Лабораторна робота 3 - Логістична регресія.

У цій роботі Ви побудуєте логістичну регресію для класифікації зображень рукописних символів за датасетом MNIST.

Реалізуйте методи з позначкою #TODO класу LogisticRegression:

Метод preprocess повинен додавати колонку з одиниць у матрицю X. Опціонально – додайте поліноміальні або будь-які інші нелінійні ознаки.

Метод onehot повинен виконувати onehot-перетворення:

$$onehot: \mathbb{R} \to \mathbb{R}^c$$
$$\overline{onehot(y_i)}_j = \begin{cases} 1, & j = y_i \\ 0, & j \neq y_i \end{cases}$$

де c – кількість класів. Метод h - гіпотеза:

$$h(X) = \sigma(X\theta)$$
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Метод `J` повинен обчислювати оціночну функцію логістичної регресії:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{c}(-y_{i,j}\cdot\log(h(x_i)_j) - (1 - y_{i,j})\cdot\log(1 - h(x_i)_j) + \alpha_1\sum_{i=1}^{N}\sum_{j=1}^{c}|\hat{\theta}_{i,j}| + \alpha_2\sum_{i=1}^{N}\sum_{j=1}^{c}\hat{\theta}_{i,j}^2$$

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

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

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 і 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` має 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), $v_1 = [0]_{N,c}$.

Код класу LogisticRegression:

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from sklearn.metrics import confusion_matrix
import pandas as pd
import random
from sklearn.utils import shuffle
import os
from datetime import datetime
from datetime import timedelta
class LogisticRegression:
    def __init__(self,
        items=[]
        # alpha1
        # alpha2,
        # learning_rate,
        # batch_size,
        # train_steps
    ):
        self.l = items
        # self.alpha1 = alpha1
        # self.alpha2 = alpha2
        # self.learning_rate = learning_rate
        # self.batch_size = batch_size
        # self.train_steps = train_steps
```

```python
    def onehot(self, y):
        OnehotY=np.zeros((len(y),10))
        for i in range(len(y)):
            OnehotY[i][y[i]]=1
        return OnehotY
    def preprocess(self, x, poly_deg): # TODO
        PolynomialX = [x]
        for degree in range(2, 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*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,poly_deg+1):
                for l in range(CountOfColumns):
                    NewX[i][k]=x[i][l]**j
                    k+=1
        return NewX"""
```

```python
    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((batch_size,len(x[0])))
        YBatch=np.zeros((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(batch_size):
            XBatch[i]=x[RandomIndexes[i]]
            YBatch[i]=y[RandomIndexes[i]]
        return [XBatch,YBatch]"""
    def PrepareX(self,x,poly_deg):
        x = self.preprocess(x,poly_deg)
        self.mu, self.sigma = self.moments(x)
        return self.normalize(x)
```

```
    def sigmoid(self,x):
        return 1/(1+np.exp(-x))
    def h(self, x, theta): # TODO
        return self.sigmoid(x@theta)
    def grad(self, x, y, theta, alpha1, alpha2): # TODO
        return ((1/len(x)*-1)*x.T@(y-self.h(x,theta)))+(alpha1*np.sign(theta)+2*alpha2*theta)
    def fit(self, x, y, alpha1, alpha2, learning_rate, batch_size, train_steps):
        (m, n), (_, c) = x.shape, y.shape
        theta = np.zeros(shape=(n, c))
        gamma = 0.9
        v_1 = np.zeros(shape=(n, c)) #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,1)
        x = self.normalize(x)
        return self.h(x, self.theta).argmax(axis=1)
    def score(self, x, y):
        y_pred = self.predict(x)
        return (y == y_pred).mean() * 100
```

Опис проведених досліджень

Запустивши код із стандартними параметрами ми отримали наступний вивід:

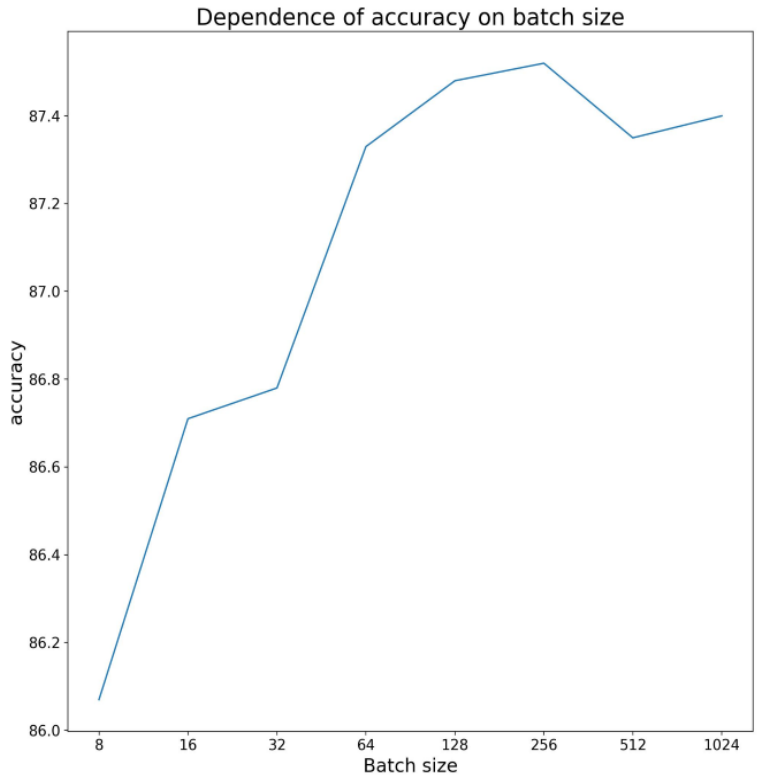
```
Processing... accuracy:  87.19000% ; time:  0 : 0 : 52
```

У рамках даної лабораторної роботи ми також виконали численні експерименти з різними значеннями таких основних параметрів моделі як коефіцієнти alpha1 та alpha2, коефіцієнт швидкості навчання (learning rate), розмір частини вибірки (batch_size), кількість кроків навчання (train_steps), будували також

поліноміальну логістичну регресію, аналізуючи параметр poly_deg. Вони підбирались так, щоб побачити межу між недостатнім навчанням та перенавчанням.

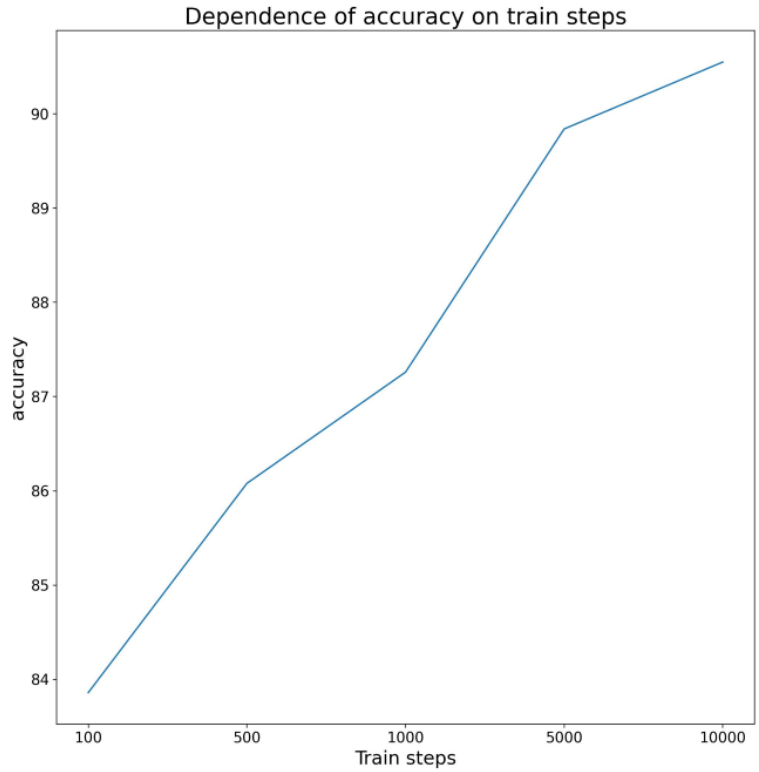
Ми створили декілька версій методів preprocess та get_batch, задіявши під час реальних випробувань їхні найбільш швидкодіючі версії. Також слід відмітити зміну конструктора та виокремлення деяких підготовчих інструкцій у процедуру PrepareX з метою оптимізації часових витрат на попередню підготовку даних, 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):

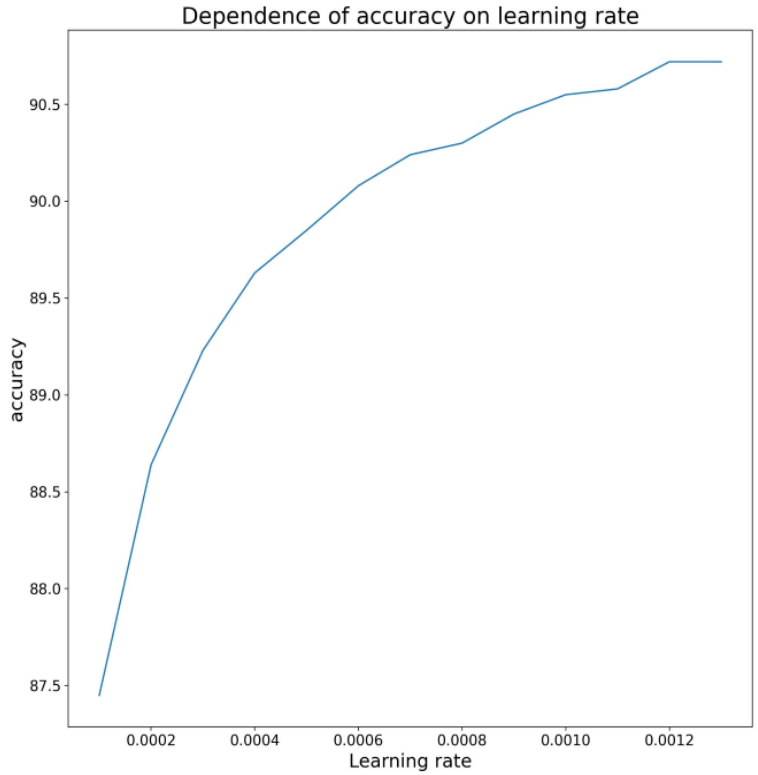


Можна зробити висновок, що найкращим значенням є 256, проте при подальших експериментах, можливо, знайдеться більш кращий розмір.

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:

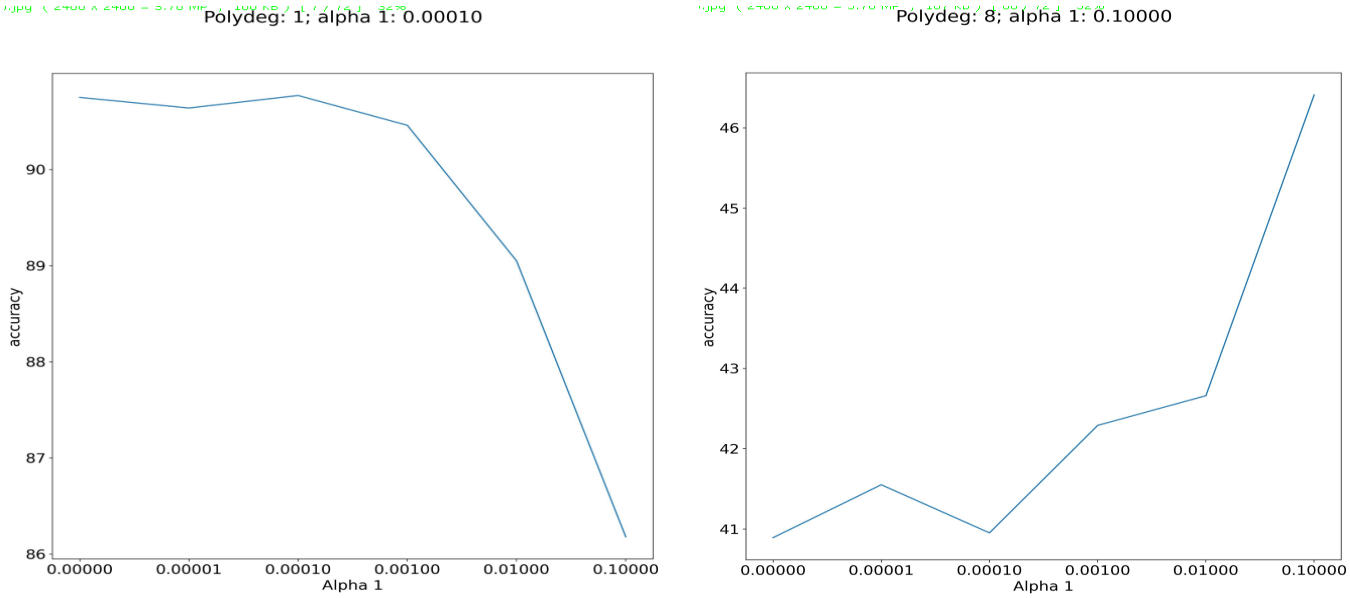


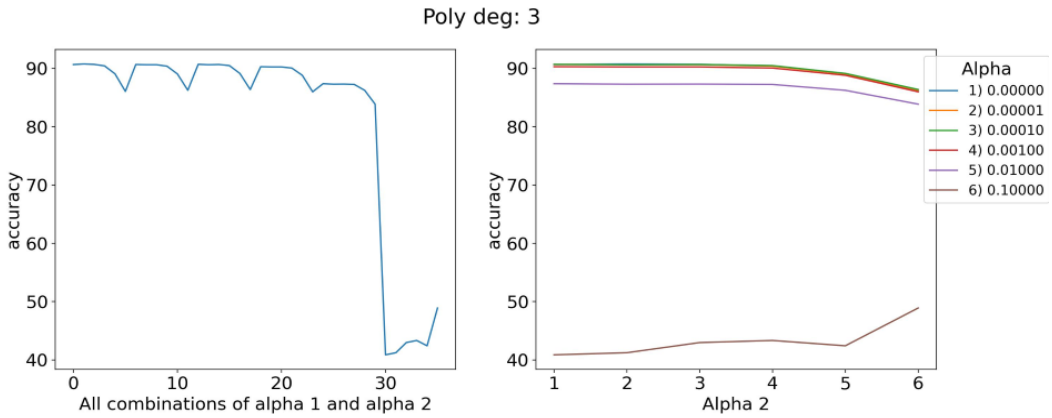
Після цього були досліджені різні значення learning_rate (коефіцієнт швидкості навчання):

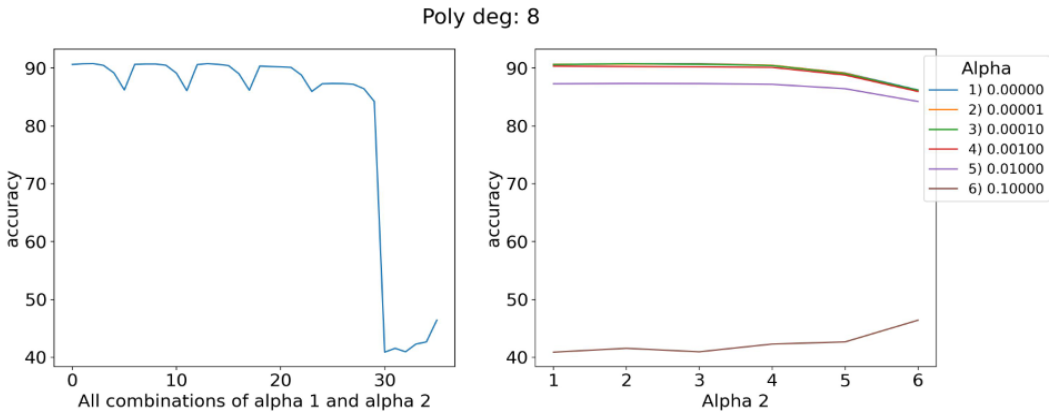


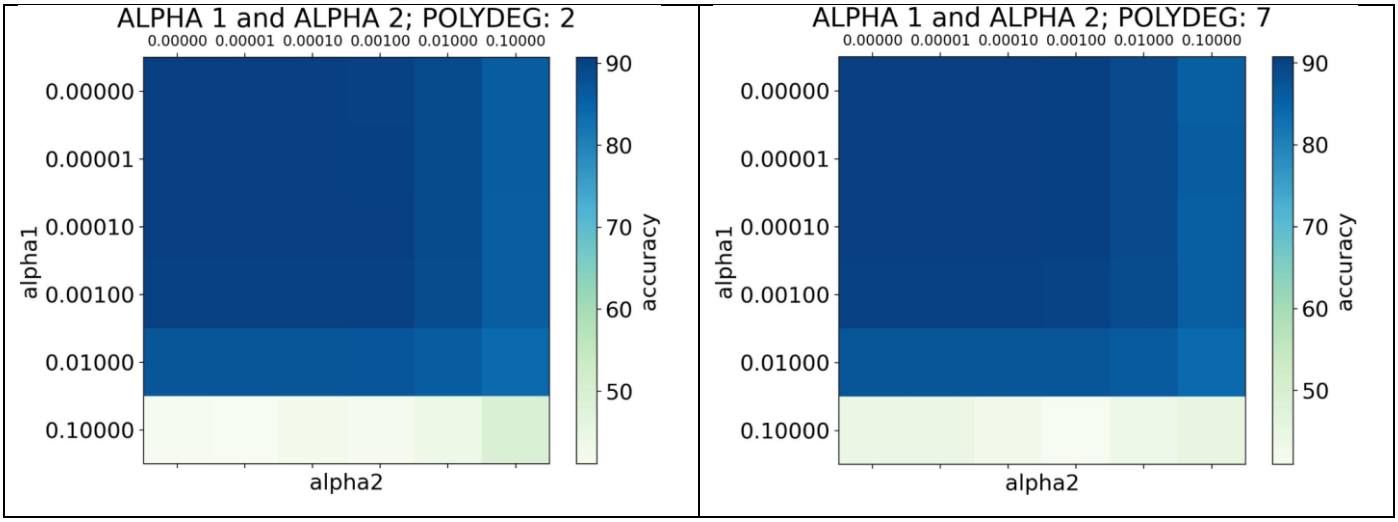
Найкращим значенням із досліджених виявилось 0.0012. Можна стверджувати, що його збільшення призводить до покращення моделі, проте визначення межі, де починається стрімке перенавчання, потребує більшої кількості додаткових експериментів.

Після цього експериментальним шляхом було встановлено, що коефіцієнт поліноміальності (poly_deg) більший ніж 5 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:







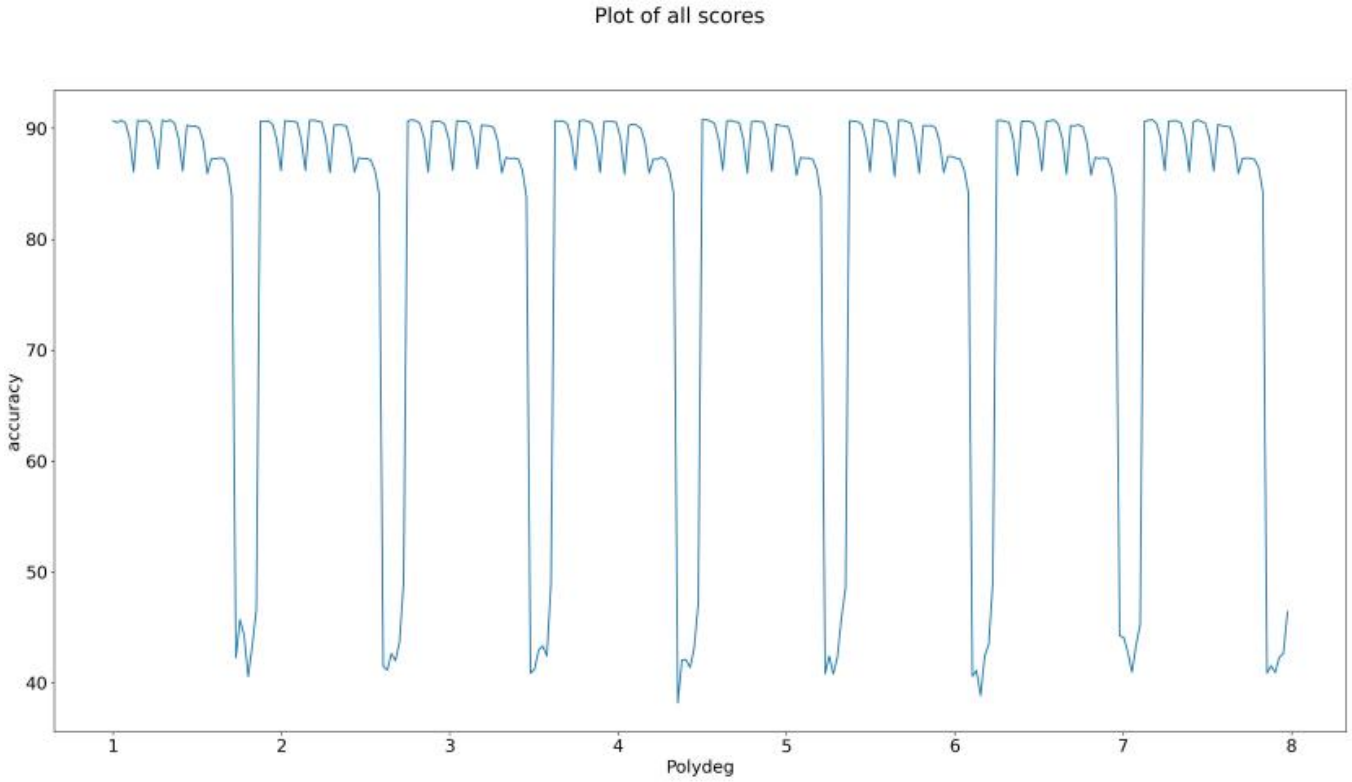


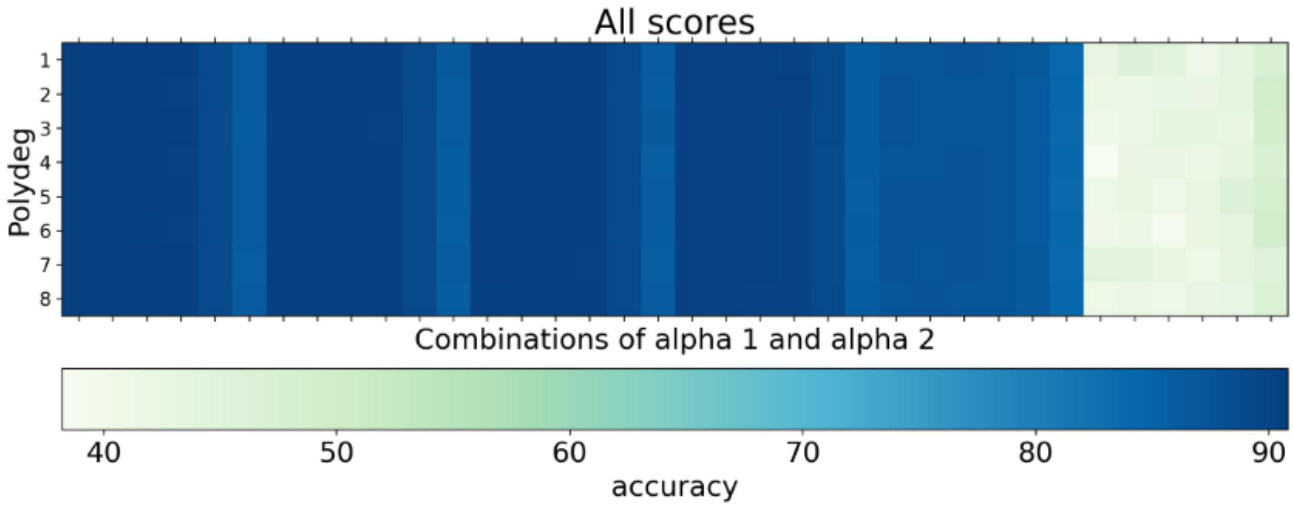
Можна зробити висновок, що оптимальні значення alpha1 та alpha1 знаходяться в межах до 0.01, вище – стрімке зменшення якості моделі, проте найкращим значенням alpha1 та alpha2 виявився 0, poly_deg – 5. Це свідчить про негативний вплив коефіцієнтів регуляризації на якість моделі, а поліноміальність покращує точність лише в певній мірі.

Найгірше значення accuracy 38.21000% було отримано при таких параметрах:

poly_deg	alpha1	alpha2	learning_rate	batch_size	train_steps
4	0.10000	0.00000	0.0010	8	100

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





Після завершення роботи програми ми отримуємо наступний вивід на екран:

```
286  /  288 ; alpha1:  0.10000 ; alpha2:  0.00100... Time:  0 : 0 : 20 ; accuracy:  42.2900
287  /  288 ; alpha1:  0.10000 ; alpha2:  0.01000... Time:  0 : 0 : 20 ; accuracy:  42.6600
288  /  288 ; alpha1:  0.10000 ; alpha2:  0.10000... Time:  0 : 0 : 20 ; accuracy:  46.4100

--------------------------------------------------------------------------------
Max accuracy:  90.82000% ; polydeg:  5; alpha 1:  0.00000 ; alpha 2:  0.00000
learning rate:  0.0012 ; batch size:  256 ; train steps:  10000
--------------------------------------------------------------------------------
Min accuracy:  38.21000% ; polydeg:  4; alpha 1:  0.10000 ; alpha 2:  0.00000
learning rate:  0.0010 ; batch size:  8 ; train steps:  100
--------------------------------------------------------------------------------
Best model... Time:  0 : 1 : 16 ; Test accuracy: 90.65000%
Total time of all calculations:  1 : 53 : 24
Drawing plots... Time:  0 : 0 : 12
```

Візуалізація матриці помилок для кращої моделі: