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 объём 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

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 $t_i \in \mathbb{R}$, $y_i \in \mathbb{R}$, $i = 1, \ldots, n$.

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

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

где $\mathbf{p} = (\lambda_1, \ldots, \lambda_p, \alpha_1, \ldots, \alpha_p)$ — 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 $\mathbf{p} = (\lambda_1, \ldots, \lambda_p, \omega_1, \ldots, \omega_p)$

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

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

где $\boldsymbol{W} = \mathrm{diag}\left(\frac{1}{\sigma_1^2}, \ldots, \frac{1}{\sigma_n^2}\right)$ - 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 $\mathbf{p}^T = \Big(1, 1, \ldots, 1\Big)$ 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 **p** заменяется новой оценкой $\mathbf{p} + \mathbf{\Delta}$. Чтобы определить $\mathbf{\Delta}$, функция $f(t_i, \mathbf{p} + \mathbf{\Delta})$ линеаризуется:

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

где

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

— это градиент f по параметрам **p**.

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

$$\mathbf{J}_{ij} = \frac{\partial f\left(t_i, \mathbf{p}\right)}{\partial \lambda_{\mathbf{j}}} = \exp(\omega_j t_i),$$

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

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

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

или в векторной форме:

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

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

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

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

Вклад Левенберга заключается в использовании демпфированной версии уравнения:

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

где λ — коэффициент демпфирования, изменяемый после каждого вычисления $\boldsymbol{\Delta}$. Если снижение L быстрое, значение λ уменьшается, приближая алгоритм к методу Гаусса–Ньютона:

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

иначе λ увеличивается, приближая шаг к направлению градиентного спуска:

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

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 $\mathbf{\Delta}$ 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-Марквадта:

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

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

Коэффициент демпфирования и параметры модели обновляются согласно следующим правилам:

Если $\rho > \varepsilon_4$:

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

иначе:

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

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

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 $[\mathbf{J}^T\mathbf{W}\mathbf{J}]\boldsymbol{\Delta} = \mathbf{J}^T\mathbf{W}(\mathbf{y} - \hat{\mathbf{y}})$ 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:

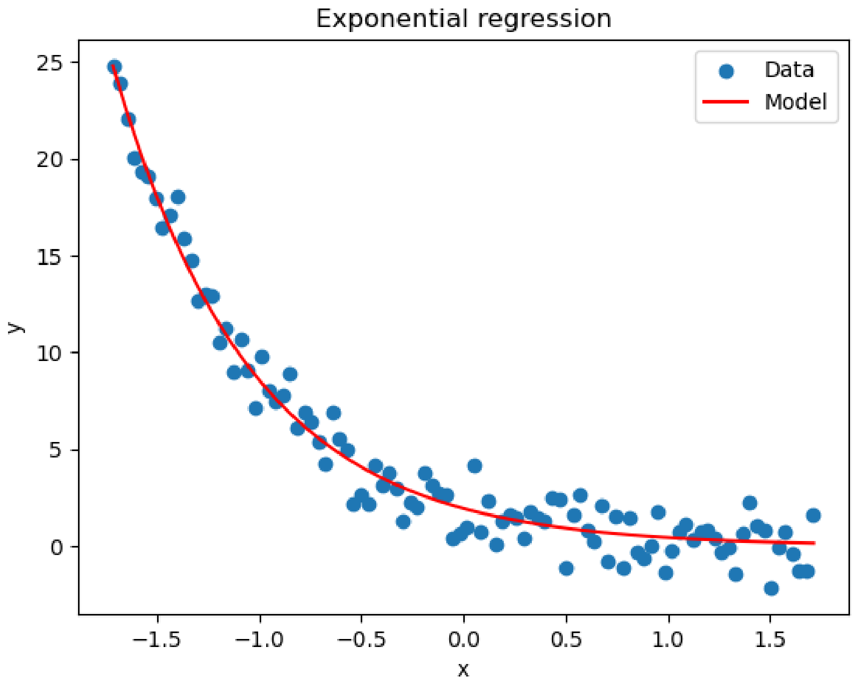

Рисунок 1 – Пример с одним экспоненциальным членом

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:

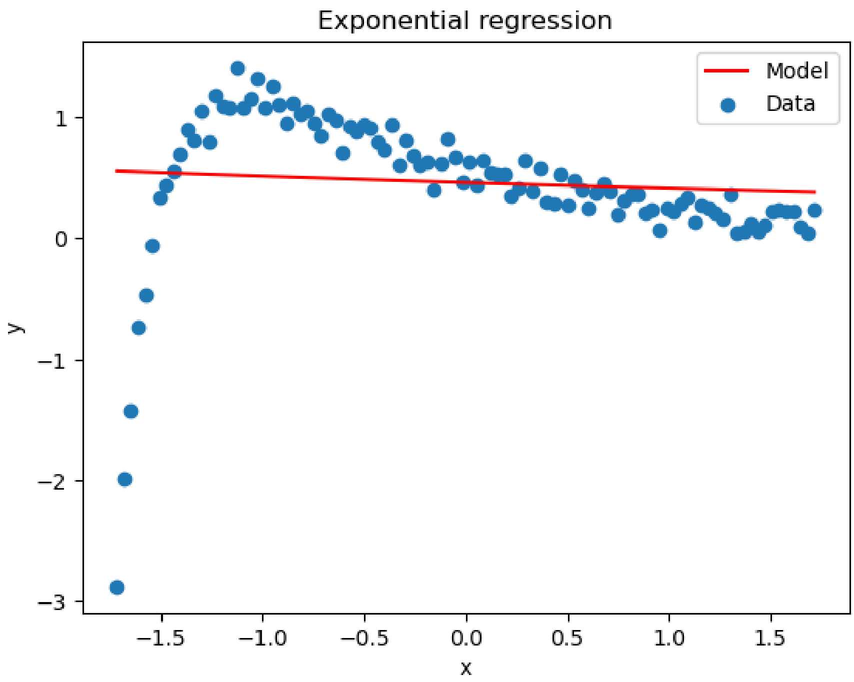

Рисунок 2 – Пример с двумя экспоненциальными членами

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.

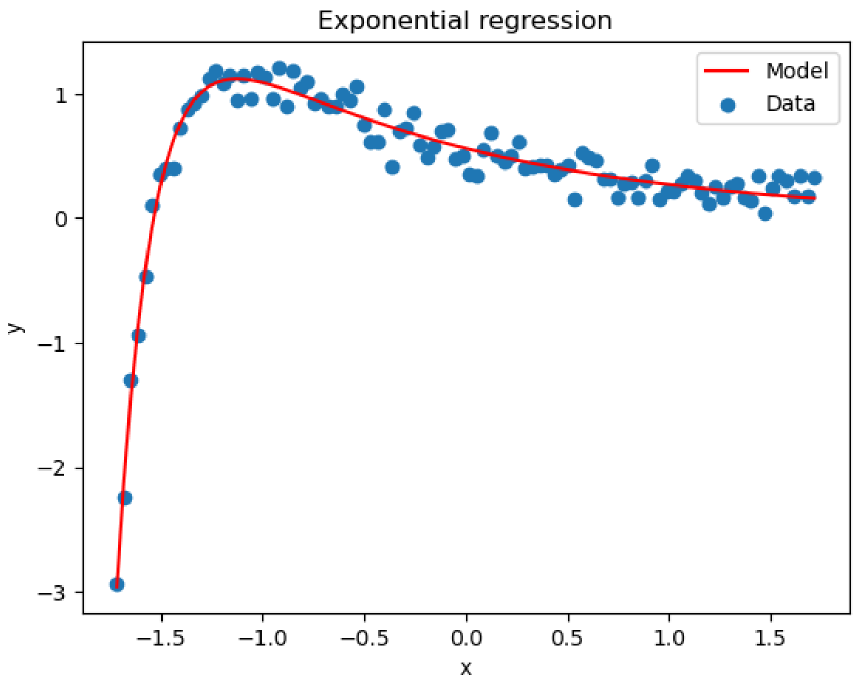

Рисунок 3 – Пример с двумя экспоненциальными членами (улучшенный результат)

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 курсовой работы была достигнута цель, заключающаяся в разработке метода аппроксимации данных линейной комбинацией экспоненциальных функций с использованием алгоритма Левенберга-Марквардта. Были детально изучены основные математические принципы алгоритма, включая оптимизационные подходы, такие как градиентный спуск и метод Гаусса–Ньютона, а также особенности их сочетания в демпфированном виде.

Важным этапом стало внедрение методов повышения численной устойчивости, включая стандартизацию данных и адаптивное управление коэффициентом регуляризации. Реализованная модель продемонстрировала свою эффективность на тестовых данных, успешно аппроксимируя зависимости с одним и двумя экспоненциальными членами. Тем не менее, результаты также выявили чувствительность метода к выбору начальных приближений параметров, что требует дополнительного внимания при его применении.

Практическая реализация была выполнена на языке программирования Python с использованием современных библиотек для численных вычислений и визуализации. Предложенный подход может быть расширен для решения более сложных задач, например, с увеличением числа экспоненциальных членов или использованием других функций потерь. Разработанный алгоритм и полученные результаты подчеркивают его потенциал для дальнейших исследований и применения в задачах 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

```python
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()
```