Report data mining

1)Abdelrahman nabil :2205115

2)john essam fawzy:2205098

3)ziad ibrahem elbatal:2205075

4)Marwan Mostafa Mostafa:2205159

1)data = pd.read_csv("diabetic_data.csv")

data.head()

#This code loads data from a CSV file named "diabetic_data.csv" into a Pandas DataFrame called data.

head() method is used to display the first few rows of the DataFrame.

----

2)data.isnull().sum()

data.drop_duplicates()

data.dropna(inplace=True)

data.replace('?', np.nan, inplace=True)

data.drop('weight', axis=1, inplace=True)

data.drop('payer_code', axis=1, inplace=True)

data.head()

#isnull().sum() counts the number of missing values in each column.

drop_duplicates() removes duplicate rows.

dropna(inplace=True) removes rows with missing values.

replace('?', np.nan, inplace=True) replaces '?' values with NaN.

drop('weight', axis=1, inplace=True) and .drop('payer_code', axis=1, inplace=True) drop the 'weight' and 'payer_code' columns, respectively.

----

3) plt.figure(figsize=(8, 6))

sns.countplot(x='gender' , data=data)

plt.title('Comparison of which gender is more at risk for having diabetes')

plt.xlabel('gender')

plt.ylabel('number of people with diabetes')

plt.show()

#This block of code creates a count plot using Seaborn to compare the counts of each gender in the dataset.

It sets the figure size, specifies the data (data), and defines the x-axis as 'gender'.

Titles, labels, and axis titles are added for better understanding.

Finally, it displays the plot.

---

4) plt.figure(figsize=(8, 6))

sns.boxplot(x='gender', y='age', data=data)

plt.title('')

plt.xlabel('gender')

plt.ylabel('age')

plt.show()

#This section creates a box plot using Seaborn to visualize the distribution of ages by gender.

It sets the figure size, specifies the data (data), and uses 'gender' for the x-axis and 'age' for the y-axis.

Titles, labels, and axis titles are adjusted, and the plot is displayed.

```
5) category_counts = data['race'].value_counts()

plt.figure(figsize=(8, 8))

plt.pie(category_counts, labels=category_counts.index, autopct='%1.1f%%', startangle=140)

plt.title('Comparison of which race is more at risk for having diabetes')

plt.show()
```

#This part calculates the count of each race in the dataset and creates a pie plot to visualize the distribution.

It sets the figure size, then uses the plt.pie() function to create the pie chart with appropriate labels, percentages, and angles.

The title is added, and the plot is displayed

```
6) inertia = []

for k in range(1, 11):

   kmeans = KMeans(n_clusters=k, random_state=42)

   kmeans.fit(data_scaled)

   inertia.append(kmeans.inertia_)
```

#This loop calculates the inertia (within-cluster sum of squares) for different numbers of clusters (k) using the KMeans algorithm.

For each value of k from 1 to 10, a KMeans model is fitted to the standardized data, and the inertia is computed and appended to the inertia list

---

```
7) plt.plot(range(1, 11), inertia, marker='o')
```

plt.xlabel('Number of clusters')

plt.ylabel('Inertia')

plt.title('Elbow Method')

plt.show()

#This code plots the inertia values against the number of clusters to visualize the "elbow" point, which helps determine the optimal number of clusters.

The x-axis represents the number of clusters (k), and the y-axis represents the inertia.

The plot is displayed to identify the elbow point.

----

8) optimal_k = method

kmeans = KMeans(n_clusters=optimal_k, random_state=42)

clusters = kmeans.fit_predict(data_scaled)

data['Cluster'] = clusters

#This part of the code selects the optimal number of clusters based on the elbow method (in this case, optimal_k is set to 3).

It initializes a KMeans model with the optimal number of clusters and fits it to the standardized data.

The fit_predict() method is used to both fit the model and predict the cluster labels for each data point.

The cluster labels are added to the original DataFrame data as a new column named 'Cluster'

---

9) plt.figure(figsize=(10, 6))

sns.scatterplot(x=data_scaled[:, 0], y=data_scaled[:, 1], hue=data['Cluster'], palette='Set1')

plt.title('Clustering Analysis')

plt.xlabel(features_for_clustering[0])

```
plt.ylabel(features_for_clustering[1])

plt.legend(title='Cluster')

plt.grid(True)

plt.show()
```

#This section creates a scatter plot to visualize the clustering analysis results.

It sets the figure size and uses Seaborn's scatterplot() function to plot the data points.

The x-axis represents the first feature selected for clustering (features_for_clustering[0]), and the y-axis represents the second feature (features_for_clustering[1]).

Data points are colored according to their assigned clusters, and a legend is added to indicate cluster labels.

The plot is displayed to visualize the clustering results.

```
10) linked = linkage(sampled_data, method='ward', metric='euclidean')

plt.figure(figsize=(10, 7))

dendrogram(linked, orientation='top', distance_sort='descending',
show_leaf_counts=True)

plt.title('Hierarchical Clustering Dendrogram')

plt.xlabel('Data Points')

plt.ylabel('Distance')

plt.show()
```

#This code performs hierarchical clustering using the linkage function from SciPy. The method used for linkage is "ward", and the distance metric is "euclidean".

The resulting linkage matrix is stored in the linked variable.

A dendrogram is then created using the dendrogram function, which visualizes the hierarchical clustering results.

The plot is displayed with appropriate titles and labels.

---

11) preprocessor = ColumnTransformer(

    transformers=[

        ('num', 'passthrough', numerical_cols), # numerical columns passed through

        ('cat', OneHotEncoder(), categorical_cols) # categorical columns one-hot
encoded

    ])

#his code defines a ColumnTransformer named preprocessor to preprocess both numerical and categorical features.

Two transformers are specified:

The first transformer ('num') passes through numerical columns unchanged (using 'passthrough').

The second transformer ('cat') applies OneHotEncoder to encode categorical columns

---

12) kmeans_pipeline = Pipeline([

    ('preprocessor', preprocessor),

    ('kmeans', KMeans(n_clusters=2, random_state=42))

])

#This section sets up a pipeline (kmeans_pipeline) for KMeans clustering.

The pipeline consists of two steps:

The first step ('preprocessor') applies the defined preprocessing transformations.

The second step ('kmeans') performs KMeans clustering with a specified number of clusters (2 in this case).

---

13) lr_pipeline = Pipeline([

    ('preprocessor', preprocessor),

    ('lr', LinearRegression())

```
])
```

#Similarly, this code sets up a pipeline (lr_pipeline) for linear regression modeling.

It also consists of two steps:

The first step ('preprocessor') preprocesses the data using the same transformations as in the KMeans pipeline.

The second step ('lr') applies linear regression modeling.

----

14) # Fitting and predicting using KMeans

```
kmeans_pipeline.fit(X_concat)

y_pred_kmeans = kmeans_pipeline.predict(X_test)

print("K-Means Clustering Results:")

print("Accuracy:", accuracy_score(y_test, y_pred_kmeans))

print("Classification Report:")

print(classification_report(y_test, y_pred_kmeans))

print("Confusion Matrix:")

print(confusion_matrix(y_test, y_pred_kmeans))

print("-" * 50)
```

#This part fits the KMeans clustering pipeline (kmeans_pipeline) to the concatenated training and testing data (X_concat).

It then predicts cluster labels for the test data (X_test) using the fitted pipeline.

Accuracy, classification report, and confusion matrix are calculated and printed to evaluate the performance of KMeans clustering on the test data.

---

15) # Fitting and predicting using Linear Regression

```
lr_pipeline.fit(X_concat, pd.concat([y_train, y_test], ignore_index=True))

y_pred_lr = lr_pipeline.predict(X_test)

y_pred_lr_binary = (y_pred_lr > 0.5).astype(int)
```

```
print("Linear Regression Results:")

print("Accuracy:", accuracy_score(y_test, y_pred_lr_binary))

print("Classification Report:")

print(classification_report(y_test, y_pred_lr_binary))

print("Confusion Matrix:")

print(confusion_matrix(y_test, y_pred_lr_binary))
```

#Similarly, this section fits the linear regression pipeline (lr_pipeline) to the concatenated training and testing data.

It then predicts the target variable for the test data using the fitted pipeline.

The predicted values are binarized using a threshold of 0.5.

Accuracy, classification report, and confusion matrix are calculated and printed to evaluate the performance of linear regression on the test data.