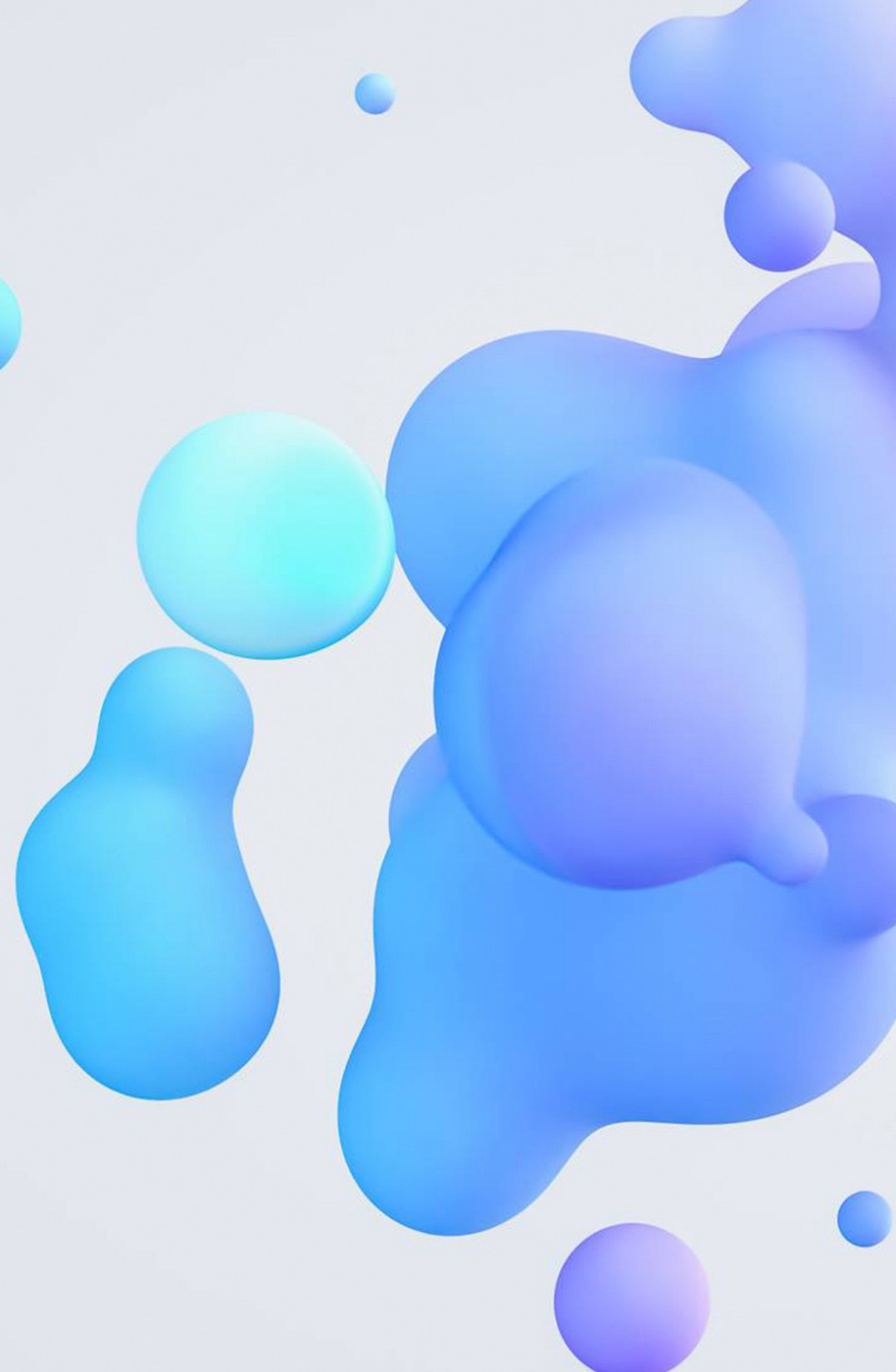
Flexibilisierung von Local Interpretable Model-Agnostic Explanations (LIME) mithilfe von Generalized Additive Models (GAMs)



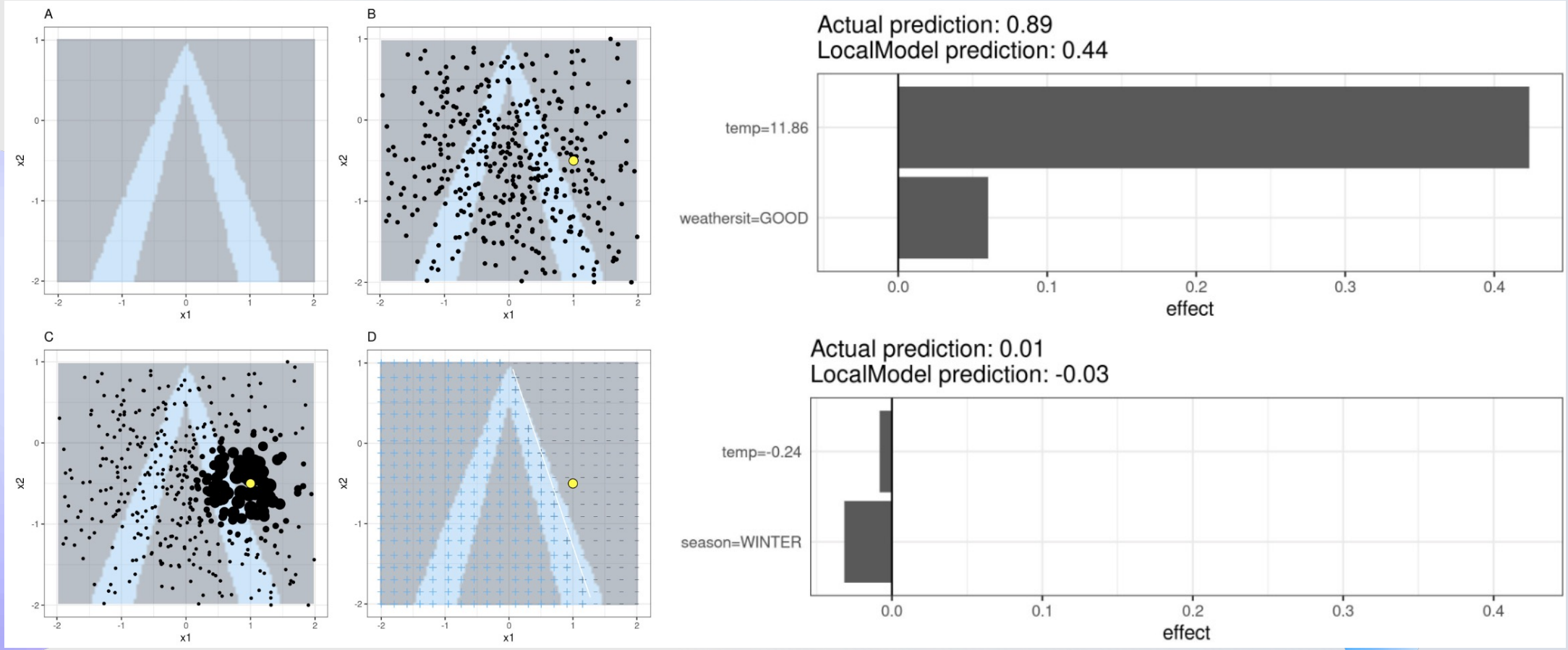
Gliederung

1. MOTIVATION
2. GRUNDLAGEN & RELATED WORK
3. KONZEPTIONELLE VORBETRACHTUNG ZU BESTEHENDEN ANSÄTZEN
4. TECHNISCHE REALISIERUNG
5. BEWERTUNG DES ANSATZES
6. KRITISCHE WÜRDIGUNG & AUSBLICK
7. FAZIT

1. Motivation

- mangelnde Interpretierbarkeit von ML-Modellen
  - ➢ geringere Akzeptanz und Vertrauen
  - ➢ Lösung: XAI
- LIME (XAI-Methode)
  - ➢ Entscheidungen individueller Instanzen lokal erklären
- GAMs (mathematischer Ansatz)
  - ➢ nichtlineare Funktionen erzeugen

2. Grundlagen & Related Work
LIME



https://christophm.github.io/interpretable-ml-book/lime.html

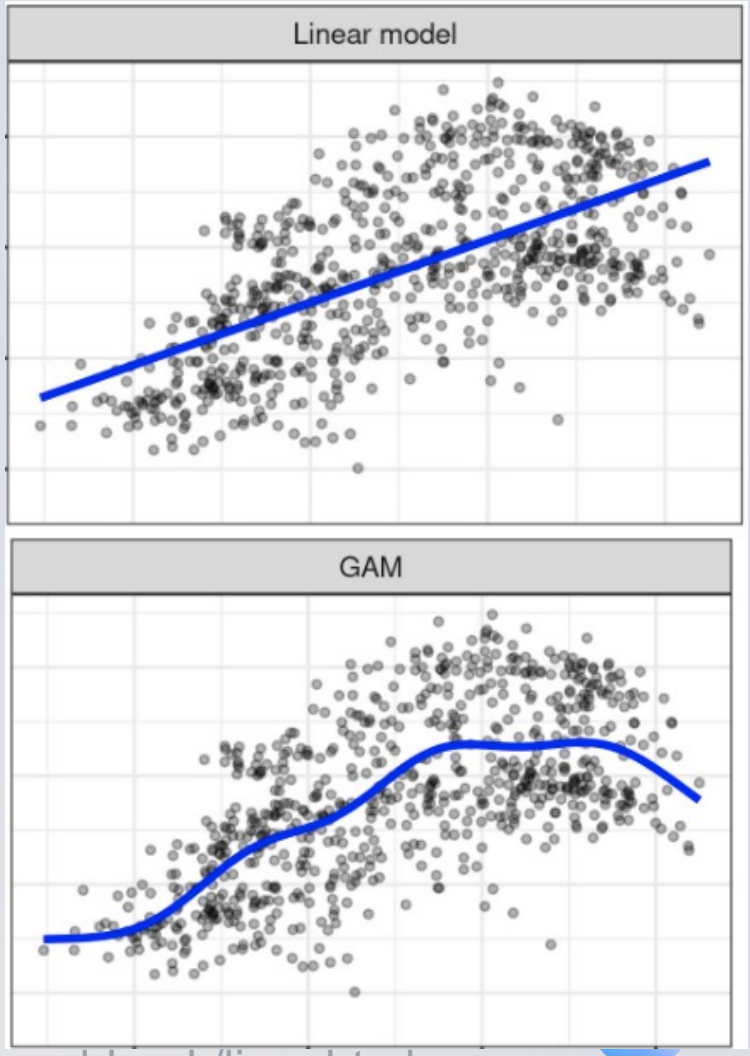
2. Grundlagen & Related Work
GAMs

- multiples lineares Modell:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon.$$

- generalisiertes additives Modell:

$$y = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_p(x_p) + \epsilon.$$



https://christophm.github.io/interpretable-ml-book/lime.html

2. Grundlagen & Related Work

BMB-LIME: LIME with modeling local nonlinearity and uncertainty in explainability, Yu-Hsin Hung & Chia-Yen Lee, 2024

- Integration Multivariaten Adaptiven Regressions-Splines (MARS) und Bootstrap-Aggregation in LIME

➢ nichtlineare Entscheidungsgrenzen präziser zu erfassen

- bayesianisches Framework

➢ quantifiziert Unsicherheiten in den Beiträgen der Features

Ergebnis: BMB-LIME **genauer** und **stabiler** als LIME

3. Konzeptionelle Vorbetrachtung zu bestehenden Ansätzen

- **Nachteil** LIME
  - ➢ Beschränkung auf lineare Modelle als lokaler Erklärer
    - ➢ Ungenaue und vereinfachte Erklärungen
- **Vorteile** GAMs
  - ➢ Merkmale durch eigene, potenziell nichtlineare Funktionen transformieren
  - ➢ bleiben interpretierbar, da isolierte Betrachtung Effekte einzelner Merkmale

3. Konzeptionelle Vorbetrachtung zu bestehenden Ansätzen

- **Nachteil** GAMs
  - ➢ Balance zwischen Flexibilität (präzise Abbildung Datenmuster) und Komplexität (Überanpassung)
- **Idee**: GAMs (pygam) als lokales Modell innerhalb von LIME
- Voraussetzung:
  - ➢ GAM verarbeitet Eingangsdaten korrekt und generiert Ausgabedaten, die von LIME genutzt werden können

4. Technische Realisierung

- Lokales Modell standardmäßig:
  - **Ridge Regression**:
    Lineare Regression,
    Regularisierungskomponente α = 1,
    um Überanpassung („Overfitting“)
    zu verhindern, indem Strafe für
    große Regressionskoeffizienten
- Funktion:
  - Eingangsdaten: erzeugte Datenpunkte, deren Gewichte und vorhergesagten Werte durch das Black-Box-Modell
  - Ausgabe: Erklärung

```
def explain_instance_with_data(self,
                               neighborhood_data,
                               neighborhood_labels,
                               distances,
                               label,
                               num_features,
                               feature_selection='auto',
                               model_regressor=None):
    weights = self.kernel_fn(distances)
    labels_column = neighborhood_labels[:, label]
    used_features = self.feature_selection(neighborhood_data,
                                           labels_column,
                                           weights,
                                           num_features,
                                           feature_selection)
    if model_regressor is None:
        model_regressor = Ridge(alpha=1, fit_intercept=True,
                                random_state=self.random_state)
    easy_model = model_regressor
    easy_model.fit(neighborhood_data[:, used_features],
                   labels_column, sample_weight=weights)
    prediction_score = easy_model.score(neighborhood_data[:,
used_features],
                                        labels_column,
sample_weight=weights)

    local_pred = easy_model.predict(
        neighborhood_data[0, used_features].reshape(1, -1))

    if self.verbose:
        print('Intercept', easy_model.intercept_)
        print('Prediction_local', local_pred, )
        print('Right:', neighborhood_labels[0, label])
    return (easy_model.intercept_,
            sorted(zip(used_features, easy_model.coef_),
                   key=lambda x: np.abs(x[1]), reverse=True),
            prediction_score, local_pred)
```

Listing 4.1: Funktion mit *Ridge Regression*

```python
def explain_instance_with_data(self,
                               neighborhood_data,
                               neighborhood_labels,
                               distances,
                               label,
                               num_features,
                               feature_selection='auto',
                               model_regressor=None):
    weights = self.kernel_fn(distances)
    labels_column = neighborhood_labels[:, label]
    used_features = self.feature_selection(neighborhood_data,
                                           labels_column,
                                           weights,
                                           num_features,
                                           feature_selection)
    if model_regressor is None:
        model_regressor = Ridge(alpha=1, fit_intercept=True,
                                random_state=self.random_state)
    easy_model = model_regressor
    easy_model.fit(neighborhood_data[:, used_features],
                   labels_column, sample_weight=weights)
    prediction_score = easy_model.score(neighborhood_data[:,
   used_features],
                                        labels_column,
   sample_weight=weights)

    local_pred = easy_model.predict(
        neighborhood_data[0, used_features].reshape(1, -1))

    if self.verbose:
        print('Intercept', easy_model.intercept_)
        print('Prediction_local', local_pred, )
        print('Right:', neighborhood_labels[0, label])
    return (easy_model.intercept_,
            sorted(zip(used_features, easy_model.coef_),
                   key=lambda x: np.abs(x[1]), reverse=True),
            prediction_score, local_pred)
```

Listing 4.1: Funktion mit *Ridge Regression*

- fit
- score
- predict
- intercept_
- coef_

```python
def explain_instance_with_data(self,
                               neighborhood_data,
                               neighborhood_labels,
                               distances,
                               label,
                               num_features,
                               feature_selection='auto',
                               model_regressor=None):
    weights = self.kernel_fn(distances)
    labels_column = neighborhood_labels[:, label]
    used_features = self.feature_selection(neighborhood_data,
                                           labels_column,
                                           weights,
                                           num_features,
                                           feature_selection)
    if model_regressor is None:
        model_regressor = LinearGAM(fit_intercept=True)
    easy_model = model_regressor
    easy_model.gridsearch(neighborhood_data[:, used_features],
                          labels_column,
                          weights=weights)
    prediction_score = easy_model.score(
        neighborhood_data[:, used_features],
        labels_column, weights=weights)
    feature_contributions = []
    for i in used_features:
        pdeps, conf_intervals = easy_model.partial_dependence(
            term=int(i), width=0.95)
        X_grid = easy_model.generate_X_grid(term=i)[:, i]
        dpdeps_dx = np.gradient(pdeps, X_grid)
        closest_idx = np.argmin(np.abs(X_grid - 0.5))
        i_derivative_value = dpdeps_dx[closest_idx]

        #Visualisierung der partiellen Abhaengigkeitsfunktionen
        ...

        feature_contributions.append((int(i), i_derivative_value)
    )
    local_pred = easy_model.predict(
        neighborhood_data[0, used_features].reshape(1, -1))
    if self.verbose:
        print('Intercept', easy_model.coef_[0])
        print('Prediction_local', local_pred, )
        print('Right:', neighborhood_labels[0, label])
    return (easy_model.coef_[0],
            sorted(feature_contributions,
                   key=lambda x: np.abs(x[1]), reverse=True),
            prediction_score, local_pred)
```

Listing 4.2: Funktion mit *LinearGAM*

```python
def explain_instance_with_data(self,
                               neighborhood_data,
                               neighborhood_labels,
                               distances,
                               label,
                               num_features,
                               feature_selection='auto',
                               model_regressor=None):
    weights = self.kernel_fn(distances)
    labels_column = neighborhood_labels[:, label]
    used_features = self.feature_selection(neighborhood_data,
                                           labels_column,
                                           weights,
                                           num_features,
                                           feature_selection)

    if model_regressor is None:
        model_regressor = Ridge(alpha=1, fit_intercept=True,
                                random_state=self.random_state)
    easy_model = model_regressor
    easy_model.fit(neighborhood_data[:, used_features],
                   labels_column, sample_weight=weights)
    prediction_score = easy_model.score(neighborhood_data[:,
   used_features],
                                        labels_column,
   sample_weight=weights)

    local_pred = easy_model.predict(
        neighborhood_data[0, used_features].reshape(1, -1))

    if self.verbose:
        print('Intercept', easy_model.intercept_)
        print('Prediction_local', local_pred, )
        print('Right:', neighborhood_labels[0, label])
    return (easy_model.intercept_,
            sorted(zip(used_features, easy_model.coef_),
                   key=lambda x: np.abs(x[1]), reverse=True),
            prediction_score, local_pred)
```

Listing 4.1: Funktion mit *Ridge Regression*

```python
def explain_instance_with_data(self,
                               neighborhood_data,
                               neighborhood_labels,
                               distances,
                               label,
                               num_features,
                               feature_selection='auto',
                               model_regressor=None):
    weights = self.kernel_fn(distances)
    labels_column = neighborhood_labels[:, label]
    used_features = self.feature_selection(neighborhood_data,
                                           labels_column,
                                           weights,
                                           num_features,
                                           feature_selection)
```

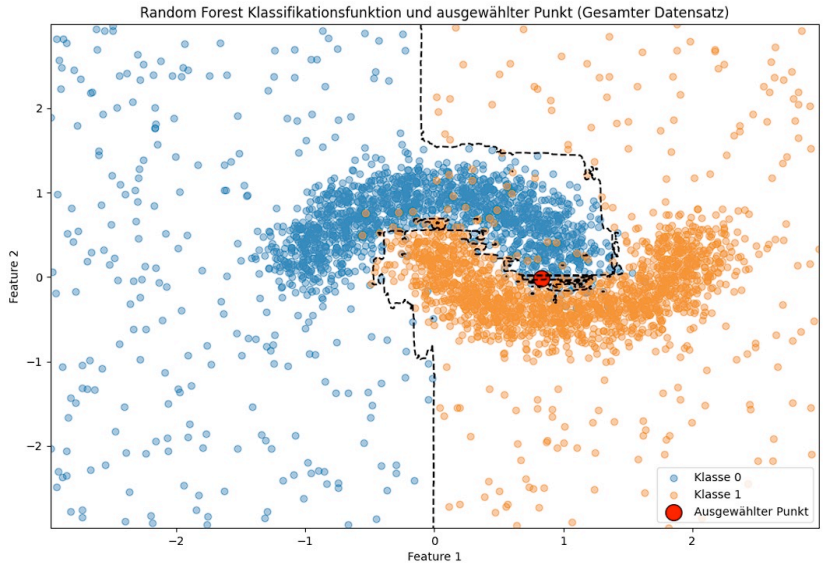
```python
def explain_instance_with_data(...):
    ...
    for i in used_features:
        ...
        #Visualisierung der partiellen Abhaengigkeitsfunktionen
        conf_intervals_lower = conf_intervals[:, 0]
        conf_intervals_upper = conf_intervals[:, 1]
        plt.figure(figsize=(10, 6))
        plt.plot(X_grid, pdeps, label='Partial Dependence')
        plt.fill_between(X_grid,conf_intervals_lower,
                         conf_intervals_upper, alpha=0.2,
                         color='orange', label='95% Confidence
   Interval')
        plt.xlabel(f'Feature {i + 1}')
        plt.ylabel('Partial Dependence')
        plt.title(f'Partial Dependence Plot with Confidence '
                  f'Interval for Feature {i + 1}')
        plt.legend()
        plt.show()
        ...
    ...
```

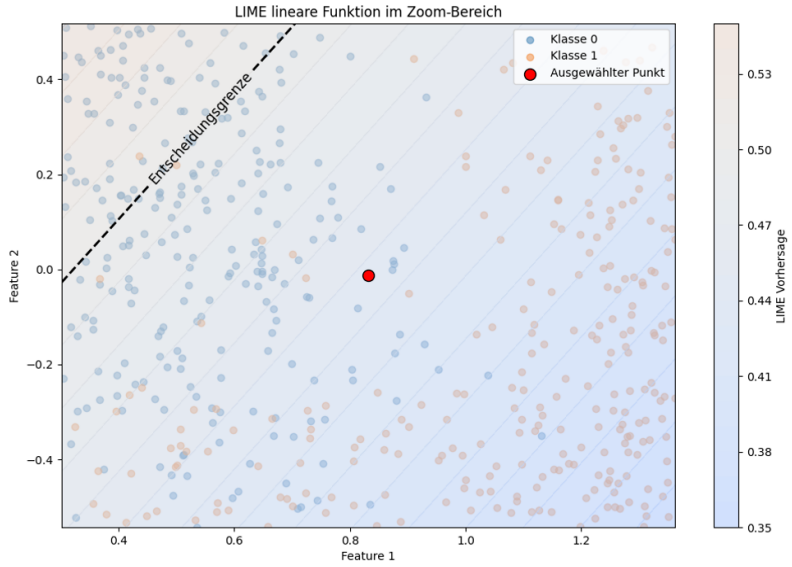
Listing 4.3: Visualisierung der partiellen Abhängigkeitsfunktionen

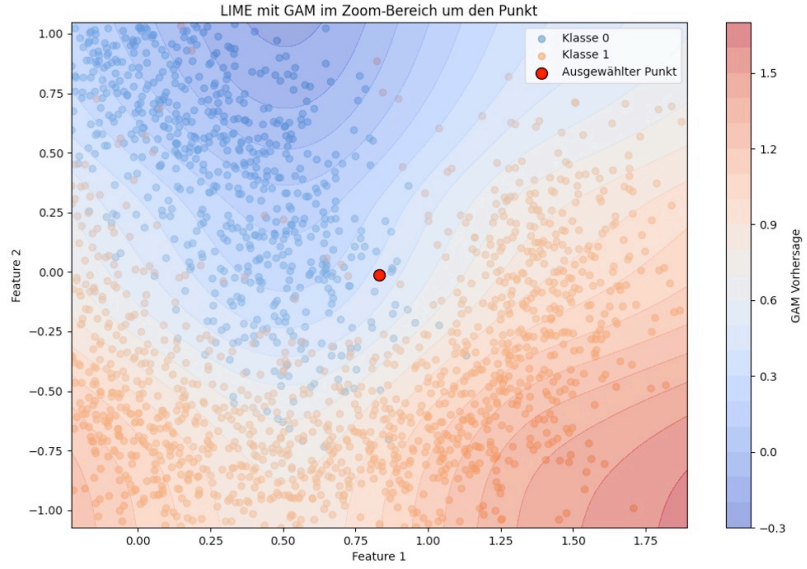
```python
            sorted(feature_contributions,
                   key=lambda x: np.abs(x[1]), reverse=True),
            prediction_score, local_pred)
```

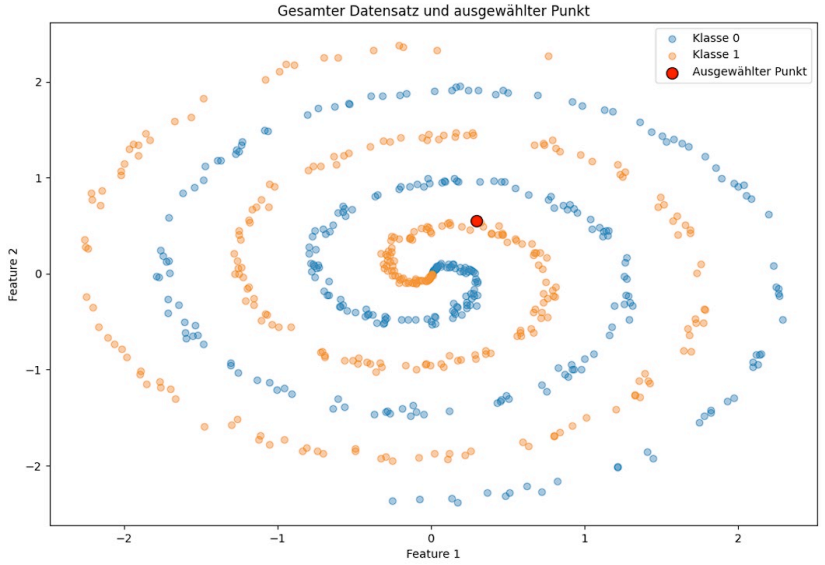
Listing 4.2: Funktion mit *LinearGAM*

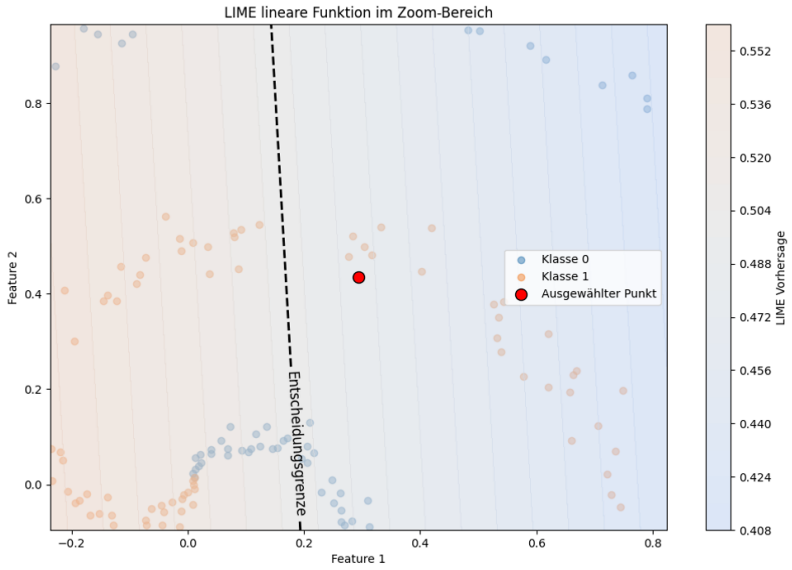
5. Bewertung des Ansatzes

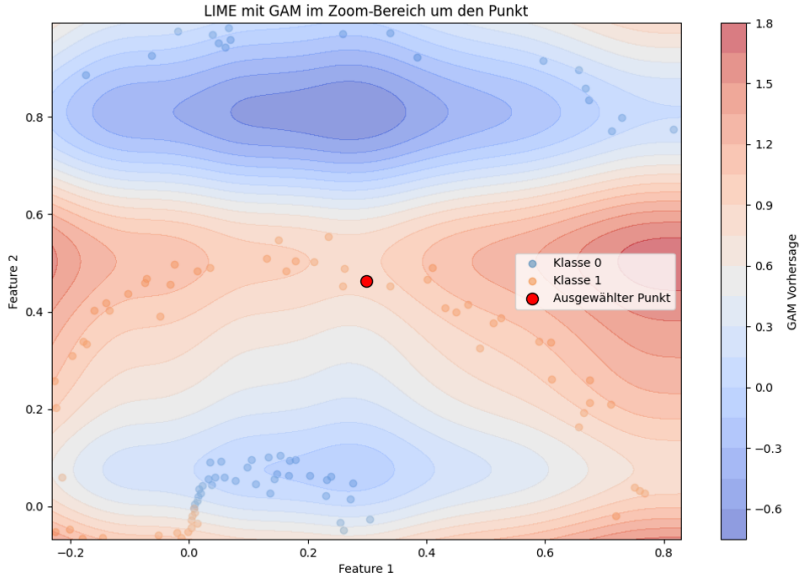












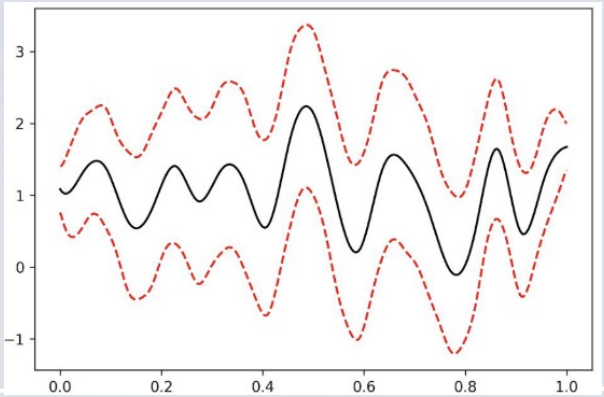
5. Bewertung des Ansatzes

- OpenXAI: Bewertung von XAI-Methoden: RIS, RRS, ROS -> **Stabilität** der Erklärungen (Ziel: 0)

Methode	RIS	RRS	ROS
LIME	3.28 ± 0.86	2.31 ± 0.41	4.67 ± 4.48
LIME + GAMs	$\mathbf{2.87 \pm 0.59}$	$\mathbf{2.09 \pm 0.08}$	$\mathbf{4.61 \pm 3.39}$

Tabelle 2: Stabilitätsergebnisse für den „*Adult Income*“-Datensatz

- Rechenaufwände („**Computational Efficiency**“):
  - LIME + GAMs: 2 *Stunden* ; LIME: 2 *Minuten*
- Benutzerfreundlichkeit („**Usability**“): Diagramme zur Darstellung der partiellen Abhängigkeiten der Merkmale



6. Kritische Würdigung & Ausblick

- wesentliche Schwächen des herkömmlichen LIME-Ansatzes (Beschränkung auf lokale Linearität) überwunden
- partielle Abhängigkeitsfunktionen der Merkmale

➢ detailliertere Modellinterpretation, die zusätzliche Informationen bereitstellt

- Problem der Koeffizientextraktion zur Merkmalsgewichtung
- Limitation: erhöhter Rechenaufwand

6. Kritische Würdigung & Ausblick

- Ausblick:
  - Stabilität, Effizienz, Koeffizientenextraktion verbessern
  - Verfahren für unterschiedliche Modelltypen und Anwendungsszenarien testen
  - Anpassung Kernel-Bereich (GAM weniger empfindlich?)
  - weitere Tests mit OpenXAI

7. Fazit

- Idee von GAMs in weiteren Anwendungen prüfen
- Limitationen beachten

- Flexibilisierung von LIME mithilfe von GAMs

➢ neue Perspektiven für die transparente und nachvollziehbare Nutzung von KI-Systemen

Flexibilisierung von Local Interpretable Model-Agnostic Explanations (LIME) mithilfe von Generalized Additive Models (GAMs)