databricks ACT18

(https://databricks.com)

```
df1 = spark.read.format("csv").load("dbfs:/FileStore/tables/Transformations.csv",header="True",inferSchema="True")
```

Here we are reading the file Transformations.csv, we are letting spark infer the schema and use the first row as the headers into our dataframe called df1

```
df1.count()
```

```
Out[3]: 20
```

we are counting the rows in our new dataframe and we have 20 rows

```
df1.show()
```

```
+-----+-----+-----+-----+-----+-----+-----+---------+
|col_1|col_2|col_3|col_4|col_5|col_6|col_7|    label|
+-----+-----+-----+-----+-----+-----+-----+---------+
|    3|   25|   21|    3|    2|   20|    5| 0.111111|
|    4|   16|   20|    1|    1|   16|    3|   2.0329|
|    2|   15|   16|    2|    1|   17|    5| 4.799999|
|    4|   24|   23|    5|    4|   12|    4|   2.9254|
|    2|   25|    4|    1|    1|   18|    2|  4.66666|
|    3|   17|   18|    4|    2|   13|    4| 4.666666|
|    4|   18|   13|    0|    2|   17|    4|1.7229999|
|    5|   17|   11|    0|    2|   14|    4|    1.521|
|    2|   18|   11|    5|    4|   19|    3|   2.0198|
|    3|   21|   12|    1|    3|   13|    2|      2.0|
|    2|   22|   18|    0|    4|   19|    3|   0.4889|
|    1|   18|   12|    4|    3|   13|    5|    0.927|
|    4|   18|   16|    0|    2|   19|    2|   0.7128|
|    3|   25|   20|    1|    2|   20|    3|    4.092|
|    5|   25|   23|    2|    3|   19|    4|    2.902|
|    1|   22|   16|    3|    2|   19|    2|    1.904|
|    2|   16|   10|    2|    2|   13|    5|   5.2567|
|    2|   19|   23|    2|    1|   13|    5|   5.2567|
```

Here we are showing the contents of our df1 dataframe. The result is text.

```
df1.display()
```

Table

New result table: ON

Here we are displaying our df1 dataframe and the .display is a native pyspark methodh that displays data in tubular and more interactive format.

```
from pyspark.ml.feature import Binarizer
binarizer = Binarizer(threshold=0.99, inputCol="label",outputCol="binarized_label")
```

Here we are first importing the Binarizer class from the PySpark ML library. The Binarizer is a feature transformer that is used to convert continuous numerical features into binary (0/1) features.
The second line creates an instance of the Binarizer.
The treshold value of 0.99 sets the value for the binarization process.
Question 1:
The Binarizer is used in data processing and machine learning to transform continuous numerical features into binary features based on a threshold. It is particularly useful when you need to categorize data into two distinct groups based on a specific criterion.

Example Use Case - Email Spam Detection:

Imagine a scenario where an email service provider wants to classify emails as either spam or not spam. One of the features they might consider is the frequency of certain trigger words known to be common in spam emails, like "lottery" or "prize."

Suppose they decide that any email containing these words more than a certain number of times is likely to be spam. Here, the Binarizer can be applied. For instance, if the threshold is set at 3, then any email with more than 3 occurrences of the trigger words would be classified as spam (1), and those with 3 or fewer occurrences as not spam (0).

By using a Binarizer to transform the count of trigger words into a binary feature, the email service provider simplifies the feature for use in a machine learning model. This binary feature effectively captures the presence or absence of a significant frequency of trigger words, which can be a strong indicator of spam.

In this case, the Binarizer helps in creating a clear, binary distinction in a key feature for spam detection, enhancing the effectiveness of the classification model in distinguishing spam from regular emails.

```
new_df=binarizer.transform(df1)
```

here we are creating a new_df that will contain the results of the transformation that is an additional column binarized_label. The .transform will actually apply the transformation defained in the Binarizer intance.

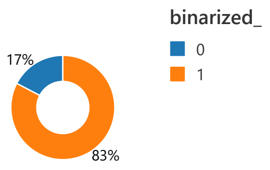
```
new_df.show()
```

```
+-----+-----+-----+-----+-----+-----+-----+---------+---------------+
|col_1|col_2|col_3|col_4|col_5|col_6|col_7|    label|binarized_label|
+-----+-----+-----+-----+-----+-----+-----+---------+---------------+
|    3|   25|   21|    3|    2|   20|    5| 0.111111|            0.0|
|    4|   16|   20|    1|    1|   16|    3|   2.0329|            1.0|
|    2|   15|   16|    2|    1|   17|    5| 4.799999|            1.0|
|    4|   24|   23|    5|    4|   12|    4|   2.9254|            1.0|
|    2|   25|    4|    1|    1|   18|    2|  4.66666|            1.0|
|    3|   17|   18|    4|    2|   13|    4| 4.666666|            1.0|
|    4|   18|   13|    0|    2|   17|    4|1.7229999|            1.0|
|    5|   17|   11|    0|    2|   14|    4|    1.521|            1.0|
|    2|   18|   11|    5|    4|   19|    3|   2.0198|            1.0|
|    3|   21|   12|    1|    3|   13|    2|      2.0|            1.0|
|    2|   22|   18|    0|    4|   19|    3|   0.4889|            0.0|
|    1|   18|   12|    4|    3|   13|    5|    0.927|            0.0|
|    4|   18|   16|    0|    2|   19|    2|   0.7128|            0.0|
|    3|   25|   20|    1|    2|   20|    3|    4.092|            1.0|
|    5|   25|   23|    2|    3|   19|    4|    2.902|            1.0|
|    1|   22|   16|    3|    2|   19|    2|    1.904|            1.0|
|    2|   16|   10|    2|    2|   13|    5|   5.2567|            1.0|
```

Here we are .show ing the rows of our new_def with our new feature with binarized data of 1.0s and 0.0s

```
new_df.display()
```

Visualization   New result table: ON



20 rows

Here we are .display ing the contents of our new_df and adding a donut graph that will show us that 83% of the rows are 1s and that the remaining 17% are 0s. I like the dounut graph for this specific feature because it shows that the 1s have a longer circumference of the dougnut than the 0s.

```
from pyspark.ml.feature import PCA
```

Here we are importing the Principal Component Analysis (PCA) feature transformer from PySpark's machine learning library,

```
from pyspark.ml.feature import VectorAssembler
```

Here we are importing the VectorAssembler feature transformer from the PySpark's machine learning library.

```
assembler = VectorAssembler(inputCols=[col for col in df1.columns if col !='label'], outputCol="features")
```

Here we are creating an instance of the VectorAssembler transformer in PySpark, which is used for combining multiple columns into a single vector column.
We are going to include all columns not named label as input columns and a new column named feature as the output column with the assembled rows.

```
df_new=assembler.transform(df1)
```

Here we are actually applying .transform that is the assembler to the df_new dataframe

```
df_new.show(truncate=False)
```

```
+-----+-----+-----+-----+-----+-----+-----+---------+--------------------------------+
|col_1|col_2|col_3|col_4|col_5|col_6|col_7|label    |features                        |
+-----+-----+-----+-----+-----+-----+-----+---------+--------------------------------+
|3    |25   |21   |3    |2    |20   |5    |0.111111 |[3.0,25.0,21.0,3.0,2.0,20.0,5.0]|
|4    |16   |20   |1    |1    |16   |3    |2.0329   |[4.0,16.0,20.0,1.0,1.0,16.0,3.0]|
|2    |15   |16   |2    |1    |17   |5    |4.799999 |[2.0,15.0,16.0,2.0,1.0,17.0,5.0]|
|4    |24   |23   |5    |4    |12   |4    |2.9254   |[4.0,24.0,23.0,5.0,4.0,12.0,4.0]|
|2    |25   |4    |1    |1    |18   |2    |4.66666  |[2.0,25.0,4.0,1.0,1.0,18.0,2.0] |
|3    |17   |18   |4    |2    |13   |4    |4.666666 |[3.0,17.0,18.0,4.0,2.0,13.0,4.0]|
|4    |18   |13   |0    |2    |17   |4    |1.7229999|[4.0,18.0,13.0,0.0,2.0,17.0,4.0]|
|5    |17   |11   |0    |2    |14   |4    |1.521    |[5.0,17.0,11.0,0.0,2.0,14.0,4.0]|
|2    |18   |11   |5    |4    |19   |3    |2.0198   |[2.0,18.0,11.0,5.0,4.0,19.0,3.0]|
|3    |21   |12   |1    |3    |13   |2    |2.0      |[3.0,21.0,12.0,1.0,3.0,13.0,2.0]|
|2    |22   |18   |0    |4    |19   |3    |0.4889   |[2.0,22.0,18.0,0.0,4.0,19.0,3.0]|
|1    |18   |12   |4    |3    |13   |5    |0.927    |[1.0,18.0,12.0,4.0,3.0,13.0,5.0]|
|4    |18   |16   |0    |2    |19   |2    |0.7128   |[4.0,18.0,16.0,0.0,2.0,19.0,2.0]|
|3    |25   |20   |1    |2    |20   |3    |4.092    |[3.0,25.0,20.0,1.0,2.0,20.0,3.0]|
|5    |25   |23   |2    |3    |19   |4    |2.902    |[5.0,25.0,23.0,2.0,3.0,19.0,4.0]|
|1    |22   |16   |3    |2    |19   |2    |1.904    |[1.0,22.0,16.0,3.0,2.0,19.0,2.0]|
|2    |16   |10   |2    |2    |13   |5    |5.2567   |[2.0,16.0,10.0,2.0,2.0,13.0,5.0]|
|2    |19   |23   |2    |1    |13   |5    |5.2567   |[2.0,19.0,23.0,2.0,1.0,13.0,5.0]|
```

Here we are .show ing the contens of the df_new dataframe. Notice the new vectorized column called features and also notice that column binarized_labels is no longer here. I think that is because we are not saving and we are not commiting our tables to be permanent.

```
pca = PCA(k=2, inputCol="features", outputCol="pca_features")
```

Here we are creating an instance of the PCA (Principal Component Analysis) transformer in PySpark. PCA is used for dimensionality reduction.
k=2 will reduce the dimensionality of our data to 2 principal components. This means it aims to capture most of the variance in the data using just two new features that are linear combinations of the original features.
We are using the column features for the input and the column pca_features for the output column.
Question 2:
Principal Component Analysis (PCA) is used in data processing and machine learning for dimensionality reduction, feature extraction, and data visualization. It's particularly useful when dealing with high-dimensional data, helping to uncover the underlying structure, reduce noise, and improve efficiency.

Example Use Case - Data Compression in Image Processing:

Consider a scenario where a tech company needs to process a large set of high-resolution images for a facial recognition system. Each image consists of thousands of pixels, which translates into high-dimensional data. Processing this high-dimensional data is computationally expensive and can lead to overfitting in machine learning models.

By applying PCA to these images, the company can reduce the dimensionality of the data. PCA achieves this by identifying the most important features (principal components) that capture the maximum variance in the dataset. These principal components are linear combinations of the original pixel values and often much fewer in number compared to the original dimensions.

For example, if each image is represented by 10,000 pixels (features), PCA might reveal that 95% of the variance in the images can be captured by just 100 principal components. By transforming the original pixel-based features into these principal components, the company can significantly reduce the size of the dataset while retaining most of the critical information.

This reduced dataset is much more efficient to process and analyze. In the facial recognition system, using these principal components instead of the full pixel data can speed up the recognition process and reduce the resources required, while maintaining accuracy.

In summary, PCA in this case helps in compressing the image data, reducing computational requirements, and potentially improving the performance of machine learning algorithms by eliminating redundant and less informative features.

```
pca_model=pca.fit(df_new)
```

Here we are training the PCA model on the df_new dataframe. This training involves finding the axes (principal components) that maximize the variance in the dataset. Once trained, the pca_model will be used to transform the data in df_new into its principal components, effectively reducing the number of features in the dataset. This is often done to simplify the dataset and can improve the performance of machine learning models.

```
pca_comp = pca_model.transform(df_new).select("pca_features")
```

Here we are applying the .transform ing to the previously trained pca_model to the df_new dataframe, and then selecting the newly created column pca_features.

```
pca_comp.show(truncate=False)
```

```
+-----------------------------------------+
|pca_features                             |
+-----------------------------------------+
|[-30.173180617979533,-22.53887729998468] |
|[-25.780299595500193,-14.316380325537374]|
|[-21.750960897320603,-15.00357203829197] |
|[-31.28436164086435,-16.725703879757454] |
|[-13.864499911628316,-27.165265081462195]|
|[-24.18078247861391,-13.67287573838345]  |
|[-19.97560937801268,-18.399953375520326] |
|[-17.62675574064221,-16.539395584214013] |
|[-18.446380898440733,-20.215193567841624]|
|[-19.673806397491724,-18.948632784992526]|
|[-25.946564733116485,-21.212826884907443]|
|[-18.789571971132045,-16.110280170959438]|
|[-22.87314965270604,-18.987303589607215] |
|[-29.042657975926005,-23.27519709861832] |
|[-32.03817799673795,-21.584238153159305] |
|[-24.173392898581064,-21.824230161153764]|
|[-16.19029654981938,-15.295639144304715] |
|[-29.2967089047592,-13.492603619968161]  |
```

So now the pca_comp contains only the pca_features column. so we are .show ing this column with the reduced dinemtionality. The truncate=false means that the contents of each row in the DataFrame should be displayed in full, without truncation.

```
from pyspark.ml.feature import Normalizer
```

Here we are importing the Normalizer feature transformer from the PySpark's machine learning library.

```
normalizer = Normalizer(inputCol="features",outputCol="norm_features", p=1.0)
```

Here we are Creating an Instance of Normalizer transfromer and assigning it to the normalizer variable. Our input column is features and tour output column is norm_features
The p parameter specifies the p-norm used for normalization. Here, p=1.0 means that L1 norm is used. In L1 normalization, the values in each row are transformed so that the sum of their absolute values equals 1. This is also known as Manhattan distance or taxicab norm.
Question 3:
Normalizer is used in data processing and machine learning to scale individual data instances (rows) to have unit norm. This technique is particularly useful when the magnitude of the data vector is important but the direction of the vector is of primary interest.

Example Use Case - Text Similarity in Document Clustering:

Imagine a scenario where a company wants to cluster a large collection of documents to identify groups of similar documents. Each document is represented as a vector of word counts or term frequencies (a common approach in text processing known as Bag-of-Words).

If we use these vectors directly for clustering, documents that are longer and have higher word counts overall might dominate the clustering process, overshadowing the actual content similarities. To prevent this, we can use the Normalizer to scale each document vector to have a unit length. This way, the comparison between documents focuses on the direction of the vectors (which indicates the distribution of words) rather than their magnitude (which indicates the length of the documents).

By normalizing the document vectors, the clustering algorithm (like K-Means or Hierarchical Clustering) can more accurately group documents based on the similarity of their content, irrespective of their length. This leads to more meaningful and content-focused document clusters.

In this case, Normalizer helps ensure that the clustering analysis is based on the patterns of word usage within the documents rather than being influenced by the length of the documents, making the clusters more relevant and useful for understanding document similarities.

```
normalised_l1_data = normalizer.transform(df_new)
```

Here we are Applying the Normalizer Transformer to the df_new dataframe. Now the normalised_l1_data dataframe includes the columns of df_new plus the transformed column norm_features

```
normalised_l1_data.select('norm_features').show(truncate=False)
```

```
+-------------------------------------------------------------------------------------------------------------------------------
---------------+
|norm_features
|
+-------------------------------------------------------------------------------------------------------------------------------
---------------+
|[0.0379746835443038,0.31645569620253167,0.26582278481012656,0.0379746835443038,0.02531645569620253,0.25316455696202533,0.06329113
924050633]     |
|[0.06557377049180328,0.26229508196721313,0.32786885245901637,0.01639344262295082,0.01639344262295082,0.26229508196721313,0.049180
32786885246]   |
|[0.034482758620689655,0.25862068965517243,0.27586206896551724,0.034482758620689655,0.017241379310344827,0.29310344827586204,0.086
20689655172414]|
|[0.05263157894736842,0.3157894736842105,0.3026315789473684,0.06578947368421052,0.05263157894736842,0.15789473684210525,0.05263157
894736842]     |
|[0.03773584905660377,0.4716981132075472,0.07547169811320754,0.018867924528301886,0.018867924528301886,0.33962264150943394,0.03773
584905660377]  |
|[0.04918032786885246,0.2786885245901639,0.29508196721311475,0.06557377049180328,0.03278688524590164,0.21311475409836064,0.0655737
7049180328]    |
|[0.06896551724137931,0.3103448275862069,0.22413793103448276,0.0,0.034482758620689655,0.29310344827586204,0.06896551724137931]
|
|[0.09433962264150944,0.32075471698113206,0.20754716981132076,0.0,0.03773584905660377,0.2641509433962264,0.07547169811320754]
```

Here we are .show ing the new transformed column norm_features. If we add up the values in each of this rows the result should be 1.

```
from pyspark.ml.feature import StandardScaler
```

Here we are importing the StandardScale feature transformer from the PySpark's machine learning library.

```
scaler = StandardScaler (inputCol="features",outputCol="scaled_features",withStd=False, withMean=True)
```

Here we are creating an Instance of StandardScaler to be save in the scaler variable. We are using the features column as input and the scaled_features as the output column. Please note that the withstd is set to False so the values in each feature will not be scaled by the standard deviation of the feature. But withMean is set to True so the mean of each feature should be subtracted, centering the data around 0. It is useful for algorithms that assume input features are centered around zero.
This line of code is configuring a StandardScaler to take the vectors in the "features" column of a DataFrame, subtract their mean (thereby centering the data), but not scale the data to unit variance. The results with the standardized features will be stored in a new column called "scaled_features"
Question 4:
Standard Scaling, also known as Standardization, is used in data processing and machine learning to transform features to have a mean of zero and a standard deviation of one. It's particularly useful in algorithms that are sensitive to the scale and distribution of input features.

Example Use Case - K-Means Clustering in Customer Segmentation:

Consider a scenario where a retail company wants to segment its customers for targeted marketing based on their annual income and spending score. These two features might be on very different scales; for instance, annual income could range in the thousands, while spending score might be on a scale of 1 to 100.

If you apply a K-Means clustering algorithm directly to this data without scaling, the model might disproportionately weigh the income feature due to its larger magnitude. This could lead to suboptimal clustering, where the algorithm overemphasizes income while underrepresenting the importance of spending score.

By using Standard Scaling, we transform both features depending on settings, to have a mean of zero and a standard deviation of one, ensuring they contribute equally to the distance calculations in the K-Means algorithm. This results in more meaningful clusters that accurately reflect both income and spending habits, allowing for more effective customer segmentation and targeted marketing strategies.

Standard Scaling here ensures that each feature contributes equally to the analysis, preventing features with larger magnitudes from dominating the model's behavior, which is crucial for distance-based algorithms like K-Means.

```
scaler_model = scaler.fit(df_new)
```

Here we are Fitting the StandardScaler Model to the dataframe df_new and also creating the scaler_model that is an instance of the fitted StandardScaler model. It contains the mean computed from df_new and can be used to transform the DataFrame to standardize its features.

```
scaled_data = scaler_model.transform(df_new)
```

Here we are applying the Trained StandardScaler Model to transform the dataframedf_new. Since your StandardScaler was configured with withStd=False and withMean=True, this transformation will subtract the mean from each feature (column) in the inputCol "features" but will not scale the features to unit variance. We are then creating a new dataframe called scaled_data.

```
scaled_data.select('scaled_features').show(truncate=False)
```

```
+----------------------------------------------------------------------------------------------------------------+
|scaled_features                                                                                                 |
+----------------------------------------------------------------------------------------------------------------+
|[-0.050000000000000266,4.700000000000003,4.75,0.7499999999999991,-0.20000000000000018,3.5,1.5000000000000004]   |
|[0.9499999999999997,-4.299999999999997,3.75,-1.2500000000000009,-1.2000000000000002,-0.5,-0.49999999999999956]  |
|[-1.0500000000000003,-5.299999999999997,-0.25,-0.2500000000000009,-1.2000000000000002,0.5,1.5000000000000004]   |
|[0.9499999999999997,3.700000000000003,6.75,2.749999999999999,1.7999999999999998,-4.5,0.5000000000000004]        |
|[-1.0500000000000003,4.700000000000003,-12.25,-1.2500000000000009,-1.2000000000000002,1.5,-1.4999999999999996]  |
|[-0.050000000000000266,-3.299999999999997,1.75,1.7499999999999991,-0.20000000000000018,-3.5,0.5000000000000004] |
|[0.9499999999999997,-2.299999999999997,-3.25,-2.250000000000001,-0.20000000000000018,0.5,0.5000000000000004]    |
|[1.9499999999999997,-3.299999999999997,-5.25,-2.250000000000001,-0.20000000000000018,-2.5,0.5000000000000004]   |
|[-1.0500000000000003,-2.299999999999997,-5.25,2.749999999999999,1.7999999999999998,2.5,-0.49999999999999956]    |
|[-0.050000000000000266,0.7000000000000028,-4.25,-1.2500000000000009,0.7999999999999998,-3.5,-1.4999999999999996]|
|[-1.0500000000000003,1.7000000000000028,1.75,-2.250000000000001,1.7999999999999998,2.5,-0.49999999999999956]    |
|[-2.0500000000000003,-2.299999999999997,-4.25,1.7499999999999991,0.7999999999999998,-3.5,1.5000000000000004]    |
|[0.9499999999999997,-2.299999999999997,-0.25,-2.250000000000001,-0.20000000000000018,2.5,-1.4999999999999996]   |
|[-0.050000000000000266,4.700000000000003,3.75,-1.2500000000000009,-0.20000000000000018,3.5,-0.49999999999999956]|
|[1.9499999999999997,4.700000000000003,6.75,-0.2500000000000009,0.7999999999999998,2.5,0.5000000000000004]       |
|[-2.0500000000000003,1.7000000000000028,-0.25,0.7499999999999991,-0.20000000000000018,2.5,-1.4999999999999996]  |
|[-1.0500000000000003,-4.299999999999997,-6.25,-0.2500000000000009,-0.20000000000000018,-3.5,1.5000000000000004] |
|[-1.0500000000000003,-1.2999999999999972,6.75,-0.2500000000000009,-1.2000000000000002,-3.5,1.5000000000000004]  |
```

Here we are selecting one column scaled_features from the new dataframe called scaled_data the . show ing it with truncate equals to false so that we can see the complete contents of each row.

```
from pyspark.ml.feature import MinMaxScaler
```

The code from pyspark.ml.feature import MinMaxScaler imports the MinMaxScaler class from PySpark's machine learning library,

```
mm_scaler = MinMaxScaler(inputCol="features",outputCol="mm_scaled_features")
```

The code mm_scaler = MinMaxScaler(inputCol="features", outputCol="mm_scaled_features") creates an instance of the MinMaxScaler transformer, configuring it to rescale the data in the "features" column and output the scaled data in a new column named "mm_scaled_features".

```
mm_scaler_model = mm_scaler.fit(df_new)
```

The code mm_scaler_model = mm_scaler.fit(df_new) fits the MinMaxScaler instance mm_scaler to the DataFrame df_new, creating a model that can be used to transform the data by scaling each feature to a range from 0 to 1.

```
rescaled_df = mm_scaler_model.transform(df_new)
```

The code rescaled_df = mm_scaler_model.transform(df_new) applies the trained MinMaxScaler model to the DataFrame df_new, producing a new DataFrame rescaled_df with features scaled to a range from 0 to 1.

```
rescaled_df.select("features", "mm_scaled_features").show(truncate=False)
```

```
+--------------------------------+----------------------------------------------------------------------------------------------+
|features                        |mm_scaled_features                                                                            |
+--------------------------------+----------------------------------------------------------------------------------------------+
|[3.0,25.0,21.0,3.0,2.0,20.0,5.0]|[0.5,1.0,0.894736842105263,0.6000000000000001,0.3333333333333333,1.0,1.0]                     |
|[4.0,16.0,20.0,1.0,1.0,16.0,3.0]|[0.75,0.1,0.8421052631578947,0.2,0.0,0.5,0.3333333333333333]                                  |
|[2.0,15.0,16.0,2.0,1.0,17.0,5.0]|[0.25,0.0,0.631578947368421,0.4,0.0,0.625,1.0]                                                |
|[4.0,24.0,23.0,5.0,4.0,12.0,4.0]|[0.75,0.9,1.0,1.0,1.0,0.0,0.6666666666666666]                                                 |
|[2.0,25.0,4.0,1.0,1.0,18.0,2.0] |[0.25,1.0,0.0,0.2,0.0,0.75,0.0]                                                               |
|[3.0,17.0,18.0,4.0,2.0,13.0,4.0]|[0.5,0.2,0.7368421052631579,0.8,0.3333333333333333,0.125,0.6666666666666666]                  |
|[4.0,18.0,13.0,0.0,2.0,17.0,4.0]|[0.75,0.30000000000000004,0.47368421052631576,0.0,0.3333333333333333,0.625,0.6666666666666666]|
|[5.0,17.0,11.0,0.0,2.0,14.0,4.0]|[1.0,0.2,0.3684210526315789,0.0,0.3333333333333333,0.25,0.6666666666666666]                   |
|[2.0,18.0,11.0,5.0,4.0,19.0,3.0]|[0.25,0.30000000000000004,0.3684210526315789,1.0,1.0,0.875,0.3333333333333333]                |
|[3.0,21.0,12.0,1.0,3.0,13.0,2.0]|[0.5,0.6000000000000001,0.42105263157894735,0.2,0.6666666666666666,0.125,0.0]                 |
|[2.0,22.0,18.0,0.0,4.0,19.0,3.0]|[0.25,0.7000000000000001,0.7368421052631579,0.0,1.0,0.875,0.3333333333333333]                 |
|[1.0,18.0,12.0,4.0,3.0,13.0,5.0]|[0.0,0.30000000000000004,0.42105263157894735,0.8,0.6666666666666666,0.125,1.0]                |
|[4.0,18.0,16.0,0.0,2.0,19.0,2.0]|[0.75,0.30000000000000004,0.631578947368421,0.0,0.3333333333333333,0.875,0.0]                 |
|[3.0,25.0,20.0,1.0,2.0,20.0,3.0]|[0.5,1.0,0.8421052631578947,0.2,0.3333333333333333,1.0,0.3333333333333333]                    |
|[5.0,25.0,23.0,2.0,3.0,19.0,4.0]|[1.0,1.0,1.0,0.4,0.6666666666666666,0.875,0.6666666666666666]                                 |
|[1.0,22.0,16.0,3.0,2.0,19.0,2.0]|[0.0,0.7000000000000001,0.631578947368421,0.6000000000000001,0.3333333333333333,0.875,0.0]    |
|[2.0,16.0,10.0,2.0,2.0,13.0,5.0]|[0.25,0.1,0.3157894736842105,0.4,0.3333333333333333,0.125,1.0]                                |
|[2.0,19.0,23.0,2.0,1.0,13.0,5.0]|[0.25,0.4,1.0,0.4,0.0,0.125,1.0]                                                              |
```

Here we are selecting one column mm_scaled_features and features from the new dataframe called rescaled_df then . show ing it with truncate equals to false so that we can see the complete contents of each row.

```
mm_scaler.getMin()
```

```
Out[33]: 0.0
```

The code mm_scaler.getMin() retrieves the lower bound of the range to which the MinMaxScaler will scale the input data. We get 0.0 as expected

```
mm_scaler.getMax()
```

```
Out[34]: 1.0
```

The code mm_scaler.getMax() retrieves the upper bound of the range to which the MinMaxScaler will scale the input data. We get 1.0 as expected.

```
mm_scaler =MinMaxScaler(inputCol="features",outputCol="mm_scaled_features", min=-1,max=1)
```

The code mm_scaler = MinMaxScaler(inputCol="features", outputCol="mm_scaled_features", min=-1,max=1) creates an instance of the MinMaxScaler transformer, configuring it to rescale the data in the "features" column and output the scaled data in a new column named "mm_scaled_features". We are also setting the range here from -1 to 1.

```
mm_scaler_model = mm_scaler.fit(df_new)
```

The code mm_scaler_model = mm_scaler.fit(df_new) fits the MinMaxScaler instance mm_scaler to the DataFrame df_new, creating a model that can be used to transform the data by scaling each feature to a range from -1 to 1.

```
rescaled_df = mm_scaler_model.transform(df_new)
```

The code rescaled_df = mm_scaler_model.transform(df_new) applies the trained MinMaxScaler model to the DataFrame df_new, producing a new DataFrame rescaled_df with features scaled to a range from -1 to 1.

```
rescaled_df.select("features", "mm_scaled_features").show(truncate=False)
```

```
+--------------------------------+-------------------------------------------------------------------------------------------------
--+
|features                        |mm_scaled_features
|
+--------------------------------+-------------------------------------------------------------------------------------------------
--+
|[3.0,25.0,21.0,3.0,2.0,20.0,5.0]|[0.0,1.0,0.7894736842105261,0.20000000000000018,-0.33333333333333337,1.0,1.0]
|
|[4.0,16.0,20.0,1.0,1.0,16.0,3.0]|[0.5,-0.8,0.6842105263157894,-0.6,-1.0,0.0,-0.33333333333333337]
|
|[2.0,15.0,16.0,2.0,1.0,17.0,5.0]|[-0.5,-1.0,0.26315789473684204,-0.19999999999999996,-1.0,0.25,1.0]
|
|[4.0,24.0,23.0,5.0,4.0,12.0,4.0]|[0.5,0.8,1.0,1.0,1.0,-1.0,0.33333333333333326]
|
|[2.0,25.0,4.0,1.0,1.0,18.0,2.0] |[-0.5,1.0,-1.0,-0.6,-1.0,0.5,-1.0]
|
|[3.0,17.0,18.0,4.0,2.0,13.0,4.0]|[0.0,-0.6,0.4736842105263157,0.6000000000000001,-0.33333333333333337,-0.75,0.33333333333333326]
|
|[4.0,18.0,13.0,0.0,2.0,17.0,4.0]|[0.5,-0.3999999999999999,-0.052631578947368474,-1.0,-0.33333333333333337,0.25,0.3333333333333332
6]|
|[5.0,17.0,11.0,0.0,2.0,14.0,4.0]|[1.0,-0.6,-0.26315789473684215,-1.0,-0.33333333333333337,-0.5,0.33333333333333326]
```

Here we are selecting one column mm_scaled_features and features from the new dataframe called rescaled_df then . show ing it with truncate equals to false so that we can see the complete contents of each row. Notice that we can now see negative numbers.

```
from pyspark.ml.feature import MaxAbsScaler
```

The code from pyspark.ml.feature import MaxAbsScaler imports the MaxAbsScaler class from PySpark's machine learning library, used for scaling each feature by its maximum absolute value to ensure all values lie within the range of [-1, 1].

```
mxabs_scaler = MaxAbsScaler(inputCol="features",outputCol="mxabs_features")
```

The code mxabs_scaler = MaxAbsScaler(inputCol="features", outputCol="mxabs_features") creates an instance of the MaxAbsScaler transformer, setting it to scale the data in the "features" column and output the scaled data in a new column named "mxabs_features".

```
mxabs_scaler_model = mxabs_scaler.fit(df_new)
```

The code mxabs_scaler_model = mxabs_scaler.fit(df_new) fits the MaxAbsScaler instance mxabs_scaler to the DataFrame df_new, creating a model to scale each feature by its maximum absolute value.

```
rescaled_df = mxabs_scaler_model.transform(df_new)
```

The code rescaled_df = mxabs_scaler_model.transform(df_new) applies the trained MaxAbsScaler model to the DataFrame df_new, resulting in a new DataFrame rescaled_df with features scaled by their maximum absolute values.

```
rescaled_df.select("features", "mxabs_features").show(truncate=False)
```

```
+--------------------------------+---------------------------------------------------------------------------+
|features                        |mxabs_features                                                             |
+--------------------------------+---------------------------------------------------------------------------+
|[3.0,25.0,21.0,3.0,2.0,20.0,5.0]|[0.6000000000000001,1.0,0.9130434782608695,0.6000000000000001,0.5,1.0,1.0] |
|[4.0,16.0,20.0,1.0,1.0,16.0,3.0]|[0.8,0.64,0.8695652173913043,0.2,0.25,0.8,0.6000000000000001]              |
|[2.0,15.0,16.0,2.0,1.0,17.0,5.0]|[0.4,0.6,0.6956521739130435,0.4,0.25,0.8500000000000001,1.0]               |
|[4.0,24.0,23.0,5.0,4.0,12.0,4.0]|[0.8,0.96,1.0,1.0,1.0,0.6000000000000001,0.8]                              |
|[2.0,25.0,4.0,1.0,1.0,18.0,2.0] |[0.4,1.0,0.17391304347826086,0.2,0.25,0.9,0.4]                             |
|[3.0,17.0,18.0,4.0,2.0,13.0,4.0]|[0.6000000000000001,0.68,0.7826086956521738,0.8,0.5,0.65,0.8]              |
|[4.0,18.0,13.0,0.0,2.0,17.0,4.0]|[0.8,0.72,0.5652173913043478,0.0,0.5,0.8500000000000001,0.8]               |
|[5.0,17.0,11.0,0.0,2.0,14.0,4.0]|[1.0,0.68,0.4782608695652174,0.0,0.5,0.7000000000000001,0.8]               |
|[2.0,18.0,11.0,5.0,4.0,19.0,3.0]|[0.4,0.72,0.4782608695652174,1.0,1.0,0.9500000000000001,0.6000000000000001]|
|[3.0,21.0,12.0,1.0,3.0,13.0,2.0]|[0.6000000000000001,0.84,0.5217391304347826,0.2,0.75,0.65,0.4]             |
|[2.0,22.0,18.0,0.0,4.0,19.0,3.0]|[0.4,0.88,0.7826086956521738,0.0,1.0,0.9500000000000001,0.6000000000000001]|
|[1.0,18.0,12.0,4.0,3.0,13.0,5.0]|[0.2,0.72,0.5217391304347826,0.8,0.75,0.65,1.0]                            |
|[4.0,18.0,16.0,0.0,2.0,19.0,2.0]|[0.8,0.72,0.6956521739130435,0.0,0.5,0.9500000000000001,0.4]               |
|[3.0,25.0,20.0,1.0,2.0,20.0,3.0]|[0.6000000000000001,1.0,0.8695652173913043,0.2,0.5,1.0,0.6000000000000001] |
|[5.0,25.0,23.0,2.0,3.0,19.0,4.0]|[1.0,1.0,1.0,0.4,0.75,0.9500000000000001,0.8]                              |
|[1.0,22.0,16.0,3.0,2.0,19.0,2.0]|[0.2,0.88,0.6956521739130435,0.6000000000000001,0.5,0.9500000000000001,0.4]|
|[2.0,16.0,10.0,2.0,2.0,13.0,5.0]|[0.4,0.64,0.43478260869565216,0.4,0.5,0.65,1.0]                            |
|[2.0,19.0,23.0,2.0,1.0,13.0,5.0]|[0.4,0.76,1.0,0.4,0.25,0.65,1.0]                                           |
```

Here we are selecting one columns mxabs_features and features from the new dataframe called rescaled_df then . show ing it with truncate equals to false so that we can see the complete contents of each row. Notice that we can see only positive number eventhou the rage is from -1 to 1.

```
from pyspark.ml.feature import Bucketizer
```

The code from pyspark.ml.feature import Bucketizer imports the Bucketizer class from PySpark's machine learning library, used for transforming continuous features into discrete categories (buckets) based on specified ranges that must be explicitly defined by the data analyst.

```
df1.show(10,False)
```

```
+-----+-----+-----+-----+-----+-----+-----+---------+
|col_1|col_2|col_3|col_4|col_5|col_6|col_7|label    |
+-----+-----+-----+-----+-----+-----+-----+---------+
|3    |25   |21   |3    |2    |20   |5    |0.111111 |
|4    |16   |20   |1    |1    |16   |3    |2.0329   |
|2    |15   |16   |2    |1    |17   |5    |4.799999 |
|4    |24   |23   |5    |4    |12   |4    |2.9254   |
|2    |25   |4    |1    |1    |18   |2    |4.66666  |
|3    |17   |18   |4    |2    |13   |4    |4.666666 |
|4    |18   |13   |0    |2    |17   |4    |1.7229999|
|5    |17   |11   |0    |2    |14   |4    |1.521    |
|2    |18   |11   |5    |4    |19   |3    |2.0198   |
|3    |21   |12   |1    |3    |13   |2    |2.0      |
+-----+-----+-----+-----+-----+-----+-----+---------+
only showing top 10 rows
```

Here we are displaying the first 10 rows of the DataFrame df1 in a tabular format without truncating the content of the columns.

```
splits = [0.0,1.0,2.0,3.0,4.0,5.0,float("inf")]
```

```
The code splits = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, float("inf")] defines a list of split points for bucketing in PySpark, creating
ranges [0.0-1.0), [1.0-2.0), [2.0-3.0), [3.0-4.0), [4.0-5.0), and [5.0-∞) for categorizing continuous data.  I find this very
powerfull.
```

```
bucketizer = Bucketizer(splits=splits, inputCol="label",outputCol="label_bins")
```

The code bucketizer = Bucketizer(splits=splits, inputCol="label", outputCol="label_bins") creates an instance of the Bucketizer transformer in PySpark, configured to categorize values in the "label" column into discrete bins based on the defined splits and output the results in a new column "label_bins".
Question 5:
Binning, also known as bucketing, is used in data processing and analysis to categorize continuous data into discrete bins or intervals. It is particularly useful for simplifying models, dealing with noisy data, and enhancing understanding of data distribution.

Example Use Case - Credit Score Categorization:

Imagine a financial institution that wants to assess the risk of loan applicants based on their credit scores. Credit scores are continuous data ranging, say, from 300 to 850. However, for risk assessment, it might be more practical to categorize these scores into discrete bins such as:

300-579: Poor 580-669: Fair 670-739: Good 740-799: Very Good 800-850: Exceptional By binning the credit scores into these categories, the institution can more easily apply policies or decisions based on these broader risk categories. For instance, it might decide that applicants in the "Poor" category are high-risk and may require a higher interest rate, or it might only offer certain loan products to those in the "Good" to "Exceptional" categories. This approach simplifies decision-making and policies, making them more manageable and less sensitive to small variations in the credit score.

Binning here helps to transform a complex, nuanced set of data (credit scores) into a simpler, more actionable form for risk management and decision-making processes.

```
binned_df = bucketizer.transform(df1)
```

The code binned_df = bucketizer.transform(df1) applies the configured Bucketizer transformation to the DataFrame df1, resulting in a new DataFrame binned_df with an additional column containing the binned categories of the "label" column.

```
binned_df.select(['label','label_bins']).show(10,False)
```

```
+---------+----------+
|label    |label_bins|
+---------+----------+
|0.111111 |0.0       |
|2.0329   |2.0       |
|4.799999 |4.0       |
|2.9254   |2.0       |
|4.66666  |4.0       |
|4.666666 |4.0       |
|1.7229999|1.0       |
|1.521    |1.0       |
|2.0198   |2.0       |
|2.0      |2.0       |
+---------+----------+
only showing top 10 rows
```

The code binned_df.select(['label', 'label_bins']).show(10, False) displays the first 10 rows of the binned_df DataFrame, specifically showing the "label" and "label_bins" columns without truncating their content.

```
binned_df.groupBy('label_bins').count().show()
```

```
+----------+-----+
|label_bins|count|
+----------+-----+
|       0.0|    4|
|       1.0|    4|
|       4.0|    4|
|       3.0|    1|
|       2.0|    5|
|       5.0|    2|
+----------+-----+
```

The code binned_df.groupBy('label_bins').count().show() groups the binned_df DataFrame by the "label_bins" column, counts the number of occurrences in each bin, and displays the result.

```
print(bucketizer.getSplits())
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, inf]
```

The code print(bucketizer.getSplits()) outputs the split points defined in the Bucketizer instance bucketizer, showing the boundaries used for bucketing the data.

We terminate the cluster here.