Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 9 по дисциплине «Методы машинного обучения»

Тема Эволюционные алгоритмы

Студент Сапожков А.М,

Группа ИУ7-23М

Преподаватель Солодовников В.И.

Москва, 2025

Содержание

ВВЕДЕНИЕ

Генетические и эволюционные алгоритмы являются разновидностью численных методов решения оптимизационных задач. Основное внимание в них уделяется использованию принципов естественного отбора, мутации и рекомбинации, имитирующих естественный процесс эволюции, для поиска оптимального или близкого к оптимальному решения сложной проблемы.

Целью данной лабораторной работы является изучение эволюционных алгоритмов на примере решения задачи кластеризации.

Задачи данной лабораторной работы:

1) описать общие этапы функционирования системы;
2) описать предлагаемый генетический алгоритм;
3) реализовать программу для кластеризации датасета MNIST;
4) оценить качество кластеризации.

1 Аналитическая часть

1.1 Общие этапы функционирования системы

Генетический алгоритм будет использоваться для решения задачи кластеризации, в частности для подбора центроидов кластеров, количество которых задано заранее.

На рисунке 1.1 представлены основные этапы работы генетического алгоритма.

Рисунок 1.1 — Основные этапы работы генетического алгоритма

2 Конструкторская часть

2.1 Генетический алгоритм

Генетический алгоритм начинается с популяции случайно выбранных потенциальных решений (индивидуумов), для которых вычисляется функция приспособленности. Алгоритм выполняет цикл, в котором последовательно применяются операторы отбора, скрещивания и мутации, после чего приспособленность индивидуумов пересчитывается. Цикл продолжается, пока не выполнено условие остановки, после чего лучший индивидуум в текущей популяции считается решением.

Начальная популяция состоит из случайным образом выбранных потенциальных решений (индивидуумов). Поскольку в генетических алгоритмах индивидуумы представлены хромосомами, начальная популяция — это, по сути дела, набор хромосом. Формат хромосом должен соответствовать принятым для решаемой задачи правилам, например это могут быть двоичные строки определённой длины. В случае аппроксимации функций хромосомой будет являться значение схожести аппроксимированного и истинного значения функции в точке. Данное значение предлагается рассчитывать по формулам

$$fitness = ARI = \frac{RI - E[RI]}{max(RI) - E[RI]}, \tag{2.1}$$

$$RI = \frac{a + b}{\binom{n}{2}}, \tag{2.2}$$

$$E[RI] = \frac{\sum_i \binom{n_i}{2} \sum_j \binom{m_j}{2}}{\binom{n}{2}}, \tag{2.3}$$

где a — количество пар элементов, которые находятся в одном кластере как в истинной, так и в прогнозируемой кластеризации, b — количество пар элементов, которые находятся в разных кластерах как в истинной, так и в прогнозируемой кластеризации, n — общее количество образцов, n_i — количество образцов в истинном кластере i, m_j — количество образцов в прогнозируемом кластере j.

Для каждого индивидуума вычисляется функция приспособленности. Это делается один раз для начальной популяции, а затем для каждого нового поколения после применения операторов отбора, скрещивания и мутации. Поскольку приспособленность любого индивидуума не зависит от всех остальных, эти вычисления можно производить параллельно.

Так как на этапе отбора, следующем за вычислением приспособленности, более приспособленные индивидуумы обычно считаются лучшими решениями, генетические алгоритмы естественно «заточены» под нахождение максимумов функции приспособленности. Если в какой-то задаче нужен минимум, то при вычислении приспособленности следует инвертировать найденное значение, например умножив его на –1.

Применение генетических операторов к популяции приводит к созданию новой популяции, основанной на лучших индивидуумах из текущей.

Оператор **отбора** отвечает за отбор индивидуумов из текущей популяции таким образом, что предпочтение отдаётся лучшим.

Оператор **скрещивания** (или рекомбинации) создаёт потомка выбранных индивидуумов. Обычно для этого берутся два индивидуума, и части их хромосом меняются местами, в результате чего создаются две новые хромосомы, представляющие двух потомков.

Оператор **мутации** вносит случайные изменения в один или несколько генов хромосомы вновь созданного индивидуума. Обычно вероятность мутации очень мала.

3 Технологическая часть

3.1 Средства реализации

В качестве языка программирования для реализации алгоритмов был выбран язык программирования Python ввиду наличия библиотек для обучения регрессионных моделей, таких как sklearn и numpy.

3.2 Реализация алгоритмов

На листинге 3.1 представлена реализация алгоритма обучения классификаторов респондентов, принимавших участие в социологическом исследовании.

Листинг 3.1 — Кластеризация с использованием генетического алгоритма

```
import numpy
import sklearn.datasets
!pip install pygad
import pygad
from sklearn.metrics import pairwise_distances
from collections import Counter
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
# %matplotlib inline

import umap
!pip install umap-learn[plot]
import umap.plot
from umap import UMAP
from sklearn.metrics import adjusted_rand_score, silhouette_score,
   davies_bouldin_score, calinski_harabasz_score

images, target = sklearn.datasets.load_digits(return_X_y=True,
   as_frame=True)
print(images.shape)

import numpy as np

X, y_true = images, target.astype(int)
X = X.astype(np.float32) / 255.0

umap_3d_embeddings = UMAP(n_components=3, random_state=7).
```

```
    fit_transform(X)

def euclidean_distance(X, Y):
  return numpy.sqrt(numpy.sum(numpy.power(X - Y, 2), axis=1))

def cluster_data(solution, solution_idx):
  global num_clusters, feature_vector_length, data
  cluster_centers = []
  all_clusters_dists = []
  clusters = []
  clusters_sum_dist = []

  for clust_idx in range(num_clusters):
    cluster_centers.append(solution[feature_vector_length*clust_idx:
       feature_vector_length*(clust_idx+1)])
    cluster_center_dists = euclidean_distance(data, cluster_centers[
       clust_idx])
    all_clusters_dists.append(numpy.array(cluster_center_dists))

  cluster_centers = numpy.array(cluster_centers)
  all_clusters_dists = numpy.array(all_clusters_dists)

  cluster_indices = numpy.argmin(all_clusters_dists, axis=0)
  for clust_idx in range(num_clusters):
    clusters.append(numpy.where(cluster_indices == clust_idx)[0])
    if len(clusters[clust_idx]) == 0:
      clusters_sum_dist.append(0)
    else:
      clusters_sum_dist.append(numpy.sum(all_clusters_dists[clust_idx
         , clusters[clust_idx]]))

  clusters_sum_dist = numpy.array(clusters_sum_dist)
  return cluster_centers, all_clusters_dists, cluster_indices,
     clusters, clusters_sum_dist

def fitness_func(ga_instance, solution, solution_idx):
  _, _, cluster_indices, _, _ = cluster_data(solution, solution_idx)
  return adjusted_rand_score(y_true, cluster_indices)

data = umap_3d_embeddings
num_clusters = 10
```

```
feature_vector_length = data.shape[1]
num_genes = num_clusters * feature_vector_length

ga_instance = pygad.GA(num_generations=500,
                       sol_per_pop=60,
                       init_range_low=-10,
                       init_range_high=20,
                       num_parents_mating=50,
                       keep_parents=40,
                       num_genes=num_genes,
                       fitness_func=fitness_func)

ga_instance.run()

best_solution, best_solution_fitness, best_solution_idx = ga_instance
   .best_solution()
print("Best solution is {bs}".format(bs=best_solution))
print("Fitness of the best solution is {bsf}".format(bsf=
   best_solution_fitness))
print("Best solution found after {gen} generations".format(gen=
   ga_instance.best_solution_generation))

cluster_centers, all_clusters_dists, cluster_indices, clusters,
   clusters_sum_dist = cluster_data(best_solution, best_solution_idx)
print(f'Clusters: {np.unique(cluster_indices)}')
print(f'ARI: {adjusted_rand_score(y_true, cluster_indices)}')
print(f'DBI: {davies_bouldin_score(data, cluster_indices)}')
print(f'Silhouette: {silhouette_score(data, cluster_indices,
   random_state=7)}')
print(f'Calinski Harabasz: {calinski_harabasz_score(data,
   cluster_indices)}')

def class_purity(y_true, y_pred, cls):
  class_mask = (y_true == cls)
  class_predictions = y_pred[class_mask]
  cluster_counts = Counter(class_predictions)
  purity = max(cluster_counts.values()) / len(class_predictions)
  return purity

def plot_clustering(title, X, y_true, y_pred, metric):
  fig, plots = plt.subplots(2, 2, figsize=(12,12))
```

```
    fig.suptitle(title)
    plt.prism()

    n_clusters = len(np.unique(y_true))
    purities = []

    ax = fig.add_subplot(2, 2, 1, projection='3d') if X.shape[1] == 3
       else plots[0, 0]
    for i in range(n_clusters):
      digit_indices = (y_true == i)
      purities.append(class_purity(y_true, y_pred, i))
      dims = [X[digit_indices, i] for i in range(X.shape[1])]
      ax.set_title('Original')
      ax.scatter(\*dims, label=f"Digit {i}")
      ax.legend()

    purities.append(np.average(purities))

    avg_dist = np.zeros((n_clusters, n_clusters))
    for i in range(n_clusters):
      for j in range(n_clusters):
        avg_dist[i, j] = pairwise_distances(
          X[y_true == i], X[y_true == j], metric=metric
        ).mean()
    avg_dist /= avg_dist.max()
    sns.heatmap(avg_dist, ax=plots[1, 0], annot=True, cmap='Reds',
       xticklabels=np.arange(n_clusters), yticklabels=np.arange(
       n_clusters))

    inner_distances = [avg_dist[i, i] for i in range(n_clusters)]
    inner_distances.append(np.average(inner_distances))
    sns.heatmap([inner_distances, purities], ax=plots[1, 1], annot=True
       , cmap='Reds', xticklabels=[*np.arange(n_clusters), 'avg'],
       yticklabels=['inner distance', 'purity'])

    n_clusters = len(np.unique(y_pred))

    ax = fig.add_subplot(2, 2, 2, projection='3d') if X.shape[1] == 3
       else plots[0, 1]
    for i in range(n_clusters):
      digit_indices = (y_pred == i)
```

```
        dims = [X[digit_indices, i] for i in range(X.shape[1])]
        ax.set_title('Prediction')
        ax.scatter(\*dims, label=f"Cluster {i}")
        ax.legend()

    plt.tight_layout()
    plt.legend(loc='best')
    plt.show()

plot_clustering('Prediction', data, y_true, cluster_indices, '
    euclidean')
ga_instance.plot_fitness()
```

4 Исследовательская часть

Для тестирования разработанного алгоритма применялась облачная платформа Google Colab, не требующая установки ПО на локальный компьютер. В качестве датасета был взят датаест MNIST, для которого был применён метод понижения размерности UMAP, чтобы сделать датасет трёхмерным и визуализировать полученные результаты.

Полученные метрики кластеризации:

— ARI: 0.848;

— DBI: 0.278;

— Silhouette: 0.795;

— Calinski-Harabasz: 10033.279;

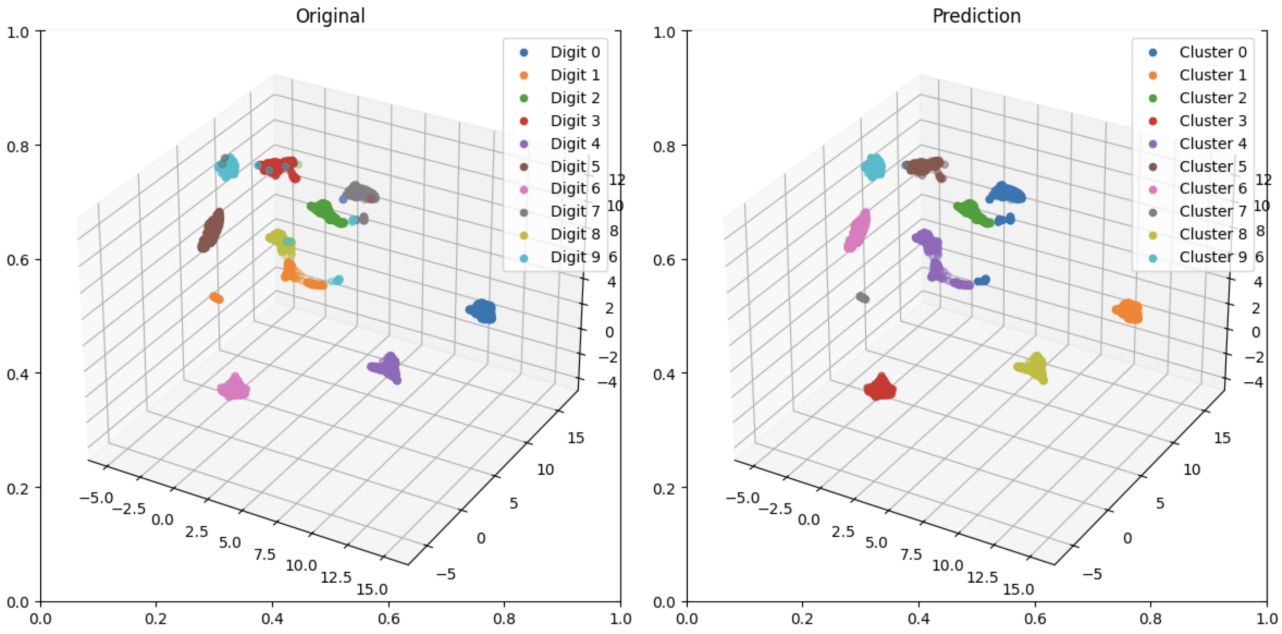

Рисунок 4.1 — Результат кластеризации в сравнении с исходным разбиением на кластеры

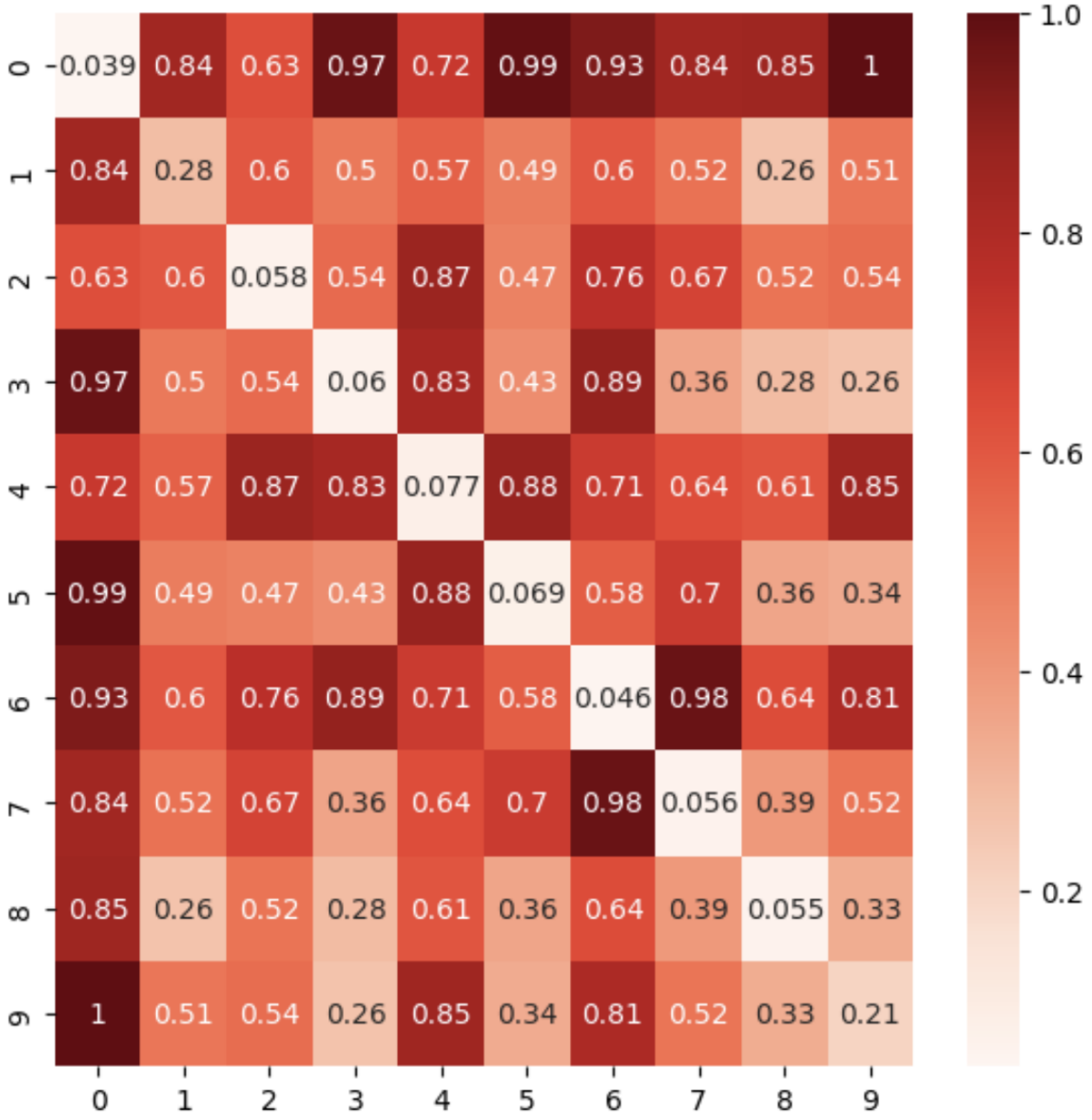

Рисунок 4.2 — Матрица внутрикластерных и межкластерных расстояний)

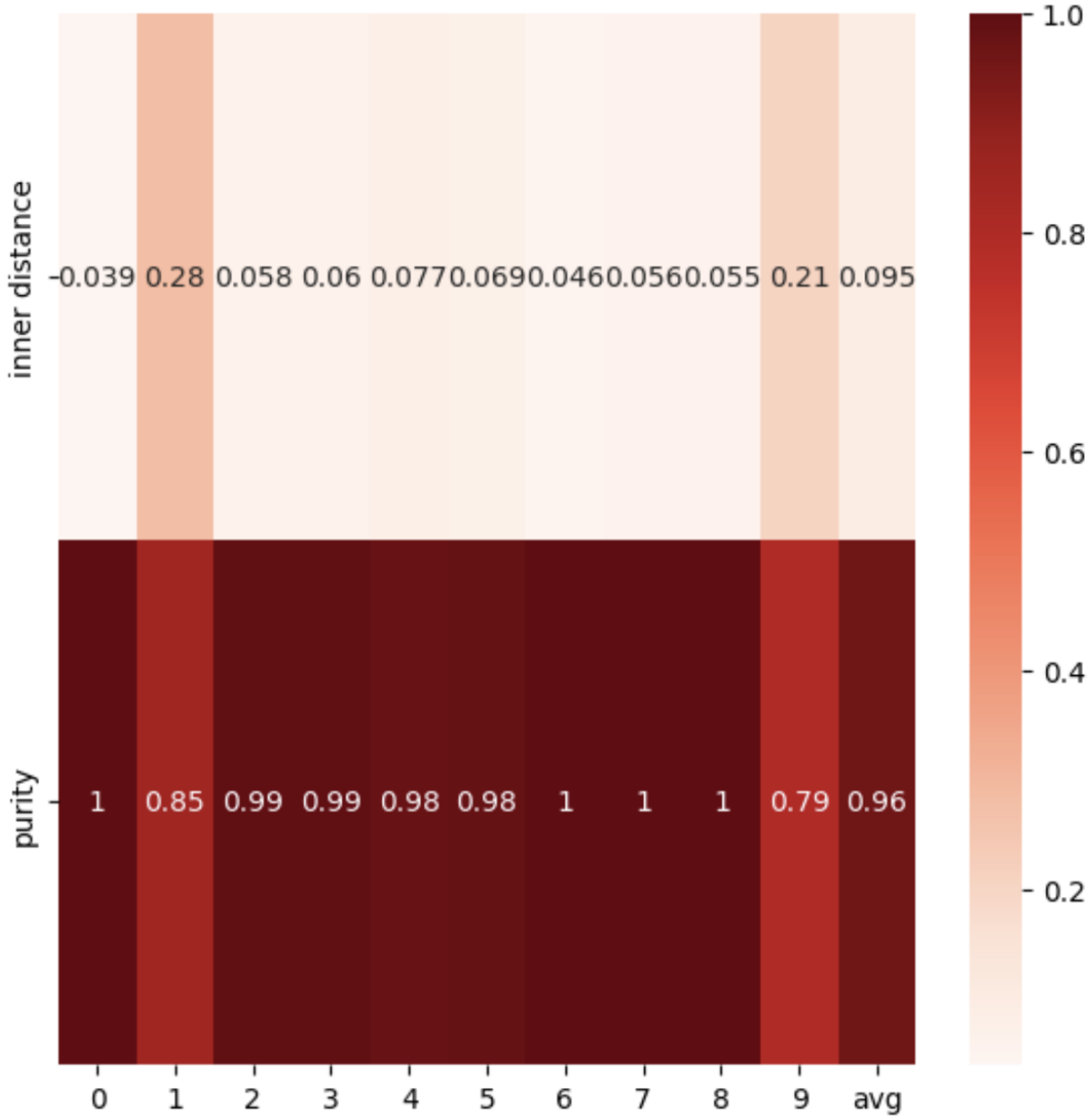

Рисунок 4.3 — Соотнесение внутрикластерных расстояний с чистотой исходных кластеров)

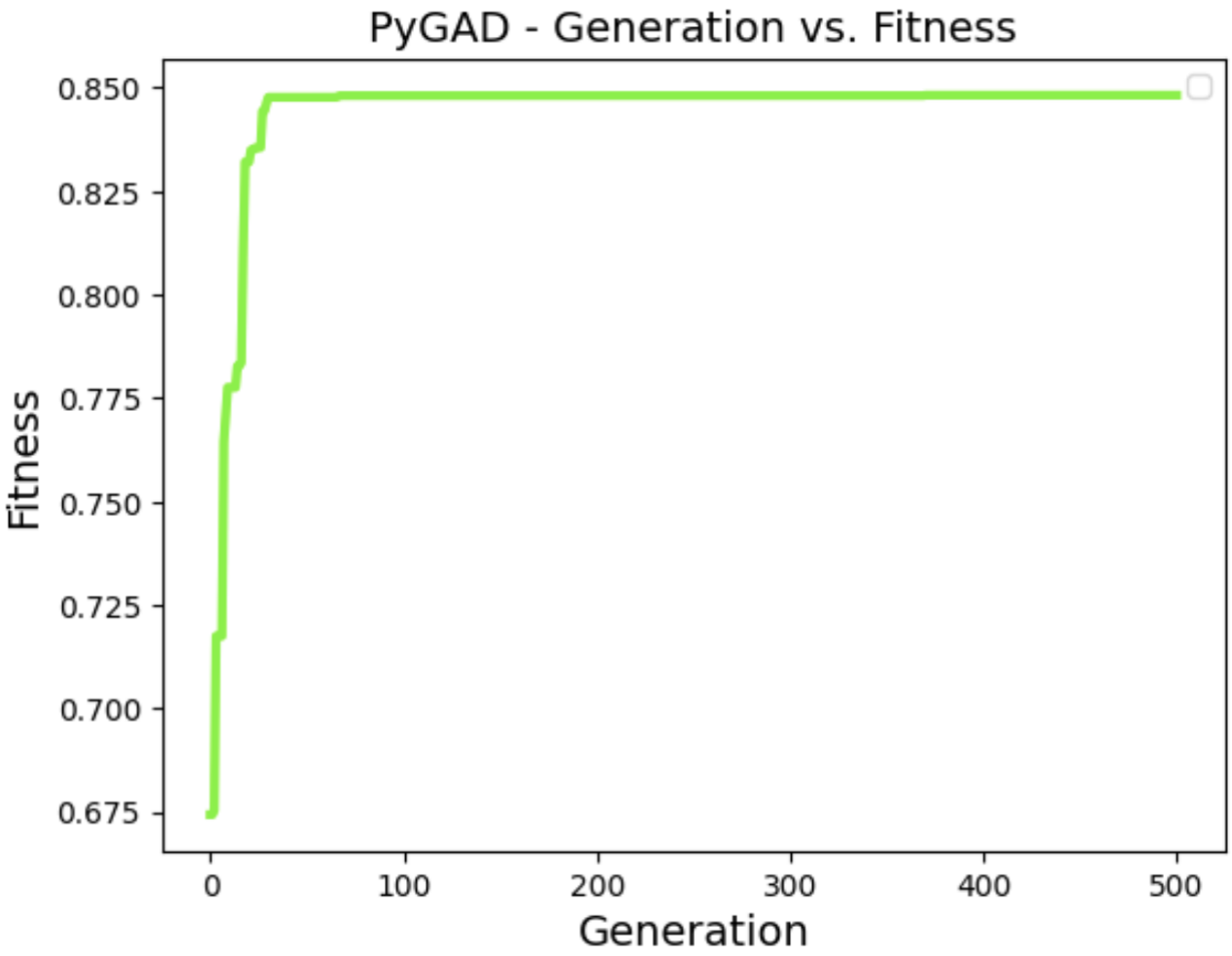

Рисунок 4.4 — График зависимости значения функции приспособленности от номера поколения)

ЗАКЛЮЧЕНИЕ

В рамках лабораторной работы было проведено изучение эволюционных алгоритмов на примере решения задачи кластеризации.

1. Описаны общие этапы функционирования системы.
2. Описан предлагаемый генетический алгоритм.
3. Реализована программа для кластеризации датасета MNIST.
4. Проведена оценка качества кластеризации.

Реализованный алгоритм кластеризации показал на датасете MNIST метрику ARI 0.848.