**Міністерство освіти і науки України**

Харківський національний університет імені В.Н. Каразіна

Лабораторна робота #3

# Середньоквадратичне наближення функцій алгебраїчними многочленами

**Виконав:**

Захаров Дмитро Олегович

Група МП-31

Харків – 2023

# Зміст

# 1 Постановка задачі

За даними з таблиці побудувати середньоквадратичне наближення функції, заданої таблично, алгебраїчним багаточленом третього ступеня.

Обчислити значення побудованого кубічного багаточлена у вузлах таблиці та оцінити похибку середньоквадратичного наближення.

На друк вивести результати у вигляді таблиць:

$$x_i \quad y_i \quad f(x_i) \quad |y_i - f(x_i)|,$$

а також $\Delta$ – відносну похибку середньоквадратичного наближення.

**Варіант 5.**

| $y$ | 0.8 | 1.2 | 1.6 | 2.0 | 2.4 | 2.8 | 3.2 | 3.6 | 4.0 | 4.4 |
|---|---|---|---|---|---|---|---|---|---|---|
| $x$ | 1.97 | 3.53 | 3.57 | 2.42 | 2.44 | 1.30 | 0.71 | 2.12 | 3.08 | 5.99 |

# 2 Опис методу

## 2.1 Класична перспектива

Нехай ми шукаємо апроксимацію у вигляді:

$$f(x; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(x)$$

де $\mathbf{w} \in \mathbb{R}^{m+1}$ – вектор ваг, $\boldsymbol{\phi}(x) = \begin{bmatrix} 1 & x & x^2 & \dots & x^m \end{bmatrix}^\top$.

Нехай ми маємо таблицю $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n_\mathcal{D}} \subset \mathbb{R} \times \mathbb{R}$, яку ми хочемо апроксимувати. Суть середньоквадратичної помилки – оптимізувати функцію втрати:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w} \mid \mathcal{D}), \text{ де } \mathcal{L}(\mathbf{w} \mid \mathcal{D}) \triangleq \sum_{i=1}^{n_\mathcal{D}} (f(x_i; \mathbf{w}) - y_i)^2$$

Набір данних легше зобразити у вигляді матриці:

$$\mathbf{X} = \begin{bmatrix} \boldsymbol{\phi}(x_1)^\top \\ \boldsymbol{\phi}(x_2)^\top \\ \vdots \\ \boldsymbol{\phi}(x_{n_\mathcal{D}})^\top \end{bmatrix} \in \mathbb{R}^{n_\mathcal{D} \times (m+1)}, \ \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n_\mathcal{D}} \end{bmatrix} \in \mathbb{R}^{n_\mathcal{D}}$$

Тоді, наша задача еквівалентна

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = \arg\min_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

Помітимо, що екстремум можна знайти за допомогою:

$$\frac{\partial \mathcal{L}(\mathbf{w} \mid \mathcal{D})}{\partial \mathbf{w}} = 0 \iff \frac{\partial (\mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{X}\mathbf{w} - \mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w})}{\partial \mathbf{w}} = 0$$

Після диференціювання, отримуємо:

$$-2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X}\mathbf{w} = 0 \implies \boxed{\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}}$$

## 2.2 Ймовірнісна перспектива

Нехай ми вважаємо, що функція $f(x; \mathbf{w})$, наведена у попередньому розділі 2.1, найкраще апроксимує набір данних $\mathcal{D}$. В такому разі будемо вважати, що джерело різниці між реальними значеннями і "теоретично" правильними – це гаусовий гум $\epsilon$, тобто:

$$y = f(x; \mathbf{w}) + \epsilon, \text{ де } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

В такому разі, умовна умовірність зустріти значення $y$ при заданому $x$:

$$p(y \mid x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{(y - f(x; \mathbf{w}))}{2\sigma^2} \right\}$$

Нам потрібно знайти $\arg\max_{\mathbf{w}} p(\mathbf{y} \mid \mathbf{X})$, тобто максимізувати ймовірність мати увесь набір данних. Тобто:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \prod_{i=1}^{n_{\mathcal{D}}} p(y_i \mid x_i)$$

Проте, знаходити максимум від добутку не дуже зручно, тому помітимо, що ми можемо оптимізовувати замість цього так званий *negative log-likelihood*:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \left( -\log \prod_{i=1}^{n_{\mathcal{D}}} p(y_i \mid x_i) \right) = \arg\min_{\mathbf{w}} \left( \frac{n_{\mathcal{D}}}{2} \log 2\pi\sigma^2 + \frac{1}{2\sigma^2} \sum_{i=1}^{n_{\mathcal{D}}} (y_i - f(x_i; \mathbf{w}))^2 \right)$$

Оскільки $\frac{n_{\mathcal{D}}}{2} \log 2\pi\sigma^2$ ніяк не залежить від $\mathbf{w}$, то задача на оптимізацію еквівалентна:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n_{\mathcal{D}}} (y_i - f(x_i; \mathbf{w}))^2,$$

що еквівалентно оптимізації у минулому пункті.

## 2.3 Оцінка точності

Після того, як ми отримали коефіцієнти $\hat{\mathbf{w}}$, ми можемо оцінити відносну похибку за формулою:

$$\Delta^2 = \frac{\sum_{i=1}^{n_{\mathcal{D}}} (y_i - f(x_i; \hat{\mathbf{w}}))^2}{\sum_{i=1}^{n_{\mathcal{D}}} y_i^2}$$

Або, на мові попередніх розділів,

$$\Delta = \frac{\|\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\|_2}{\|\mathbf{y}\|_2}$$

# 3 Текст програми

Повний текст програми можна знайти за цим посиланням (← напис клікабельний) на *GitHub* сторінку.

## 3.1 Пошук коефіцієнтів

Створимо файл `solver.py` та реалізуємо алгоритм з розділу 2.1 у класі `LeastSquaresSolver`. При ініціалізації ми будемо вказувати ступінь полінома за яким ми хочемо апроксимувати, а також додамо 2 методи:

1. `fit`: приймає набір данних $\mathcal{D}$ і повертає набір коефіцієнтів $\hat{\mathbf{w}}$ (а також зберігає їх в самому класі);
2. `predict`: приймає $x \in \mathbb{R}$ і повертає $\hat{\mathbf{w}}^\top \boldsymbol{\phi}(x)$, тобто значення полінома зі знайденими коефіцієнтами у вказаній точці.

```python
from typing import TypeAlias, Tuple, List
import numpy as np

Point: TypeAlias = Tuple[float, float]
Dataset: TypeAlias = List[Point]

class LeastSquaresSolver:
    """
    A class to solve the least squares problem
    """

    def __init__(self, degree: int) -> None:
        """
        Args:
            dataset (Dataset): A list of points
            degree (int): The degree of the polynomial to fit
        """

        self._degree = degree

    def _generate_power_vector(self, x: float) -> List[float]:
        """ Generates a row consisting of powers of x

        Args:
            x (float): The x value of the point

        Returns:
            List[float]: A row for the dataset matrix
        """

        return [x**i for i in range(self._degree + 1)]

```

```
    def fit(self, dataset: Dataset) -> np.ndarray:
        """ Returns a list of coefficients for the polynomial of the given degree

        Returns:
            np.ndarray: A list of coefficients for the polynomial of the given degree (degree + 1)
        """

        X = np.empty((len(dataset), self._degree + 1))
        Y = np.empty((len(dataset), 1))

        for i in range(len(dataset)):
            x, y = dataset[i]
            X[i] = self._generate_power_vector(x)
            Y[i] = y

        self._coefficients = np.linalg.inv(X.T @ X) @ X.T @ Y
        self._coefficients = np.squeeze(self._coefficients, axis=-1)
        return self._coefficients

    def predict(self, x: float) -> np.floating:
        """ Predicts the y value for the given x value

        Args:
            x (float): The x value to predict the y value for

        Returns:
            float: The predicted y value
        """

        return self._coefficients @ self._generate_power_vector(x)
```

Лістинг 1: Реалізація пошуку коефіцієнтів методом найменших квадратів

## 3.2 Програма запуску

При запуску, ми робимо наступні дії:

1. Ініціалізуємо `LeastSquaresSolver` та запускаємо `fit` метод на обраному наборі данних.
2. Будуємо графік, де зображуємо поліном та точки з набору данних.
3. Будуємо графік для різних ступенів поліному для перевірки.
4. Будуємо таблицю, котру потребує завдання.
5. Оцінюємо відносну похибку $\Delta$.

Отже, наводимо код знизу (файл `main.py`), котрий робить вищезгадані кроки.

```python
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np
from typing import List

# Rich logging
from rich import print
from rich.console import Console
from rich.table import Table

from solver import LeastSquaresSolver, Dataset

def _show_predictions(dataset: Dataset, solver: LeastSquaresSolver) -> None:
    """ Shows the predictions of the fitted polynomial

    Args:
        dataset (Dataset): A list of points
        solver (LeastSquaresSolver): A LeastSquaresSolver instance
    """

    # Initializing an instance of rich.Table to make an attractive table in the console
    console = Console()
    table = Table(title='Degree vs real values for a linear regression')
    table.add_column('x', justify='center', style='white')
    table.add_column('y', justify='center', style='green')
    table.add_column('f(x)', justify='center', style='blue')
    table.add_column('|y - f(x)|', justify='center', style='red')

    # Adding rows to the table
    for (x, y) in dataset:
        x_label = '{:.8f}'.format(x)
        y_label = '{:.8f}'.format(y)
        prediction_label = '{:.8f}'.format(float(solver.predict(x)))
        difference_label = '{:.8f}'.format(np.abs(y - solver.predict(x)))

        table.add_row(x_label, y_label, prediction_label, difference_label)

    console.print(table)

def _print_predicted_polynomial(coefficients: np.ndarray) -> None:
    """ Prints the predicted polynomial

    Args:
        coefficients (np.ndarray): A list of coefficients for the polynomial of the given degree (degree + 1)
```

```
45      """
46
47      # Initializing an instance of rich.Table to make an attractive
        table in the console
48      console = Console()
49      table = Table(title='Predicted polynomial')
50      table.add_column('Degree', justify='center', style='white')
51      table.add_column('Coefficient', justify='center', style='green')
52
53      # Adding rows to the table
54      for i, coefficient in enumerate(coefficients):
55          degree_label = f'x^{i}'
56          coefficient_label = '{:.4f}'.format(float(coefficient))
57
58          table.add_row(degree_label, coefficient_label)
59
60      console.print(table)
61
62  def _show_plot(dataset: Dataset, solver: LeastSquaresSolver) -> None:
63      """ Shows a plot of the dataset and the fitted polynomial
64
65      Args:
66          dataset (Dataset): A list of points
67          solver (LeastSquaresSolver): A LeastSquaresSolver instance
68      """
69
70      # Finding a range of values for x
71      min_x = min([point[0] for point in dataset])
72      max_x = max([point[0] for point in dataset])
73
74      # Drawing the plot
75      t = np.arange(min_x, max_x, 0.01)
76      _, ax = plt.subplots()
77      ax.grid()
78      ax.scatter([point[0] for point in dataset], [point[1] for point in
        dataset], marker='x', color='red')
79      ax.plot(t, [solver.predict(x) for x in t], color='blue')
80      plt.tight_layout()
81      plt.savefig('images/plot.png', dpi=300)
82      plt.show()
83
84  def _show_plot_of_different_degrees(dataset: Dataset) -> None:
85      """ Shows a plot of the dataset and the fitted polynomial of
        different degrees
86
87      Args:
88          dataset (Dataset): A list of points
89      """
90
91      # Finding a range of values for x
```

```
92      min_x = min([point[0] for point in dataset])
93      max_x = max([point[0] for point in dataset])
94
95      # Fitting the dataset with different degrees
96      solvers = [LeastSquaresSolver(degree=i) for i in range(1, 8)]
97      for solver in solvers:
98          solver.fit(dataset)
99
100     # Drawing the plot
101     t = np.arange(min_x, max_x, 0.01)
102     colors = ['cornflowerblue', 'royalblue', 'blue', 'mediumblue', '
    darkblue', 'navy', 'midnightblue']
103
104     _, ax = plt.subplots()
105     ax.grid()
106
107     handles: List[mpatches.Patch] = []
108     ax.scatter([point[0] for point in dataset], [point[1] for point in
     dataset], marker='x', color='red')
109     for solver, color in zip(solvers, colors):
110         ax.plot(t, [solver.predict(x) for x in t], color=color)
111         handles.append(mpatches.Patch(color=color, label=f'Degree {
    solver._degree}'))
112     ax.legend(handles=handles)
113     plt.tight_layout()
114     plt.savefig('images/degrees.png', dpi=300)
115     plt.show()
116
117 def _estimate_relative_error(dataset: Dataset, solver:
    LeastSquaresSolver) -> np.floating:
118     """ Estimates the relative error of the fitted polynomial
119
120     Args:
121         dataset (Dataset): A list of points
122         solver (LeastSquaresSolver): A LeastSquaresSolver instance
123
124     Returns:
125         np.floating: Relative error
126     """
127     # Set of y real values
128     y = np.array([point[1] for point in dataset])
129     # Set of y predicted values
130     y_hat = np.array([solver.predict(point[0]) for point in dataset])
131
132     return np.linalg.norm(y - y_hat) / np.linalg.norm(y)
133
134 if __name__ == '__main__':
135     dataset = [
136         (0.8, 1.97),
137         (1.2, 3.53),
```

```
        (1.6, 3.57),
        (2.0, 2.42),
        (2.4, 2.44),
        (2.8, 1.30),
        (3.2, 0.71),
        (3.6, 2.12),
        (4.0, 3.08),
        (4.4, 5.99)
    ]

    # Using Least Squares Solver to find the coefficients of the polynomial
    solver = LeastSquaresSolver(degree=3)
    coefficients = solver.fit(dataset)

    # Showing the plot of the dataset and the fitted polynomial of different degrees
    _show_plot_of_different_degrees(dataset)

    # Showing the predictions of the fitted polynomial
    _show_predictions(dataset, solver)

    # Showing the coefficients of the fitted polynomial
    _print_predicted_polynomial(coefficients)

    # Showing the plot with the dataset and the fitted polynomial
    _show_plot(dataset, solver)

    # Estimating the relative error of the fitted polynomial
    relative_error = _estimate_relative_error(dataset, solver)
    print('Relative error: {:.4f}'.format(float(relative_error)))
```

Лістинг 2: Запуск методу найближчих квадратів

# 4 Результати

## 4.1 Таблиця

Спочатку, побудуємо таблицю, як і просили в умові

*Degree vs real values for a linear regression*

| x | y | f(x) | \|y − f(x)\| |
|---|---|---|---|
| 0.80000000 | 1.97000000 | 2.15506294 | 0.18506294 |
| 1.20000000 | 3.53000000 | 3.24534266 | 0.28465734 |
| 1.60000000 | 3.57000000 | 3.37650816 | 0.19349184 |
| 2.00000000 | 2.42000000 | 2.88956177 | 0.46956177 |
| 2.40000000 | 2.44000000 | 2.12550583 | 0.31449417 |
| 2.80000000 | 1.30000000 | 1.42534266 | 0.12534266 |
| 3.20000000 | 0.71000000 | 1.13007459 | 0.42007459 |
| 3.60000000 | 2.12000000 | 1.58070396 | 0.53929604 |
| 4.00000000 | 3.08000000 | 3.11823310 | 0.03823310 |
| 4.40000000 | 5.99000000 | 6.08366434 | 0.09366434 |

*Predicted polynomial*

| Degree | Coefficient |
|---|---|
| x^0 | −4.2668 |
| x^1 | 12.4144 |
| x^2 | −6.1941 |
| x^3 | 0.8880 |

Relative error: 0.1019

Рис. 1: Таблиця з результатами

Бачимо, що поліном має вигляд:

$$\mathbf{w} \approx \begin{bmatrix} -4.267 \\ 12.414 \\ -6.194 \\ 0.888 \end{bmatrix} \implies f(x;\mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(x) \approx 0.888x^3 - 6.194x^2 + 12.414x - 4.267$$

Відносна похибка вийшла $\Delta \approx 10.2\%$.

## 4.2 Графік

Графік, на якому ми зобразили наш побудований поліном:

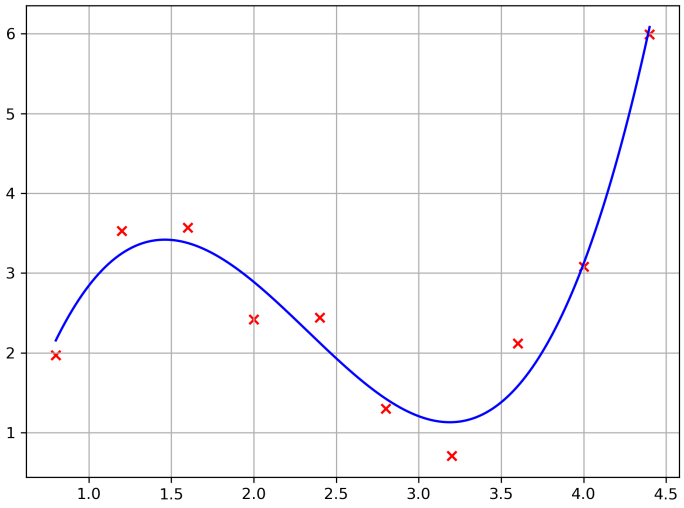


Рис. 2: Синім ми зобразили графік $f(x; \mathbf{w})$, а червоним набір данних

Для перевірки, додатково побудуємо поліноми різних ступенів:

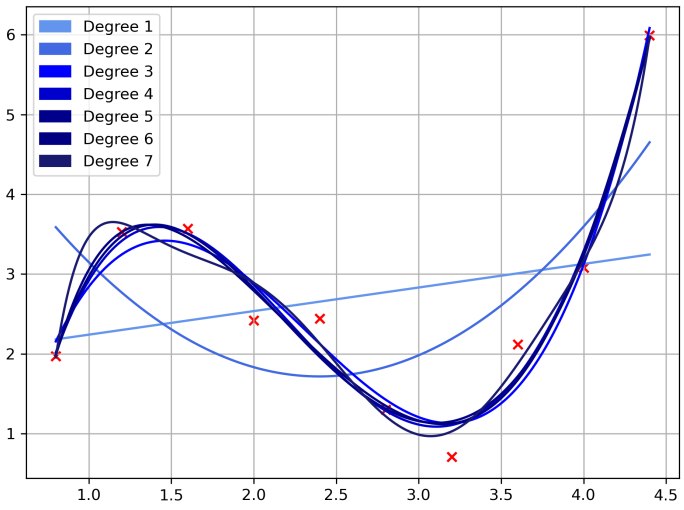


Рис. 3: Різними відтінками синього ми помітили різні ступені $f(x; \mathbf{w})$

# 5 Висновки

В цій лабораторній роботі ми:

- навчилися будувати поліном, котрий апроксимує набір точок за допомогою методу найближчих квадратів.
- навчилися писати комп'ютерну програму (на прикладі мови `Python`), що будує поліном за методом найближчих квадратів з можливістю обрання ступені поліному та задаванням довільного розміру початкових данних.
- Оцінювати написану програму (ми це зробили у випадку $\deg f(x; \mathbf{w}) = 3$).
- Візуалізовувати графік та набір точок.

Щодо *LSE* методу можемо зробити наступні зауваження:

1. В порівнянні з минулими методами інтерполяції (поліном Лагранжа, сплайни тощо), точність на заданому наборі виявилась меньшою. Проте, це ніяк не каже про те, що метод найменьших квадратів є гіршим. Навпаки – на великому наборі данних, точна інтерполяція часто дає дуже нестабільні результати на проміжках між вузлами, оскільки для інтерполяції нам потрібно поліном ступеня $n_{\mathcal{D}} - 1$, навіть якщо ми дуже впевнені про, скажімо, лінійну залежність $y(x)$.
2. Для *LSE* методу ми розв'язуємо лінійну систему $\mathbf{X}^\top \mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^\top \mathbf{y}$ за допомогою знаходження оберненої матриці $(\mathbf{X}^\top \mathbf{X})^{-1}$, проте з цією операцією є безліч проблем. Існує багато інших більш точних чисельних методів лінійної алгебри для цього. Як альтернатива, найбільш очевидний спосіб – це метод градієнтного спуску. У найпростішому випадку, ми на кожному кроці знаходимо:

$$\mathbf{w}^{\langle i+1 \rangle} \leftarrow \mathbf{w}^{\langle i \rangle} - \eta \frac{\partial \mathcal{L}(\mathbf{w}^{\langle i \rangle} \mid \mathcal{D})}{\partial \mathbf{w}}, \ \eta \ll 1$$

3. Для цього конкретно набору данних, значно меньшу похибку дала би степінь полінома 8. Проте, з цього набору точок видно, що скоріше за все залежність кубічна. Хоча поліном 8 ступеня і ближче підігнав криву до них, це б не означало, що якщо ми додамо до набору інші точки, то наша функція би їх добре наближала. У машинному навчанні таке явище називають *overfitting*'ом.