in (0, 1)$ (manchmal als Schrumpfung bezeichnet) einführen und erhalten $M_2 = M_1 + \nu\gamma_2 T_2$.
3. Wiederholen Sie 2. und trainieren Sie weitere Entscheidungsstümpfe auf den Residuen von M_2, dann M_3, bis eine gute Anpassung an die Zielvariable erreicht ist

a) **Regression**

Beginnen wir mit dem Lernen einer einfachen Sinuskurve.

Mit dem Modul `GradientBoostingRegressor` von `sklearn` können wir ein GradientBoosting-Regressionsmodell an die erzeugte Sinuskurve anpassen. Wir werden die Residuen für insgesamt 300 Schritte anpassen und die Funktion der kleinsten Quadrate als Fehlerfunktion verwenden.

Wenden Sie einen Gradient Boosting Regressor in der Funktion `fit_gradient_boosting_regressor` an und geben Sie das gelernte Model und die Trainingsfehler für jede Iteration zurück.

```
def fit_gradient_boosting_regressor(X: np.ndarray, n_samples: int, y: np.ndarray,
                                    learning_rate: float, max_depth: int, random_state: int, loss:
    str):
    """Fits gradient tree boosting regressor and returns the learned model and training errors."""
    gb = GradientBoostingRegressor(
        n_estimators=n_samples, learning_rate=learning_rate, max_depth=max_depth, random_state=
    random_state, loss=loss
    )
    gb.fit(X, y)
    # Obtain loss after each training iteration
    train_errors = gb.train_score_
    return gb, train_errors
```

b) **Konzept des Algorithmus**

Um den iterativen Fortschritt des Modells auf den Trainingsdaten zu visualisieren, ist es möglich, den Fortschritt nach jeder Iteration zu erhalten (s. Abb. 2).

Beschreiben Sie das Konzept des Algorithmus von Gradient Boosting.

Nach jeder Iteration verbessert sich das Modell und macht einen Schritt in die richtige Richtung, ohne über die Lösung hinauszugehen. Da wir mit einer einfachen Sinuskurve als Datensatz begonnen haben, sehen wir wie schrittweise die Residuen verringert werden.

Konzept

- Erstellen eines additiven Modells $f(x) = \sum_{m=1}^{M} \alpha_m f_m(x)$ *(Ensemble) Schritt für Schritt von* $1 \ldots M$.

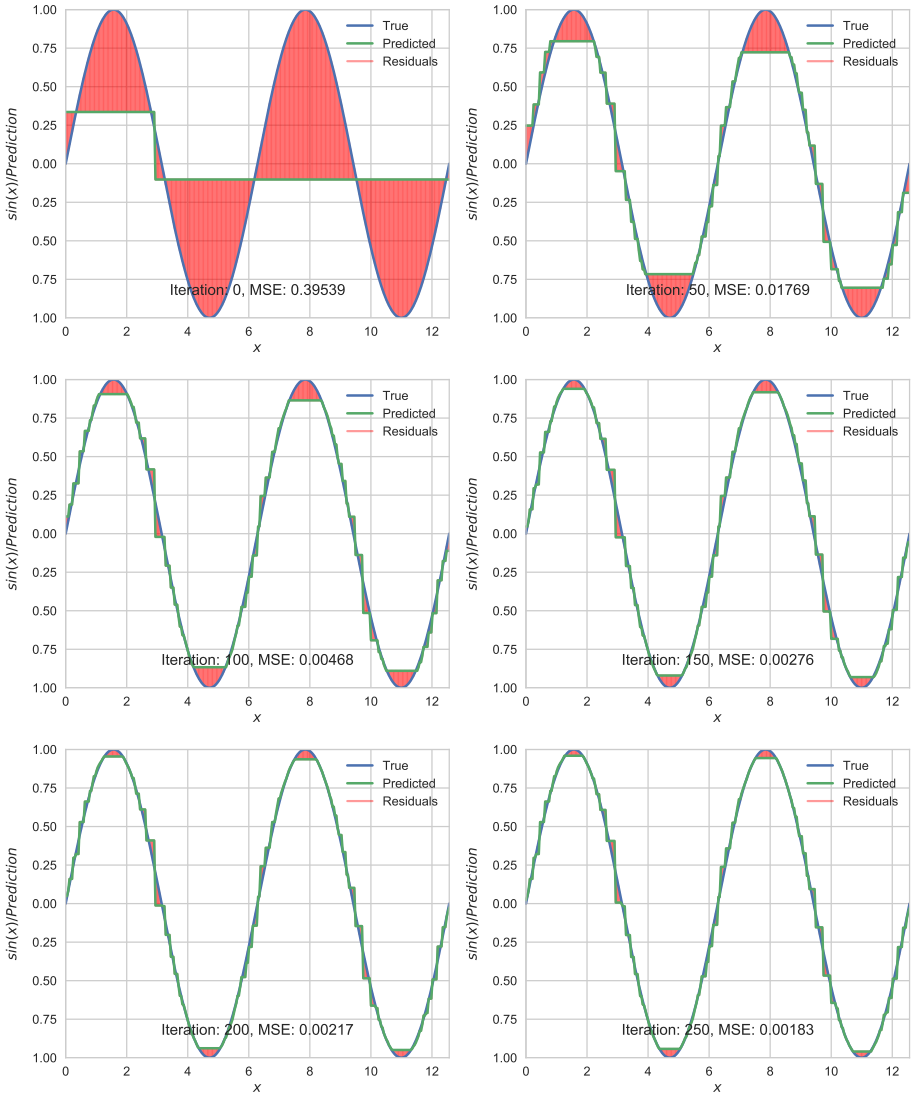

Abbildung 2: Iterationen von Gradient Boost mit Entscheidungsstümpfen.

- In jeder Iteration trainieren wir einen neuen schwachen Lerner, der versucht, die Schwachstellen des vorherigen Ensembles auszusortieren.

- Nutzen der Gradienten der Verlustfunktion, um insbesondere den neuen Lerner zu verbessern

6.4 Gradient Boosting - Fehlerfunktionen

Die folgenden Fehlerfunktionen werden für Gradient Boosting unterstützt [1] und können in `sklearn` mit dem Parameter 'loss' spezifiziert werden:

Regression

- Kleinste Quadrate (engl. Least Squares) ('ls'): Die natürliche Wahl für die Regression aufgrund ihrer überlegenen rechnerischen Eigenschaften. Das Ausgangsmodell ist durch den Mittelwert der Zielwerte gegeben.
- Geringste absolute Abweichung ('lad'): Eine robuste Verlustfunktion für die Regression. Das Anfangsmodell ist durch den Median der Zielwerte gegeben.
- Huber ('huber'): Eine robuste Verlustfunktion für die Regression: Eine weitere robuste Verlustfunktion, die kleinste Quadrate und geringste absolute Abweichung kombiniert; verwenden Sie Alpha, um die Sensitivität in Bezug auf Ausreißer zu kontrollieren (siehe [F2001] für weitere Details).
- Quantil ('quantil'): Eine Verlustfunktion für die Quantil-Regression. Verwenden Sie $0 <$ alpha < 1, um das Quantil zu spezifizieren. Diese Verlustfunktion kann verwendet werden, um Prädiktionsintervalle zu erstellen.

Klassifikation

- Binomische Abweichung ('deviance'): Die negative Binomial-Log-Wahrscheinlichkeitsverlust-Funktion für binäre Klassifikation (liefert Wahrscheinlichkeitsschätzungen). Das anfängliche Modell ist durch das Log-Wahrscheinlichkeits-Verhältnis gegeben.
- Multinomiale Abweichung ('deviance'): Die negative binomiale logarithmische Wahrscheinlichkeitsverlust-Funktion für binäre Klassifikation (liefert Wahrscheinlichkeitsschätzungen): Die negative multinomiale logarithmische Wahrscheinlichkeitsverlustfunktion für die Mehrklassenklassifikation mit *n_classes* sich gegenseitig ausschließender Klassen. Sie liefert Wahrscheinlichkeitsschätzungen. Das anfängliche Modell ist durch die vorherige Wahrscheinlichkeit jeder Klasse gegeben. Bei jeder Iteration müssen n_Klassen-Regressionsbäume konstruiert werden, was die GBRT für Datensätze mit einer großen Anzahl von Klassen ziemlich ineffizient macht.
- Exponentieller Verlust ('exponential'): Dieselbe Verlustfunktion wie AdaBoostClassifier. Weniger robust gegenüber falsch vorhergesagten Datenpunkte als 'deviance'; kann nur für binäre Klassifikation verwendet werden.

Regularisierung

[2] schlug eine einfache Regularisierungsstrategie vor, die den Beitrag jedes schwachen Lerners um einen Faktor ν skaliert:

$$F_m(x) = F_{m-1}(x) + \nu \gamma_m h_m(x)$$

a) **Fehlerfunktionen**

Evaluieren Sie die Performanz von Gradient Boosting in der Funktion `evaluate_loss_fn` unter Nutzung der verschiedenen, oben genannten Fehlerfunktionen und geben Sie jeweils die mittleren Fehlerquadrate (MSE) für den Trainings- und Testdatensatz zurück.

```python
def evaluate_loss_fn(X_train: np.ndarray, y_train: np.ndarray,
                     X_test: np.ndarray, y_test: np.ndarray,
                     loss_fn="ls", learning_rate=0.5) -> np.ndarray:
    """Fits a GradientBoostingRegressor on the training data and returns the training and test mse
    for all iterations"""
    gb = GradientBoostingRegressor(
        loss=loss_fn,
        learning_rate=learning_rate,
        n_estimators=200,
        random_state=0,
    )
    gb.fit(X_train, y_train)

    # Collect train/test mse
    mses = []
    for y_train_pred, y_test_pred in zip(
```

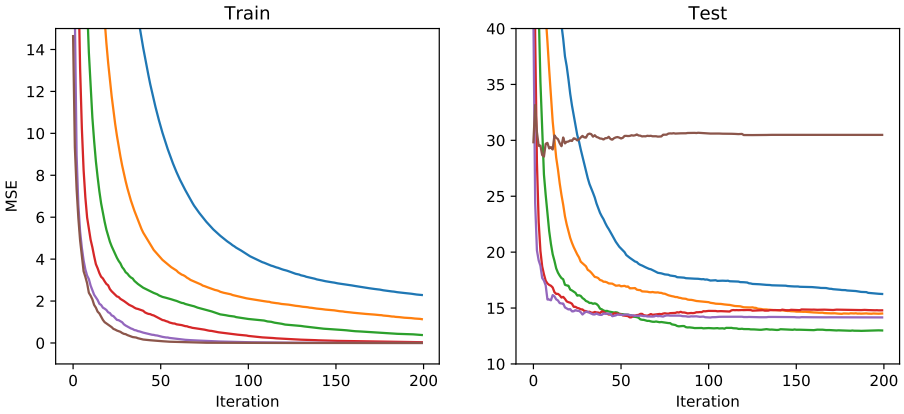

Abbildung 3: Performanz von Gradient Boosting mittels MSE unter Anwendung unterschiedlicher Lernraten auf dem Bosten Datensatz.

```
        gb.staged_predict(X_train), gb.staged_predict(X_test)
    ):
        mse_train = mse(y_train, y_train_pred)
        mse_test = mse(y_test, y_test_pred)
        mses.append([mse_train, mse_test])
    return np.array(mses)
```

b) **Lernrate**

In folgender Abbildung 3 wird eine Auswertung des Bostoner Datensatzes für Train- und Test-MSE unter Verwendung von Lernraten $lr \in \{2^{-5}, \ldots, 2^{0}\}$ gezeigt. Beurteilen Sie wie sich unterschiedliche Lernraten auswirken.

Der Parameter `learning_rate` *interagiert stark mit dem Parameter* `n_estimators`*, der die Anzahl der schwachen Lerner anpasst. Kleinere Werte von* `learning_rate` *erfordern eine größere Anzahl von schwachen Lernenden, um einen konstanten Trainingsfehler aufrechtzuerhalten. Empirische Belege deuten darauf hin, dass kleine Werte der* `learning_rate` *bessere Testfehler begünstigen. [3] empfehlen, die Lernrate auf eine kleine Konstante zu setzen (z.B.* `learning_rate` *<= 0.1) und* `n_estimators` *durch frühes Anhalten (engl. „Early Stopping“) zu wählen.*

-

Literatur

[1] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[2] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.