دانشگاه تربیت دبیر شهید رجائی

دانشکدهٔ مهندسی کامپیوتر

گزارش پروژهٔ داده‌کاوی

پاک‌سازی، دسته‌بندی و خوشه‌بندی دیتاست دیابت

نام و نام خانوادگی دانشجو: محسن الهی‌فرد

شمارهٔ دانشجویی: ۴۰۰۱۲۳۱۰۰۵

رشته: مهندسی کامپیوتر

استاد درس: دکتر نگین دانشپور

فهرست مطالب

چکیده ۳

فصل اول – معرفی پروژه، ابزارها و دیتاست ۴

۱-۱- مقدمه ۴

۱-۲- معرفی ابزارها ۵

۱-۳- بارگذاری و معرفی دیتاست ۶

فصل دوم – پاک‌سازی داده ۷

۲-۱- مقدمه ۷

۲-۲- شناسایی و پاک‌سازی مقادیر گمشده، تکراری و خارج از محدوده ۸

فصل سوم – پیاده‌سازی الگوریتم‌های دسته‌بندی ۱۳

۳-۱- مقدمه ۱۳

۳-۲- پیاده‌سازی دسته‌بندی با الگوریتم درخت تصمیم ۱۳

۳-۳- پیاده‌سازی دسته‌بندی الگوریتم k-نزدیک‌ترین همسایه ۲۲

فصل چهارم – پیاده‌سازی الگوریتم‌های خوشه‌بندی ۲۵

۴-۱- مقدمه ۲۵

۴-۲- پیاده‌سازی خوشه‌بندی با الگوریتم K-Means ۲۵

۴-۳- پیاده‌سازی خوشه‌بندی با الگوریتم سلسله‌مراتبی ۲۹

چکیده

در این گزارش، به بررسی الگوریتم‌های مختلف از جمله دسته‌بندی با درخت تصمیم و KNN و خوشه‌بندی K-Means و سلسله‌مراتبی پرداخته می‌شود. ابتدا، نحوهٔ پاک‌سازی داده‌ها توضیح داده می‌شود؛ سپس، مفاهیم درخت تصمیم و محاسبات مربوط به آنتروپی تشریح می‌شود و در ادامه، مراحل ساخت و ارزیابی مدل با استفاده از معیارهای دقت و ماتریس درهم‌ریختگی توضیح داده می‌شود. سپس، الگوریتم KNN با انتخاب نزدیک‌ترین همسایه‌ها براساس فاصلهٔ اقلیدسی بررسی می‌گردد. در بخش خوشه‌بندی، روش‌های K-Means و سلسله‌مراتبی تحلیل می‌شوند و مفاهیمی مانند روش Elbow Method برای تعیین تعداد خوشه‌ها توضیح داده می‌شوند. در نهایت، تجسم ساختار خوشه‌ها با استفاده از نمودار دندروگرام ارائه و مزایا و معایب این الگوریتم‌ها مقایسه می‌گردد.

فصل اول

معرفی پروژه، ابزارها و دیتاست

۱-۱- مقدمه

در این گزارش به مطالعه بر روی دیتاست دیابت پرداخته می‌شود و فرایندهای مختلفی ازجمله آماده‌سازی داده‌ها، پاک‌سازی[1] و مدل‌سازی را شامل می‌شود. تحلیل داده‌های پزشکی اهمیت بسیاری در پیش‌بینی و مدیریت بیماری‌ها دارد و می‌تواند به تصمیم‌گیری بهتر در حوزهٔ سلامت کمک کند. منابع این پروژه در آدرس github.com/mohsenelahifard/diabetes-data-mining قابل مشاهده اند.

ابتدا داده‌های خام که شامل اطلاعاتی همچون سن، شاخص توده بدنی[2]، سطح گلوکز خون و دیگر ویژگی‌ها است، با استفاده از روش‌های پاکسازی داده‌ها مانند جایگزینی مقادیر گمشده[3]، شناسایی و حذف داده‌های پرت و کدگذاری داده‌ها آماده‌سازی شده اند. سپس داده‌ها برای مراحل بعدی، شامل دسته‌بندی[4] و خوشه‌بندی[5]، تقسیم‌بندی و متعادل‌سازی شده اند.

---

[1] Data cleaning

[2] BMI

[3] Missing values

[4] Classification

[5] Clustering

برای دسته‌بندی، الگوریتم‌های مختلفی مانند درخت تصمیم[۱] و k-نزدیک‌ترین همسایه[۲] مورد استفاده قرار گرفته‌اند. همچنین، برای خوشه‌بندی، از الگوریتم‌های K-Means و سلسله‌مراتبی[۳] استفاده شده است.

۲-۱- معرفی ابزارها

در این پروژه، از ابزارها و کتابخانه‌های مختلف پایتون برای تحلیل داده‌ها و کمک گرفتن در پیاده‌سازی مدل‌های یادگیری ماشین استفاده شده است. کتابخانهٔ Pandas به منظور مدیریت و تحلیل داده‌ها مورد استفاده قرار گرفت و عملیات‌هایی نظیر خواندن فایل‌های CSV، شناسایی مقادیر گمشده و پردازش داده‌های ساختاریافته با آن انجام شد. همچنین، NumPy برای انجام محاسبات عددی و عملیات ماتریسی به کار رفت.

برای تجسم داده‌ها و نمایش گرافیکی نتایج، از دو کتابخانهٔ Matplotlib و Seaborn بهره گرفته شد. این ابزارها به نمایش هیستوگرام‌ها، نمودارهای پراکندگی و تحلیل ویژگی‌های داده کمک کردند. همچنین، از توابع آماری کتابخانهٔ SciPy برای محاسبهٔ چولگی داده‌ها و خوشه‌بندی سلسله‌مراتبی استفاده شد.

در بخش یادگیری ماشین، از ابزارهای موجود در Scikit-learn برای ارزیابی مدل‌ها، محاسبهٔ دقت[۴] و ایجاد ماتریس درهم‌ریختگی[۵] استفاده شد. برای پیاده‌سازی مدل‌های سفارشی مانند درخت تصمیم و روش نزدیک‌ترین همسایه‌ها، الگوریتم‌های دستی طراحی و پیاده‌سازی گردیدند.

همچنین، خوشه‌بندی داده‌ها با استفاده از الگوریتم‌های K-Means و سلسله‌مراتبی انجام شد که در این پروژه به‌صورت دستی پیاده‌سازی گردیدند. روش Elbow Method برای تعیین تعداد بهینه خوشه‌ها در الگوریتم K-Means به کار رفت. علاوه بر این، تجزیه و تحلیل ویژگی‌ها و مدیریت مقادیر گمشده نیز بخش مهمی از این پروژه بود که از تکنیک‌های مختلفی نظیر جای‌گذاری مقادیر تصادفی استفاده شد.

در نهایت، این ترکیب از ابزارهای مختلف به تحلیل جامع داده‌ها، نمایش مؤثر نتایج و توسعهٔ مدل‌های دقیق و قابل اعتماد کمک کرد و درک عمیق‌تری از مفاهیم پردازش داده و یادگیری ماشین فراهم آورد.

---

[1] Decision tree

[2] K-Nearest Neighbor (KNN)

[3] Hierarchical

[4] Accuracy

[5] Confusion matrix

۳-۱- بارگذاری و معرفی دیتاست

دیتاست در قالب فایل modified_diabetes_prediction_dataset.csv موجود بود که به‌کمک read_csv خوانده شد. این دیتاست حاوی ستون gender (جنسیت)، age (سن)، hypertension (فشار خون بالا)، heart_disease (سابقهٔ حملهٔ قلبی)، smoking_history (سابقهٔ سیگار کشیدن)، bmi، HbA1c_level (متوسط گلوکز خون)، blood_glucose_level (سطح گلوکز خون) و ستون هدف diabetes است که نشان می‌دهد یک فرد با ویژگی‌های مذکور دارای دیابت هست یا خیر. ستون‌های gender و smoking_history اسمی، ستون‌های hypertension، heart_disease و diabetes باینری و دیگر ستون‌ها، عددی اند. دیگر ویژگی‌ها در مورد این ستون‌ها در فصل دوم گفته خواهند شد.

فصل دوم

پاک‌سازی داده

۱-۲- مقدمه

از آن‌جایی که در دنیای واقعی، داده‌ها به‌ندرت تمیز و آماده برای تحلیل هستند، یکی از مهارت‌های کلیدی و مراحل مهم و ضروری در علم داده و یادگیری ماشین، پاک‌سازی داده است. این مرحله به‌معنای شناسایی و اصلاح مشکلات مختلفی است که ممکن است در داده‌ها وجود داشته باشد و مانع از تحلیل صحیح و مدل‌سازی درست داده‌ها شوند. این مشکلات شامل مقادیر گمشده، داده‌های تکراری، مقادیر خارج از محدودهٔ منطقی و... هستند که برای هر کدام، روش‌هایی برای برطرف‌سازی اشکال‌ها وجود دارند. پاک‌سازی داده‌ها می‌تواند از ایجاد نتایج

گمراه‌کننده و تصمیمات نادرست جلوگیری کند و دقت مدل‌های پیش‌بینی را بالا ببرد.

۲-۲- شناسایی و پاک‌سازی مقادیر گمشده، تکراری و خارج از محدوده

وجود مقادیر گمشده در داده‌ها یکی از چالش‌های رایج در تحلیل داده‌ها و یادگیری ماشین است که می‌تواند نتایج تحلیل و مدل‌سازی را تحت تأثیر قرار دهد. مقادیر گمشده می‌توانند به دلایل مختلفی رخ دهند؛ ازجمله خطای انسانی در فرآیند جمع‌آوری داده‌ها، نقص‌های فنی در ابزارهای اندازه‌گیری، محدودیت‌های دسترسی به اطلاعات یا عدم پاسخ‌دهی افراد در نظرسنجی‌ها.

برای مقابله با این مسئله، روش‌های متعددی برای مدیریت و برطرف‌سازی مقادیر گمشده ارائه شده است. ساده‌ترین روش، حذف رکوردها یا ویژگی‌های حاوی مقادیر گمشده است که البته ممکن است باعث کاهش حجم داده‌ها و از دست رفتن اطلاعات مفید شود. روش دیگر، جایگزینی مقادیر گمشده با مقادیری مانند میانگین، میانه یا مد ویژگی مربوطه است. در روش‌های پیشرفته‌تر، از مدل‌های پیش‌بینی برای تخمین مقادیر گمشده استفاده می‌شود، مانند رگرسیون خطی، درخت‌های تصمیم، یا الگوریتم‌های مبتنی بر یادگیری ماشین. انتخاب روش مناسب برای مدیریت مقادیر گمشده به عوامل متعددی نظیر نوع داده‌ها، میزان و الگوی گم‌شدگی و اهمیت حفظ ساختار داده‌ها بستگی دارد. بهینه‌سازی این فرآیند می‌تواند دقت تحلیل‌ها را بهبود بخشیده و از کاهش کیفیت نتایج جلوگیری کند.

```
print(f"The count of null cells in each column is:\n{df.isnull().sum()}")
print(f"Records with null values:\n{df[df.isnull().any(axis=1)]}")
```

شکل (۲-۱) شناسایی مقادیر گمشده

بعد از بارگذاری دیتاست، مطابق شکل (۲-۱)، تعداد سلول‌های گمشده در هر ستون و رکوردهایی که دارای حداقل یک سلول گمشده هستند درخواست شده است. نتایج در شکل (۲-۲) نشان داده شده اند. ظاهراً تعداد کمی مقادیر گمشده داریم!

```
The count of null cells in each column is:
gender                0
age                   2
hypertension          0
heart_disease         0
smoking_history       1
bmi                   0
HbA1c_level           1
blood_glucose_level   0
diabetes              0
dtype: int64
Records with null values:
        gender   age  hypertension  heart_disease smoking_history       bmi  \
1       Female   NaN             0              0         No Info  28.034572
2         Male  28.0             0              0            NaN  25.369152
3       Female  37.0             0              0            yes  25.262602
100000  Female   NaN             0              0         No Info  25.698752

        HbA1c_level  blood_glucose_level  diabetes
1               6.6                   80         0
2               5.7                 9999         0
3               NaN                  155         0
100000          6.6                   80         0
```

شکل (۲-۲) تعداد سلول‌های گمشده در هر ستون و رکوردهای حاوی سلول گمشده

برای این‌که بتوانیم متوجه تعارضات در داده‌ها شویم، مقادیر یکتای هر ستون در شکل (۲-۳) خواسته شده اند و در شکل (۲-۴) این مقادیر را مشاهده می‌کنیم. توجه کنید که با اجرای تابع sort_index، مقادیر یکتا برای ستون‌های عددی به‌صورت صعودی مرتب شده اند. مطابق این گزارش، ستون gender دارای چهار مقدار Male، Female، Other و unknown است. با توجه به تعداد بسیار پایین رکوردهای Other و unknown، این دو مقدار را گمشده فرض می‌کنیم و در مراحل بعد آن‌ها را با NaN جایگزین می‌کنیم. همچنین، در ستون age، مشاهده می‌کنیم که تعدادی سن منفی داریم که با توجه به مقادیری که گرفته اند، به اشتباه منفی وارد نشده اند و پرت هستند. این مقادیر را نیز باید با NaN جایگزین کنیم. ستون smoking_history دارای یک مقدار yes است که احتمالاً به اشتباه وارد شده و باید current قرار می‌گرفته است. ستون blood_glucose_level دارای یک مقدار ۹۹۹۹ است که به اشتباه وارد شده است. ستون bmi را جلوتر بیشتر مورد بررسی قرار می‌دهیم. بقیهٔ ستون‌ها از لحاظ محدودهٔ مقادیر یکتا مشکلی ندارند.

```python
print("Distinct Values for each column with count of each one are: ")
for column in df.columns:
    print(df[column].value_counts().sort_index())
```

شکل (۲-۳) شناسایی مقادیر یکتای هر ستون به‌صورت مرتب‌شده

```
Distinct Values for each column with count of each one are:
gender
Female     58552
Male       41430
Other         18
unknown        1
Name: count, dtype: int64
age
-4.92       1
-4.84       3
-4.76       9
-4.68      10
-4.60       6
          ...
 80.00    865
 81.00    732
 82.00    652
 83.00    614
 84.00    571
Name: count, Length: 222, dtype: int64
hypertension
0    92516
1     7485
Name: count, dtype: int64
heart_disease
0    96059
1     3942
Name: count, dtype: int64
```

شکل (۲-۴) مقادیر یکتای بعضی ستون‌ها

مطابق با توضیحات داده‌شده، جایگزینی‌ها در شکل (۲-۵) انجام شده اند.

```
df.replace({"gender": {"unknown": np.nan, "Other": np.nan}}, inplace=True)
df.replace({"smoking_history": {"yes": "current"}}, inplace=True)
df.replace({"blood_glucose_level": {9999: np.nan}}, inplace=True)
df["age"] = df["age"].where(df["age"] >= 0, np.nan)
```

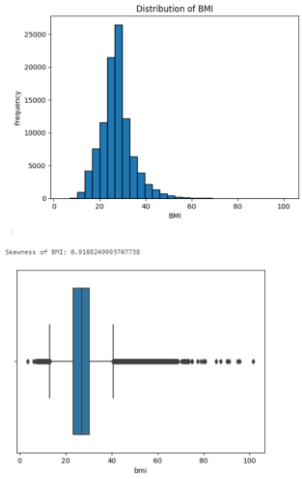
شکل (۲-۵) جایگزینی مقادیر نامعتبر

اما برای ستون bmi، از آن‌جایی که تصمیم‌گیری در خصوص معیار محدودهٔ صحیح، اندکی دشوار بود، توزیع آن را در قالب یک هیستوگرام مطابق شکل‌های (۲-۶) و (۲-۷) ترسیم کردیم و مشاهده کردیم این ستون دارای چولگی مثبت و راست است و بنابراین، استفاده از معیار IQR برای شناسایی داده‌های پرت، روش خوبی نیست. همچنین، روش z-score، روش مناسبی نبود، چرا که داده‌های پرت را فقط در یک سمت شناسایی می‌کرد. بنابراین، در حالت ساده، فرض کردیم bmi می‌توانید مقادیر معتبر ۱۰ تا ۶۰ را داشته باشد و رکوردهای دارای bmi خارج از این محدوده را مطابق شکل (۲-۸) حاوی مقدار نامعتبر فرض کردیم[۱].

[۱] تصمیم‌گیری در خصوص معتبر بودن bmi به‌خصوص برای افراد زیر ۱۹ سال، پیچیده است؛ چرا که در این بازهٔ سنی از معیارهای دیگری برای اندازه‌گیری شاخص تودهٔ بدنی استفاده می‌شود.

```
df["bmi"].plot(kind="hist", bins=30, edgecolor="black")
plt.xlabel("BMI")
plt.title("Distribution of BMI")
plt.show()
skewness = skew(df["bmi"])
print(f"Skewness of BMI: {skewness}")
sns.boxplot(x="bmi", data=df)
plt.show()
```

شکل (۲-۶) رسم هیستوگرام و نمودار جعبه‌ای و محاسبهٔ چولگی ستون bmi



شکل (۲-۷) نمودارهای نشان‌دهندهٔ توزیع ستون bmi

```
df["bmi"] = df["bmi"].where((df["bmi"] >= 10) & (df["bmi"] <= 60), np.nan)
```

شکل (۲-۸) جایگزینی مقادیر خارج از محدودهٔ ۱۰ تا ۶۰ ستون bmi با NaN

در این‌جا مطابق شکل (۲-۹) و شکل (۲-۱۰) مشاهده می‌کنیم که با اضافه‌شدن مقادیر خالی، دو رکورد تکراری داریم که یکی از آن‌ها را حذف می‌کنیم.

```
print("Duplicate records are: ")
duplicates = df[df.duplicated(keep=False)]
print(duplicates)
df = df.drop_duplicates()
```

شکل (۲-۹) شناسایی رکوردهای تکراری

```
Duplicate records are:
      gender  age  hypertension  heart_disease smoking_history  bmi  \
47023   Male  NaN             0              0         No Info  NaN
63979   Male  NaN             0              0         No Info  NaN

       HbA1c_level  blood_glucose_level  diabetes
47023          4.5                126.0         0
63979          4.5                126.0         0
```

شکل (۲-۱۰) رکوردهای تکراری

در ادامه با توجه به این‌که تعداد رکوردهای دارای سلول‌های خالی برای ستون‌های age و bmi حدوداً هزار رکورد هستند (در مقایسه با صد هزار رکورد)، بهتر است که از آن‌ها صرف نظر کنیم. اما با دید آموزشی بودن این پروژه، تصمیم به پر کردن آن‌ها به‌صورت نمونه‌گیری تصادفی (بدون ایجاد مشکلی در توزیع داده‌ها) گرفته شده است. بقیهٔ رکوردهای دارای سلول‌های خالی با توجه به انگشت‌شمار بودن، حذف شده اند و بعد از اجرای دستورات شکل‌های (۲-۱۱) و (۲-۱۲)، دیگر خانهٔ خالی و یا مقدار نامعتبری نداریم.

```python
for column in ["age", "bmi"]:
    value_counts = df[column].value_counts(normalize=True)
    df.loc[df[column].isna(), column] = np.random.choice(
        value_counts.index.tolist(), size=df[column].isna().sum(), p=value_counts.values
    )
```

شکل (۲-۱۱) پر کردن خانه‌های خالی ستون‌های age و bmi با مقادیر تصادفی

```python
df = df.dropna()
```

شکل (۲-۱۲) حذف رکوردهای دارای حداقل یک سلول خالی

فصل سوم

پیاده‌سازی الگوریتم‌های دسته‌بندی

۳-۱- مقدمه

در این قسمت به پیاده‌سازی الگوریتم‌های دسته‌بندی درخت تصمیم و k–نزدیک‌ترین همسایه می‌پردازیم. در ادامه، ابتدا داده‌ها با ساختار درختی تحلیل خواهند شد و پس از محاسبهٔ معیارهای مختلف برای داده‌های آموزشی و تست، به بررسی الگوریتم k–نزدیک‌ترین همسایه و پیش‌بینی داده‌های تست براساس آن می‌پردازیم.

۳-۲- پیاده‌سازی دسته‌بندی با الگوریتم درخت تصمیم

قبل از توضیح الگوریتم درخت تصمیم، توضیحاتی در خصوص آماده‌سازی‌های لازم قبل از استفاده از این الگوریتم داده می‌شود. با توجه به حذف تعدادی از رکوردها، لازم است تا اندیس‌گذاری مجدداً انجام شود تا در ادامه با مشکل مواجه نشویم. با توجه به این‌که توزیع داده‌ها با در نظر گرفتن ستون هدف، نامتوازن است و تعداد صفرها بسیار بیشتر از یک‌هاست، دیتاست آموزش را حاصل ترکیب ۷۰٪ رکوردهایی که مقدار ستون هدفشان، یک

است به‌علاوهٔ همان میزان رکورد که مقدار ستون هدفشان، صفر است (به‌صورت تصادفی) در نظر می‌گیریم. مابقی رکوردها را در دیتاست تست قرار می‌دهیم. مجدداً برای دیتاست آموزش و تست، اندیس‌گذاری را انجام می‌دهیم. ویژگی‌هایی که باید درخت تصمیم براساس آن‌ها تشکیل شود (تمام ویژگی‌ها به‌جز ویژگی هدف) را استخراج می‌کنیم. برای این‌که درخت تصمیم بیش از حد بزرگ نشود، ستون age را در دیتاست‌های آموزش و تست به یک ستون دسته‌ای تبدیل می‌کنیم، طوری که با توجه به توزیع، افراد زیر ۲۴ سال، جوان، افراد بین ۲۴ تا ۵۴ سال، میان‌سال و دیگر افراد، پیر محسوب شوند. این کار را برای ستون‌های bmi، HbA1c_level و blood_glucose_level نیز متناسب با سطح‌بندی علمی انجام می‌دهیم تا به کم، نرمال و زیاد برسیم. برای دیتاست‌های آموزش و تست، مقادیر صفر و یک ستون diabetes را جهت معنادارتر شدن برگ‌های درختمان به Yes و No نگاشت می‌کنیم. توضیحات داده‌شده در شکل (۳-۱) پیاده‌سازی شده اند.

```
df = df.reset_index(drop=True)
df_zero = df[df["diabetes"] == 0]
df_one = df[df["diabetes"] == 1]
sample_size = int(len(df_one) * 0.7)
zero_sample = df_zero.sample(n=sample_size, random_state=42)
one_sample = df_one.sample(n=sample_size, random_state=42)
balanced_train = pd.concat([zero_sample, one_sample])
balanced_train = balanced_train.sample(frac=1, random_state=42).reset_index(drop=True)
test_data = df.drop(balanced_train.index).reset_index(drop=True)
train_data = balanced_train
attribute = train_data.columns.to_list()[:-1]
for col in attribute:
    if col == "age":
        bins = [0, 24, 54, float("inf")]
        labels = ["Youth", "Middle", "Old"]
        train_data[col] = pd.cut(
            train_data[col], bins=bins, labels=labels, include_lowest=True
        )
        test_data[col] = pd.cut(
            test_data[col], bins=bins, labels=labels, include_lowest=True
        )
    elif col in ["bmi", "HbA1c_level", "blood_glucose_level"]:
        if col == "bmi":
            bins = [0, 18.5, 24.9, float("inf")]
        elif col == "HbA1c_level":
            bins = [0, 5.7, 6.5, float("inf")]
        elif col == "blood_glucose_level":
            bins = [0, 140, 200, float("inf")]
        train_data[col] = pd.cut(
            train_data[col], bins=bins, labels=["Low", "Normal", "High"]
        )
        test_data[col] = pd.cut(
            test_data[col], bins=bins, labels=["Low", "Normal", "High"]
        )
    elif col in ["gender", "hypertension", "heart_disease"]:
        pass
train_data["diabetes"] = train_data["diabetes"].map({0: "No", 1: "Yes"})
test_data["diabetes"] = test_data["diabetes"].map({0: "No", 1: "Yes"})
X_train = train_data.iloc[:, :].values
X_test = test_data.iloc[:, :].values
```

شکل (۳-۱) متعادل‌سازی و آماده‌سازی دیتاست‌های آموزش و تست

حال، نوبت به پیاده‌سازی الگوریتم درخت تصمیم می‌رسد. این پیاده‌سازی، نسخهٔ بهبودیافتهٔ یک مخزن در گیت‌هاب[1] است که امکاناتی نظیر بصری‌سازی درخت به آن افزوده شده است. در ابتدا کلاس گره را تعریف کردیم که حاوی مقدار گره (نام ویژگی)، تصمیم یا مقدار مرتبط با این گره (شرطی که از ویژگی بررسی‌شده در گره گرفته می‌شود) و لیستی از گره‌های فرزند است که این گره، والد آن‌هاست. تابع findEntropy، میزان بی‌نظمی یا اطلاعات در داده را اندازه‌گیری می‌کند. این تابع، داده‌ها را به‌صورت لیستی از لیست‌ها و لیستی از سطرهای بررسی‌شونده را به‌عنوان ورودی دریافت می‌کند و آنتروپی محاسبه‌شده و ans که نتیجهٔ قطعی را نشان می‌دهد را به‌عنوان خروجی برمی‌گرداند. این تابع با استفاده از ستون آخر جدول، بررسی می‌کند که مقدار ستون هدف هر ردیف چیست. احتمال Yes و No بودن محاسبه می‌شود و در حالتی که این احتمال، قطعی است، جهت تصمیم‌گیری برای تبدیل به برگ شدن آن گره، ans مقدار صفر یا یک می‌گیرد. نحوهٔ محاسبهٔ آنتروپی مشابه محاسبهٔ information gain در ID3 است. به شکل‌های (۳-۲) و (۳-۳) توجه کنید.

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

شکل (۳-۲) فرمول محاسبهٔ اطلاعات خواسته‌شدهٔ مورد انتظار برای دسته‌بندی رکورد D

[1] github.com/vidhikhatwani/Decision-Tree-ID3-Algorithm

```
class Node(object):
    def __init__(self):
        self.value = None
        self.decision = None
        self.childs = []


def findEntropy(data, rows):
    yes, no, ans = 0, 0, -1
    idx = len(data[0]) - 1
    for i in rows:
        if data[i][idx] == "Yes":
            yes += 1
        else:
            no += 1
    if yes + no != 0:
        x, y = yes / (yes + no), no / (yes + no)
    else:
        x, y = 0, 0
    if x == 1:
        ans = 1
    elif y == 1:
        ans = 0
    entropy = -1 * (x * math.log2(x) + y * math.log2(y)) if x != 0 and y != 0 else 0
    return entropy, ans
```

شکل (۳-۳) کلاس گره و تابع پیدا کردن آنتروپی

تابع findMaxGain وظیفهٔ انتخاب بهترین ویژگی برای تقسیم داده‌ها در یک گره از درخت تصمیم را دارد. این تابع، ورودی دیتاست شامل ویژگی‌ها و مقادیر برچسب‌ها، لیستی از ایندکس ردیف‌هایی که باید بررسی شوند و لیستی از اندیس ستون‌های ویژگی‌هایی که باید برای تقسیم بررسی شوند دارد و به‌عنوان خروجی، بیشترین gain، اندیس ستونی که بهترین ویژگی برای تقسیم است و نتیجهٔ قطعی را برمی‌گرداند. برای محاسبهٔ آنتروپی از findEntropy استفاده می‌شود. در صورت صفر بودن آنتروپی، برچسب تمامی داده‌ها یکسان است و در غیر این‌صورت، تابع برای هر ویژگی، gain را محاسبه می‌کند. با استفاده از یک دیکشنری، مقادیر یکتا در ستون جاری و تعداد وقوع هر مقدار محاسبه می‌شود و برای هر مقدار یکتا در ویژگی جاری، مقادیر Yes و No شمارش می‌شوند. احتمال Yes و No محاسبه شده و آنتروپی این مقدار یکتا به gain اضافه می‌شود. اگر مقدار gain بیشتر از مقدار فعلی maxGain باشد، مقدار maxGain و اندیس ستون retidx به‌روزرسانی می‌شود. در پایان، تابع مقدار بیشترین gain، اندیس بهترین ویژگی و نتیجهٔ قطعی را برمی‌گرداند. به شکل (۳-۴) توجه کنید.

```
def findMaxGain(data, rows, columns):
    maxGain, retidx = 0, -1
    entropy, ans = findEntropy(data, rows)
    if entropy == 0:
        return maxGain, retidx, ans
    for j in columns:
        mydict = {}
        for i in rows:
            key = data[i][j]
            mydict[key] = mydict.get(key, 0) + 1
        gain = entropy
        for key in mydict:
            yes = no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == "Yes":
                        yes += 1
                    else:
                        no += 1
            if yes + no != 0:
                x, y = yes / (yes + no), no / (yes + no)
            else:
                x, y = 0, 0
            if x != 0 and y != 0:
                gain += (mydict[key] * (x * math.log2(x) + y * math.log2(y))) / len(
                    rows
                )
        if gain > maxGain:
            maxGain = gain
            retidx = j
    return maxGain, retidx, ans
```

شکل (۳-۴) تابع findMaxGain

تابع buildTree مسئول ساخت یک درخت تصمیم است. این تابع، findMaxGain را فراخوانی می‌کند، یک گره ایجاد می‌کند و در صورت صفر شدن gain، مقدار نهایی گره را مشخص می‌کند. مقدار گره برابر با نام ویژگی انتخاب‌شده تنظیم می‌شود. یک دیکشنری ساخته می‌شود تا مقادیر یکتای ویژگی و تعداد آن‌ها نگهداری شود. یک کپی از ستون‌ها گرفته شده و ویژگی انتخاب‌شده حذف می‌شود. برای هر مقدار یکتای ویژگی، داده‌ها فیلتر شده و تابع buildTree به صورت بازگشتی فراخوانی می‌شود. در نهایت، گره ساخته‌شده برگردانده می‌شود.

تابع predict برای پیش‌بینی مقدار برچسب دادهٔ ورودی براساس درخت تصمیم ساخته‌شده استفاده می‌شود. این تابع، ریشهٔ درخت تصمیم و یک نمونه داده که شامل مقادیر ویژگی‌هاست را به‌عنوان ورودی می‌گیرد. اگر گره جاری هیچ فرزندی نداشته باشد، مقدار برچسب برگردانده می‌شود. مقدار ویژگی مربوط به گره فعلی در دادهٔ ورودی بررسی می‌شود. اگر مقدار ویژگی با تصمیم یکی از فرزندان مطابقت داشته باشد، پیش‌بینی به صورت

بازگشتی در آن فرزند ادامه می‌یابد. اگر هیچ فرزندی مطابقت نداشت، مقدار پیش‌فرض "No" برگردانده می‌شود.

تابع evaluate_tree برای ارزیابی عملکرد درخت تصمیم ساخته‌شده بر روی مجموعه داده استفاده می‌شود. این تابع، ریشهٔ درخت تصمیم و دیتاست را به‌عنوان ورودی می‌گیرد و براساس برچسب‌های واقعی و پیش‌بینی‌شده، معیارهای دقت، دقت مثبت، حساسیت، F1-score و ماتریس درهم‌ریختگی را محاسبه می‌کند.

پیاده‌سازی این سه تابع در شکل (۳-۵) قابل مشاهده است.

سپس، اندیس‌های مربوط به داده‌های آموزشی و تست ایجاد می‌شوند. اندیس تمام ویژگی‌ها ذخیره می‌شود. درخت تصمیم بر اساس داده‌های آموزشی ساخته می‌شود و عملکرد درخت تصمیم روی داده‌های آموزشی و تست ارزیابی می‌شود. نتایج تست برای داده‌های آموزشی و تست چاپ می‌شوند.

تابع collect_tree_data، ریشهٔ درخت تصمیم، موقعیت گره‌ها، مختصات فعلی گره، سطح فعلی درخت، گره والد، لیست یال‌ها و مقادیر تصمیم‌گیری گره‌ها را به‌عنوان ورودی دریافت می‌کند. موقعیت گره فعلی به همراه مقدار آن در pos ذخیره می‌شود. اگر گره والد موجود باشد،یال بین والد و گره فعلی ثبت می‌شود. برای هر فرزند، مختصات گره فرزند محاسبه شده و تابع به صورت بازگشتی فراخوانی می‌شود. موقعیت‌ها، یال‌ها و مقادیر تصمیم‌گیری بازگردانده می‌شوند.

در تابع draw_tree، هر یال بین گره‌ها رسم می‌شود و مقدار تصمیم‌گیری آن در وسط یال نمایش داده می‌شود. هر گره به صورت دایره‌ای رسم شده و مقدار آن داخل دایره نمایش داده می‌شود. در نهایت، نمودار رسم می‌شود و این توابع، فراخوانی می‌شوند. پیاده‌سازی این قسمت در شکل (۳-۶) و نتایج حاصل از این الگوریتم در شکل (۳-۷) قابل مشاهده اند.

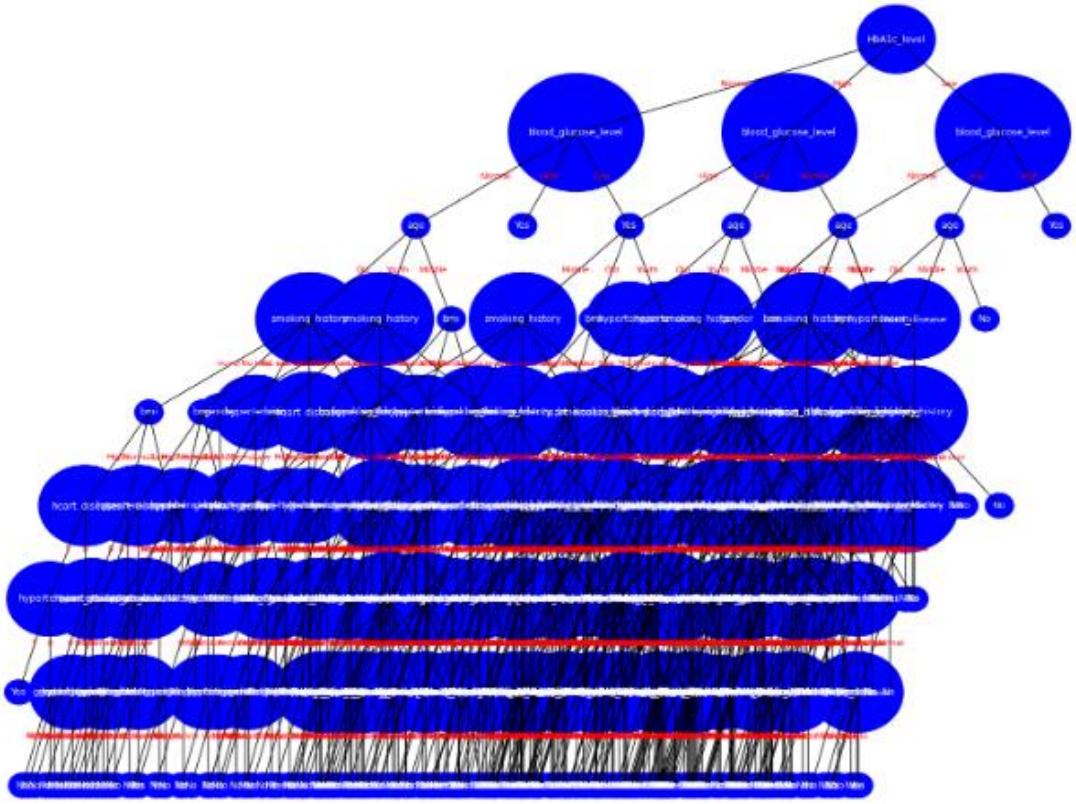
```python
def buildTree(data, rows, columns):
    maxGain, idx, ans = findMaxGain(data, rows, columns)
    root = Node()
    root.childs = []
    if maxGain == 0:
        root.value = "Yes" if ans == 1 else "No"
        return root
    root.value = attribute[idx]
    mydict = {}
    for i in rows:
        key = data[i][idx]
        mydict[key] = mydict.get(key, 0) + 1
    newcolumns = copy.deepcopy(columns)
    newcolumns.remove(idx)
    for key in mydict:
        newrows = [i for i in rows if data[i][idx] == key]
        temp = buildTree(data, newrows, newcolumns)
        temp.decision = key
        root.childs.append(temp)
    return root


def predict(root, data_row):
    if not root.childs:
        return root.value
    for child in root.childs:
        if data_row[attribute.index(root.value)] == child.decision:
            return predict(child, data_row)
    return "No"


def evaluate_tree(root, X):
    y_true = [row[-1] for row in X]
    y_pred = [predict(root, row) for row in X]
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, pos_label="Yes")
    rec = recall_score(y_true, y_pred, pos_label="Yes")
    f1 = f1_score(y_true, y_pred, pos_label="Yes")
    cm = confusion_matrix(y_true, y_pred, labels=["Yes", "No"])
    return acc, prec, rec, f1, cm
```

شکل (۳-۵) توابع buildTree، predict و evalute_tree

```
rows_train = [i for i in range(len(X_train))]
rows_test = [i for i in range(len(X_test))]
columns = [i for i in range(len(attribute))]
root = buildTree(X_train, rows_train, columns)
acc_train, prec_train, rec_train, f1_train, cm_train = evaluate_tree(root, X_train)
acc_test, prec_test, rec_test, f1_test, cm_test = evaluate_tree(root, X_test)
print("Train Metrics:")
print(
    f"Accuracy: {acc_train:.2f}, Precision: {prec_train:.2f}, Recall: {rec_train:.2f}, F1-Score: {f1_train:.2f}"
)
print("Confusion Matrix (Train):\n", cm_train)
print("\nTest Metrics:")
print(
    f"Accuracy: {acc_test:.2f}, Precision: {prec_test:.2f}, Recall: {rec_test:.2f}, F1-Score: {f1_test:.2f}"
)
print("Confusion Matrix (Test):\n", cm_test)


def collect_tree_data(
    root, pos={}, x=0, y=0, level=1, parent=None, edges=[], labels=[]
):
    if root is None:
        return pos, edges, labels
    pos[(x, y)] = f"{root.value if root.value else root.decision}"
    if parent is not None:
        edges.append((parent, (x, y)))
        labels.append(root.decision)
    n = len(root.childs)
    for i, child in enumerate(root.childs):
        child_x = x + (i - n / 2) * (3 / level)
        child_y = y - 1.5
        pos, edges, labels = collect_tree_data(
            child, pos, child_x, child_y, level + 1, (x, y), edges, labels
        )
    return pos, edges, labels


def draw_tree(pos, edges, labels):
    fig, ax = plt.subplots(figsize=(20, 15))
    for idx, ((x1, y1), (x2, y2)) in enumerate(edges):
        ax.plot([x1, x2], [y1, y2], "k-", lw=1)
        label_x, label_y = (x1 + x2) / 2, (y1 + y2) / 2
        ax.text(label_x, label_y, labels[idx], color="red", fontsize=8, ha="center")
    for (x, y), label in pos.items():
        circle_radius = max(0.2, 0.05 * len(label))
        ax.add_patch(plt.Circle((x, y), circle_radius, color="blue"))
        ax.text(x, y, label, color="white", ha="center", va="center", fontsize=9)
    ax.axis("off")
    plt.show()


pos, edges, labels = collect_tree_data(root)
draw_tree(pos, edges, labels)
```

شکل (۳-۶) ساخت اندیس‌ها، ساخت و ارزیابی درخت تصمیم و توابع collect_tree_data و draw_tree

```
Decision-Tree classification results:
Train Metrics:
Accuracy: 0.76, Precision: 1.00, Recall: 0.52, F1-Score: 0.68
Confusion Matrix (Train):
 [[3099 2851]
 [   0 5950]]

Test Metrics:
Accuracy: 0.94, Precision: 0.74, Recall: 0.52, F1-Score: 0.61
Confusion Matrix (Test):
 [[ 3869  3617]
 [ 1390 79203]]
```



شکل (۳-۷) نتایج الگوریتم درخت تصمیم

همان‌طور که مشاهده می‌شود توانستیم معیارهای گزارش‌شده را که قابل قبول هستند کسب کنیم. از آن‌جایی که شناسایی دیابتی‌ها در اولویت است، دقت مثبت ۱۰۰درصد برای داده‌های آموزشی و دقت مثبت ۷۴درصد برای داده‌های تست عملکرد خیلی خوبی است. در صدر درخت تصمیم، HbA1c_level قرار گرفته است و براساس Normal، High و یا Low بودن آن به سطح بعدی می‌رویم. برای مثال، این مدل می‌گوید که کسی که HbA1c_level پایین و blood_glucose_level بالا دارد دارای دیابت است.

۳-۳- پیاده‌سازی دسته‌بندی الگوریتم k-نزدیک‌ترین همسایه

همان‌طور که می‌دانیم، این الگوریتم براساس k تا همسایهٔ نزدیک خود، برچسب را پیش‌بینی می‌کند. مشابه الگوریتم قبل، ابتدا آماده‌سازی داده‌ها صورت گرفته است. از آن‌جایی که این الگوریتم نیاز به مقایسه‌های زیاد و زمان‌بر دارد و بسیار کند است، در ابتدا از داده‌ها نمونه گرفته شده است. طبیعی است که این نمونه‌گیری ممکن است معیارهای ارزیابی این مدل را تحت‌الشعاع قرار دهد. مطابق شکل (۳-۸)، تابع preprocess_dataset تعریف شده است تا ستون‌ها را جهت ایجاد امکان مقایسه، عددی و نرمال‌سازی کند. در این‌جا، ستون smoking_history یک چالش بود و من راهی به‌جز نگاشت به‌صورتی که مشاهده می‌شود به ذهنم نرسید. ستون‌های دیگر براساس نرمال‌سازی کمینه-بیشینه[1] نرمال شده اند.

```
def preprocess_dataset(data):
    data = data.copy()
    data["gender"] = data["gender"].map({"Male": 1, "Female": 0})
    smoking_map = {
        "No Info": 0,
        "never": 0.2,
        "not current": 0.4,
        "former": 0.6,
        "ever": 0.8,
        "current": 1,
    }
    data["smoking_history"] = data["smoking_history"].map(smoking_map)
    numeric_columns = ["age", "bmi", "HbA1c_level", "blood_glucose_level"]
    for col in numeric_columns:
        data[col] = (data[col] - data[col].min()) / (data[col].max() - data[col].min())
    return data
```

شکل (۳-۸) تابع preprocess_dataset

در ادامه به توابع مرتبط با این الگوریتم می‌پردازیم.

تابع euclidean_distance، فاصلهٔ اقلیدسی دو بردار ورودی را محاسبه می‌کند که همان ریشهٔ دوم مجموع مربعات اختلاف‌هاست.

تابع predict_knn، فاصلهٔ اقلیدسی نمونهٔ row با هر نمونه در داده‌های آموزشی را محاسبه می‌کند؛ سپس، لیست distances که شامل فاصله و کلاس مربوطه است، بر اساس فاصله مرتب می‌شود. k نمونه نزدیک‌تر انتخاب می‌شوند. کلاس هر یک از نزدیک‌ترین همسایگان استخراج می‌شود. کلاسی که بیشترین تکرار را در بین همسایگان

[1] Min-Max Normalization

دارد، به‌عنوان پیش‌بینی نهایی انتخاب می‌شود.

تابع evaluate_knn برای هر نمونه از داده‌های تست، کلاس را پیش‌بینی می‌کند و معیارهای ارزیابی را محاسبه می‌کند.

در نهایت، تعداد همسایگان تنظیم می‌شود و مدل، ارزیابی می‌شود. تعداد همسایگان براساس چندین بار آزمون و خطا طوری قرار گرفته است تا بهترین دقت را به ما بدهد.

پیاده‌سازی این الگوریتم در شکل (۳-۹) و نتایج آن در شکل (۳-۱۰) نشان داده شده اند.

```python
def euclidean_distance(row1, row2):
    return np.sqrt(np.sum((row1 - row2) ** 2))


def predict_knn(X_train, y_train, row, k):
    distances = []
    for i, train_row in enumerate(X_train):
        dist = euclidean_distance(train_row, row)
        distances.append((dist, y_train[i]))
    distances.sort(key=lambda x: x[0])
    k_nearest = distances[:k]
    classes = [neighbor[1] for neighbor in k_nearest]
    prediction = max(set(classes), key=classes.count)
    return prediction


def evaluate_knn(X_train, y_train, X_test, y_test, k):
    y_pred = [predict_knn(X_train, y_train, row, k) for row in X_test]
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, pos_label=1)
    rec = recall_score(y_test, y_pred, pos_label=1)
    f1 = f1_score(y_test, y_pred, pos_label=1)
    cm = confusion_matrix(y_test, y_pred, labels=[1, 0])
    return acc, prec, rec, f1, cm


k = 2
acc_test, prec_test, rec_test, f1_test, cm_test = evaluate_knn(
    X_train, y_train, X_test, y_test, k
)
print("\nTest Metrics:")
print(
    f"Accuracy: {acc_test:.2f}, Precision: {prec_test:.2f}, Recall: {rec_test:.2f}, F1-Score: {f1_test:.2f}"
)
print("Confusion Matrix (Test):\n", cm_test)
```

شکل (۳-۹) پیاده‌سازی و ارزیابی الگوریتم k-نزدیک‌ترین همسایه

```
KNN classification results:

Test Metrics:
Accuracy: 0.91, Precision: 0.47, Recall: 0.80, F1-Score: 0.60
Confusion Matrix (Test):
 [[ 293   71]
 [ 327 3721]]
```

شکل (۳-۱۰) نتایج الگوریتم k-نزدیک‌ترین همسایه

مشاهده می‌شود که در این پیاده‌سازی با وجود بالا بودن دقت، دقت مثبت خوبی نداریم. یکی از دلایل آن می‌تواند نمونه‌گیری اولیه از داده‌ها باشد که به‌دلیل زمان‌بر بودن بیش از حد این الگوریتم روی دیتاست صورت گرفته است.

فصل چهارم

پیاده‌سازی الگوریتم‌های خوشه‌بندی

۴-۱- مقدمه

در این قسمت به پیاده‌سازی الگوریتم‌های خوشه‌بندی K-Means و سلسله‌مراتبی می‌پردازیم. در ابتدا داده‌ها را با استفاده از الگوریتم K-Means و استفاده از Elbow Method خوشه‌بندی کردیم و سپس، خوشه‌بندی سلسله‌مراتبی را انجام دادیم.

۴-۲- پیاده‌سازی خوشه‌بندی با الگوریتم K-Means

همان‌طور که می‌دانیم، الگوریتم K-Means با دریافت تعداد خوشه‌ها، مراکز اولیه را به‌صورت تصادفی انتخاب می‌کند و داده‌ها را طوری در هر خوشه قرار می‌دهد که دادهٔ مورد نظر با مرکز آن خوشه کمترین فاصله را داشته باشد. بعد از خوشه‌بندی، میانگین هر خوشه با مرکز آن جابه‌جا می‌شود و این کار تا چند مرحله جهت بهبود مجموع مربعات فاصلهٔ نقاط تا مراکز خوشه‌هایشان ادامه پیدا می‌کند.

در ابتدا به پیاده‌سازی کلاس K-Means پرداختیم. این کلاس یک سازنده دارد که ویژگی‌های تعداد خوشه‌ها

(مراکز)، تعداد تکرارهای بیشینه برای الگوریتم و مقدار تحمل برای توقف الگوریتم[۱]. متد fit (یادگیری) مراکز اولیه را به‌صورت تصادفی انتخاب می‌کند، داده‌ها به نزدیک‌ترین خوشه اختصاص داده می‌شوند و مراکز خوشه‌ها به‌روزرسانی می‌شوند. اگر مراکز جدید به مراکز قبلی نزدیک‌تر از مقدار تحمل باشد، الگوریتم متوقف می‌شود. متد _assign_clusters با محاسبهٔ فاصلهٔ اقلیدسی هر نمونه از داده‌ها تا تمامی مراکز خوشه، داده‌ها را به خوشه‌ها براساس این‌که آن نمونه تا کدام خوشه کمترین فاصله را دارد اختصاص می‌دهد. متد _update_centroids مرکز هر خوشه را با محاسبهٔ میانگین داده‌هایی که به آن خوشه تعلق دارند به‌روزرسانی می‌کند. مراکز جدید خوشه‌ها به‌صورت آرایه‌ای برگردانده می‌شوند. متد _is_coverged فاصلهٔ بین مراکز قدیمی و مراکز جدید را برای هر خوشه محاسبه می‌کند؛ اگر تمام این فاصله‌ها از مقدار تحمل کمتر باشند، الگوریتم همگرا شده است و متوقف می‌شود. متد predict مشابه متد _assign_clusters فاصلهٔ هر نمونه از داده‌های جدید با مراکز خوشه‌ها محاسبه می‌شود و خوشه‌ای که کمترین فاصله را دارد به هر نمونه اختصاص داده می‌شود تا خوشه برای داده‌های جدید پیش‌بینی شود. پیاده‌سازی این کلاس در شکل (۴-۱) قابل مشاهده است.

```python
class KMeans:
    def __init__(self, n_clusters, max_iter=300, tol=1e-4):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol

    def fit(self, X):
        np.random.seed(42)
        random_indices = np.random.choice(X.shape[0], self.n_clusters, replace=False)
        self.centroids = X[random_indices]
        for _ in range(self.max_iter):
            self.labels = self._assign_clusters(X)
            previous_centroids = self.centroids.copy()
            self.centroids = self._update_centroids(X)
            if self._is_converged(previous_centroids, self.centroids):
                break

    def _assign_clusters(self, X):
        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
        return np.argmin(distances, axis=1)

    def _update_centroids(self, X):
        return np.array(
            [X[self.labels == k].mean(axis=0) for k in range(self.n_clusters)]
        )

    def _is_converged(self, old_centroids, new_centroids):
        return np.all(np.linalg.norm(old_centroids - new_centroids, axis=1) < self.tol)

    def predict(self, X):
        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
        return np.argmin(distances, axis=1)
```

شکل (۴-۱) کلاس KMeans

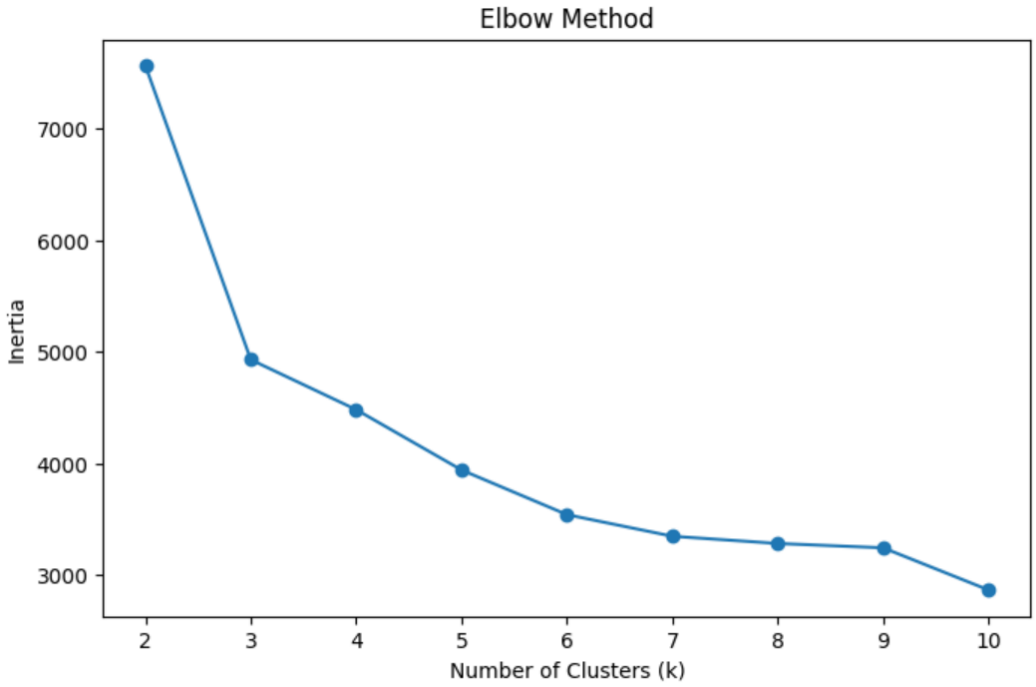
[۱] اگر تغییر مراکز خوشه‌ها کمتر از این مقدار باشد، الگوریتم متوقف می‌شود.

در ادامه، پیش‌پردازش داده‌ها مشابه الگوریتم درخت تصمیم و عددی کردن آن‌ها براساس preprocess_dataset که در فصل گذشته پیاده‌سازی شد انجام شده است. ستون diabetes به‌دلیل اینکه می‌خواهیم یادگیری بدون نظارت داشته باشیم حذف شده است. قبل از اجرای الگوریتم K-Means از معیار Elbow Method برای یافتن بهترین K را مورد بررسی قرار می‌دهیم.

اما Elbow Method چیست؟ روشی برای پیدا کردن تعداد مناسب خوشه‌ها (k) در الگوریتم K-Means است. این روش به تغییرات مقدار Inertia با افزایش k توجه می‌کند. Inertia مجموع مربعات فاصلهٔ داده‌ها از مراکز خوشه‌هاست. مقادیر کمتر نشان‌دهندهٔ خوشه‌بندی بهتر است. هدف، پیدا کردن نقطه‌ای است که Inertia به آرامی کاهش می‌یابد. برای پیاده‌سازی، inertia را یک لیست در نظر گرفتیم و محدودهٔ تعداد خوشه‌ها را از ۲ تا ۱۰ فرض کردیم. برای هر مقدار k، یک مدل K-Means با k خوشه ساخته می‌شود و مدل روی داده‌ها (فرضاً در متغیر data.values) اجرا می‌شود. برای هر خوشه، داده‌های اختصاص داده‌شده به خوشه استخراج می‌شوند و فاصلهٔ اقلیدسی هر داده از مرکز خوشه محاسبه شده و مربع این فاصله‌ها جمع زده می‌شود؛ مقدار محاسبه‌شده برای خوشه به متغیر inertia_k افزوده می‌شود و در نهایت، مقدار Inertia برای k به لیست inertia اضافه می‌شود. پس از آن، نمودار ترسیم می‌شود. به شکل‌های (۴-۲) و (۴-۳) توجه کنید.

```
inertia = []
K_range = range(2, 11)
for k in K_range:
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(data.values)
    inertia_k = 0
    for cluster, centroid in enumerate(kmeans.centroids):
        cluster_points = data.values[kmeans.labels == cluster]
        inertia_k += np.sum(np.linalg.norm(cluster_points - centroid, axis=1)**2)
    inertia.append(inertia_k)

plt.figure(figsize=(8, 5))
plt.plot(K_range, inertia, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Inertia')
plt.show()
```

شکل (۴-۲) یافتن بهترین k برای الگوریتم با استفاده از Elbow Method



شکل (۴-۳) نمودار Elbow Method

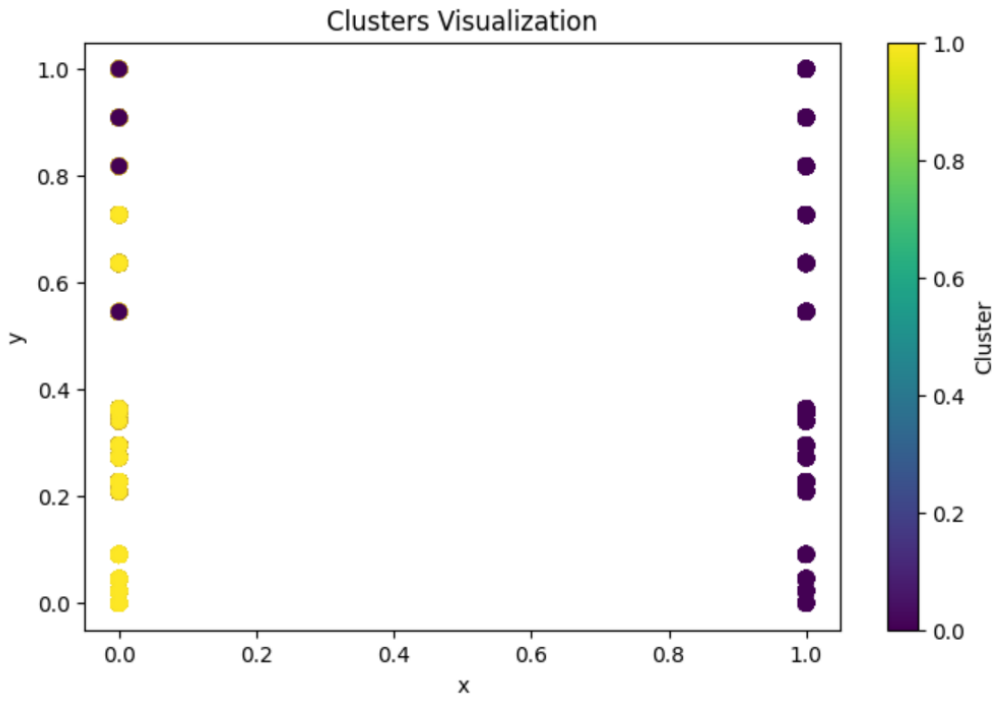
با وجود استفاده از این نمودار، به نظر می‌آید استفاده از دو خوشه گزینهٔ منطقی‌تری باشد!

در ادامه، مدل K-Means با دو خوشه تعریف شده، آموزش مدل روی داده‌ها انجام شده و پیش‌بینی صورت گرفته است. ستون جدیدی با نام Cluster ظاهر می‌شود که شمارهٔ خوشه هر داده را نگه می‌دارد. جهت تحلیل خوشه‌بندی، نمودار پراکندگی برای ویژگی‌های hypertension و blood_glucose_level ترسیم شده و داده‌ها بر اساس مقدار ستون Cluster رنگ‌آمیزی می‌شوند. جهت تحلیل هر خوشه، داده‌ها بر اساس مقدار خوشه و مقدار یک ویژگی دیگر (heart_disease و diabetes) گروه‌بندی می‌شوند. تعداد داده‌ها در هر گروه محاسبه شده و داده‌ها به صورت جدول دو بعدی نمایش داده می‌شوند، طوری که سطرها نشان‌دهندهٔ خوشه‌ها و ستون‌ها نشان‌دهندهٔ ویژگی‌ها هستند. به شکل‌های (۴-۴) و (۴-۵) توجه کنید.

```
kmeans_manual = KMeans(n_clusters=2)
kmeans_manual.fit(data.values)
sampled_df['Cluster'] = kmeans_manual.predict(data.values)
plt.figure(figsize=(8, 5))
plt.scatter(data['hypertension'], data['blood_glucose_level'], c=sampled_df['Cluster'], cmap='viridis', s=50)
plt.title('Clusters Visualization')
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(label='Cluster')
plt.show()
cluster_summary = sampled_df.groupby(['Cluster', 'heart_disease']).size().unstack(fill_value=0)
print(sampled_df.groupby(['Cluster', 'heart_disease']).size().unstack(fill_value=0))
print(sampled_df.groupby(['Cluster', 'diabetes']).size().unstack(fill_value=0))
```

شکل (۴-۴) اجرای الگوریتم K-Means و تهیهٔ گزارش توزیع داده‌ها در هر خوشه

K-means Clustering results:



heart_disease	0	1
Cluster		
0	1519	976
1	9367	38

diabetes	0	1
Cluster		
0	448	2047
1	5502	3903

شکل (۴-۵) نتایج الگوریتم K-Means و توزیع هر خوشه

مطابق شکل (۴-۵) مشاهده می‌شود که خوشه‌بندی به‌شدت تحت تأثیر hypertension و heart_disease بوده است.

۴-۳- پیاده‌سازی خوشه‌بندی با الگوریتم سلسله‌مراتبی

الگوریتم خوشه‌بندی سلسله‌مراتبی یک روش محبوب برای گروه‌بندی داده‌ها است که به صورت درختی یا دندروگرام نتایج را نمایش می‌دهد. این الگوریتم به دو نوع تجمیعی و تجزیه‌ای تقسیم می‌شود. در روش تجمیعی، هر داده ابتدا یک خوشه جداگانه در نظر گرفته شده و خوشه‌ها به تدریج با یکدیگر ادغام می‌شوند تا زمانی که تمام داده‌ها در یک خوشه بزرگ قرار گیرند. در مقابل، در روش تجزیه‌ای ابتدا تمام داده‌ها یک خوشه بزرگ تشکیل می‌دهند و به تدریج به خوشه‌های کوچک‌تر تجزیه می‌شوند. یکی از مهم‌ترین اجزای این الگوریتم، نحوه اندازه‌گیری فاصله بین خوشه‌ها است. معیارهای متداول شامل فاصلهٔ کوتاه‌ترین جفت نقاط بین دو خوشه، فاصلهٔ بلندترین جفت نقاط بین دو خوشه، فاصلهٔ میانگین نقاط بین دو خوشه و میانگین فاصلهٔ تمام نقاط بین دو خوشه

هستند.

در پیاده‌سازی این الگوریتم، تابعی برای محاسبهٔ فاصله به‌کمک np.linalg.norm داریم. مطابق شکل (۴-۶)، تابع hierarchical_clustering ماتریسی از داده‌ها را به‌عنوان ورودی می‌گیرد. تعداد نمونه‌ها محاسبه می‌شود. در ابتدا، هر داده به‌صورت یک خوشه مجزا در نظر گرفته می‌شود. ساختار clusters یک دیکشنری است که کلیدهای آن شناسهٔ خوشه‌ها هستند و مقادیر آن فهرستی از نقاط موجود در هر خوشه است. فاصلهٔ بین هر جفت داده محاسبه و در دیکشنری distances ذخیره می‌شود. کلیدهای دیکشنری به صورت جفت داده‌های i و j و مقدار آن برابر فاصلهٔ محاسبه‌شده است. لیست merge_steps برای ذخیرهٔ مراحل ادغام خوشه‌ها و فاصله بین آنها استفاده می‌شود. تا زمانی که تنها یک خوشه باقی مانده باشد، نزدیک‌ترین جفت خوشه‌ها (کمترین فاصله) از میان مقادیر دیکشنری distances پیدا می‌شود. خوشه‌های نزدیک ادغام شده و به clusters اضافه می‌شوند. خوشه‌های قدیمی حذف می‌شوند و همچنین، فاصله‌های مرتبط با خوشه‌های حذف‌شده از دیکشنری distances حذف می‌شوند. فاصلهٔ بین خوشهٔ جدید و سایر خوشه‌ها دوباره محاسبه و به distances اضافه می‌شود. در نهایت، merge_steps برگردانده می‌شود.

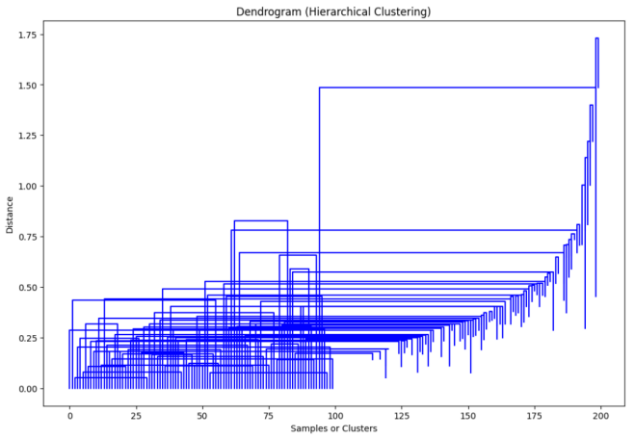
```python
def hierarchical_clustering(X):
    n_samples = X.shape[0]
    clusters = {i: [i] for i in range(n_samples)}
    distances = {
        (i, j): calculate_distance(X[i], X[j])
        for i in range(n_samples)
        for j in range(i + 1, n_samples)
    }
    merge_steps = []
    while len(clusters) > 1:
        (cluster_i, cluster_j), min_distance = min(
            distances.items(), key=lambda x: x[1]
        )
        merge_steps.append((cluster_i, cluster_j, min_distance))
        new_cluster = clusters[cluster_i] + clusters[cluster_j]
        new_cluster_id = max(clusters.keys()) + 1
        clusters[new_cluster_id] = new_cluster
        del clusters[cluster_i]
        del clusters[cluster_j]
        distances = {
            (i, j): dist
            for (i, j), dist in distances.items()
            if i not in [cluster_i, cluster_j] and j not in [cluster_i, cluster_j]
        }
        for cluster in clusters.keys():
            if cluster != new_cluster_id:
                dist = np.mean(
                    [
                        calculate_distance(X[p1], X[p2])
                        for p1 in clusters[cluster]
                        for p2 in new_cluster
                    ]
                )
                distances[
                    min(cluster, new_cluster_id), max(cluster, new_cluster_id)
                ] = dist
    return merge_steps
```

شکل (۴-۶) تابع hierarchical_clustering

در ادامه، تابع plot_dendrogram یک درخت سلسله‌مراتبی (دندروگرام) رسم می‌کند. این تابع دارای ورودی‌های merge_steps و n_samples است. نموداری ایجاد می‌شود و ارتفاع هر خوشه در نمودار و شناسهٔ خوشه‌های جدید تنظیم می‌شوند. برای هر مرحلهٔ ادغام، مختصات نقاط برای رسم خطوط مشخص می‌شوند، ارتفاع خوشه جدید برابر با فاصلهٔ ادغام آن با خوشه‌های دیگر قرار می‌گیرد و شناسهٔ خوشه جدید افزایش می‌یابد. در نهایت، نمودار رسم‌شده نمایش داده می‌شود. به‌عنوان پیش‌پردازش، داده‌ها آماده‌سازی می‌شوند، ستون diabetes حذف می‌شود و یک نمونهٔ صدتایی از داده‌ها انتخاب می‌شود. خوشه‌بندی سلسله‌مراتبی روی داده‌های نمونه‌شده اجرا می‌شود و مراحل ادغام خوشه‌ها تولید می‌شود. مراحل ادغام به همراه تعداد نمونه‌ها به تابع plot_dendrogram ارسال می‌شود تا دندروگرام رسم شود. این مراحل در شکل (۴–۷) پیاده‌سازی شده اند و در شکل (۴–۸) نمودار ترسیم‌شده قابل مشاهده است.

```python
def plot_dendrogram(merge_steps, n_samples):
    plt.figure(figsize=(12, 8))
    cluster_heights = {i: 0 for i in range(n_samples)}
    current_cluster_id = n_samples
    for cluster_i, cluster_j, distance in merge_steps:
        x1 = cluster_i if cluster_i < n_samples else current_cluster_id
        x2 = cluster_j if cluster_j < n_samples else current_cluster_id + 1
        y1 = cluster_heights[cluster_i]
        y2 = cluster_heights[cluster_j]
        plt.plot([x1, x1, x2, x2], [y1, distance, distance, y2], c="b")
        cluster_heights[current_cluster_id] = distance
        current_cluster_id += 1
    plt.title("Dendrogram (Hierarchical Clustering)")
    plt.xlabel("Samples or Clusters")
    plt.ylabel("Distance")
    plt.show()


data = preprocess_dataset(df)
data.drop(columns=["diabetes"], inplace=True)
sampled_data = data.sample(n=100, random_state=42).values
merge_steps = hierarchical_clustering(sampled_data)
plot_dendrogram(merge_steps, sampled_data.shape[0])
```

شکل (۴–۷) تابع plot_dendogram، پیش‌پردازش و اجرای توابع

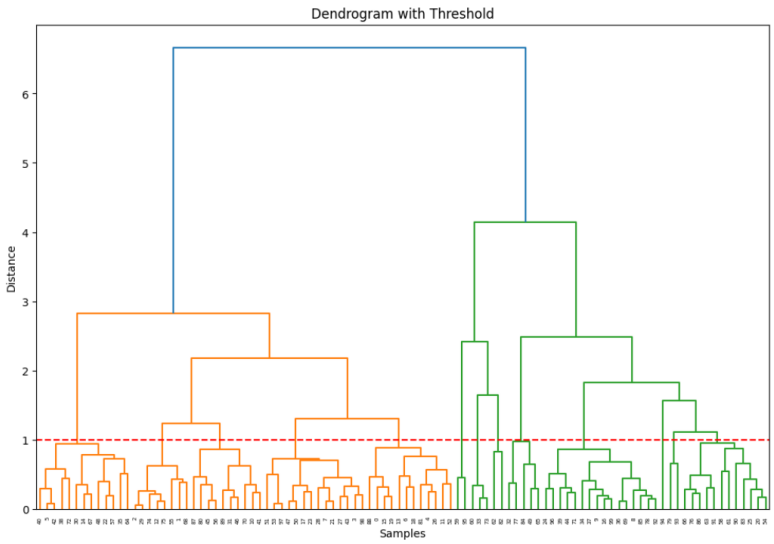


شکل (۴–۸) دندروگرام الگوریتم سلسله‌مراتبی روی نمونهٔ صدتایی از داده‌ها

در ادامه برای اعمال آستانه، مجدداً نمونه‌گیری انجام دادیم. از کتابخانهٔ scipy.cluster.hierarchy برای اجرای خوشه‌بندی سلسله‌مراتبی کمک گرفتیم که ورودی داده‌های نمونه‌شده و نوع متد را به‌عنوان ورودی دریافت می‌کند. ما در اینجا از متد ward استفاده کردیم تا واریانس بین خوشه‌ها را به حداقل برساند. خروجی، آرایه‌ای شامل اطلاعات خوشه‌بندی و فاصله‌های ادغام است. دندروگرام را با یک خط آستانه (y = 1) ترسیم کردیم که نشان می‌دهد ادغام خوشه‌ها تا این سطح، ادامه پیدا می‌کند. از fcluster برای خوشه‌بندی نهایی براساس آستانه استفاده کردیم که ورودی‌های آرایهٔ اطلاعات خوشه‌بندی، مقدار آستانه و معیار تعیین خوشه‌ها (که این‌جا فاصله است) را می‌گیرد و به‌عنوان خروجی، آرایه‌ای که نشان می‌دهد هر نمونه به کدام خوشه تعلق دارد را برمی‌گرداند. در ادامه، خوشه‌ها را به داده‌ها افزودیم و برای تحلیل، شمارهٔ خوشه و داده‌های مربوط به آن را نمایش دادیم. به شکل‌های (۴–۹)، (۴–۱۰) و (۴–۱۱) توجه کنید.

```
sampled_data = data.sample(n=100, random_state=42).values
linked = linkage(sampled_data, method='ward')
plt.figure(figsize=(12, 8))
dendrogram(linked)
plt.axhline(y=1.0, color='r', linestyle='--')
plt.title('Dendrogram with Threshold')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.show()
threshold = 1.0
clusters = fcluster(linked, t=threshold, criterion='distance')
clustered_data = data.sample(n=100, random_state=42).copy()
clustered_data['Cluster'] = clusters
for cluster_num in np.unique(clusters):
    print(f"Cluster {cluster_num}:")
    print(clustered_data[clustered_data['Cluster'] == cluster_num])
```

شکل (۴–۹) خوشه‌بندی سلسله‌مراتبی آماده با در نظر گرفتن آستانه و چاپ خوشه‌ها



شکل (۴–۱۰) دندروگرام الگوریتم سلسله‌مراتبی روی نمونهٔ صدتایی از داده‌ها با در نظر گرفتن آستانه

Cluster 1:

	gender	age	hypertension	heart_disease	smoking_history	\
43557	0	0.690476	0	0	0.6	
47435	0	0.357143	0	0	0.6	
7115	0	0.595238	0	0	1.0	
39354	0	0.392857	0	0	0.6	
89556	0	0.750000	0	0	0.8	
40150	0	0.750000	0	0	0.6	
48644	0	0.607143	0	0	0.8	
15057	0	0.630952	0	0	0.6	
2180	0	0.297619	0	0	1.0	
40365	0	0.500000	0	0	1.0	
20166	0	0.333333	0	0	1.0	
52461	0	0.380952	0	0	0.6	
14771	0	0.321429	0	0	0.6	

	bmi	HbA1c_level	blood_glucose_level	Cluster
43557	0.366623	0.400000	0.363636	1
47435	0.238931	0.272727	0.022727	1
7115	0.297245	0.272727	0.209091	1
39354	0.355051	0.000000	0.209091	1
89556	0.489625	0.090909	0.227273	1
40150	0.222767	0.472727	0.022727	1
48644	0.351246	0.563636	0.359091	1
15057	0.367161	0.454545	0.363636	1
2180	0.221443	0.490909	0.090909	1
40365	0.301553	0.272727	0.045455	1
20166	0.626354	0.000000	0.363636	1
52461	0.198884	0.236364	0.227273	1
14771	0.298454	0.563636	0.045455	1

شکل (۴-۱۱) نمونه‌داده‌های قرار گرفته در خوشهٔ اول به‌واسطهٔ الگوریتم سلسله‌مراتبی

بدین‌صورت، خوشه‌بندی روی داده‌ها انجام شد و می‌توان از آن‌ها، اطلاعات مختلفی کسب کرد.

در این گزارش سعی داشتیم تا بعد از آماده‌سازی داده‌ها با پیاده‌سازی درخت تصمیم و k-نزدیک‌ترین همسایه به پیش‌بینی کلاس نمونه‌ها بپردازیم. معیارهایی برای ارزیابی این الگوریتم‌ها داشتیم که به‌علت نمونه‌گیری در k-نزدیک‌ترین همسایه (به‌علت زمان‌بر بودن آن)، چندان قابل مقایسه با هم نبودند. مشاهده کردیم که درخت تصمیم با وجود سریع‌تر بودن، انعطاف‌پذیری کمتری نسبت به KNN دارد. نتایج کسب‌شده از معیارهای ارزیابی ممکن است تصادفی باشند و باید با کمک گرفتن از روش‌هایی نظیر Holdout Method، Cross-validation و Bootstrap و برآورد فاصلهٔ اطمینان (T-test) نسبت به بررسی تفاوت معنادار بین مدل‌ها پرداخت. در این فصل، خوشه‌بندی‌های K-Means و سلسله‌مراتبی را معرفی کردیم و از Elbow Method برای تعیین تعداد بهینهٔ خوشه‌ها استفاده کردیم. می‌دانیم که به‌دلیل استفاده از میانگین برای تعیین مراکز خوشه‌ها، محاسبات در این الگوریتم سریع‌تر است و پیاده‌سازی آسان‌تری دارد؛ اما به‌عنوان نقطهٔ ضعف، تعداد خوشه‌ها را از ما می‌خواهد و همچنین، به داده‌های پرت و مقیاس‌ها حساس است. این الگوریتم برای توزیع‌های کروی‌شکل، عملکرد خوبی دارد. الگوریتم‌های مشابهی که از مد به‌جای میانگین استفاده می‌کنند یا به‌جای میانگین‌گیری در هر مرحله یکی از داده‌های خوشه را با مرکز آن جابه‌جا می‌کنند نیز وجود دارند. سپس به خوشه‌بندی سلسله‌مراتبی پرداختیم و نشان دادیم که چگونه دندروگرام برای نمایش سلسله‌مراتب و ادغام تدریجی نمونه‌ها استفاده می‌شود و با تعیین آستانه می‌توان داده‌ها را به خوشه‌های نهایی تقسیم کرد. روش دیگری نیز ارائه شد و فرآیند را ساده‌تر کرد. این الگوریتم به نویز حساس است و در آن، انتخاب سطح مناسب برای برش درخت و تعیین تعداد خوشه‌ها چالش‌برانگیز است.