دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

درس هوش مصنوعی و کارگاه

گزارش ۶: خوشه‌بندی دیتاست Iris با کمک الگوریتم ژنتیک

نگارش

مهسا گودرزی

۹۹۱۲۰۴۳

استاد اول

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

دی ۱۴۰۲

چکیده

خوشـــه‌بندی داده‌ها یکی از روش‌های مهم تجزیه و تحلیل داده‌ها اســـت که به دســـته‌بندی داده‌ها بر اسـاس شـباهت‌های آن‌ها می‌پردازد. در این مقاله، ما دو روش خوشـه‌بندی K-Means و الگوریتم ژنتیک را بر روی دیتاسـت Iris که شـامل اطلاعات درباره گونه‌های مختلف گل زنبق اسـت، اعمال کرده و نتایج آن‌ها را با هم مقایسه می کنیم. هدف ما از این مقاله، درک درست از الگوریتم ژنتیک و نحوه حل مسئله خوشه‌بندی به کمک این الگوریتم می‌باشد. با توجه به این که درصد درستی خوشه‌بندی به نحوه تعریف توابع ما در الگوریتم ژنتیک بستگی دارد، بالا بودن دقت خوشه‌بندی هدف ما در این‌جا نمی‌باشد.

واژه‌های کلیدی:

خوشه‌بندی، K-means، الگوریتم ژنتیک، Iris

فهرست مطالب

صفحه

فهرست اشکال

صفحه

فصل اول

مقدمه

مقدمه

دیتاست Iris یک مجموعه داده مشهور و کلاسیک در حوزه یادگیری ماشین[۱] و داده‌کاوی[۲] است که شامل اطلاعاتی در مورد سه گونه مختلف از گل‌های زنبق است: زنبق نوک‌زبر[۳]، زنبق رنگارنگ[۴] و زنبق ویرجینیا[۵]. این دیتاست توسط رونالد فیشر[۶]، یک آماردان و زیست‌شناس بریتانیایی، در سال ۱۹۳۶ معرفی شد. هر گل زنبق در این دیتاست با چهار ویژگی توصیف می‌شود: طول و عرض کاسبرگ و طول و عرض گلبرگ. هدف از این دیتاست این است که بتوان با استفاده از این ویژگی‌ها، گل‌های زنبق را به سه گونه موجود طبقه‌بندی کرد.

خوشه‌بندی[۷] داده‌ها یکی از روش‌های مهم تجزیه و تحلیل داده‌ها است که به دسته‌بندی داده‌ها بر اساس شباهت‌های آن‌ها می‌پردازد. برای خوشه‌بندی دیتاست Iris، می‌توان از روش‌های مختلفی مانند K-Means و الگوریتم ژنتیک[۸] استفاده کرد.

الگوریتم K-Means یکی از روش‌های پرکاربرد و ساده خوشه‌بندی است که بر اساس فاصله اقلیدسی بین نقاط داده و مراکز خوشه‌ها عمل می‌کند. این الگوریتم با تعیین تعداد خوشه‌ها به صورت از پیش تعیین شده، به طور تکراری مراکز خوشه‌ها را بهینه می‌کند تا مجموع مربعات فاصله‌ها کمینه شود. الگوریتم K-Means سریع و ساده است اما ممکن است در برخی موارد به حداقل محلی گیر کند یا به تعداد خوشه‌ها حساس باشد.

---

۱ Machine Learning

۲ Data Mining

۳ Iris Setosa

۴ Iris Versicolor

۵ Iris Virginica

۶ Ronald Fisher

۷ Clustering

۸ Genetic Algorithm

الگوریتم ژنتیک یک روش جستجوی ابتکاری و بهینه‌سازی است که از نظریه انتخاب طبیعی الهام گرفته شده است. این الگوریتم با شبیه‌سازی فرایند تکامل در طبیعت، با هدف یافتن بهترین جواب ممکن برای یک مسئله، به جستجو در فضای جواب‌های کاندید می‌پردازد. الگوریتم ژنتیک می‌تواند برای خوشه‌بندی داده‌ها نیز استفاده شود. در این روش، هر جواب کاندید یا کروموزوم، یک تقسیم‌بندی از داده‌ها به خوشه‌ها را نشان می‌دهد. این الگوریتم با اعمال عملگرهای ژنتیکی مانند جهش[۹]، انتخاب[۱۰] و ترکیب[۱۱]، جمعیتی از کروموزوم‌ها را تولید می‌کند که با توجه به تابع هدف[۱۲] یا تابع برازش[۱۳]، بهترین تقسیم‌بندی را انتخاب می‌کند. الگوریتم ژنتیک مزایایی مانند تنوع و انعطاف‌پذیری دارد اما معایبی مانند پیچیدگی و زمان‌بر بودن نیز دارد.

در ادامه، ما برخی از جزئیات پیاده‌سازی و نتایج آن را ارائه می‌دهیم.

[۹] Mutation

[۱۰] Selection

[۱۱] Crossover

[۱۲] Objective Function

[۱۳] Fitness Function

فصل دوم

خوشه‌بندی K-means

خوشه‌بندی K-means

در این فصل برای خوشه‌بندی دیتاست Iris از روش K-means استفاده می‌کنیم. برای این کار ابتدا دیتاست را بارگیری و سپس ساختار داده و اطلاعات موجود را بررسی می‌کنیم. سپس از الگوریتم K-means استفاده می‌کنیم و به بررسی خوشه‌ها و تحلیل نتایج به دست آمده از این روش می‌پردازیم.

۲-۱- آماده‌سازی داده

برای بارگیری اطلاعات موجود در Dataset می‌توان از کتابخانه Pandas استفاده کرد. با کمک توابع مختلف این کتابخانه، ما می‌توانیم ساختار کلی داده را ببینیم؛ این مجموعه داده شامل ۴ ویژگی مختلف است که نشان‌دهنده طول و عرض کاسبرگ و طول و عرض گلبرگ هستند. در ستون class این دیتاست که نشان دهنده نوع گل است، سه نوع گل متفاوت داریم که هر کدام ۵۰ نمونه را شامل می‌شوند. پس در کل این مجموعه داده شامل ۵ ستون و ۱۵۰ سطر است.

حال ما می‌خواهیم که نمونه‌های خود و برچسب‌ها که هدف ما برای تحلیل نتایج هستند به صورت متغیرهای جدا ذخیره کنیم. نمونه‌ها را که شامل ویژگی‌های داده هستند در متغیر basic_samples و برچسب‌ها در متغیر target_labels ذخیره می‌شوند. تصویر کد این بخش در شکل ۲-۱ آورده شده است.

```
basic_samples = iris_data.drop(columns='class')
target_labels = iris_data['class'].ravel()
```

شکل ۲-۱- تعریف متغیرهای basic_samples و target_labels

بعد از تعریف این متغیرها و مشخص کردن samples و target به نرمال‌سازی نمونه‌ها و سپس کاهش بعد آن‌ها با کمک PCA می‌پردازیم. همان‌طور که در شکل ۲-۲ آورده شده است، ما با کمک کلاس StandardScaler از کتابخانه sklearn.preprocessing نمونه‌ها را استاندارد کرده‌ایم، یعنی آنها را به گونه‌ای تغییر می‌دهیم که دارای میانگین ۰ و واریانس ۱ شوند. سپس با کمک کلاس PCA از کتابخانه sklearn.decomposition بعد داده‌ها را به ۲ کاهش می‌دهیم. در واقع PCA یک روش کاهش بعد است که برای کاهش تعداد متغیرهای یک داده بزرگ استفاده می‌شود. هدف از این کار این

اســـت که با حفظ بیشـــترین اطلاعات ممکن در داده، آن را به یک داده کوچک‌تر تبدیل کنیم. برای این کار، PCA از یک تبدیل خطی اســتفاده می‌کند که داده را به یک ســیســتم مختصــات جدید (مؤلفه‌های اصــلی) منتقل می‌کند که در آن جهت‌هایی که بیشــترین تغییرات را در داده نشــان می‌دهند، به راحتی قابل شناسایی هستند.

```
normalized_samples = StandardScaler().fit_transform(basic_samples)
samples_pca = PCA(n_components=2).fit_transform(normalized_samples)
```

شکل ۲-۲- نرمال‌سازی نمونه‌ها و اعمال PCA بر روی آن‌ها

با انجام تمامی این کارها، داده‌های ما برای ورود به الگوریتم K-means آماده شده‌اند.

۲-۲- الگوریتم K-means

برای پیاده‌ســـازی الگوریتم K-means بر روی داده‌های خود، از کتابخانه sklearn.cluster کمک می‌گیریم. کلاس KMeans با گرفتن ورودی‌های لازم، به راحتی می‌تولند خوشـــه‌بندی را انجام دهد. از آنجا که ما می‌خواهیم نمونه‌های خود را به سه خوشه تقسیم کنیم، ورودی n_clusters در KMeans را برابر با ۳ قرار می‌دهیم. همچنین برچســـب‌های مربوط به نتایج خوشـــه‌بندی K-means را در متغیر kmeans_labels ذخیره می‌کنیم. تصویر کد این بخش در شکل ۲-۳ آورده شده است.

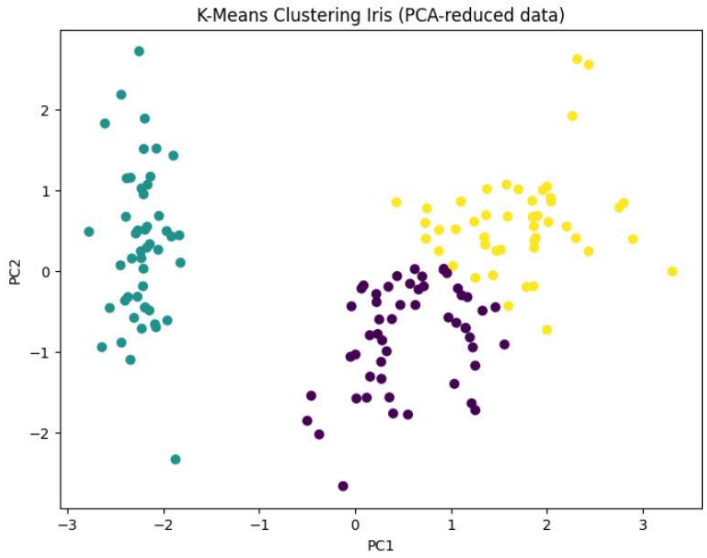
```
kmeans = KMeans(init='k-means++', n_clusters=3, n_init=10, random_state=42)
kmeans.fit(samples_pca)
kmeans_labels = kmeans.predict(samples_pca)
```

شکل ۲-۳- تصویر کد الگوریتم K-means

برای بصـری‌سـازی نتایج خوشـه‌بندی K-means می‌توانیم از کتابخانه matplotlib.pyplot اسـتفاده کنیم. این نمودار نحوه توزیع نمونه‌ها بر اســـاس PCA را نشـــان می‌دهد و برای مشـــخص کردن نحوه خوشـه‌بندی، از رنگ جداگانه برای هر خوشـه اسـتفاده کرده اسـت. کد این بخش در شـکل ۲-۴ و رسـم نمودار مربوطه در شـکل ۲-۵ دیده می‌شـود. همان‌طور که به صـورت شـهودی نیز دیده می‌شـود، تعداد داده‌های موجود در ســـه خوشـــه تفاوت زیادی با یکدیگر ندارند، بنابراین می‌توان پیش‌بینی کرد که خوشه‌بندی ما کیفیت نسبتا خوبی دارد.

```
plt.figure(figsize=(8, 6))
plt.scatter(samples_pca[:, 0], samples_pca[:, 1], c=kmeans_labels)
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("K-Means Clustering Iris (PCA-reduced data)")
plt.show()
```

شکل ۲-۴- تصویر کد مربوط به بصری‌سازی نتایج خوشه‌بندی K-means



شکل ۲-۵- نمودار بصری‌سازی نتایج خوشه‌بندی K-means

۲-۳- تحلیل نتایج خوشه‌بندی K-Means

برای تحلیل و ارزیابی عملکرد الگوریتم خوشه‌بندی K-means، چندین معیار متفاوت وجود دارد. از آنجا که داده‌های ما دارای برچسب هستند، می‌توان از معیار خلوص[۱] استفاده کرد. این معیار درصد نمونه‌هایی را که به درستی دسته‌بندی شده‌اند را نشان می‌دهد. فرمول آن به صورت زیر است:

$$Purity = \frac{۱}{N}\sum_{i=۱}^{k} max_j(n_{ij})$$

[۱] Purity Score

که در آن N تعداد کل نمونه‌ها، k تعداد کل دسته‌ها، n_{ij} تعداد نمونه‌هایی است که همزمان در دسته i و خوشه j قرار دارند و $max_j(n_{ij})$ بیشترین تعداد نمونه‌هایی است که در دسته i و یکی از خوشه‌ها قرار دارند. به عبارت دیگر، ما برای هر دسته، خوشه‌ای را انتخاب می‌کنیم که بیشترین اشتراک با آن دسته را دارد و سپس تعداد نمونه‌های مشترک را جمع می‌کنیم. سپس این مقدار را بر تعداد کل نمونه‌ها تقسیم می‌کنیم تا معیار خلوص را به دست آوریم. این معیار بین صفر و یک متغیر است و هر چه بیشتر باشد نشان‌دهنده‌ی کیفیت بالاتر خوشه‌بندی است. در ادامه نحوه پیاده‌سازی این معیار در کد را توضیح می‌دهیم.

۲-۳-۱- ماتریس پیشایندی

ماتریس پیشایندی یا Contingency Matrix در خوشه‌بندی یک ماتریسی است که نشان می‌دهد که چه تعداد از نمونه‌های داده‌ها که دسته‌های واقعی آن‌ها مشخص است، به چه خوشه‌هایی توسط الگوریتم خوشه‌بندی تخصیص داده شده‌اند. این ماتریس می‌تواند عملکرد الگوریتم خوشه‌بندی را ارزیابی کند و بگوید که چقدر دقیق است.

برای محاسبه این ماتریس بر روی داده‌های خود، می‌توانیم از کتابخانه sklearn.metrics.cluster استفاده کنیم. در شکل ۲-۶ کد این بخش و در شکل ۲-۷ جزئیات به دست آمده از Contingency Matrix نشان داده شده است.

```
k_conting_matrix = cluster.contingency_matrix(labels_true=target_labels, labels_pred=kmeans_labels)
print(f"K-Means Contingency Matrix: \n{k_conting_matrix}")
print()

for i in range(len(k_conting_matrix)):
  for j in range(len(k_conting_matrix[i])):
    print(f"(Actual Label: {label_names[i]}, Preticted Label: {j}) = {k_conting_matrix[i][j]}")
```

شکل ۲-۶- تصویر کد مربوط به K-means Contingency Matrix و جزئیات آن

```
K-Means Contingency Matrix:
[[ 0 50  0]
 [39  0 11]
 [14  0 36]]

(Actual Label: Iris-setosa, Preticted Label: 0) = 0
(Actual Label: Iris-setosa, Preticted Label: 1) = 50
(Actual Label: Iris-setosa, Preticted Label: 2) = 0
(Actual Label: Iris-versicolor, Preticted Label: 0) = 39
(Actual Label: Iris-versicolor, Preticted Label: 1) = 0
(Actual Label: Iris-versicolor, Preticted Label: 2) = 11
(Actual Label: Iris-virginica, Preticted Label: 0) = 14
(Actual Label: Iris-virginica, Preticted Label: 1) = 0
(Actual Label: Iris-virginica, Preticted Label: 2) = 36
```

شکل ۲-۷- جزئیات به دست آمده از K-Means Contingency Matrix

۲-۳-۲- مقدار Purity Score

تصویر کد محاسبه معیار خلوص یا همان Purity Score و مقدار آن در شکل ۲-۸ آورده شده است.

```
kmeans_clustering_purity = np.sum((np.max(k_conting_matrix, axis=1))) / np.sum(k_conting_matrix)
print(f"K-Means Clustering Purity: {kmeans_clustering_purity}")
```

```
K-Means Clustering Purity: 0.8333333333333334
```

شکل ۲-۸- محاسبه K-Means Purity Score

پس مقدار Purity Score حدودا برابر با ۰.۸۳ شد که این نشان دهنده آن است که الگوریتم خوشه‌بندی K-means ما عملکرد نسبتا خوبی دارد اما به طور کاملا درست نمی‌تواند خوشه‌بندی را انجام دهد.

فصل سوم

خوشه‌بندی با استفاده از الگوریتم ژنتیک

خوشه‌بندی با استفاده از الگوریتم ژنتیک

در این فصل برای خوشه‌بندی دیتاست Iris از الگوریتم ژنتیک استفاده می‌کنیم. بارگیری و آماده‌سازی دیتا را از قبل انجام داده‌ایم. در ادامه اجزای این الگوریتم را توضیح می‌دهیم.

۳-۱- جمعیت اولیه

در اینجا ما دو تابع random_chromosome و generate_population تعریف کرده‌ایم که این دو کمک می‌کنند که یک جمعیت اولیه از کروموزوم‌ها را برای حل مسئله خوشه‌بندی دیتاست Iris ایجاد کنیم.

تابع random_chromosome یک کروموزوم تصادفی با طول ۱۵۱ عنصر ایجاد می‌کند. این کروموزوم نشان‌دهنده یک جواب کاندید برای مسئله خوشه‌بندی دیتاست Iris است. هر کدام از ۱۵۰ عنصر اول کروموزوم یک عدد صحیح بین ۰ تا ۲ است که نشان‌دهنده برچسب خوشه‌ای است که به نمونه متناظر تخصیص داده شده است. برای مثال، اگر عنصر اول کروموزوم برابر با ۱ باشد، به این معنی است که نمونه اول دیتاست Iris به خوشه ۱ تعلق دارد. عنصر آخر کروموزوم یک عدد اعشاری است که نشان‌دهنده میزان سازگاری (fitness) آن کروموزوم است. میزان سازگاری یک کروموزوم نشان‌دهنده عملکرد آن کروموزوم در حل مسئله است و معمولا با استفاده از یک تابع هدف محاسبه می‌شود. در ابتدا میزان سازگاری هر کروموزوم را برابر با صفر قرار می‌دهیم و بعدا با استفاده از تابع Fitness آن را به‌روزرسانی می‌کنیم. برای ایجاد یک کروموزوم تصادفی، از تابع randint از کتابخانه numpy استفاده می‌کنیم که یک آرایه از اعداد صحیح تصادفی با مقادیر و اندازه مشخص را برمی‌گرداند. سپس با استفاده از تابع append از همان کتابخانه، عنصر صفر را به انتهای آرایه اضافه می‌کنیم و کروموزوم را برمی‌گردانیم.

تابع generate_population یک جمعیت از کروموزوم‌ها را با استفاده از تابع random_chromosome ایجاد می‌کند. این جمعیت شامل تعداد مشخصی از کروموزوم‌ها است که به عنوان پارامتر به تابع داده می‌شود. برای مثال، اگر تعداد کروموزوم‌ها برابر با ۱۰۰۰ باشد، این تابع یک جمعیت از ۱۰۰۰ کروموزوم تصادفی را تولید می‌کند. برای ایجاد یک جمعیت، از یک حلقه for استفاده می‌کنیم که به تعداد کروموزوم‌ها، تابع random_chromosome را صدا می‌زند و نتیجه را در یک آرایه numpy ذخیره می‌کنیم. سپس آرایه حاوی کروموزوم‌ها را برمی‌گردانیم.

تصویر کد این دو تابع در شکل ۳-۱ آورده شده است.

```
def random_chromosome():
  chromosome = np.random.randint(0, 3, size=150)
  # The last element of each chromosome indicates the fitness of that chromosome, which is initially set to zero
  chromosome = np.append(chromosome, 0.0)
  return chromosome

def generate_population(number_of_chromosomes):
  return np.array([random_chromosome() for _ in range(number_of_chromosomes)])
```

شکل ۳-۱- تصویر کد دو تابع random_chromosome و generate_population

بعد از تعریف این دو تابع، ما یک جمعیت اولیه از کروموزوم‌های تصادفی ایجاد می‌کنیم و تعداد اولیه این جمعیت را برابر با ۱۰۰۰ قرار می‌دهیم. این جمعیت اولیه در متغیری با نام initial_population ذخیره می‌شود.

۳-۲- تابع Fitness

تابع fitness که تصویر کد آن در شکل ۳-۲ آورده شده است، به ما کمک می‌کند که میزان سازگاری یک کروموزوم را برای مسئله خوشه‌بندی دیتاست Iris محاسبه کنیم. در این تابع، ابتدا یک لیست از شاخص‌های نمونه‌هایی که به هر یک از ۳ خوشه تعلق دارند ایجاد می‌کنیم. این کار با استفاده از تابع where از کتابخانه numpy انجام می‌شود که یک آرایه بولی را دریافت می‌کند و شاخص‌های عناصری را که True هستند را برمی‌گرداند. در اینجا ما از کروموزوم به عنوان آرایه بولی استفاده می‌کنیم و برای هر یک از مقادیر ۰ تا ۲، شاخص‌های نمونه‌هایی را پیدا می‌کنیم که با آن مقدار برابر هستند. نتیجه در متغیر cluster_indices ذخیره می‌شود.

سپس یک لیست از داده‌های نمونه‌هایی که به هر یک از ۳ خوشه تعلق دارند ایجاد می‌کنیم. این کار با استفاده از تابع iloc از کتابخانه pandas انجام می‌شود که یک شیء DataFrame را دریافت می‌کند و با استفاده از شاخص‌های عددی، سطرها یا ستون‌های مورد نظر را برمی‌گرداند. در اینجا ما از شیء iris_data به عنوان DataFrame استفاده می‌کنیم و با استفاده از شاخص‌هایی که در مرحله قبل پیدا کرده‌ایم، داده‌های نمونه‌های هر خوشه را انتخاب می‌کنیم. نتیجه در متغیر clusters ذخیره می‌شود.

در ادامه، تابع میانگین فاصله نمونه‌های هر خوشه از مرکز خوشه را محاسبه می‌کند. این کار با استفاده از تابع mean و تابع linalg.norm از کتابخانه numpy انجام می‌شود که به ترتیب میانگین و نرم یک

آرایه را محاسبه می‌کنند. در اینجا ما برای هر یک از سه خوشه، فاصله نمونه‌ها را از میانگین ویژگی‌های خوشه که نشان‌دهنده مرکز خوشه است را با استفاده از نرم اقلیدسی محاسبه می‌کنیم و سپس میانگین این فاصله‌ها را برای هر خوشه به دست می‌آوریم و نتیجه را در متغیر differences ذخیره می‌کنیم.

سپس تابع نسبت اندازه هر خوشه به اندازه کل دیتاست را محاسبه می‌کند. در اینجا ما برای هر یک از سه خوشه، تعداد نمونه‌های آن را تقسیم بر تعداد کل نمونه‌های دیتاست می‌کنیم و نسبت را به دست می‌آوریم و نتیجه را در متغیر sizes ذخیره می‌کنیم.

میزان سازگاری کروموزوم با استفاده از فرمول زیر محاسبه می‌شود:

$$\sum_{i=1}^{k} d_i \times s_i$$

که در آن k تعداد خوشه‌ها، d_i میانگین فاصله نمونه‌های خوشه i از مرکز خوشه و s_i نسبت اندازه خوشه i به اندازه کل دیتاست است. این فرمول نشان‌دهنده میزان پراکندگی داده‌ها در خوشه‌ها است که هر چه کمتر باشد، نشان‌دهنده خوشه‌بندی بهتر است. این کار را با استفاده از تابع sum و تابع zip از کتابخانه numpy انجام می‌شود. در اینجا ما مقادیر متغیرهای differences و sizes را با هم ضرب می‌کنیم و سپس جمع آن‌ها را محاسبه می‌کنیم و نتیجه را در متغیر fitness_value ذخیره می‌کنیم.

در نهایت مقدار متغیر fitness_value به عنوان خروجی تابع برگردانده می‌شود. این مقدار نشان‌دهنده میزان سازگاری کروموزوم با مسئله خوشه‌بندی دیتاست Iris است که هر چه کمتر باشد، نشان‌دهنده کروموزوم بهتر است.

```python
def fitness(chromosome):
    cluster_indices = [np.where(chromosome == i)[0] for i in range(3)]
    clusters = [iris_data.iloc[indices] for indices in cluster_indices]

    differences = [np.mean([np.linalg.norm(cluster[col] - cluster[col].mean()) for col in iris_data.columns[:-1]]) for cluster in clusters]
    sizes = [len(cluster) / len(iris_data) for cluster in clusters]

    fitness_value = sum(diff * size for diff, size in zip(differences, sizes))
    return fitness_value
```

شکل ۳-۲- کد تابع fitness

در ادامه مقدار fitness را برای هر کروموزوم جمعیت اولیه محاسبه و آن را در آخرین عنصر هر کروموزوم جایگذاری می‌کنیم.

۳-۳- تابع Selection

تابع selection به ما کمک می‌کند که از میان یک جمعیت اولیه از کروموزوم‌ها، تعداد مشخصی از آن‌ها را به عنوان والدین برای تولید نسل بعدی انتخاب کنیم. این تابع دو پارامتر را دریافت می‌کند؛ initial_population و num. پارامتر initial_population یک جمعیت اولیه از کروموزوم‌ها و پارامتر num یک عدد صحیح است که نشان‌دهنده تعداد کروموزوم‌هایی است که می‌خواهیم به عنوان والدین انتخاب کنیم.

تابع selection با استفاده از تابع sorted از کتابخانه numpy، کروموزوم‌های جمعیت اولیه را بر اساس مقدار fitness آن‌ها مرتب می‌کند. برای مرتب‌سازی کروموزوم‌ها، از پارامتر key استفاده می‌کنیم که یک تابع لامبدا است که عنصر آخر هر کروموزوم را برمی‌گرداند. این کار باعث می‌شود که کروموزوم‌ها به صورت صعودی بر اساس مقدار fitness مرتب شوند.

تابع selection تعداد num عنصر اول آرایه مرتب شده را انتخاب می‌کند. با این کار، کروموزوم‌هایی که مقدار fitness بیشتری دارند، حذف شوند و کروموزوم‌هایی که مقدار fitness کمتری دارند، حفظ شوند. این کروموزوم‌های انتخاب شده، به عنوان والدین برای تولید نسل بعدی در نظر گرفته می‌شوند. در نهایت کروموزوم‌های انتخاب شده به عنوان خروجی برگردانده می‌شوند.

تصویر کد این تابع در شکل ۳-۳ قابل مشاهده است.

```
def selection(initial_population, num):
    return np.array(sorted(initial_population, key=lambda x: x[-1]))[:num]
```

شکل ۳-۳- تصویر کد تابع selection

در ادامه به کمک این تابع، دو کروموزوم که مقدار fitness کمتری دارند یعنی کاندید بهتری برای خوشه‌بندی هستند را به عنوان والدین نسل بعدی انتخاب می‌کنیم و در همان متغیر initial_population ذخیره می‌کنیم.

۳-۴- تابع Mutation

تابع mutation یک کروموزوم را با احتمال مشخصی تغییر می‌دهد تا تنوع ژنتیکی را حفظ کند. این تابع دو پارامتر chromosome و mutation_rate را دریافت می‌کند. پارامتر mutation_rate یک عدد اعشاری بین ۰ تا ۱ است که نشان‌دهنده احتمال تغییر هر عنصر از کروموزوم است.

تابع mutation با استفاده از یک حلقه for، به ترتیب هر عنصر از کروموزوم را بررسی می‌کند که در اینجا با توجه به دیتاست Iris، ما ۱۵۰ عنصر را برای هر کروموزوم باید بررسی کنیم. این تابع با استفاده از تابع random.rand از کتابخانه numpy، یک عدد اعشاری تصادفی بین ۰ تا ۱ تولید می‌کند و آن را با مقدار mutation_rate مقایسه می‌کند. اگر عدد تصادفی کوچکتر از mutation_rate باشد، به این معنی است که تغییری در عنصر مورد نظر کروموزوم ایجاد شود. در غیر این صورت، عنصر کروموزوم بدون تغییر باقی می‌ماند. اگر عنصری باید تغییر داده شود، تابع mutation با استفاده از تابع random.randint از کتابخانه numpy، یک عدد صحیح تصادفی بین ۰ تا ۲ تولید می‌کند و آن را به عنوان مقدار جدید عنصر مورد نظر کروموزوم قرار می‌دهد. این کار باعث می‌شود که برچسب خوشه‌ای که به نمونه متناظر تخصیص داده شده است، تغییر کند. سپس در ادامه، کروموزوم تغییر یافته به عنوان خروجی تابع برگردانده می‌شود.

تصویر کد این تابع در شکل ۳-۴ آورده شده است.

```
def mutation(chromosome, mutation_rate):
    for i in range(150):
        if np.random.rand() < mutation_rate:
            chromosome[i] = np.random.randint(0, 3)
    return chromosome
```

شکل ۳-۴- تصویر کد تابع mutation

۳-۵- تابع Crossover

تابع crossover این تابع به ما کمک می‌کند که از میان یک جمعیت از کروموزوم‌ها، نسل جدیدی از کروموزوم‌ها را با کمک ترکیب و تغییر کروموزوم‌ها ایجاد کنیم. این کار باعث می‌شود که اطلاعات مفید

از والدین به فرزندان منتقل شود و تنوع ژنتیکی در جمعیت افزایش یابد. این تابع سه پارامتر population، cross_rate و mutation_rate را دریافت می‌کند. پارامتر population یک آرایه از کروموزوم‌ها است که نشان‌دهنده جمعیت فعلی است. پارامتر cross_rate یک عدد اعشاری بین ۰ تا ۱ است که نشان‌دهنده احتمال پیوند بین دو والد است. پارامتر mutation_rate یک عدد اعشاری بین ۰ تا ۱ است که نشان‌دهنده احتمال تغییر هر عنصر از یک کروموزوم است.

در ابتدا تابع crossover یک کپی از آرایه population را در متغیر new_population ذخیره می‌کند. این کپی برای نگه‌داری کروموزوم‌های جدید استفاده می‌شود. سپس با استفاده از دو حلقه for، همه جفت‌های ممکن از کروموزوم‌های جمعیت فعلی را بررسی می‌کند. این جفت‌ها به عنوان والدین برای تولید فرزندان در نظر گرفته می‌شوند و در ادامه با استفاده از یک حلقه for، دو فرزند از هر جفت والد تولید می‌شود.

تابع crossover با استفاده از تابع where از کتابخانه numpy، یک کروموزوم فرزند را با انتخاب تصادفی عناصر از دو کروموزوم والد ایجاد می‌کند. این کار با استفاده از یک آرایه از اعداد تصادفی بین ۰ تا ۱ انجام می‌شود که با مقدار cross_rate مقایسه می‌شوند. اگر عدد تصادفی کوچکتر از cross_rate باشد، عنصر از کروموزوم parent1 انتخاب می‌شود. در غیر این صورت، عنصر از کروموزوم parent2 انتخاب می‌شود.

در ادامه، تابع با استفاده از تابع mutation، کروموزوم فرزند را با احتمال مشخصی تغییر می‌دهد و سپس کروموزوم فرزند را به آرایه new_population اضافه می‌کند. در نهایت آرایه new_population به عنوان خروجی تابع برگردانده می‌شود. این آرایه شامل کروموزوم‌های جمعیت فعلی و کروموزوم‌های جدید تولید شده است.

تصویر کد این تابع در شکل ۳-۵ نشان داده شده است.

```
def crossover(population, cross_rate, mutation_rate):
  new_population = population.copy()

  for parent1 in new_population:
    for parent2 in new_population:

      # Number of childs = 2
      for _ in range(2):
        child = np.where(np.random.rand() < cross_rate, parent1, parent2)
        child = mutation(child, mutation_rate)

        new_population = np.append(new_population, [child], axis=0)
  return new_population
```

شکل ۳-۵- تصویر کد تابع crossover

۳-۶- سیر تکامل

در ادامه الگوریتم ژنتیک کد ما یک حلقه را اجرا می‌کند. این حلقه ۱۰۰۰ نسل از کروموزوم‌ها را تولید می‌کند و مقدار fitness بهترین کروموزوم هر نسل را ذخیره می‌کند. برای این کار ابتدا جمعیت اولیه کروموزوم‌ها در متغیر new_population کپی می‌شوند. سپس یک لیست خالی به نام best_fitness برای ذخیره مقدار fitness بهترین کروموزوم هر نسل ایجاد می‌شود. در ادامه در حلقه‌ای که ۱۰۰۰ بار تکرار می‌شود، به ترتیب مراحل زیر اجرا می‌شوند:

۱- پیوند دادن کروموزوم‌های جمعیت فعلی با cross_rate برابر با ۰.۵ و mutation_rate برابر با ۰.۰۵ و ایجاد نسل جدید

۲- محاسبه مقدار fitness هر کروموزوم جدید

۳- انتخاب دو کروموزوم برتر از جمعیت جدید

۴- افزودن مقدار fitness کروموزوم برتر جمعیت جدید به لیست best_fitness

بعد از اجرای این حلقه، لیست best_fitness چاپ می‌شود که نشان دهنده‌ی مقدار fitness بهترین کروموزوم در هر نسل است.

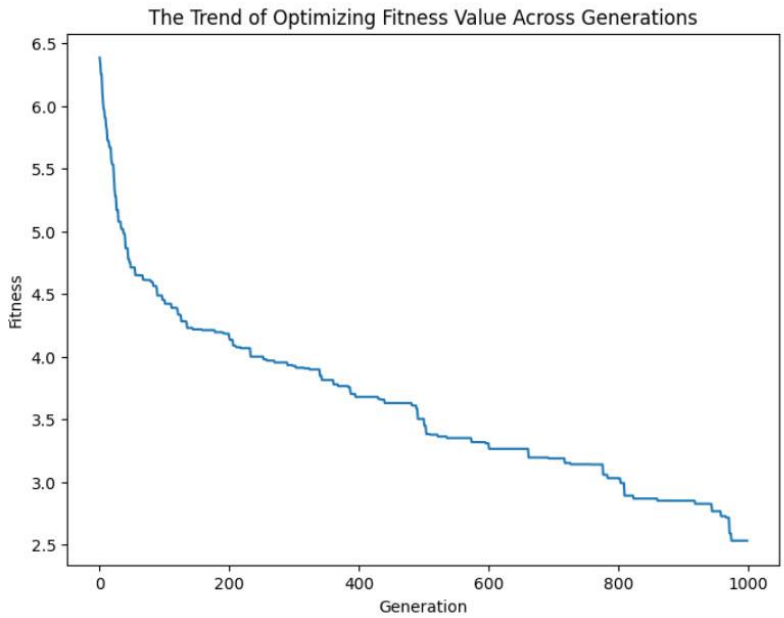
تصویر کد این قسمت در شکل ۳-۶ آورده شده است.

```
# Create an Empty List for the Best Fitness
new_population = initial_population.copy()
best_fitness = []

# Loop for 1000 Generations
for i in range(1000):
    # Perform Crossover on the Current Population
    new_population = crossover(new_population, 0.5, 0.05)
    # Calculate the Fitness for the New Population
    for chromosome in new_population:
      chromosome[-1] = fitness(chromosome[:-1])
    # Select the Best 2 individuals from the New Population
    new_population = selection(new_population, 2)
    best_fitness.append(new_population[0][-1])

for i in range(len(best_fitness)):
  print(f"The Best Amount of Fitness in Generation {i}: {best_fitness[i]}")
```

شکل ۳-۶- تصویر کد حلقه اصلی الگوریتم ژنتیک

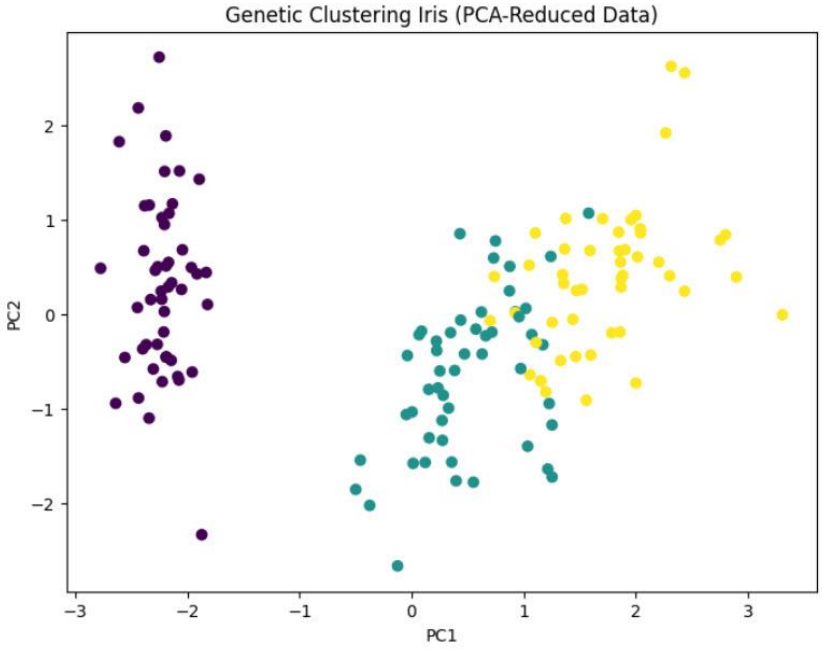
۳-۷- ارزیابی الگوریتم ژنتیک

همان‌طور که در شکل ۳-۷ دیده می‌شود، مقدار fitness بهترین کروموزوم هر نسل از نسل‌های قبلی کمتر است، که این به معنی آن است که آن کروموزوم کاندید بهتری برای خوشه‌بندی است و نتایج بهتری را می‌تواند ارائه بدهد.



شکل ۳-۷- روند تغییر مقدار Fitness بهترین کروموزوم هر نسل

بنابراین ما بهترین کروموزوم آخرین نسل را به عنوان کاندید خوشه‌بندی خود انتخاب و آن را در یک متغیر به نام best_solution ذخیره می‌کنیم. این متغیر شامل برچسب‌های پیش‌بینی شده برای خوشه‌بندی می‌باشد. نمودار پراکندگی و نحوه خوشه‌بندی الگوریتم ژنتیک با توجه به متغیر best_solution، در شکل ۳-۸ نشان داده شده است. همان‌طور که به صورت شهودی از این نمودار می‌توان برداشت کرد، خوشه‌بندی ما به کمک الگوریتم ژنتیک عملکرد نسبتا خوبی داشته است.



شکل ۳-۸- نمودار بصری‌سازی نتایج خوشه‌بندی به کمک الگوریتم ژنتیک

نحوه محاسبه Contingency Matrix و Purity Score مانند الگوریتم خوشه‌بندی K-Means است. در شکل ۳-۹ جزئیات Genetic Contingency Matrix و در شکل ۳-۱۰ مقدار محاسبه شده Genetic Purity Score آورده شده است.

```
Genetic Contingency Matrix:
[[50  0  0]
 [ 0 43  7]
 [ 0  8 42]]

(Actual Label: Iris-setosa, Preticted Label: 0) = 50
(Actual Label: Iris-setosa, Preticted Label: 1) = 0
(Actual Label: Iris-setosa, Preticted Label: 2) = 0
(Actual Label: Iris-versicolor, Preticted Label: 0) = 0
(Actual Label: Iris-versicolor, Preticted Label: 1) = 43
(Actual Label: Iris-versicolor, Preticted Label: 2) = 7
(Actual Label: Iris-virginica, Preticted Label: 0) = 0
(Actual Label: Iris-virginica, Preticted Label: 1) = 8
(Actual Label: Iris-virginica, Preticted Label: 2) = 42
```

شکل ۳-۹- جزئیات به دست آمده از Genetic Contingency Matrix

```
genetic_clustering_purity = np.sum((np.max(gen_conting_matrix, axis=1))) / np.sum(gen_conting_matrix)
print(f"Genetic Clustering Purity: {genetic_clustering_purity}")
```

```
Genetic Clustering Purity: 0.9
```

شکل ۳-۱۰- محاسبه Genetic Purity Score

مقدار Purity Score برای خوشه‌بندی الگوریتم ژنتیک ما برابر با ۰.۹ شده است که نشان‌دهنده‌ی عملکرد خوب این الگوریتم در مسئله خوشه‌بندی می‌باشد. با توجه به نقش اعداد تصادفی در عملکرد این الگوریتم، در هر بار اجرای این برنامه، مقدار purity score می‌تواند کمتر یا بیشتر از این مقدار شود، ولی در هر صورت، این الگوریتم عملکرد خوبی در خوشه‌بندی دیتاست Iris دارد. یکی از دلایل این امر به تعداد نسل‌های گسترش یافته برمی‌گردد، با توجه به این‌که بهترین کروموزوم هر نسل، نسبت به نسل‌های قبلی مقدار fitness کمتری دارد، پس هر چقدر تعداد نسل‌های گسترش یافته بیشتر شود، الگوریتم ما نیز در خوشه‌بندی بهتر عمل می‌کند. اما باید به این نکته توجه داشت که افزایش تعداد نسل، یا پیچیده‌تر کردن توابع مربوط به الگوریتم ژنتیک علیرغم بهبود عملکرد الگوریتم ژنتیک، ممکن است حافظه و زمان بیشتری را مصرف کند.

فصل چهارم

لینک کد گزارش

لینک کد گزارش

کد خوشه‌بندی دیتاست Iris به دو روش K-Means و الگوریتم ژنتیک در Google Colab نوشته شده و لینک آن در زیر قابل دسترسی است:

لینک کد گزارش ۶ در Google Colab

فصل پنجم

جمع‌بندی و نتیجه‌گیری

جمع‌بندی و نتیجه‌گیری

در این مقاله، ما دو روش خوشه‌بندی K-Means و الگوریتم ژنتیک را بر روی دیتاست Iris که شامل اطلاعات درباره گونه‌های مختلف گل زنبق است، اعمال و مقایسه کردیم. ما از معیار purity score برای مقایسه دقت هر دو روش با توجه به برچسب‌های واقعی داده‌ها استفاده کردیم. این مقدار برای خوشه‌بندی با الگوریتم ژنتیک تقریبا ۰.۹ و برای خوشه‌بندی با الگوریتم K-Means تقریبا ۰.۸۳ شد. نتایج نشان داد که در این‌جا الگوریتم ژنتیک نسبت به الگوریتم K-Means عملکرد بهتری دارد و می‌تواند خوشه‌هایی با همبستگی داخلی بالا و همبستگی بین خوشه‌ای پایین تشکیل دهد. این می‌تواند به خاطر نوع تعریف ما از توابع مربوط به الگوریتم ژنتیک و همچنین تعداد نسل‌های تولید شده باشد. واضح است که هر چقدر تعاریف این نوع توابع دقیق‌تر و از لحاظ پیچیدگی معنادارتر باشند، عملکرد الگوریتم نیز بهبود می‌یابد و نتایج بسیار بهتری را ارائه می‌دهد. در سیر تکاملی الگوریتم ما، مقدار Fitness بهترین کروموزوم در هر نسل کمتر شد و این بدان معنی است که الگوریتم با پیشروی در نسل‌های بعدی، دیتاست را دقیق‌تر و بهتر خوشه‌بندی می‌کند. هر چقدر تعاریف توابع پیچیده‌تر و تعداد نسل‌های تولید شده نیز بیشتر باشد، الگوریتم حافظه و زمان بیشتری را مصرف می‌کند.

منابع

"Faradars," [Online]. Available: https://b.fdrs.ir/1l4.

"Faradars," [Online]. Available: https://b.fdrs.ir/2az.

"Maktabkhooneh," [Online]. Available: https://maktabkhooneh.org/mag/genetic-algorithm-in-artificial-intelligence/.

"Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Genetic_algorithm.

"Geeks," [Online]. Available: https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/.

"Geeks," [Online]. Available: https://www.geeksforgeeks.org/genetic-algorithms/.

"UCIrvine," [Online]. Available: https: https://archive.ics.uci.edu/dataset/53/iris.