

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА
по дисциплине «Дифференциальные уравнения»
Тема: Аппроксимация данных линейной комбинацией
экспоненциальных функций

Студент гр. 2384

Поглазов Н.В.

Преподаватель

Колоницкий С.Б.

Санкт-Петербург

2024

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Поглазов Н.В.

Группа 2384

Тема работы: Аппроксимация данных линейной комбинацией экспоненциальных функций

Исходные данные: ...

Содержание пояснительной записки: содержание, введение, ..., заключение, список использованных источников, ...

Предполагаемый объем пояснительной записки: не менее 00 страниц.

Дата выдачи задания: 17.10.2024

Дата сдачи реферата: 14.12.2024

Студент _____

Поглазов Н.В.

Преподаватель _____

Колоницкий С.Б.

АННОТАЦИЯ

Кратко (в 8–10 строк) указать основное содержание курсового проекта (курсовой работы), методы исследования (разработки), полученные результаты.

SUMMARY

Summary in English.

СОДЕРЖАНИЕ

Введение	5
1. Выполнение работы	9
1.1. Детали реализации	9
1.1.1. Предобработка данных	9
1.1.2. Обновление параметров	9
1.1.3. Критерии сходимости	10
1.2. Программная реализация алгоритма	10
1.2.1. Структура проекта	11
1.2.2. Класс <code>ExponentialRegression</code>	11
1.2.3. Класс <code>Chi2Loss</code>	14
2. Демонстрация работы программы	15
2.1. Пример с одним экспоненциальным членом	15
2.2. Пример с двумя экспоненциальными членами	15
Заключение	18
Список использованных источников	19
Приложение А. Исходный код программы	20
A.1. <code>exponential_regression.py</code>	20
A.2. <code>loss/loss_function.py</code>	23
A.3. <code>loss/chi2_loss.py</code>	24
A.4. <code>main.py</code>	25

ВВЕДЕНИЕ

Цель работы.

Разработка и реализация метода аппроксимации ряда данных линейной комбинацией экспоненциальных функций с неизвестными вещественными показателями.

Задание.

Даны $(t_i, y_i)_{i=1}^n$, где $t_i \in \mathbb{R}$, $y_i \in \mathbb{R}$, $i = 1, \dots, n$.

Рассмотрим p — количество экспоненциальных членов, тогда мы подбираем функцию:

$$f : X \rightarrow Y; f(t, \mathbf{p}) = \sum_{i=1}^p \lambda_i \alpha_i^t$$

где $\mathbf{p} = (\lambda_1, \dots, \lambda_p, \alpha_1, \dots, \alpha_p)$ — параметры, которые необходимо подобрать.

Предполагая, что $\forall i \alpha_i > 0$, можно переписать $f(t, \mathbf{p})$ как:

$$f(t, \mathbf{p}) = \sum_{i=1}^p \lambda_i \exp(\ln(\alpha_i)t) = \sum_{i=1}^p \lambda_i \exp(\omega_i t),$$

где $\mathbf{p} = (\lambda_1, \dots, \lambda_p, \omega_1, \dots, \omega_p)$

Функция потерь для оптимизации — это χ^2 , или взвешенный MSE, так как он часто используется в задачах аппроксимации кривых:

$$L(\mathbf{p}) = \chi^2(\mathbf{p}) = \sum_{i=1}^n \left(\frac{y_i - f(t_i, \mathbf{p})}{\sigma_i} \right)^2 = [\mathbf{y} - \mathbf{f}(\mathbf{p})]^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})],$$

где $\mathbf{W} = \text{diag} \left(\frac{1}{\sigma_1^2}, \dots, \frac{1}{\sigma_n^2} \right)$ - матрица весов: $\sigma_i^2 = \mathbb{D}[y_i]$.

Метод оптимизации — алгоритм Левенберга-Марквардта, встроенный во множество библиотек для оптимизации, например, в SciPy.

Основные теоретические положения.

Подобно другим численным методам минимизации, алгоритм Левенберга-Марквардта является итеративной процедурой. Для начала минимизации необходимо задать начальное приближение для вектора параметров. Начальное значение $\mathbf{p}^T = (1, 1, \dots, 1)$ подходит в большинстве случаев; в задачах с множеством локальных минимумов алгоритм сходится к глобальному минимуму, только если начальное приближение достаточно близко к решению.

На каждом шаге итерации вектор параметров \mathbf{p} заменяется новой оценкой $\mathbf{p} + \Delta$. Чтобы определить Δ , функция $f(t_i, \mathbf{p} + \Delta)$ линеаризуется:

$$f(t_i, \mathbf{p} + \Delta) \approx f(t_i, \mathbf{p}) + \mathbf{J}_i \Delta,$$

где

$$\mathbf{J}_i = \frac{\partial f(t_i, \mathbf{p})}{\partial \mathbf{p}}$$

— это градиент f по параметрам \mathbf{p} .

Таким образом, для текущей задачи $\forall j \leq p$:

$$\mathbf{J}_{ij} = \frac{\partial f(t_i, \mathbf{p})}{\partial \lambda_j} = \exp(\omega_j t_i),$$

$$\mathbf{J}_{ij+p} = \frac{\partial f(t_i, \mathbf{p})}{\partial \omega_j} = \lambda_j t_i \exp(\omega_j t_i).$$

Функция потерь достигает минимума, когда её градиент по \mathbf{p} равен нулю. Для первого приближения $f(t_i, \mathbf{p} + \Delta)$:

$$L(\mathbf{p} + \Delta) \approx \sum_{i=1}^n \left[\frac{y_i - f(t_i, \mathbf{p}) - \mathbf{J}_i \Delta}{\sigma_i} \right]^2$$

или в векторной форме:

$$\begin{aligned}
L(\mathbf{p} + \Delta) &\approx \left\| \mathbf{W}^{1/2} [\mathbf{y} - \mathbf{f}(\mathbf{p}) - \mathbf{J}\Delta] \right\|_2^2 \\
&= [\mathbf{y} - \mathbf{f}(\mathbf{p}) - \mathbf{J}\Delta]^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p}) - \mathbf{J}\Delta] \\
&= [\mathbf{y} - \mathbf{f}(\mathbf{p})]^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})] - 2 [\mathbf{y} - \mathbf{f}(\mathbf{p})]^T \mathbf{W} \mathbf{J} \Delta + \Delta^T \mathbf{J}^T \mathbf{W} \mathbf{J} \Delta.
\end{aligned}$$

Взяв производную от $L(\mathbf{p} + \Delta)$ по Δ и приравняв её к нулю, получим:

$$(\mathbf{J}^T \mathbf{W} \mathbf{J}) \Delta = \mathbf{J}^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})].$$

Выражение выше соответствует методу Гаусса–Ньютона. Матрица Якоби \mathbf{J} обычно не квадратная, а прямоугольная размерности $m \times n$, где n — количество параметров. Перемножение $\mathbf{J}^T \mathbf{W} \mathbf{J}$ дает квадратную матрицу размерности $n \times n$. Результат — это система из n линейных уравнений, решаемая для Δ .

Вклад Левенберга заключается в использовании демпфированной версии уравнения:

$$(\mathbf{J}^T \mathbf{W} \mathbf{J} + \lambda \mathbf{E}) \Delta = \mathbf{J}^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})].$$

где λ — коэффициент демпфирования, изменяемый после каждого вычисления Δ . Если снижение L быстрое, значение λ уменьшается, приближая алгоритм к методу Гаусса–Ньютона:

$$\Delta \approx [\mathbf{J}^T \mathbf{W} \mathbf{J}]^{-1} \mathbf{J}^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})],$$

иначе λ увеличивается, приближая шаг к направлению градиентного спуска:

$$\Delta \approx \lambda^{-1} \mathbf{J}^T \mathbf{W} [\mathbf{y} - \mathbf{f}(\mathbf{p})].$$

На старте алгоритма λ обычно берется достаточно большим (≈ 1), чтобы делать первые шаги в направлении градиентного спуска. После каждой итерации λ умножается или делится на определенный фактор, чтобы менять скорость сходимости. Подробности см. в разделе "Выполнение работы" подраздел "Детали реализации".

1. ВЫПОЛНЕНИЕ РАБОТЫ

1.1. Детали реализации

Так как производительность реализации, основанной исключительно на теоретических выкладках, оказалась недостаточной для практического применения, в итоговом решении были внесены несколько улучшений.

1.1.1. Предобработка данных

Для повышения численной устойчивости было принято решение стандартизировать данные перед обучением модели. Для этого используется формула:

$$x_i = \frac{x_i - \mu}{\sigma},$$

то есть, после предобработки все значения признаков будут иметь нулевое среднее и единичное стандартное отклонение. Также этот метод не меняет форму распределения данных, что важно для сохранения интерпретируемости результатов.

1.1.2. Обновление параметров

Ранее, найденная Δ применялась, если функция потерь уменьшалась, иначе она отклонялась, а коэффициент демпфирования увеличивался. Теперь шаг принимается, если метрика ρ (была предложена Нильсеном в его статье 1999 года [3]) больше порогового значения $\varepsilon_4 > 0$. Эта метрика измеряет фактическое уменьшение χ^2 по сравнению с улучшением, достигаемым шагом метода Левенберга-Марквардта:

$$\begin{aligned}\rho &= \frac{\chi^2(\mathbf{p}) - \chi^2(\mathbf{p} + \Delta)}{|(\mathbf{y} - \hat{\mathbf{y}})^T \mathbf{W}(\mathbf{y} - \hat{\mathbf{y}}) - (\mathbf{y} - \hat{\mathbf{y}} - \mathbf{J}\Delta)^T \mathbf{W}(\mathbf{y} - \hat{\mathbf{y}} - \mathbf{J}\Delta)|} \\ &= \frac{\chi^2(\mathbf{p}) - \chi^2(\mathbf{p} + \Delta)}{|\Delta^T (\lambda \Delta + \mathbf{J}^T \mathbf{W}(\mathbf{y} - \hat{\mathbf{y}}))|}\end{aligned}$$

где $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{p})$.

Коэффициент демпфирования и параметры модели обновляются согласно следующим правилам:

Если $\rho > \varepsilon_4$:

$$\lambda = \max \left[\frac{\lambda}{L_{\downarrow}}, 10^{-7} \right], \mathbf{p} \leftarrow \mathbf{p} + \Delta$$

иначе:

$$\lambda = \min [\lambda L_{\uparrow}, 10^7], \mathbf{p} \leftarrow \mathbf{p}$$

где $L_{\downarrow} \approx 9$ и $L_{\uparrow} \approx 11$ — фиксированные константы. Эти значения хорошо показали себя на практике и были выбраны на основе статьи [2].

1.1.3. Критерии сходимости

Алгоритм останавливается, когда выполняется *одно* из следующих условий:

- Сходимость по градиенту: $\max |\mathbf{J}^T \mathbf{W}(\mathbf{y} - \hat{\mathbf{y}})| < \varepsilon_1$
- Сходимость по коэффициентам: $\max |\Delta/\mathbf{p}| < \varepsilon_2$
- Сходимость по (редуцированному) χ^2 : $\chi_{\nu}^2 = \chi^2/(m - n) < \varepsilon_3$

1.2. Программная реализация алгоритма

Для реализации алгоритма был выбран язык программирования Python, так как он предоставляет широкие возможности для научных вычислений и

имеет большое количество библиотек для работы с данными. В качестве основной библиотеки для работы с данными была выбрана библиотека NumPy, а для работы с графиками — Matplotlib.

1.2.1. Структура проекта

Проект разделен на следующие модули:

- `exponential_regression.py` — модуль с реализацией алгоритма в классе `ExponentialRegression`
- `loss/loss_function.py` — модуль с реализацией базового класса для функций потерь `LossFunction`
- `loss/chi2_loss.py` — модуль с реализацией функции потерь χ^2 в классе `Chi2Loss`
- `main.py` — точка входа в программу, содержит пример использования алгоритма

1.2.2. Класс `ExponentialRegression`

Класс `ExponentialRegression` реализует регрессионную модель на основе экспоненциальных функций. Он наследует `BaseEstimator` и `RegressorMixin` из библиотеки `scikit-learn`, что делает его совместимым с её API. Основное назначение класса `ExponentialRegression` — обучение и предсказание на основе экспоненциальной зависимости.

При создании экземпляра класса задаются следующие параметры:

- `n_terms` (int, по умолчанию 1): число экспоненциальных членов в модели.
- `max_iter` (int, по умолчанию 1000): максимальное число итераций для процедуры оптимизации.

- `gradient_tol` (float, по умолчанию 10^{-3}): порог для остановки оптимизации по величине градиента.
- `coefficients_tol` (float, по умолчанию 10^{-3}): порог для изменения коэффициентов модели.
- `chi2_reduced_tol` (float, по умолчанию 0.1): порог для среднего значения χ^2 .
- `step_acceptance` (float, по умолчанию 0.1): минимальное значение отношения улучшения шага для принятия шага.
- `reg_init` (float, по умолчанию 0.1): начальное значение регуляризации.
- `loss_function` (LossFunction, по умолчанию Chi2Loss): функция потерь, используемая для вычисления градиента, гессиана и значения ошибки.

В дополнение к параметрам инициализации, класс определяет несколько предустановленных констант, таких как коэффициенты изменения регуляризации `REG_INCREASE_FACTOR`, `REG_DECREASE_FACTOR`, а также минимальные и максимальные значения регуляризации `REG_MIN` и `REG_MAX`.

```
fit(data, target, initial_lambda=None,
    initial_omega=None):
```

Метод обучает модель на основе входных данных:

- `data` (np.ndarray): одномерный массив временных значений.
- `target` (np.ndarray): одномерный массив значений целевой переменной.
- `initial_lambda` (Optional[np.ndarray]): начальные значения коэффициентов, если заданы.

- `initial_omega` (Optional[np.ndarray]): начальные значения параметров, если заданы.

Для решения системы $[\mathbf{J}^T \mathbf{W} \mathbf{J}] \Delta = \mathbf{J}^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}})$ используется функция `scipy.linalg.solve` из библиотеки SciPy.

`predict(data):`

Метод делает предсказания для входных данных `data`:

- `data` (np.ndarray): одномерный массив временных значений.

Приватные (вспомогательные) методы:

- `_init_parameters(n_terms):` инициализация параметров **p**.
- `_jacobian(t):` вычисление якобиана.
- `_regularize_hessian(hessian):` добавление регуляризации к гессиану.
- `_accept_step(t, y_true, y_pred, delta, gradient):` проверка, является ли шаг улучшением, и обновление параметров в случае успеха.
- `_increase_regularization():` увеличение параметра регуляризации.
- `_decrease_regularization():` уменьшение параметра регуляризации.
- `_check_convergence(gradient, lambda_delta, omega_delta, loss, measurements_amount):` проверка условий сходимости.
- `_model(t, lambda_=None, omega_=None):` вычисление $f(\mathbf{t}, \mathbf{p})$.

1.2.3. Класс Chi2Loss

Класс `Chi2Loss` реализует функцию потерь χ^2 . Он наследует `LossFunction` и реализует методы для вычисления градиента, гессиана и значения функции потерь.

При создании экземпляра класса задаются следующие параметры:

- `measurements_amount (int)`: количество измерений;
- `measurement_variance (np.ndarray | float | None, по умолчанию None)`: дисперсия каждого измерения или общая дисперсия;

с помощью которых вычисляется матрица весов \mathbf{W} .

`loss(y_true, y_pred):`

Метод вычисляет значение функции потерь:

$$\chi^2 = [\mathbf{y} - \hat{\mathbf{y}}]^T \mathbf{W} [\mathbf{y} - \hat{\mathbf{y}}],$$

`gradient(t, y_true, y_pred, jacobian):`

Метод вычисляет (анти-) градиент функции потерь деленный на 2 (для удобства в использовании):

$$-\frac{1}{2} \frac{\partial \chi^2}{\partial \mathbf{p}} = \mathbf{J}^T \mathbf{W} (\mathbf{y} - \hat{\mathbf{y}})$$

`hessian(jacobian):`

Метод (приближенно) вычисляет гессиан функции потерь:

$$\frac{\partial^2 \chi^2}{\partial \mathbf{p}^2} \approx \mathbf{J}^T \mathbf{W} \mathbf{J}$$

где $\hat{\mathbf{y}}$ - `y_pred`, \mathbf{y} - `y_true`, \mathbf{J} - `jacobian`, \mathbf{W} - диагональная матрица весов.

2. ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ

2.1. Пример с одним экспоненциальным членом

Для примера были сгенерированы данные, соответствующие зависимости $f(t) = 2 \exp(-0.25t) + \varepsilon$, $\varepsilon \in \mathcal{N}(0, 1)$ — случайный шум.

Было проведено 100 замеров в диапазоне $t \in [-10, 10]$. После обучения модели были получены следующие результаты:

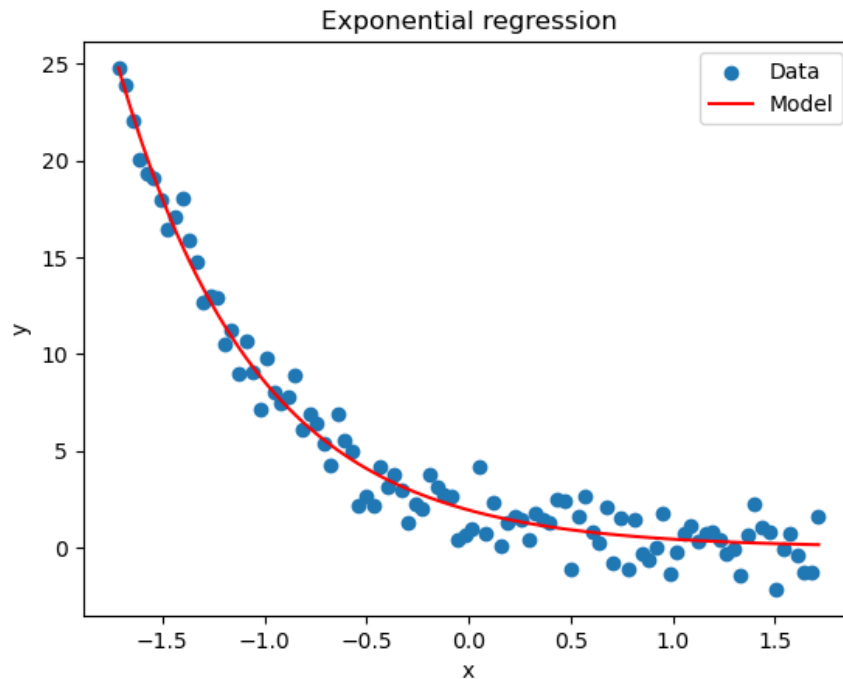


Рисунок 1 – Пример с одним экспоненциальным членом

Как видно из графика, модель хорошо отфильтровала шум и восстановила исходную зависимость.

2.2. Пример с двумя экспоненциальными членами

Для примера были сгенерированы данные, соответствующие зависимости $f(t) = 2 \exp(-0.25t) - 5 \exp(-2t) + \varepsilon$, $\varepsilon \in \mathcal{N}(0, 0.1^2)$ — случайный шум.

Было проведено 100 замеров в диапазоне $t \in [0, 10]$. После обучения модели были получены следующие результаты:

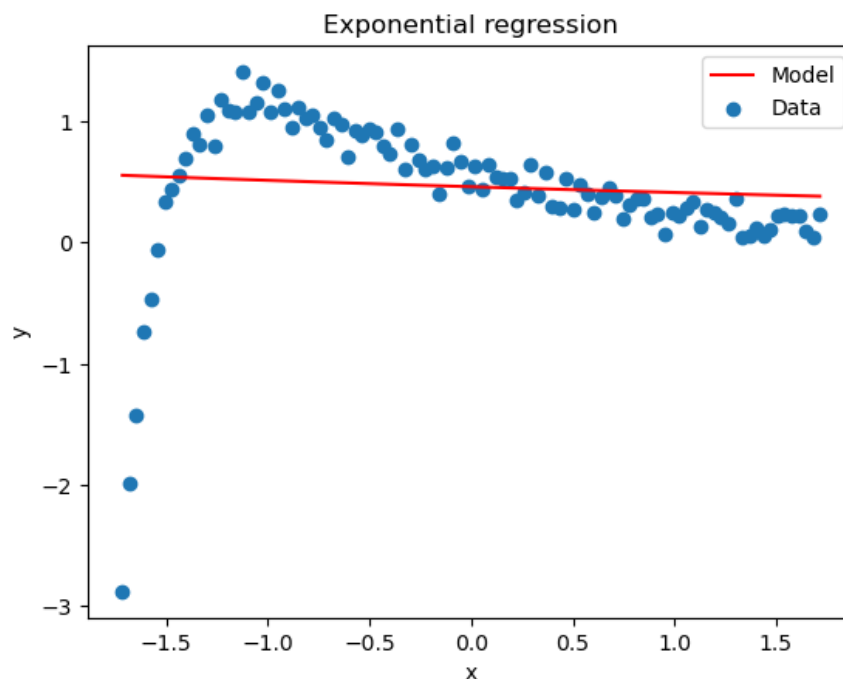


Рисунок 2 – Пример с двумя экспоненциальными членами

Как видно из графика, модель не смогла восстановить исходную зависимость. Это связано с тем, что функция потерь $\chi^2(\mathbf{p})$ может иметь множество локальных минимумов. В таких случаях метод Левенберга-Марквардта может сходиться к неудовлетворительному решению. Если это происходит, пользователь может попытаться задать лучшее начальное приближение для параметров, например, с помощью случайного поиска, или поиска по сетке, либо путем анализа данных.

Попробуем улучшить результат, вручную задав начальные приближения для параметров:

$$\lambda = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

$$\omega = \begin{bmatrix} -1 & -1 \end{bmatrix}$$

После обучения модели с начальными приближениями были получены следующие результаты:

Как видно из графика, модель восстановила исходную зависимость. Это показывает, что правильный выбор начальных приближений для параметров может существенно повлиять на результат.

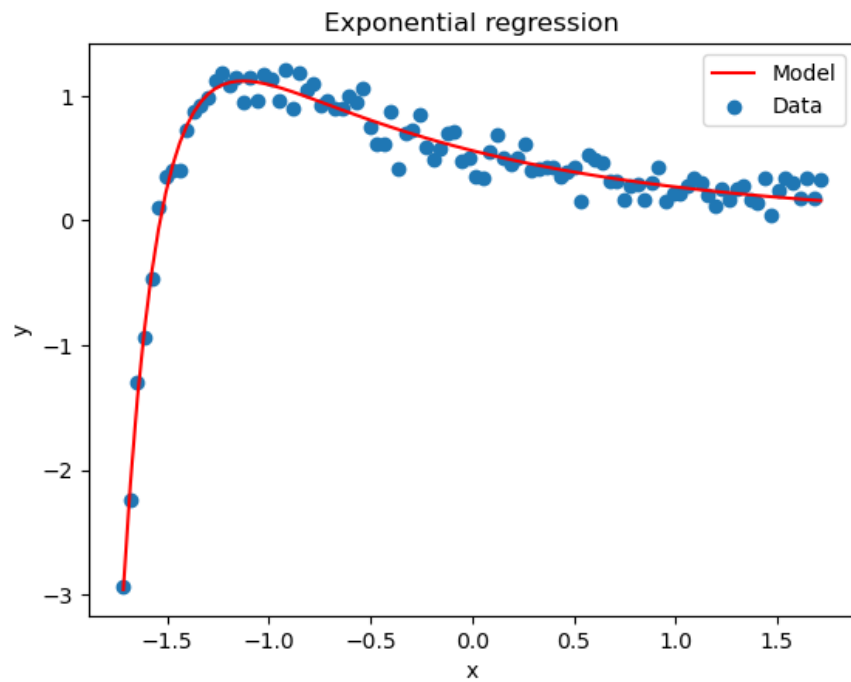


Рисунок 3 – Пример с двумя экспоненциальными членами (улучшенный результат)

Написанный алгоритм позволяет удобно и быстро решать задачи аппроксимации данных линейной комбинацией экспоненциальных функций. Он легко расширяется на случай большего числа экспоненциальных членов, а также на случай других функций потерь.

Реализованный код см. в приложении А.

ЗАКЛЮЧЕНИЕ

В процессе выполнения работы удалось глубже изучить тему экспоненциальной регрессии, разработать её алгоритм и проверить его эффективность. Были рассмотрены различные методы оптимизации, такие как алгоритм Левенберга-Марквардта.

В итоге был разработан и протестирован алгоритм, который подтвердил свою эффективность и практическую ценность. Полученные результаты открывают возможности для дальнейшей работы над улучшением методов регрессии и их применением для решения более сложных задач анализа данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Wikipedia contributors. *Levenberg–Marquardt algorithm*. Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm.
- [2] H. P. Gavin. *The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems*. 2020. Available at: <https://people.duke.edu/~hpgavin/ce281/lm.pdf>.
- [3] H. B. Nielsen. *Damping Parameter in Marquardt's Method*. 1999. Available at: https://www2.imm.dtu.dk/documents/ftp/tr99/tr05_99.pdf.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

A.1. exponential_regression.py

```
from typing import Self, Optional

import numpy as np

from sklearn.base import BaseEstimator, RegressorMixin
from sklearn.utils.validation import check_X_y, check_array

from .loss import LossFunction, Chi2Loss


class ExponentialRegression(BaseEstimator, RegressorMixin):
    GRADIENT_TOL: float = 1e-3
    COEFFICIENTS_TOL: float = 1e-3
    CHI_SQR_REDUCED_TOL: float = 1e-1
    STEP_ACCEPTANCE: float = 1e-1

    REG_INIT: float = 1e-1
    REG_INCREASE_FACTOR: float = 11.
    REG_DECREASE_FACTOR: float = 9.
    REG_MIN: float = 1e-7
    REG_MAX: float = 1e7

    def __init__(self,
                 n_terms: int = 1,
                 max_iter: int = 1000,
                 gradient_tol: float = GRADIENT_TOL,
                 coefficients_tol: float = COEFFICIENTS_TOL,
                 chi2_reduced_tol: float = CHI_SQR_REDUCED_TOL,
                 step_acceptance: float = STEP_ACCEPTANCE,
                 reg_init: float = REG_INIT,
                 loss_function: LossFunction = None) -> None:
        self.n_terms = n_terms
        self.max_iter = max_iter

        self.gradient_tol = gradient_tol
        self.coefficients_tol = coefficients_tol
        self.chi2_reduced_tol = chi2_reduced_tol
        self.step_acceptance = step_acceptance

        self.reg_init = reg_init

        self.loss_function = loss_function

        self.regularization_: None | float = None
        self.lambda_: None | np.ndarray = None
        self.omega_: None | np.ndarray = None

    def fit(self,
            data: np.ndarray,
            target: np.ndarray,
            initial_lambda: Optional[np.ndarray] = None,
            initial_omega: Optional[np.ndarray] = None) -> Self:
        data, target = check_X_y(data, target, ensure_2d=False)
        t = data.ravel()
```

```

if initial_lambda is not None and initial_omega is not None:
    self.lambda_ = initial_lambda
    self.omega_ = initial_omega
else:
    self._init_parameters(self.n_terms)

self.regularization_ = self.reg_init

new_y_pred = self._model(t)
is_step_accepted = True
hessian = gradient = None

for _ in range(self.max_iter):
    y_pred = new_y_pred
    if is_step_accepted:
        jacobian = self._jacobian(t)
        gradient = self.loss_function.gradient(
            target, y_pred, jacobian)
        loss = self.loss_function.loss(target, y_pred)

        if self._check_convergence(gradient, self.lambda_, self.omega_,
            , loss, data.size):
            break

        hessian = self.loss_function.hessian(jacobian)

    hessian = self._regularize_hessian(hessian)

    delta = np.linalg.solve(hessian, gradient)

    is_step_accepted, new_y_pred = self._accept_step(
        t, target, y_pred, delta, gradient)
    if is_step_accepted:
        self._decrease_regularization()
    else:
        self._increase_regularization()
else:
    print(f"Failed to converge after {self.max_iter} iterations")

return self

def _init_parameters(self, n_terms: int) -> None:
    self.lambda_ = np.ones(n_terms)
    self.omega_ = np.ones(n_terms)

def _jacobian(self, t: np.ndarray) -> np.ndarray:
    exp_terms = np.exp(np.outer(t, self.omega_))
    jacobian_lambda = exp_terms
    jacobian_omega = exp_terms * self.lambda_ * t[:, np.newaxis]
    jacobian = np.hstack((jacobian_lambda, jacobian_omega))
    return jacobian

def _regularize_hessian(self, hessian: np.ndarray) -> np.ndarray:
    return hessian + np.eye(hessian.shape[0]) * self.regularization_

def _accept_step(self, t: np.ndarray, y_true: np.ndarray, y_pred: np.
    ndarray, delta: np.ndarray,
        gradient: np.ndarray) -> tuple[bool, np.ndarray]:
    new_y_pred = self._model(t,
        self.lambda_ + delta[:self.n_terms],
        self.omega_ + delta[self.n_terms:])

    chi_sqr = self.loss_function.loss(y_true, y_pred)

```

```

new_chi_sqr = self.loss_function.loss(y_true, new_y_pred)

rho = (
    (chi_sqr - new_chi_sqr) /
    np.abs(delta.T @ (self.regularization_ * delta + gradient))
)

if rho > self.step_acceptance:
    self.lambda_ += delta[:self.n_terms]
    self.omega_ += delta[self.n_terms:]
    return True, new_y_pred
return False, y_pred

def _increase_regularization(self) -> None:
    self.regularization_ *= self.REG_INCREASE_FACTOR
    self.regularization_ = min(self.regularization_, self.REG_MAX)

def _decrease_regularization(self) -> None:
    self.regularization_ /= self.REG_DECREASE_FACTOR
    self.regularization_ = max(self.regularization_, self.REG_MIN)

def _check_convergence(self, gradient: np.ndarray, lambda_delta: np.
ndarray, omega_delta: np.ndarray,
                        loss: float, measurements_amount: int) -> bool:
    return (
        np.max(np.abs(gradient)) < self.gradient_tol or
        np.max(np.abs(lambda_delta / self.lambda_)) < self.
        coefficients_tol or
        np.max(np.abs(omega_delta / self.omega_)) < self.
        coefficients_tol or
        loss / (measurements_amount -
                gradient.shape[0]) < self.chi2_reduced_tol
    )

def predict(self, data: np.ndarray) -> np.ndarray:
    data = check_array(data, ensure_2d=False)
    t = data.ravel()
    return self._model(t)

def _model(self, t: np.ndarray, lambda_: np.ndarray = None, omega_: np.
ndarray = None) -> np.ndarray:
    lambda_ = self.lambda_ if lambda_ is None else lambda_
    omega_ = self.omega_ if omega_ is None else omega_

    exp_terms = np.exp(np.outer(t, omega_))
    return exp_terms @ lambda_

```

A.2. loss/loss_function.py

```
from abc import ABC, abstractmethod
import numpy as np
from typing import Optional

class LossFunction(ABC):
    def __repr__(self) -> str:
        return self.__class__.__name__

    @abstractmethod
    def loss(self, y_true: np.ndarray, y_pred: np.ndarray) -> float:
        """
        Compute the loss function given the residuals
        param y_true: true values
        param y_pred: predicted values
        """
        pass

    @abstractmethod
    def gradient(self, y_true: np.ndarray, y_pred: np.ndarray, jacobian: np.
        ndarray) -> np.ndarray:
        """
        Compute the gradient of the loss function
        param y_true: true values
        param y_pred: predicted values
        param jacobian: jacobian of the model
        """
        pass

    @abstractmethod
    def hessian(self, jacobian: np.ndarray) -> np.ndarray:
        """
        Compute the hessian of the loss function
        param jacobian: jacobian of the model
        """
        pass
```

A.3. loss/chi2_loss.py

```
import numpy as np
from typing import Optional

from .loss_function import LossFunction

class Chi2Loss(LossFunction):
    # Chi-squared (or weighted MSE) loss function
    def __init__(self, measurements_amount: int, measurement_variance: float |
        np.ndarray | None = None) -> None:
        if isinstance(measurement_variance, float):
            self._weights = np.eye(measurements_amount) / measurement_variance
        elif isinstance(measurement_variance, np.ndarray):
            self._weights = np.diag(1 / measurement_variance)
        else:
            self._weights = np.eye(measurements_amount)

    def loss(self, y_true: np.ndarray, y_pred: np.ndarray) -> float:
        residuals = y_true - y_pred
        return np.float_(residuals.T @ self._weights @ residuals)

    def gradient(self, y_true: np.ndarray, y_pred: np.ndarray, jacobian: np.
        ndarray) -> np.ndarray:
        residuals = y_true - y_pred
        return jacobian.T @ self._weights @ residuals

    def hessian(self, jacobian: np.ndarray) -> np.ndarray:
        return jacobian.T @ self._weights @ jacobian
```


A.4. main.py

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from src.exponential_regression import ExponentialRegression
from src.loss.chi2_loss import Chi2Loss


def main():
    measurements = 100
    measurement_errors = 1 / 20
    x = np.linspace(0, 10, measurements).reshape(-1, 1)
    y = (2 * np.exp(-0.25 * x) + (-5) * np.exp(-2 * x)).ravel()
    y += np.random.normal(0, measurement_errors, measurements)

    scaler = StandardScaler()
    x = scaler.fit_transform(x)

    loss = Chi2Loss(measurements, measurement_errors)

    er = ExponentialRegression(n_terms=2, loss_function=loss)
    er.fit(x, y, initial_lambda=np.array([1., -1.]), initial_omega=np.array(
        [-1., -1.]))
    coefficients = np.hstack((er.lambda_, er.omega_))
    print(coefficients)

    plt.scatter(x, y)
    plt.plot(x, er.predict(x), color='red')
    plt.show()


if __name__ == '__main__':
    main()
```