دانشگاه
خواجه نصیرالدین طوسی
K. N. Toosi University
of Technology

دانشکده مهندسی برق

# پروژه پایان‌ترم

## دانشجوها:

**نیلوفر ملا 40122903**

**محدثه علیرضایی طهرانی 40121123**

## استاد درس: جناب آقای دکتر علیاری

## لینک مخزن گیتهاب:

https://github.com/m24ath/Smart-System

## لینک گوگل کولب:

**سوال اول**

https://colab.research.google.com/drive/1F_obeBIzWlV4xwq3r5SRYFp8tdgA4rJA?usp=sharing

**سوال دوم**

https://colab.research.google.com/drive/1pgebpLHt1k29h-einOA97qcD12gsRYFk?usp=sharing


بسم الله الرحمن الرحيم

**عنوان** **صفحه**

**فهرست مطالب**

# فصل 1- مقدمه

در این بخش از پروژه درس مبانی سیستم های هوشمند، سه محور اصلی از جریان کاری یادگیری ماشین و یادگیری تقویتی بررسی می شود. ابتدا روی یک مسئله طبقه بندی واقعی تمرکز می کنیم تا با انتخاب ویژگی، هم دقت مدل حفظ شود و هم پیچیدگی آن کاهش یابد. سپس وارد خوشه بندی بدون ناظر می شویم تا با مقایسه چند خانواده الگوریتمی، ساختار پنهان داده های مشتریان استخراج و تفسیر شود. در نهایت، مفاهیم پایه یادگیری تقویتی با تاکید بر الگوریتم کیو لرنینگ مرور می گردد تا نقش پارامترهای یادگیری، نحوه به روزرسانی مقدار Q و عوامل موثر بر همگرایی به صورت شفاف تحلیل شود.

# فصل 2- بخش اول

## پرسش یک

**لینک مجموعه‌داده:** Loan Dataset (در فایل فشرده نیز موجود است).

**توضیح داده:** مجموعه‌دادهٔ مورد استفاده مربوط به وضعیت وام‌های اعطایی یک مؤسسهٔ مالی است. هر ردیف نمایانگر متقاضی یک وام بوده و شامل اطلاعاتی همچون جنسیت، وضعیت تأهل، تعداد افراد تحت تکفل، وضعیت شغلی، میزان درآمد، سابقهٔ تحصیلی، نوع منطقهٔ محل سکونت، مبلغ وام و در نهایت برچسب خروجی «وضعیت وام» (تأیید یا رد) می‌باشد. هدف، شناسایی ویژگی‌هایی است که بیشترین نقش را در تصمیم نهایی (تأیید یا عدم تأیید وام) دارند.

با توجه به تعداد بالای ویژگی‌ها و احتمال وجود ویژگی‌های غیرمؤثر یا هم‌بسته، انتخاب زیرمجموعهٔ بهینه از آن‌ها می‌تواند منجر به بهبود عملکرد مدل یادگیری گردد. در این تمرین، شما باید از **الگوریتم‌های تکاملی** برای انجام فرآیند Feature Selection استفاده کنید و نتایج دو روش را مقایسه نمایید.

آ) داده را بارگذاری کرده و تمامی مقادیر NaN را حذف کنید. سپس ستون‌های متغیرهای اسمی مانند Gender، Married، Education، Self_Employed، Area و Status را با استفاده از LabelEncoder به مقادیر عددی تبدیل نمایید.

```
#1.1
import pandas as pd

!gdown --fuzzy "https://drive.google.com/file/d/1eyJ3ZHx-pH-bBN0UJV6L2dj2-V6A-q2-
/view?usp=sharing"

!unzip -o Loan_Dataset.zip -d loan_data

train_data = pd.read_csv('loan_data/loan_train.csv')
test_data = pd.read_csv('loan_data/loan_test.csv')
train_data['Dependents'] = train_data['Dependents'].replace(['3+'], '3')
test_data['Dependents'] = test_data['Dependents'].replace(['3+'], '3')
train_data['Dependents'] = pd.to_numeric(train_data['Dependents'],
errors='coerce')
test_data['Dependents'] = pd.to_numeric(test_data['Dependents'], errors='coerce')

print("Before processing - Train Data Info:")
train_data_info = train_data.info()

print("\nBefore processing - Test Data Info:")
test_data_info = test_data.info()

train_data = train_data.dropna()
test_data = test_data.dropna()
```

```
print("\nAfter dropna - Train Data Info:")
train_data_info_after = train_data.info()

print("\nAfter dropna - Test Data Info:")
test_data_info_after = test_data.info()

from sklearn.preprocessing import LabelEncoder

categorical_columns = ['Gender', 'Married', 'Education', 'Self_Employed', 'Area']

label_encoders ={}

for column in categorical_columns:
    le = LabelEncoder()
    train_data[column] = le.fit_transform(train_data[column])
    test_data[column] = le.transform(test_data[column])
    label_encoders[column] = le
le_status = LabelEncoder()
train_data['Status'] = le_status.fit_transform(train_data['Status'])
label_encoders['Status'] = le_status


print("\nAfter encoding categorical columns - Train Data Info:")
train_data_info_final = train_data.info()

print("\nAfter encoding categorical columns - Test Data Info:")
test_data_info_final = test_data.info()

train_data_info, test_data_info, train_data_info_after, test_data_info_after,
train_data_info_final, test_data_info_final
```

پس از بارگذاری داده ها با استفاده از دستور gdown یک بار داده ها را نمایش می‌دهیم. به دلایلی pandas ویژگی dependents را اسمی درنظر می‌گرفت که در همان ابتدا با تبدیل 3+ به 3 و تبدیل آن به متغیر عددی از ایرادهای پیش رو جلوگیری کردیم.

طبق خواسته سوال ردیف‌هایی که دارای مقادیر گمشده (NaN) بودن را از دیتاست‌ها با استفاده از دستور dropna() حذف کردیم.بعد از این مرحله، تعداد ردیف‌ها کاهش پیدا کرد زیرا هر ردیف که حاوی مقادیر گمشده بود، حذف شد.

حالا که داده‌ها آماده شدند، باید مقادیر اسمی (مثل Gender, Married, Education, و.. ) را به مقادیر عددی (طبق متن زیر) تبدیل می‌کردیم. برای اینکار از LabelEncoder استفاده کردیم.

Gender: Female = 1 , Male = 0
Married: No = 1 , Yes = 0
Education: Not Graduate = 1 , Graduate = 0
Self_Employed: No = 1 , Yes = 0
Area: No = 1 , Yes = 0

ستون Status که نشان‌دهنده وضعیت تایید یا رد وام است، نیز به عدد تبدیل شد. در این ستون مقادیر Y (برای تایید) و N (برای رد) بود. این مقادیر رو با استفاده از LabelEncoder به عدد تبدیل کردیم.

**خروجی**

```
Before processing - Train Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 12 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Gender              601 non-null    object
 1   Married             611 non-null    object
 2   Dependents          599 non-null    float64
 3   Education           614 non-null    object
 4   Self_Employed       582 non-null    object
 5   Applicant_Income    614 non-null    int64
 6   Coapplicant_Income  614 non-null    float64
 7   Loan_Amount         614 non-null    int64
 8   Term                600 non-null    float64
 9   Credit_History      564 non-null    float64
 10  Area                614 non-null    object
 11  Status              614 non-null    object
dtypes: float64(4), int64(2), object(6)
Before processing - Test Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 367 entries, 0 to 366
Data columns (total 11 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Gender              356 non-null    object
 1   Married             367 non-null    object
 2   Dependents          357 non-null    float64
 3   Education           367 non-null    object
 4   Self_Employed       344 non-null    object
 5   Applicant_Income    367 non-null    int64
 6   Coapplicant_Income  367 non-null    int64
 7   Loan_Amount         367 non-null    int64
```

```
 8   Term               361 non-null    float64
 9   Credit_History     338 non-null    float64
 10  Area               367 non-null    object
dtypes: float64(3), int64(3), object(5)

After dropna - Train Data Info:
<class 'pandas.core.frame.DataFrame'>
Index: 499 entries, 0 to 613
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             499 non-null    object
 1   Married            499 non-null    object
 2   Dependents         499 non-null    float64
 3   Education          499 non-null    object
 4   Self_Employed      499 non-null    object
 5   Applicant_Income   499 non-null    int64
 6   Coapplicant_Income 499 non-null    float64
 7   Loan_Amount        499 non-null    int64
 8   Term               499 non-null    float64
 9   Credit_History     499 non-null    float64
 10  Area               499 non-null    object
 11  Status             499 non-null    object
dtypes: float64(4), int64(2), object(6)

After dropna - Test Data Info:
<class 'pandas.core.frame.DataFrame'>
Index: 293 entries, 0 to 366
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             293 non-null    object
 1   Married            293 non-null    object
 2   Dependents         293 non-null    float64
 3   Education          293 non-null    object
 4   Self_Employed      293 non-null    object
 5   Applicant_Income   293 non-null    int64
 6   Coapplicant_Income 293 non-null    int64
 7   Loan_Amount        293 non-null    int64
 8   Term               293 non-null    float64
 9   Credit_History     293 non-null    float64
 10  Area               293 non-null    object
dtypes: float64(3), int64(3), object(5)

After encoding categorical columns - Train Data Info:
<class 'pandas.core.frame.DataFrame'>
Index: 499 entries, 0 to 613
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             499 non-null    int64
 1   Married            499 non-null    int64
 2   Dependents         499 non-null    float64
 3   Education          499 non-null    int64
```

```
 4   Self_Employed       499 non-null    int64
 5   Applicant_Income    499 non-null    int64
 6   Coapplicant_Income  499 non-null    float64
 7   Loan_Amount         499 non-null    int64
 8   Term                499 non-null    float64
 9   Credit_History      499 non-null    float64
 10  Area                499 non-null    int64
 11  Status              499 non-null    int64
dtypes: float64(4), int64(8)

After encoding categorical columns - Test Data Info:
<class 'pandas.core.frame.DataFrame'>
Index: 293 entries, 0 to 366
Data columns (total 11 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Gender              293 non-null    int64
 1   Married             293 non-null    int64
 2   Dependents          293 non-null    float64
 3   Education           293 non-null    int64
 4   Self_Employed       293 non-null    int64
 5   Applicant_Income    293 non-null    int64
 6   Coapplicant_Income  293 non-null    int64
 7   Loan_Amount         293 non-null    int64
 8   Term                293 non-null    float64
 9   Credit_History      293 non-null    float64
 10  Area                293 non-null    int64
dtypes: float64(3), int64(8)
```

**پیش پردازش**

در مرحله‌ی پیش‌پردازش، ستون‌های Gender، Married، Education، Self_Employed، Area و Status به عنوان نوع داده object شناسایی شدند چرا که این ستون‌ها حاوی مقادیر متنی بودند.

**بعد از حذف NaN**

پس از حذف مقادیر NaN، تعداد ردیف‌ها در دیتاست train_data از ۶۱۴ به ۴۹۹ و در دیتاست test_data از ۳۶۷ به ۲۹۳ کاهش یافت. همانطور که انتظار داشتیم هیچ تغییری در نوع داده‌ها ایجاد نشد و ساختار ستون‌ها به همان صورت اولیه باقی ماند.

**بعد از تبدیل مفادیر اسمی به عددی**

ستون Gender که پیش از این شامل مقادیر رشته‌ای بود، به اعداد ۰ و ۱ تبدیل شد. همچنین ستون Status که با مقادیر Y و N مشخص شده بود، به ترتیب به ۱ و ۰ تبدیل گشت.

پس از تمام مراحل، اکنون ستون‌های Gender, Married, Education, Self_Employed, Area, Status به درستی به مقادیر عددی تبدیل شده‌اند و آماده استفاده در مدل‌های یادگیری ماشین هستند.

ب) الگوریتم Particle Swarm Optimization (PSO) را برای انتخاب ویژگی‌ها در فضای باینری پیاده‌سازی کنید (با استفاده از کتابخانهٔ pyswarms). هدف تابع برازندگی باید بیشینه کردن دقت مدل Random Forest و همزمان حداقل‌سازی تعداد ویژگی‌های انتخاب‌شده باشد. می‌توانید از ترکیب زیر برای تابع برازندگی استفاده کنید:

$$J = \alpha(1 - Acc) + (1 - \alpha)\left(1 - \frac{\text{تعداد ویژگی‌های انتخاب‌شده}}{\text{تعداد کل ویژگی‌ها}}\right)$$

که در آن $\alpha$ مقدار وزن بین دو عامل است.

```
#1.2
!pip install pyswarms
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import pyswarms as ps

X = train_data.drop('Status', axis=1)
y = train_data['Status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

def evaluate_features_pso(particles):
    scores = []

    for particle in particles:
        use_features = particle > 0.5

        if sum(use_features) == 0:
            scores.append(1.0)
            continue

        X_train_sel = X_train.iloc[:, use_features]
        X_test_sel = X_test.iloc[:, use_features]

        model = RandomForestClassifier(n_estimators=50, random_state=42,
max_depth=15)
```

```
        model.fit(X_train_sel, y_train)

        predictions = model.predict(X_test_sel)
        accuracy = accuracy_score(y_test, predictions)

        num_features = sum(use_features)
        fitness = 0.5 * (1 - accuracy) + 0.5 * (num_features / len(particle))

        scores.append(fitness)

    return np.array(scores)

options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

optimizer = ps.single.GlobalBestPSO(
    n_particles=50,
    dimensions=X.shape[1],
    options=options,
    bounds=(np.zeros(X.shape[1]), np.ones(X.shape[1]))
)

best_fitness_pso, best_solution_pso = optimizer.optimize(evaluate_features_pso,
iters=100)


selected_pso = best_solution_pso > 0.5
selected_names_pso = X.columns[selected_pso].tolist()

X_train_final_pso = X_train.iloc[:, selected_pso]
X_test_final_pso = X_test.iloc[:, selected_pso]

final_model_pso = RandomForestClassifier(n_estimators=100, random_state=42)
final_model_pso.fit(X_train_final_pso, y_train)
final_pred_pso = final_model_pso.predict(X_test_final_pso)
final_acc_pso = accuracy_score(y_test, final_pred_pso)

print(f"\nFinal Accuracy: {final_acc_pso*100:.2f}%")
print(f"Best Fitness: {best_fitness_pso:.4f}")
```

در ابتدا داده‌ها را به دو بخش ویژگی‌ها (x) و برچسب‌ها (y) تقسیم می‌کنیم. x شامل تمام ستون‌ها به جز ستون Status است که در نهایت پیش‌بینی می‌شود.

سپس داده‌ها را به دو مجموعه X_train, y_train برای آموزش و X_test, y_test برای تست تقسیم می‌کنیم. نسبت تقسیم داده‌ها 70٪ به 30٪ است.

**تابع ارزیابی ویژگی‌ها(evaluate_features_pso)**

PSO به صورت یک آرایه از ذرات عمل می‌کند که در هر ذره (هر فرد)، ویژگی‌های انتخابی (0 یا 1) قرار دارد. در اینجا، برای هر ذره، ویژگی‌هایی که مقدار آن‌ها بیشتر از 0.5 است را انتخاب می‌کنیم. سپس مدل Random Forest با ویژگی‌های انتخاب‌شده آموزش داده می‌شود و دقت آن بر روی مجموعه داده تست محاسبه می‌شود.

**خروجی**

```
Selected 1 features out of 11:
  1. Credit_History

Final Accuracy: 82.00%
Best Fitness: 0.1355
```

در فرآیند انتخاب ویژگی با استفاده از PSO، تنها ویژگی Credit_History به عنوان موثرترین متغیر برای پیش‌بینی Status (وضعیت تایید وام) انتخاب شد. این انتخاب اگرچه غیرمعمول به نظر می‌رسد، اما نشان می‌دهد که الگوریتم PSO این ویژگی را دارای بیشترین تاثیر در پیش‌بینی تشخیص داده است. به طور معمول در مسائل پیچیده‌تر، چندین ویژگی در کنار یکدیگر برای پیش‌بینی به کار گرفته می‌شوند با این حال، در سوال ما، به دلیل محدود بودن تعداد ویژگی‌های تاثیرگذار، PSO تنها به همین یک ویژگی اکتفا کرده است.

دقت 82٪ که با استفاده از تنها یک ویژگی (Credit_History) به دست آمده، نشان‌دهنده قدرت بالای این ویژگی در پیش‌بینی وضعیت تایید وام است. هرچند در بسیاری از مسائل، مدل‌های پیچیده‌تر با چندین ویژگی به کار گرفته می‌شوند، اما این دقت نسبتا خوب بیانگر آن است که حتی با یک ویژگی کلیدی نیز می‌توان به عملکرد قابل قبولی دست یافت. از سوی دیگر، مقدار fitness برابر با 0.1355 نشان‌دهنده ترکیب بهینه دقت و تعداد ویژگی‌هاست. هرچه این مقدار کمتر باشد، مدل هم از دقت بالاتری برخوردار است و هم از ویژگی‌های کمتری استفاده کرده است. در اینجا، مقدار پایین fitness تأییدی بر کارایی مدل با حداقل تعداد ویژگی است.

این قسمت از کد را انواع تنظیمات مختلف امتحان کردیم. مرز مقداری که ویژگی ها انتخاب شوند را هم نیز از 0.5 به اعداد دیگر تغییر دادیم. ولی باز هم به همین نتیجه رسیدیم ممکن است با تغییر مقدار آلفا چون وزن

تعادل را عوض می‌کند، به خروجی متفاوتی برسیم اما در سوال گفته شده که ما بیشینه دقت با کمترین ویژگی انتخاب شده مدنظرمان است پس به نظر می‌رسد همین تنظیمات و خروجی مناسب است.

یکی از تغییرات تنظیماتی که انجام شده بود برای مثال:

```
model = RandomForestClassifier(n_estimators=50, random_state=42, max_depth=15)
```

ج) الگوریتم Genetic Algorithm (GA) را با استفاده از کتابخانهٔ DEAP برای همین مسئله پیاده‌سازی کنید. در آن، هر فرد یک رشتهٔ باینری است که مقدار ۱ نشان‌دهندهٔ انتخاب آن ویژگی و مقدار ۰ نشان‌دهندهٔ عدم انتخاب است. از عملگرهای Two-Point Crossover و Bit Flip Mutation برای ایجاد نسل‌های جدید استفاده کنید.

در کدی که نوشته‌ایم، از الگوریتم ژنتیک برای انتخاب ویژگی‌ها استفاده کرده‌ایم. ابتدا داده‌ها را به ویژگی‌ها و برچسب‌ها تقسیم کرده و سپس از DEAP (کتابخانه الگوریتم‌های ژنتیک) برای اجرای الگوریتم ژنتیک استفاده کردیم. در این الگوریتم، برای هر فرد (کروموزوم)، یک بردار دودویی تولید می‌شود که نشان‌دهنده انتخاب یا عدم انتخاب ویژگی‌ها است. سپس مدل Random Forest با ویژگی‌های انتخاب‌شده آموزش داده می‌شود و دقت آن محاسبه می‌شود. در نهایت، بهترین فرد از الگوریتم ژنتیک (یعنی بهترین مجموعه ویژگی‌ها) را انتخاب کرده و دقت نهایی مدل محاسبه می‌شود.

```
#1.3
!pip install deap
import random
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from deap import base, creator, tools, algorithms

X = train_data.drop('Status', axis=1)
y = train_data['Status']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

def evaluate(individual):
    selected_features = [index for index, value in enumerate(individual) if value
== 1]
```

```
    if len(selected_features) == 0:
        return (1.0,)

    X_train_selected = X_train.iloc[:, selected_features]
    X_test_selected = X_test.iloc[:, selected_features]

    model = RandomForestClassifier(n_estimators=50, random_state=42,
max_depth=15)
    model.fit(X_train_selected, y_train)
    y_pred = model.predict(X_test_selected)

    accuracy = accuracy_score(y_test, y_pred)

    num_features = len(selected_features)
    fitness = 0.5 * (1 - accuracy) + 0.5 * (num_features / len(individual))

    return (fitness,)

if hasattr(creator, "FitnessMin"):
    del creator.FitnessMin
if hasattr(creator, "Individual"):
    del creator.Individual

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_bool, n=X_train.shape[1])
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluate)

population = toolbox.population(n=50)

hof = tools.HallOfFame(1)

pop, logbook = algorithms.eaSimple(
    population,
    toolbox,
```

```
    cxpb=0.9,
    mutpb=0.1,
    ngen=50,
    halloffame=hof,
    verbose=True
)

best_individual = hof[0]
selected_features_ga = [index for index, value in enumerate(best_individual) if
value == 1]
selected_names_ga = X_train.columns[selected_features_ga].tolist()

X_train_final_ga = X_train.iloc[:, selected_features_ga]
X_test_final_ga = X_test.iloc[:, selected_features_ga]

final_model_ga = RandomForestClassifier(n_estimators=100, random_state=42)
final_model_ga.fit(X_train_final_ga, y_train)
final_pred_ga = final_model_ga.predict(X_test_final_ga)
final_acc_ga = accuracy_score(y_test, final_pred_ga)

print(f"\nFinal Accuracy: {final_acc_ga*100:.2f}%")
print(f"Best Fitness: {best_individual.fitness.values[0]:.4f}")
```

مقدار FitnessMin به این معناست که هدف الگوریتم، کاهش fitness و دستیابی به کمترین مقدار ممکن برای آن است. در واقع، هرچه مقدار fitness کمتر باشد، مدل عملکرد بهتری از نظر ترکیب دقت و تعداد ویژگی‌های انتخابی دارد. در این قسمت از سوال نیز رسیدن به مقدار پایین fitness نشان‌دهنده موفقیت الگوریتم PSO در انتخاب ویژگی‌ای تاثیرگذار با حداقل تعداد ممکن است.

Individual نشان‌دهنده فردی است که یک کروموزوم (بردار از ویژگی‌های انتخاب‌شده) دارد.

Toolbox برای اجرای الگوریتم ژنتیک (GA) پیکربندی می‌شود. برای تولید ویژگی‌های دودویی (۰ یا ۱) از تابع attr_bool استفاده کردیم. هر فرد (کروموزوم) با تابع individual و جمعیت اولیه نیز با تابع population ساخته می‌شود. عملیات‌های ترکیب و جهش به ترتیب توسط توابع mate و mutate انجام می‌شوند. در نهایت، برای انتخاب افراد برتر از روش Tournament Selection به کمک تابع select استفاده می‌شود.

**خروجی**

```
gen   nevals
0     50
1     48
2     40
3     46
4     46
5     43
6     42
7     46
8     46
9     44
10    47
11    46
12    50
13    50
14    42
15    47
16    50
...
50    48
Selected 1 features out of 11:
  1. Credit_History

Final Accuracy: 82.00%
Best Fitness: 0.1355
```

تنها یک ویژگی به نام `Credit_History` انتخاب شده است، که نشان می‌دهد این ویژگی برای پیش‌بینی وضعیت وام از سایر ویژگی‌ها مهم‌تر است. این به این معناست که الگوریتم ژنتیک تصمیم گرفته که فقط از این ویژگی برای مدل استفاده کند.

دقت نهایی مدل 82% است که نشان می‌دهد مدل با ویژگی انتخاب‌شده (`Credit_History`) توانسته به دقت خوبی برسد. بهترین مقدار fitness برابر با 0.1355 است که نشان‌دهنده این است که این ویژگی برای پیش‌بینی وضعیت وام بهینه‌ترین ویژگی بوده است.

د) برای هر الگوریتم، دقت مدل Random Forest را پس از انتخاب ویژگی‌ها محاسبه کرده و میانگین دقت و تعداد ویژگی‌های انتخاب‌شده را گزارش دهید. نتایج دو الگوریتم را از نظر عملکرد (دقت و سادگی مدل) مقایسه نمایید.

در این قسمت محاسبه دقت از ما خواسته شده است که ما در قسمت های قبل بعد اموزش مدل دقت هم محاسبه کردیم.

```
#1.4
from sklearn.model_selection import train_test_split, cross_val_score

cv_scores_pso = cross_val_score(final_model_pso, X_train_final_pso, y_train,
cv=5, scoring='accuracy')
mean_accuracy_pso = np.mean(cv_scores_pso)
std_accuracy_pso = np.std(cv_scores_pso)

print("\nPSO Results:")
print(f"Selected {len(selected_names_pso)} features out of {X.shape[1]}: {',
'.join(selected_names_pso)}")
print(f"Mean Accuracy: {mean_accuracy_pso * 100:.2f}%")
print(f"Accuracy Std Dev: {std_accuracy_pso * 100:.2f}%")
print(f"Final Accuracy: {final_acc_pso * 100:.2f}%")
print(f"Best Fitness: {best_fitness_pso:.4f}")

cv_scores_ga = cross_val_score(final_model_ga, X_train_final_ga, y_train, cv=5,
scoring='accuracy')
mean_accuracy_ga = np.mean(cv_scores_ga)
std_accuracy_ga = np.std(cv_scores_ga)

print("\nGA Results:")
print(f"Selected {len(selected_names_ga)} features out of {X.shape[1]}: {',
'.join(selected_names_ga)}")
print(f"Mean Accuracy: {mean_accuracy_ga * 100:.2f}%")
print(f"Accuracy Std Dev: {std_accuracy_ga * 100:.2f}%")
print(f"Final Accuracy: {final_acc_ga * 100:.2f}%")
print(f"Best Fitness: {best_individual.fitness.values[0]:.4f}")
```

خروجی

```
PSO Results:
Selected 1 features out of 11: Credit_History
Mean Accuracy: 79.67%
Accuracy Std Dev: 5.41%
Final Accuracy: 82.00%
Best Fitness: 0.1355

GA Results:
Selected 1 features out of 11: Credit_History
```

```
Mean Accuracy: 79.67%
Accuracy Std Dev: 5.41%
Final Accuracy: 82.00%
Best Fitness: 0.1355
```

**مقایسه PSO و GA**

**دقت**

PSO: 82%

GA: 82%

این بدان معناست که از نظر توان پیش‌بینی، عملا هیچ تفاوتی ندارند. هر دو به یک نقطه‌ی بهینه رسیده اند که با همان ویژگی‌ها همین دقت رو می‌دهد.

**سادگی مدل**

سادگی مدل رو اینجا با تعداد ویژگی‌های انتخاب‌شده می‌سنجیم:

PSO: 11 ویژگی از 1

GA: 11 ویژگی از 1

پس از نظر سادگی هم برابراند. هر دو مدل را به ساده‌ترین حالت (خیلی شدیدا) فشرده کردند.

**Best Fitness**

PSO: 0.1355

GA: 0.1355

چون تابع فیتنس  برای هر دو الگوریتم یکسان است و نتیجه‌ی نهایی هم دقیقا یکسان شده است، مقدار fitnessهم طبیعتا برابر درمی‌آید.

PSO و GA در این کد از نظر دقت و سادگی دقیقا هم‌سطح‌اند. هیچ‌کدام از دیگری بهتر نیست زیرا هر دو به یک جواب رسیده اند.

اینکه نتایج الگوریتم‌های PSO و GA در انتخاب ویژگی دقیقا یکسان شده، اتفاقی کاملا طبیعی و قابل انتظار است و دلایل مشخصی دارد.

نخستین و مهم‌ترین دلیل، وجود یک ویژگی غالب در دیتاست است. در مجموعه‌داده‌های مرتبط با وام، ویژگی Credit_History معمولا قوی‌ترین پیش‌بینی کننده محسوب می‌شود. وقتی این ویژگی به تنهایی بتواند

دقتی در حدود 82٪ ارائه دهد، الگوریتم‌های بهینه‌یابی به سرعت تشخیص می‌دهند که افزودن سایر ویژگی‌ها نه تنها کمک چشمگیری نمی‌کند، بلکه ممکن است مقدار تابع فیتنس را بدتر کند.

دومین عامل به ساختار تابع فیتنس برمی‌گردد. در این تابع، به تعداد ویژگی‌های انتخاب شده وزن داده شده است. فرمول فیتنس به این صورت است که ترکیبی از کاهش خطای طبقه‌بندی و نسبت تعداد ویژگی‌های انتخاب شده به کل ویژگی‌ها را شامل می‌شود. این طراحی باعث می‌شود الگوریتم‌ها به سمت راه‌حل‌هایی با کمترین تعداد ویژگی تمایل پیدا کنند، به شرطی که دقت قربانی نشود. بنابراین وقتی یک ویژگی به تنهایی عملکرد قابل قبولی دارد، انتخاب همان یک ویژگی بهترین تصمیم ممکن خواهد بود.

سومین دلیل به یکسان بودن شرایط آزمایش بازمی‌گردد. هر دو الگوریتم با همان تقسیم داده به آموزش و آزمون، همان مدل جنگل تصادفی با همان پارامترها و همان تابع فیتنس ارزیابی شده‌اند. وقتی فضای مسئله و شرایط ارزیابی کاملا یکسان باشد و یک پاسخ مشخص به عنوان پاسخ برتر وجود داشته باشد، انتظار می‌رود الگوریتم‌های مختلف به همان نقطه همگرا شوند.

در نهایت، اندازه محدود فضای جستجو نیز در این مسئله بی‌تأثیر نیست. با تنها ۱۱ ویژگی، تعداد کل زیرمجموعه‌های ممکن ۲۰۴۸ حالت است که برای یک مسئله انتخاب ویژگی عدد نسبتا کوچکی محسوب می‌شود. در چنین فضای محدودی، شانس رسیدن الگوریتم‌های مختلف به جواب بهینه یکسان بسیار بالا خواهد بود، به ویژه زمانی که آن جواب به وضوح برتر از سایر گزینه‌ها باشد.

ه) ویژگی‌هایی را که هر الگوریتم «مهم‌تر» تشخیص داده است چاپ کنید و در چند خط تحلیل کنید که چرا انتخاب‌ها ممکن است متفاوت باشند.

```
#1.5
print(f"Selected {len(selected_names_pso)} features out of {X.shape[1]}: {',
'.join(selected_names_pso)}")
print(f"Selected {len(selected_names_ga)} features out of {X.shape[1]}: {',
'.join(selected_names_ga)}")
```

این قسمت قبلا انجام شده بود فقط در اینجا مجدد چاپ شد

**خروجی**

```
Selected 1 features out of 11: Credit_History
Selected 1 features out of 11: Credit_History
```

الگوریتم‌های PSO و GA دو الگوریتم objective و تصادفی هستند که از روش‌های متفاوتی برای جستجوی فضای ویژگی‌ها استفاده می‌کنند. به همین دلیل، ممکن است انتخاب ویژگی‌ها در هر یک از این الگوریتم‌ها متفاوت باشد. اما در کد ما، چون هر دو الگوریتم در نهایت فقط ویژگی Credit_Historyرا انتخاب کرده‌اند، دلیل این انتخاب مشابه به دلایل زیر است:

### 1. ویژگی Credit_History به طور خاص بسیار تاثیرگذار است

در بسیاری از مسائل مرتبط با وام‌دهی و اعتبارسنجی، تاریخچه اعتباری (Credit History) فرد از مهم‌ترین شاخص‌ها برای تصمیم‌گیری است. این ویژگی به راحتی می‌تواند تاثیر زیادی بر تصمیمات مدل بگذارد، چرا که می‌تواند به وضوح نمایانگر خطرات یا امنیت یک فرد در بازپرداخت وام باشد.
اگر مدل‌های Random Forestیا سایر مدل‌های یادگیری ماشین بر اساس این ویژگی به دقت بسیار بالایی برسند، الگوریتم‌های انتخاب ویژگی به سمت انتخاب این ویژگی هدایت می‌شوند تا دقت مدل افزایش یابد و تعداد ویژگی‌ها کم نگه داشته شود.

### 2. توابع fitness و هزینه انتخاب ویژگی‌ها

در هر دو الگوریتم، تابع fitness به گونه‌ای طراحی شده که نه تنها به دقت مدل توجه می‌کند، بلکه تعداد ویژگی‌های انتخاب‌شده نیز برای کاهش پیچیدگی مدل و جلوگیری از overfitting تشویق می‌شود.
به همین دلیل، وقتی که مدل با استفاده از تنها Credit_History به دقت بالا می‌رسد، هر دو الگوریتم ترجیح می‌دهند تنها از این ویژگی استفاده کنند و ویژگی‌های اضافی را کنار بگذارند.

### 3. انتخاب ویژگی مشابه در هر دو الگوریتم PSO و GA

علی‌رغم این که PSO و GA روش‌های متفاوتی دارند PSO با استفاده از ذرات در فضای جستجو حرکت می‌کند و GA با استفاده از عملگرهای انتخاب، Crossover و جهش، چون فضای جستجو محدود و ویژگی‌ها کم است تنها 11 ویژگی، این احتمال وجود دارد که هر دو الگوریتم در نهایت به یک نتیجه مشابه برسند.
به علاوه، در هر دو الگوریتم fitness functionبه سمت ساده‌سازی مدل و کاهش تعداد ویژگی‌ها تمایل دارد. بنابراین، الگوریتم‌ها ممکن است همان ویژگی مهم و موثر (Credit_History) را انتخاب کنند و دیگر ویژگی‌ها را کنار بگذارند.

### 4. حساسیتم نسبت به ویژگی‌های اثرگذار

در اینجا، انتخاب Credit_History به این معناست که مدل به وضوح از این ویژگی برای پیش‌بینی وضعیت وام استفاده می‌کند و برای الگوریتم‌ها از اهمیت خاصی برخوردار است.

حتی در الگوریتم‌های پیچیده‌تری مانند PSOیا GA، اگر مدل به خوبی با این ویژگی کار کند، الگوریتم‌ها ممکن است فقط به همین یک ویژگی بسنده کنند.

### 5. پایداری در انتخاب ویژگی‌ها

در این داده‌ها، که ویژگی Credit_History به وضوح با دقت بالا پیش‌بینی می‌کند، انتخاب فقط این ویژگی احتمالاً در هر دو الگوریتم به دلیل پایداری در تصمیم‌گیری‌ها و ساده‌سازی مدل اتفاق افتاده است. به عبارت دیگر، در هر دو الگوریتم، الگوریتم‌ها ترجیح می‌دهند ویژگی‌های اضافی را انتخاب نکنند چون تاثیر آن‌ها بسیار کم و مدل پیچیده‌تر می‌شود.

اگر جواب‌ها (انتخاب ویژگی‌ها) در الگوریتم‌های PSOو GAمتفاوت بودند، این می‌توانست دلایل مختلفی داشته باشد که به خصوصیات هر الگوریتم و نحوه جستجوی فضای ویژگی‌ها مربوط است. در اینجا چند دلیل احتمالی که می‌تواند موجب تفاوت در انتخاب ویژگی‌ها شود را آورده ایم:

### 1. روش‌های جستجوی مختلف در PSO وGA

PSO (Particle Swarm Optimization) این الگوریتم از الگوریتمی الهام گرفته شده از رفتار جمعی پرندگان یا ماهی‌ها استفاده می‌کند. درPSO ، ذرات (ذرات نماینده هر ویژگی) در فضای جستجو حرکت می‌کنند و از موقعیت‌های قبلی خود برای پیدا کردن بهترین موقعیت استفاده می‌کنند. این الگوریتم به دنبال بهترین حل (حداکثر دقت و حداقل پیچیدگی) در کل فضای جستجو می‌گردد.

GA (Genetic Algorithm) این الگوریتم از فرایندهای ژنتیکی مانند انتخاب طبیعی، Crossover و جهش (Mutation) برای جستجوی فضای جستجو استفاده می‌کند. این الگوریتم‌ها نسل به نسل به جستجوی ویژگی‌های بهینه می‌پردازند.

از آنجا که این دو الگوریتم از اصول مختلفی برای جستجو و ترکیب ویژگی‌ها استفاده می‌کنند، ممکن است انتخاب ویژگی‌ها در هر الگوریتم متفاوت باشد. مثلاً ممکن است یکی از الگوریتم‌ها به ویژگی خاصی با تاثیرگذار بودن بسیار زیاد بیشتر توجه کند، در حالی که دیگری ممکن است ویژگی‌های دیگری را که ترکیب بهتری ایجاد می‌کنند، انتخاب کند.

### 2. اثر انتخاب تصادفی در PSO وGA

PSOبه طور تصادفی ذرات خود را در فضا جابجا می‌کند و گاهی ممکن است در مسیرهای مختلف به حل بهینه برسد. اما این جابجایی می‌تواند به ویژگی‌های مختلفی منتهی شود.

GAبا استفاده از عملیات Crossover و جهش ممکن است به انتخاب ویژگی‌های مختلفی برسد که در نهایت به بهترین ترکیب ویژگی‌ها منتهی شود.
در بعضی موارد، تصادفی بودن در جابجایی ذرات PSO یا عملگرهای جهش در GA می‌تواند به انتخاب ویژگی‌های متفاوت منجر شود.

### 3. تنظیمات متفاوت در الگوریتم‌ها

هر یک از این الگوریتم‌ها به پارامترهای خاص خود نیاز دارند:

- در PSOپارامترهایی مانند c1، c2و wتاثیر زیادی بر جستجو در فضای ویژگی‌ها دارند.
- در GAپارامترهایی مانند تعداد نسل‌ها(ngen) ، احتمال Crossover و احتمال جهش می‌توانند انتخاب ویژگی‌ها را تحت تاثیر قرار دهند.

تنظیمات این پارامترها برای مثال تعداد ذرات در PSO یا تعداد نسل‌ها در GA می‌توانند به انتخاب ویژگی‌های متفاوتی منتهی شوند. تغییرات کوچک در این پارامترها می‌تواند الگوریتم را به سمت انتخاب ویژگی‌های مختلف سوق دهد.

### 4. تابع Fitness و هدف الگوریتم‌ها

در هر دو الگوریتم، تابع fitnessبا توجه به دو پارامتر دقت مدل و تعداد ویژگی‌های انتخاب‌شده طراحی شده است. اما وزن‌دهی به این دو پارامتر می‌تواند بر انتخاب ویژگی‌ها تاثیر بگذارد:

- در PSO و GA، اگر وزن به تعداد ویژگی‌های انتخاب‌شده بیشتر باشد (یعنی اگر هدف بر کاهش پیچیدگی مدل باشد)، الگوریتم ممکن است ویژگی‌های کمتری انتخاب کند.
- اگر وزن به دقت مدل بیشتر باشد، الگوریتم ممکن است ویژگی‌های بیشتری انتخاب کند تا دقت مدل بالا رود.

به عنوان مثال، اگر در یکی از الگوریتم‌ها وزن بیشتری به کاهش پیچیدگی مدل داده شده باشد، این الگوریتم ممکن است ویژگی‌های کمتری انتخاب کند، در حالی که دیگری ممکن است ویژگی‌های بیشتری انتخاب کند چون بیشتر بر دقت مدل تمرکز داشته باشد.

### 5. نوسان در پاسخ به تغییرات در فضای جستجو

چون الگوریتم‌های PSO و GA هر کدام در فضای جستجوی خود به صورت تصادفی عمل می‌کنند، ممکن است انتخاب‌های مختلفی را در مراحل مختلف اجرا پیدا کنند. به عبارت دیگر، در یک بار اجرا ممکن است PSO به ویژگی‌های خاصی برسد، در حالی که در اجرای دیگر به ویژگی‌های متفاوتی برسد.
نوسان در انتخاب ویژگی‌ها ممکن است به دلیل ویژگی‌های تصادفی یا پارامترهای الگوریتم در یک مرحله خاص باشد.

و) **(بخش مقایسه و تحلیل)** نمودار تغییرات مقدار برازندگی یا دقت را برای هر دو الگوریتم (PSO و GA) در طول تکرارها رسم کنید. محور افقی را «تعداد تکرارها / نسل‌ها» و محور عمودی را «مقدار برازندگی یا دقت» در نظر بگیرید. در یک نمودار واحد، منحنی هر الگوریتم را با رنگ متفاوت نمایش دهید و نتایج را از نظر سرعت همگرایی، پایداری و دقت نهایی مقایسه کنید. در پایان، تحلیلی کوتاه از مشاهدهٔ نمودارها ارائه دهید.

```
#1.6
import random
import matplotlib.pyplot as plt

def evaluate_features_pso(particles):
    scores = []
    for particle in particles:
        use_features = particle > 0.5

        if use_features.sum() == 0:
           scores.append(1.0)
           continue

        X_train_sel = X_train.iloc[:, use_features]
        X_test_sel = X_test.iloc[:, use_features]

        model = RandomForestClassifier(n_estimators=10, random_state=42,
max_depth=5)
        model.fit(X_train_sel, y_train)
        pred = model.predict(X_test_sel)
        acc = accuracy_score(y_test, pred)

        num_feat = use_features.sum()
        fitness = 0.5 * (1 - acc) + 0.5 * (num_feat / len(particle))
        scores.append(fitness)

    return np.array(scores)

options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

optimizer = ps.single.GlobalBestPSO(
    n_particles=50,
    dimensions=X.shape[1],
    options=options,
    bounds=(np.zeros(X.shape[1]), np.ones(X.shape[1]))
)

best_fitness_pso, best_solution_pso = optimizer.optimize(evaluate_features_pso,
iters=100, verbose=False)
```

```
pso_fitness_history = optimizer.cost_history
pso_iters = np.arange(1, len(pso_fitness_history) + 1)

selected_pso = best_solution_pso > 0.5
selected_names_pso = X.columns[selected_pso].tolist()

X_train_final_pso = X_train.iloc[:, selected_pso]
X_test_final_pso = X_test.iloc[:, selected_pso]

final_model_pso = RandomForestClassifier(n_estimators=100, random_state=42)
final_model_pso.fit(X_train_final_pso, y_train)
final_pred_pso = final_model_pso.predict(X_test_final_pso)
final_acc_pso = accuracy_score(y_test, final_pred_pso)


def evaluate_ga(individual):
    selected_features = [i for i, v in enumerate(individual) if v == 1]

    if len(selected_features) == 0:
       return (1.0,)

    X_train_sel = X_train.iloc[:, selected_features]
    X_test_sel = X_test.iloc[:, selected_features]

    model = RandomForestClassifier(n_estimators=10, random_state=42, max_depth=5)
    model.fit(X_train_sel, y_train)
    pred = model.predict(X_test_sel)
    acc = accuracy_score(y_test, pred)

    num_feat = len(selected_features)
    fitness = 0.5 * (1 - acc) + 0.5 * (num_feat / len(individual))
    return (fitness,)

if hasattr(creator, "FitnessMin"):
    del creator.FitnessMin
if hasattr(creator, "Individual"):
    del creator.Individual

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("attr_bool", random.randint, 0, 1)
```

```
toolbox.register("individual", tools.initRepeat, creator.Individual,
toolbox.attr_bool, n=X.shape[1])
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("evaluate", evaluate_ga)

population = toolbox.population(n=50)
hof = tools.HallOfFame(1)

stats = tools.Statistics(lambda ind: ind.fitness.values[0])
stats.register("min", np.min)
stats.register("avg", np.mean)

pop, logbook = algorithms.eaSimple(
    population,
    toolbox,
    cxpb=0.9,
    mutpb=0.1,
    ngen=50,
    halloffame=hof,
    stats=stats,
    verbose=False
)

ga_gens = logbook.select("gen")
ga_min_fitness = logbook.select("min")


best_individual = hof[0]
selected_features_ga = [i for i, v in enumerate(best_individual) if v == 1]
selected_names_ga = X.columns[selected_features_ga].tolist()

X_train_final_ga = X_train.iloc[:, selected_features_ga]
X_test_final_ga = X_test.iloc[:, selected_features_ga]

final_model_ga = RandomForestClassifier(n_estimators=100, random_state=42)
final_model_ga.fit(X_train_final_ga, y_train)
final_pred_ga = final_model_ga.predict(X_test_final_ga)
final_acc_ga = accuracy_score(y_test, final_pred_ga)

best_fitness_ga = best_individual.fitness.values[0]
```

```
print("PSO Results:")
print(f"Selected {len(selected_names_pso)} features out of {X.shape[1]}:
{selected_names_pso}")
print(f"Final Accuracy: {final_acc_pso*100:.2f}%")
print(f"Best Fitness: {best_fitness_pso:.4f}\n")

print("GA Results:")
print(f"Selected {len(selected_names_ga)} features out of {X.shape[1]}:
{selected_names_ga}")
print(f"Final Accuracy: {final_acc_ga*100:.2f}%")
print(f"Best Fitness: {best_fitness_ga:.4f}\n")

plt.figure(figsize=(10,6))
plt.plot(pso_iters, pso_fitness_history, marker='x', linestyle='--', label='PSO
(best fitness per iter)')
plt.plot(ga_gens, ga_min_fitness, marker='o', linestyle='-', label='GA (min
fitness per gen)')
plt.xlabel("Iterations / Generations")
plt.ylabel("Fitness Value")
plt.title("Comparison of PSO and GA Fitness Over Time")
plt.grid(True)
plt.legend()
plt.show()
```

در این قسمت همان مسئله‌ی انتخاب ویژگی را با دو روش PSO و GA اجرا کردیم، ولی این‌بار علاوه بر نتیجه‌ی نهایی، کاری کردیم که در طول اجرا هم روند Fitness ذخیره بشود تا بتوان نمودار تغییرات رو کشید. بعد از اینکه هر الگوریتم بهترین زیرمجموعه ویژگی‌ها را پیدا کرد، با همان ویژگی‌ها یک مدل نهایی RandomForestبا n_estimators=100 ساختیم و Final Accuracy را حساب کردیم.

```
pso_fitness_history = optimizer.cost_history
```

`cost_history` تاریخچه‌ی بهترین fitness پیدا شده تا آن iteration رو نگه می‌دارد. یعنی دقیقا همان چیزی که برای رسم نمودار تغییرات لازم داشتیم.

بخش جدیدی که باعث شد history داشته باشیم اینه که `stats` تعریف کردیم:

```
stats = tools.Statistics(lambda ind: ind.fitness.values[0])
stats.register("min", np.min)
stats.register("avg", np.mean)
```

و بعد این `stats` را به `eaSimple` دادیم:

```
pop, logbook = algorithms.eaSimple(..., stats=stats, ...)
```

این کار باعث شد `logbook` برای هر نسل، مقدارهای آماری مثل کمترین Fitness بهترین فرد نسل را ذخیره کند. بعد هم این را برای نمودار گرفتیم.

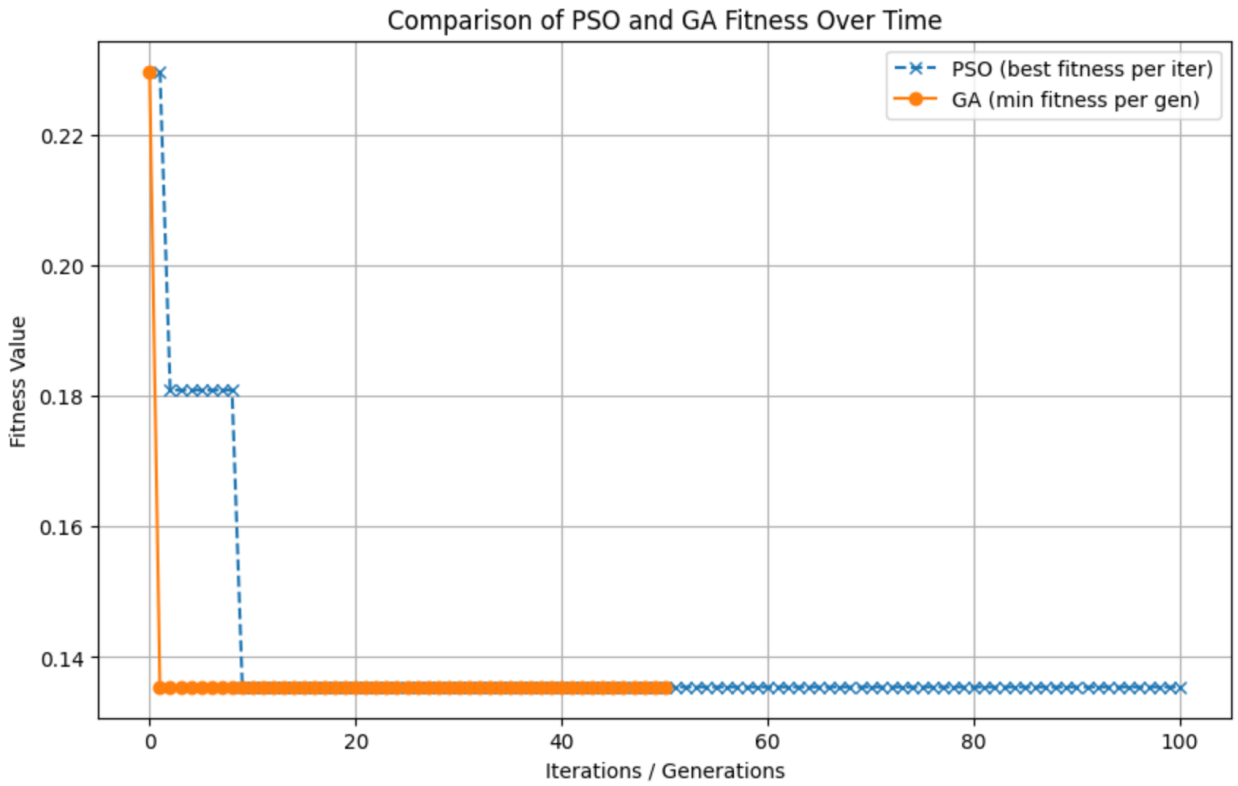
```
ga_min_fitness = logbook.select("min")
```

**خروجی**

```
PSO Results:
Selected 1 features out of 11: ['Credit_History']
Final Accuracy: 82.00%
Best Fitness: 0.1355

GA Results:
Selected 1 features out of 11: ['Credit_History']
Final Accuracy: 82.00%
Best Fitness: 0.1355
```

هر دو فقط ویژگی Credit_History انتخاب کردند و دقت نهایی هر دو Final Accuracy = 82% و Best Fitness هر دو 0.1355 بود که این کاملا با فرمول Fitness هم می‌خواند.




## تحلیل نمودار Fitness

در نمودار همگرایی، هر دو الگوریتم در نهایت به یک مقدار فیتنس حدود 0.1355 می‌رسند، اما مسیر رسیدن به این نقطه برای آنها کاملا متفاوت بوده است.

الگوریتم ژنتیک با منحنی نارنجی رنگ، از همان نسل‌های اولیه به سرعت به مقدار بهینه فیتنس دست پیدا می‌کند و پس از آن تا پایان روند، تقریباً ثابت می‌ماند. این نشان می‌دهد که GA خیلی زود جواب بهینه را شناسایی کرده و پس از آن نتوانسته راه‌حل بهتری بیابد. چنین رفتاری بیانگر همگرایی سریع و پایداری بالای این الگوریتم در مسئله مورد نظر است.

در مقابل، الگوریتم PSO با منحنی آبی رنگ، روند متفاوتی را نشان می‌دهد. این الگوریتم در نسل‌های ابتدایی مقادیر فیتنس بزرگ‌تری در حدود 0.23 و 0.18 دارد و به تدریج و طی چند مرحله پله‌ای کاهش می‌یابد تا نهایتاً به همان مقدار بهینه 0.1355 برسد. اگرچه PSO نیز در پایان به همان جواب GA دست پیدا می‌کند، اما سرعت همگرایی آن در این اجرا کندتر بوده و مسیر طولانی‌تری را طی کرده است.

# فصل 3- بخش سوم

## ۱ پرسش دو

**داده‌ها:** Mall Customer Segmentation (در فایل فشرده نیز موجود است).

آ. داده را بارگذاری، پاک‌سازی و ویژگی‌های عددی را انتخاب کنید. فقط روی ویژگی‌ها StandardScaler اعمال کنید. یک PCA دوبعدی صرفاً برای نمایش بسازید (مدل‌ها در فضای اصلی آموزش ببینند).

در گام اول، دیتاست بارگذاری می‌شود و سپس مراحل پاکسازی شامل حذف مقادیر گمشده و ردیف‌های تکراری و همچنین بررسی و اصلاح نوع ستون‌ها انجام می‌گیرد. در ادامه، تنها ویژگی‌های عددی را که از دیتاست استخراج شده و با استفاده از StandardScaler نرمال‌سازی می‌کنیم. در نهایت، برای تصویر کردن، یک PCA دوبعدی روی داده‌های مقیاس‌شده اعمال و خروجی آن به صورت نمودار نمایش می‌دهیم.

```
#2.1
import pandas as pd
!pip -q install gdown

!gdown --fuzzy
"https://drive.google.com/file/d/1mWHneibejp026KCQbHmgly16L9wAuq8Q/view?usp=drive
_link" -O Mall_Customers.csv

data = pd.read_csv("Mall_Customers.csv")

data_info = data.info()
data_info
data = data.copy()

data = data.drop_duplicates()
data = data.dropna()
print("Shape after cleaning:", data.shape)
data.head()

numeric_cols = data.select_dtypes(include=["number"]).columns.tolist()
print("Numeric columns:", numeric_cols)

if "CustomerID" in numeric_cols:
    numeric_cols.remove("CustomerID")

X = data[numeric_cols].copy()
print("Selected numeric features for scaling/model:", X.columns.tolist())
X.head()
```

```
from sklearn.preprocessing import StandardScaler
import pandas as pd

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns, index=data.index)

print("Scaled means (approx 0):")
print(X_scaled_df.mean().round(4))
print("\nScaled stds (approx 1):")
print(X_scaled_df.std(ddof=0).round(4))

from sklearn.decomposition import PCA

pca = PCA(n_components=2, random_state=23)
X_pca2 = pca.fit_transform(X_scaled_df)

pca_df = pd.DataFrame(X_pca2, columns=["PC1", "PC2"], index=data.index)

print("Explained variance ratio:", pca.explained_variance_ratio_)
pca_df.head()

import matplotlib.pyplot as plt

plt.figure(figsize=(7,5))

if "Gender" in data.columns:
    genders = data["Gender"].astype(str)
    colors = genders.map({"Male": 0, "Female": 1}).fillna(2)

    plt.scatter(pca_df["PC1"], pca_df["PC2"], c=colors, alpha=0.8)
    plt.title("PCA (2D) Visualization - colored by Gender (optional)")
else:
    plt.scatter(pca_df["PC1"], pca_df["PC2"], alpha=0.8)
    plt.title("PCA (2D) Visualization")

plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True, alpha=0.3)
plt.show()
```

پس از بارگذاری داده ها و نمایش محتوای آن ها اقدام به پاکسازی داده ها کردیم:

drop_duplicates() ردیف‌های تکراری را حذف می‌کند (اگر وجود داشته باشد).

dropna() ردیف‌هایی که مقدار گمشده دارند را حذف می‌کند (اگر وجود داشته باشد).

سپس ویژگی های عددی را انتخاب کردیم:
با select_dtypes(include=["number"]) فقط ستون‌هایی که عددی هستند انتخاب می‌شوند.

CustomerID عددی است، اما ویژگی یادگیری نیست پس حذفش کردیم پس از استاندارد سازی اقدام به نمایش دو بعدی PCA کردیم.

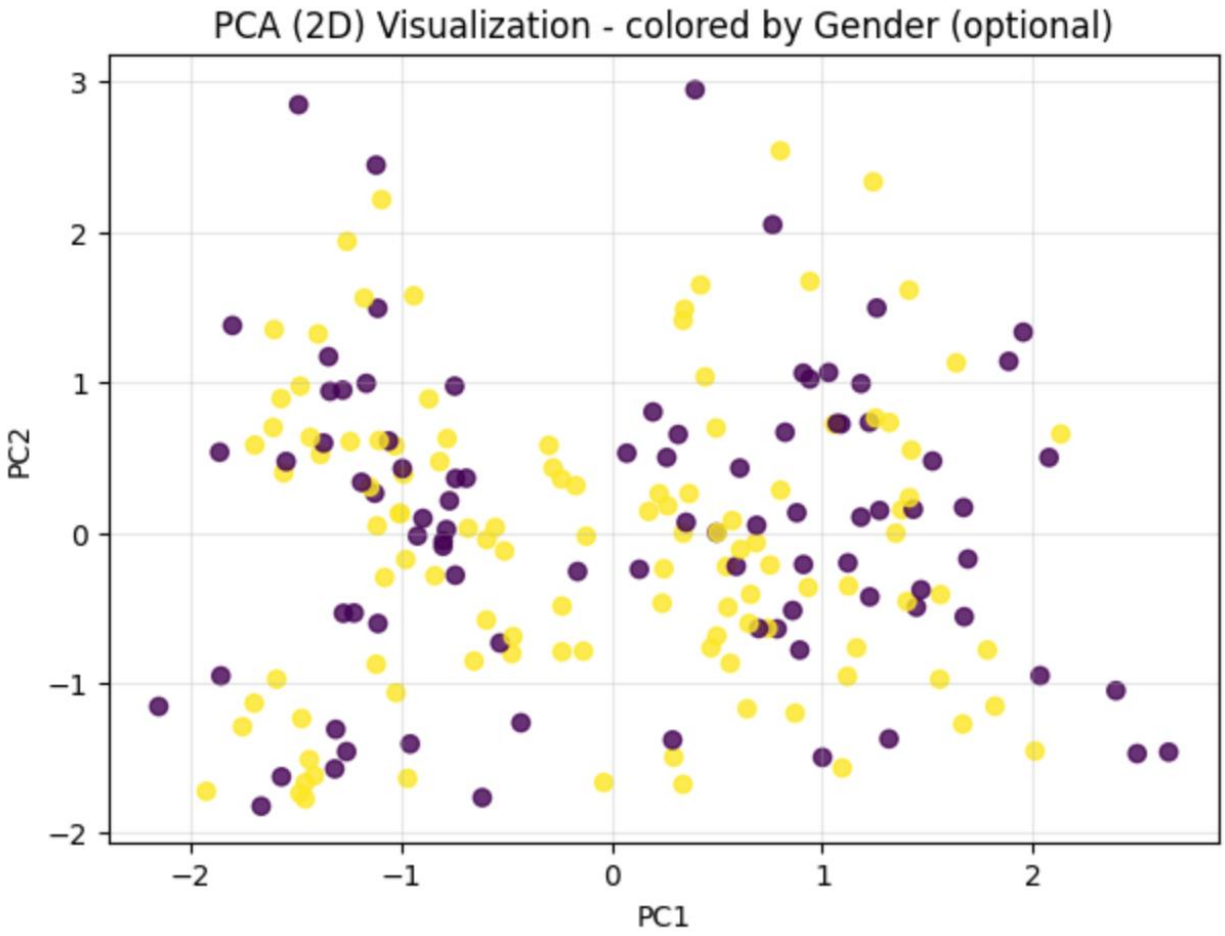
در صوال گفته شده بود random_state دو رقم اخر شماره دانشجویی یکی از اعضای گروه باشد 40121123

**خروجی**

```
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Gender                  200 non-null    object
 2   Age                     200 non-null    int64
 3   Annual Income (k$)      200 non-null    int64
 4   Spending Score (1-100)  200 non-null    int64
dtypes: int64(4), object(1)

Shape after cleaning: (200, 5)

Numeric columns: ['CustomerID', 'Age', 'Annual Income (k$)', 'Spending Score (1-
100)']
Selected numeric features for scaling/model: ['Age', 'Annual Income (k$)',
'Spending Score (1-100)']
Scaled means (approx 0):
Age                     -0.0
Annual Income (k$)      -0.0
Spending Score (1-100)  -0.0
...
Annual Income (k$)       1.0
Spending Score (1-100)   1.0
dtype: float64
Explained variance ratio: [0.44266167 0.33308378]
```




باتوجه به نتایج معلوم است که نه مقدار گمشده ای وجود داشته و نه ردیف تکراری ای و ویژگی های عددی هم به درستی انتخاب شده اند.

```
Scaled means (approx 0): -0.0
Scaled stds (approx 1): 1.0
```

این دقیقا همان چیزی است که باید رخ دهد. 0.0- فقط اثر نمایش floating point است، یعنی همان 0.

std(ddof=0) در نظر گرفته شده است، و چون StandardScaler هم با استاندارد جمعیت کار می‌کند، دقیقاً 1.0 درآمد.

```
Explained variance ratio: [0.44266167 0.33308378]
```

PC1 حدود 44.27% از واریانس کل داده را توضیح می‌دهد.

PC2 حدود 33.31% را توضیح می‌دهد.

0.44266167 + 0.33308378 = 0.77574545 (حدود 77.57%)

پس نمایش 2بعدی PCA حدود 77.6% از اطلاعات پراکندگی داده را حفظ کرده که برای Visualization عدد خوبی است .پراکندگی نقاط بیشتر در امتداد PC1 دو توده‌ی اصلی می‌سازد یعنی بیشترین تفاوت‌ها یا الگوهای داده احتمالا توسط ترکیبی از ویژگی‌ها در جهت PC1 توضیح داده می‌شود.

جنسیت جداسازی واضحی ایجاد نکرده چون هر دو رنگ (Male/Female) در هر دو سمت و نواحی نمودار حضور دارند. بر اساس این نمایش PCA ، جنسیت به تنهایی عامل تفکیک خوشه‌ها نیست و الگوها بیشتر از Age/Income/SpendingScore می‌آیند.

نقاط دورافتاده (outliers) محدودند و چند نقطه در بالا/پایین نمودار وجود دارد ولی شدید و غیرعادی نیست.

male = بنفش

female = زرد

ب. برای $K \in \{2, \ldots, 10\}$ KMeans را آموزش دهید. برای هر $K$: inertia و silhouette را گزارش کنید و $K$ نهایی را با استدلال انتخاب کنید.

برای انتخاب تعداد بهینه خوشه‌ها در الگوریتم K-Means، مراحل زیر را انجام دادیم:

ابتدا مدل K-Means برای تعداد خوشه‌های مختلف در بازه ۲ تا ۱۰ آموزش دادیم. برای هر مقدار K، دو معیار ارزیابی محاسبه می‌گردد: inertia و silhouette score.

معیار inertia که همان مجموع مربعات فاصله نقاط تا مرکز خوشه خودشان است، با افزایش تعداد خوشه‌ها همواره کاهش می‌یابد. هرچه این مقدار کمتر باشد، نقاط در خوشه‌های خود متراکم‌تر هستند.

در مقابل، silhouette score کیفیت جداسازی خوشه‌ها را در بازه [-1,1] نشان می‌دهد. هرچه این امتیاز به ۱ نزدیک‌تر باشد، خوشه‌بندی از کیفیت بهتری برخوردار است.

برای انتخاب K نهایی، روند تغییرات inertia و silhouette score به صورت همزمان بررسی می‌شود. بهترین K معمولاً نقطه‌ای است که در آن:
- کاهش inertia کندتر شده باشد (نقطه آرنج)
- silhouette score در بالاترین مقدار خود باشد

با بررسی این دو معیار در کنار یکدیگر، عددی را به عنوان K بهینه انتخاب می‌کنیم که هم تراکم درون‌خوشه‌ای قابل قبول و هم جدایی مطلوب بین خوشه‌ها را فراهم کند.

```python
#2.2
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import pandas as pd
import matplotlib.pyplot as plt

Ks = range(2, 11)

inertias = []
sil_scores = []

for k in Ks:
    kmeans = KMeans(n_clusters=k, random_state=23, n_init=10)
    labels = kmeans.fit_predict(X_scaled_df)

    inertias.append(kmeans.inertia_)
    sil_scores.append(silhouette_score(X_scaled_df, labels))
```

```
results = pd.DataFrame({
    "K": list(Ks),
    "Inertia": inertias,
    "Silhouette": sil_scores
})

print(results)

plt.figure(figsize=(7,5))
plt.plot(results["K"], results["Inertia"], marker="o")
plt.title("Elbow Method (Inertia vs K)")
plt.xlabel("K")
plt.ylabel("Inertia")
plt.grid(True, alpha=0.3)
plt.show()

plt.figure(figsize=(7,5))
plt.plot(results["K"], results["Silhouette"], marker="o")
plt.title("Silhouette Score vs K")
plt.xlabel("K")
plt.ylabel("Silhouette Score")
plt.grid(True, alpha=0.3)
plt.show()
```

Kmeans رو برای K های مختلف اجرا کردیم:

n_init=10 الگوریتم KMeans را 10 بار با شروع‌های مختلف اجرا می‌کند و بهترین جواب (کمترین inertia) را نگه می‌دارد این باعث پایداری بهتر نتایج می‌شود.

Inertia مجموع مربعات فاصله‌ی نقاط تا مرکز خوشه‌ی خودشان را بدست می‌آورد (WCSS/SSE) هرچه کمتر بهتر، ولی با افزایش K همیشه کاهش می‌یابد پس به‌تنهایی معیار انتخاب نیست.

Silhouette Score معیار کیفیت جداسازی خوشه‌ها در بازه‌ی [-1, 1] هرچه بزرگ‌تر بهتر (نزدیک 1 یعنی خوشه‌ها فشرده و از هم جدا) در نهایت رسم کردیم.

**خروجی**

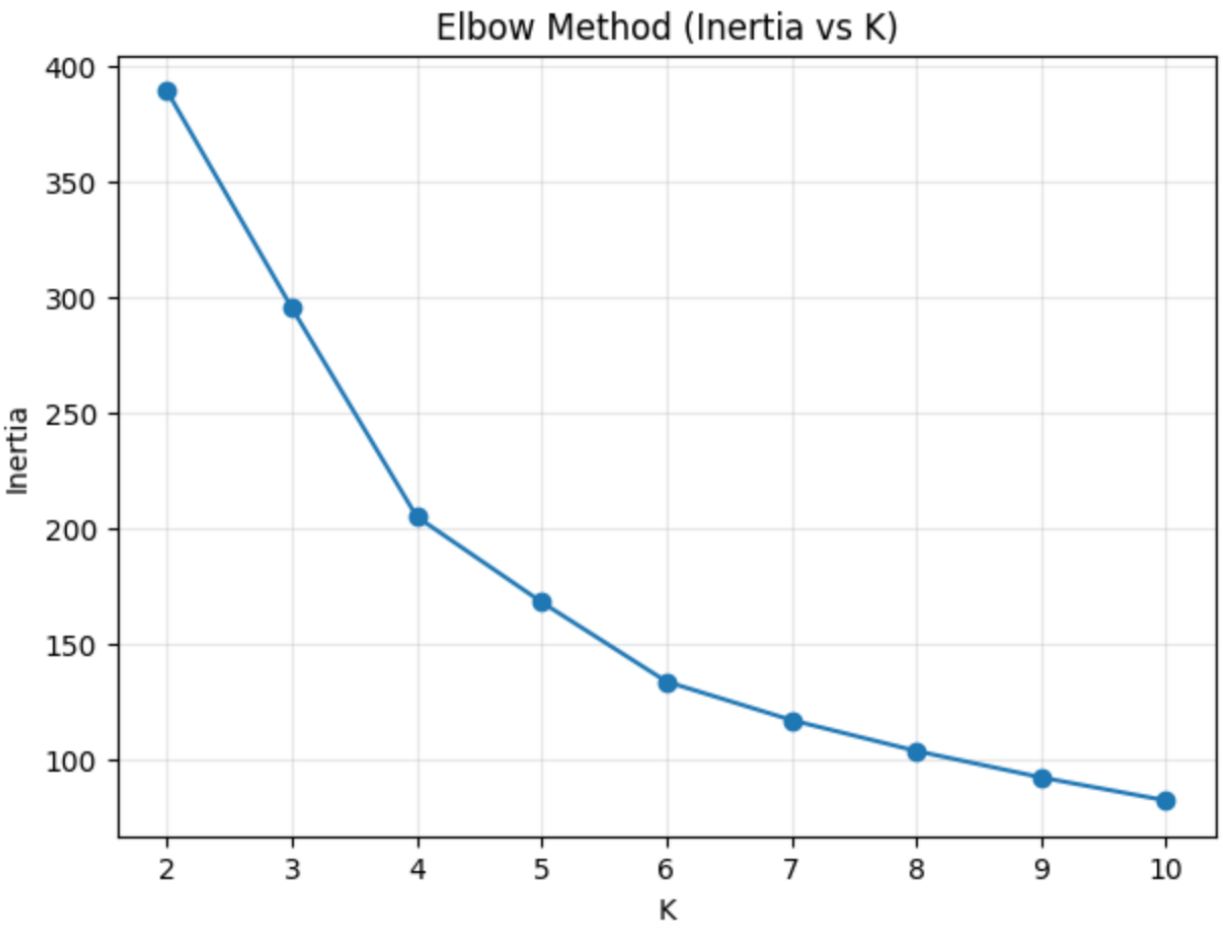
```
    K     Inertia  Silhouette
0   2  389.386189    0.335472
1   3  295.501955    0.358958
2   4  205.225147    0.403958
3   5  168.247580    0.416643
4   6  133.868334    0.427428
5   7  117.231929    0.419607
6   8  103.876008    0.407537
7   9   92.432574    0.418168
8  10   82.456952    0.401493
```

برای انتخاب K نهایی به صورت عملی و استاندارد، به این شکل عمل می‌کنیم.

از نمودار Elbow (آرنج) استفاده می‌کنیم. با افزایش K، مقدار inertia همیشه کاهش می‌یابد، اما در نقطه‌ای این کاهش شیب تند خود را از دست می‌دهد و کند می‌شود. همان نقطه که شبیه آرنج منحنی است، می‌تواند K مناسبی باشد. معمولا Kهای قبل از آن کاهش زیادی در inertia دارند و Kهای بعد از آن بهبود چشمگیری ایجاد نمی‌کنند.

از نمودار Silhouette هم کمک می‌گیریم. برخلاف inertia، silhouette همیشه با افزایش K بهتر نمی‌شود. معمولا در یک یا دو مقدار خاص، امتیاز silhouette به اوج خود می‌رسد. بالاترین مقدار silhouette (یا نزدیک به آن) گزینه‌ی مناسبی است، به شرطی که K بسیار کوچکی نباشد (مثلاً ۲ یا ۳).

در نهایت این دو معیار را کنار هم می‌گذاریم. ممکن است نقطه‌ی آرنج K=4 باشد، اما silhouette در K=5 امتیاز بالاتری بدهد، یا برعکس. در این حالت یک trade-off منطقی انجام می‌دهیم و Kای را انتخاب می‌کنیم که هم به آرنج نزدیک باشد و هم silhouette قابل قبولی داشته باشد. معمولاً Kهایی که هم در ناحیه‌ی آرنج قرار می‌گیرند و هم silhouette بالای ۰.۵ دارند، انتخاب‌های مطمئنی هستند.

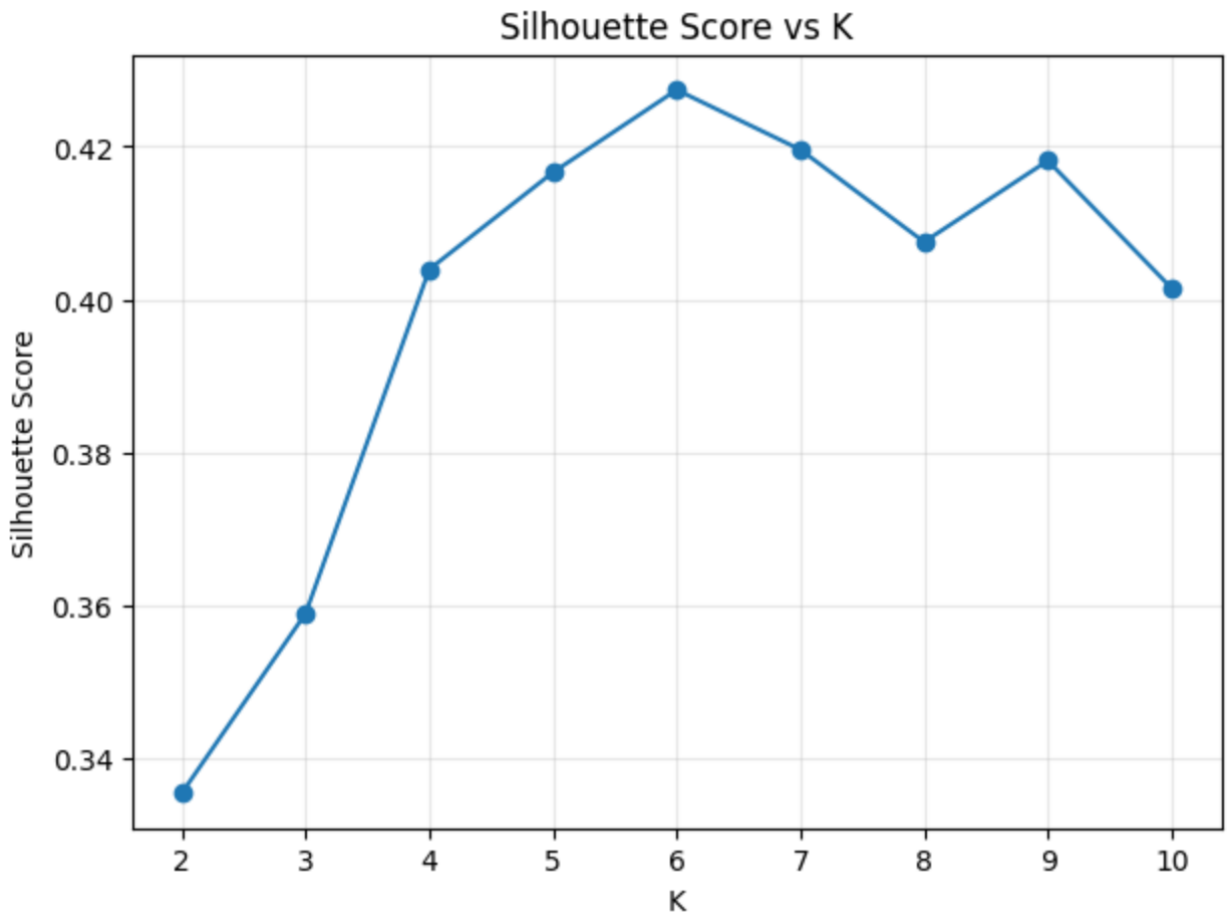



**تحلیل نمودار Elbow (Inertia)**

همانطور که گفته شد برای انتخاب تعداد بهینه خوشه‌ها، مدل K-Means را برای Kهای ۲ تا ۱۰ آموزش دادیم و دو معیار inertia و silhouette را بررسی کردیم.

نمودار inertia روندی نزولی داشت که کاملا طبیعی است. از K=2 تا K=4 کاهش inertia بسیار شدید بود که نشان از بهبود قابل توجه در تراکم خوشه‌ها دارد. از K=4 به بعد، شیب کاهش کمتر شد و در حوالی K=5 و K=6 منحنی به طور محسوسی صاف‌تر گردید که بیانگر بازده نزولی افزایش تعداد خوشه‌هاست.

بنابراین از منظر روش آرنج (Elbow)، ناحیه منطقی برای انتخاب K همان ۵ یا ۶ است. برای تصمیم نهایی، امتیاز silhouette را مقایسه کردیم که در K=5 مقدار بالاتری داشت. از این رو، K=5 را به عنوان تعداد بهینه خوشه‌ها برگزیدیم.




**تحلیل نمودار Silhouette**

silhouette از K=2 تا K=6 روندی افزایشی دارد و در K=6 به بیشترین مقدار خود یعنی ۰.۴۲۷۴ می‌رسد. این نقطه بهترین کیفیت جداسازی و فشردگی خوشه‌ها را نشان می‌دهد.

پس از K=6، امتیاز silhouette کاهش می‌یابد؛ به طوری که در K=7 به ۰.۴۱۹۶، در K=8 به ۰.۴۰۷۵ می‌رسد. در K=9 اندکی افزایش یافته و به ۰.۴۱۸۲ می‌شود اما همچنان از مقدار K=6 کمتر است و در نهایت در K=10 به ۰.۴۰۱۵ افت می‌کند.

بنابراین، با استناد به معیار silhouette، بهترین انتخاب برای تعداد خوشه‌ها K=6 است، زیرا بالاترین امتیاز را در این معیار کسب کرده و از کیفیت بهتری در تفکیک و تراکم خوشه‌ها برخوردار است.

با بررسی همزمان دو معیار:

- از دید Elbow، کاهش inertia از K=5 و K=6 به بعد کم‌اثر می‌شود.

- از دید Silhouette، بیشترین امتیاز دقیقاً در K=6 به دست می‌آید.

بنابراین هر دو معیار K=6 را تأیید می‌کنند و این انتخاب نهایی ماست.

ج. AgglomerativeClustering را با linkageهای single/complete/average/Ward و تعداد خوشهٔ انتخابی اجرا کنید؛ سیلوئت را مقایسه و بهترین linkage را گزارش کنید.

چهار نوع linkage را امتحان می‌کنیم:

single

complete

average

ward

یک تعداد خوشه‌ی انتخابی تعیین کنیم که همان K نهایی بخش قبل است K=6 . برای هرlinkage ، Silhouette Scoreرا حساب و مقایسه می‌کنیم. در نهایت بهترین linkage را بر اساس بیشترین silhouette گزارش کنیم.

```
#1.3
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score
import pandas as pd

K_final = 6

linkages = ["single", "complete", "average", "ward"]
rows = []

for link in linkages:
    try:
        model = AgglomerativeClustering(n_clusters=K_final, linkage=link,
metric="euclidean")
    except TypeError:
        model = AgglomerativeClustering(n_clusters=K_final, linkage=link,
affinity="euclidean")

    labels = model.fit_predict(X_scaled_df)
    sil = silhouette_score(X_scaled_df, labels)

    rows.append({"linkage": link, "K": K_final, "silhouette": sil})
```

```
results_hc = pd.DataFrame(rows).sort_values("silhouette",
ascending=False).reset_index(drop=True)
print(results_hc)

best = results_hc.iloc[0]
print(f"Best linkage: {best['linkage']}  |  silhouette:
{best['silhouette']:.4f}  |  K={int(best['K'])}")
```

single نزدیک‌ترین دو نقطه بین دو خوشه معیار ادغام است.⟵ ممکن است خوشه های کشیده بدهد.

Complete دورترین دو نقطه معیار است ⟵ خوشه‌های فشرده‌تر، حساس‌تر به نویز.

Average میانگین فاصله‌ها معیار است ⟵ رفتار متعادل.

Ward ادغام را طوری انجام می‌دهد که افزایش واریانس درون‌خوشه‌ای کمینه شود ⟵ معمولا برای داده‌های عددیِ اسکیل‌شده خیلی خوب جواب می‌دهد اما فقطEuclidean

برای هر linkage یک مدل می‌سازیم.

metric="euclidean" یعنی فاصله اقلیدسی مبنای محاسبه فاصله‌هاست

try/exceptگذاشتیم چون نسخه‌های مختلف sklearn فرق دارند

metric :نسخه‌های جدید

affinity :نسخه‌های قدیمی

**خروجی**

```
    linkage  K  silhouette
0      ward  6    0.420117
1   average  6    0.389573
2  complete  6    0.374561
3    single  6   -0.042750
Best linkage: ward  |  silhouette: 0.4201  |  K=6
```

روش Ward با امتیاز ۰.۴۲۰۱ بهترین عملکرد را دارد و بالاترین کیفیت جداسازی و انسجام خوشه‌ها را میان این چهار روش نشان می‌دهد. روش Average با امتیاز ۰.۳۸۹۶ در رتبه بعدی قرار می‌گیرد عملکرد آن قابل

قبول است اما به کیفیت Ward نمی‌رسد. روش Complete با امتیاز ۰.۳۷۴۶ ضعیف‌تر از Average عمل کرده است.

روش Single با امتیاز ۰.۰۴۲۸- عملکرد بسیار ضعیفی دارد. امتیاز منفی silhouette به این معناست که به طور میانگین، هر نقطه به خوشه‌های مجاور نزدیک‌تر از خوشه خودش است. این مشکل مستقیما به پدیده chaining در Single linkage بازمی‌گردد جایی که خوشه‌ها به صورت زنجیره‌ای و کشیده شکل می‌گیرند و مرزبندی مشخصی میان آنها ایجاد نمی‌شود.

در مقابل، روش Ward با کمینه‌سازی افزایش واریانس درون‌خوشه‌ای در هر مرحله از ادغام، خوشه‌هایی فشرده و تفکیک‌شده ایجاد می‌کند. به همین دلیل روی داده‌های عددیِ مقیاس‌شده، بهترین عملکرد را از نظر silhouette به همراه داشته است.

از این رو، روش Ward را به عنوان روش نهایی برای خوشه‌بندی سلسله‌مراتبی انتخاب می‌کنیم.

د. DBSCAN را با یک شبکهٔ کوچک برای $\varepsilon \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ و $\text{min\_samples} \in \{3, 5, 10\}$ اجرا کنید. تعداد خوشه‌ها، نسبت نویز و سیلوئت روی نقاط غیرنویز را گزارش و بهترین تنظیم را انتخاب کنید.

برای ارزیابی و انتخاب بهترین پارامترها در الگوریتم‌های خوشه‌بندی مبتنی بر چگالی نظیر DBSCAN، سه معیار اصلی را برای هر ترکیب از پارامترها گزارش می‌کنیم:

- تعداد خوشه‌ها: تعداد برچسب‌های منحصربه‌فرد به جز نویز (نقاط با برچسب ۱-)

- نسبت نویز: درصد نقاطی که به عنوان نویز شناسایی شده‌اند

- امتیاز Silhouette: تنها روی نقاط غیرنویز محاسبه می‌شود و تنها زمانی معتبر است که حداقل دو خوشه تشکیل شده باشد

در انتخاب بهترین تنظیم، هر سه معیار را به صورت همزمان در نظر می‌گیریم. یک تنظیم مطلوب باید:

۱. امتیاز silhouette قابل قبولی داشته باشد (ترجیحاً بالای ۰.۵)

۲. نسبت نویز خیلی بالا نباشد (معمولاً زیر ٪۳۰-۲۰)

۳. تعداد خوشه‌ها معقول و تفسیرپذیر باشد (نه خیلی کم، نه خیلی زیاد)

برای مثال، اگر تنظیمی silhouette حدود ۰.۶، نویز ٪۱۲ و ۵ خوشه ایجاد کند، در مقابل تنظیمی با silhouette ۰.۶۵ اما نویز ٪۴۵ و ۲ خوشه، گزینه اول ارجحیت دارد. به همین ترتیب، تنظیمی که تنها ۱ خوشه (به جز نویز) تولید کند، اساسا قابل ارزیابی با silhouette نیست.

در نهایت، بهترین تنظیم را بر اساس بیشترین امتیاز silhouette در میان ترکیب‌هایی که نویز معقول و تعداد خوشه‌های مناسب دارند، انتخاب می‌کنیم.

```
#1.4
import numpy as np
import pandas as pd

from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score

eps_list = [0.2, 0.4, 0.6, 0.8, 1.0]
min_samples_list = [3, 5, 10]

rows = []

for eps in eps_list:
    for ms in min_samples_list:
        db = DBSCAN(eps=eps, min_samples=ms, metric="euclidean")
        labels = db.fit_predict(X_scaled_df)

        n_total = len(labels)
        noise_mask = (labels == -1)
        n_noise = noise_mask.sum()
        noise_ratio = n_noise / n_total

        unique_labels = set(labels)
        n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
        non_noise_mask = ~noise_mask
        labels_non_noise = labels[non_noise_mask]
        X_non_noise = X_scaled_df.loc[non_noise_mask]

        sil = np.nan
        if n_clusters >= 2 and len(X_non_noise) > 1:
            sil = silhouette_score(X_non_noise, labels_non_noise)

        rows.append({
            "eps": eps,
            "min_samples": ms,
            "n_clusters": n_clusters,
```

```
            "noise_ratio": noise_ratio,
            "silhouette_non_noise": sil,
            "n_noise": int(n_noise),
            "n_non_noise": int(n_total - n_noise)
        })

results_dbscan = pd.DataFrame(rows)

results_dbscan_sorted = results_dbscan.sort_values(
    by=["silhouette_non_noise", "noise_ratio", "n_clusters"],
    ascending=[False, True, True],
    na_position="last"
).reset_index(drop=True)

display(results_dbscan_sorted)
```

یک Grid Search دستی روی DBSCAN انجام می‌دهیم.

پارامترها:

```
eps ∈ {0.2, 0.4, 0.6, 0.8, 1.0}
min_samples ∈ {3, 5, 10}
```

برای هر ترکیب این دو پارامتر DBSCAN را روی فضای اصلی اجرا کردیم. و خروجی DBSCAN یعنی labelsرا گرفتیم. برچسب -1 یعنی نویز.

این موارد را محاسبه و گزارش کردیم:

n_clusters تعداد خوشه‌ها (به جز نویز)

noise_ratio نسبت نقاط نویز از کل نقاط

silhouette_non_noise سیلوئت فقط روی نقاط غیرنویز (labels != -1)

فقط وقتی حساب می‌شود که حداقل ۲ خوشه داشته باشیم وگرنه NaN می‌شود

n_noise و n_non_noise تعداد نویز و غیرنویز

eps پارامتر اصلی در الگوریتم DBSCAN است که شعاع همسایگی را مشخص می‌کند. هر نقطه‌ای که فاصله آن تا نقطه مرکزی کمتر از eps باشد، همسایه محسوب می‌شود.

- eps کوچک: شرط همسایگی سخت‌گیرانه‌تر می‌شود، نقاط کمتری در همسایگی قرار می‌گیرند، در نتیجه تعداد نقاط نویز افزایش می‌یابد.

- eps بزرگ: شرط همسایگی سهل‌گیرانه‌تر می‌شود، خوشه‌های مجاور با هم ادغام می‌شوند و در حالت حدی ممکن است همه نقاط یک خوشه در نظر گرفته شوند.

min_samples دومین پارامتر اصلی در الگوریتم DBSCAN است که تعیین می‌کند یک نقطه برای هسته‌ای شدن، حداقل به چند همسایه در شعاع eps نیاز دارد.

- min_samples بزرگ‌تر: شرط هسته‌ای شدن سخت‌گیرانه‌تر می‌شود، نقاط کمتری به عنوان هسته انتخاب می‌شوند و تعداد نقاط نویز افزایش می‌یابد.

- min_samples کوچک‌تر: شرط هسته‌ای شدن آسان‌تر می‌شود، خوشه‌ها راحت‌تر تشکیل می‌شوند و نویز کاهش می‌یابد.

```
for eps in eps_list:
  for ms in min_samples_list:
```

یعنی همه‌ی ترکیب‌ها اجرا می‌شوند.

eps ۵ مقدار ، min_samples 3 مقدار مختلف وجود دارد که یعنی $3 \times 5 = 15$ حالت را بررسی می‌کنیم.

برای هر ترکیب، برچسب نویز را به صورت یک آرایه بولین تعریف می‌کنیم

- True: برچسب ۱- (نویز)

- False: غیرنویز

با استفاده از sum() روی این آرایه، تعداد نقاط نویز محاسبه شده و نسبت نویز (noise_ratio) از تقسیم این تعداد بر کل نقاط به دست می‌آید.

```
unique_labels = set(labels)
n_clusters = len(unique_labels) - (1 if -1 in unique_labels else 0)
```

set(labels) لیست برچسب‌های یکتا را می‌دهد.

اگر -1 وجود داشته باشد یعنی نویز داریم و باید آن را از تعداد خوشه‌ها کم کنیم.

```
sil = np.nan
if n_clusters >= 2 and len(X_non_noise) > 1:
 sil = silhouette_score(X_non_noise, labels_non_noise)
```

silhouette فقط وقتی تعریف دارد که حداقل 2 خوشه داشته باشیم.

اگر n_clusters=0 یا 1 باشد، silhouette معنی ندارد و sklearn خطا می‌دهد. پس در این حالت‌ها NaN می‌گذاریم تا بعداً هم قابل تشخیص باشد.

```
results_dbscan_sorted = results_dbscan.sort_values(
  by=["silhouette_non_noise", "noise_ratio", "n_clusters"],
  ascending=[False, True, True],
  na_position="last"
  reset_index(drop=True)
```

اولویت اول: silhouette بزرگ‌تر بهتر (False یعنی نزولی)

اولویت دوم: noise کمتر بهتر (True یعنی صعودی)

اولویت سوم: تعداد خوشه کمتر بهتر (True)

**خروجی**

| | eps | min_samples | n_clusters | noise_ratio | silhouette_non_noise | n_noise | n_non_noise |
|---|---|---|---|---|---|---|---|
| **0** | 0.4 | 10 | 2 | 0.850 | 0.766073 | 170 | 30 |
| **1** | 0.2 | 3 | 11 | 0.805 | 0.645880 | 161 | 39 |
| **2** | 0.6 | 10 | 4 | 0.330 | 0.529589 | 66 | 134 |
| **3** | 0.4 | 5 | 6 | 0.490 | 0.519023 | 98 | 102 |
| **4** | 0.4 | 3 | 10 | 0.295 | 0.442575 | 59 | 141 |
| **5** | 0.6 | 5 | 2 | 0.140 | 0.273047 | 28 | 172 |
| **6** | 0.6 | 3 | 3 | 0.070 | 0.214881 | 14 | 186 |
| **7** | 1.0 | 3 | 1 | 0.005 | NaN | 1 | 199 |
| **8** | 1.0 | 5 | 1 | 0.010 | NaN | 2 | 198 |
| **9** | 0.8 | 3 | 1 | 0.015 | NaN | 3 | 197 |
| **10** | 1.0 | 10 | 1 | 0.025 | NaN | 5 | 195 |
| **11** | 0.8 | 5 | 1 | 0.030 | NaN | 6 | 194 |
| **12** | 0.8 | 10 | 1 | 0.115 | NaN | 23 | 177 |
| **13** | 0.2 | 5 | 1 | 0.975 | NaN | 195 | 5 |
| **14** | 0.2 | 10 | 0 | 1.000 | NaN | 200 | 0 |

**eps=0.4, min_samples=10**

- n_clusters=2
- noise_ratio=0.85 (خیلی زیاد)
- silhouette=0.7661 (خیلی بالا، ولی روی فقط **30** نقطه‌ی غیرنویز)
- n_non_noise=30, n_noise=170

silhouette عالی است، اما 85% داده نویز شده. این یعنی DBSCAN فقط یک زیرمجموعه خیلی کوچک را خوشه‌بندی کرده و بقیه را کنار گذاشته. این تنظیم معمولا برای گزارش نهایی مناسب نیست چون عملا بیشتر دیتا را استفاده نکرده.

**eps=0.2, min_samples=3**

- n_clusters=11
- noise_ratio=0.805
- silhouette=0.6459
- n_non_noise=39

باز هم نویز بسیار زیاد (80.5٪). مشابه مورد اول silhouette بالا ولی روی تعداد کم.

**eps=0.6, min_samples=10 (گزینه‌ی بسیار خوب از نظرtrade-off )**

- n_clusters=4
- noise_ratio=0.33 (معقول)
- silhouette=0.5296 (خوب)
- n_non_noise=134 (خیلی بهتر اکثر داده‌ها خوشه‌بندی شده‌اند)

اینجا هم silhouette قابل قبول است و هم دو سوم داده‌ها داخل خوشه‌ها قرار گرفته‌اند. از نظر عملی بهترین تعادل را دارد.

**eps=0.4, min_samples=5**

- n_clusters=6
- noise_ratio=0.49
- silhouette=0.5190
- n_non_noise=102

silhouette نزدیک به مورد 3 است ولی نویز تقریبا نصف دیتا است بنابراین از نظر کیفیت استفاده از داده، از (0.6,10) ضعیف‌تر است.

بسیاری از ترکیب‌های پارامترها در DBSCAN منجر به خروجی silhouette = NaN شدند. دلیل این امر آن است که محاسبه امتیاز silhouette به وجود حداقل دو خوشه نیاز دارد. در epsهای بزرگ (مانند 0.8 و 1.0)، الگوریتم تقریبا تمام نقاط را در یک خوشه واحد می‌ریزد (n_clusters=1) و در epsهای بسیار کوچک همراه با min_samples بزرگ (مانند eps=0.2, min_samples=10)، تقریباً همه نقاط به عنوان نویز شناسایی شده و هیچ خوشه‌ای تشکیل نمی‌شود (n_clusters=0). در هر دو حالت، شرط حداقل دو خوشه برقرار نیست و بنابراین امتیاز silhouette قابل محاسبه نمی‌باشد.

ه. برای بهترین تنظیم هر خانواده (مرکزی/سلسله‌مراتبی/چگالی)، برچسب خوشه‌ها را روی فضای PCA دوبعدی رسم و تحلیل کنید.

برای ارزیابی نهایی و مقایسه عملکرد سه خانواده اصلی خوشه‌بندی، برچسب خوشه‌های هر یک از روش‌های زیر را روی فضای دوبعدی حاصل از PCA رسم کرده و تحلیل می‌کنیم:

- خوشه‌بندی مرکزی (Centroid-based): مدل K-Means با تعداد بهینه K=6

- خوشه‌بندی سلسله‌مراتبی (Hierarchical): مدل Agglomerative Clustering با linkage=ward و K=6

- خوشه‌بندی چگالی‌محور (Density-based): مدل DBSCAN با تنظیمات بهینه eps=0.6 و min_samples=10

```
#1.5
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN
from sklearn.decomposition import PCA

K_final = 6
best_linkage = "ward"
best_eps = 0.6
best_min_samples = 10

kmeans = KMeans(n_clusters=K_final, random_state=23, n_init=10)
labels_kmeans = kmeans.fit_predict(X_scaled_df)

plt.figure(figsize=(7,5))
plt.scatter(pca_df["PC1"], pca_df["PC2"], c=labels_kmeans, alpha=0.85)
plt.title(f"KMeans on PCA (K={K_final})")
plt.xlabel("PC1"); plt.ylabel("PC2")
plt.grid(True, alpha=0.3)
plt.show()

try:
    agg = AgglomerativeClustering(n_clusters=K_final,
linkage=best_linkage, metric="euclidean")
except TypeError:
    agg = AgglomerativeClustering(n_clusters=K_final,
linkage=best_linkage, affinity="euclidean")
```

```
28.
29.labels_agg = agg.fit_predict(X_scaled_df)
30.
31.plt.figure(figsize=(7,5))
32.plt.scatter(pca_df["PC1"], pca_df["PC2"], c=labels_agg, alpha=0.85)
33.plt.title(f"Agglomerative on PCA (linkage={best_linkage}, K={K_final})")
34.plt.xlabel("PC1"); plt.ylabel("PC2")
35.plt.grid(True, alpha=0.3)
36.plt.show()
37.
38.db = DBSCAN(eps=best_eps, min_samples=best_min_samples,
   metric="euclidean")
39.labels_db = db.fit_predict(X_scaled_df)
40.
41.noise_mask = (labels_db == -1)
42.n_noise = noise_mask.sum()
43.n_clusters_db = len(set(labels_db)) - (1 if -1 in set(labels_db) else 0)
44.
45.plt.figure(figsize=(7,5))
46.
47.plt.scatter(pca_df.loc[~noise_mask, "PC1"], pca_df.loc[~noise_mask,
   "PC2"],
48.            c=labels_db[~noise_mask], alpha=0.85, label="clusters")
49.
50.plt.scatter(pca_df.loc[noise_mask, "PC1"], pca_df.loc[noise_mask, "PC2"],
51.            alpha=0.85, label="noise (-1)")
52.
53.plt.title(f"DBSCAN on PCA (eps={best_eps}, min_samples={best_min_samples})
   | "
54.          f"clusters={n_clusters_db}, noise={n_noise}")
55.plt.xlabel("PC1"); plt.ylabel("PC2")
56.plt.grid(True, alpha=0.3)
57.plt.legend()
58.plt.show()
59.
60.labels_df = pd.DataFrame({
61.    "kmeans": labels_kmeans,
62.    "agglomerative": labels_agg,
63.    "dbscan": labels_db
64.}, index=X_scaled_df.index)
65.
66.labels_df.head()
```

این بخش آخر، نتایج بهترین تنظیمِ هر خانواده‌ی خوشه‌بندی را (که قبلا پیدا کرده بودیم) روی فضای ۲بعدی PCAرسم می‌کند تا به‌صورت بصری مقایسه کنیم که هر الگوریتم چگونه داده‌ها را گروه‌بندی کرده است.

آموزش (خوشه‌بندی) همچنان روی فضای اصلی انجام می‌شود فقط نمایش روی فضای کم‌بعد pca_df شامل PC1و PC2 انجام می‌شود.

```
kmeans = KMeans(n_clusters=K_final, random_state=23, n_init=10)
labels_kmeans = kmeans.fit_predict(X_scaled_df)
plt.scatter(pca_df["PC1"], pca_df["PC2"], c=labels_kmeans, ...)
```

labels_kmeans یک عدد از 0 تا 5 به هر نقطه می‌دهد.

سپس همان نقاط را روی pca_dfرسم می‌کنیم و رنگ هر نقطه را بر اساس برچسب خوشه می‌گذاریم.

شماره خوشه‌ها (0....5) و رنگ‌ها معنای ذاتی ندارند فقط برای تفکیک بصری‌اند و ممکن است در هر الگوریتم جابه‌جا شوند.

```
agg = AgglomerativeClustering(n_clusters=K_final, linkage="ward",
metric="euclidean")
labels_agg = agg.fit_predict(X_scaled_df)
plt.scatter(..., c=labels_agg, ...)
```

خروجی labels_agg باز هم 0…5 است، اما این شماره‌ها هیچ ارتباطی با شماره‌های KMeans ندارند.

```
db = DBSCAN(eps=0.6, min_samples=10)
labels_db = db.fit_predict(X_scaled_df)
```

در DBSCAN:

برچسب‌های 0..(C-1) = خوشه‌ها

برچسب -1 = noise(نقطه‌ای که در ناحیه کم‌چگال افتاده)

```
noise_mask = (labels_db == -1)
n_noise = noise_mask.sum()
n_clusters_db = ...
```

تعداد نویز و تعداد خوشه‌ها را حساب کردیم.

نقاط غیرنویز رنگی رسم می‌شوند

```
c=labels_db[~noise_mask]
```

نقاط نویز جداگانه رسم می‌شوند

```
labels_df = pd.DataFrame({
"  kmeans": labels_kmeans,
"  agglomerative": labels_agg,
"  dbscan": labels_db
})
```

**خروجی**

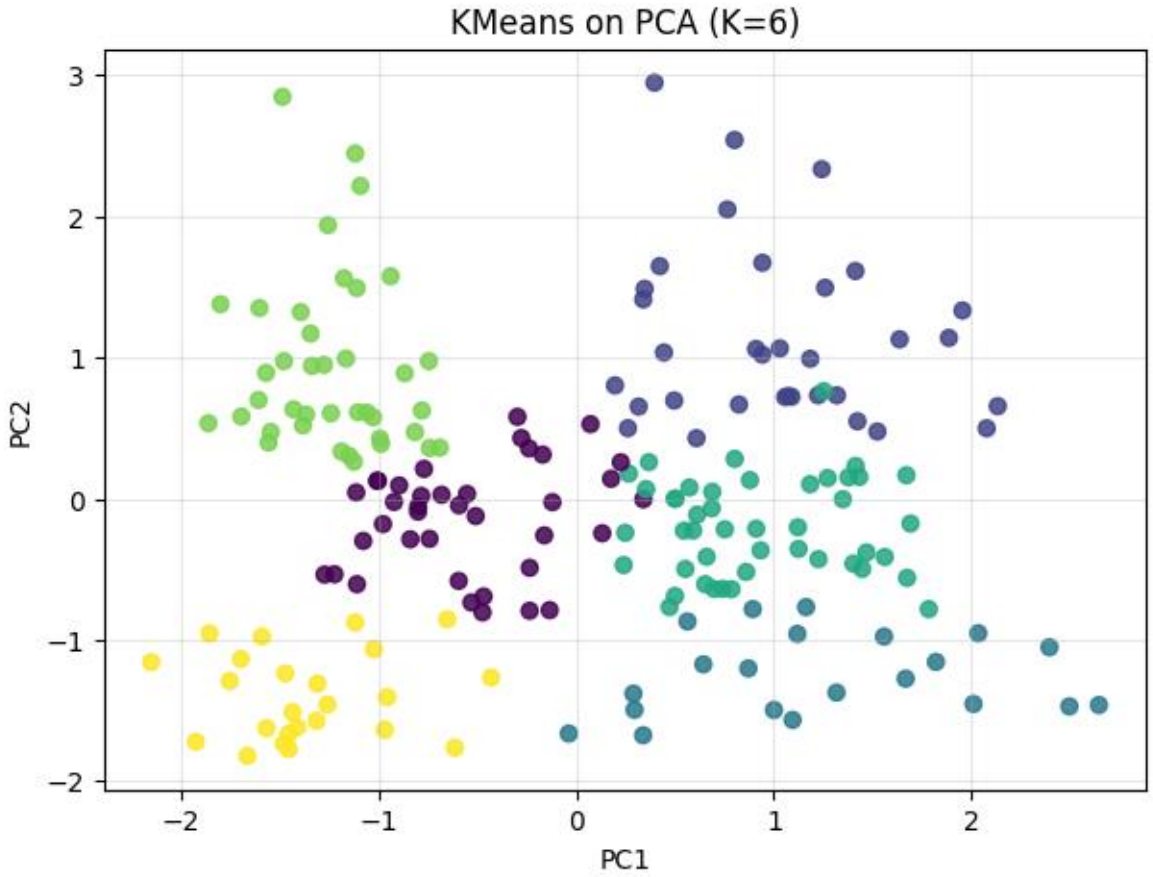
در بررسی جدول labels_df.head() مشاهده می‌کنیم

| | kmeans | agglomerative | dbscan |
|---|---|---|---|
| **0** | 5 | 4 | -1 |
| **1** | 5 | 5 | 0 |
| **2** | 2 | 4 | -1 |
| **3** | 5 | 5 | 0 |
| **4** | 2 | 4 | -1 |

- برچسب DBSCAN برای برخی سطرها برابر ۱- (نویز) است

- برچسب K-Means و Agglomerative برای همان نقاط معمولا مقادیر متفاوتی دارند

این نتایج کاملا طبیعی است. K-Means و Ward به تمام نقاط یک برچسب خوشه نسبت می‌دهند، در حالی که DBSCAN نقاط کم‌تراکم را به عنوان نویز شناسایی می‌کند. همچنین شماره خوشه‌ها بین روش‌های مختلف قابل مقایسه مستقیم نیستند.

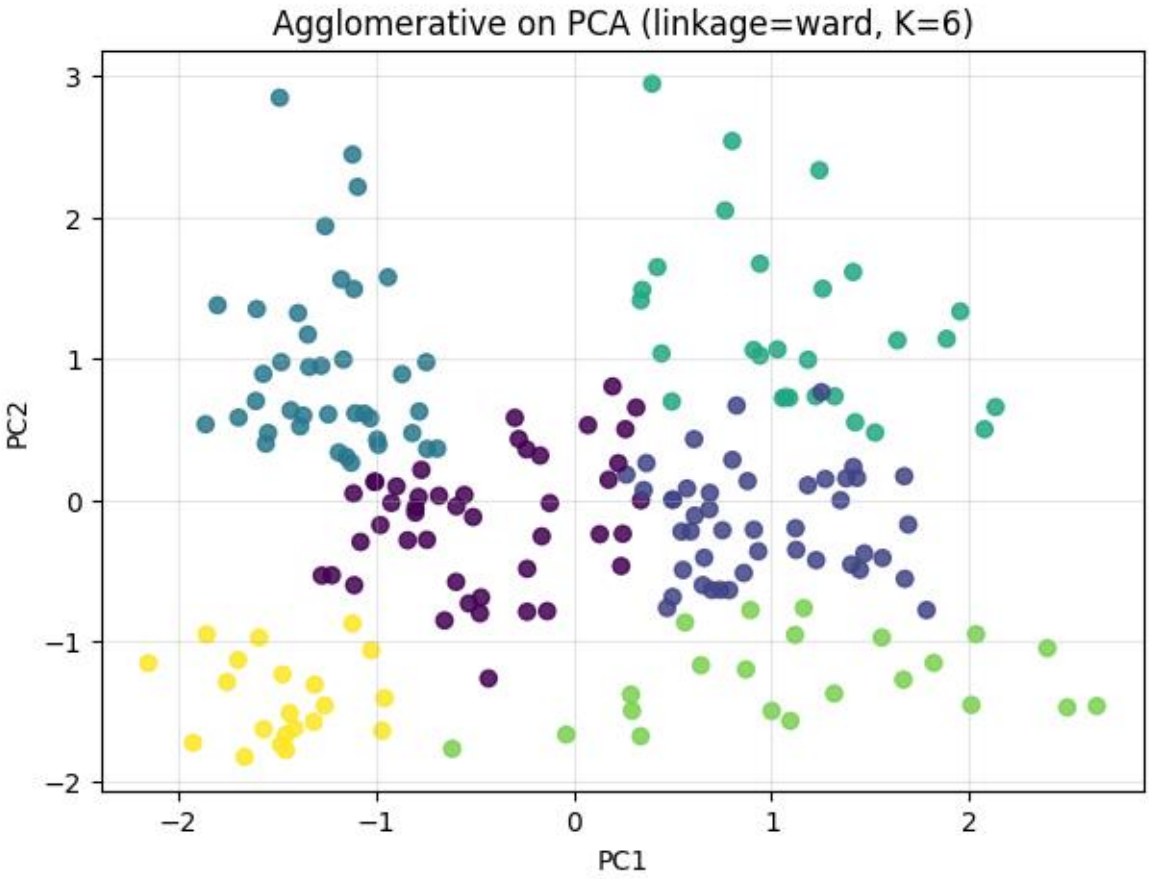



**نمودار KMeans روی PCA (K=6)**

شش گروه رنگی مشخص در فضای دوبعدی قابل مشاهده است. داده‌ها به دو نیم‌فضای کلی تقسیم شده‌اند: در سمت چپ (PC1 منفی) چند خوشه نسبتاً فشرده و در سمت راست (PC1 مثبت) چند خوشه با همپوشانی بیشتر قرار دارند.

این الگو با ماهیت K-Means سازگار است. این الگوریتم با کمینه‌سازی فاصله نقاط تا مراکز، خوشه‌هایی تقریبا کروی و مرکزگرا ایجاد می‌کند و به همه نقاط برچسب می‌دهد. برخی خوشه‌ها مانند خوشه‌های پایین‌چپ و بالاچپ تفکیک بسیار خوبی دارند، اما در نواحی میانی و راست همپوشانی جزئی دیده می‌شود.

این مشاهده با امتیاز silhouette ~0.43 همخوانی دارد یعنی تفکیکی متوسط-روبه‌بالا اما نه عالی.

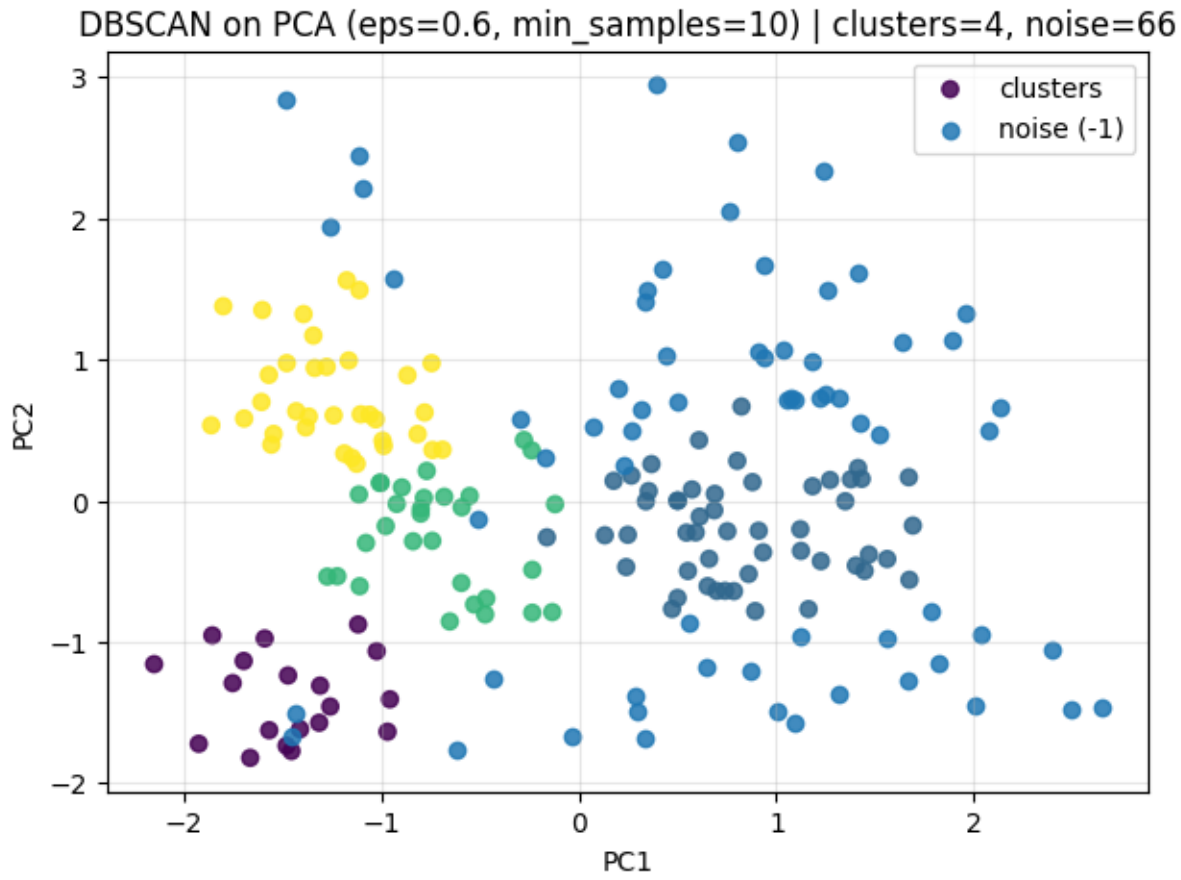



**نمودار Agglomerative (ward) روی PCA (K=6)**

در نمودار Ward با K=6 نیز شش خوشه مشخص دیده می‌شود، اما مرزبندی‌ها با K-Means کاملا یکسان نیست:

ناحیه‌های سمت راست و میانی که در K-Means به چند خوشه مجزا تقسیم شده بودند، در Ward مرزهای متفاوتی پیدا کرده‌اند. این تفاوت به ماهیت Ward برمی‌گردد که در هر مرحله ادغام، افزایش واریانس درون‌خوشه‌ای را کمینه می‌کند.

جداسازی کلی خوشه‌ها در Ward قابل قبول است، اما مرزها در برخی نواحی نرم‌تر و درهم‌تر از K-Means به نظر می‌رسند. این مشاهده با امتیاز silhouette ~0.42 برای Ward همخوانی دارد که بسیار نزدیک به K-Means است.




**نمودار DBSCAN روی PCA (eps=0.6, min_samples=10)**

clusters = 4

noise = 66 (حدود 33 درصد داده ها)

در فضای PCA، خوشه‌ها عمدتا در سمت چپ و میانی (چگالی بالاتر) قرار گرفته‌اند، در حالی که بیشتر نقاط سمت راست (PC1 مثبت) به‌عنوان نویز با رنگ آبی مشخص شده‌اند.

DBSCAN برخلاف K-Means و Ward که همه نقاط را خوشه‌بندی می‌کنند، محافظه‌کارانه عمل کرده و تنها توده‌های متراکم را به‌عنوان خوشه در نظر گرفته است. این رویکرد باعث شده امتیاز silhouette روی نقاط غیرنویز به 0.53 برسد (بالاتر از دو روش دیگر)، اما در مقابل یک‌سوم داده‌ها خوشه‌بندی نشده‌اند.

# فصل 4- بخش سوم

## پرسش سه

آ) معادله‌ی به‌روزرسانی Q-learning را بنویسید و توضیح دهید هر کدام از پارامترهای $\alpha$ (نرخ یادگیری)، $\gamma$ (ضریب تنزیل) و $\epsilon$ در سیاست $\epsilon$-greedy چه نقشی در یادگیری دارند. همچنین توضیح دهید چرا Q-learning یک روش off-policy محسوب می‌شود.

$$Q_{t+1}(s_t, a_t) = (1 - \alpha)\, Q_t(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q_t(s_{t+1}, a)\Big)$$

این بخش از ما ۳ کار می‌خواهد:

1. معادله‌ی به‌روزرسانی Q-learning را بنویسیم (همان فرمولی که در تصویر هم آمده).
2. نقش سه پارامتر را توضیح دهیم:
   - $\alpha$ نرخ یادگیری step size
   - $\gamma$ ضریب تنزیل discount factor
   - $\varepsilon$ در سیاست $\varepsilon$-greedy
3. توضیح دهیم چرا Q-learning یک روش off-policy است.

### معادله‌ی به‌روزرسانیQ-learning

Standard form:
Q_{t+1}(s_t, a_t) = (1 - alpha) × Q_t(s_t, a_t) + alpha × ( r_t + gamma × max_a Q_t(s_{t+1}, a) )

Equivalent TD-error form: خطای TD

delta_t = r_t + gamma × max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)
Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + alpha × delta_t

عبارت زیر هدف است:

r_t + gamma × max_a Q_t(s_{t+1}, a)

و اختلافش با Q_t(s_t, a_t) می‌شود خطای TD که نشان می‌دهد چقدر باید آن را اصلاح کنیم.

### نقش α (نرخ یادگیری)

$\alpha \in [0,1]$ تعیین می‌کند چقدر سریع Q را به سمت هدف جدید حرکت دهیم. α بزرگ (مثلاً 0.5 یا 1) وزن بیشتری به تجربه‌های جدید می‌دهد و یادگیری سریع‌تر است، اما نوسان زیاد و ناپایداری در محیط‌های نویزی محتمل‌تر می‌شود. α کوچک (مثلاً 0.01) تغییرات آرام و یادگیری کندتر اما پایدارتری دارد. در فرمول، (1-α) وزن دانسته‌های قبلی و α وزن اطلاعات جدید است.

**نقش γ (ضریب تنزیل)**

$\gamma \in [0,1]$ مشخص می‌کند پاداش‌های آینده چقدر ارزش دارند. $\gamma \approx 0$ یعنی عامل فقط به پاداش فوری توجه می‌کند (کوتاه‌بین) و $\gamma \approx 1$ یعنی عامل پاداش‌های دورتر را هم جدی می‌گیرد (دوراندیش). در هدف Q-learning یعنی $r_t + \gamma \cdot \max_a Q(s_{t+1},a)$، $\gamma$ دقیقا وزن بخش ارزش آینده را تعیین می‌کند.

**نقش ε در سیاست ε-greedy**

در سیاست ε-greedy، با احتمال 1-ε عمل حریصانه (بهترین عمل طبق Q فعلی) و با احتمال ε یک عمل تصادفی انتخاب می‌شود. ε بزرگ (مثلاً 0.2 یا 0.3) اکتشاف را افزایش می‌دهد و احتمال یافتن مسیرهای بهتر را بالا می‌برد، اما عملکرد کوتاه‌مدت را تضعیف می‌کند. ε کوچک (مثلاً 0.01) بهره‌برداری را افزایش می‌دهد و عملکرد پایدارتری دارد، اما خطر گیر افتادن در راه‌حل زیربهینه بیشتر می‌شود. معمولا ε را در طول آموزش کاهش می‌دهند: ابتدا زیاد برای اکتشاف، سپس کم برای تثبیت.

در off-policy، سیاستی که با آن داده جمع می‌کنیم می‌تواند با سیاستی که داریم یاد می‌گیریم فرق داشته باشد.

در Q-learning

behavior policy معمولا ε-greedyاست (برای جمع‌آوری تجربه و اکتشاف).

اما target policy که Q-learning در واقع به سمت آن یاد می‌گیرد، سیاست کاملا greedy است:

$\pi_{target}(s) = \text{argmax}_a Q(s, a)$

در به‌روزرسانی Q-learning از این عبارت استفاده می‌شود:

$\max_a Q(s_{t+1}, a)$

یعنی صرف‌نظر از اینکه واقعا در قدم بعد چه عملی انجام داده‌ایم که ممکن است تصادفی باشد برای یادگیری از بهترین عمل ممکن در حالت بعدی استفاده می‌کنیم. همین استفاده از max مستقل از عمل واقعی انتخاب‌شده باعث می‌شود Q-learning ، off-policy باشد.

ب) فرض کنید جدول $Q$ در ابتدا صفر است و گذار زیر مشاهده می‌شود:

$$(s_t = s_0,\ a_t = a_1,\ r_t = +2,\ s_{t+1} = s_1)$$

همچنین در همان لحظه داریم $\max_a Q(s_1, a) = 1.5$. با $\alpha = 0.2$ و $\gamma = 0.9$، مقدار جدید $Q(s_0, a_1)$ را محاسبه کنید (همه‌ی مراحل را شفاف بنویسید).

در این بخش از ما می‌خواهد یک آپدیت Q-learning را عددی حساب کنیم .جدول Q در ابتدا همه‌جا صفر است پس

Q(s0, a1) = 0

یک تجربه (transition) دیده شده:

(s_t = s0, a_t = a1, r_t = +2, s_{t+1} = s1)

همچنین در همان لحظه داریم:

max_a Q(s1, a) = 1.5

پارامترها:

alpha = 0.2
gamma = 0.9

Q_new(s_t, a_t) = (1 - alpha) × Q_old(s_t, a_t) + alpha × ( r_t + gamma × max_a Q_old(s_{t+1}, a) )

gamma × max_a Q(s1, a) = 0.9 × 1.5 = 1.35

Target = r_t + gamma × max_a Q(s1, a) = 2 + 1.35 = 3.35

(1 - alpha) × Q_old(s0, a1) = (1 - 0.2) × 0 = 0.8 × 0 = 0

alpha × Target = 0.2 × 3.35 = 0.67

Q_new(s0, a1) = 0 + 0.67 = 0.67

Q_new(s0, a1) = 0.67

ج) دو عامل که می‌توانند باعث کندی همگرایی یا ناپایداری یادگیری شوند را نام ببرید و برای هرکدام یک راهکار عملی پیشنهاد کنید. برای مثال می‌توانید به مواردی مثل «طراحی/مقیاس‌دهی پاداش»، «تنظیم و کاهش تدریجی $\epsilon$»، «افزایش تعداد بازدید از حالت‌ها»، «کلیپ کردن پاداش»، یا «گسسته‌سازی مناسب حالت‌ها» اشاره کنید.

**عامل Exploration 1 نامناسب ( $\varepsilon$خیلی بزرگ/خیلی کوچک یا ثابت)**

اگر $\varepsilon$ زیاد و ثابت باشد، عامل زیاد تصادفی عمل می‌کند، سیاست پایدار نمی‌شود و همگرایی کند می‌شود. اگر $\varepsilon$ خیلی کم باشد یا زود صفر شود، اکتشاف کافی انجام نمی‌شود، عامل در سیاست زیربهینه گیر می‌افتد و یادگیری عملاً قفل می‌شود. در محیط‌های تصادفی، $\varepsilon$ بدتنظیم واریانس آپدیت‌ها را بالا برده و نوسان و ناپایداری ایجاد می‌کند.

**راهکار عملی**

1. کاهش تدریجی $\varepsilon$: شروع با $\varepsilon$ بالا برای اکتشاف، سپس کاهش به مقدار کم.

$$\text{epsilon\_t} = \max(\text{epsilon\_min}, \text{epsilon\_0} \times \text{decay}^{t})$$

.2

GLIE: Greedy in the Limit with Infinite Exploration

در بلندمدت حریصانه عمل می‌کنیم اما تعداد بازدیدها کافی باقی می‌ماند.

.3

Boltzmann/Softmax exploration

به‌جای انتخاب تصادفی یکنواخت، اعمال با Q بهتر احتمال بیشتری می‌گیرند ⟵ نوسان کمتر و اکتشاف هدفمندتر.

**نرخ یادگیری $\alpha$ نامناسب یا ثابت بودن $\alpha$ در حضور نویز**

اگر $\alpha$ خیلی بزرگ باشد، آپدیت‌ها جهشی می‌شوند، Q نوسان می‌کند یا حتی واگرا می‌شود و ناپایداری ایجاد می‌کند. اگر $\alpha$ خیلی کوچک باشد، هر آپدیت اثر کمی دارد و رسیدن به مقادیر درست خیلی طول می‌کشد که منجر به همگرایی کند می‌شود. اگر $\alpha$ ثابت باشد و محیط یا پاداش نویزی باشد، Q ممکن است حول مقدار درست دائم بالا و پایین شود و تثبیت صورت نگیرد.

**راهکار عملی**

1. کاهش تدریجی $\alpha$ یا استفاده از learning-rate schedule که معمولا وابسته به تعداد بازدید از زوج حالت-عمل است؛ مانند $\text{alpha}(s,a) = 1 / N(s,a)$ یا $\text{alpha\_t} = \text{alpha\_0} / (1 + k \times t)$.

2. استفاده از $\alpha$ متفاوت برای هر زوج حالت-عمل؛ به این صورت که حالت‌هایی که بیشتر دیده شده‌اند $\alpha$ کمتری بگیرند تا تثبیت بهتری داشته باشند.

3. نرمال‌سازی یا مقیاس‌دهی پاداش، زیرا پاداش بزرگ باعث بزرگ شدن TD error و جهش آپدیت می‌شود.

**Reward scaling / Reward design بد (پاداش‌های خیلی بزرگ، نامتوازن یا نویزی)**

اگر پاداش‌ها خیلی بزرگ باشند، مثلاً $\pm 1000$ به‌جای $\pm 1$، یا دامنه شدیداً متفاوتی داشته باشند، گاهی ۰ و گاهی ۵۰۰، یا نویز زیاد داشته باشند، آنگاه Target و در نتیجه TD error بزرگ و پرنوسان می‌شود. این مسئله باعث می‌شود آپدیت‌ها جهشی شوند، Q نوسان کند و یادگیری ناپایدار شود. همچنین ممکن است عامل به‌جای یادگیری پایدار، دچار overshoot شود، به‌خصوص اگر $\alpha$ نسبتا بزرگ باشد. از طرف دیگر، اگر طراحی پاداش بد باشد، مثلاً پاداش خیلی sparse یا کم‌رخداد باشد، عامل خیلی دیر بازخورد می‌گیرد، سیگنال یادگیری ضعیف می‌شود و همگرایی کند اتفاق می‌افتد.

**راهکار عملی**

1. طراحی یا مقیاس‌دهی پاداش: یعنی پاداش را به دامنه منطقی ببریم، مثلا تقسیم بر ۱۰۰ یا نرمال‌سازی کنیم. هدف این است که $|r_t|$ به حدی باشد که TD error معقول بماند.

2. کلیپ کردن پاداش: مانند $r\_clipped = clip(r, -1, +1)$. با این کار حتی اگر پاداش خام بزرگ باشد، آپدیت‌های Q کنترل می‌شوند و نوسان کاهش می‌یابد. هرچند کلیپ کردن گاهی جزئیات شدت پاداش را از بین می‌برد، اما در بسیاری از مسائل باعث پایداری بهتر می‌شود.

**فضای حالت بزرگ یا نمایش نامناسب حالت‌ها (یا گسسته‌سازی بد)**

اگر تعداد حالت‌ها خیلی زیاد باشد یا حالت‌ها پیوسته باشند ولی گسسته‌سازی ریز و بی‌هدفی انجام شده باشد، بسیاری از حالت‌ها و زوج حالت-عمل خیلی کم دیده می‌شوند. در نتیجه Q برای آنها دیر و ناقص یاد می‌گیرد. از آنجا که بوت‌استرپینگ از $max\_a\ Q(s',a)$ استفاده می‌کند، خطاهای تخمین در حالت‌های کم‌دیده می‌تواند به حالت‌های قبلی نیز منتقل شود و یادگیری پراکنده و کندی ایجاد کند. به زبان ساده، وقتی حالت‌ها زیادند، تجربه‌ها پخش می‌شوند و هر خانه جدول Q سهم کمی از داده می‌گیرد.

**راهکار عملی**

1. گسسته‌سازی مناسب حالت‌ها: هدف کاهش تعداد حالت‌های مؤثر بدون از دست دادن اطلاعات مهم است. برای مثال اگر یک متغیر پیوسته داریم، به‌جای ۱۰۰ بازه، ۱۰ تا ۲۰ بازه معنادار قرار دهیم و بازه‌ها را با توجه به حساسیت مسئله تنظیم کنیم؛ یعنی جاهایی که تغییرات مهم‌تر است ریزتر و بقیه درشت‌تر گسسته‌سازی شوند.

2. افزایش تعداد بازدید از حالت‌ها: اگر فضای حالت بزرگ است باید تجربه بیشتری جمع شود تا زوج حالت-عمل بهتر پوشش داده شوند. این کار از طریق اپیزودهای بیشتر، شروع از حالت‌های متنوع‌تر و exploration بهتر مانند $\varepsilon$ بالاتر در ابتدا امکان‌پذیر است. هدف این است که بازدیدها یکنواخت‌تر شوند و حالت‌هایی وجود نداشته باشند که تقریباً هیچ‌وقت دیده نشوند.