Лабораторна робота 2 - Лінійна та поліноміальна регресія.

Завдання: одне з безлічі завдань, яким займається сучасна фізика - це пошук матеріалу для виготовлення надпровідника, що працює за кімнатної температури. Крім теоретичних методів є і підхід з боку статистики, який передбачає аналіз бази даних матеріалів для знаходження залежності критичної температури від інших фізичних характеристик. Саме цим Ви і займетеся.

У файлі data.csv міститься весь датасет. Разом маємо 21 тисячу рядків і 169 колонок, з яких перші 167 - ознаки, колонка critical_temp містить величину, яку треба передбачити. Колонка material - містить хімічну формулу матеріалу, її можна відкинути.

Реалізуйте методи з позначкою #TODO класу PolynomialRegression:

Метод preprocess повинен виконувати таке перетворення:

$$X = [\,x_{i,j}\,]_{m \times n}$$

$$preprocess(X)$$

$$= \begin{bmatrix} 1 & x_{1,1} & \dots & x_{1,1} & x_{1,1}^2 & \dots & x_{1,1}^2 & \dots & x_{1,1}^p & \dots & x_{1,1}^p \\ 1 & x_{2,1} & \dots & x_{2,n} & x_{2,1}^2 & \dots & x_{2,n}^2 & \dots & x_{2,1}^p & \dots & x_{2,n}^p \\ \vdots & & & & & & & & & & \\ 1 & x_{m,1} & \dots & x_{m,n} & x_{m,1}^2 & \dots & x_{m,n}^2 & \dots & x_{m,1}^p & \dots & x_{m,n}^p \end{bmatrix}_{m,N}$$

де p - ступінь полінома (self.poly_deg у коді). Таким чином, preprocess додає поліноміальні ознаки до X. Метод J має обчислювати оціночну функцію регресії:

$$J(\theta) = MSE(Y,\, h_\theta(X)) + \alpha_1 \sum_{i=1}^{N} \sum_{j=1}^{k} |\hat{\theta}_{i,j}| + \alpha_2 \sum_{i=1}^{N} \sum_{j=1}^{k} \hat{\theta}_{i,j}^2$$

Метод `grad` має обчислювати градієнт $\frac{\partial J}{\partial \theta}$:

$$\frac{\partial J}{\partial \theta} = -\frac{2}{m} X^T (Y - X\theta) + \begin{bmatrix} 0 & & \\ & 1 & \\ & & \ddots \\ & & & 1 \end{bmatrix} \times (\alpha_1 sign(\theta) + 2\alpha_2 \theta)$$

Метод `moments` має повертати вектор-рядки μ, σ для середнього і стандартного відхилення кожної колонки. Пам'ятайте, що колонку з одиницями не потрібно нормалізувати, тож відповідні середнє і стандартне відхилення для неї вкажіть рівними 0 і 1 відповідно. Можна використовувати функції np.mean и np.std.

Метод `normalize` має виконувати нормалізацію X на основі статистик μ, σ, що повернув метод **moments**. Для того щоб уникнути ділення на 0, можете до σ додати маленьку величину, наприклад 10^{-8}.

Метод `get_batch` повинен повертати матриці X_b, Y_b з довільно обраних b елементів вибірки (b у коді - `self.batch_size`).

Метод `fit` виконує оптимізацію $J(\theta)$. Для кращої збіжності реалізуйте алгоритм оптимізації **Momentum**:

$$v_t = \gamma v_{t-1} + \alpha \nabla J(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - v_t$$

де γ встановіть рівним 0.9 (можете поекспериментувати з іншими величинами), $v_1 = [0]_{N,k}$.

Код класу PolynomialRegression:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import seaborn as sns
import pandas as pd
import random
from sklearn.utils import shuffle
import os
from datetime import datetime
from datetime import timedelta
class PolynomialRegression:
    def __init__(
        self,
        # alpha1,
        # alpha2,
        poly_deg,
        # learning_rate,
        # batch_size,
        # train_steps
    ):
        # self.alpha1 = alpha1
        # self.alpha2 = alpha2
        self.poly_deg = poly_deg
        # self.learning_rate = learning_rate
        # self.batch_size = batch_size
        # self.train_steps = train_steps
```

```
    def preprocess(self, x): # TODO
        PolynomialX = [x]
        for degree in range(2, self.poly_deg + 1):
            PolynomialX.append(x ** degree)
        newx_poly = np.concatenate((PolynomialX[0], np.ones((x.shape[0], 1))), axis = 1)
        newx_poly[:, :1:] = 1
        return newx_poly
        # previous preprocess (it works slowly)
        """CountOfRows=len(x)
        CountOfColumns=len(x[0])
        NewX=np.zeros((CountOfRows,CountOfColumns*self.poly_deg+1))
        for i in range(CountOfRows):
            NewX[i][0]=1
            l=1
            for j in range(CountOfColumns):
                NewX[i][j+l]=x[i][j]
            k=CountOfColumns+1
            for j in range(2,self.poly_deg+1):
                for l in range(CountOfColumns):
                    NewX[i][k]=x[i][l]**j
                    k+=1
        return NewX"""
```

```
    def normalize(self, x): # TODO
        # Z-масштабування даних на основі середнього значення та стандартного відхилення:
        ділення різниці між змінною та середнім значенням на стандартне відхилення.
        CountOfRows=len(x)
        CountOfColumns=len(x[0])
        VerySmallNumber=pow(10,-8)
        for i in range(CountOfRows):
            for j in range(CountOfColumns):
                x[i][j]=((x[i][j]-self.mu[j]))/(self.sigma[j]+VerySmallNumber)
        return x
    def moments(self, x): # TODO
        CountOfRows=len(x)
        CountOfColumns=len(x[0])
        MeanDeviations=[0]
        StandardDeviations=[1]
        for i in range(1,CountOfColumns):
            column=[]
            for j in range(CountOfRows):
                column.append(x[j][i])
            MeanDeviations.append(np.mean(column,axis=0))
            StandardDeviations.append(np.std(column,axis=0))
        return [MeanDeviations,StandardDeviations]
```

```
    def get_batch(self, x, y, batch_size): # TODO
        RandomIndexes=np.random.randint(len(x),size=batch_size)
        return x[RandomIndexes],y[RandomIndexes]
        # previous get_batch 2 (it works faster)
        """XSize=len(x)
        RandomIndexes=np.array([i for i in range(XSize)])
        random.shuffle(RandomIndexes)
        return np.array([x[RandomIndexes[i]] for i in
        range(batch_size)]),np.array([y[RandomIndexes[i]] for i in range(batch_size)])"""
        # previous get_batch 1 (it works slowly)
        """XSize=len(x)
        YSize=len(y)
        XBatch=np.zeros((self.batch_size,len(x[0])))
        YBatch=np.zeros((self.batch_size,len(y[0])))
        RandomIndexes=[]
        for i in range(XSize):
            RandomIndexes.append(i);
        for i in range(XSize):
            j=random.randrange(0,XSize)
            t=RandomIndexes[i]
            RandomIndexes[i]=RandomIndexes[j]
            RandomIndexes[j]=t
        for i in range(self.batch_size):
            XBatch[i]=x[RandomIndexes[i]]
            YBatch[i]=y[RandomIndexes[i]]
        return [XBatch,YBatch]"""
```

```
    def grad(self, x, y, theta, alpha1, alpha2): # TODO
        return ((2/len(x)*-1)*x.T@(y-(x@theta)))+(alpha1*np.sign(theta)+2*alpha2*theta)
    def J(self, x, y, theta, alpha1, alpha2): # TODO
        prev_y=x@theta
        mse=np.square(np.subtract(y,prev_y)).mean()
        return mse+self.alpha1*sum(sum(abs(theta)))+self.alpha2*sum(sum(theta**2))
```

```
    def PrepareX(self,x,batch_size):
        x = self.preprocess(x)
        self.mu, self.sigma = self.moments(x)
        x = self.normalize(x)
        try:
            assert np.allclose(x[:, 1:].mean(axis=0), 0, atol=1e-3)
            assert np.all((np.abs(x[:, 1:].std(axis=0)) < 1e-2) | (np.abs(x[:, 1:].std(axis=0
            ) - 1) < 1e-2))
        except AssertionError as e:
            print('Something wrong with normalization')
            raise e
        x_batch, y_batch = self.get_batch(x, y, batch_size)
        try:
            assert x_batch.shape[0] == batch_size
            assert y_batch.shape[0] == batch_size
        except AssertionError as e:
            print('Something wrong with get_batch')
            raise e
        return x
    def fit(self, x, y, alpha1, alpha2, learning_rate, batch_size, train_steps):
        (m, N), (_, k) = x.shape, y.shape
        theta = np.zeros(shape=(N, k))
        gamma = 0.9
        v_1 = np.zeros(shape=(N, k)) # TODO
        v_t = v_1
        for step in range(train_steps):
            x_batch, y_batch = self.get_batch(x, y, batch_size)
            theta_grad = self.grad(x_batch, y_batch, theta,alpha1, alpha2)
            # TODO Update v_t and theta
            v_t = gamma * v_t + learning_rate * theta_grad
            theta = theta - v_t
        self.theta = theta
        return self
```

```
    def predict(self, x):
        x = self.preprocess(x)
        x = self.normalize(x)
        return x @ self.theta
    def score(self, x, y):
        y_pred = self.predict(x)
        return np.abs(y - y_pred).mean()
```

Опис проведених досліджень

Запустивши код із стандартними параметрами ми отримали наступний вивід:

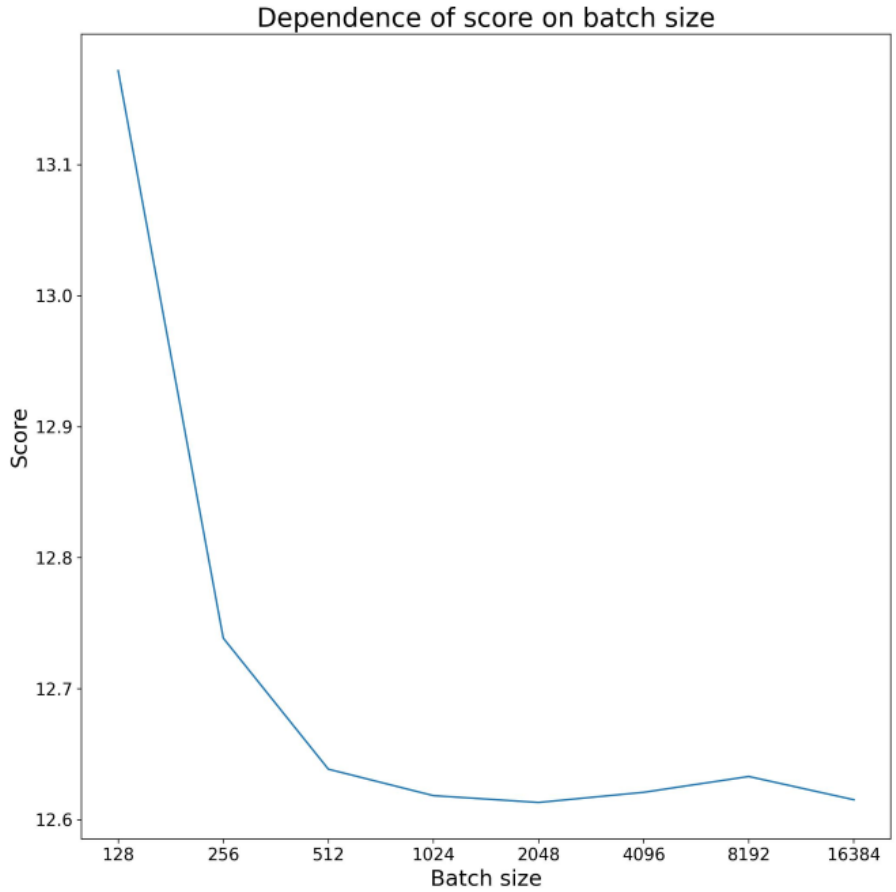
```
Processing... Score:  12.63077 ; time:  0 : 0 : 4
```

У рамках даної лабораторної роботи ми також виконали численні експерименти з різними значеннями таких основних параметрів моделі як ступінь поліноміальності (poly_deg), коефіцієнти alpha1 та alpha2, коефіцієнт швидкості навчання (learning rate), розмір частини вибірки (batch_size), кількість кроків навчання (train_steps). Вони підбирались так, щоб побачити межу між недостатнім навчанням та перенавчанням.

Ми створили декілька версій методів preprocess та get_batch, задіявши під час реальних випробувань їхні найбільш швидкодіючі версії. Також слід відмітити зміну конструктора та виокремлення деяких підготовчих інструкцій у процедуру PrepareX з метою оптимізації часових витрат на попередню підготовку даних, яка виконується лише один раз перед певним набором експериментів. Окрім тестів з різними параметрами програма також містить код для визначення часових витрат,

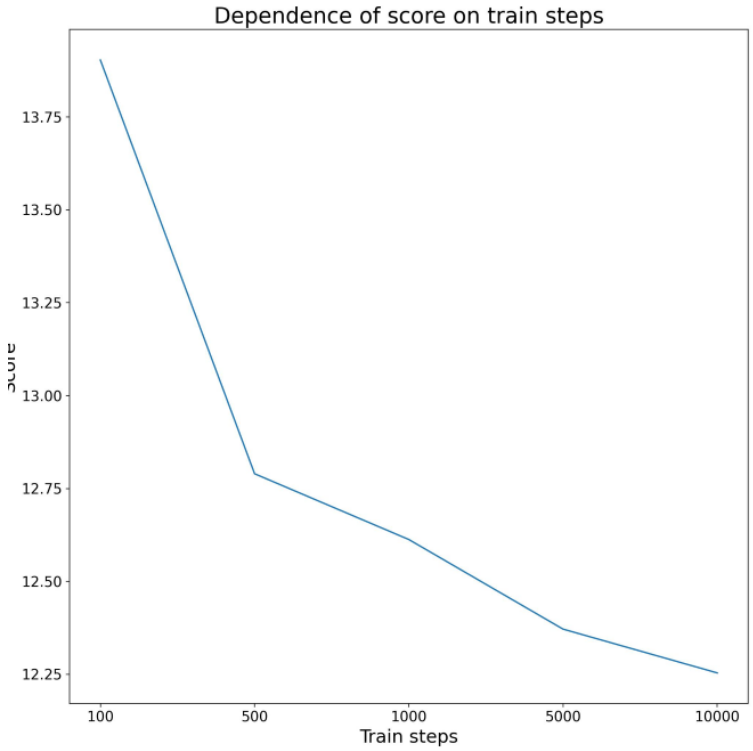
фіксації результатів у вигляді графіків, діаграм різних типів та текстового csv-файлу, який при необхідності може бути проаналізований табличним процесором.

Тестування створеного рішення проводилось на ПК з процесором Intel Core i5-6600 без використання окремого GPU та хмарних сервісів. Тому деякі набори експериментів виконувались у послідовності за спаданням обчислювальної складності, використовуючи в кожній наступній серії експериментів оптимальні параметри з попередніх. Спочатку ми проаналізували залежність score від різних розмірів частини вибірки (batch_size):

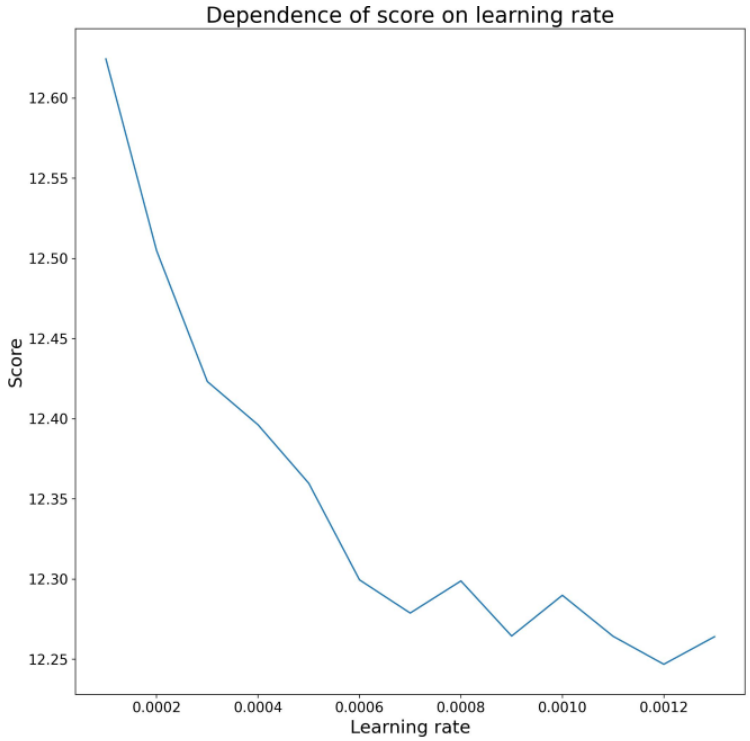


Можна зробити висновок, що найкращим значенням є 2048, проте при подальших експериментах, можливо, знайдеться більш кращий розмір.

Після цього, вже з використанням кращого batch_size, був проаналізований параметр train_steps (кількість кроків навчання). Звісно, чим він більший, тим точнішою є модель, проте визначення можливої межі початку перенавчання потребує значних обчислювальних потужностей або великих часових витрат:

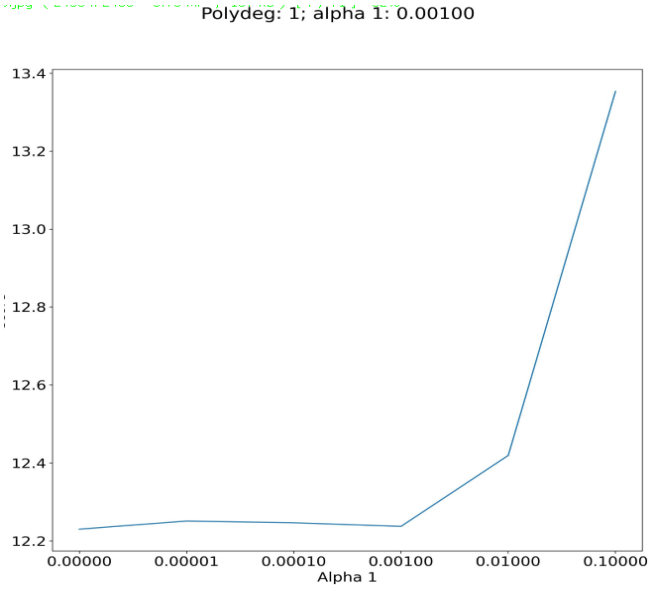


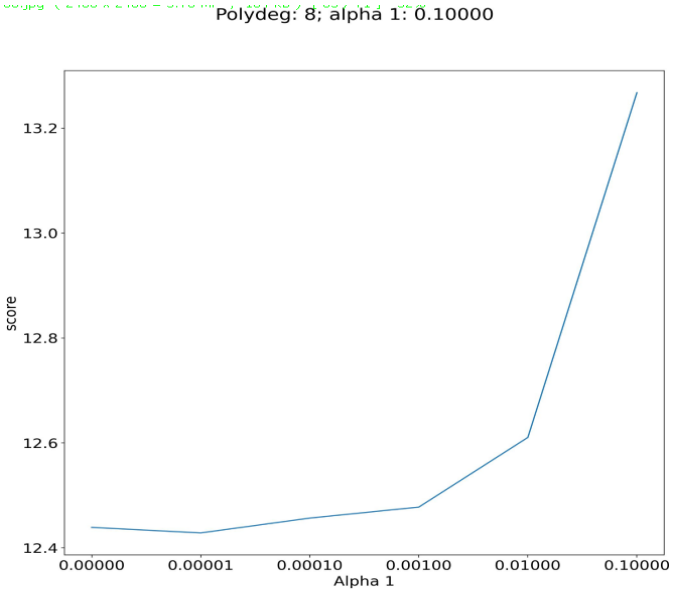
Після цього були досліджені різні значення learning_rate (коефіцієнт швидкості навчання):



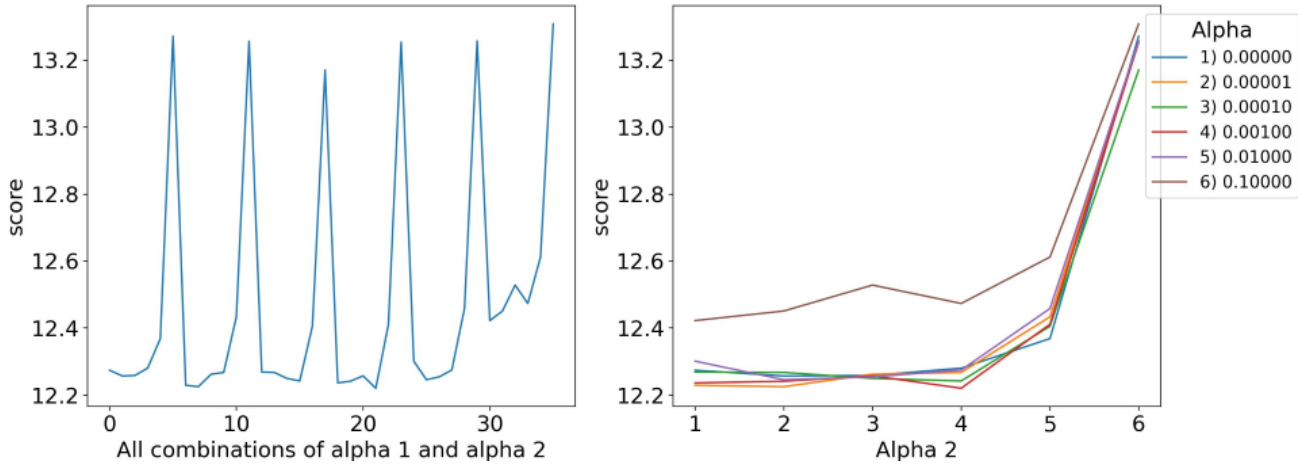
Найкращим значенням із досліджених виявилось 0.0012. Можна стверджувати, що його збільшення призводить до покращення моделі, проте визначення межі, де починається стрімке перенавчання, потребує більшої кількості додаткових експериментів.

Після цього експериментальним шляхом було встановлено, що коефіцієнт поліноміальності (poly_deg) більший ніж 6 призводить до перенавчання, тобто до збільшення score. Ми намагались підібрати його у різних комбінаціях разом із параметрами alpha1 та alpha2, отримавши схожі один на одного графіки та діаграми:

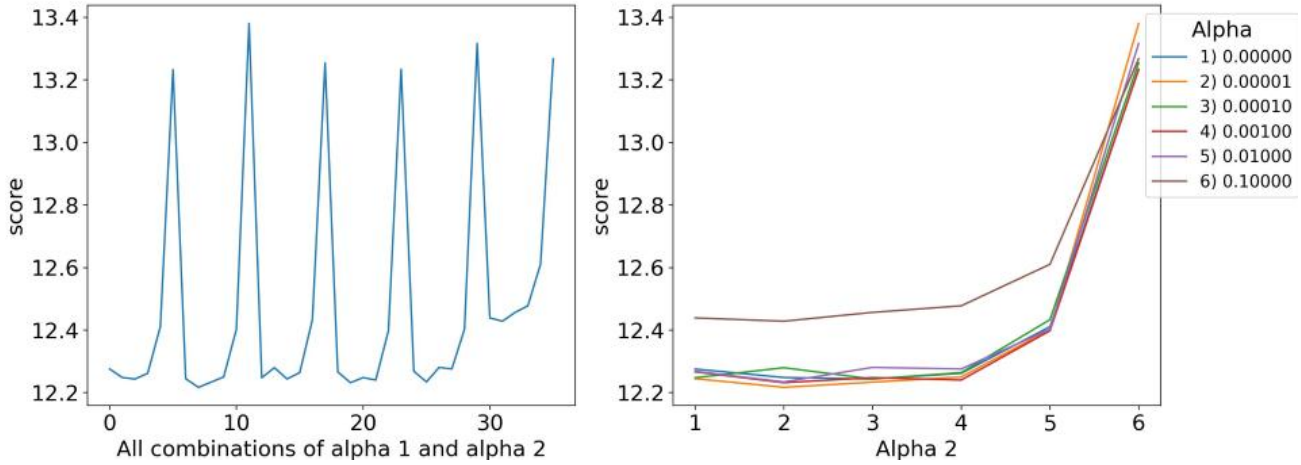


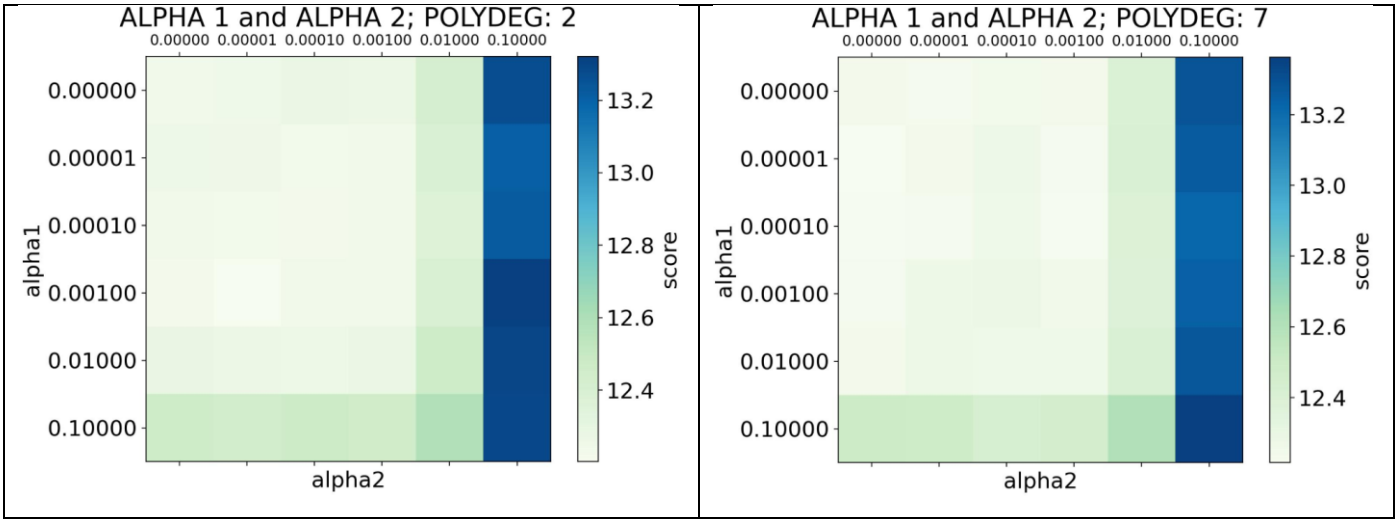


Poly deg: 3



Poly deg: 8



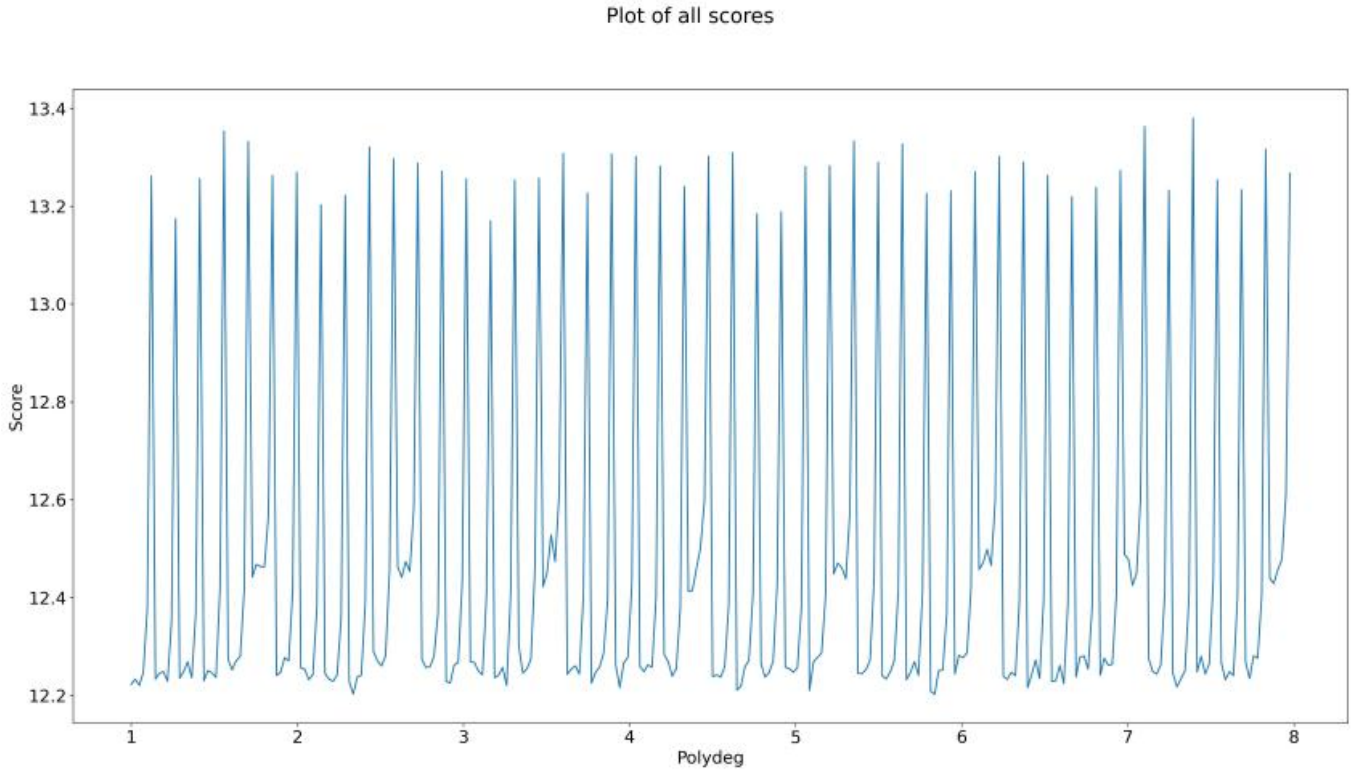


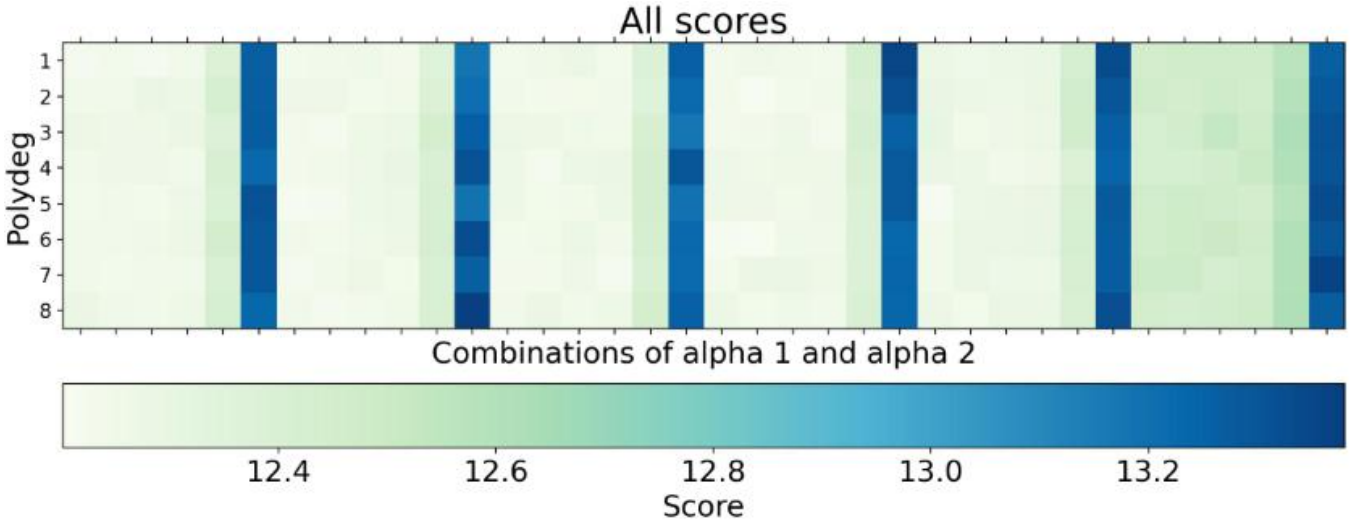
Можна зробити висновок, що оптимальні значення alpha1 та alpha1 знаходяться в межах до 0.01, вище – стрімке зменшення якості моделі. Найкращим alpha1 виявилось 0.001, alpha2 – 0.00001, poly_deg – 6.

Найгіршим значенням score є 13.90356, отримане при таких параметрах:

poly_deg	alpha1	alpha2	learning_rate	batch_size	train_steps
1	0.00000	0.00000	0.001	128	100

Наведемо глобальний графік та діаграму типу matshow для всіх комбінацій коефіцієнтів (результат пошуку параметрів alpha1 та alpha2 разом з poly_deg):





Можна зробити висновок, що коефіцієнти alpha1 та alpha2, які використовуються при обчисленні градієнту, є досить впливовими параметрами.

Після завершення роботи програми ми отримуємо наступний вивід на екран:

```
287  /  288 ; alpha1:  0.10000 ; alpha2:  0.01000... Time:  0 : 0 : 21 ; score:  12.61061
288  /  288 ; alpha1:  0.10000 ; alpha2:  0.10000... Time:  0 : 0 : 21 ; score:  13.26785
--------------------------------------------------------------------------------
Min score:  12.20193 ; polydeg:  6; alpha 1:  0.00100 ; alpha 2:  0.00001
learning rate:  0.0012000000000000003 ; batch size:  2048 ; train steps:  10000
--------------------------------------------------------------------------------
Max score:  13.90356 ; polydeg:  1; alpha 1:  0.00000 ; alpha 2:  0.00000
learning rate:  0.001 ; batch size:  128 ; train steps:  100
--------------------------------------------------------------------------------
Calculation the dependence of the predicted critical temperature on the true one...
Time:  0 : 0 : 25 ; Test MAE: 12.24457
Total time of all calculations:  1 : 49 : 45
Drawing plots... Time:  0 : 0 : 12
```

Візуалізація залежності передбаченої критичної температури від істинної (обчислення виконані з параметрами кращої моделі):